QF-Test - The Manual

Version 7.1.2

Quality First Software GmbH[1]

March 13, 2024

---

[1] https://www.qfs.de

Preface

As the name indicates, Quality First Software GmbH is dedicated to quality assurance for the software development process. Our contribution to this area is the product QF-Test the manual of which you are currently reading.

QF-Test is a professional tool for automating functional tests for Java or web applications with a graphical user interface. Depending on size and structure of a company the ungrateful task of testing sometimes falls to a QA department or team, sometimes to the developers and sometimes to the unlucky customer. Users of QF-Test are therefore usually developers or testers with varying knowledge about software development and testing in general and Java or web GUIs in particular.

The video

> ▷ 'Overview'
> https://www.qfs.de/en/yt/overview-42.html

gives a general overview of QF-Test.

You will find a more technical overview in the video

> ▷ 'Technical insights'
> https://www.qfs.de/en/yt/technial-insights.html

This manual is the primary source of information for QF-Test. We have tried to explain things in a way that is understandable for all users, independent of their technical knowledge, yet provide a complete and concise reference. In case of Java specific questions, testers may fare best by contacting their developers who will surely be able to assist.

Initially QF-Test did only support Java Swing GUIs. With version 2.0 support for Eclipse/SWT was added and web support with version 3.0. Parts of QF-Test and this manual owe to this history and most things are explained from the perspective of testing a Swing GUI. In most cases the concepts are universal apply similarly to all GUIs. Where things differ, specific notes explain the particularities of a web or SWT GUI.

How to use this manual

This manual is available in HTML and PDF versions. The HTML version is split across multiple files for better navigation and to avoid excessive memory consumption of the web browser. Due to extensive cross-linking, the HTML document is the preferred version for online viewing, while the PDF version is better suited for printing. Searching should also be done in the PDF version since the splitting of the HTML files prevents searches through the whole manual.

The PDF version of the manual is located at `qftest-7.1.2/doc/manual_en.pdf`, the entry page of the HTML version is at `qftest-7.1.2/doc/manual/en/manual.html`.

A web-browser for the HTML manual can be started directly from QF-Test. The `Help→Manual...` menu item will take you to the entry page of the manual and `Help→News...` will bring up the section documenting the latest changes. Context-sensitive help is also available for all kinds of tree nodes, attributes and configurable options by clicking with the right mouse button and selecting `What's this?` from the popup menu. This may not work if the system's browser is not accessible from QF-Test.

The manual consists of three parts which are kept in one document for technical reasons (it simplifies cross-linking and index generation). They may be split in a later version. These parts are

User manual[(2)]

This part explains how to install and run QF-Test and how to work with its user interface. It shows how to create and organize tests, then continues with more advanced material. To avoid duplication of text, the user manual often refers to the reference manual for detailed explanation. We recommend that you follow these links.

Reference manual[(417)]

This is a complete reference that covers all configurable options, all parts of a test suite , etc. When looking for specific information, this is the place to go. The reference manual also serves as the source for context-sensitive help.

Technical reference[(862)]

The part about technical details contains in-depth and background information about miscellaneous topics as well as a comprehensive API reference for the scripting interface. Beginners will rarely need to take a look at this part, but for the advanced user and the technically interested it is a valuable resource.

A learning-by-doing tutorial is also available in HTML and PDF versions. The HTML version, which is directly accessible from the `Help→Tutorial (HTML)...` menu item, is

located at `qftest-7.1.2/doc/tutorial/en/tutorial.html`. The PDF version is to be found at `qftest-7.1.2/doc/tutorial/tutorial_en.pdf` and is also directly accessible through ‖Help→Tutorial (PDF)...‖.

The following notations are used throughout the manual:

- ‖Menu→Submenu‖ represents a menu or menu item.

- (Modifier-Key) stands for a keystroke, where the modifier is one (or a combination) of (Shift)/(⇧), (Control)/(⌃), (Alt)/(⌥), or (⌘).

- `Monospaced font` is used for names of directories and files, user input and program output.

- In order to transfer at least part of the convenience of cross-linking to the paper version, references[ii] in the PDF version are underlined and show the target page number in small braces.

Contents

List of Figures

List of Tables

Part I

User manual

Chapter 1

Installation and startup

The video

> ▷ 'Installation & Trial License'
> https://www.qfs.de/en/yt/installation-trial-license.html

first explains the download and installation of QF-Test, then (starting at min 8:20) the installation of a trial license.

The installation of QF-Test on the supported operating systems is explained in detail in the subsequent sections. The following packages are available for download:

Windows (section 1.2[(5)])
    On Windows QF-Test is normally installed via the setup program `QF-Test-7.1.2.exe` which requires administrator privileges. If you are lacking the required permissions or prefer to keep all QF-Test files together in one place you can unpack the self-extracting archive `QF-Test-7.1.2-sfx.exe` instead.

Linux / Unix (section 1.3[(8)])
    On Linux and other Unix systems please unpack the archive `QF-Test-7.1.2.tar.gz`.

macOS (section 1.4[(9)])
    The disk-image `QF-Test-7.1.2.dmg` is provided for the installation on macOS.

It is possible to have different versions of QF-Test installed in parallel. Existing configuration files will not be overwritten during setup.

In section 34.2[(396)] you can find best practices about the QF-Test installation.

1.1 System requirements

1.1.1 Hard- and Software

QF-Test itself runs with Java 17. The required 64 bit Java Runtime Environment (JRE) is provided with QF-Test, so Java does not need to be installed on your system unless required by the SUT.

Note If your system under test (SUT) uses Java it should typically use its own JRE, not that of QF-Test. The Java command for the SUT can be configured separately when creating the setup sequence for your SUT. Supported Java versions for the SUT are listed below.

For a QF-Test installation you need to reserve about 1 GB on your hard disk. The required RAM to work with QF-Test is in the same region but depends on the sizes of your test suites and the length of your test run, see I've got a long-running test and QF-Test runs out of memory. How can I prevent that? A[(1222)]. Note that you need to add the resources required by the SUT.

1.1.2 Supported technologies

The following table summarizes the officially supported versions of operating systems and required software for this QF-Test version 7.1.2. Support for additional systems and versions may be available on request but is not owed by QFS. Another option to get support for older software can be to use one of the older QF-Test versions that are still available for download at https://www.qfs.de/en/qf-test/download.html.

Note In QF-Test version 7.0 support for 32 bit software was deprecated for removal in a future QF-Test version.

Technology	Version restriction	Comment
Windows	10, 11, Server 2012 R2, Server 2016, Server 2019, Server 2022	
Linux		No special restrictions.
Unix		Swing and JavaFX testing possible on Solaris, HP-UX, AIX,...
macOS	macOS 10.14 and higher	Swing, JavaFX, Web and Android testing possible.
JDK/JRE	17 embedded for QF-Test; 8 - 22 for the SUT	
Swing		All platforms.
JavaFX	8 and higher	All platforms.
SWT	3.7 - 4.30 (i.e. 2023-12)	Windows and Linux GTK only, GTK3 with SWT 4.6 and higher. For Eclipse/SWT 3.5 - 3.6 simply download https://archive.qfs.de/pub/qftest/swt_legacy.zip and extract the contents into the `swt` directory of your QF-Test installation.

Table 1.1: Supported operating systems and Java versions

Technology	Version restriction	Comment
Chrome	Version 120 via QF-Driver, current versions via Chrome DevTools Protocol (CDP-Driver) (at the moment Chrome 121) and automatic ChromeDriver download (WebDriver).	Headless Chrome supported. See also Browser connection mode[993]
Firefox (WebDriver)	As supported by the included GeckoDriver, i.e. currently 102esr and higher.	Headless Firefox supported
Microsoft Edge	Current versions via Chrome DevTools Protocol (CDP-Driver) and automatic MSEdgeDriver download (WebDriver).	Headless Edge supported.
Opera	Current versions via Chrome DevTools Protocol (CDP-Driver).	
Safari		WebDriver with Safari[999]
JxBrowser	version 6 and 7, embedded into Swing, JavaFX or SWT	
Electron	1.7 and newer	
Web component libraries	Detailed list of supported toolkits in section 49.2[988]	

Table 1.2: Supported web browsers and toolkits

1.2   Windows Installation

On Windows QF-Test can be installed in two variants.

1.2.1   Installing via the Windows setup file `QF-Test-7.1.2.exe`

This setup requires administrator privileges and follows the Windows standard of separating read-only program files from writable configuration files. If an older QF-Test version is detected it is also possible to skirt Windows standards and install QF-Test and its system configuration together at the place of the old installation.

4.2+   **Windows compliant installation**
Program files are saved to `C:\Program Files\QFS\QF-Test` or whichever target directory you choose. The system configuration with writable data is stored in `%PROGRAMDATA%\QFS\QF-Test`, irrespective of the selected target directory.

Note   `%PROGRAMDATA%` usually refers to the directory `C:\ProgramData` but the name may

vary depending on the Windows system. By default is is hidden in Windows Explorer. A simple way to navigate to this directory is to enter `%PROGRAMDATA%` into the address bar of Windows Explorer. In a PowerShell window use `cd $env:PROGRAMDATA`, in a cmd console window `cd /d %PROGRAMDATA%` to change to the respective drive and directory.

4.2+ **Installation together with an existing QF-Test version**

If an older QF-Test installation is found and there is no system configuration in `%PROGRAMDATA%\QFS\QF-Test` yet, you choose to follow the Windows compliant installation using `%PROGRAMDATA%` or to stick with the existing structure and install QF-Test there.

In the first case, after selecting the target directory for the QF-Test program files, the system configuration files are copied - just this once - from the existing installation to `%PROGRAMDATA%\QFS\QF-Test`.

When installing into the existing structure, QF-Test is installed into that directory and shares the system configuration that is already present there.

In both cases the directory `%PROGRAMDATA%\QFS\QF-Test\qftestpath` is added to the system PATH and the program files qftest.exe and qftestc.exe are copied there. This allows to start QF-Test from anywhere.

Independent of the installation choice both old and new QF-Test can be run in parallel. In case of the Windows compliant installation with `%PROGRAMDATA%` the old and new system configuration are independent. If the old structure is kept, all versions share the same system configuration. In the medium to long term we advise to move to `%PROGRAMDATA%` because the old structure requires changing access rights in the program directory, which is questionable. However, while migrating tests from QF-Test 4.1 to 4.2 it may be convenient to keep both versions close together. The move to a Windows compliant installation using `%PROGRAMDATA%` can also be made in the course of a later installation.

Silent Installation

For an automatic distribution on test systems it might be necessary to install QF-Test silently. QF-Test supports this kind of installation because the installer is based on Inno Setup. This allows to use nearly all documented parameters from https://jrsoftware.org/ishelp/index.php?topic=setupcmdline at the installation of QF-Test.

You can perform a silent default installation with `QF-Test-7.1.2.exe /VERYSILENT`.
If you want to not create a Deskop icon you can run `QF-Test-7.1.2.exe /VERYSILENT /MERGETASKS="!desktopicon"`. This executes a standard installation without the task "desktopicon".
Both the `minisetup-admin.exe` and `minisetup-noadmin.exe` can also be installed silently. For example via `minisetup-admin.exe /VERYSILENT`.

Please note that the installation for all users always requires elevated administrative rights. To also automatically accept the Windows UAC dialog the calling process must already have elevated rights. The parameter `/CURRENTUSER` does not help here, because the installation always requires elevated rights independent of installing for all or just the current user.

An exception of this rule is `minisetup-noadmin.exe`, which allows to configure an already installed QF-Test for the current user only. It does not need elevated rights.

Instead of performing a silent installation you can also use the portable self-extracting archive `QF-Test-7.1.2-sfx.exe`.

1.2.2 Unpacking the self-extracting archive `QF-Test-7.1.2-sfx.exe`

If you don't have administrator privileges or want to keep all QF-Test files together in a single place, unpack the archive `QF-Test-7.1.2-sfx.exe` at a suitable place. To do so, copy the file to the desired location and execute it there. If 7-Zip is installed on your system you can also right-click the archive to open and extract it with 7-Zip. This will create a directory named `qftest` at the target location which we will refer to as the root directory of QF-Test and that will also hold QF-Test's system configuration files.

After unpacking the files you can run the program `minisetup-noadmin.exe` in the sub-directory `qftest-7.1.2`. It will create associations for the file extensions belonging to QF-Test and optionally a startup menu entry and a desktop icon for QF-Test. If you have administrator privileges you can run `minisetup-admin.exe` instead which applies the same settings for all users and also adds the directory `%PROGRAMDATA%\QFS\QF-Test\qftestpath` to the system PATH and copies the program files `qftest.exe` and `qftestc.exe` there.

If you'd rather have a fully portable installation instead, you can create a folder named `userdir` in the `qftest` directory which will then serve as the user-specific configuration directory in place of `%APPDATA%\QFS\QF-Test` so that really all files belonging to QF-Test are kept together in one place and no changes are made to the system.

1.2.3 Completing the installation and configuring Java

As the last step each of the setup programs will offer to configure the Java program for QF-Test which is done with the help of a small dialog in which you can make your choices. With a portable installation you can run the program `qftest\qftest-7.1.2\bin\qfconfig.exe` to achieve the same.

4.2+ A 64 bit Java Runtime Environment is installed with QF-Test into its installation folder. It is recommended to use it. On a 32 bit Windows machine (beware: support for 32

bit software was deprecated in QF-Test version 7.0) you must provide a Java 8 JDK or JRE yourself. You can either copy a 32 bit JRE 8 into QF-Test's installation directory as `...\qftest-7.1.2\jre\win32` and also chose to use the QF-Test Java or you can install it as the system JRE or in any other convenient place and select that as the alternative Java program.

The dialog also lets you adjust the maximum amount of memory to be used by QF-Test with a default of 1024 MB.

The third value to be configured is the language for QF-Test. Normally the language is determined by the system settings, but you can also choose to always use the English or the German version.

The values above are stored in the file `launcherwin.cfg` in QF-Test's system configuration directory from where they are read by the `qftest.exe` start program. You can run the configuration program any time from the system menu to change these settings.

1.3 Linux/Unix Installation

First select a convenient directory that will contain this release of QF-Test as well as future updates. Common choices are `/opt` or `/usr/local`. Make sure you have write access to this directory and change to it. When upgrading to a new QF-Test version, use the same directory again.

Unpack the `.tar.gz` archive with `tar xfzv QF-Test-7.1.2.tar.gz`. This will create a directory named `qftest`, which we will refer to as the main or root directory of QF-Test. On a Linux/Unix system this also serves as the system directory holding the system configuration files of QF-Test.

After unpacking a QF-Test archive for the first time, QF-Test's root directory will hold only the version-specific subdirectory `qftest-7.1.2`. When upgrading, a new subdirectory for the current version will be added.

To finish the installation, change to the specific directory for the current QF-Test version with `cd qftest/qftest-7.1.2` and run the setup script provided (`setup.sh`).

The setup script will create the directories `log`, `jython groovy` and `javascript` under QF-Test's root directory unless they already exist. Additionally it will offer to create a symbolic link from the `/usr/local/bin` directory (or `/usr/bin` if there is no `/usr/local/bin`) to the shell run script for the `qftest` command. You need to have write permission to the `/usr/local/bin` directory for the link to be created.

On Linux QF-Test should normally use its own JRE. Alternatively the default `java` program for QF-Test can be defined now. Either way it can be overridden at execution time with the `-java <executable>` (deprecated)[868] argument. The setup script searches `PATH` and proposes to use the first `java` program it detects. If you want to use a differ-

ent `program` or if none was found, you can enter one. The script determines the JDK version automatically.

Next setting to perform is the maximum amount of memory to be used by QF-Test. As default 1024 MB are taken. Alternatively QF-Test can be started with the `-J-XmxZZZm` command line argument, where `ZZZ` defines the memory in MB.

Finally the language for QF-Test can be configured. By default the language depends on the system settings, but you can also choose to always use the English or the German version. Note that this setting will affect all QF-Test users. Alternatively you can run QF-Test with the `-J-Duser.language=XX` option using `en` for English or `de` for German.

Those of the above settings that differ from the default are written to the file `launcher.cfg` in QF-Test's root directory. This file is read by the `qftest` launch-script and also evaluated during an update of QF-Test.

1.4 macOS Installation

To install QF-Test on a macOS System, simply mount the `QF-Test-7.1.2.dmg` disk image and copy the QF-Test app to your `Applications` directory (or any other folder) and start it from there.

Note To configure custom program arguments like memory used by QF-Test or the language there is a separate options-page in the QF-Test options (General->Startup). You can configure the settings there and they will then be applied after restarting QF-Test.

1.5 The license file

Video The video

▷ 'Installation & Trial License'
https://www.qfs.de/en/yt/installation-trial-license.html

first explains the download and installation of QF-Test, then (starting at min 8:20) the installation of a trial license.

The video

▷ 'License update'
https://www.qfs.de/en/yt/license-update.html

shows how to update a license.

QF-Test requires a license file to run, which you should have received from Quality First Software GmbH.

4.0+   Since QF-Test 4.0 the preferred way to activate or update your QF-Test license is by way of the menu ‖Help→Update license...‖.
The traditional way as described below is also still valid.

Place the license file into the system directory of QF-Test. On Windows, depending on the type of installation, this will be `%PROGRAMDATA%\QFS\QF-Test` (see section 1.2[(5)]) or the root directory of your QF-Test installation as on Linux. Make sure the file is named `license` **with no extension**. Some mail clients try to guess the file type and add an extension on their own. When upgrading to a new QF-Test version you can simply keep the license file provided that it is valid for the new version.

Note   For a complete list of the directories relevant to QF-Test please open the info dialog via the menu ‖Help→Info‖ and select the "System info" tab.

If you need to upgrade your license, for example to increase the number of concurrent QF-Test instances or when upgrading to a new version, you will receive a file called `license.new` from Quality First Software GmbH which is typically not a valid license in itself but must be combined with your current license. To do so, proceed as follows:

- Place the file `license.new` in the same directory as the current license. Make sure that this directory and the file `license` are writable by you.

- Start QF-Test in interactive mode. QF-Test will detect the license update, verify its validity and offer to upgrade your license file.

- If you agree, the current license will be renamed to `license.old` and the new, combined license will be written to `license`. When you are satisfied that everything is OK, you can remove the files `license.old` and `license.new`.

- If QF-Test doesn't seem to recognize the license upgrade, make sure that the timestamp of the file `license.new` is newer than that of the file `license`. Also make sure that no other instance of QF-Test is running on your computer.

In case you need to specify a special name or location for the license file or work with more than one license, this can be achieved with help of the `−license <file>`[(873)] argument as described in chapter 42[(862)].

1.6   The configuration files

Note   For a complete list of the directories relevant to QF-Test please open the info dialog via the menu ‖Help→Info‖ and select the "System info" tab.

QF-Test saves all of its window configuration and those global options that represent personal preferences together in a file named `config` located in the QF-Test user con-

figuration directory which also holds run logs for tests run in interactive mode, profile directories for web testing and temporary files for editing and running scripts.

4.2+ On Windows the user configuration directory defaults to `%APPDATA%\QFS\QF-Test` for new installations. If that directory doesn't exist and you already used a QF-Test version older than 4.2 on the same system that created the directory `.qftest` in your home directory for the user configuration, QF-Test will continue to use that `.qftest` directory.

You can manually move the content of the directory `.qftest` to `%APPDATA%\QFS\QF-Test` and delete `.qftest` afterwards. QF-Test since version 4.2.0 will then use this directory only. You should not move the files if you still want to use a version older than 4.2.0!

On Linux the user configuration directory is always ∼/.qftest.

On macOS it is located at `/Users/<username>/Library/Application Support/de.qfs.apps.qftest`.

The personal config file is not read when QF-Test is run in batch mode (see section 1.7[11]). Irrespective of the system default you can always specify an explicit location for the user configuration directory as a whole with the `-userdir <directory>`[883] command line argument and just for the user config file with `-usercfg <file>`[883].

System specific options that need to be shared between users are saved in a file called `qftest.cfg` in the system configuration directory which also serves as the home for the license file, script modules, Java plugins and other customization files. On Windows the location of the system configuration directory depends on the installation variant (c.f. section 1.2[5]). It is either located in `%PROGRAMDATA%\QFS\QF-Test` or in the root directory of QF-Test.

On Linux and macOS the default system configuration directory is the root directory of QF-Test.

The location of the system config file can be changed with the command line argument and `-systemcfg <file>`[881] and that of the entire system directory with `-systemdir <directory>`[881].

1.7 Starting QF-Test

QF-Test can be run in two modes. In normal mode QF-Test is the editor for test suites and run logs and the control center for running programs, capturing events and executing tests. When run with the `-batch`[867] argument, QF-Test goes into "batch" mode. Instead of opening an editor window, the test suites given on the command line are loaded and executed automatically without the need for supervision. The result of the test is reflected in QF-Test's exit code[885], optional run logs (see section 7.1[117]) and reports (see chapter 22[271]).

The setup script for Linux/Unix offers to create a symbolic link from `/usr/local/bin`

to the `qftest` start script in the `qftest-7.1.2/bin` directory under QF-Test's root directory. That way you can simply enter `qftest` at the shell prompt to launch the application.

On Windows a menu shortcut is created as well as an optional desktop icon. You can either launch QF-Test from one of these or by double-clicking a test suite or a run log, since these files are associated with the QF-Test application. To run QF-Test from the console type `qftest`.

When run from the command line, QF-Test offers a wide range of arguments for customization, like selecting the Java VM to use. These are explained in detail in chapter 42[(862)].

In case different versions of QF-Test are installed at the same time, a specific version can be started by calling the `qftest` executable directly from the respective `qftest-X.Y.Z/bin` directory.

Mac In case QF-Test is not starting up anymore because of some incorrect settings under Options->General->Startup the default startup settings need to be restored. This can be done through running the following two commands from a macOS shell terminal.

```
defaults write de.qfs.qftest /de/qfs/qftest/ \
        -dict-add JVMOptions/ '{"Xmx"="-Xmx1024m";"Xms"="-Xms16m";}'
defaults write de.qfs.qftest /de/qfs/qftest/ \
        -dict-add JVMArguments/ '{"args"="";}'
```

Example 1.1: Resetting startup settings to defaults under macOS

1.8 Firewall Security Warning

On startup of QF-Test and/or the System Under Test (SUT) via QF-Test you might get a security warning from the Windows firewall asking whether to block Java or not. As QF-Test communicates with the SUT by means of network protocols, this must **not** be blocked by the local firewall in order to allow automated testing.

Chapter 2

The user interface

This chapter explains the structure of QF-Test's main window. When you are done reading, it might be a good idea to run QF-Test and try things out. In the $\boxed{\text{Help}}$ menu there is an entry labeled $\boxed{\text{Tutorial}}$ that should bring up your web browser with a hands-on, learning-by-doing tutorial. Should this fail because QF-Test cannot determine your system's standard browser, the tutorial can be found in the directory `qftest-7.1.2/doc/tutorial`, where a PDF version is also available.

Video The first part of the video

> ▷ 'Main window and the System under Test'
> https://www.qfs.de/en/yt/main-window-sut-40.html

shows the structure of the QF-Test main window.

2.1   The test suite

Automating a GUI test basically requires two things: control structure and data. The control structure defines what to do and when to do it. The data for a test consists of information about the SUT's GUI components, the events that will be triggered and the expected results.

QF-Test combines all of these into one data structure, a tree hierarchy that we call a *test suite*. The elements of the tree are called *nodes*. Nodes can contain *child nodes* (often just called *children*) and are themselves contained in a *parent node* (or just *parent*). The *root node* of the tree represents the test suite as a whole.

There are more than 60 different kinds of nodes all of which are explained in detail in the reference manual[515]. Some nodes are used as containers for data while others control the execution of a test. All of them have their own unique set of attributes.

The attributes of the currently selected node are displayed to the right of the tree in a detail view which can be toggled on and off via the View→Details menu item.

Figure 2.1: Structure of a test suite

The image above shows an example for a simple test suite. The attributes of the node named "Simple test" can be edited in the detail view to the right.

The basic structure of a test suite and thus the child nodes of the Test suite[(515)] root node is fixed. An arbitrary number of Test set[(525)] and Test case[(518)] with or without Dependency[(548)] nodes are followed by the Procedures[(595)], Extras[(547)] and Windows and components[(835)] nodes. The Procedures node holds Packages[(594)] and Procedures[(586)] which are explained further in section 8.5[(135)]. The Extras node is a kind of playground or clipboard where all kinds of nodes can be added for experimentation or temporary storage. The windows and components of the SUT's user interface are represented as Window[(812)] and Component[(823)] nodes which are located below the Windows and components node.

To get detailed information about a node or one of its attributes, click on it with the right mouse button and select What's this? from the context menu. This will bring up a browser displaying the corresponding section of the reference manual.

2.2 Basic editing

Editing a test suite falls into two categories: operations like Cut/Copy/Paste on the tree's nodes and changing the attributes of a node. The latter can be done either by editing the fields in the detail view and selecting *OK* or pressing $\boxed{\text{Return}}$, or by bringing up a dialog for the selected node with $\boxed{\text{Edit→Properties}}$ or $\boxed{\text{Alt-Return}}$ and changing the values there. If you change some values in the detail view and forget to press *OK* before moving the selection to another node, QF-Test will pop up a dialog with the changed values, asking you to either confirm your changes or discard them. This feature can be turned off with the option Ask before implicitly accepting detail modifications[(428)].

Some non-obvious key-bindings may come in handy when editing multi-line text attribute: $\boxed{\text{Ctrl-TAB}}$ and $\boxed{\text{Shift-Ctrl-TAB}}$ move the focus out of the text field, while $\boxed{\text{Ctrl-Return}}$ is a shortcut to select the *OK* button.

An extremely useful feature is the $\boxed{\text{Edit→Undo}}$ function ($\boxed{\text{Ctrl-Z}}$) which will take back any kind of change made to the test suite, including recordings or use of the replace dialog. Changes are undone step by step. If you find you went too far and undid more than you wanted, you can use $\boxed{\text{Edit→Redo}}$ ($\boxed{\text{Ctrl-Y}}$) to undo the undone. The number of steps that can taken back are limited only by available memory and can be configured with the option Number of undo levels per suite[(429)] (default 30).

2.2.1 Navigating the tree

Though the key-bindings for tree navigation are similar to those of most tree components, it won't hurt to mention them here. Besides, QF-Test comes with a few non-standard bindings that may come in handy.

The cursor keys are used for basic navigation. $\boxed{\text{Up}}$ and $\boxed{\text{Down}}$ are obvious. $\boxed{\text{Right}}$ either expands a closed node or moves down one row while $\boxed{\text{Left}}$ closes an open node or moves to its parent.

QF-Test's trees support a special variant of multi-selection. Multiple discontinuous regions can be selected, but only among siblings, i.e. children of the same node. If multi-selection across the whole tree were allowed, cutting and pasting nodes would become a real brain-teaser. Keys to try are $\boxed{\text{Shift-Up}}$ and $\boxed{\text{Shift-Down}}$ to extend the selection, $\boxed{\text{Ctrl-Up}}$ and $\boxed{\text{Ctrl-Down}}$ to move without affecting the selection and $\boxed{\text{Space}}$ to toggle the selection of the current node. Similarly, mouse-clicks with $\boxed{\text{Shift}}$ extend the selection while clicks with $\boxed{\text{Ctrl}}$ toggle the selection of the node being clicked on.

Special bindings include $\boxed{\text{Alt-Right}}$ and $\boxed{\text{Alt-Left}}$ which recursively expand or collapse a node and all of its children. $\boxed{\text{Alt-Down}}$ and $\boxed{\text{Alt-Up}}$ can be used to move to the next or previous sibling of a node, skipping the intermediate child nodes.

QF-Test keeps a history of recently visited nodes. $\boxed{\text{Ctrl-Backspace}}$ will take you back to the previously selected node. Also worthy of note are $\boxed{\text{Ctrl-Right}}$ and $\boxed{\text{Ctrl-Left}}$ which will scroll the tree to the right or left if it doesn't fit its frame.

2.2.2 Insertion marker

When inserting a new node or pasting in a copy of some other nodes, the insertion marker shows the place where the nodes will end up.

Figure 2.2: Insertion marker

Nodes are always inserted after the selected node. If the selected node is expanded, the new node is inserted as the first child of the selected node, otherwise it becomes a sibling of same. This behavior will take a little to get used to, especially for long-time users of the Windows explorer. However, there is no other way to insert a node at a definite position. In the example shown in figure 2.2[(16)] above, a new node would be inserted as the first child of the sequence called "Events", just before the Mouse event[(681)].

2.2.3 Moving nodes

Nodes can be copied and pasted or moved around within a test suite or to another suite. The standard keyboard shortcuts for cut, copy and paste, $\boxed{\text{Ctrl-X}}$, $\boxed{\text{Ctrl-C}}$ and $\boxed{\text{Ctrl-V}}$ are available as well as entries in the context menu.

2.0+ Alternatively, nodes can be moved using standard Drag&Drop operations. The default operation will move the selected node(s). If the $\boxed{\text{CTRL}}$ key is held down during the drop, the nodes are copied instead.

While dragging the mouse over the tree of a test suite, the insertion marker shows where the nodes will be dropped when the mouse button is released and whether the

operation is allowed. A green marker signals a legal operation, a red marker an illegal one. Nothing will happen if the nodes are dropped on an illegal target position.

During the drag you can expand or collapse nodes by dragging the mouse cursor over the expansion toggle and keeping it there for a moment. That way you can easily navigate to the desired target location without interrupting and restarting the drag.

The Drag&Drop operation can be aborted at any time by pressing Esc.

2.2.4 Transforming nodes

Some nodes can be transformed into different node types, which is a lot more convenient than first creating the desired target node and then copying over the required attributes. Examples of interchangeable nodes are Sequence[536] and Test step[539] or Server script[628] and SUT script[631]. The transformation of a node is possible only if its childnodes and its current position in the tree are also valid for the desired target node. The potential transformation targets can be selected from the entry Transform node into in the context menu. If the entry is not available there are no valid target nodes. In that case, moving the node to the Extras[547] node first may help.

You can find more details about the conversion mechanism under Details about transforming nodes[1164].

2.2.5 Tables

In various places QF-Test employs tables to view and edit a set of values, e.g. when defining variables[103] or for checks[707] of multiple elements.

+ / ✗ ↑ ↓ Variable definitions	
Name	Value
rootdir	some/directory

Figure 2.3: Example table

The buttons above the tables have the following keyboard shortcuts and effects:

⬜➕ ⌈Shift-Insert⌉
Insert a new row.

⬜✏ ⌈Shift-Return⌉, ⌈Alt-Return⌉
Edit a row. Opens a dialog with fields for every cell of the selected row.

⬜✖ ⌈Shift-Delete⌉
Delete the selected row.

⬜⬆ ⌈Shift-Ctrl-Up⌉
Move the selected row up by one.

⬜⬇ ⌈Shift-Ctrl-Down⌉
Move the selected row down by one.

Some tables also offer the ability to add and remove columns and edit the column title. For these, the following additional buttons are available:

➕ Insert a new column.

✖ Delete the selected column.

✏ Edit the title of the selected column.

To enter a value directly into the selected cell just start typing. This way you overwrite the current value of the cell. To edit the current value, either double click the cell or press ⌈F2⌉. To finish editing press ⌈Return⌉, to cancel and restore the old value press ⌈Escape⌉. If you try to enter an invalid value the cell's border will turn red and you can't accept the value.

Multi-selection of table rows is supported via mouse-clicks with ⌈Shift/Ctrl⌉ and ⌈Shift/Ctrl-Up/Down⌉. Cut copy and paste of the selected rows is done with ⌈Ctrl-X/C/V⌉. Pasting is restricted to tables with a similar column structure.

In the table's context menu additional actions might be available, e.g. show line numbers, locate component, etc.

A mouse click in a column header will activate sorting of table rows. A double-click in a column header will resize the column to fit the largest value in the column or opens the editor for the header text (data table).

2.2.6 Packing and Unpacking

During test development it is often necessary to move several nodes into a new parent node. A typical situation could be the re-factoring of procedures to re-organize them in packages or to wrap a workflow into a Try/Catch block.

For such requirements QF-Test allows the user to pack nodes into others. This can be achieved by selecting the nodes to pack, right-clicking and selecting $\boxed{\text{Pack nodes}}$ and the desired parent node.

QF-Test also allows the user to unpack such nodes and remove their parent. This can be used to remove unnecessary packages or test sets from the structure or to dispense with sequences or Try/Catch blocks that are no longer required. For unpacking right-click the node to unpack and select $\boxed{\text{Unpack nodes}}$.

Note The packing and unpacking actions are only shown in the menu if the desired target structure is legal.

2.2.7 Sorting Nodes

QF-Test allows sorting nodes. This can be achieved by clicking at a node with the right mouse button and selecting $\boxed{\text{Sort child nodes}}$. Alternatively you can also select multiple nodes, perform a right mouse click and then choose $\boxed{\text{Sort nodes}}$, which will sort the current selected nodes.

To guarantee a better overview the sorting algorithm puts ciphers prior to capital letter and those prior to small letters. Sorting doesn't modify the base structure of QF-Test nodes. It also follows the rule to keep Package nodes always prior to Dependency nodes and those always prior to Procedure nodes.

Note The base structure of a test suite will not be altered during sorting. You can sort test cases or procedures but the Procedures node will always stay prior to the Windows and components node.

2.3 Advanced editing

This section explains how to use the more advanced editing techniques such as search/replace and multiple views on the same test suite.

2.3.1 Searching

QF-Test provides two kinds of search operations, a general search through all nodes and attributes of a test suite or run log and in incremental search through the contents of a text area, including script consoles or program output.

General search

Though search and replace operations in QF-Test have much in common, there are significant differences, especially in the scope of the operation. Searching normally starts at the selected node and traverses the whole tree depth-first to the end. After asking for confirmation the search continues from the root of the tree to the original start of the search so each node is traversed exactly once. This is not unlike search operations in common text processors and should be intuitive to use.

By default QF-Test shows the search dialog in 'simple' mode, which allows searching for any appearance of a given text.

Figure 2.4: The simple search dialog

For a more specific search QF-Test allows limiting the search to specific attributes, node types or specific states of nodes. Therefore you have to switch to the 'advanced' mode by clicking the 'Switch mode' button in the toolbar of the search dialog.

Figure 2.5: The advanced search dialog

By default QF-Test will search all attributes in all kinds of nodes for the requested string.

Use the "In attribute" option to limit the search to a specific attribute.

The "Node type" option allows to limit the search to nodes of a specific kind.

The option "Scope of search operation" tells QF-Test where to search for the given expression, below the selected node(s), in the current test suite or in all currently opened

suites.

Activating options in "Only nodes with the following states" limits the search to nodes that have all of the activated states, e.g. a green mark and a breakpoint.

If "Match whole attribute" is selected, a search for the word "tree", for example, will not match an attribute value of "treeNode".

Regular expressions are explained in section 47.3[909].

Note To search for values of boolean attributes like Replay as "hard" event[684], use "true" or "false" (no quotes). If you want to search for an empty value you have to check "Match whole attribute".

If the search is successful, the resultant node is selected and a message in the status line displays the name of the attribute that contains the value.

3.4+ As already mentioned the searching process usually starts from the currently selected node. In case you want to select other nodes during your search process you can continue the previous search by using the "Search continue" button.

Once you have closed the search dialog you can still continue the search pressing F3. You can even trigger the same search from a new node pressing Ctrl-G.

A very useful feature is the ability to quickly locate all Procedure call nodes that call a given Procedure or all event nodes that refer to a given Component node, etc. Simply select the entry Locate references... from the context menu of a node that can be called or referred to. This will show a new frame showing all available references of it. You can reach the node in the test suite via a double click at the row in the list.

Figure 2.6: Result list for 'Locate references'

3.1+   It is also possible to get a list of all found nodes via pressing the "Show result list" button in the search dialog. From this dialog you can then reach any single node in your test suite.

Incremental text search

3.1+   In addition to searching the tree, components containing text like terminal areas or respective attributes in the details view can be searched independently by use of QF-Test's incremental search feature. This feature can be invoked either by selecting Search... from the component's context menu or by pressing Ctrl-F when the component is selected and owns the keyboard focus. Then the incremental search popup dialog appears at the upper right corner of the respective component. The figure below shows an incremental search for the terminal with highlighted search hits.

Figure 2.7: Incremental search

The search can be limited to a certain part of the contained text by selecting the region to be searched and invoking Search in selection... from the component's context menu or pressing Ctrl-Shift-F.

Beside this, the way the incremental search works as well as the available options should be self-explanatory.

2.3.2 Replacing

Once you understand how the scope of the replace operation differs from searching, the replace dialog should be just as intuitive to use as the search dialog. When the replace operation is in progress, you have a choice of replacing one match at a time or all matches at once. To avoid unexpected results when selecting the latter, there needs to be a way to limit the nodes that will possibly be affected. To that end, replace operations can be limited to the currently selected nodes and their direct or indirect child

nodes. For a replace operation that covers the whole tree, either select the root node or choose the respective "Scope of replace operation" option in the dialog.

Figure 2.8: The replace dialog

The options are identical to the ones for searching. When "Match whole attribute" is turned off, multiple replacements within one attribute are possible. When replacing "a"

with "b" for example, "banana" would change to "bbnbnb". Be sure to read section 47.3[(909)] about how to use regular expressions for replacing.

If the search is successful, the resultant node is selected and a confirmation dialog is brought up that shows the target attribute and its value before and after the change.

Figure 2.9: The replace query dialog

It offers the following choices:

- When *Replace* is selected, the attribute's value is changed and the search carries on, showing the query again for the next match.

- *Replace all* means change this value and all the rest of the matches in one go without asking again.

- *Skip* leaves this attribute unchanged. The search continues and the query dialog is shown again in case of another match.

- Obviously *Cancel* ends the replace operation.

If you know what to expect you can skip the query entirely by selecting *Replace all* in the replace dialog. After the attributes have been changed, the number of affected nodes is shown in a message dialog.

3.1+ After performing the actual replacement QF-Test will show a list of all touched nodes. You can also open a list of all nodes, which will be touched before the actual replacement pressing the "Show result list" button in the replace dialog.

Note Whether values are replaced one by one or all at once also affects the way the undo function will take these changes back. All changes of a *Replace all* operation are taken back in one step, while single changes are undone one at a time.

2.3.3   Complex searches and replace operations

3.3+   Sometimes a simple search is not enough. Imagine, for example, that you want to set a Timeout of 3000 milliseconds for all text checks on a certain component. You know the component's QF-Test ID, but if you search for that QF-Test ID you will also find events and other kinds of checks referencing it. And if you search for the node text 'Check text' you will find all Check text nodes, not just those for the given component.

Instead of providing several combinable levels of search criteria QF-Test offers complete flexibility through its marks. First perform a search for your first criterion, e.g. the node text and select 'Show result list'. In the resulting dialog select all entries in the table by pressing ⟨Ctrl-A⟩, press 'Set mark' to assign the blue mark to all result nodes and close the dialog. You can now perform a second search or a replacement with the scope limited to nodes with a given mark. In our example you would perform a replacement of the empty string with '3000' on all Timeout attributes with the search scope set the all nodes with the blue mark in the whole tree.

2.3.4   Multiple views

It is possible to open multiple views that show different parts of the same tree structure simultaneously. This can be useful when managing large test suites or to compare the attributes of different nodes.

Additional views are opened via the ⟦View→New window...⟧ menu item. The current node will be the root node for the new view. Additional views are similar to the primary views, but with a limited range of menus.

2.3.5   Hiding toolbar buttons

You can reduce the number of buttons shown in the toolbar by right-clicking the button you want to hide and selecting ⟦Hide toolbar button⟧ from the popup menu.

In order to restore the original toolbar right-click it somewhere and select ⟦Show all tooolbar buttons⟧. In case you hid all toolbar buttons, please select ⟦View→Show toolbar⟧.

Chapter 3

Quickstart your application

This chapter provides instructions on how to quickly set up your application as the SUT (System Under Test).

The video

 'The Quickstart Wizard Java'
https://www.qfs.de/en/yt/quickstart-wizard-java-42.html

shows how to work with the Quickstart Wizard for Java applications.

The video

 'The Quickstart Wizard Web'
https://www.qfs.de/en/yt/quickstart-wizard-web-42.html

shows the Quickstart Wizard for web applications.

In order to make you application recognized by QF-Test as SUT it basically needs to be started out of QF-Test. There are a number of special process nodes available within the ‖Insert→Process nodes‖ to perform this task but the straight forward way is to use the Quickstart Wizard as described below. For those with an aversion to wizard dialogs, the manual way is explained at section 44.1[889].

A precondition for testing Java based SUTs is that QF-Test can hook into the GUI toolkit:

For Swing/JavaFX or combined Swing/JavaFX and SWT applications QF-Test hooks into the Java's JVM Tool Interface. Normally QF-Test can do this directly. Only for some non-standard JDKs it may be necessary to instrument those first. See JRE deinstrumentation[899] for details if necessary.

For JavaFX applications and respective combinations the connection works exclusively via the QF-Test agent. Please ensure the option Connect via QF-Test agent[514] is activated.

For Eclipse/SWT based applications, an instrumentation of the SWT library may be necessary. The Quickstart Wizard, which is described below, will automatically add the

necessary step to the setup sequence. For detailed technical information please see section 45.2[900].

Web Web application testing does not require instrumentation but there are some constraints to consider that are explained in chapter 14[200].

3.1 Setup sequence creation

With the Quickstart Wizard QF-Test offers a convenient utility for creating a startup sequence for your application.

You can open the Quickstart Wizard via the ‖Extras→Quickstart Wizard...‖ menu item or the 🪄 toolbar button. Please follow the steps which should be self explanatory.

Figure 3.1: Quickstart Wizard

As result the Wizard delivers a startup sequence under the "Extras", as shown in the following figure:
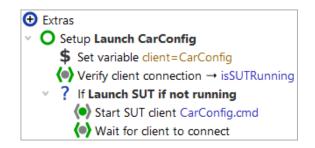
Figure 3.2: Startup sequence created by the Quickstart Wizard

The created setup sequence varies depending on the specific type of the application. But all of them follow the same standard. At the beginning you will find a Set variable node which specifies the name of the SUT client for QF-Test. This node is followed by a Wait for client to connect node that checks whether its necessary to start the application. The subsequent If evaluates the result of its predecessor and contains the start steps for your application. The actual launch takes place in the start node which is specific to the type of application. That node is followed by an other Wait for client to connect node which ensures that QF-Test connects to your application during the startup procedure. (Details about the different start node types and attributes can be found in section 44.1[889].)

SWT For SWT based applications an additional procedure call node for SWT instrumentation[900] is added.

Web The standard startup sequence for web includes some additional nodes for setting variables, initializing browser cache and cookie settings and possibly install a toolkit resolver. See chapter 14[200] for further information about starting a web based SUT.

3.2   Executing the setup sequence

The setup sequence can be executed directly after creation via selecting the green setup sequence node in the tree and pressing "Replay" toolbar button ▶ .

When executing the setup sequence your application should start up and the "Start recording" button ● in the QF-Test toolbar should become activated which indicates that QF-Test is properly connected to the SUT.

Now you are able to record and replay your first test sequences as described in chapter 4[35]. There is also a learning by doing tutorial available from the QF-Test help menu which guides you through all features of QF-Test.

In case you are facing an error message or the red "Start recording" button stays inactive, please proceed with the following paragraph.

3.3   In case the client does not connect ...

If your application (or the browser window in case of web testing) doesn't come up at all:

- The error dialog QF-Test typically displays should provide a first indication.

- Please look for error messages in the terminal window. If there is no terminal window visible in the bottom area of QF-Test, it can be activated through the menu

item ‖View→Terminal→Show‖. Additional information about program output can be found in section 3.4[(32)].

- Be sure to double-check the attribute values in the setup sequence nodes are correct. Possibly a typo has crept in somewhere. Details about the different start node types and attributes can be found in section 44.1[(889)].

Web
- As browser development cycles i.e. those of Firefox tend to shorten, be sure the installed browser is supported by the QF-Test version you are using. The terminal output should show a respective error message. See section 1.1.2[(3)] for the latest browser versions supported. Possibly you need to update QF-Test to a later version or temporarily use another browser. See chapter 14[(200)] for further information.

If the SUT gets visible but QF-Test is not able to connect to the client (`ClientNotConnectedException`[(855)]):

- Please double-check the terminal output content (see also above) for possible error messages.

- In case the case the red record button in the toolbar gets activated after the error message occurred, the timeout value in the Wait for client to connect[(670)] node needs to be increased.

SWT
- For an Eclipse/SWT application first make sure that you specified the correct application directory. You may want to take a look at the run log (see section 7.1[(117)]) to see if any warnings or errors were logged during execution of the Procedure `qfs.swt.instrument.setup`.

- Check the run log in general for possible additional error indications (see section 7.1[(117)]).

After possibly having adapted your test suite or settings retry executing your setup sequence. If you are not getting any further you might want to consider trying a sample test suite from the tutorial or you contact our support.

3.4 Program output and the Clients menu

The standard output and error streams of all processes started by QF-Test are captured and stored in the run log under the node that represents the respective starter node. In this QF-Test does not distinguish between SUT clients and arbitrary processes or shell scripts started with a Start process[(642)] or Execute shell command[(652)] node.

The main window contains a shared terminal view that shows the output of all processes started by a test that was run from this window. The ‖View→Terminal‖ sub-menu holds items to configure whether this terminal is visible, whether the tree or the terminal should use the are in the lower left corner, whether long lines are wrapped and whether it is automatically scrolled to the end when new output arrives. Other items let you clear the terminal or save its contents to a file. The maximum amount of text that the terminal holds is configurable in the option Maximum size of shared terminal (kB)[(463)].

In addition to the shared terminal, for each active or recently terminated process there is an individual terminal window that shows its output. These individual terminal windows can be opened from the ‖Clients‖ menu. The shared terminal's intention is to provide visual feedback whenever new output arrives, while the individual terminals are better suited for actually studying that output.

Active processes can also be stopped with the help of the ‖Clients‖ menu, either individually in the respective sub-menu or all at once with ‖Clients→Stop all clients‖.

The number of terminated clients that are kept in the ‖Clients‖ menu is set with the option Number of terminated clients in menu[(461)]. If your processes generate lots of output and you are low on memory you may want to reduce that number.

Note The ‖Clients‖ menu also serves well in case you are not sure which specific QF-Test product you need to purchase. The GUI technologies used by your applications are shown in '[ ]' next to the active client name. The example below shows two clients using Java swing and web which suggests to buy a QF-Test/swing+web license.

Figure 3.3: GUI technology information

3.5 An application started indirectly from an already connected SUT

If a second Java VM is started from an already connected SUT, QF-Test will recognize this as an indirect connection attempt from a child process of the first SUT and automatically assign an artificial client name to this new SUT. The name is created by appending ':2' to the client name of the parent SUT, signifying that this is the second process for

this client. Yet another Java VM started by either of these SUTs would get ':3' appended to the original client name unless the second process was already terminated so the ':2' was again free for use.

To summarize, the sequence for connecting to an indirectly started SUT typically consists of an event node that triggers something like a button click or menu selection, causing the SUT to launch the second SUT, followed by a <u>Wait for client to connect</u>⁽⁶⁷⁰⁾ node for the ':2' extended client name.

Chapter 4

Capture and replay

Once the SUT is up and running under QF-Test, the next step is to record sequences of events and play them back.

The video

> ▷ 'Capture and Replay'
> https://www.qfs.de/en/yt/capture-replay-40.html

explains capturing and replaying of sequences.

The video

> ▷ 'Creating a test case'
> https://www.qfs.de/en/yt/test case-40.html

shows how to create a test case.

4.1 Recording sequences

In order to record a sequence of events in the SUT, the SUT must have been run from QF-Test (see chapter 3[(28)]) and the connection between QF-Test and the SUT must be established. A visual indicator of this is the color of the record button which turns red when it is enabled.

Figure 4.1: Disabled and enabled Record button

To record a sequence, simply start recording be pressing the record button ● or se-

lecting ‖Record→Start‖. Then switch to the SUT, execute a few commands, switch back to QF-Test and stop the recording with the stop button ■ or ‖Record→Stop‖. The recorded events will be added to the test suite, either directly at the position indicated by the insertion marker[(16)] or as a new Sequence[(536)] under the Extras[(547)] node, depending on the setting of the Insert recording at current selection[(439)] option. Pause the recording with ‖Record→Pause‖ or the pause button ▌▌ if you need to execute some steps in the SUT that should not be recorded and you don't want to stop and restart the recording.

Recording mode can be started and stopped directly in the SUT by use of the Hotkey for recording[(439)]. Default key is ‖F11‖.

Any Components[(823)] referred to by the newly recorded events are added automatically to the Windows and components[(835)] node if they are not there already.

There are many options that influence the way QF-Test records events and how it treats the components of the GUI. All of these are explained in detail in section 39.2[(439)] of the reference manual. Once you are familiar with QF-Test you should take the time to skim through it.

Here's some general advice for recording:

- Record short sequences at a time.

- After recording, take a look at the sequence, try to understand what you got and whether it represents the actions you took.

- Edit the sequence to remove unnecessary events, especially those at the beginning and end caused by switching windows. QF-Test has excellent filters that should catch nearly all of these, but some might remain and have to be removed manually.

- Finally, try out the new sequence to see whether it replays OK. Then you can cut/copy/paste as needed to integrate it into larger parts.

Mac   For SUT's running on macOS QF-Test disables use of the screen menu bar and activates normal menu bar behavior like on other platforms. This is due to the fact that QF-Test cannot fully access the screen menu bar which prevents proper capture/replay of menu actions. In case the typical Mac screen menu bar behavior is necessary for any reason, this can be forced by adding the line `qfs.apple.noScreenMenuBar=false` to the file `qfconnect.properties` that is located in QF-Test's root directory. After restarting the SUT the screen menu bar is supposed to work as normal on Mac.

4.2 Running tests

To run some tests, select the node or nodes to execute and press $\boxed{\text{Return}}$ or the play button ▶ or select $\boxed{\text{Run→Start}}$. QF-Test will mark each node with a small arrow as it is executed and also show progress messages in the status bar. This can slow down execution a little and can be turned off with the options Mark nodes during replay[(457)] and Show replay messages in status line[(457)].

When the test is finished, the result is shown in the status bar. If things are fine you should see "No errors", otherwise the number of warnings, errors and exceptions is shown. Additionally, a message dialog is shown in case of errors or exceptions to make sure you don't miss these.

As for recording there are many options that influence the replay of tests. Some of these are only for convenience while others have a major impact on the outcome of the tests. Be sure to read section 39.3[(456)] some time to familiarize yourself with these.

To abort execution before the test is finished, press the stop button ■ or select $\boxed{\text{Run→Stop}}$. You can also suspend execution temporarily via the pause button ⏸ or by selecting $\boxed{\text{Run→Pause}}$. This will also enable the debugger (see chapter 7[(116)]). To continue, press pause again.

While a test is run at full speed it can be tricky to stop or interrupt it, especially when the mouse cursor is actually moved across the screen or the SUT's windows are raised on every event. To regain control, press the Hotkey for pausing test run ("Don't Panic" key)[(456)] (the default is $\boxed{\text{Alt-F12}}$). This will pause all running tests immediately. To continue, press the same combination again.

While building a test suite you will often want to execute some sequences to get the SUT to a point where you can continue recording. Sometimes you may want to skip certain nodes at this stage because they don't get you where you want, but you don't want to delete them or move them to some other place. In that case use the $\boxed{\text{Edit→Toggle disabled state}}$ menu item to disable the node(s). When you want to use them again later you can re-enable them.

The current error state during replay as well as the final result is shown in the status line at the bottom of the QF-Test main window. The visibility of this status line can be controlled via $\boxed{\text{View→Show status line}}$.

In case Test set[(525)] or Test case[(518)] nodes (section 8.2[(131)] describes their usage) are executed the status line also contains relevant result counters from the following list.

Counter Icon	Description
#	Total number of test cases. This counter value starts with a '>' symbol in case there are skipped test sets.
❶	Number of test cases with exceptions.
➖	Number of test cases with errors.
➖	Number of test cases with expected errors. <u>Expected to fail if...</u>[(523)] marks a test case expected to fail.
➕	Number of successful test cases.
⏩	Number of skipped test cases. A test case is skipped when its (optional) <u>Condition</u>[(522)] fails. This counter value starts with a '>' symbol in case there are skipped test sets.
⏩	Number of skipped test sets. A test set is skipped when its (optional) <u>Condition</u>[(529)] fails.
⊘	Number of not implemented test cases. A test case is not implemented when it doesn't contain nodes that were executed during the test run.
▶	Number of executed test cases.
%	Percent test cases passed.

Table 4.1: Test result counter in the status line

The final test result counts also appear in the report which can be created for any test run. Reports are discussed in <u>chapter 22</u>[(271)].

The counter values above can also be accessed as <u>variables</u>[(108)] during the test run. A <u>TestRunListener</u>[(1087)] can help to keep track of counter values and trigger dependent actions.

4.3 Recording checks

Though it can be quite entertaining to record sequences and watch the magic dance of the SUT as they are played back, the task at hand is to find out whether the SUT actually works as expected. This is where <u>checks</u>[(707)] come into play. The most common check, the <u>Check text</u>[(708)] node, reads the text displayed by a component, e.g. a text field, and compares it to a given value. If the values differ, an error is signaled.

How checks work is explained in detail in the <u>Reference manual</u>[(707)]. There is a range available from simple text check to advance <u>Check image</u>[(729)] and even <u>custom check types</u>[(1073)] can be implemented. Here we are going to concentrate on the most convenient way to create checks, which is to record them.

While recording, the SUT is in *record mode*, which means that all events are collected and sent to QF-Test. With the help of the check button ✔ or by selecting Record→Check you can bring it into *check mode*, recognizable through the different

mouse cursor. In this mode, recording events is suspended. Instead, the mouse cursor is tracked and the component under it is highlighted. When you click on the component, a check for the component is recorded using the value that is currently displayed. To get back to record mode, select the check button or menu item again.

There are different kinds of checks[707] that can be performed. Which kinds of checks are applicable depends on the selected component. Some components don't display any text, so a Check text[708] node doesn't make sense for, say, a scroll bar. Clicking on a component with the right mouse button while in check mode brings up a menu of applicable checks for this component. Select one of the items to create the respective check node. Clicking with the left mouse button always records the default check, which is the topmost one in the popup menu.

If you hold down the [Shift] or [Ctrl] key while clicking with the right mouse button, the check menu will stay open after making a selection. That way, you can easily record multiple kinds of checks for the same component.

Checks integrate well with events and you'll soon develop a recording style a'la *click, click, type, click, check, click, click, check...* Having to switch back and forth between QF-Test and the SUT every time you want to create a check can be a real pain. That is where the Hotkey for checks[440] option comes into play. It defines a key which toggles the SUT between record mode and check mode. The default value is [F12], but if this key has some defined meaning for your application you can change it to whatever you like. To record a sequence of interspersed events and checks, simply start recording, switch to the SUT and record the sequence. Whenever you want to add some checks, just press [F12] (or whatever you defined), record the checks, then switch back to record mode by pressing [F12] again and continue. This way you can work with the SUT for the whole sequence and need to switch back to QF-Test only to stop the recording.

One word of warning should be repeated: Don't let this convenience tempt you into recording extremely long sequences. When something changes that causes such a sequence to fail you will be hard put to find out what went wrong and how to cope.

4.4   Recording components

As already described component information is automatically stored when recording events or checks. Nevertheless there are situations where capturing of just components proves useful.

To activate *component recording mode* you simply need to press the record components button ⬜ or select ‖Record→Record components‖ from the menu. Then switch to the SUT window where you will notice that the component below the mouse cursor is now highlighted.

Clicking with the left mouse button on a component will record the single component while pushing the right button instead will pop up a menu with choices to record the nested components as well or all components in the window. Multiple components can be captured in this way. Then switch back to QF-Test and release the record components button ☐ or deactivate ‖Record→Record components‖. Now the recorded component information is stored in shape of respective Component[823] nodes under the Windows and components[835] node.

Component recording mode can be alternatively controlled by a configurable hotkey. Default binding is [F11], the option that configures it is Hotkey for components[447].

Pressing [F11] (default setting) *in the SUT window* starts component recording. Further details can be found in the documentation for the option Hotkey for components[447].

Note Only one test suite at a time can receive the recorded components. If more than one test suite is open and each is shown in an individual window, i.e. workbench mode is deactivated, either the test suite in which the recording is stopped (toolbar button or menu) or - when using [F11], the test suite that can be selected via the menu item ‖Record→Suite is receiver for recording‖ will receive the components.

The component recording feature can also be used to quickly locate a component independent of whether it has been recorded before. When creating event or checks by hand or changing the target component, the QF-Test component ID[682] attribute needs to be specified. When a component is recorded, its QF-Test ID is saved in the clipboard and can be pasted directly into the QF-Test component ID field with [Ctrl-V]. You can also jump directly to the Component node with [Shift-Ctrl-Backspace] or by choosing the ‖Edit→Select next node‖ menu item or clicking the respective toolbar button.

The context menu which appears when clicking with the right mouse button on a component also contains an entry ‖Show methods‖ which allows of some kind of component inspection (see section 5.12[96]).

Components play a central role in the structure of a test suite which is explained further in chapter 5[42].

4.5   Recording of HTTP requests (GET/POST)

Web

In order to record a (GET/POST) request sent by the SUT, the SUT must have been launched from QF-Test (see chapter 3[28]) and the connection between QF-Test and the SUT must be established.

While recording, the SUT is in *record mode*, which means that all events are collected and sent to QF-Test. By selecting ‖Record→Record HTTP Requests‖ you can bring it into *request recording mode*. In contrast to Recording sequences[35] all GET/POST-

request send by the web browser are saved as http-request nodes in this special recording mode. To get back to record mode, select the menu item again.

In section Web options[490] the ability to change the type of the recorded request is described. By default Browser HTTP request[808] is recorded. This Request type is likely used to automate large web form inputs, the use of separate input nodes will be avoided. The form data will be submitted within the browser, so that the response will be shown as well. At this point the test execution could be continued. In opposition to this the Server HTTP request[802] will be directly submitted through QF-Test without the need of a running browser. The response is also only available in QF-Test and a eventually running browser will not be affected.

All attributes of an accordingly recorded request node as well as the parametrization of requests are explained in detail in the HTTP Requests[802] section of the reference part of this manual.

Chapter 5

Components

Though they often go unnoticed, at least until the first `ComponentNotFoundException`[(850)] occurs, the Component[(823)] nodes are the heart of a test suite, since stable component recognition is the central challenge of a good GUI testing tool. QF-Test takes care of it most of the time, but some special situations require manual definitions or interventions. Thus it is important to understand components and their handling in QF-Test and this chapter aims to explain the fundamentals.

Videos

The video

    'Component recognition'
        https://www.qfs.de/en/yt/component-recognition.html

first explains the criteria for component recognition, then (starting at min 13:07) the use of generic components using regular expressions, followed by generic components using variables in component recognition attributes.

There are two videos available explaining in detail how to deal with a `ComponentNotFoundException`:

-     'ComponentNotFoundException - Simple case'
    https://www.qfs.de/en/yt/componentnotfoundexception-simple-40.html
-     'ComponentNotFoundException - Complex case'
    https://www.qfs.de/en/yt/componentnotfoundexception-complex-40.html

The video

 'Dealing with the explosion of complexity in web test automation'
https://www.qfs.de/en/yt/web-test-automation-40.html

gives you a good idea of how QF-Test handles a deeply nested DOM structure.

Live recording of the special webinar

 'Component recognition'
https://www.qfs.de/en/yt/component-recognition-51.html

.

GUI actions and components   Actions by the end-user of an application are transformed into *events* by QF-Test. Every event has a target component. For a mouse click this is the component under the mouse cursor, for a key press it is the component that has *keyboard focus*. When an event is recorded by QF-Test, additional information about the target component is recorded as well, so that the event can later be replayed for the same component.

Recognition
Component recognition is one of the most complex part of QF-Test. The reason is the need to allow for some degree of change. QF-Test is a tool designed for regression testing, so when a new version of the SUT is tested, tests should continue to run, ideally unchanged. So when the GUI of the SUT changes, QF-Test needs to adapt. If, for example, the "OK" and "Cancel" buttons were moved from the bottom of the detail view to its top, QF-Test would still be able to replay events for these buttons correctly. The extent to which QF-Test is able to adapt depends on the available recognition criteria. In this area software developers often can, with relatively low effort, make a great contribution to the creation of robust regression tests.

The following criteria are available for component recognition:

- Class[56], correlates with the component's function
- Name[58], based on the Component identifiers[59]
- Feature[63], a piece of text belonging to the component
- Extra features[66], further recognition features like a description or tooltip
- Index[70]
- Geometry[70]
- Component hierarchy[70]

These criteria figure into recognition with varying importance. Especially important are a component's class and Component identifiers[59]. With the latter, developers can make a great contribution to test stability (see How to achieve robust component recognition[49]). For more information, see Weighting of recognition features for recorded components[902].

Storing recognition information

Information about recognition can be stored by QF-Test either in a Component node[71] or directly in the event nodes as a SmartID[73]. In Component nodes versus SmartID[46] you will learn which option is preferrable for which use case.

By default, QF-Test will record Component nodes.

Child elements and nested components

There are some components that QF-Test addresses relative to a parent component. Among these are table cells, list entries, tree nodes, button icons, or a checkbox inside a table cell.

QF-Test has special ways of addressing these components. This topic is explained in detail in Sub-items: Addressing relative to a parent component[82].

Also, QF-Test offers the ability to define Scope[80] to limit actions (clicks, text entry, checks) to components contained within.

5.1   Components of a GUI

The graphical user interface (GUI) of an application consists of one or more windows which hold a number of components. These components are nested in a hierarchical structure. Components that hold other components are called containers. As QF-Test is itself a complex application, its main window will serve well as an example:

Figure 5.1: Components of a GUI

The *window* contains a *menu bar* which holds the *menus* for QF-Test. Below that is the *toolbar* with its *toolbar buttons*. The main area employs a *split pane* to separate the tree view from the details. The tree view consists of a *label* ("Test suite") and the *tree* itself. The detail view contains a complex hierarchy of various components like *text fields*, *buttons*, a *table*, etc. Actually there are many more components that are not obvious. The tree, for example, is nested in a *scroll pane* which will show *scroll bars* if the tree grows beyond the visible area. Also, various kinds of *panes* mainly serve as containers and background for other components, like the region that contains the "OK" and "Cancel" buttons in the detail view.

Unless explicitly stated otherwise, the term "component" in this manual refers to elements of a GUI, regardless of what the individual components are called in the respective GUI technology.

5.2 Component nodes versus SmartID

Regognition criteria can be linked to events in tests in two different ways. With the classic method, the criteria are stored as attributes of a Component[(823)] node (see Component node[(71)]). These are then referenced in the tests via their QF-Test component ID. Alternatively, GUI elements can be addressed directly by the recognition criteria via SmartID[(73)]. Components are not neccessary in that case.

SmartIDs and classic recorded Component nodes can be used alternatingly, and even combined when neccessary. The following points may help you decide whether to use SmartIDs or record components.

- Improved readability of tests[(46)]

- Test-driven development[(47)]

- Keyword-driven testing[(47)]

- Stability of recognition[(48)]

- Maintainability[(48)]

- Performance[(49)]

5.2.1 Improved readability of tests

SmartIDs offer advantages over recorded Component nodes in the following situation:

The referenced GUI components should be directly recognizable in event and check steps. If the Component identifiers[(59)] of the components are cryptic, but usable descriptions are available, SmartIDs have the advantage.

Figure 5.2: Readability of SmartIDs

Figure 5.3: Readability of identifiers

The readability of a test can also be improved in procedures, if description-based Smar-tIDs can be used instead of cryptic identifiers.

SmartIDs also can increase readability for fields with the same recognition criteria but differently labeled container panels. In the following example, the SmartIDs `#Shipping address@#Last name` and `#Billing address@#Last name` could be used.

Figure 5.4: Readability of SmartIDs in panels with description

5.2.2 Test-driven development

With test driven development, the big advantage offered by SmartIDs is that no Components need to be created. Additionally, in test driven development Component identifiers[59] are often defined in the technical design and can then be used for test creation. For example, if the component identifier is `btnOK`, the component can be referenced via the SmartID `#btnOK`.

5.2.3 Keyword-driven testing

Keyword-driven tests are implemented on a technical level with procedure calls and pa-rameters. This way, the test creator does not record any components and is dependent

on visual information from the GUI for identifying components. This could be the label of the component or its function (class). In a SmartID, these possibilities for recognition can also be combined with each other and with an index.

5.2.4   Stability of recognition

The stability of recognition is equally good with recorded components and SmartIDs if the SmartID uses the name, if possible in combination with the class. At it's core, stability of recognition depends on the probability of change of the used criteria. If, for example, the label of a component is stable across versions of an application, the recognition via a label-based SmartID (Feature[63] or `qfs:label*` variants[67]) will be stable as well.

Recorded Component nodes use a predefined algorithm for recognition. It gives different importance to individual recognition criteria. Class, name, and hierarchy have the highest priority. If no name is present, hierarchy, label, index, and geometry (in descending importance) are combined into a probability. That probability is the basis for deciding if a GUI element is the wanted component.

This algorithm has proven very good for most usage scenarios. However, there are cases in which subordinate recognition criteria (like the label) offer greater stability than the higher-weighted criteria. With recorded Component nodes, we could intervene via a resolver, see The `resolvers` module[1016]. However, the strengths of the SmartID get to shine here, because it can specifically target a stable recognition criterium (or even a combination of multiple criteria).

This is the case when, for example, the label is more stable than the Component identifiers[59].

SmartIDs also have advantages when there is a big chance of changes to the component hierarchy during version changes (or even just during application start) or if recognition features of parent components change. SmartIDs don't consider the component hierarchy by default.

5.2.5   Maintainability

Regarding the maintainability, recorded Component nodes have the upper hand, because recognition criteria are stored centrally in the node, and later changes only need to be performed in this one place.

With SmartIDs however, recognition criteria are stored decentrally. It is possible to perform changes across tests via the powerful search-and-replace feature. For SmartIDs with the same recognition criteria for different components, manual tweaking may still be necessary.

5.2.6 Performance

SmartIDs that use Component identifiers[59] can keep up well with a Component node performance-wise, because the recognized Names are indexed.

However, if the SmartID uses the label (Feature[63] or `qfs:label*` variants[67]) or other Extra features[66], performance will not be as good as with recorded Component nodes, since the GUI elements are not filtered by Class name beforehand and all GUI elements with matching classes must be searched.

5.2.7 Combining Component nodes and SmartIDs

Recorded Component nodes can be combined with SmartIDs. You can find details about this in Sub-items: Addressing relative to a parent component[82] and Component QF-Test ID as SmartID[80].

Recorded components can be used to overlay the SmartID syntax by setting their QF-Test ID[824] to a SmartID including prefix "#". This allows simple, data-driven or pre-generated tests to be created with SmartID and only at neuralgic points to define individual components more specifically without having to adapt the tests or procedures for this.

5.3 How to achieve robust component recognition

The most important feature of a GUI test tool is the recognition of the graphical components. QF-Test offers a lot of configuration options for this. This section presents an overview over the most common strategies and settings to make component recognition as stable as possible.

Note You should define a component recognition strategy for your project **before** starting to implement tests in QF-Test. Otherwise, test maintenance can create larger expenses.

The recognition of components in the SUT during test playback is very complex. The challenge lies in the changes the interface of the SUT can go through all the time even during normal use. Windows are opened and closed or varied in size, changing the position and size of components within. Menus and combo boxes are opened and closed, components are added or removed, made visible or inisible, activated or disabled. In addition, the application under test itself will develop over time, which will reflect in changes to its interface. All these changes must be handled flexibly by QF-Test to be able to match components as reliably as possible.

In many cases, QF-Test can manage this with the default settings. QF-Test uses an intelligent, probability-based algorithm to achieve a stable and fault tolerant component

recognition. It assesses the attributes described in The following criteria are available for component recognition: 5[43] and weights them. However, if no good recognition attributes are available, even the best algorithm will struggle. For this case, there are possiblilites for configuration and optimization which are described in Opportunities for optimization[54].

The first question is whether the default settings are already sufficient, so:

5.3.1 How to judge robust component recognition

This section is intended to enable you to assess whether the current component recognition will, in all likelihood, be robust.

The following are important elements of robust component recognition:

- Class[56] of the component

- Name[58]

- label (Feature[63] or `qfs:label*` variants[67])

- moderate hierarchy depth of the component tree

In most cases, the class and the name are the most robust criteria for recognition. (In rare cases, however, they change from one version of the application to the next. We consider this messy case in Opportunities for optimization[54], item 2.) Usually the label of the component also rarely changes and is thus also well suited. Detailed information on all the detection features can be found in Recognition criteria[56].

With the class, QF-Test tries to derive which functionality a component has from the class used by the developer. Based on this generic class, QF-Test optimizes the inclusion and provides function-specific checks (for example, checking an entire row for a table).

First, let's show you how to quickly see if generic classes have been detected and if names or labels are present based on the recorded Component[823] nodes.

Figure 5.5: Component tree 1

The class is the black text of the COMPONENT nodes. If the class starts with an uppercase letter followed by a lowercase letter, it is generally one of the Generic classes[(1186)], for example `Button`. For browser elements, if the class consists only of uppercase letters, QF-Test could not determine the functionality. In the example `HTML` and `BODY`.

Whether names or labels are present can be seen from the brown texts. This is the QF-Test component ID, which allows the following conclusions to be drawn:

- If the class does not show up in the QF-Test ID, it means that either a Name[(58)] is present (in the example, `CalculatorPanel` and `CalculatedPrice`) or, if no generic class was recognized, a label (Feature[(63)] or `qfs:label*` variants[(67)]) is present. In the example, `CarConfigWeb`.

- If the QF-Test ID starts with the class, no Name[(58)] could be determined and the following part is the label of the component (Feature[(63)] or `qfs:label*` variants[(67)]). In the example `labelTotal` and `buttonOK`.

- If neither name nor label are found, the QF-Test ID repeats the class in lowercase letters. In the example `html`.

- If multiple components would be assigned the same QF-Test ID with the described algorithm, an ongoing number will be appended. In the example `buttonOK2`

Video A certain hierarchy for components is helpful for recognition. Only deep nestings are problematic. For component recognition, only few hierarchy levels are actually relevant. The others can be ignored. The video

▷ 'Dealing with the explosion of complexity in web test automation'
https://www.qfs.de/en/yt/web-test-automation-40.html

visualizes the problem of deep nestings - and also the solutions. The example above only has a shallow hierarchy depth. This is optimal.

Note The component tree in the example above was created with the following settings in the section ‖Record→Components‖:

- Prepend QF-Test ID of window parent to component QF-Test ID[(452)] disabled, which corresponds to the default setting.

- Prepend parent QF-Test ID to component QF-Test ID[(452)] set to `Never`, which also corresponds to the default setting.

Access these settings via the menu item ⸤Edit→Settings⸥

As an alternative to evaluating the QF-Test ID in the component tree you can get a list of all components with their names via the QF-Test search. To do this, in the search dialog, set `In Attribute` to `Name` and `Node type` to `Component` and click on `Show Result List`.

If you record something, the components you interact with will automatically be recorded. To record all components at once for analysis, choose ⸤Record→Record components⸥. Then right-click in the GUI and select `Whole window`. (After the analysis it makes sense to delete the components to avoid unnecessary ballast.)

Here are two more example component trees with evaluation of how robust the component recognition is.

Example 1
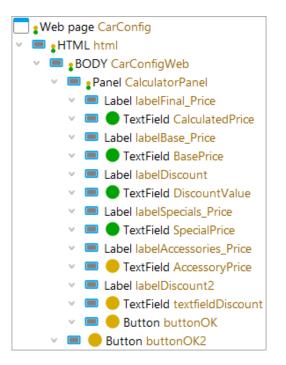
Figure 5.6: Stable component recognition - Example 1

Positive: <u>Generic classes</u>[1186] are recognized: `MenuBar`, `TabPanel`, `Panel`, `Label`, and `TextField`.

Positive: Names were determined for the text fields marked green, identifiable by the QF-Test ID (brown text) not starting with the class, for example `BasePrice`.

Positive: For the text fields and buttons marked yellow, no names were determined, identifiable by the QF-Test ID (brown text) starting with the class (`textfield`, `button`). But the second part of the QF-Test ID shows that at least a label was found.

Not important: the labels don't have names. However, they are rarely relevant for testing.

Not important: the containers 'HTML' and 'BODY' don't have a generic class. They could be mapped to 'Panel'. In this case, this would neither improve recognition nor unlock additional functionality in QF-Test (such as additional checks for check recording).

Positive: No superfluous containers except for `BODY`.

Example 2
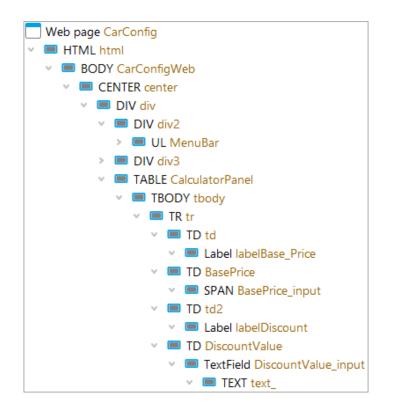
Figure 5.7: Stable component recognition - Example 2

Positive: Names or labels could be determined for test-relevant components, identifiable by the QF-Test ID (brown text) not starting with the class, for example `BasePrice` and

`DiscountValue_input`.

Negative: Generic classes were only recognized for few components. A component mapping with a CustomWebResolver is missing here, see <u>Improving component recognition with a `CustomWebResolver`⁽⁹⁴⁵⁾</u>.

Negative: Superfluous hierarchy levels. The `DIV`, `TR TD`, `CENTER`, and `TABLE` components should be mapped to `Panel` (see <u>The Install CustomWebResolver node⁽⁹⁴⁹⁾</u>) or ignored (see <u>Install CustomWebResolver node − Syntax⁽⁹⁵⁰⁾</u>, parameter `ignoreTags`).

5.3.2 Opportunities for optimization

If generic classes and names are available for the relevant components, you can assume that component detection is robust in the vast majority of cases and skip the rest of this section.

If there are problems with recognition, there are two fundamentally different cases to consider:

Is the component displayed (in time)?
This case is unrelated to component recognition itself. It occurs if QF-Test is too fast for the application, so to speak. In this case, you should explicitly wait for the appearance of the component in your test case. Find more information at <u>Timing synchronisation⁽⁹⁰⁾</u>.

Is the displayed component recognized?
There are several options here:

Web: Assignment of generic classes to GUI element classes
For web applications, please first perform component assignment as described in <u>The Install CustomWebResolver node⁽⁹⁴⁹⁾</u>. If this does not lead to sufficient stability, then continue reading in this section.

Unstable Component Identifiers
<u>Component identifiers⁽⁵⁹⁾</u> have been assigned, but they are not stable across application versions. In this case, it is better to remove the identifiers using resolvers and work with the remaining detection criteria if stable identifiers cannot be set by the developers.

In the case of web tests, a corresponding setting in CWR parameter 'customIdAttributes' (see <u>Install CustomWebResolver node − Syntax⁽⁹⁵⁰⁾</u>) can help.

No component identifiers
No <u>Component identifiers⁽⁵⁹⁾</u> has been assigned and the other criteria are not sufficiently stable. Here it is also often worthwhile to contact development and explain to them the relevance of component identifiers for regression tests - or to convince the person who is responsible for

development and testing in terms of budget that a small amount of additional work in development for entering the identifiers can mean a significant reduction in effort in the test department.

If this is not possible, there may be other stable recognition criteria which QF-Test does not use by default. These can be announced via a name resolver (see section 52.1.7[(1023)]).

Component identifiers contain stable parts

Only parts of the Component identifiers[(59)] are stable. If a computer-readable schema is available, this may be a case for a name resolver (see section 52.1.7[(1023)]). For a web application, this can also be defined via the CWR configuration category 'autoIdPatterns' (see Install CustomWebResolver node – Syntax[(950)]).

The components have labels wich QF-Test does not recognize out of the box

There is no name and the default QF-Test algorithm does not detect a feature or extra feature 'qfs:label', even though there are possible candidates available. In this case you can announce the labels through a FeatureResolver (see section 52.1.10[(1027)]) or ExtraFeatureResolver (see section 52.1.11[(1028)]).

Web components sometimes have an attribute which can be used as a label. This can be announced through the CustomWebResolver category 'attributesToQftFeature' (see Install CustomWebResolver node – Syntax[(950)]).

Parts of the feature or extra feature 'qfs:label' are stable

In this case you can either use regular expressions directly in the Component node or in the SmartIDs. But the solution could also be a FeatureResolver (see section 52.1.10[(1027)]) or ExtraFeatureResolver (see section 52.1.11[(1028)]).

Parent components are unstable

The component itself is stable, but one of its parent containers is not stable. Here, regular expressions or resolvers for the affected containers can help. If all test-relevant components have names, the option Name override mode (record)[(450)] in section Record→Components→Name override mode can also be set to "Override everything".

The use of SmartIDs is also an option here.

Additional or missing parent components

The component itself is stable, but its containing hierarchy is not stable because containers can appear or disappear. If all test-relevant components have names, the option Name override mode (record)[(450)] in section Record→Components→Name override mode can be set to "Override everything".

Alternatively, the component can be moved up in the component tree hierarchy, so it is no longer influenced by the unstable containers.

The use of SmartIDs is an option here as well.

Component structure or index

The attribute Class index[(828)] plays a subordinate role, but comes to effect if the component recognition must do without name and feature or the extra feature 'qfs:label'. If the Class index is unstable as well, it can be deleted so the geometry comes to effect. In this case the window size of the application to be tested should always be set to the same value after lauch (see Component event[(695)]).

5.4 Recognition criteria

5.4.1 Class

The class of a component is a very important attribute as it describes the type of the recorded component. Once QF-Test records a button, it will only look for a button on replay, not for a table or a tree. Thus the component class conveniently serves to partition the components of a GUI. This improves performance and reliability of component recognition, but also helps you associate the component information recorded by QF-Test with the actual component in the GUI.

Besides its role in component identification, the class of a component is also important for registering various kinds of resolvers that can have great influence on the way QF-Test handles components. Resolvers are explained in detail in section 52.1.7[(1023)].

The Name is used here for generating the QF-Test component ID. Examples for this can be found in How to judge robust component recognition[(50)].

In a SmartID[(73)] components can also be directly addressed via their Names, without recording a Component node[(71)].

The influence of the class on the QF-Test ID[(824)] of the component is described below, usage as SmartID in section 5.6[(73)].

Generic classes

Each UI toolkit usially defines its own system-specific classes for components like Buttons or Tables. In case of Buttons, that definition could be `javax.swing.JButton` for Java Swing, or `org.eclipse.swt.widgets.Button` for Java SWT, or `javafx.scene.control.ButtonBase` For JavaFX, or `INPUT:SUBMIT` for web applications. In order to allow your tests to run independently of the utilised concrete technology QF-Test unifies those classes via so-called generic classes, for example all buttons are simply called `Button` now.

You can find a detailed description of generic classes in chapter 59[(1186)]. In addition to the generic class, system specific classes are recorded as Extra features[(825)], but with the status "ignore". In case of recognition problems because of too many similiar components, these can be enabled to sharpen recognition, even if detracting from flexibility.

Swing

JavaFX Even if the class was extended, the generic class will be recorded. Additionally, it should be mentioned that this concept allows QF-Test to easily create tests with obfuscated classes without having to change the default settings. During replay, QF-Test compares the recorded Class name attribute of the component with each class of the object in the SUT. Therefore, QF-Test can handle class name changes as long as the base type remains the same.

Web HTML is a very flexible language to describe content and structure of a website. There is only a minimum of quasi-standards like `INPUT:SUBMIT`, for which you can always expect the same functionality and which can therefore be assigned to a QF-Test class by default. The development of web applications usually happens via toolkits which use their own standards. QF-Test includes class mappings for a range of commonly used toolkits, see Special support for various web frameworks[(988)]. If the developers of the application extended a toolkit or used a custom one, it will be neccessary to announce the class mapping to QF-Test. This is described in Improving component recognition with a `CustomWebResolver`[(945)].

If QF-Test can assign a component a generic class, this will gain the following advantages for test creation and execution:

Independence from concrete technical classes

With generic classes, a certain independence from the concrete technical classes is established. This concept allows you to create tests independent of the concrete technology.

Improved component recognition

If the functionality of the component is known, the most suitable recognition criteria can be stored.

Example button: The button label is the first choice for the Feature and the extra feature 'qfs:label'.

Example text field: It does not make sense to use the text value for recognition. Instead, QF-Test searches for a label nearby and stores this in the extra feature 'qfs:label'.

The generic class itself also is a differentiation criterium. This is especially noticeable in web applications, where most components will be recorded with the class `DIV`, matching their HTML tag by default.

Optimal mouse position during replay

The generic class also influences the optimal mouse position during event replay.

Example button: The mouse click is ideally placed in the middle of the button.

Example text field: The mouse click is ideally placed in the same place where the tester clicked during recording, so text can be inserted in exactly the same place if needed.

Class-specific checks

In addition, QF-Test offers additional class-specific checks during recording. For example, text fields can be checked for their editable state. Check items[719] however only make sense for lists, tables or trees.

5.4.2 Name

In case the developers have assigned Component identifiers[59] to a component, QF-Test will recognize this and use it, if suitable, for the attribute Name[825].

If a value for Name[825] was found, it will also be used for generating the QF-Test ID[824] of the component. Examples for this can be found in How to achieve robust component recognition[49].

The value of the Name attribute is also the first choice during recording of SmartID[73]s.

The reason for the tremendous impact of names is the fact that they make component recognition reliable over time. Obviously, locating a component that has a unique name assigned is trivial. Without the help of a name, QF-Test uses lots of different kinds of information to locate a component. The algorithm is fault-tolerant and configurable and has been fine-tuned with excellent results. However, every other kind of information besides the name is subject to change as the SUT evolves. At some time, when the changes are significant or small changes have accumulated, component recognition will fail and manual intervention will be required to update the test suite.

Another aspect of names is that they make testing of multi-lingual applications independent of the current language because the name is internal to the application and does not need to be translated.

Test automation can be improved tremendously if the developers of the SUT have either planned ahead or are willing to help by defining names for at least some of the components of the SUT. Such names have two effects: They make it easier for QF-Test to locate components even after significant changes were made to the SUT and they are highly visible in the test suite because they serve as the basis for the QF-Test IDs QF-Test assigns to components. The latter should not be underestimated, especially for components without inherent features like text fields. Nodes that insert text into components called "textName", "textAddress" or "textAccount" are far more readable and maintainable than similar nodes for "text", "text2" or "text3". Indeed, coordinated naming of components is one of the most important factors for the efficiency of test automation and the return of investment on QF-Test. If development or management is reluctant to

spend the little effort required to set names, please try to have them read this chapter of the manual.

If developers used another consistent scheme for assigning identifiers which QF-Test does not recognize out of the box, please take a look at Influencing the 'Name' attribute by implementing a NameResolver[(62)].

When determining distinct Names, the options Name override mode (replay)[(472)] and Name override mode (record)[(450)] can be set to "Override everything", which makes the component recognition independent from the component hierarchy. Because of name caching, this will gain maximum performance.

To simplify assigning of identifiers, QF-Test offers a feature to suggest identifiers for components whose testing would benefit from it. Read more about this in Hotkey for components[(447)].

Note   Changes to identifiers in the application under test should be avoided as much as possible, as this undermines component recognition and can mean a lot of rework in the tests. Please note that if changes do occur, they should be made in the Name[(825)] attribute of the component and not in the QF-Test ID[(824)] attribute, which is only used to reference the component in the tests! Another possible difficulty can be that the name change occurs directly in the test in the reference to the component, for example when a mouse click occurs in the QF-Test component ID[(682)] attribute. The test then fails with an `UnresolvedComponentIdException`[(857)].

Component identifiers

Component identifiers are called differently in the different UI technologies. In the manual, the term 'name' is also used for them. In addition, the criteria for whether and how the identifiers are transferred to the 'name' attribute are slightly different depending on the technology.

The following is valid for the default settings, especially of Name override mode (replay)[(472)] and Name override mode (record)[(450)] (default value: "Hierarchical resolution"). The use of resolvers could change the described behavior as well.

Java Swing/AWT
     The component identifier is called 'Name' here. If set, it will be transferred to the Name attribute. If there are duplicate component identifiers inside a container, QF-Test will create the Extra feature[(825)] `qfs:matchindex` with the appropriate index for the duplicates.

All AWT and Swing components are derived from the AWT class `Component`. That is why their `setName` method is the standard for Swing SUTs. Thanks to this standard, many developers make use of it even without considering test automation, which is a great help.

JavaFX

The component identifier is called 'ID', here. If set, it will be transferred to the Name attribute. If there are duplicate component identifiers inside a container, QF-Test will create the Extra feature[(825)] qfs:matchindex with the appropriate index for the duplicates.

For JavaFX, setId is used to assign names to components (here called "nodes"). Alternatively, IDs can be set in FXML via the attribute fx:id. Although IDs of nodes are supposed to be unique, this is not enforced.

Java SWT

The component identifier is also called 'Name', here. If set, it will be transferred to the Name attribute. If there are duplicate component identifiers inside a container, QF-Test will create the Extra feature[(825)] qfs:matchindex with the appropriate index for the duplicates.

Unfortunately SWT has no inherent concept for naming components. An accepted standard convention is to use the method setData(String key, Object value) with the String "name" as the key and the designated name as the value. If present, QF-Test will retrieve that data and use it as the name for the component. Obviously, with no default naming standard, very few SWT applications today have names in place, including Eclipse itself. Fortunately QF-Test can derive names for the major components of Eclipse/RCP based applications from the underlying models with good results - provided that IDs were specified for those models. See the Automatic component names for Eclipse/RCP applications[(451)] option for more details.

Web

The natural candidate for naming the DOM nodes of a web application is the 'id' attribute of a DOM node - not to be confused with the QF-Test ID attribute of QF-Test's Component nodes. Unfortunately the HTML standard does not enforce IDs to be unique. Besides, 'id' attributes are a double-edged sword because they can play a major role in the internal JavaScript operations of a web application. Thus there is a good chance that 'id' attributes are defined, but they cannot be defined as freely as the names in a Swing, JavaFX or SWT application. Worse, many DHTML and Ajax frameworks need to generate 'id' attributes automatically, which can make them unsuited for naming. The option Turn 'ID' attribute into name where "unique enough"[(491)] determines whether QF-Test uses 'id' attributes as names.

Fortunately, component identifiers can be realized via different attributes of the GUI element. Mostly it is the attribute 'id', sometimes also 'name' - but other attributes can be used as well.

The option Turn 'ID' attribute into name where "unique enough"[(491)] determines whether QF-Test uses 'id' attributes for names or not. Please keep in mind that

the option <u>Eliminate all numerals from 'ID' attributes</u>[(491)] can also cause originally unique identifiers to not be unique anymore after the deletion of the numbers. When checking if the resolved Name is unique, the component's parent containers will be considered when judging uniqueness if the options <u>Name override mode (replay)</u>[(472)] and <u>Name override mode (record)</u>[(450)] are set to the default value "Hierarchical resolution".

The automatically generated 'id' attributes sometimes contain a static part which can be used as identifier. This can be configured through the CWR category `autoIdPatterns`, see <u>Install CustomWebResolver node – Syntax</u>[(950)]. Also, this procedure can be used with the `customIdAttributes` parameter to use any other HTML attribute as a component identifier.

In case of web applications that use a UI toolkit supported by QF-Test, you can look at <u>section 49.2.2</u>[(990)] to learn more about setting unique identifiers for each toolkit.

Win

 The component identifier is called 'AutomationId' here. If set, it will be transferred to the Name attribute. If there are duplicate component identifiers inside a container, QF-Test will create an <u>Extra feature</u>[(825)] named `qfs:matchindex` and an appropriate index for the duplicates.

Android

 The component identifier is called 'ID', here. It will only be transferred to the Name attribute if it is not a trivial class name (see <u>Android - list of trivial component identifiers</u>[(906)]). If there are duplicate component identifiers inside a container, QF-Test will create the <u>Extra feature</u>[(825)] `qfs:matchindex` with the appropriate value for the duplicates.

About setting identifiers

There is one critical requirement for identifiers: They must not change over time, not from one version of the SUT to another, not from one invocation of the SUT to the next and not while the SUT executes, for example when a component is destroyed and later created anew. Once an identifier is set it must be persistent. Unfortunately there is no scheme for setting identifiers automatically that fulfills this requirement. Such schemes typically create identifiers based on the class of a component and an incrementing counter and invariably fail because the result depends on the order of creation of the components. Because identifiers play such a central role in component identification, non-persistent identifiers, specifically automatically generated ones, can cause a lot of trouble. If development cannot be convinced to replace them with a consistent scheme or at least drop them, such identifiers can be suppressed with the help of a `NameResolver` as described in <u>section 52.1.7</u>[(1023)].

QF-Test does not require ubiquitous use of identifiers. In fact, over-generous use can even be counter-productive because QF-Test also has a concept for components being "interesting" or not. Components that are not considered interesting are abstracted away so they can cause no problem if they change. Typical examples for such components are panels used solely for layout. If a component has a non-trivial identifier QF-Test will always consider it interesting, so naming trivial components can cause failures if they are removed from the component hierarchy in a later version.

Global uniqueness of identifiers is also not required. Each class of components has its own namespace, so there is no conflict if a button and a text field have the same identifier. Besides, only the identifiers of components contained within the same window should be unique because this gives the highest tolerance to change. If your component identifiers are unique on a per-window basis, set the options Name override mode (replay)[(472)] and Name override mode (record)[(450)] to "Override everything". If identifiers are not unique per window but identically named components are at least located inside differently named ancestors, "Hierarchical resolution" is the next best choice for those options.

Two questions remain: Which components should have identifiers assigned and which identifiers to use? As a rule of thumb, all components that a user directly interacts with should have an identifier, for example buttons, menus, text fields, etc. Components that are not created directly, but are automatically generated as children of complex components don't need an identifier, for example the scroll bars of a `JScrollPane`, or the list of a `JComboBox`. The component itself should have an identifier, however.

If components were not named in the first place and development is only willing to spend as little effort as possible to assign identifiers to help with test automation, a good strategy is to assign identifiers to windows, complex components like trees and tables, and to panels that comprise a number of components representing a kind of form. As long as the structure and geometry of the components within such forms is relatively consistent, this will result in a good compromise for component recognition and useful QF-Test ID attributes. Individual components causing trouble due to changing attributes can either be named by development when identified or taken care of with a `NameResolver`.

Influencing the 'Name' attribute by implementing a NameResolver

In GUI testing projects you can face a lot of interesting naming concepts. Sometimes the components in an application have no names, but the testers know an algorithm how to name them reliably. Sometimes existing names change from time to time or are completely dynamic, for example you can get a name 'button1' after the first recording and after the second recording you get 'button2'. Another situation could be that the current version of the application is part of the name of a dialog window.

Sometimes the tester knows an algorithm for setting unique Namen. In such cases you should take a closer look at The `NameResolver` Interface[(1023)] in the chapter The `resolvers` module[(1016)].

A NameResolver can be used to change or remove names set by developers for the QF-Test perspective. They are only removed for QF-Test not from the real source code.

You can think about utilizing NameResolvers in following cases:

- The SUT has dynamically changing names.

- You know a method to set the names uniquely.

- You want to map names to other names (for example due to new versions or for testing other languages.)

- You want to tune the names of components, for example to remove some parts and get nicer QF-Test component IDs in QF-Test.

If you can achieve per-window uniqueness of names with the help of a NameResolver you can also think about setting the options Name override mode (replay)[(472)] and Name override mode (record)[(450)] to "Override everything".

Whenever possible it is preferable that developers set the names directly in their source code as they best know the context of that component. Implementing a NameResolver can become an excruciating task if the developers change the content of the GUI a lot.

NameResolvers are described in detail in section 52.1.7[(1023)].

5.4.3 Feature

The Feature attribute stores, roughly said, a text that is useful for recognition and is directly connected to the component itself. This can be either the text of the component (for example the label on a button), a programmatically assigned identifier or label of a component (for example CheckBox, RadioButton, TextField), a title (Window[(812)], 'Dialog', 'TitledPanel'), or for a Web page[(818)] the URL.

Frequently, the value of the Feature is identical to the Extra feature[(825)] `qfs:label`. This is because the label of the component is stored in `qfs:label` and this is often the text that is directly connected to the component. The redundancy still makes sense, since a status can be set for the extra feature: 'Ignore', 'Should match', or 'Must match'. The Feature implicitly always has the status 'Should match'. For backwards compatibility reasons it cannot be replaced by `qfs:label`.

If no Name[(58)] can be determined, the Feature is used for generating the QF-Test component ID. Examples for this can be found in How to judge robust component recognition[(50)].

Components can also be addressed directly in a SmartID⁽⁷³⁾ via the Feature without recording a Component node⁽⁷¹⁾.

Using regular expressions for working with dynamic window titles

The Video

▷ 'Component recognition'
    https://www.qfs.de/en/yt/component-recognition.html

shows the use of regular expressions with window titles starting from minute 13:07.

In a lot of applications the developers do not use unique names and QF-Test keeps recording the same components again and again in different places. Playback with previously recorded components may still work, unless the window geometry changes significantly.

In this case it is very likely that the title of the main window changes frequently, for example to display a version string, a user name, a file name or some other variable information. If you want to keep your tests working and prevent recording multiple variants of this window and all its components, you have to select the respective Window node and edit its Feature attribute to replace the dynamic parts of the title with a regular expression. Be sure to check 'Use regexp'. Now your tests should work again.

Here you can see the use of a regular expression for a component of the CarConfigurator. Its Feature attribute has to start with 'Edit' followed by an optional dynamic part:

Figure 5.8: Using a regular expression in the Feature attribute

QF-Test uses regular expressions in many places. You can find detailed information in section 47.3[(909)] to learn more about how to use them.

Feature for web components

For web components, the Feature is determined according to the following logic:

- HTML elements of class `Frame` or `Document` use the URL as Feature attribute.

- If none of the following special cases apply, the `id` attribute of the HTML element is used, if it exists and is sufficiently unique. Otherwise, the inner text of the HTML element, truncated if necessary, is used as Feature attribute.

- Special cases:

HTML tag name	Value of the Feature attribute
TEXT	Text of the HTML element, truncated if necessary
A	Text of the HTML element (truncated if necessary), otherwise the window title or the URL
FIELDSET	Text of a contained HTML element with tag name "LEGEND"
FORM	Value of the "name" attribute
IMG	Value of the "alt" attribute, otherwise "src", otherwise part of the URL
INPUT	Value of the "name" attribute. For radio buttons with identical names, the value of the attribute "name" with the value of the "value" attribute appended
BUTTON	Value of the "name" attribute, otherwise text of the HTML element, truncated if necessary
LABEL	Text of the HTML element, truncated if necessary
SELECT	Value of the "name" attribute
OPTION	If the option `OPT_WEB_USE_OPTION_LABEL` is set, value of the "label" attribute, otherwise the text of the HTML element, truncated if necessary
OPTGROUP	Value of the "label" attribute
IFRAME	Value of the "id" attribute, otherwise "name", otherwise "src"

Table 5.1: Feature attribute special cases for web components

In rare special cases, the Feature attribute may also receive a value which is not described by the logic above.

5.4.4   Extra features

The table Extra features[825] stores various information useful for the recognition of the component. In the chapter about Extra features[825] you will find a list of the default entries. But you can also add your own via an `ExtraFeatureResolver` (see section 52.1.11[1028]).

Some of the additional features are recorded preventatively and are not usually used for component recognition. This mostly concerns information about the component class, which QF-Test uses to derive the Generic classes[56]. By default, they have the status "Ignore". This can be changed if the original value is of interest in special cases.

Among the Extra features, the `qfs:label*` variants are interesting for recognition.

`qfs:label*` variants

For component recognition labels are very important. There are a number of different types such as the text of the component itself, for example with a button. Or a label component programmatically assigned to another component using `labelFor` for example. Then there are label components close the component, a tooltip or even an icon description.

Up to QF-Test version 6 the best label is saved in the extra feature `qfs:label`.

7.0+ From QF-Test version 7.0 all labels identified for a certain component will be stored in the extra features, starting with `qfs:label` and a string showing the type, for example `qfs:labelText` for the text of a button or `qfs:labelLeft` for the label left of a text field. The advantage of the specific `qfs:label*` types is replay performance on the one hand, because QF-Test can search directly for the specific label, and flexibility on the other hand.

Note When you want to use the new algorithm for the extra feature `qfs:label` on existing component nodes please change the name to `qfs:labelBest`, telling QF-Test to look for the best of the available labels. You will find more information about the transition in the chapter The `ExtraFeatureResolver` Interface[(1028)].

The following table shows the available positional labels:

qfs:labelTopleft	qfs:labelTop	-
qfs:labelLeft	the component	qfs:labelRight
-	qfs:labelBottom	-

Table 5.2

The following list shows the available `qfs:label*` types and the qualifier to be used when you want the address the component directly via SmartID[(73)]. Descriptions are below the list.

The entries in the column "Category" correspond to the terms used in the section "qfs:label" in Generic classes[(1186)].

qfs:label variant	SmartID qualifier	Category
qfs:labelText	#text=	Own text
qfs:labelFor	#for=	Associated label
qfs:labelLeft	#left=	Label close to it
qfs:labelTop	#top=	Label close to it
qfs:labelTopleft	#topleft=	Label close to it
qfs:labelRight	#right=	Label close to it
qfs:labelBottom	#bottom=	Label close to it
qfs:labelInherited	#inherited=	Label close to it
qfs:labelTooltip	#tooltip=	Tooltip
qfs:labelImage	#image=	Description of icon
qfs:labelTitle	#title=	Title
qfs:labelPlaceholder	#placeholder=	Prompt

Table 5.3: `qfs:label*` variants

qfs:labelText
> The text of the component itself.

qfs:labelFor
> Text of a label assinged to the component in the code. For example via `labelFor` with web applications.

qfs:labelLeft
> The text of a 'Label' component to the left of the component.

qfs:labelTop
> The text of a 'Label' component above the component.

qfs:labelTopleft
> The text of a 'Label' component top left of the component.

qfs:labelRight
> The text of a 'Label' component to the right of the component.

qfs:labelBottom
> The text of a 'Label' component beneath the component.

qfs:labelInherited
> The text of a 'Label' component for a different component. Example: "Street: Main road 11", street name and number are split into separate TextFields. The field for street number here receives "qfs:labelInherited" with the value "Street:".

qfs:labelTooltip
> The tooltip of the component itself.

qfs:labelImage
> The name of the icon belonging to the component.

qfs:labelTitle
> The title of the component, for example of a Window or a titled panel of the component class "Panel:titledPanel".

qfs:labelPlaceholder
> Only web applications. The placeholder showing when no text has been entered by the user.

The influence of the extra feature `qfs:label*` variant representing Best label[69] on the QF-Test component ID is described in Generating the component QF-Test ID[904].

For information about switching from the old to the new algorithm please refer to The `ExtraFeatureResolver` Interface[1028].

Best label

When analyzing a component QF-Test looks for different types of labels which might be used for the component. The labels found will be saved in the Extra features table, the names starting with `qfs:label` (see table 5.3[68]). The one with the best ranking (best label) will get the status "Should match", the others "Ignore". The order of the entries in above table rouhly represents the ranking for most component classes. The exact order can be found in the section "qfs:label*" of the properties of the Generic classes[1186]. Distance and overlapping also have an influence of the ranking with `qfs:label*` variants of the category "Label close to it".

The value of the label ranking highest will additionally be stored as an extra feature with the name `qfs:labelBest` and the state "Ignore". In a SmartID this extra feature can be referenced by the qualifier `qlabel`. See also SmartID syntax for Extra features[77].

qfs:text

`qfs:text` contains the text of the component itself. For text fields or PDF components, this is an additional information which could not be used for component recognition without an additional resolver up until QF-Test version 5.3.

value

Web For certain HTML elements like checkboxes and radio buttons, `value` will contain the value of the HTML attribute of the same name, as long as it is distinct. The `value`

attribute does not reflect the currently selected value or even the selection state of the element, but the static value which would be transferred if the element is selected.

5.4.5 Index

The index of a component can also be used for recognition. However, you need to differentiate between the Class index[(828)] of a Component node[(71)] and the index used in a SmartID[(73)]: The first always refers to GUI elements of this Class name in reference to the parent component. In the case of SmartIDs, the index refers to the eligible components for the specified SmartID (see SmartID with index[(79)]).

5.4.6 Geometry

Geometry only has a small part in component recognition if other criteria are available. It also is possible for a component to neither have a name, nor a label or usable extra features or index. If then even an application-specific resolver (see section 52.1[(1016)]) cannot provide any recognition criteria, the recognition will rely on the component class (which is always available), the component hierarchy, and position and size of the component.

If in this case you take care that the window sizes during replay are the same as during recording (see Component event[(695)]), the component recognition should be stable. However, the modification effort for version changes of the application can be somewhat higher, since position changes of components must be explicitly traced.

5.4.7 Component hierarchy

The nesting is used in Component event[(695)] for recognition as well.

With Component nodes, the containers of a component are recorded as well. Whether they should be used for recognition with a present Name attribute can be controlled with the options Name override mode (replay)[(472)] and Name override mode (record)[(450)]. The best setting for each is described in About setting identifiers[(61)].

Component Nodes can be moved from deep nestings to higher-level nodes in the component tree, as long as this flattening of the component hierarchy does not affect recognition. Sometimes this approach can even lead to better recognition stability, especially if the recognition criteria for the parent nodes are unstable, or if parent components do not always exist (for example scroll panels that are inserted into the GUI hierarchy only as needed).

With a SmartID[(73)], the component hierarchy can also be used for recognition. Here, the to be used component (or even multiple components) is explicitly specified in the

SmartID. More information can be found in SmartID syntax for component hierarchies[(79)].

Via Scope[(80)], the search area for components can be limited to a certain level of the hierarchy during replay.

5.5   Component node

When Component[(823)] nodes are used in place of SmartID[(73)]s QF-Test stores the recognition criteria of the recorded components in Window[(812)] and Component[(823)] nodes, whose hierarchical order matches the structure in the GUI of the SUT. These nodes are located below the Windows and components[(835)] node. The following image shows a section of the Component nodes that are part of the QF-Test main window:

Test suite components.qft

Q Test suite
> Test set **Test**
> Procedures
> Extras
∨ Windows and components
  ∨ Window SuiteFrame
    ∨ Panel treeAndTerminalPane
      ∨ Panel treeAndDetailPane
        ∨ SplitPanel split
          ∨ Panel treeTitlePanel.scroll
            ∨ Tree SuiteTree
          > Panel detailTitlePanel
    > MenuBar menubar
    > ToolBar toolbar
    > StatusLine StatusLine
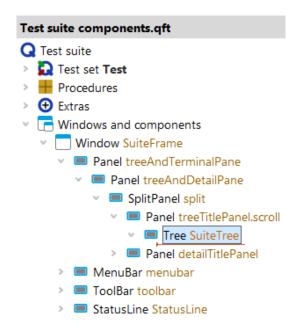
Figure 5.9: Component hierarchy of a Swing SUT

In the detail area of a Component[(823)] node, the Recognition criteria[(56)] are stored. In addition, it contains the QF-Test ID attribute. This is the reference ID for all nodes in the tests that refer to that component.

Figure 5.10: Component node

Each node in a test suite has a <u>QF-Test ID</u>[(824)] attribute, which does not have any special meaning and is managed automatically for most nodes. For Component nodes on the other hand, the QF-Test ID has an important functionality. Other nodes with a target component, like events or checks, have the attribute QF-Test component ID, which refers to the QF-Test ID of the Component. This indirect reference of GUI elements is very useful: If the interface of the SUT changes in a way that QF-Test cannot automatically compensate for, only the Component nodes of the unrecognized components need to be

adjusted. Then, the test will run again.

It is very important to understand that the QF-Test ID of a Component is only an artificial concept for the internal use in QF-Test, not to be confused with the attribute Name, which serves to identify the component in the SUT, which will be explained further in the next section. The actual value of the QF-Test ID is completely irrelevant and has no relation to the GUI of the SUT. The only important thing is that the QF-Test ID is unique and that other nodes correctly refer to it. On the other hand, the QF-Test ID of the Component node is displayed in the tree view, and not only for the Component itself, but even for events and other nodes that refer to it. That is why Components should have expressive QF-Test IDs that indicate the actual component in the GUI.

When QF-Test creates a Component, it has to automatically assign it a QF-Test ID. It does its best to construct a meaningful identifier from the available information. Details about this can be found in Generating the component QF-Test ID[904]. Should a generated QF-Test ID not be to your liking, you can change it. If you choose a value that is already taken QF-Test will output a warning. If you have already recorded events referring to this component, QF-Test offers to automatically adjust their QF-Test component ID attribute. This automatic mechanism does not work for references with variables in the QF-Test component ID attribute.

Note   A frequent mistake is to change the attribute QF-Test component ID of an event instead of the QF-Test ID itself. This destroys the connection between event and its target component, leading to an `UnresolvedComponentIdException`[857]. So, you should only do this if you actually want to change the target component.

Frequently, tests are assembled from existing procedures. For this it is often helpful to use the process described in Recording components[39]. The QF-Test component ID recorded this way will be stored in the operating system clipboard to be easily inserted into the corresponding procedure parameter.

Using Recording components[39] you can also first create the entire component structure of the SUT to get an overview and assign sensible QF-Test IDs. During recordings, theses QF-Test IDs will then be used.

5.6 SmartID

6.0+   SmartIDs enable simple and flexible recognition of components based directly on the ID without storing recognition criteria in a separate place. This noticeably slims down the recorded component tree in "Windows and Components". When using SmartIDs only, the component tree is not used anymore at all. However, you have to consider the price of this flexibility and ease and the possible - depending on the situation - impact on performance and maintainability.

SmartIDs use the same recognition criteria which are stored during classic component

recognition in a Component node[71]. The difference is that, of all the possible recognition criteria, one or multiple are explicitly selected and entered in place of the reference to a recorded Component, for example, directly in the attribute QF-Test component ID[682] of a mouse click node.

The goal of SmartIDs is to slim down the component tree - which is useful, but not at all costs. The idea is to keep simple things simple, but if addressing a component gets difficult, Component[823] nodes are preferable. As an alternative you can handle issues of uniqueness or performance via the scope concept as shown in Scope[80].

The SmartID is characterized by a leading #. The simplest version of a SmartID is either the name or the label of a component with a prefixed #. For example, `#username` to select a component with the name `username` or `#User name` if `User name` is the label of the component.

Typically, the SmartID consists of #, followed by the class of the component delimited by a colon. When the SmartID value is a label, a qualifier and = come after the colon, then the value of the SmartID. For example `#TextField:left=username`.

The qualifier denotes the type of the SmartID value. When replaying a SmartID without qualifier the option Priority for recording SmartIDs with qualifier[485] sets the priority for the recognition criteria. The following qualifier exist:

- `name`: Name[58], see as well SmartID syntax for Name[76]. By default, the qualifier will not be recorded.

- `feature`: Feature[63], see as well SmartID syntax for Feature[77]. By default, the qualifier will not be recorded.

- `label`: Feature[63] is a special form, accepting `feature` or one of the `qfs:label*` variants. By default, the qualifier will not be recorded.

- The names of the Extra features[66], respectively their short forms, for example `qlabel` for Best label[69]. See as well SmartID syntax for Extra features[77]. By default, the respective qualifier will be recorded.

Specifying the class and the qualifier help with readability and performance. The option Always record class for SmartID[484] influences whether the component class will be recorded. The option only has effect on components where the class belongs to one of the Generic classes[1186]. With other classes it is mandatory, for example `#DIV:compid`.

The SmartID takes the place of the QF-Test component ID, for example in the QF-Test component ID[682] attribute of event or check nodes. It can, just like the QF-Test component ID, be stored in variables, passed in parameters, or used in scripts. For complex components like tables, lists or trees, the SmartID can also replace the QF-Test component ID. The index describing the child element remains unchanged.

Following a SmartID, a child component can be addressed either via another SmartID or an XPath, see Addressing via XPath and/or CSS selectors[87].

SmartIDs can be used with all client technologies.

As with generic components you need to consider that updating a component is not as comfortable as with a Component node[71]. However, QF-Test provides a powerful "Search and Replace" feature, which can also be used to bulk-modify SmartIDs.

Note For a brief introduction to SmartID, also read our blog post SmartID - The next generation of component recognition .

5.6.1 Use cases for SmartIDs

The application areas are generally the same as those of Generic components[81]. SmartIDs mostly replace generic components and are easier to use.

Readability
When directly recording test cases, the use of SmartIDs can make recorded event and check nodes more readable. Especially if the recorded component names are cryptic and stable labels are available, it makes sense to change the recording order of recognition criteria to "First label, then name" by setting the option Priority for recording SmartIDs with qualifier[485] to `label,name`.

Ignoring the component hierarchy
Some applications have deeply nested component hierarchies. SmartIDs make it easy to reduce the component tree, which is especially helpful if the hierarchy is not stable across versions. (Until now, Generic components[81] were used in these cases. This is still possible, even in parallel to SmartIDs.)

Test-driven development
For test-driven development, SmartIDs offer the big advantage of not having to create any Component node[71] nodes. In addition, Component identifiers[59] are often defined in the technical design during test-driven development. These can then be used for test creation.

Keyword-based tests
Keyword-based tests are implemented via procedure calls and parameters. The test creator does not record components and depends on visual information from the GUI to identify components. This could be the component label or its function (class). Further information can be found in Keyword-driven testing with QF-Test[352].

Integration with other testing tools
When controlling test execution in QF-Test via other testing tools like Robot Framework, the recognition criteria can be specified directly via SmartIDs.

5.6.2 SmartID syntax for Class name

The Class[56] is specified in the SmartID directly after the # and followed by a :, for example `#Button:`.

You do not need to specify the class in the SmartID when you address a component typically used in tests. It is sufficient to specify the Component identifiers[59] or a component label (either Feature[63] or one of the `qfs:label*` variants[67]), for example `#btnOK`, where "btnOK" is the identifier of the button, or `#Save`, where "Save" is the label of the Button. It makes the handling of the SmartID easier. However, to a certain extent at the expense of performance, as without a specified class QF-Test has to check more candidates for matches.

Because of better performance QF-Test records the class with the SmartID by default. If you want to suppress it, please set the option Always record class for SmartID[484] to 'false'.

chapter 59[1186] documents the properties for each class, including where you have to specify the class with a SmartID. Any class not mentioned in the chapter has to be specified in the SmartID, too. Example: `#DIV:addresses` where "addresses" is the Name of the DIV element in a web application.

Panels with a label are a special case, being useful for nested SmartIDs (see section 5.6.7[79]) or scopes (see section 5.7[80]). For this reason the class type `Panel:TitledPanel` belongs to the SmartID classes and does not need to be explicitly specified.

If you use a predefined class type in addition to the generic class, you can write this combination as usual, for example `#Button:ComboBoxButton:`. You can find the prefedined class types in Generic classes[1186]. For your own class types any internal colons must be escaped via \, for example `#Panel\:myPanel:`.

You can find more information about the combination possibilities in section 46.3[904]. .

5.6.3 SmartID syntax for Name

A Name[58] can be specified in the SmartID directly after the #, for example `#txtUsername`. If the class of the component belongs to the Generic classes[1186], just stating the name is sufficient. Otherwise, the Class[56] must be prefaced, for example `#DIV:txtUsername`.

The name can contain SmartID-specific special characters , but they must be escaped with a prefixed \.

To force component recognition to refer to the Name[58], the SmartID can be prefixed with `Name=`, for example `#Name=txtUsername`. Upper/lower casing does not matter for `Name=`.

You can find more information about the combination possibilities in section 46.3[904] .

5.6.4 SmartID syntax for Feature

The recognition criterium Feature[63] can be specified in the SmartID directly after the #, for example `#User name`. If the class of the component belongs to the Generic classes[1186], just stating the name is sufficient. Otherwise, the Class[56] must be prefaced, for example `#DIV:User name`.

The feature can contain SmartID-specific special characters , but they must be escaped with a prefixed \.

To force component recognition to refer to the Feature[63], the SmartID can be prefixed with `Feature=`, for example `#Feature=User name`. Upper/lower casing does not matter for `Feature=`.

You can find more information about the combination possibilities in section 46.3[904] .

5.6.5 SmartID syntax for Extra features

Recognition criteria from the group of Extra features[66] are also available for SmartIDs. They can be referenced via qualifiers inserted before the SmartID value. An equal sign = separates the qualifier and the SmartID. For all Extra features the name of the extra feature corresponds to the qualifier. They are case-sensitive. The SmartID value corresponds to the value of the extra Feature, also case-sensitive.

Examples:

- The SmartID `#module=module1` references a component with an extra feature named `module` and the value `module1`.

- The SmartID `#my\:foo=Any\&thing` references a component with an extra feature named `my:foo` and the value `Any&thing`.

Short forms exist for the qualifiers for `qfs:label*` variants. They will be explained further down in the chapter.

SmartID specific special characters ":", "@", "&" und "%" (see SmartIDs - special characters[905]) in the value, the qualifier or the class name of the SmartID have to be escaped with a prefixed \.

You can find more information about the combination possibilities in section 46.3[904] .

Extra feature `qfs:label`
    `qfs:label*` variants[67] representing the labels of a component have prominant

role for component recognition. When a component has labels use can use either the Best label⁽⁶⁹⁾ or a specific label. The advantage of a specific label is performance at replay, because QF-Test knows which label to go for and does not have to check all possibilities. When you want to reference a specific label you need to write the hash tag, `#` then the qualifier, i.e. the short form of the name of the Extra feature (see `qfs:label*` variants⁽⁶⁸⁾), followed by "=" and the SmartID value, for example `#left=First name`. You can address the best label directly after the hash tag or via the qualifier `qlabel=`. When you write `#label=`, the value of the SmartID can either refer to the Feature or one of the `qfs:label*` variants. The qualifiers are not case-sensitive. SmartIDs without a qualifier, for example `#First name` will be evaluated following the priority set via the option Priority for recording SmartIDs with qualifier⁽⁴⁸⁵⁾, by default Name - Feature - 'Best label'.

Beispiele:

- `#left=First name` - The label to the left of the component has to be "First name".

- `#qlabel=First name` - The Best label⁽⁶⁹⁾ for the component has to be "First name".

- `#label=First name` - Either the Best label⁽⁶⁹⁾ for the component or the Feature has to be "First name".

- `#First name` - Either the name or the Best label⁽⁶⁹⁾ for the component or the Feature has to be "First name".

Extra features `qfs:text` and `text`

The extra features `qfs:text` and `text` have a special status as well. Both can be addressed via the qualifier `text=`. If you want to explicitly use `qfs:text` you can use `qtext=`.

Examples: `#text=Anna`, `#qtext=Benno`

Looking closely, with the qualifier `#text=` QF-Test will first look for the Extra feature `#qfs:labelText=`, then for `qfs:text` and `text`. As the latter two have a special use with text components for which nor `#qfs:labelText=` will be recorded, there won't hardly be any conficts.

If you want to use the extra features `qfs:text` and `text` without the prefix `#text=` you need to set the option Priority for recording SmartIDs with qualifier⁽⁴⁸⁵⁾ accordingly, for example to "name,feature,qlabel,text".

Extra Feature `qfs:type`

The extra feature `qfs:type` denotes the type of a class. If a type is not predefined by QF-Test (see Generic classes⁽¹¹⁸⁶⁾) any colons contained within must be escaped via `\`.

5.6.6 SmartID with index

All SmartIDs can be equipped with an index in case multiple components match the same SmartID. For this, the technical order of the components in the hierarchy counts. This does not have to be the same as the visual order. The count of the index starts at 0. The index is specified in between angled brackets. If no index is given, 0 is used implicitly.

Examples: `#Name<2>`, `#TextField:<2>`

Special cases

For components of the class `Label`, the standard order does not apply. Because they are mostly used as label for other component classes and are stored there in the feature or extra feature `"qfs:label"`, components of the class `Label` are treated as subordinate. Label components must be addressed explicitly with the prefixed class `Label:`, for example `#Label:First Name`.

You can find information about the SmartID syntax in general in <u>section 46.3</u>[(904)].

5.6.7 SmartID syntax for component hierarchies

The <u>Component hierarchy</u>[(70)] can also be used with SmartIDs for recognition. As divider between hierarchy levels, `@` is used.

Examples:

Component inside container
    The SmartID `#Customer information@#Name` references a component with the SmartID `#Name` in a parent component (like a `TitledPanel`) with the SmartID `#Customer information`.

Component inside "normal" component
    Sometimes, components like a Button will not have any good recognition criteria themselves but can be addressed well via their parent component. A typical example is the Button for expanding the list of a combo box: `#ComboBoxSmartID@#Button:`

Component inside Sub-Element
    Links or buttons inside list oder table elements can be addressed with nested SmartIDs: `#ListSmartID&22@#Link:<1>` Here, the part before "@" addresses a list element, `#Link:<1>` addresses the second link inside.

5.6.8 Recording and replaying SmartIDs

To record SmartIDs instead of component notes, please activate the option SmartID recording[(484)] or simply check the menu item Recording→SmartID recording.

When recording SmartIDs, QF-Test first checks if a Name[(58)] is present. If it is, it will be used for the SmartID. If not, QF-Test searches for a label (in Feature[(63)] or Extra features[(66)]). If the determined SmartID is valid for multiple components, an Index is attached.

Note

For many cases recording SmartIDs is straightforward. However, depending on the target component and the information available it may happen that no SmartID can be recorded so that a classic Component node[(71)] node gets recorded instead. This is the case if, for example, the GUI element cannot be assigned a generic class or if QF-Test can determine neither a Name[(58)] nor a Feature[(63)] nor the extra feature `qfs:label*` variants[(67)].

By default the generic class is prefixed to the recorded SmartID. This not only improves readability, it also has a significant effect on replay performance. This can be turned off via the option Always record class for SmartID[(484)]. Please note that the 'Label' or 'Panel' prefixes are always recorded to ensure correct replay.

Replaying nodes with SmartIDs is no different than with recorded components. Both variants can be used inside the same test case. SmartIDs can also be used in combination with recorded components to address descendant components. The example `recordedList&10@#Button:` illustrates the combination of the QF-Test ID of a recorded list with an Index and the SmartID of the Button contained inside the list element.

5.6.9 Component QF-Test ID as SmartID

It is possible to set the QF-Test ID of a recorded Component to a SmartID icluding prefixed `#`. This can be used to essentially reroute this SmartID and perform the recognition via the classic recognition criteria of the recorded component. Recording individual components makes the most sense if the SmartID gets long and cumbersome, has bad performance, or is hard to make unique. The SmartID indicator `#` can be used for consistency's sake, but it does not have to be.

5.7 Scope

6.0+

A scope can be used to narrow the search area for components. This is useful to make component references unique or to improve the readability of a test. Example: There

are three panels with address data with identically labeled text fields. The scope can now be set to one of the panels. Now the specified SmartIDs refer exclusively to the fields in this panel.

Scope can also be used to speed up component detection under certain circumstances, especially for windows or web pages that contain a large number of components. An example of this is web applications that load all GUI elements with the status "invisible" from the start, and only make the relevant ones visible. Here it can be useful to use the scope to limit component detection to at least the visible window.

The scope is set in the comment of a node, via the SmartID or even the QF-Test ID of the recorded component with prefixed `@scope`, for example `@scope #myDialog`. If the scope should apply to multiple event or check nodes, the scope is set in the comments of a node (for example Sequence, Test step, or Test case) which contains these nodes directly or indirectly via procedure calls.

The scope currently active can be referenced via a SmartID consisting just of the hash symbol `#`.

If a component is not part of the scope, a `ComponentNotFoundException` will occur. Scopes can be bypassed when needed by inserting the doctag `@noscope` in the comments of the respective event or check node or by inserting `noscope:` at the start of the SmartID. For example, its possible to click a button "Save" via the SmartID `#noscope:Save`, even though the button is outside of the scope set for the sequence in which the click event is located (see).

Scopes can be nested, whereby the inner scope must lie in the outer scope and further restrict it. This, too, can be bypassed with the doctag `@noscope`. To do this, the doctags `@noscope` and `@scope NEWSCOPE` are specified in the comments of the node whose components lie in a scope outside the current one (NEWSCOPE denotes the new scope). The order of the doctags does not matter.

The scope always refers only to the respective nodes and nodes executed within it. Therefore the nodes of a procedure called from inside a scope must either be inside this scope or be marked with `#noscope:...` or doctag `@noscope`.

Scopes can be set via SmartIDs or via the QF-Test ID of a recorded Component. Still, they are only respected when referencing a component via SmartID. Referencing a recorded Component will always ignore the current scope.

5.8 Generic components

Before the introduction of the SmartID[73] in QF-Test Version 6.0, generic components were the pattern of choice for avoiding recording components. With SmartIDs, this goal can be achieved easier and more flexibly. Still, the concept of generic components is

described here for backwards compatibility.

A typical use case is the testing of localized applications.

Another situation could be the use of a GUI framework during development. This generates, for example, a lot of similar dialogs which differ by only a few components. But you must re-record them for each dialog, for example global navigation buttons, because they are located inside a new window each time.

With generic components, you use variables in the component properties or simply delete non-dynamic parts of them.

The following is a general approach for generalizing components:

1. Record some components you want to generalize and compare them.

2. Create a new generic component with 'generic' in the QF-Test ID so you can find it again later.

3. Remove all attributes you do not want to use for recognition from this generic component.

4. Define the recognition criterium, like 'name', 'feature' or 'index'.

5. Place a variable in this attribute, for example $(name).

6. To avoid false positives, deactivate geometry recognition by placing a '-' in the 'X' and 'Y' attributes.

7. Specify '@generic' in the Comment attribute so this component is not inadvertently removed by the 'Remove unused components' action.

8. Create a procedure for accessing this generic component and use the variable from before as procedure parameter.

Note   Generic components are very useful for replay of tests, but QF-Test does not use them for recording. It always records concrete components and you need to manually replace these with generic components afterwards.

5.9 Sub-items: Addressing relative to a parent component

In QF-Test it is possible to address components in relation to a parent component. This is most interesting if the child component can only be unequivocally addressed in combination with its parent. There are various usage scenarios for this and also various ways to implement.

Addressing via index

With tables, lists, and trees it makes sense to use an index for sub-items. The main component is specified via the QF-Test component ID or a SmartID. The index for the sub-item is appended.

Examples: `listid@Entry`, `#Table@Column Heading&5`

If the main component is addressed via a SmartID, tabs in a `TabPanel` or list items of a `ComboBox` can be referenced simplified.

Examples `#Tab:Tab1`, `#Item:EntryX`

More information can be found in Addressing via index[84].

Addressing sub-items via SmartID

SmartIDs can be attached to the QF-Test component ID or to the SmartID identifying the parent component. As divider between parent and child component, `@` is used. The nesting can also be multi-level. The individual components can also be given an index.

Examples: `#Dialog:@#OK`, `comboboxid@#Button:`, `#Table:&0&0@#CheckBox:`.

Addressing sub-items via QPath

A QPath can be used simliarly to the attached SmartID, but is not as powerful as it by far. A QPath can be attached to a QF-Test component ID. As divider, `@` is used. The QF-Test component ID can also be given an index.

Examples: `buttonid@:Icon`, `tableid&0&0@:CheckBox`

More information can be found in Addressing via QPath[86].

Addressing sub-items via XPath and CSS selectors

Web

With web applications, an XPath and/or a CSS-Selektor can also be appended to a QF-Test component ID or a SmartID. As divider `@:xpath=` or `@:css=` is used. The QF-Test component ID can also be given an index.

Example: `genericDocument@:xpath=${quoteitem:$(xpath)}`

More information can be found in Addressing via XPath and/or CSS selectors[87].

Scope

The parent component can also be specified via a scope, see Scope[80].

Recording sub-items as nodes

In the case of tables, lists and trees it can also make sense to record the sub-item as Item[829] node. It depends on the situation if a sub-item is addressed via index or if it makes more sense to record it. You can use both methods as preferred and even combine them. The rule of thumb is that an Item node is better for components with few, constant elements, like columns of a table or tabs

in a tab panel. The syntax is preferrable if QF-Test variables are used in indexes or if the names of elements vary or are editable. The option Sub-item type⁽⁴⁵⁴⁾ determines if QF-Test creates Item nodes during recording or uses the QF-Test ID syntax. With the default setting "Intelligent", QF-Test follows the rules above.

More information about recording sub-items can be found in Addressing via Items nodes⁽⁸⁸⁾.

Possible combinations

Note: In the following listing, the SmartID of the parent component may already consist of nested references.

Reference of the parent component	Reference of the child component	Example
QF-Test component ID	Index	list item with text index: `listid@Entry`
SmartID	Index	table cell with numeric indexes: `#Table:&0&2`
QF-Test component ID	SmartID	Icon in button: `buttonid@#Icon:`
SmartID	SmartID	Text field in dialog: `#Dialog:@#TextField:`
SmartID with index	SmartID	Button in table cell: `#Table:&0&2@#Button:`
QF-Test component ID with index	SmartID	Button in table cell: `tableID&0&2@#Button:`
QF-Test component ID with index	QPath	Button in table cell: `tableID&0&2@:Button`
QF-Test component ID with or without index	XPath and/or CSS selector	`genericHtml@:css=${quoteitem:$(css)}`
SmartID with or without index	XPath and/or CSS selector	`#genericDocument@:xpath=${quoteitem:$(xpath)}`
Scopes	SmartID	Scope as doctag in a Test step, SmartID in check node

Table 5.4: Addressing sub-items

5.9.1 Addressing via index

The sub-item is described using a special syntax. The QF-Test component ID which is used in the test consists of the QF-Test ID or the SmartID of the complex component (tree, table, etc.), followed by a special separator and the index of the sub-item. The kind of separator determines if the index is numeric, textual, or a regular expression (see section 47.3⁽⁹⁰⁹⁾):

Separator	Index format
@	Text index
&	Numeric index
%	Regular expression

Table 5.5: Separator and index format for accessing sub-items

To access a table cell with the Primary index and the Secondary index, simply append another separator, followed by the second index. The two indices can be different types.

With trees the index consists of the path of tree nodes leading to the node you want to address. The nodes are separated by a valid index separator, followed by "/". When a separator holds for more than one tree node in a row, you do not need to repeat it before each "/".

Note The special meaning of the separators '@', '&' and '%' makes them special characters which must be escaped if they appear in the index itself. More about the topic can be found in Quoting and escaping special characters[(912)].

Negative index
   In most cases you can use a negative index to start the count from the end.

SmartIDs: Easy indexes for TabPanels and lists
   With SmartIDs, tabs in tab panels can be addressed according to above syntax, for example via `#TabPanel:@Tab1`, where `Tab1` is the name of the tab. Alternatively, the shortcut `#Tab:Tab1` can be used. If no other component has the SmartID `#Tab1`, the tab can even be addressed by `#Tab1`.

   List entries can be addressed according to above syntax via `#List:@EntryX`. As a shortcut, `#Item:EntryX` is possible, too, or just `#EntryX` when no other component on the page has the SmartID `EntryX`. The shortcuts can also be applied to drop down lists of combo boxes.

   Both shortcuts are very comfortable, but the high flexibility has its price in performance. How much it is depends on a number of factors, so the decision between comfort and performance must be made on a case-by-case basis.

Examples

Component	Index	Comment
Table	`@Name&5`	Table cell in the sixth row and the column with the title "Name". Full QF-Test ID of the component: `tableid@Name&5`
List	`&0`	Numeric index: first entry in a list. Full SmartID: `#List:&0`
List	`@Europe`	Text index: list entry with the text "Europe". Full SmartID: Default syntax: `#List:@Europe` Shortcut 1 (alternative): `#Item:Europe` Shortcut 2 (alternative) when no other component has the SmartID `Europe`: `#Europe`
Tree	`@/root/b1/b1-2/leaf`	Text index: tree path addressing all nodes via their text. Full QF-Test ID of the component: `treeid@/root/b1/b1-2/leaf`
Tree	`&/0/5/1/3/`	Numeric index: tree path addressing all nodes via the numeric index. Full SmartID: `#Tree:&/0/5/1/3/`
Tree	`%/W.*/A.*`	Regular expressions for the tree nodes.
Tree	`&/0@/Ast1%/B.*`	Mixed indices: numeric index for the first, text index for the second and regular expression for the third node.
Table	`&-1&-1`	Negative indices: bottom row, right-most column.
TabPanel	`@Tab1`	Text index: addressed via the tab label. Full SmartID: Default syntax: `#TabPanel:@Tab1` Shortcut 1 (alternative): `#Tab:@Tab1` Shortcut 2 (alternative) when no other component has the same SmartID: `#Tab1`

Table 5.6: Indices for sub item

5.9.2 Addressing via QPath

Each QF-Test component ID attribute in an event or check node (with or without sub-item) can be appended one or more indexes in the form of `@:ClassName<idx>`, whereby `<idx>` is optional. This instructs QF-Test to first determine the target component (and if needed the sub-item) for the part of the QF-Test component ID attribute in front of the `@:` and then to search for visible components of class ClassName within. If `<idx>` is specified, this is interpreted as 0-based index of the list of visible candidates. No `<idx>` is equivalent to `<0>`.

The QPath syntax expects a generic class after the `@:`. An overview over generic classes can be found in <u>chapter 59</u>[(1186)]. If the component cannot be recorded with a generic class, the QPath must contain the complete class name. In JavaFX for exam-

ple, some of them are called `ImageView`, `VBox`, `GridPane`, or `BorderPane`.

The following example references the second `ImageView` on the third position of a list:
`panelSecond.list&3@:javafx.scene.image.ImageView<1>`

5.9.3 Addressing via XPath and/or CSS selectors

XPath and CSS selectors are standardised formats for addressing elements in web browsers. (Official specifications: www.w3.org/TR/xpath and www.w3.org/TR/css3-selectors).

QF-Test supports addressing components via XPaths and CSS selectors for web elements, to allow for easier migration of existing web tests of other tools into QF-Test.

There are already a lot of tutorials on the internet on how to address elements with CSS selectors (for example w3schools CSS Selector Reference) and with XPaths (for example w3schools XPath Syntax). Because of this, the peculiarities of these ways of addressing components are not described here.

Use in the QF-Test ID

Assuming that a web component is to be recognized in QF-Test using the XPath "$(xpath)" or a CSS selector "$(css)", this can be done in several ways. The easiest/fastest way is usually to specify the XPath or the CSS selector in the QF-Test component ID[(682)] attribute of any event node. The following syntax is used for this:

```
genericHtml@:xpath=${quoteitem:$(xpath)}
genericHtml@:css=${quoteitem:$(css)}
```

Or to the same effect:

```
genericDocument@:xpath=${quoteitem:$(xpath)}
genericDocument@:css=${quoteitem:$(css)}
```

The syntax can be nested as needed. For example, you can use:

```
genericDocument@:xpath=${quoteitem:$(xpath)}@:css=${quoteitem:$(css)}
```

to direct QF-Test to first search for a component using an XPath and then search for a child component using a CSS selector.

Note Please note that the `@:xpath`/`@:css` syntax understandably expects the given XPath/CSS statement to return a single component. Using an XPath which returns a number (for example `count(.//input[@id!='Google'])`) or a boolean (for example `nilled($in-xml//child[1])`) can lead to unexpected behavior.
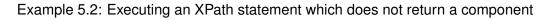
Use in scripts

The `rc` module in SUT scripts also allows to find web components via XPath or CSS selectors.

```
com = rc.getComponent("genericHtml") # or rc.getComponent("genericDocument")
res = com.getByXPath(rc.lookup("xpath"))    # find subcomponent via xpath
res = com.getByCSS(rc.lookup("css"))        # find subcomponent via css
res = com.getAllByXPath(rc.lookup("xpath")) # find all subcomponent via xpath
res = com.getAllByCSS(rc.lookup("css"))     # find all subcomponent via css
```

Example 5.1: Finding components by XPath or CSS selectors in scripts

To use an XPath which does not return any component(s), please use the `callJS` method:

```
node = rc.getComponent('genericDocument')
print node.callJS("""return document.evaluate("count(.//input[@id='Google'])",
  document, null, 0, null).numberValue;""")
```

Example 5.2: Executing an XPath statement which does not return a component

Use in component nodes

Inside a component node QF-Test can also be instructed to use an XPath or CSS selector for component recognition. To do this, specify a recognition criterium like the following in "Extra features":

Status	must match
Regexp	No
Negate	No
Name	qfs:item
Value	@:xpath=${quoteitem:$(xpath)} oder @:css=${quoteitem:$(css)}

Figure 5.11: Extra feature attribute for component recognition via XPath or CSS selector.

5.9.4 Addressing via Items nodes

An Item is defined by two things: The component it belongs to and an index inside the component. The parent node of the Item defines the component. The index can be either a number or a text. Numeric indexes start with 0. For example, in a `JList` component the element with index 1 corresponds to the second list entry. For trees,

simple numeric indexes are almost useless, since by opening and closing branches, the indexes of all nodes below are changed.

A text index defines an element by the text it displays in the interface. a list item called "Entry1" in a `JList` component would be recorded with the text index "Entry1". The textual representation is more flexible than the numeric one, but can cause problems if the displayed texts of the elements in a component are not unique. In these cases, the first matching element is selected. A text index can also be a regular expression (see section 47.3[909]). In this case, the first element matching the expression is selected.

The option Sub-item format[454] determines which format is used during element recording.

Almost all kinds of Item have only one index. This is not sufficient for the cell of a `JTable` component, since tables are two-dimensional structures. Two indexes are needed to exactly describe a cell. The first, the Primary index[830], determines the table column, the second, the Secondary index[830], the row.

Figure 5.12: An Item for a table cell

Tree nodes also occupy a special position. As described above, the hierarchical structure cannot be easily mapped to a linear structure. In addition, tree nodes with the same names often occur in trees. If, on the other hand, the direct and indirect parent nodes

are included in the name, uniqueness can usually be achieved.

QF-Test uses a special syntax to represent tree nodes. An index starting with a '/' character is interpreted as a path index. Just think of a file system: The file named "/tmp/file1" can either be represented flat as "file1", which can lead to conflicts with other files named "file1" in other directories. Alternatively, the full and unique path "/tmp/file1" can be used. QF-Test also supports numeric indexes with this syntax: A numeric index of the form "/2/3" denotes the fourth child node of the third child node of the root node. A combined form to address the third node in the "tmp" node using "/tmp/2", for example, is currently not possible.

Note   This special syntax makes '/' a special character for Items in a tree component. If this character appears in a name itself, it therefore must be escaped. More about this topic can be found in <u>section 47.5</u>[(912)].

Everything said in <u>section 5.5</u>[(71)] about the QF-Test ID attribute of Components also applies to the <u>QF-Test ID</u>[(830)] attribute of an Item. This attribute must be unique and is referenced by events and checks.

When QF-Test automatically assigns the QF-Test ID of an Item, it creates it by taking the QF-Test ID of the Component of the parent node and appending the index (or indexes). This kind of QF-Test ID is normally well readable and understandable. Unfortunately, it also is the source of a frequent misunderstanding: If you want to change the index of a recorded Item to refer to another element, you must **not** change the attribute QF-Test component ID of the node which refers to it. Instead you must change the Primary index of the Item node.

5.10 Troubleshooting component recognition problems

5.10.1 Timing synchronisation

If you get exceptions because a component was not found, one of the reasons may be that you did not wait for the component long enough. A mouse click has a certain default waiting time, but this is not always sufficient. Therefore you should check if there are enough synchronization points, like Wait for component to appear or Check nodes with waiting times to only execute the test steps if the SUT is really ready for it.

- A Wait for component to appear node can be used when a new component appears. The maximum wait time (in milliseconds) is set in Timeout.

- A Check node is used to wait for a state change of components. The maximum wait time (in milliseconds) is set in Timeout here as well.

- Sometimes it is also necessary to loop, waiting for the state change and, if not already done, performing an action, for example, clicking a "Refresh" button.

- You can also wait for the number of rows in a table to change in a loop.

- Many applications use indicators that symbolize waiting times, for example progress bars or "egg timers". Here you can wait first for the component to appear and then for it to disappear.

The maximum wait time is set in the Timeout attributes. As soon as the desired application state is reached, QF-Test continues execution. These waiting times can therefore be chosen generously.

You should only change the option for default waiting times (section 39.3.6[(477)]) if generally longer waiting times make sense across your whole application.

Note   As a last resort you can also use with a fixed delay. When the attribute Delay before/after is set, QF-Test will wait the entire given time. Delay before/after should therefore only be used if there is no state change detectable by QF-Test in the application which QF-Test could wait for.

5.10.2 Recognition

If your SUT changes in a way that makes it impossible for QF-Test to find a component again, your test will fail with a `ComponentNotFoundException`[(850)]. This should not be confused with an `UnresolvedComponentIdException`[(857)], which can be caused by removing a Component node from the test suite or by changing the attribute QF-Test component ID of an Event node to a non-existing QF-Test ID.

Video   There are two videos that comprehensively explain how to handle a `ComponentNotFoundException`:

- [▷] 'ComponentNotFoundException - Simple case'
  https://www.qfs.de/en/yt/componentnotfoundexception-simple-40.html

- [▷] 'ComponentNotFoundException - Complex case'
  https://www.qfs.de/en/yt/componentnotfoundexception-complex-40.html

When run into an `ComponentNotFoundException`, run the test again with the test debugger enabled so that the test stops and you can examine the node that caused the problem. This is where it pays to have QF-Test ID attributes that are meaningful, because you need to understand which component the test was trying to address. If you can't make any sense of what the node in question is supposed to be, disable it and see if the test goes through without it. It could be a spurious effect that was not filtered

during the recording and that does not contribute anything to the actual test. Basically, your tests should always be reduced to the minimum number of nodes that can be used to achieve the desired effect.

If the node must be preserved, next take a look at the SUT to see if the target component is currently visible. If not, you will need to adjust your test accordingly to handle this situation. If the component is visible, use the screenshot in the log to verify that this was the case at the time of the failure and try executing the failed node again as a single step. If execution now works you have a timing problem which you can solve by including a Wait for component to appear⁽⁷⁷¹⁾ node, a Check node with Timeout, or another waiting action (see Timing synchronisation⁽⁹⁰⁾).

If the component is visible and the replay continuously fails, the reason is a change in the component or one of its parents. Now you must determine what changed and where. For this, record a new click on the component and compare the new and old Component node in the hierarchy below Windows and components⁽⁸³⁵⁾.

Note   You can jump directly from the Event node to the associated Component by pressing ⌐Ctrl-W⌐ or by selecting ‖Find component‖ from the context menu. You can use ⌐Ctrl-Backspace⌐ or ‖Edit→Select previous node‖ to jump back again. A smart move is to denote the Components to be compared using markers with ‖Edit→Mark‖ to easily find them again.

The crux is where the hierarchy of the two nodes branches. If they are located under different Window nodes, the difference is in the respective Windows themselves. Otherwise, there is a common predecessor just above the branch. The crucial difference is then found in the respective nodes directly below this common predecessor. When you have found the place of divergence, compare the attributes of the respective nodes from top to bottom and look for differences.

Note   You can use ‖View→New window...‖ to open another QF-Test window and place the detail views of both nodes next to each other.

The only differences that will always lead to an error during recognition are changes to the attributes Class name or Name. Differences to Feature, structure or geomertry can usually be compensated for, provided they do not accumulate.

A change to the Class name should seldomly happen when using Generic classes⁽¹¹⁸⁶⁾. Using generic classes offers a range of advantages, but in the case of web applications it is sometimes only introduced after creating first tests (see Improving component recognition with a `CustomWebResolver`⁽⁹⁴⁵⁾). In this case you must adapt the Class name attribute of the already created Component nodes to this change.

The Component identifiers⁽⁵⁹⁾ can change, too. If the change seems to be on purpose, for example correcting a grammatical error, you can adjust the Name attribute accordingly. More probably it is an automatically generated Component identifiers⁽⁵⁹⁾ which could change again at any time. Here it can make sense as well to discuss the issue

with the developers and find a solution on the development side. Otherwise, for web applications the Name can be influenced by Install CustomWebResolver node – Syntax[950] via the catagories `autoIdPatterns` and `customIdAttributes`. In all technologies the Name can be influenced using a `NameResolver` as described in section 52.1.7[1023]. It can be suppressed entirely or reduced to the relevant parts.

Changes to the attribute Feature are not unusual, especially for Window nodes. There, the Feature corresponds to the title of the window. Combined with a significant change to geometry, this can cause the recognition to fail. This can be fixed by adjusting the Feature attribute to the new circumstances, or - preferably - by using a regular expression (see section 47.3[909]) which covers all variants.

Depending on the type and scope of the changes, there are two basic options for correction:

- Adjust the attributes of the old node and remove the newly recorded nodes. If the changes to the SUT were small enough and the component recognition still works, changes can also be performed automatically via the QF-Test feature Update Components[94].

- Keep the new nodes and remove the old ones. For this you first must make sure that all nodes that refer to the old components are updated to the new QF-Test ID. This can be achieved with a little trick: Change the QF-Test ID of the old Component node to the QF-Test ID of the new one. QF-Test will initially complain that the QF-Test ID is not unique, which you can ignore, and then will offer to update all references, which you need to confirm with "Yes". Then, you can remove the old node.

Note   The automatic adjustment of references in other test suite only works if they belong to the same project or if the attribute Dependencies (reverse includes)[517] of the Test suite node is set correctly.

5.11   Component tree maintenance

During the course of test creation, some unused components can collect in the component tree. From time to time you can Clean up the component tree[94]. On the other hand, recognition characteristics can change because of changes in the application interface. Before changes accumulate across multiple interface changes and break recognition, it makes sense to Update Components[94] in the affected windows and dialogs.

5.11.1 Clean up the component tree

Each time a sequence is recorded, new nodes are created for the components which are not yet part of the test suite. If the sequence is deleted later, theses Components remain which gives Components a certain tendency to accumulate.

The context menu for Window and Component nodes has two entries called Mark unused components... and Remove unused components, which mark or remove entirely those Components which no other node in this test suite refers to.

Be careful if you use variables in QF-Test component ID attributes since the automatic mechanism does not resolve them.

If Components from other test suites are referenced, these should be part of the same project or the attribute Dependencies (reverse includes)[517] of the Test suite must be set correctly.

5.11.2 Update Components

It is almost unavoidable that components of the SUT change with time. As described, this is not a big problem as long as identifiers were consequently used, since QF-Test can then handle almost every kind of change.

Without identifiers, changes will accumulate with time and can reach a point at which the recognition fails. To avoid this problem, you should adjust the Components in QF-Test to the current SUT from time to time. This can be done with the help of the menu entry Update component(s) which you can find in the context menu of any node below the Windows and components[835] node.

Note This feature can change a lot of information at once, so it can be difficult to judge if everything went well or if a component was recognized wrongly. You should always create a backup copy before updating lots of components. You should also proceed Window by Window and make sure that the components you want to update are visible in the SUT (excluding menu entries). After every step make sure that the tests still run cleanly.

Provided there is a connection to the SUT the following dialog will appear when using this function:
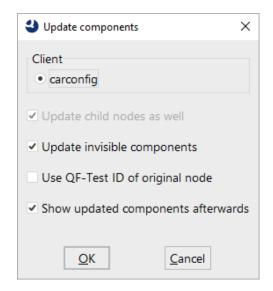
Figure 5.13: Update components dialog

If you are connected to multiple SUT clients you need to select one for the update.

Decide if you only want to update the selected Components themselves or their child nodes as well.

You can also include components that are not currently visible in the SUT. This is most useful for menu entries.

The QF-Test ID[(824)] of an updated node is retained if "Keep QF-Test ID of original node" is selected. Otherwise, the node is given a QF-Test ID generated by QF-Test, if meaningful information is available. Other nodes that refer to this QF-Test ID are automatically adjusted. QF-Test also checks dependencies in the test suites that belong to the same project or that are listed in the attribute Dependencies (reverse includes)[(517)] of the Test suite[(515)] node. These test suites are automatically loaded and indirect dependencies are also resolved.

Note In this case, the modified test suites are automatically opened so that the changes can be saved or undone.

After confirming with "OK", QF-Test will attempt to find the affected components in the SUT and collect current information. Components which cannot be found are skipped. Then the Component nodes are adjusted to the current structure of the SUT GUI, which can also result in nodes being moved.

Note For large hierarchies of components, this extremely complex operation can take some time, up to a few minutes in extreme cases.

This feature is most useful if identifiers are used for the first time in the SUT. If you have already created some tests before convincing developers to assign identifiers, you can

use this to apply these identifiers to your Components and at the same time adjust the QF-Test IDs. This works best if you can get a version of the SUT which is identical to the previous version except for the identifiers.

Note   Very important note: Updating whole windows or hierarchies of components above a certain size often leads to an attempt to update components that are not present or invisible at that moment. In such a case it is very important to prevent false positive hits for these components. You can do this by temporarily setting the '... bonus' and '... penalty' options for recognition (section 39.3.4[471]). In particular, set the 'Feature penalty' to a value below the 'Minimum probability' value, for example to 49, if you otherwise use the default values. Do not forget to restore the original values afterwards.

If you need to change the setting of the Name override mode (replay)[472] and Name override mode (record)[450] options, for example because component identifiers have turned out to be ambiguous, first change only the option for recording. When the update is finished, follow up the option for playback accordingly.

5.12   Inspecting components

3.1+

Sometimes it is useful to get extra information about the components saved in the section Windows and components[835] or to view the saved information interacting directly with the application.

It is particularly relevant when mapping components of web applications. This should be done before starting to write tests, as described in Improving component recognition with a `CustomWebResolver`[945]. The UI inspector can be used to examine the UI elements, currently for Android and web applications.

When working with scripts, it is sometimes helpful to be able to display a list of the methods of a GUI element.

5.12.1   Show methods

Every GUI object has certain (public) methods and fields which can be accessed in a SUT script[631] as soon as it has access to the object (see section 11.2.4[169]). To display these, select the Show methods for component... from the context menu of a node below the Windows and components[835] branch or right-click the component itself in component recording mode (see section 4.4[39]).

Web   The methods and fields that are displayed for HTML elements in a browser cannot be used directly with the object returned by `rc.getComponent()`. They are JavaScript methods and properties that must be embedded in `callJS` (see section 52.11[1118]).

5.12.2 UI Inspector

7.0+

The UI inspector shows the component hierarchy of the client and the properties of every component. This can be useful to resolve component recognition problems. Furthermore, it also makes it easier to set up resolvers thanks to the information displayed in the detailed view.

The UI Inspector is available for Android and Web applications. As of QF-Test version 7.1, Windows and Swing/AWT are also supported.

The node representation in the component tree will already give an overview of the most important information. If the class name (section 5.4.1[56]) is written in blue, the component will be considered as interesting. This in turn will determine whether a Component[823] node will be created for the component. In case a generic class (see chapter 59[1186]) can be determined, the generic class will be represented in bold. Furthermore the original class will be added in brackets. By default, all generic classes will be regarded as interesting. Invisible components will be displayed in grey.

Figure 5.14: UI Inspector

Opening the inspector

In order to open the UI inspector you have the following possibilities:

- Via the menu | Clients→Show inspector |.

- Via the menu entry | Show inspector | in the context menu upon component recording.

- Via the context menu | Show inspector | of any Component node or any node containing a QF-Test component ID or a SmartID[(73)].

Web
- Via the button | Inspector | in the Install CustomWebResolver[(796)] node.

Android
- In the recording window via the crosshairs-button in the toolbar, see QF-Test Android recording window[(236)].

UI Inspector toolbar

The buttons in the toolbar have the following meaning:

Switch inspector mode on and off. In the inspector mode the record- and check functionality of QF-Test is deactivated. Furthermore actions will not be forwarded to the SUT. That way the information in the UI inspector can be analysed via mouse clicks.

Navigating in the component tree. Together with inspector mode the history function also gets activated, thus remembering selections made in the UI inspector and the client, which makes it possible to jump forward and backwards in the component tree.

Refresh component tree. Updating is required when components have changed or an resolver has been installed.

Show invisible components in the component tree. Components with visible childs will always be displayed.

Show geometry information in the component tree.

UI Inspector details

The UI inspector view gives an overview of the most important properties of a component. The view is divided into three sections:

General

This section covers the basic properties of a component, for example its class.

Figure 5.15: General information

Engine-related

The engine-related details cover additional technical information of a component. The information shown differs depending on the technology used.

Web   For web applications it is information about the DOM element, for example the HTML tag, HTML attributes and the readable text.

Figure 5.16: Web-specific information

Android   For Android applications information concerning the content description, resource ID, package name as well as information about the window type and its arrangement is shown.

Figure 5.17: Android-specific information

In Windows applications, the information pertains to the Automation Element and includes the most important UI Automation properties. Detailed descriptions of this information can be found in section 52.13.1[1135].

Figure 5.18: Windows-specific information

Swing In Swing applications, the UI inspector contains information about the Swing component and its key accessible properties as well as the name, tooltip, and client properties.

Figure 5.19: Swing-specific information

QF-Test specific

The information shown in this section relates to the Component node[71] node. The information can be used to verify whether a resolver works as expected.

Figure 5.20: QF-Test specific information

The detail view can also show information about two different components next to each other, making it easy to compare them. So right click one component in the hierarchy tree and choose Compare from the context menu.

Via Reset comparison in the context menu or via the 'Close' button in the detail view you can leave the comparison mode again.

Chapter 6

Variables

Video:

▷   'Variables'
     https://www.qfs.de/en/yt/variables.html

.

Variables are the primary means to add flexibility to a test suite. Though they are used mainly as parameters for Procedures[(586)], they are also useful in many other cases.

Variable syntax can be used in almost every attribute of the nodes of a suite, except for boolean values (check boxes). There are three versions of variable references:

- `$(varname)` expands to the value of some previously defined variable.

- `${group:name}` accesses external data[(108)] from a resource bundle or property file. The groups `system` and `qftest` are always defined and have a special meaning (see section 6.5[(108)]).

- `$[expression]` evaluates some mathematical expressions[(114)].

6.1 Variable lookup
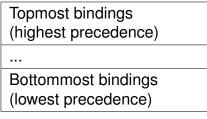
To understand the reasons of why and how variables are defined in multiple places, you first have to learn about how the values of variables are determined.

Each set of variable definitions, called bindings, is placed on one of two stacks of bindings. One stack is used for actual - or direct - definitions and one for fallback bindings or default values. When a variable's value is requested via `$(...)`, QF-Test first searches the stack of direct bindings from top to bottom, then the stack of fallbacks, also top down. The first value found is used. If there is no binding at all for a

name, an `UnboundVariableException`[(853)] is thrown unless you use the special syntax `${default:varname:defaultvalue}` to provide a default value for this case as described in section 6.5[(108)].

Topmost bindings (highest precedence)
...
Bottommost bindings (lowest precedence)

Topmost bindings (highest precedence)
...
Bottommost bindings (lowest precedence)

Primary stack(Direct bindings) **Secondary stack(Default values)**

Figure 6.1: Direct and fallback bindings

This mechanism supports recursive or self-referencing variable definitions. For example, setting a variable named `classpath` to the value `some/path/archive.jar:$(classpath)` will extend a binding for `classpath` with lower precedence. If no such binding exists, a `RecursiveVariableException`[(853)] is thrown.

6.2 Defining variables

Variables are defined in various places with different orders of precedence. You define a variable by adding a row to the appropriate table with the name and value of the variable in the respective columns.

Figure 6.2: System variables

Four sets of variable definitions are accessible from the global options[512]:

System variables
The most basic set of bindings, commonly used for system-specific definitions like path names, JDK version, operating system stuff etc. These are placed at the bottom of the fallback stack so they have the least possible precedence. The system bindings are saved in the system configuration file together with the other QF-Test system options.

Suite variables
These are general definitions for a test suite and saved as part of the suite. The bindings of the suite from which a test run is started, are placed at the bottom of the direct bindings stack. When a procedure from a different suite is called, or a different suite's window or component is accessed, the bindings of that suite are temporarily placed on top of the fallback stack.

Variables from the command line
You can define variables on the command line when starting QF-Test with the `-variable <name>=<value>`[883] argument which can be used multiple times. These bindings are placed above the suite bindings on the direct bindings stack, so they override the bindings of the suite. For example, a variable `count=1` can be defined for the suite and used as `$(count)` for the Number of iterations[598] of a Loop[597] node for quick trial runs. Then you can use `qftest -batch`

`-variable count=100` ... for the actual testing. The command line bindings are accessible mainly for your information, but you can also edit them for experimentation.

Global variables

Placed above the command line variables on the direct call stack, these bindings are a means to pass values between unrelated parts of a test run. Values read from the SUT by a Fetch text[740] or a Fetch index[744] node are bound here, as well as definitions from a Set variable[768] node. The globals are not cleared in between test runs as a convenience and you can edit them for the same reason. To simulate a real test run you should clear them via the $\boxed{\text{Run}\rightarrow\text{Clear global variables}}$ menu first. When run in batch mode (see section 1.7[11]) QF-Test clears the global variables before the execution of each test given with the `-test <n>|<ID>`[881] command line argument.

The rest of the variable definitions are part of the test suite itself:

- Each sequence type node defines a set of bindings that are placed on top of the direct bindings stack when the test run enters the node and removed when the node is exited.

- A Procedure call's[589] bindings are placed on top of the direct bindings stack before executing the Procedure[586]. The bindings defined by a Procedure are placed on top of the fallback stack during execution of the Procedure. Think of them as default values for the expected parameters. Once the procedure returns, both sets of bindings are removed.

6.3 Variable example

Consider the following example:
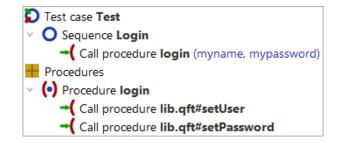
Figure 6.3: Variable example
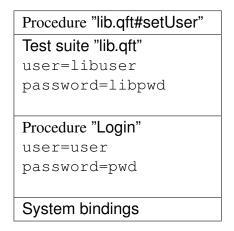
The Sequence "Login" contains a Procedure call of the Procedure "login" which expects two parameters: `user` and `password`.   The Procedure's default bindings are `user=username` and `password=pwd`.   The Procedure call overrides these with `user=myname` and `password=mypassword`.

The "login" Procedure itself holds Procedure calls of Procedures in another test suite called "lib.qft" to write the user and password to some GUI components. We assume that the Procedures in this library have most parameters in common, so they don't define default values themselves but in the suite bindings of "lib.qft" where they are set to `user=libuser` and `password=libpwd`. This is useful for creating and editing the procedures in the library since they can all be executed and tried out separately without having to define Procedure calls.

The following diagram shows the State of the binding stacks during the execution of the Procedure "lib.qft#setUser":

Procedure   call "lib.qft#setUser"

Procedure call "login"
 `user=myname`
 `password=mypassword`

Sequence "Login"
Global bindings
Command line bindings
Suite bindings

Primary stack(Direct bindings)

Procedure "lib.qft#setUser"

Test suite "lib.qft"
 `user=libuser`
 `password=libpwd`

Procedure "Login"
 `user=user`
 `password=pwd`

System bindings

Secondary stack(Default values)

Figure 6.4: Variable bindings example

The important thing to note here is that the Procedure call to "lib.qft#setUser" inside the Procedure "login" does not need to define the `user` parameter again, the parameter is "passed through". As a rule of thumb, when calling one Procedure from another, define a parameter value in the Procedure call if and only if the value has not been explicitly defined yet or you want to pass a different value.

6.4   Fetching data from the GUI

Often it is necessary to fetch some kind of data from the SUT's GUI to use it as test input.

QF-Test offers a special set of query nodes[(740)] for this kind of task, available via ‖Insert→Miscellaneous‖:

- Fetch text[(740)] to fetch a component or item text,

- Fetch index[(744)] to fetch the index of an components item,

- Fetch geometry[(747)] to fetch a component or items geometry.

The retrieved values are assigned to local or global variables that can be declared in the fetch node.

Instead of inserting fetch nodes by hand they can easily be created by first recording a mouse click node the the respective component and then use the transform operation[(17)] to convert the same to the fetch node you need.

6.5 External data and special groups

External data is made available by the Load resources[(785)] and Load properties[(788)] nodes. These assign a group name to a set of definitions from a resource bundle or properties file. To access the value of the definition for *name*, use the syntax ${*group*:*name*}.

When run in batch mode (see section 1.7[(11)]) QF-Test clears the resources and properties before the execution of each test given with the -test <n>|<ID>[(881)] command line argument. In normal mode QF-Test keeps them around to ease building a suite, but for a true trial run you should clear them via the ‖Run→Clear resources and properties‖ menu first.

Some special group names are predefined and always available:

system

    The group `system` gives access to the system properties of the Java VM (for programmers: `java.lang.System.getProperties()`), e.g. `${system:user.home}` for the user's home directory or `${system:java.class.path}` for the `CLASSPATH` with which QF-Test was started. Which names are defined in the group `system` depends on the utilised JDK.

    The group always refers to the VM QF-Test was started with, because variable expansion takes place there.

env

    On operating systems which support environment variables like `PATH`, `CLASSPATH` or `JAVA_HOME` (practically all systems QF-Test runs on), these environment variables can be accessed with the help of the group `env`.

default

You can specify a default value for a variable with the group `default`. The syntax is `${default:varname:defaultvalue}`. This is extremely useful for things like generic components or in almost every place where there is a reasonable default for a variable because the default value is then tightly coupled with the use of the variable and doesn't have to be specified at Sequence or test suite level. Of course you should only use this syntax if the variable lookup in question is more or less unique. If you are using the same variable with the same default in different places it is preferable to use normal syntax and explicitly set the default, so that the default for all values can be changed in a single place.

id

The group `id` can be used to reference QF-Test component IDs. Values in this group simply expand to themselves, i.e. "${id:whatever}" expands to "whatever". Though QF-Test component IDs can be referenced without the help of this group, its use increases the readability of tests. Most notably however, QF-Test component ID references in this group will be updated automatically in case the referenced target component gets moved or its QF-Test ID changed.

idlocal

The group `idlocal` is similar to the `id` group but includes the path to the current test suite, i.e. "${idlocal:x}" expands to "path/to/current/suite/suite.qft#x". This enforces use of the component referenced in the suite that is current at the time of expansion, irrespective of whether there is a component with the same %attId; in the target suite of a procedure call.

quoteitem

Via the `quoteitem` group you can conveniently escape special characters like '@', '&' and '%' in the name of a textual sub-item index to prevent it from being treated as several items, e.g. "${quoteitem:user@host.org}" will result in "user\@host.org".

quoteregex, quoteregexp

The group `quoteregex` with its alias `quoteregexp` can be used to escape characters with special meaning in regular expressions. This is often useful when building regular expressions dynamically or when referencing subitems with special characters in their name by a regular expression index, e.g. "componentid%${quoteregex:foo(baa)}.*" allows you to address the first occurrence of items beginning with 'foo(baa)'.

quotesmartid

The `quotesmartid` group is similar to `quoteitem`. In addition to the item syntax special characters '@', '&' and '%' it also escapes the characters ':', '=', '<' and '>' that have special meaning in SmartIDs, e.g. "${quotesmartid:Name: A & B}" will result in "Name\: A \& B".

qftest

The special group named `qftest` provides miscellaneous values that may be useful during a test run. The following tables list the values currently defined.

Name	Meaning
`32` or `32bit`	*true* if QF-Test is running in a 32bit Java VM - which is not the same as running on a 32bit Operating System - *false* otherwise.
`64` or `64bit`	*true* if QF-Test is running in a 64bit Java VM, *false* otherwise.
`batch`	*true* if QF-Test is running in batch mode, *false* for interactive mode.
`client.exitcode.<name>`	The exit-code of the last process started with the Client[636] attribute set to *<name>*. In case the process is still alive the result is the empty string.
`client.output.<name>`	The output of the last process started with the Client[636] attribute set to *<name>*. The maximum size for buffered output is defined by the option Maximum size of client terminal (kB)[461].
`client.stdout.<name>`	The output on the standard output stream (stdout) of the last process (started with the Client[636] attribute set to *<name>*). The maximum size for buffered output is defined by the option Maximum size of client terminal (kB)[461].
`client.stderr.<name>`	The output on the standard error stream (stderr) of the last process (started with the Client[636] attribute set to *<name>*). The maximum size for buffered output is defined by the option Maximum size of client terminal (kB)[461].
`clients`	A list of the names of all active process clients, separated by a newline.
`clients.all`	A list of the names of all process clients, separated by a newline. This includes live clients as well as the recent dead clients similar to those listed in the "Clients" menu.
`count.exceptions`	Number of exceptions in the current test run.
`count.errors`	Number of errors in the current test run.
`count.warnings`	Number of warnings in the current test run.
`count.testcases`	Total number of total test cases (run and skipped) in the current test run.
`count.testcases.exception`	Number of test cases with exceptions in the current test run.
`count.testcases.error`	Number of test cases with errors in the current test run.
`count.testcases.expectedtofail`	Number of test cases expected to fail in the current test run.
`count.testcases.ok`	Number of successful test cases in the current test run.

`count.testcases.ok.percentage`	Percentage of successful test cases in the current test run.
`count.testcases.skipped`	Number of skipped test cases in the current test run.
`count.testcases.notimplemented`	Number of not implemented test cases in the current test run.
`count.testcases.run`	Number of run test cases in the current test run.
`count.testsets.skipped`	Number of skipped test sets in the current test run.
`dir.cache`	Cache directory of QF-Test
`dir.groovy`	Directory of Groovy
`dir.javascript`	Directory of JavaScript
`dir.jython`	Directory of Jython
`dir.log`	Log directory of QF-Test
`dir.plugin`	Plugin directory of QF-Test
`dir.root`	Root directory of QF-Test
`dir.runlog`	Run log directory of QF-Test
`dir.system`	System-specific configuration directory of QF-Test.
`dir.user`	User-specific configuration directory of QF-Test
`dir.version`	Version-specific directory of QF-Test
`engine.<componentId>`	Retrieves the GUI engine responsible for the given component (see chapter 43[887]).
`language`	The language in which QF-Test displays its graphical user interface.
`license`	The path to the license file
`systemcfg`	The path to the system configuration file
`usercfg`	The path to the user specific configuration file
`executable`	The `qftest` executable matching the currently running QF-Test version, including the full path to its `bin` directory and with `.exe` appended on Windows. Useful if you need to run QF-Test from QF-Test for example to call a daemon or create reports.
`isInRerun`	"true", if current execution is in rerun mode, "false" otherwise, see section 23.3.2[295].
`isInRerunFromLog`	"true", if test run has been re-started from run log, "false" otherwise, see section 23.3.1[292].
`java`	Standard Java program (`javaw` under Windows, `java` under Linux/Unix) or the explicit Java argument if QF-Test is started with `-java <executable>` (deprecated)[868]
`java.mainversion`	The major version of the JRE that QF-Test currently runs on, using 8 for Java 1.8 so the result is something like 8, 11 or 17.

`java.subversion`	The sub-version of the JRE that QF-Test currently runs on. For Java 8 the sub-version taken from after the '_', so for java.version 1.8.0_302 this results in 302. For Java 9 or higher this is the minor version, e.g. 9 in case of java.version 11.0.9.
`linux`	"true" under Linux, "false" otherwise
`macos`	"true" under macOS, "false" otherwise
`os.fullversion`	The whole version of the operating system
`os.mainversion`	The main version of the operating system, e.g. "10" for Windows 10
`os.name`	The name of the operating system
`os.version`	The version of the operating system. In some cases that's not the whole one then you should use `os.fullversion` instead.
`project.dir`	The directory to the current project. This variable is not defined in case the current test suite is not part of a project.
`reruncounter`	Number of current rerun attempt, default is 0, for details see section 23.3.2[(295)].
`return`	The most recent value returned from a Procedure[(586)] through a Return[(592)] node.
`runid`	The runid of the current test run. See section 22.1[(272)] for further information about the runid.
`screen.height`	Screen height in pixels
`screen.width`	Screen width in pixels
`skipnode`	This magic value is not for the casual user. It causes QF-Test to skip execution of the current node. Its primary use is as the value for a variable defined in the Text[(691)] attribute of a Text input[(689)] node which also has its Clear target component first[(691)] attribute set. An empty value would clear the field whereas $_{qftest:skipnode} leaves the field unchanged. But skipnode is also applicable for fine-grained execution control by placing a variable in the comment of a node and selectively passing $_{qftest:skipnode} to that variable. Please note that you almost always want to use lazy syntax '$_' with this variable. Otherwise its expansion as the parameter in a Procedure call node would cause skipping the whole call.
`suite.dir`	Directory of the current suite
`suite.file`	File name of the current suite without directory
`suite.path`	File name of the current suite including directory
`suite.name`	Get the name of the current test suite.
`testcase.name`	The name of the current Test case, empty if no Test case is currently being executed.

`testcase.id`	The QF-Test ID of the current Test case, empty if no Test case is currently being executed.
`testcase.qname`	The qualified name of the current Test case, including the names of its parent Test sets. Empty if no Test case is currently being executed.
`testcase.reportname`	The expanded report name of the current Test case, empty if no Test case is currently being executed.
`testcase.splitlogname`	The qualified name of the current Test case converted to a filename, including the names of its parent Test sets as directories. Empty if no Test case is currently being executed.
`testset.name`	The name of the current Test set, empty if no Test set is currently being executed.
`testset.id`	The QF-Test ID of the current Test set, empty if no Test set is currently being executed.
`testset.qname`	The qualified name of the current Test set, including the names of its parent Test sets. Empty if no Test set is currently being executed.
`testset.reportname`	The expanded report name of the current Test set, empty if no Test set is currently being executed.
`testset.splitlogname`	The qualified name of the current Test set converted to a filename, including the names of its parent Test sets as directories. Empty if no Test set is currently being executed.
`teststep.name`	The name of the current Test step, empty if no Test step is currently being executed.
`teststep.qname`	The qualified name of the current Test step, including the names of its parent Test steps, but not including Test cases or Test sets. Empty if no Test step is currently being executed.
`teststep.reportname`	The expanded report name of the current Test step, empty if no Test step is currently being executed.
`thread`	The index of the current thread. Always 0 except if QF-Test is started with the argument `-threads <number>`[(883)].
`threads`	The number of parallel threads. Always 1 except if QF-Test is started with the argument `-threads <number>`[(883)].
`version`	QF-Test version
`version.build`	QF-Test build number
`windows`	"true" under Windows, "false" otherwise

Table 6.1: Definitions in the special group `qftest`

6.6   Expressions

The `$[...]` syntax for mathematical expressions supports operations like `+`, `-`, `*`, `/` and `%` (modulo) as well as braces. It comes in handy when calculating coordinates, e.g. in a loop.

In fact these expressions are far more powerful, since they are evaluated by the Jython interpreter. You can use arbitrary constructs that are legal syntax for the Jython method `eval`. See chapter 11[(161)] for details about Jython scripting.

Note   Accessing QF-Test variables in an expression follows the same rules as in Jython scripts (see section 11.2.3[(165)]). You can use the standard QF-Test syntax `$(...)` and `${...:...}` for numeric or Boolean values. String values should be accessed with `rc.lookup(...)`.

6.7   Immediate and lazy binding

3.0+

There is a very subtle issue in using QF-Test variables that requires further explanation:

When a new set of variable bindings is pushed on one of the variable stacks, there are two possibilities for handling variable references in the value of a binding, for example when the variable named 'x' is bound to the value '$(y)'. The value '$(y)' can be stored literally, in which case it will be expanded some time in the future when '$(x)' is referenced somewhere, or it can be expanded immediately, so that the value of the variable 'y' is bound instead. The first approach is called 'lazy' or 'late binding', the second approach 'immediate binding'.

The difference, of course, is the time and thus the context in which a variable is expanded. In most cases there is no difference at all, but there are situations where it is essential to use either lazy or immediate binding. Consider the following two examples:

A utility test suite contains a procedure for starting the SUT with different JDK versions. The variable 'jdk' is passed as a parameter to this procedure. For ease of use, the author of the test suite defines some additional useful variables at test suite level, for example a variable for the java executable named 'javabin' with the value '/opt/java/$(jdk)/bin/java'. At the time 'javabin' is bound in the test suite variables, 'jdk' may be undefined, so immediate binding would cause an exception. But even if 'jdk' were bound to some value, immediate binding would not have the desired effect, because the java executable is supposed to be the one from the JDK defined later by passing the parameter 'jdk' to a procedure. Thus lazy binding is the method of choice here.

Imagine another utility test suite with a procedure to copy a file. Two parameters called 'source' and 'dest' specify the source file and destination directory. The caller of the procedure wants to copy a file called 'data.csv' from the same directory as the call-

ing test suite to some other place. The natural idea is to bind the variable 'source' to the value '${qftest:suite.dir}/data.csv' in the procedure call. With immediate binding, '${qftest:suite.dir}' will indeed expand to the directory in which the calling suite resides. However, if lazy binding were used, the actual expansion would take place inside the procedure. In that case, '${qftest:suite.dir}' would expand to the directory of the utility suite, which most likely is not what the caller intended.

In versions of QF-Test up to and including 2.2 all variable expansion was lazy. As the examples above show, both variants are sometimes necessary. Since immediate binding is more intuitive it is now the default. This can be changed with the option When binding variables, expand values immediately[(513)]. The option Fall back to lazy binding if immediate binding fails[(513)] complements this and helps to ease migration of old test suites to the use of Immediate Binding. The warnings issued in this context help locating the few spots where you should use explicit lazy binding as described below. Except for very rare cases where lazy binding is required but immediate binding also works so that the fallback is not triggered, all tests should work out of the box.

In the few cases where it makes a difference whether a variable is expanded immediately or lazily, the expansion of choice can be selected individually, independent of the setting of the above option, by using an alternative variable syntax. For immediate binding use '$!' instead of just '$'. Lazy binding is selected with '$_'. For example, to define a variable at test suite level that specifies a file located in this test suite's directory, use '$!{qftest:suite.dir}/somefile'. If immediate binding is the default and you require lazy binding as in the 'jdk' example above, use '$_(jdk)'.

Note   With lazy binding the order of variable or parameter definitions in a node or a data driver did not matter because nothing was expanded during the binding stage. With immediate bindings, variables are expanded top-to-bottom or, in a data driver, left-to-right. This means that if you define x=1 and y=$(x) it will work, with y being set to 1, if x is defined first. If y is defined first the definition will either fail or trigger the lazy definition fallback described above.

Chapter 7

Problem analysis and debugging

The whole point of creating automated tests is to uncover problems in the SUT. Therefore we can justifiably expect the tests to fail occasionally.

After the execution of a test has finished, a message will appear in the status line that will hopefully say "No errors". If something went wrong, the numbers of warnings, errors and exceptions that occurred is shown. Additionally an error dialog may pop up. In that case you will need to find out what went wrong.

For some problems the cause may be obvious, but very often it is not. First and foremost in this case is the need to determine whether the test failed due to a bug in the SUT or whether the SUT behaved correctly but the logic of the tests was wrong. The dilemma here is that any potential problem in the SUT must be duly reported as soon as possible, yet every false bug report is a waste of time and will cause resentment among the developers. Therefore, each problem needs to be thoroughly analyzed and every alleged bug in the SUT should ideally be reproducible before a report is submitted.

QF-Test supports testers in this crucial task in two ways. A detailed log is created for each test run that holds all the relevant information for post mortem error analysis, including screenshots taken at the time that an error occurred. The integrated test debugger helps with analyzing and understanding the flow of control and information during a test run.

Video The video

> 'Error analysis'
> https://www.qfs.de/en/yt/error-analysis-40-html

shows a brief example for error handling.

7.1 The run log

During test replay QF-Test creates a run log that records everything that is going on. The run logs for recent tests are accessible from the ⟦Run⟧ menu, the current or most recent run log can also be opened by typing ⟦Ctrl-L⟧ or the respective ≡ᵥ button in the toolbar. See section 7.1.7[(123)] for information about options influencing run log creation.

The structure of a run log is similar to that of a test suite, but there are subtle differences. Nodes are added to the run log when they are executed. Setup[(554)] and Cleanup[(557)] nodes, for example, are typically executed more than once, in which case multiple copies will be recored in the run log as shown below:

Test suite   Run log
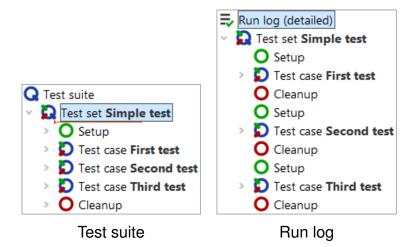
Figure 7.1: A simple test and its run log

A run log is the essential resource for determining **what** went wrong in a test, **where** it went wrong and maybe even get an idea about **why** it went wrong. Therefore the emphasis is on completeness of information rather than readability and a run log should not be confused with a report or summary. Report generation is covered in chapter 22[(271)].

In addition to the nodes copied from the test suite, a run log contains failure information, optional annotations, various kinds of messages as well as information about variable expansion and run-time behavior.

The information gathered from a long test run accumulates and can eat up enormous amounts of memory and QF-Test has several means to cope with that. The best one, which is also the default, is to create split run logs as described in section 7.1.6[(122)]. The resulting *.qzp files in ZIP format not only preserve memory on disk - partial run logs can be saved during test execution and removed from memory to free up necessary space. The same applies when processing logs, e.g. for report creation. The older op-

tion <u>Create compact run log</u>⁽⁵¹⁰⁾ as well as the alternative file formats `*.qrz` and `*.qrl` add flexibility but are mostly retained for compatibility reasons.

7.1.1 Error states

There are three kinds of failures differing in the level of severity:

Warnings
    Warnings indicate problems that are typically not serious, but might lead to trouble in the future and may need looking at. For example, QF-Test issues a warning, if the best match for a component barely meets the requirements and differs in some significant way.

Errors
    Errors are considered to be serious defects that require closer inspection. They indicate that the SUT does not fulfill some requirement. A typical cause for an error is a mismatch in a <u>Check text</u>⁽⁷⁰⁸⁾ node.

Exceptions
    Exceptions are the most serious kinds of errors. An exception is thrown when a situation occurs in which QF-Test cannot simply continue with the execution of the test. Most exceptions indicate problems with the logic of the test, though they can just as well be caused by the SUT. A <u>`ComponentNotFoundException`</u>⁽⁸⁵⁰⁾, for example, is thrown when no component in the SUT matches the intended target for an event. A list of all possible exceptions is available in <u>chapter 41</u>⁽⁸⁵⁰⁾.

Each node of a run log has an associated state which can be one of *normal*, *warning*, *error* or *exception*. This state is visually represented by a frame around the node's icon which is orange for *warning*, red for *error* and bold red for *exception*.
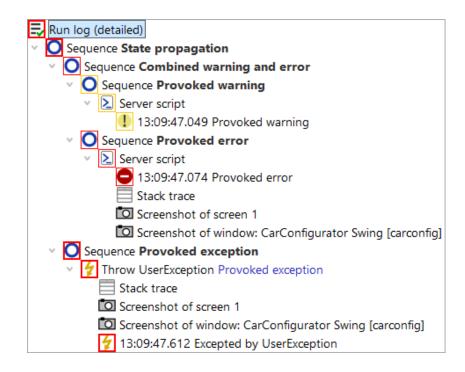
Figure 7.2: Error states in a run log

As shown in the (somewhat reduced) screenshot above, error states propagate from bottom to top. The *exception* state takes precedence over the *error* state, which in turn overrides *warning*. The most severe kind of error that propagates to the top of the tree determines the final result of a test and QF-Test's exit code when run in batch mode (see section 42.3[(885)]).

If necessary, the propagation of errors can be restricted for all kinds of Sequence[(536)] nodes with the help of the Maximum error level[(537)] attribute. This can be useful for sequences which are known to contain errors that should not be taken into account just yet. Exceptions can be handled with the help of the Try[(616)] and Catch[(619)] nodes. The Maximum error level[(621)] attribute of the Catch node determines the state to propagate for a caught exception.

7.1.2 Navigating the run log tree

All of the basic editing methods for a run log are similar to those for a test suite. One significant difference is that can neither add or remove any nodes nor edit the attributes of the nodes copied from the test suite. You can add annotations though, for example to document the reason for an error if it is known.

The first question to answer when looking at a run log is **"What happened"?**

The ⬤ Button, or ‖Edit→Find next error‖, (Ctrl-N) for short, moves the selection to the next place at which an error actually occurred.

Respectively, 🔍 or ‖Edit→Find previous error‖ ((Ctrl-P)) moves backwards.

The option Skip suppressed errors[503] determines whether to ignore errors that didn't propagate up to the root node. There's a menu item shortcut ‖Edit→Skip suppressed errors‖ to quickly toggle the latter option.

The next question might be **"Where did this happen?".**

Though a run log is similar in many ways to a test suite, the connection isn't always obvious. The function ‖Edit→Find node in test suite‖ ((Ctrl-T)) will take you to the exact node in the test suite that is represented by the selected node in the run log, always provided that the test suite can be located and hasn't changed in a way that prevents this. If the run log is loaded from a file, the corresponding test suite may not be located at the same place as when the test was executed. If the test suite cannot be found, a dialog will pop up that lets you select a different file. In case you select a wrong file or some other test suite is found instead of the one the run log was created from, you may end up at some totally different node, or none at all. In that case you can use the menu item ‖Edit→Locate corresponding test suite‖ to explicitly change the test suite.

If you want to set the link between the file path of the executed test suite and the file path where you develop the test suite permanently you can do so in the options menu for the log-file as explained in Directory map for test suites[512].

7.1.3 Run-time behavior

QF-Test tracks the start time and the time spent for each node executed during a test, the latter in two forms: 'Real time spent' is the wall time elapsed between entry and exit of the respective node. It includes explicit delays introduced in nodes via the 'Delay before/after' attribute, user interrupts when debugging tests or other overhead like taking screenshots. The actual time spent testing is collected in the 'Duration' attribute, making it a better indicator for the performance of the SUT.

To get a better understanding of the run-time behavior of a test run you can activate display of duration indicators via the toolbar button ▬ , the menu ‖View→Show relative duration indicators‖ or the option Show relative duration indicators[502]. A colored bar is shown for each node with the length based on the percentage of time spent in the node relative to the time of its parent node. Bottlenecks in the performance of a test run can be located by drilling down into the nodes with the longest bars:
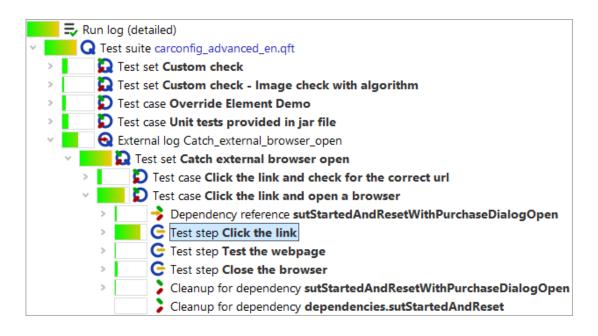
Figure 7.3: Display of duration indicators in the run log

Via the option Duration indicator kind[(502)] or the sub-menu
‖View→Duration indicator kind‖ the display can be toggled to show the relative duration,
real time or both. The latter is especially helpful but takes a bit of getting used to.

7.1.4 Showing return values

If the option Show return values of procedures[(503)] is active (in a run log also accessi-
ble via the ▯ menu), return values of Procedures[(586)] are displayed in the tree after the
respective Procedure call[(589)] node.

7.1.5 Accepting values of failed checks as good

A noteworthy feature of QF-Test's run log is the ability to quickly accept the actual val-
ues of a failed Check node as good. When QF-Test logs a failed Check it includes the
complete current state of the Check node's target component in the SUT. This is much
more useful than the failure message alone, which, for example, might just tell you that
a table column has 10 rows instead of the expected 9, but not what its contents are.

If you are analyzing a failed Check and see that the value in the SUT was actually correct
and the expected value stored in the test suite wrong, you can press ⌐Ctrl-U⌐ or select
‖Update check node with current data‖ from the context menu to accept the data from
the failed Check as the new correct value for the Check node.

Warning: QF-Test currently doesn't preserve regular expressions in Check text[708] or Check items[719] nodes, they will simply get overwritten.

7.1.6   Split run logs

Run logs for long-running tests can get very large and consume an enormous amount of memory, even more so in case many screenshots are kept. Compact run logs can help, but not enough to make tests that run for days on end possible without turning off the run log entirely. The best way to overcome the memory problem are split run logs.

For split run logs, whenever a certain part of a test has finished, QF-Test takes the run log for that part, removes it from the main run log, saves it to a separate file and replaces it with a single node that references that file. The partial logs are complete run logs in themselves and can be viewed and archived independently, though normally they are accessed through the main run log. When navigating the main run log or when creating reports, QF-Test transparently loads the partial run logs from the separate files as required and removes them from memory when no longer needed. This makes it possible to navigate huge run logs while still retaining a relatively small memory footprint. Of course operations like searching or report creation that need to traverse the whole run log become slower, but jumping from error to error remains quite fast and loading the main run log is sped up drastically.

There are two ways for storing a main run log and its partial logs: All combined together in a single ZIP file with the extension `.qzp` or with the partial logs in a separate directory. The latter is named after the main run log with the extension `.qrl` or `.qrz` removed and the suffix `_logs` appended. Inside a `.qzp` ZIP file the same layout is used so that it is possible to zip or unzip files manually without breaking the internal references in the run log. This compatibility is also the reason why by default partial logs inside the ZIP are are stored compressed with the extension `.qrz`. This is slightly less efficient than storing uncompressed `.qrl` files, but that way a `.qzp` run log can be unzipped without its overall size exploding.

To make use of split run logs you can explicitly define points at which a run log is broken and split into parts. This is done via the Name for separate run log[564] attribute of Data driver[562], Test case[518], Test set[525], Test call[531] or Test step[539] nodes. When used with a Data driver, the logs for each iteration are saved separately, in the other cases the node with the Name for separate run log attribute is split off. Otherwise partial run logs are split off automatically when they reach a certain size. This functionality can be configured via the option Minimum size for automatic splitting (kB)[505].

When working with split run logs it is advisable to turn Create compact run log[510] off, in order to keep all details in the run log. This will consume a bit more disc space, but is very helpful when analyzing errors.

Split run logs are also very handy for tracking the progress of a test in batch mode. In

that context it is extremely useful that the file names for the partial logs can be created using the same placeholders as when specifying the name of the main run log on the command line. In particular the error state of the partial log can be made part of its filename. Please see the documentation for the attribute <u>Name for separate run log</u>⁽⁵⁶⁴⁾ for details.

7.1.7 Run log options

There are several options affecting the creation of run logs and their content. Among others, you can choose whether to create compact or detailed run logs, whether to log screenshots of the whole screen and/or the client windows or whether to suppress run logs altogether. All options are explained in detail in <u>section 39.9</u>⁽⁵⁰⁰⁾.

7.1.8 Creating a test suite from the run log

3.3+

If several people are involved in the test development process, it might be useful to generate a test suite from the run log directly. The generated test suite could be used to reproduce a test run on-the-fly without having the entire structure of test suites.

You can create a test suite from the run log via performing a right mouse click at any node in the run log and selecting Create test suite from the context menu.

QF-Test creates a new file containing all executed steps of the respective tests under <u>Extras</u>⁽⁵⁴⁷⁾ as well as the used components.

Note

QF-Test only adds the executed steps to the new test suite. Variables will be expanded immediately, so you can only see their value in the new file. Organizational nodes like procedures or control structures will not become created.

You have to set a couple of options in order to get this feature properly working (Under Run log -> Content):

- <u>Create compact run log</u>⁽⁵¹⁰⁾ needs to be switched off.

- <u>Log variable expansion</u>⁽⁵⁰⁸⁾ needs to be switched on.

- <u>Log parent nodes of components</u>⁽⁵⁰⁸⁾ needs to be switched on.

If you have access to all test suites, you can use also use information from them for creating the new one. Therefore select Create test suite from existing structure from the context menu. In contrast to the approach described above, it is not required to switch on the option <u>Log parent nodes of components</u>⁽⁵⁰⁸⁾.

7.1.9 Merging run logs

4.1+

During test development you might face the requirement, that you have a run log with the test results for your test-cycle. But in several cases you might need to rerun one test case which was failing due to subtle reasons during the previous test run. Once that rerun has taken place you would like to update your test-report to have that new execution in that test-report instead the previous one. For this purpose it's possible to merge run logs via command line.

A typical merge command looks like this:

```
qftest -batch -mergelogs -mergelogs.mode=replace
                -mergelogs.masterlog full_log.qzp
                -mergelogs.resultlog newresult_log.qzp rerun.qzp
```

Example 7.1: Sample call of merging run logs for update

The command above takes the result of the rerun from the run log `rerun.qzp`, searches for the test case in the run log `full_log.qzp` and store the updated run log to `newresult_log.qzp`. If you set the parameter `mergelogs.mode` to `merge` the new test cases will be added to the existing structure and the original test cases will remain in the run log.

Another case might be to add run logs of several test runs into one large run log in order to get a more readable report. This kind of merging is also implemented and can be achieved by another command line call like this:

```
qftest -batch -mergelogs -mergelogs.mode=append
                -mergelogs.resultlog newresult_log.qzp run1.qzp run2.qzp
```

Example 7.2: Sample call of merging run logs for adding

The call above takes the run logs `run1.qzp` and `run2.qzp` and creates a run log `newresult_log.qzp` which contains the results from both run logs. In this mode the parameter `mergelogs.masterlog` is optional. If the parameter is set, the corresponding run log will be used as a root for a resulting run log.

7.2 The debugger

As in any development environment, at some point the need will arise to debug problems introduced into a test suite which cannot readily be solved by a straight-forward analysis of the elements and structure of the suite. To this end, QF-Test includes an intuitive

debugger. Those of you familiar with debugging programs in Java or other programming languages will find this debugger similar in function and usefulness.

7.2.1   Entering the debugger

The QF-Test debugger can be started directly by selecting a node (or some nodes) to execute and pressing the step-in 🔄 or step-over 🔄 buttons, or by using the menu operations ‖Debugger→Step in‖ and ‖Debugger→Step over‖ or the keyboard shortcuts [F7] and [F8]. See section 7.2.3[(126)] for a detailed explanation of these operations.

If you are running tests on your test suite and use the play button to start execution (see section 4.2[(37)]), the debugger will normally not be entered. However, the debugger will be activated automatically when any one of the following occur:

- A user-defined breakpoint is reached (see section 7.2.4[(127)] on turning on/off break-points).

- Execution is interrupted manually by pressing the pause button or [F9] or selecting the ‖Run→Pause‖ menu item.

- A caught or uncaught exception is thrown, an error happens or a warning is logged and the respective option to break under that condition is set (see option Automatic breaks[(499)]).

When the debugger suspends execution of the test, the node about to be executed will be shown with a colored frame around its icon that reflects the cause for the break. If the debugger is stopped due to manual intervention, a user breakpoint or when stepping, the frame will be black. When stopping due to a warning, error or exception the frame will be orange, red or thick red respectively, exactly like the error indicators in the run log.

Note   When the debugger is entered due to a warning, error or exception it will move execution back to the beginning of the node that caused it, giving you a chance to fix the cause of the problem and re-execute that node. If this is undesirable or impossible you can simply skip the node (see section 7.2.3[(126)]).

7.2.2   The debugger window

The debugger can be run either from within the normal test suite view, or by opening a dedicated debugger window by selecting ‖Debugger→Show debugger window‖ once the debugger has been entered.

You can also cause the debugger window to open automatically whenever the debugger is entered by setting the option Always open debugger window[(499)] in the global options dialog or under the ‖Debugger→Options‖ menu. If you open or close the debugger window explicitly, this is considered a "manual override" and this option will be ignored for the rest of the test run.

The debugger window is similar to a normal test suite window. You can select nodes and edit their attributes, but you cannot delete or insert nodes, there are no file operations and no recorder. For the more complex operations you can quickly jump from the debugger window to the same node in the respective test suite window by pressing ⌞Ctrl-T⌝ or selecting ‖Find node in test suite‖ from the ‖Edit‖ menu or the context menu.

The lower half of the debugger window shows the nodes that are binding variables on the primary and fallback variable stacks (see chapter 6[(103)]). For the primary stack all nodes are shown, even if they are not binding any variables. This is useful because it serves as a kind of stack-trace for the current point of execution of the test run. You can double-click on any node to quickly navigate to the node in its associated test suite.

A single click on a node brings up its variable bindings in the right half of the variable display. There the variable values can be edited, new variables can be added or existing ones removed. These changes immediately affect the current test run, but are of a temporary nature. They are not propagated to the nodes in which the variables were bound in the first place.

7.2.3   Debugger commands

Most of the debugger commands are similar to those of any other debugger. However, some additional commands are included that deal with special situations.

Step-wise debugging of a test suite is available through three operations:

- The step-in button ↻ (⌞F7⌝, ‖Debugger→Step in‖) executes the currently selected node and will set the execution mark to the next deepest node, regardless of how deep that node may lie in the node structure. This operation is useful, for example, to step into and debug a Procedure or Sequence.

- The step-over button ↻ (⌞F8⌝, ‖Debugger→Step over‖) executes the currently selected node as well as any child nodes that lie under it and then sets the execution mark to the next node at the same level. This is helpful for being able to execute an entire Procedure or Sequence without stepping through each step individually.

- The step-out button ↻ (⌞Ctrl-F7⌝, ‖Debugger→Step out‖) executes the currently selected node as well as any other nodes at the same level (including any child

nodes of these nodes) and then sets the execution mark to the next node at the next higher level. This type of operation is useful when, for example, you are debugging a Procedure or Sequence and don't want to step through the rest of the nodes in it. By simply using step-out, you can execute the rest of the nodes and return.

The *skip* functions expand the QF-Test debugger in a powerful way which is not typically possible for a debugger in a standard programming environment. In short, they allow you to jump over one or more nodes without having to execute those nodes at all.

- The skip-over button ↱ (Shift-F9, Debugger→Skip over) jumps over the current node without executing it, moving the execution mark to the next node.

- The skip-out button ↰ (Ctrl-F9, Debugger→Skip out) ends the execution of the current Procedure or Sequence and jumps to the next node at the next higher level.

Even more powerful is the ability to continue the test run at any arbitrary node, even in a completely different test suite. QF-Test will keep as much of the current execution context as possible, including variable bindings. The closer the new target location is to the current point of execution, the more information can be salvaged.

You can switch execution to a different node by pressing Ctrl-, or by selecting the menu item Run→Continue execution from here or the respective item in the context menu. When you do so, execution will not continue immediately, only the next node to be executed will change. You can continue the test as usual by single-stepping or resuming the test run.

The following additional commands are available:

- The rethrow-exception button ⚡ (Debugger→Rethrow exception) is only active when the debugger was entered due to an exception. It lets you rethrow the exception to be handled by the test suite just as if the debugger had never caught it in the first place.

- The locate-current-node button ◉ (Debugger→Locate current node) quickly moves the selection in the tree-view to the node that is about to be executed. It is a useful shortcut in case you get lost while moving around the test suite.

7.2.4 Manipulating breakpoints

Setting a breakpoint on a node will tell the debugger to suspend execution just before it enters that node. Breakpoints are displayed in the tree-view by prepending "(B)" to the name of a node.

Breakpoints can be set or removed individually with $\boxed{\text{Ctrl-F8}}$ or with the $\boxed{\text{Debugger}\rightarrow\text{Breakpoint on/off}}$ menu item. After finishing a debugging session you can use $\boxed{\text{Debugger}\rightarrow\text{Clear all breakpoints}}$ to remove any breakpoints that might have been left hanging around. This command will remove all breakpoints from *all* test suites.

Note   Breakpoints are transient and will not be saved with the test suite.

Chapter 8

Organizing the test suite

Creating useful, reliable tests requires more than just recording sequences and playing them back. You can fill a test suite with lots of sequences in a short time, but you are bound to lose track of what you've got sooner or later if you do not organize your tests in some logical structure. QF-Test provides you with a number of structure elements to achieve this.

Before you start recording sequences and put them into a structure make sure that you

- You have a good idea of what you are testing.

- You are testing the right thing.

- Your tests are reliable and repeatable.

- Your tests are easy to maintain.

- The results of your tests are conclusive.

The essential prerequisite of getting the components right has been discussed in chapter 5[(42)]. Here we are going to concentrate on structuring the actual test sets, test cases, test steps, sequences, events, checks, etc.

8.1 Test suite structure

Figure 8.1: Test suite structure

QF-Test provides structure elements on different levels:

The QF-Test files for saving the tests and components in the file directory. These can be bundled in projects.

The Test suite has a set structure starting with the testing section that can hold any number of Test set[525] nodes, which in turn can have any number of Test case[518] nodes or more Test sets.

Next comes the Procedures[586] section, where you can place any number of Procedure[586] nodes. QF-Test provides Package[594] nodes as structure element in this section. A package node can hold any number of procedure nodes or more package nodes.

After that you will find the Extras[547] where you place any type of node and try out tests before moving them to the testing section.

The last section, Windows and components[835], is reserved for the components referenced

by the tests.

QF-Test provides a number of structure elements for the tests themselves like Test case and Test set nodes as well as Setup and Cleanup nodes for setting up the preconditions for the tests, cleaning up after the test and error handling.

8.2   Test set and Test case nodes

8.2.1   Test management with Test set and Test case nodes

2.0+

The Test set[525] and Test case[518] nodes provide a small-scale, pragmatic form of test management right inside QF-Test. Their main feature is the smart dependency management described in Dependency nodes[138] that allows Test cases to be implemented completely independent from each other. With properly written Dependencies[548], cleanup of the SUT for previously executed tests is handled automatically along with the setup for the next test and all error handling.

8.2.2   Concept

Conceptually a Test case node represents a single elementary test case. As such it is the main link between test planning, execution and result analysis. With the help of Dependencies[548], Test cases can be isolated from each other so that they can be run in any arbitrary order. QF-Test automatically takes care of the necessary test setup. Cleanup is also automatic and will be performed only when necessary in order to minimize overhead in the transition from one test to the next. This enables things like running subsets of functional test suites as build tests or retesting only failed Test cases.

Test sets basically are bundles of Test cases that belong together and typically have similar requirements for setup and cleanup. Test sets can be nested. The whole structure of Test sets and Test cases is very similar to Package[594] and Procedure[586] nodes. The Test suite root node can be considered a special kind of Test set.

Test suite, Test set and Test case nodes can be called from other places using a Test call node. That way, tests that run only a subset of other tests can easily be created and managed. Test call nodes are allowed everywhere, but should not be executed from within a Test case node because that would break the atomicity of a Test case from the report's point of view. A warning is issued if Test case execution is nested.

8.2.3   Variables and special attributes

Default value

As both Test sets and Test case can be called via a Test call node they each have a set of default parameters similar to those of a Procedure[586]. These will be bound on the secondary variable stack and can be overridden in the Test call node.

Variable definitions

A Test case has an additional set of variable bindings. These are direct bindings for the primary variable stack that will be defined during the execution of the Test case and cannot be overridden via a Test call node or the command line parameter `-variable <name>=<value>`[883]. Primary and secondary variable stack are described in section 6.1[103].

Characteristic variables

The list of Characteristic variables is a set of names of variables that are part of the characteristics of the test for data-driven testing. Each execution of the Test case with a different set of values for these variables is considered a separate test case. The expanded values of these variables are shown in the run log and report for improved error analysis.

Condition

Another useful attribute is the Condition which is similar to the Condition[606] of an If[605] node. If the Condition is not empty, the test will only be executed if the expression evaluates to true. Otherwise the test will be reported as skipped.

Expected to fail if...

Sometimes a Test case is expected to fail for a certain period of time e.g. when it is created prior to the implementation of the respective feature or before a bug-fix is available in the SUT. The Expected to fail if... attribute allows marking such Test cases so they are counted separately and don't influence the percentage error statistics.

8.3   Sequence **and Test step** nodes

The primary building block of a test are the Sequence[(536)] and Test step[(539)] nodes which execute their child nodes one by one in the order as they appear. They are used to structure the child nodes of a Test case.

The difference between Sequence[(536)] and Test step[(539)] nodes is that Test step[(539)] nodes will show up in the report whereas Sequences[(536)] will not.

8.4   Setup **and Cleanup** nodes

Since it is in the nature of testing that tests may fail from time to time it is crucial to have structure elements that will help you set up a defined initial state for a test. Setup and Cleanup nodes are for simple cases and are inserted as child nodes of Test case nodes. However, in most cases Dependency nodes, that contain Setup and Cleanup nodes, will prove far more efficient.

Test case nodes with well designed Setup and Cleanup nodes have the following properties important to successful testing:

- The Test case can be executed independently of previous test cases that may have failed.

- Test case nodes can be added at any position in Test suite and Test set nodes without influencing other Test cases

- You can work on a Test case or just run it without having to execute previous Test cases to get the SUT into the state required by your test.

- You can execute any number of Test case nodes in case you do not want to run the whole Test set or Test suite.

In the simplest case exactly the same initial condition is required by all the Test case nodes of a Test set. This can be implemented via the following structure:
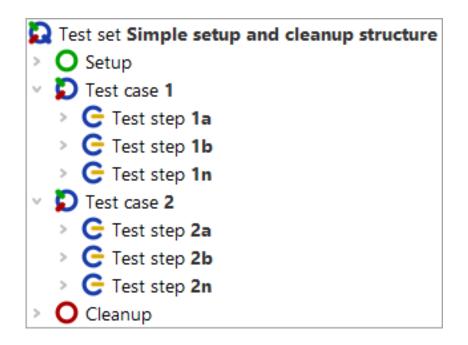
Figure 8.2: Test structure with simple Setup and Cleanup

In the run log you can see that for each Test case node first the Setup node and then the Cleanup node is run:

Figure 8.3: Test execution with simple Setup and Cleanup

In this simple example the cleanup is done in any case, even if the next test could be executed with the state the previous test left the SUT in. QF-Test provides a more comprehensive structure for setting up the SUT and handling cleanup much more efficiently, and even including error handling. This is explained in chapter section 8.6[(138)] in detail.

8.5 Procedures and Packages

In a way, writing good tests is a little like programming. After mastering the initial steps, tests and source code alike tend to proliferate. Things work fine until some building block that was taken for granted changes. Without a proper structure, programs as well as tests tend to collapse back upon themselves at this point as the effort of adapting them to the new situation is greater than the one needed for recreating them from scratch.

The key to avoiding this kind of problem is reuse or avoidance of redundancy. Generating redundancy is one of the main dangers of relying too much on recording alone. To give an example, imagine you are recording various sequences to interact with the components in a dialog. To keep these sequences independent of each other, you start

each one by opening the dialog and finish it by closing the dialog again. This is good thinking, but it creates redundancy because multiple copies of the events needed to open and close the dialog are contained in these sequences. Imagine what happens if the SUT changes in a way that invalidates these sequences. Let's say a little confirmation window is suddenly shown before the dialog is actually closed. Now you need to go through the whole suite, locate all of the sequences that close the dialog and change them accommodate the confirmation window. Pure horror.

To stress the analogy again, this kind of programming style is called *Spaghetti Programming* and it leads to the same kind of maintenance problems. These can be avoided by collecting the identical pieces in one place and referring to them wherever they are needed. Then the modifications required to adapt to a change like the one described above are restricted to this place only.
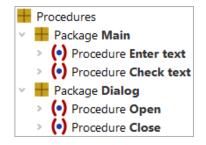
Figure 8.4: Packages and Procedures

QF-Test comes with a set of nodes that help to achieve this kind of modularization, namely the Procedure[(586)], Procedure call[(589)] and Package[(594)] nodes. A Procedure is similar to a Sequence[(536)] except that its Name[(587)] attribute is a handle by which a Procedure call node can refer to it. When a Procedure call is executed, the Procedure it refers to is looked up and execution continues there. Once the last child node of the Procedure has finished, the Procedure call has completed as well.

Packages are just a way to give even more structure to Procedures. A hierarchy of Packages and Procedures, rooted at the special Procedures[(595)] node, is used to group sets of Procedures with a common context together and to separate them from other Procedures used in different areas.

A Procedure that always does exactly the same, no matter where it is called from, is only marginally useful. To expand on the above example, let's say we want to extend the Procedure that opens the dialog to also set some initial values in some of its fields. Of course we don't want to have these initial values hard-coded in the Procedure node, but want to specify them when we call the Procedure to get different values in different contexts. To that end, parameters can be defined for the Procedure. When the Procedure call is executed, it specifies the actual values for these parameters during this run. How all of this works is explained in Variables[(103)]. Also please take a look at the detailed

explanation for the Procedure[(586)] and Procedure call[(589)] nodes for a better understanding of how these complement each other.

A test suite library with a set of commonly useful Procedures is provided with QF-Test under the name `qfs.qft`. An entire chapter of the Tutorial is devoted to this library and section 24.1[(298)] explains how to include it in your test suites.

8.5.1 Local Procedures and Packages

3.1+

If you work with several test suite libraries you might face a situation, where you define reusable test steps or sequences, which you only want to use within a dedicated test suite. If you want to create such local Procedures, you can put a '_' as first sign of the procedure's name. This marks a Procedure as test suite local.

A call of a local Procedure can only be inserted within the test suite, where it is defined. You can use the same concept for local Packages.

8.5.2 Relative Procedures

3.1+

If you call Procedures from other Procedures, it could be convenient not to specify the full procedure name all the time.

So called 'relative' procedure calls can only be added to a Package, which has the Border for relative calls (see Border for relative calls[(595)]) attribute specified. The structure of that call follows the concept below:

Level	Call
Procedures of the same level	.Name of Procedure
Procedures one level higher	..Name of Procedure
Procedures one level deeper	.Name of Package.Name of Procedure

Table 8.1: Relative procedure calls

As you can see each dot stands for one level. So calling a Procedure two levels higher requires three dots (Current level also requires a dot.)

8.5.3 Inserting Procedure call nodes

As you should organize your tests in separate test steps, which are ideally the same like QF-Test's procedures, QF-Test offers several ways to insert those Procedure call nodes:

1. Via the menu $\boxed{\text{Insert}{\rightarrow}\text{Procedures}{\rightarrow}\text{ProcedureCall}}$

2. Via right mouse click and selecting $\boxed{\text{Insert node}{\rightarrow}\text{Procedures}{\rightarrow}\text{ProcedureCall}}$

3. Copy a Procedure node and insert it at the location of the Procedure call using the normal Copy/Paste actions

4. Via Drag&Drop operation, i.e. dragging the Procedure node to its target node

5. Via the keyboard shortcut $\boxed{\text{Ctrl-A}}$

6. By converting a Sequence or a Test step into a procedure, as described in section 8.5.5[138]. Shortcut $\boxed{\text{Ctrl-Shift-P}}$

This approach is also valid for inserting Dependency reference nodes, except the keyboard shortcut.

8.5.4 Parameterizing nodes

3.1+

You can create parameters for a Procedure, Dependency or Test case automatically via the menu $\boxed{\text{Operations}{\rightarrow}\text{Parameterize node}}$.

The parameter details dialog allows you to define for which actions you want to create parameters, e.g. only text-inputs or check nodes.

8.5.5 Transforming a Sequence into a Procedure

3.0+

This transformation is very useful for developing procedures immediately after recording! Under Extras you can convert a recorded Sequence node into a Procedure and move that to the Procedures node.

3.1+

If you transform a Sequence under Test cases QF-Test automatically creates a Procedure node and inserts a Procedure call to the previous location of the transformed node.

8.6 Dependency nodes

Video:

Dependencies
https://www.qfs.de/en/yt/dependencies-basics-50.html

8.6.1 Concept

Dependencies are a powerful and optimized concept for handling pre- and post-conditions. They are indispensable when running tests in the QF-Test <u>Daemon mode</u>⁽¹¹³⁹⁾ mode. They basically work the following way:

1. Set up a list of all dependencies required for the test case.

2. Compare the list of dependencies needed for the current test case with the list of dependencies of the test case executed last.

3. Execute all Cleanup nodes of the dependencies no longer part of the current dependencies list plus the ones where the values of <u>Characteristic variables</u>⁽¹⁴⁵⁾ changed, including the Cleanup nodes of the Dependencies based on them.

4. Execute **all** Setup nodes of the current dependencies list.

Test cases as well as other dependencies can make use of <u>Dependency</u>⁽⁵⁴⁸⁾ nodes placed in the <u>Procedures</u>⁽⁵⁸⁶⁾ section via <u>Dependency reference</u>⁽⁵⁵¹⁾ nodes. Therefore, Setup and Cleanup nodes placed in a Dependency node can be used by various test cases - in contrast to those placed directly in Test case or Test set nodes.

In order to understand the concept of Dependency nodes it might be helpful to have a look at how a manual tester would proceed: He would do the setup for the first test case and then run it. In case of errors he may want to run special error cleanup routines. After that he would first check the requirements of the second test case. **Only then** would he do any cleanup. And he would only clean up as much as is necessary. Next he would check that the SUT still meets **all** preconditions required by the next test case and if not execute the necessary steps. In case the previous test case failed badly he might need to clean up the SUT completely before being able to set up the initial condition for the second test case.

This is exactly what you can implement using QF-Test Dependencies.

Dependencies give an answer to the disadvantages of the classical <u>Setup and Cleanup nodes</u>⁽¹³³⁾ where Setup nodes can only be nested by nesting test sets and where Cleanup nodes will be executed in any case, both of which is not very efficient. Moreover, Dependency nodes provide structure elements for handling errors and exceptions.

Quite a number of the provided sample test suites make use of Dependencies, e.g.:

- The test suite in the directory `doc/tutorial` named `dependencies.qft`. You will find a detailed description in the tutorial in chapter 16.

- The test suite in `demo/carconfigSwing` named `carconfigSwing_en.qft`, showing a realistic example.

- The SWT demo test suite named `swt_addressbook.qft`, with an example for SWT users

- The test suite in `demo/eclipse` named `eclipse.qft`, containing nested Dependencies.

- The data driver demo `datadriver.qft` in `doc/tutorial` also uses Dependencies.

Single-stepping through these suites in the debugger, looking at the variable bindings and examining the run logs should help you to familiarize yourself with this feature. Please take care to store modified test suites in a project-related folder.

8.6.2   Usage of Dependencies

You can define Dependencies in two places:

- You can implement Dependency nodes at the beginning of Test suite, Test set or Test case nodes. Additionally to their own Dependency a Test case or Test set may inherit the Dependency of its parent node.

- Dependencies used by a number of Test cases or used as a basis for other Dependencies may be implemented just like a Procedure[(586)] node and be placed in the Procedures[(595)] section, e.g. in a Package[(594)] node named "Dependencies". The fully qualified name has the same structure as that of a Procedure. Just like Procedures[(586)] Dependencies can be referred to by other nodes, in this case via Dependency reference nodes.

One Dependency should deal with one precondition. Then you can reduce the test overhead generated by cleanup activities. In case a Dependency itself relies on preconditions these should be implemented in separate Dependency nodes. Dependencies can either be inherited from a parent node or referred to explicitly via Dependency reference nodes.

The implementation of the actual pre- and post-conditions is done in the Setup and Cleanup nodes of the Dependency.

In case a Test set or Test case node has a Dependency node as well as Setup and Cleanup nodes the Dependency will be executed first. Setup and Cleanup nodes have no influence on the dependency stack.

8.6.3   Dependency execution and Dependency stack

The execution of a Dependency has three phases:

1. Generate a list of required Dependencies and check with list of previously executed Dependency nodes

2. Execute Cleanup nodes if required

3. Execute Setup nodes of all required Dependency nodes

The examples used in this chapter all refer to tests with the following preconditions and cleanup activities:

Sample

Dependency A:
Setup: start application if necessary
Cleanup: stop application

Dependency B:
Setup: log in user if necessary
Cleanup: log off user

Dependency C:
Setup: load application module 1 if necessary
Cleanup: close application module 1

Dependency D:
Setup: load application module 2 if necessary
Cleanup: close application module 2

Dependency E:
Setup: open dialog in module 2 if necessary
Cleanup: close dialog

Dependency C depends on B, B in turn on A.
Dependency E depends on D, D on B therefore also on A.

Before executing a Test case node QF-Test checks whether it has a Dependency node of its own and/or inherits one from its parent nodes. In that case QF-Test checks whether the Dependency node itself relies on other dependencies. Based on this analysis QF-Test generates a list of the dependencies required. This is done in step 1 of the example below.

Next, QF-Test checks if previous tests have already executed dependencies. If so, QF-Test checks if it has to execute any Cleanup nodes. After that QF-Test goes through all the setup nodes, starting with the most basic ones. The name of each Dependency executed is noted down in a list called dependency stack. See step 2 of below example.

Example: Test case 1
    Test of application module 1. First test case to be executed.

1. step
    Analyze the dependencies: the dependencies A-B-C have to be executed.

2. step
    Compare the dependencies to be executed with the dependency stack: In this example the dependency stack is still empty as the test case is the first one to be executed.

3. step
    Execute the Setup nodes, starting with A (start application), then B (login user (user name: Standard)), and last C (load application module 1).
    The dependency stack now reads A-B-C.

4. step
    Execute the Test case.

In the run log you can see exactly what QF-Test did:
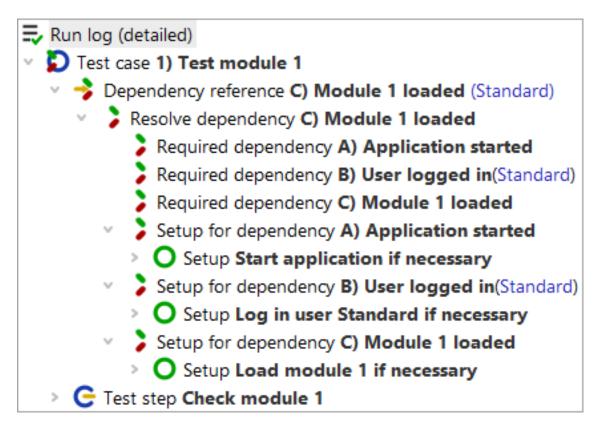
Figure 8.5: Dependency stack A-B-C

After executing the test case the application remains in the condition the last test case left it in. Only after analyzing the dependencies of the next test case Cleanup nodes might be run and the respective Dependency be deleted from the dependency stack. When Cleanup nodes need to be run they are executed in reverse order to the Setup nodes. After maybe clearing up dependencies no longer needed the Setup nodes of all required Dependencies are executed. Just like a manual tester will check that all requirements for the next test case are fulfilled QF-Test will do the same. A manual tester may not be conscious of checking the basic requirements. However, if he notices that the last test case left the application in a very bad state like a deadlock, he will probably kill the process if nothing else helped and start it again. To this end QF-Test explicitly runs all Setup nodes. These should be implemented in a way that they first check if the application is already in the required state and just in case not run the whole Setup node.
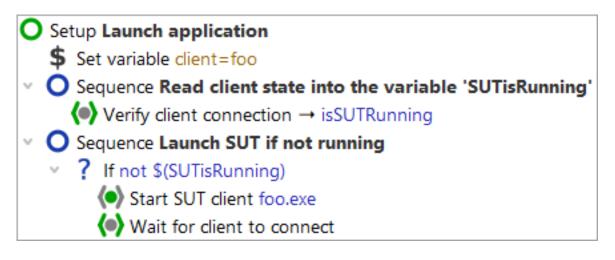
Figure 8.6: Good practice Setup node

Setups nodes should first check if the required condition already exists before actually executing the node. Cleanup nodes should first check if the requested cleanup action (e.g. closing a dialog) has already been performed. Also they should be programmed in such a way that they are in grade of clearing up error states of the application (e.g. error messages) so that a failed test case will not affect the following ones.

Example: test case 2
Test a dialog in application module 2

1. step:
   Analyze the dependencies: the dependencies A-B-D-E have to be executed.

2. step:
   Compare the list of dependencies to be executed with the dependency stack:

Dependency C is not required for test case 2. Therefore Cleanup node of Dependency C is executed (close application module 1).

3. step:
   Execute the Setup nodes, starting with A (check the application is already started and skip the rest of the setup), then B (check the user is already logged in, and skip the rest of the setup), then D (check if application module 2 is loaded and as it is not execute the complete Cleanup node), then E (same as D).

4. step:
   Execute the Test case.

You can see in the run log that the cleanup was done:
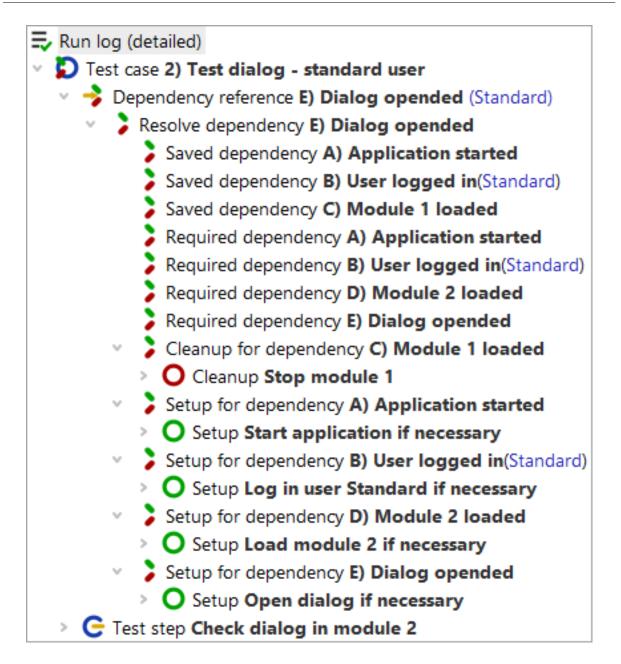
Figure 8.7: Dependency stack A-B-D-E

8.6.4   Characteristic variables

Values of certain variables may determine whether a dependency has to be cleared up and the setup re-executed, like the user name for dependency B 'Login'. These variables are called Characteristic variables. The values of the Characteristic variables are always taken into account when comparing dependency stacks. Two Dependencies on the stack

are only considered identical if the values of all Characteristic variables from the previous and the current run are equivalent. Consequently it is also possible for a Dependency to directly or indirectly refer to the same base Dependency with different values for its Characteristic variables. In that case the base Dependency will appear multiple times in the linearized dependency stack.

Furthermore, QF-Test stores the values of the Characteristic variables during execution of the Setup of a Dependency. When the Dependency is rolled back, i.e. its Cleanup node is executed, QF-Test will ensure that these variables are bound to the same value as during execution of the Setup. This ensures that a completely unrelated Test case with conflicting variable definitions can be executed without interfering with the execution of the Cleanup nodes during Dependency rollback. Consider for example the commonly used "client" variable for the name of an SUT client. If a set of tests for one SUT has been run and the next test will need a different SUT with a different name, the "client" variable will be changed. However, the Cleanup node for the previous SUT must still refer to the old value of "client", otherwise it wouldn't be able to terminate the SUT client. This is taken care of automatically as long as "client" was added to the list of Characteristic variables.

Example: Test case 3:
Test of the same dialog for the user Administrator.

1. Step:
   Analyze the dependencies: same list of dependencies A-B-D-E as in Test case 2. However, the Characteristic variables of Dependency B has a different value, i.e. 'Administrator'.

2. Step:
   Compare the dependency list with the dependency stack: The required Dependency B differs from the Dependency B saved on the dependency stack because of the values of the Characteristic variables 'username', which is 'Standard' on the dependency stack. This means that the dependency stack will be rolled back including Dependency B, starting with the Cleanup for Dependency E (close dialog), then Cleanup for Dependency D (stop module 2), then Cleanup for Dependency B (log off user - the variable 'username' then has the value 'Standard' saved via the Characteristic variables on the dependency stack).

3. Step:
   Execute the Setup nodes, starting with A (check the application is already started and skip the rest of the setup), then B (log in user 'Administrator'), then D (load module 2), then E (open dialog).

4. Step:
   Execute the Test case.

In the run log you can see the values of the Characteristic variables behind the respective Dependency:

Run log (detailed)
- Test case **3) Test dialog - admin**
  - Dependency reference **E) Dialog opended** (Administrator)
    - Resolve dependency **E) Dialog opended**
      - Saved dependency **A) Application started**
      - Saved dependency **B) User logged in**(Standard)
      - Saved dependency **D) Module 2 loaded**
      - Saved dependency **E) Dialog opended**
      - Required dependency **A) Application started**
      - Required dependency **B) User logged in**(Administrator)
      - Required dependency **D) Module 2 loaded**
      - Required dependency **E) Dialog opended**
      - Cleanup for dependency **E) Dialog opended**
        - Cleanup **Close dialog**
      - Cleanup for dependency **D) Module 2 loaded**
        - Cleanup **Stop module 2**
      - Cleanup for dependency **B) User logged in**(Standard)
        - Cleanup **Log off user Standard**
      - Setup for dependency **A) Application started**
        - Setup **Start application if necessary**
      - Setup for dependency **B) User logged in**(Administrator)
        - Setup **Log in user Administrator if necessary**
      - Setup for dependency **D) Module 2 loaded**
        - Setup **Load module 2 if necessary**
      - Setup for dependency **E) Dialog opended**
        - Setup **Open dialog if necessary**
  - Test step **Check dialog in module 2**

Figure 8.8: Dependency with Characteristic variables

Other examples for Characteristic variables are JDK versions when the SUT needs to be tested for various JDK versions or the browser name with web applications. In our example these would be specified as Characteristic variables for Dependency A.

8.6.5   Forced cleanup

In some use cases it may be necessary to execute the Cleanup node of a Dependency after each Test case. Then you should set the attribute Forced cleanup.

If Forced cleanup is activated for a Dependency node on the list of dependencies the Cleanup node of this and maybe of subsequent Dependencies will be executed.

Example:
> In this example the test logic requires module 2 to be stopped after test execution. The attribute Forced cleanup is activated for Dependency D.

> In our example the Cleanup nodes of Dependencies E (close dialog) and D (stop modul) would be executed after each Test case.

8.6.6   Rolling back Dependencies

QF-Test rolls back Dependencies depending on the needs of the Test cases.

If you want to clear the list of dependencies explicitly there are two ways to do it:

- The menu item Run→Roll back dependencies rolls back the list of dependencies 'cleanly' executing all the Cleanup nodes in reverse order to the setup activities.

- The menu item Run→Reset dependencies just deletes the list of dependencies without executing any nodes.

When a Test case does not use Dependencies the list of dependencies remains untouched, i.e. no Cleanup nodes are executed.

8.6.7   Error escalation

Another thing that is just grand about Dependencies is the convenient way that errors can be escalated without any additional effort. Let's again consider the example from the previous section after the first dependency stack has been initialized to A-B-C (Application started, user logged in, module one loaded) and the Setups have been run. Now what happens if the SUT has a really bad fault, like going into a deadlock and not reacting to user input any longer?

When a Cleanup node fails during rollback of the dependencies stack, QF-Test will roll back an additional Dependency and another one if that fails again and so on until the stack has been cleared. Similarly, if one of the Setups fails, an additional Dependency is rolled back and the execution of the Setups started from scratch.

Example:

In the example Test case 1 above the SUT would for example get deadlocked. In Test case 1 an exception would be thrown, Test case 1 would be stopped and execution passed on to Test case 2.

1. Step:
   Analyze the dependencies: the list of dependencies A-B-D-E has to be executed. (Application started, user logged in, module 2 loaded, dialog opened)

2. Step:
   Comparing the dependency list with the dependency stack set up by Test case 1 A-B-C (application started, user logged in, module 1 loaded) results in the execution of the Cleanup node of Dependency C (stop module 1). Of course, the exception is thrown again. Now QF-Test runs the Cleanup node for the next Dependency B (log off user). This will fail again so that now the basic Dependency A will be rolled back, which successfully stops the application.

3. Step:
   Execute the Setup nodes, starting with A (start the application), then B (log in user), then D (load module 2), then E (open dialog).

4. Step:
   Test case 2 will be executed despite the deadlock in Test case 1.

Figure 8.9: Exception in forced cleanup sequence of C causes B to clean up

For this to work it is very important to write Cleanup sequences in a way that ensures that either the desired state is reached or that an exception is thrown and that there is a more basic dependency with a more encompassing Cleanup. For example, if the Cleanup node for the SUT Dependency just tries to cleanly shut down the SUT through its File->Exit menu without exception handling and further safeguards, an exception in that sequence will prevent the SUT from being terminated and possibly interfere with all subsequent tests. Instead, the shutdown should be wrapped in a Try/Catch with a Finally node that checks that the SUT is really dead and if not, kills the process as a last resort.
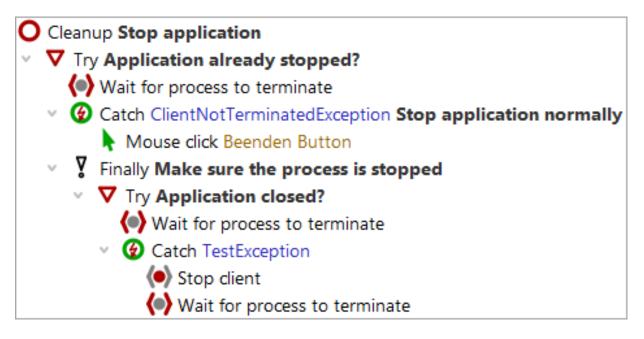
Figure 8.10: Typical Cleanup node

With good error handling in place, Test cases will rarely interfere with each other even in case of really bad errors. This helps avoid losing a whole night's worth of test runs just because of a single error.

8.6.8   Error handling

Besides supporting automatic escalation of errors a Dependency can also act as an error or exception handler for the tests that depend on it. Catch nodes, which can be placed at the end of a Dependency, are used to catch and handle exceptions thrown during a test. Exceptions thus caught will still be reported as exceptions in the run log and the report, but they will not interfere with subsequent tests or even abort the whole test run.

An Error handler node is another special node that may be added to a Dependency after

the Cleanup and before the Catch nodes. It will be executed whenever the result of a Test case is "error". In case of an exception, the Error handler node is not executed automatically because that might only cause more problems and even interfere with the exception handling, depending on the kind of exception. To do similar things for errors and exception, implement the actual handler as a Procedure and call it from the Error handler and the Catch node. Error handlers are useful for capturing and saving miscellaneous states that are not automatically provided by QF-Test. For example, you may want to create copies of temporary files created during execution of your SUT that may hold information pertaining to the error.

Only the topmost Error handler that is found on the dependency stack is executed, i.e. if in a dependency stack of [A,B,C,D] both A and C have Error handlers, only C's Error handler is run. Otherwise it would be difficult to modify the error handling of the more basic Dependency A in the more specialized Dependency C. To reuse A's error handling code in C, implement it as a Procedure.

8.6.9   Name spaces for Dependencies

Note
You might be interested in reading this section in case you want to run several SUTs at the same time where you do not want the Dependency node for a test on one of the SUTs to trigger cleanup actions for another SUT. Otherwise feel free to skip it.

A typical use case would be the test of whole process chains over several applications.

Consider the following situation: Sales representatives enter data for offers via a web application into a database at headquarters. There, the offers will be completed, printed and posted. A copy of each printed offer will be saved in a document management system (DMS).
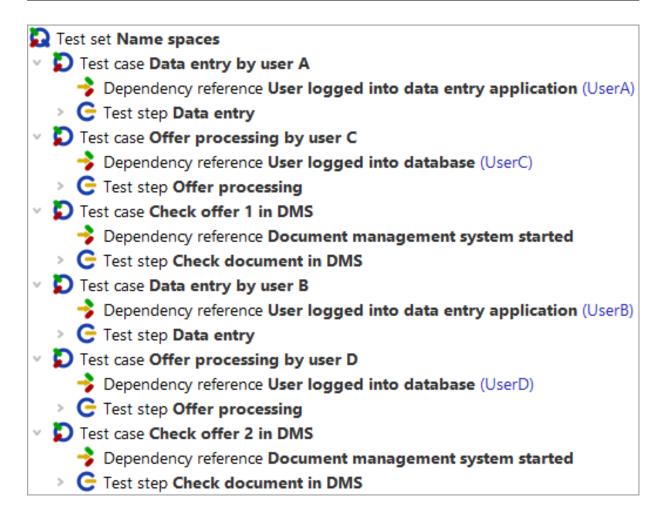
Figure 8.11: Example Test set for name spaces

In above example two sales representatives (UserA and UserB) enter offers and two different persons (UserC and UserD) process the offers at headquarters. Then the offers will be checked in the document management system. Since you do not want the dependencies of the test cases to interfere with one another you need to add a suitable name in the Dependency namespace[(553)] attribute of each Dependency reference node.

After running the test set you can see in the run log that a dependencies stack was set up in the name space 'data entry' for the first test case:

Figure 8.12: Dependency handling for test case 'Data entry by User A'

A dependencies stack is set up in the name space 'database' for the second test case. The dependencies stack in the name space 'data entry' remains unheeded. Looking at the applications, this means the database is started whereas the application for data entry is left as it is.

Figure 8.13: Dependency handling for test case 'Offer processing by User C'

A dependencies stack is set up in the name space 'DMS' for the third test case. The dependencies stacks in the name spaces 'data entry' and 'database' remain unheeded. Looking at the applications, this means the document management system is started whereas the other two applications are left as they are.

Figure 8.14: Dependency handling for test case 'Check offer 1 in DMS'

In test case number four the required dependencies are checked against the ones on the dependencies stack in the name space 'data entry' of the first test case. The dependencies stacks in the other two name spaces remain unheeded. Looking at the applications, this means User A is logged off, User B is logged into the data entry application and the other two applications are left as they are.
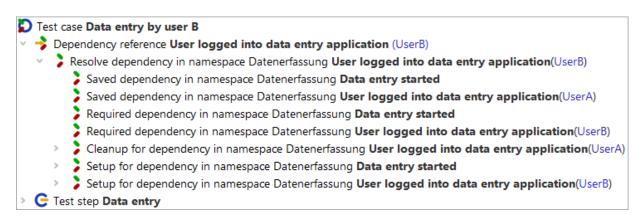
Figure 8.15: Dependency handling for test case 'Data entry by User B'

In test case number five the required dependencies are checked against the ones on the dependencies stack in the name space 'database' of the second test case. The dependencies stacks in the other two name spaces remain unheeded. Looking at the applications, this means User C is logged off, User D is logged into the database application and again the other two applications are left as they are.

Figure 8.16: Dependency handling for test case 'Offer processing by User D'

In the last test case the required dependencies are checked against the ones on the dependencies stack in the name space 'DMS' of the third test case. The dependencies

stacks in the other two name spaces remain unheeded. Looking at the applications, this means no clean up action has to be done on the DMS. The other two applications are left as they are, anyway.

Figure 8.17: Dependency handling for test case 'Check offer 2 in DMS'

8.7   Documenting test suites

Like with any programming-related task it is important for successful test-automation to properly document your efforts. Otherwise there is a good chance (some might say a certainty) that you will lose the overview over what you have done so far and start re-implementing things or miss out tests that should have been automated. Proper documentation will be invaluable when working through a run log, trying to understand the cause of a failed test. It will also greatly improve the readability of test reports.

An easy option for readable and documented tests is to group the recorded nodes into Sequence⁽⁵³⁶⁾ und Test step⁽⁵³⁹⁾ nodes.

For inline documentation you can use the Comment⁽⁷⁵¹⁾ node.

When you want to set up a documentation available outside QF-Test you can do so based on the Comment⁽⁵³¹⁾ attributes of Test set⁽⁵²⁵⁾, Test case⁽⁵¹⁸⁾, Package⁽⁵⁹⁴⁾ and Procedure⁽⁵⁸⁶⁾ nodes, and create a set of comprehensive HTML documents that will make all required information readily available. The various kinds of documents and the methods to create them are explained in detail in chapter 22⁽²⁷¹⁾.

Chapter 9

Projects

Projects provide a better overview, improve navigation between test suites and expand the scope for search and replace operations. Also, QF-Test automatically manages dependencies resulting from includes or absolute references between test suites that belong to the same project (see section 24.1[(298)]). Many other features have already been implemented or are under development.

Technically a QF-Test project is a set of test suites located in one or more directories with a common root. There is a 1:1 relation between the project and its directory and the name of the directory automatically becomes the name of the project.

To create a new project, select the menu item $\boxed{\text{File}\rightarrow\text{New project...}}$ and choose the directory. QF-Test then creates a file called `qftest.qpj` in that directory which identifies it as a project. All test suites located below that directory, except those specified in the option Project files and directories to exclude[(422)] automatically belong to this project. Please see section 39.1.1[(421)] for options affecting projects, including the exclusion list.

A subproject is sub-directory of a project which is itself a project. Test suites within a subproject also belong to all outer projects containing the subproject. *The* project of a test suite is the innermost subproject it belongs to. Automatic dependency resolution always covers the whole outermost project of a suite including all subprojects.

Figure 9.1: The project view

The project view with one or more projects can be turned on or off via the menu item View→Show projects. The project tree shows the hierarchy of directories and test suites starting from the project root, possibly limited by the filter at the top of the tree which matches on test suite names. Double clicking a test suite opens it, as does pressing the Return key. You can select several files or directories to be opened in one go, including all test suites located below the selected directories.

The hierarchy is refreshed automatically at intervals defined in the option Project refresh interval (s)[(422)]. You can refresh a directory including its complete hierarchy at any time by selecting it and pressing F5. For a more thorough rescan that does not rely on modification times but may take significantly longer for large projects, press Shift-F5 instead.

To switch keyboard focus back and forth between the test suite and the project view, press F6. Via Shift-F6 you can navigate to the node representing the current test suite in the project tree. If necessary, project view and project are automatically shown.

Chapter 10

The standard library

The standard library `qfs.qft`, a test suite that is part of the QF-Test distribution, con-
tains many useful procedures for a diverse set of tasks.

Figure 10.1: Standard library `qfs.qft`

Among others there are procedures for accessing and checking components (AWT, Swing, JavaFX, SWT, Web) in a generic manner, file system and database access, logging messages or screenshots to the run log and report and for performing cleanup.

A complete description of all packages and procedures including parameters and return values is provided in the library's HTML documentation, also accessible from the QF-Test Help menu. The latest version is also available online.

qfs.qft is included by default in every newly created test suite. As its directory is on the <u>library path</u>[(436)], specifying just qfs.qft in the <u>Include files</u>[(516)] of the Test suite node is sufficient.

Note   All procedures referring to an SUT use the generic variable $(client) as an implicit parameter. You must make sure that this variable is set correctly either globally or locally or specified as an explicit parameter in the procedure call.

Chapter 11

Scripting

The video

> ▷ 'Scripting in QF-Test (Basics)'
> https://www.qfs.de/en/yt/scripting-basics-45.html

explains the basic concepts about scripting.

If you want to know more about scripting have a look at the video

> ▷ 'Scripting in QF-Test (Advanced)
> https://www.qfs.de/en/yt/scripting-advanced-47.html

explains the basic concepts about scripting.

One of QF-Test's benefits is that complex tests can be created without writing a single line of code. However, there are limits to what can be achieved with a GUI alone. When testing a program which writes to a database, for example, one might want to verify that the actual values written to the database are correct; or one might want to read values from a database or a file and use these to drive a test. All this and more is possible with the help of powerful scripting languages like Jython, Groovy or JavaScript.

4.2+ While Jython is supported since the beginning of QF-Test, Groovy has found its way into QF-Test a bit later (QF-Test version 3). This language might be more convenient than Jython for those who are familiar with Java. Version 4.2 enabled JavaScript which might be more suitable for web developers. It's mainly a matter of individual preference whether to utilize Jython, Groovy or JavaScript scripting inside QF-Test.

In this chapter the basics of the scripting features available in all supported languages are explained. Most of the examples can be applied exactly or with few changes in other script languages. Methods calls which vary in syntax are exemplified in the affected languages. Particularities of the script languages are described in the sections Fundamentals of the Jython integration[(172)], Scripting with Groovy[(181)] and Scripting with JavaScript[(184)].
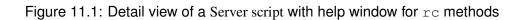
3.0+ The scripting language to use for a given Server script[(628)] or SUT script[(631)] node is determined by its Script language[(630)] attribute, so you can mix all three languages within a test suite. The default language to use for newly created script nodes can be set via the options Default script language for script nodes[(420)] and Default script language for conditions[(420)].

11.1 General

The approach to scripting in QF-Test is inverse from that of other GUI test tools. Instead of driving the whole test from a script, QF-Test embeds scripts into the test suite. This is achieved with the two nodes Server script[(628)] and SUT script[(631)].

Both nodes have a Script[(629)] attribute for the actual code.

Figure 11.1: Detail view of a Server script with help window for `rc` methods

3.0+ The internal script editor has some useful features to ease the typing of code. Reserved keywords, built-in functions, standard types, literals and comments are highlighted. Indentation is handled automatically inside of code blocks. With TAB and Shift-TAB respectively several selected lines can be indented manually.

However, the probably most useful feature - at least for the QF-Test newbie - might be the input assistance for many built-in methods. Type, for example, `rc.` and maybe some initial letters of a method name. Then press Ctrl-Space to open a pop-up window displaying the appropriate methods and descriptions of QF-Test's *run context* (cf. chapter 48[(915)]). Select one of the methods and confirm with Return to insert it into the script code. To get a list of all objects equipped with help, just press Ctrl-Space with the mouse cursor positioned after white space.

Server scripts are useful for tasks like calculating the values of variables or reading and parsing data from a file and using it to drive a test. SUT scripts on the other hand give full access to the components of the SUT and to every Java API that the SUT exposes. An SUT script might be used to retrieve or check values in the SUT to which QF-Test doesn't have access. The SUT script node has a Client[(632)] attribute which requires the name of the SUT client to run in.

Server scripts are run in script interpreters for the different script languages embedded in QF-Test itself, while SUT scripts are run in a script interpreter embedded in the SUT. These interpreters are independent of each other and do not share any state. However, QF-Test uses the RMI connection between itself and the SUT for seamless integration of SUT scripts into the execution of a test.

Through the menu items Extras→Jython console or Extras→Groovy console etc. you can open a window with an interactive command prompt for the language interpreters embedded in QF-Test. You can use the console to experiment with the scripts, get a feeling for the language, but also to try out some sophisticated stuff like setting up database connections. The keystrokes Ctrl-Up and Ctrl-Down let you cycle through previous input and you can also edit any other line or mark a region in the console and simply press Return to send it to the interpreter. In that case QF-Test will filter the '»>' and '...' prompts from previous interpreter output.

Similar consoles are available for each SUT client. The respective menu items are located below the Clients menu.

Note   When working in a SUT script terminal, there's one thing you need to be aware of: The commands issued to the interpreter are not executed on the event dispatch thread, contrary to commands executed via SUT script nodes. This may not mean anything to you and most of the time it doesn't cause any problems, but it may deadlock your application if you access any Swing or SWT components or invoke their methods. To avoid that, QF-Test provides the global method `runAWT` (and `runSWT` respectively) that executes arbitrary code on the event dispatch thread. For example, to get the number of visible nodes in a `JTree` component named `tree`, use `runAWT("tree.getRowCount()")` (or `runAWT { tree.getRowCount() }` in Groovy) to be on the safe side.

11.2 The run context `rc`

When executing Server scripts and SUT scripts, QF-Test provides a special environment in which a variable named `rc` is bound. This variable represents the *run context* which encapsulates the current state of the execution of the test. It provides an interface (fully documented in section 48.5[(917)]) for accessing QF-Test variables, for calling QF-Test procedures and can be used to add messages to the run log. To SUT scripts it also provides access to the actual Java components of the SUT's GUI.

For those cases where no run context is available, i.e. Resolvers, TestRunListeners, code executing in a background thread etc. QF-Test also provides a module called `qf` with useful generic methods for logging and other things. Please see section 48.6[(937)] for details.

11.2.1 Logging messages

One thing the run context can be used for is to add arbitrary messages to the run log that QF-Test generates for each test run. These messages may also be flagged as warnings or errors.

```
rc.logMessage("This is a plain message")
rc.logWarning("This is a warning")
rc.logError("This is an error")
```

Example 11.1: Logging messages from scripts

When working with compact run logs (see the option Create compact run log[(510)]), nodes which most likely will not be needed for error analysis may be deleted from the run log to preserve memory. This does not apply to error messages (`rc.logError`). They are kept, along with about 100 nodes preceding the error. Warnings (`rc.logWarning`) are also kept, however, without preceding nodes. Normal messages (`rc.logMessage`) may be subject to deletion. If you really need to make sure that a message will definitely be kept in the run log you can enforce this by specifying the optional second parameter `dontcompactify`, e.g.

```
rc.logMessage("This message will not be removed", dontcompactify=true)
# or simply
rc.logMessage("This message will not be removed", 1)
```

Example 11.2: Logging messages that will not get removed in compact run logs

11.2.2 Performing checks

Most of the time logging messages is tied to evaluating some condition. In that case, it is often desirable to get a result in the HTML or XML report equivalent to that of a Check node. The methods `rc.check` and `rc.checkEqual` will do just that:

```
var = 0
rc.check(var == 0, "Value of var is 0")
rc.checkEqual('${system:user.language}', 'en', "English locale required",
              rc.EXCEPTION)
```

Example 11.3: Performing checks

The optional last argument changes the error level in case of failure. Possible values are `rc.EXCEPTION`, `rc.ERROR`, `rc.OK` or `rc.WARNING`.

11.2.3 Variables

QF-Test has different kinds of variables. On the one hand you find variables belonging to the QF-Test environment and on the other variables of the script languages. Variables of the script languages are separated in server and SUT side variables of the specific script interpreter. The following graphic clarifies theses differences:
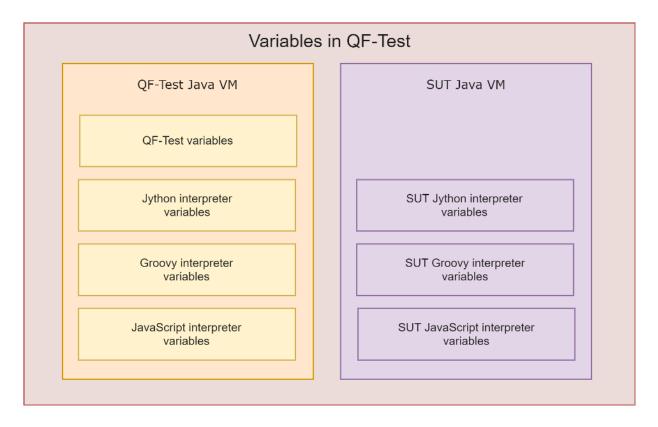
Figure 11.2: Overview of the types of variables in QF-Test

To share the different kinds of variables between QF-Test and the script interpreters provides the `rc` object which has several methods for the purpose. The methods are explained in the next section.

Accessing variables

Using QF-Test variables in scripts is not difficult. You can use the run context's `lookup` method (see section 48.5[(917)] for API reference) whenever you want to access a QF-Test value as a string.

```
# access a simple variable
text = rc.lookup("someText")
# access a property or resource
version = rc.lookup("qftest", "version")
```

Example 11.4: Using `rc.lookup` to access string variables

Setting variables

To make the results of a script available during further test execution, values can be stored in global or local variables. The effect is identical to that of a <u>Set variable</u>[(768)] node. The corresponding methods in the run context are `rc.setGlobal` and `rc.setLocal`.

```
# Test if the file /tmp/somefile exists
from java.io import File
rc.setGlobal("fileExists", File("/tmp/somefile").exists())
```

Example 11.5: Using `rc.setGlobal`

After executing the above example `$(fileExists)` will expand to true if the file `/tmp/somefile` exists and to false if it doesn't.

To clear a variable, set it to `None`, to clear all global variables use `rc.clearGlobals()` from a Server script.

Global variables

Sometimes it is helpful to have a variable available in several scripting nodes of the same language. If the value of the variable is not a simple string or integer, it is normally not sufficient to use `rc.setGlobal(...)` to store it in a global QF-Test variable because the value will be converted to a string in the process. Instead, such a variable should be declared `global` as shown in the following example.

```
global globalVar
globalVar = 10000
```

Example 11.6: Global Jython variable

The `globalVar` is now accessible within all further scripting nodes of the same type (Server scripts or SUT scripts of the same client). For changing the value of `globalVar` within another script, the `global` declaration is necessary again. Otherwise, a new local variable is created instead of accessing the existing global. Use the `del` statement to remove a global Jython variable:

```
global globalVar
del globalVar
```

Example 11.7: Delete a global Jython variable

In Groovy and JavaScript the global variables declaration is even easier than in Jython. All variables that are not declared locally are assumed to be global.

```
myGlobal = 'global'
```

Example 11.8: Defining a global variable in Groovy or JavaScript

```
assert myGlobal == 'global'
def globals = binding.variables
assert globals['myGlobal'] == 'global'
globals.remove('myGlobal')
assert globals.find { it == 'myGlobal' } == null
```

Example 11.9: Usage and deletion of a global Groovy variable

Exchanging variables between interpreters

Sometimes one would like to use variable values that have been defined in one interpreter in a different interpreter. For example, an SUT script might have been used to create a list of items displayed in a table. Later we want to iterate over that list in a Server script.

To simplify such tasks, the run context provides a symmetrical set of methods to access or set global variables in a different interpreter. For SUT scripts these methods are named `toServer` and `fromServer`. The corresponding Server script methods are `toSUT` and `fromSUT`.

The following example illustrates how an SUT script can set a global variable in the QF-Test Jython interpreter:

```
cellValues = []
table = rc.lookup("idOfTable")
for i in range(table.getRowCount()):
    cellValues.append(table.getValueAt(i, 0))
rc.toServer(tableCells=cellValues)
```

Example 11.10: Setting a server variable from an SUT script

After the above script is run, the global variable named "tableCells" in the QF-Test Jython interpreter will hold the array of cell values.

Note The cell values in the above example are not necessarily strings. They could be numbers, date values, anything. Unfortunately Jython's *pickle* mechanism isn't smart enough to transport instances of Java classes (not even realizable ones), so the whole

exchange mechanism is limited to primitive types like strings and numbers, along with Jython objects and structures like arrays and dictionaries.

11.2.4 Accessing the SUT's GUI components

For SUT scripts the run context provides an additional method that is extremely useful. Calling `rc.getComponent("componentId")` will retrieve the information of the Component[(823)] node in the test suite with the QF-Test ID[(824)] "componentId" and pass that to QF-Test's component recognition mechanism. The whole process is basically the same as when simulating an event, including the possible exceptions if the component cannot be found.

If the component is located, it will be passed to Jython, not as some abstract data but as the actual Java object. All methods exposed by the Java API for the component's class can now be invoked to retrieve information or achieve effects which are not possible through the GUI alone. To get a list of a component's method see section 5.12[(96)].

```
# get the custom password field
field = rc.getComponent("tfPassword")
# read its encrypted value
passwd = field.getCryptedText()
rc.setGlobal("passwd", passwd)
# get the table component
table = rc.getComponent("tabAddresses")
# get the number of rows
rows = table.getRowCount()
rc.setGlobal("tableRows", rows)
```

Example 11.11: Accessing components with `rc.getComponent`

You can also access sub-items this way. If the `componentId` parameter references an item, the result of the `getComponent` call is a pair, the component and the item's index. The index can be used to retrieve the actual value. The following example shows how to get the value of a table cell. Note the convenient way Jython supports sequence unpacking during assignment.

```
# first get the table and index
table, (row,column) = rc.getComponent("tableAddresses@Name@Greg")
# then get the value of the table cell
cell = table.getValueAt(row, column)
```

Example 11.12: Accessing sub-items with `rc.getComponent`

11.2.5 Calling Procedures

The run context can also be used to call back into QF-Test and execute a Procedure[(586)] node.

```
rc.callProcedure("text.clearField",
        {"component": "nameField", "message" : "nameField cleared"})
```

Example 11.13: Simple procedure call in Jython

In the example above the Procedure named "clearField" in the Package[(594)] named "text" will be called. The parameter named "component" is set to the value "nameField" and the parameter named "message" is set to the value "nameField cleared".

The same example with Groovy syntax:

```
rc.callProcedure("text.clearField",
        ["component" : "nameField", "message" : "nameField cleared"])
```

Example 11.14: Simple procedure call in Groovy

And in JavaScript:

```
rc.callProcedure("text.clearField",
        {"component" : "nameField", "message" : "nameField cleared"})
```

Example 11.15: Simple procedure call in JavaScript

The value returned by the Procedure through a Return[(592)] node is returned as the result of the `rc.callProcedure` call.

Note Great care must be taken when using `rc.callProcedure(...)` in SUT script nodes. Only short-running Procedures should be called that won't trigger overly complex actions in the SUT. Otherwise, a `DeadlockTimeoutException`[(852)] might be caused. For data-driven tests where for some reason the data must be determined in the SUT, use `rc.toServer(...)` to transfer the values to QF-Test interpreter, then drive the test from a Server script node where these restrictions do not apply.

11.2.6 Setting options

3.1+

Many of the options described in chapter 39[(417)] can also be set at runtime via `rc.setOption`. Constants for option names are predefined in the class `Options`. It is automatically available for all script languages.

A real-life example where this might be useful is if you want to replay an event on a disabled component, so you need to temporarily disable QF-Test's check for the enabled/disabled state. For setting and immediately resetting an option there is the variant `pushOption` / `popOption` which leaves a potentially preceding `setOption` call intact:

```
rc.pushOption(Options.OPT_PLAY_THROW_DISABLED_EXCEPTION, false)
```

Example 11.16: Example for pushOption

After replaying this special event, the previous option setting can be restored as shown in the following example:

```
rc.popOption(Options.OPT_PLAY_THROW_DISABLED_EXCEPTION)
```

Example 11.17: Example for popOption

To be on the safe side and ensure, that the value is always restored, the two script nodes should be placed into a Try[616] / Finally[623] combination. Otherwise, for example a `ComponentNotFoundException` during event replay would prevent restoring the option.

Note Be sure to set QF-Test options in a Server script node and SUT options in an SUT script node, otherwise the setting will have no effect. Some options - most notably for SmartIDs - have effect on QF-Test and SUT side. Those must be set in a Server script node. QF-Test automatically takes care of the SUT side as well. The option documentation in chapter 39[417] includes information about the effected side - server and/or SUT.

11.2.7 Override components

You might face a situation where you want to work with a component which you have to determine at script level before working with it, either for performance reasons when using the same component multiple times or for special cases where default component recognition is too difficult or inefficient. For such cases you can use the method `rc.overrideElement` to associate the component found with a QF-Test ID or SmartID after which you can work use the assigned ID QF-Test event, check or similar nodes.

Note The following example could alternatively be resolved using SmartIDs but still illustrates the case well. For more complex cases `overrideElement` remains a relevant alternative.

Let's imagine that we have a panel and we want to work with the first textfield, but because of changing textfields we cannot rely on the standard way of the recognition. Now we can implement a script, which looks for the first textfield and assigns that

textfield to the `PriorityAwtSwingComponent` from the standard library `qfs.qft`. Once we have executed that script we can work with any QF-Test nodes using the `PriorityAwtSwingComponent`, which actually performs all actions on the found textfield.

```
panel = rc.getComponent("myPanel")
for component in panel.getComponents():
    if qf.isInstance(component, "javax.swing.JTextField"):
        rc.overrideElement("PriorityAwtSwingComponent", component)
        break
```

Example 11.18: Using rc.overrideElement

This concept is very useful if you know an algorithm to determine the target component of your test steps.

You can find (old-style, see below) priority components for all engines in the standard library `qfs.qft`. You can also find an illustrative example in the provided demo test suite `carconfigSwing_advanced_en.qft`, located in the directory `demo/carconfigSwing` in your QF-Test installation.

7.0+   Before the introduction of SmartIDs QF-Test ID[(824)] of an existing Component[(823)] node had to be used as the `id` parameter. When using SmartIDs these are no longer necessary. You are free to assgin a pseudo SmartID as long as it starts with #. This functionality is based on simple string comparison. Potentially defined scopes are not taken into account! Also new in QF-Test 7.0 is the ability to query overridden elements via `rc.getOverrideElement`. Following is an example based on SmartID that overrides a component only if necessary.

```
if not rc.getOverrideElement("#FirstTextField"):
    panel = rc.getComponent("myPanel")
    for component in panel.getComponents():
if qf.isInstance(component, "javax.swing.JTextField"):
    rc.overrideElement("#FirstTextField", component)
    break
```

Example 11.19: Conditional rc.overrideElement with SmartID

11.3   Fundamentals of the Jython integration

Note   Jython is based on Python 2, not Python 3, so whenever just "Python" is mentioned in relation to Jython throughout this manual it refers to Python 2.

Python is an object oriented scripting language written in C by Guido van Rossum. A wealth of information including an excellent Python tutorial is available at

http://www.python.org. Python is a standard language that has been around for years with extensive freely accessible documentation. Therefore, this manual only explains how Jython is integrated into QF-Test, not the language itself. Python is a very natural language. Its greatest strength is the readability of Python scripts, so you should have no problems following the examples.

Jython (formerly called JPython) is a Java implementation of version 2 of the language Python. It has the same syntax as Python and almost the same set of features. The object systems of Java and Jython are very similar and Jython can be integrated seamlessly into applications like QF-Test. This makes it an invaluable tool for Java scripting. Jython has its own web page at http://www.jython.org. There is also an extensive tutorial available which may help you get started with this scripting language.

QF-Test uses Jython version 2.7 which supports a large majority of the standard Python 2 library.

The Jython language is not only used in Server script⁽⁶²⁸⁾ and SUT script⁽⁶³¹⁾ nodes but also in $[...] expressions and to evaluate conditions like the Condition⁽⁶⁰⁶⁾ attribute of an If⁽⁶⁰⁵⁾ node.

Note   In Jython scripts QF-Test variables with the syntax $(var) or ${group:name} are expanded before the execution of the script. This can lead to unexpected behavior, especially if the values of those variables contain multi-line strings or backslash characters ('\'). `rc.lookup(...)`, which will be evaluated during execution of the script, is the preferred method in this case (see section 11.2.3⁽¹⁶⁶⁾ for details).

11.3.1 Modules

Modules for Jython in QF-Test are just like standard Python modules. You can import the modules into QF-Test scripts and call their methods, which simplifies the development of complex scripts and increases maintainability since modules are available across test suites.

Modules intended to be shared between test suites should be placed in the directory `jython` under QF-Test's root directory. Modules written specifically for one test suite can also be placed in the test suite's directory. The version-specific directory `qftest-7.1.2/jython/Lib` is reserved for modules provided by Quality First Software GmbH. Jython modules must have the file extension `.py`.

The following Jython module defines a procedure sorting an array of numbers.

```
def insertionSort(alist):
    for index in range(1, len(alist)):
        currentvalue = alist[index]
        position = index
        while position > 0 and alist[position-1] > currentvalue:
            alist[position] = alist[position-1]
            position = position-1
        alist[position] = currentvalue
```

Example 11.20: The Jython module `pysort.py`

The procedure defined in above module is beeing called in the following Jython script:

```
import pysort
alist = [54,26,93,17,77,31,44,55,20]
pysort.insertionSort(alist)
print(alist)
```

Example 11.21: Jython script using a module

11.3.2 Post-mortem debugging of Jython scripts

Python comes with a simple line-oriented debugger called `pdb`. Among its useful features is the ability for post-mortem debugging, i.e. analyzing why a script failed with an exception. In Python you can simply import the `pdb` package and run `pdb.pm()` after an exception. This will put you in a debugger environment where you can examine the variable bindings in effect at the time of failure and also navigate up to the call stack to examine the variables there. It is somewhat similar to analyzing a core dump of a C application.

Though Jython comes with `pdb`, the debugger doesn't work very well inside QF-Test for various reasons. But at least post-mortem debugging of Jython scripts is supported from the Jython consoles (see section 11.3[(172)]). After a Server script[(628)] node fails, open QF-Test's Jython console, for a failed SUT script[(631)] node open the respective SUT Jython console, then just execute `debug()`. This should have a similar effect as `pdb.pm()` described above. For further information about the Python debugger please see the documentation for pdb at https://docs.python.org/2/library/pdb.html.

11.3.3 Boolean type

Jython now has a real boolean type with values `True` and `False` whereas in older versions integer values 0 and 1 served as boolean values. This can cause problems if

boolean results from calls like `file.exists()` are assigned to a QF-Test variable, e.g. "fileExists" and later checked in a $\underline{\text{Condition}}^{(606)}$ attribute in the form `$(fileExists) == 1`. Such conditions should generally be written as just `$(fileExists)` or `rc.getBool("fileExists")` which work well with all Jython versions.

11.3.4 Jython strings and character encodings

Summary and advice

5.3+ Characters in Jython literal strings like `"abc"` used to be limited to 8 bit, causing problems when trying to work with international characters.

QF-Test version 5.3 introduces a solution for international characters in Jython scripts and $\underline{\text{Condition}}^{(606)}$ attributes based on the option $\underline{\text{Literal Jython strings are unicode (16-bit as in Java)}}^{(420)}$.

If you start using QF-Test with version 5.3. or higher, that option is turned on by default.

A small percentage of existing scripts will need to be updated when switching to unicode literals, so if QF-Test encounters an existing older system configuration the option remains off until explicitly turned on. Turning the option on is strongly recommended. The "Trouble shooting" section below explains what to do in case you encounter problems.

If Jython unicode literals are activated, the option $\underline{\text{Default character encoding for Jython}}^{(421)}$ should be set to "utf-8" for maximum flexibility.

The main thing to avoid, regardless of the option setting, is expansion of QF-Test variables in literal Jython strings like `"$(somevar)"`. It can cause syntax errors or have unexpected results if the expanded variable contains newlines or backslash characters. Use `rc.lookup("somevar")` instead.

Background and history of Jython in QF-Test

In Java all strings are sequences of 16-bit characters, whereas Jython has two kinds of Strings: 8-bit "byte strings" (type <str>) and 16-bit "unicode strings" (type <unicode>). The majority of strings used in QF-Test Jython scripts are either string constants like `"abc"`, called literal strings, or Java string values converted to Jython, e.g. the result of `rc.lookup("varname")`. Conversion from a Java string always results in a 16-bit unicode Jython string. For literal strings the result depends on the setting of the option $\underline{\text{Literal Jython strings are unicode (16-bit as in Java)}}^{(420)}$.

When unicode and byte strings are compared or concatenated, Jython needs to convert one into the other. Conversion from unicode to byte strings is called encoding, the other way decoding. There are many different ways to encode 16-bit strings to 8-bit sequences and the rules to do so are called encodings. Common examples include "utf-8" or "latin-1". The option $\underline{\text{Default character encoding for Jython}}^{(421)}$ specifies the

default encoding to use. For backwards compatibility the default used to be "latin-1" before QF-Test 5.3 and is now "utf-8", which is preferable because it is the most flexible and supports all international character sets.

Jython in QF-Test is based on Python version 2. In early Python versions strings were made of 8-bit characters. Later, unicode strings with 16-bit characters were added. In Python 2 literal strings like ″abc″ are 8-bit byte strings, prepending 'u', i.e. `u″abc″` turns them into unicode strings. In Python 3 literal strings are unicode and one needs to prepend 'b', i.e. `b″abc″` to get 8-bit strings.

In Jython 2.2, Java strings were converted to 8-bit Python strings based on the default encoding of the Java VM, typically ISO-8859-1 (also known as latin-1) in western countries. Since Jython 2.5, every Java string gets interpreted as a unicode Jython string. With 8-bit literal string this results in a lot of implicit conversion between 8-bit and unicode strings, for example when concatenating a Java string - now unicode - and a literal string like `rc.lookup(″path″) + ″/file″`.

5.3+ Before QF-Test version 5.3 the Jython script nodes had further problems with characters outside the 8-bit range, because of the way scripts were passed from QF-Test to the Jython compiler. In the process of fixing these issues it was decided that the best way to reduce problems with Jython literal strings was to adapt a feature already available in Python 2, namely `from future import unicode_literals` and make it possible to treat Jython literal strings in QF-Test as unicode. This results in literal strings being the same in all three scripting languages of QF-Test and fully compatible with Java strings, so the interaction of Jython scripts with everything else in QF-Test gets far more natural. The new option <u>Literal Jython strings are unicode (16-bit as in Java)</u>[420] determines whether or not literal Strings in QF-Test Jython scripts are treated as unicode. For backwards compatibility reasons the default remains 8-bit if QF-Test encounters an exsiting system configuration, otherwise unicode literals are now the default.

The recommended Jython option settings are on for <u>Literal Jython strings are unicode (16-bit as in Java)</u>[420] and "utf-8" for <u>Default character encoding for Jython</u>[421].

Trouble shooting Jython encoding issues

As explained in the previous sections, Jython has two string types, `<type ′str′>` for 8-bit "byte" strings and `<type ′unicode′>` for 16-bit "unicode" strings. Literal strings can be prepended with 'b' (`b″abc″`) to get byte strings or with 'u' (`u″abc″`) for unicode strings. Plain literal strings (`″abc″`) are unicode if the option <u>Literal Jython strings are unicode (16-bit as in Java)</u>[420] is turned on and byte strings otherwise. Java strings resulting from Java function calls like `rc.lookup(″somevar″)` are unicode strings.

The following advice should help minimizing Jython string encoding issues:

- Turn the option <u>Literal Jython strings are unicode (16-bit as in Java)</u>[420] on and set the option <u>Default character encoding for Jython</u>[421] to "utf-8".

- Literal strings containing $() expansion like `″$(varname)″` have always been

problematic and should be replaced with `rc.lookup("varname")`.

- Strings containing Windows filenames need special treatment because of the backslash '\' character. In 8-bit strings backslashes are retained unless they have special meaning like '\t' for tab or '\n' for newline. In 16-bit strings there are several more special escape sequences that are likely to cause syntax errors or unexpected results. Issues are avoided by using `rc.lookup("filename")` (see above) and prepending 'r' (for "raw string") to literal strings, e.g. `qftestDir = r"C:\Program Files\QFS\QF-Test"`.

- Generally use `qf.println(...)` instead of `print ...` because the latter gets passed through an 8-bit stream with the default Java encoding (and in case of an SUT script[631] node also of the operating system) and thus may lose international characters on the way.

- Converting an object to a string in Jython was traditionally done via `str(some_object)`. As `str` is the byte string type this always creates a byte string and triggers encoding. Unless you specifically need a byte string it is much better to use `unicode(some_object)`.

- The `types` Jython module provides the constant `types.StringType` and `types.UnicodeType` as well as the list `types.StringTypes` cotaining both. The latter is very useful when checking if an object is any type of string, regardless of 8-bit or 16-bit. Instead of `if type(some_object) == types.StringType`
this should be written as
`if type(some_object) in types.StringTypes`

- In the very few cases where you really need a literal byte string, prepend a 'b', e.g. `array.array(b'i', [1, 2, 3])`

And of course our support is always there to help.

11.3.5 Getting the name of a Java class

This simple operation is surprisingly difficult in Jython. Given a Java object you would expect to simply write `obj.getClass().getName()`. For some objects this works fine, for others it fails with a cryptic message. This can be very frustrating. Things go wrong whenever there is another `getName` method defined by the class, which is the case for AWT `Component`, so getting the class name this way fails for all AWT/Swing component classes.

In Jython 2.2.1 the accepted workaround was to use the Python idiom `obj.__class__.__name__`. This no longer works in Jython 2.5 because it no

longer returns the fully qualified class name, only the last part. Instead of `java.lang.String` you now get just `String`. The only solution that reliably works for version 2.5 is:

```
from java.lang import Class
Class.getName(obj.getClass())
```

This also works for 2.2, but it is not nice, so we initiated a new convenience module with utility methods called `qf` that gets imported automatically. As a result you can now simply write

```
qf.getClassName(obj).
```

11.3.6   A complex example

We are going to close this section with a complex example, combining features from Jython and QF-Test to execute a data-driven test. For the example we assume that a simple table with the three columns "Name", "Age" and "Address" should be filled with values read from a file. The file is assumed to be in "comma-separated-values" format with "|" as the separator character, one line per table-row, e.g.:

```
John Smith|45|Some street, some town
Julia Black|35|Another street, same town
```

The example verifies the SUT's functionality in creating new table rows. It calls a QF-Test procedure that takes the three parameters, "name", "age", and "address", creates a new table-row and fills it with these values. Then the Jython script is used to read and parse the data from the file, iterate over the data-sets and call back to QF-Test for each table-row to be created. The name of the file to read is passed in a QF-Test variable named "filename". After filling the table, the script compares the state of the actual table component with the data read from the file to make sure everything is OK.

```
import string
data = []
# read the data from the file
fd = open(rc.lookup("filename"), "r")
line = fd.readline()
while line:
    # remove whitespace
    line = string.strip(line)
    # split the line into separate fields
    # and add them to the data array
    if len(line) > 0:
        data.append(string.split(line, "|"))
    line = fd.readline()
# now iterate over the rows
for row in data:
    # call a qftest procedure to create
    # one new table row
    rc.callProcedure("table.createRow",
                     {"name": row[0], "age": row[1],
                      "address": row[2]})
# verify that the table-rows have been filled correctly
table = rc.getComponent("tabAddresses")
# check the number of rows
rc.check(table.getRowCount() == len(data), "Row count")
if table.getRowCount() == len(data):
    # check each row
    for i in range(len(data)):
        rc.check(table.getValueAt(i, 0)) == data[i][0],
                "Name in row " + str(i))
        rc.check(table.getValueAt(i, 1)) == data[i][1],
                "Age in row " + str(i))
        rc.check(table.getValueAt(i, 2)) == data[i][2],
                "Address in row " + str(i))
```

Example 11.22: Executing a data-driven test

Of course, the example above serves only as an illustration. It is too complex to be edited comfortably in QF-Test and too much is hard-coded, so it is not easily reusable. For real use, the code to read and parse the file should be parameterized and moved to a module, as should the code that verifies the table.

This is done in the following Jython script with the methods `loadTable` to read the data from the file and `verifyTable` to verify the results. It is saved in a module named `csvtable.py`. An example module is provided in `qftest-7.1.2/doc/tutorial/csvtable.py`. Following is a simplified version:

```
import string
def loadTable(file, separator="|"):
    data = []
    fd = open(file, "r")
    line = fd.readline()
    while line:
        line = string.strip(line)
        if len(line) > 0:
            data.append(string.split(line,separator))
        line = fd.readline()
    return data
def verifyTable(rc, table, data):
    ret = 1
    # check the number of rows
    if table.getRowCount() != len(data):
        if rc:
            rc.logError("Row count mismatch")
        return 0
    # check each row
    for i in range(len(data)):
        row = data[i]
        # check the number of columns
        if table.getModel().getColumnCount() != len(row):
            if rc:
                rc.logError("Column count mismatch " +
                            "in row " + str(i))
            ret = 0
        else:
            # check each cell
            for j in range(len(row)):
                val = table.getModel().getValueAt(i, j)
                if str(val) != row[j]:
                    if rc:
                        rc.logError("Mismatch in row " +
                                    str(i) + " column " +
                                    str(j))
                    ret = 0
    return ret
```

Example 11.23: Writing a module

The code above should look familiar. It is an improved version of parts of example 11.22[(179)]. With that module in place, the code that has to be written in QF-Test is reduced to:

```
import csvtable
# load the data
data = csvtable.loadTable(rc.lookup("filename"))
# now iterate over the rows
for row in data:
    # call a qftest procedure to create
    # one new table row
    rc.callProcedure("table.createRow",
                     {"name": row[0], "age": row[1],
                      "address": row[2]})
# verify that the table-rows have been filled correctly
table = rc.getComponent("tabAddresses")
csvtable.verifyTable(rc, table, data)
```

Example 11.24: Calling methods in a module

11.4 Scripting with Groovy

Groovy is another established scripting language for the Java Platform. It was invented by James Strachan and Bob McWhirter in 2003. All you need for doing Groovy is a Java Runtime Environment (JRE) and the `groovy-all.jar` file. This library contains a compiler to create Java class files and provides the runtime when using that classes in the Java Virtual Machine (JVM). You may think of Groovy as being Java with an additional `.jar` file. In contrast to Java, Groovy is a dynamic language, meaning that the behavior of an object is determined at runtime. Groovy also allows to load classes from sources without creating class files. Finally, it is easy to embed Groovy scripts into Java applications like QF-Test.

The Groovy syntax is similar to Java, maybe more expressive and easier to read. When coming from Java you can embrace the Groovy style step by step. Of course we cannot explain all aspects of the Groovy language here. For in-depth information, please take a look at the Groovy home page at http://groovy-lang.org/ or read the excellent book "Groovy in Action" by Dierk Koenig and others. Perhaps the following tips may help a Java programmer getting started with Groovy.

- The semicolon is optional as long as a line contains only one statement.

- Parentheses are sometimes optional, e. g. `println 'hello qfs'` means the same as `println('hello qfs')`.

- Use `for (i in 0..<len) { ...  }` instead of `for (int i = 0; i < len; i++) { ...  }`.

- The following imports are made by default: `java.lang.*`, `java.util.*`,

> java.io.*, java.net.*, groovy.lang.*, groovy.util.*,
> java.math.BigInteger, java.math.BigDecimal.

- Everything is an object, even integers like '1' or booleans like 'true'.

- Instead of using getter and setter methods like `obj.getXxx()`, you can simply write `obj.xxx` to access a property.

- The operator `==` checks for equality, not identity, so you can write `if (somevar == "somestring")` instead of `if (somevar.equals("somestring"))`. The method `is()` checks for identity.

- Variables have a dynamic type when being defined with the `def` keyword. Using `def x = 1` allows for example to assign a `String` value to the variable `x` later in the script.

- Arrays are defined differently from Java, e. g. `int[] a = [1, 2, 3]` or `def a = [1, 2, 3] as int[]`. With `def a = [1, 2, 3]` you define a `List` in Groovy.

- Groovy extends the Java library by defining a set of extra methods for many classes. Thus, you can for example apply an `isInteger()` method to any `String` object in a Groovy script. That's what is called *GDK* (according to the *JDK* in Java). To get a list of those methods for an arbitrary object `obj`, you can simply invoke `obj.class.metaClass.metaMethods.name` or use the following example:

```
import groovy.inspect.Inspector
def s = 'abc'
def inspector = new Inspector(s)
def mm = inspector.getMetaMethods().toList().sort() {
    it[Inspector.MEMBER_NAME_IDX] }
for (m in mm) {
    println(m[Inspector.MEMBER_TYPE_IDX] + ' ' +
            m[Inspector.MEMBER_NAME_IDX] +
            '(' + m[Inspector.MEMBER_PARAMS_IDX] + ')')
}
```

Example 11.25: GDK methods for a `String` object

- Inner classes are not supported, in most cases you can use *Closures* instead. A `Closure` is an object which represents a piece of code. It can take parameters and return a value. Like a block, a `Closure` is defined with curly braces `{ ... }`. Blocks only exist in context with a `class`, an `interface`, static or object initializers, method bodies, `if`, `else`, `synchronized`, `for`, `while`, `switch`, `try`,

catch, and `finally`. Every other occurrence of `{...}` is a `Closure`. As an example let's take a look at the `eachFileMatch` GDK method of the `File` class. It takes two parameters, a filter (e. g. a `Pattern`) and a `Closure`. That `Closure` takes itself a parameter, a `File` object for the current file.

```
def dir = rc.lookup('qftest', 'suite.dir')
def pattern = ~/.*\.qft/
def files = []
new File(dir).eachFileMatch(pattern) { file ->
    files.add(file.name)
}
files.each {
    // A single Closure argument can also be referred with "it"
    rc.logMessage(it)
}
```

Example 11.26: Closures

• Working with `Lists` and `Maps` is simpler than in Java.

```
def myList = [1, 2, 3]
assert myList.size() == 3
assert myList[0] == 1
myList.add(4)
def myMap = [a:1, b:2, c:3]
assert myMap['a'] == 1
myMap.each {
    this.println it.value
}
```

Example 11.27: Working with lists and maps

11.4.1 Groovy packages

Just like Java classes, Groovy source files (`.groovy`) can be organized in packages. Those intended to be shared between test suites should be placed in the directory `groovy` under QF-Test's root directory. Others that are written specifically for one test suite can also be placed in the directory of the test suite. The version-specific directory `qftest-7.1.2/groovy` is reserved for Groovy files provided by Quality First Software GmbH.

```
package my
class MyModule
{
    public static int add(int a, int b)
    {
        return a + b
    }
}
```

Example 11.28: `MyModule.groovy`

The file `MyModule.groovy` could be saved in a sub directory `my` below the suite directory. Then you can use the `add` method from `MyModule` as follows:

```
import my.MyModule as MyLib
assert MyLib.add(2, 3) == 5
```

Example 11.29: Using MyModule

This code also shows another groovy feature: *Type aliasing*. By using `import` and `as` in combination you can reference a class by a name of your choice.

11.5   Scripting with JavaScript

JavaScript has become the most widely used programming language in the web area and is one of the most popular script languages. QF-Test supports scripting with EC-MAScript, which provides a common standard for the variety of different implementations of JavaScript.

QF-Test must run with at least Java 8 to use JavaScript.

It is possible to write code for the ECMAScript 6 standard. QF-Test automatically transpiles the code to the EcmaScript 5 standard before the execution.

Special features of JavaScript as compared to other programming languages:

- There are two different null values: `undefined` and `null`. A variable is `undefined` when it has no value. `null` is an intended null value that has to be assigned.

- The `==` operator checks for equality instead of identity. So you can use `if (somevar == "somestring")` to check for equality. To check for identity use the `===` operator.

- Variables declared with the `let` keyword are dynamically typed. E.g. `let x = 1` makes it possible to assign `String` to x. Constants can be declared with `const`.

11.5.1 JavaScript imports

The following example shows how functionality can be transfered in a module. The module must be placed in the `javascript` directory inside the QF-Test root directory. The module can look like this:

```
var fibonacci = function(n) {
   return n < 1 ? 0
       : n <= 2 ? 1
       : fibonacci(n - 1) + fibonacci(n - 2);
}
function sumDigits(number) {
  var str = number.toString();
  var sum = 0;
  for (var i = 0; i < str.length; i++) {
    sum += parseInt(str.charAt(i), 10);
  }
  return sum;
}
// Module exports (Node.js style)
exports.fibonacci = fibonacci;
exports.sumDigits = sumDigits;
```

Example 11.30: The `moremath.js` module

The `moremath.js` module defines the two function: `fibonacci` and `sumDigits`. Each function has to be exported to . This can be achieved via Node.js like function `exports`.

The following code can now be used inside the script node to take advantage of the `moremath.js` modules functions:

```
moremath = require('moremath');
console.log(moremath.fibonacci(13));
console.log(moremath.sumDigits(123));
```

Example 11.31: Usage of the moremath.js module

There are multiple ways to import modules. Modules provided by QF-Test can be imported using the `import` function.

```
import {Autowin} from 'autowin';
Autowin.doClickHard(0, 0, true);
```

Example 11.32: `Using the autowin module`

Java classes can also be imported using the import function.

```
import {File} from 'java.io';
```

Example 11.33: `Importing Java classes`

It is also possible to use the "require" function for importing npm modules, which are explained in the following section.

11.5.2 NPM modules

npm is a package manager for JavaScript with over 350.000 packages. The available packages are listed here https://www.npmjs.com/. The packages can be used in QF-Test scripts. They need to be installed in the javascript folder of the QF-Test root directory.

```
npm install underscore
```

This line installs the npm underscore package from the os command line.

There are a few npm modules that are incompatible with the ECMAScript standard as they were written for Node.js.

```
_ = require('underscore');
func = function(num){ return num % 2 == 0; }
let evens = _.filter([1, 2, 3, 4, 5, 6], func);
console.log(evens);
```

Example 11.34: `Usage of the 'underscore' package`

11.5.3 Print statements

Besides `console.log()` there is another method implemented in QF-Test to show output on the terminal. Note that this `print` is not defined in ECMAScript and was added for convenience in QF-Test.

```
print([1,2,3,4]);
```

Example 11.35: `Printing an array`

11.5.4   Execution

JavaScript scripts are not executed inside the browser but in a specific engine on the server or SUT side. This engine is called Oracle Nashorn Engine and comes with JDK 8. It allows the execution of EcmaScript directly in the JVM.

Chapter 12

Unit Tests

With Unit Tests, i.e. component tests, you can check the functional units. They explicitly test the functionality of single components. For this reason they are much less complex compared to integration and system tests.

The Unit test[(790)] node executes Unit Tests via the JUnit framework as part of a QF-Test test run. The results are available in the run log as well as in the report. In section 27.5[(346)] you find information on how to include QF-Test test suites into existing JUnit tests.

The tests can be started from two possible sources: Java classes containing the JUnit test cases or Unit Tests scripted directly in QF-Test. The parameters of the node vary with the use case.

The JUnit 5 framework is used to execute the tests. This enables executing JUnit 5 Tests using the JUnit Jupiter engine as well as executing JUnit 4 and JUnit 3 tests using the JUnit Vintage engine. With JUnit 5 you can use features like parameterized tests, nested tests and test with a different display name.

12.1 Java Classes as the Source for the Unit Test

It is possible to execute Unit Tests from Jar or Class files. It is also possible to execute Unit Tests that are available from the SUT's runtime. QF-Test executes the tests of the test classes specified in the respective attribute of the Unit test[(790)] node. In the report they will be displayed as test steps within a test case. The following example demonstrates the usage of a Unit test[(790)] node with Java test classes.

Figure 12.1: Unit Test node with Java classes

```
package de.qfs.test;
import org.junit.Assert;
import org.junit.Test;
public class StringTest {
    @Test
    public void testubstring() {
        String s = new String("Long text");
        s = s.substring(5, 9;
        assert("text".equals(s));
    }
    @Test
    public void testReplace() {
        String s = new String("example");
        s = s.replace('e', 'i');
        Assert.assertEquals("ixampli", s);
    }
}
```

Example 12.1: Java Unit test

The class de.qfs.test.StringTest must exist in the unittests.jar specified in the Classpath[(794)] attribute. The path is determined relative to the path of the directory of the current suite. In this example the jar file is in the same directory as the suite.

JUnit test classes are Java classes where the methods have the `@Test` annotation. The Unit test node executes all classes specified in the Test classes[(794)] table. Thus a Unit test node can execute several test classes.

12.2   Basics of the Test Scripts

The second option to execute Unit Tests is to script the Unit Test directly in the Unit test node. You can use any of the Script languages QF-Test offers. The most appropriate one is Groovy because it supports the Java annotations. The JUnit framework is used to execute the scripts.

12.2.1 Groovy Unit Tests

```
@BeforeClass
static void onbefore(){
    println("Set Up")
}
@Test(expected=IndexOutOfBoundsException.class)
void indexOutOfBoundsAccess() {
    def numbers = [1,2,3,4]
    numbers.get(4)
}
@Test
void noFailure() {
    assert true
}
```

Example 12.2: Unit Test Script with Groovy

In Groovy the required JUnit 4 classes are automatically imported at run-time. Just like in Java all tests with the `@Test` annotation will be executed. You can ignore expected exceptions using the `expected` parameter of the `@Test` annotation. The methods with the `@BeforeClass` annotation will be executed before the test methods will be run.

12.2.2 Jython Unit Tests

```
def setUp(self):
    print "Set Up"
def testMathCeil(self):
    import math
    self.assertEqual(2, math.ceil(1.01))
    self.assertEqual(1, math.ceil(0.5))
    self.assertEqual(0, math.ceil(-0.5))
    self.assertEqual(-1, math.ceil(-1.1))
def testMultiplication(self):
    self.assertAlmostEqual(0.3, 0.1 * 3)
```

Example 12.3: Unit Test script with Jython

Because Jython does not support Java annotations, the tests run as JUnit 3 tests. All methods beginning with the keyword `test` are considered to be a test and executed as QF-Test checks. The methods must have the `self` parameter because they are automatically enclosed in a class. The `setUp` method is executed at the beginning of each test.

12.2.3 JavaScript Unit test

```
setUp(){
    print("Set up");
}
tearDown() {
    print("Tear Down");
}
testUpperCase(){
    let s = "text";
    assertEquals("TEXT", s.toUpperCase());
}
testOk() {
    assertTrue(true);
}
```

Example 12.4: Unit Test Script with JavaScript

Also JavaScript does not support Java annotations, the tests are executed as JUnit-3 Tests (cf. section 12.2.2[191]). Just like in Jython all functions beginning with the keyword `tests` are executed as QF-Test checks.

12.3 Injections

It is possible to use the Unit Test node for the so called Live Tests. In this case the Unit Tests are executed in the running SUT. Using 'Injections' the Unit Tests inject QF-Test objects like WebDriver, components and variables into the Unit Tests or scripts directly.

12.3.1 Component-Injections

```java
import static org.junit.Assert.*;
import javax.swing.JComponent;
import org.junit.Test;
public class ComponentTest
{
    /** The component to test in this unit test */
    static JComponent component;
    /** Expected value */
    static String accessibleName;
    @Test
    public void accessibleNameIsCorrect()
    {
        /** component and accessible name are injected at run-time */
        final String currentName =
            component.getAccessibleContext().getAccessibleName();
        assertEquals(accessibleName,currentName);
    }
}
```

Example 12.5: Java Unit test

Figure 12.2: Example Unit Test node with Injections

The example shows the injection of two QF-Test objects: component and variable. The parameter "Field" corresponds to the name of the field `static JComponent component;` in the Java class. The java field must be `static`.

12.3.2 WebDriver-Injections

```
import static org.junit.Assert.*;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
public class WebdriverTest
{
    /** The driver of the window currently opened by QF-Test. */
    static WebDriver driver;
    @Test
    public void urlIsCorrectedLoaded()
    {
        // driver is injected at run-time
        final String currentUrl = driver.getCurrentUrl();
        assertEquals("http://www.example.com", currentUrl);
    }
}
```

Example 12.6: Java Unit Test with WebDriver Injections

Figure 12.3: Example Unit Test node with Injections

This example shows how to inject a WebDriver object into a Java class. When no value for the `WebDriver driver` is specified QF-Test determines the value via the given client.

12.4 Unit Tests in Report

The greatest benefit from using the Unit test node is that the results are displayed nicely formatted in the HTML report. All Unit Test classes executed via this node are considered QF-Test test cases. Unit test nodes should not be run separately. In order to see them correctly displayed in the HTML report, run them as part of a Test case. Each test method is handled like a QF-Test check, e.g. a failed check does not abort the tests execution.
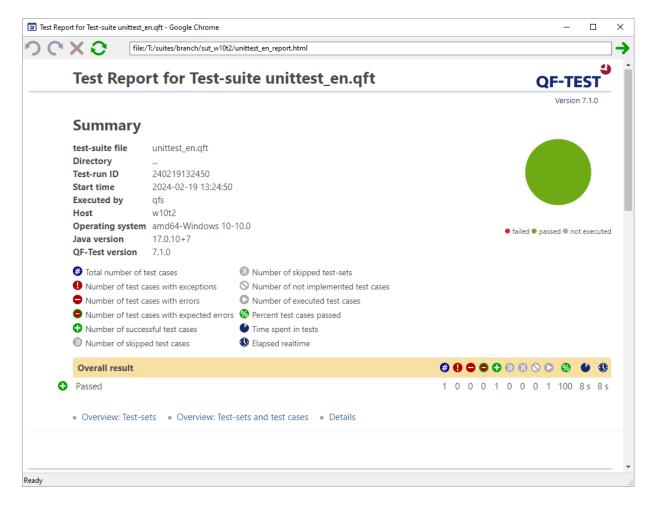
Figure 12.4: Unit Test Report

Chapter 13

Testing Java desktop applications

QF-Test's origin is testing of Java desktop applications and since 1999 a very profound support has been achieved for the basic Java UI toolkits as there are:

- Java Swing - the UI toolkit from Sun/Oracle

- SWT (Standard Widget Toolkit) - the tookit behind Eclipse, developed by IBM

- JavaFX - the intended successor of Java Swing from Oracle

There are also extensions/libraries available for this toolkits providing special components or framework functionalities:

- Rich Client Platform (RCP)

- Eclipse Plug-Ins

- Netbeans Platform

- JFace GUI toolkit (library based on SWT)

- JIDE Common Layer components

- ULC (UltraLightClient) and RIA (Rich Internet Application)

- Java WebStart

- ...

Desktop application based on those technologies can be tested with QF-Test in a convenient and efficient way, leading to robust and reliable test-cases with low maintenance efforts and a high benefit for the software quality assurance. All general techniques described in this manual can be applied for Java desktop testing.

There are also hybrid applications, like Java desktop applications with an embedded browser component or being displayed/rendered by a browser. Also those kind of systems can be perfectly tested with QF-Test. Please refer to Web testing[200] and Testing Java desktop applications in a browser with Webswing and JPro[249] respectively for further details.

Video   There are short introductory videos about

    Java Swing testing
https://www.qfs.de/en/yt/java-swing-testing.html

and

    JavaFX testing
https://www.qfs.de/en/yt/javafx-testing.html

available on our QF-Test YouTube Channel.

Chapter 14

Web testing

QF-Test allows intuitive testing of web pages in a browser from the point of view of the user. Just as with the other supported GUI technologies you can record actions and checks directly, and then rework, structure and replay them.

There is a short

▷ introductory video about web testing
https://www.qfs.de/en/yt/web-testing.html

available on our QF-Test YouTube Channel.

14.1 Supported browsers

QF-Test supports test automation for the following browsers:

- Google Chrome (also headless mode, cf. section 14.7[205])

- Mozilla Firefox (also headless mode, cf. section 14.7[205])

- Microsoft Edge (also headless mode, cf. section 14.7[205])

- Safari

- Opera

- JxBrowser embedded into Swing, JavaFX or SWT

- WebView embedded into JavaFX

- Internet Explorer or Webkit embedded into SWT

Please refer to section 1.1.2[3] for details on the supported browser versions.

14.2 General approach

In the tutorial, part II, you will find a step-by-step instruction for starting with test automation for web applications.

You can also take a look at the Video tutorials. If you just want to set up the startup sequence for the web application we recommend the video

▷ 'Quickstart Wizard'
https://www.qfs.de/en/yt/quickstart-wizard-web-42.html

on our QF-Test YouTube Channel.

The approach towards test automation and execution regarding web applications does not really differ from that for other GUI technologies, as described in the general part of the manual, starting from chapter 2[(13)]. However, you should pay special attention to component recognition, which highly depends on the precise implementation of the web application. In order to find out how well the direct recognition will work we recommend you do a test recording of different components in several dialogs of the web application and check the replay is correct. Please refer to section 14.4[(202)] for more information on component recognition and configuration.

14.3 Browser connection

First, you need to start and connect to the desired browser via QF-Test. As soon as the web application, defined by its URL, has been loaded you can start with recording and test automation.

QF-Test uses three different methods, so-called drivers, to get access to the browser and to set up the connection: the **QF-Driver**, **CDP-Driver** and the **WebDriver**.

Note For some browsers QF-Test supports more than one connection mode. QF-Test tries to set the best mode for accessing the browser automatically. However, it is possible to take control over the connection mode via the attribute Browser connection mode[(647)] of the Start web engine[(645)] node. For details please refer to section 49.3[(993)].

QF-Driver embeds the browser installed on the testing machine into a wrapper window, and the wrapper window into the locally installed web browser, thus gaining access to the automation interfaces of the browser and being able to listen to the events in the browser and on the other hand to inject events into the browser.

The embedding of the browser into a separate window unfortunately does not work with all browsers anymore, requiring an alternative mechanism to be implemented.

5.3+ The **CDP-Driver** mechanism brings into play debugging interface integrated in Chromium (and browsers based on it). QF-Test uses for it Chrome DevTools Protocol.

It is the same API that is used by browser development tools. The API provides close communication with a browser and efficient test execution. Unfortunately, an implementation of this interface exists so far not for all the browsers supported by QF-Test.

4.1+ The **WebDriver** mechanism uses the Selenium WebDriver as link between the browser and QF-Test, the WebDriver having become a W3C standard for controlling of web browsers (http://www.w3.org/TR/webdriver/).

Note For reasons inherent to WebDriver, unfortunately, the WebDriver connection mode is not yet on par with QF-Driver mode in terms of performance and feature completeness (see section 49.3.4[995]). For the time being we recommend to primarily use QF-Driver or CDP-Driver mode for recording.

Note QF-Test needs to deep inspect the browser content in order to enable the familiar testing features (e.g. event and check recording, feature based component recognition, web and custom resolvers) in WebDriver mode. With some browsers this might trigger in a warning related to mixed content display or an untrusted certificate. If this warning or error message only appears while running the web site in testing mode, you can safely ignore it.

14.4 Recognition of web components and toolkits

With web applications developers are quite free as to how to implement graphical objects with HTML, leading to a multitude of different implementations for functional GUI components like buttons, text fields, data tables etc. Some samples for an OK Button:

1. `<button id="ok1">OK</button>`

2. `<div class="toolkit-btn">OK</div>`

3. `<span class="btn" name="ok2">OK</span>`

4. `<div role="button">OK</div>`

By default, QF-Test records the GUI elements with the HTML tag as class plus basic features for recognition.

For the first sample QF-Test would record a component of the class `BUTTON`, the name `ok1` and the structure and geometry as resulting from the GUI. When replaying the tests, the component recognition should work all right.

Moreover, QF-Test checks whether the HTML implementation is a quasi-standard, and if so, maps the object to a generic QF-Test class. In the first sample this would be the case and the component would be recorded as `Button`.

One benefit of generic classes is that QF-Test additionally records class specific features for component recognition, e.g. the text of a button would be saved in the Feature attribute. Another advantage are the class specific checks, like a check for a whole row or column with tables. For more benefits of generic classes please read the introduction of Generic classes[(1186)]. For detailed information on class specific features please have a look at Generic classes[(1186)].

There are a variety of web component libraries on the market, such as Angular Material or Vaadin, which are very helpful when creating web pages. Each of these libraries has their own implementation of GUI objects.

The second sample could be from a framework using the css class `toolkit-btn` for buttons.

For some web frameworks the mapping of the GUI elements to generic classes has already been implemented with QF-Test. For such frameworks you can work with the stable component recognition you are used to with QF-Test. For more information please refer to Special support for various web frameworks[(988)]. By default, QF-Test should automatically recognize the web framework used to implement the application. Else you can choose the correct framework manually.

The third and the fourth sample do not follow any standard. For sample three the standard component recognition would probably be sufficient regarding stability because of the `name` attribute. The fourth sample would not have a good standard component recognition. However, it has an HTML attribute defining the functionality of the node. In both cases you could map the defining attribute to the generic class `Button`. For detailed instructions please refer to Improving component recognition with a `CustomWebResolver`[(945)].

Note   We recommend checking which category the GUI objects of the application to be tested belong to before starting test automation. In case standard recognition should not be sufficient we recommend to improve it by mapping the GUI objects to generic QF-Test classes as described in section 49.1[(945)].

For general information on QF-Test components refer to chapter 5[(42)].

Note   Working with several browser windows is explained in FAQ 25.

14.5 Cross browser tests

Cross browser tests are easy to implement. You can implement test cases working on one browser and then replay them on other browsers. You just need to implement a Data driver[(562)] defining adequately the variable `$(browser)`. If you want to try it out just open the web demo test suite and add a Data driver node in the test set ": (To open the demo test suite select the menu item |Help→Explore sample test suites...| and click

the 'open' link behind 'Web CarConfig Suite'.)

Figure 14.1: Cross-Browser Tests

Then the four test sets which are in the same test set as the data driver will be run once for each of the browsers.

14.6 Emulation of mobile browsers

4.2.1+

An important aspect of web page testing is the user experience on mobile devices like smartphones or tablets, since due to varying browser identifiers ("user agent") and device-specific screen sizes rendering of web pages differs between mobile and desktop browsers ("responsive design").

QF-Test supports such scenarios by emulating mobile browsers: A desktop browser (e.g. Google Chrome) is started in a special mode, where the page size and the browser identifier mimics those of browsers on mobile devices.

Google Chrome in particular is able to simulate specific characteristics of mobile browsers like adjusted pixel ratios and automatic scaling of non-responsive web pages.

To use mobile emulation in QF-Test, select the corresponding entry in the Quickstart wizard (see chapter 3[(28)]) and specify the required device parameters, together with the URL of the web page and additional test requirements. QF-Test ships with a great number of predefined definitions of well-known mobile devices, which can be freely adapted as needed.

For a demo please open the test suite 'carconfigWeb_advanced_en.qft' and run the test set 'Emulation of Mobile Devices'. An easy way to open it is via the menu Help→Explore sample test suites... and then selecting the 'open' link behind 'Web CarConfig Suite' (at the very bottom).

14.7   Web-Tests in headless mode

4.2+   Using the CDP-Driver and WebDriver connection mode it is possible to run Firefox, Chrome and Microsoft Edge in a so called "headless" mode. In this mode, the browser is started in the background, without any visible window on the screen. All interactions with the web page are executed inside this "invisible" window.

A use case for headless browsers might be load testing (section 31.5[(386)]). Or you could use it for tests you want to run in the background on the same machine as you are developing tests on.

To execute an existing web test using the "headless"-mode, simply change the type of browser in the Start web engine[(645)] step from `chrome` to `headless-chrome`, from `firefox` to `headless-firefox`, or from `edge` to `headless-edge`.

14.8   Integrating existing Selenium web tests

You can run existing Selenium Scripts with QF-Test when using the WebDriver mode.

One way is the direct use of the WebDriver Java APIs in the SUT script[(631)] node (cf. section 52.12[(1132)]).

Another option is embedding Selenium Scripts as Unit tests, as described in chapter 12[(188)]. As a nice side effect, you get an integrated report including the executed Unit tests.

For a demo please open the test suite 'carconfigWeb_advanced_en.qft' and run the test set 'Integrating Selenium tests'. An easy way to open it is via the menu Help→Explore sample test suites... and then selecting the 'open' link behind 'Web

CarConfig Suite' (at the very bottom).

14.9   Selecting the browser installation

If you use the CDP-Driver or WebDriver mode, you are no longer limited to Firefox if you want to specify a browser installation folder using the attribute <u>Directory of browser installation</u>[647] of the <u>Start web engine</u>[645] node. If no matching browser can be localized in the specified directory, an exception will be thrown. If no directory is specified, QF-Test will try to start a default browser of the given browser type.

Chapter 15

Testing native Windows applications

15.1 Getting started

Video about testing of native Windows desktop applications:

▶ 'QF-Test Version 5.0 - Testing Windows applications'
https://www.qfs.de/en/yt/version-50-testing-windows-applications-50.html

This chapter covers automation and testing of Windows desktop applications, in particular

- Classical Win32 applications,

- .NET applications based on Windows Presentation Foundation (WPF) or Windows Forms,

- Universal Windows Platform (UWP) applications using XAML controls.

All these kinds of applications support Microsoft UI Automation or Microsoft Active Accessibility (MSAA) for software test automation, joined together in the Windows Automation API, please see section 15.2⁽²⁰⁸⁾ for background information.

For test execution a connection between the process of the application being tested and QF-Test is needed. In order to create a Setup that is able to create such a connection, the Setup sequence creation⁽²⁹⁾, launchable via the Extras menu can be used. Choose 'A native Windows application' as application type then. Further information on how to use the Quickstart-Manager can be found in section 15.3⁽²⁰⁹⁾ and section 3.1⁽²⁹⁾.

If your application is already up and running, you can use the Attach to windows application node. You just need to specify the title of its (main) window. Therefore a regular expression can also be defined. In this case the checkbox for As regexp needs to be activated as well. For example .*- Editor for the Windows Notepad application would

be sufficient. However, it should be ensured that the given regular expression, does not match any another window title. Otherwise you can use the Start windows application node to give the path to your Windows executable (.exe) so that QF-Test can start the program, please see Launching/Attaching to an application[209] for details.

Once the application is connected to QF-Test (as GUI engine[633] win), capture and replay can be performed as described in chapter 4[35]. However, due to the nature of UI Automation, you should observe the recording rules listed in section 15.4[210].

The QF-Test installation provides the following example files:

- `qftest-7.1.2/demo/carconfigForms/winDemoForms_en.qft`

- `qftest-7.1.2/demo/carconfigWpf/winDemoWPF_en.qft`

- `qftest-7.1.2/demo/windows/Win10Calculator_en.qft`

Also have a look at the (Current) Limitations[215], most of which are expected to be fixed or improved in future releases of QF-Test.

15.2   Technical background

A common framework for all Windows-based applications is the Windows Automation API consisting of Microsoft's Active Accessibility and its successor, Microsoft UI Automation. These frameworks provide the core of the `win` engine, whereby QF-Test is now able to control virtually any Windows applications.

A Windows application has to expose so-called `Providers` in order to follow the rules of UI Automation. This is done automatically when a framework like WPF is used to develop a program. This is also done for Win32 applications via proxy providers. That means, how good an application can be controlled and tested depends on the quality of the respective providers, i.e. usually on the framework used in application development. Like this, applications created via the WPF framework tend to be easily testable, as the WPF framework was introduced along with the UI Automation framework. If the framework does not provide an integration for the UI Automation the situation is different. For example if you try to test a Java Swing application. However, QF-Test already provides another very good connection mode for Java applications.

A test application that wants to control a program via UI Automation can get hold of so called `Automation Elements` which represent the actual UI elements in the SUT (System Under Test). Though every automation element has a control type (like `Button`, `MenuItem`, etc.), its actual functionality - for example, setting a value in a text field - depends on `Control Patterns` implemented by the respective providers.

To deal with the UI Automation framework, QF-Test starts a special Java program which serves as UI Automation client application. That program can access all UI Automation elements in a given process and handle them according to the rules of QF-Test (e.g. create a snapshot of an element as Component[823]).

15.3 Launching/Attaching to an application

Testing a native Windows application does not require you to launch that application from QF-Test. You can also connect to a running process and that way even control parts of the operating system, for example the Windows Taskbar.

In order to connect to a process you can specify a window title (optionally as a regular expression) or the respective process ID or the window's UI Automation class name. Strictly speaking, that window must not be a `Window` but could also be a `Pane` or a `Menu` in terms of UI Automation control types. Whatever feature is used for attaching, QF-Test will eventually determine the respective process ID and treat exactly that process as the actual client application (SUT).

To connect just define the attribute Window title[661] in the Attach to windows application[660] node and this can be

- a regular expression for the title

- -pid <process ID>

- -class <class name>

For example, by specifying `.*- Editor` you can attach to a running Windows Notepad application, while `-class Shell_TrayWnd` will address the Windows Taskbar.

In order to find out the titles, process IDs and class names of running programs, you can run the procedure `qfs.autowin.logUIAToplevels` in `qfs.qft`, cf. The standard library[158].

Besides attaching to a running process, it is also possible to launch a program from the Start windows application[657] node. To this end, specify the path to the respective executable in the Windows application[658] attribute.

In some cases, it can also be useful to define both the Windows application and the Window title attribute. QF-Test will then first try to attach. If that fails, the given program will be started and connected via its process ID. If that fails too - the process may launch a child process and terminate itself or may not display a (graphical) user interface - another attempt to attach is made.

When you terminate a `win` client in QF-Test (either via the Stop client[675] node or from the Clients menu), the respective UI Automation client process will be stopped along

with its sub-processes. That is, your actual SUT will terminate as well, if you started it from QF-Test. On the other hand, the SUT will not be stopped when it was running before you attached to it.

When you close the SUT, the UI Automation client will terminate as well.

To attach to an elevated processes (presenting the UAC prompt), you have to launch QF-Test as administrator.

15.4 Recording

After connecting QF-Test with the SUT, you can record events (section 4.1[35]), checks (section 4.3[38]) and components (section 4.4[39]).

However, as the communication between the SUT and the QF-Test UI Automation client is handled by Windows (the UI Automation core), accessing elements is not quite as fast as you may know it from the QF-Test Java automation. Furthermore, in contrast to Java and Web testing (QF-Driver), events are processed asynchronously, i.e. you cannot expect that an application's dispatch thread is blocked while QF-Test is handling an event.

That makes recording more difficult, because a target element might be destroyed by the action to be recorded, for example when selecting an entry from a ComboBox or clicking on a button that closes its parent window.

So you'd best get into the habit of following a few recording rules:

- Activate the recording mode and move the mouse over the element for which you want to record an event.

- As QF-Test may take a little time to retrieve information about the element below the mouse cursor, a red pane is displayed until it is done; the little 'QF-Test Element Information' window will then show which automation element was found.

- Now perform the mouse click to be recorded.

- When a mouse click will close a dialog or window (might also be a popup displaying a list), make sure to perform the click slowly, i.e. do not release the mouse button immediately after pressing it so that QF-Test has the opportunity to gather information before the window will disappear (when the mouse release is done).

- When recording checks or components, the respective frame around the element is drawn almost immediately when the mouse is hovering over an automation element. Before recording a check, you should wait until the frame disappears.

Sometimes check recording (and transforming the node afterwards) may work better than event recording, for example when a click on a button (like OK, Cancel) closes the respective dialog or when a mouse down event recreates elements (for example the accessory table in the CarConfiguratorNet WPF demo application). In check recording mode QF-Test covers the SUT with an (almost) invisible window to prevent mouse clicks from triggering an action in the client application.

15.5 Components

In QF-Test an automation element is recorded (or can be inserted manually, of course) as Window[812] or Component[823] and stored within the Windows and components[835] node. The QF-Test (generic) Class name often corresponds to the type of the UI Automation element, for example `Button`. To be able to differentiate between the UI Automation type and the generic class name, QF-Test adds a prefix `Uia.` to the type. Similarly, the UI Automation framework specifier is used as prefix for the automation element's class. So you may for example see a `classname:   WPF.DataGrid` in the Extra features of a Table component recorded in a WPF application.

QF-Test does not strictly follow the hierarchy of the UI Automation elements. That is often the case with dialogs (like Notepad's Font dialog) which are usually listed below the main application window in the UI Automation tree. From the Win32 perspective as well as what QF-Test users would expect, such dialogs are also top-levels and thus listed as a sibling of the main window below Windows and components. On the other hand, a context menu can be a top-level in the UI Automation tree, but may be a window's child in QF-Test.

15.6 Playback and Patterns

UI Automation supports various "soft" actions which do not rely on mouse events. For example, to trigger a button's action you can play back

```
+Select:   invoke [myButtonID]
```

The effect should be the same as with

```
+Mouse click [myButtonID]
```

but no mouse is involved when using the Selection node. Instead, the UI Automation core will trigger the execution of a provider's `Invoke()` method in the SUT.

The Selection node does support the following actions in its Detail attribute:

Detail	Description	Pattern
`invoke`	Usually equivalent with a mouse click.	InvokePattern
`expand, collapse`	Should expand/collapse a ComboBox, MenuItem or TreeItem.	ExpandCollapsePattern
`select[:0|-1|1]`	Should select an item in a list. If -1 or 1 is specified, a multi-selection is extended or reduced by one.	SelectionItemPattern
`toggle[:on|off]`	Change the state of a CheckBox element.	TogglePattern
`scroll:horiz%,vert%`	Values between 0 and 100 are possible, defining the position of the scroll location in percent; specify -1 when you do not want to change a position (horizontally or vertically).	ScrollPattern

Table 15.1: Supported details for a Selection

The actions actually supported depend on an automation element's patterns. They are recorded among the Extra features of a component or can be determined in an SUT script like `print rc.getComponent(id).getPatterns()`.

What exactly a pattern means can vary from application to application. If, for example, both `SelectionItem` and `Invoke` patterns are supported, `Invoke` might be preferable because

```
+Select [list@item]
```

may only highlight the element but not trigger the respective action (e.g. Notepad Fonts).

The formal support of a pattern does unfortunately not mean that applying it has any effect, for example scrolling an (invisible) entry in the list of Windows Calculator's modes into view (`ScrollItem` pattern). To get around the problem, you can simply play back `select` here, whether or not the entry is currently visible.

As already mentioned above, because "soft" playback may simply not work (due to the provider implementation).

Regarding Key events, a text can only be set directly by a Text input node if the `Value` pattern is supported. Otherwise single key events have to be played back.

15.7 Scripting

Internally, the `win` engine represents automation elements by a class `WinControl`. To obtain an element in a Groovy SUT script[(631)] node, run

```
def ctrl = rc.getComponent("myComponentID")
println ctrl
```

Example 15.1: Retrieving a `WinControl` in a Groovy SUT script

with the respective QF-Test component ID. The methods of the `WinControl` class are described in section 52.13.1[(1135)]:

- `getUiaType()`, `getUiaClassName()`, `getFramework()`, `getUiaName()`, `getUiaId()`, `getUiaDescription()`, `getUiaHelp()`, `getHwnd()`, `getLocation()`, `getSize()`, `getLocationOnScreen()`, `getPatterns()`, `hasPattern()` to retrieve UI Automation properties of the element

- `getChildren()`, `getParent()`, `getChildrenOfType()`, `getAncestorOfType()`, `getElementsByClassName()` to traverse the element hierarchy

- `getUiaControl()` to retrieve an `AutomationBase` control, compatible with the `uiauto` script module (chapter 50[(1000)]).

15.8 Options

The behavior of the `win` engine can be influenced via a set of QF-Test options and additionally by defining preferences which affect the native part of the UI Automation Client. Those options and preferences can be set in an SUT script node via

```
rc.setOption(<name>, <value>)
```

or

```
rc.engine.preferences().setPref(<name>, <value>)
```

respectively. To reset an option, use

```
rc.unsetOption(<name>)
```

15.8.1 Windows scaling

As the display resolution increased over the years, Windows allows to define a scale factor so that application windows and controls are enlarged and text becomes more readable. Usually, UWP, WPF and Windows Forms application do scale automatically, but especially Win32 programs may retain the size of its controls or scale differently.

QF-Test works with physical display coordinates by default so that geometry values will change when an application is scaled. Say the scale factor is set to 125%, a button which originally (100%) resides at location (24, 40) with size (100, 20) will be moved to location (30, 50) within its container and grow to an area of 125 times 25 pixels. The consequences are

- different geometry when recording the component anew

- geometry mismatch when QF-Test tries to identify an element which was recorded at 100% now in the scaled environment

- a (hard) mouse click onto a given region within an element may fail because the scaled region is farther away from the element's top-left corner.

To make QF-Test work with logical coordinates (as seen with a scale factor of 1), you can set `Options.OPT_WIN_USE_SCALING` to `true`. QF-Test then uses the scale factor of the primary connected monitor to adapt the geometry of components and mouse event coordinates. Note that rounding errors may occur when calculating new integer coordinates so that the mouse may not hit a given point exactly.

15.8.2 Visibility

You sometimes may want to play back an event on an element that is actually not visible (it may not be scrolled into view). To perform an `invoke` event then, you may need to get rid of the visibility test which is usually part of the component recognition.

This can be achieved by setting `Options.OPT_WIN_TEST_VISIBILITY` to `false`. After playing back the event, you should reset the option to re-enable the visibility test.

15.8.3 Attaching to a window with a given class

If you attach to an application via `-class <class name>`, QF-Test by default ignores all toplevels in the application which do not have the given class name. That way, you can for example deal solely with the Windows Taskbar (and set apart the desktop and its icons which run in the same process).

To be able to access all toplevels in the process, you can set the preference ```"windriver.restrict.tops.to.class"``` to ```"false"```.

15.8.4 Child count limitation

Unfortunately, big hierarchies of automation elements may cause performance problems. To avoid that recording and playback slow down drastically, QF-Test limits the number of children when retrieving automation elements from the client.

The default value is 100. It can be changed by setting ```Options.OPT_WIN_MAX_CHILDREN``` to another value.

15.9 (Current) Limitations

There are a number of limitations in the current implementation status of the Windows testing functionality. We will try to further improve things within the next versions, but possibly not all of the following points will be resolved soon.

As the support for UI Automation depends on the framework used for application development, the recording in QF-Test may not always be consistent. For example, a Wait for component to appear node may or may not be recorded when opening a dialog.

Dealing with applications consisting of several processes requires several ```win``` clients and can be tricky.

Further limitations / not yet implemented features (January 2020) are among other things

- Supported check types are more less complete There may be some special check types missing. This is supposed to be fixed in a future release of QF-Test.

- Elements in the title bar of a Windows app cannot be accessed (easily), because they live in a different process. This might be improved in a future release of QF-Test.

- Redirection from a Button's Text element to the Button element is done when recording a mouse click, but may be missing elsewhere. This is supposed to be improved in a future release of QF-Test.

15.10 Links

The Windows Automation API is described here: https://docs.microsoft.com/en-US/windows/desktop/WinAuto/windows-automation-api-portal.

More about Mark Humphrey's ui-automation Java library can be found on https://github.com/mmarquee.

Chapter 16

Testing Android applications

This chapter covers test automation of Android native applications.

There is a short

> introductory video about Android testing
> https://www.qfs.de/en/yt/android-testing.html

available on our QF-Test YouTube Channel.

In June 2022, a special webinar took place about Android Testing with QF-Test. After a bit of theory the detailed way of working with emulator and real device is demonstrated.

Here you can find the

> special webinar video recording
> https://www.qfs.de/en/yt/android-special-webinar.html

available on our QF-Test YouTube Channel.

In case you want to test mobile Web applications, we recommend to check out the options of the mobile emulation mode of the chrome desktop browser as describe in section 14.6[204]. Even though is possible to use a web browser on an Android device (given that it supports the accessibility interface), the mobile emulation mode offers better automation features and less overhead for mobile web testing.

16.1 Preconditions and known restrictions

16.1.1 Preconditions

In order to perform Android tests with QF-Test, the following preconditions need to be fulfilled for the machine QF-Test is running on:

- In case an Android emulator shall be used, a sufficiently powerfull machine is required (not an old scrap mill :-). It might be even necessary to enable hardware acceleration (typical via the bios) if the emulator works too slow. Further details can be found at https://developer.android.com/studio/run/emulator-acceleration.

- Android SDK command line tools need to be available on your machine, even better an installation of Android Studio as described in section 16.3[(219)].

- Either a real Android mobile device needs to be connected via cable and USB debugging need to be enabled for this device (see section 16.4[(225)]),

- Or an Android emulator needs to be installed, running an adequate Android Virtual Device (AVD) (see section 16.3.2[(219)]).

- The Android API Level of the real or virtual Android device needs to be greater or equal 24, which means Android Version 7 Nougat or later (see Andriod version history at Wikipedia).

16.1.2 Known restrictions

Note

There are some restrictions within this version listed below:

- Only one Android client can be connected and controlled in parallel so far. It is planned to enhance this to multiple clients as it is supported with the other UI technologies.

16.2 Emulator or real device

At the beginning of Android testing, you need to decide how to start: with a virtual or real device.

A real Android device can be used for testing with QF-Test. It needs to have the USB debugging developer option activated and requires a connection via cable to the machinge QF-Tests runs on. By help of respective setup sequence, QF-Test connects to the real device and is able to control it. Now actions and tests can be recored and executed.

An Android virtual device (AVD) is the emulation of a real device. It runs by using an emulator software on a computer that replicates the hardware and behavior of the real device. An Android emulator is therefore a software to execute and test Android apps on a computer. The emulator is able to load different Android virtual devices with specific Android versions or products of a certain vendor.

When using an emulator, it is typically started by QF-Test at the beginning of the test, then QF-Test loads and connect to the defined virutal Android device. Finally the app to be tested is opened. Actions and tests can now be recorded and executed.

An advantage of the emulator is, that there is no dependency of external devices and testing of different virtual devices is possible. Though, it needs a bit more of initial setup and may cause more load on your maschine.

A real device allows a quicker start, is less flexible and needs USB debugging enabled.

16.3 Installing Android Studio, emulator and virtual devices (AVD)

The easiest way to install the Android SDK command line tools and emulator and configure a virtual device is via the Android Studio. There is also the option to just install the Android SDK or even just the SDK command line tools but there are some pitfalls, which is why we decided to focus on Android Studio.

If you have not already installed Android Studio, the following steps are necessary:

16.3.1 Android Studio installation

- Download Android Studio from https://developer.android.com/studio. The installation will need about 3 GB on your hard disk.

- Just install it in the standard way with default settings and allow it to get started afterwards.

- In the setup wizard you can go with the proposed settings.

- During installation you will be asked for a change in command line settings which you may want to accept.

- Installation is finished.

16.3.2 Android Studio virtual device configuration

- As we just want to use the Android Studio to configure our virtual device, we choose Virtual Device Manager from the menu (sometimes indicated by the vertical dots top right). Otherwise, you can find the Device Manager in the Tools menu.

Figure 16.1: Android studio start screen

- Here we can select Create device....

Figure 16.2: Android studio virtual device creation screen

- Select an appropriate virtual device. It is recommended to go for smaller screen sizes to both spare memory and allow the virtual device to fully fit on your screen.

Figure 16.3: Android studio screen to chose a device definition

- Now select the System Image representing the Android version. Press the respective "Download" link to start the component installer. Then you can also proceed with "Next".

Figure 16.4: Android studio screen to download and select the system image

- Finally press "Finish" on the last configuration dialog.

Figure 16.5: Android studio screen to finish the AVD configuration procedure

- Now you have a first configured virtual device ready for QF-Test to use.

Figure 16.6: Android studio screen showing available AVDs

16.4 Connecting to a real Android device

To use a real Android device for testing it needs to have USB debugging enabled and connected to the machine via USB cable.

Also an Android SDK is required on your machine. Even though it may be sufficient to just install a dedicated package for the Android SDK command line tools we strongly recommend install the full Android Studio as described in section 16.3[219]

Enable USB debugging

Please activate USB debugging for your real device. Typically following steps are necessary:

1. Navigate to "Setings" -> "About 'Device'".

2. Click seven times at Build-Number, then "Settings" -> "Developer options" will get visible

3. There activate the option "USB debugging"

The reference documentation how to activate USB debugging can be found at https://developer.android.com/studio/debug/dev-options.

Connect to PC via USB cable

 • After connecting the Android device to the PC, you may be asked on the device whether to allow USB debugging from this PC and whether to allow this permanently. You need to confirm this.

16.5   Create a QF-Test setup sequence for Android testing

 • As always when creating a setup sequence, open the Quickstart Wizard via the [Extras] menu or the 🪄 toolbar button.

 • Select "An Android application".

Figure 16.7: Quickstart wizard screen to select the application type

16.5.1 Usage of an Android emulator

- Select first option "Launch emulator and connect to an Android virtual device".

Figure 16.8: Quickstart wizard screen to select the emulate as test device

- Select the virtial device from the drop-down list. Press "Refresh" in case no AVD is visible. If it is still not visible also try to restart QF-Test in order to make the virtual device visible. Then press "Next".

Figure 16.9: Quickstart wizard screen to select the AVD

- As the next step you may want to specify the Android .apk file you want to test. If you want to test an App already installed, leave it empty.
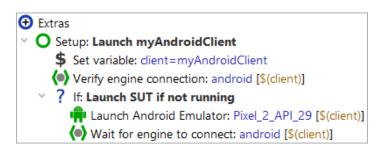
Figure 16.10: Quickstart wizard screen to select an APK
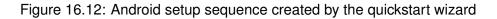
- In the next step you can specify a client name and press "Next" or "Finish" to finalize the Wizard.

Figure 16.11: Quickstart wizard screen to specify the client name

- As a result, a Setup sequence will be created in the "Extras" node of your test suite. It should be pretty much self explanatory and also contains the hint, that in this early access phase the qfsandroid.qft suite is needed for this setup sequence to run successfully. A resepective include has been added automatically.

Figure 16.12: Android setup sequence created by the quickstart wizard

- When executing the Setup sequence, the emulator window is supposed to appear and in case you have provided an .apk file to be started, the same should be visible there. Also the Record button ● is supposed to become active in order to indicate an established connection.

Figure 16.13: Android emulator window

16.5.2 Usage of a real Android device

- Select "Connect to a real Android device or running Android virtual device".

Figure 16.14: Quickstart wizard screen to select the application type

- From the Combo box select the shown entry, which is the id of your connected device. Press "Refresh" in case no device is visible. If it is still not visible also try to restart QF-Test in order to make the device visible. Then press "Next".

Figure 16.15: Quickstart wizard screen to select the real device

- As the next step you may want to specify an Android .apk file you want to test. If you want to test an App already installed, leave it empty.

Figure 16.16: Quickstart wizard screen to select a .apk file

- In the next step you can specify a client name and press "Next" or "Finish" to finalize the Wizard.

Figure 16.17: Quickstart wizard screen to specify the client name

- As a result, a Setup sequence will be created in the "Extras" node of your test suite. It should be pretty much self explanatory and also contains the hint, that in this early access phase the qfsandroid.qft suite is needed for this setup sequence to run successfully. A resepective include has been added automatically.

Figure 16.18: Android setup sequence created by the quickstart wizard

- When executing this Setup sequence the Record button ● is supposed to become active in order to indicate an established connection. Also in case you have provided an .apk file to be started, the same should get visible on you real device.

16.6   Record actions and checks for Android

- Please press on the Record Button to see what is going to happen for Android testing in QF-Test.

- A special recording windows will open showing the content of either the emulator or the real device. This special window is necessary, as it is currently not possible to directly capture events from the emulator or real device. So you need to capture actions and checks via this special window.

- It offers buttons for resizing the content area. Please also note that the content area just shows an image of the device screen. It needs to be updated manually by using the ⟳ refresh button. There is also an auto refresh toggle button ⟳ᴬ to perform this automatically.

Figure 16.19: QF-Test Android recording window

- Now you can try and record and also replay actions or checks. It may feel a bit clumsy in the beginning, but you will get used to the recording window soon.

- Despite these Android recording specifics, QF-Test should work and behave as with any other GUI Technology, except for the known restrictions described in

section 16.1.2[218].

- Goodies:

  The recording window also has some goodies to mention. In the bottom line left it shows the mouse coordinates, which may become handy if you need to work with absolute mouse clicks. On the right, it indicates the type of the last highlighted component.

  There is a ⬚ toolbar button on the recording window to open a UI inspector window, see UI Inspector[97], displaying all visible components including the size and coordinates. This is more to help tracking down issues with component recording and recognision issues, but may become handy here and there.

16.7 Android utility procedures

There are a number of Android utility procedures available in the standard library[158] and are located in respecitive "android" package.

Some are similar to those avaialbe for other UI technologies but some are very specific for mobile testing e.g. for performing guestures or swipe actions, scrolling and setting of a certain component status.
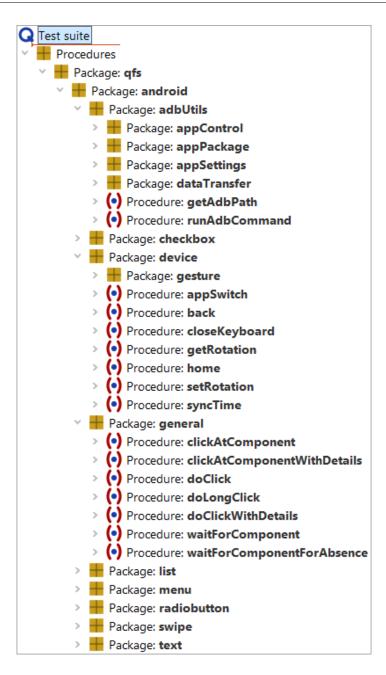
Figure 16.20: Android utility procedures

Chapter 17

Testing PDF documents

From version 4.2 onwards QF-Test offers the possibility to test PDF documents similarly to GUIs, i.e. QF-Test analyzes the structure of a PDF document and recognizes single components, which can be tested individually.
Using <u>Capture and replay</u>[35] QF-Test can directly record and replay <u>Events</u>[241] as well as <u>Checks</u>[241].

Video Video:

> ▷ 'Testing PDF Documents with QF-Test'
> https://www.qfs.de/en/yt/testing-pdf-documents-42.html

17.1   PDF Client

QF-Test loads the PDF document to be tested into a viewer, which QF-Test starts as a client process.

17.1.1   PDF Client start

The Quickstart Wizard allows to create a setup sequence to start the PDF client. Please choose "PDF document" as Type of Application. (c.f. <u>chapter 3</u>[28]). This allows to start the viewer. The node <u>Start PDF client</u>[654] is used as start node.

17.1.2   PDF Client window

The left side of the window of the PDF Client displays a column with an overview of all pages of the PDF document.

The right part of the window shows the currently selected page.
The following screenshot shows the PDF Client with a demo PDF document.
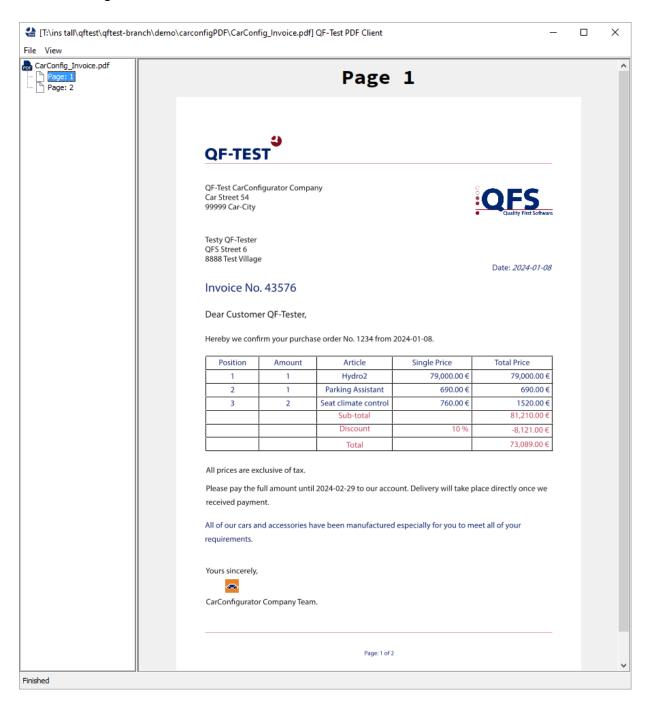
Figure 17.1: PDF Client main window with PDF document

17.2 PDF events

To change the opened document during test execution or the shown page you may use a Selection[(697)] step. These actions can be recorded directly in the recording mode. The following events can be replayed:

17.2.1 Open a PDF document

You can load another PDF document during the test execution. To do so you have to set the Detail[(699)] attribute of the Selection[(697)] step to `open:`.

Now you can set the path to the PDF document. If a relative path is specified it is resolved relatively to the directory containing the current test suite.

```
open:C:\Users\qfs\meinPDFDokument.pdf
```

Example 17.1: Loading a PDF document

If the document cannot be found or opened a `TestException`[(850)] will be thrown.

17.2.2 Switch page

To switch to a specific page the Detail[(699)] attribute of the Selection[(697)] step can be set to `goto:`.

Just like the Page of PDF Document[(656)] attribute, the page can either be set as an integer to set the page number or a string in quotation marks for the page name.

```
goto:3 or goto:"Introduction"
```

Example 17.2: Open a specific page

If the desired page cannot be determined, a `PageNotFoundException` will be thrown.

17.3 Checks for PDF components

The following checks exist for PDF components (c.f. section 17.4[(247)]). These checks can be directly recorded via the Check recording mode.

17.3.1 Check text

For a description of the CHECK TEXT node please refer to <u>Check text</u>[(708)]. There are two check types available for PDF text components: 'default' and 'Text positioned'.

PDF documents do not necessarily contain line breaks, and spaces. The spaces between words and rows result from the coordinates of the single letters. The check type 'default' checks the text as it is represented in the PDF document - without line breaks and spaces when the text object does not contain any. From the coordinates of the single letters QF-Test calculates where there should be line breaks and spaces. The check type 'Text positioned' checks this processed text.

Text

QF-Test CarConfigurator CompanyCar Street 5499999 Car-City

Figure 17.2: Check text 'default'

Text

QF-Test CarConfigurator Company↵
Car Street 54↵
99999 Car-City

Figure 17.3: Check text 'Text positioned'

4.4+ Additionally the whole text of the current page can be checked with the check types "Text (whole page)", "Text positioned (whole page)", "Text as items (whole page)" and "Text positioned as items (whole page)" available at the Main stage.
These check types record all Text components of the current page sorted by their Y/X position. The check types differ between recording the Text components positioned/processed (see above) or not and whether they are recorded as <u>Check items</u>[(719)] or as combined <u>Check text</u>[(708)].

Figure 17.4: Check Items 'Text as items (whole page)'

Figure 17.5: Check Items 'Text positioned as items (whole page)'

Figure 17.6: Check text 'Text (whole page)'

Figure 17.7: Check text 'Text positioned (whole page)'

17.3.2 Check image

For a description of the Check image node please refer to Check image[(729)]. The check type 'default' is provided for all PDF object types.

The check type 'default' checks the object as it is displayed on the PDF page, with scaling and with overlapping objects or parts of objects.

Figure 17.8: Check Image 'default' recording of a Text object

Figure 17.9: Check Image 'default' recording of an Image object

In a PDF document there may also be real embedded images. The images can be scaled for display on the PDF page. Moreover, other objects may overlap the image on the displayed page. For these Images the following check types are also available:

The check type 'unscaled' checks the original unscaled image embedded into the file.

Figure 17.10: Check Image 'unscaled' recording of an Image object

The check type 'scaled' checks the image displayed on the PDF page without overlapping objects, however, taking into account scaling. This allows to check partly invisible images.

Figure 17.11: Check Image 'scaled' recording of an Image object

17.3.3 'Check Font'

The Check text node with the check type 'text_font' allows to check the font of a text object.

The letters within one PDF text object may have various fonts. 'Check Font' checks the font which is used predominantly.

17.3.4 'Check Font size'

The Check text node with the check type 'text_fontsize' allows to check the fontsize of a text object.

The letters within one PDF text object may have various font sizes. 'Check Font size' checks the font size which is used predominantly.

17.4 PDF component types

QF-Test recognizes the following object types:

PDF object type	QF-Test component type	Comment
Text	Text or Label	Collection of letters, which have a font and a font size.
Image	Graphics	Collection of pixels. May also have the form of a letter.
Shader and Vectors	Graphics	Collection of vectors, which either represent geometrical figures or maybe also letters.
Main stage	MainPanel	The basic page for all objects.

Table 17.1: Supported PDF objects

QF-Test highlights all recognized PDF objects with a colored border line if this feature is enabled via the menu View -> Show components or the keyboard shortcut CTRL-T. This feature must be disabled during capture and replay, otherwise image checks will show the colored border lines.

The following color code applies to the object types:

Color	PDF object type
Red	Text
Blue	Image
Green	Shader and Vectors
Cyan	Main stage

Table 17.2: Color code for PDF objects

17.5   PDF component recognition

QF-Test represents PDF objects as Swing components, which can be accessed via the Swing API by SUI scripts, for example (c.f. chapter 11[161]).

The basic data QF-Test needs to identify the PDF object on the page are the same as with all QF-Test components: class, geometry and structure information (index). For text components QF-Test also saves the predominantly used font and predominantly used font size in the Extra features table. For image objects QF-Test also records the image hash and saves it in the Extra features table, for shader objects the shader type.

Moreover, QF-Test checks for every text object whether according to its features it might be a label. If so, the text object is given the class 'Label'. Via the standard algorithm for 'qfs:label' in the Extra features table QF-Test will assign the label component to other components where appropriate.

As the standard algorithm for the recognition of labels is based on assumptions and probabilities it may happen that labels are not recognized or the falsely identified. In this case you may want to use resolvers (section 52.1[1016]) to improve recognition. Resolvers can also be used to improve the assignment of label components to the respective field.

Chapter 18

Testing Java desktop applications in a browser with Webswing and JPro

Webswing and JPro are two highly interesting solutions that bring Swing and JavaFX desktop applications into a browser. The underlying technologies, concepts and goals differ significantly, but the challenge for QF-Test mainly boils down to the same thing: There are two SUT clients that need to be tested together in a coordinated way.

Migrating existing applications is one of the most common scenarios, so the ability to reuse existing QF-Test tests for the Swing or Java desktop application is crucial. This is one reason why testing through the browser alone is not sufficient. The other reason is that in the browser QF-Test only sees either a `CANVAS` node with colored pixels (Webswing) or a hierarchy of very similar `DIV` nodes (JPro). Though the latter is at least moderately useful for testing and may become interesting for special cases like load testing, it is still very limited compared to the deep access QF-Test has to Java applications.

Enter "JiB" - QF-Test's solution for "Java in Browsers".

In addition to QF-Test engine licenses for Swing and/or JavaFX, JiB support requires QF-Test licenses for the web engine.

A demo test suite for Webswing is provided for a better understanding of the concepts described in the section below. You can open it via the menu $\boxed{\text{Help}\rightarrow\text{Explore example test suites...}}$, entry "Webswing SwingSet Suite".

18.1 Technical concepts of JiB for Webswing and JPro

With the JiB concept QF-Test treats the Swing or JavaFX application as the primary SUT. Nearly all interaction is triggered through the respective Swing or JavaFX SUT

engine. QF-Test also opens a browser window and uses its web engine to interact with this frontend through which the application is displayed and through which the user interacts with it.

There are two modes of interaction between QF-Test and the application:

Java-mode

QF-Test can keep the event handling entirely within the Swing or JavaFX application. In that mode the browser serves only as a trigger to launch the application, as a reference for the user and for handling special cases where the workflow in the application had to be adapted to use web interfaces, most notably for file upload and download.

This mode is very similar to testing a plain Swing or JavaFX application, event simulation happens in an identical way. Images for image checks are taken via Swing or JavaFX off-screen-rendering to a memory buffer, also identical to the desktop version.

Web-mode

What the above doesn't cover is the verification that the Webswing or JPro integration actually works end-to-end as expected, i.e. that the user really sees the interface as expected and that the user can interact with the application via mouse and keyboard through the browser. Though it is debatable to which degree underlying technologies should simply be trusted or covered by one's own tests, the ability to perform real end-to-end tests via the browser is a very important aspect in this scenario.

To that end QF-Test can redirect the actual replay of mouse and key events to the browser via a number of option settings. Tests are still written and executed against the Java application, component recognition works unchanged and QF-Test performs all the necessary synchronization and setup like scrolling the target component visible or implicitly opening tree nodes. At the final step the Swing or JavaFX engine doesn't replay the mouse or key event itself but uses a special connection instead to forward the event information to the QF-Test web engine, then waits for the event to be performed there and received back into the Java application via Webswing or JPro.

The final building block for end-to-end tests is verification of what gets displayed in the browser via image checks. Instead of using off-screen-rendering, QF-Test can delegate taking images to the web engine which captures a screenshot of the respective region in the browser window. These images will vary from the Java off-screen variants in subtle ways for font-rendering or anti-aliasing which can be accommodated for by using QF-Test's image check algorithms as described in chapter 57[1167].

Tests using Java-mode are very robust and more efficient. Our recommendation is to use that mode for migrating existing tests and for running the bulk of the functional tests.

These should be supplemented with various tests using web-mode to ensure end-to-end reliability. As a rule of thumb, testing the same UI with different values and the focus on functionality should mostly use Java-mode. Testing different components with the focus on interaction should use web-mode.

Procedures for switching between the various option settings are provided in the package `qfs.jib` of the standard library suite `qfs.qft`.

Chapter 19

Testing Electron applications

Electron[1] is a framework for executing cross-platform desktop applications using the web browser Chromium and the Node.js framework. HTML, CSS und JavaScript can be used for the development of the applications. Electron applications can access native functionality of the operating system such as menus, files or the task bar.

Since version 4.5 QF-Test can handle applications developed via the Electron framework. All features QF-Test supports for the web engine can also be used for Electron testing.

19.1   Starting the Electron Client

Connecting to an Electron application can be realized using the recommended CDP-Driver connection mode (see section 49.3.2[(994)]) or the WebDriver connection mode (see section 49.3.3[(995)]).

The quickstart wizard (see chapter 3[(28)]) helps you generate the correct Setup[(554)] sequence for an easy start of the application.

The Electron specific parameters for the quickstart wizard will be explained in this chapter. For the remaining optional parameters you will find an explanation within the quickstart wizard itself.

Video The video shows

▷   ' Starting your Electron application via the Quickstart wizard'.
https://www.qfs.de/en/yt/electron-45.html

---

[1]https://electronjs.org

19.1.1   Electron settings for the quickstart wizard

In the quickstart wizard select `An Electron application` in the section 'Type of the Application'.

Please enter the fully qualified executable for the application in the section 'Electron application'. You can make use of the file selection dialog by clicking the button to the right of the text field. If your application requires specific command line arguments, you can provide them here.

Electron is based on Node.js, which is executed in the JavaScript runtime environment 'V8'. Since Electron 6 and QF-Test 5.4.0 the CDP-Driver connection mode is used to control the application. Older applications require the WebDriver connection mode in combination with a ChromeDriver. In most cases, QF-Test detects the required ChromeDriver automatically and downloads it. The downloaded driver will be saved in the subdirectory `chromedriver` of the QF-Test installation directory.

19.2   Electron specific functionality of QF-Test

For Electron testing you can use all the features QF-Test offers for web testing plus the following:

19.2.1   Native Menus

The Selection[(697)] node allows you to control native menus in Electron applications.

Please enter the QF-Test ID of the node `Web page` of the SUT in the attribute QF-Test component ID.

The menu item to be selected goes in the Detail[(699)] attribute using the following syntax: `clickmenu:/@<menu path>`, where `<menu path>` is the menu name plus the menu item(s), separated by `/`. For example, if you want to select the menu item `Save as` in the menu `File` the correct entry would be `clickmenu:@/File/Save as...`.

19.2.2   Native Dialogs

5.1.0+   QF-Test supports capture, check and control of dialogs instantiated with the `dialog`-module of Electron. For technical reasons, during the test the dialogs can be optically different from the usual Electron-dialogs.

A capture of a native dialog results in a component-node with the class `Dialog`. It is possible to check the text of the dialog-window using a node Check text[(708)]. The interac-

tion with the dialog-window can be performed using a node Selection[697]. The Detail[699] value of a Selection-node depends on a type of the dialog:

- Message Box: The value of the Detail[699] attribute is the number of the button to select, e.g. `2`. If the Message Box contains a CheckBox, it is possible to append its value separated with `:`, e.g. `2:true`.

- Error Box: An Error Box contains just one button, so the value of the Detail[699] attribute should be `0`.

- Open File Dialog: The Detail[699] attribute should contain the name of the file to select. In order to select multiple files, the Detail-attribute should be set to a Json-Array containing their names, e.g. `["file.txt","C:\\TEMP\\other.txt"]`. It is possible to cancel the dialog by setting the Detail-attribute to `<CANCEL>`.

- Save File Dialog: A Save File Dialog can be controlled in the same way as an Open File Dialog. Selection of multiple files in a Save File Dialog is not supported by Electron.

19.2.3 Extended Javascript-API

5.4.0+

In Electron applications separate render processes control the content view of the application windows. In addition, a so called main process, built upon the Node.js engine, executes the main application logic. To execute individual code in the context of this process, QF-Test provides the methods `mainCallJS` and `mainEvalJS` as powerfull extension of the `DocumentNode`-API (see section 52.11.2[1126]).

`Object mainCallJS(String code)`

Runs Javascript code as function in the main process of the Electron application.

Parameters

code	The code to run.
Returns	Whatever the code returns explicitly using a `return` statement, converted to the proper object type. General Javascript objects will be converted to Json objects. The specific variable `_qf_window` will be replaced by the `BrowserWindow` objekt, which corresponds to the current `DocumentNode`.

`Object mainEvalJS(String script)`

Evaluates Javascript code in the main process of the Electron application.

Parameters

`script`	The script to execute.
Returns	Whatever the script returns, converted to the proper object type. General Javascript objects will be converted to Json objects. The specific variable `_qf_window` will be replaced by the `BrowserWindow` objekt, which corresponds to the current `DocumentNode`.

In the example, the "Chrome Developer Tools"" will be displayed in the current Electron window.

```
rc.getComponent("genericDocument").mainCallJS("_qf_window.webContents.openDevTools()")
```

Example 19.1: SUT script to display the Dev Tools in an Electron window

19.3 Technical remarks on testing Electron applications in WebDriver connection mode

To support testing the Electron APIs, e.g. record and replay native menu interaction, QF-Test has to be able to access the core Electron APIs in WebDriver connection mode from the renderer processes of your application. In practice, this means that the `nodeIntegration` preference of the BrowserWindow should not be set to `false`. In addition, `contextIsolation` must be left deactivated and `enableRemoteModule` must remain `true`:

```
mainWindow = new BrowserWindow({
   webPreferences: {
      nodeIntegration: true,
      enableRemoteModule: true,
      contextIsolation: false,
      ...
   },
   ...
})
```

Example 19.2: Basic example for good testability in Electron apps

If you want to avoid to expose the complete node integration into the browser window web content, you can enable QF-Test to access the API integration using a preload script:

```
mainWindow = new BrowserWindow({
  webPreferences: {
      nodeIntegration: false,
      ...
      preload: `${__dirname}/preload.js` // absolute pathname required
  },
  ...
})
```

Example 19.3: The require preferences for limited node integration

```
// Expose require API in test mode:
if (process.env.NODE_ENV === 'test') {
  window.electronRequire = require;
}
```

Example 19.4: The corresponding `preload.js`

Since QF-Test always sets the `NODE_ENV` environment variable to `test`, you can use this to dynamically loosen the access security during test:

```
const inTestMode = (process.env.NODE_ENV === 'test');
mainWindow = new BrowserWindow({
  webPreferences: {
      nodeIntegration: inTestMode,
      enableRemoteModule: inTestMode,
      contextIsolation: ! inTestMode,
      ...
  },
  ...
})
```

Example 19.5: Dynamic example for good testability in Electron apps

Starting with Electron 14, the `remote` module is not part of the Electron API anymore, but must be explicitly included. To do so, add at development time a reference to the `@electron/remote` module in your `package.json` and initialize the module in your `main.js`:

```
// in the main process:
require('@electron/remote/main').initialize()
```

Example 19.6: How to initialize the `@electron/remote` module

QF-Test automatically uses the new module if detected. More information about the module can be found it the documentation at https://github.com/electron/remote/.

When using the CDP-Driver connection mode, no specific adaptation of the Electron application is required for QF-Test.

Chapter 20

Testing web services

From version 4.2 onwards QF-Test offers the possibility to test web services.

Unlike the well known capture replay model here you must take care yourself to build the HTTP request and verify or validate the responses and/or the results. It is highly suggested to use the existing documentation of the web services you will test. For testing SOAP web services you have to build every HTTP request, there is no automated creation from a WSDL file.

20.1   RESTful web services

The node Server HTTP request[(802)] is used for sending arbitary HTTP packets to a host.

20.1.1   HTTP standards and web services

The web services and web sites all use the Hypertext Transfer Protocol. It is a text based communication made of requests and responses. Here are the most useful and suprisingly short internet standards: Hypertext Transfer Protocol – HTTP/1.1

HTTP Authentication, 2 Basic Authentication Scheme

A list of currently supported HTTP request methods

Supported HTTP Methods
GET
POST
PUT
DELETE
HEAD
OPTIONS
TRACE

Table 20.1: Supported HTTP Methods

20.1.2 HTTP request

Let's examine a simple browser GET request. When you open a web page/URL in the browser, the browser makes a HTTP GET request for you. Here is an example taken via the developer tools in Chrome. The HTTP request consists of `headers`, `URL` and optional `payload` (body).

Figure 20.1: Browser send HTTP GET

The response from the server has `response code`, `headers` and optional `payload`.

Figure 20.2: GET response

20.1.3 Examples

Unlike web browsers when using the Server HTTP Request node you must enter all required data in the respective places, e.g. headers payload etc. Response handling should also be created if needed by using the variables filled in by the server response. Such examples can be found in the example test suite `demo/webservices` named `webservice_testing.qft`.

The examples are built with the help of a HTTP Proxy used for development purposes. Such a proxy is Charles (https://www.charlesproxy.com/) or the free alternative James (https://github.com/james-proxy/james).

Chapter 21

Data-driven testing

Data-driven testing is a very important aspect of test automation. In short, the goal is to run a given test or set of tests multiple times with different sets of input data and expected results. QF-Test has various means to store data or load external data for use in data-driven tests. The most convenient is based on a Data driver node which sets up an environment for iterating over the sets of data combined with one or more Data binder nodes to bind the variables for test execution. Note that there is no Data binder node as such. The name serves as a generic term for the specific nodes like a Data table or a CSV data file. This is best explained through some examples. A demo test suite with simple and advanced examples named `datadriver.qft` is provided in the directory `doc/tutorial` below QF-Test's root directory. Please take care to store modified test suites in a project-related folder.

21.1 Data driver examples
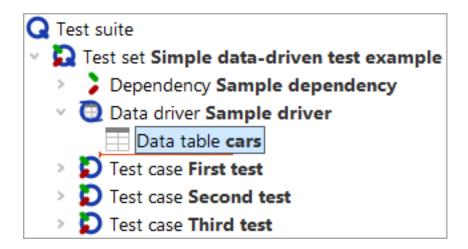
Figure 21.1: A simple data-driven test

The image above shows a Test set with a Data driver node that contains a single Data binder in the form of a Data table node. The contents of the Data table are as follows:

Figure 21.2: Data table example

When the Test set is executed, it will iterate over the rows of the Data table shown above. For each of the three iterations the values of the respective row in the table are bound to the variable named in the matching column header. Thus, during the first iteration the variable named "Model" is bound to "Rolo", "Variant" to "None" and "Price" to "19000". During the second iteration, "Model" is set to "I5" and to "Minigolf" during the third and last iteration. For each iteration, all Test case child nodes of the Test set are run.

The following image shows a run log for the above Test set.
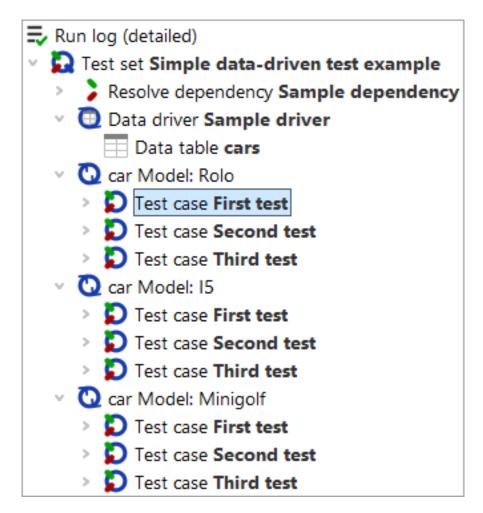
Figure 21.3: Run log of a data-driven test

The next example shows that data-driven testing is not limited to a single loop:
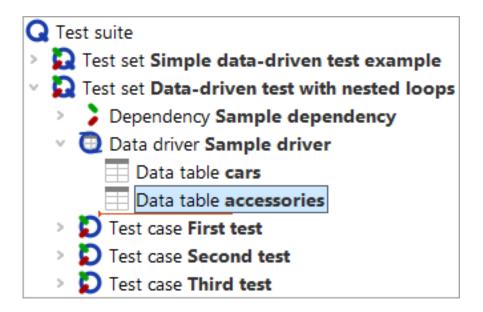
Figure 21.4: Data-driven test with nested loops

The Data driver now contains a second Data table node with the following contents:

Data table

Name

accessories

Iteration counter

Iteration ranges

Data bindings

	Name	APrice
0	Alloy Rims	900
1	Radio with CD	400

QF-Test ID

Delay before (ms)

Delay after (ms)

Comment

Figure 21.5: Second data table example

The Test set will now go through a total of six iterations because for each of the three iterations of the outer loop "cars", both iterations of the inner loop "accessories" will be run as shown in the following run log.
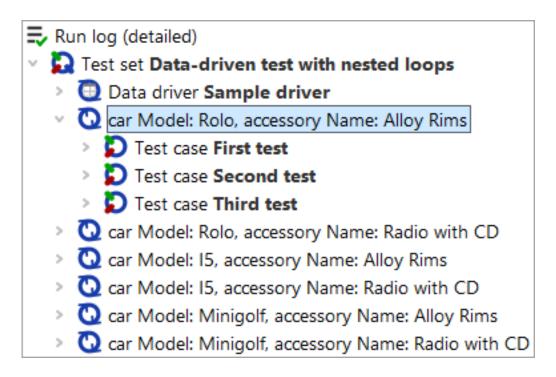
Figure 21.6: Run log of a data-driven test with nested loops

Note The extremely useful dynamic names of the loop nodes in the run logs are obtained by setting the attribute Name for loop pass in the run log to the value "car Model: $(Model)"in the first and to "car Model: $(Model), accessory Name: $(Accessory)" in the second example. As you can see, that name is expanded individually for each iteration, so you can make use of the variables bound for that iteration.

21.2 General use of Data drivers

As seen in the example above the Data driver node must be placed in a Test set node, between the optional Dependency and Setup nodes. When the Test set is executed it will check for Data driver and run it. The contents of the Data driver node are not limited to Data binders. Like a normal Sequence the Data driver node can hold any executable node to be able to perform any setup that may be required to retrieve the data. Thus it is also possible to share Data binders by putting them inside a Procedure and calling the Procedure from inside the Data driver.

Conceptually, a Data binder represents a loop where a different set of variables is bound for each iteration. A Data binder must be registered with a name in the Data driver context of a Test set. This ensures that the loop can be interrupted by a Break[604] node with the same name. Once the Test set has run the Data driver node, it will iterate over the

registered data loops and perform the tests.

In case of nested loops the Data binder that was registered first represents the outermost loop. Its variables are bound first and have lesser precedence than the variables from the inner loop(s).

21.3   Examples for Data drivers

We provide a couple of examples for reading CSV or Excel files in the test suite `doc/tutorial/datadriver.qft`.

21.4   Advanced use

Besides the Data table[(566)] node there are various other means for binding data in a data driver. The Excel data file[(574)], CSV data file[(579)], Database[(569)] and Data loop[(583)] nodes are all explained in detail in section 40.4[(562)].

It is also possible to bind data by calling the Procedures `qfs.databinder.bindList` or `qfs.databinder.bindSets` in the standard library `qfs.qft`. These take as parameters strings with lists or sets of values to split and iterate over. Please see tutorial chapter 8 for information about the standard library.

And finally, data can be bound directly from Jython (and analogous from Groovy and JavaScript) with the help of the `databinder` module, which offers the following methods:

`void bindDict(Object rc, String loopname, dictionary dict,`
`String counter=None, String intervals=None)`
Create and register a databinder that binds data from a dictionary. The keys of the dictionary are the names of the variables and the values are sequences of values to be bound.
Parameters

`rc`	The current run context.
`loopname`	The name under which to bind the data, equivalent to the Name attribute of a Data binder node.
`dict`	The dictionary to bind.
`counter`	An optional variable name for the iteration counter.
`intervals`	Optional ranges of indices, separated by comma, e.g. "0,2-3".

```
void bindList(Object rc, String loopname, String varname,
Object values, String separator=None, String counter=None,
String intervals=None)
```

Create and register a databinder that binds a list of values to a variable.

Parameters

`rc`	The current run context.
`loopname`	The name under which to bind the data, equivalent to the Name attribute of a Data binder node.
`varname`	The name of the variable to bind to.
`values`	The values to bind. Either a sequence type or a string to split.
`separator`	Optional separator character to split the values at in case they're a string. Default is whitespace.
`counter`	An optional variable name for the iteration counter.
`intervals`	Optional ranges of indices, separated by comma, e.g. "0,2-3".

```
void bindSets(Object rc, String loopname, Object varnames,
Object values, String separator=None, String counter=None,
String intervals=None)
```

Create and register a databinder that binds a list of value-set to a set of variables.

Parameters

`rc`	The current run context.
`loopname`	The name under which to bind the data, equivalent to the Name attribute of a Data binder node.
`varnames`	The names of the variables to bind to. Either a sequence type or a string to split.
`values`	The value-sets to bind. Either a sequence of sequences - each inner sequence being one set of data to bind - or a string to split.
`separator`	Optional separator character to split the varnames and the values of a value-set at in case they're a string. Default is whitespace. Value-sets are separated by line-breaks.
`counter`	An optional variable name for the iteration counter.
`intervals`	Optional ranges of indices, separated by comma, e.g. "0,2-3".

Some examples:

```
import databinder
# Three iterations with the values "spam", "bacon" and "eggs"
# bound to the variable named "ingredient"
databinder.bindList(rc, "meal", "ingredient", ["spam", "bacon", "eggs"])
# Same with string values
databinder.bindList(rc, "meal", "ingredient", "spam bacon eggs")
# Same with string values and special separator
databinder.bindList(rc, "meal", "ingredient", "spam|bacon|eggs", "|")
# Two iterations, the first with item="apple" and number="5",
# the second with item="orange" and number="3"
databinder.bindSets(rc, "fruit", ["item", "number"],
                    [["apple",5], ["orange",3]])
# Same with string values, note the linebreak
databinder.bindSets(rc, "fruit", "item number", """apple 5
orange 3""")
# Same as before with the data stored in a dict
databinder.bindDict(rc, "fruit",
                    {"item": ["apple", "orange"],
                     "number": [5,3]})
```

Example 21.1: Examples for use of the `databinder` module

Chapter 22

Reports and test documentation

Besides test suites and run logs QF-Test can create a number of additional documents. Most important of these is the report, which summarizes the overall results of a test run along with an overview over the test suites executed and their individual results. The report is easy to read and understand without further knowledge about QF-Test and thus complements the run log which is geared towards error analysis and requires some QF-Test experience to fully understand.

Following is an example of a report summary:

Figure 22.1: Example report

The other documents are more static in nature, describing the content of test suites instead of test run results. The *testdoc* document gives an overview over the structure of Test set[(525)] and Test case[(518)] nodes in a given set of test suites. It is intended for the test architect or QA project manager and documents the current state of test development. The *pkgdoc* documentation is similar, but focuses on Package[(594)], Procedure[(586)] and Dependency[(548)] nodes instead. The result is a library reference comparable to Java's JavaDoc. The target audience for pgkdoc are test implementers requiring information about already existing procedures and their parameters.

22.1 Reports

Before we can start explaining how to create reports, some basic terms and concepts need to be defined.

22.1.1 Report concepts

A report represents the results of one or more *test runs*. A test run comprises the execution of either a single test suite or multiple test suites, typically executed together in one batch-run. A test run is identified by a *runid*. It is possible to execute a test run in several steps by assigning the same runid to the resulting run logs.

A report is identified by a *report name*. For a report that covers a single test run, the report name is usually the same as the runid. For reports summarizing the results of several test runs a distinct report name can be specified.

Reports can be created in multiple variants: XML, HTML and JUnit. Most users will probably use the HTML variant which can be viewed in a browser, printed and archived. The XML variant can serve as the basis for collecting the results of a test run for further processing, for example to collect test results in a database or to create customized HTML reports. We suggest that you always create both HTML and XML reports unless you have a good reason to do otherwise. JUnit reports base on the JUnit XML format as created by Apache Ant by use of its JUnitReport task. This format is not as pretty and detailed as the first two report variants QF-Test offers but it is directly understood by many continuous integration tools and may prove useful for a quick integration with those.

A report consists of one summary document, plus one document per run log. These files are collected together with complementary files like icons, stylesheets and screenshot images in a directory. At the file level, this directory represents the report.

The layout of the files inside the report directory depends on some command line options explained below. Basically there are two ways to lay out the files: Based on the file structure of the original test suites or based on the file structure of the run logs.

22.1.2 Report contents

In advance to the overall test result, a report as shown above starts with a summary containing informational system data and a legend describing the meaning of counter icons used in the report (see Running tests[(37)]).

Note The difference between "Time spent in tests" and "Elapsed time" are explicit delays introduced in nodes via the 'Delay before/after' attribute or user interrupts.

The contents of a report are based on the original structure of the executed test suites. The main structure is created from Test set[(525)] and Test case[(518)] nodes. The Comment[(531)] attributes of the root node as well as the Test set and Test case nodes share the doctags with testdoc documents as explained in section 22.2[(276)]. In addition to those doctags the '@title' doctag can be specified in the comment of the root node to set a title for the report document created for the respective test suite.

If `-report-teststeps`[(878)] is specified in batch mode (true by default) or the respective option is active in the interactive dialog, Test cases can be further broken down into steps with the help of Test step[(539)] nodes. In addition to explicitly wrapping steps into a Test step, any node can be turned into a test step by specifying the doctag '@teststep' in its Comment, followed by an optional name for the step. For Test step nodes the '@author', '@version' and '@since' doctags are also applicable. The names, comments and tag values of the various nodes can contain variables that will be expanded at execution time so that the expanded value is shown in the report. This is especially useful for test steps within a procedure.

If listing of test steps is active, Setup, Cleanup and Dependency nodes are also listed and checks, screenshots and messages, including warnings, errors and exceptions are properly integrated into the nested steps. If the test suites are set up properly the resulting report can serve as a very readable summary of what was going on during the execution of a test.

Whether warnings and checks are listed is determined by the command line arguments `-report-warnings`[(878)] and `-report-checks`[(876)] or the respective interactive options. Warnings from component recognition are never listed because they are too technical and could easily flood the report. For checks one must distinguish between checks that represent an actual verification step and those that are used solely for control flow, for example to check whether a checkbox is already selected and click it only in case it is not. By default QF-Test lists those Check nodes in the report that have the default result settings, i.e. the Error level of message is 'Error', no exception is thrown and no result variable bound. All others are treated as helpers for control flow and not listed in the report. For cases where this default treatment is not appropriate, you can force a Check into the report via the doctag '@report' in its Comment attribute or prevent its listing via '@noreport'. Of course failed checks are treated as warnings, errors or exceptions (depending on their Error level of message) and cannot be excluded from the report if messages at the respective level are shown.

Additional messages, checks and screenshots can be added to the report by scripts via the methods `rc.logMessage`, `rc.logImage` and `rc.check` and its variants, which have an optional `report` parameter. For details, please see the run context API documentation in section 48.5[(917)].

22.1.3 Creating reports

There are three ways to create reports:

- Interactively from a run log through the menu item File→Create report... .

- In batch mode as the result of a test run.

- In batch mode by transforming already existing run logs.

The interactive variant is easy to use. Just select the target directory for the report and whether you want the XML and/or the HTML variant.

For report creation in batch mode there are a number of command line options which are listed and explained in section 42.2[(867)]. Let's look at the variant of creating reports as the result of a test run first:

The command line syntax for plain test execution in batch mode is `qftest -batch <test suite> [<test suite>...]`

To create a combined XML and HTML report, use `-report <directory>`[(876)]. To create only one version or to separate the XML, HTML variants, use `-report-xml <directory>`[(878)] and/or `-report-html <directory>`[(877)]. For JUnit reports `-report-junit <directory>`[(877)] works respectively.

The runid of a test run is specified with `-runid <ID>`[(878)], the name of the report with `-report-name <name>`[(877)]. If the report name is unspecified it will default to the runid.

To lay out the files in the report directory according to the file structure of the test suites, use `-sourcedir <directory>`[(880)]. To use the file structure of the run log as the basis, use `-runlogdir <directory>`[(879)].

The following is a typical example of a command line for a batch run making use of the placeholders explained in section 42.2.4[(884)]:

```
qftest -batch -runid +M+d -runlog logs/+i -report report_+i
        -sourcedir . suite1.qft subdir/suite2.qft
```

Example 22.1: Creating a report as the result of a test run

Creating a report as a separate step by transforming a set of run logs is similar in many respects. The run logs to transform have to be specified instead of the test suites to execute and the `-runid <ID>` and `-sourcedir <directory>` command line options have no effect. The following is an example for how to create a weekly summary report based on the assumption that you have collected all run logs below the directory named `logdir`, possibly in subdirectorys thereof:

```
qftest -batch -genreport -report report_+M+d
        -report.name week_of_+y+M+d logdir
```

Example 22.2: Creating a weekly summary report

22.1.4   Customizing reports

The XML and HTML reports are created from the run log via XSLT. By changing the XSLT stylesheets used it is possible to change the content and structure of the resulting documents.

You can find more on this possibility in our blog article "Creating custom HTML/XML/Junit reports" at https://www.qfs.de/en/blog/article/2019/02/28/creating-custom-htmlxmljunit-reports.html

As an alternative it is possible to customize the display of the HTML report using established web techniques via JavaScript. A file named `user.js` is copied to the report directory and included in all pages of the HTML report. To change the report layout you can replace this file with your own version after creating the report. See the comments in the default `user.js` file for examples.

22.2   Testdoc documentation for Test sets and Test cases

The type of test documents called *testdoc* provide overview and detailed information over the Test set[(525)] and Test case[(518)] nodes of one or more test suites. When Test cases contain Test steps[(539)] those steps will be included in the testdoc. By default QF-Test ignores Test call[(531)] nodes during testdoc creation. By setting the option `-testdoc-followcalls`[(882)]=true the real targets Test case, Test set or the whole test suite are processed as if they were part of the original test suite.

This documentation is a valuable tool for QA project managers to keep track of the current state of test development. Similar to reports, testdoc documents are laid out as directories with one summary file and one detailed file per test suite.

A testdoc document for a single suite can be created interactively from a test suite by selecting ⟦Create testdoc documentation...⟧ from the ⟦File⟧ menu. This is very useful during test development to quickly check whether all tests are properly documented.

For actual use as a reference it is preferable to create complete sets of documents spanning multiple test suites for a whole project. This can be done by running QF-Test in batch mode with the `-gendoc`[(872)] command line argument. In its simplest form, a call to create testdoc documentation for a whole directory tree would look as follows:

```
qftest -batch -gendoc -testdoc test_documentation
        directory/with/test suites
```

Example 22.3: Creating testdoc documentation

Please see chapter 42[(862)] for detailed information about the available command line

arguments.

To get optimal results you can use HTML markup in the Comment[531] attributes of Test set and Test case nodes and also make use of doctags. A *doctag* is a keyword beginning with '@', sometimes followed by a name and always by a description. This is a proven concept in JavaDoc, the standard documentation format for Java programs (see   http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#tag).

Note   All doctags must appear after the main description. Description after the doctags will be ignored, as well doctags inside the descripiton are not allowed.

The following doctags are supported for Test set and Test case nodes:

@deprecated
> If a Test set or Test case is no longer to be used, this description should explain when and why the node was deprecated and especially which replacement should be used.

@condition
> Non-formal explanation of the condition under which the node is executed or skipped.

@param
> Description for a parameter. Following are the name of the parameter and its description.

@charvar
> Description for a characteristic variable. Following are the name of the variable and its description.

@author
> Author of the Test set or Test case.

@version
> Version of the Test case or Test case.

@since
> The version since which this Test set or Test case has been available.

In addition to the doctags described above, the doctag '@title' in the comment of the root node can be used to specify a title for the testdoc document created for the respective test suite.

22.3 Pkgdoc documentation for Packages, Procedures and Dependencies

The concepts of and methods for creation of *pkgdoc* documents are nearly identical to testdoc, so this section is brief. Instead of Test set and Test case nodes, pkgdoc documents cover Package[(594)], Procedure[(586)] and Dependency[(548)] nodes. They are intended for the test developer to keep track of the procedures available for use in implementing tests.

Please refer to the standard library `qfs.qft` as a good example how a pkgdoc may look like.

A pkgdoc document can also either be created interactively using File→Create HMTL/XML pkgdoc... or in batch mode. Again, please see chapter 42[(862)] for detailed information about the available command line arguments.

As the following example shows, testdoc and pkgdoc can even be created together in a single batch run:

```
qftest -batch -gendoc -testdoc tests -pkgdoc procedures
        directory/with/test suites
```

Example 22.4: Creating testdoc and pkgdoc documentation in a single run

Of course pkgdoc also supports HTML markup and doctags. The following doctags are supported for Package, Procedure and Dependency nodes:

@deprecated
> If a Procedure, Dependency or Package is no longer to be used, this description should explain when and why the node was deprecated and especially which replacement should be used.

@param (Procedure and Dependency only)
> A parameter of a Procedure or Dependency. Following are the name of the parameter and its description.

@charvar (Dependency only)
> Description for a characteristic variable of a Dependency. Following are the name of the variable and its description.

@return (Procedure only)
> The return value of the procedure.

@result (Procedure and Dependency only)
> Can be used to document side-effects of the Procedure or Dependency like setting a global variable.

@throws (Procedure only)
>   Expected exception. Following are the name of the exception and a description of its cause.

@catches (Dependency only)
>   An exception being caught by the Dependency. Following are the name of the exception and a description of the handler.

@author
>   Author of the Package, Procedure or Dependency.

@version
>   Version of the Package, Procedure or Dependency.

@since
>   The version since which this Package, Procedure or Dependency is available.

In addition to the doctags described above, the doctag '@title' in the comment of the root node can be used to specify a title for the pkgdoc document created for the respective test suite.

Chapter 23

Test execution

When talking about test execution, there are two aspects to be considered. On one hand you need to run tests while they are developed to check them for proper operation. This situation has already been described in section 4.2[(37)]. Basically all you have to do to run a test interactively is invoking Run→Start from the main menu.

On the other hand you want to run your tests periodically to ensure the stability of the system under test, for example in nightly regression tests. Instead of launching QF-Test, loading the test suite and running it from the graphical user interface, it is much more convenient here to execute tests from the command line in batch mode. This kind of running tests is explained in the first section of this chapter (Test execution in batch mode[(280)]).

Sometimes, for instance when you want to run the test on a remote computer, a second variant comes into play: the daemon mode. This type of test execution, which uses a running QF-Test instance to execute tests, is the topic of the second section (Executing tests in daemon mode[(286)]).

For integration of QF-Test with build tools like `ant`, `maven` or `Hudson/Jenkins`, please refer to chapter 27[(336)].

23.1   Test execution in batch mode

There are a lot of command line arguments when running QF-Test in batch mode; an overview can be found in chapter 42[(862)]. Here we will present examples showing the most important of them.

The examples are written for the Windows operating system, but you may easily adapt them for the Linux platform. What is different is the path specification and also the syntax for placeholders (section 42.2.4[(884)]): On Linux you can use +X as well as %X. On Windows there's a separate console application `qftestc.exe`. In contrast to its

GUI variant `qftest.exe`, it waits until the execution of QF-Test has terminated and also displays print output from a Server script[(628)]. You can use `qftestc.exe` in place of `qftest.exe` wherever you'll find it convenient.

23.1.1 Command line usage

Let's start with the most simple QF-Test command to execute a test:

```
qftest -batch -run c:\mysuites\suiteA.qft
```

Example 23.1: Test execution from the command line

The argument `-batch` makes QF-Test start without a graphical user interface. The second argument, `-run`, is the specifier for test execution. Finally, at the end of the command line, you find the test suite to be executed.

Note   The argument `-run` is optional, i. e. the test execution is defined as default for the batch mode.

When running the above command, all top-level Test case and Test set nodes of `suiteA.qft` will be executed one after another. After the test run you will find a run log file in the current directory; it has the same name as the test suite (except from the extension, which can be `.qrl`, `.qrz` or `.qzp`). The run log file shows the result of the test run.

By specifying `-nolog` you can suppress the creation of a run log. Probably this only makes sense, if you have extended your test by your own log output (written to a file). Otherwise you'd have to check the result code of QF-Test, whereas 0 means that everything is alright. A positive value in contrast indicates that warnings, errors or exceptions occurred during the test run (see section 42.3[(885)]). That's why in most situations you'll probably prefer to create a run log and save it at a fixed place in the file system. This can be achieved with the parameter `-runlog`:

```
qftest -batch -compact -runlog c:\mylogs\+b c:\mysuites\suiteA.qft
```

Example 23.2: Test execution with run log creation

A run log file `suiteA.qrz` will now be created in the specified directory `c:\mylogs`. The placeholder `+b` is responsible for its name being identical with that of the test suite. The additional switch `-compact` prevents the run log from growing too large: Only the nodes needed for a report and those immediately before an error or an exception are kept in the run log. Especially in case of very long test runs this may help to reduce

the amount of required memory. The newer method of using split run logs is even more powerful. For more information about that see section 7.1[(117)].

Whether the file is indeed created as compressed run log (to be distinguished from the above "compact") with extension `.qrz`, depends on the system settings. To force the creation of a particular format you can set the file extension explicitly. With `-runlog c:\mylogs\+b.qrl`, for example, an uncompressed XML file will be produced.

Sometimes you may want to execute not the whole test suite but only parts of it. By using the parameter `-test` you can run a specific node of the test suite:

```
qftest -batch -runlog c:\mylogs\+b -test "My test case" c:\mysuites\suiteA.qft
```

Example 23.3: Executing a specified node

The parameter `-test` expects the QF-Test ID attribute of the node to follow or the qualified name of a Test case or Test set. If you want to execute several nodes, you can define `-test <ID>` multiple times. Apart from the node's QF-Test ID, `-test` accepts also the numerical index of a top-level node. For example, `-test 0` will run the first child of the Test suite node.

The run log provides a rather technical view of the test run; it is helpful mainly when analyzing errors (cf. section 7.1[(117)]). The report in contrast contains a summary of the executed test cases and errors (cf. chapter 22[(271)]) in XML or HTML format. It is created from the run log either in a separate step after running the test or automatically with the test run:

```
qftest -batch -runlog c:\mylogs\+b
       -report c:\mylogs\rep_+b_+y+M+d+h+m
       c:\mysuites\suiteA.qft
```

Example 23.4: Creating a report

In this example the XML and HTML files are saved in a directory which name consists of the test suite and a timestamp like `c:\mylogs\rep_suiteA_0806042152`. When replacing the argument `-report` with `-report.xml` or `-report.html` respectively, only an XML or HTML report will be created.

Test cases often uses variables to control the execution of the test. For example, you may have defined the variable `myvar` in the Test suite[(515)] node of the suite. You can overwrite its default value when running the test suite from the command line:

```
qftest -batch -variable myvar="Value from command line"
       -runlog c:\mylogs\+b c:\mysuites\suiteA.qft
```

Example 23.5: Test execution with variables

If needed, you can specify `-variable <name>=<wert>` multiple times to set values for different variables.

23.1.2 Windows batch script

Running tests from the command line is fundamental for integrating QF-Test in test management systems (see Interaction with Test Management Tools[(312)]). Otherwise, living without such a tool, you may find it convenient to embed the command for the test execution into a script. A simple Windows batch script (`qfbatch.bat`) looks like this:

```
@echo off
setlocal
if "%1" == "" (
    echo Usage: qfbatch Testsuite
    goto end
) else (
    set suite=%~f1
)
set logdir=c:\mylogs
pushd c:\programs\qftest\qftest-7.1.2\bin
@echo on
.\qftest -batch -compact -runlog %logdir%\+b %suite%
@echo off
if %errorlevel% equ 0 (
  echo Test terminated successfully
  goto end
)
if %errorlevel% equ 1 (
  echo Test terminated with warnings
  goto end
)
if %errorlevel% equ 2 (
  echo Test terminated with errors
  goto end
)
if %errorlevel% equ 3 (
  echo Test terminated with exceptions
  goto end
)
if %errorlevel% leq -1 (
  echo Error %errorlevel%
  goto end
)
:end
popd
```

Example 23.6: Batch script `qfbatch.bat` to execute a test suite

Now you can simply run that script with only the file name of the test suite as parameter. Everything else is done automatically: The test suite will be executed, the run log file stored in `logdir` and finally the script will print out the state of the test run (depending on the QF-Test result code).

23.1.3   Groovy

Since version QF-Test 3 the language Groovy is part of the release (cf. chapter 11[(161)]). It is meant mainly for scripting inside QF-Test (Server and SUT scripts), but it can, like Jython, also be used outside of QF-Test. Groovy is probably well suited to create a little test execution management system by yourself. By the way, Groovy simplifies working with Ant, too: Instead of dealing with bulky XML files, which makes it hard to define conditions, you can work with the Groovy `AntBuilder`. However, that's out of scope here, the following example doesn't rely on Ant but only on the basic Groovy features:

```
def suite = ''
if (args.size() == 0) {
    println 'Usage: groovy QfExec Testsuite'
    return
}
else {
    suite = args[0]
}
def qftestdir = 'c:\\programs\\qfs\\qftest\\qftest-7.1.2'
def qftest = qftestdir + '\\bin\\qftest.exe'
def command = [qftest,
               "-batch",
               "-compact",
               "-runlog", "c:\\mylogs\\+b",
               suite]
def printStream = { stream ->
    while (true) {
        try {
            stream.eachLine { println it }
        } catch (IOException) {
            break
        }
    }
}
println "Running command: $command"
def proc = command.execute()
new Thread().start() { printStream(proc.in) }
new Thread().start() { printStream(proc.err) }
proc.waitFor()
switch (proc.exitValue()) {
    case '0': println 'Test terminated successfully'; break
    case '1': println 'Test terminated with warnings'; break
    case '2': println 'Test terminated with errors'; break
    case '3': println 'Test terminated with exceptions'; break
    default: println "Error ${proc.exitValue()}"
}
```

Example 23.7: Groovy script `QfExec.groovy` to execute a test suite

If you have Groovy installed on your computer independently of QF-Test, you can run the example test suite simply via `groovy QfExec c:\mysuites\suiteA.qft`. Otherwise you can use the Groovy jar file from the QF-Test installation, preferably again with help of a batch script:

```
@echo off
setlocal
if "%1" == "" (
    echo Usage: qfexec Testsuite
    goto end
)
set qftestdir=c:\programs\qftest\qftest-7.1.2
set scriptfile=QfExec.groovy
java -cp %qftestdir%/lib/groovy-all.jar groovy.ui.GroovyMain %scriptfile% %*
:end
```

Example 23.8: Batch script `qfexec.bat` to run a Groovy script (here: `QfExec.groovy`)

Now execute the test suite with `qfexec c:\mysuites\suiteA.qft`.

23.2 Executing tests in daemon mode

In daemon mode QF-Test listens to RMI connections and provides an interface for remote test execution. This is useful for simplifying test execution in a distributed load-testing scenario (chapter 31[(375)]), but also for integration with existing test management or test execution tools (chapter 26[(312)]).

Note GUI tests require an active user session. Chapter Hints on setting up test systems[(410)] contains useful tips and tricks to set-up the daemon process. In FAQ 14 you can find technical details.

23.2.1 Launching the daemon

!!! Warning !!!

Anybody with access to the QF-Test daemon can start any program on the machine running the daemon with the rights of the user account that the daemon is running under, so access should be granted only to trusted users.

If you are not running the daemon in a secure environment where every user is trusted or if you are creating your own library to connect to the QF-Test daemon, you definitely should **read section 53.3**[(1154)] about how to secure daemon communication with SSL.

To work with a daemon, you must first launch it on any computer in your network (of course, this host can also be `localhost`):

```
qftest -batch -daemon -daemonport 12345
```

Example 23.9: Launching a QF-Test daemon

Note

Important compatibility note:

3.5+

Starting with QF-Test version 3.5, SSL is used for daemon communication by default. To interact with a QF-Test version older than 3.5 you must start the daemon with an empty `-keystore <keystore file>`[(873)] argument in the form:

```
qftest -batch -keystore= -daemon -daemonport 12345
```

Example 23.10: Launching a QF-Test daemon without SSL

If you omit the argument `-daemonport`, the daemon will listen on QF-Test's standard port 3543. You may check whether the daemon is running by means of the `netstat` utility:

Windows

```
netstat -a -p tcp -n | findstr "12345"
```

Linux

```
netstat -a --tcp --numeric-ports | grep 12345
```

If you want to launch a daemon on a remote host, you may use for instance ssh or VNC. Your network administrator knows whether and how this works. To follow the examples below, a local daemon will be sufficient.

3.0+

23.2.2 Controlling a daemon from QF-Test's command line

The easiest way to get in touch with a daemon is running QF-Test from the command line in the *calldaemon* mode. The following example checks if a daemon is listening at the specified host and port:

```
qftestc -batch -calldaemon -daemonhost localhost -daemonport 12345 -ping
```

Example 23.11: Pinging a QF-Test daemon

In contrast to the `netstat` command from above `-ping` also works between different computers (if you check the daemon on your local computer, you can omit the argument `-daemonhost`).

What you actually want from a daemon is executing your test case(s) and getting back a run log file. It sounds and indeed looks quite similar to what you have seen before when running a test in batch mode:

```
qftest -batch -calldaemon -daemonhost somehost -daemonport 12345
        -runlog c:\mylogs\+b
        -suitedir c:\mysuites
        suiteA.qft#"My test case"
```

Example 23.12: Running a test case with the QF-Test daemon

Note
In contrast to the batch mode, a Test case or a Test set node is always referenced here by its qualified name, for instance "My Test set.My Test case" (just to remember: -test <ID> may be used in batch mode). To execute the complete suite suiteA.qft, you can simply omit the test case or write suiteA.qft#..

If the daemon is running on a remote host, you have to specify it explicitly via -daemonhost (default is -daemonhost localhost). Note that the parameter -suitedir refers to the remote host (where the daemon is running) while -runlog defines a local file.

3.4+
In case you cannot easily observe the test running on a remote host, you may find it convenient to add the argument -verbose to get status output in the console (on Windows, use qftestc to see the output).

A running daemon, no matter whether local or remote, can be terminated with the *call-daemon* command -terminate:

```
qftest -batch -calldaemon -daemonport 12345 -daemonhost localhost -terminate
```

Example 23.13: Terminating a QF-Test daemon

A complete list of the *calldaemon* parameters can be found in the chapter <u>Command line arguments and exit codes</u>[(862)].

23.2.3 Controlling a daemon with the daemon API

Using the QF-Test command line to control a daemon was quite easy. On the other hand, to get all capabilities of a daemon, you have to deal with the daemon API. In this section we will concentrate on some basic examples, the whole interface is described in <u>chapter 53</u>[(1139)].

To get started with the daemon API, insert a Server script node with the following code:

```
from de.qfs.apps.qftest.daemon import DaemonRunContext
from de.qfs.apps.qftest.daemon import DaemonLocator
host = "localhost"
port = 12345
# Leading r means raw string to allow normal backslashes in the path string.
testcase = r"c:\mysuites\suiteA.qft#My test case"
timeout = 60 * 1000
def calldaemon(host, port, testcase, timeout=0):
    daemon = DaemonLocator.instance().locateDaemon(host, port)
    trd = daemon.createTestRunDaemon()
    context = trd.createContext()
    context.runTest(testcase)
    if not context.waitForRunState(DaemonRunContext.STATE_FINISHED, timeout):
        # Run did not finish, terminate it
        context.stopRun()
        if not context.waitForRunState(DaemonRunContext.STATE_FINISHED, 5000):
            # Context is deadlocked
            raise UserException("No reply from daemon RunContext.")
        rc.logError("Daemon call did not terminate and had to be stopped.")
    result = context.getResult()
    log = context.getRunLog()
    rc.addDaemonLog(log)
    context.release()
    return result
result = calldaemon(host, port, testcase, timeout)
rc.logMessage("Result from daemon: %d" %result)
```

Example 23.14: Daemon API in a Server script

The script shows the basic mechanisms to control a daemon:

- First find a running daemon with `locateDaemon`.

- Provide an environment for test runs by calling `createTestRunDaemon`.

- To run a test, you need a context object (`createContext`). The creation of that object requires a QF-Test run-time license.

- Now the context enables you to start a test run (`runTest`) and to query about its current state. `waitForRunState` waits during the defined timeout (in milliseconds) until the specified state has occurred. In the example above, we wait for the test to terminate within one minute.

- Finally, when the test run has terminated, the context can query the test result with the method `getResult` (cf. Exit codes for QF-Test[(885)]).

- Moreover, you can use the context to get the run log of the daemon test run. It can be included in the local run log by means of the `rc` method `addDaemonLog`.

Note To keep it small and simple, the example script does not contain any error handling. However, particularly when working with a daemon, you should check every method call.

Note Driving a daemon from a Server script has the disadvantage of consuming an additional QF-Test license to run the script node interactively or in batch mode. However, this doesn't apply nor for the above-mentioned *calldaemon* mode neither for the case when controlling a daemon outside QF-Test (see below).

The usage of the daemon API is not restricted to Server scripts. Outside QF-Test a daemon can be contacted by means of a Java program or, more easily, a Groovy script. The following Groovy script works with several running daemons and may serve as a starting point for load tests. Suppose we have started some daemons in our network, each on a separate machine. We want to execute a test case simultaneously by all of the daemons and we want to save a run log for every single test run (`daemon1.qrl`, `..., daemonN.qrl`). The test suite containing the test case to be executed may be available for all daemon instances via the network drive `z:`).

```
import de.qfs.apps.qftest.daemon.DaemonLocator
import de.qfs.apps.qftest.daemon.DaemonRunContext
def testcase = "z:\\mysuites\\suiteA.qft#My test case"
def logfile = "c:\\mylogs\\daemon"
def timeout = 120 * 1000
def keystore = "z:\\mysuites\\mydaemon.keystore"
def password = "verySecret"
def locator = DaemonLocator.instance()
locator.setKeystore(keystore)
locator.setKeystorePassword(password)
def daemons = locator.locateDaemons(10000)
def contexts = []
// Start tests
for (daemon in daemons) {
    def trd = daemon.createTestRunDaemon()
    trd.setGlobal('machines', daemons.size().toString())
    def context = trd.createContext()
    contexts << context
    context.runTest(testcase)
}
// Wait for tests to terminate
for (i in 0..<contexts.size()) {
    def context = contexts[i]
    context.waitForRunState(DaemonRunContext.STATE_FINISHED, timeout)
    byte[] runlog = context.getRunLog()
    def fos = new FileOutputStream("$logfile${i + 1}.qrl")
    fos.write(runlog)
    fos.close()
    context.release()
}
```

Example 23.15: Groovy daemon script `CallDaemon.groovy`

To run that Groovy script, you need the QF-Test libraries `qftest.jar`, `qfshared.jar`, and `qflib.jar` as well as the Groovy library, which is also part of the QF-Test installation. The following batch script shows how it works:

```
@echo off
setlocal
set qftestdir=c:\programs\qftest\qftest-7.1.2
set qflibdir=%qftestdir%\qflib
set classpath=%qftestdir%\lib\groovy-all.jar
set classpath=%classpath%;%qflibdir%\qftest.jar;%qflibdir%\qfshared.jar;
            %qflibdir%\qflib.jar
java -cp %classpath% groovy.ui.GroovyMain CallDaemon
```

Example 23.16: Batch script `calldaemon.bat` to run `Calldaemon.groovy`

When accessed from externally, the `DaemonLocator` can only determine the default

keystore to encrypt the daemon communication automatically, if the `qftest.jar` file is loaded from the QF-Test-directory (as shown in the batch script). Alternatively (as seen in the groovy script), the keystore can be specified explicitly by calling `setKeystore` and `setKeystorePassword`, or indirectly with the system properties `javax.net.ssl.keyStore` and `javax.net.ssl.keyStorePassword`.

To make the daemon example a load test (cf. chapter 31[375]), you have to synchronize the test runs inside of "My test case" (e. g. after starting the SUT). This can be done by means of the `rc` method `syncThreads`:

```
def machines = rc.getNum('machines')
rc.syncThreads('startup', 60000, -1, machines)
```

Example 23.17: Groovy Server script node to synchronize the test runs

The variable `machines` denotes the number of hosts with a daemon running on them. Best define it in the Test suite node of the test suite with a default value of `1`. When running the Groovy script, it will be overwritten with the correct value.

23.3 Re-execution of nodes (Rerun)

4.1+

23.3.1 Triggering rerun from a run log

When a test run has finished, the run log or report is a good entry point for evaluating the results. In case of errors you may face various challenges. You might want to re-execute the failed test cases to investigate the reason for the error or because you want to perform an official re-test of this failing situation after removing the error condition. If the re-test results are to be shown in the test-report, you may want to replace the previous results or append them to the existing ones. Or you might just want to repeat the test case with the previous variable settings and keep the new run logs and reports separately.

To that end QF-Test offers the capability to trigger re-execution from the run log. You can trigger a rerun via selecting the run log node or any test set node and choose ⟦Rerun test cases⟧ from the ⟦Edit⟧ menu or from the context menu. Alternatively you can select the nodes to rerun it the error-list and use the context menu entry ⟦Rerun test cases of selected nodes⟧. The dialog then shown lets you select the test cases for the rerun and choose how to handle run logs via the selection box `Mode for merging run logs` with the following options:

Choice	Meaning
Replace test cases	Replace the test cases from the original run log with the results from the rerun, i.e. the previous results will get lost. The previous run log will be saved in a backup copy.
Merge run logs	The new test results will be merged into the existing structure.
Append run log	The new test results will be appended to the end of the run log. The test set structure will be ignored.
Keep run logs separated	The new test results will be written to a new run log, the original one remains unchanged.

Table 23.1: Choices for handling the run log of a rerun

Figure 23.1: Dialog to rerun test cases

For each test case the variable values are taken from the run log of the original test run. During re-execution the variable `${qftest:isInRerunFromLog}` is set to `true`, making it possible to distinguish between a normal test run and a a rerun.

Note   Merging of run logs makes use of names of test cases and test sets. Therefore, those

names should be unique. In case of data-driven testing you should take care to keep those names unique via the attributes Name for separate run log or Characteristic variables.

23.3.2   Rerunning failing nodes immediately

During your test-automation project you can sometimes face situations where some test steps don't provide reliable results, failing sometimes but not always. Most of the time such cases depend on timing and can be stabilized using <u>Wait for component to appear</u>[771] nodes, or checks for conditions, delays, scripts or other control structures. As an alternative or additional approach QF-Test offers the capability to repeat such steps whenever they fail.

This automated rerunning in case of error can be applied to every executable node using a certain doctag in the comment attribute. This doctag can look like this:

```
@rerun attempts=3;errorlevel>=ERROR;newerrorlevel=WARNING;
        handler=handlers.errorhandler
```

Example 23.18: Example for a rerun definition

In the example above a failed node will be repeated up to three times until an attempt succeeds. Failed attempts will be downgraded to warning in the run log. In case all attempts fail the last attempt will be reported as error or exception. After every failing attempt QF-Test will execute the procedure `handlers.errorhandler`.

If you are interested in the number of the current rerun attempt, you can use the variable `reruncounter` from the `qftest` variable group, see <u>section 6.5</u>[108].

The `@rerun` doctag has parameters like `attempts` or `errorlevel` with possibilities as follows:

attempts
    The maximum number of attempts.

errorlevel (optional)
    Defines the error states to be handled. Possible values are `EXCEPTION`, `ERROR` or `WARNING` with = for an exact match or > or >= for a range. Specifying `errorlevel=ERROR` means to rerun that node only in case of an error whereas `errorlevel>=ERROR` triggers the rerun in case of errors or exceptions. If this parameter isn't specified the value `errorlevel>=ERROR` will be taken as default.

newerrorlevel (optional)
    Specifies the error-level in the run log for the initial run and possible additional

failed runs. You can again choose between `EXCEPTION`, `ERROR` or `WARNING` with the additional options `NOLOG` and `KEEP`. The level `NOLOG` stands for removing the failing results from the run log entirely. `NOLOG` should be used with extreme care. Using the level `KEEP` doesn't override the original error level and reports it unchanged. If this parameter isn't specified the value `WARNING` will be taken as default.

handler (optional)

The name of the procedure which should be called in case a caught error state occurs. This procedure will be called after each failed attempt.

reusevariables (optional, default=true)

Specifies whether to reuse the variable values from the beginning of the first run. When set to `false` the current variable values will be used.

logmessages (optional, default=true)

If that parameter is set to `true` a message will be written into the run log, when an attempt begins and when the execution of that sequence terminates. In addition every node gets an annotation in the run log with the current attempt.

logmessagesintoreport (optional, default=true)

If this parameter and the parameter `logmessages` are set to `true`, all messages will be written to the report as well.

keepfirst (optional, default=false)

If this value is set to `true` the first failing attempt will be logged with its original error level. In case of further failing attempts those errors will be logged with the `newerrorlevel` level.

exceptiontype (optional)

In case you want to catch only one specific exception you can specify the exception type here, e.g. `CheckFailedException` or just `ClientNotConnected` for a `ClientNotConnectedException`. This parameter should only be used if you set `Exception` as value for the parameter `errorlevel`. Please see the Catch[(619)] node for details about exceptions.

exceptionmessage (optional)

In case you want to catch only one specific exception with one text, you can specify the text here. This parameter should only be used if you set `Exception` as error level. Please see the Catch[(619)] node for details about exceptions.

exceptionregex (optional)

If `true`, the value of `exceptionmessage` is a regular expression. This parameter should only be used if you set `Exception` as error level and an exception message. Please see the Catch[(619)] node for details about exceptions.

exceptionlocalized (optional)

    If `true`, the value of `exceptionmessage` should be the localized exception message, e.g. mostly the full text. This parameter should only be used if you set `Exception` as error level and an exception message. Please see the Catch[619] node for details about exceptions.

Chapter 24

Distributed test development

The previous chapters all focused on creating and organizing sets of reliable tests in a single test suite. However, when testing a large application, a single suite may not be enough. There are at least two scenarios where splitting tests into multiple test suites becomes essential:

- Multiple developers are creating and editing tests. To avoid redundancy and duplication of code, separately developed tests should use common Procedures and Components where possible, but only one person can edit a test suite at a time.

- Tests are simply getting too large. Run logs for extensive test runs may cause the system to run out of memory and organizing a large number of tests in a single suite is difficult. Things may become unwieldy. It may also be desirable to be able to run some of the tests as part of the whole test as well as standalone.

QF-Test provides a set of advanced features that make it possible to split and arrange tests across a set of test suites. Multiple developers can work on separate parts of a test, then coordinate their efforts, merge the Components of their suites and create libraries of shared Procedures.

This chapter first explains the various mechanisms for distributed test development and how they interact. The final section then summarizes these in concise step-by-step instructions on how to approach large testing efforts with QF-Test.

24.1   Referencing nodes in another test suite

It is possible to reference Procedures[(586)] and Components[(823)] in a test suite other than the current one. These references can be explicit or implicit through included files:

- Explicit references use a syntax similar to the one used in URLs to specify an item inside a web page. The referenced suite must be prepended to the Procedure name⁽⁵⁹⁰⁾ attribute of a Procedure call⁽⁵⁸⁹⁾ of the QF-Test component ID attribute of a Component dependent node, separated by a '#' character. The usual `packagepath.procedure` becomes `suite#packagepath.procedure`.

- Implicit references make use of the Include files⁽⁵¹⁶⁾ attribute of the Test suite⁽⁵¹⁵⁾ node. Whenever a node is not found in the current suite, QF-Test will search for a matching Procedure or Component within all the suite's directly or indirectly included files (a file is considered indirectly included by a suite if it is found as an included file within one of the suite's own included files; for example, if suite A includes B, and suite B includes C, then C is indirectly included by A).

A test suite that references a node in another test suite becomes dependent on that suite. If the Name of a Procedure or the QF-Test ID of a Component in the referenced suite changes, the suite with the reference must get updated, otherwise the link is broken and the suite will no longer work correctly. In such cases QF-Test will automatically update references if it knows about them. The best way to ensure that is to have both test suites in a common project because QF-Test automatically tracks all includes and all explicit references within a project. Alternatively you can list the calling suite in the Dependencies (reverse includes)⁽⁵¹⁷⁾ attribute of Test suite root node of the referenced suite.

While implicit references are more convenient in most cases, they can make tests harder to understand because it is not immediately obvious where the Procedure or Component referenced by some node is actually located. One possibility to find out is to select "Locate procedure" (Ctrl-P) or "Locate component" (Ctrl-W) from the context menu. Additionally, QF-Test provides the menu items Operations→Make references explicit and Operations→Make references implicit which let you toggle quickly between the two modes without changing the actually referenced nodes.

In both cases, the referenced suite can either be given a relative or absolute filename. Relative filenames will be resolved relatively to the directory of current suite, or - if that fails - relatively to the directories on the library path (see option Directories holding test suite libraries⁽⁴³⁶⁾). Always use the forward '/' as the directory separator, even under Windows. QF-Test will map it to the correct character for the system it runs on. This keeps your test suites independent from the operating system.

Note Your Package and Procedure names as well as Component QF-Test IDs should not contain any '\' or '#' characters. If they do, you need to include an escape character in the Procedure call or the QF-Test component ID attribute. See section 47.5⁽⁹¹²⁾ for details about escaping and quoting special characters.

When choosing the Procedure for a Procedure call or the Component for some event in the dialog, QF-Test offers a selection of all currently opened test suites. If a Procedure or Component from another test suite is selected, QF-Test automatically creates the correct

reference, taking included suites into account. When the test is run at a later time, the referenced test suite is loaded automatically if necessary.

During execution QF-Test keeps a stack of currently executing suites. Whenever a Procedure is called in another suite, the called suite is pushed on to the top of this stack and removed when execution returns to the calling suite. Whenever during the execution of a Procedure a Window or Component is referenced by its QF-Test ID, QF-Test searches through this stack of suites from the top to the bottom, i.e. first in the test suite of the called Procedure and then in the calling suite, always checking any included files along the way. This process is quite complicated and you should take care to keep your include hierarchies simple. In case you encounter problems anyway, a detailed explanation is given in section 47.6[(913)].

24.2 **Managing** Components

As we have emphasized in chapter 5[(42)], the Components are the essential part of a test suite. If the SUT changes between releases, these will likely be affected most. If changes are so massive that QF-Test cannot adapt automatically, the Components will have to be updated manually. This is why you should try to avoid redundancy in the Component hierarchy of your tests more than in any other part.

Therefore, when splitting your tests across multiple test suites you should try to keep the Components together in one central test suite and include this suite from the other suites. For very large applications you may want to split the Component hierarchy into parts, each related to a separate part of the SUT's GUI.

Maintaining this central Component library is not trivial. The problems that will arise can be resolved with QF-Test as follows:

- When multiple test developers are recording new Components simultaneously, they cannot be integrated immediately into the central suite, because only one user can edit the central suite at a time. Instead, Components must be integrated later by importing them into the central suite when the new tests have stabilized. This is explained in the following section.

- When the SUT changes, Components in the central suite may need to be updated. If this involves changing any Component QF-Test IDs, this will break any references to these Components from other suites. To avoid that, QF-Test must update those references and it will do so, provided that the suites that depend on the central suite are currently loaded, belong to the same project or are listed in the Dependencies (reverse includes)[(517)] attribute of the Test suite[(515)] node of the central suite.

24.3 Merging test suites

test suites can be merged by importing one test suite into another with the ⟦File→Import...⟧ menu item.

You can select the areas of the test suite, which should be imported.

You have to take care about a correct Include/Reverse-Include of your test suites to ensure, that all calls and component references are still valid. See <u>chapter 35</u>[(399)] for details.

24.3.1 Importing Components

During import, all Windows and Components of the imported test suite are integrated into the component hierarchy of the importing suite. Components that already exist are not copied. A QF-Test ID conflict (identical components with different QF-Test IDs or differing components with identical QF-Test IDs) is resolved automatically by changing the QF-Test ID of the imported component.

Afterwards, all Windows and Components are removed from the imported suite. All nodes in the imported suite that referred to these Components are updated accordingly. Ideally, the imported suite should include the importing suite so no explicit suite references will have to be created.

24.3.2 Importing Procedures and Testcases

3.3+

As you can import Components QF-Test also allows to import Procedures, Packages, Dependencies and Test cases as well as Test sets by choosing 'Procedures' or 'Tests' in the import dialog. You should take care about keeping all calls consistent, e.g. in most cases it does not make sense to import Procedures without their required Components.

In case you only want to import one dedicated Procedure or Test case you can use the button 'Detailimport' on the importdialog. Here you can choose any node you want to import separately.

24.4 Strategies for distributed development

There is no single *best way* of test development or organization, but one approach that works well is the following:

• Start with a central test suite that has the functionality needed to start and stop the

SUT and a basic set of Tests and Procedures. This will become your master suite which will contain all Components.

- Make sure that your developers have understood the importance of assigning names with `setName()` and that unique names are assigned consistently where needed. Where `setName()` is not an option, try to implement `ComponentNameResolvers` to achieve this (see section 52.1.7[1023]). You should be able to record and replay sequences without much ado and without "polluting" the Component hierarchy after trivial changes in the user interface.

- Move as much functionality as possible into Procedures, especially commonly-used stuff and the setup and cleanup routines for the SUT.

- To create new tests, start with an empty test suite. Include the master test suite by editing the Include files[516] attribute of the Test suite[515] node of the new suite. Create the Setup and Cleanup nodes to start and stop the SUT by calling the respective Procedures in the master suite.

- Create your tests as required. When recording sequences, the Components of the master-suite will be used if possible. New Components are added to the new suite, so the master suite will not be modified at this stage.

- Where possible, call Procedures in the master suite for common operations.

- When your new set of tests is complete and you are satisfied that they work well, import any required nodes of your new test suite into the master suite. You have to ensure that all new Component nodes that you recorded are imported into the master suite's Component hierarchy in any case. The master suite's existing Components will not be affected by this, so other suites that depend on the master suite will not need to be modified.

- After importing Components you can import all or only the required Procedures into the master suite.

- You now have various options of how to arrange the actual sequences of events and checks that form your tests. In any case it is a good idea to move everything to Procedures and Packages structured after your test-plan. Then the top-level Test set or Test case nodes of the master suite and your new suite will only contain the required hierarchy of Test set, Test case, Test step and Sequence nodes filled with Procedure calls to the actual test cases. Such an arrangement has several advantages:

  - All your tests are structured cleanly.
  - You can easily create different sets of tests with varying complexity and run-time.

- – You have the option to keep the test cases in separate test suites and have the master suite call them. These "test case-libraries" must include the master-suite, so they need not contain any Components themselves. You can organize your tests so that the master-suite will run the whole set of tests, while each separate suite can also be run standalone.

- – The tests can be maintained by several developers as long as modifications to the master suite are coordinated.

- If you decide to keep your new tests in the newly created test suite instead of moving them to the master suite, modify the master suite to tell QF-Test that there is a new test suite that depends on it. To do so, either ensure that both test suites belong to the same project or add the new test suite to the Dependencies[(517)] attribute of the master suite's Test suite node.

- If you need to modify or extend the new test suite later, proceed as before. You can record new sequences as needed. When you are done, merge any newly created Components back into the master suite.

- If your SUT changes in a way that requires updates or adaptions to the master-suite's Component hierarchy, you must coordinate your test developers. Before you start updating the Components, make sure that all suites that directly or indirectly include the master suite belong to the same project as the master suite or are listed in the Dependencies attribute of the master suite's Test suite node. If modifying the Components of the master suite involves any QF-Test component ID changes, QF-Test will update the depending test suites accordingly, so they should not be edited simultaneously by others.

- The file format for QF-Test test suites is XML and thus plain text. As a result, test suites can be managed very well by version control systems like CVS. Changes to some QF-Test component ID attributes of the depending suites can typically be merged with other changes without conflicts, alleviating the need for coordination.

Of course, the above scheme can be extended to have several master suites for testing different parts or aspects of an application. It may be a good idea to ensure that the component hierarchies in these suites don't overlap too much though. This will save you the effort of maintaining all these hierarchies in case the user interface of the SUT changes significantly.

24.5 Static validation of test suites

3.1+

Working in a project over time will cause modifications, refactoring or deletion of steps in your test suite structure, e.g. you may consider renaming Procedures or simply removing them once they are not required anymore.

24.5.1   Avoiding invalid references

In such cases it is quite important that you adapt all references of the according Procedure in order to guarantee that the remaining tests keep running. For this purpose QF-Test automatically updates all references during the process of renaming or moving elements on demand.

If you want to ensure that your created test structure doesn't contain any call of non-existing Procedures anymore, you can also use the "Analyze references" command of QF-Test in order to perform a static validation of your test suite. This command will open a dialog showing all references and whether they are still okay or something is missing.

You can trigger the analysis via a right mouse-click and selecting Additional node operations→Analyze references... or selecting the according entry from the main menu under Operations . This method is also available in batch mode.

Figure 24.1: Result of analyzing references

3.5+ QF-Test also provides features to search through your test suites for duplicate nodes, empty Packages or Procedures or to analyze for nodes having invalid characters in their names.

This kind of static validation is available for Procedures, Dependencies, Test cases, Test sets and Components and their references.

24.5.2 Unused procedures

4.0.3+

During test development it could happen that procedures, which were used in the first version of your tests will not be used in newer versions due to re-factoring of tests. If those procedures won't get deleted immediately they will stay in the test suite and the test suite will grow and grow. Sometimes you could get the feeling that you have too many procedures or that you have lost the overview of your procedures. In order to check for such unused procedures or dependencies in your test suite you can open the context menu via a right mouse click at Test suite or Procedures and select Additional node operations→Find unused callables... . This operation creates a report showing any procedures or dependencies which had been created but haven't been used yet. Now you could decide what you want to do with those.

Sometimes you might simply remove all of those unused nodes immediately via Additional node operations→Remove unused callables .

Chapter 25

Automated Creation of Basic Procedures

3.0+

25.1   Introduction

At the beginning of a typical QF-Test project the tester records the first tests and starts them. After a couple of such recordings and first success stories he notices that only recording or performing copy/paste bares some hidden pitfalls in maintaining the tests. Just think about a possible workflow change in a main panel, then the tester might have to adapt all test cases. That's why we recommend to make use of the modularization concept using procedures and variables as early as possible in a project. For more information about the modularization concept, please see section 8.5[135].

In projects containing a lot of dialogs and graphical components it might be sufficient to split those procedures into component-specific ones, e.g "press button ok" and separate workflow procedures, e.g. "create a vehicle" combining the component-specific steps together. This approach enables the tester to create new test cases very fast. However, he has to put a lot of efforts into creating those basic procedures first.

QF-Test comes with a Procedure Builder, which will create those basic procedures for you. Using the Procedure Builder will drastically decrease the efforts of recording and creating procedures for graphical components. So the tester can solely concentrate on his main focus, i.e. designing the workflow of the test itself and the according test data.

25.2   How to use the Procedure Builder

For creating the basic procedure automatically, you have to perform following steps:

- Start the SUT from QF-Test.

- Navigate to the window or frame you want to create procedures for in the SUT.

- Press the 'Record Procedures' button, select the according menu item in QF-Test or use the Hotkey for procedure recording[(455)].

- Perform a right mouse-click on the respective component in the SUT.

- Select the according recording-mode.

- Stop procedure recording by releasing the 'Record Procedures' button, by deselecting the according menu item in QF-Test or using the Hotkey for procedure recording[(455)].

Now you should be able to find a newly created package in the Procedures[(595)] node of the test suite where you stopped the recording, containing the created procedures for the components. By default this is called `procbuilder`. If the package `procbuilder` already exists, a package `procbuilder1` will be created and so on. When you open the package you will see a set of packages for functions like `check`, `get`, `select`, `wait` and so on. These contain the procedures created for the components you selected when running the procedure builder, apart from the `check` package, where another level with packages for the various check modes is inserted. This is the default structure, which you can adapt to your needs by modifying the definition file for the procedure builder as described in the following section.
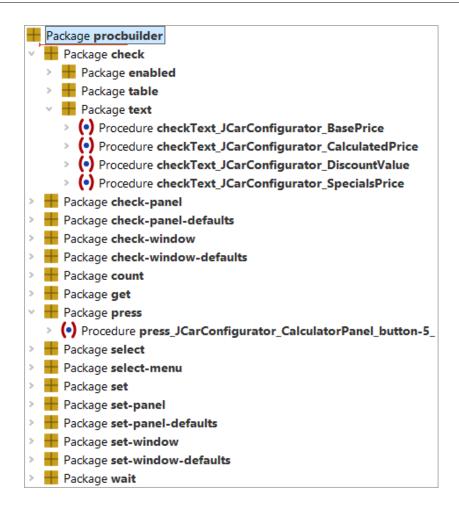
Figure 25.1: Recorded procedures

25.3   Configuration of the Procedure Builder

The act of building procedures is controlled by a template suite, which is located at `qftest-7.1.2/include/procbuilderdef.qft`. This file should be copied to any project-specific location, if you want to adapt it to your project. You can define its location in the options at Configuration file for recorded procedures [(455)].

The template suite contains procedures for the most common GUI elements and actions. If you require other test steps, you can add the according procedure to this test suite.

The file itself is a test suite with a dedicated structure. You can find a detailed explanation of this structure in the subsection section 25.3.1[(310)]. The definition file allows the tester to define procedures for components of dedicated classes or to define procedures

for working with all components of one certain window.

You can find some demo configurations at `qftest-7.1.2/demo/procbuilder`.

25.3.1 The Procedure Builder definition file

The automated creation of basic procedures delivers different procedures depending on the components. A text-field requires a setter procedure for setting its text, a button requires a press procedure for pressing it or a window could require a setter which calls the setter procedures of all text-fields or combo-boxes on that window to call just one procedure for using the window etc..
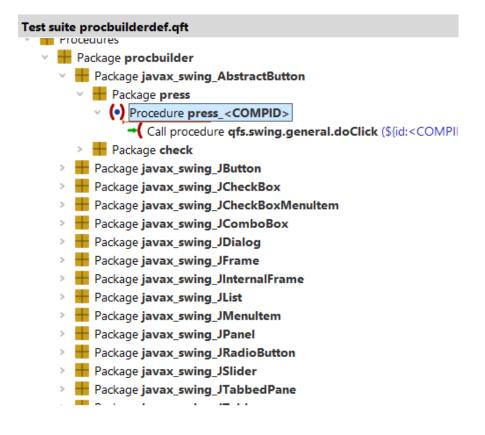
Figure 25.2: The Procedure Builder definition file

The topmost package in the Procedures[(595)] node is the name of the target package for the newly created packages. By default this is called `procbuilder`. This package will be inserted below the Procedures[(595)], when you have finished recording procedures. If the package `procbuilder` already exists, a package `procbuilder1` will be created and so on.

The next level is the class level. Here you can define a package per class. The package

name represents the full class name, but with '_' as separators instead of '.'. That's because '.' is not allowed in package names. The Procedure Builder creates the procedures also for descendants of specified classes. In case the names of your classes contain a '_', you have to mark this via '_'.

The following levels can be chosen freely because those levels are intended to structure the procedures.

At the last level you have to define the steps of the procedure itself.

Of course there are a lot of variable data in that definition, e.g. like <COMPID>.

Using those you can specify variables for the procedure names, like the QF-Test ID of the current component or the component-name. You can also record the current value of the text-field or the current selected status of a checkbox. It's even possible to keep the package structure variable. For an overview of all possible variables, please see chapter 54[(1156)].

Chapter 26

Interaction with Test Management Tools

QF-Test contains some pragmatic test management approaches, like creating a test case overview or documenting test cases within QF-Test. In bigger projects it might be necessary to make use of an own dedicated test management system to track the development status of test cases or to link test cases and their results to defects, use cases or features. Besides support for planning of test cases and tracking their results a test management system could also contain a test execution engine, which supervises the occupation of test systems during different test runs.

As QF-Test doesn't come with all of those features though continuously improving in that area, it is very easy to integrate QF-Test with such a test management or test execution system using the QF-Test Batch mode or the QF-Test Daemon mode. For more information about the Batch mode or the Daemon mode, please see chapter 23[(280)].

The following chapters describe some exemplary solutions which we provide for established test management systems. If you cannot find your test management system in that list, please contact us at support@qfs.de to get hints about a possible integration approach.

26.1 HP ALM - Quality Center

26.1.1 Introduction

The current integration of QF-Test and HP ALM - Quality Center utilizes the built-in VAPI-XP-TEST type of Quality Center.
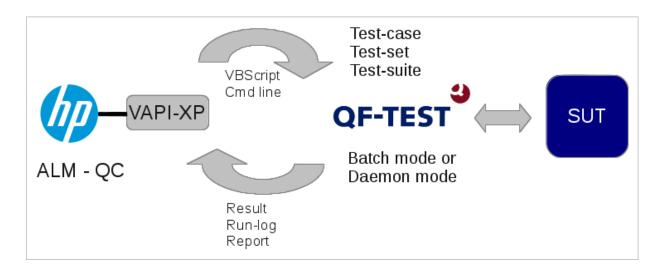
Figure 26.1: Integration with ALM - QualityCenter

The VAPI-XP-TEST type is intended to be an automated test case for any test-tool. QF-Test comes with a template file for the VAPI-XP-TEST script, which is `qcVapiXPTemplate.txt`, see `qftest-7.1.2/ext/qualitycenter`. This script can be used as template for all QF-Test tests in Quality Center. Please see section 26.1.2[(314)] for a detailed step-by-step description.

The QF-Test VAPI-XP-TEST template script employs an external worker VBScript script, called `qcTemplate.vbs`. This script is also part of the QF-Test distribution (see `qftest-7.1.2/ext/qualitycenter`) and has to be adapted to your specific needs. So we encourage you to copy that file to a project specific location and adapt it according to your needs.

The worker script launches QF-Test in batch mode on each test system locally, i.e. it has to be accessible for each test system. As the test suite files and the configuration files have to be available on the test system too, we recommend to put all those files on a shared network drive or into the version management system.

After the execution of the test the run log of QF-Test will be appended to the test instance as well as the status of the test will be set to the result.

You can also change the worker script to make use of a daemon call (for details about the daemon mode, please see chapter 53[(1139)]). In this case QF-Test will establish the network connection to the test system and launch the test by itself. In case of the normal batch call Quality Center establishes the connection to the test system and triggers the local QF-Test installation to perform the test. If you make use of the daemon call, the worker script has to be located on the Quality Center system, but the test suite still needs to be accessible on each test system.

If you do not make use of VBScript in your project, feel free to port the QF-Test demo

scripts to JScript or any other supported language.

The following figure shows the VAPI-XP-TEST test case in Quality Center:
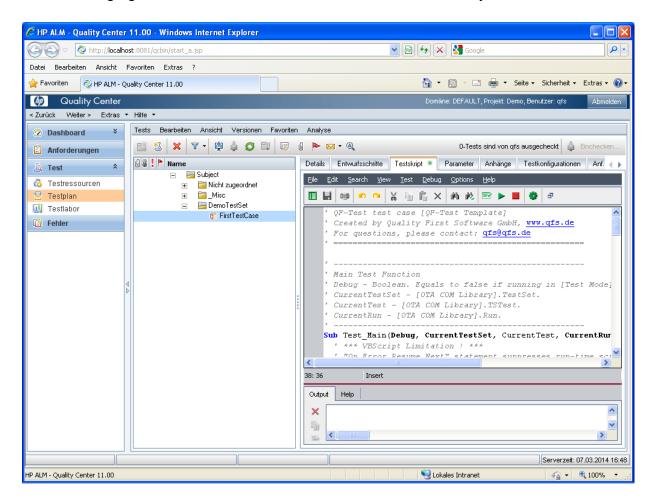
Figure 26.2: QF-Test VAPI-XP-TEST test case in HP ALM - QualityCenter

26.1.2 Step-by-step integration guide

General steps to be performed on the the test system:

1. Copy the template worker script from `qftest-7.1.2/ext/qualitycenter/qcTemplate.vbs` to your project location and rename it to a proper name. We recommend to use the same path on all test systems. Perhaps you should use a shared network drive.

2. Within the worker script you can define certain default option, e.g. whether the batch or daemon mode should be used as default and what should be the name

for the default run log file. This also can be done at a later stage, which might be recommendable when initially starting with the integration process to keep things simple.

Steps in Quality Center to create a test case:

1. Start Quality Center and log in to your project.

2. You might want to create a new test set e.g. called "DemoTestSet" in the "Test plan" area.
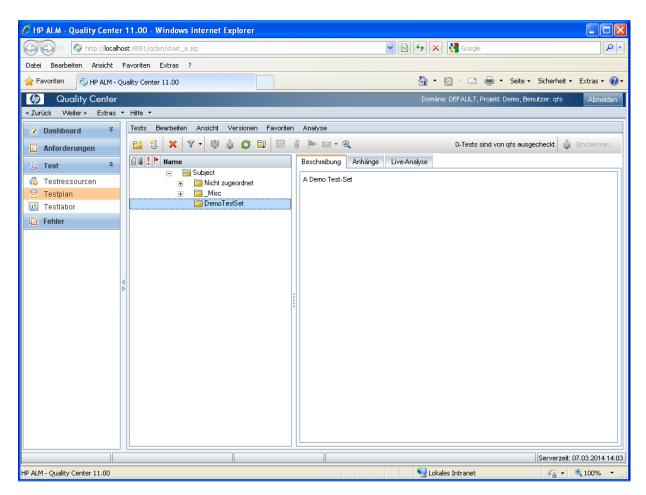
Figure 26.3: In Test plan create new Test set

3. In this test set create a new test with type VAPI-XP-TEST.

Figure 26.4: Create new test of type VAPI-XP-TEST

Note   4. On the HP VAPI-XP Wizard window just press finish without any modifications. (That means you have VBScript as script language and COM/DCOM Server Test as test type).

Figure 26.5: HP VAPI-XP Wizard

5. You will then get a new test as shown below.

Figure 26.6: Test details

6. Change to the 'Test script' tab of the test and copy the content of the template
   file `qftest-7.1.2/ext/qualitycenter/qcVapiXPTemplate.txt` into the
   Script Viewer's text area.

Figure 26.7: Copy template content to script text area

7. Within the script please do following adaptations:

   - Change the `pathToModule` variable to the location you have copied the worker script `qcTemplate.vbs` to.

   - Change the `testSuiteFile` variable to your desired test suite file.

   - If you want to execute one specific test, you can also change the `testCase` variable to the desired test case name.

Please read the comments in the script carefully, because you can also use test case specific settings optionally.

Steps to be performed to run the sample test case

1. Change to the "Test lab" section in Quality Center.

2. You might want to create a new sample test set.
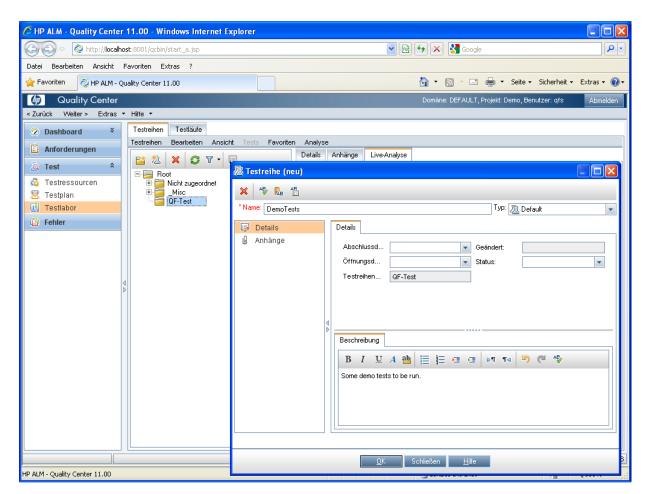
Figure 26.8: New test set in Test lab section

3. Add the test case to the new test sets' execution grid by choosing it from the test plan structure.

Figure 26.9: Add test to execution grid

4. Now you can launch the test case. Ensure the "Run all tests locally" checkbox is activated unless you really have a remote system with a QF-Test environment already set up.

Figure 26.10: Run the test

5. Now Quality Center should start the test run - possibly on your machine, then you should see the SUT coming up after some time, actions being performed and closed at the end. When the run has finished, the result is noted down with the test: Passed or Failed.

Figure 26.11: Test result

6. After the test has terminated, in addition to the result the run log of the test will be uploaded as attachment to the test instance.

7. To view the run log, please double-click to the test in the execution grid, then change to "Runs" and again double-click at the paper-clip attachments symbol for the respective test run.

Figure 26.12: Uploaded run log

26.1.3 Troubleshooting

First of all we need to state that we are not QualityCenter experts. Therefore there might be better and advance options for troubleshooting. Hence we want to at least provide some hints we used so far.

Unfortunately the process output during the test execution in QualityCenter is only visible for a fraction of time, not allowing a direct analysis. Therefore we need to find a work around.

The text editor of the VAPI-XP-TEST node in the "Test plan" area allows to directly execute the script. Then in the output area below the output gets visible permanently showing possibly something helpful.

Figure 26.13: Script debug run

But the direct execution from the script node needs to be handled with care. Of course it is not considered as a real test run, so no run log can be uploaded which results in a respective Run-time error "Object required" in the output. Don't get confused by that!

For more debugging, additional statements like `TDOutput.Print "Some text"` can be added to both the test script and the worker script. By this you can see how far the script runs until a possible error occurs.

The text script editor has a "Syntax check" button which is helpful for validation after every change.

26.2 Imbus TestBench

26.2.1 Introduction

The current integration of QF-Test and the TestBench consists of two parts:

- Creating a QF-Test template file using the TestBench interactions.

- Importing QF-Test results into TestBench.

You can find all required libraries and test suites in the folder `qftest-7.1.2/ext/testbench/Version_1.1_TestBench_2.3`. Please take care to copy all test suites to a project-related folder first and modify them there.

The following section provides a short overview about the integration concept.

26.2.2 Creating QF-Test template from interactions

After planning your tests and designing the interactions in the TestBench, you can create a template QF-Test file using the QF-Test export plug-in for interactions. Imbus will provide all required information, how to install this plugin.

After exporting the interactions you will find all interactions as procedures and their structure as packages in the QF-Test file. Now you can start recording the respective steps in QF-Test and fill the empty procedures.

The completed file has to be saved in your project-specific test suite folder, because this file should be used as input file for the test execution later. We recommend to use a project-specific location, perhaps a shared network drive or the version management system.

26.2.3 Importing test execution results

For running tests you need specific test cases and procedures, which can be found in the provided test suites in the folder `qftest-7.1.2/ext/testbench/Version_1.1_TestBench_2.3/suite`. Please take care to copy all test suites to a project-related folder first and modify them there.

You can also find a demo implementation for the CarConfigurator in the folder `qftest-7.1.2/ext/testbench/Version_1.1_TestBench_2.3/suite/demo`. You have to take care that you need to include the test suite `TestBench_Automation.qft` to your test suite. If you have created procedures via

the iTEP exporting as described in chapter section 26.2.2⁽³²⁶⁾ you will need to include that test suite as well.

The next step is to adapt the configuration for the output files. This can be achieved by modifying the files `testaut.properties` and `user.properties`.

Now you are ready to call the test case `Standalone test executor` from `TestBench_Automation.qft`.

When the test run has been completed, you can import all those results using the iTEP or iTORX import plug-in into the TestBench. The single QF-Test run logs will then be attached to the test-instances.

26.3 QMetry

26.3.1 Introduction

The current integration between QF-Test and QMetry relies on planning the tests and its steps within QMetry and forwarding the actual test execution to QF-Test. Once the the test run terminates the QF-Test run log and its HTML report will be automatically uploaded to QMetry to the respective result area as well as the state of the test case in QMetry will be set to the according result.

You need to prepare your test system in order to run QF-Test tests. Please perform the following steps:

- In the 'Admin' area of the QMetry Testmanagement view install a test execution agent at the 'Agent' view.

- Download the required agent and configuration files to install the QMetry execution agent on your test system.

- Install the respective QF-Test QMetry Launcher at your test-agent.

- Install and set-up a platform at the 'Platform' view, which is also located in the 'Admin' area of QMetry.

- Configure `QMetryAgent.properties` correctly to use the required environment variables of QMetry's QF-Test wrapper.

- Configure `QMetryConfig.properties` correctly to show to the right QF-Test executable.

- Configure additional parameters for the QF-Test call in `QMetryConfig.properties`, see next section for details.

- Launch the QMetry agent. Please do not launch the agent as Windows-Service to avoid running GUI-Tests within the service-session. If you launch the agent as service you should run the QF-Test tests via the QF-Test daemon, which shouldn't run in a service session then.

After setting up the agent and launcher, you need to plan the test execution. QMetry supports several ways of integrating QF-Test test cases. You can find all supported integrations in QMetry's integration guide document. Please perform following steps for a simple integration:

- In the Testmanagement view change to 'Test Cases' and plan the test cases there.

- At the individual test case you have to set the value 'Test Script Name' to the path of the required QF-Test test suite holding the actual implementation of the test case.

- The name of the test case must be exactly the same as the specified value for the QF-Test ID attribute in QF-Test.

- Add the test case to an executable test suite in the 'Test Suites' view.

Now you are ready to run the test cases:

- Open the 'Test Suites' view and select the required test suite for execution.

- Select the 'Execute TestSuite' tab.

- Run or schedule a test run via assigning an agent to the 'Automation' column.

- The next time when the local QMetry agent is polling the QMetry server it will get the necessary information to run the test case.

- Once the test run terminates you will find the run log of QF-Test and its HTML report attached to the 'Execution History' of the executed test suite. The state of the test case will also be updated accordingly.

The following figure shows the 'Execution History' tab in 'Test Suites' holding the run log:

Figure 26.14: QF-Test run log in QMetry

You will find a more detailed description of how to setup QMetry in the manual of QMetry and in QMetry's integration guide document.

26.3.2 Sample Configuration

It's recommended to set following values in the configuration file `QMetryConfig.properties`:

- Set the value of `generic.adapter.success.code` to `0,1`.

- Set `qftest.additional.arguments` to `-test ${QMTestCaseName}` in case of local test execution.

- In case of using QF-Test's daemon set `qftest.additional.arguments` to `-test ${QMTestCaseName} -calldaemon -daemonhost <testsystem> -daemonport <daemonport>` .

As already mentioned in the previous section, you need to use the same name for the test case within QMetry and for the value of the QF-Test ID attribute within QF-Test.

Further ways for integrating QMetry and QF-Test can be found in the integration guide document provided by QMetry.

26.4 Klaros

26.4.1 Introduction

Klaros is a test management tool developed and supported by verit Informationssysteme GmbH, Kaiserslautern, Germany.

Klaros is available in two kinds of editions, a free community edition and an enterprise edition with an extended set of functionality, individual configuration options and full customer support.

The current integration of QF-Test with Klaros comprises:

- Import of QF-Test results into Klaros.

26.4.2 Importing QF-Test results into Klaros

After creating the XML report file as described in chapter 22[(271)], you can upload the results to Klaros. An example for a QF-Test import URL may look like this, where the result file is contained in the HTTP request body.

```
http://localhost:18080/klaros-web/seam/resource/rest/importer?
            config=P00001&env=ENV00001&sut=SUT00001&type=qftest&
            time=05.02.2013_12:00&username=me&password=secret
```

Example 26.1: Importing test results into Klaros

The `curl` command line tool can be used on Linux or Windows/Cygwin to trigger an import in a single command line.

```
curl -v -H "Content-Type: text/xml" -T "my_qftest_report.xml" \
"http://localhost:18080/klaros-web/seam/resource/rest/importer\
?config=P00001&env=ENV00001&sut=SUT00001&type=qftest\
&time=05.02.2013_12:00&user=me&password=secret"
```

Example 26.2: Using `curl` command to import test results into Klaros

Further information can be found within the Klaros online manual at https://www.klaros-testmanagement.com/files/doc/html/User-Manual.Import-Export.html.

26.5 TestLink

26.5.1 Introduction

The current integration of QF-Test with the open-source tool TestLink consists of two parts:

- Generating template test suites for QF-Test from the planned test cases of TestLink.

- Importing QF-Test results into TestLink.

3.5.1+ If you use TestLink 1.9.4 or newer you can use the TestLink API for interacting with TestLink. The TestLink API requires a valid development key. Therefore open TestLink and go to 'My Settings'. In the settings you can generate a development key by pressing 'Generate key' under the 'API interface' section.

For TestLink 1.9.3 or older versions the integration mechanism accesses the database of TestLink directly. This approach requires a JDBC database driver to use the provided scripts. You can download those drivers from the web page of their providers.

Exporting the planned test cases including its test steps from TestLink to QF-Test supports the test-creator to implement the test cases exactly as planned.

Importing the test results into TestLink provides a better overview over all executed manual and automated tests-cases in one tool.

Note Test results can also be uploaded to TestLink without exporting them before. Therefore you have to take care, that the ID of the test case from TestLink is part of the test case's name in QF-Test. The name has to be called like this: `<TestLink-ID>: Name of the test case`.

26.5.2 Generating template test suites for QF-Test from test cases

QF-Test offers the capability to generate template test suites following the same structure as the planned tests in TestLink to guarantee a synchronized structure of automated tests and test planning.

In the QF-Test file you can find one Test case node per test case and one Test set node per suite from TestLink. If you have specified the fields "Steps" and "Expected Results" of a test case, the generating-script will also create an empty Test step for each test step in the according test case. The expected result will be shown in the Comment attribute of the Test step node.

Now the template test suite has to be filled by the test automation engineer with the according steps by adding QF-Test steps to the generated Test step nodes.

3.5.1+ In case you use TestLink 1.9.4 or newer you need to perform following steps:

1. Take care that test automation is enabled in TestLink. Therefore set the respective `enable_test_automation` key to `ENABLED` in the configuration file `config.inc.php`.

2. Copy the folder `qftest-7.1.2/ext/testlink/api` to a project-specific location.

3. Open the launcher script you want to use with a text editor. The launcher scripts are `exportTests.bat` for Windows and `exportTests.sh` for Unix.

4. Adapt the paths of the variables `JAVA`, `QFTDIR` and `TESTLINKINTEGRATOR`.

5. Open the file `TestLinkUserSpecifics.py` with a text editor.

6. Adjust the variables `serverurl` and `devkey`.

7. If you want to export custom fields from TestLink, also adjust the variable `custom_fields`.

8. Run the adapted export script, like shown below.

```
exportTests.bat --testproject projectname
                --targetsuite /path/to/testsuite.qft
```

Example 26.3: Sample call of exporting test cases from 1.9.4

If you use TestLink 1.9.3 or older, please perform those steps:

1. Copy the folder `qftest-7.1.2/ext/testlink/export` to a project-specific location.

2. Open the launcher script you want to use with a text editor. The launcher scripts are `exportTestLinkToQFT.bat` for Windows and `exportTestLinkToQFT.sh` for Unix.

3. Adapt the paths of the variables `JAVA`, `QFTDIR` and `TESTLINKINTEGRATOR`.

4. Open the file `TestLinkDBIntegrator.py` with a text editor.

5. Adjust the variables `dbdriver`, `conncetionstr`, `dbuser` and `dbpass` according to your database connection.

6. If you want to export custom fields from TestLink, also adjust the variable `custom_fields`.

7. Run the adapted export script, like shown below.

```
exportTestLinkToQFT.bat --testproject projectname
                --targetsuite /path/to/testsuite.qft
```

Example 26.4: Sample call of exporting test cases till 1.9.3

26.5.3 Execution of test cases

Executing the QF-Test tests can be performed as usual. But you should create a XML-report at the end of the test run, because the import mechanism is using this report. Therefore you have to use the '-report.xml' parameter during test execution. If you create the reports via the GUI, you have to check the checkbox 'Create XML report'.

Note In case you did not export test cases from TestLink the ID of the test case from TestLink has to be part of the test case's name in QF-Test. The name has to be called like this: `<TestLink-ID>: Name of the test case.`

```
qftest -batch -report.xml reportFolder testsuite.qft
```

Example 26.5: Sample execution to create a XML report

26.5.4 Importing QF-Test results into TestLink

After creating the XML report file, you can upload the results to TestLink.

Per default the import mechanism creates a new build for every test run. The build number of TestLink will be created by the run-ID of the QF-Test report. You can change the run-ID, by setting the parameter '-runid' when launching the tests with QF-Test. But you can also set the '-build' parameter during import to specify a custom build name.

3.5.1+ In case you use TestLink 1.9.4 or newer you need to perform following steps:

1. Take care that test automation is enabled in TestLink. Therefore set the respective `enable_test_automation` key to `ENABLED` in the configuration file `config.inc.php`.

2. Copy the folder `qftest-7.1.2/ext/testlink/api` to a project-specific location. (If you have copied them already for exporting you can use the same files.)

3. Open the launcher script you want to use with a text editor. The launcher scripts are `importResults.bat` for Windows and `importResults.sh` for Unix.

4. Adapt the paths of the variables JAVA, QFTDIR and TESTLINKINTEGRATOR.

5. Open the file TestLinkUserSpecifics.py with a text editor.

6. Adjust the variables serverurl and devkey. (If you have adapted them already for exporting you can use the same values.)

7. Run the adapted import script, like shown below.

```
importResults.bat --testproject projectname
            --resultfile qftestReport.xml --testplan testplanname
        --platform system1
```

Example 26.6: Importing test results into TestLink from 1.9.4

If you want to overwrite the build name you can use the '-build' parameter.>

```
importResults.bat --testproject projectname
            --resultfile qftestReport.xml --testplan testplanname
        --platform system1 --build myBuild
```

Example 26.7: Importing test results into TestLink from 1.9.4 with custom build

If you use TestLink 1.9.3 or an older version, please perform following steps:

1. Copy the folder qftest-7.1.2/ext/testlink/import to a project-specific location.

2. Open the launcher script you want to use with a text editor. The launcher scripts are importToTestLink.bat for Windows and importToTestLink.sh for Unix.

3. Adapt the paths of the variables JAVA, QFTDIR and TESTLINKINTEGRATOR.

4. Open the file ReportParser.py with a text editor.

5. Adjust the variables dbdriver, conncetionstr, dbuser and dbpass according to your database connection.

6. If you want to export custom fields from TestLink, also adjust the variable custom_fields.

7. Run the adapted import script, like shown below.

```
importToTestLink.bat --testproject projectname
                --resultfile qftestReport.xml --testplan testplanname
                --tester tester
```

Example 26.8: Importing test results into TestLink till 1.9.3

Chapter 27

Integration with Development Tools

Automating GUI testing is just one part of the development cycle. Requirements like automating the compilation or build process, running tests, creating documentation or providing a deliverable package led on to a variety of different development tools like IDEs (e.g. `Eclipse`) or build tools (e.g. `make`, `ant`, `maven`) or so called continuous integration systems (like `Jenkins/Hudson`, `Cruise Control`, `Continuum`).

In general, by use of QF-Test's command line interface as documented in chapter 23[(280)] and chapter 42[(862)] a straight forward integration with those tools should be possible.

Note   GUI tests require an active user session. Chapter Hints on setting up test systems[(410)] contains useful tips and tricks to set-up your test systems. In FAQ 14 you can find technical details.

The following sections contain examples for integrations with a some of the tools mentioned above.

27.1   Eclipse

Eclipse (http://eclipse.org) is an Open Source software developer tool for java applications.

QF-Test offers an Eclipse plugin enabling you to start an application directly from Eclipse and run tests on it - anything from whole test sets, single test cases or even just a mouse click.

Video   Video instructions:

 'The QF-Test Eclipse Plugin'
https://www.qfs.de/en/yt/eclipse-42.html

27.1.1 Installation

For the installation please copy the Eclipse plugin file `de.qfs.qftest_7.1.2.jar` from the subdirectory `qftest-7.1.2/misc/` of the QF-Test installation directory to the subdirectory 'dropins' of the Eclipse installation directory. Then (re-)start Eclipse and the plugin will be available.

27.1.2 Configuration of the test nodes

Open the Eclipse menu ‖Run→Run Configurations‖. Enter the QF-Test nodes to be started in the tab 'Main' and if necessary enter parameters in the tabs 'Settings' and 'Initial Settings'. (The tabs 'Environment' and 'Common' are standard Eclipse tabs that are not needed for the configuration of the QF-Test Plugin.)

Then save the configuration by pressing 'Apply'. To start a test run press 'Run'.

Tab 'Main'

Figure 27.1: Eclipse plugin configuration - tab 'Main'

Enter the fully qualified path to the QF-Test executable 'qftest.exe' in the field QF-Test executable, e.g. `C:\Program Files (x86)\qfs\qftest\qftest-4.1.0\bin\qftest.exe`.

'Run configuration to be used as SUT' is an optional entry. You may enter an existing Eclipse 'Run Configuration' for starting the application to be tested. At the start of the application the QF-Test plugin sets up the connection to QF-Test so you can replay or record tests on the application. Use this option when you specify QF-Test nodes in the 'Startup nodes' section which do not start the application themselves. Please be aware that the run configuration to be used as SUT will be started and then right away the listed 'startup nodes' will be executed. So, to make sure the SUT is started when executing the 'startup nodes' the first action of the first 'startup nodes' should be to wait for the SUT. This can be done either by inserting a Wait for client to connect node at the beginning of the first 'startup node' or by adding a first 'startup node' just calling a Wait for client to connect node in QF-Test.

Enter all QF-Test nodes to be executed in the table 'Startup nodes'. You need to specify the QF-Test ID of the node as well as its test suite. Please be aware that the QF-Test ID is a separate attribute of the node and not its name. The QF-Test ID attribute is empty by default and has to be set before use.

Tab 'Settings'

Figure 27.2: Eclipse plugin configuration - Tab 'Settings'

Variables specified in this tab will be read each time before executing the run configuration.

'Path to run log folder' specifies the directory where to save the run logs of the test runs of the run configuration. It is optional. When empty the run logs are saved as configured in QF-Test itself.

If required enter variables to be passed to QF-Test on command line level in the table 'Variables'. This will overwrite default values of the variable.

Tab 'Initial Settings'

Figure 27.3: Eclipse plugin configuration - Tab 'Initial Settings'

The values set in this tab are optional and only read in once before the start of QF-Test When changing them you need to restart QF-Test before they take effect.

Path to license file: Path of the license file to be used.

Path to qftest system config file: Path of the qftest.cfg file to be used.

Path to qftest user config file: Path of the user configuration file to be used.

27.2 Ant

People who are using Apache Ant (http://ant.apache.org) as build system may easily integrate QF-Test in their build file:

```
<project name="QF-Test" default="runtest">
    <property name="qftest"
      location="c:\Program Files\qfs\qftest\qftest-7.1.2\bin\qftest.exe" />
    <property name="logdir" value="c:\mylogs" />
    <target name="runtest" description="Run a test in batchmode">
        <echo message="Running ${suite} ..." />
        <exec executable="${qftest}" failonerror="false"
    resultproperty="returncode">
            <arg value="-batch" />
            <arg value="-compact" />
            <arg value="-runlog" />
            <arg value="${logdir}\+b" />
            <arg value="${suite}" />
        </exec>
        <condition property="result"
    value="Test terminated successfully.">
            <equals arg1="${returncode}" arg2="0" />
        </condition>
        <condition property="result"
    value="Test terminated with warnings.">
            <equals arg1="${returncode}" arg2="1" />
        </condition>
        <condition property="result"
    value="Test terminated with errors.">
            <equals arg1="${returncode}" arg2="2" />
        </condition>
        <condition property="result"
    value="Test terminated with exceptions.">
            <equals arg1="${returncode}" arg2="3" />
        </condition>
        <echo message="${result}" />
    </target>
</project>
```

Example 27.1: Ant build file `build.xml` to execute a test suite

The above example assumes the test suite to be defined as property when running `ant`: `ant -Dsuite="...\qftest-7.1.2\demo\carconfigSwing\carconfigSwing_en.qft"`.

27.3   Maven

People who are using Apache Maven (http://maven.apache.org) as build system may easily integrate QF-Test in their build. This can be achieved by using the antrun plugin of Maven. A demo `pom.xml` file, where QF-Tests tests are executed in the `test` phase

could look like this:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <artifactId>testant</artifactId>
  <packaging>jar</packaging>
  <name>testant</name>
  <groupId>de.qfs</groupId>
  <version>1</version>
  <properties>
   <qf.exe>"C:\Program Files\qfs\qftest\qftest-7.1.2\bin\qftest.exe"</qf.exe>
   <qf.reportfolder>qftest</qf.reportfolder>
   <qf.log>logFile.qrz</qf.log>
   <qf.suite>"c:\path\to\testsuite.qft"</qf.suite>
  </properties>
  <build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-antrun-plugin</artifactId>
            <executions>
                <execution>
                    <phase>test</phase>
                    <configuration>
                        <tasks>
                            <exec executable="${qf.exe}">
                                <arg value="-batch"/>
                                <arg value="-report"/>
                                <arg value="${qf.reportfolder}"/>
                                <arg value="-runlog"/>
                                <arg value="${qf.log}"/>
                                <arg value="${qf.suite}"/>
                            </exec>
                        </tasks>
                    </configuration>
                    <goals>
                        <goal>run</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
  </build>
</project>
```

Example 27.2: Maven build file `pom.xml` to execute a test suite

In your project it might become required to run the tests during another build phase, than the configured `test` phase in the example. In this case you have to configure the plugin accordingly, like described in the Maven documentation.

27.4 Jenkins/Hudson

The video

▷ 'QF-Test Jenkins Plugin'
https://www.qfs.de/en/yt/jenkins-plugin-40.html

shows installation and configuration of the plugin.

Jenkins/Hudson (http://jenkins-ci.org) are continuous integration build tools. They are meant to control and monitor the build process within a software project. One important step in this build process is automated testing.

There are number of benefits to be gained when integrating QF-Test with Jenkins/Hudson:

- In case Jenkins/Hudson is already used for the continuous integration process, integration of automated GUI tests can be easily achieved.

- Easy-to-use administration of scheduled test runs and notification of results via email or RSS.

- Jenkins/Hudson's web based GUI provides good overview and control of test results.

- By use of the HTML Publisher Plugin it is possible to embed QF-Test's HTML reports directly into the Jenkins/Hudson GUI.

- Results generated during the test run such as run logs and reports can be archived automatically. Therefore maintaining an own directory structure is not needed anymore.

As Jenkins and Hudson share a common history, following sections will use Jenkins as representative.

27.4.1 Install and start Jenkins

For GUI tests, Jenkins must not be configured to run as a service but within a real user session. On Windows the `.msi` installer unfortunately directly installs Jenkins as

service without any further inquiry. Please beware of it therefore and ensure Jenkins is started as real user process as described below.

To install Jenkins download the `war` Archive (which can be found here) and start it via `java -jar jenkins.war`.

As soon as Jenkins is started its web interface can be accessed via `http://localhost:8080`. It should look like the following:
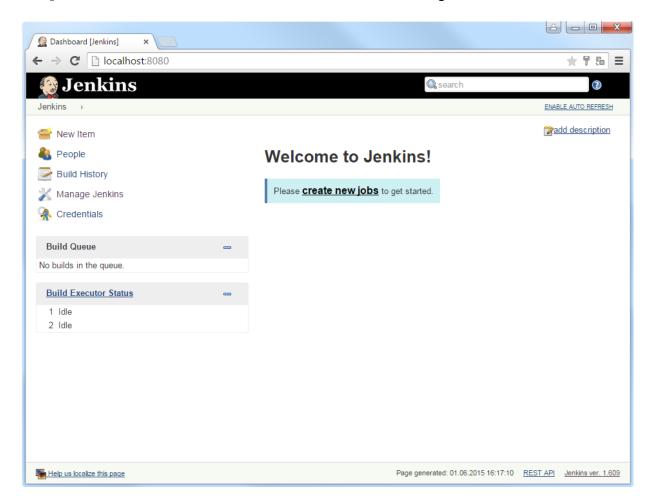
Figure 27.4: Jenkins after start-up.

27.4.2 Requirements for GUI tests

GUI testing requires an unlocked, active desktop. That is the only way to ensure that the SUT behaves the same as if a normal user interacts with it. Chapter Hints on setting up test systems[410] contains useful tips and tricks to set-up the Jenkins/Hudson process.

Jenkins allows execution of tasks on remote machines. This is of course also relevant for

GUI testing. Due to its nature GUI tests are typically not intended to run on the central build server. In addition, tests might need to be executed for different environments, operating systems and SUT versions.

On a remote machine, a Jenkins agent needs to be launched in order to connect to the Jenkins server and wait for jobs to be processed. As described in the Jenkins documentation, there are several options to launch this agent, but for the GUI tests to properly work the only possible launch method is to use Java Web Start.

For GUI tests it is vital to have an active, unlocked user session. Therefore it is not possible to start the agent via a windows service but a real (test) user must be logged in (e.g. via auto login) using Windows Autostart to launch the Jenkins agent. Furthermore screen locking needs to be disabled.

Note Please see also FAQ 14 for more technical background details.

27.4.3 Install QF-Test Plugin

The QF-Test Plugin enables QF-Test to interact with Jenkins. To install the plugin open the Jenkins dashboard and navigate via "Manage Jenkins" to "Manage Plugins". Select the QF-Test Plugin from the "Available" tab. When installing the QF-Test Plugin the JUNIT and HTML-Publisher Plugin will also be downloaded automatically, in case they were not already installed. Finally restart Jenkins to complete the installation. Now the QF-Test Plugin will show up under the Installed tab, as shown in Figure 20.2.

Note Jenkins will automatically use the latest installed version of QF-Test. In case you want to use a different version, you can provide its path under the QF-Test section in the Jenkins configuration (Manage Jenkins -> Configure System).

Figure 27.5: Install QF-Test Plugin.

As soon as the QF-Test Plugin has been installed successfully, test execution with QF-Test can be included in the build jobs. A detailed explanation about the configuration of jobs can be found in the QF-Test Plugin documentation at https://www.qfs.de/en/jenkins.

27.5 JUnit 5 Jupiter

7.0+

In chapter 12[(188)] we described how to integrate JUnit tests into a QF-Test test suite, which creates a common run log combining the results from the unit tests with those from the other QF-Test test cases. With the help of the Java annotation `@QFTest.Test` it is possible to go the opposite way and include QF-Test test suites into a Junit 5 test case, integrating the results from the QF-Test test run into the JUnit test results. This simplifies the inclusion of QF-Test test runs into Maven or Gradle builds, as well as software development environments like Eclipse or IntelliJ IDEA.

To do so, extend the test class, which should include the execution of one or several QF-Test test suites, with a method annotated with `de.qfs.apps.qftest.junit5.QFTest.Test`. The method must return an object of the type `de.qfs.apps.qftest.junit5.QFTest`, which is created using the static method `QFTest.runSuite(...)` or `QFTest.runSuites(...)`. If required, this object can be further configured e.g. to include QF-Test options or variables. The

provided methods are documented in the file `doc/javadoc/qftest-junit5.zip` inside the QF-Test installation.

```
import de.qfs.apps.qftest.junit5.QFTest;
import java.io.File;
public class QFTestDemoTest
{
  @QFTest.Test
  QFTest demoTest() throws Exception {
    // Get location of demo testsuite
    final File qftestVerdir = QFTest.getVersionDir();
    final File demo = new File(qftestVerdir,
            "demo/carconfigSwing/carconfigSwing_en.qft");
    return QFTest.runSuite(demo)
                .withVariable("buggyMode","True")
                .withArgument("-verbose")
                .withReportOpen();
  }
}
```

Example 27.3: Example of a JUnit 5 test case including a QF-Test test suite.

To execute the test it is required to include the following libraries from the QF-Test installation into the classpath:

- `lib/truezip.jar`

- `qflib/qflib.jar`

- `qflib/qfshared.jar`

- `qflib/qftest.jar`

If the project is based on Gradle build, you can apply the `de.qfs.qftest` gradle plugin to automatically resolve those dependencies. For more information, refer to the plugin homepage.

```
plugins {
    id 'java'
    id 'de.qfs.qftest' version '1.1.0'
}
repositories {
    mavenCentral()
}
test {
    useJUnitPlatform()
}
```

Example 27.4: Excerpt from a gradle.build file, which calls QF-Test during the JUnit test run.

27.6 TeamCity CI

QF-Test can easily be integrated with TeamCity CI, so that tests are automatically executed by TeamCity CI and test results, run logs and HTML reports can be inspected right through the TeamCity UI.

You can find step-by-step instructions for how to set this us in our blog post Integrating QF-Test with TeamCity in three easy steps .

Chapter 28

Integration with Robot Framework (Preview)

Though fully functional, the Robot Framework integration is still considered a preview feature because there are no pre-defined Robot Framework keyword libraries available for QF-Test yet so you have to create your own (which is quite easy, see below).

28.1 Introduction

Robot Framework is a very popular framework for test automation and robotic process automation (RPA). Based on Python, it comes with a plethora of ready-to-use keyword libraries for many scenarios. Most of the time the decision will be to use either QF-Test or Robot Framework, but there are situations where an integration makes perfect sense: If you have an existing infrastructure based on Robot Framework or testers with in-depth Robot Framework knowledge combined with the need for QF-Test's unique abilities in UI automation.

28.2 Prerequisites and installation

You need a current version of Python 3 installed.

If not already available, Robot Framework can be installed via `pip install robotframework`. Robot Framework version 4 or higher is required.

The integration requires a bridge between Python and Java. JPype serves that role very well. It needs to be installed via `pip install JPype1`.

QF-Test comes with a Robot Framework library called `qftest` that

Robot Framework needs to know about. It is located in the directory `.../qftest-7.1.2/ext/robotframework`. You can either add that directory to your PYTHONPATH environment variable or create a file called `qftest_robot.pth` in the site-packages of your Python 3 installation - i.e `.../python3/Lib/site-packages/qftest_robot.pth` - with just one line, the full path to that directory.

28.3   Getting started

Robot Framework talks to QF-Test via its daemon mode, so you need to start QF-Test with daemon mode enabled as described in chapter 53[(1139)]. For test development it is best to use interactive daemon mode in which you can activate the QF-Test debugger and step through your keywords at QF-Test level in addition to using the debugger of whichever IDE you run your Robot Framework scripts from. So please start QF-Test from the command line with

```
qftest -daemon -daemonport 5454 -keystore=
```

The port 5454 is just an example, choose whatever you like, but make sure you use the same in your robot file as described below.

As explained in the documentation for the `-keystore <keystore file>`[(873)] command line argument, `-keystore=` tells the daemon to use unsecured communication, which speeds up communication setup and should be OK for internal use on your local machine. The third argument to the `qftest` library shown below should be "false" if the QF-Test daemon is started with -keystore= and "true" otherwise.

Before creating your own Robot Framework tests with QF-Test you should try to run the demo robot script provided with QF-Test to ensure that your setup is complete. It is provided in the directory `.../qftest-7.1.2/demo/robotframework`. Please change to that location and run

```
robot carconfigSwing_en.robot
```

The script should launch the Swing Carconfig demo application and perform a few clicks and checks. If you run it several times you'll see another great advantage of this integration: Becaus the application is started via the QF-Test daemon its lifetime is no longer dependent on that of the Python process running the Robot Framework script. Subsequent scripts can make use of the already running application and rely on QF-Test dependencies (section 40.3[(548)]) to ensure a well-defined state.

Preview SmartIDs (see section 5.6[(73)]) are ideal for specifying target components in Robot Framework keyword calls. Unfortunately the leading '#' of SmartIDs introduces a comment in Robot Framework so that it would always need to be escaped which significantly reduces readability. There is a not-yet-public option in QF-Test that makes it possible to

treat every Comment reference automatically as a SmartID if no <u>Component</u>⁽⁸²³⁾ node exist with that ID. Until that option becomes public it can be set at script level as shown in the procedure "use smartids without marker" in the `robot.qft` demo test suite:

```
rc.setOption(Options.OPT_SMARTID_WITHOUT_MARKER, true)
```

28.4   Using the library

As you can see in the file `resource.txt` in the Robot Framework demo directory, the `qftest` library should be initialized as follows:

```
Library     qftest     localhost     5454     false     ${SUITE}
```

The arguments are optional with the first three defining the host and port of the QF-Test daemon to contact and whether to use a keystore or not. The fourth one defaults to `robot.qft` and specifies the primary test suite from which to determine the keywords that Robot Framework can use.

28.5   Creating your own keywords

The keywords for Robot Framework are determined by parsing the primary test suite specified as argument in the Library definition of the robot script as well as all test suites included directly or indirectly from that suite.

The @keyword doctag is used to designate a Procedure or an entire Package hierarchy as keywords. Details are explained in <u>section 60.2</u>⁽¹²¹⁷⁾.

Chapter 29

Keyword-driven testing with QF-Test

29.1 Introduction

The concept of keyword-driven testing allows business analysts and testers to describe test cases without any deep QF-Test knowledge. Those test cases can either be described in a meta-language or directly in a testmanagement system. QF-Test will then read and execute those test cases. The implementation of the underlying test steps inside QF-Test needs to be done by engineers having QF-Test knowledge.

Business testers and QF-Test engineers are free in defining the way how to describe test steps. They can define a wide range of keywords like simple action-related keywords (e.g. click a button) or more complex business-related keywords (e.g. create an object inside the SUT). Combining test steps into a test case could be achieved by a strict table-oriented approach like clickButton=OK or formulating the test steps in continuous text like "Close the dialog via click on ok".

The following examples illustrate the most popular variants of defining test cases in a keyword-driven manner. The demo test case will create a vehicle in the CarConfigurator demo application of QF-Test.

Option 1: Business-related test steps (see <u>section 29.2.1</u>[355]). You find a sample at `qftest-7.1.2/demo/keywords/simple_business`.

Test step
Launch SUT, if necessary
Open vehicles dialog via Options -> Vehicles
Enter data (name and price)
Press new in order to create the vehicle
Press OK in order to close the dialog
Check creation of vehicle in table

Table 29.1: Test case using business-related keywords

Option 2: Atomic test steps (see section 29.2.2[358]). You find a sample at `qftest-7.1.2/demo/keywords/simple_atomic`.

Test step
Launch SUT, if necessary
Select menu 'Options'
Select menu 'Vehicles...'
Fill text-field 'Name'
Fill text-field 'Price'
Press button 'New' in order to create the vehicle
Press button 'OK' in order to close the dialog
Check table, whether new created vehicle appears

Table 29.2: Test case using atomic keywords

Option 3: Behavior-Driven Testing (BDT) from a technical perspective (see section 29.4.1[363]). You find a sample at `qftest-7.1.2/demo/keywords/behaviordriven`.

Test step
Given SUT is running
Given vehicles dialog is opened
When vehicle name is set to <name>
And vehicle price is set to <price>
And button new clicked
And button ok clicked
Then row with <name> and <formatted-price> appears in table
And column model has value <name>
And column price has value <formatted-price>

Table 29.3: Test case with Behavior-Driven Testing from a technical perspective

Option 4: Behavior-Driven Testing (BDT) with business keywords (see section 29.4.2[(366)]). You find a sample at qftest-7.1.2/demo/keywords/behaviordriven_business.

Test step
Given application is ready to enter vehicle data
When vehicle created with <model> and <price>
Then <model> with <formatted price> appears in table

Table 29.4: Test case with Behavior-Driven Testing from a business perspective

In the subsequent sections you will find a more detailed description of those variants of keyword-driven testing with QF-Test. The concept of Behavior-driven testing (BDT) is described in section 29.4[(363)]. From a perspective of QF-Test BDT is just a special variation of keyword-driven testing.

All samples use the CarConfigurator of QF-Test which is also provided by the installation. You can find all samples in the folder qftest-7.1.2/demo/keywords/. In order to show the test planning aspects we provide Excel files containing the required test steps. Of course you can also go ahead and use your testmanagement tool to plan test steps. We are using Excel files as Excel is a very common piece of software and reading Excel file is quite simple in QF-Test.

The section keywords with dynamic components (see section 29.3[(359)]) describes how to use QF-Test just as test-executor without really recording steps within QF-Test. You need to interact with QF-Test only in rare cases when applying this scenario.

Please take care to copy all test suites to a project-related folder first and modify them there.

29.2 Simple Keyword-driven testing with QF-Test

The simplest way of using keywords is to use existing procedures. Procedures can be designed as business-related procedures or as atomic component-oriented procedures. Business-related procedures perform real workflow in the application, e.g. creating a new vehicle. Atomic component-oriented procedures perform very basic steps like click the ok-button.

29.2.1 Business-related Procedures

As stated in the previous section business-related procedures represent a real business workflow in your application. You can find a sample at `qftest-7.1.2/demo/keywords/simple_business/ SimpleKeywords.qft`. The respective test-plan can be found at `qftest-7.1.2/demo/keywords/simple_business/ simple_keywords.xlsx`. Please take care to copy the demo folder to a project-related folder first and modify them there.

The sample shows the "Create vehicle" test case of the QF-Test CarConfigurator. It consists of following test steps:

1. Launch SUT, if necessary

2. Open vehicles dialog via Options -> Vehicles

3. Enter data (name and price)

4. Press new in order to create the vehicle

5. Press OK in order to close the dialog

6. Check creation of vehicle in table

Let's take a look at the Excel file now:

Figure 29.1: Excel file business-related keywords

As QF-Test can read excel files row by row, we have decided to go for that excel structure. Reading that file follows the data-driven concept (see section 40.4[(562)]). It's also possible to use another structure of the excel file, but then we lose the advantage of using the QF-Test functionality directly without any scripts or if-conditions.

In the first row we find the values `teststep`, `value1` and `value2`. That row will be interpreted as variable names by QF-Test. Every subsequent row will then contain respective values for those variables. This mechanism allows QF-Test to walk through that Excel file in order to execute the planned test steps.

Now let's take a look at the test suite `SimpleKeywords.qft`. The test suite looks like this:

Figure 29.2: Test suite business-related keywords

Node	Purpose
Test set "Scenario"	Represents the container node for the test execution. This node could be omitted theoretically.
Dependency reference dependencies.sutStarted	The concept of dependencies allows QF-Test to manage the starting and stopping process of your application in an intelligent way. Dependencies can contain a strategy in case of any unexpected behavior during test run. See section 40.3[(548)] for details.
Test case "Create vehicle"	This node represents the implemented test case.
Teststep "Walk through test steps"	This node is required to read the excel file with a data driver node.
Data driver "steps from excel"	A data driver reads the content of the Excel file row by row.
Excel file "steps from excel"	Points to the Excel file.
Test step "Generic test step"	This node will be filled with the current test step name from Excel during execution. This approach will create a readable report.
Call procedure "business.$(teststep)"	Here the respective procedure defined in Excel is called. The variable teststep will be filled with the planned data of Excel because of the data driver mechanism.

Table 29.5: Structure of SimpleKeywords.qft

All required procedures are implemented in the package business. In order to allow a simple variable definition any steps in Excel use the variables value1 and value2. Every procedure maps those generic names to the specific parameters of the procedure itself.
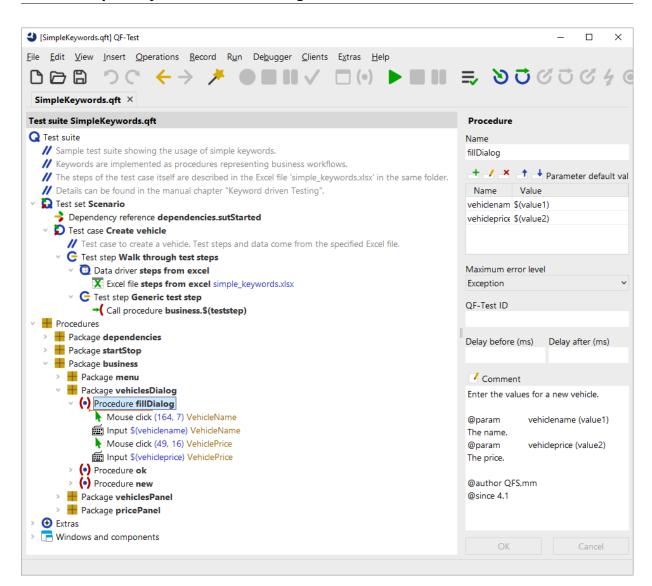
Figure 29.3: Procedure fillDialog

This concept requires that any used keyword has to be implemented in QF-Test already before using it. If your steps require more than two parameters you need to extend the excel file with more columns. In addition you need one test case node in QF-Test per test case in Excel. You can make this more flexible, see section 29.5[(367)].

29.2.2 Atomic component-oriented procedures

Besides the already known business-related procedures representing workflows you can describe every action individually. Applying this concept

results in a very detailed description. You can find the sample test suite at `qftest-7.1.2/demo/keywords/simple_atomic/` `SimpleAtomicKeywords.qft`. The respective test-plan can be found at `qftest-7.1.2/demo/keywords/simple_atomic/` `simple_atomic_keywords.xlsx`. Please take care to copy the demo folder to a project-related folder first and modify them there.

Let's take a look at the "Create vehicle" test case of the CarConfigurator again. The test case now consists of the following steps:

1. Launch SUT, if necessary

2. Select menu 'Options'

3. Select menu 'Vehicles...'

4. Fill text-field 'Name'

5. Fill text-field 'Price'

6. Press button 'New' in order to create the vehicle

7. Press button 'OK' in order to close the dialog

8. Check table, whether new created vehicle appears

Similar to the business-related keywords you need to specify an Excel file containing the planned test steps and you need to create the procedures in QF-Test as well. You can find the implemented procedures in the package `atomic` of the test suite `qftest-7.1.2/demo/keywords/SimpleAtomicKeywords.qft`. In case you choose this approach you can also think about using the automated procedure generation (see chapter 25[(307)]).

The next section describes how to use dynamic procedures. This means that we will still write atomic component-oriented procedures in our test-plan, but there will be no need to create a procedure for each and every step or component. Instead of individual procedures we will create procedures like clickButton or setText. Those procedures will then be re-used every time.

29.3 Keyword-driven testing using dynamic or generic components

The previous section shows how we can apply keyword-driven testing to call various procedures depending on the test-plan. But the graphical components and their recog-

nition still stays in QF-Test and the respective procedures. This approach requires that every procedure needs to be recorded or created before actually running the tests.

However, it's also possible to specify the actual component information directly in the test-plan. This plan should then be interpreted by QF-Test. You can find a sample test suite at `qftest-7.1.2/demo/keywords/generic/Keywords_With_Generics.qft`. The respective test-plan can be found at `qftest-7.1.2/demo/keywords/generic/keywords-generic.xlsx`. Please take care to copy the demo folder to a project-related folder first and modify them there.

This approach depends on the concept of generic component recognition in QF-Test. Generic component recognition allows the user to apply variables to the recorded component information or to move components out of hierarchical component structure. Please see section 5.8[81] for details.

Let's go back to our sample test case. The test case "Create vehicle" looks like this.

1. Launch SUT, if necessary

2. Select menu 'Options'

3. Select menu 'Vehicles...'

4. Fill text-field 'Name'

5. Fill text-field 'Price'

6. Press button 'New' in order to create the vehicle

7. Press button 'OK' in order to close the dialog

8. Check table, whether new created vehicle appears

As you can see the test case follows the same description like in the previous section about atomic keywords.

The Excel file looks like this:

Figure 29.4: Excel file of generic components

The used Excel file contains values like `selectMenu` or `dialog.clickButton` for the `teststep` column. Additionally a new column `target` was introduced. That new variable will be explained later. Like in the previous samples you can find a demo implementation at `qftest-7.1.2/demo/keywords/generic/Keywords_With_Generics.qft`. You can find the respective procedures in the package `generic`. Please take care to copy the demo folder to a project-related folder first and modify them there.

Figure 29.5: Test suite for generic components

Let's investigate the procedure `selectMenu` first. This procedure consists of a mouse-click at the component `GenericMenuItem`. If we analyze that component in the test suite, we see that the attribute `class` is set to `MenuItem`. We also see that the attributes Name and Feature are empty, but there is one entry for the `Extra features` table. This entry has the name `qfs:label` with the state `Must match` and the value `$(target)`. The next attributes `Structure` is empty again and the values for `Geometry` are set to '-'. You can details about the '-' at <u>section 5.8</u>[(81)].

This way of defining a component means that the recognition of that component relies on the variable `target`. The variable itself is used in the extra feature `qfs:label`. That extra feature `qfs:label` represents the best describing text of a component, e.g. the text on a button or a label close to a text-field. The excel file got the column `target` which contains the exact label of the respective target components. This method has been applied to all other components as well.

Another noteworthy aspect is the package `dialog` under the package `generic`. This has been introduced because QF-Test also takes the window or dialog objects into account in order to recognize the graphical components correctly. QF-Test also distinguishes between windows and dialogs. Standard windows which allow the user to work within a second window of the application as well and so-called modal windows preventing the user to work in a second window of your application. In most cases it's simpler to separate those two kinds of windows in several packages. If you want, it might be possible to unify them in one window, but that's not shown in the current samples. By the way, you don't need to separate between those window types if you test web applications as there every component is part of a web-page.

In this section we have seen how to make the component recognition more flexible using variables. Additionally we have created one procedure per action and type of target component. This concept allows us to define all test cases within Excel. The required procedures including the generic components have to be created at the beginning of the project. Of course you can also mix this approach with some recorded procedures. Those recorded procedures can then be used like business-related procedures described in section 29.2.1⁽³⁵⁵⁾)

29.4   Behavior-driven testing (BDT)

Besides the traditional concept of keyword-driven testing a second concept called Behavior-driven testing (BDT) is widely being used. Tools for behaviour driven testing like Cucumber/Gherkin can easily be integrated into QF-Test (please contact Quality First Software GmbH for a description about how to set up and configure BDT with QF-Test). Using this approach allows testers to describe test cases more or less in continuous text and sentences. But the tester needs to follow a predefined vocabulary at the beginning of the sentence. Test cases described like this can be more readable for persons without any knowledge of the test cases. Test cases can be described from a technical perspective (see section 29.4.1⁽³⁶³⁾) or from a business perspective (see section 29.4.2⁽³⁶⁶⁾) like in keyword driven testing. You find samples for both variants in the following sections.

29.4.1   Behavior-Driven Testing (BDT) from technical perspective

Describing a test case from a technical perspective using Behavior-Driven Testing (BDT) uses more or less elementary actions for designing a test case. You can find a sample test suite at `qftest-7.1.2/demo/keywords/behaviordriven/ BehaviorDrivenTesting.qft`. The respective test-plan can be found at `qftest-7.1.2/demo/keywords/behaviordriven/     createvehicle.xlsx`.

Please take care to copy the demo folder to a project-related folder first and modify them there.

The "Create vehicle" test case looks like this if it's described in the BDT manner from a technical view:

1. Given SUT is running

2. Given vehicles dialog is opened

3. When vehicle name is set to <name>

4. And vehicle price is set to <price>

5. And button new clicked

6. And button ok clicked

7. Then row with <name> and <formatted-price> appears in table

8. And column model has value <name>

9. And column price has value <formatted-price>

BDT requires to use the terms `Given`, `When`, `And` and `Then` at the beginning of any sentence. You will find more information about this approach in the testing literature.

QF-Test requires matching procedures for above test steps, so we need to build respective procedures again. It's an established method to divide the BDT-keywords in separate packages. The provided test suite therefore contains the packages `Given`, `When_And` and `Then`.

Figure 29.6: Test suite Behavior-driven testing technical

The provided sample test suite contains all procedures in the respective packages, e.g. a procedure `vehicles dialog opened` inside the package `Given`. In order to prevent annoying typos a Server-script `read test steps and fix typos` formats any steps to lower case and tries to replace multiple blanks by one. This script is called directly before the procedure call of `$(teststep)`.

In order to run the test case on multiple test data the sample was extended.

Of course you can apply the concept of generic component recognition as described in the previous section (see section 29.3[(359)]). To that end you would need to specify a very exact description or implement a script filtering the target components from the test step

itself.

29.4.2 Behavior-Driven Testing (BDT) from business perspective

Describing test cases from business perspective using Behavior-Driven Testing (BDT) requires actions from a user's point of view. So those actions contain several interactions likes mouse-clicks or text-inputs. You can find a sample test suite at `qftest-7.1.2/demo/keywords/behaviordriven_business/ BehaviorDrivenTesting-Business.qft`. The respective test-plan can be found at `qftest-7.1.2/demo/keywords/behaviordriven_business/ createvehicle-business.xlsx`. Please take care to copy the demo folder to a project-related folder first and modify them there.

The "Create vehicle" test case looks like this if it's described in the BDT manner from a business perspective:

1. Given application is ready to enter vehicle data

2. When vehicle created with <model> and <price>

3. Then <model> with <formatted price> appears in table

This approach uses the keywords `Given`, `When`, `And` and `Then` at the beginning of any sentence like the technical one. The provided test suite therefore contains the packages `Given`, `When_And` and `Then`.

Figure 29.7: Test suite Behavior-driven testing from business perspective

In order to prevent annoying typos a Server-script `read test steps and fix typos` formats any steps to lower case and tries to replace multiple blanks by one. This script is called directly before the procedure call of `$(teststep)`.

In order to run the test case on multiple test data the sample was extended.

29.5   Scenario files

Apart from defining single test cases you can also specify the entire test scenario in Excel files or even within your test management tool. In our sample we will go on using Excel to keep things simple. Of course it's also possible to use your testmanagement tool there. For reasons of simplification we have used Excel files again. You can find a

sample test suite at `qftest-7.1.2/demo/keywords/generic_with_scenario/`
`Keywords_With_Generics.qft`. The scenario itself can be found in
`qftest-7.1.2/demo/keywords/generic_with_scenario/`
`scenario.xlsx`. All used test cases are described in a separate excel file,
see `qftest-7.1.2/demo/keywords/generic_with_scenario/`
`keywords-generic-testcases.xlsx`. Please take care to copy the demo folder to
a project-related folder first and modify them there.

The provided scenario consists of two test cases using the concept of generic proce-
dures and components (see <u>section 29.3</u>[(359)]). You can use any other approach, if you
want.

Let's take a look at the scenario Excel file.

Figure 29.8: Excel file as scenario file

The worksheet "Scenario" contains a column "testcase". This value will be used as vari-
able later. Each subsequent row represents a test case name. Those test cases cor-
respond with the worksheets in `keywords-generic-testcases.xlsx`. The work-
sheets "tc_createvehicle" and "tc_createvehicle_cheap" contain the respective test case
description.

How does the test suite look like?

Figure 29.9: Test suite scenario file

Node	Purpose
Test set "Generic samples with scenario"	Represents the container node for the test execution. This node could be omitted theoretically.
Dependency reference dependencies.sutStarted	The concept of dependencies allows QF-Test to manage the starting and stopping process of your application in an intelligent way. Dependencies can contain a strategy in case of any unexpected behavior during test run. See section 40.3[(548)] for details.
Data driver "scenario"	Here we read the required test cases to execute.
Excel file "test cases"	Points to the scenario excel file.
Test case "Generic test case"	This node represents the implemented test case.
Teststep "Walk through test steps"	This node is required to read the test steps from the excel file with a data driver node.
Data driver "steps from excel"	A data driver reads the content of the Excel file describing the test case row by row.
Excel file "steps from excel"	Points to the Excel file describing the test case.
Test step "Generic test step"	This node will be filled with the current test step name from Excel during execution. This approach will create a readable report.
Call procedure "generic.$(teststep)"	Here the respective procedure defined in Excel is called. The variable `teststep` will be filled with the test data of Excel because of the data driver mechanism.

Table 29.6: Structure of Keywords_With_Generics.qft

29.6   Custom test case description

The previous sections show all samples using Excel files. As already mentioned it's also possible to use different file-types, e.g. XML or CSV files. But you can also think about evaluate the result of a web-service. Therefore, you will need to implement Server-scripts reading the required information like test step name, component names or variables for QF-Test. Setting variables can be achieved by the methods `rc.setLocal` bzw. `rc.setGlobal`.

Once those variables have been set it will become necessary to call test case or procedures. Therefore, you can use the methods `rc.callTest` or `rc.callProcedure`. You can find a full API description at chapter 11[(161)].

You can also find a few samples in the provided test suite of the manual tester (`qftest-7.1.2/demo/manualtester`) or in the integration of the imbus TestBench (`qftest-7.1.2/ext/testbench`).

29.7 Adapting to your software

All examples make use of the CarConfigurator of QF-Test. You can use those samples as templates to adapt the existing concept to your strategy of keyword-driven testing. The provided samples can only act as templates because of the huge variety of ways of creating applications and the many different testing strategies. They will never serve as out-of-the-box solution without any need of adapting them. In order to find a matching solution to your project you can also get in touch with our support team.

Nevertheless you can find a full sample for the CarConfigurator at `qftest-7.1.2/demo/keywords/ full_sample_for_carconfig`. The sample uses the concept of scenario files as described in section 29.5[(367)].

Technology	Necessary Adjustments
JavaFX	1. Replace the value `awt` with `fx` in the attribute GUI engine on all window components. 2. Perhaps you need to extend the recorded window components with additional variables. 3. You need to adapt the procedure `startStop.startSUT` in order to start your application. Simply copy the created steps from the quickstart wizard. 4. You might possibly have to create some resolver scripts to tune component recognition.
Java/Swing	1. Perhaps you need to extend the recorded window components with additional variables. 2. You need to adapt the procedure `startStop.startSUT` in order to start your application. Simply copy the created steps from the quickstart wizard. 3. You might possibly have to create some resolver scripts to tune component recognition.
Java/SWT	1. Replace the value `awt` with `swt` in the attribute GUI engine on all window components. 2. Perhaps you need to extend the recorded window components with additional variables. 3. You need to adapt the procedure `startStop.startSUT` in order to start your application. Simply copy the created steps from the quickstart wizard. 4. You might possibly have to create some resolver scripts to tune component recognition.
Web	1. Replace the window component with a `Web page` node. 2. As dialogs are already part of the web page you don't need to keep them separated and can create a component within the web page.. 3. You need to adapt the procedure `startStop.startSUT` in order to start your application. Simply copy the created steps from the quickstart wizard. 4. You might possibly have to tune component recognition using the CustomWebResolver concept.

Table 29.7: Necessary adaptions to your SUT

Chapter 30

Usage of QF-Test in Docker Environments

30.1 What is Docker?

Docker is a free virtualization software that makes it very easy to install and run arbitrary applications on physical computers or in the cloud.

Docker was originally developed for the Linux operating system. Docker is now also available for other platforms, including Microsoft Windows and macOS. The virtualization software also runs on cloud services such as Amazon Web Services (AWS) and Microsoft Azure.

Unlike virtual machines, Docker containers are much more resource-efficient, as they do not require the installation of a guest operating system.

30.2 QF-Test Docker Images

Since QF-Test version 6.0.3 official Docker images have been available, which allow to virtualize QF-Test relatively easy. To enable different application scenarios, there are currently 4 different Docker images per QF-Test version on Docker Hub. The exact details and application possibilities for those individual images are also described there.

There is an image with and without preinstalled web browser especially for web tests as well as a variant with or without additional VNC server, which allows the visual control of the test execution as well as possibly necessary debugging of tests.

The provided images only serve as a basis for your own application scenarios and can be extended accordingly by using them as a base image in a Dockerfile.

In February 2023, a special webinar took place about using Docker with QF-Test. After a bit of theory the detailed steps of using the QF-Test images from Docker Hub.

Video   Here you can find the

 special webinar video recording
https://www.qfs.de/en/yt/docker-special-webinar.html

available on our QF-Test YouTube Channel.

Chapter 31

Performing GUI-based load tests

Video:

▷   Load testing
https://www.qfs.de/en/yt/loadtests-5.1.html

31.1 Background and comparison with other techniques

In addition to functional and system tests, QF-Test can also be used to perform load tests, stress tests or performance tests. The idea is to test the performance of some server applications by running a number of GUI clients concurrently.

Performance is measured by running multiple GUI clients in parallel. QF-Test enables you to measure the actual end-to-end response times (the time span from user action until the result shows up). For the following paragraphs we will use the term load testing.

There are many different ways for setting up and performing load tests, most of which are not using real GUI clients. Instead they directly make use of the protocol between the client and server, e.g. by sending HTTP request or performing RMI or other kinds of remote procedure calls.

There are a number of pros and cons for protocol-based or GUI-based load testing:

- Use of resources:
  Protocol-based testing uses very little resources at the client side, so it can easily scale up to the breaking point of the server without requiring too much hardware. GUI-based tests incur the full memory and performance overhead for each client, which can be quite significant, especially in case of Swing or JavaFX based rich clients. In addition, every client creates a GUI and, therefore, a real active user

session is required.

- Efforts creating tests:
  Rich clients typically represent a complex user interface, which correlates with a certain complexity of the client/server API. Creating protocol-based tests that cover most of that API can be quite an effort. On the other hand, GUI-based tests that have already been implemented for functional testing may be available for reuse. If not, it is still much easier to automate complete use cases with QF-Test than at protocol level.

- Measuring response times:
  With GUI-based testing, actual end-to-end response times (the time span from user action until the result shows up) are measured, while protocol-based tests measure only the times for the server call. Either can be useful, depending on the situation.

In some cases it can be quite reasonable to combine both approaches. You can think about running GUI tests on a few systems in order to measure those end-to-end times and in parallel you can trigger protocol-based tests in order to create some load.

In summary, GUI-based load tests can be very useful and efficient - especially if functional tests can be reused - provided that either the number of clients that need to be simulated is not too high, or that sufficient hardware is available for the client side.

At the end of this section here is a overview diagram showing all involved systems.
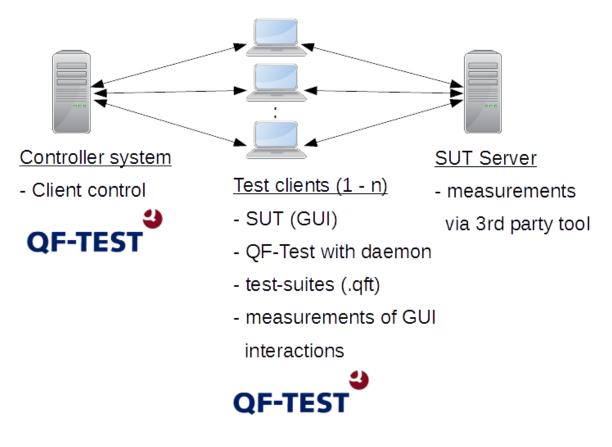
Figure 31.1: Load testing scenario
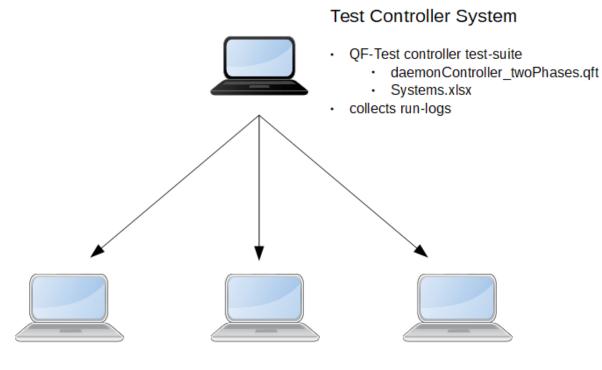
31.2 Load testing with QF-Test

As load testing is a sophisticated subject QF-Test provides a demo tests-suite which can be used as initial point for your project. You can find that demo solution at `qftest-7.1.2/demo/loadtesting/`. This folder contains the following files:

File	Purpose
Systems.xlsx	You can configure which test systems are involved in the test run. Furthermore you can configure global variables for the test run there.
carconfig_Loadtesting.qft	This test suite contains the GUI tests which will get executed on the test systems.
daemonController_twoPhases.qft	This test suite represents the controlling test suite for the entire test run. Using this test suite allows you to launch and co-ordinate the test run on multiple systems.
checkForRunningDaemons.qft	This test suite contains test cases to check for running daemon processes on individual test systems.

Table 31.1: Content of load testing directory

The test suites and files mentioned above can be applied to a load testing project which makes use of multiple test systems. Please take care to copy the demo folder to a project-related folder first and modify them there. The subsequent figure shows an illustration:

Figure 31.2: Overview load testing project

The provided sample test suite for controlling the test run looks like this:

Figure 31.3: Sample test suite daemonController_twoPhases.qft

In order to execute load tests with QF-Test you should take care of the following:

1. Provision of test systems

2. Conception of the test run

3. Preparing test systems prior to the test run

4. Test execution

5. Evaluating results

You can find brief explanations as well as some hints for each item in the following sections.

Note   For tips on how to do parallel website testing with QF-Test read our blog post Running Website Tests in Parallel with QF-Test .

31.2.1 Provision of test systems

You perform load tests with QF-Test via the GUI. GUI tests require an active user session and shouldn't get executed in parallel at the same desktop. That's why we recommend to set-up a virtual or physical system for every client involved. It's indeed possible to run multiple GUI tests in parallel on the same desktop, but this can end up in very subtle problems, e.g. issues with the current focus. That's why running multiple GUI tests on the same desktop is not recommended and should only be taken into consideration in exception.

QF-Test needs to be installed on every system. In addition, the required test suites, the configuration file of QF-Test and necessary test data files need to be deployed to the test systems as well. You can either copy those files to every system locally or you establish a common network share. Furthermore, every test system requires at least a runtime license in order to run the tests. QFS offers to lease such runtime licenses even for a certain period of time.

31.2.2 Conception of the test run

The simplest case is to run the same test on all involved test systems. However, many load testing projects require different sets of GUI tests to be executed. You can think about running tests for various roles of users or user groups. A possible group can represent standard users another some kind of administrator users.

Besides designing the test run for multiple roles load tests are often split into several phases. A phase represents a certain thematic priority. As an example you can divide your project into four phases. The first phase stands for the "Launch" phase. There, the SUT is getting launched on all involved test systems and some initial actions as the log-in can take place. During the second phase 50 clients perform their specific test-scenario. The third phase is performed using 100 clients and the final fourth phase downgrades to 50 clients again. This kind of scaling is also called ramp-up phase (incrementally increasing load) and ramp-down phase (incrementally decreasing load).

Such a conception using several phases increasing the load allows you to test the load capacity of your application in several steps. Like this you will get the information that your application was ok in phase one and problems occurred in the second phase, rather than just a statement about all or nothing.

Using several phases makes sense if multiple roles are the actual focus of your tests. In some cases launching the application on all involved test system can break the environment. So you can think about splitting your project at least into a "launch" phase and "test" phase.

You should create one test suite per role to keep track of your test cases.

Implementation in the sample test suite:

You can find a sample project with two phases in the provided controller test suite `daemonController_twoPhases.qft` The first phase (`Launch Phase` Phase) launches the application. The second phase (`Test Phase`) represents the actual test phase. You can configure the required test suite in the corresponding `Run...Phase` test nodes of each phase.

The provided sample focuses on several roles instead of phases. In case you would like to create a third phase, simply copy the test node `Test Phase` and rename it accordingly.

31.2.3   Preparing test systems prior to the test run

You need to launch the QF-Test daemon before you can start your test run. This QF-Test daemon requires a vacant network port. In order to work effectively we recommend to use the same port on all systems, e.g. 5555.

You can launch the daemon like this:

```
qftest -batch -daemon -daemonport 5555
```

Example 31.1: Launching QF-Test daemon

Please note, that the daemon needs to be started in an active user session. You can accomplish this using tools like the task planer. You can find further details about the daemon at section 23.2[(286)]. Chapter Hints on setting up test systems[(410)] contains useful tips and tricks to set-up the daemon process. In FAQ 14 you can find technical details.

If you want to check whether the daemons are up and running you can either run individual ping commands of the daemon or you run the provided test suite `checkForRunningDaemons.qft`.

```
qftest -batch -calldaemon -ping -daemonhost localhost -daemonport 5555
```

Example 31.2: Ping of QF-Test daemon at localhost

Note On Windows you should use the command `qftestc.exe` instead of `qftest.exe` for every command.

31.2.4   Test execution

During test execution you will need some scripts that will contact the various QF-Test daemons in order to co-ordinate the test run. Such scripts can use QF-Test's daemon

API (see section 53.2$^{(1140)}$) or its command line (see chapter 42$^{(862)}$).

Implementation in the sample test suite:
The provided test suite `daemonController_twoPhases.qft` allows you to run such a load testing scenario and collect the run logs of the test runs afterwards. In the provided Excel file `Systems.xlsx` you can configure which test systems should be involved. That files also contains some variables to organize your tests in roles as described in section 31.2.2$^{(381)}$.

Once all test systems have been correctly configured you can start the test run via running the entire test suite.

Besides the pure execution of such a load testing project you can also meet further requirements. The provided test suites shows samples for the following aspects:

1. Synchronizing the test run on several test systems, see section 31.3.1$^{(383)}$.

2. Measure end-to-end times, see section 31.3.2$^{(385)}$.

31.2.5   Evaluating results

Evaluating results can become quite challenging just because of that huge amount of data. You can analyze the QF-Test run logs as well as the QF-Test reports. Perhaps you receive some measurements at server side or you find a couple of logs which you can analyze by specific tools.

During test execution you can also create custom log-files with QF-Test as described in section 31.3.2$^{(385)}$ for details.

31.3   Hints on test execution

31.3.1   Synchronization

To get consistent results, it may sometimes be necessary to coordinate the tests in the parallel threads, either to make sure that all clients access the server simultaneously, or to prevent just that. Furthermore a role (see previous section) might require all test systems to be in a certain state before running a specific action.

Test runs can be synchronized with the help of a Server script$^{(628)}$ node. That script should contain the following:

```
rc.syncThreads("identifier", timeout, remote=3)
```

`identifier` is a name for the synchronization point, `timeout` is the maximum time in milliseconds to wait for all threads to reach the given synchronization point and `remote` specifies how many systems should wait for that synchronization point.

If the timeout is exceeded without the expected number of threads reaching the synchronization point, a $\underline{TestException}^{(850)}$ is thrown. To log an error instead of raising an exception, set the optional parameter `throw` to 0 (default value 1) or you pack that $\underline{Server\ script}^{(628)}$ step into a Try step.

```
rc.syncThreads("case1", 120000, remote=3, throw=0)
```

You can find a sample implementation in `carconfig_Loadtesting.qft`.
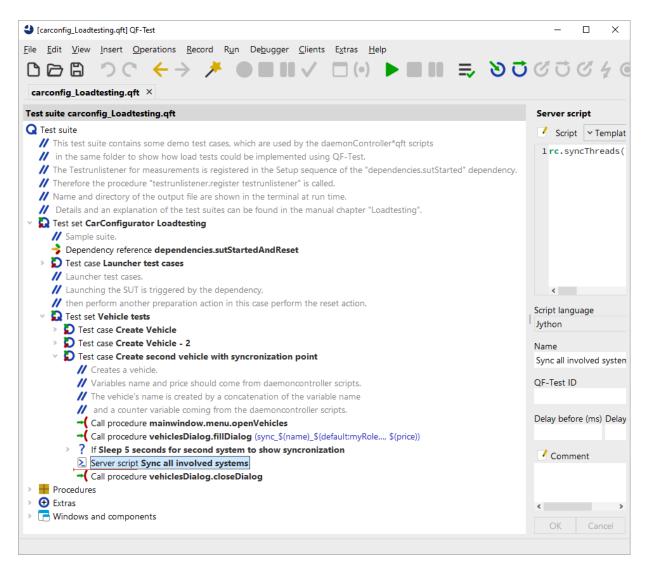
Figure 31.4: Call of rc.syncThreads in demo test suite

31.3.2 Measuring end-to-end response times

It's a very common requirement for GUI tests to measure end-to-end response times.

QF-Test logs those times into its run log. Instead of having to parse that run log in order to retrieve those values you can implement a so-called `TestRunListener` to write a dedicated log file, which just contains the required measurements.

In order to measure the interesting parts, you will need to mark your test steps or sequence using a dedicated keyword. The provided sample implementation uses the keyword `@transaction` for that purpose. If you want to use another keyword, you have to change the code of the provided TestRunListener.

In the provided sample test suite all measurements will be logged into a simple CSV file. That CSV file can be used later for the actual evaluation by another tool. Furthermore, writing that CSV file doesn't brake the test run. If you want to create Excel files or fill databases in order to evaluate the results you should do that after the test run due to performance reasons.

You can find details about the TestRunListener at section 52.7[(1087)]. The sample implementation can be found in `carconfig_Loadtesting.qft`.

The created CSV file looks like this:

```
open vehicles;118;20150921145057;OK
close vehicles;84;20150921145057;OK
```

Example 31.3: CSV file for time measurements

In that CSV file the first value represents the name of the measurement, the second value stands for the duration of the action in milliseconds, the third shows the time when the step was performed, the fourth value shows whether the step was successful.

31.4 Troubleshooting

Due to the complexity of load testing projects you may face issues in several areas.

1. Why are wrong test cases executed?

   Adapt the variable `testsuite` in the respective test case. You can also address a test case directly via `testsuite#testset.testcase`.

2. The QF-Test daemon cannot be started.

   Is the network port vacant? You can check this using the `netstat` command. Here is a sample for the port 5555.

`netstat -a -p tcp -n | findstr "5555"`

`netstat -a --tcp --numeric-ports | grep 5555`

3. The test systems cannot be reached although the QF-Test daemon is running.

   Check whether the QF-Test daemon can be reached on the test-sytem, see section 31.2.1[381]. If the QF-Test daemon is running, please perform following steps:

   (a) Can you reach the QF-Test daemon on the local systems using the ping command, see section 31.2.3[382]?

   (b) Ensure that the daemon or its Java process is not blocked by your system firewall.

   (c) Perhaps there are issues resolving the host name of your test system. Then try to launch the daemon with the additional parameter `-serverhost localhost` or `-serverhost IP address` or `-serverhost <host name>`. In case you use the IP address, please also access that system using the IP-address, otherwise use the host name.

31.5   Web load testing with headless browsers

For loadtesting of web applications you may also use a browser in headless mode. The advantage is the browser does not have a GUI and therefore does not need its own user session. The drawback is that the GUI test then has some restrictions compared to 'normal' browser tests:

- Hard and semi hard mouse clicks as well as drag and drop operations have to be simulated via the browser interface and thus may have a somewhat different behaviour from 'normal' browser tests.

- Screenshots can be generated, but may have a slightly different optic from the one with a normal browser as there is no GUI where the picture could be taken from.

- The application itself has to be runnable on several browser instances in the same user session.

For further informationen on headless browsers please see section 14.7[205].

Chapter 32

Executing Manual Tests in QF-Test

3.0+

32.1 Introduction

QF-Test is primarily a tool for the creation and execution of automated tests. However, it is rarely possible - or economical - to automate 100% of the required tests for a project. In most projects some manual tests need to be performed as well. One of the biggest challenges in testing a project is consolidating the different results and reports of automated and manual testing to get an overview about the execution status of all tests. To facilitate reporting the results of manual test execution along with those of automated testing, QF-Test now offers the capability of tracking manual tests from within itself.

The steps to be performed during a manual test have to be defined in an Excel file which is read by a test suite called `ManualTestRunner.qft`. This test suite is provided along with a sample specification file in the directory `demo/manualtester` below the QF-Test installation directory. The test designer has to specify each step in that Excel file including the expected result. After stepping through the manual tests QF-Test provides the usual results - a run log, HTML and XML reports. Additionally, a newly created Excel file with the results of the respective test run is created. For a detailed description please see section 32.2[(388)].

The dialog used for the test execution is called `ManualStepDialog` and looks like this:

Figure 32.1: Example for a ManualStepDialog

The title of the dialog shows the name of the test case. The detailed step description and the expected result are shown in the first two text-boxes. After performing the test the tester has to specify whether the test succeeded or not. In case the test failed the tester also has to enter the received result which is intended to show the differences between the actual and the expected result. This dialog can also be used for your own purposes, see section 55.1[(1162)].

32.2   Step-by-step Guide

Please perform the following steps on your system to launch a manual test from QF-Test.

- Copy   the   definition   Excel   file   from
  `qftest-7.1.2/demo/manualtester/SampleTestDescription.xlsx` to
  your project location and rename it to a suitable name. We recommend to use the
  same path on all test systems. Perhaps you can make use of a shared network
  drive.

- Also copy the execution test suite from `qftest-7.1.2/demo/manualtester/ManualTestRunner.qft` to your project location. You may want to rename it as well.

- Open the Excel file and define the test steps.

- After saving the changes to the Excel file, open the execution test suite and adapt the global variable `testFile` variable to target your specific Excel file.

- Turn off the QF-Test debugger. It would only interfere with the steps of the manual tester.

- Start the test suite via selecting the test suite node and pressing "Start test run".

- QF-Test will now read the data from the Excel file and open a dialog containing the first test step.

- Enter the result of the test step and proceed with executing each test step.

- At the end of the test execution QF-Test will write a new Excel file containing the test description and the according results. You can also store the run log of that execution or create an HTML report.

Please read the comments in the test suite and Excel file carefully, because you can adapt this concept according to your needs. It is even possible to start only specific tests.

32.3 Structure of the Excel file

The Excel file has a specific structure which allows you to describe the manual test steps quite flexibly. The meaning of the columns is explained in the following table:

Column	Description
TestCase	A unique identifier for each test case. If the step belongs to the same test case as the previous step, just leave this column empty.
Type of Test	Optional definition of the kind or function of the test or step, e.g. a functional test or a usability test, startup, etc.
Comment	An individual comment for the test case. This comment will be shown in the run log of QF-Test.
Short Description	A short description about the content of the test.
Step Description	The detailed description of the manual step.
Expected Result	The description of the expected result of that test step.

Table 32.1: Description of the Excel file for the definition of manual tests

The Excel file with the results of the manual test execution will contain two additional columns as follows:

Column	Description
Received Result	The result the tester received during test execution. If a test step fails, the tester must specify a received result.
State	The state of the test, i.e. PASSED, FAILED, CANCELED or SKIPPED.

Table 32.2: Description of the Excel file with the results of manual tests

32.4 The ManualTestRunner test suite

The `ManualTestRunner.qft` test suite contains some global variables at suite-level which provide fine-grained control over test run. These are explained in the following table. All variables not listed here are used internally by the test suite and should not be changed.

Global Variable	Description
testFile	The path to the test step definition Excel file.
testSheet	The worksheet of the Excel file containing the test steps.
resultSheet	The name of the worksheet for the results.
tests	A list of tests to be intended to execute. If this variable is empty, all tests will be executed. If you want to execute only test 5 and 6, you can specify `5,6` or `5-6`. It is even possible to specify things like: `1,3-5,7` to execute the tests 1, 3, 4, 5 and 7.
defaultState	The default selection of the state. You can set it either to PASSED or FAILED. All other states will be converted to FAILED.
testCaseColumn	The heading of the column containing the test case number.
commentColumn	The heading of the column containing the comment.
shortDescColumn	The heading of the column containing the short step description.
stepDescColumn	The heading of the column containing the full step description.
expResultColumn	The heading of the column containing the expected result.
recResultColumn	The heading of the column containing the received result.
stateColumn	The heading of the column containing the state of the test.

Table 32.3: Description of the global variables in the ManualTestRunner test suite

32.5 Results

An executed test step can be set to one of the following states:

Result	Description
PASSED	The test step was successful.
FAILED	The test step failed.
CANCELED	The test step was canceled.
SKIPPED	The test step was skipped.

Table 32.4: States of manual test execution

Part II

Best Practices

Chapter 33

Introduction

This part of the manual describes best practices based on lessons learned from several customer projects and user feedback. The concepts described should assist you in finding the best strategy for using QF-Test in your projects.

Note   QF-Test is a very generic tool. The hints and experiences described here are just suggestions from our point of view, which we hope will support you in working efficiently and successfully with QF-Test in your project. But they are just one way of doing things and you will have to find your own solution that works best for your specific project.

Chapter 34

How to start a testing project

This chapter talks about the most important aspects that should be considered **before** you start to use QF-Test widely in your testing project. It mostly raises questions and gives general answers with references to more detailed information.

The aim of this chapter is to provide hints about issues which you should take care of in order to make your GUI tests reliable, stable, repeatable and especially maintainable.

34.1   Infrastructure and testing environment

Before you start creating and running automated tests you should think about some general matters pertaining to the environment where the tests have to run. In order to make tests reliable and repeatable you have to take into account that you must be able to bring your SUT into a well-defined state, which includes the state of its backend, e.g. a server and/or a database. If you do not think about such aspects it might become very difficult and sometimes quite tricky to rerun a test or simply to analyze test results and maintenance of tests can become a nightmare.

Please consider the following topics:

1. What is the initial state of your SUT?

    - Which user is the actual user running the tests in your SUT? Most projects work with dedicated test users for running tests. Another approach could be to have one test user per test engineer.

    - Which language setting of your SUT is the primary one? Is it really required to reach a full coverage of all supported languages or is it sufficient to run the bulk of the tests in one primary language and create only a few tests to specifically test localization? In most cases repeating tests in several languages just

covers the same functionality, so you gain no real new information after running them. However, unless you take precautions, the language setting will influence the component recognition of QF-Test, please see <u>section 5.3</u>[(49)] for details.

2. What is the initial state of your database?

   - Can you work with an extra test database or do you have to use a production database? Test databases contain test data with designed and planned content whereas production databases contain real-life data. Is the latter predictable and reliable? What about the danger that tests can mess with and possibly destroy production data? If at all possible you should avoid running automated tests in a production environment.

   - Can you clean up or reset the environment after one test run for rerunning the test? Is it possible to undo changes in the database or is it required to use new test data for the next regression phase?

   - How can you read or write test data? Do you want to use standard SQL scripts or can you reuse libraries from development? Some projects even reinstall the whole database before every test run because they cannot reuse any test data or clean the database correctly.

3. Do you want to integrate QF-Test with other tools, e.g. build tools or test management tools?

   - How to integrate QF-Test with a test management tool? If you can reuse already planned test steps you can avoid redundant work in planning tests and creating them. For the standard integration for such tools, please see <u>chapter 26</u>[(312)].

   - Should test's be launched by a build-tool? If you have created tests you can run them unattended and trigger the run by a build-system like Ant or CruiseControl. Please see <u>chapter 23</u>[(280)] for details about test execution.

   - Should test results be uploaded to a reporting system or into a test management system or is it more sufficient to put the HTML reports and run logs on a centralized HTTP-server?

4. Who will work with QF-Test?

   - Do only one or two engineers work with QF-Test or do all developers and business testers participate in test development? You can find some hints about working in a team with different roles in <u>section 35.5</u>[(403)].

   - What are the skills of the engineers? It is recommended to have at least one dedicated person with a good QF-Test knowledge in the team, who is also capable of implementing scripts and understanding software development principles.

Of course there will be more issues to take care about which are specific for your project. Try to figure them out.

34.2 Location of files

You should also think about following aspects of saving or installing files:

1. Where to install QF-Test to? QF-Test can be installed locally on every system but this forces you to update every system manually whenever you need to upgrade to a new version. You can also install QF-Test on a shared network drive, if your network is reliable, see section 34.2.1[397] for details.

2. Where to store the configuration file `qftest.cfg`? Among other things that file contains information about how QF-Test should recognize components or what should go into the run log. These options have to be the same for every QF-Test user, otherwise you cannot share tests in your team. To ensure that you can either use a shared network installation for QF-Test or specify the config file via command-line parameters when launching QF-Test. Make sure the shared config file is write-protected unless you explicitly want to change it. For details, see section 1.6[10].

3. Where to store the license file `license`? You should put the license file to a central place in order to update the license only once when you receive an update for it. Again, you can either have a shared network installation for QF-Test or can use command-line parameters to specify the location of that file when launching QF-Test. For details, see section 1.5[9].

4. Where to store the test suites? The best place to store test suites is a version management system where you can track the changes and access any version of the files. If this is not possible you should store them on a shared network drive.

5. Where to store the test data files? Test data files are associated with test suites so you should store them closely to the suites, i.e. either in the same version management system or on a shared network drive.

6. Where to store the HTML reports and run logs? You should put those files in a centralized place where any engineer can take a look at them to evaluate the test results. Most people tend to use an HTTP server or a shared network drive for that.

34.2.1   Network installation

If you plan to install QF-Test on a shared network drive you have to take care about some specific things.

The main source of conflict is the system settings file `qftest.cfg`. It is actually a good (and necessary) thing to have all users use the same system settings, especially for the recognition options. Sharing the system settings file facilitates this. However, this file should be made read-only so that one user will not inadvertently change the system settings for everyone. If the file is read-only, QF-Test will not save the system settings upon exit. Any change to these settings will have to be made by explicitly making that file writable, then exiting QF-Test and then making it read-only again. Alternatively each user could specify a different system settings file via <u>`-systemcfg <file>`</u>[881] but that's not advisable.

The running QF-Test instances will also share the log directory (internal logging, not a problem) and the Jython package cache which can occasionally cause problems so that QF-Test cannot initialize its Jython interpreter. This doesn't happen often and can be fixed by clearing (not removing) the Jython cachedir.

For Windows, each user should also execute the `setup.exe` for the primary QF-Test version, located in the installed qftest-x.y.z directory, to get proper registry settings and documentation links on his machine.

In the rare case when a QF-Test patch overwrites existing jar files of QF-Test, running instances based on those jars may crash on Windows.

34.3   Component Recognition

The most important aspect of a GUI testing tool is a stable and reliable recognition of the graphical components. In that area QF-Test is very flexible and can be configured in several ways. In most cases the default configuration for the component recognition works well, but sometimes you may have to change it.

If you change the component recognition options after creating lots of test cases, those test cases may break. Therefore you should try to find the most appropriate settings for your project as early as possible. It is worth spending time in that area before starting to implement a huge amount of tests because in the worst case you might have to re-record or at least update all or most of the existing test cases after a critical change of the recognition options.

Best start by recording some demo test cases and figure out how QF-Test recognizes your SUT's components. The recognition mechanism is described in <u>chapter 5</u>[42] and <u>section 5.9</u>[82]. If you rerun those demo test cases - ideally on different versions of your SUT - and run into recognition problems, you have to ask yourself following questions

about those tests:

1. Are there enough synchronization points, like Wait for component to appear or Check nodes with timeouts to execute test steps only if the SUT is ready for them?

   (a) Sample 1: After opening a window you can only work in that window, if it is really there -> Use a Wait for component to appear node.

   (b) Sample 2: After pressing on a search button, you can only continue with the test when the search is really over -> Use a Check node with a timeout.

Another important aspect besides synchronization points is the correct approach of recognizing components. You have to ask yourself the following questions to determine, which recognition approach might be the most appropriate one:

1. Do the developers use unique and stable names for their components? Please take a closer look at section 40.13[811].

2. Perhaps it is sufficient to use a regular expression for the Feature attribute of the component of the main window under the Windows and components node. Please see section 5.4.3[64] for details.

3. If development did not set useful or even dynamic names it may be required to implement a NameResolver. Please take a closer look at section 5.3[49].

4. Do any of the QF-Test recognition options need to be changed? These are described in section 5.3[49].

5. Is it possible to use generic components? See section 5.8[81] for details.

In some cases it is sufficient to change the default configuration. Let us assume the developers have set unique and stable names for the target components, i.e. buttons, textfields, checkboxes etc. In such cases it may be sufficient to just change the 'Name override mode' setting of QF-Test to 'Name overrides everything'. This setting tells QF-Test to ignore any changes in the component hierarchy and just work with the target components and the window directly.

Note   You have to change this option in two places: Once at 'Record' -> 'Components' -> 'Name override mode' and at 'Replay' -> 'Recognition' -> 'Name override mode'. See section 5.3[49] for more details.

Chapter 35

Organizing test suites

One of the most challenging tasks in a project is keeping the test suites maintainable over a long period of time. Especially if some window or workflow changes significantly the maintenance effort should be kept to a minimum.

It is also worth thinking about how to minimize the creation efforts of tests that contain a lot of similar or even the same steps. A typical use case is launching the SUT or the login process or a very important and basic workflow like navigating to a certain part of the SUT.

Another aspect to think about is how to efficiently organize test suites if different people work in your testing project.

3.5+ In any case you should create your test suites within a QF-Test project as described in chapter 9[(156)]. This feature provides a better overview over your test suites and directories.

The following sections show some best practices how to keep your tests maintainable, extensible and well-organized.

35.1   Organizing tests

In chapter section 8.2[(131)] we describe the concepts of Test set and Test case nodes. A Test case node stands for one dedicated test case and its test data. A typical Test case could be derived from a use case, a requirement or a defect description in your environment, e.g 'Calculate the price for vehicle xyz at 10% discount' for the CarConfigurator application.

Test set nodes are collections of Test sets and Test cases which can be used for organizing test cases, e.g. 'Tests for calculating prices'.

Test step nodes represent the individual test steps of a Test case like 'Open window' or 'Check calculation'.

If you have an external description of the Test case or any other associated information which might be important for it, it is recommended to add an HTML link to it in the Comment attribute of the Test case. You will see that link in the report later. It is even possible to create a separate test documentation using the menu action File→Create testdoc documentation . More details about documentation can be found in chapter 22[(271)].

3.1+   The report and test documentation also contain the Test step nodes which are used in a Test case.

If a Test case consists of lots of Procedure calls or Sequences, you should organize the single test steps in Test step nodes. Those Test step nodes have the advantage that you can really see every significant step in the QF-Test window and also in the report later.

If you want to put several nodes into a Test step you can pack the respective nodes into a Test step via selecting them, perform a right-mouse click and selecting Pack nodes→TestStep .

35.2   Modularization

One of the most important concepts for effective test automation is modularization. Modularization here means placing reusable sequences in a dedicated location and calling these whenever possible. This concept enables you to create a sequence only once and reuse it as often as you require it in your tests without re-recording the same steps all the time. Changes in the SUT that require an update of the tests that rely on such a sequence, e.g. a change to some basic workflow, can then be handled by updating just the procedure in a single location instead of many identical sequences spread all over the test suites.

The modularization concept in QF-Test is implemented via Procedure nodes. Procedures are well described in section 8.5[(135)].

If you have lots of test cases, it is best to have almost every test step as a Procedure and create those procedures up front, if possible. With those procedures in place you can fill your test cases very fast by just adding the respective Procedure call nodes.

In larger projects it is useful to have Procedures at different level, e.g. component specific procedures like 'Click OK' and workflow oriented procedures like 'Create a vehicle'.

35.3   Parameterization

The concept of modularization enables you to maintain test steps at a single location in your test suites. But how to use different test data for different tests?

If you have a Procedure that can be called with different test data, e.g. a typical 'Login' process with name and password or the 'Select accessory' procedure of the CarConfigurator, you can use variables within the QF-Test nodes. Those variables should be used at places where test data is usually being accessed. In most cases variables will be used for Text input nodes or for selections of items in a list or a table or tree nodes.

If a procedure requires variables you should define the required variables in its list of Variable definitions[587]. This is to ensure that you get a list of all required parameters whenever you add a respective Procedure call node for that Procedure to your tests. Some customers even set dummy default values for parameters so they can recognize immediately when a parameter has not been initialized by the calling test.

The next step is to move those variables from the Procedure call either into the Variable definitions[523] section of the Test case node or to put the test data into a Data driver node with a Data table or using an external data source.

The usage of variables and parameters is well described in section 8.5[135]. Parameters can also be created automatically, please see section 8.5.4[138]. You can find more details about the data driver concept for loading test data from a data source in chapter 21[261].

35.4 Working in multiple test suites

Up to now you have read about using the modularization and parameterization concept to avoid unnecessary and redundant work in the creation process of your tests. You should have recognized that those concepts will reduce the maintenance efforts of your test suites by changing or updating only one single sequence instead of several ones. But we still do not know how to organize our work for different test engineers or for a very large project with a lot of GUI elements.

The answer for an effective working organization comes again from the software development area and it is to use several 'libraries' for different areas and different responsibilities.

Importing other test suites into a test suite enables you to follow that encapsulation approach. A typical organization of test suites in your project could look like this:
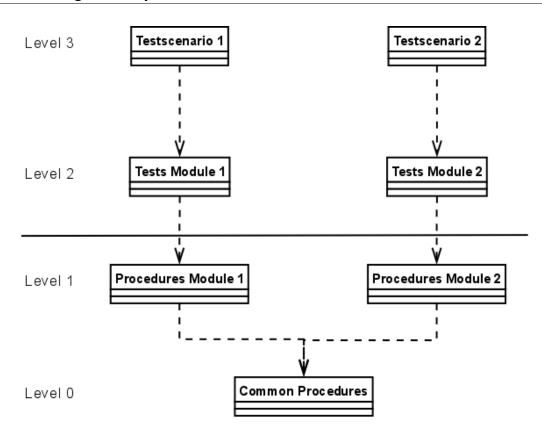
Figure 35.1: Structure of multiple test suites

Level 0 is the level that contains test steps (i.e. Procedures) which are required for nearly all test cases in your projects. Such test steps could be 'Launch SUT' or 'Perform the login'.

Level 1 contains test steps for a specific part of the SUT. For the CarConfigurator you can think about a test suite 'Vehicles' containing Procedures like 'Create a vehicle', 'Remove a vehicle' and another test suite 'Accessories', which contains Procedures like 'Create an accessory' or 'Remove an accessory'.

Level 2 is the test case level. It contains Test cases and Test sets for the respective area of your software, e.g. 'Tests for vehicle creation' or 'Tests for accessory creation'. Of course you could also have a test suite like 'Integration tests' which refers to test steps from different test suites at level 1 and level 0.

Level 3 is the so called scenario level. Those test suites usually just contain Test calls to level 2 and stand for different scenarios within your test project, e.g. 'Nightly test scenario', 'Defect verification scenario' or 'Quick-test build verification'.

Note   The structure described in this document is of course just one possible solution for handling test suites in a project and is not a strict rule you have to follow. You could also think about splitting level 1 into a GUI-element level and a workflow level or merge level

2 and level 3 to one level. Which structure you finally implement also depends on the experience and knowledge of the test-engineers working in your project.

The including area of level 1 test suites looks like this:

Figure 35.2: Including test suites of level 1

You can find a detailed description of how to include test suites in section 24.1[298] and section 47.6[913].

In section 36.5[408] you can find a step-by-step description how to extend an already existing test suite and in section 36.4[407] you can find strategies of handling components in such a scenario.

35.5   Roles and responsibilities

If you take a closer look at the organization shown in the previous section section 35.4[401] you may recognize that it is also possible to organize your test suites based on different knowledge levels of test engineers.
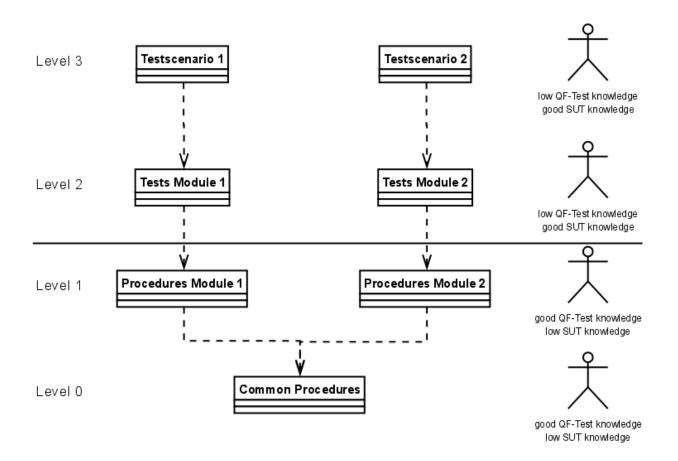
Figure 35.3: Structure of different test suites with roles

Level 0 and level 1 require a good knowledge in working with QF-Test but not a deep knowledge of the SUT. On the other hand level 2 and level 3 require a very good knowledge of the SUT and the planned test cases but those engineers usually do not require a very deep knowledge of QF-Test as long as they just use procedures from level 0 and level 1.

Test engineers working in level 0 and level 1 should be capable of implementing scripts or control structures like the $\mathrm{Try/Catch}$ concept which enables them to create strong and powerful test libraries. At least the engineers working on level 0, but also recommended for engineers working on level 1, should have a good knowledge about component recognition in QF-Test. Please see chapter 5[(42)], section 5.9[(82)] and section 5.3[(49)].

Note Even if you are working alone on a project it is strongly recommended to split the tests and procedures into different levels because maintenance will become easier than with everything kept in one huge test suite.

35.6 Managing components at different levels

If you follow the approach suggested in the previous section (section 35.4[(401)]) you have to define where the components belong. There are two possibilities:

1. Put all components into level 0.

2. Split the components like the procedures into several levels.

Storing all components in level 0 is the most simple solution but this could cause you to update level 0 very often, just because one single component in your project changes. You have to assign responsible persons to keep that structure cleanly.

In big projects you may consider storing the common components like the login dialog or the main frame menus, that are important for everyone, in level 0. Components specific to a certain area, e.g. a dedicated vehicle dialog, appear only in the test suite that holds the procedures operating on those components.

The workflow for moving components between test suites is described in section 36.4[(407)] and the workflow for extending existing test suites is described in section 36.5[(408)].

35.7 Reverse includes

3.5+ You don't need to care about Dependencies[(517)] of test suites belonging to a QF-Test project as described in chapter 9[(156)], because QF-Test automatically resolves dependent suites. So, if you use the concept of projects, you can skip this chapter.

If you work in different test suites in your project you might sometimes want to rename a Procedure or a Test case. If you do that you may encounter some troubles in updating the references to that Procedure or Test case in other test suites. If you want to keep the other files also being updated after renaming such an element you have to maintain the Dependencies[(517)] attribute of the root node of the library test suite.

If you follow the approach described in section section 35.4[(401)], you should ensure that level 0 contains a reverse include to level 1, level 1 should contain one to level 2 and level 2 should contain another one to level 3. A sample from the provided demo test suites is shown in figure 35.2[(403)].

Chapter 36

Efficient working techniques

This chapter will help you in optimizing your working techniques and avoid unnecessary steps when working with QF-Test.

36.1   Using QF-Test projects

The previous chapter describes the creation of several test suites. As you can image the amount of test suites will increase over time. You can get a better overview over them using a QF-Test project.

QF-Test projects show all involved test suites in a very nice way. Furthermore projects automatically take care of propagating modifications to referring test suites. You can find more information about projects at chapter 9⁽¹⁵⁶⁾.

36.2   Creating test suites from scratch

In the previous chapter we described the concept of creating maintainable test suites by utilizing procedures and variables within QF-Test. Normally people start recording very long sequences and splitting those into smaller parts or even procedures later. It is very hard to split up long sequences as you have really to walk through the whole one to find proper boundaries. Another disadvantage is that you cannot see parts which have already been implemented in existing test cases or procedures.

Instead of that described workflow we recommend to plan the tests and their test steps including procedures first. Then you can start recording procedure by procedure. We came to the conclusion that anticipatory recording and creation is very helpful especially for working in bigger teams. A typical workflow for creating those procedures looks like this:

1. Plan the required procedures first.

2. Also plan the required package structure first.

3. Record every procedure as a separate sequence.

4. Rename the recorded sequence like the procedure should be called.

5. Transform the recorded sequence into a Procedure, using the ⎹Transform node into...⎸ action from the context-menu of QF-Test.

6. Move the Procedure to the correct location.

7. Replace test data with variables, either manually or by using the parameterizer (see section 8.5.4⁽¹³⁸⁾)

8. Add the variables to the Variable definitions of the Procedure node, possibly specifying default values.

9. Describe the procedure in its Comment attribute, see section 8.7⁽¹⁵⁵⁾.

An alternative approach of creating procedures is the automated creation provided by QF-Test. This concept is described in chapter 25⁽³⁰⁷⁾.

36.3   The standard library qfs.qft

QF-Test provides the standard library `qfs.qft` which is included per default in every test suite.

This suite contains many useful procedures for accessing components, the file system or a database. Please take a look at that library before you begin implementing something that has already been solved by us.

36.4   Component storage

QF-Test records new components in the test suite where you press the 'Stop recording' button. Therefore it could happen that you record components in the wrong test suite.

If you want to move those components into another test suite you must always use ⎹File→Import⎸ from the target suite. Take care to ensure that both suites belong to the same project or to specify correct 'Include files'/'Dependencies' relations in those test suites. This workflow is described in detail in section 24.2⁽³⁰⁰⁾.

3.1+   For cleaning your component structure in a test suite you can first import the test suite

into itself. Then you can select the Windows and components step, open the context menu via a right mouse-click and click at ‖Mark unused components...‖. You will get a list of all components which are not used in the project. If you are sure that those components can be removed, then select the ‖Remove unused components‖ in the context menu of the Windows and components step.

Note As soon as a Component's comment contains any text it is considered used, even if it has no references.

36.5   Extending test suites

Several workflows can be followed to extend existing test suites:

1. Simply record and extend the target test suite directly.

2. Work with a scratch suite as described in chapter 24[(298)].

If you extend testsuites directly via clicking at 'Stop recording' in that suite, then you have to care about the recorded components. In case you have changed the recorded component hierarchy under Windows and components the newly recorded components will be recorded in the normal hierarchy again and you have to move the new components to the optimized hierarchy. Another aspect is that it could become very difficult for moving single components into another test suite under the right location there.

If you work in a scratch suite you can create the new test steps temporarily in a completely new test suite and import the recorded components and the created procedures and test cases into the target suites together.

A detailed workflow for extending a test suite from level 1 (from section 35.4[(401)]) could look as follows:

1. Create a new test suite.

2. Add the test suite to be extended to the Include files area of the new one.

3. Save the new test suite.

4. Ensure that both test suites belong to the same project or add the new test suite to the Dependencies area of the test suite to extend.

5. Record the new test steps in the scratch suite. Also create Procedures, if required.

6. Then import the components, procedures and tests into the target test suite as described in section 36.4[(407)] and section 24.3[(301)].

A more detailed description of working in multiple test suite can be found in chapter 24[(298)].

36.6 Working in the script editor

The script editor of QF-Test contains some fancy features to avoid too much typing actions.

If you want to call methods of the run context `rc`, you can simply type `rc.` and press Ctrl-Space, then you will get a list of all available methods.

The autocompletion is also working for several variable names, which are:

Variables	Methods
doc	Methods of a DocumentNode.
frame	Methods of a FrameNode.
iw	Methods of the ImageWrapper.
node	Methods of a DOMNode.
Options	The keys and values of QF-Test options to set.
qf	Methods of the qf module.
rc	Methods of the Run context.
resolvers	Methods of the Resolvers module.
Just press Ctrl-Space without typing anything	A list of all variables with autocompletion.

Table 36.1: List of variables with autocompletion.

Chapter 37

Hints on setting up test systems

This chapter provides some hints how to set up your test systems and processes in order to get a stable test execution.

37.1   Using the task scheduler

In order to execute tests or similar processes on a regular basis it is very common to set up a Windows service. Unfortunately those services have the disadvantage that they don't run in an active Windows user session. Because of this such processes should never be started as services, because running GUI tests without an active user session brings up very subtle problems during test execution. You can find further technical details at FAQ 14.

We recommend to define a scheduled task via the task scheduler instead of using services. This can be directly applied via the graphical UI of the task scheduler. The following instruction should work on Windows 7, Windows 8, Windows 8.1 and Windows 10. There might be a few differences depending on the exact operation system:

1. Start the Windows Task scheduler via 'System Control Panel' -> 'Administrative Tools' -> 'Task Scheduler'.

2. Click "Create Task" on the right.

3. At the "General" tab define a "Name", e.g. QF-Test.

4. Now click at the "Change User or Group" button and select the user for the session which should be used to run QF-Test. Do not use the System- or Service session but a real user session.

5. Press OK to close the dialog.

6. Now select "Run only when user is logged in".

7. Do not select "Run with highest privileges".

8. Choose the correct Windows version for your task.

9. Now select the "Triggers" tab. Click "New.." and define the desired time/trigger at "Begin the task".

10. Click "OK" to close.

11. Now select the "Actions" tab, click at "New." and specify "Start a program" as "Action" and "Browse" to the .cmd or .bat file you've just created.

12. Press "OK" to close this dialog.

13. You can now have a look at the "Conditions" and "Settings" tab if you need anything else from your side.

14. If you have finished the setup press "OK" and the task has been created.

It is very important that the user which is configured to run that process is logged in correctly. This user can be logged in manually or automatically (see section 37.3[(412)]). It is recommended to use a virtual system for running GUI tests. On such virtual systems the user needs to be logged in only on the guest system and the host can be locked.

37.2   Remote access to Windows systems

Accessing remote Windows systems via RDP is subject to some restrictions and requires a dedicated configuration of your systems. That's because the implemented RDP server of Windows desktop versions allows only one active user. Thus the (virtual) monitor will be locked as soon as you access the system via RDP. After closing the RDP session the monitor of the test system remains locked. Usually you cannot use a graphical user interface at a locked screen, thus you cannot test it, too.

Note On Windows 10 or Windows Server 2016 systems you can make use of RDP if you modify the Registry. Therefore navigate to `HKEY_CURRENT_USER\Software\Microsoft\Terminal Server Client` or `HKEY_LOCAL_MACHINE\Software\Microsoft\Terminal Server Client` and add a new value `RemoteDesktop_SuppressWhenMinimized` as `DWORD` having the value `2`. Once that setting has been set you are allowed to minimize RDP connections, but you have to keep the connection alive. The tests will still fail if you disconnect or close the session.

Instead of this approach you should use the capabilities your virtual server provides. For VMware server the vSphere client would be the first choice. With VirtualBox you can connect to VirtualBox with RDP (not with the Windows client). Of course this kind of RDP connection has not the impacts on the test system as explained above.

37.3 Automated logon on Windows systems

A simple possibility to get an active user session is to logon a test user automatically after start-up of your system. This chapter describes how to configure your system for that purpose.

Note   An automatic logon at Windows is always a security risk. Therefore you have to ensure that the corresponding test systems are not accessible from outside the test environment.

Although this guidance is generally valid it will be used commonly together with virtual systems which will be accessed remotely. What to keep in mind for this remote access will be explained at section 37.2[411].

The following method is running with Windows 7, Windows 8, Windows 8.1 and Windows 10. There might be a few differences depending on the exact operation system:

1. Start the command line interface with administrator privileges.

2. Enter `control userpasswords2`.

3. Now the dialog 'User Accounts' appears.

4. Remove the check at the checkbox 'Users must enter user name and password'.

5. In the next appearing dialog ('Automatic Logon') enter the user password twice.

6. Click the 'Ok' button to finish the configuration.

Of course also other methods exist to get the same result. Thus you could modify the corresponding registry entry directly. Or you could download the 'Autologon' tool from Microsoft from https://technet.microsoft.com/en-us/sysinternals/bb963905. But all those different methods lead to the same result, which is the modified registry entry. We recommend to use the method explained above as in this way no download is needed and a type mismatch in the registry entry is avoided. By the way, an automated logon will never run for domain users. In fact this would be quite awkward in conjunction with test system. This may be an information which may calm your administrators down.

37.4 Test execution on Unix

On Unix systems you can set up virtual displays using tools like VNC server. A very useful window manager for those displays could be xfce.

Chapter 38

Test execution

This chapter gives some hints about how to implement your tests to get stable and reliable test execution.

38.1 Dependencies

The 'Dependencies' concept of QF-Test provides functionality to guarantee that all pre-requisites for a test case are fulfilled before running it. It is also capable of reacting to unexpected behavior, e.g. closing an error dialog, which pops up and blocks your tests.

The concept is described in section 40.3[(548)] and a use case can be found in the tutorial in the chapter 'Dependencies'.

You should at least implement a Dependency which is responsible for launching the SUT, containing a Setup for launching, a Cleanup for a normal exit and a Catch to react on any unexpected behavior.

Note   If you implement a Cleanup, try to close the SUT normally first and only if the SUT does not terminate correctly, kill it via Stop client.

For SWING and SWT applications please use the procedures `qfs.cleanup.swing.closeAllModalDialogs` and `qfs.cleanup.swt.closeAllModalDialogsAndShells` from the standard library `qfs.qft` for closing unexpected error dialogs.

38.2 Timeout vs. delay

Instead of using the 'Delay before' and 'Delay after' attributes you should try to use QF-Test's synchronization nodes to optimize test execution time.

The first kind of synchronization nodes are the 'waiter' nodes like Wait for component to appear, Wait for client to connect, Wait for document to load and Wait for process to terminate. You can specify the Timeout attribute to wait for a component, process or document. The Wait for component to appear node even provides the functionality to wait for the absence of a component.

The second kind are the 'check' nodes which allow you to specify the Timeout attribute as well. Those nodes can be used to continue the test when a GUI element of your SUT has reached a defined state.

38.3   What to do if the run log contains an error

If the test report contains an error message or exceptions, the following steps should be performed to find the source of that failure very fast:

1. Analyze the run log, especially the screenshots and any other messages.

2. If you cannot find the cause immediately, jump to the failing location in your test suite by typing $\boxed{\text{Ctrl-T}}$ in the run log.

3. Set a breakpoint before or at the failing step.

4. Ensure that the debugger of QF-Test is enabled.

5. Run the failing test.

6. When QF-Test reaches the breakpoint and stops, open the debugger window and check the active variable bindings to wee whether they contain any wrong values.

7. Perhaps at that time you can also see the error immediately in your SUT.

8. If you cannot see any source of that error, run the failing step.

9. If you still encounter errors you might have to re-debug some steps executed before the failing step. Use the 'Continue execution from here' menu entry to jump to previous steps instead of rerunning the whole test again.

3.1+   Since QF-Test version 3.1 it is possible to mark nodes via the context menu item $\boxed{\text{Set mark}}$ or setting bookmarks for specific nodes via the menu item $\boxed{\text{Add bookmark}}$. These features enable you to find important nodes very fast again.

If you encounter problems with component recognition, please see section 5.10[(90)] and section 5.3[(49)].

Part III

Reference manual

Chapter 39

Options

Since QF-Test is a tool that is intended for a wide range of applications, the "one size fits all" approach doesn't quite work. That's why QF-Test has a great number of options that control its functionality.

There are two kinds of options for QF-Test: *user* options and *system* options. User options adjust the behavior of QF-Test's own GUI while system options influence how tests are recorded and replayed. Each user has its own set of user options whereas system options are saved in a common system file. See section 1.6[(10)] for details about configuration files.

3.1+ Many options can have their value changed at run time from a script via `rc.setOption` as described in section 48.5[(917)]. Depending on whether the option takes effect in QF-Test itself or in the SUT, the documentation for those options shows a "Server script name" or "SUT script name" matching the constant from the `Options` class. Obviously the option has to be set in a matching Server script[(628)] or SUT script[(631)] node. Where the option's value can be selected from a drop-down list, the documentation also lists the constants that can be specified as the option's value.

Though the number of options may look daunting, don't let yourself be deterred by it. All options have reasonable default values, so QF-Test works well out of the box for most cases. However, if you find you need to change something or simply want to explore the range of QF-Test's abilities, this chapter is for you.

The options can be set in the dialog available through the menu item Edit→Options... . The settings are saved in two configuration files, one for personal settings and one for system-wide settings (see section 1.6[(10)]).

QF-Test Options

Search for options

∨ General
Projects
Test suites
Display
Editing
Bookmarks
External tools
Backup
Library
License
Updates
∨ Record
Events
Packing
Components
Subitems
Procedures
∨ Replay
Clients
Terminal
Events
Recognition
Delays
Timeouts
Backward compatibility
SmartID and qfs:label
∨ Android
Recording Window
∨ Web
HTTP Requests
Backward compatibility
SWT
Debugger
∨ Run logs
Splitting
Content
Directories
∨ Variables
System
Suite
Command line
Globals

Figure 39.1: Options tree

To get at an option, first select the appropriate node of the tree. The options for that topic are then displayed in the right part of the view. When switching from one group to the other, the current values are verified but not adopted yet. This happens only after confirmation with the OK button.

39.1 General options

This is the node for general QF-Test settings.

Figure 39.2: General options

Ask before closing (User)

When a test suite or a test run log has been modified, QF-Test asks whether it should be saved before it closes its main window. That query can be suppressed by turning off this option. Be warned that auto saving is not implemented yet, so you may lose data if you forget to save before closing.

Ask before overwriting (User)

When you try to save a test suite or a run log or generate a report, pgkdoc or testdoc or save the image of a Check image$^{(729)}$ over an existing file or directory, QF-Test asks for confirmation unless you turn off the option for the respective type of file.

Restore last session on startup (User)

If this option is set and QF-Test is opened in the workbench view, the previous session is restored by loading previously opened test suites and selecting the previously selected node in each suite. If one or more test suites are specified on the command line, this are loaded in addition to the previous session and receive the initial focus on startup.

Number of recent files in menu (User)

The File menu offers quick access to recently used test suites or run logs. This option determines the maximum number of recent file entries in the menu.

Default script language for script nodes (User)

This option can be set to either "Jython", "Groovy" or "JavaScript" and determines the default setting for the Script language$^{(630)}$ attribute of newly created Server script$^{(628)}$ or SUT script$^{(631)}$ nodes.

Default script language for conditions (User)

This option can be set to either "Jython", "Groovy" or "JavaScript" and determines the default setting for the Script language$^{(607)}$ attribute of newly created If$^{(605)}$, Elseif$^{(609)}$, While$^{(600)}$, Test set$^{(525)}$ or Test step$^{(539)}$ nodes.

Literal Jython strings are unicode (16-bit as in Java) (System)

5.3+

Server (automatically forwarded to SUT) script name: OPT_JYTHON_UNICODE_LITERALS
This option defines how to treat literal strings (explicitly specified string constants like ″abc″) in Jython scripts in Server script$^{(628)}$ and SUT script$^{(631)}$ nodes,

Condition[606] attributes in If[605] and other nodes as well as the interactive Jython consoles for QF-Test and the SUT.

If set, it defines that literal Jython strings should be treated as 16-bit unicode strings, just like in Java itself. Otherwise literal strings are 8-bit Python 2 strings that don't integrate well with Java and thus QF-Test. Please see section 11.3.4[175] for detailed information and examples.

If QF-Test encounters an existing older system configuration, the default value in QF-Test is off, meaning 8-bit literal strings. For new installations the option is turned on.

Default character encoding for Jython (System)

Server (automatically forwarded to SUT) script name: OPT_JYTHON_DEFAULT_ENCODING
This option defines the default encoding for converting between Jython 16-bit unicode strings and 8-bit byte strings. It applies to explicit conversions like `str(...)` and to implicit conversions. If the previous option Literal Jython strings are unicode (16-bit as in Java)[420] is unset, implicit conversions include all occurrences of literal Jython strings (explicitly specified string constants like `"abc"`). Please see section 11.3.4[175] for detailed information and examples.

5.3+   Starting with QF-Test 5.3 the default value is "utf-8" (it used to be "latin-1"). Existing system configurations are not be affected by that change.

Use native file chooser on Windows or macOS systems (User)

3.5+

Server script name: OPT_USE_NATIVE_FILECHOOSER
On Windows or macOS systems the native file chooser is more advanced and more convenient to use than the Swing file chooser so QF-Test uses the native one by default. In case you prefer the Swing file chooser you can get it back by deactivating this option.

Show complete file path and QF-Test version in the title bar (User)

4.1.3+

If set, QF-Test shows the full path of the current test suite and the QF-Test version in the title bar of the main window.

39.1.1 Project settings

3.5+

There are several options that influence the way QF-Test manages and displays projects.

Figure 39.3: Projects

Project refresh interval (s) (User)

The interval at which a project automatically gets completely refreshed. You can refresh a directory at any time by selecting it and pressing F5. To refresh the complete hierarchy below the selected directory, press Shift-F5 instead.

Number of test suites to open in one go without warning (User)

From the project tree you can open all test suites contained in one directory hierarchy in one go. If you accidentally select too many test suites, QF-Test will first issue a warning with the number of test suites, allowing you to cancel that action. This option determines the threshold for that warning.

Project files and directories to exclude (System)

In many cases a directory hierarchy holds files and directories that don't really belong to a project, most notably sub-directories created by version control systems like git, subversion or cvs. In this option you can specify patterns for files and directories to generally exclude from projects.

The patterns used here are not regular expressions but a simpler form often used by development tools: An '*' stands for any number of characters up to the next file separator - for compatibility reasons only forward '/' is used - while '**' means 0 or more characters of any kind, including '/'. Every pattern is relative to the root directory of the project. Some examples:

**/.svn
> All directories named `.svn` at any depth.

**/.*
> All directories starting with a '.' at any depth.

deprecated
> A directory named `deprecated` directly below the project root.

39.1.2   Saving test suites

7.0+

The following options specify the format for saving test suites as XML files. The format for saving run logs is determined by the option Save run log in current XML format with UTF-8 encoding[504].

Note   Regardless of the chosen XML format, test suites and run logs can also be opened with QF-Test versions older than 7.0 (although node types introduced later are obviously not recognized). Saving with an old QF-Test version always causes the old XML format to be used.

Figure 39.4: Saving test suites

Overwrite XML format of existing test suites (System)

7.0+

> Changing the XML file format can lead to a lot of purely syntactical changes the next time a test suite gets saved. These changes will show up in version control and possibly hide the real semantic changes. To avoid this, the XML format defined in these options only applies to new files by default. QF-Test will not change the format of already existing test suites unless this option is activated.

Note   Changing the XML format should be a project-wide decision and ideally performed

in one go with all files checked into version control as a single commit with no other changes. Conversion can easily be done using QF-Test in batch mode with the -convertxml[(870)] command line argument as described in section 42.1[(862)].

Use UTF-8 encoding for saving test suites (System)

7.0+

If this option is active (the default), test suites are saved using UTF-8 encoding. Otherwise the encoding is ISO-8859-1.

QF-Test versions older than 7.0 always use ISO-8859-1 encoding.

Number of blanks for indentation when saving test suites (System)

7.0+

XML files that use indentation are easier for humans to read. However, test suites are mostly processed by QF-Test and the only time a typical QF-Test user need to work at that level is when resolving merge conflicts. The latter are reduced by using an indentation level of 0, the new default, because otherwise all lines in the XML will change for nodes that are wrapped into or moved out of a parent node.

QF-Test versions older than 7.0 always use 2 characters for indentation.

Line length for saving test suites (System)

7.0+

The only lines in test suite XML files that can safely be wrapped are those containing attributes of XML nodes. Wrapping actual text, e.g. from scripts or comment attributes would change its meaning so this option does not impose a hard limit for line length.

Unfortunately there is no ideal default value for this option. The current default of 160 is a compromise between the following two extremes:

A negative or extremely large value results in practically unlimited line length, enough to always keep all attributes of an XML node on a single line. This is both compact and good for merging because attribute changes are kept on a single line that also includes the - typically unique - ID of the node.

The value 0 introduces a special format that causes each attribute and even the closing > character to be written on its own line. As a result, line-based version control tools like git blame can show the most recent change of each individual attribute whereas for a long line only the most recently changed attribute is shown. Also, it is easier to interpret the changes in diffs between two versions of an XML file. The downside of this mode is that the context of a change - typically 3 lines - might not include the ID attribute which increases the chance of an incorrect merge.

QF-Test versions older than 7.0 always use a line length of 78 characters.

39.1.3 Display

The following options specify the display of the test suite tree and its nodes.

Figure 39.5: Display

UI theme (User)

This option determines which QF-Test UI theme to use. There is a shortcut for changing this option via the menu $\boxed{\text{View→UI theme}}$.

UI mode - light or dark (User)

This option determines whether the current QF-Test UI theme is displayed in light or in dark mode. The default setting is to follow the setting of the underlying operating system. There is a shortcut for changing this option via the menu $\boxed{\text{View→UI theme}}$.

Paint lines in trees (User)

The option controls whether to display vertical lines between tree nodes of the same indentation.

Syntax highlighting for tree nodes (User)

4.0+

This option controls activation of syntax highlighting for tree nodes within test suites and run logs. If active, specific text parts of nodes (e.g. node name, parameters, client) are outlined in different colors and styles. This significantly improves readability.

Show step types for named tree nodes (User)

7.0+

If this option is deactivated, labels like "Test case" are hidden in tree nodes of test suites and run logs provided the respective node has a name and its icon is unique.

Show script language for script nodes (User)

7.1+

When you activate the option the script language of a Server script[628] or SUT script[631] node will be displayed in the test suite tree.

Show result variables (User)

7.1+

When you activate the option the name of the result variable will be shown in the tree nodes of the test suite and the run log.

Show result values (User)

7.1+

When you activate the option the value of the result variable will be shown in the node of the run log tree. The option Maximum length for values in trees[427] determines the maximum length of the value displayed.

Show client name in tree (User)

7.1+

Use the option to display the name of the client the node relates to in the test suite tree, either "Always" or "Never" or only when the value of the Client attribute is not the default value $(client).

Maximum length for values in trees (User)

7.1+

The option is only relevant when Show result values[(426)] has been activated. It determines the maximum length of the result value displayed in a test suite tree node.

Maximum length for component IDs in trees (User)

7.1+

The option determines the maximum lenght of the QF-Test component ID displayed in the tree node.

Minimum font size (pt) (User)

This option lets you set the minimum font size (as point value) used within QF-Test. A change in this value becomes operative after restarting QF-Test.

Show symbols for tab and line break characters (User)

3.5+

If this option is set, QF-Test shows symbols for tabulator and linebreaks in tables and relevant textareas.

39.1.4 Editing

These options are used to configure various settings regarding editing in the tree or detail view.

Figure 39.6: Editing

Warn when modifying a test suite that cannot be saved (User)

If saving test suites is prohibited, for example when working without a license, QF-Test will warn you that you will not be able to save your changes when a test suite is modified for the first time. Deactivating this option suppresses that warning.

Ask before discarding detail modifications (User)

4.0+

When you have started making changes to an existing or newly inserted node and then abort by pressing (Escape) or clicking the "Cancel" button, QF-Test asks for confirmation before discard your modifications. This dialog can be suppressed by disabling this option. In this case, please be aware that - especially in case of scripts - a lot of work may get lost in case of a mistake.

Ask before implicitly accepting detail modifications (User)

A very common mistake made while editing a test suite is to forget pressing OK after making changes in the detail view of a node before switching to some other node, running a test, etc. If that happens QF-Test can either accept the modified values automatically or ask for confirmation by popping up a dialog with the detail view. The following options are available:

Always
 Don't accept values implicitly, always ask for confirmation.

Only if values are suspect or invalid
 Try to accept values implicitly as long as they are valid and not suspect. Currently "being suspect" is defined as having leading or trailing whitespace which can lead to subtle problems which are very hard to locate.

Never
 Accept all valid values implicitly without asking for confirmation.

This option doesn't change the effect of explicitly discarding your modifications with the Cancel button or by pressing Escape.

Number of undo levels per suite (User)

This option lets you set the number of edits that can be undone in a test suite or run log.

Intelligent scrolling in trees (User)

The default methods for interacting with Swing trees are not ideal. Moving the selection around causes a log of unnecessary horizontal scrolling and Swing has the tendency to scroll trees to a position where little context is visible around the selected node.

Because tree navigation is essential for QF-Test, some of these methods are implemented differently to provide a more natural interface and to make sure that there is always enough context visible around the selected node. However, your mileage may vary, so if you don't like the alternative methods you can switch back to the default Swing way of things by deactivating this option.

Check references before deletion (User)

3.4+

If this option is set, QF-Test searches for references of nodes before nodes will be deleted. If references can be found, they will be shown in a dialog.

Ask after changing QF-Test component IDs or use default (User)

3.5.3+

If this option is set, QF-Test asks whether the user wants to update the QF-Test component IDs of any referring node after the QF-Test ID of a component has been changed. If this option isn't set QF-Test updates all references in case of unique QF-Test component IDs.

Ask after changing callable names or use default (User)

3.5.3+

If this option is set, QF-Test asks whether the user wants to update the callable names (i.e. procedures, packages, tests and dependencies) of any referring node after the name of a callable node has been changed. If this option isn't set QF-Test updates all references in case of unique names.

Check if configuration files are writable (User)

4.1.2+

If this option is set, QF-Test checks whether the configuration files have writing permissions once opening the 'Options' dialog. If one configuration file has no writing privileges, QF-Test will show a message.

Automatically open created nodes (User)

6.1.0+

Automatically open nodes that just have been created.

39.1.5 Bookmarks

Here you can edit your bookmarks, a list of files and nodes that can be accessed quickly via the menu ⟦File→Bookmarks⟧.

4.0+

Instead of a file you can also specify a directory. When the respective bookmark is selected, the file selection dialog is opened directly for this directory. The QF-Test ID for the node is ignored in this case.

Figure 39.7: Bookmarks

Though you can also create new bookmarks manually, it is preferable to use the menu item ⟦File→Add to bookmarks⟧ to add a bookmark for a whole test suite or run log or

to select ⟦Add to bookmarks⟧ in the context menu of a node in a test suite to add a bookmark for this specific node.

39.1.6 External tools

The following options determine which external programs are called by QF-Test.

External tools

External editor command

Directory passed temporary files to external editor

☑ Warn when running external editor without file name

☑ Warn when test suite is changed by an external editor

External imaging program

HTML browser

PDF reader

Figure 39.8: External tools options

External editor command (User)

Scripts can be edited in an external editor by pressing ⟦Alt-Return⟧ or by clicking the ✎ button above the text area. The contents of the text area are then saved to a temporary file and the external editor is run to edit that file. It is recommended to define a name for the script before opening it in the external editor (see also Warn when running external editor without file name[432]). Otherwise a random number is chosen as file name, which makes it difficult to distinguish several scripts opened in the external editor.

Changes made to an external file are picked up automatically by QF-Test. Depending on your settings, you may get a warning message when this happens (see

Warn when test suite is changed by an external editor[(432)]). In case you are tempted to edit your script code parallel in the internal QF-Test editor: These changes are also saved in the temporary file. Editors like jEdit on their part are smart enough to detect the change and reload the file automatically.

This option determines the external editor command to use. There are two variants, the plain name of an executable file or a complex command including options. The latter is distinguished by the string $(file) which is the placeholder for the name of the temporary file. Additionally, $(line) may be used to pass the current line number to the editor as well.

The $(file)/$(line) syntax is used simply to avoid yet another different convention for variable attributes. No standard QF-Test $(...) variable expansion is taking place.

Plain commands need never be quoted. Examples are:

- emacsclient
- notepad
- C:\Program Files\Crimson Editor\cedt.exe

Complex commands on the other hand may need to use quotes, especially on windows. QF-Test takes care of quoting the $(file) argument itself:

- "C:\Program Files\eclipse-3.6\eclipse.exe" –launcher.openFile $(file)
- javaw.exe -jar C:\Programs\jEdit4.2\jedit.jar -reuseview $(file)
- "C:\Program Files\Crimson Editor\cedt.exe" $(file)
- xterm -e vi +$(line) $(file)

If this option is left empty, the value of the environment variable EDITOR is used, if it is defined when QF-Test is started.

Directory passed temporary files to external editor (User)

This option can be used to change the directory in which QF-Test saves temporary files for opening in an external editor (see External editor command[(431)]). If empty, the user configuration directory[(10)] is used.

Warn when test suite is changed by an external editor (User)

Display a warning message when changes to a script made by an external editor are picked up by QF-Test (see also External editor command[(431)]).

Warn when running external editor without file name (User)

Display a warning message when a script without name is opened in an external editor (see also External editor command[(431)]).

External imaging program (User)

The Image[(731)] of a Check image[(729)] node can be edited in an external imaging program. The image is saved to a temporary PNG file and the external imaging program is run to edit that file. When finished editing, the file must be saved and the program exited. QF-Test will read the image back from the temporary file.

This option determines the program to use for the operation. There are two variants, the plain name of an executable file or a complex command including options. The latter is distinguished by the string $(file) which is the placeholder for the name of the temporary file.

Note The $(file)/$(line) syntax is used simply to avoid yet another different convention for variable attributes. No standard QF-Test $(...) variable expansion is taking place.

Plain commands need never be quoted. Examples are:

- gimp
- mspaint
- C:\Windows\System32\mspaint.exe

Complex commands on the other hand may need to use quotes, especially on windows. QF-Test takes care of quoting the $(file) argument itself:

- gimp –no-splash $(file)
- "C:\Windows\System32\mspaint.exe" $(file)

HTML browser (User)

This option allows you to set the HTML browser used to open HTML pages (e.g. report files or the context sensitive help). You can specify a complex command using '$url' as placeholder for the URL to show, e.g.

```
netscape -remote openURL($url)
```

or just a simple command like

```
firefox
```

in which case the URL is passed as the last argument. If the option is empty the system browser is used.

PDF reader (Unix only) (User)

This option allows you to set the program used to display PDF files. This is necessary to open the PDF manual from the |Help| menu. If the option is empty, the program associated with the `.pdf` extension is used.

39.1.7 Backup files

Unless told to do otherwise, QF-Test creates backups of existing files when saving test suites or run logs. These backup files are useful only in protecting against failures when saving a file. They are by no means a replacement for proper system backups. The following options determine if, when and how backup files are created.

Figure 39.9: Backup file options

Create backup files for test suites (User)

Backup files for test suites are created only if this option is activated. Please be careful and don't turn it off without a good reason, such as using a version control

system for test suites which obviates the need to create backups. Just think about the amount of work that goes into creating a useful test suite and imagine the frustration if it gets destroyed accidentally.

Create backup files for test run logs (User)

Usually a run log is far less valuable than a test suite, so there is a separate option that determines whether backups are created for run logs.

Backup frequency (User)

There are two possibilities for the frequency with which backup files are created.

With the first option, "One backup per session", QF-Test creates a backup file only the first time a file is saved. If you continue editing the suite or run log and save it again, the backup file is left unchanged. Only when you edit a different file or restart QF-Test, a new backup is created. This setting is useful is you keep only one backup per test suite.

If, on the other hand, you keep multiple backups per suite, "Backup on every save" may be the preferred choice.

Name of the backup file (User)

Like many other things, the conventions for the names of backup files differ between Unix and Windows. While the common extension for a backup file under Windows is `.bak`, there are many variants under Unix. One of the most common is appending a tilde character '~'.

Number of backup files to keep (User)

You can keep more than one backup file for each test suite or run log. If you do so, backup files are named after the scheme `.bak1`, `.bak2`... for the `.bak` naming style and ~1~, ~2~... for the other. The most recent backup is always numbered 1. When a new backup is created, the old number 1 is renamed to 2, 2 renamed to 3 and so on. When the maximum is reached, the oldest files are deleted.

Auto-save interval (s) (User)

Interval after which a modified test suite is saved automatically. Setting this value to 0 will disable auto-saving. Otherwise values less than about 20 seconds are not useful. Run logs are never saved automatically. Auto-save files are created in the same directory as the test suite or - in the case of new suites that have never been saved - in the user configuration directory[10].

39.1.8   Library path

Figure 39.10: Library path option

Directories holding test suite libraries (System)

This is a list of directories that are searched for test suites whenever a suite reference is given as a relative path that cannot be resolved relative to the current suite. This includes direct suite references in the Procedure name[(590)] attribute of a Procedure call[(589)] or a Component[(823)] QF-Test ID[(824)] reference as well as suites included through the Include files[(516)] attribute of the Test suite[(515)] node.

The `include` directory belonging to the current version of QF-Test is automatically and invisibly placed at the end of the library path. This ensures that the common library `qfs.qft` can always be included without knowing its actual location and that its version is matching the version of QF-Test at all times.

Note   If the command line argument `-libpath <path>`[(873)] is given it will override the settings of this option. In interactive mode, the value of the command line argument is displayed here, but it will not be saved with the system configuration unless it is modified.

39.1.9   License

Figure 39.11: License options

Normally QF-Test license bundles contain a homogeneous mix of GUI engines. For example, a bundle of QF-Test/swing licenses only supports the AWT/Swing engine, QF-Test/suite licenses support both AWT/Swing and SWT for all included licenses. For these kinds of simple licenses these license settings can be ignored.

A small problem arises in case of mixed engine licenses where some GUI engine is included only for a part of the licenses. An example for this is a license bundle that was formerly purchased for qftestJUI, upgraded to QF-Test/suite with QF-Test 2.0 and then extended with further QF-Test/swing licenses, say two licenses for QF-Test/suite and another two for QF-Test/swing. Such a license allows running four concurrent instances of QF-Test, but only two of these can make use of the SWT engine. If more than two instances are started with SWT support there will be a license conflict.

When QF-Test detects such a mixed license bundle for the first time it asks the user which engine licenses to use. The choice made then can be changed here at any

time. Besides, QF-Test can be started with the command line argument <u>—engine <engine></u>[871] to override the supported GUI engines for this execution.

39.1.10 Updates

To get the latest features and bug-fixes QF-Test can check for updates automatically. The following options determine whether to check and when to notify for available updates. Using the command line argument <u>-noupdatecheck 42.2.3</u>[875] you can disable the automatic update check.

Figure 39.12: Update options

Automatically check for updates (User)

QF-Test automatically checks for updates upon startup. Deactivating this option disables this feature.

Ask for Update (User)

If a new version is available QF-Test shows a notification with links to the release notes and the download page. This option limits those notifications to specific kinds of versions.

- Minor updates contain mostly bug-fixes and small improvements.
- Medium upgrades are released to provide new features.
- Major upgrades include significant new features and may change the behavior of QF-Test.

39.2 Recording options

The following options determine which kinds of events are recorded and which filters are applied, how components are recorded and arranged, and how sub-items are handled.

Figure 39.13: Recording options

Show initial quickstart help on record button (User)

Controls the display of an initial question mark on the record button in order to directly lead new users to the quickstart wizard.

Hotkey for recording (User)

SUT script name: OPT_RECORD_HOTKEY
Event recording can be directly started/stopped by pressing a key in the SUT. The key to be used for this function is set through this option. To set an option, click into the field and press the desired key combination. The default key is F11.

Keep variables in recorded client names (System)

This option is very useful if the client name assigned to your SUT contains variables (e.g. `$(client)`), which generally makes sense when creating procedures. If this option is set, the client attribute of all recorded nodes is set to the unexpanded Client[(640)] value of the Start SUT client[(639)] node through which the SUT was started.

Insert recording at current selection (User)

Depending on what you are currently working on, it may or may not make sense to add newly recorded sequences directly at the current insertion mark. If you deactivate this option, new recordings are placed in a new Sequence[(536)] under Extras[(547)].

Hotkey for checks (User)

SUT script name: OPT_RECORD_CHECK_HOTKEY
To simplify recording a sequence of events with interspersed checks, you can switch between plain recording and recording checks by pressing a key in the SUT. The key that triggers this switch is set through this option. To set an option, click into the field and press the desired key combination. The default key is F12.

Highlight components when checking (User)

SUT script name: OPT_RECORD_CHECK_HIGHLIGHT
When QF-Test is recording checks, it can give visual feedback on the component the mouse is currently over by inverting its foreground and background colors. Rarely this may have unwanted visual side effects, so you can turn that feature off with this option.

Show message if no events were recorded (User)

Server script name: OPT_SHOW_EMPTY_RECORDING_MESSAGE
When starting to record and stopping without interacting with the SUT in between, a message is shown indicating that no events were recorded. By deactivating this option that message can be suppressed.

39.2.1 Events to record

These options specify which kinds of events are recorded and which aren't. You should not tinker with these unless you know what you are doing.

Figure 39.14: Options for events to record

Abstract 'Mouse click' events (System)

Activating this option causes a sequence of MOUSE_MOVED, MOUSE_PRESSED, MOUSE_RELEASED and MOUSE_CLICKED events to be recorded as a single 'Mouse click' pseudo event (see section 40.8.1[(681)]).

Simplified 'Mouse click' recording (System)

4.1+

When recording 'Mouse click' events, this option should also be activated. Except for Drag&Drop and special case MOUSE_MOVED events, recording is then based primarily on MOUSE_PRESSED events, turned into 'Mouse clicks'. This gives best results in most cases, even when QF-Test receives too few or too many events from the SUT. If this option is turned off, the algorithm from QF-Test 4.0 and older is used. This is worth a try in case a recorded sequence cannot be replayed directly.

Abstract 'Keystroke' events (System)

This option lets you record a sequence of KEY_PRESSED, KEY_TYPED and KEY_RELEASED events (or just KEY_PRESSED and KEY_RELEASED for function keys) as a single 'Keystroke' pseudo event (see section 40.8.2[(685)]).

Record MouseEvents without coordinates where possible (System)

SUT script name: OPT_RECORD_REPOSITION_MOUSE_EVENTS
For many types of components and sub-items it doesn't matter where exactly a MouseEvent occurs. However, if large values are recorded for the X or Y

coordinate of a Mouse event⁽⁶⁸¹⁾, there's the danger that the event might miss its target upon replay if the component has shrunk a little due to font changes or because the window has been resized. This is also a possible source for problems when converting a recorded sequence to a Procedure⁽⁵⁸⁶⁾ with variable target components.

If this option is activated, QF-Test ignores the coordinates of recorded MouseEvents if it thinks that the coordinates don't matter for the target component, e.g. for all kinds of buttons, for menu items, table cells, list items and tree nodes. For the latter QF-Test distinguishes between clicks on the node itself and on the expand/collapse toggle. When MouseEvents without coordinates are played back, QF-Test targets the center of the respective component or item except that the X coordinate for items is limited to 5 because item bounds cannot always be calculated correctly.

Convert opening of a window into Wait for component to appear⁽⁷⁷¹⁾ (System)

When replaying a sequence during which a new window is opened, it may be useful to allow for a longer than usual delay until the window is opened. By activating this option, a recorded `WINDOW_OPENED` event will be turned into a Wait for component to appear⁽⁷⁷¹⁾ node automatically.

Web

For web clients this option causes a Wait for document to load⁽⁷⁷⁶⁾ node to be inserted whenever loading of a document completes. This is important for proper synchronization when navigating to another page.

39.2.2   Events to pack

In order to keep the amount of raw event data generated during normal use of a Java GUI manageable, QF-Test employs a set of recording filters and packers. These do their best to keep everything needed for successful replay and throw away the rest. To get an impression of the actual data behind a recorded sequence, try recording a short sequence with all of these turned off, and with 'Mouse click'⁽⁴⁴¹⁾ and 'Keystroke'⁽⁴⁴¹⁾ pseudo events disabled.

Figure 39.15: Options for events to pack

MOUSE_MOVED events (System)

SUT script name: OPT_RECORD_PACK_MOUSE_MOVED
MOUSE_MOVED events are especially frequent. Every mouse cursor motion generates a handful of these. Under most circumstances, only the last of a consecutive series of MOUSE_MOVED events is actually useful, so all events except the last one are dropped, if this option is activated. An example where this is not advisable is recording some freehand drawing in a graphics application.

Note One might think that MOUSE_MOVED events are completely useless, since MOUSE_PRESSED or MOUSE_RELEASED events have their own set of coordinates, but this is not the case. Some Java components require a MOUSE_MOVED event before a MOUSE_PRESSED event is recognized.

MOUSE_DRAGGED events (System)

SUT script name: OPT_RECORD_PACK_MOUSE_DRAGGED
MOUSE_DRAGGED events are like MOUSE_MOVED events, but with one mouse button held down. Similarly, only the last of a series of MOUSE_DRAGGED events is recorded unless you turn off this option, except for special cases (see next option).

Mouse drag hover delay (System)

SUT script name: OPT_RECORD_MOUSE_DRAGGED_HOVER
There are situations where not only the final target of a mouse drag is of interest, but intermediate points as well. The most common is invoking a menu item in a sub-menu.

Note

As of QF-Test 1.05.2, the following no longer applies because MOUSE_MOVED or MOUSE_DRAGGED events that are required for opening sub-menus are not "optimized away" anymore. However, there may be other situations where intermediate stops are useful when recording drags.
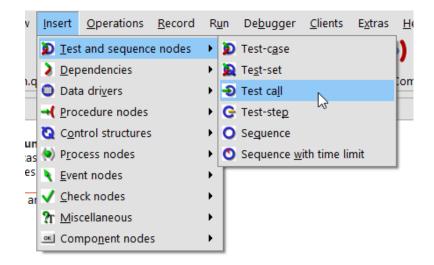
Figure 39.16: Dragging to a sub-menu

As illustrated above, creating a new Test call node for a suite could be done by clicking on the Insert menu button, dragging the mouse to the Test and Sequence nodes item for the sub-menu, so the sub-menu pops up, then dragging on to the Testcall menu item and releasing the mouse button. Normally such a sequence would be reduced to a press, a drag to the final Testcall item and a release. It would fail to replay, since the sub-menu would never be popped up. To work correctly, an additional drag to the Test and Sequence nodes item must be recorded.

For that reason QF-Test recognizes a MOUSE_DRAGGED event as important if you hover over an intermediate component for a while during the drag. The delay (in milliseconds) needed to recognize such an event is set through this option.

To record the above example correctly with this option set to 1000, you'd have to click on the Insert menu button, drag to the Test and Sequence nodes item and

keep the mouse pointer stationary for one second, then move on to the $\boxed{\text{TestCall}}$ sub-menu item and release the mouse button.

Maximum drag distance for 'Mouse click' event (System)

It sometimes happens unintentionally that the mouse cursor is moved between pressing the mouse button and releasing it. This movement may be registered as a `MOUSE_DRAGGED` event, depending on the JDK version and the distance of the move. QF-Test is able to compensate for small movements and still convert the click into an abstract 'Mouse click' event. This option defines the maximum distance between pressing and releasing the mouse button that QF-Test will ignore. Every `MOUSE_DRAGGED` event above that distance will be left unchanged.

Collect key events into a Text input node (System)

Another example where a lot of events are generated is entering a short string of text into a text field. Each character typed leads to at least one `KEY_PRESSED`, one `KEY_TYPED` and one `KEY_RELEASED` event. For additional fun, the `KEY_RELEASED` events may arrive out of order or not at all, depending on operating system and JDK version.

If this option is activated, sequences of KeyEvents on a text component (to be exact: a component whose class is derived from `java.awt.TextField` or `javax.swing.text.JTextField`) are converted into a Text input[(689)] node. Only true character input is packed, function or control keys or key combinations with $\boxed{\text{Control}}$ or $\boxed{\text{Alt}}$ are left unchanged.

When the packed sequence is replayed, only `KEY_TYPED` events are generated. `KEY_PRESSED` and `KEY_RELEASED` events cannot be generated, since the required key code is system dependent and cannot be determined from the character alone. This is not a problem however, since text components usually handle only `KEY_TYPED` events and some special keys.

Automatically set 'Clear...' attribute of recorded Text input nodes (System)

This option determines the value of the Clear target component first[(691)] attribute of a recorded Text input[(689)] node. If the option is not set, the attribute will not be set either. Otherwise, the Clear target component first attribute is set if and only if the text field or text area was empty before the input started.

Always set 'Replay single events' attribute of recorded Text input nodes (System)

This option determines the value of the Replay single events[(691)] attribute of a recorded Text input[(689)] node. If the option is set, the attribute will be set and vice

versa. The conservative way is to keep the option set, but for a typical application that does not add its own KeyListeners to text fields it should be safe to turn it off so as to speed up replay of Text input nodes.

39.2.3 Components

General information regarding the settings for recording of class names:

QF-Test can record classes of component in various ways, therefore it organizes component classes in various categories. Those categories are called as the specific class, the technology-specific system class, the generic class and the dedicated type of the generic class. Each category is recorded at Extra features[(825)].

The option Record generic class names for components[(449)] is checked by default. Using this option allows you to record generic classes in order to share and re-use your tests when testing a different technology with just minor changes to the existing tests.

Swing
In case you work with one Java engine only and you prefer to work with the "real" Java classes, you could also work without the generic class recording. In this case you should consider to check the option Record system class only[(449)]. This option makes QF-Test record the technology-specific system class instead of the derived class. If you switch off this option you will get the derived class which enables you to make a very well targeted recognition but could cause maintenance efforts in case of changes coming from refactoring. In case to derived classes were obfuscated you must not set this option.

Figure 39.17: Options for recording components

Hotkey for components (User)

SUT script name: OPT_RECORD_COMPONENT_HOTKEY
This option defines the key for a very useful functionality: recording components directly from the SUT. Pressing this key in the SUT will switch it to "component recording" mode, regardless of whether event recording is currently activated or not. To set an option, click into the field and press the desired key combination. The default key is Shift-F11.

In this mode, clicking on a component with the mouse will cause the component to be recorded and added to the test suite if it was unknown before. If more than one test suite is currently opened, the menu item Record→Receive components determines the suite that will receive the components. The QF-Test ID[824] of the Component[823] is put on the clipboard so it can be pasted into any text field with

[Control-V]. This latter feature is very handy when creating an event or a check from scratch.
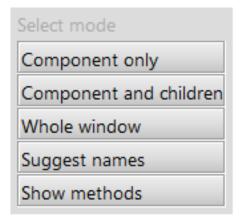
Figure 39.18: Popup menu for recording components

You can also record a whole component hierarchy at once by clicking with the right mouse button instead. This will bring up a popup menu with the following four options:

Component only
This is similar to clicking with the left mouse button. Only the selected component is recorded.

Component and children
The selected component and all components contained therein are recorded.

Whole window
Records every component in the whole window.

Suggest names
This is a special feature to improve the collaboration between testers and developers in deciding which components should have names set with `setName()`. All components in the whole window are recorded and put into a test suite of their own. Each unnamed component for which a name will improve testability is marked with a name of the form "SUGGESTED NAME (n): suggestion". The running count in braces is just used to avoid duplicates. The suggested name is built from the component's class and other available information. It should be taken with a grain of salt.

Show methods
This is another special feature that brings up a component inspector window showing the attributes and methods of the selected component's class. See section 5.12[96] for further information.

Normally "component recording" mode is turned off by either pressing the hotkey again or by selecting a component. If you want to record multiple single components, hold down an additional modifier key when switching to "component recording" mode. That way selecting a component will not turn off the mode, only pressing the hotkey again will.

Record generic class names for components (System)

4.0+

SUT script name: OPT_RECORD_COMPONENT_GENERIC_CLASS
Where possible QF-Test assigns generic class names to components like "Button", "Table" or "Tree" in addition to the actual Java, DOM or framework-specific class names like "javax.swing.JButton", "javafx.scene.control.Button", "INPUT" or "X-BUTTON". These generic class names are more descriptive and robust, improve compatibility between different UIs and enable creation of generic utility procedures. Generic class names can be used for component recognition or registering resolvers. If this option is active, generic class names are recorded where available.

Record system class only (System)

SUT script name: OPT_RECORD_COMPONENT_SYSTEM_CLASS_ONLY
If this option is set, QF-Test does not record any custom classes for Components[(823)]. Instead it moves up the class hierarchy until it encounters a system class and records that. Set this option if the class names of your custom GUI classes tend to differ between releases.

Note   You must activate this option if you intend to obfuscate the jar files of your application or if you are using a custom class loader to load your application's GUI classes.

Web   This option does not apply to web SUTs.

Match any class when recording components (System)

4.0+

SUT script name: OPT_RECORD_TOLERANT_CLASS_MATCH
For compatibility with older QF-Test versions that did not have generic classes, QF-Test now matches against several classes of a component when recording, the concrete class, the generic class and the system class. This is very useful if you want to retain as many of your old components as possible. If you would rather get new components based on generic classes in new recordings you should deactivate this option. Components recorded for the first time will always be recorded with the class determined by the preceding two options Record generic class names for components[(449)] and Record system class only[(449)].

Validate component recognition during recording (System)

SUT script name: OPT_VALIDATE_RECORDED_COMPONENTS
In case non-unique names are assigned to components QF-Test can still distinguish between these components with the help of the Extra feature⁽⁸²⁵⁾ qfs:matchindex that specifies the index of the component with the given name. If this option is set, QF-Test will check the name of the component during recording and try to assign qfs:matchindex correctly.

Note   You should only deactivate this option if you are sure that component names are "reasonably unique" and the component validation significantly impacts performance during recording.

Convert HTML components to plain text (System)

Swing

SUT script name: OPT_RECORD_COMPONENT_CONVERT_HTML
Swing supports HTML markup in various kinds of labels, buttons and sub-elements of complex components. For component identification and validation, the HTML markup is often not useful and will clutter up things. If this option is set, QF-Test converts HTML to normal text by removing all HTML markup so only the actual text content is left.

Name override mode (record) (System)

Server (automatically forwarded to SUT) script name: OPT_RECORD_COMPONENT_NAME_OVERRIDE
Possible Values: VAL_NAME_OVERRIDE_EVERYTHING,
VAL_NAME_OVERRIDE_HIERARCHY,
VAL_NAME_OVERRIDE_PLAIN

Note   There are two versions of this option which are closely related. This one is effective during recording, the other one⁽⁴⁷²⁾ during replay. Obviously, both options should always have the same value. There's one exception though: When migrating from one setting to another, QF-Test's components have to be updated. During that process, keep the replay option at the old setting and change this option to the new one. Be sure to update the replay setting after updating the components.

This option determines the weight given to the names of components for recording. Possible choices are:

Override everything
This is the most effective and adaptable way of searching components, but it requires that the names of the components are unique, at least within the same window. If that uniqueness is given, use this choice.

Don't use this value for a web page with frames. Use "Hierarchical resolution" instead.

Hierarchical resolution
This choice should be used if component names are not unique on a per-window basis, but naming is still used consistently so that two components with identical names have at least parent components or ancestors with distinct names. That way, component recognition is still tolerant to a lot of change, but if a named component is moved to a different named parent in the SUT, the test suite will have to be updated to reflect the change.

Plain attribute
If there are components with identical names in the SUT within the same parent component you must use this setting. The name will still play an important role in component recognition, but not much more than the Feature[825] attribute.

Automatic component names for Eclipse/RCP applications (System)

SUT script name: OPT_RECORD_COMPONENT_AUTOMATIC_RCP_NAMES
Eclipse and applications based on the Rich Client Platform (RCP) have a complex GUI with support for changing perspectives. Such a change causes components to be rearranged which can make it hard for QF-Test to recognize them unless names are set at least on the major components. This is further complicated by the fact that the structure of the components is not what it appears to be - the main components are all arranged in a relatively flat hierarchy within the workbench. On the upside, RCP based applications have a uniform inner structure based on `Views` and `Editors`, many of which are named.

If this option is turned on, QF-Test will do its best to automatically associate GUI elements with their RCP counterparts and assign names based on that association. This can drastically improve component recognition for such applications. However, if some names thus assigned turn out not to be reliable over time, they can also interfere. In such a case, names can be assigned to the affected components either using `setData` as described in chapter 5[42] or with the help of a `NameResolver` as described in section 52.1.7[1023]. Both will override automatically generated names.

Component hierarchy (System)

Server script name: OPT_RECORD_COMPONENT_HIERARCHY
Possible Values: VAL_RECORD_HIERARCHY_INTELLIGENT,
VAL_RECORD_HIERARCHY_FULL,
VAL_RECORD_HIERARCHY_FLAT
QF-Test supports three different kinds of views for the components of the SUT.

For more information about their effect on component recognition, see section 46.1[902].

The flat view collects all components of a window as direct child nodes of the respective Window[812] node. The advantage of this view is that structural changes of the component hierarchy have little effect on component recognition. This is also its greatest disadvantage: since structural information is not available, this view gives reasonable recognition quality only if `setName()` is used ubiquitously. Another drawback is the lack of clearness.

The complement to the flat view is the full hierarchy. It includes every single component of the SUT's GUI, emulating all parent/child relationships. This view can be a useful tool for developers or testers that want to gain insights into the SUT's structure, but is not very applicable for testing, since structural changes affect it too much. As long as you don't change the GUI however, it will give you excellent recognition without the help of `setName()`.

A compromise between flat and full hierarchy is available through the choice "Intelligent". For this view only the "interesting" components of the SUT are recorded. "Interesting" in this case means that either the user can interact with the component, or it is located at some important point in the hierarchy, like the children of a split pane or a tabbed pane. In some later version of QF-Test this decision may be made configurable as well.

Prepend QF-Test ID of window parent to component QF-Test ID (System)

Server script name: OPT_RECORD_COMPONENT_PREPEND_WINDOW_ID
If selected, QF-Test prepends the QF-Test ID of the Window[812] parent of a Component[823] to its QF-Test ID[824] during recording. This is useful to disambiguate QF-Test IDs of components with identical names in different windows.

Prepend parent QF-Test ID to component QF-Test ID (System)

Server script name: OPT_RECORD_COMPONENT_PREPEND_PARENT_ID
Possible Values: VAL_RECORD_COMPONENT_PREPEND_PARENT_ALWAYS,
VAL_RECORD_COMPONENT_PREPEND_PARENT_NAMED,
VAL_RECORD_COMPONENT_PREPEND_PARENT_FEATURE,
VAL_RECORD_COMPONENT_PREPEND_PARENT_NEVER
When a Component[823] is recorded for the first time, QF-Test assigns an automatically generated QF-Test ID[824]. The QF-Test ID of a direct or indirect parent node may be prepended to this QF-Test ID. This is useful to distinguish between similar components that don't have a name of their own.

Example: Imagine two `JScrollPanes`, one named "TreeScrollPane" and the other named "DetailScrollPane". Without this functionality, their vertical scrollbars

would get the QF-Test IDs "scrollbarVertical" and "scrollbarVertical2". With this function turned on, the IDs would be "TreeScrollPane.scrollbarVertical" and "DetailScrollPane".scrollbarVertical". That way it is immediately obvious which component is the target of an event.

There are four possible settings:

- "Never" turns this option off.

- "Nearest named ancestor" is a useful setting, if your developers have assigned names to all major components with the Java method `setName`. A component that doesn't have a name of its own, gets the QF-Test ID of its nearest named ancestor node prepended.

- If `setName` is used sparingly or not at all, it is better to set this option to "Nearest ancestor with name or feature". That way either the name or a distinctive feature of an ancestor node will be applicable.

- "Always" is only useful if the option Component hierarchy[(451)] is set to "Flat". With this setting, every component gets the QF-Test ID of its parent node prepended, which can lead to unusably long QF-Test IDs when components are nested deeply.

39.2.4 Recording sub-items

Events on complex components like tables or trees can be recorded relative to a sub-item of the component.

Figure 39.19: Options for recording sub-items

Record sub-items as event targets (System)

SUT script name: OPT_RECORD_SUBITEM
This option activates recording sub-items. When turned off, events on complex components are no different from events on simple components.

Multi-level sub-items (System)

Server (automatically forwarded to SUT) script name: OPT_RECORD_SUBITEM_MULTILEVEL
Via this option you can completely disable multi-level sub-items (even for replay). However, you should only turn this feature off in case you are running into problems caused by test suites that contain unquoted special characters like '@' or '%' in textual sub-item indexes. Even then it is preferable to update the test suites with properly quoted items, possibly using the special variable syntax ${quoteitem:...} (see section 6.5[108]).

Sub-item format (System)

SUT script name: OPT_RECORD_SUBITEM_FORMAT
Possible Values: VAL_RECORD_SUBITEM_FORMAT_INTELLIGENT,
VAL_RECORD_SUBITEM_FORMAT_TEXT,
VAL_RECORD_SUBITEM_FORMAT_NUMBER
When recording an event for a sub-item, the Item's[829] index can be defined As string[831] or As number[831].

The third choice, "Intelligent", causes QF-Test to record the index in the format most appropriate for the item. If the name of the item is unique within the complex component, a string index is recorded, a numeric index otherwise.

Sub-item type (System)

Server script name: OPT_RECORD_SUBITEM_TYPE
Possible Values: VAL_RECORD_SUBITEM_TYPE_INTELLIGENT,
VAL_RECORD_SUBITEM_TYPE_NODE,
VAL_RECORD_SUBITEM_TYPE_SYNTAX
With this option you control whether a Item[829] node is created for a sub-element during event recording or the element is referenced directly in the attribute QF-Test component ID[682] of the Mouse event[681] node. (see section 5.9[82]).

Choosing "Intelligent" will only cause a node to be created if the Index is given as text[831] and the sub-element is not editable in the SUT.

Represent tree node as path (System)

SUT script name: OPT_RECORD_SUBITEM_TREE_PATH

It is not uncommon that trees have identically named nodes under different parent nodes, e.g. a file system with the directories `/tmp` and `/usr/tmp`. By using a path format in the Items[(829)] for tree nodes QF-Test can make full use of the hierarchical structure to distinguish between these nodes. The slash character '/' is used as separator.

If this option is deactivated, trees will be treated as flat lists.

39.2.5   Recording procedures

The following options determine the configuration of the Procedure Builder which is described in detail in Automated Creation of Basic Procedures[(307)].

Figure 39.20: Procedure Builder options

Hotkey for procedure recording (User)

SUT script name: OPT_RECORD_PROCEDURE_HOTKEY

This option defines a key for turning on automatic procedure recording directly in the SUT. To set a hotkey, click into the field and press the desired key combination. The default key is Shift-F12 .

Configuration file for recorded procedures (System)

Here you can specify your own template file for the Procedure Builder. If a relative path is given, QF-Test looks for the definition file in the directory that QF-Test was started from and in the default include directory.

39.3 Replay options

The following settings change the way test suites are executed.

Figure 39.21: Replay options

Hotkey for pausing test run ("Don't Panic" key) (User)

> Server (automatically forwarded to SUT) script name:
> OPT_PLAY_DONT_PANIC_HOTKEY
> When running a test at full speed it can be rather difficult to get the focus to
> QF-Test's window and interrupt the test so you can do something different without

having all these windows flashing around the screen. This is all the more true when the options <u>Actually move mouse cursor</u>[(468)] or <u>Raise SUT windows automatically</u>[(467)] are activated or when running in batch mode.

This option lets you define a key combination (the default being [Alt-F12]) that will instantly pause all running tests if it is pressed in any SUT or QF-Test window (unless multiple QF-Test instances are run simultaneously, of course). Pressing the same key combination again will resume all tests, unless you manually resume or stop any of them. In that case its effect is automatically reset to suspend tests.

To set a hotkey, click into the field and press the desired key combination.

Call stack size (System)

Server script name: OPT_PLAY_CALLSTACK_SIZE
The call stack size is a limit for the nesting depth of <u>Sequences</u>[(536)] or <u>Procedure calls</u>[(589)] during replay. This limit is needed to detect and handle endless recursion. When the nesting depth exceeds the call stack size, a <u>StackOverflowException</u>[(857)] is thrown. The default value of 200 should be sufficient but can be increased for very complex tests.

Log warning for nested test cases (System)

Server script name: OPT_PLAY_WARN_NESTED_TEST_CASE
Execution of <u>Test case</u>[(518)] nodes should not be nested because such Test cases cannot be listed properly in the report. If this option is active, a warning is logged in case a Test case is executed within another Test case.

Mark nodes during replay (User)

If set, tree nodes that are currently executed are marked with an arrow.

Show replay messages in status line (User)

Determines whether the name of the currently executing node is shown in the status line.

Raise test suite window after replay (User)

This option is mainly used together with the option <u>Raise SUT windows automatically</u>[(467)]. It causes the window of a test suite to be raised after a test run.

See also option <u>Force window to the top when raising</u>[(467)].

Minimize test suite window during replay (User)

If this option is set, QF-Test will minimize the window of a test suite while its tests are being executed. The window will pop back up automatically when the test is stopped or suspended. This feature is especially useful on Windows 2000/XP systems where programs are prohibited from bringing their windows to the top so QF-Test cannot raise the windows of the SUT.

Show message dialog after (User)

After replay is finished, the status line shows the number of errors and warnings that occurred. If an uncaught exception was thrown, an error dialog is displayed. Additionally, a message dialog can be displayed in case of warnings or errors or every time a run is finished. This option sets the minimum error level that triggers such a message dialog.

Always locate the source of an error (User)

When an exception is thrown during replay, the node that caused the exception will be made visible and selected. If you don't like this, you can turn this feature off and locate the node via the $\boxed{\text{Run→Find last error source...}}$ menu item instead.

Salt for crypting passwords (System)

3.0+

QF-Test can store encrypted passwords in the $\text{Text}^{(691)}$ attribute of a $\text{Text input}^{(689)}$ node for a password field or the $\text{Detail}^{(699)}$ attribute of a $\text{Selection}^{(697)}$ used for a login dialog in a web SUT. When such passwords are en- or decrypted, QF-Test combines the key with the salt specified in this option. Without this salt, anybody with sufficient knowledge is able to decrypt your passwords to get the plain-text version.

Note

Don't let this option give you a false sense of security. Anybody that gains access to this salt and anybody that can execute your tests can also gain access to the plain-text version of the password. However, encrypting passwords is still useful to prevent obvious plain-text passwords getting stored in test suites and run logs, and encrypted passwords are reasonably safe from someone who only gets hold of a test suite or run log without access to this salt.

How to handle disabled components (System)

4.0+

Server script name: OPT_PLAY_ERROR_STATE_DISABLED_COMPONENT
Possible Values: VAL_PLAY_DISABLED_COMPONENT_WARNING,
VAL_PLAY_DISABLED_COMPONENT_ERROR,

VAL_PLAY_DISABLED_COMPONENT_EXCEPTION
In case you replay an event on a component, which is disabled you can configure QF-Test's behavior for that case. You can

- Log a warning message

- Log an error message

- Throw a `DisabledComponentStepException`[(851)]

How to handle exceeded execution timeout (System)

4.1+

Server script name: OPT_PLAY_ERROR_STATE_EXECUTION_TIMEOUT
Possible Values: VAL_PLAY_EXECUTION_TIMEOUT_WARNING,
VAL_PLAY_EXECUTION_TIMEOUT_ERROR,
VAL_PLAY_EXECUTION_TIMEOUT_EXCEPTION,
VAL_PLAY_EXECUTION_TIMEOUT_WARNING_IMMEDIATE,
VAL_PLAY_EXECUTION_TIMEOUT_ERROR_IMMEDIATE,
VAL_PLAY_EXECUTION_TIMEOUT_EXCEPTION_IMMEDIATE
In case of an exceeding execution timeout, you can configure QF-Test's behavior for that case. You can

- Log a warning message and run through possible cleanup nodes.

- Log an error message and run through possible cleanup nodes.

- Throw an `ExecutionTimeoutExpiredException`[(852)] and run through possible cleanup nodes.

- Log a warning message and stop the node immediately.

- Log an error message and stop the node immediately.

- Throw an `ExecutionTimeoutExpiredException`[(852)] and stop the node immediately.

The definition of running cleanup nodes includes that Cleanup and Catch nodes get executed. Stopping the node immediately stands for not executing possible Cleanup and Catch nodes at the end.

39.3.1 Client options

Various settings for process and SUT clients can be adjusted with the following options:

Figure 39.22: Client options

Ignore empty arguments when starting a client (System)

Server script name: OPT_PLAY_CLIENT_START_IGNORE_EMPTY_ARGUMENT
If the option is set (default), empty program parameters or class arguments will be ignored in 'starter' nodes like Start Java SUT client[(635)], Start SUT client[(639)], Start process[(642)], Start web engine[(645)], Start windows application[(657)] or Launch Android emulator[(663)]. This is very useful for using variables in parameters because an empty variable is then equivalent to removing the parameter from the list. If the option is turned off, an empty command line argument (i.e. ") will be passed to the starting program instead.

Ask whether to stop clients before exiting (User)

If there are still active clients upon exit of QF-Test, these are terminated after asking for confirmation. If this option is turned off, the clients are terminated unconditionally.

When terminating a process, kill its whole process tree (System)

Server script name: OPT_PLAY_KILL_PROCESS_TREE
The process of an SUT or a helper program started during a test can be terminated via a Stop client[(675)] node or manually via the ⟦Client⟧ menu. In case of an SUT, QF-Test first tries to communicate with it and initiate a clean `System.exit` call. Non-Java programs have to be killed. If the program has started further child processes these may or may not get terminated by a normal shutdown or kill, depending on circumstances.

It is normally undesirably to keep such processes around as they might interfere with other tests or lock files that need to be removed or overwritten. Unless this option is disabled, QF-Test will try to determine the whole process hierarchy for any program it started and make sure that the main process and all child processes get killed explicitly.

Number of terminated clients in menu (User)

Server script name: OPT_PLAY_MAX_CLIENTS
This option limits the number of menu items for terminated clients that are kept in the ⟦Clients⟧ menu.

Maximum size of client terminal (kB) (User)

3.0+

Server (automatically forwarded to SUT) script name: OPT_PLAY_TERMINAL_SIZE
The maximum amount of text (in kilobyte) that the individual client terminal will hold. If the limit is exceeded, old text will be removed when new text arrives. A value of 0 means no limit.

Note
This option also determines the amount of output available for the special variables `${qftest:client.output.<name>}`, `${qftest:client.stdout.<name>}` and `${qftest:client.stderr.<name>}`.

Highlight selected component in the SUT (User)

Server script name: OPT_PLAY_HIGHLIGHT_COMPONENTS
If this option is set, QF-Test will highlight the associated component in the SUT whenever a Component[(823)] node or a node that references a Component is selected.

How to handle exceptions in the SUT (System)

SUT script name: OPT_PLAY_SUT_EXCEPTION_LEVEL
Possible Values: VAL_PLAY_EXCEPTION_LEVEL_WARNING,
VAL_PLAY_EXCEPTION_LEVEL_ERROR,
VAL_PLAY_EXCEPTION_LEVEL_EXCEPTION
Exceptions that are thrown during event handling in the SUT are typically a sure sign for a bug in the SUT. This option determines what to do if such an exception is caught. You can

- Log a warning message

- Log an error message

- Throw an <u>UnexpectedClientException</u>[856]

Reuse IDs for SUT clients in nested sub-processes (System)

Server script name: OPT_PLAY_REUSE_SUT_IDS
This is a complex option which you should hopefully never care about. When an SUT client launches another process that itself connects to QF-Test, the new SUT client is identified by the name of the original SUT client with a ':' and a numeric ID appended. The first ID will always be 2, with increasing numbers for additional sub-processes.

When a sub-process terminates and another sub-process connects, QF-Test can either reuse the ID of the terminated process or continue incrementing to create a new ID.

In most cases it is preferable to reuse the sub-process ID. The most common case is a single sub-process that is started, terminated, then started again. By activating this option you can always address the single sub-process with the same client name.

In a more complex situation, multiple sub-processes may be launched and terminated more or less at random, depending on progression of the test run. In such a case, always incrementing the ID for a new process is more deterministic.

In either case the ID counter will be reset when the original SUT client is started anew.

Automatically perform garbage collection in the SUT (System)

SUT script name: OPT_PLAY_SUT_GARBAGE_COLLECTION
By default QF-Test performs a full garbage collection in the SUT once every few hundred <u>SUT script</u>[631] executions. This is necessary due to a limitation in Java's default garbage collection mechanism that allows an OutOfMemoryError to happen for the so called PermGen space, even though the required memory could easily be reclaimed by a garbage collection.

When you are trying to analyze the memory performance of your application, these explicit garbage collections might influence the results. For that case you can disable QF-Test's garbage collection via this option.

39.3.2 Terminal options

Various settings for the shared terminal can be adjusted with the following options:

Shared Terminal

Maximum size of shared terminal (kB)

20

Regular expression to suppress display of certain text

☑ Use rich text terminal

Terminal font size (pt)

14

☑ Use monospaced font

☑ Regular expression coloring

Regular expression for red highlighting

(?md).+Exception\b.*\n(?>\n?^(?>\sat|Caused by:)\s.+\n)+|.*(

Regular expression for orange highlighting

(?md)^[1-2] \(\d\d:\d\d:\d\d\.\d\d\d\) .*|.*(?i)(?>error(?>s)?|feh

Regular expression for yellow highlighting

(?md)^[3-4] \(\d\d:\d\d:\d\d\.\d\d\d\) .*|.*(?i)(?>warning(?>s)?

Regular expression for blue highlighting

Regular expression for green highlighting

Figure 39.23: Terminal options

Maximum size of shared terminal (kB) (User)

Server script name: OPT_PLAY_SHARED_TERMINAL_SIZE
The maximum amount of text (in kilobyte) that the shared terminal will hold. If the limit is exceeded, old text will be removed when new text arrives. A value of 0 means no limit.

Regular expression to suppress display of certain text (User)

4.0+

By defining a regular expression for this option, certain text in the terminal output can be suppressed.

Default value is empty.
See also Regular expressions[(909)].

Use rich text terminal (User)

4.0+

If activated the rich text terminal allowing monospaced font type and coloring of given regular expressions. Deactivate this option if you want to switch back to the simple terminal as it were before QF-Test version 4.

Note   QF-Test needs to be restarted in order to make a change in this option become visible.

Terminal font size (pt) (User)

4.0+

This option lets you set the font size (as point value) used within the shared terminal.

This option only has an effect if the Use rich text terminal[(464)] is active.

Use monospaced font (User)

4.0+

If activated the rich text terminal will use a monospaced font.

This option only has an effect if the Use rich text terminal[(464)] is active.

Regular expression coloring (User)

4.0+

If activated the shared terminal output is processed for given regular expressions to be highlighted in different colors.

This option only has an effect if Use rich text terminal[(464)] and Regular expression coloring[(464)] are active.

Regular expression for red highlighting (User)

This option allows to define a regular expression for output to be shown in red color.

This option only has an effect if Use rich text terminal[464] and Regular expression coloring[464] are active.

Default value is:
(?md).+Exception\b.*\n(?%gt;\n?ˆ(?%gt;\sat|Caused
by:)\s.+\n)+|.*(?i)exception(?%gt;s)?\b.*
With this also typical Java stack traces will be highlighted.
See also Regular expressions[909].

Regular expression for orange highlighting (User)

This option allows to define a regular expression for output to be shown in orange color. The default regular expression matches error log output and lines containing respective text.

This option only has an effect if Use rich text terminal[464] and Regular expression coloring[464] are active.

Default value is:
(?md)ˆ[1-2] \(\d\d:\d\d:\d\d\.\d\d\d\) .*|.*(?i)(?%gt;error(?%gt;s)?|fehler)\b.*
With this also error log messages will be highlighted.
See also Regular expressions[909].

Regular expression for yellow highlighting (User)

This option allows to define a regular expression for output to be shown in yellow color. The default regular expression matches with typical stack traces and exceptions.

This option only has an effect if Use rich text terminal[464] and Regular expression coloring[464] are active.

Default value is:
(?md)ˆ[3-4]   \(\d\d:\d\d:\d\d\.\d\d\d\)
.*|.*(?i)(?%gt;warning(?%gt;s)?|warnung(?%gt;en)?)\b.*
With this also warning log messages will be highlighted.
See also Regular expressions[909].

Regular expression for blue highlighting (User)

This option allows to define a regular expression for output to be shown in blue

color. The default regular expression matches with typical stack traces and exceptions.

This option only has an effect if Use rich text terminal[(464)] and Regular expression coloring[(464)] are active.

Regular expression for green highlighting (User)

4.0+

This option allows to define a regular expression for output to be shown in green color. The default regular expression matches with typical stack traces and exceptions.

This option only has an effect if Use rich text terminal[(464)] and Regular expression coloring[(464)] are active.

39.3.3 Event handling

These options influence some details of how events are simulated in the SUT during replay.

Event handling

☑ Raise SUT windows automatically

☑ Force window to the top when raising

☑ Check for modal dialogs

☐ Actually move mouse cursor

Delay for hard events during replay (ms)

1000

☑ Scroll automatically to display sub-items

☑ Expand tree nodes as needed

☑ Throw DisabledComponentExceptions

Tolerance for checking images

5

How to handle events coming from the wrong thread
Error level

Error

☐ Strict checking

Maximum number of errors to log per SUT client

20

Figure 39.24: Event handling options

Raise SUT windows automatically (System)

SUT script name: OPT_PLAY_RAISE_SUT_WINDOWS
If this option is set, windows of the SUT for which a MouseEvent or KeyEvent is replayed will be raised automatically when they get activated. This eases switching between QF-Test and the SUT to visually verify that a sequence is replaying correctly.

See also options Raise test suite window after replay[457] and Force window to the top when raising[467].

Force window to the top when raising (System)

SUT script name: OPT_PLAY_RAISE_SUT_WINDOWS_FORCED

This option only has an effect on Windows systems.

Windows only allows an application to bring one of its own windows to the front if that application currently has the focus. This can make it difficult for QF-Test to raise SUT windows and to automatically switch between the SUT and QF-Test. If this option is activated, QF-Test temporarily sets the "always on top" attribute to force windows to the top.

See also options Raise test suite window after replay[(457)] and Raise SUT windows automatically[(467)].

Check for modal dialogs (System)

SUT script name: OPT_PLAY_CHECK_MODAL

A modal dialog is a window that blocks all input to other windows until it is closed. It is often used to display an error message or request user input.

Because the events simulated by QF-Test are artificial, they are not blocked by a modal dialog and can reach any window. This is normally not desirable, since the existence of a modal dialog may signal an unexpected error. Activating this option causes QF-Test to check for modal dialogs itself before replaying MouseEvents or KeyEvents. If such an event is targeted to a window that is blocked by a modal dialog, a `ModalDialogException`[(851)] is thrown.

Actually move mouse cursor (System)

SUT script name: OPT_PLAY_MOVE_MOUSE_CURSOR

If this option is set, the mouse cursor is actually moved across the screen as MouseEvents are simulated. This feature requires a working AWT robot.

While this option is mainly intended to give visual feedback, it can have a positive impact on test reliability because it reduces side-effects through events from the underlying system that might interfere with the automated test. However, for tests where precise mouse movement is essential, for example a drawing tool, this option should be turned off.

Delay for hard events during replay (ms) (System)

SUT script name: OPT_PLAY_DELAY_HARD_EVENTS

During event replay QF-Test blocks or delays some "hard" events, i.e. events that come directly from the system, for example when the user is moving the mouse cursor. This is done to prevent them from interfering with the SUT in an unlucky

moment. Popup windows, which are used for menus and combo boxes among other things, are especially sensitive to such events which can cause them to pop down accidentally. Therefore, these filters improve testing stability considerably.

This option sets the maximum time that such events may be delayed. In the unlikely case that the filters have unwanted side effects with your application, you can turn them off by setting the value to 0.

Scroll automatically to display sub-items (System)

SUT script name: OPT_PLAY_SCROLL_ITEM
If this option is set, accessing a sub-item of a complex component inside a scroll pane will automatically cause the sub-item to be scrolled into view. In that case you can remove most recorded events on scroll bars or scroll buttons, which are not required for correct replay.

Expand tree nodes as needed (System)

SUT script name: OPT_PLAY_EXPAND_TREE
When accessing nodes of a tree component as hierarchic sub-items it is possible to select a node that is not visible because one of its ancestral nodes is not expanded. If this option is set, all ancestors of the node will be expanded as needed. Otherwise this situation leads to a `ComponentNotFoundException`[850].

Throw DisabledComponentException (System)

SUT script name: OPT_PLAY_THROW_DISABLED_EXCEPTION
If QF-Test replays events on a component that is not enabled, these events are ignored silently. In most cases this indicates an error which is signaled by throwing a `DisabledComponentException`[851].

Old test suites may not be prepared to deal with this exception. These test suites should be fixed, but as a quick workaround `DisabledComponentExceptions` can be suppressed by deactivating this option.

Tolerance for checking images (System)

SWT

Note

SUT script name: OPT_PLAY_IMAGE_TOLERANCE
Initially this option was intended for SWT/Gtk only but it turned out to be universally applicable and useful.

Image rendering in Java applications and web browsers is not always fully deterministic. Even within the same session on a display with limited color depth the RGB values of an icon image can vary slightly and it becomes worse when running

tests on different machines. Graphics driver, JDK version and operating system settings also play a role. This makes strict image checks almost unusable in some cases.

To work around the problem, this option defines a tolerance setting for default image checks. For a pixel's red, green and blue color values a deviation from the expected value by the given amount is tolerated. Thus exact image checking for the default algorithm can be enforced by setting this value to 0 but it is preferable to use the "identity" check algorithm instead (see Details about the algorithm for image comparison[(1167)]). The default setting of 5 is a good compromise, allowing checks with differences that are normally not visually perceivable to succeed.

How to handle events coming from the wrong thread (System)

Swing

SUT script name: OPT_PLAY_WRONG_THREAD_ERROR_LEVEL
Possible Values: VAL_PLAY_THREAD_LEVEL_WARNING,
VAL_PLAY_THREAD_LEVEL_ERROR,
VAL_PLAY_THREAD_LEVEL_EXCEPTION
It is a rather common mistake in Swing based Java applications to access GUI components from the wrong thread. Since Swing is not thread-safe, such calls may only be made from the AWT event dispatch thread. Otherwise the potential consequences are race conditions, leading to very subtle and hard-to-debug errors, or deadlocks, freezing the application and making it unusable. Background information about this topic is available from
http://download.oracle.com/javase/tutorial/uiswing/concurrency/index.html ,
specifically the sections on "Initial Threads" and "The Event Dispatch Thread".

When QF-Test registers an event on a thread other than the AWT event dispatch thread it issues an error message including a stack trace which can be useful in fixing the problem. This set of options defines the severity of the message, whether to perform strict checking and a maximum for the number of messages to log.

The possible choices for the option "Error level" are "Error" and "Warning". We strongly suggest that you keep the default setting of "Error" and make sure that such problems are fixed sooner rather than later because they represent a serious risk.

Strict checking (System)

Swing

SUT script name: OPT_PLAY_WRONG_THREAD_STRICT
If strict checking for wrong thread errors is activated, error or warning messages will be issued for all kinds of events triggered from the wrong thread. Otherwise the problem will be ignored for "less relevant" events.   Which events are considered more or less relevant is arbitrary, based on the fact that there's a lot of Java literature (including early Java documentation from Sun) claiming that it is

safe to create components on any thread and that thread-safety only needs to be enforced once a component is made visible. A lot of code follows this pattern and the risk for causing problems in this case is indeed minimal. In the presence of such code disabling "Strict checking" will cause error messages to be logged only for the more relevant problems. If you want to get rid of all thread violations - as we recommend - you should turn strict checking on.

Maximum number of errors to log per SUT client (System)

Swing

SUT script name: OPT_PLAY_WRONG_THREAD_MAX_ERRORS
In case an SUT contains code that violates thread safety it is possible that a very large number of events are triggered from the wrong thread. Logging all those errors can significantly impact test performance, yet logging more than the first few errors does not really contribute much. The option "Maximum number of errors to log per SUT client" limits the possible number of error messages for this case.

39.3.4   Component recognition

How component recognition works - and the impact of these options on it - is explained in section 46.1[(902)]. The pre-defined values should give good results. If you experience problems with component recognition, you can try to improve it by adjusting the probabilities.

Figure 39.25: Component recognition options

The name of a component plays a special role. The following option affects the impact of names:

Name override mode (replay) (System)

> SUT script name: OPT_PLAY_RECOGNITION_NAME_OVERRIDE
> Possible Values: VAL_NAME_OVERRIDE_EVERYTHING,
> VAL_NAME_OVERRIDE_HIERARCHY,
> VAL_NAME_OVERRIDE_PLAIN

Note   There are two versions of this option which are closely related. This one is

effective during replay, the other one[450] during recording. Obviously, both options should always have the same value. There's one exception though: When migrating from one setting to another, QF-Test's components have to be updated. During that process, keep this option at the old setting and change the record option to the new one. Be sure to update the replay setting after updating the components.

This option determines the weight given to the names of components for component recognition. Possible choices are:

Override everything
This is the most effective and adaptable way of searching components, but it requires that the names of the components are unique, at least within the same window. If that uniqueness is given, use this choice.

Web Don't use this value for a web page with frames. Use "Hierarchical resolution" instead.

Hierarchical resolution
This choice should be used if component names are not unique on a per-window basis, but naming is still used consistently so that two components with identical names have at least parent components or ancestors with distinct names. That way, component recognition is still tolerant to a lot of change, but if a named component is moved to a different named parent in the SUT, the test suite will have to be updated to reflect the change.

Plain attribute
If there are components with identical names in the SUT within the same parent component you must use this setting. The name will still play an important role in component recognition, but not much more than the Feature[825] attribute.

The algorithm for component recognition is very tolerant and biased towards finding a match. If the best match is not perfect QF-Test logs information about the encountered differences, either as a warning or a plain message, depending on the following options:

Log missing name (System)

SUT script name: OPT_PLAY_WARN_MISSING_NAME
If this option is set, a message will be logged whenever a component is targeted that does not have a name, but QF-Test "thinks" it should have one. A plausible name is suggested where possible.

Log ambiguous name (System)

SUT script name: OPT_PLAY_WARN_AMBIGUOUS_NAME
If the option Name override mode (replay)[472] is set to "Override everything" or

"Hierarchical resolution", a message is logged whenever QF-Test encounters more than one potential target components with the same name. That message can be suppressed with the help of this option.

Log feature mismatch (System)

SUT script name: OPT_PLAY_WARN_FEATURE_MISMATCH
A component is considered to have a "feature mismatch" if it is determined by QF-Test as the target best suited for an event or check even though at one or more levels of the hierarchy the recorded Feature[(825)] attribute did not match the current state of the component. If this option is activated, feature mismatches are logged, notifying you that it may be a good idea to update the affected components.

Log extra feature mismatch (System)

SUT script name: OPT_PLAY_WARN_EXTRA_FEATURE_MISMATCH
An "extra feature mismatch" is similar to a feature mismatch as explained above, except that it applies to extra features with status "should match". If this option is activated, extra feature mismatches are logged, notifying you that it may be a good idea to update the affected components.

Log structure mismatch (System)

SUT script name: OPT_PLAY_WARN_STRUCTURE_MISMATCH
A "structure mismatch" is similar to a feature mismatch as explained above, except that instead of the feature it is the structure information represented by the attributes Class index[(828)] and Class count[(828)] where the mismatch occurred. If this option is activated, structure mismatches are logged, notifying you that it may be a good idea to update the affected components.

Log intermediate named ancestor (System)

SUT script name: OPT_PLAY_WARN_NAMED_ANCESTOR
An "intermediate named ancestor" is a direct or indirect parent component of the target component in the SUT which is not part of the component hierarchy in QF-Test even though it has a name. If the option Name override mode (replay)[(472)] is set to "Hierarchical resolution", this is considered as a mismatch, comparable to a feature or structure mismatch. If this option is activated, interferences through intermediate named ancestors are logged, notifying you that it may be a good idea to update the affected components.

Log warning instead of message (System)

4.2+

SUT script name: OPT_PLAY_COMPONENT_WARNINGS
If this option is activated, deviations from the expected values during component recognition are logged as warnings instead of plain messages. It can be useful to temporarily activate this option to increase the visibility of such deviations, either to update the respective component information or to seek out false positive matches. For normal testing this creates too much noise and tends to obscure more interesting warnings.
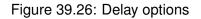
For an explanation of the remaining options for component recognition please refer to section 46.1[902]. The respective SUT script names for these options are:
OPT_PLAY_RECOGNITION_BONUS_NAME
OPT_PLAY_RECOGNITION_PENALTY_NAME
OPT_PLAY_RECOGNITION_BONUS_FEATURE
OPT_PLAY_RECOGNITION_PENALTY_FEATURE
OPT_PLAY_RECOGNITION_BONUS_EXTRAFEATURE
OPT_PLAY_RECOGNITION_PENALTY_EXTRAFEATURE
OPT_PLAY_RECOGNITION_BONUS_STRUCTURE
OPT_PLAY_RECOGNITION_PENALTY_STRUCTURE
OPT_PLAY_RECOGNITION_PENALTY_MODAL
OPT_PLAY_RECOGNITION_MINIMUM_PROBABILITY

39.3.5 Delays

Here values can be set for various delays.

Figure 39.26: Delay options

Default delays (ms) (System)

> Server script name: OPT_PLAY_DELAY_BEFORE, OPT_PLAY_DELAY_AFTER
> These two options set the delay in milliseconds before and after the execution of a node. If a node defines its own Delay before/after[538], its value overrides this default.

> These options are useful to slow a test down so you'll be able to follow it.

Drag&Drop and interpolation of mouse movement

Simulating Drag&Drop is non-trivial. It is made possible only by generating "hard" mouse events that actually move the mouse cursor. On Windows systems, some mouse drivers can interfere with these "hard" events. See section 47.1[908] for further details about Drag&Drop.

To make Drag&Drop as reliable as possible, movement of the mouse cursor is highly configurable. Since the requirements for Drag&Drop and hard mouse events differ from those for general mouse moves which only provide visual feedback, two sets of options are provided. The settings for demo mouse moves are ignored unless the respective option Actually move mouse cursor[468] is activated.

Typically movements for Drag&Drop and hard events should be slower and involve more interpolation steps than those for demo moves, which could slow down a test considerably. All of the following values influence the overall speed of mouse moves. A little experimentation may be required to get the desired effect.

Delay (ms) (System)

> SUT script name: OPT_PLAY_DND_DELAY, OPT_PLAY_MOVEMOUSE_DELAY
> After each single movement of the mouse cursor QF-Test will wait for the specified number of milliseconds. This value should be between 2 and 20 if interpolation is enabled and between 20 and 200 if interpolation is turned off. With interpolation, 10 is a good value for Drag&Drop and 5 for demo mouse moves.

Interpolation step size (System)

> SUT script name: OPT_PLAY_DND_STEP, OPT_PLAY_MOVEMOUSE_STEP
> The size of the steps for interpolation of mouse movement. Set this to 0 to turn interpolation off. Good values are between 1 and 3 for Drag&Drop and between 2 and 10 for demo mouse moves.

Acceleration (System)

> SUT   script   name: OPT_PLAY_DND_ACCELERATION,
> OPT_PLAY_MOVEMOUSE_ACCELERATION
> To avoid needless slowdown of tests, long distance mouse movement can be accelerated. A value of 0 turns off acceleration. Useful values range from 1 for very little acceleration to 10 or more for high acceleration. Good values are between 3 and 5 for Drag&Drop and between 6 and 20 for demo mouse moves.

Acceleration threshold (System)

> SUT   script   name: OPT_PLAY_DND_THRESHOLD,
> OPT_PLAY_MOVEMOUSE_THRESHOLD
> To ensure that small movements as well as the start end end of each movement remain precise, acceleration is turned off for movements that require less than this threshold's number of steps. Good values are between 4 and 8 for Drag&Drop and between 0 and 6 for demo mouse moves.

39.3.6 Timeouts

These timeouts are essential for reliable replay of tests under varying conditions. They define how long QF-Test waits for a component to be in the proper state for an event

before throwing an exception.

Don't make these values too small, so a little hiccup due to high load won't interrupt a test needlessly. QF-Test continues as soon as the conditions for replaying an event are met, so higher values for the timeouts won't slow down execution (except for focus, see below). On the other hand, don't set any values higher than a few seconds or you'll have to wait too long until you finally get an error message when a component is truly not found.

Timeouts

Deadlock detection (s)

120

Wait for GUI engine (ms)

5000

Wait for non-existent component (ms)

5000

Wait for non-existent item (ms)

2000

Default timeout for checks (ms)

500

Wait for modal dialog (ms)

5000

Wait for 'busy' GlassPane (ms)

3000

Wait for button/menu enable (ms)

3000

Wait for focus (ms)

20

Poll interval for component wait (ms)

300

Sub-item poll interval (ms)

300

Retry check interval (ms)

300

Figure 39.27: Timeout options

Deadlock detection (s) (System)

Server script name: OPT_PLAY_TIMEOUT_DEADLOCK

If the SUT does not react for the given time, a `DeadlockTimeoutException`[(852)] is thrown. Setting this value to 0 will suppress deadlock detection.

Wait for GUI engine (ms) (System)

SUT script name: OPT_PLAY_TIMEOUT_ENGINE

This option is useful only for multi-engine SUTs, like Eclipse with embedded AWT/Swing components. A Wait for client to connect[(670)] node finishes as soon as the first GUI engine connects to QF-Test, unless its GUI engine[(672)] attribute specifies to wait for a specific engine. To prevent a subsequent Wait for component to appear[(771)] node for a component of the wrong engine from failing immediately, QF-Test first waits for the time specified with this option to give the second GUI engine a chance to connect also.

Wait for non-existent component (ms) (System)

SUT script name: OPT_PLAY_TIMEOUT_COMPONENT

The maximum time in milliseconds that QF-Test waits for the target component of an event to become visible. When the connection between QF-Test and the SUT is established, this option is temporarily set to at least 30000 ms so as to allow the SUT time for its initialization.

Wait for non-existent item (ms) (System)

SUT script name: OPT_PLAY_TIMEOUT_ITEM

If an event is targeted on a sub-item of a complex component, QF-Test first waits for the component to become visible. Then it gives the SUT the chance to make the intended sub-item available before this timeout is exceeded.

Default timeout for checks (ms) (System)

Server script name: OPT_PLAY_CHECK_TIMEOUT

This option defines a default timeout for Check nodes that have no Timeout[(711)] attribute set and that represent an actual check in the report instead of being used for test control flow, i.e. checks that don't throw an exception and don't set a result variable or that have a @report doctag.

If your tests include a lot of Check nodes without explicitly defined Timeout that are expected to fail - which is unlikely for the actual checks described above - you may be able speed up test execution by setting this value to 0. However, it would be preferable to set the Timeout attribute of the respective nodes to 0 instead and leave this option unchanged because it increases general stability of check execution.

Wait for modal dialog (ms) (System)

SUT script name: OPT_PLAY_TIMEOUT_MODAL

If an event is targeted at a window that is blocked by a modal dialog, a `ModalDialogException`[(851)] will be thrown. However, modal dialogs are often temporary, informing the user about some ongoing processing. If this option is set to a non-zero value, QF-Test will wait for the given time before it throws the exception. If the modal dialog disappears before the time limit is exceeded, the test will continue immediately. This greatly simplifies handling of temporary modal dialogs.

Note

If the option Convert opening of a window into Wait for component to appear[(771)] [(442)] is activated, recording a sequence of events during which a temporary dialog is displayed may result in a Wait for component to appear[(771)] node for that dialog. If the dialog is displayed for a short time only, it is best to remove such nodes to avoid timing issues. If the SUT employs temporary modal dialogs often it may be best to disable the option Convert opening of a window into Wait for component to appear[(771)] [(442)].

Wait for 'busy' GlassPane (ms) (System)

Swing

SUT script name: OPT_PLAY_TIMEOUT_GLASSPANE

As an alternative to temporary modal dialogs some applications employ a so called GlassPane together with a 'busy' mouse cursor (typically in the form of an hourglass) to inform the user that the application is busy. The GlassPane is an invisible component that covers an entire window. If an event is delivered to such a window, the GlassPane will typically intercept it, preventing normal event processing, which can throw a test run severely off course.

QF-Test handles this case automatically by waiting until the GlassPane disappears before delivering an event, performing a check, etc. If the timeout given in this option is exceeded and the GlassPane is still active, a `BusyPaneException`[(852)] is thrown.

If the value of this option is set to 0, GlassPane checking is disabled and the event is delivered regardless. A `BusyPaneException` is never thrown in this case.

The Wait for component to appear[(771)] node is a special case. When waiting for a component (not its absence) covered by a busy GlassPane, the Timeout[(774)] attribute of the node overrides this option and is used to wait for both, the appearance of the component and the removal of the busy GlassPane. Thus it is possible to handle cases where the SUT is expected to be busy for an exceptionally long time on an individual basis without changing the default timeout of this option.

Wait for button/menu enable (ms) (System)

SUT script name: OPT_PLAY_TIMEOUT_ENABLED
A MouseEvent on a button or menu item is simply ignored, if the component is not enabled. This could lead to unwanted side effects during a test, so QF-Test waits until the component is enabled or the specified timeout is exceeded. If the component does not become activated within the given time, a `DisabledComponentException`[(851)] is thrown unless the option Throw DisabledComponentException[(469)] is deactivated.

Wait for focus (ms) (System)

SUT script name: OPT_PLAY_TIMEOUT_FOCUS
If set, this timeout causes QF-Test to wait for a component to get the keyboard focus before it simulates any KeyEvents on it. This option actually can slow down a test noticeably if a component fails to get the focus, so don't set it higher than about 100. A good value is 20.

Poll interval for component wait (ms) (System)

SUT script name: OPT_PLAY_POLL_COMPONENT
When waiting for a component in the SUT to appear, QF-Test can't always rely on Java's event mechanism. Instead it has to repeatedly scan the SUT for the component. This option determines the interval between searches.

Sub-item poll interval (ms) (System)

SUT script name: OPT_PLAY_POLL_ITEM
When waiting for a non-existent sub-item in the SUT, QF-Test can't rely on the event mechanism. Instead it has to repeatedly search the complex component for the sub-item. This option determines the interval between searches.

Retry check interval (ms) (System)

SUT script name: OPT_PLAY_POLL_CHECK
If a Check[(707)] fails for which a Timeout[(711)] is defined, QF-Test repeatedly queries the component's state until either it matches the given values or the time is up. The interval to wait between queries is set with this option.

39.3.7 Backward compatibility

These options can re-set QF-Test to older behavior. Those settings have changed that much that QF-Test cannot keep backward compatibility over all versions.

Figure 39.28: Options for replay backward compatibility

Don't evaluate variable boolean attributes (before 4.2) (System)

4.2+

Server script name: OPT_PLAY_DONT_EVALUATE_BOOLEAN_OPTIONS
Expressions in values of attributes with variable boolean values will be evaluated by Jython since 4.2.0. Such attributes are attributes like the Replay as "hard" event attribute of Mouse event node or the Modal attribute of Window nodes.

Use old-style GNU regexps (before 3.1) (System)

Server (automatically forwarded to SUT) script name: OPT_USE_GNU_REGEXP
Since version 3.1 QF-Test uses the standard Java regular expression syntax by default. In case this causes problems with regular expressions in older tests, you can switch back to using the GNU regexp package via this option. For further information about regular expressions please see section 47.3[909].

Old style image grab (before 2) (System)

Swing

SUT script name: OPT_RECORD_CHECK_IMAGE_OLD_STYLE
When recording images for Check image[729] nodes, non-opaque components used to be recorded with a black background. Thus, the image could deviate from what the user was seeing. This error has been corrected and normally the background is now recorded correctly. If this option is activated, the old, broken mechanism is used instead which may be useful if many checks for non-opaque components have already been recorded using that method.

39.4   SmartID und qfs:label

7.0+

The following settings define details for capture and replay of SmartIDs and qfs:label* variants. Please see section 5.6[73] and section 5.4.4[67] for details.

Figure 39.29: SmartID und qfs:label-Optionen

SmartID recording (System)

6.0+

Server (automatically forwarded to SUT) script name: OPT_RECORD_SMARTID
If this option is active, SmartIDs will be recorded instead of components where possible.

Always record class for SmartID (System)

6.0+

Server (automatically forwarded to SUT) script name:
OPT_RECORD_SMARTID_CLASS
Prepending the class of the target component is optional for SmartIDs. This option determines whether classes are always prepended when recording SmartIDs or only if required for uniqueness. The option is active by default because - in addition to improved readability and clarity - having the class in a SmartID improves replay performance significantly.

Always record qualifier for SmartID (System)

7.0+

Server (automatically forwarded to SUT) script name:
OPT_RECORD_SMARTID_QUALIFIER
This option determines whether the qualifier for a SmartID gets recorded or not.

In the following cases the qualifier will always be rercorded, regardless of the setting:

- If the SmartID is a qfs:label* variant and the option <u>Recording of qfs:label* variants</u>[486] is set to either "Record all variants" or "Record specific only".

- If the SmartID is based on an extra feature not belonging to the qfs:label* variants and the option <u>Priority for recording SmartIDs with qualifier</u>[485] has been set to record the extra feature.

- If the option <u>Priority for recording SmartIDs with qualifier</u>[485] deviates from its default value and the recorded SmartID is not based on the name.

Priority for recording SmartIDs with qualifier (System)

7.0+

Server (automatically forwarded to SUT) script name: OPT_RECORD_SMARTID_PRIORITIES

This comma-separated list of qualifiers determines the order in which criteria for component recognition are applied when recording SmartIDs. The default value of "name,qlabel,feature" tells QF-Test to first look for a name and use it for the SmartID, if available. Next is the test for a qfs:label*-variant and last for a feature. See <u>SmartID</u>[73] for a list of available qualifiers and their meanings.

Note For replay of a SmartID with no explicit qualifier the order is the same as the default value. Thus the search starts by looking for a named component only, then for qfs:label* and feature, which are taken as equivalent and implicitly combined.

Maximum length for recorded SmartID value (System)

7.0+

Server (automatically forwarded to SUT) script name: OPT_SMARTID_MAX_VALUE_LENGTH

In rare cases, the feature or associated label of a component can be very long. This is not a problem as such and often goes unnoticed if stored in a <u>Component</u>[823] node, but it can be rather unwieldy in a SmartID. For ease of use, values longer than the value given in this option are automatically converted into a regular expression of the given length.

Use SmartID instead of QPath for components inside items (System)

7.0+

Server (automatically forwarded to SUT) script name: OPT_RECORD_SMARTID_INSTEAD_OF_QPATH

A component inside an item, e.g. a CheckBox inside a table cell, has to be represented as a pseudo item with special syntax. Starting with QF-Test version 7, SmartID has replaced the outdated QPath for this task. If this option is

deactivated, QPath will be used except when general SmartID recording is active.

Recording of qfs:label* variants (System)

SUT script name: OPT_RECORD_QFSLABEL_MODE
Possible Values: VAL_RECORD_QFSLABEL_MODE_ALL,
VAL_RECORD_QFSLABEL_MODE_SPECIFIC,
VAL_RECORD_QFSLABEL_MODE_BEST,
VAL_RECORD_QFSLABEL_MODE_LEGACY
This option determines which qfs:label* variants are getting recorded as
Extra features[825]:

- The value "Record all variants" causes all `qfs:label*` variants found to be stored in the Extra features. The best label found gets the state "Should match", all others the state "Ignore". For a SmartID[73] the specific qualifier is recorded (see table `qfs:label*` variants[68]).

- With the value "Record specific only", only the best rated qfs:label* variant is stored in the Extra features or used as the qualifier in a SmartID.

- The value "Record qfs:labelBest only" tells QF-Test to only record the best rated qfs:label* variant and store it in the Extra features with the name `qfs:labelBest`, see Best label[69]. When recording the best label as SmartID with "Record qfs:labelBest only", depending on the setting of the option Always record qualifier for SmartID[484], either no qualifier will be recorded or the specific qualifier of the label.

- If set to "Legacy qfs:label mode", the associated label is determined via the old algorithm from before QF-Test 7.0. It is stored as `qfs:label` in the Extra features. For a SmartID the qualifier `qlabel` is used, but not recorded by default ().

39.5 Android

The following options influence the testing of Android applications.

Figure 39.30: Options for Android

Android SDK (System)

> Server script name: OPT_ANDROID_SDK_PATH
> Enter the installation path of the Android SDK here. This directory is usually named `sdk` and contains a subdirectory `tools` or `cmdline-tools`.
>
> This setting is only neccessary if QF-Test can not determine the path automatically.

Log extra android information to run log (System)

> Server script name: OPT_ANDROID_DEBUG
> If this option is activated further information is logged in the log.

Forward emulator output to QF-Test terminal (System)

> Server script name: OPT_ANDROID_FORWARD_EMULATOR_OUTPUT
> If this option is activated and an emulator was launched via the Launch Android emulator node, then the stdout/stderr output of the launched emulator will be passed on to the QF-Test terminal.

39.5.1   Recording Window

6.0+

The following options can be used to customize the look and functionality of the Android recording window.
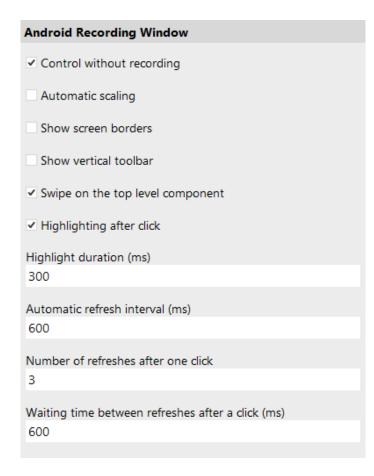
Figure 39.31: Options for the Android recording window

Control without recording (System)

Server script name: OPT_RECORDING_CONTROL_STATE
If this option is set, clicks and other inputs in the Android recording window are passed on to the active device or emulator, even if the recording mode is not active.

Automatic scaling (System)

Server script name: OPT_RECORDING_DISPLAY_AUTO_SCALING
If this option is set, the preview image in the Android recording window will be automatically scaled according to the window size.

Show screen borders (System)

Server script name: OPT_SHOW_BORDER
Draws a Border around the edge of the virtual screen inside the Android recording window to make the edge clearly visible.

Show vertical toolbar (System)

Server script name: OPT_VERTICAL_TOOLBAR_STATE
Inserts an additional toolbar at the edge of the Android recording window containing actions for controlling possible hardware buttons of the device. In recording mode, these are also recorded as events.

Swipe on the top level component (System)

Server script name: OPT_SWIPE_ON_TOPLEVEL_COMPONENT
When the option is activated swipes will always be recorded on the top level component.

Due to differing screen sizes and resolutions of devices some components may not be located in the visible area. Then, swipes where the lower level component is not relevant may become unreliable. Especially for navigation swipes, the option can improve the recording.

Highlighting after click (System)

Server script name: OPT_CLICK_HIGHLIGHT
When the option is set clicks to recording window will highlight the respective component, showing a colored border around it for a short time. This can be useful to check whether a click was interpreted correctly.

Highlight duration (ms) (System)

Server script name: OPT_HIGHLIGHT_DURATION
With this you control how long a border around a selected component should be displayed.

Automatic refresh interval (ms) (System)

Server script name: OPT_RECORDING_AUTO_REFRESH_INTERVALL
This controls how often QF-Test will try to refresh the contents of the Android recording window.

Please be aware that the maximum refresh speed depends on the device or emulator used. A interval value lower than the maximum refresh speed will not have any effect.

Too low values may negatively impact system performance.

Number of refreshes after one click (System)

Server script name: OPT_REFRESH_STEPS
The option defines the number of times the recording window will be updated after a click. The option is only relevant when automatic updating has been disabled.

Depending on the configurable animation speed of Android it can happen that the preview window may be updated whilst an animation is executed. In that case the option can be useful.

Waiting time between refreshes after a click (ms) (System)

Server script name: OPT_INTERVAL_TIME_AFTER_CLICK
The option defines the time between two refreshes. The option is only relevant when automatic refreshing has been disabled.

Please be aware the maximum refresh speed depends on the device or emulator used. A interval value lower than the maximum refresh speed will not have any effect.

39.6   Web options

Web

The following options are used specifically for web testing.

Figure 39.32: Web options

Turn 'ID' attribute into name where "unique enough" (System)

> SUT script name: OPT_WEB_ID_AS_NAME
> Use the ID of a DOM node as the component name, provided the ID is sufficiently unique. Uniqueness is determined per node type and depending on the setting of the options Name override mode (replay)[(472)] and Name override mode (record)[(450)].

Eliminate all numerals from 'ID' attributes (System)

SUT script name: OPT_WEB_SUPPRESS_NUMERALS
Of course this option only changes the way QF-Test treats ID attributes. The attributes themselves are left unchanged or the application would most likely no longer work.

With this option activated QF-Test removes all numerals from 'ID' attributes to prevent problems caused by automatically generated IDs often found in Ajax frameworks like GWT. Such dynamic IDs can change after the slightest modification to a web application which causes tests to break, especially if names are based on IDs. By removing the dynamic part from such IDs they become less useful, because they are no longer unique, but also less harmful. Uniqueness of names is taken care of by QF-Test. Since IDs also serve as a basis for Feature[(825)] and Extra features[(825)] attributes, this option is helpful even if IDs are not used as names.

Limit URL feature of 'Web page' node to host or file (System)

SUT script name: OPT_WEB_LIMIT_URL
If this option is active, all pages coming from the same host are recorded as the same page by reducing the URL feature to the host part of the URL. This is often useful when pages share a common look and navigation structure.

For file URLs, the URL is reduced to the filename, with intermediate directories removed.

Retarget mouse event on trivial node to parent (System)

SUT script name: OPT_WEB_RETARGET_MOUSE_EVENT
When recording mouse events on DOM nodes in a web page it is often useful to ignore "trivial" nodes and concentrate on the important ones. For example, when clicking on a text hyperlink it is typically not of interest whether part of the link is formatted with a bold font. It is the link that is important.

If this option is active QF-Test does not simply record the event for the deepest DOM node under the mouse cursor. Instead it moves up the hierarchy until it finds an "interesting" node. In the example above, QF-Test would record the event on the A node with the option active and on the contained B node otherwise.

Tolerate intermediate parent components (System)

SUT script name: OPT_WEB_TOLERATE_INTERMEDIATE_PARENT
Normally QF-Test's component recognition is tolerant to changes in the component hierarchy. For web pages with deeply nested tables this can lead to performance problems because the potential variants of determining the target component grow exponentially with the nesting depth. If you experience such

problems, try to deactivate this option. It will reduce adaptability but should help with performance.

The by far preferable solution is to set unique ID attributes for the different tables and other components so that QF-Test's name override mechanism can apply. This not only speeds up recognition drastically, it is also much more reliable and tolerant to change.

Take visibility of DOM nodes into account (System)

SUT script name: OPT_WEB_TEST_VISIBILITY
Similar to AWT/Swing or SWT, QF-Test normally only recognizes visible DOM nodes as target components. However, visibility of DOM nodes is not as well defined as that of components in a Java GUI. For example it is possible that an invisible DOM node has visible child nodes. Also, if a web page contains illegal HTML constructs it is possible that a DOM node is considered invisible, even though it is displayed in the browser window. If you come across such a problem you can turn off this option.

Let the browser determine the target element for check recording (System)

SUT script name: OPT_WEB_CHECK_VIA_BROWSER
When recording checks, components or procedures QF-Test needs to determine the target element under the mouse cursor. In case of overlapping nodes there are two different ways for calculating which one should be used. By default QF-Test lets the browser decide, which is usually the best choice. Since the different browsers don't always behave in the same reliable way, this option can be turned off in case of problems to use the older mechanism based on the z-order of elements instead. This option has no effect on check replay.

Stabilize event recording by displaying an overlay (User)

SUT script name: OPT_INTERACTION_OVERLAY_MODE
Possible Values: VAL_INTERACTION_OVERLAY_MODE_NONE,
VAL_INTERACTION_OVERLAY_MODE_MUTATION,
VAL_INTERACTION_OVERLAY_MODE_TRACKER
During recording QF-Test generates component steps. To gather the required data QF-Test analyzes - supported by the registered resolvers - the website in parallel to the user interaction. Depending on the complexity of the website and the registered resolvers this analysis might take some moments. With this option QF-Test can show a page overlay when it is busy analyzing the components for later recognition. This makes the process more transparent for the user.

How to handle errors in a web application (System)

SUT script name: OPT_WEB_JAVASCRIPT_ERROR_LEVEL
Possible Values: VAL_WEB_JAVASCRIPT_LEVEL_WARNING,
VAL_WEB_JAVASCRIPT_LEVEL_ERROR
Dynamic HTML is implemented via a lot of JavaScript code that is executed in the Browser. If an error occurs in such a script, browsers either ignore it or show an error dialog with some details about the error, depending on user setting. Many of these errors are harmless, others can be severe. QF-Test intercepts the error message and logs an error or warning in the run log instead. This set of options defines the severity of the message and a maximum for the number of such kinds of messages to log.

The possible choices for the option "Error level" are "Error" and "Warning". We advise that you set it to "Error" and make sure that such problems are reported to development and fixed sooner rather than later because they can represent a bug in the application you are testing. Known messages that are not going to be fixed by development can be excluded and ignored via the option Errors that should be ignored[(494)].

Maximum number of errors to log per SUT client (System)

SUT script name: OPT_WEB_JAVASCRIPT_MAX_ERRORS
In case a web page contains erroneous code it is possible that a lot of errors are triggered. Logging all those errors can significantly impact test performance, yet logging more than the first few errors does not really contribute much. The option "Maximum number of errors to log per SUT client" limits the possible number of error messages for this case.

Errors that should be ignored (System)

3.5+

SUT script name: OPT_WEB_JAVASCRIPT_ERROR_FILTERS
It is possible that some JavaScript errors cannot or will not be fixed, for example when they are coming from third-party code. In such a case it is preferable to ignore the known errors while still having QF-Test report unexpected ones. When the browsers reports a JavaScript error, QF-Test searches its error message for the occurrence of any of the regular expressions specified in this option. In case of a match, the error is ignored. If no exceptions are defined or none match, the error is reported in accordance with the previous options.

39.6.1 HTTP Requests

These options influence the behavior of HTTP requests.

HTTP Requests

☑ Record HTTP Request as browser request

☑ Synchronize with HTTP requests

Timeout for HTTP request synchronization (ms)

1000

Extended HTTP request timeout for new documents (ms)

5000

Figure 39.33: Options for HTTP Requests

Record HTTP Request as browser request (System)

4.1+

SUT script name: OPT_WEB_RECORD_CLIENT_REQUEST_STEP
When recording HTTP Requests a Browser HTTP request[808] is created. This
request will be submitted directly via the browser so that the response is shown
afterwards and test execution can be continued directly in the browser. By
deactivating this option a Server HTTP request[802] will be recorded. This request will
be submitted by QF-Test and doesn't have any effect on the browser. The
response is only accessible in QF-Test.

Synchronize with HTTP requests (System)

4.1+

SUT script name: OPT_WEB_TRACK_HTTP_REQUESTS

Note
HTTP request tracking is supported for browsers in QF-Driver or CDP-Driver
mode only. It does not apply to WebDriver based tests or to browsers embedded
in Java like WebView.

Because most things in JavaScript-based web applications run asynchronously,
one of the main challenges in testing such applications is timing. QF-Test uses
various means to synchronize with the SUT and this option controls one of them.
If turned on, QF-Test will keep track of all HTTP requests the browser sends to the
server. Before and after replaying an event, QF-Test will wait until no requests are
outstanding. The following two options Timeout for HTTP request synchronization

(ms)[495] and Extended HTTP request timeout for new documents (ms)[496] are used for fine-tuning this feature.

Timeout for HTTP request synchronization (ms) (System)

4.1+

SUT script name: OPT_WEB_HTTP_REQUEST_TIMEOUT
When synchronizing with the SUT by tracking HTTP requests as explained for the option Synchronize with HTTP requests[495], QF-Test cannot wait indefinitely for outstanding requests as this would impact test performance too much. This option defines the maximum time to wait for outstanding requests in normal situations and the following option Extended HTTP request timeout for new documents (ms)[496] is used right after a page has finished loading.

Extended HTTP request timeout for new documents (ms) (System)

4.1+

SUT script name: OPT_WEB_HTTP_REQUEST_TIMEOUT_DC
When synchronizing with the SUT by tracking HTTP requests as explained for the option Synchronize with HTTP requests[495], QF-Test cannot wait indefinitely for outstanding requests as this would impact test performance too much. Right after a page has finished loading, JavaScript applications tend to send a lot of requests and may take a while to build the final user interface via JavaScript. This options defines the maximum time to wait for outstanding requests at that point. The previous option Timeout for HTTP request synchronization (ms)[495] is used to set the standard timeout for other situations.

39.6.2   Backward compatibility

These options can re-set QF-Test to older behavior. Those settings have changed that much that QF-Test cannot keep backward compatibility over all versions.

Figure 39.34: Options for web backward compatibility

Don't fetch text for generic classes with generic class approach (before 4.2) (System)

SUT   script   name:
OPT_WEB_TEXT_DONT_TRAVERSE_ALL_NODES_FOR_GENERICS
In versions older than 4.2.0 Check text⁽⁷⁰⁸⁾ and Fetch text⁽⁷⁴⁰⁾ nodes provided too much or too less text for components of generic classes in some cases. Especially `SELECT` or `TableCell` components containing text-fields returned wrong text. Now the any child component with a generic class will be taken into account.

Treat underscores like blanks in extra features (before 4.2) (System)

SUT script name: OPT_WEB_TREAT_UNDERSCORES_AS_BLANKS_IN_EF
Older QF-Test versions than 4.2.0 turned all underscores into blanks once Extra features were compared. This behavior could cause troubles in case you were really searching for underscores.

Reset web-document load state during rescan (before 5.4) (System)

SUT   script   name:
OPT_WEB_RESET_DOCUMENT_KNOWN_STATE_DURING_RESCAN
Before version 5.4 QF-Test had a bug that caused an inadvertent reset of the internal loading state of a document.   In the same context, documents contained in frames were not handled correctly.   Both could lead to a Wait for document to load⁽⁷⁷⁶⁾ succeeding solely because a document existed without regard for whether it was loaded anew or not.

Since QF-Test version 5.4 document reload is checked more precisely again. This might lead to new errors in case existing tests contain too many or misplaced Wait for document to load⁽⁷⁷⁶⁾ nodes. In this case, it is possible to restore the previous behavior by activating this option, instead of correcting the errornous test suites.

39.7   SWT options

The following options are used specifically for SWT testing.

Figure 39.35: SWT options

Connect without SWT instrumentation (System)

4.5+

Server script name: OPT_PLAY_SWT_VIA_AGENT
With this option activated there is no need to instrument SWT based applications, except for older SWT versions on Linux. For detailed technical information please see section 45.2[(900)].

Preferred GTK version for SWT (Linux only) (System)

4.5+

Server script name: OPT_PLAY_SWT_PREFERRED_GTK_VERSION
On Linux systems Eclipse/SWT applications with SWT versions 4.3 through 4.9 can be run either in GTK2 or GTK3 mode. Older version support GTK2 only, newer version GTK3. This option can be set to "2" or "3" in order to enforce a specific GTK version or left empty to use the default for the respective SWT version.

Activate XSync for SWT with GTK2 (Linux only) (System)

4.5+

Server script name: OPT_PLAY_SWT_GTK2_XSYNC
SWT with GTK2 has become unstable on newer Linux systems and can crash under heavy load which is not uncommon when driven by QF-Test at full speed. A fix for this is to turn on XSync, an option specific to X11 that causes X11 events to be synchronized. This can have a performance impact however, so if you need to run your SWT application with GTK2 and it appears to be slow, you can try to deactivate this option to see if this speeds things up without causing the SUT to crash occasionally.

39.8 Debugger options

These options modify the behavior of the debugger.

Figure 39.36: Debugger options

Enable debugger (User)

By default the debugger is disabled unless this option, which can also be modified through the menu item ⟦Debugger→Enable debugger⟧, is activated.

If a test is interrupted by a breakpoint or be pressing the "Pause" button, the debugger is activated automatically. Similarly, starting a test run with "Step in" or "Step over" will activate the debugger for the duration of that test run.

Always open debugger window (User)

When test execution is halted and the debugger entered QF-Test can optionally open a separate window for the debugger. This option determines whether debugging should happen in a separate window or the normal test suite window.

Show variable bindings automatically (User)

When test execution is halted and the debugger entered QF-Test can display the current variable bindings in the workbench window. If this option is active the variables are shown automatically each time a test run first enters the debugger.

Alternatively the variables can be viewed in the debugger window or shown in the workbench window via the menu ‖Debugger→Show variables‖.

Automatic breaks (User)

These options describe the situations in which execution of a test will be suspended and the debugger entered:

Break on uncaught exception
  The debugger will break if an exception is thrown that will not be handled by a $\mathrm{Catch}^{(619)}$ node.

Break on error
  The debugger will break if an error occurs.

Break on warning
  The debugger will break if a warning occurs.

39.9   Run log options

These options let you control which information is collected in a run log, if and when a run log is shown and how to locate errors.

39.9.1   General run log options

Figure 39.37: General run log options

When to show run logs (User)

A run log is created for every execution of a test. A number of recent run logs are available from the ⟦Run⟧ menu, the most recent run log can also be opened by pressing ⟦Control-L⟧. Additionally the run log can be shown during execution or after an error as follows:

At start
This choice causes QF-Test to open the run log when it begins executing a test sequence. The nodes of the log will be added as execution proceeds.

After finish

With this choice the run log is shown after replay is finished.

On exception

The run log is shown only if an uncaught exception is thrown.

Don't show

The run log is not displayed automatically. You have to open it via the ⟦Run⟧ menu or by pressing ⟦Control-L⟧.

Show relative duration indicators (User)

6.0+

Server script name: OPT_LOG_DURATION_INDICATORS

To analyze the run-time behaviour of a test it is helpful to see at a glance which branches are taking up the most time. To that end, display of duration indicators can be activated via this option, which is also directly accessible via the ⟦View⟧ menu of a run log.

The length of the bars shown represents the percentage of the time taken for execution of the respective node relative to the time taken by its parent node.

Duration indicator kind (User)

6.0+

Server script name: OPT_LOG_DURATION_INDICATORS_KIND
Possible Values: VAL_LOG_DURATION_INDICATORS_KIND_DURATION,
VAL_LOG_DURATION_INDICATORS_KIND_REALTIME,
VAL_LOG_DURATION_INDICATORS_KIND_BOTH
This option determines whether duration indicators show duration, real time spent or both.

Note   The difference between the values of "Duration" and "Real time spent" are explicit delays introduced in nodes via the 'Delay before/after' attribute or user interrupts.

Number of run logs in menu (User)

A limit for the number of menu items for recent run logs kept in the ⟦Run⟧ menu.

Automatically save run logs (User)

3.0+

To prevent excessive memory use through run logs and also to make the most recent run logs persistent between QF-Test sessions, the recent run logs kept in the ⟦Run⟧ menu are saved automatically to the user configuration directory[10] or the directory defined in the option Directory for run logs[503]. The filename for the run log is based on a timestamp. QF-Test uses file locks to prevent collisions and accidental removal in case of parallel sessions and automatically keeps the user

configuration directory[10] clean by removing unreferenced logs, so there should be no reason to disable this feature. Still, you can do so by disabling this option.

Directory for run logs (User)

By default, run logs created during interactive use of QF-Test are stored in the user configuration directory[10]. This option can be used to specify a different target directory.

This option is interpreted by QF-Test whenever a test is started. At that point, test suite and global variable bindings are already in place and in contrast to other options it is possible to use QF-Test variable syntax here. This includes special variables like ${env:HOME} to look up an environment variable or even ${qftest:suite.dir} to save the run log next to the test suite. If the directory is dynamic, like in the latter case, QF-Test may not be able to clean up old run logs regularly. Errors in variable expansion are silently ignored and the user configuration directory[10] is used instead.

Show expanded variable values in tree nodes (User)

The nodes in the tree view of a run log can either be displayed with variables expanded to the value they had at run-time or with the original variables. Both views have their use, so you can toggle between them via this option or more quickly via the menu item │View→Show nodes expanded│.

Show return values of procedures (User)

If this option is active, return values of Procedures[586] are displayed in the tree after the respective Procedure call[589] node. In a run log you can also toggle this option's value via the menu item │View→Show procedure return values│.

Skip suppressed errors (User)

Like the previous one, this option controls the search for errors in a run log. If activated, warnings, errors or exceptions that have not propagated to the top, are not found. Thus exceptions caught by a Try[616]/Catch[619] clause or messages suppressed through the Maximum error level[537] attribute are skipped.

This option is also accessible through the │Edit→Skip suppressed errors│ menu item.

Cleanup tree when jumping to next or previous error (User)

When repeatedly jumping to errors in a run log the tree can easily get cluttered with many expanded nodes. If this option is activated, QF-Test will automatically clean up the tree each time you navigate to an error so that only the parents of the error node are expanded.

When viewing split run logs, partial run logs containing an error will remain in memory as long as their nodes are expanded. Keeping this option activated will ensure that partial run logs will be released as soon as possible, keeping memory use manageable even viewing the errors of a very large run logs.

Save compressed run logs (*.qrz) (System)

Server script name: OPT_LOG_SAVE_COMPRESSED
Run logs can either be saved as plain or as compressed XML files. For large run logs without screenshots the compression factor can be as high as 10, so it's advisable to use compressed logs where possible. The only reason not to use compression is if you want to transform the XML run log afterwards. But even then compressed run logs are an option because the compression method used is standard gzip format, so converting to and from compressed run logs can easily be done using `gzip`.

When saving a log-file interactively you can always switch between compressed or non-compressed format by choosing the appropriate filter or by giving the file the extension `.qrz` or `.qrl`.

In batch mode, the default run log format is compressed. To create an uncompressed run log, simply specify the extension `.qrl` in the parameter for the `-runlog [<file>]`[(878)] command line argument.

Save run log in current XML format with UTF-8 encoding (System)

Server script name: OPT_XML_LOG_NEW
Starting with QF-Test version 7.0, run logs are saved in the current XML format with UTF-8 encoding, no indentation and arbitrarily long lines. If you need to fall back to the old format (ISO-8859-1 encoding, 2 characters indentation, line length 78) for use in external tools you can do so by deactivating this option.

39.9.2   Options for splitting run logs

Figure 39.38: Options for splitting run logs

Create split run logs (User)

3.0+

Server script name: OPT_LOG_SAVE_SPLIT
A run log can be split into several parts by setting the Name for separate run log[(564)] attribute of a Data driver[(562)] or any of the various test nodes. By turning this option off you can temporarily disable support for split run logs in order to get a normal, single run log without having to modify any Name for separate run log attributes.

See section 7.1.6[(122)] for further information about split run logs.

Save split run logs as ZIP files (*.qzp) (User)

3.0+

Server script name: OPT_LOG_SAVE_SPLIT_ZIP
Split run logs can either be saved as a single ZIP file with the extension `.qzp`, containing the main run log and all partial logs together, or as a normal `.qrl` or `.qrz` run log that is accompanied by a directory with the same base name and the suffix `_logs`, e.g. the file `runlog.qrz` plus the directory `runlog_logs`. This option determines the format in which split run logs are created in interactive mode. It has no effect if the option Automatically save run logs[(502)] is turned off.

See section 7.1.6[(122)] for further information about split run logs.

Minimum size for automatic splitting (kB) (System)

3.4+

Server script name: OPT_LOG_AUTO_SPLIT_SIZE
This option exclusively applies to Test case and Test set nodes. At other places run logs are split only when Name for separate run log has explicitly been set.

Split run logs are the only reliable way to prevent running out of memory during very long running tests or when the run log grows quickly due to screenshots or output from the SUT. They are also more efficient when transforming run logs into reports. However, explicit setting of Name for separate run log attributes requires an understanding of the issues involved and either making decisions about where best to split a run log or tedious typing when trying to split into small pieces.

As a compromise, QF-Test makes a very rough calculation about the size of a run log during executing, taking screenshots and program output into account. When execution of a Test case or Test set has finished and the approximate size of the run log pertaining to that node is larger than the threshold specified in this option, the run log is split off and saved automatically. A value of 0 prevents automatic splitting.

See section 7.1.6[(122)] for further information about split run logs.

Name for automatically split 'Test case' run log (System)

3.4+

Server script name: OPT_LOG_AUTO_SPLIT_TESTCASE_NAME
This option specifies the name to use for an external log when it is split off automatically after execution of a Test case has finished as described in the previous option. Variables can be used as well as the '%...' placeholders documented for the attribute Name for separate run log[(521)].

The special variable ${qftest:testcase.splitlogname} is a good base. It expands to a path name created from the name of the Test case with possible parent Test set nodes as directories.

See section 7.1.6[(122)] for further information about split run logs.

Name for automatically split 'Test set' run log (System)

3.4+

Server script name: OPT_LOG_AUTO_SPLIT_TESTSET_NAME
This option specifies the name to use for an external log when it is split off automatically after execution of a Test set has finished as described in the option Minimum size for automatic splitting (kB)[(505)]. Variables can be used as well as the '%...' placeholders documented for the attribute Name for separate run log[(528)].

The special variable ${qftest:testset.splitlogname} is a good base. It expands to a path name created from the name of the Test set with possible parent Test set nodes as directories.

See section 7.1.6[(122)] for further information about split run logs.

39.9.3 Options determining run log content

Content of run logs

☑ Log variable expansion

☐ Log parent nodes of components

Log level for the SUT

Warnings and errors ⌄

Number of events to log for error diagnosis

0

Maximum number of errors with screenshots per run log

3

☑ Count screenshots individually for each split log

☑ Create screenshots of the whole screen upon error

☑ Limit screenshots to relevant screens

☑ Create screenshots of the client's windows upon error in client

☑ Create screenshots of all client windows upon error

☐ Create screenshots for warnings

☐ Log successful advanced image checks

☑ Compress images in run logs and test suites

☐ Create compact run log

☐ Don't create run log

☑ Log SUT output individually

☑ Compactify individually logged SUT output

☑ Log comments to run log

☑ Wrap lines in exception messages

Figure 39.39: Options determining run log content

Log variable expansion (System)

Server script name: OPT_LOG_CONTENT_VARIABLE_EXPANSION
If this option is activated, every variable expansion[103] is logged.

Log parent nodes of components (System)

Server script name: OPT_LOG_CONTENT_PARENT_COMPONENTS
Setting this option will cause all direct and indirect parent nodes to be logged in addition to the target component node for every event, check, etc.

Log level for the SUT (System)

Server script name: OPT_LOG_CONTENT_SUT_LEVEL
Possible Values: VAL_LOG_SUT_LEVEL_MESSAGE,
VAL_LOG_SUT_LEVEL_WARNING,
VAL_LOG_SUT_LEVEL_ERROR
The level for automatically generated messages in the SUT during replay, e.g. details for component recognition. Only messages with the respective level, i.e. plain messages, warnings or errors will be logged. This option has no effect on messages created explicitly via the Message node, `rc.logMessage` or `qf.logMessage`.

Number of events to log for error diagnosis (System)

SUT script name: OPT_LOG_CONTENT_DIAGNOSIS
During replay of a test QF-Test logs various events and other things going on behind the scenes. This information is quickly discarded except when an error happens. In that case the most recent events are written to a special run log node. The information may also be useful to developers but is mostly required for error diagnosis when requesting support from Quality First Software GmbH.

This option determines the number of recent internal events to keep. Setting it to 0 disables the feature altogether. You should not set this value to less than about 400 without a good reason. Because the information is logged only for errors, the cost for gathering it is minimal.

Maximum number of errors with screenshots per run log (System)

Server script name: OPT_LOG_CONTENT_SCREENSHOT_MAX
The maximum number of screenshots that QF-Test takes and stores in the run log during a test run on situations of exception or errors. Setting this value to 0

disables taking screenshots entirely, a negative value means unlimited screenshots.

Count screenshots individually for each split log (System)

Server script name: OPT_LOG_CONTENT_SCREENSHOT_PER_SPLIT_LOG
If this option is set, each partial log of a split run log may contain the maximum number of screenshots defined above without affecting the count for the main run log. Otherwise the limit applies for the sum of all parts belonging to the same main run log.

See section 7.1.6[122] for further information about split run logs.

Create screenshots of the whole screen upon error (System)

Server script name: OPT_LOG_CONTENT_SCREENSHOT_FULLSCREEN
Activating this option causes QF-Test to take an image of the whole screen and save it in the run log when a screenshot is triggered by an exception or error.

Limit screenshots to relevant screens (System)

6.0+

Server script name: OPT_LOG_CONTENT_SCREENSHOT_RELEVANT_ONLY
If multiple monitors are connected when taking screenshots it may not be desirable to take screenshots of all screens, especially on personal workstations where this might expose private or confidential information.

If this option is active (the default), QF-Test will try to determine the screens on which SUT or QF-Test windows are showing and exclude the rest.

Create screenshots of the client's windows upon error in client (System)

Server script name: OPT_LOG_CONTENT_SCREENSHOT_WINDOW
Activating this option causes QF-Test to record images of all windows and dialogs of the SUT and store them in the run log when screenshots are triggered due to exceptions or errors coming from that SUT. In many cases this will work even for windows that are covered by other windows or in cases where a full screenshot is not possible, for example when a screen is locked.

Create screenshots of all client windows upon error (System)

4.3+

Server script name: OPT_LOG_CONTENT_SCREENSHOT_ALL_WINDOWS
If this option is active QF-Test will log images of all windows of all connected SUTs for any exception or error, regardless of where the exception or error is coming from. This option also affects which screens are considered relevant, depending on where SUT windows are showing.

Create screenshots for warnings (System)

6.0+

Server script name: OPT_LOG_CONTENT_WARNING_SCREENSHOTS
If this option is active, screenshots will also be saved when warnings are logged
in the run log. Otherwise this is done for errors and exceptions only.

Log successful advanced image checks (System)

3.4+

Server   script   name:
OPT_LOG_CONTENT_SCREENSHOT_ADVANCED_ALWAYS
If this option is activated, QF-Test will store the expected and actual images as
well as the transformed images for successful advanced image checks in the run
log. Otherwise these details are kept for failed image checks only.

Activating this option can raise the size of the run log drastically so be sure to use
it in combination with compact run logs and/or split run logs.

Compress images in run logs and test suites (System)

4.5+

Server script name: OPT_LOG_CONTENT_COMPRESS_SCREENSHOTS
If this option is activated, QF-Test will store new images in test suites and run
logs losslessly compressed.

This option can reduce the memory consumption of protocols and test suites on
disk and in memory significantly. On the other hand, compression and decom-
pression requires some CPU time, so the option can be deactivated for very time-
critical test executions.

Create compact run log (System)

Server script name: OPT_LOG_CONTENT_COMPACT
Activating this option causes QF-Test to discard every node from a run log that is
neither relevant for error diagnosis, nor for the XML/HTML report. After an error
or exception, as well as at the end of a test run, the 100 most recent nodes are
not discarded, so the most relevant information should remain available.

Even large tests should not cause memory issues, provided the option <u>Create split</u>
<u>run logs</u>[505] is turned on and used as described in <u>Split run logs</u>[122]. But if you do
run out of memory, activating this option can be useful.

This option is used only when QF-Test is run in interactive mode. It is ignored in
batch mode (see <u>section 1.7</u>[11]) to avoid accidental loss of information. To create
a compact run log in batch mode, use the <u>`-compact`</u>[870] command line argument.

Don't create run log (System)

Server script name: OPT_LOG_CONTENT_SUPPRESS

For very-long-running tests or demos that are run in an endless loop, memory consumption of the run log is an issue, but split run logs are an ideal solution. Before split run logs were available, turning run logs off completely via this option was sometimes the only way to get long-running tests to work. Now this option is only retained for backwards compatibility.

In batch mode this option is ignored. To suppress the run log, use the argument -nolog[(874)].

Log SUT output individually (System)

3.0+

SUT script name: OPT_LOG_CONTENT_SPLIT_IO

If set, any text that an SUT client prints to its stdout or stderr stream is also logged in the run log. For each interaction with the SUT QF-Test collects text printed before the event and after the event during synchronization. This makes it possible to associate output like an exception stacktrace that is triggered by an event with the event itself, something that is impossible if all output is kept in a single piece.

Compactify individually logged SUT output (System)

3.0+

Server script name: OPT_LOG_CONTENT_COMPACTIFY_SPLIT_IO

Output from an SUT client tends to accumulate and can consume a lot of memory. If this option is activated, individually logged SUT output for events that are no longer of interest can be removed along with the events in compact run logs. Please see the option Create compact run log[(510)] for further information about compact run logs.

Log comments to run log (System)

5.0+

Server script name: OPT_LOG_COMMENTS_TO_RUNLOG

If this option is activated, Component[(823)] steps are added to the run log.

Wrap lines in exception messages (System)

5.3+

Server script name: OPT_LOG_WRAP_EXCEPTION_MESSAGE

If activated, display of exceptions messages in the run log or an error dialog are displayed using word-wrap to break long lines.

39.9.4   Options for mapping between directories with test suites

4.0+

Figure 39.40: Options for mapping between directories with test suites

Directory map for test suites (System)

When analyzing a run log it is often necessary to quickly switch between the run log and the respective test suite. However, when running automated tests on different systems like Windows and Linux, the directories from which test suites are loaded during the test vary and there is no automatic way to map between different directory layouts.

With this option you can assist QF-Test in locating test suites. The 'From' column is a glob pattern that must match from the beginning of the path of the test suite stored in the run log to the end of some directory in that path. The 'To' column is the respective replacement for the matching part and can also contain a glob pattern. When searching for the test suite, QF-Test processes this list top to bottom, performing the replacement for every match found and the first match leading to an actual test suite is used.

Note   A glob patterns is a simpler form of a regular expression often used by development tools: An '*' stands for any number of characters up to the next file separator while '**' means 0 or more characters of any kind, including '/'. Some examples:

**/test-suites
     All directories named `test-suites` at any depth.

T:/test/sut_*
     All directories starting with `sut_` in the `T:/test` directory.

39.10   Variables

The following options pertain to variable binding.
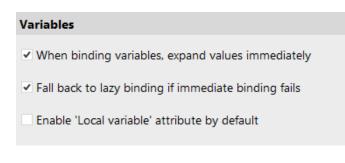
Figure 39.41: Variable options

When binding variables, expand values immediately (System)

3.0+

Server script name: OPT_VARIABLE_IMMEDIATE_BINDING
When a set of variable bindings is pushed onto a variable stack, any additional variable references in the values of these variables can either be expanded immediately, or the values can be left unchanged to be expanded lazily as needed. This is explained in detail in section 6.7[114].

For immediate expansion turn this option on, for lazy expansion turn it off.

Fall back to lazy binding if immediate binding fails (System)

3.0+

Server script name: OPT_VARIABLE_LAZY_FALLBACK
Old tests that rely on lazy variable binding may fail with the new default of immediate binding. If this option is activated, variables that cannot be bound immediately due to references to not-yet-defined variables are bound lazily instead and a warning is issued. To get rid of the warning simply change the value of the respective variable to use explicit lazy binding with '$_'. Please see section 6.7[114] for further information.

Enable 'Local variable' attribute by default (System)

5.1+

Many nodes have a 'Local variable' attribute that determines whether a result variable should get stored locally or as a global variable. If this option is activated, newly created nodes of that kind will have the 'Local variable' attribute set.

Furthermore, various kinds of variables can be defined here. These are explained in chapter 6[103].

39.11 Runtime only

Some options of a more technical nature are not available via the user interface, especially if they are useful only in very specific cases and will do more harm than good in general.

As mentioned in the introduction to this chapter, options can be set in Server script[628] or SUT script[631] nodes via `rc.setOption(Options.<OPTION_NAME>, <value>)` (see section 48.5[917] for details). Which kind of step to use is indicated by the text "Server script name" or "SUT script name" in the documentation of the respective option.

Connect via QF-Test agent (System)

4+

> Server script name: OPT_PLAY_CONNECT_VIA_AGENT
> QF-Test version 4 introduced a new mechanism for connecting to an SUT, based on Java agents. It is far more powerful and flexible than the older mechanisms and even a hard requirement for most current SUTs. Thus it should not be turned off without a very good reason.

Note

> Without the agent an SUT based on Java 9 or newer will not even start. Besides, the agent is a prerequisite for access to JavaFX, embedded browsers and the full functionality of live unit testing.

Instrument AWT EventQueue (System)

4.1+

> Server script name: OPT_PLAY_INSTRUMENT_EVENT_QUEUE

Swing

> In earlier versions QF-Test replaced the AWT system EventQueue with its own, taking pains to handle custom EventQueues and many other subtle details. Starting with QF-Test 4.1 the default is to instrument the EventQueue class instead which has less impact on the SUT, handles border cases well and is preferable in almost all situations. In case of connection problems, the old mechanism can be used by turning off this option.

Note

> Instrumenting the AWT EventQueue is only possible with the QF-Test agent.

Chapter 40

Elements of a test suite

40.1 The test suite and its structure

There are more than 60 different kinds of nodes for a test suite which are all listed in this chapter. Each node type has a unique set of features. The attributes of a node are displayed and edited in the detail view of the editor. The restrictions that apply to each attribute are listed as well as whether it supports variable expansion (see chapter 6[(103)]).

Additional features of a node include the behavior of the node during execution of a test and the kinds of parent and child nodes allowed.

40.1.1 Test suite

The root node of the tree represents the test suite itself. Its basic structure is fixed. The root node contains an arbitrary number of Test sets[(525)] or Test cases[(518)], followed by the Procedures[(595)] the Extras[(547)] and the Windows and components[(835)]. When executed, the top-level Test sets are executed one by one.

Contained in: None

Children: An arbitrary number of Test set or Test case nodes, followed by the Procedures, Extras and Windows and components nodes.

Execution: The top-level Test set nodes are executed one by one.

Attributes:
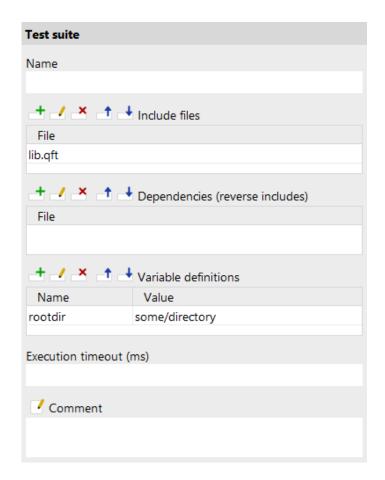
Figure 40.1: Test suite attributes

Name

A kind of short description for the test suite. The name is displayed in the tree view, so it should be concise and tell something about the function of the test suite.

Variable: No

Restrictions: None

Include files

This is a list of test suites that are included by the suite. If a Component[(823)] or Procedure[(586)] reference cannot be resolved in the current suite, the included suites are searched for it. When recording new components, QF-Test will search the included suites for a matching Component node before creating a new one.

Relative pathnames are treated relative to the directory of the suite, or to a directory on the library path (see option Directories holding test suite libraries[(436)]).

When you change anything in this attribute QF-Test will offer to update all affected nodes to compensate for the change. For example, if you add or remove a suite from the includes, QF-Test will make all references to that suite's Procedures or Components implicit or absolute so that the actual referenced nodes remain unchanged. In such a case, choose "Yes". If, on the other hand, you renamed a suite or moved it to some other directory and are simply updating the includes to reflect that, chose "No" so all former implicit references into the old suite will now point to the new one.

Variable: No

Restrictions: None

Dependencies (reverse includes)

This list of test suites is the reverse of the Include files attribute. It has no impact on test execution. Instead it serves a hint to tell QF-Test which test suites depend on Components in this suite, either because they (directly or indirectly) include this suite or because they explicitly reference Components in it. This information is used when QF-Test IDs of Components are manipulated (for example after updating components, see (section 5.11.2[(94)])) and the QF-Test component ID attributes of nodes depending on these components have to be updated. QF-Test always checks all currently loaded suites for dependencies. Additionally, it will automatically load and check all suites listed here.

Relative pathnames are treated relative to the directory of the suite, or to a directory on the library path (see option Directories holding test suite libraries[(436)]).

Note Like the Include files, the Dependencies are also resolved indirectly. For example if suite A has the dependency B which has the dependency C, both suites B and C will be loaded and checked for references automatically when components in suite A are manipulated.

Variable: No

Restrictions: None

Variable definitions

These variables definitions are identical to the suite variable bindings accessible in the Variables[(512)] pane of the global options. A detailed explanation of variable definition and lookup is given in chapter 6[(103)]. See section 2.2.5[(17)] about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Execution timeout

Time limit for the node's execution in milliseconds.   If that limit expires the execution of that node will get interrupted.

Variable: Yes

Restrictions: >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the ✏ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.2   Test and Sequence nodes

Tests and sequences are the main structural elements of a test suite. Test case[518] nodes represent logical test cases and are implemented as specialized sequences. A sequence is a container that executes its child nodes one by one. It can define variables (see chapter 6[103]) that remain bound during the execution of the children.

Other kinds of sequences differ either in the way their child nodes are executed, or in the restrictions they impose on their child or parent nodes.

40.2.1   Test case

A Test case node represents one or more logical test cases. In a sense it is the most important of all QF-Test nodes and everything else only serves to lend structure to Test cases or to implement their logic. Functionally it is a highly specialized Sequence[536] with a number of important extensions and special attributes.

A Test case should focus on the actual test it supposed to perform. Setup and cleanup tasks required to ensure that the Test case executes in the required environment and does not interfere with subsequent tests should be implemented in the form of <u>Dependencies</u>[(548)] as described in <u>section 8.6</u>[(138)]. Alternatively - or in addition to Dependencies - a Test case can have <u>Setup</u>[(554)] and <u>Cleanup</u>[(557)] nodes to be executed before and after the Test case.

Because a Test case can be called from a <u>Test call</u>[(531)] node it is also somewhat similar to a <u>Procedure</u>[(586)] in that its <u>Name</u>[(521)] attribute is mandatory and that it has a list of <u>Parameter default values</u>[(523)] that can be overridden in the calling node.

Test cases also play a central role in run logs and test reports. During a test run a Test case node can be executed several times in different contexts and with different parameters. Logically these executions may represent the same or different test cases. By defining a set of <u>Characteristic variables</u>[(521)] you can specify which variable values are used to differentiate between executions, thus characterizing the run-time environment of the test. The values of these variables at the time of entry to the Test case are stored in the run log. To emphasize the fact that each execution of a Test case node may represent a separate logical test case there is an alternative name attribute called <u>Name for run log and report</u>[(521)]. Its value may contain references to the Characteristic variables of the test. In the run log or report the test will then be listed with this name, including the expanded run-time variable values.

Finally, there are situations in which a test cannot or should not be executed for specific variable settings. If the <u>Condition</u>[(522)] attribute of the Test case is defined, the Test case will only be executed if that expression expands to a true value. If the Test case is not executed due to the Condition it will be listed as *skipped* in the report.

Contained in: <u>Test suite</u>[(515)], <u>Test set</u>[(525)].

Children: Optional <u>Dependency</u>[(548)] or <u>Dependency reference</u>[(551)] as the first element. A <u>Setup</u>[(554)] may be the next and a <u>Cleanup</u>[(557)] the last node with an arbitrary number of normal child nodes in between. A Test case that does not contain normal child nodes will be listed as *not implemented* in the report.

Execution: First the <u>Variable definitions</u>[(523)] of the Test case are bound on the primary and its <u>Parameter default values</u>[(523)] on the fallback variable stack (see <u>chapter 6</u>[(103)]). With these in place the <u>Condition</u>[(522)] is evaluated and the node will be skipped if a non-empty Condition evaluates to false. Next the dependency of the Test case - possibly inherited from its parent node - is determined and resolved as described in <u>section 8.6</u>[(138)]. Then the optional Setup is executed once, followed by the normal child nodes and a single execution of the optional Cleanup. An exception raised during the course of the Test case will be caught and passed to the Dependency for handling in a <u>Catch</u>[(619)] node. Even if the exception is not handled by a Dependency it is not propagated beyond the Test case to prevent aborting the whole test run. The error state is duly noted in run log and report however.

Attributes:

Figure 40.2: Test case attributes

Name

A Test case is identified by its name and the names of its Test set[525] ancestors, so you should assign a name with a meaning that is easy to recognize and remember.

Variable: No

Restrictions: Must not be empty or contain the characters '.' or '#'.

Name for run log and report

A separate name to be used for run log and report. This is useful to differentiate between multiple executions with potentially different values for the Characteristic variables.

Variable: Yes

Restrictions: None

Characteristic variables

These variables are part of the characteristics of a Test set or Test case. Two executions of a Test case are considered to represent the same logical test case if the run-time values of all Characteristic variables are identical. The run-time values of the Characteristic variables are stored in the run log.

Variable: No

Restrictions: None

Name for separate run log

If this attribute is set it marks the node as a breaking point for split run logs and defines the filename for the partial log. When the node finishes, the respective log entry is removed from the main run log and saved as a separate, partial run log. This operation is completely transparent, the main run log retains references to the partial logs and is fully controllable. Please see section 7.1.6[122] for further information about split run logs.

This attribute has no effect if the option Create split run logs[505] is disabled or split run logs are explicitly turned off for batch mode via the `-splitlog`[880] command line argument.

There is no need to keep the filename unique. If necessary, QF-Test appends a number to the filename to avoid collisions. The filename may contain directories and, similar to specifying the name of a run log in batch mode on the command line, the following placeholders can be used after a '%' or a '+' character:

Character	Replacement
%	Literal '%' character.
+	Literal '+' character.
i	The current runid as specified with `-runid <ID>`[878].
r	The error level of the partial log.
w	The number of warnings in the partial log.
e	The number of errors in the partial log.
x	The number of exceptions in the partial log.
t	The thread index to which the partial log belongs (for tests run with parallel threads).
y	The current year (2 digits).
Y	The current year (4 digits).
M	The current month (2 digits).
d	The current day (2 digits).
h	The current hour (2 digits).
m	The current minute (2 digits).
s	The current second (2 digits).

Table 40.1: Placeholders for the Name for separate run log attribute

Variable: Yes

Restrictions: None, characters that are illegal for a filename will be replaced with '_'.

Inherit dependency from parent node

This option allows inheriting the Dependency[548] from the parent node as a replacement for or in addition to specifying a Dependency for the node itself.

Variable: No

Restrictions: None

Condition

If the condition is non-empty it will be evaluated and if the result is false the execution of the current node is aborted. In this case the node will be reported as skipped.

Like the Condition[606] of an If[605] node, the Condition is evaluated by the Jython interpreter, so the same rules apply.

Variable: Yes

Restrictions: Valid syntax

Script language

This attribute determines the interpreter in which to run the script, or in other words, the scripting language to use. Possible values are "Jython", "Groovy" and "JavaScript".

Variable: No

Restrictions: None

Expected to fail if...

This attribute allows specifying a condition under which the Test case is expected to fail. This is helpful to distinguish new problems from already known ones. Of course the latter should be fixed, but if and when that happens may be outside the tester's sphere of influence.

Most of the time it is sufficient to simply set this attribute to 'true' to mark the Test case as an expected failure. But if the Test case is executed multiple times, e.g. within a Data driver[562] or on multiple systems and fails only in specific cases, the condition should be written so that it evaluates to true for exactly those cases, e.g. ${qftest:windows} for a test that fails on Windows but runs fine on other systems.

If this attribute evaluates to true and the Test case fails with an error, it will be listed separately in the run log, report and on the status line. It still means that there is an error in the application, so the overall percentage of successful tests is not changed.

It is treated as an error if this attribute evaluates to true and the Test case does not fail because this means that either the test is not able to reproduce the problem reliably or that the problem has been fixed and the Test case must be updated accordingly.

Variable: Yes

Restrictions: Valid syntax

Variable definitions

These variables are bound on the direct bindings stack (see chapter 6[103]). They remain valid during the execution of the Test case's child nodes and cannot be overridden by a Test call[531] node. See section 2.2.5[17] about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Parameter default values

Here you can define default or fallback values for the Test case's parameters (see chapter 6[103]). Defining these values also serves as documentation and is a valuable time-saver when using the dialog to select the Test case for the

Test name[(533)] attribute of a Test call[(531)]. See section 2.2.5[(17)] about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run log node for a sequence will propagate, can be limited.

This value has no effect on the way exceptions are handled. It only affects the error states of the run log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.7[(11)]). It also has no effect on the creation of compact run logs (see command line argument -compact[(870)]). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

Execution timeout

Time limit for the node's execution in milliseconds. If that limit expires the execution of that node will get interrupted.

Variable: Yes

Restrictions: >= 0

Border for relative calls

This flag determines whether relative procedure calls, test calls or dependency references are allowed within that certain node. Relative calls passing that border are not allowed. If that attribute is not specified in the hierarchy, no relative calls are allowed.

Variable: No

Restrictions: None

QF-Test ID

When using the command line argument -test <n>|<ID>[(881)] for execution in batch mode you can specify the QF-Test ID of the node as an alternative to its qualified name.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the 🖍 button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.2.2   Test set

The main purpose of a Test set is to give structure to the Test cases[(518)] of a test suite. Test sets can be nested. The Name[(527)] of a Test set is part of the Test case's fully qualified name to which a Test call[(531)] node refers. Test sets are also callable themselves and thus have a set of Parameter default values[(530)] that can be overridden in the Test call.

One way of structuring Test cases is to collect those with similar setup and cleanup requirements in the same Test set which can provide a Dependency[(548)] or Dependency reference[(551)] node to be inherited by the Test case or nested Test set child nodes. If the Inherit dependency from parent node[(529)] attribute of the Test set is set, the Dependency nodes of the Test set's parent(s) are also inherited. See section 8.6[(138)] for information about QF-Test's dependency mechanism. Alternative or additional setup

and cleanup is available in the form of Setup[554] and Cleanup[557] nodes which will be executed before and after each of the Test set and Test case child nodes.

Another important feature of a Test set is data-driven testing. This can be achieved by adding a Data driver[562] node with one or more Data binders as described in chapter 21[261].

Like a Test case, a Test set plays an important role for reports. As it can also be executed several times with different parameter settings, it has a set of Characteristic variables[528] and an alternative Name for run log and report[528] that work just like for a Test case. The same is true for the Condition[529] which can be used to skip an entire Test set depending on current variable values.

Contained in: Test suite[515], Test set[525].

Children: Optional Dependency[548] or Dependency reference[551] as the first child, followed by an optional Data driver[562]. A Setup[554] may come next and a Cleanup[557] at the end with an arbitrary number of Test set[525], Test case[518] and Test call[531] nodes in between.

Execution: First the Parameter default values[530] of the Test set are bound on the fallback variable stack (see chapter 6[103]). With these in place the Condition[529] is evaluated and the node will be skipped if a non-empty Condition evaluates to false. The Dependency[548] or Dependency reference[551] node will not be resolved at this time unless its Always execute, even in test suite and test set nodes[550] attribute is set. If there is a Data driver[562] node, it is executed to create a data driving context, bind one or more Data binders and then repeatedly execute the other child nodes as described in chapter 21[261]. For each loop iteration - or once in case no Data driver is present - the Test set executes each of its Test set or Test case child nodes. If an optional Setup[554] or Cleanup[557] node are present, they are executed before and after each of these child nodes respectively.

Attributes:

Figure 40.3: Test set attributes

Name

The name of a Test set is part of its own identification and of that of the Test case[518] and Test set nodes it contains, so you should assign a name with a meaning that is easy to recognize and remember.

Variable: No

Restrictions: Must not be empty or contain the characters '.' or '#'.

Name for run log and report

A separate name to be used for run log and report. This is useful to differentiate between multiple executions with potentially different values for the Characteristic variables.

Variable: Yes

Restrictions: None

Characteristic variables

These variables are part of the characteristics of a Test set or Test case. Two executions of a Test case are considered to represent the same logical test case if the run-time values of all Characteristic variables are identical. The run-time values of the Characteristic variables are stored in the run log.

Variable: No

Restrictions: None

Name for separate run log

If this attribute is set it marks the node as a breaking point for split run logs and defines the filename for the partial log. When the node finishes, the respective log entry is removed from the main run log and saved as a separate, partial run log. This operation is completely transparent, the main run log retains references to the partial logs and is fully controllable. Please see section 7.1.6[122] for further information about split run logs.

This attribute has no effect if the option Create split run logs[505] is disabled or split run logs are explicitly turned off for batch mode via the -splitlog[880] command line argument.

There is no need to keep the filename unique. If necessary, QF-Test appends a number to the filename to avoid collisions. The filename may contain directories and, similar to specifying the name of a run log in batch mode on the command line, the following placeholders can be used after a '%' or a '+' character:

Character	Replacement
%	Literal '%' character.
+	Literal '+' character.
i	The current runid as specified with `-runid <ID>`[878].
r	The error level of the partial log.
w	The number of warnings in the partial log.
e	The number of errors in the partial log.
x	The number of exceptions in the partial log.
t	The thread index to which the partial log belongs (for tests run with parallel threads).
y	The current year (2 digits).
Y	The current year (4 digits).
M	The current month (2 digits).
d	The current day (2 digits).
h	The current hour (2 digits).
m	The current minute (2 digits).
s	The current second (2 digits).

Table 40.2: Placeholders for the Name for separate run log attribute

Variable: Yes

Restrictions: None, characters that are illegal for a filename will be replaced with '_'.

Inherit dependency from parent node

This option allows inheriting the Dependency[548] from the parent node as a replacement for or in addition to specifying a Dependency for the node itself.

Variable: No

Restrictions: None

Condition

If the condition is non-empty it will be evaluated and if the result is false the execution of the current node is aborted. In this case the node will be reported as skipped.

Like the Condition[606] of an If[605] node, the Condition is evaluated by the Jython interpreter, so the same rules apply.

Variable: Yes

Restrictions: Valid syntax

Script language

This attribute determines the interpreter in which to run the script, or in other words, the scripting language to use. Possible values are "Jython", "Groovy" and "JavaScript".

Variable: No

Restrictions: None

Parameter default values

Here you can define default or fallback values for the Test set's parameters (see chapter 6[(103)]). Defining these values also serves as documentation and is a valuable time-saver when using the dialog to select the Test set for the Test name[(533)] attribute of a Test call[(531)]. See section 2.2.5[(17)] about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run log node for a sequence will propagate, can be limited.

Note

This value has no effect on the way exceptions are handled. It only affects the error states of the run log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.7[(11)]). It also has no effect on the creation of compact run logs (see command line argument -compact[(870)]). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

Execution timeout

Time limit for the node's execution in milliseconds. If that limit expires the execution of that node will get interrupted.

Variable: Yes

Restrictions: >= 0

Border for relative calls

This flag determines whether relative procedure calls, test calls or dependency references are allowed within that certain node. Relative calls passing that border

are not allowed. If that attribute is not specified in the hierarchy, no relative calls are allowed.

Variable: No

Restrictions: None

QF-Test ID

When using the command line argument `-test <n>|<ID>`[(881)] for execution in batch mode you can specify the QF-Test ID of the node as an alternative to its qualified name.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the [button] button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.2.3   Test call

With this node a call to some other executable test node can be made. Possible targets are Test suite[515], Test set[525] and Test case[518] nodes in the same or a different test suite. Execution will continue in the called node and when finished return to the Test call and thus to its parent node.

The name of a Test case or Test set to call is determined by its Name[521] and the Names[527] of its Test set ancestors. These are concatenated with a dot ('.') as separator, starting with the outermost Test set and ending in the Test case's name. Thus to call a Test case named `nodeTest` in a Test set named `Tree` that is itself a child of a Test set named `Main`, the Test name[533] attribute would be set to '`Main.Tree.nodeTest`'. A node in a different test suite is addressed by prepending the filename of the test suite followed by a '#' to the Test name. A Test suite node is addressed by a single '.', so calling a whole test suite is done with a Test name attribute of the form '`suitename.qft#.`'. It is generally easiest to pick the target node interactively from a dialog by clicking on the button above the Test name attribute.

See also section 24.1[298] for further information about cross-suite calls.

Contained in: All kinds of sequences[518].

Children: None

Execution: The Variable definitions[534] of the Test call are bound, the target node is determined and execution is passed to it. After the Test call returns, the Test call's variables are unbound again.

Attributes:

Figure 40.4: Test call Attributes

Name

In case of a Test case or Test set it is the full name, created from the Name[527] of its Test set parents and its own Name[521], joined by a dot. A Test suite node is addressed by a single '.' character. The "Select test" button above the attribute brings up a dialog in which you can select the target node interactively. You can also get to this dialog by pressing Shift-Return or Alt-Return when the focus is in the text field. By selecting the "Copy parameters" checkbox you can adopt the Test case's or Test set's Parameter default values as parameters for the Test call node to save typing.

Variable: Yes

Restrictions: Must not be empty.

Name for separate run log

If this attribute is set it marks the node as a breaking point for split run logs and defines the filename for the partial log. When the node finishes, the respective log entry is removed from the main run log and saved as a separate, partial run log. This operation is completely transparent, the main run log retains references to the partial logs and is fully controllable. Please see section 7.1.6[(122)] for further information about split run logs.

This attribute has no effect if the option Create split run logs[(505)] is disabled or split run logs are explicitly turned off for batch mode via the `-splitlog`[(880)] command line argument.

There is no need to keep the filename unique. If necessary, QF-Test appends a number to the filename to avoid collisions. The filename may contain directories and, similar to specifying the name of a run log in batch mode on the command line, the following placeholders can be used after a '%' or a '+' character:

Character	Replacement
%	Literal '%' character.
+	Literal '+' character.
i	The current runid as specified with `-runid <ID>`[(878)].
r	The error level of the partial log.
w	The number of warnings in the partial log.
e	The number of errors in the partial log.
x	The number of exceptions in the partial log.
t	The thread index to which the partial log belongs (for tests run with parallel threads).
y	The current year (2 digits).
Y	The current year (4 digits).
M	The current month (2 digits).
d	The current day (2 digits).
h	The current hour (2 digits).
m	The current minute (2 digits).
s	The current second (2 digits).

Table 40.3: Placeholders for the Name for separate run log attribute

Variable: Yes

Restrictions: None, characters that are illegal for a filename will be replaced with '_'.

Variable definitions

This is where you define the parameter values for the target node. These variables are bound on the primary variable stack (see chapter 6[(103)]) so they

override any Parameter default values. See section 2.2.5[(17)] about how to work with the table.

In case you want to re-set the order of the parameters like they are sorted in the called test case or test set, you can select Re-set parameter order.

Variable: Variable names no, values yes

Restrictions: None

Act like a procedure call

If the Test call node is executed inside a Test case, this attribute determines how exceptions are handled in the called node(s). If it is activated a single exception terminates the whole call irrespective of Test set and Test case nesting, just like it would for a regular Procedure call node. If this attribute is unset the special roles of Test set and Test case nodes with their local exception handling is maintained.

Variable: Yes

Restrictions: None

Execution timeout

Time limit for the node's execution in milliseconds. If that limit expires the execution of that node will get interrupted.

Variable: Yes

Restrictions: >= 0

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes,

this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ✏️ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.2.4 Sequence

This is the most general kind of sequence. Its children are executed one by one and their number or type is not limited in any way.

Contained in: All kinds of sequences[518].

Children: Any

Execution: The Variable definitions[537] of the Sequence are bound and its child nodes executed one by one. After the execution of the last child is complete, the variables are unbound again.

Attributes:

Figure 40.5: Sequence attributes

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and say something about the function of the sequence.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see chapter 6[(103)]). See section 2.2.5[(17)] about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the

corresponding node of the run log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run log node for a sequence will propagate, can be limited.

Note   This value has no effect on the way exceptions are handled. It only affects the error states of the run log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.7[(11)]). It also has no effect on the creation of compact run logs (see command line argument `-compact`[(870)]). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note   For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.2.5 Test step

A Test step is a special Sequence that serves to divide a Test case into steps that can be documented individually and will show up in the report or testdoc documentation. In contrast to Test cases, which should not be nested, Test steps can be nested to any depth.

Contained in: All kinds of sequences[(518)].

Children: Any

Execution: The Variable definitions[(542)] of the Test step are bound and its child nodes executed one by one. After the execution of the last child is complete, the variables are unbound again.

Attributes:

Figure 40.6: Test step attributes

Name

> The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and say something about the function of the sequence.

> **Variable:** No

> **Restrictions:** None

Name for run log and report

> A separate name to be used for run log and report. This is useful to differentiate

between multiple executions of the surrounding Test case with potentially different values for its Characteristic variables.

Variable: Yes

Restrictions: None

Name for separate run log

If this attribute is set it marks the node as a breaking point for split run logs and defines the filename for the partial log. When the node finishes, the respective log entry is removed from the main run log and saved as a separate, partial run log. This operation is completely transparent, the main run log retains references to the partial logs and is fully controllable. Please see section 7.1.6[(122)] for further information about split run logs.

This attribute has no effect if the option Create split run logs[(505)] is disabled or split run logs are explicitly turned off for batch mode via the -splitlog[(880)] command line argument.

There is no need to keep the filename unique. If necessary, QF-Test appends a number to the filename to avoid collisions. The filename may contain directories and, similar to specifying the name of a run log in batch mode on the command line, the following placeholders can be used after a '%' or a '+' character:

Character	Replacement
%	Literal '%' character.
+	Literal '+' character.
i	The current runid as specified with -runid <ID>[(878)].
r	The error level of the partial log.
w	The number of warnings in the partial log.
e	The number of errors in the partial log.
x	The number of exceptions in the partial log.
t	The thread index to which the partial log belongs (for tests run with parallel threads).
y	The current year (2 digits).
Y	The current year (4 digits).
M	The current month (2 digits).
d	The current day (2 digits).
h	The current hour (2 digits).
m	The current minute (2 digits).
s	The current second (2 digits).

Table 40.4: Placeholders for the Name for separate run log attribute

Variable: Yes

Restrictions: None, characters that are illegal for a filename will be replaced with '_'.

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see chapter 6[103]). See section 2.2.5[17] about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run log node for a sequence will propagate, can be limited.

Note

This value has no effect on the way exceptions are handled. It only affects the error states of the run log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.7[11]). It also has no effect on the creation of compact run logs (see command line argument −compact[870]). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

Execution timeout

Time limit for the node's execution in milliseconds. If that limit expires the execution of that node will get interrupted.

Variable: Yes

Restrictions: >= 0

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by

pressing ⌊Alt-Return⌋ or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.2.6 Sequence with time limit

This node extends the basic Sequence[(536)] node with time-constraint checking. Child nodes are executed as usual, but upon completion of the sequence the elapsed time is compared to the time-limit. Exceeding the time limit will result in a warning, error or exception, depending on the value of the attribute Error level if time limit exceeded[(545)]. Explicit delays like Delay before/after[(546)] or user interrupts are deducted from the duration before the constraints are checked, unless Check realtime[(545)] is activated.

For report generation, time-constraints are treated like checks. If the Comment[(546)] attribute starts with an '!' character, the result will be logged in the report.

The function of this node is to check time constraints in the user-acceptance range, i.e. between a few hundred milliseconds and a few seconds. Real-time constraints of a few milliseconds or less are beyond its limits.

Contained in: All kinds of sequences[(518)].

Children: Any

Execution: The Variable definitions[(545)] of the Sequence with time limit are bound and its child nodes executed one by one. After the execution of the last child is complete, the variables are unbound again. The elapsed time is compared to the given time limit.

Attributes:

Figure 40.7: Sequence with time limit attributes

Name

> The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and say something about the function of the sequence.

Variable: No

Restrictions: None

Time limit for execution

The time (in milliseconds) allowed for the execution of the sequence.

Variable: Yes

Restrictions: Must not be negative.

Check realtime

Normally explicit delays like Delay before/after[(546)] or user interrupts are deducted from the duration, before the time constraints are checked. To prevent this deduction and therefore check the real-time, this attribute can be activated.

Variable: No

Restrictions: None

Error level if time limit exceeded

This attribute determines what happens in case the time limit is exceeded. If set to "exception", a `CheckFailedException`[(854)] will be thrown. Otherwise a message with the respective error-level will be logged in the run log.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see chapter 6[(103)]). See section 2.2.5[(17)] about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run log node for a sequence will propagate, can be limited.

Note   This value has no effect on the way exceptions are handled. It only affects the error states of the run log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.7[(11)]). It also has no effect on the creation of compact

run logs (see command line argument $-\mathtt{compact}^{(870)}$). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

Execution timeout

Time limit for the node's execution in milliseconds. If that limit expires the execution of that node will get interrupted.

Variable: Yes

Restrictions: >= 0

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.2.7   Extras

This node is a kind of clipboard or playground where the usual restrictions on the parent of a node don't apply. You can add any kind of node here to assemble and try out some test sequences.

Contained in: Root node

Children: Any

Execution: Cannot be executed

Attributes:

Figure 40.8: Extras attributes

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the 🖊 button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.3 Dependencies

Dependencies are a very powerful feature for handling setup and cleanup requirements for test cases automatically. The goal is to isolate test cases so that each one can be run independently without interfering with others. This is a very important requirement for things like testing arbitrary sub-sets of test cases, for example to re-test only failed tests, or during test-development where it must be possible to quickly run and debug any given test case.

40.3.1 Dependency

Dependency nodes are used to implement advanced, automatic handling of setup and cleanup requirements for Test sets[(525)] and Test cases[(518)]. A detailed description of the Dependency mechanism is given in section 8.6[(138)]. This section focuses on formal requirements for the Dependency node and its attributes.

As Dependencies are complex, they should be reused as much as possible. This can be done by grouping Test cases with identical dependencies in a Test set and have them inherit the Dependency of the Test set. However, this mechanism alone is not flexible enough, so a Dependency can also be implemented just like a Procedure[(586)] and placed among the Procedures[(595)] of a test suite to be referenced from a Dependency reference node. For this to work, the Name[(521)] attribute is mandatory and it also has a list of Parameter default values[(550)] that can be overridden in the referencing node.

The Characteristic variables[(550)] of a Dependency are part of its identity and play an important role in the dependency resolution mechanism.

Contained in: Test suite[(515)], Test set[(525)], Test case[(518)], Procedures[(595)], Package[(594)].

Children: Zero or more Dependency references[(551)] on which the Dependency is based, optional Setup[(554)] and Cleanup[(557)] nodes and an optional Error handler[(560)] followed by zero or more Catch[(619)] nodes.

Execution: Normally Dependencies are executed only indirectly in the setup phase of a Test set or Test case. If a Dependency node is executed interactively, the dependency stack is resolved as described in section 8.6[(138)].

Attributes:

Figure 40.9: Dependency attributes

Name

A Dependency is identified by its name, so you should assign a name with a meaning that is easy to recognize and remember.

Variable: No

Restrictions: Must not be empty or contain the characters '.' or '#'.

Name for run log and report

A separate name to be used for run log and report. This is useful to differentiate between multiple executions with potentially different values for the Characteristic variables.

Variable: Yes

Restrictions: None

Characteristic variables

These variables are part of a Dependency's identity. During Dependency resolution as described in section 8.6[(138)] two Dependencies are considered equal only if they are one and the same node and the run-time values of all their Characteristic variables are identical. Additionally, the values of the Characteristic variables are stored during the setup phase of the Dependency. Later, when the Dependency is rolled back, these settings will be temporarily restored for the cleanup phase.

Characteristic variables have the same value at a Cleanup node as during the execution of the corresponding Setup node - regardless of the value of the variables in the current test case.

Variable: No

Restrictions: None

Always execute, even in test suite and test set nodes

Normally a Dependency is only executed if it belongs to a Test case node. Dependencies in Test suite or Test set nodes are simply inherited by the Test case descendants of these nodes. However, in some cases it is useful to resolve a Dependency early, for example when the Dependency provides parameters for a test run that are required to evaluate a Condition[(522)] of a subsequent Test case. This can be achieved by activating this option.

Variable: No

Restrictions: None

Forced cleanup

Normally Dependencies are only rolled back and their cleanup code executed as required by the dependency resolution mechanism described in section 8.6[(138)]. In some cases it makes sense to force partial cleanup of the dependency stack immediately after a Test case finishes. This is what the Forced cleanup attribute is for. If this option is activated, the dependency stack will be rolled back at least up to and including this Dependency .

Variable: No

Restrictions: None

Parameter default values

Here you can define default or fallback values for the Dependency's parameters (see chapter 6[(103)]). Defining these values also serves as documentation and is a valuable time-saver when using the dialog to select the Dependency for the

Referenced dependency[(552)] attribute of a Dependency reference[(551)]. See section 2.2.5[(17)] about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the 🖉 button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.3.2   Dependency reference

A Dependency reference is simply a stand-in for a Dependency[(548)] defined in some other place, typically a Package[(594)]. The name of the referenced Dependency is determined by its Name[(549)] and the Names[(594)] of its Package parents. These are concatenated with a dot ('.') as separator, starting with the outermost Package and ending in the Dependency's name. Thus to reference a Dependency named `demoStarted` in a Package named `Demo` that is itself a child of a Package named `Main`, set the Referenced dependency[(552)] attribute to 'Main.Demo.demoStarted'.

See section 24.1[(298)] about how to reference a Dependency in a different test suite.

Contained in: Test suite[(515)], Test set[(525)], Test case[(518)] and Dependency[(548)]

Children: None

Execution: Normally Dependency references are executed only indirectly in the setup phase of a Test set or Test case. If a Dependency reference node is executed interactively, the referenced Dependency is determined and the dependency stack resolved accordingly as described in section 8.6[(138)].

Attributes:

Figure 40.10: Dependency reference attributes

Referenced dependency

The full name of the Dependency⁽⁵⁴⁸⁾, created from the Names⁽⁵⁹⁴⁾ of its Package⁽⁵⁹⁴⁾ parents and its own Name⁽⁵⁴⁹⁾, joined by a dot. The "Select dependency" button above the attribute brings up a dialog in which you can select the Dependency interactively. By selecting the "Copy parameters" checkbox you can adopt the Dependency's Parameter default values⁽⁵⁵⁰⁾ as parameters for the Dependency reference node to save typing.

Variable: Yes

Restrictions: Must not be empty.

Dependency namespace

Normally there is only a single stack of dependencies, but in some cases, e.g. when mixing tests for independent SUT clients, it can be useful to have independent sets of dependencies for different parts of the test, so that resolving the dependencies for one part doesn't necessarily tear down everything required for a different part.

By setting the Dependency namespace attribute of a Dependency reference node you can tell QF-Test to resolve the target dependency in that namespace. The default dependency stack will be completely ignored in that case. If there is a dependency stack remaining from a previous dependency resolution in the same namespace it will be used for comparison instead, otherwise a new stack will be created. For an example please refer to Name spaces for Dependencies⁽¹⁵¹⁾.

Variable: Yes

Restrictions: None

Always execute, even in test suite and test set nodes

Normally a Dependency is only executed if it belongs to a Test case node. Dependencies in Test suite or Test set nodes are simply inherited by the Test case descendants of these nodes. However, in some cases it is useful to resolve a Dependency early, for example when the Dependency provides parameters for a test run that are required to evaluate a Condition⁽⁵²²⁾ of a subsequent Test case. This can be achieved by activating this option.

Variable: No

Restrictions: None

Variable definitions

These variables override the Parameter default values⁽⁵⁵⁰⁾ of the Dependency referenced by this Dependency reference. See section 2.2.5⁽¹⁷⁾ about how to work with the table.

4.2+ In case you want to re-set the order of the parameters like they are sorted in the

called dependency, you can select $\boxed{\text{Re-set parameter order}}$.

Variable: Variable names no, values yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note   For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by

pressing $\boxed{\text{Alt-Return}}$ or by clicking the ✎ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.3.3   Setup

This node is just like a Sequence[536] except for its special place in a Dependency[548], Test set[525] or Test case[518] node.

Contained in: Dependency[548], Test set[525] or Test case[518]

Children: Any

Execution: The Variable definitions[(555)] of the Setup are bound and its child nodes executed one by one. After the execution of the last child is complete, the variables are unbound again.

Attributes:

Figure 40.11: Setup attributes

Name

> The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and say something about the function of the sequence.
>
> **Variable:** No
>
> **Restrictions:** None

Variable definitions

> This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see chapter 6[(103)]). See section 2.2.5[(17)] about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run log node for a sequence will propagate, can be limited.

This value has no effect on the way exceptions are handled. It only affects the error states of the run log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.7[(11)]). It also has no effect on the creation of compact run logs (see command line argument −compact[(870)]). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.3.4 Cleanup

This node is just like a Sequence[(536)] except for its special place in a Dependency[(548)], Test set[(525)] or Test case[(518)] node.

Contained in: Dependency[(548)], Test set[(525)] or Test case[(518)]

Children: Any

Execution: The Variable definitions[(558)] of the Cleanup are bound and its child nodes executed one by one. After the execution of the last child is complete, the variables are unbound again.

Attributes:

Figure 40.12: Cleanup attributes

Name

> The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and say something about the function of the sequence.
>
> **Variable:** No
>
> **Restrictions:** None

Variable definitions

> This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see chapter 6[103]). See section 2.2.5[17] about how to work with the table.
>
> **Variable:** Variable names no, values yes
>
> **Restrictions:** None

Maximum error level

> When a warning, error or exception occurs during a test run, the state of the corresponding node of the run log is set accordingly. This state is normally

propagated to the parent node in a way that ensures that the error state of a run log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run log node for a sequence will propagate, can be limited.

This value has no effect on the way exceptions are handled. It only affects the error states of the run log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.7[(11)]). It also has no effect on the creation of compact run logs (see command line argument -compact[(870)]). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the 🖉 button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.3.5 Error handler

This node is just like a Sequence[(536)] except for its special place in a Dependency[(548)] (see section 8.6[(138)]).

Contained in: Dependency[(548)]

Children: Any

Execution: The Variable definitions[(560)] of the Error handler are bound and its child nodes executed one by one. After the execution of the last child is complete, the variables are unbound again.

Attributes:

Figure 40.13: Error handler attributes

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and say something about the function of the sequence.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see chapter 6[103]). See section 2.2.5[17] about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run log node for a sequence will propagate, can be limited.

This value has no effect on the way exceptions are handled. It only affects the error states of the run log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.7[11]). It also has no effect on the creation of compact run logs (see command line argument `-compact`[870]). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes,

this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ✏ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.4 Data driver

The term *Data-driven Testing* refers to a common method in automated testing where test cases are executed several times with different sets of data defined. With QF-Test's highly flexible variables there is no limit on how this data can be used but the most common case is for event input values and expected check values.

QF-Test's data-driving mechanism consists of the Data driver[562] node used to provide a data-driving context and several kinds of Data binders. Currently available are the Data table[566] node that stores data internally within QF-Test the CSV data file[579] node that reads data from a CSV file, the Database[569] node that reads data from a database and the Excel data file[574] that reads data from an Excel file. An extension API for plugging in arbitrary external data is also available.

Further information about how the various parts of the data driver mechanism are working together is provided in chapter 21[261].

40.4.1 Data driver

Except for its special place in a Test set[525] or Test step[539], a Data driver is just like a normal Sequence[536]. It provides a context for one or more Data binders to register themselves during the execution of the Data driver. The Test set then iterates over the sets of data provided by the registered Data binders and executes its child nodes as described in chapter 21[261]. For this purpose a Data driver node needs to be placed in a Test set[525] node, between the optional Dependency[548] and Setup[554] nodes. Data driver nodes can also be placed in a Test step[539] as first steps.

Contained in: Test set[525], Test step[539].

Children: Any

Execution: When a Test set or Test step is executed it checks for a Data driver and runs it. The contents of the Data driver node are not limited to Data binders, but can hold any executable node so that they can perform any setup that may be required to retrieve the data. Thus it is also possible to share Data binders by placing them inside a Procedure[586] and calling the Procedure from inside the Data driver. Any Data binders registered within this Data driver's context will then be queried for data by the Test set or Test step.

Attributes:

Figure 40.14: Data driver attributes

Name

    The name of a Data driver is a kind of short description. It is displayed in the tree view, so it should be concise and say something about the kind of data provided.

    Variable: No

    Restrictions: None

Name for loop pass in the run log

> A separate name for each iteration to be used in the run log. It can make use of the variables bound as a result of the data-driving which makes it easier to locate a specific step of the iteration.

> **Variable:** Yes

> **Restrictions:** None

Name for separate run log

> If this attribute is set it marks the node as a breaking point for split run logs and defines the filename for the partial log. Every time an iteration of the Data driver finishes, the respective log entry is removed from the main run log and saved as a separate, partial run log. This operation is completely transparent, the main run log retains references to the partial logs and is fully controllable. Please see section 7.1.6[122] for further information about split run logs.

> This attribute has no effect if the option Create split run logs[505] is disabled or split run logs are explicitly turned off for batch mode via the -splitlog[880] command line argument.

> There is no need to keep the filename unique. If necessary, QF-Test appends a number to the filename to avoid collisions. The filename may contain directories and, similar to specifying the name of a run log in batch mode on the command line, the following placeholders can be used after a '%' or a '+' character:

Character	Replacement
%	Literal '%' character.
+	Literal '+' character.
i	The current runid as specified with -runid <ID>[878].
r	The error level of the partial log.
w	The number of warnings in the partial log.
e	The number of errors in the partial log.
x	The number of exceptions in the partial log.
t	The thread index to which the partial log belongs (for tests run with parallel threads).
y	The current year (2 digits).
Y	The current year (4 digits).
M	The current month (2 digits).
d	The current day (2 digits).
h	The current hour (2 digits).
m	The current minute (2 digits).
s	The current second (2 digits).

Table 40.5: Placeholders for the Name for separate run log attribute

Variable: Yes

Restrictions: None, characters that are illegal for a filename will be replaced with '_'.

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see chapter 6[(103)]). See section 2.2.5[(17)] about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ✎ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.4.2 Data table

A Data table provides a convenient interface for storing test data in tabular form directly inside QF-Test. For details about the data-driving mechanism please see chapter 21[(261)].

Contained in: Any

Children: None

Execution: The Data table expands variable values in the table according to the option When binding variables, expand values immediately[(513)]. Each row is expanded individually left to right, meaning that - within the same row - a cell may refer to a variable bound in a column further to the left. Then the Data table registers itself with the Data driver context. A property group called like the node will be created additionally. That group contains the variables `size` and `totalsize`. `size` shows the number of data rows with taking care about iteration intervals. The variable `totalsize` shows the total number of data rows without taking care about iteration intervals. When all Data binders have been registered the Test set[(525)] will query the Data table in order to iterate over the available sets of data. If no such context is available the respective property group will be extended with all variables.

Attributes:

Figure 40.15: Data table attributes

Name

> The Name of a Data binder is mandatory. It is used to distinguish Data binders in the same Data driver context. A <u>Break</u>[604] node executed during data-driven testing can be used to break out of a specific loop by referring to the Data binder's Name.
>
> **Variable:** Yes
>
> **Restrictions:** None

Iteration counter

> The name of the variable that the iteration counter will be bound to.
>
> **Variable:** Yes
>
> **Restrictions:** None

Iteration ranges

> An optional set of indexes or ranges to use from the bound data.   This is especially useful during test development in order to run sample tests with just a single index or a subset of the given data.

Ranges are separated by ','. Each range is either a single index or an inclusive range of the form 'from-to' or 'from:to' where 'to' is optional. Indexes or ranges may be specified multiple times, overlap or be given in descending order. Indexes are 0-based, negative indexes are counted from the end, -1 being the last item. An invalid syntax or an index outside the valid data range will cause a `BadRangeException`[(854)].

The following table shows some example range specifications and the resulting indexes, based on a set of 20 entries.

Iteration ranges	Resulting indexes
0	[0]
-2, -1	[18, 19]
1-2,4:5	[1, 2, 4, 5]
18:,-3-	[18, 19, 17, 18, 19]
3-2,16:15	[3, 2, 16, 15]

Table 40.6: Iteration range examples

Note

The value bound for the Iteration counter reflects the index in the current interval, not the counter of actual iterations, e.g. if you specify '2' there will be a single iteration with the Iteration counter bound to '2', not '0'.

Variable: Yes

Restrictions: Valid syntax and index values

Data bindings

This is where the actual test data is defined. Each column of the table represents one variable with its name specified in the column header. Each row is a set of data, one value per variable. Thus the number of rows determines the number of iterations of the data-driven loop. To start entering the data you first need to add columns to the table to define the variables, then add rows to fill in the values. See also section 2.2.5[(17)] about how to work with the table.

Variable: Yes, even the column headers

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ✎ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.4.3 Database

A Database node is used to load external data from a database. To access the database it is required to have the jar file with the database driver in QF-Test's plugin-directory (section 48.2[(916)]) or in the CLASSPATH environment variable at QF-Test's start-up time. For more information about the connection mechanism to your database ask the developers or see www.connectionstrings.com.

For further details about the data-driving mechanism please see chapter 21[(261)].

Contained in: Any

Children: None

Execution: The Database loads the data from the database and expands variable values according to the option When binding variables, expand values immediately[(513)]. Each row is expanded individually left to right, meaning that - within the same row - a cell may refer to a variable bound in a column further to the left. Then the Database node registers itself with the Data driver context. The result columns of the used SQL statement will be used for the variable names. The capitalization depends on the database driver,

e.g. completely capitalized letters for Oracle. A property group called like the node will be created additionally. That group contains the variables `size` and `totalsize`. `size` shows the number of data rows with taking care about iteration intervals. The variable `totalsize` shows the total number of data rows without taking care about iteration intervals. When all Data binders have been registered the Test set[525] will query the Database in order to iterate over the available sets of data. If no such context is available the respective property group will be extended with all variables.

Attributes:

Figure 40.16: Database attributes

Name

> The Name of a Data binder is mandatory. It is used to distinguish Data binders in the

same Data driver context. A <u>Break</u>[(604)] node executed during data-driven testing can be used to break out of a specific loop by referring to the Data binder's Name.

Variable: Yes

Restrictions: None

Iteration counter

The name of the variable that the iteration counter will be bound to.

Variable: Yes

Restrictions: None

Iteration ranges

An optional set of indexes or ranges to use from the bound data. This is especially useful during test development in order to run sample tests with just a single index or a subset of the given data.

Ranges are separated by ','. Each range is either a single index or an inclusive range of the form 'from-to' or 'from:to' where 'to' is optional. Indexes or ranges may be specified multiple times, overlap or be given in descending order. Indexes are 0-based, negative indexes are counted from the end, -1 being the last item. An invalid syntax or an index outside the valid data range will cause a <u>BadRangeException</u>[(854)].

The following table shows some example range specifications and the resulting indexes, based on a set of 20 entries.

Iteration ranges	Resulting indexes
0	[0]
-2, -1	[18, 19]
1-2,4:5	[1, 2, 4, 5]
18:,-3-	[18, 19, 17, 18, 19]
3-2,16:15	[3, 2, 16, 15]

Table 40.7: Iteration range examples

The value bound for the Iteration counter reflects the index in the current interval, not the counter of actual iterations, e.g. if you specify '2' there will be a single iteration with the Iteration counter bound to '2', not '0'.

Variable: Yes

Restrictions: Valid syntax and index values

SQL statement

The SQL query that should be executed to get the desired test data. This statement is supposed to be a *select* statement. Each column will stand for a variable. The capitalization of the columns depends on the kind of the used database driver, e.g. most of the Oracle drivers return completely capitalized variables.

Variable: Yes

Restrictions: Must not be empty

Driver class

The class name of the database driver.

Note   The jar file with the database driver has to be placed in the CLASSPATH or in QF-Test's plugin directory before launching QF-Test.

Here is a list of the most common database drivers:

Database	Classname of JDBC-driver
Borland Interbase	`interbase.interclient.Driver`
DB2	`com.ibm.db2.jcc.DB2Driver`
Informix	`com.informix.jdbc.IfxDriver`
IDS Server	`ids.sql.IDSDriver`
MS SQL Server 2000	`com.microsoft.jdbc.sqlserver.SQLServerDriver`
MS SQL Server 2005	`com.microsoft.sqlserver.jdbc.SQLServerDriver`
mSQL	`COM.imaginary.sql.msql.MsqlDriver`
MySQL	`com.mysql.jdbc.Driver`
Oracle	`oracle.jdbc.driver.OracleDriver`
Pointbase	`com.pointbase.jdbc.jdbcUniversalDriver`
PostgreSQL	`org.postgresql.Driver`
Standard Driver	`sun.jdbc.odbc.JdbcOdbcDriver`
Sybase	`com.sybase.jdbc2.jdbc.SybDriver`
SQLite	`org.sqlite.JDBC`

Table 40.8: Database drivers

Variable: Yes

Restrictions: Must not be empty

Connection string

The connection string for database connection, typically something like: `jdbc:`*databasetype*`://`*databasehost*`/`*databasename*.

Here is a list of the most common database connection strings:

Database	Example
Derby	`jdbc:derby:net://`*`databaseserver`*`:`*`port`*`/`
IBM DB2	`jdbc:db2://`*`database`*
HSQLB	`jdbc:hsqldb:file:`*`database`*
Interbase	`jdbc:interbase://`*`databaseserver`*`/`*`database`*`.gdb`
MS SQL Server 2000	`jdbc:microsoft:sqlserver://`*`databaseserver`*`:` *`port`*`;DatabaseName=`*`database`*`;`
MS SQL Server 2005	`jdbc:sqlserver://`*`databaseserver`*`:` *`port`*`;DatabaseName=`*`database`*`;`
MySQL	`jdbc:mysql://`*`databaseserver`*`/`*`database`*
PostgreSQL	`jdbc:postgresql://`*`databaseserver`*`/`*`database`*
ODBC Data Sources	`jdbc:odbc:`*`database`*
Oracle Thin	`jdbc:oracle:thin:@`*`databaseserver`*`:`*`port`*`:` *`database`*
Sybase	`jdbc:sybase:tds:`*`databaseserver`*`:`*`port`*`/`*`database`*
SQLite	`jdbc:sqlite:`*`sqlite_database_file_path`*

Table 40.9: Database connection strings

Variable: Yes

Restrictions: Must not be empty

Database user

The name of the user to use when connecting to the database. If your database connection doesn't require a user you can leave this field empty.

Variable: Yes

Restrictions: None

Database password

The password to use when connecting to the database. If your database connection doesn't require a password you can leave this field empty. To that end the password can be encrypted by inserting the plain-text password, right-clicking and selecting Crypt password from the popup menu. Please be sure to specify a password salt before encrypting via the option Salt for crypting passwords[458].

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the 🖊 button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.4.4   Excel data file

An Excel data file is used to load external data from an Excel-file to provide test data for data-driven testing. The first row must contain the names of the variables to bind. The rest of the rows should contain the values to be used for the iteration steps.

Note

It is possible that the contents of some cells are not read correctly. This can happen especially for cells of the type "date", "time" or "currency". The reason is that Excel stores the value and the format of a cell separately and the Java package used to parse Excel files doesn't support all possibilities. Still, the Excel data file should read most cells correctly, but in case of problems please change the type of the problematic cells in the Excel file to "text". Once read by QF-Test, all values are treated as strings anyway.

For further details about the data-driving mechanism please see chapter 21[261].

Contained in: Any

Children: None

Execution: The Excel data file node loads the data from the Excel file and expands variable values in the table according to the option When binding variables, expand values immediately[513]. Each row is expanded individually left to right, meaning that - within the same row - a cell may refer to a variable bound in a column further to the left. Then the Excel data file registers itself with the Data driver context. A property group called like the node will be created additionally. That group contains the variables `size` and `totalsize`. `size` shows the number of data rows with taking care about iteration intervals. The variable `totalsize` shows the total number of data rows without taking care about iteration intervals. When all Data binders have been registered the Test set[525] will query the Excel data file in order to iterate over the available sets of data. If no such context is available the respective property group will be extended with all variables.

Attributes:

Figure 40.17: Excel data file attributes

Name

>   The Name of a Data binder is mandatory. It is used to distinguish Data binders in the same Data driver context. A Break[(604)] node executed during data-driven testing can be used to break out of a specific loop by referring to the Data binder's Name.

>   **Variable:** Yes

>   **Restrictions:** None

Iteration counter

>   The name of the variable that the iteration counter will be bound to.

>   **Variable:** Yes

>   **Restrictions:** None

Iteration ranges

An optional set of indexes or ranges to use from the bound data. This is especially useful during test development in order to run sample tests with just a single index or a subset of the given data.

Ranges are separated by ','. Each range is either a single index or an inclusive range of the form 'from-to' or 'from:to' where 'to' is optional. Indexes or ranges may be specified multiple times, overlap or be given in descending order. Indexes are 0-based, negative indexes are counted from the end, -1 being the last item. An invalid syntax or an index outside the valid data range will cause a BadRangeException[854].

The following table shows some example range specifications and the resulting indexes, based on a set of 20 entries.

Iteration ranges	Resulting indexes
0	[0]
-2, -1	[18, 19]
1-2,4:5	[1, 2, 4, 5]
18:,-3-	[18, 19, 17, 18, 19]
3-2,16:15	[3, 2, 16, 15]

Table 40.10: Iteration range examples

The value bound for the Iteration counter reflects the index in the current interval, not the counter of actual iterations, e.g. if you specify '2' there will be a single iteration with the Iteration counter bound to '2', not '0'.

Variable: Yes

Restrictions: Valid syntax and index values

Excel file name

The name of the Excel file to read the test data. Relative path names are resolved relative to the directory of the test suite.

The button above the attribute brings up a dialog in which you can select the Excel file interactively. You can also get to this dialog by pressing Shift-Return or Alt-Return when the focus is in the text field.

Variable: Yes

Restrictions: Must not be empty

Worksheet name

The name of the worksheet that contains the test data. If empty, the first sheet will be used.

Variable: Yes

Restrictions: None

Override date format

This value specifies a date format which will be used for all date values specified in the Excel file. The format must be specified as for the Java class `SimpleDateFormat`, i.e. 'd' for day, 'M' for month, 'y' for year. Thus 'dd' stands for two digits of a day and similar for 'MM', "yy' or 'yyyy'.

Variable: Yes

Restrictions: None

Variables in rows

If this attribute is checked, the names of the variables will be taken from the first row. If it's not checked the names will come from the first column.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note   For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing [Alt-Return] or by clicking the 🖉 button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.4.5 CSV data file

A CSV data file node is used to load external data from a file and make it available for data-driven testing. *CSV* stands for *Comma-separated Values*, a more or less standard plain text file format. Each line in the file contains one set of data with the values separated by a separator character, often but not always a comma (','). For use with a CSV data file node the first line of the CSV file must contain the names of the variables to bind. The rest of the lines should contain the values to be used for the iteration steps.

Unfortunately, with CSV files there are non-uniform definitions for things like quoting, white-space, multi-line values or embedded separator characters. Two de-facto standards exist, one used by Microsoft Excel and one by the rest of the world. QF-Test supports both of these. For further details about the data-driving mechanism please see chapter 21[(261)].

Contained in: Any

Children: None

Execution: The CSV data file node loads the data from the CSV file and expands variable values in the table according to the option When binding variables, expand values immediately[(513)]. Each row is expanded individually left to right, meaning that - within the same row - a cell may refer to a variable bound in a column further to the left. Then the CSV data file registers itself with the Data driver context. A property group called like the node will be created additionally. That group contains the variables `size` and `totalsize`. `size` shows the number of data rows with taking care about iteration intervals. The variable `totalsize` shows the total number of data rows without taking care about iteration intervals. When all Data binders have been registered the Test set[(525)] will query the CSV data file in order to iterate over the available sets of data. If no such context is available the respective property group will be extended with all variables.

Attributes:

Figure 40.18: CSV data file attributes

Name

The Name of a Data binder is mandatory. It is used to distinguish Data binders in the same Data driver context. A Break[(604)] node executed during data-driven testing can be used to break out of a specific loop by referring to the Data binder's Name.

Variable: Yes

Restrictions: None

Iteration counter

The name of the variable that the iteration counter will be bound to.

Variable: Yes

Restrictions: None

Iteration ranges

An optional set of indexes or ranges to use from the bound data. This is especially useful during test development in order to run sample tests with just a single index or a subset of the given data.

Ranges are separated by ','. Each range is either a single index or an inclusive range of the form 'from-to' or 'from:to' where 'to' is optional. Indexes or ranges may be specified multiple times, overlap or be given in descending order. Indexes are 0-based, negative indexes are counted from the end, -1 being the last item. An invalid syntax or an index outside the valid data range will cause a `BadRangeException`[(854)].

The following table shows some example range specifications and the resulting indexes, based on a set of 20 entries.

Iteration ranges	Resulting indexes
0	[0]
-2, -1	[18, 19]
1-2,4:5	[1, 2, 4, 5]
18:,-3-	[18, 19, 17, 18, 19]
3-2,16:15	[3, 2, 16, 15]

Table 40.11: Iteration range examples

Note   The value bound for the Iteration counter reflects the index in the current interval, not the counter of actual iterations, e.g. if you specify '2' there will be a single iteration with the Iteration counter bound to '2', not '0'.

Variable: Yes

Restrictions: Valid syntax and index values

CSV file name

The name of the CSV file to get the test data from. Relative path names are resolved relative to the directory of the test suite.

The button above the attribute brings up a dialog in which you can select the CSV file interactively. You can also get to this dialog by pressing ⟨Shift-Return⟩ or ⟨Alt-Return⟩ when the focus is in the text field.

Variable: Yes

Restrictions: Must not be empty

File encoding

An optional encoding for the CSV file, "UTF-8" for example. If no encoding is specified, the file will be read with the default encoding of the Java VM.

Variable: Yes

Restrictions: The encoding must be supported by the Java VM.

Read Microsoft Excel CSV format

If this option is active QF-Test will try to parse the CSV file using the format used by Microsoft Excel.

Variable: Yes

Restrictions: None

Separator character

In this attribute you can specify the character to be used as separator for data values within the CSV file. If no separator is defined a comma (',') is used as the default value. If Read Microsoft Excel CSV format[582] is activated this attribute is ignored.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ✏ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.4.6 Data loop

A Data loop node is a simple loop with a single variable bound as the iteration counter. It is useful for executing Test cases[(518)] multiple times by placing them inside a Test set[(525)] with a Data driver[(562)] holding a Data loop.

Contained in: Any

Children: None

Execution: During execution all the Data loop node does is register itself with the Data driver context. The iteration counter will be used as variable. A property group called like the node will be created additionally. That group contains the variables `size` and `totalsize`. `size` shows the number of data rows with taking care about iteration intervals. The variable `totalsize` shows the total number of data rows without taking care about iteration intervals. When all Data binders have been registered the Test set[(525)] will query the Data loop in order to iterate over the available sets of data. If no such context is available the respective property group will be extended with all variables.

Attributes:

Figure 40.19: Data loop attributes

Name

>   The Name of a Data binder is mandatory. It is used to distinguish Data binders in the same Data driver context. A Break[(604)] node executed during data-driven testing can be used to break out of a specific loop by referring to the Data binder's Name.

>   **Variable:** Yes

>   **Restrictions:** None

Iteration counter

>   The name of the variable that the iteration counter will be bound to.

>   **Variable:** Yes

>   **Restrictions:** None

Iteration ranges

>   An optional set of indexes or ranges to use from the bound data.   This is especially useful during test development in order to run sample tests with just a single index or a subset of the given data.

>   Ranges are separated by ','. Each range is either a single index or an inclusive range of the form 'from-to' or 'from:to' where 'to' is optional. Indexes or ranges

may be specified multiple times, overlap or be given in descending order. Indexes are 0-based, negative indexes are counted from the end, -1 being the last item. An invalid syntax or an index outside the valid data range will cause a `BadRangeException`[(854)].

The following table shows some example range specifications and the resulting indexes, based on a set of 20 entries.

Iteration ranges	Resulting indexes
0	[0]
-2, -1	[18, 19]
1-2,4:5	[1, 2, 4, 5]
18:,-3-	[18, 19, 17, 18, 19]
3-2,16:15	[3, 2, 16, 15]

Table 40.12: Iteration range examples

The value bound for the Iteration counter reflects the index in the current interval, not the counter of actual iterations, e.g. if you specify '2' there will be a single iteration with the Iteration counter bound to '2', not '0'.

Variable: Yes

Restrictions: Valid syntax and index values

Number of iterations

The number of iterations of the loop.

Variable: Yes

Restrictions: > 0

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing [Alt-Return] or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.5 Procedures

Procedures are a means to collect some of the basic building blocks of a test suite like events[(681)] and checks[(707)] into a larger, reusable structure.

Procedures can be called from any other part of the test suite and even from different suites (see section 24.1[(298)]). You can pass parameters to a procedure in the form of variable definitions (see chapter 6[(103)]).

A typical example would be a procedure that selects a menu item in a menu. Its parameters could be the client name of the SUT, the name of the menu and the name of the menu item.

40.5.1 Procedure

A Procedure is a Sequence[(536)] that is executed from some other place by a Procedure call[(589)].

The parameters of the procedure are not defined explicitly. Instead they are a consequence of the variable references in the children of the Procedure. You may want to define fallback values for some or all of the parameters in the Variable definitions[(587)]. In any case it is a good idea to document the required parameters in the Comment[(588)] attribute.

A Procedure can return a value to the calling node with the help of a Return[592] node. Without such a node a Procedure implicitly returns the empty string.

Contained in: Package[594], Procedures[595]

Children: Any

Execution: The Procedure's variables are bound as fallback values. The child nodes are executed one by one, then the fallback values are unbound again.

Attributes:

Figure 40.20: Procedure Attributes

Name

> A Procedure is identified by its name and the names of its Package[594] ancestors, so you should assign a name with a meaning that is easy to recognize and remember.

> **Variable:** No

> **Restrictions:** Must not be empty or contain the characters '.' or '#'.

Variable definitions

Here you can define default or fallback values for the Procedure's parameters (see chapter 6[103]). Defining these values also serves as documentation and is a valuable time-saver when using the dialog to select the Procedure for the Procedure name[590] attribute of a Procedure call[589]. See section 2.2.5[17] about how to work with the table.

4.2+   In case you want to re-set the order of the parameters like they are sorted in the called procedure, you can select Re-set parameter order .

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run log node for a sequence will propagate, can be limited.

Note   This value has no effect on the way exceptions are handled. It only affects the error states of the run log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.7[11]). It also has no effect on the creation of compact run logs (see command line argument `-compact`[870]). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing [Alt-Return] or by clicking the ![button icon] button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.5.2 Procedure call

This node calls a Procedure[(586)] in the same or a different test suite, meaning that execution continues in the Procedure. When the Procedure is finished, the value returned by the Procedure is bound to the variable defined in the Variable for return value[(590)] attribute and execution returns to the Procedure call and thus to its parent node.

The name of the Procedure[(586)] to call is determined by its Name[(587)] and the Names[(594)] of its Package[(594)] parents. These are concatenated with a dot ('.') as separator, starting with the outermost Package and ending in the Procedure's name. Thus to call a Procedure named expandNode in a Package named tree that is itself a child of a Package named main, set the Procedure name[(590)] attribute to **main.tree.expandNode**.

See section 24.1[(298)] about how to call a Procedure in a different test suite.

Contained in: All kinds of sequences[(518)].

Children: None

Execution: The Variable definitions[(591)] of the Procedure call are bound, the target Procedure[(586)] is determined and execution passed to it. After the Procedure returns, the Procedure call's variables are unbound again.

Attributes:

Figure 40.21: Procedure call Attributes

Name

The full name of the Procedure[(586)], created from the Names[(594)] of its Package[(594)] parents and its own Name[(587)], joined by a dot. The "Select procedure" button (•) above the attribute brings up a dialog in which you can select the Procedure interactively. By selecting the "Copy parameters" checkbox you can adopt the Procedure's default values as parameters for the Procedure call node to save typing.

Variable: Yes

Restrictions: Must not be empty.

Variable for return value

The value returned by the Procedure, either through a Return[(592)] node or the empty string, is bound to the variable defined in this optional attribute. Additionally, the most recent return value is always available as the special variable ${qftest:return}.

6.1+   The name of the variable shows in the test suite tree - in blue when it is a global

variable, in black when it is local.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure[(586)], Dependency[(548)] or Test case[(518)] node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test case node or, if there is no such node in the topmost node on the variables stack, falling back to the global bindings if necessary. See chapter 6[(103)] for a detailed explanation of variable binding and lookup.

In order to predefine the option use Enable 'Local variable' attribute by default[(513)].

Variable: No

Restrictions: None

Variable definitions

This is where you define the parameter values for the Procedure[(586)] (see chapter 6[(103)]). See section 2.2.5[(17)] about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note   For detailed documentation, especially for Test set, Test case or Procedure nodes,

this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ✏ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.5.3 Return

This node can be used to return from a Procedure[586] prematurely and also to pass a return value to the calling node.

From a script, the same effect can be achieved by raising a ReturnException.

Contained in: All kinds of sequences[518].

Children: None

Execution: A ReturnException is thrown. If it is caught by a matching Procedure, the Procedure is terminated and the return value passed to the caller. If the node is executed outside a Procedure it will lead to an error.

Attributes:

Figure 40.22: Return Attributes

Return value

The value to return from the PROCEDURE. May be empty in which case the empty string is returned.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-TEST ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for TEST SET, TEST CASE or PROCEDURE nodes,

this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ✎ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.5.4 Package

The only use of Packages is to structure the Procedures[(586)] of a test suite. The Name[(594)] of a Package is part of the Procedure's fully qualified name, as required by a Procedure call[(589)].

Contained in: Package[(594)], Procedures[(595)]

Children: Package[(594)], Procedure[(586)]

Execution: Cannot be executed.

Attributes:
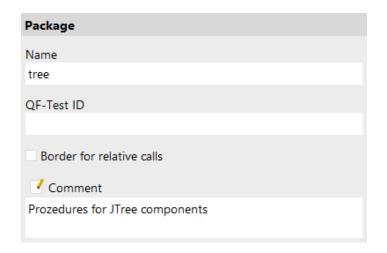
Figure 40.23: Package Attributes

Name

The name of a Package is part of the identification of the Procedures[586] it contains, so you should assign a name with a meaning that is easy to recognize and remember.

Variable: No

Restrictions: Must not be empty or contain the characters '.' or '#'.

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Border for relative calls

This flag determines whether relative procedure calls, test calls or dependency references are allowed within that certain node. Relative calls passing that border are not allowed. If that attribute is not specified in the hierarchy, no relative calls are allowed.

Variable: No

Restrictions: None

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.5.5 Procedures

This is the root of all Packages⁽⁵⁹⁴⁾ and Procedures⁽⁵⁸⁶⁾.

Contained in: Root node

Children: Package⁽⁵⁹⁴⁾, Procedure⁽⁵⁸⁶⁾

Execution: Cannot be executed.

Attributes:

Procedures

QF-Test ID

☐ Border for relative calls

✎ Comment

Figure 40.24: Procedures Attributes

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Border for relative calls

This flag determines whether relative procedure calls, test calls or dependency references are allowed within that certain node. Relative calls passing that border are not allowed. If that attribute is not specified in the hierarchy, no relative calls are allowed.

Variable: No

Restrictions: None

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note   For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that

are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the 🖉 button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.6   Control structures

Besides the standard sequence[(518)] types QF-Test has a set of special control structures. Conditional processing is handled via If[(605)], Elseif[(609)] and Else[(613)] nodes. Loops[(597)] and While[(600)] nodes can be aborted with a Break[(604)] node. Exceptions are handled by Try[(616)], Catch[(619)] and Finally[(623)] nodes.

Beyond that, full scripting is available for the Jython language (formerly called *JPython*), Groovy and JavaScript as documented in chapter 11[(161)].

40.6.1   Loop

This node is basically the same as a Sequence[(536)] except that its children can be executed more than once. This is useful in two ways. For one thing, a test sequence that executes OK a hundred times is more trustworthy than a sequence that only runs once. The other use is to run a number of similar jobs with slight variations. To that end, the count of the current iteration is bound as a variable during execution.

Special Loops with varying increments can be achieved by changing the value of the Iteration counter[(599)] during execution.

Execution of a Loop can be terminated prematurely with the help of a Break[(604)] node. An optional Else[(613)] node may be placed at the end of the Loop. It is executed if all iterations of the Loop are run through completely without hitting a Break.

Contained in: All kinds of sequences[(518)].

Children: Any

Execution: The Variable definitions[(599)] of the Loop are bound. The Iteration counter[(599)] is initialized to 0 and the child nodes are executed one by one. For each iteration the

Iteration counter is increased by one and the children are executed again. After the final execution of the last child is complete, the Iteration counter and the Variable definitions are unbound again.

Attributes:

Figure 40.25: Loop attributes

Name

> The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and say something about the function of the sequence.

> **Variable:** No

> **Restrictions:** None

Number of iterations

> The number of iterations of the loop.

Variable: Yes

Restrictions: > 0

Iteration counter

The name of the variable that will hold the iteration count during the execution. Make sure to use different Iteration counter names for nested loops.

Variable: Yes

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see chapter 6[(103)]). See section 2.2.5[(17)] about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run log node for a sequence will propagate, can be limited.

Note   This value has no effect on the way exceptions are handled. It only affects the error states of the run log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.7[(11)]). It also has no effect on the creation of compact run logs (see command line argument −compact[(870)]). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

The QF-Test ID of the Loop node can be used in a Break[(604)] node to terminate an outer loop explicitly when loops are nested.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note
For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing [Alt-Return] or by clicking the ✎ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.6.2 While

This is a sequence that is executed repeatedly as long as a condition is fulfilled.

The loop can be terminated prematurely with the help of a Break[(604)] node.

Contained in: All kinds of sequences[(518)].

Children: Any

Execution: The Variable definitions[(602)] of the While node are bound. If the condition evaluates to true, the child nodes are executed one by one. This is repeated until the condition evaluates to false or the loop is terminated through a Break[(604)] node or an Exception. Finally the Variable definitions are unbound again.

Attributes:

Figure 40.26: While attributes

Condition

A condition is an expression that evaluates to either true or false. QF-Test discriminates between simple expression that it evaluates itself and complex expressions that are passed to the Jython script language to evaluate.

An empty string or the string `false` (regardless of case) is interpreted as false, the string `true` as true. Whole numbers are true if and only if they are non-zero.

Evaluating expressions in Jython opens the way for powerful expression handling. Jython supports the standard operators ==, !=, >, >=, < and <=. You can combine expressions with `and` and `or` and define their priority with braces.

Note — Accessing QF-Test variables in a condition follows the same rules as in Jython scripts (see section 11.2.3[165]). You can use the standard QF-Test syntax `$(...)` and `${...:...}` for numeric or boolean values. String values should be accessed with `rc.lookup(...)`.

Important: Im you want to compare strings (as opposed to numbers) you need to excape them by single or double inverted commas for Jython. Else Jython would

interpret the string as a Jython variable, which, of course would not be defined, and thus lead to a syntax error.

Some examples:

Expression	Value
Empty String	False
0	False
21	True
False	False
True	True
abc abc	Syntax error
25 > 0	True
${qftest:batch}	True if QF-Test is run in batch mode
not ${qftest:batch}	True if QF-Test is run in interactive mode
rc.lookup("system", "java.version") == "1.3.1"	True if JDK Version is 1.3.1
rc.lookup("system", "java.version")[0] == "1"	True is JDK Version starts with 1
(1 > 0 and 0 == 0) or 2 < 1	True

Table 40.13: Condition examples

Variable: Yes

Restrictions: Valid syntax

Script language

This attribute determines the interpreter in which to run the script, or in other words, the scripting language to use. Possible values are "Jython", "Groovy" and "JavaScript".

Variable: No

Restrictions: None

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and say something about the function of the sequence.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the

execution of the sequence's child nodes (see chapter 6⁽¹⁰³⁾). See section 2.2.5⁽¹⁷⁾ about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run log node for a sequence will propagate, can be limited.

Note This value has no effect on the way exceptions are handled. It only affects the error states of the run log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.7⁽¹¹⁾). It also has no effect on the creation of compact run logs (see command line argument -compact⁽⁸⁷⁰⁾). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

The QF-Test ID of the While node can be used in a Break⁽⁶⁰⁴⁾ node to terminate an outer loop explicitly when loops are nested.

Variable: No

Restrictions: Must not be empty, contain any of the characters '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽⁴⁷⁶⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that

are much better suited to this task. The option <u>External editor command</u>[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the 🖊 button.

You can trigger special behaviors of some nodes using doctags, please see <u>Doctags</u>[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.6.3   Break

This node is used to terminate a <u>Loops</u>[(597)] or a <u>While</u>[(600)] node prematurely.

Contained in: All kinds of <u>sequences</u>[(518)].

Children: None

Execution: A `BreakException` is thrown. If it is caught by a matching loop, the loop is terminated, otherwise it will lead to an error.

Attributes:

Break

QF-Test loop ID

QF-Test ID

Delay before (ms)   Delay after (ms)

🖊 Comment

Break innermost loop

Figure 40.27: Break attributes

QF-Test loop ID

For nested loops you can specify the loop to terminate by specifying the QF-Test ID[(599)] of a Loop[(597)] node and refer to it here. It works with QF-Test ID[(603)] for a While[(600)] node respectively. In case this field is empty the innermost loop is terminated. In case you want to break an iteration raised by a Data driver[(562)] node, you should specify the value of the 'Name' attribute of the respective Data driver[(562)] node.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.6.4 If

Like in Java the child nodes of this node are executed only if a condition evaluates to true. However QF-Test differs from common programming languages in the way alternative branches are arranged.

Contained in: All kinds of sequences[(518)].

Children: Any executable node, followed by an arbitrary number of Elseif[(609)] nodes with an optional Else[(613)] at the end.

Execution: The Variable definitions[(608)] of the If node are bound. If the condition evaluates to true, the normal child nodes are executed one by one. Otherwise the conditions of the Elseif[(609)] nodes are evaluated and the first Elseif node whose condition evaluates to true is executed. If none of the conditions are true of no Elseif nodes exist, the Else[(613)] node is executed, if one exists. Finally the Variable definitions are unbound again.

Attributes:

Figure 40.28: If attributes

Condition

A condition is an expression that evaluates to either true or false. QF-Test discriminates between simple expression that it evaluates itself and complex expressions that are passed to the Jython script language to evaluate.

An empty string or the string `false` (regardless of case) is interpreted as false, the string `true` as true. Whole numbers are true if and only if they are non-zero.

Evaluating expressions in Jython opens the way for powerful expression handling. Jython supports the standard operators ==, !=, >, >=, < and <=. You can combine expressions with `and` and `or` and define their priority with braces.

Note   Accessing QF-Test variables in a condition follows the same rules as in Jython scripts (see section 11.2.3[(165)]). You can use the standard QF-Test syntax `$(...)` and `${...:...}` for numeric or boolean values. String values should be accessed with `rc.lookup(...)`.

Important: Im you want to compare strings (as opposed to numbers) you need to excape them by single or double inverted commas for Jython. Else Jython would interpret the string as a Jython variable, which, of course would not be defined, and thus lead to a syntax error.

Some examples:

Expression	Value
Empty String	False
0	False
21	True
False	False
True	True
abc abc	Syntax error
25 > 0	True
${qftest:batch}	True if QF-Test is run in batch mode
not ${qftest:batch}	True if QF-Test is run in interactive mode
rc.lookup("system", "java.version") == "1.3.1"	True if JDK Version is 1.3.1
rc.lookup("system", "java.version")[0] == "1"	True is JDK Version starts with 1
(1 > 0 and 0 == 0) or 2 < 1	True

Table 40.14: Condition examples

Variable: Yes

Restrictions: Valid syntax

Script language

This attribute determines the interpreter in which to run the script, or in other

words, the scripting language to use. Possible values are "Jython", "Groovy" and "JavaScript".

Variable: No

Restrictions: None

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and say something about the function of the sequence.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see chapter 6[(103)]). See section 2.2.5[(17)] about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run log node for a sequence will propagate, can be limited.

Note   This value has no effect on the way exceptions are handled. It only affects the error states of the run log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.7[(11)]). It also has no effect on the creation of compact run logs (see command line argument `-compact`[(870)]). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ✎ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.6.5   Elseif

This node is an alternative branch in an If[(605)] node. If the condition of the If node evaluates to false, the first Elseif node whose condition is true is executed.

Contained in: If[(605)]

Children: Any

Execution: The Variable definitions[(611)] of the Elseif are bound and its child nodes executed one by one. After the execution of the last child is complete, the variables are unbound again.

Attributes:

Figure 40.29: Elseif attributes

Condition

A condition is an expression that evaluates to either true or false. QF-Test discriminates between simple expression that it evaluates itself and complex expressions that are passed to the Jython script language to evaluate.

An empty string or the string `false` (regardless of case) is interpreted as false, the string `true` as true. Whole numbers are true if and only if they are non-zero.

Evaluating expressions in Jython opens the way for powerful expression handling. Jython supports the standard operators ==, !=, >, >=, < and <=. You can combine expressions with `and` and `or` and define their priority with braces.

Note Accessing QF-Test variables in a condition follows the same rules as in Jython scripts (see section 11.2.3[165]). You can use the standard QF-Test syntax `$(...)` and `${...:...}` for numeric or boolean values. String values should be accessed with `rc.lookup(...)`.

Important: Im you want to compare strings (as opposed to numbers) you need to excape them by single or double inverted commas for Jython. Else Jython would

interpret the string as a Jython variable, which, of course would not be defined, and thus lead to a syntax error.

Some examples:

Expression	Value
Empty String	False
0	False
21	True
False	False
True	True
abc abc	Syntax error
25 > 0	True
${qftest:batch}	True if QF-Test is run in batch mode
not ${qftest:batch}	True if QF-Test is run in interactive mode
rc.lookup("system", "java.version") == "1.3.1"	True if JDK Version is 1.3.1
rc.lookup("system", "java.version")[0] == "1"	True is JDK Version starts with 1
(1 > 0 and 0 == 0) or 2 < 1	True

Table 40.15: Condition examples

Variable: Yes

Restrictions: Valid syntax

Script language

This attribute determines the interpreter in which to run the script, or in other words, the scripting language to use. Possible values are "Jython", "Groovy" and "JavaScript".

Variable: No

Restrictions: None

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and say something about the function of the sequence.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the

execution of the sequence's child nodes (see chapter 6[103]). See section 2.2.5[17] about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run log node for a sequence will propagate, can be limited.

This value has no effect on the way exceptions are handled. It only affects the error states of the run log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.7[11]). It also has no effect on the creation of compact run logs (see command line argument -compact[870]). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets

you define an external editor in which comments can be edited conveniently by pressing ⌑Alt-Return⌑ or by clicking the ✎ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.6.6 Else

An Else node is executed if neither the condition of its If[(605)] parent, nor the condition of its Elseif[(609)] siblings evaluate to true.

Contained in: If[(605)], Loop[(597)], Try[(616)]

Children: Any

Execution: The Variable definitions[(614)] of the Else are bound and its child nodes executed one by one. After the execution of the last child is complete, the variables are unbound again.

Attributes:

Figure 40.30: Else attributes

Name

> The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and say something about the function of the sequence.

> **Variable:** No

> **Restrictions:** None

Variable definitions

> This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see chapter 6[103]). See section 2.2.5[17] about how to work with the table.

> **Variable:** Variable names no, values yes

> **Restrictions:** None

Maximum error level

> When a warning, error or exception occurs during a test run, the state of the corresponding node of the run log is set accordingly. This state is normally

propagated to the parent node in a way that ensures that the error state of a run log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run log node for a sequence will propagate, can be limited.

This value has no effect on the way exceptions are handled. It only affects the error states of the run log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.7[11]). It also has no effect on the creation of compact run logs (see command line argument -compact[870]). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing [Alt-Return] or by clicking the ✎ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.6.7 Try

A Try is a special sequence whose semantic equates the Java `try-catch-finally` composition. As in Python, this composition was extended to support optional `else` blocks. A Try behaves the same way as a special Sequence[(536)] with the extension, that exception handling is possible. Like a Sequence it has a set of normal child nodes that it executes one by one. After these may come an arbitrary number of Catch[(619)] nodes with an optional Else[(613)] node followed by an optional Finally[(623)] node at the end.

If an exception is thrown during the execution of one of the normal child nodes, the Catch nodes are tested for whether they are able to catch that exception. The first one found is executed and the Try will be exited normally afterwards without continuing with the execution of the normal child nodes and without passing the exception on. If no matching Catch is found, the exception will terminate the Try immediately (almost, see below) and be passed onto the Try's parent.

A possible Else[(613)] child node at the end of a try node will be executed, if and only if no Catch[(619)] nodes had been executed. This means, it is executed, when no exception in the try block was thrown.

If the Try has a Finally[(623)] child node, this node will be executed just before the Try finishes, no matter whether an exception is thrown and whether it is handled or not.

Contained in: All kinds of sequences[(518)].

Children: Any executable node, followed by an arbitrary number of Catch[(619)] nodes with an optional Else[(613)] node and/or an optional Finally[(623)] node at the end.

Execution: The Variable definitions[(617)] of the Try are bound and its normal child nodes executed one by one. If an exception is thrown, execution of the normal children is terminated. If a Catch node with a matching Exception class[(620)] is found it is executed. Before exiting the Try its Finally node is executed unconditionally. After unbinding the Variable definitions, the Try is either exited cleanly if no exception was thrown or the exception was caught, or it passes on the uncaught exception.

Attributes:

Figure 40.31: Try attributes

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and say something about the function of the sequence.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see chapter 6[(103)]). See section 2.2.5[(17)] about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run log is set accordingly. This state is normally

propagated to the parent node in a way that ensures that the error state of a run log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run log node for a sequence will propagate, can be limited.

This value has no effect on the way exceptions are handled. It only affects the error states of the run log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.7[(11)]). It also has no effect on the creation of compact run logs (see command line argument -compact[(870)]). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

For a Try node, the error state of the run log is additionally affected by the Maximum error level[(621)] of the Catch[(619)] node that handles an exception.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.6.8   Catch

A Catch is a special Sequence[(536)] that can only be placed inside a Try[(616)] node or a Dependency[(548)]. Its job is to catch exceptions that may arise during the execution of a Try's or a Test case[(518)] with a Dependency.

A Catch can handle an exception if the class of the exception is the same as the Catch node's Exception class[(620)] attribute or a derived class thereof, just as in Java.

Contained in: Try[(616)], Dependency[(548)]

Children: Any

Execution: The Variable definitions[(621)] of the Catch are bound and its child nodes executed one by one. After the execution of the last child is complete, the variables are unbound again.

Attributes:

Figure 40.32: Catch attributes

Exception class

> This *ComboBox* lets you select the class of the exception that is to be caught. All QF-Test exceptions are derived from the class <u>TestException</u>[(850)]. For details about the possible exceptions see <u>chapter 41</u>[(850)].
>
> **Variable:** No
>
> **Restrictions:** None

Expected message

You can further qualify the exception to catch by specifying a message to look for. If this attribute is empty, all exceptions of the specified class are caught. Otherwise it is compared to `exception.getMessage()` and the exception is caught only in case of a match.

Variable: Yes

Restrictions: Valid regexp if required.

As regexp

If this attribute is set, the exception message is matched against a regexp (see section 47.3[909]) instead of comparing plain strings.

Variable: Yes

Restrictions: None

Match against localized message

Most exceptions have two kinds of error message: The raw message is typically some short English text whereas the localized message contains more details and is either English or German, depending on the current language settings of QF-Test. Both are shown in the run log. If this attribute is set, the localized exception message is used for comparison, otherwise the raw message. The latter is usually preferable as it doesn't depend on language settings so no regexp is needed in order to ensure the correct handling of the different languages.

Variable: Yes

Restrictions: None

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and say something about the function of the sequence.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see chapter 6[103]). See section 2.2.5[17] about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

In contrast to the Maximum error level⁽⁵³⁷⁾ of other sequences⁽⁵¹⁸⁾, this attribute does not determine the error state propagated by the run log node for the Catch node itself, but for the log of its parent Try⁽⁶¹⁶⁾ node, provided that the Catch is executed in order to handle an exception.

The error state for any warnings, errors or exceptions that happen during the execution of the Catch node are **not** limited by the setting of this attribute. Otherwise problems occurring during exception handling might accidentally go unnoticed.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽⁴⁷⁶⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note   For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽⁴³¹⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the ✏ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags⁽¹²¹⁵⁾.

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.6.9   Finally

A Finally node, which can only be placed at the end of a Try node, will always be executed as the last thing just before exiting the Try, no matter what happened there. This is used primarily to ensure that cleanup code like removing a temporary file or terminating a process is executed under any conditions.

Contained in: Try[(616)]

Children: Any

Execution: The Variable definitions[(624)] of the Finally are bound and its child nodes executed one by one. After the execution of the last child is complete, the variables are unbound again.

Attributes:

Figure 40.33: Finally attributes

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and say something about the function of the sequence.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see chapter 6[(103)]). See section 2.2.5[(17)] about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run log node for a sequence will propagate, can be limited.

Note This value has no effect on the way exceptions are handled. It only affects the error states of the run log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.7[(11)]). It also has no effect on the creation of compact run logs (see command line argument `-compact`[(870)]). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.6.10   Throw

If you need to handle an exceptional situation, you can use this node to throw an explicit Exception.

Contained in: All kinds of sequences[(518)].

Children: None

Execution: A   UserException[(858)]   is   thrown,   its   message   taken   from   the Exception message[(626)] attribute.

Attributes:

Figure 40.34: Throw attributes

Exception message

An arbitrary message for the <u>UserException</u>[(858)] to throw.

Variable: Yes

Restrictions: Must not be empty.

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the <u>Default delay</u>[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note   For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that

are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ▨ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.6.11   Rethrow

An exception that was caught by a Catch[619] node can be thrown again with the help of a Rethrow node. This is especially useful if you need to catch all kinds of exceptions except one. To handle that case, create a Try[616] node with a Catch for the special exception followed by a Catch for a `TestException`[850]. Then place a Rethrow in the first Catch node.

Contained in: All kinds of sequences[518]. The Rethrow node doesn't have to be placed directly below a Catch node.

Children: None

Execution: The last exception caught by a Catch[619] node is thrown again. If no such exception exists, a `CannotRethrowException`[858] is thrown.

Attributes:

Rethrow exception
QF-Test ID
Delay before (ms)   Delay after (ms)
▨ Comment
Pass this exception on

Figure 40.35: Rethrow attributes

QF-Test ID

> At the moment the QF-Test ID attribute has no meaning for this type of node.
>
> **Variable:** No
>
> **Restrictions:** Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

> These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.
>
> **Variable:** Yes
>
> **Restrictions:** Valid number >= 0

Comment

> Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note

> For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing ⌊Alt-Return⌋ or by clicking the ✎ button.
>
> You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].
>
> If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.
>
> **Variable:** Yes
>
> **Restrictions:** None

40.6.12 Server script

Server side scripts are executed by an interpreter (Jython, Groovy or JavaScript) embedded into QF-Test. Scripting is explained in chapter 11[161] and chapter 48[915]. As server side scripts run embedded to QF-Test those scripts cannot interact with the SUT. It is recommended that Server side scripts be used for cases that work without the SUT or time consuming operations like accessing databases or files.

Contained in: All kinds of sequences[518].

Children: None

Execution: The script is executed by an embedded interpreter.

Attributes:

Figure 40.36: Server script attributes

Script

The script to execute.

Note   You may use QF-Test variables of the syntax $(var) or ${group:name} in Jython scripts. They will be expanded before the script is passed to the Jython interpreter. This can lead to unexpected behavior. `rc.lookup(...)` is the preferred method in this case (see section 11.2.3[(166)] for details).

Note   In spite of syntax highlighting and automatical indentation this attribute might not be the right place to write complex scripts. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which scripts can be edited conveniently by pressing

Alt-Return or by clicking the ✏ button. Complex scripts can also be written as separate modules which can then be imported for use in this attribute. See chapter 48[915] for details.

Variable: Yes

Restrictions: Valid syntax

Templates

This dropdown menu contains a list of useful template scripts. The available templates will differ depending on the chosen script type and interpreter.

When you choose one of these templates, the current contents of your script will be replaced.

You can add your own templates to this menu by choosing "Open user templates directory" and placing your template files there. The following file types are valid:

- **[directory]:** Will be converted into a sub menu.
- **.py:** A Jython script template.
- **.groovy:** A Groovy script template.
- **.js:** A JavaScript script template.

Script language

This attribute determines the interpreter in which to run the script, or in other words, the scripting language to use. Possible values are "Jython", "Groovy" and "JavaScript".

Variable: No

Restrictions: None

Name

The name of a script is a kind of short description. It is displayed in the tree view, so it should be concise and say something about the function of the script.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.6.13 SUT script

Client side scripts are executed by an interpreter (Jython, Groovy or JavaScript) that QF-Test embeds into the SUT. Scripting is explained in chapter 11[161] and chapter 48[915]. As client side scripts run in the SUT you should use them to access the components and properties of the SUT.

Contained in: All kinds of sequences[518].

Children: None

Execution: The script is executed inside the SUT by an embedded interpreter.

Attributes:

Figure 40.37: SUT script attributes

Client

The name of the SUT client process in which to execute the script.

Variable: Yes

Restrictions: Must not be empty.

Script

The script to execute.

Note You may use QF-Test variables of the syntax $(var) or ${group:name} in Jython scripts. They will be expanded before the script is passed to the Jython interpreter.

This can lead to unexpected behavior. `rc.lookup(...)` is the preferred method in this case (see section 11.2.3⁽¹⁶⁶⁾ for details).

In spite of syntax highlighting and automatical indentation this attribute might not be the right place to write complex scripts. There are many excellent editors that are much better suited to this task. The option External editor command⁽⁴³¹⁾ lets you define an external editor in which scripts can be edited conveniently by pressing

Alt-Return or by clicking the ✏ button. Complex scripts can also be written as separate modules which can then be imported for use in this attribute. See chapter 48⁽⁹¹⁵⁾ for details.

Variable: Yes

Restrictions: Valid syntax

Templates

This dropdown menu contains a list of useful template scripts. The available templates will differ depending on the chosen script type and interpreter.

When you choose one of these templates, the current contents of your script will be replaced.

You can add your own templates to this menu by choosing "Open user templates directory" and placing your template files there. The following file types are valid:

- **[directory]:** Will be converted into a sub menu.
- **.py:** A Jython script template.
- **.groovy:** A Groovy script template.
- **.js:** A JavaScript script template.

Script language

This attribute determines the interpreter in which to run the script, or in other words, the scripting language to use. Possible values are "Jython", "Groovy" and "JavaScript".

Variable: No

Restrictions: None

GUI engine

The GUI engine in which to execute the script. Only relevant for SUTs with more than one GUI engine as described in chapter 43⁽⁸⁸⁷⁾.

Variable: Yes

Restrictions: See chapter 43⁽⁸⁸⁷⁾

Name

> The name of a script is a kind of short description. It is displayed in the tree view, so it should be concise and say something about the function of the script.
>
> **Variable:** No
>
> **Restrictions:** None

QF-Test ID

> At the moment the QF-Test ID attribute has no meaning for this type of node.
>
> **Variable:** No
>
> **Restrictions:** Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

> These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.
>
> **Variable:** Yes
>
> **Restrictions:** Valid number >= 0

Comment

> Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

> For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.
>
> You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].
>
> If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.
>
> **Variable:** Yes
>
> **Restrictions:** None

40.7 Processes

In order to enable QF-Test to communicate with the SUT, the SUT's process must be started by QF-Test. For details about how the application is started by QF-Test and

which kind of node is most applicable in your case, see chapter 44[(889)].

The following nodes are used to start or stop processes, to wait for a connection with the SUT or to wait for the termination of a process and check its exit value. The number of concurrent processes managed by QF-Test is limited only by the underlying system, but you have to assign a unique name to each process by which other nodes will identify it.

We will refer to processes run by QF-Test as clients. QF-Test distinguishes between two types of clients: arbitrary processes that are simply started and stopped and SUT clients, the actual Java applications that QF-Test interacts with.

Standard input and output of a client is redirected to a terminal that you can open from the ‖Clients‖ menu. The client's output is also stored in the log of a test run.

40.7.1 Start Java SUT client

This node provides the standard and most flexible way of starting an SUT client. To use it, you must know the Java command that starts your application. If the SUT is run through a script, use a Start SUT client[(639)] node instead (see chapter 44[(889)]).

Contained in: All kinds of sequences[(518)].

Children: None

Execution: The command line for the program is built from the attributes and the process is started. Its input and output are redirected to QF-Test.

Attributes:

Figure 40.38: Start Java SUT client attributes

Client

This is the identifier for the SUT client. It must remain unique as long as the process is alive. Other nodes will refer to the client by this name.

Variable: Yes

Restrictions: Must not be empty

Executable

The Java program to run, typically `java` or `javaw` on Windows. If you want to run a specific Java virtual machine you should give the full path name of the executable.

The "Select file" button ▯ brings up a dialog in which you can select the program file interactively. You can also get to this dialog by pressing Shift-Return or Alt-Return, when the focus is in the text field.

Variable: Yes

Restrictions: Must not be empty

Directory

Here you can set the working directory for the program. If you leave this value empty, the program will inherit QF-Test's working directory.

The "Select directory" button ▯ brings up a dialog in which you can select the directory interactively. You can also get to this dialog by pressing Shift-Return or Alt-Return, when the focus is in the text field.

Variable: Yes

Restrictions: Must be empty or an existing directory

Class name

The fully qualified name of the Java class of the SUT whose `main(String[])` method will be called. If your application is run from an executable jar archive with the `-jar` parameter, you must leave this attribute empty.

Note QF-Test can only run the `main` method of the class if both the class itself and the main method are declared `public`. Please verify this if you get an `IllegalAccessException` when you try to start your application.

Variable: Yes

Restrictions: Valid class name

Executable parameters

The command line arguments for the Java executable. Put each parameter on a line of its own. Special quoting of whitespace or symbols is not required.

For example to set the classpath, set one line to `-classpath` and the following line to the desired value of the classpath. You don't need to concern yourself with QF-Test's jar files.

By default, empty parameters will be ignored. In case you explicitly want to pass an empty command line argument (i.e. "), you can deactivate the option Ignore empty arguments when starting a client [460].

See section 2.2.5 [17] about how to work with the tables.

Variable: Yes

Restrictions: None

Class arguments

These are the arguments that are passed to the `main(String[])` method of the class to be started. Again, each argument requires a row of its own and special quoting is not required.

By default, empty arguments will be ignored. In case you explicitly want to pass an empty command line argument (i.e. "), you can deactivate the option Ignore empty arguments when starting a client [(460)].

See section 2.2.5[(17)] about how to work with the tables.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.7.2 Start SUT client

 If your application is normally started through a shell script or a special executable starter, the most convenient way to run it from QF-Test is through this kind of node. Depending on how the application is actually started by this script, some modifications may be required. A typical symptom is that the SUT starts up fine, but the connection to QF-Test is not established. In that case, read chapter 44[(889)].

Contained in: All kinds of sequences[(518)].

Children: None

Execution: The command line for the program is built from the attributes and the process is started. Its input and output are redirected to QF-Test.

Attributes:

Figure 40.39: Start SUT client attributes

Client

> This is the identifier for the SUT client. It must remain unique as long as the process is alive. Other nodes will refer to the client by this name.
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty

Executable

> The program to run. If the executable file is not located in a directory on the `PATH` you must give the full path name.
>
> The "Select file" button ⎙ brings up a dialog in which you can select the program file interactively. You can also get to this dialog by pressing ⎡Shift-Return⎤ or ⎡Alt-Return⎤, when the focus is in the text field.
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty

Directory

Here you can set the working directory for the program. If you leave this value empty, the program will inherit QF-Test's working directory.

The "Select directory" button ⊟ brings up a dialog in which you can select the directory interactively. You can also get to this dialog by pressing Shift-Return or Alt-Return, when the focus is in the text field.

This directory will become the current working directory **after** executing the program. As a result, a script named, say, `./copy_data` will be looked up relative to QF-Test's working directory and not the directory given. Only the path names referred to in the script itself will be resolved relative to the new directory.

Variable: Yes

Restrictions: Must be empty or an existing directory

Executable parameters

The command line arguments for the executable. Put each parameter on a line of its own. Special quoting of whitespace or symbols is not required.

By default, empty parameters will be ignored. In case you explicitly want to pass an empty command line argument (i.e. "), you can deactivate the option Ignore empty arguments when starting a client [(460)].

See section 2.2.5[(17)] about how to work with the tables.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the [button] button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.7.3 Start process

To run an arbitrary program during a test, you can either use this node or an Execute shell command[652] node. This node is preferable if you need to pass possibly complex arguments to the executable.

Contained in: All kinds of sequences[518].

Children: None

Execution: The command line for the program is built from the attributes and the process is started. Its input and output are redirected to QF-Test.

Attributes:

Figure 40.40: Start process attributes

Client

> This is the identifier for the SUT client. It must remain unique as long as the process is alive. Other nodes will refer to the client by this name.
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty

Executable

> The program to run. If the executable file is not located in a directory on the `PATH` you must give the full path name.
>
> The "Select file" button ⬜ brings up a dialog in which you can select the program file interactively. You can also get to this dialog by pressing $\boxed{\text{Shift-Return}}$ or $\boxed{\text{Alt-Return}}$, when the focus is in the text field.
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty

Directory

> Here you can set the working directory for the program here. If you leave this value empty, the program will inherit QF-Test's working directory.
>
> The "Select directory" button 🗁 brings up a dialog in which you can select the directory interactively. You can also get to this dialog by pressing $\boxed{\text{Shift-Return}}$ or $\boxed{\text{Alt-Return}}$, when the focus is in the text field.

Note

> This directory will become the working directory of the newly started process. It does not affect the working directory of QF-Test. As a result, a program named, say, `./startserver` will be looked up relative to QF-Test's working directory and not the directory given. Only the path names referred to in the program itself will be resolved relative to the given directory.
>
> **Variable:** Yes
>
> **Restrictions:** Must be empty or an existing directory

Executable parameters

> The command line arguments for the executable. Put each parameter on a line of its own. Special quoting of whitespace or symbols is not required.
>
> By default, empty parameters will be ignored. In case you explicitly want to pass an empty command line argument (i.e. "), you can deactivate the option Ignore empty arguments when starting a client [460].
>
> See section 2.2.5[17] about how to work with the tables.
>
> **Variable:** Yes
>
> **Restrictions:** None

QF-Test ID

> At the moment the QF-Test ID attribute has no meaning for this type of node.
>
> **Variable:** No
>
> **Restrictions:** Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

> These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.
>
> **Variable:** Yes
>
> **Restrictions:** Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ✏ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.7.4 Start web engine

This node is used to start a web browser specifically for web testing. If the process for the SUT is already running, this node can also be used to open an additional browser window in the same process.

Contained in: All kinds of sequences[518].

Children: None

Execution: The command line for the browser program is built from the attributes and the process is started. Its input and output are redirected to QF-Test.

Attributes:

Figure 40.41: Start web engine attributes

Client

This is the identifier for the SUT client. It must remain unique as long as the process is alive. Other nodes will refer to the client by this name.

Variable: Yes

Restrictions: Must not be empty

Browser type

The kind of browser to start. Officially supported values are currently "firefox" (or "mozilla") for Mozilla Firefox, "chrome" for Google Chrome, "edge" for Microsoft

Edge, "opera" for Opera, "safari" for Apple Safari, "headless-firefox" for Mozilla Firefox without visible window, "headless-chrome" (or "headless") for Google Chrome without visible window, and "headless-edge" for Microsoft Edge without visible window.

Variable: Yes

Restrictions: Legal values are "ie", "firefox" (or "mozilla"), "chrome", "edge", "msedge", "opera", "safari", "headless-firefox", "headless-chrome" (or "headless")", and "headless-edge".

Directory of browser installation

For most browsers this field can be left empty. For Firefox with QF-Driver connection mode the installation directory of the browser must be specified here. When connecting via CDP-Driver or WebDriver, the default installation of the given browser is used. If several versions of that browser can be installed in parallel, a specific version may be chosen be specifying its installation directory here.

Variable: Yes

Restrictions: Must be empty or an existing directory.

Browser connection mode

QF-Test can connect to a browser in three different ways: By embedding the browser in a Java VM (the traditional QF-Driver mode), directly via Chrome DevTools Protocol or via Selenium WebDriver. Further information about connection modes and those support for browsers can be found in <u>section 49.3</u>[993]. This attribute determines what to do in cases, when more than one connection mode is available for a browser. Possible values are:

Prefer QF-Driver
Prefer QF-Driver mode if possible, otherwise use CDP-Driver or WebDriver. This is the standard, used also if this attribute is left empty.

QF-Driver only
Exclusively use QF-Driver mode.

Prefer CDP-Driver
Prefer CDP-Driver mode if possible, otherwise use QF-Driver or WebDriver.

CDP-Driver only
Exclusively use CDP-Driver mode.

Prefer WebDriver
Prefer WebDriver mode if possible, otherwise use QF-Driver or CDP-Driver.

WebDriver only
Exclusively use WebDriver mode.

Variable: Yes

Restrictions: Must be empty or one of the values 'Prefer QF-Driver', 'QF-Driver only', 'Prefer CDP-Driver' or 'CDP-Driver only', 'Prefer WebDriver' or 'WebDriver only'

Executable

The Java program to start the browser with, typically `java` or `javaw` on Windows. If you want to run a specific Java virtual machine you should give the full path name of the executable.

The default value is ${qftest:java} which is the Java program that QF-Test itself runs on. When using QF-Driver (only Windows/Linux) please ensure that the Java program used here is of the same 32bit or 64bit variant as the Firefox browser specified in the Directory of browser installation attribute.

The "Select file" button ⬜ brings up a dialog in which you can select the program file interactively. You can also get to this dialog by pressing [Shift-Return] or [Alt-Return], when the focus is in the text field.

Variable: Yes

Restrictions: Must not be empty

Executable parameters

Special command line arguments for the java executable in which the browser is embedded. Can be used to specify the available memory (default is 200MB), define system properties, enable debugging, etc. Put each parameter on a line of its own. Special quoting of whitespace or symbols is not required.

By default, empty parameters will be ignored. In case you explicitly want to pass an empty command line argument (i.e. "), you can deactivate the option Ignore empty arguments when starting a client [460].

See section 2.2.5 [17] about how to work with the tables.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ✏ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.7.5 Open browser window

This node opens a particular web page in a new browser window in a running web engine process. You should start that process via a Start web engine node.

Contained in: All kinds of sequences[518].

Children: None

Execution: Open a given web-page in the already running web engine process.

Attributes:

Figure 40.42: Open browser window attributes

Client

This is the identifier for the SUT client. It must remain unique as long as the process is alive. Other nodes will refer to the client by this name.

Variable: Yes

Restrictions: Must not be empty

URL

The URL of the web page to be shown in the browser. As a special case the URL "about:nowindow" can be used to start the SUT process without opening an initial browser window. In that state, cache or preference settings for the browser can be modified which may be a prerequisite for opening a given URL. The browser window with the actual target URL can then be opened via another Open browser window node.

Variable: Yes

Restrictions: Must not be empty

Name of the browser window

This attribute can be ignored unless you need to test a web application with multiple open browser windows holding similar documents. In that case the Name of the browser window attribute can be used to identify the browser window. Here you can specify the name for the browser window to be opened. You find a brief description how to handle multiple browser windows in FAQ 25.

Variable: Yes

Restrictions: None

Geometry of browser window

These optional attributes for the X/Y coordinate, width and height can be used to specify the geometry of the browser window to open.

Variable: Yes

Restrictions: Width and height must not be negative.

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.7.6 Execute shell command

This node is a convenient way to execute shell commands during a test. The shell that will execute the command can be specified with the command line arguments `-shell <executable>`[(879)] and `-shellarg <argument>`[(879)] for QF-Test. The default for Unix is `/bin/sh`, on Windows either `command.com` or `cmd.exe` is used, the value of the `COMSPEC` environment variable.

When the shell is started it is treated like every other process started by QF-Test, so you can kill it or wait for it to terminate and check its exit code.

Contained in: All kinds of sequences[(518)].

Children: None

Execution: A shell is started to execute the command. Its input and output are redirected to QF-Test.

Attributes:

Figure 40.43: Execute shell command attributes

Client

This is the identifier for the SUT client. It must remain unique as long as the process is alive. Other nodes will refer to the client by this name.

Variable: Yes

Restrictions: Must not be empty

Shell command

The command to execute in the shell. Enter this just as you would at the command prompt.

Windows On Windows systems, quoting of arguments with blanks can be a little tricky. If you're using the standard Windows shell, simply use double quotes as always, for example `dir "C:\Program Files"`. If you're using a Unix shell on Windows by specifying the `-shell <executable>`[(879)] command line argument, use single quotes instead, i.e. `ls 'C:/Program Files'`.

Variable: Yes

Restrictions: Must not be empty

Directory

Here you can set the working directory for the shell. If you leave this value empty, the shell will inherit QF-Test's working directory.

The "Select directory" button ⊟ brings up a dialog in which you can select the directory interactively. You can also get to this dialog by pressing (Shift-Return) or (Alt-Return), when the focus is in the text field.

Variable: Yes

Restrictions: Must be empty or an existing directory

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note   For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing (Alt-Return) or by clicking the 🖊 button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.7.7   Start PDF client

Use this node to load the PDF document to be tested into a viewer, which QF-Test will start as a client process. Please refer to chapter 17[239] for information about the testing of PDF documents.

Contained in: All kinds of sequences[518].

Children: None

Execution: The viewer is started and the PDF document loaded.

Attributes:

Figure 40.44: Start PDF client attributes

Client

This is the identifier for the SUT client. It must remain unique as long as the process is alive. Other nodes will refer to the client by this name.

Variable: Yes

Restrictions: Must not be empty

PDF document

The PDF document to be opened. If a relative path is specified it is resolved relative to the directory containing the current test suite.

The "Select file" button ⬚ brings up a dialog in which you can select the program file interactively. You can also get to this dialog by pressing ⌈Shift-Return⌉ or ⌈Alt-Return⌉, when the focus is in the text field.

Variable: Yes

Restrictions: Must not be empty

Page of PDF document

The page to show.

An integer number is interpreted as page number. A string in quotation marks is interpreted as page name.
Example:
5 opens page number 5
"Index IV" opens the page with the name Index IV

Variable: Yes

Restrictions: Must be a integer number or valid page name in quotation marks.

Password

If the document requires a password, it can be specified here.

Note
When using a password it may be desirable to avoid having the password show up as plain text in the test suite or a run log. To that end the password can be encrypted by inserting the plain-text password, right-clicking and selecting ⌈Crypt password⌉ from the popup menu. Please be sure to specify a password salt before encrypting via the option Salt for crypting passwords[(458)].

Variable: Yes

Restrictions: Empty or a valid password, if required only.

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.7.8 Start windows application

Launch and attach to a native Windows application. Please refer to chapter 15[207] for further information about Windows testing.

Attributes:

Figure 40.45: Start windows application attributes

Client

> This is the identifier for the SUT client. It must remain unique as long as the process is alive. Other nodes will refer to the client by this name.
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty

Windows application

The Windows executable to be launched.

Variable: Yes

Restrictions: Must not be empty

Directory

Here you can set the working directory for the program. If you leave this value empty, the program will inherit QF-Test's working directory.

The "Select directory" button ⊟ brings up a dialog in which you can select the directory interactively. You can also get to this dialog by pressing Shift-Return or Alt-Return, when the focus is in the text field.

Variable: Yes

Restrictions: Must be empty or an existing directory

Window title

The title of the window to attach to.

You can select ‖Escape text for regular expressions‖ from the context menu for escaping special characters of regular expressions of that text.

Variable: Yes

Restrictions: Valid regexp if required.

As regexp

If this attribute is set, the title is interpreted as a regexp (see section 47.3[(909)]) instead of a plain string.

Variable: Yes

Restrictions: None

Timeout

Time limit in milliseconds.

Variable: Yes

Restrictions: >= 0

Executable parameters

There is one tab for each of the following parameter lists:

The command line arguments for the Windows executable. Put each parameter on a line of its own. Special quoting of whitespace or symbols is not required. Parameters that start with "-qfengine:" are directly forwarded to the QF-Test win-engine process.

By default, empty parameters will be ignored. In case you explicitly want to pass an empty command line argument (i.e. "), you can deactivate the option Ignore empty arguments when starting a client [(460)].

See section 2.2.5[(17)] about how to work with the tables.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ✎ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.7.9   Attach to windows application

Attach to a native Windows application. Please refer to chapter 15[207] for further information about Windows testing.

Attributes:

Attach to Windows application

Client
$(client)

Window title
Calculator

$ ☐ As regexp

Timeout
15000

QF-Test ID

Delay before (ms)   Delay after (ms)

✏ Comment

Figure 40.46: Attach to windows application attributes

Client

This is the identifier for the SUT client. It must remain unique as long as the process is alive. Other nodes will refer to the client by this name.

Variable: Yes

Restrictions: Must not be empty

Window title

The title of the window to attach to. Via `-pid <process ID>` you can also specify the ID of the process. Via `-class <class name>`, the UI Automation class name of a window can be defined.

You can select Escape text for regular expressions from the context menu for escaping special characters of regular expressions of that text.

Variable: Yes

Restrictions: Must not be empty. Valid regexp if required.

As regexp

If this attribute is set, the title is interpreted as a regexp (see section 47.3[(909)]) instead of a plain string.

Variable: Yes

Restrictions: None

Timeout

Time limit in milliseconds.

Variable: Yes

Restrictions: >= 0

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note
For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the ✎ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.7.10 Launch Android emulator

Use the node to start an Android emulator and connect to it. Quickstart your application[28] helps you to set up a sequence for launching the emulator and starting an app along with the appropriate waiter nodes.

Please have a look at chapter 16[217] for detailed information on testing of Android apps.

Attributes:

Figure 40.47: Launch Android emulator attributes

Client

This is the identifier for the SUT client. It must remain unique as long as the process is alive. Other nodes will refer to the client by this name.

Variable: Yes

Restrictions: Must not be empty

Name of the Android emulator to use

The name of the emulator to be used, as shown, for example, in the Android Virtual Device Manager in the column 'Name' (see Android studio screen showing available AVDs[(225)]). Using the variable `deviceName` you do not need to hardcode it here.

In case you want to use an emulator created after you started QF-Test it may not be showing in the drop down list. Then, just type the name in the text field.

Variable: Yes

Restrictions: Must not be empty

Executable parameters

QF-Test communicates with the emulator via a Java program started additionally to the emulator, running in the background. Here, you can specify parameters for that Java program. Put each parameter on a line of its own. Special quoting of whitespace or symbols is not required. Sample: `-Xmx1G` to assign 1 GM memory to the Java program.

By default, empty parameters will be ignored. In case you explicitly want to pass an empty command line argument (i.e. ''), you can deactivate the option Ignore empty arguments when starting a client [(460)].

See section 2.2.5[(17)] about how to work with the tables.

Variable: Yes

Restrictions: None

Emulator arguments

The command line arguments for the emulator to be launched, for example `-no-snapshot-save`. Put each parameter on a line of its own. Special quoting of whitespace or symbols is not required.

By default, empty arguments will be ignored. In case you explicitly want to pass an empty command line argument (i.e. ''), you can deactivate the option Ignore empty arguments when starting a client [(460)].

See section 2.2.5[(17)] about how to work with the tables.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

> These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

> **Variable:** Yes

> **Restrictions:** Valid number >= 0

Comment

> Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note

> For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ✎ button.

> You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

> If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

> **Variable:** Yes

> **Restrictions:** None

40.7.11   Connect to Android device

> The node connects to a running Android emulator or a real device. Quickstart your application[(28)] helps you to set up a sequence for connecting to an Android device, either emulator or real, and starting an app along with the appropriate waiter nodes.

Please have a look at chapter 16[(217)] for detailed information on testing of Android apps.

Attributes:

Figure 40.48: Connect to Android device Attribute

Client

> This is the identifier for the SUT client. It must remain unique as long as the process is alive. Other nodes will refer to the client by this name.
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty

Name of the Android device to use

> The name of a real Android device attached to the computer or the emulator. The available names are shown in the drop down list. Alternatively, you will find them via the menu item ⟦Running Android Devices⟧ in the menu ⟦Extras⟧.
>
> You do not need to hardcode the name if you use the variable `deviceName` instead.

Note
> In case you want to use a real Android device attached to the computer after you started QF-Test it may not be showing in the drop down list. Then, just type the name in the text field. The same applies to an emulator you created after you started QF-Test.
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty

Executable parameters

> QF-Test communicates with the real device, respectively the emulator, via a Java program started additionally to the emulator, running in the background. Here, you can specify parameters for that Java program. Put each parameter on a line of its own. Special quoting of whitespace or symbols is not required. Sample: `-Xmx1G` to assign 1 GM memory to the Java program.
>
> By default, empty parameters will be ignored. In case you explicitly want to pass an empty command line argument (i.e. "), you can deactivate the option Ignore empty arguments when starting a client [460].
>
> See section 2.2.5 [17] about how to work with the tables.
>
> **Variable:** Yes
>
> **Restrictions:** None

QF-Test ID

> At the moment the QF-Test ID attribute has no meaning for this type of node.
>
> **Variable:** No
>
> **Restrictions:** Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

> These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay [476] from the global options is used.
>
> **Variable:** Yes
>
> **Restrictions:** Valid number >= 0

Comment

> Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.
>
> **Note**   For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command [431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the 🖉 button.
>
> You can trigger special behaviors of some nodes using doctags, please see Doctags [1215].
>
> If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.
>
> **Variable:** Yes

Restrictions: None

40.7.12 Launch Android app

Use the node to install and start an Android app. Quickstart your application[28] helps you to set up a sequence for launching the emulator - or attaching to a real Android device or running emulator - and starting an app along with the appropriate waiter nodes.

Please have a look at chapter 16[217] for detailed information on testing of Android apps.

Attributes:

Figure 40.49: Launch Android app attributes

Client

This is the identifier for the SUT client. It must remain unique as long as the process is alive. Other nodes will refer to the client by this name.

Variable: Yes

Restrictions: Must not be empty

APK file path

The path of the Android app to be used. If you want the specify the path relative to the test suite use the variable `${qftest:suite.dir}` (cf. <u>section 6.5</u>[(108)]). Sample: `${qftest:suite.dir}/apps/myapp.apk`.

In case the app has been preinstalled and the .apk file is not available you can start the app via the attributes Package und Activity.

Variable: Yes

Restrictions: Must not be empty

Force app reinstallation

Select the attribute to install the app in any case, irrespective of an existing installation of the app on the Android emulator or real device.

Note Whether the reinstallation will delete existing settings and data of the app depends on the place where they have been saved.

Variable: No

Restrictions: None

Launch app

Select the attribute to start the app.

Variable: No

Restrictions: None

Package

The package name of the Android app to be launched. Only required in case the .apk file cannot be specified with the attribute APK file path. You also need to specify the attribute Activity. You can either use the Android Debug Bridge to get the package name or launch or attach to an Android emulator or real device, start the app manually and record a component. The package name will then be recorded in the top level node of the component hierarchy.

Variable: Yes

Restrictions: None

Activity

> The activity name of the Android app to be launched. Only required in case the .apk file cannot be specified with the attribute APK file path. You also need to specify the attribute Package. You can use the Android Debug Bridge to get the activity name.
>
> **Variable:** Yes
>
> **Restrictions:** No

QF-Test ID

> At the moment the QF-Test ID attribute has no meaning for this type of node.
>
> **Variable:** No
>
> **Restrictions:** Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

> These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽⁴⁷⁶⁾ from the global options is used.
>
> **Variable:** Yes
>
> **Restrictions:** Valid number >= 0

Comment

> Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note

> For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽⁴³¹⁾ lets you define an external editor in which comments can be edited conveniently by pressing [Alt-Return] or by clicking the ✎ button.
>
> You can trigger special behaviors of some nodes using doctags, please see Doctags⁽¹²¹⁵⁾.
>
> If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.
>
> **Variable:** Yes
>
> **Restrictions:** None

40.7.13 Wait for client to connect

This node is used to make sure that the connection to a client is established. If after a configurable timeout the client is still not connected, a `ClientNotConnectedException`[(855)] is thrown. You can also use the Variable for result attribute to store the result into a variable and the Throw exception on failure attribute to suppress the exception.

Contained in: All kinds of sequences[(518)].

Children: None

Execution: QF-Test waits until either the connection to the Java client is established or the timeout is up.

Attributes:

Wait for client to connect

Client

SUT

Timeout

15000

GUI engine

Result handling

Variable for result

☐ Local variable

Error level of message

Error ⌄

$ ☑ Throw exception on failure

QF-Test ID

Delay before (ms)   Delay after (ms)

✏ Comment

Wait until the connection between qftest
and the SUT is established.

Figure 40.50: Wait for client to connect attributes

Client

> The name of the Java client to wait for.
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty

Timeout

> Time limit in milliseconds.
>
> **Variable:** Yes
>
> **Restrictions:** >= 0

GUI engine

> The GUI engine to wait for. Only relevant for SUTs with more than one GUI engine as described in chapter 43[(887)].
>
> **Variable:** Yes
>
> **Restrictions:** See chapter 43[(887)]

Variable for result

> This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

Note

> If this attribute is set, the attribute Error level of message is ignored and no error is reported. The attribute Throw exception on failure always remains effective, so it is possible to set a result variable and still throw an exception.
>
> **Variable:** Yes
>
> **Restrictions:** None

Local variable

> This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure[(586)], Dependency[(548)] or Test case[(518)] node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test case node or, if there is no such node in the topmost node on the variables stack, falling back to the global bindings if necessary. See chapter 6[(103)] for a detailed explanation of variable binding and lookup.
>
> In order to predefine the option use Enable 'Local variable' attribute by default[(513)].
>
> **Variable:** No
>
> **Restrictions:** None

Error level of message

> This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

> If the attribute Throw exception on failure is set, this attribute is irrelevant and if Variable for result is set this attribute is ignored.

> **Variable:** No

> **Restrictions:** None

Throw exception on failure

> Throw an exception in case of failure. For 'Check...' nodes a CheckFailedException[(854)] is thrown, for 'Wait for...' nodes the respective specific exception.

> **Variable:** No

> **Restrictions:** None

QF-Test ID

> At the moment the QF-Test ID attribute has no meaning for this type of node.

> **Variable:** No

> **Restrictions:** Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

> These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

> **Variable:** Yes

> **Restrictions:** Valid number >= 0

Comment

> Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

> For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.

> You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.7.14 Wait for Android device

This node is used to make sure that the connection to a virtual or real Android device has been established. If after a configurable timeout, the Android device still has not been connected a `ClientNotConnectedException`[(855)] is thrown. If QF-Test detects before the end of the timeout that a connection cannot be established it will throw a `ConnectionFailureException`[(856)].

Contained in: All kinds of sequences[(518)].

Children: None

Execution: QF-Test waits until either the connection to the emulator or the real device has been established or the timeout is up.

Attributes:

Wait for Android device
Client
ANDROID
Timeout
60000
QF-Test ID
Delay before (ms)   Delay after (ms)
✎ Comment

Figure 40.51: Wait for Android device Attributes

Client

The name of the Android emulator or real device to wait for.

Variable: Yes

Restrictions: Must not be empty

Timeout

Time limit in milliseconds.

Variable: Yes

Restrictions: >= 0

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note   For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing [Alt-Return] or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.7.15   Stop client

This node forcibly terminates a client process started by QF-Test. If the process has already terminated, nothing is changed. Otherwise, if the client is an SUT client, QF-Test first tries to call the Java method `System.exit(-1)` in the SUT to achieve a clean shutdown. If the client is not an SUT client or the process is still alive after the `exit` call, the process is killed and its exit code is set to -1.

Windows **Note:** On Windows child processes started by a process are not terminated when the parent process is killed. As explained in chapter 44[(889)], the `java` program for the SUT is not started directly by QF-Test, but through an intermediate program. This means that when the `System.exit(-1)` call fails to terminate the SUT for some reason, the process for the SUT's Java VM will be left hanging around.

Contained in: All kinds of sequences[(518)].

Children: None

Execution: Terminates the last process that was started under the given name.

Attributes:

Figure 40.52: Stop client attributes

Client

The client name of the process that is to be killed.

Variable: Yes

Restrictions: Must not be empty

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing [Alt-Return] or by clicking the ✎ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.7.16 Wait for process to terminate

This node waits until a process that was started by QF-Test terminates. If a given time limit is exceeded and the process is still alive. Per default a `ClientNotTerminatedException`[856] is thrown. Otherwise the client's exit code is read and tested against a given value. You can also use the Variable for result attribute to store the result into a variable and the Throw exception on failure attribute to suppress the exception.

Contained in: All kinds of sequences[518].

Children: None

Execution: Waits for the end of a process and checks its exit value unless the timeout is exceeded.

Attributes:

Wait for process to terminate

Client

SUT

Timeout

3000

Expected exit code

0

Result handling

Variable for result

☐ Local variable

Error level of message

Error

$ ☑ Throw exception on failure

QF-Test ID

Delay before (ms)   Delay after (ms)

✎ Comment

Wait for the SUT to exit and check its exit value

Figure 40.53: Wait for process to terminate attributes

Client

The client name of the process to wait for.

Variable: Yes

Restrictions: Must not be empty

Timeout

Time limit in milliseconds.

Variable: Yes

Restrictions: >= 0

Expected exit code

If this attribute is set it is used to validate the exit code of the process. The type of the comparison is defined by prepending one of the four operators **==**, **!=**, **<** and **>**. Just a number without preceding operation test for equality. If the test fails, an UnexpectedExitCodeException[856] is thrown. The exception won't be thrown, if the attribute Throw exception on failure is not set.

Examples: If this attribute is set to **0**, every exit code that is not 0 causes an exception. This is the same as for **==0**. A value of **>0** causes an exception for every exit code equal to or less than 0. This exception can be suppressed by un-checking the attribute Throw exception on failure.

Variable: Yes

Restrictions: See above

Variable for result

This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

Note   If this attribute is set, the attribute Error level of message is ignored and no error is reported. The attribute Throw exception on failure always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure[586], Dependency[548] or Test case[518] node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test case node or, if there is no such node in the topmost node on the variables stack, falling back to the global bindings if necessary. See chapter 6[103] for a detailed explanation of variable binding and lookup.

In order to predefine the option use Enable 'Local variable' attribute by default[513].

Variable: No

Restrictions: None

Error level of message

> This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

> If the attribute Throw exception on failure is set, this attribute is irrelevant and if Variable for result is set this attribute is ignored.

> **Variable:** No

> **Restrictions:** None

Throw exception on failure

> Throw an exception in case of failure. For 'Check...' nodes a CheckFailedException[854] is thrown, for 'Wait for...' nodes the respective specific exception.

> **Variable:** No

> **Restrictions:** None

QF-Test ID

> At the moment the QF-Test ID attribute has no meaning for this type of node.

> **Variable:** No

> **Restrictions:** Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

> These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.

> **Variable:** Yes

> **Restrictions:** Valid number >= 0

Comment

> Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

> For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.

> You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.8 Events

This section lists all kinds of nodes that trigger actions in the SUT. Besides true Java events these include pseudo events with special behavior.

40.8.1 Mouse event

Mouse events simulate mouse movement and clicks as well as drag and drop operations.

Contained in: All kinds of sequences[(518)].

Children: None

Execution: The coordinates and other attributes of the event are sent to the SUT together with the data about the target component. The TestEventQueue determines the corresponding component in the SUT, adjusts the coordinates and triggers the resulting event.

Attributes:

Figure 40.54: Mouse event attributes

Client

The name of the SUT client process to which the event is sent.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID[(813)] of the Window[(812)], Component[(823)] or Item[(829)] node that is the target of the event.

The "Select component" button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing Shift-Return

or ⟨Alt-Return⟩, when the focus is in the text field. As an alternative you can copy the target node with ⟨Ctrl-C⟩ or ⟨Edit→Copy⟩ and insert its QF-Test component ID into the text field by pressing ⟨Ctrl-V⟩.

This attribute supports a special format for referencing components in other test suites (see section 24.1[(298)]). Furthermore, sub-elements of nodes can be addressed directly without requiring separate nodes for them (see section 5.9[(82)]). When using SmartIDs, you can address a GUI element directly via its recognition criteria. For more information, refer to SmartID[(73)] and Component nodes versus SmartID[(46)].

Variable: Yes

Restrictions: Must not be empty.

Event

This *ComboBox* lets you choose the type of the event. MOUSE_MOVED, MOUSE_PRESSED, MOUSE_RELEASED, MOUSE_CLICKED and MOUSE_DRAGGED are the standard event IDs of the Java class MouseEvent.

The abstract 'Mouse click' event is a compound of the events MOUSE_MOVED, MOUSE_PRESSED, MOUSE_RELEASED and MOUSE_CLICKED. During replay the pseudo event is simulated as four separate events. This adds to the clarity of a test suite and simplifies editing.

4.1.3+ The special 'Double click' event comprises all the individual events required to simulate a complete double click.

The events MOUSE_DRAG_FROM, MOUSE_DRAG_OVER and MOUSE_DROP_TO are used to simulate Drag&Drop in the SUT. See section 47.1[(908)] for details.

Variable: No

Restrictions: None

X/Y

These are the coordinates of the *MouseEvent*. They are relative to the upper left corner of the Window[(812)], Component[(823)] or Item[(829)] that is the target of the event. They can be negative, e.g. to simulate a click on the expansion toggle of a node in a JTree.

Most of the time the exact coordinates for a mouse don't really matter, any place within the target will do. In this case you should leave the X and Y values empty to tell QF-Test to aim at the center of the target. Where possible QF-Test will leave the values empty when recording, provided the option Record MouseEvents without coordinates where possible [(441)] is active.

Variable: Yes

Restrictions: Valid number or empty

Modifiers

This value reflects the state of the mouse buttons and the modifier keys $\boxed{\text{Shift}}$, $\boxed{\text{Control}}$, $\boxed{\text{Alt}}$ and $\boxed{\text{Meta}}$ during a mouse or key event. States are combined by adding up their values.

Value	Key/Button
1	Shift
2	Control
4	Meta or right mouse button (Longclick for Android)
8	Alt or middle mouse button
16	Left mouse button

Table 40.16: Modifier values

Variable: Yes

Restrictions: Valid number

Click count

This value lets a Java program distinguish between a single and a double (or even multiple) click.

Variable: Yes

Restrictions: Valid number

Popup trigger

If this attribute is set, the event can trigger a *PopupMenu*. This is Java's somewhat peculiar way of supporting different conventions for triggering *PopupMenus* on different platforms.

Variable: Yes

Restrictions: None

Replay as "hard" event

If this attribute is set the event is replayed as a hard event, meaning it is triggered as a real system event that moves the mouse around and not just inserted as soft event into the event queue. Soft events are are typically better because they avoid impact of concurrent user mouse actions and are less likely to break due to interference from an overlapping window. Nevertheless there are certain special situations where hard events are helpful.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.8.2 Key event

Key events simulate keyboard input in the SUT. These are used mainly for control and function keys. Input of text is better represented as a Text input[689].

The special keyboard event `InputMethodEvent` supports international character codes. QF-Test doesn't support `InputMethodEvents` directly. Instead, it converts events of type `INPUT_METHOD_TEXT_CHANGED` into Key events of type `KEY_TYPED`.

Contained in: All kinds of sequences[518].

Children: None

Execution: The key codes of the event are sent to the SUT together with the data about the target component. The `TestEventQueue` determines the corresponding component in the SUT and triggers the resulting event.

Attributes:

Figure 40.55: Key event attributes

Client

    The name of the SUT client process to which the event is sent.

    Variable: Yes

    Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID[(813)] of the Window[(812)], Component[(823)] or Item[(829)] node that is the target of the event.

The "Select component" button ⬚ brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing [Shift-Return] or [Alt-Return], when the focus is in the text field. As an alternative you can copy the target node with [Ctrl-C] or [Edit→Copy] and insert its QF-Test component ID into the text field by pressing [Ctrl-V].

This attribute supports a special format for referencing components in other test suites (see section 24.1[(298)]). Furthermore, sub-elements of nodes can be addressed directly without requiring separate nodes for them (see section 5.9[(82)]). When using SmartIDs, you can address a GUI element directly via its recognition criteria. For more information, refer to SmartID[(73)] and Component nodes versus SmartID[(46)].

Variable: Yes

Restrictions: Must not be empty.

Event

This *ComboBox* lets you choose the type of the event. `KEY_PRESSED`, `KEY_TYPED` and `KEY_RELEASED` are the standard event IDs of the Java class `KeyEvent`.

The 'Keystroke' pseudo event is a compound of the events `KEY_PRESSED`, `KEY_TYPED` and `KEY_RELEASED`. During replay the pseudo event is simulated as two or three separate events, depending on whether it is a printable character key, or a control or function key. In the latter case, no `KEY_TYPED` event is generated.

Variable: No

Restrictions: None

Key

This is a convenience method to set Key code[(687)], Key char[(688)] and Modifiers[(688)] directly by pressing the corresponding key while this component has the keyboard focus. For `KEY_TYPED` events Key char[(688)] is set to 0.

Unfortunately you can't select the [Tab] key this way since it is used for keyboard traversal. Key code[(687)] and Key char[(688)] for the [Tab] key are both 9.

Variable: No

Restrictions: None

Key code

This is a Java specific code for the key, the `keyCode` member of the Java class `KeyEvent`.

Variable: Yes

Restrictions: Valid number

Key char

This is the `keyChar` member of the Java class `KeyEvent`. Its value is the character generated created by the last key press, taking the state of the ⌑Shift⌑ key into account. Control and function keys always have a Key char value of 65535.

Variable: Yes

Restrictions: Valid number

Modifiers

This value reflects the state of the mouse buttons and the modifier keys ⌑Shift⌑, ⌑Control⌑, ⌑Alt⌑ and ⌑Meta⌑ during a mouse or key event. States are combined by adding up their values.

Value	Key/Button
1	Shift
2	Control
4	Meta or right mouse button (Longclick for Android)
8	Alt or middle mouse button
16	Left mouse button

Table 40.17: Modifier values

Variable: Yes

Restrictions: Valid number

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the 🖉 button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.8.3 Text input

This is a pseudo event for simulating text input from the keyboard. A single Text input node replaces a whole sequence of Key event[685] nodes. To achieve this, QF-Test takes advantage of the fact that AWT and Swing text fields listen only to `KEY_TYPED` events, so input of text can be simulated by triggering one `KEY_TYPED` event per text character. If your SUT uses custom text components that actually require the `KEY_PRESSED` and `KEY_RELEASED` events, you cannot use this node type but have to resort to plain Key events[685].

Contained in: All kinds of sequences[518].

Children: None

Execution: The text is sent to the SUT together with the data about the target component. The `TestEventQueue` determines the corresponding component in the SUT and triggers the resulting events.

Attributes:

Figure 40.56: Text input attributes

Client

> The name of the SUT client process to which the event is sent.
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty.

QF-Test component ID

> The QF-Test ID$^{(813)}$ of the Window$^{(812)}$, Component$^{(823)}$ or Item$^{(829)}$ node that is the target of the event.
>
> The "Select component" button ⬚ brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing [Shift-Return] or [Alt-Return], when the focus is in the text field. As an alternative you can copy the target node with [Ctrl-C] or [Edit→Copy] and insert its QF-Test component ID into the text field by pressing [Ctrl-V].
>
> This attribute supports a special format for referencing components in other test suites (see section 24.1$^{(298)}$). Furthermore, sub-elements of nodes can be ad-

dressed directly without requiring separate nodes for them (see section 5.9[82]). When using SmartIDs, you can address a GUI element directly via its recognition criteria. For more information, refer to SmartID[73] and Component nodes versus SmartID[46].

Variable: Yes

Restrictions: Must not be empty.

Text

The text that is to be sent to the SUT.

Note

When inserting text into a password field it may be desirable to avoid having the password show up as plain text in the test suite or a run log. To that end the password can be encrypted by inserting the plain-text password, right-clicking and selecting ⟦Crypt password⟧ from the popup menu. Please be sure to specify a password salt before encrypting via the option Salt for crypting passwords[458].

Variable: Yes

Restrictions: No line breaks are possible.

Clear target component first

If this attribute is set, and the target component is a text field or text area, the contents of the target component are removed before the new text is inserted.

Variable: Yes

Restrictions: None

Replay single events

If the target component is a text field or a text area, the text can optionally be inserted by manipulating the component directly through its API. This is much faster, but if KeyListeners are registered on the component they will not be notified of the change. If this attribute is set, key events are simulated for every single character.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

> These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.
>
> **Variable:** Yes
>
> **Restrictions:** Valid number >= 0

Comment

> Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note
> For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the 🖉 button.
>
> You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].
>
> If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.
>
> **Variable:** Yes
>
> **Restrictions:** None

40.8.4 Window event

WindowEvents are of limited use for a test suite since most of them are not generated as a direct consequence of some user interaction. The only exception is the `WINDOW_CLOSING` event that is triggered when the user closes a window. It is also possible to simulate `WINDOW_ICONIFIED` and `WINDOW_DEICONIFIED` events.

Contained in: All kinds of sequences[(518)].

Children: None

Execution: The event is sent to the SUT together with the data about the target window. The `TestEventQueue` determines the corresponding window in the SUT and triggers the resulting event.

Attributes:

Figure 40.57: Window event attributes

Client

The name of the SUT client process to which the event is sent.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID[(813)] of the Window[(812)], Component[(823)] or Item[(829)] node that is the target of the event.

The "Select component" button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing Shift-Return or Alt-Return, when the focus is in the text field. As an alternative you can copy the target node with Ctrl-C or Edit→Copy and insert its QF-Test component ID into the text field by pressing Ctrl-V.

This attribute supports a special format for referencing components in other test suites (see section 24.1[(298)]). Furthermore, sub-elements of nodes can be addressed directly without requiring separate nodes for them (see section 5.9[(82)]). When using SmartIDs, you can address a GUI element directly via its recognition

criteria. For more information, refer to SmartID⁽⁷³⁾ and Component nodes versus SmartID⁽⁴⁶⁾.

Variable: Yes

Restrictions: Must not be empty.

Event

This *ComboBox* lets you choose the type of the event. Possible values are `WINDOW_OPENED`, `WINDOW_CLOSING`, `WINDOW_CLOSED`, `WINDOW_ACTIVATED`, `WINDOW_DEACTIVATED`, `WINDOW_ICONIFIED` and `WINDOW_DEICONIFIED`, the standard event IDs of the Java class `WindowEvent`.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽⁴⁷⁶⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽⁴³¹⁾ lets you define an external editor in which comments can be edited conveniently by pressing ⌗Alt-Return⌗ or by clicking the ▱ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags⁽¹²¹⁵⁾.

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.8.5 Component event

The name Component event for this node may be misleading, since QF-Test filters all of these events except for windows, but since they represent the Java class `ComponentEvent`, it is better to stick to that name. Except for `COMPONENT_MOVED` and `COMPONENT_SIZED` events on a window, which are the result of the user moving or resizing the window interactively, all *ComponentEvents* are artificial and thus ignored. If you replay the event during a Web test not on a Web page[(818)] component, but on an HTML component, the browser window will be moved in a way that size and position of the inner rendering area match the given values.

Contained in: All kinds of sequences[(518)].

Children: None

Execution: The event is sent to the SUT together with the data about the target window. The `TestEventQueue` determines the corresponding window in the SUT and triggers the resulting event.

Attributes:

Figure 40.58: Component event attributes

Client

> The name of the SUT client process to which the event is sent.
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty.

QF-Test component ID

> The QF-Test ID[(813)] of the Window[(812)], Component[(823)] or Item[(829)] node that is the target of the event.
>
> The "Select component" button 🔲 brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing [Shift-Return] or [Alt-Return], when the focus is in the text field. As an alternative you can copy the target node with [Ctrl-C] or [Edit→Copy] and insert its QF-Test component ID into the text field by pressing [Ctrl-V].
>
> This attribute supports a special format for referencing components in other test suites (see section 24.1[(298)]). Furthermore, sub-elements of nodes can be addressed directly without requiring separate nodes for them (see section 5.9[(82)]). When using SmartIDs, you can address a GUI element directly via its recognition criteria. For more information, refer to SmartID[(73)] and Component nodes versus SmartID[(46)].
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty.

Event

> This *ComboBox* lets you choose the type of the event. Possible values are `COMPONENT_SIZED` and `COMPONENT_MOVED`, the standard event IDs of the Java class `ComponentEvent`.
>
> **Variable:** No
>
> **Restrictions:** None

Y/Height

> For a `COMPONENT_MOVED` event set this to the new Y-coordinate of the window, for a `COMPONENT_SIZED` event to its new height.
>
> **Variable:** Yes
>
> **Restrictions:** Valid number, height > 0

X/Width

> For a `COMPONENT_MOVED` event set this to the new X-coordinate of the window, for a `COMPONENT_SIZED` event to its new width.

Variable: Yes

Restrictions: Valid number, width > 0

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note
For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ✎ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.8.6 Selection

A Selection node represents an abstract event like selecting a menu item, choosing an entry in a combo box or selecting something from or closing a system dialog. Currently this event node is used only for SWT, web and Electron SUT clients, where some events cannot be triggered by "soft" mouse events, . The alternative of using "hard" events has some disadvantages as described for the Replay as "hard" event[(684)] attribute of a Mouse event[(681)].

The Detail[(699)] attribute determines the kind of operation to perform, or the value to select, depending on the target component.

Contained in: All kinds of sequences[(518)].

Children: None

Execution: The event is sent to the SUT together with the data about the target component. The component is resolved and an action performed which depends on the type of component as listed in the table above.

Attributes:

Figure 40.59: Selection attributes

Client

>   The name of the SUT client process to which the event is sent.
>
>   **Variable:** Yes
>
>   **Restrictions:** Must not be empty.

QF-Test component ID

>   The QF-Test ID[(813)] of the Window[(812)], Component[(823)] or Item[(829)] node that is the target of the event.

The "Select component" button ⬒ brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing $\boxed{\text{Shift-Return}}$ or $\boxed{\text{Alt-Return}}$, when the focus is in the text field. As an alternative you can copy the target node with $\boxed{\text{Ctrl-C}}$ or $\boxed{\text{Edit}\rightarrow\text{Copy}}$ and insert its QF-Test component ID into the text field by pressing $\boxed{\text{Ctrl-V}}$.

This attribute supports a special format for referencing components in other test suites (see section 24.1[298]). Furthermore, sub-elements of nodes can be addressed directly without requiring separate nodes for them (see section 5.9[82]). When using SmartIDs, you can address a GUI element directly via its recognition criteria. For more information, refer to SmartID[73] and Component nodes versus SmartID[46].

Variable: Yes

Restrictions: Must not be empty.

Detail

The Detail attribute specifies the kind of operation to perform or the value to select, depending on the target component. The possible combinations are listed in detail below.

SWT   The following combinations of widgets and Detail values are currently supported for SWT:

Class	Detail attribute	Action
CCombo sub-item	Empty	Choose item as current value
ColorDialog	Color value in hexadecimal #rrggbb format	Select given color
ColorDialog	CANCEL	Abort color selection
Combo sub-item	Empty	Choose item as current value
CTabFolder sub-item	Empty	Select tab
CTabFolder sub-item	close	Close tab
DirectoryDialog	Directory name	Select given directory
DirectoryDialog	CANCEL	Abort directory selection
FileDialog	File name, including directory	Select given file
FileDialog	CANCEL	Abort file selection
FontDialog	Font description, system specific	Select given font
FontDialog	CANCEL	Abort file selection
Menu	close	Cancel the menu (close without selection)
MenuItem	Empty	Select item
MessageBox	Any of OK YES NO CANCEL ABORT RETRY IGNORE OPEN SAVE	Close with the given value as the user's choice
ProgressBar	ProgressBar value	Set the given value
Scale	Scale value	Set the given value
Slider	Slider value	Set the given value
Spinner	Spinner value	Set the given value
TabFolder sub-item	Empty	Select tab

Table 40.18: Supported SWT widgets for a Selection event

Note   Eclipse/RCP makes heavy use of dynamic CTabFolders where the items can be moved between folders. The items represent the actual business objects whereas the folders are just scaffolding to hold them. Besides, the layout of the folders and items can change drastically when switching perspectives. Thus it is often desirable to be able to select or close an item independent of the CTabFolder it currently resides in. This can be done by using the Procedures[586] qfs.qft#qfs.swt.ctabfolder.selectTab and qfs.qft#qfs.swt.ctabfolder.closeTab, provided in the qfs.qft standard library. Besides the ubiquitous client parameter, the only other parameter tabname must be set to the name of the item to select or close.

Web   The following combinations of DOM nodes and Detail values are currently supported for web SUT clients:

Node type	Detail attribute	Action
Confirmation dialog	Any of `OK YES NO CANCEL RETRY`	Close the dialog with the given value as the user's choice
File dialog for download	The file to save to or `CANCEL`	Close the dialog and save to the specified file or abort the download
Login dialog	Username|Password or `CANCEL`	Close the dialog and login with the specified data or abort. The password can by encrypted by right-clicking the Detail field and selecting Crypt password. Please be sure to specify a password salt before encrypting via the option Salt for crypting passwords[(458)].
Prompt dialog	The text to enter or `CANCEL`	Close the dialog and return the specified text or abort
Top-level DOCUMENT	`back`	Navigate back to the previous page
Top-level DOCUMENT	`forward`	Navigate forward to the next page
Top-level DOCUMENT	`goto:`URL	Navigate to the specified URL
Top-level DOCUMENT	`refresh`	Reload the current page
Top-level DOCUMENT	`stop`	Stop loading the current page
OPTION or SELECT sub-item	`0`	Choose the OPTION as current value
OPTION or SELECT sub-item	`1`	Add the OPTION to the selection
OPTION or SELECT sub-item	`-1`	Remove the OPTION from the selection

Table 40.19: Supported DOM nodes for a Selection event

Electron The following Combinations of DOM nodes and the Detail attribute are currently support for Electron applications (see chapter 19[(252)]), additionally to the ones listed for Web as above.

Node type	**Detail Attribute**	**Action**
Menu	`clickmenu:@/<Menu path>`	Where `<Menu path>` specifies the menu and the sub-menu item(s), seperated by `/`. For example, if you want to trigger the menu item `Save as` in the menu `File` you would enter `clickmenu:@File/Save as` for the attribute. You also need to enter the QF-Test ID of the node `Web page` of the SUT in the attribute QF-Test component ID.
Dialog	`select:@/<Return value>`	Salt for crypting passwords[458] Close a previously opened dialog. For details about possible return values please refer to: Native Dialogs[253] You also need to enter the QF-Test ID of the node `Web page` of the SUT in the attribute QF-Test component ID.
Error dialog	select:1	Closes the open dialog.
Message box	select:2:true	Select the button with id "2" and sets the checkbox value to true.
Save dialog	select:"C:\path\to\my.file"	Closes the save dialog and returns the given path.
Open dialog	select:["C:\path\to\my.file"]	Closes the open dialog and returns the given path.
Open dialog	select:["C:\path\to\my\first.file", "C:\path\to\my\second.file"]	Closes the open dialog and returns the given paths.

Table 40.20: Supported DOM nodes for Electron SUTs in a Selection Event

Android

The following Combinations of DOM nodes and the Detail attribute are currently support for Android applications (see chapter 16[217]):

Node type	Detail attribute	Action
All (will be ignored)	HOME	Click on the Home-Button of the Emulator.
All (will be ignored)	BACK	Click on the Back-Button of the Emulator.
All (will be ignored)	APP_SWITCH	Click on the App-Switch-Button of the Emulator.
All	swipe: <Direction>	Swipe on a component into a certain direction. Possible values are: "right", "→", "go_left", "prevPage", "", "left", "←", "go_right", "nextPage", "", "up", "↑", "go_down", "scrollDown", "", "down", "↓", "go_up", "scrollUp", "", "down_right", "↘", "go_up_left", "", "up_left", "↖", "go_down_right", "", ""down_left", "", "go_up_right", "", "up_right", "", "go_down_left" and "" (all without quotes).
All	swipe: <Start Coordinate X> <Start Coordinate Y> <End Coordinate X> <End Coordinate X>	Swipe on a component whereby the Swipe-start and the swipe-end coordinate are given. Instead of Integer-Numbers it is also possible to specify "W", "E", "S", "N", "C", "CH" or "CW". In this case "W" stands for West, "E" for East, "S" for the south of the component, "N" for the north of the component and "C" for the center of a component. Thus swipe: W C E C will swipe in the middle of the component from west (left) to the east (right) and swipe: C N C S from up to down.

Table 40.21

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ✏ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.8.7 File selection

A File selection is a pseudo event that you only need in a special case.

If your SUT uses the standard AWT file selection dialog, implemented by the class `java.awt.FileDialog`, QF-Test has no chance to record the events the user generates while selecting a file, since they are all handled by the underlying system and never passed on to Java. For the same reason, the selection cannot be simulated as a sequence of mouse and key events. This is not the case with Swing's `javax.swing.JFileChooser` which is implemented as a normal dialog.

Therefore QF-Test just records the result of the file selection in the form of this node and stores that data in the FileDialog upon replay before closing it. For the SUT there is no difference to an actual selection by the user.

Contained in: All kinds of sequences[518].

Children: None

Execution: File and directory are stored in an open `java.awt.FileDialog` and the dialog is closed.If no FileDialog is open, a `ComponentNotFoundException`[850] is thrown.

Attributes:

Figure 40.60: File selection attributes

Client

> The name of the SUT client process to which the event is sent.
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty.

File

> The name of the file (without the directory part) that is to be selected.
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty

Directory

> The directory of the file that is to be selected.
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty

GUI engine

> The GUI engine in which to look for a file selection dialog. Only relevant for SUTs with more than one GUI engine as described in chapter 43[(887)].
>
> **Variable:** Yes
>
> **Restrictions:** See chapter 43[(887)]

QF-Test ID

> At the moment the QF-Test ID attribute has no meaning for this type of node.
>
> **Variable:** No
>
> **Restrictions:** Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

> These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.
>
> **Variable:** Yes
>
> **Restrictions:** Valid number >= 0

Comment

> Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.
>
> **Note**   For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.
>
> You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].
>
> If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.
>
> **Variable:** Yes
>
> **Restrictions:** None

40.9 Checks

Checks are the means by which QF-Test validates correctness of the SUT. If the current state of a component doesn't match the expected value of the check node a message with a configurable error level is written to the run log. Additionally or alternatively a `CheckFailedException`[(854)] can be thrown and the result of the check can be assigned to a variable.

By setting a Timeout[(711)] checks can also be used to wait for a component to attain a certain state, e.g. to wait for a `MenuItem` to become enable or for a `CheckBox` to become selected.

Like events[(681)] each check must have a target window, component or sub-item. Depending on that target, different kinds of checks are supported which can be recorded in check-mode as described in section 4.3[(38)] via right-clicking the target component and selecting the check from the resulting menu. In case you need a check for a special component that is not offered by default you can implement custom checks via the `Checker` extension API described in section 52.5[(1073)].

There are six different data types available for checks and each of these corresponds to a specific check node. Because it is possible to have different checks of the same data type for the same target component, e.g. for the enabled state and the editable state of a text field, both boolean, checks are further qualified by their Check type identifier attribute. In most cases, when data type and target component are sufficient to uniquely identify the kind of check, the value of this attribute is *default*. In case the specified check is not supported for the given target component a `CheckNotSupportedException`[(854)] is thrown.

You can display checks independetly of the result in the HTML report. The option can be set via the check box `List checks` when interactively creating a report or via the command line argument `-report-checks`[(876)]. Please note: the option refers only to checks with default result handling, i.e. just logging to the run log, not setting a variable or throwing an exception. For more information please see section 22.1.2[(273)].

The following check nodes are available (if the respective component supports it):

- Check text[(708)]

- Boolean check[(713)]

- Check items[(719)]

- Check selectable items[(724)]

- Check image[(729)]

- Check geometry[(734)]

40.9.1 Check text

Validates the text displayed in a component or sub-item.

Contained in: All kinds of <u>sequences</u>⁽⁵¹⁸⁾.

Children: None

Execution: The text is sent to the SUT together with the data about the target component. The `TestEventQueue` determines the corresponding component in the SUT, reads its displayed text and compares it to the required value.

Attributes:

Figure 40.61: Check text attributes

Client

The name of the SUT client process to which the check is sent.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID[(813)] of the Window[(812)], Component[(823)] or Item[(829)] node that is the target of the check.

The "Select component" button ⬚ brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing [Shift-Return] or [Alt-Return], when the focus is in the text field. As an alternative you can copy the target node with [Ctrl-C] or [Edit→Copy] and insert its QF-Test component ID into the text field by pressing [Ctrl-V].

This attribute supports a special format for referencing components in other test suites (see section 24.1[(298)]). Furthermore, sub-elements of nodes can be addressed directly without requiring separate nodes for them (see section 5.9[(82)]). When using SmartIDs, you can address a GUI element directly via its recognition criteria. For more information, refer to SmartID[(73)] and Component nodes versus SmartID[(46)].

Variable: Yes

Restrictions: Must not be empty.

Text

The value to which the text displayed by the component or sub-item is compared.

4.0+ You can select [Escape text for regular expressions] from the context menu for escaping special characters of regular expressions of that text.

Variable: Yes

Restrictions: Valid regexp if required.

As regexp

If this attribute is set, the component's text is matched against a regexp (see section 47.3[(909)]) instead of comparing plain strings.

Variable: Yes

Restrictions: None

Negate

This flag determines whether to execute a positive or a negative check. If it is set,

a negative check will be performed, i.e. the checked property must not match the expected value.

Variable: Yes

Restrictions: None

Check type identifier

This attribute specifies the kind of check to perform. This makes it possible to support different kinds of checks of the same data type for a given target component without any ambiguity. With the help of a `Checker` additional check types can be implemented as shown in section 52.5[(1073)]. By default a Check text node provides the following Check types (if the respective component supports it):

Check	Description	Engines
default	Text of component	All
tooltip	Tooltip of component. In some cases you need to run a Mouse event step before that check to initialize the tooltip.	All
text_positioned	For details see PDF Check text[(242)].	PDF only
text_font	The text font of the component. For details see PDF 'Check Font'[(247)].	PDF only
text_fontsize	The text font size of the component. For details see PDF 'Check Font size'[(247)].	PDF only
class	CSS class(es) of component	Web only
id	'id' attribute of component	Web only
name	'name' attribute of component	Web only
value	'value' attribute of component	Web only
href	'href' attribute of component	Web only
attribute:NAME	attribute named NAME of component	Web only

Table 40.22: Provided Check types of Check text

Variable: Yes

Restrictions: Must not be empty

Timeout

Time limit in milliseconds until the check must succeed. To disable waiting, leave this value empty or set it to 0.

Variable: Yes

Restrictions: Must not be negative.

Variable for result

This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

If this attribute is set, the attribute Error level of message is ignored and no error is reported. The attribute Throw exception on failure always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure[(586)], Dependency[(548)] or Test case[(518)] node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test case node or, if there is no such node in the topmost node on the variables stack, falling back to the global bindings if necessary. See chapter 6[(103)] for a detailed explanation of variable binding and lookup.

In order to predefine the option use Enable 'Local variable' attribute by default[(513)].

Variable: No

Restrictions: None

Error level of message

This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

If the attribute Throw exception on failure is set, this attribute is irrelevant and if Variable for result is set this attribute is ignored.

Variable: No

Restrictions: None

Throw exception on failure

Throw an exception in case of failure. For 'Check...' nodes a `CheckFailedException`[(854)] is thrown, for 'Wait for...' nodes the respective specific exception.

Variable: No

Restrictions: None

Name

An optional name for the Check, mostly useful for the report.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.9.2 Boolean check

 Compares the current state of a component or sub-item with an expected value.

Contained in: All kinds of sequences[(518)].

Children: None

Execution: The expected state is sent to the SUT together with the data about the target component. The `TestEventQueue` determines the corresponding component in the SUT, determines its state and compares it to the required value.

Attributes:

Figure 40.62: Boolean check attributes

Client

The name of the SUT client process to which the check is sent.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID[(813)] of the Window[(812)], Component[(823)] or Item[(829)] node that is the target of the check.

The "Select component" button 🔲 brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing ⎣Shift-Return⎦ or ⎣Alt-Return⎦, when the focus is in the text field. As an alternative you can copy the target node with ⎣Ctrl-C⎦ or ⎣Edit→Copy⎦ and insert its QF-Test component ID into the text field by pressing ⎣Ctrl-V⎦.

This attribute supports a special format for referencing components in other test suites (see section 24.1[(298)]). Furthermore, sub-elements of nodes can be addressed directly without requiring separate nodes for them (see section 5.9[(82)]). When using SmartIDs, you can address a GUI element directly via its recognition criteria. For more information, refer to SmartID[(73)] and Component nodes versus SmartID[(46)].

Variable: Yes

Restrictions: Must not be empty.

Expected state

The value to which the current state of the component or sub-item is compared.

Variable: Yes

Restrictions: None

Check type identifier

This attribute specifies the kind of check to perform. This makes it possible to support different kinds of checks of the same data type for a given target component without any ambiguity. Boolean check nodes in particular are often used to check different kinds of states like 'enabled', 'editable' or 'selected' (like described in section 40.9[(707)]). The following table explains some of the check types. It depends on the component class which check types are actually available. With the help of a `Checker` additional check types can be implemented as shown in section 52.5[(1073)].

Check type	Example 1	Example 2	Details
visible		Text ☑ Box	Example 1 shows an invisible Textfield and CheckBox. Both are visible in Example 2.
editable	Text ☑ Box	Text ☑ Box	Example 1 shows a Textfield which is not editable. The Textfield in Example 2 is editable. A CheckBox does not support this kind of check.
enabled	Text ☑ Box	Text ☑ Box	Neither the Textfield nor the CheckBox is enabled in Example 1, which means you cannot interact with them. Example 2 shows enabled components.
checked (former selected)	Text ☐ Box	Text ☑ Box	Example 1 shows a CheckBox which is not selected. It is selected in Example 2. A Textfield does not support this kind of check.
selected (Table)	A / B / ☐ 1 / ☐ 1	A / B / ☑ 1 / ☐ 1	Example 1 shows a Table where no cell is selected. The lower cells are selected in Example 2. The check concerns the row selection, not the CheckBox.
focused	Text| ☑ Box	Text ☑ Box	Example 1 shows a focused Textfield (indicated by the cursor). The CheckBox is focused in Example 2 (indicated by a frame).

attribute:NAME	”attribute:sel” succeeds:
	`<p sel></p>`
	`<p sel=""></p>`
	`<p sel="text"></p>`
	”attribute:sel” fails:
	`<p sel="0"></p>`
	`<p sel="False"></p>`
	`<p></p>`
			Web only: If the attribute named NAME of the component exists (even when empty), this check succeeds unless the attribute value is ”0” or ”false” (case-insensitive). If the attribute does not exist, the check fails.

Table 40.23: Provided Check types of Boolean check

Variable: Yes

Restrictions: Must not be empty

Timeout

Time limit in milliseconds until the check must succeed. To disable waiting, leave this value empty or set it to 0.

Variable: Yes

Restrictions: Must not be negative.

Variable for result

This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

Note
If this attribute is set, the attribute Error level of message is ignored and no error is reported. The attribute Throw exception on failure always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure[586], Dependency[548] or Test case[518] node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test case node or, if there is no such node in the topmost node on the variables stack, falling back to the global bindings if necessary. See chapter 6[103] for a detailed explanation of variable binding and lookup.

In order to predefine the option use Enable 'Local variable' attribute by default[513].

Variable: No

Restrictions: None

Error level of message

This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

Note
If the attribute Throw exception on failure is set, this attribute is irrelevant and if Variable for result is set this attribute is ignored.

Variable: No

Restrictions: None

Throw exception on failure

Throw an exception in case of failure. For 'Check...' nodes a `CheckFailedException`[(854)] is thrown, for 'Wait for...' nodes the respective specific exception.

Variable: No

Restrictions: None

Name

An optional name for the Check, mostly useful for the report.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.9.3 Check items

Validates multiple displayed text strings in a component or sub-item.

Contained in: All kinds of <u>sequences</u>[(518)].

Children: None

Execution: The text strings are sent to the SUT together with the data about the target component. The `TestEventQueue` determines the corresponding component in the SUT, reads its displayed values and compares them to the required values.

Attributes:

Check items

Client

SUT

QF-Test component ID

tabNames.FirstName

Items

	Text	Regexp
0	Greg	☐
1	James	☐
2	Tina	☐

Check type identifier

default

Timeout

Result handling

Variable for result

☐ Local variable

Error level of message

Error   ⌄

$ ☐ Throw exception on failure

Name

QF-Test ID

Delay before (ms)   Delay after (ms)

Comment

Check column "First name" including sort order

Figure 40.63: Check items attributes

Client

The name of the SUT client process to which the check is sent.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID[(813)] of the Window[(812)], Component[(823)] or Item[(829)] node that is the target of the check.

The "Select component" button ⬚ brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing [Shift-Return] or [Alt-Return], when the focus is in the text field. As an alternative you can copy the target node with [Ctrl-C] or [Edit→Copy] and insert its QF-Test component ID into the text field by pressing [Ctrl-V].

This attribute supports a special format for referencing components in other test suites (see section 24.1[(298)]). Furthermore, sub-elements of nodes can be addressed directly without requiring separate nodes for them (see section 5.9[(82)]). When using SmartIDs, you can address a GUI element directly via its recognition criteria. For more information, refer to SmartID[(73)] and Component nodes versus SmartID[(46)].

Variable: Yes

Restrictions: Must not be empty.

Items

This table holds the values to which the text strings displayed by the component or sub-item are compared. Each row represents one sub-item of the target component. The "Text" column holds the actual value while the "Regexp" column determines if this value is treated as a plain string or as a regexp (see section 47.3[(909)]). The "Regexp" column allows using a variable via performing a double click on the respective cell.

See section 2.2.5[(17)] about how to work with tables in QF-Test.

4.0+ You can select [Escape text for regular expressions] from the context menu for escaping special characters of regular expressions of that cell text.

Variable: Yes for the "Text" column, otherwise no.

Restrictions: Valid regexp if required.

Check type identifier

This attribute specifies the kind of check to perform. This makes it possible to support different kinds of checks of the same data type for a given target

component without any ambiguity. With the help of a `Checker` additional check types can be implemented as shown in section 52.5[(1073)].

The check types implemented for the generic classes are described in chapter 59[(1186)] in "Additional checks", as far as appropriate, for example for Accordion[(1187)], List[(1195)], Table[(1205)], TabPanel[(1208)], TextArea[(1209)], Tree[(1212)] and TreeTable[(1213)].

With the generisc classes Table[(1205)] and TreeTable[(1213)] you can specify a subset of items to be checked for the two check types `column` and `row`. This is done via the parameters `start` and `count`. For example, the expression `row;start=2;count=3` would only check the items in row three to five, `column;start=0;count=4` would check the entries in the first four columns.

Variable: Yes

Restrictions: Must not be empty

Timeout

Time limit in milliseconds until the check must succeed. To disable waiting, leave this value empty or set it to 0.

Variable: Yes

Restrictions: Must not be negative.

Variable for result

This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

Note

If this attribute is set, the attribute Error level of message is ignored and no error is reported. The attribute Throw exception on failure always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure[(586)], Dependency[(548)] or Test case[(518)] node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test case node or, if there is no such node in the topmost node on the variables stack, falling back to the global bindings if necessary. See chapter 6[(103)] for a detailed explanation of variable binding and lookup.

In order to predefine the option use Enable 'Local variable' attribute by default[(513)].

Variable: No

Restrictions: None

Error level of message

This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

If the attribute Throw exception on failure is set, this attribute is irrelevant and if Variable for result is set this attribute is ignored.

Variable: No

Restrictions: None

Throw exception on failure

Throw an exception in case of failure. For 'Check...' nodes a CheckFailedException[(854)] is thrown, for 'Wait for...' nodes the respective specific exception.

Variable: No

Restrictions: None

Name

An optional name for the Check, mostly useful for the report.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option <u>External editor command</u>[431] lets you define an external editor in which comments can be edited conveniently by pressing [Alt-Return] or by clicking the ✎ button.

You can trigger special behaviors of some nodes using doctags, please see <u>Doctags</u>[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.9.4   Check selectable items

Validates multiple displayed text strings in a component or sub-item and additionally checks their selection state.

Contained in: All kinds of <u>sequences</u>[518].

Children: None

Execution: The text strings and selection state data are sent to the SUT together with the data about the target component. The `TestEventQueue` determines the corresponding component in the SUT, reads its displayed values, compares them to the required values and checks the required selection state.

Attributes:

Figure 40.64: Check selectable items attributes

Client

> The name of the SUT client process to which the check is sent.
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty.

QF-Test component ID

> The QF-Test ID[(813)] of the Window[(812)], Component[(823)] or Item[(829)] node that is the target of the check.
>
> The "Select component" button ⬚ brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing Shift-Return or Alt-Return, when the focus is in the text field. As an alternative you can copy the target node with Ctrl-C or Edit→Copy and insert its QF-Test component ID into the text field by pressing Ctrl-V.
>
> This attribute supports a special format for referencing components in other test suites (see section 24.1[(298)]). Furthermore, sub-elements of nodes can be addressed directly without requiring separate nodes for them (see section 5.9[(82)]). When using SmartIDs, you can address a GUI element directly via its recognition criteria. For more information, refer to SmartID[(73)] and Component nodes versus SmartID[(46)].
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty.

Items

> This table holds the values to which the text strings displayed by the component or sub-item are compared. Each row represents one sub-item of the target component. The "Text" column holds the actual value while the "Regexp" column determines if this value is treated as a plain string or as a regexp (see section 47.3[(909)]). The "Selected" column holds the required selection state. The "Regexp" and the "Selected" column allow using a variable via performing a double click on the respective cell.
>
> See section 2.2.5[(17)] about how to work with tables in QF-Test.
>
> **4.0+** You can select Escape text for regular expressions from the context menu for escaping special characters of regular expressions of that cell text.
>
> **Variable:** Yes for the "Text" column, otherwise no.
>
> **Restrictions:** Valid regexp if required.

Check type identifier

> This attribute specifies the kind of check to perform. This makes it possible to

support different kinds of checks of the same data type for a given target component without any ambiguity. With the help of a `Checker` additional check types can be implemented as shown in section 52.5[1073].

The check types implemented for the generic classes are described in chapter 59[1186] in "Additional checks", as fas as appropriate, for example for Accordion[1187], List[1195], Table[1205], Tree[1212] and TreeTable[1213].

Variable: Yes

Restrictions: Must not be empty

Timeout

Time limit in milliseconds until the check must succeed. To disable waiting, leave this value empty or set it to 0.

Variable: Yes

Restrictions: Must not be negative.

Variable for result

This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

Note If this attribute is set, the attribute Error level of message is ignored and no error is reported. The attribute Throw exception on failure always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure[586], Dependency[548] or Test case[518] node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test case node or, if there is no such node in the topmost node on the variables stack, falling back to the global bindings if necessary. See chapter 6[103] for a detailed explanation of variable binding and lookup.

In order to predefine the option use Enable 'Local variable' attribute by default[513].

Variable: No

Restrictions: None

Error level of message

This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

If the attribute Throw exception on failure is set, this attribute is irrelevant and if Variable for result is set this attribute is ignored.

Variable: No

Restrictions: None

Throw exception on failure

Throw an exception in case of failure. For 'Check...' nodes a CheckFailedException[854] is thrown, for 'Wait for...' nodes the respective specific exception.

Variable: No

Restrictions: None

Name

An optional name for the Check, mostly useful for the report.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets

you define an external editor in which comments can be edited conveniently by pressing $\boxed{\text{Alt-Return}}$ or by clicking the ✏ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.9.5 Check image

Checks the displayed image of a component. This check is supported for all kinds of components and for sub-items.

It is possible to check only parts of an image. To that end, a region within the image can be defined, either by dragging a rectangle or by editing the region attributes. Also, if the check image is smaller than the component, an offset into the component can be given. When recording only the visible part of a component or when cropping the check image to size after defining a region, the offset is determined automatically.

Contained in: All kinds of sequences[(518)].

Children: None

Execution: The image data is sent to the SUT together with the data about the target component. The `TestEventQueue` determines the corresponding component in the SUT and compares its current image to the required value.

Attributes:

Check image

Client

SUT

QF-Test component ID

tb_debugger.singlestep

Position of image relative to component

X offset

2

Y offset

2

Image (32x32) [1:1]

Actual check region inside image

X

Y

Width

Height

Algorithm for image comparison

$ ☐ Negate

Check type identifier

default

Timeout

Result handling

Variable for result

☐ Local variable

Error level of message

Error

$ ☐ Throw exception on failure

Name

QF-Test ID

Delay before (ms)

Delay after (ms)

✎ Comment

Figure 40.65: Check image attributes

Client

> The name of the SUT client process to which the check is sent.
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty.

QF-Test component ID

> The QF-Test ID[813] of the Window[812], Component[823] or Item[829] node that is the target of the check.
>
> The "Select component" button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing Shift-Return or Alt-Return, when the focus is in the text field. As an alternative you can copy the target node with Ctrl-C or Edit→Copy and insert its QF-Test component ID into the text field by pressing Ctrl-V.
>
> This attribute supports a special format for referencing components in other test suites (see section 24.1[298]). Furthermore, sub-elements of nodes can be addressed directly without requiring separate nodes for them (see section 5.9[82]). When using SmartIDs, you can address a GUI element directly via its recognition criteria. For more information, refer to SmartID[73] and Component nodes versus SmartID[46].
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty.

'Position of image relative to component'

> If the image is smaller than the component, these offset coordinates together with the size of the image determine the sub-region of the component that is checked.
>
> **Variable:** Yes
>
> **Restrictions:** Must not be negative.

Image

> The recorded image of the component. It can be displayed at different zoom levels, saved to or loaded from disk in the PNG format, or edited in an external imaging tool. The tool to use must be defined with the option External imaging program[433].
>
> The text next to the icons displayes the size and the current zoom setting of the image. Furthermore the text next to these icons may also display the current color value of the pixel over which the mouse is positioned. QF-Test may either use the hexadecimal or the rgba format in order to represent the color value. By clicking on this text it is possible to switch between these two representations.
>
> **Variable:** Yes

Restrictions: Must not be negative.

'Actual check region inside image'

If only some part of the image of a component needs to be checked, a region within the image can be defined that will then be searched for in the displayed image of the component.

Variable: Yes

Restrictions: Must not be negative.

'Algorithm for image comparison'

This attribute defines a special algorithm for image comparison. See Details about the algorithm for image comparison[1167] for further information.

Variable: Yes

Restrictions: Must match a special syntax.

Check type identifier

This attribute specifies the kind of check to perform. This makes it possible to support different kinds of checks of the same data type for a given target component without any ambiguity. The standard check type is 'default'. There are two special check types for PDF, 'scaled' and 'unscaled', which are described in the PDF section 17.3.2[245]. With the help of a `Checker` additional check types can be implemented as shown in section 52.5[1073].

Variable: Yes

Restrictions: Must not be empty

Negate

This flag determines whether to execute a positive or a negative check. If it is set, a negative check will be performed, i.e. the checked property must not match the expected value.

Variable: Yes

Restrictions: None

Timeout

Time limit in milliseconds until the check must succeed. To disable waiting, leave this value empty or set it to 0.

Variable: Yes

Restrictions: Must not be negative.

Variable for result

This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

Note
If this attribute is set, the attribute Error level of message is ignored and no error is reported. The attribute Throw exception on failure always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure[586], Dependency[548] or Test case[518] node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test case node or, if there is no such node in the topmost node on the variables stack, falling back to the global bindings if necessary. See chapter 6[103] for a detailed explanation of variable binding and lookup.

In order to predefine the option use Enable 'Local variable' attribute by default[513].

Variable: No

Restrictions: None

Error level of message

This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

Note
If the attribute Throw exception on failure is set, this attribute is irrelevant and if Variable for result is set this attribute is ignored.

Variable: No

Restrictions: None

Throw exception on failure

Throw an exception in case of failure. For 'Check...' nodes a `CheckFailedException`[854] is thrown, for 'Wait for...' nodes the respective specific exception.

Variable: No

Restrictions: None

Name

An optional name for the Check, mostly useful for the report.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ✎ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.9.6   Check geometry

Checks the location and size of a component. This check is supported for all kinds of components but not for sub-items.

Contained in: All kinds of <u>sequences</u>⁽⁵¹⁸⁾.

Children: None

Execution: The geometry data is sent to the SUT together with the data about the target component. The `TestEventQueue` determines the corresponding component in the SUT and compares location and size to the required values.

Attributes:

Check geometry

Client

SUT

QF-Test component ID

winMain

Geometry

X	Y
200	300

Width	Height
600	400

$ ☐ Negate

Check type identifier

default

Timeout

Result handling

Variable for result

☐ Local variable

Error level of message

Error ⌄

$ ☐ Throw exception on failure

Name

QF-Test ID

Delay before (ms)	Delay after (ms)

Comment

Check the geometry of the main window

Figure 40.66: Check geometry attributes

Client

> The name of the SUT client process to which the check is sent.
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty.

QF-Test component ID

> The QF-Test ID[(813)] of the Window[(812)], Component[(823)] or Item[(829)] node that is the target of the check.
>
> The "Select component" button ⬚ brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing [Shift-Return] or [Alt-Return], when the focus is in the text field. As an alternative you can copy the target node with [Ctrl-C] or [Edit→Copy] and insert its QF-Test component ID into the text field by pressing [Ctrl-V].
>
> This attribute supports a special format for referencing components in other test suites (see section 24.1[(298)]). Furthermore, sub-elements of nodes can be addressed directly without requiring separate nodes for them (see section 5.9[(82)]). When using SmartIDs, you can address a GUI element directly via its recognition criteria. For more information, refer to SmartID[(73)] and Component nodes versus SmartID[(46)].
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty.

Geometry

> The X/Y coordinates, width and height to which the location and size of the component are compared. To check only some of these, e.g. just the location or just the size, leave the others empty.
>
> **Variable:** Yes
>
> **Restrictions:** Valid number, width and height > 0

Check type identifier

> This attribute specifies the kind of check to perform. This makes it possible to support different kinds of checks of the same data type for a given target component without any ambiguity. With the help of a `Checker` additional check types can be implemented as shown in section 52.5[(1073)].
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty

Negate

This flag determines whether to execute a positive or a negative check. If it is set, a negative check will be performed, i.e. the checked property must not match the expected value.

Variable: Yes

Restrictions: None

Timeout

Time limit in milliseconds until the check must succeed. To disable waiting, leave this value empty or set it to 0.

Variable: Yes

Restrictions: Must not be negative.

Variable for result

This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

Note If this attribute is set, the attribute Error level of message is ignored and no error is reported. The attribute Throw exception on failure always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure[(586)], Dependency[(548)] or Test case[(518)] node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test case node or, if there is no such node in the topmost node on the variables stack, falling back to the global bindings if necessary. See chapter 6[(103)] for a detailed explanation of variable binding and lookup.

In order to predefine the option use Enable 'Local variable' attribute by default[(513)].

Variable: No

Restrictions: None

Error level of message

This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

Note If the attribute Throw exception on failure is set, this attribute is irrelevant and if

Variable for result is set this attribute is ignored.

Variable: No

Restrictions: None

Throw exception on failure

Throw an exception in case of failure.   For 'Check...'   nodes a CheckFailedException[(854)] is thrown, for 'Wait for...' nodes the respective specific exception.

Variable: No

Restrictions: None

Name

An optional name for the Check, mostly useful for the report.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.10 Queries

Automating a GUI test gets difficult whenever the SUT shows dynamic or unpredictable behavior, i.e. when it displays values that change with every run of the program. This is typically the case for fields that are filled automatically with things like the current time, an automatic ID from a database, etc.

QF-Test addresses this issue with means to read values from the SUT's components or to determine the numerical index of a sub-item when given its name. These values are stored in variables to be used again later as the test proceeds.

40.10.1 Fetch text

This node lets you read a value from the SUT during the execution of a test run. The text is assigned to the local or global variable (see chapter 6[103]) named by the Variable name[743] attribute.

Not all components display text and some complex components contain multiple textual items, so this operation is only supported for certain components or sub-items. If you try to fetch the text from the wrong component, an `OperationNotSupportedException`[855] is thrown, while an unsupported sub-item leads to an `UnexpectedIndexException`[855]. The following table lists the supported component and sub-item targets for this operation. (P/S) means primary/secondary index.

Web In web applications every node could contain some text, so QF-Test returns either the text or an empty value, but never throws an `OperationNotSupportedException`.

Class	Index (P/S)	Result
AbstractButton	-/-	getText()
Dialog	-/-	getTitle()
Frame	-/-	getTitle()
ComboBox	-/-	Current value (use renderer)
ComboBox	List item/-	List item (use renderer)
JEditorPane	Character index/-	Structural element at index (experimental)
Label	-/-	getText()
List	Item/-	Item (use renderer)
TabbedPane	Tab/-	Title of tab
Table	Column/Row	Cell contents (use renderer)
JTableHeader	Column/-	Column title (use renderer)
TextArea	Line number/-	Line of text
JTextComponent	-/-	getText()
Tree	Node/-	Node (use renderer)
Label	-/-	getText()
TextField	-/-	getText()

Table 40.24: Components supported by Fetch text

Contained in: All kinds of sequences[(518)].

Children: None

Execution: The data of the target component is sent to the SUT. The TestEventQueue determines the corresponding component, retrieves the requested value and sends it back to QF-Test, where it is stored in a global variable.

Attributes:

Figure 40.67: Fetch text attributes

Client

> The name of the SUT client process from which to query the data.
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty.

QF-Test component ID

> The QF-Test ID[(813)] of the Window[(812)], Component[(823)] or Item[(829)] node that is to be queried.
>
> The "Select component" button ⬚ brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing [Shift-Return] or [Alt-Return], when the focus is in the text field. As an alternative you can copy the target node with [Ctrl-C] or [Edit→Copy] and insert its QF-Test component ID into the text field by pressing [Ctrl-V].
>
> This attribute supports a special format for referencing components in other test suites (see section 24.1[(298)]). Furthermore, sub-elements of nodes can be addressed directly without requiring separate nodes for them (see section 5.9[(82)]).

When using SmartIDs, you can address a GUI element directly via its recognition criteria. For more information, refer to SmartID[73] and Component nodes versus SmartID[46].

Variable: Yes

Restrictions: Must not be empty.

Variable name

The name of the global variable to which the result of the query is assigned (see chapter 6[103]).

Variable: Yes

Restrictions: Must not be empty.

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure[586], Dependency[548] or Test case[518] node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test case node or, if there is no such node in the topmost node on the variables stack, falling back to the global bindings if necessary. See chapter 6[103] for a detailed explanation of variable binding and lookup.

In order to predefine the option use Enable 'Local variable' attribute by default[513].

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.10.2   Fetch index

With the help of this node you can determine the index of a sub-item during the execution of a test, provided its displayed text is known. Obviously only an Item[829] is supported as the target component. The result is assigned to the local or global variable (see chapter 6[103]) named by the Variable name[746] attribute.

Contained in: All kinds of sequences[518].

Children: None

Execution: The data of the target component is sent to the SUT. The TestEventQueue determines the corresponding component, searches for the requested sub-item and sends its index back to QF-Test, where it is stored in a global variable.

Attributes:

Figure 40.68: Fetch index attributes

Client

The name of the SUT client process from which to query the data.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID[813] of the Window[812], Component[823] or Item[829] node that is to be queried.

The "Select component" button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing Shift-Return or Alt-Return, when the focus is in the text field. As an alternative you can copy the target node with Ctrl-C or Edit→Copy and insert its QF-Test component ID into the text field by pressing Ctrl-V.

This attribute supports a special format for referencing components in other test suites (see section 24.1[298]). Furthermore, sub-elements of nodes can be addressed directly without requiring separate nodes for them (see section 5.9[82]).

When using SmartIDs, you can address a GUI element directly via its recognition criteria. For more information, refer to SmartID[(73)] and Component nodes versus SmartID[(46)].

Variable: Yes

Restrictions: Must not be empty.

Variable name

The name of the global variable to which the result of the query is assigned (see chapter 6[(103)]).

Variable: Yes

Restrictions: Must not be empty.

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure[(586)], Dependency[(548)] or Test case[(518)] node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test case node or, if there is no such node in the topmost node on the variables stack, falling back to the global bindings if necessary. See chapter 6[(103)] for a detailed explanation of variable binding and lookup.

In order to predefine the option use Enable 'Local variable' attribute by default[(513)].

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ✐ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.10.3 Fetch geometry

Use this node to find out the geometry of a window, component or sub-item in the SUT. The result is stored in up to four local or global variables, one each for the X and Y coordinates, width and height.

This node is useful if you want to set the X[683] and Y[683] coordinates of a Mouse event[681] relative to the right or bottom border of the target component. Simply fetch the component's width and height and define the coordinates using the extended variable syntax for expressions (see section 6.6[114]).

The following table lists the supported sub-item targets for this operation. (P/S) means primary/secondary index.

Class	Index (P/S)	Result
JList	Item/-	Item
JTabbedPane	Tab/-	Tab
JTable	Column/-	Column
JTable	Column/Row	Cell
JTableHeader	Column/-	Column title
JTree	Node/-	Node

Table 40.25: Components supported by Fetch geometry

Contained in: All kinds of sequences[518].

Children: None

Execution: The data of the target component is sent to the SUT. The `TestEventQueue` determines the corresponding component, retrieves its geometry and sends it back to QF-Test, where the values are stored in global variables.

Attributes:

Fetch geometry

Client
SUT

QF-Test component ID
winMain

☑ Location relative to window

Variable for x
xMain

Variable for y
yMain

Variable for width

Variable for height

☐ Local variable

QF-Test ID

Delay before (ms)

Delay after (ms)

✎ Comment
Fetch the position of the main window.

Figure 40.69: Fetch geometry attributes

Client

The name of the SUT client process from which to query the data.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The $\underline{\text{QF-Test ID}}^{(813)}$ of the $\underline{\text{Window}}^{(812)}$, $\underline{\text{Component}}^{(823)}$ or $\underline{\text{Item}}^{(829)}$ node that is to be queried.

The "Select component" button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing Shift-Return or Alt-Return, when the focus is in the text field. As an alternative you can copy the target node with Ctrl-C or Edit→Copy and insert its QF-Test component ID into the text field by pressing Ctrl-V.

This attribute supports a special format for referencing components in other test suites (see $\underline{\text{section 24.1}}^{(298)}$). Furthermore, sub-elements of nodes can be addressed directly without requiring separate nodes for them (see $\underline{\text{section 5.9}}^{(82)}$). When using SmartIDs, you can address a GUI element directly via its recognition criteria. For more information, refer to $\underline{\text{SmartID}}^{(73)}$ and $\underline{\text{Component nodes versus}}$ $\underline{\text{SmartID}}^{(46)}$.

Variable: Yes

Restrictions: Must not be empty.

Location relative to window

This attribute determines whether the X and Y coordinate of a component or sub-item is calculated relative to its parent component or relative to its parent window. For windows there is no difference.

Variable: No

Restrictions: None

Variable for x

The name of the global variable to which the X coordinate of the window, component or sub-item is assigned (see $\underline{\text{chapter 6}}^{(103)}$). If you are not interested in the X coordinate, leave this value empty.

Variable: Yes

Restrictions: None

Variable for y

The name of the global variable to which the Y coordinate of the window, component or sub-item is assigned (see $\underline{\text{chapter 6}}^{(103)}$). If you are not interested in the Y coordinate, leave this value empty.

Variable: Yes

Restrictions: None

Variable for width

The name of the global variable to which the width of the window, component or

sub-item is assigned (see chapter 6[(103)]). If you are not interested in the width, leave this value empty.

Variable: Yes

Restrictions: None

Variable for height

The name of the global variable to which the height of the window, component or sub-item is assigned (see chapter 6[(103)]). If you are not interested in the height, leave this value empty.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create local or global variable bindings. If unset, the variables are bound in the global variables. If set, the topmost current binding for a variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure[(586)], Dependency[(548)] or Test case[(518)] node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test case node or, if there is no such node in the topmost node on the variables stack, falling back to the global bindings if necessary. See chapter 6[(103)] for a detailed explanation of variable binding and lookup.

In order to predefine the option use Enable 'Local variable' attribute by default[(513)].

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the   button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.11   Miscellaneous

This section lists the remaining nodes that don't fit in well with any of the other sections.

40.11.1   Comment

This node is thought for documentation purpose. It allows you to add a comment to your test suite.

Contained in: Everywhere

Children: None

Execution: A comment node does not affect execution.

Attributes:

Figure 40.70: Comment attributes

Heading

> The string that should get displayed in the tree, a summary of the comment or the comment itself.
>
> This attribute allows to use the following HTML-Tags <i>italic Text</i>, <u>underlined text</u>, <s>stroke text</s> and bold text in order to beautify the representation in the tree. Furthermore it is possible to specify a text color via a style="color:colorname" or the color="colorname" attribute (also in combination with the <span>, <em>, <strong> or <font>tag).
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty.

Comment

> Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note
> For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ✎ button.
>
> You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].
>
> If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.
>
> **Variable:** Yes
>
> **Restrictions:** None

40.11.2   Error

With the Error node you can write an error to the run log. Use the attributes to specify the information to be written to the run log.

The node can replace scripts containing only `rc.logError(...)`. It can also replace calls to the procedure `qfs.run-log.logError` of The standard library[(158)].

Contained in: All kinds of sequences[(518)].

Children: None

Execution: The Error node writes an error to the run log.

Attributes:

Figure 40.71: Error attributes

Text

The message text. In the tree node long texts will be truncated.

Variable: Yes

Restrictions: None

Add diagnostic client information

When selected, further information concerning each client process will be written to the run log.

Variable: Yes

Restrictions: None

Print message to terminal also

If active, the message is also written to the QF-Test Terminal.

Variable: Yes

Restrictions: None

Create screenshots

The attribute indicates whether QF-Test should log screenshots to the run-log of the whole monitor(s) attached, taking into account the setting of the option Limit screenshots to relevant screens[(509)] relevant to security and data protection. The default setting of the option only allows screenshots of monitors where either a window of QF-Test or the tested application is showing. By default, in batch mode (no QF-Test GUI) no screenshot will be logged when no SUT window is showing.

Setting	Description
Always	Always log screenshots to the run-log. This may result in memory issues in case a test-run has many errors. Options for splitting run logs[505] may be used to reduce memory consumption in such a case. The option overrules the following settings: • Maximum number of errors with screenshots per run log[508] • Count screenshots individually for each split log[509] • Create screenshots of the whole screen upon error[509]
Never	Do NOT log screenshots.
Based on options	Log and create screenshots depending on the options set. See Options determining run log content[507] for further information.
Variables, for example `$(logScreenshots)`	A reference to a variable containing one of the following values (case insensitive). Depending on the value QF-Test will either always log the screenshots or never or take the options into account. The following values indicate QF-Test should always log screenshots: `always`, `1`, `true` or `yes`. The following values indicate QF-Test should never log screenshots: `never`, `0`, `false` or `no`. The following values indicate QF-Test should decide based on the options: `based on options`, `options` or `option`.

Table 40.26: Settings for "Create Screenshots"

Variable: Yes

Restrictions: None

Create client screenshots

The attribute indicates whether QF-Test should log screenshots of all client windows to the run-log, even if they may be hidden by other windows.

Setting	Description
Always	Always log screenshots to the run-log. This may result in memory issues in case a test-run has many errors. Options for splitting run logs[505] may be used to reduce memory consumption in such a case. The option overrules the following settings: • Maximum number of errors with screenshots per run log[508] • Count screenshots individually for each split log[509] • Create screenshots of the client's windows upon error in client[509] • Create screenshots of all client windows upon error[509]
Never	Do NOT log screenshots.
Based on options	Log and create screenshots depending on the options set. See Options determining run log content[507] for further information. The setting of the option Create screenshots of the client's windows upon error in client[509] is irrelevant, as the node has effect on all clients, just like a Server script.
Variables, for example `$(logScreenshots)`	A reference to a variable containing one of the following values (case insensitive). Depending on the value QF-Test will either always log the screenshots or never or take the options into account. The following values indicate QF-Test should always log screenshots: `always`, `1`, `true` or `yes`. The following values indicate QF-Test should never log screenshots: `never`, `0`, `false` or `no`. The following values indicate QF-Test should decide based on the options: `based on options`, `options` or `option`.

Table 40.27: Settings for "Create Client Screenshots"

Variable: Yes

Restrictions: None

QF-Test ID

> At the moment the QF-Test ID attribute has no meaning for this type of node.
>
> **Variable:** No
>
> **Restrictions:** Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

> These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.
>
> **Variable:** Yes
>
> **Restrictions:** Valid number >= 0

Comment

> Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note

> For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing ⌈Alt-Return⌋ or by clicking the ✎ button.
>
> You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].
>
> If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.
>
> **Variable:** Yes
>
> **Restrictions:** None

40.11.3   Warning

> With the Warning node you can write a warning to the run log. Use the attributes to specify the information to be written to the run log.
>
> The node can replace scripts containing only `rc.logWarning(...)`. It can also replace calls to the procedure `qfs.run-log.logWarning` of The standard library[158].

Contained in: All kinds of sequences[518].

Children: None

Execution: The Warning node writes a warning to the run log.

Attributes:

Figure 40.72: Warning attributes

Text

The message text. In the tree node long texts will be truncated.

Variable: Yes

Restrictions: None

Include in report

When this option has been selected, the node will be written to the report.

Variable: Yes

Restrictions: None

Add diagnostic client information

When selected, further information concerning each client process will be written to the run log.

Variable: Yes

Restrictions: None

Print message to terminal also

If active, the message is also written to the QF-Test Terminal.

Variable: Yes

Restrictions: None

Create screenshots

The attribute indicates whether QF-Test should log screenshots to the run-log of the whole monitor(s) attached, taking into account the setting of the option Limit screenshots to relevant screens[(509)] relevant to security and data protection. The default setting of the option only allows screenshots of monitors where either a window of QF-Test or the tested application is showing. By default, in batch mode (no QF-Test GUI) no screenshot will be logged when no SUT window is showing.

Setting	Description
Always	Always log screenshots to the run-log. This may result in memory issues in case a test-run has many errors. Options for splitting run logs[(505)] may be used to reduce memory consumption in such a case. The option overrules the following settings: • Maximum number of errors with screenshots per run log[(508)] • Count screenshots individually for each split log[(509)] • Create screenshots of the whole screen upon error[(509)] • Create screenshots for warnings[(509)]
Never	Do NOT log screenshots.
Based on options	Log and create screenshots depending on the options set. See Options determining run log content[(507)] for further information. Please note: Screenshots will only be logged when the option Create screenshots for warnings[(509)] is active.
Variables, for example `$(logScreenshots)`	A reference to a variable containing one of the following values (case insensitive). Depending on the value QF-Test will either always log the screenshots or never or take the options into account. The following values indicate QF-Test should always log screenshots: `always`, `1`, `true` or `yes`. The following values indicate QF-Test should never log screenshots: `never`, `0`, `false` or `no`. The following values indicate QF-Test should decide based on the options: `based on options`, `options` or `option`.

Table 40.28: Settings for "Create Screenshots"

Variable: Yes

Restrictions: None

Create client screenshots

The attribute indicates whether QF-Test should log screenshots of all client windows to the run-log, even if they may be hidden by other windows.

Setting	Description
Always	Always log screenshots to the run-log. This may result in memory issues in case a test-run has many errors. Options for splitting run logs[505] may be used to reduce memory consumption in such a case. The option overrules the following settings: • Maximum number of errors with screenshots per run log[508] • Count screenshots individually for each split log[509] • Create screenshots of the client's windows upon error in client[509] • Create screenshots of all client windows upon error[509] • Create screenshots for warnings[509]
Never	Do NOT log screenshots.
Based on options	Log and create screenshots depending on the options set. See Options determining run log content[507] for further information. The setting of the option Create screenshots of the client's windows upon error in client[509] is irrelevant, as the node has effect on all clients, just like a Server script. Please note: Screenshots will only be logged when the option Create screenshots for warnings[509] is active.
Variables, for example `$(logScreenshots)`	A reference to a variable containing one of the following values (case insensitive). Depending on the value QF-Test will either always log the screenshots or never or take the options into account. The following values indicate QF-Test should always log screenshots: `always`, `1`, `true` or `yes`. The following values indicate QF-Test should never log screenshots: `never`, `0`, `false` or `no`. The following values indicate QF-Test should decide based on the options: `based on options`, `options` or `option`.

Table 40.29: Settings for "Create Client Screenshots"

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note   For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing [Alt-Return] or by clicking the ✏ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.11.4   Message

With the Message node you can write a message to the run log. Use the attributes to specify the information to be written to the run log.

The node can replace scripts containing only `rc.logMessage(...)`. It can also replace calls to the procedure `qfs.run-log.logMessage` of The standard library[(158)].

Contained in: All kinds of sequences[518].

Children: None

Execution: The Message step writes a message to the run log.

Attributes:

Figure 40.73: Message attributes

Text

>   The message text. In the tree node long texts will be truncated.

>   **Variable:** Yes

>   **Restrictions:** None

Prevent compactification

>   The option is only relevant when the option Create compact run log[510] has been set to compact run logs. In that case activate the option keep the node in the run log.

> **Variable:** Yes
>
> **Restrictions:** None

Include in report

> When selected, the node will be written to the report.
>
> **Variable:** Yes
>
> **Restrictions:** None

Add diagnostic client information

> When selected, further information concerning each client process will be written to the run log.
>
> **Variable:** Yes
>
> **Restrictions:** None

Print message to terminal also

> If active, the message is also written to the QF-Test Terminal.
>
> **Variable:** Yes
>
> **Restrictions:** None

Create screenshots

> The attribute indicates whether QF-Test should log screenshots to the run-log of the whole monitor(s) attached, taking into account the setting of the option Limit screenshots to relevant screens[(509)] relevant to security and data protection. The default setting of the option only allows screenshots of monitors where either a window of QF-Test or the tested application is showing. By default, in batch mode (no QF-Test GUI) no screenshot will be logged when no SUT window is showing.

Setting	Description
Always	Always log screenshots to the run-log. This may result in memory issues in case a test-run has many errors. Options for splitting run logs[505] may be used to reduce memory consumption in such a case. The option overrules the following settings: • Maximum number of errors with screenshots per run log[508] • Count screenshots individually for each split log[509] • Create screenshots of the whole screen upon error[509]
Never	Do NOT log screenshots.
Based on options	Log and create screenshots depending on the options set. See Options determining run log content[507] for further information.
Variables, for example `$(logScreenshots)`	A reference to a variable containing one of the following values (case insensitive). Depending on the value QF-Test will either always log the screenshots or never or take the options into account. The following values indicate QF-Test should always log screenshots: `always`, `1`, `true` or `yes`. The following values indicate QF-Test should never log screenshots: `never`, `0`, `false` or `no`. The following values indicate QF-Test should decide based on the options: `based on options`, `options` or `option`.

Table 40.30: Settings for "Create Screenshots"

Variable: Yes

Restrictions: None

Create client screenshots

The attribute indicates whether QF-Test should log screenshots of all client windows to the run-log, even if they may be hidden by other windows.

Setting	Description
Always	Always log screenshots to the run-log. This may result in memory issues in case a test-run has many errors. Options for splitting run logs[505] may be used to reduce memory consumption in such a case. The option overrules the following settings: • Maximum number of errors with screenshots per run log[508] • Count screenshots individually for each split log[509] • Create screenshots of the client's windows upon error in client[509] • Create screenshots of all client windows upon error[509]
Never	Do NOT log screenshots.
Based on options	Log and create screenshots depending on the options set. See Options determining run log content[507] for further information. The setting of the option Create screenshots of the client's windows upon error in client[509] is irrelevant, as the node has effect on all clients, just like a Server script.
Variables, for example $(logScreenshots)	A reference to a variable containing one of the following values (case insensitive). Depending on the value QF-Test will either always log the screenshots or never or take the options into account. The following values indicate QF-Test should always log screenshots: `always`, `1`, `true` or `yes`. The following values indicate QF-Test should never log screenshots: `never`, `0`, `false` or `no`. The following values indicate QF-Test should decide based on the options: `based on options`, `options` or `option`.

Table 40.31: Settings for "Create Client Screenshots"

Variable: Yes

Restrictions: None

QF-Test ID

> At the moment the QF-Test ID attribute has no meaning for this type of node.
>
> **Variable:** No
>
> **Restrictions:** Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

> These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.
>
> **Variable:** Yes
>
> **Restrictions:** Valid number >= 0

Comment

> Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note

> For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ⬚ button.
>
> You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].
>
> If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.
>
> **Variable:** Yes
>
> **Restrictions:** None

40.11.5 Set variable

> $ This node lets you set the value of a global variable. If the test is run interactively from QF-Test and not in batch mode (see section 1.7[(11)]) you can optionally set the value interactively.

Contained in: All kinds of sequences[(518)].

Children: None

Execution: If the test is run interactively and the Interactive[(770)] attribute is set, a dialog is shown in which the value for the variable can be entered. If the Timeout[(770)] is exceeded

or the value is confirmed with the OK button, the variable is bound accordingly in the global variables. If the dialog is canceled, the test run is stopped. In the non-interactive case the variable is bound directly to the Default value[(770)].

Attributes:

Figure 40.74: Set variable attributes

Variable name

The name of the global variable to which the value is assigned (see chapter 6[(103)]).

Variable: Yes

Restrictions: Must not be empty.

Local variable

This flag determines whether to create a local or global variable binding. If unset,

the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure[586], Dependency[548] or Test case[518] node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test case node or, if there is no such node in the topmost node on the variables stack, falling back to the global bindings if necessary. See chapter 6[103] for a detailed explanation of variable binding and lookup.

In order to predefine the option use Enable 'Local variable' attribute by default[513].

Variable: No

Restrictions: None

Default value

The default value for the variable if the test is run non-interactively, the Interactive[770] attribute is not set or the Timeout[770] is exceeded.

Variable: Yes

Restrictions: None

Interactive

Whether a dialog should be shown in which the value for the global variable can be entered. Ignored if the test is run non-interactively.

Variable: Yes

Restrictions: None

Description

A short description to display in the dialog. If you leave this value empty, the description `Value for` <Variable name> will be used.

Variable: Yes

Restrictions: None

Timeout

An optional timeout value for the dialog. If the dialog is shown and the value is left unchanged for the specified period of time, the dialog is closed automatically and the default value is used. This avoids blocking a test that is started interactively from QF-Test and then left to run unattended.

Variable: Yes

Restrictions: Empty or > 0.

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note   For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the [✎] button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.11.6 Wait for component to appear

This node is very important for the timing of a test run. The reaction time of the SUT varies depending on system and memory load, so it may take a while until, say, a complex dialog is opened. This node will delay further execution of the test until the desired component or sub-item is available. If the time limit is exceeded without success, a ComponentNotFoundException[850] is thrown. You can also use the Variable for result attribute to store the result into a variable and the Throw exception on failure attribute to suppress the exception.

This node is intended only for relatively long delays. Short delays are handled automatically by the Timeout options[477].

By setting the Wait for absence[774] attribute this node can also be used to ensure the absence of a component.

Contained in: All kinds of sequences[(518)].

Children: None

Execution: The data of the target component are sent to the SUT. The `TestEventQueue` waits until either the corresponding component becomes available or the time limit is exceeded.

Attributes:

Figure 40.75: Wait for component to appear attributes

Client

The name of the Java process in which to wait.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID[(813)] of the Window[(812)], Component[(823)] or Item[(829)] to wait for.

The "Select component" button 🔲 brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing Shift-Return or Alt-Return, when the focus is in the text field. As an alternative you can copy the target node with Ctrl-C or Edit→Copy and insert its QF-Test component ID into the text field by pressing Ctrl-V.

This attribute supports a special format for referencing components in other test suites (see section 24.1[(298)]). Furthermore, sub-elements of nodes can be addressed directly without requiring separate nodes for them (see section 5.9[(82)]). When using SmartIDs, you can address a GUI element directly via its recognition criteria. For more information, refer to SmartID[(73)] and Component nodes versus SmartID[(46)].

Variable: Yes

Restrictions: Must not be empty.

Timeout

Time limit in milliseconds.

Variable: Yes

Restrictions: >= 0

Wait for absence

If this attribute is set, QF-Test waits for the absence of a component or a sub-item. This is useful e.g. to ensure that a dialog has been closed or was never opened in the first place. If the component or sub-item remains visible for the whole time, a `ComponentFoundException`[(851)] is thrown.

Variable: Yes

Restrictions: None

Variable for result

This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

Note   If this attribute is set, the attribute Error level of message is ignored and no error is

reported. The attribute Throw exception on failure always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure[(586)], Dependency[(548)] or Test case[(518)] node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test case node or, if there is no such node in the topmost node on the variables stack, falling back to the global bindings if necessary. See chapter 6[(103)] for a detailed explanation of variable binding and lookup.

In order to predefine the option use Enable 'Local variable' attribute by default[(513)].

Variable: No

Restrictions: None

Error level of message

This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

If the attribute Throw exception on failure is set, this attribute is irrelevant and if Variable for result is set this attribute is ignored.

Variable: No

Restrictions: None

Throw exception on failure

Throw an exception in case of failure. For 'Check...' nodes a `CheckFailedException`[(854)] is thrown, for 'Wait for...' nodes the respective specific exception.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

> These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.
>
> **Variable:** Yes
>
> **Restrictions:** Valid number >= 0

Comment

> Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

> For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.
>
> You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].
>
> If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.
>
> **Variable:** Yes
>
> **Restrictions:** None

40.11.7   Wait for document to load

This node is a variant of the Wait for component to appear[(771)] node specifically for web pages. It not only checks the existence of a given document. If the target document was already known to exist the last time an event was replayed, this node waits for the document to get reloaded. When working with web pages it is often the case that the same or very similar documents are loaded many times. Without this node's functionality QF-Test could not discern the case where the old document is still around from the one where the document is already reloaded. In the former case, replaying an event could cause it to have no effect at all because at the same time reloading of the document begins.

The Name of the browser window attribute can be used to limit the search to a given browser window or to define a name for a new window. If Stop loading if timeout exceeded is set, QF-Test will abort loading the document if it doesn't finish in time. You can also use the Variable for result attribute to store the result in a variable and the Throw exception on failure attribute to suppress the `DocumentNotLoadedException`[(850)].

Contained in: All kinds of sequences[(518)].

Children: None

Execution: The data of the target document are sent to the SUT. The
`TestEventQueue` waits until corresponding document gets loaded or the time limit is
exceeded in which case a `DocumentNotLoadedException` is thrown.

Attributes:

Figure 40.76: Wait for document to load attributes

Client

> The name of the SUT client process from which to query the data.
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty.

QF-Test component ID

> The QF-Test ID[(813)] of the Window[(812)], Component[(823)] or Item[(829)] node that is to be queried.
>
> The "Select component" button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing Shift-Return or Alt-Return, when the focus is in the text field. As an alternative you can copy the target node with Ctrl-C or Edit→Copy and insert its QF-Test component ID into the text field by pressing Ctrl-V.
>
> This attribute supports a special format for referencing components in other test suites (see section 24.1[(298)]). Furthermore, sub-elements of nodes can be addressed directly without requiring separate nodes for them (see section 5.9[(82)]). When using SmartIDs, you can address a GUI element directly via its recognition criteria. For more information, refer to SmartID[(73)] and Component nodes versus SmartID[(46)].
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty.

Name of the browser window

> This attribute has a dual use. If set to an existing name for a browser window, QF-Test waits for the document to load in that window. If the name is set but no browser window by that name exists, the search is limited to documents in new or not-yet-named windows. If the wait succeeds and a new document is loaded, the window name is assigned to the document's browser window. This is the only way to define a name for a popup window. Explicitly launched browsers can have their name set via the Name of the browser window[(651)] attribute of a Open browser window[(649)] node. You find a brief description how to handle multiple browser windows in FAQ 25.
>
> **Variable:** Yes
>
> **Restrictions:** None

Timeout

> Time limit in milliseconds.
>
> **Variable:** Yes
>
> **Restrictions:** >= 0

Stop loading if timeout exceeded

If this attribute is set and the timeout is exceeded without a matching document finishing to load, QF-Test will cause loading to stop, either in all browsers or, if Name of the browser window is set, the browser window specified therein. Result and exception handling are not affected by this attribute. If the timeout is exceeded the operation is considered a failure regardless of whether loading is stopped or not.

Variable: Yes

Restrictions: None

Variable for result

This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

Note   If this attribute is set, the attribute Error level of message is ignored and no error is reported. The attribute Throw exception on failure always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure[(586)], Dependency[(548)] or Test case[(518)] node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test case node or, if there is no such node in the topmost node on the variables stack, falling back to the global bindings if necessary. See chapter 6[(103)] for a detailed explanation of variable binding and lookup.

In order to predefine the option use Enable 'Local variable' attribute by default[(513)].

Variable: No

Restrictions: None

Error level of message

This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

Note   If the attribute Throw exception on failure is set, this attribute is irrelevant and if Variable for result is set this attribute is ignored.

Variable: No

Restrictions: None

Throw exception on failure

Throw an exception in case of failure.   For 'Check...'   nodes a `CheckFailedException`[(854)] is thrown, for 'Wait for...'   nodes the respective specific exception.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the   button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.11.8   Wait for download to finish

Web

This specialized node is applicable only for web Clients. It can be used to wait for the completion of a download that was earlier started via QF-Test. This is important in case you need to verify the contents of the file or to measure the time it took to download it.

If the timeout is exceeded without the download finishing, a DownloadNotCompleteException is thrown unless suppressed via the Throw exception on failure attribute. Either way the download can be canceled by activating the Cancel download if timeout exceeded attribute, which may be necessary to enable another download to the same file. The result can also be stored in a variable by defining its name in the Variable for result attribute.

Contained in: All kinds of sequences[(518)].

Children: None

Execution: The target file identifying the download is sent to the SUT where QF-Test waits for the download to finish or the time limit is exceeded in which case a `DownloadNotCompleteException` is thrown.

Attributes:

Figure 40.77: Wait for download to finish attributes

Client

The name of the SUT client process from which to query the data.

Variable: Yes

Restrictions: Must not be empty.

File

The target file for the downloaded.

Variable: Yes

Restrictions: Valid file name

Timeout

Time limit in milliseconds.

Variable: Yes

Restrictions: >= 0

Cancel download if timeout exceeded

If this attribute is set and the timeout is exceeded without the download finishing the download is canceled.

Variable: Yes

Restrictions: None

Variable for result

This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

Note If this attribute is set, the attribute Error level of message is ignored and no error is reported. The attribute Throw exception on failure always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure[(586)], Dependency[(548)] or Test case[(518)] node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test case node or, if there is no such node in the topmost node on the variables stack, falling back to the global bindings if necessary. See chapter 6[(103)] for a detailed explanation of variable binding and lookup.

In order to predefine the option use Enable 'Local variable' attribute by default[(513)].

Variable: No

Restrictions: None

Error level of message

This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

If the attribute Throw exception on failure is set, this attribute is irrelevant and if Variable for result is set this attribute is ignored.

Variable: No

Restrictions: None

Throw exception on failure

Throw an exception in case of failure. For 'Check...' nodes a CheckFailedException[(854)] is thrown, for 'Wait for...' nodes the respective specific exception.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing [Alt-Return] or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.11.9 Load resources

This node is used to load a `ResourceBundle` and make its values available for the extended variable syntax `${group:name}` (see section 6.5[(108)]). To learn more about `ResourceBundles` see the description of the ResourceBundle[(786)] attribute.

Contained in: All kinds of sequences[(518)].

Children: None

Execution: The `ResourceBundle` is loaded and its values are made available under the Group[(785)] name.

Attributes:

Load resources

Group

msg

ResourceBundle

rsc.messages

Locale

en_US

QF-Test ID

Delay before (ms)   Delay after (ms)

✏ Comment

Load American message strings for ${msg:...}

Figure 40.78: Load resources attributes

Group

The name of the group by which values of the `ResourceBundle` are referred to.

The value of a definition of the form `name=value` in the `ResouceBundle` can be retrieved with `${group:name}` (see section 6.5[(108)]).

Variable: Yes

Restrictions: Must not be empty and should not contain special characters like ':' or '$'.

ResourceBundle

The name of the `ResourceBundle` to load. A little Java background is needed to understand this attribute.

The resources are read with the help of the Java method `ResourceBundle.getBundle()`. For this to work, a matching file with the extension `.class` or `.properties` must be located somewhere on the `CLASSPATH`. Use the fully qualified name for the file, including packages, with a dot ('.') as separator, but without extension or locale identifier.

Example: QF-Test comes with a German `ResourceBundle` in the file `de/qfs/apps/qftest/resources/properties/qftest_de.properties`, which is contained in the archive `qfshared.jar`. To load that `ResourceBundle`, set this attribute to `de.qfs.apps.qftest.resources.properties.qftest` and the Locale[(786)] to `de`.

Variable: Yes

Restrictions: Must name a `ResourceBundle` on the `CLASSPATH`.

Locale

The main use of `ResourceBundles` is to provide data in different languages. This attribute determines, which version of a `ResourceBundle` is retrieved. The value must follow the ISO standard *language_country_variant*. *Language* is a two letter lowercase code like `en` for English, *country* a two letter uppercase code like `US` for American or `UK` for British English. The *variant* discriminates further but is rarely used.

As mentioned, QF-Test relies on the Java method `ResourceBundle.getBundle()` to load the `ResourceBundle`, which is described in detail in the Java documentation and works as follows:

To load a `ResourceBundle` named `res` for the locale `en_US`, Java first searches the `CLASSPATH` for a file named `res_en_US.class` or `res_en_US.properties`, then for `res_en.class` or `res_en.properties` and finally for `res.class` and `res.properties`. The less specific files are loaded even if more specific files are found, but only values not defined in the more specific files are used. That way you can define all English

resources in `res_en.properties` and place only those that differ in `res_en_UK.properties` and `res_en_US.properties`.

Unfortunately Java has a "feature" that can lead to surprising results. If no specific file but only the base file `res.properties` is found, Java tries the whole process a second time, this time for the current default locale of the VM. As a result, if the current locale for QF-Test is German and you want to load English resources that are defined in `res.properties` and no `res_en.properties` exists, Java will load the German version from `res_de.properties`, even if you request the locale `en`. You can work around this be setting this attribute to the underscore '_'. In that case, only the base file `res.properties` is loaded.

To use the current locale of the VM, leave this value empty.

Variable: Yes

Restrictions: Empty or valid locale identifier.

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note   For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

40.11.10 Load properties

This node is used to load data from a `Properties` file and make its values available for the extended variable syntax $\${group:name}$ (see section 6.5[108]).

`Properties` files are easier to handle than `ResourceBundles` since you request the file directly, but they are less powerful. The format of a `Properties` file is simple: lines of the form `name=value` with arbitrary whitespace around the '=' character. Complex definitions spanning multiple lines are possible. Please see the Java documentation for details or ask a developer.

Contained in: All kinds of sequences[518].

Children: None

Execution: The `Properties` file is loaded and its values are made available under the Group[788] name.

Attributes:

Figure 40.79: Load properties attributes

Group

> The name of the group by which values of the `Properties` file are referred to. The value of a definition of the form `name=value` in the `Properties` file can be retrieved with `${group:name}` (see section 6.5[(108)]).
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty and should not contain special characters like ':' or '$'.

Properties file

> The file to load the `Properties` from. This can either be an absolute path name or a path relative to the directory of the current suite. In either case you should always use '/' as the separator for directories, even under Windows. QF-Test will translate this to the correct value for the current operating system.
>
> The "..." button brings up a dialog in which you can select the file interactively. You can also get to this dialog by pressing Shift-Return or Alt-Return, when the focus is in the text field.
>
> **Variable:** Yes
>
> **Restrictions:** Must be an existing `Properties` file.

File encoding is UTF-8

> Up to Java 8 the class `java.util.Properties` enforced a file encoding of ISO-Latin-1 for properties files. In Java 9 the default encoding is UTF-8. QF-Test supports both and uses the UTF-8 encoding if this attribute is activated and ISO-Latin-1 otherwise.
>
> **Variable:** Yes
>
> **Restrictions:** None

QF-Test ID

> At the moment the QF-Test ID attribute has no meaning for this type of node.
>
> **Variable:** No
>
> **Restrictions:** Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

> These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.
>
> **Variable:** Yes
>
> **Restrictions:** Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing [Alt-Return] or by clicking the ✏ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.11.11   Unit test

This node is used to execute JUnit tests.

`JUnit` tests are made for executing unit and integration tests. They should be short and easily repeatable. Unit Tests can be defined in an SUT script or loaded from the SUT or other classpaths.

Contained in: All kinds of sequences[(518)].

Children: None

Execution: The required resources and injections are loaded and the test classes are executed step by step.

Attributes:

Unit test

☑ Run in Unit Test Execution Environment

Source for the tests

Script

Script | ∨ Templates

```
1 @Test
2 void indexOutOfBoundsAccess() {
3     def x = 4
4     def y = 3
5     def sum = x + y
6     assert(sum == 7)
7 }
8
```

Script language

Groovy

+ ╱ ✗ ↑ ↓ Classpath

Type	Path

+ ╱ ✗ ↑ ↓ Injections

Type	Field	Value

Name

Server UnitTest

QF-Test ID

Delay before (ms)   Delay after (ms)

╱ Comment

Figure 40.80: Unit test server attributes

Figure 40.81: Unit test client attributes

Run in Unit Test Execution Environment

Whether to execute the unit tests inside the SUT. If disabled a execution environment is setup for the tests.

Variable: No

Restrictions: None

Client

The name of the SUT client process in which to execute the script.

Variable: Yes

Restrictions: Must not be empty.

Source for the tests

The source for the JUnit tests. This can either be an SUT script or Java classes that are loaded into the SUT.

Script

The script to execute.

Note

You may use QF-Test variables of the syntax $(var) or ${group:name} in Jython scripts. They will be expanded before the script is passed to the Jython interpreter. This can lead to unexpected behavior. `rc.lookup(...)` is the preferred method in this case (see section 11.2.3[(166)] for details).

Note

In spite of syntax highlighting and automatical indentation this attribute might not be the right place to write complex scripts. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which scripts can be edited conveniently by pressing

Alt-Return or by clicking the ✏ button. Complex scripts can also be written as separate modules which can then be imported for use in this attribute. See chapter 48[(915)] for details.

Variable: Yes

Restrictions: Valid syntax

Templates

This dropdown menu contains a list of useful template scripts. The available templates will differ depending on the chosen script type and interpreter.

When you choose one of these templates, the current contents of your script will be replaced.

You can add your own templates to this menu by choosing "Open user templates directory" and placing your template files there. The following file types are valid:

- **[directory]:** Will be converted into a sub menu.
- **.py:** A Jython script template.
- **.groovy:** A Groovy script template.

- **.js:** A JavaScript script template.

Script language

This attribute determines the interpreter in which to run the script, or in other words, the scripting language to use. Possible values are "Jython", "Groovy" and "JavaScript".

Variable: No

Restrictions: None

Test classes

These are the classes that are executed. They have to be loaded with the defined classpath. They are executed as test steps.

Instead of address the classes by their full name regular expressions can be used.

Test classes can be found if they contain a JUnit 4 Test annotation, if they extend the JUnit 3 `unit.org.TestCase` or if they contain a RunWith annotation.

You can use one of the following regular expressions:

Regular expression	Explanation
**.MainTest	All MainTest classes in all packages.
de.qfs.test.*	All test-classes from the de.qfs.test package.
de.qfs.**.*	All test classes from all sub packages of de.qfs.

Table 40.32: Possible regular expressions

Note   During the search of the test classes all classes in the given directory are loaded. The statement `**.*` loads all classes in the classpath and their static initializers. So this should be used carefully.

Variable: Yes

Restrictions: The class has to be loaded.

Classpath

Files and folders to load for the execution of the Unit Test.

Variable: Yes

Restrictions: The path has to be valid.

Injections

Injections enable working with Objects from QF-Test inside the tests.

Type	Description
String	QF-Test variables or direct values.
Component	Components of QF-Test.
WebDriver	WebDriver objects of the current browser.

Table 40.33: Injection types

The value of 'field' can be left empty. In this case the default value `instance` is used.

Variable: Yes

Restrictions: Object has to be available.

GUI engine

The GUI engine in which to execute the unit test. Only relevant for SUTs with more than one GUI engine as described in chapter 43[887].

Variable: Yes

Restrictions: See chapter 43[887]

Name

The name of a Unit test is a kind of short description. It is displayed in the tree view, so it should be concise and say something about the function of the script.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing [Alt-Return] or by clicking the 🖊 button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.11.12   Install CustomWebResolver

This node is used to install or update the CustomWebResolver.

The configuration of the Install CustomWebResolver node is described in detail in The Install CustomWebResolver node[(949)].

Contained in: All kinds of sequences[(518)].

Children: Dependency[(548)], SUT scripts[(631)], Procedure call[(589)] and Comment[(751)].

Execution: The CustomWebResolver is installed or updated according to the given configuration.

Afterwards, all contained child nodes are executed one by one. In the context of the Install CustomWebResolver node, the variables $(client) and $(guiengine) are set to the values of the attributes Client and GUI engine.

If the Install CustomWebResolver node contains a Setup node, it will be executed *before* the configuration is applied. Execution of a contained Cleanup node will be delayed until uninstallation of the CustomWebResolver.

Attributes:

Figure 40.82: Install CustomWebResolver attributes

Client

The name of the SUT client process in which to execute the CustomWebResolver.

Variable: Yes

Restrictions: Must not be empty.

YAML

The configuration instructions for the CustomWebResolver. These are described in section 49.1.2[949]. The YAML syntax described there must be followed.

Figure 40.83: CustomWebResolver configuration template actions

If the syntax is known, the YAML configuration can be edited directly. In case of an invalid configuration, corresponding error dialogs are displayed during execution or reformatting.

Variable: Yes

Restrictions: Valid syntax

Edit menu

This menu serves to simplify editing of the YAML configuration. Depending on the position in the document, it will offer different actions.

It can also be invoked at any time in the YAML editor via $\boxed{\text{Ctrl-Space}}$.

Among others, the following actions can be available:

Name	Description
Reformat data	Formats the given data in the most compact form possible. In case of an invalid configuration, a dialog with all configuration errors is displayed instead.
New entry for "..."	Creates a new mapping for the category or generic class.
Add/Remove HTML tag	Controls if the mapping is dependent on the name of the HTML tag of the element.
Add/Remove CSS class	Controls if the mapping is dependent on a CSS class of the element.
Add/Remove HTML attribute	Controls if the mapping is dependent on a HTML attribute of the element.
Change generic class	Controls the generic class that is assigned to the element.
Add/Remove ancestor	Controls if the mapping is dependent on an ancestor of the element.
Convert "..." to/Remove regular expression	Controls if the value is interpreted as a regular expression.
Show configuration errors	Displays a dialog containing a list of all problems with the current configuration. As long as the configuration is invalid, the Install CustomWebResolver can not be executed.

Table 40.34: Actions of the edit menu

Figure 40.84: CustomWebResolver edit menu

More information about the possiblities of the configuration syntax can be found in section 49.1.2[(949)].

New mapping

Clicking this button opens a list of available configuration categories. When you select an entry, an entry is created in the appropriate category. You then can replace any placeholders with the desired values.

Generic classes

Clicking on this button opens a list from which you can create a new mapping for the respective generic class. You can read about the properties assigned to each class in Generic classes[(1186)].

Script resolvers

Clicking this button opens a list from which you can select a template for one of the resolvers described in The `resolvers` module[(1016)]. The template is created as a separate SUT script[(631)] node in the CustomWebResolver node. If the text cursor is on a mapping in `genericClasses`, the resolver will be registered for the respective generic class.

Inspector

Clicking this button opens the UI inspector, see UI Inspector[(97)].

You should use the UI inspector to check your application for characteristics suitable for CustomWebResolver mappings, and to check the effect of the CustomWebResolver on the component structure of your application.

This button is available only when a web client is active.

Reformat

When clicking this button, the existing YAML code is reformatted as compactly as possible according to the syntax described in section 49.1.2[(949)]. This can also be used to detect syntax errors.

This action is also performed implicitly every time the configuration is modified e.g. through the edit menu.

Update installed CustomWebResolver during execution

If this attribute is set, then when the node is executed, the currently installed CustomWebResolver is not replaced by a new one, but the assignments of the installed CustomWebResolver are supplemented by the values specified in this node. If no CustomWebResolver was installed, then a `TestException`[(850)] is raised.

Variable: Yes

Restrictions: None

GUI engine

The GUI engine in which the CustomWebResolver is to be installed or updated. Only relevant for SUTs with more than one GUI engine as described in chapter 43[(887)].

Variable: Yes

Restrictions: See chapter 43[(887)]

Name

The name is a kind of short description. It is displayed in the tree view, so it should be concise and say something about the function of the node.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the 🖊 button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.12 HTTP Requests

This section describes how to send HTTP Request using QFTest.

40.12.1 Server HTTP request

Web

This highly specialized node sends a web request via HTTP/HTTPS directly to a web server. Such a request can be very helpful for load tests or mass data computing scenarios (e.g. filling out a form) since the simulation of user interactions and the respective loading time of the SUT are omitted during replay. The use of requests is an enhancement of the functionalities for load tests and data-driven testing described in chapter 31[(375)] and section 40.4[(562)].

If the status code returned from the server is 400 or higher, an exception is thrown. This behaviour can be changed using the Error level if status code >= 400 attribute. A detailed description of the different status codes can be found at http://www.w3.org/Protocols/HTTP/HTRESP.html. Additionally you can store the response from the server in a variable and if the attribute Add server response to run log is active the response is also written to the run log.

Contained in: All kinds of sequences[(518)].

Children: None

Execution: The web request is sent directly by QF-Test via HTTP/HTTPS to the specified URL. If the status code returned from the server is >= 400, an exception is thrown. This behaviour can be changed using Error level if status code >= 400 attribute.

Attributes:

Figure 40.85: Server HTTP request Attribute

URL

The URL to which to send the request, not including parameters. HTTP and HTTPS are acceptable values for the protocol.

Variable: Yes

Restrictions: Must not be empty.

Method

This attribute defines the method of the request, GET, POST,HEAD, PUT, DELETE, TRACE or CONNECT .

Variable: Yes

Restrictions: None

Executable parameters

Here you can specify the parameters for the request. The parameters will be URL encoded by QF-Test at execution. See section 2.2.5[17] for further information how to work with the table.

Variable: Yes

Restrictions: None

Headers

To use custom headers you can set them with this value. You can specify the name of the header and the header value. See section 2.2.5[17] for further information how to work with the table.

Variable: Yes

Restrictions: None

Additional headers

As an alternative or in addition to the headers table, this field holds additional headers in text form. This makes it easier to use variables and maybe exclude some headers. Each line holds one header in the format Header: Value.

Variable: Ja

Restrictions: Keine

Payload

For POST Methods additional payload can be attached to the request. It can be of various types like plain text, JSON or XML. To successfully add the specific format the "Content-Type" header needs to be set to the corresponding value. For more information about the "Content-Type" see: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Type.

Variable: Yes

Restrictions: None

Variable for HTTP status code

The name of the variable to which the HTTP status code is assigned (see chapter 6[(103)]).

Variable: Yes

Restrictions: None

Variable for response headers

The name of the variable to which the response headers value is assigned (see chapter 6[(103)]).

Variable: Yes

Restrictions: None

Variable for response body

The name of the variable to which the server response is assigned (see chapter 6[(103)]).

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create local or global variable bindings. If unset, the variables are bound in the global variables. If set, the topmost current binding for a variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure[(586)], Dependency[(548)] or Test case[(518)] node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test case node or, if there is no such node in the topmost node on the variables stack, falling back to the global bindings if necessary. See chapter 6[(103)] for a detailed explanation of variable binding and lookup.

In order to predefine the option use Enable 'Local variable' attribute by default[(513)].

Variable: No

Restrictions: None

Add server response to run log

If activated the server response is written to the run log in addition to the status code.

Variable: Yes

Restrictions: None

Save response to file

If set the response is written to this file. This enables QF-Test to download files.

Variable: Yes

Restrictions: QF-Test must be able to write to the file.

Error level if status code >= 400

This attribute can change the error level of requests that return with a status code greater than or equal to 400.

Variable: No

Restrictions: None

Timeout

Time limit in milliseconds until the HTTP Request must succeed. To disable the limit, leave this value empty.

Variable: Yes

Restrictions: Must not be negative.

Error level if time limit exceeded

This attribute determines what happens in case the time limit is exceeded. If set to "exception", a `CheckFailedException`[(854)] will be thrown. Otherwise a message with the respective error-level will be logged in the run log.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing [Alt-Return] or by clicking the ✎ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.12.2 Browser HTTP request

http This highly specialized node sends a GET or POST request via HTTP/HTTPS directly to a web server. Such a request can be very helpful for load tests or mass data computing scenarios (e.g. filling out a form) since the simulation of user interactions and the respective loading time of the SUT are omitted during replay. The use of requests is an enhancement of the functionalities for load tests and data-driven testing described in chapter 31[(375)] and section 40.4[(562)].

Contained in: All kinds of sequences[(518)].

Children: None

Execution: The GET/POST request is sent within the browser via HTTP/HTTPS to the specified URL. The response is shown directly in the Browser.

Attributes:

Figure 40.86: Browser HTTP request Attribute

Client

> The name of the SUT client process in which to execute the request.
>
> **Variable:** Yes
>
> **Restrictions:** Must not be empty.

QF-Test component ID

> The Web page[(818)] in which the request should be submitted.
>
> The "Select component" button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing Shift-Return

or ⌜Alt-Return⌝, when the focus is in the text field. As an alternative you can copy the target node with ⌜Ctrl-C⌝ or ‖Edit→Copy‖ and insert its QF-Test component ID into the text field by pressing ⌜Ctrl-V⌝.

This attribute supports a special format for referencing components in other test suites (see section 24.1[(298)]). Furthermore, sub-elements of nodes can be addressed directly without requiring separate nodes for them (see section 5.9[(82)]). When using SmartIDs, you can address a GUI element directly via its recognition criteria. For more information, refer to SmartID[(73)] and Component nodes versus SmartID[(46)].

Variable: Yes

Restrictions: Must not be empty.

URL

The URL to which to send the request, not including parameters. HTTP and HTTPS are acceptable values for the protocol.

Internationalized domain names (IDN) are not supported in the URL attribute as well as links to local file system starting with 'file:///'.

Variable: Yes

Restrictions: Must not be empty.

Method

This attribute defines the method of the request, GET or POST.

Variable: Yes

Restrictions: None

Executable parameters

Here you can specify the parameters for the request. The parameters will be URL encoded by QFTest at execution. See section 2.2.5[(17)] for further information how to work with the table.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[476] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.13 Windows, Components and Items

Windows[812], Components[823] and Items[829] are the foundation on which a test suite is built. They represent the windows and components of the SUT as well as the sub-items of complex components.

Many of the other node types need a window or component as a target, e.g. events[681] or checks[707]. To that end their QF-Test component ID attribute must be set to the QF-Test ID[813] of an existing Window[812], Component[823] or Item[829] node.

All of the windows and components of a test suite are collected under the Windows and components[835] node, which is always located at the bottom of the suite.

In the chapters Components[42] and the section How to achieve robust component recognition[49] in the "Best Practices" chapter you will find more information about the usage of components.

40.13.1   Window

This node is a surrogate for a window in the SUT. Events[681], checks[707] and other nodes refer to it by its QF-Test ID[813].

Contained in: Window group[832], Windows and components[835].

Children: Component group[833], Component[823].

Execution: Cannot be executed.

Attributes:

Figure 40.87: Window attributes

QF-Test ID

> This ID is the means by which other nodes refer to this window. Therefore it may
> appear in many places and you should take care to assign an ID with a meaning,
> i.e. one that is easy to remember and recognize. A QF-Test ID must be unique
> within the test suite.

Variable: No

Restrictions: Must not be empty, contain any of the characters '#', '$', '@', '&', or '%' or start with an underscore ('_').

Class name

The fully qualified name of the Java class of the window or one of its superclasses.

Variable: Yes

Restrictions: Must not be empty.

Name

The name of the window that was set by the developers of the SUT with the `setName(String)` method. See section 46.1[(902)] for why and how names should be set on all "interesting" components.

Variable: Yes

Restrictions: None

Feature

If there is no Name[(814)] available for the window, QF-Test tries to recognize it by a characteristic feature. In the case of a *Frame* or a *Dialog* this is the window's title.

See section 46.1[(902)] for more information about component recognition.

4.0+   You can select Escape text for regular expressions from the context menu for escaping special characters of regular expressions of that text.

Variable: Yes

Restrictions: None

As regexp

If this attribute is set, the Feature[(814)] is a regular expression (see section 47.3[(909)]).

Variable: Yes

Restrictions: None

Extra features

Besides the main Feature a Component can have additional features represented as name/value pairs. The kind of component determines which extra features are recorded. Each extra feature can have one of three states:

Ignore (Hint for searching: currently represented as 0)
     This extra feature is just for information. It has no influence on component recognition.

Should match (Hint for searching: currently represented as 1)
Target components matching this extra feature have a higher probability for component recognition than those that don't match it.

Must match (Hint for searching: currently represented as 2)
The target component must match this extra feature. Any component not matching it is no candidate for component recognition.

Additional columns allow for matching against a regular expression or to negate the expression, e.g. to define that the "class" attribute of a DOM node in a web page should not be "dummy". The absence of an extra feature can be enforced by adding one with an empty value. It's also allowed to use variables for those columns. You can open a textual editor via double clicking the cell and specify the respective variable then.

4.0+ You can select $\boxed{\text{Escape text for regular expressions}}$ from the context menu for escaping special characters of regular expressions of the value.

QF-Test automatically assigns some extra features to recorded components.

Name	Engine	Description
columns	Web	Column count in TABLE components.
imagehash	Swing, SWT	Shows the hash value of icons of a button or menuitem.
qfs:class	All	Dedicated component class, e.g. de.qfs.QfsTextField.
qfs:genericclass	All	Generic class of the component, e.g. TextField.
qfs:item	Web	Shows the item index of a DomNode, if it's an item of a complex GUI component. This could get recorded, if some child nodes are additionally recorded as those component might be interesting as well. This could affect content of lists, tables, tabfolders or trees.
qfs:label	All	Up to QF-Test version 6 qfs:label shows a matching label for the component, e.g. the text of a button or a label close to the respective component. If no own text or label could be found, it also tries to use tooltips or icon descriptions. From QF-Test Version 7.0 several labels for the component may be recording in an extra feature starting with "qfs:label" - the one ranking highest with the status "Should match", the others with "Ignore". For example `qfs:labelText` for the text of the component itself or `qfs:labelLeft` for a label left of the component. For more information please see `qfs:label*` variants[67].
qfs:matchindex	All	Index of components with the same name. Possibly assigned automatically when the Validate component recognition during recording[449] option is active.
qfs:modal	Web	Shows if a component of the class "Window" is modal.
qfs:originalid	Web	Shows the real 'ID' attribute specified in the DOM for that node.
qfs:systemclass	All	The toolkit-specific system class, e.g. javax.swing.JTextField.
qfs:text	All	Contains the text of the component. Not recorded by default. However, it can be used for component recognition during replay in SmartIDs or when added (manually) to Component nodes.
qfs:type	All	The generic type of the component, e.g. TextField:PasswordField.

Table 40.35: Extra features assigned by QF-Test

Variable: Yes

Restrictions: Names must not be empty

Modal

In case of a *Dialog* this attribute determines whether the dialog is modal.

Variable: Yes

Restrictions: None

Geometry

The X/Y coordinate, width and height of the window form the basis for the recognition[902] of the window. However, they play a minor roll as long as either a Name[814] was provided or a Feature[814] is available.

For windows whose location and size vary widely you should clear these attributes.

Note

If no values are specified, the recognition algorithm starts with a perfect geometry match for all candidates. To prevent false positive hits you can disable geometry matching by setting the values to a single '-' character.

Variable: Yes

Restrictions: Width and height must not be negative.

GUI engine

The GUI engine to which the Window and all its Component children belong. QF-Test records awt for AWT/Swing and swt for SWT. Only really relevant for SUTs with more than one GUI engine as described in chapter 43[887].

Variable: Yes

Restrictions: See chapter 43[887]

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ✏ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.13.2   Web page

Web

A Web page is a variant of a Window[(812)] node specifically used for testing web applications. It represents the top-level document in a Browser. Nested documents in FRAME nodes are represented as Components[(823)].

In contrast to a Window a Web page has no Class name, Modal, geometry or GUI engine attributes as these are either implicitly defined or redundant.

Contained in: Window group[(832)], Windows and components[(835)].

Children: Component group[(833)], Component[(823)].

Execution: Cannot be executed.

Attributes:

Figure 40.88: Web page attributes

QF-Test ID

> This ID is the means by which other nodes refer to this page. Therefore it may appear in many places and you should take care to assign an ID with a meaning, i.e. one that is easy to remember and recognize. A QF-Test ID must be unique within the test suite.

> **Variable:** No

> **Restrictions:** Must not be empty, contain any of the characters '#', '$', '@', '&', or '%' or start with an underscore ('_').

Name of the browser window

> This attribute can be ignored unless you need to test a web application with multiple open browser windows holding similar documents. In that case the NAME of the browser window attribute can be used to identify the browser window. The

name of a browser window can be defined via the Name of the browser window[(651)] attribute of a Open browser window[(649)] node. You find a brief description how to handle multiple browser windows in FAQ 25.

Variable: Yes

Restrictions: None

Name

A web page has no name unless one is implemented via a NameResolver. See section 52.1.7[(1023)] about the extension API for NameResolvers.

Variable: Yes

Restrictions: None

Feature

The main Feature of a web page is its URL with the parameters removed. If the option Limit URL feature of 'Web page' node to host or file[(492)] is set, the URL is further reduced to the host or file name.

See section 46.1[(902)] for more information about component recognition.

4.0+ You can select ⬚ Escape text for regular expressions ⬚ from the context menu for escaping special characters of regular expressions of that text.

Variable: Yes

Restrictions: None

As regexp

If this attribute is set, the Feature[(820)] is a regular expression (see section 47.3[(909)]).

Variable: Yes

Restrictions: None

Extra features

Besides the main Feature a Component can have additional features represented as name/value pairs. The kind of component determines which extra features are recorded. Each extra feature can have one of three states:

Ignore (Hint for searching: currently represented as 0)
This extra feature is just for information. It has no influence on component recognition.

Should match (Hint for searching: currently represented as 1)
Target components matching this extra feature have a higher probability for component recognition than those that don't match it.

Must match (Hint for searching: currently represented as 2)
>   The target component must match this extra feature. Any component not matching it is no candidate for component recognition.

Additional columns allow for matching against a regular expression or to negate the expression, e.g. to define that the "class" attribute of a DOM node in a web page should not be "dummy". The absence of an extra feature can be enforced by adding one with an empty value. It's also allowed to use variables for those columns. You can open a textual editor via double clicking the cell and specify the respective variable then.

4.0+   You can select ⟦Escape text for regular expressions⟧ from the context menu for escaping special characters of regular expressions of the value.

QF-Test automatically assigns some extra features to recorded components.

Name	Engine	Description
columns	Web	Column count in TABLE components.
imagehash	Swing, SWT	Shows the hash value of icons of a button or menuitem.
qfs:class	All	Dedicated component class, e.g. de.qfs.QfsTextField.
qfs:genericclass	All	Generic class of the component, e.g. TextField.
qfs:item	Web	Shows the item index of a DomNode, if it's an item of a complex GUI component. This could get recorded, if some child nodes are additionally recorded as those component might be interesting as well. This could affect content of lists, tables, tabfolders or trees.
qfs:label	All	Up to QF-Test version 6 qfs:label shows a matching label for the component, e.g. the text of a button or a label close to the respective component. If no own text or label could be found, it also tries to use tooltips or icon descriptions. From QF-Test Version 7.0 several labels for the component may be recording in an extra feature starting with "qfs:label" - the one ranking highest with the status "Should match", the others with "Ignore". For example `qfs:labelText` for the text of the component itself or `qfs:labelLeft` for a label left of the component. For more information please see `qfs:label*` variants[67].
qfs:matchindex	All	Index of components with the same name. Possibly assigned automatically when the <u>Validate component recognition during recording</u>[449] option is active.
qfs:modal	Web	Shows if a component of the class "Window" is modal.
qfs:originalid	Web	Shows the real 'ID' attribute specified in the DOM for that node.
qfs:systemclass	All	The toolkit-specific system class, e.g. javax.swing.JTextField.
qfs:text	All	Contains the text of the component. Not recorded by default. However, it can be used for component recognition during replay in SmartIDs or when added (manually) to Component nodes.
qfs:type	All	The generic type of the component, e.g. TextField:PasswordField.

Table 40.36: Extra features assigned by QF-Test

Variable: Yes

Restrictions: Names must not be empty

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing ⎡Alt-Return⎤ or by clicking the 🖍 button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.13.3   Component

Component nodes represent the components of the SUT. Other nodes refer to them by their QF-Test ID[(824)], similar to the way Windows[(812)] are referenced.

The Window[(812)] node that is the direct or indirect parent of the component must be the equivalent of the component's window parent in the SUT.

Contained in: Component group[(833)], Window[(812)].

Children: Item[(829)].

Execution: Cannot be executed.

Attributes:

Figure 40.89: Component attributes

QF-Test ID

> This ID is the means by which other nodes refer to this component. Therefore it may appear in many places and you should take care to assign an ID with a meaning, i.e. one that is easy to remember and recognize. A QF-Test ID must be unique within the test suite.

Variable: No

Restrictions: Must not be empty, contain any of the characters '#', '$', '@', '&', or '%' or start with an underscore ('_').

Class name

For SWT and Swing application this is the fully qualified name of the Java class of the component or one of its super-classes whereas for web applications there is a pseudo class hierarchy described in section 5.4.1[(56)].

The actual class recorded by QF-Test depends on the setting of the option Record system class only[(449)] and on potentially registered `ClassNameResolvers` (see section 52.1.9[(1026)]). Upon replay, class matching is based on the actual or pseudo class hierarchy, so you can manually change this attribute to any of the element's base classes.

Variable: Yes

Restrictions: Must not be empty.

Name

The name of the component that was set by the developers of the SUT with the `setName(String)` method. See section 46.1[(902)] for why and how names should be set on all "interesting" components.

Variable: Yes

Restrictions: None

Feature

If there is no Name[(825)] available for the component, QF-Test tries to recognize it by a characteristic feature.

See section 46.1[(902)] for more information about component recognition.

4.0+ You can select Escape text for regular expressions from the context menu for escaping special characters of regular expressions of that text.

Variable: Yes

Restrictions: None

As regexp

If this attribute is set, the Feature[(825)] is a regular expression (see section 47.3[(909)]).

Variable: Yes

Restrictions: None

Extra features

Besides the main Feature a Component can have additional features represented as name/value pairs. The kind of component determines which extra features are recorded. Each extra feature can have one of three states:

Ignore (Hint for searching: currently represented as 0)
This extra feature is just for information. It has no influence on component recognition.

Should match (Hint for searching: currently represented as 1)
Target components matching this extra feature have a higher probability for component recognition than those that don't match it.

Must match (Hint for searching: currently represented as 2)
The target component must match this extra feature. Any component not matching it is no candidate for component recognition.

Additional columns allow for matching against a regular expression or to negate the expression, e.g. to define that the "class" attribute of a DOM node in a web page should not be "dummy". The absence of an extra feature can be enforced by adding one with an empty value. It's also allowed to use variables for those columns. You can open a textual editor via double clicking the cell and specify the respective variable then.

4.0+   You can select ⎨Escape text for regular expressions⎬ from the context menu for escaping special characters of regular expressions of the value.

QF-Test automatically assigns some extra features to recorded components.

Name	Engine	Description
columns	Web	Column count in TABLE components.
imagehash	Swing, SWT	Shows the hash value of icons of a button or menuitem.
qfs:class	All	Dedicated component class, e.g. de.qfs.QfsTextField.
qfs:genericclass	All	Generic class of the component, e.g. TextField.
qfs:item	Web	Shows the item index of a DomNode, if it's an item of a complex GUI component. This could get recorded, if some child nodes are additionally recorded as those component might be interesting as well. This could affect content of lists, tables, tabfolders or trees.
qfs:label	All	Up to QF-Test version 6 qfs:label shows a matching label for the component, e.g. the text of a button or a label close to the respective component. If no own text or label could be found, it also tries to use tooltips or icon descriptions. From QF-Test Version 7.0 several labels for the component may be recording in an extra feature starting with "qfs:label" - the one ranking highest with the status "Should match", the others with "Ignore". For example `qfs:labelText` for the text of the component itself or `qfs:labelLeft` for a label left of the component. For more information please see `qfs:label*` variants[67].
qfs:matchindex	All	Index of components with the same name. Possibly assigned automatically when the Validate component recognition during recording[449] option is active.
qfs:modal	Web	Shows if a component of the class "Window" is modal.
qfs:originalid	Web	Shows the real 'ID' attribute specified in the DOM for that node.
qfs:systemclass	All	The toolkit-specific system class, e.g. javax.swing.JTextField.
qfs:text	All	Contains the text of the component. Not recorded by default. However, it can be used for component recognition during replay in SmartIDs or when added (manually) to Component nodes.
qfs:type	All	The generic type of the component, e.g. TextField:PasswordField.

Table 40.37: Extra features assigned by QF-Test

Variable: Yes

Restrictions: Names must not be empty

Structure

These two fields hold additional structural information needed for <u>component recognition</u>[(902)] during a test run. The Class count is the number of components within the same parent container and with the same class (or a class derived thereof). The Class index is the index that this component has in the list of these components with matching class. As usual the first component has index 0.

When counting the components of matching class, invisible components are considered as well. This is more robust but means that the values may be higher than expected.

Note   It is possible to specify just the Class index or the Class count attribute.

Variable: Yes

Restrictions: None

Geometry

The X/Y coordinate, width and height of the component form the basis for the <u>recognition</u>[(902)] of the component. However, they play a minor roll as long as either a <u>Name</u>[(825)] was provided or a <u>Feature</u>[(825)] or <u>structural information</u>[(828)] is available.

For components whose location or size typically vary widely at runtime, these values are not recorded.

Note   If no values are specified, the recognition algorithm starts with a perfect geometry match for all candidates. To prevent false positive hits you can disable geometry matching by setting the values to a single '-' character.

Variable: Yes

Restrictions: Width and height must not be negative.

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note   For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option <u>External editor command</u>[(431)] lets you define an external editor in which comments can be edited conveniently by

pressing [Alt-Return] or by clicking the ✎ button.

You can trigger special behaviors of some nodes using doctags, please see <u>Doctags</u>[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.13.4 Item

For complex Swing components like `JTable` or `JTree` it is possible to define Mouse events[681], checks[707] or queries[740] relative to a sub-item of the component instead of the component itself. Such a sub-item is identified with the help of an index, the Primary index[830]. This index can be given in one of three ways: as a string, a number or a regexp (see section 47.3[909]). A string or regexp designate a sub-item with a corresponding representation while a number refers to a sub-item by its index. Like in Java the first sub-item's index is 0.

The `JTable` supports an additional index to refer directly to a table cell. The Primary index[830] determines the column and the Secondary index[830] the row of the cell.

There are two representations for the nodes of a `JTree` component, flat like a list or as a hierarchy using paths. A node named `tmp` under a node named `usr` is represented as just `tmp` in the first case, as `/usr/tmp` in the latter. The option Represent tree node as path[454] determines the representation used when recording sub-items.

Currently sub-items are supported for the following Swing components:

Class	Primary	Secondary
JComboBox	List element	-
JEditorPane	Structural element (experimental)	-
JList	List element	-
JTabbedPane	Tab	-
JTable	Table column	Row
JTableHeader	Table column	-
JTextArea	Line	-
JTree	Node/Row	-

Table 40.38: Sub-items of complex Swing components

Contained in: Component[823].

Children: None.

Execution: Cannot be executed.

Attributes:

Figure 40.90: Item attributes

QF-Test ID

This ID is the means by which other nodes refer to this sub-item. Therefore it may appear in many places and you should take care to assign an ID with a meaning, i.e. one that is easy to remember and recognize. A QF-Test ID must be unique within the test suite.

Variable: No

Restrictions: Must not be empty, contain any of the characters '#', '$', '@', '&', or '%' or start with an underscore ('_').

Primary index

Designates the sub-item. Depending on whether As string[831], As number[831] or As regexp[831] is selected, the sub-item is determined by its index or by a string or regexp match.

It is OK to have an empty index, e.g. to designate a table column with an empty heading.

Variable: Yes

Restrictions: Must be a valid number or regexp if required.

Secondary index

For the `JTable` class two kinds of sub-items are supported. If only the

Primary index[(830)] is given, a whole column is referenced. An additional Secondary index designates a cell in this column.

To define a secondary index its checkbox must be selected first. This is necessary to tell an empty index from a non-existent one. An empty secondary index designates a table cell with empty content.

Variable: Yes

Restrictions: Must be a valid number or regexp if required.

As string

The Primary index[(830)] or Secondary index[(830)] is interpreted as a plain string. The sub-item is determined by matching its representation against that string.

Variable: No

Restrictions: None

As number

The Primary index[(830)] or Secondary index[(830)] is interpreted as a number. The sub-item is determined by its index.

Variable: No

Restrictions: None

As regexp

The Primary index[(830)] or Secondary index[(830)] is interpreted as a regexp (see section 47.3[(909)]). The sub-item is determined by matching its representation against that regexp.

Variable: No

Restrictions: None

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note   For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ✏ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.13.5   Window group

A Window group's only purpose is to provide structure to the Windows[(812)] of a test suite. If you have to manage a large number of Windows[(812)] you can even nest Window groups.

Contained in: Windows and components[(835)], Window group[(832)].

Children: Window group[(832)], Window[(812)].

Execution: Cannot be executed.

Attributes:

Window group

Name

Main window

QF-Test ID

Comment

The main window and its dialogs.

Figure 40.91: Window group attributes

Name

You can choose an arbitrary name. It is displayed in the tree view of the suite.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note
For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the ✎ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.13.6 Component group

A Component group's only purpose is to provide structure to the Components[823] of a Window[812]. If you have to manage a large number of Components[823] inside a Window[812] you can even nest Component groups.

Contained in: Window[812], Component group[833].

Children: Component group[833], Component[823].

Execution: Cannot be executed.

Attributes:

Figure 40.92: Component group attributes

Name

You can choose an arbitrary name. It is displayed in the tree view of the suite.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[431] lets you define an external editor in which comments can be edited conveniently by pressing Alt-Return or by clicking the button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[1215].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.13.7 Windows and components

The Windows and components node is always located at the end of the suite. It is the place where all of the Windows[(812)], Components[(823)] and Items[(829)] of the test suite are collected.

Contained in: Root node

Children: Window group[(832)], Window[(812)].

Execution: Cannot be executed.

Attributes:

Figure 40.93: Windows and components attributes

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note
For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing ⟨Alt-Return⟩ or by clicking the ✏ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.14   Deprecated nodes

The following nodes have been replaced by newer node types. The nodes can still be executed, but you shouldn't use them anymore.

40.14.1   Test

Note

Before QF-Test version 2 the Test node was one of the main building blocks of a test suite. It had a few shortcomings however: Its setup/cleanup structure was too linear and inflexible for complex scenarios and it was never clear whether a Test node represented a logical test case or was just used to implement some sequence. Thus Test nodes have been replaced with Test set[525] and Test case[518] nodes.

A Test is a special Sequence[536] that executes extra setup and cleanup code before and after the execution of its child nodes to ensure that each of the children runs under similar conditions and to prevent unwanted side effects between the execution of one child and the next. To do so, a Test has two special, optional child nodes besides its normal children, a Setup[554] as the first and a Cleanup[557] as the last node.

With its Implicitly catch exceptions[839] attribute a Test also offers special exception handling to prevent exceptions in one Test from aborting a whole test run.

For special cases of data driven testing a Test may also contain a Data driver[562], whereas such is typically done in combination with Test sets[525] as described in chapter 21[261]. That functionality can be achieved by using Test step nodes.

For backwards compatibility and to ease transition from old-style Test nodes to Test set and Test case nodes QF-Test treats nodes as a Test set or Test case for documentation and report if their place in the hierarchy allows it. In some cases Test nodes have been treated as Test step nodes, e.g. if data-driven test steps have been used.

Old test suites with a structure based on Test nodes can by migrated to make use of the new features of Test sets and Test cases. To this end, right-click on a Test node to bring up the context menu. If a transformation is allowed, QF-Test will offer to transform the Test node into a Test set, Test case or Test step node.

3.0+

It is possible to convert a whole hierarchy of Test nodes to a hierarchy of Test set and Test case nodes by selecting the recursive conversion option in the popup menu.

Both Test set and Test case nodes may contain Setup or Cleanup nodes for backwards compatibility. In a Test set, these work just as in a Test: Setup and Cleanup are executed for each test contained in the Test set. In a Test case however, Setup and Cleanup are only run once at the beginning and end of its execution. If a Test set or Test case has both a Dependency and Setup/Cleanup nodes, the Dependency will be executed first. Setup and Cleanup will have no impact on the dependency stack described in <u>section 8.6.3</u>[(140)].

Contained in: All kinds of <u>sequences</u>[(518)].

Children: Optional <u>Data driver</u>[(562)] followed by an optional <u>Setup</u>[(554)] at the beginning, then any kind of executable nodes and an optional <u>Cleanup</u>[(557)] as last node.

Execution: The <u>Variable definitions</u>[(839)] of the Test are bound. If there is a <u>Data driver</u>[(562)] node, it is executed to create a data driving context and bind one or more Data binders for iteration over the determined data sets as described in <u>chapter 21</u>[(261)]. For each of its normal child nodes, the <u>Setup</u>[(554)] is executed, then the child and then the <u>Cleanup</u>[(557)]. After the last execution of the Cleanup is complete, the variables are unbound again.

Attributes:

Figure 40.94: Test attributes

Name

> The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and say something about the function of the sequence.

> **Variable:** No

> **Restrictions:** None

Name for separate run log

> If this attribute is set it marks the node as a breaking point for split run logs and defines the filename for the partial log. When the node finishes, the respective log entry is removed from the main run log and saved as a separate, partial run

log. This operation is completely transparent, the main run log retains references to the partial logs and is fully controllable. Please see section 7.1.6[(122)] for further information about split run logs.

This attribute has no effect if the option Create split run logs[(505)] is disabled or split run logs are explicitly turned off for batch mode via the `-splitlog`[(880)] command line argument.

There is no need to keep the filename unique. If necessary, QF-Test appends a number to the filename to avoid collisions. The filename may contain directories and, similar to specifying the name of a run log in batch mode on the command line, the following placeholders can be used after a '%' or a '+' character:

Character	Replacement
%	Literal '%' character.
+	Literal '+' character.
i	The current runid as specified with `-runid <ID>`[(878)].
r	The error level of the partial log.
w	The number of warnings in the partial log.
e	The number of errors in the partial log.
x	The number of exceptions in the partial log.
t	The thread index to which the partial log belongs (for tests run with parallel threads).
y	The current year (2 digits).
Y	The current year (4 digits).
M	The current month (2 digits).
d	The current day (2 digits).
h	The current hour (2 digits).
m	The current minute (2 digits).
s	The current second (2 digits).

Table 40.39: Placeholders for the Name for separate run log attribute

Variable: Yes

Restrictions: None, characters that are illegal for a filename will be replaced with '_'.

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see chapter 6[(103)]). See section 2.2.5[(17)] about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Implicitly catch exceptions

When an exception is thrown during the execution of one of the Test's normal child nodes, the Test is usually terminated prematurely. This may not be what you want, since no information is gained from the execution of the rest of the child nodes.

If the Setup and Cleanup of the test are set up so you can guarantee the same initial conditions for each child node even in the case of an exception, you can set this attribute to make the Test catch the exception implicitly. That way, if an exception is caught from a normal child node, the exception is logged and the execution of that child node is stopped. Then the Test continues with the Cleanup as if nothing had happened.

Exceptions thrown during the execution of either the Setup or the Cleanup cannot be caught that way and will always terminate the Test

Variable: No

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run log node for a sequence will propagate, can be limited.

Note   This value has no effect on the way exceptions are handled. It only affects the error states of the run log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.7[(11)]). It also has no effect on the creation of compact run logs (see command line argument -compact[(870)]). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

Execution timeout

Time limit for the node's execution in milliseconds. If that limit expires the execution of that node will get interrupted.

Variable: Yes

Restrictions: >= 0

QF-Test ID

When using the command line argument -test <n>|<ID>[(881)] for execution in

batch mode you can specify the QF-Test ID of the node as an alternative to its qualified name.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay[(476)] from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test suite.

Note   For detailed documentation, especially for Test set, Test case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command[(431)] lets you define an external editor in which comments can be edited conveniently by pressing [Alt-Return] or by clicking the ✎ button.

You can trigger special behaviors of some nodes using doctags, please see Doctags[(1215)].

If you enter text in the comment field of a Component node, the node will be considered as 'used' when you want to mark or delete unused components.

Variable: Yes

Restrictions: None

40.14.2   Procedure `installCustomWebResolver`

Until QF-Test 7, the mapping of HTML objects to QF-Test components was done via the procedure `qfs.web.ajax.installCustomWebResolver` from the standard library `qfs.qft`. It has been replaced by the Install CustomWebResolver[(796)] node.
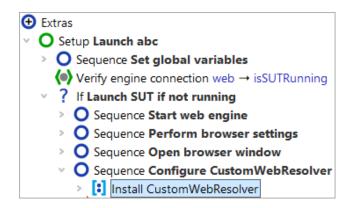
Figure 40.95: CustomWebResolver call in Setup node of the Quickstart Wizard

The sections <u>installCustomWebResolver - Parameters</u>[(842)] and <u>installCustomWebResolver - Parameter syntax</u>[(846)] explain the parameters and the syntax of the procedure.

installCustomWebResolver - Parameters

The parameters are sorted by relevance. So, for example, as component recognition is mostly based on determining QF-Test generic class names from CSS classes or other attributes the parameters `genericClasses` and `attributesToGenericClasses` come first.

---

installCustomWebResolver
Configure component recognition for web applications.

Parameters

resolver
> The short name of the resolver to extend. Either
> - `autodetect` (default) to determine the framework automatically or
> - `custom` if you do not use any of the frameworks supported by QF-Test or
> - the framework name, e.g. `zk` or `vaadin`. For the correct name please refer to the table <u>table 49.7</u>[(989)].
> If you created the start sequence via the quick start wizzard and entered a framework there, it will be shown in the parameter.

version
> The resolver version, e.g. `1` or `1.0` or `1.1.1`. The latest available version with the given restriction will be used. So, if `1.0` is given then the latest 1.0.x will apply. If empty, the latest available version will be used.
> Delete or leave empty when auto detection is used.

genericClasses

(Optional) A list of assignments mapping a css class to a generic QF-Test class. The parameter evaluates the `class` attribute of the GUI element only. The `class` attribute can hold several css classes, separated by spaces. For the mapping of a GUI element you can specify one of the css classes.

Can be overridden by `attributesToGenericClasses`.

e.g. `css-button=Button,ui-table=Table`.

Sample for a GUI element with sereral css classes:

`class="button css-button active"`. Above example makes use of the css class `css-button`.

(uses node.getAttribute(`class`)).

attributesToGenericClasses

(Optional) A list of assignments mapping GUI elements with the given attribute value to QF-Test components of the given generic class.

The mapping refers to the whole value of the attribute.

Assignments from here can override mappings done in the `genericClasses` parameter.

e.g. `id=table=Table,name=%.*combo.*=ComboBox`.

Sample for overriding the parameter `genericClasses` with several css classes:

`class=button css-button active=Button`.

tagsToGenericClasses

(Optional) A list of assignments mapping a tag to a generic QF-Test class.

Tags have to be written in capital letters,

e.g. `LI=ListItem`.

ignoreTags

(Optional) A list of class names or tags for which to ignore nodes when creating the parent hierarchy of a node. Tags have to be written in capital letters,

e.g. `DIV,TBODY`. In this example all `DIV` and `TBODY` nodes not mapped to some other class will be ignored.

ignoreByAttributes

(Optional) A list of attributes values for which to ignore nodes when creating the parent hierarchy of a node.

e.g. `id=container,id=header`.

autoIdPatterns

(Optional) A list of patterns specifying ids generated automatically by the framework. If the `id` attribute matches the pattern the node will be ignored,

e.g. `myAutoId,%auto.*`.

customIdAttributes

(Optional) A list of attribute names which can act as id for the component,

e.g. `myid,qft-id` will use the attributes `myid` and `qft-id` for id resolution.

interestingByAttributes
(Optional) A list of attribute values telling QF-Test to create a node in the component tree for the respective GUI object,
e.g. `id=container,id=header`.

attributesToQftFeature
(Optional) A list of attributes where the values will to be used for the Feature attribute of the QF-Test component.

documentJS
(Optional) Javascript code to be injected into the web page. Can be used to inject custom Javascript functions.

attributesToQftName
(Optional) A list of attributes which will be used for QF-Test name recognition of components.
Use with care. If not sure contact the QF-Test support team.

nonTrivialClasses
(Optional) A list of CSS classes of objects which shouldn't be ignored by QF-Test. Trivial nodes are I, FONT, BOLD etc. If you want to keep them, you need to activate them here specifying a proper CSS class.
Use with care. If not sure contact the QF-Test support team.

allBrowsersSemihardClasses
(Optional) A list of classes to activate semi-hard events for, e.g. `Button`, for all browsers.
Use with care. If not sure contact the QF-Test support team.
An Alternative might be setting the global `Options.OPT_WEB_SEMI_HARD_EVENTS` option to true, which works for all components.

chromeSemihardClasses
(Optional) A list of classes to activate semi-hard events for, e.g. `Button`, for Chrome.
Use with care. If not sure contact the QF-Test support team.
An Alternative might be setting the global `Options.OPT_WEB_SEMI_HARD_EVENTS` option to true, which works for all components.

ieSemihardClasses
(Optional) A list of classes to activate semi-hard events for, e.g. `Button`, for IE.
Use with care. If not sure contact the QF-Test support team.

An Alternative might be setting the global `Options.OPT_WEB_SEMI_HARD_EVENTS` option to true, which works for all components.

mozSemihardClasses

(Optional) A list of classes to activate semi-hard events for, e.g. `Button`, for Firefox.

Use with care. If not sure contact the QF-Test support team.

An Alternative might be setting the global `Options.OPT_WEB_SEMI_HARD_EVENTS` option to true, which works for all components.

edgeSemihardClasses

(Optional) A list of classes to activate semi-hard events for, e.g. `Button`, for Edge.

Use with care. If not sure contact the QF-Test support team.

An Alternative might be setting the global `Options.OPT_WEB_SEMI_HARD_EVENTS` option to true, which works for all components.

allBrowsersHardClasses

(Optional) A list of classes to activate hard events for, e.g. `Button`, for all browsers.

Use with care. If not sure contact the QF-Test support team.

An alternative might be activating "Replay as hard event" for mouse-click nodes.

chromeHardClasses

(Optional) A list of classes to activate hard events for, e.g. `Button`, for Chrome.

Use with care. If not sure contact the QF-Test support team.

An alternative might be activating "Replay as hard event" for mouse-click nodes.

ieHardClasses

(Optional) A list of classes to activate hard events for, e.g. `Button`, for IE.

Use with care. If not sure contact the QF-Test support team.

An alternative might be activating "Replay as hard event" for mouse-click nodes.

mozHardClasses

(Optional) A list of classes to activate hard events for, e.g. `Button`, for Firefox.

Use with care. If not sure contact the QF-Test support team.

An alternative might be activating "Replay as hard event" for mouse-click nodes.

edgeHardClasses

(Optional) A list of classes to activate hard events for, e.g. `Button`, for Edge.

Use with care. If not sure contact the QF-Test support team.

An alternative might be activating "Replay as hard event" for mouse-click nodes.

`installCustomWebResolver` - Parameter syntax

If a parameter can take more than one entry you need to separate the entries by commas. The comma may be followed by a line break, however, **not** by a space.

The parameter syntax consists of the following expressions, sorted by relevance:

%
> `%` denotes the following string as a regular expression.
>
> `%list.*` refers to all values starting with `list`
>
> Can be used with all parameters.

`css-class=generic class`
> Maps an HTML element with the given css class to a QF-Test component of the given generic class.
>
> `css-button=Button` maps an HTML element with the css class `css-button` to a QF-Test component of the generic class `Button`.
>
> Can be used with the parameter `genericClasses`.

`attribute=value=generic class`
> Maps an HTML element with the given attribute value to a QF-Test component with the given generic class.
>
> `role=datatable=Table` assigns the generic QF-Test class `Table` if the attribute `role` has the value `datatable`.
>
> Can be used with the parameter `attributesToGenericClasses`.

`TAG=generic class`
> Maps an HTML element with the given tag to a QF-Test component of the given generic class. Tags have to be written in capital letters.
>
> `LI=ListItem` maps the HTML element with the tag `li` to a QF-Test component with the generic class `ListItem`.
>
> Can be used with the parameter `tagsToGenericClasses`.

`@::ancestor=class name or TAG`
> Suffix to entries in the parameter lists. The entry will only be evaluated when one of the ancestors of the GUI element has the given class name or the given tag. Please use capital letters for the tag.
>
> Can be used with all parameters.
>
> Sample for parameter `tagsToGenericClasses`:

`LI=TableCell@::ancestor=TableRow` maps an HTML element with the tag `li` to a QF-Test component of the generic class `TableCell` if an ancestor has the class `TableRow`.

...=TAG

The preceding expression is only evaluated if the tag of the HTML element matches. Precedes the `@::` operator.

Tags have to be written in capital letters.

Can be used with all parameters.

Sample for parameter `genericClasses`:

`row=TableRow=SPAN` maps an HTML element with the class `row` to a QF-Test component of the generic class `TableRow` if the tag is `SPAN`.

Sample for parameter `genericClasses`:

`row=TableRow=SPAN@::ancestor=Table` maps an HTML element with the css class `row` to a QF-Test `TableRow` only if the tag is `SPAN` and if it has a some parent of the class `Table`.

Sample for parameter `interestingByAttributes`:

`myid=%.*=CONTAINER` maps HTML elements with the tag `container` only if they have the attribute `myid`.

@::parent=class name or TAG

Suffix to entries in the parameter lists. The entry will only be evaluated when the direct parent of the GUI element has the given class name or the given tag. Please use capital letters for the tag.

Can be used with all parameters.

Sample for parameter `genericClasses`:

`css-data-row=TableRow@::parent=Table`
maps the HTML element with the css class `css-data-row` to a QF-Test component of the generic class `TableRow` only if the direct parent has the class `Table`.

@::parent<level>=class name or TAG

Suffix to entries in the parameter lists. The entry will only be evaluated when the parent of the given level of the GUI element has the given class name or the given tag. Please use capital letters for the tag.

The level relies on the component structure recorded by QF-Test or the generated DomNode, so they could fail if the web-page or your resovler get changed. You should consider using the normal `@::ancestor` operator in that case or mapping a dedicated parent to a specific parent class which you can then use with `@::parent` or `@::ancestor`. The sample in CustomWebResolver − Tables[(961)] shows this technique.

Can be used with all parameters.

Sample for parameter `genericClasses`:

`css-button=Button:ComboBoxButton@::parent<3>=ComboBox`   maps the button as of type `ComboBoxButton` if the parent at level three has the class `ComboBox`.

@::ancestor<level>=class name or TAG
 Suffix to entries in the parameter lists. The entry will only be evaluated when a parent of the GUI element up to the given level has the given class name or the given tag. Please use capital letters for the tag.

Can be used with all parameters.

Sample for parameter `genericClasses`:

`cbx=CheckBox:ListItemCheckBox@::ancestor<3>=List` maps the check box as of type `ListItemCheckBox` if an ancestor within three parent levels has the class `List`. (Count of levels as with @::parent.)

attribute=value
    Can be used with the parameters `ignoreByAttributes` and `interestingByAttributes`.

With the parameter `ignoreByAttributes` the entry has the effect that no nodes will be created in the component hierarchy for HTML elements with the given attribute value.

With the parameter `interestingByAttributes` the entry has the effect that a component will be recorded for HTML elements where the attribute has the given value.

Sample for parameter `ignoreByAttributes`:

`type=container` ignores all nodes where the attribute `type` has the value `container` when creating the parent hierarchy of a node.

Sample for parameter `interestingByAttributes`:

`type=splitpane` creates a node in the parent hierarchy if the attribute `type` has the value `splitpane`.

In case you are interested:
@::ancestor=`class`   internally   uses   the   object   method
obj.getAncestorOfClass(`class`).
@::ancestor<`level`>=`class`   internally   uses   the   object   method
obj.getAncestorOfClass(`class`, level)
@::parent= internally uses the object method obj.getParent()
@::parent<`level`>= internally uses the object method obj.getNthParent(`level`)

For details of the methods please refer to Pseudo DOM API[1118].

Chapter 41

Exceptions

There are quite a lot of exceptions that can be thrown during the execution of a test. This chapter lists the exceptions in hierarchical order, shows the typical error messages and gives a short explanation.

If you want to work with those exceptions in scripts, please take a look into section 48.8[(943)].

TestException

> This is the base class of all exceptions that can be thrown during a test run. The actual exception thrown should almost always be of a derived class. A Catch[(619)] with the Exception class[(620)] set to `TestException` will catch all possible exceptions. Just like in Java you should normally not use such a Catch[(619)] since it may hide unexpected Exceptions.

ComponentNotFoundException

> This exception is thrown whenever the target component for an event[(681)] or a check[(707)] cannot be determined. Failure of a Wait for component to appear[(771)] will also cause a `ComponentNotFoundException` unless the node's Wait for absence[(774)] attribute is set.

ScopeNotFoundException

> This exception is thrown whenever the target component of an explicit scope for an event[(681)] or a check[(707)] cannot be determined. Failure of a Wait for component to appear[(771)] with an explicit scope will also cause a `ComponentNotFoundException` unless the node's Wait for absence[(774)] attribute is set.

DocumentNotLoadedException

This exception is a variant of ComponentNotFoundException and thrown specifically if a Wait for document to load[776] node fails.

PageNotFoundException

This exception is a variant of a ComponentNotFoundException and thrown specifically if a Selection[697] node fails to select a desired page in a PDF-Document.

ComponentFoundException

This is the opposite of a ComponentNotFoundException, thrown by a Wait for component to appear[771] with the Wait for absence[774] attribute set.

ModalDialogException

This exception is thrown when an event[681] is blocked by a modal dialog. See the option Check for modal dialogs[468] for details.

ComponentCannotGetFocusException

Note

This exception is obsolete and should not occur anymore.

This exception is thrown when the target component for a Key event[685] or Text input[689] is a text component that cannot get the keyboard focus for some reason. With JDK 1.4 the event cannot be delivered to the component in that case.

DisabledComponentException

This exception is thrown when the target component for a Mouse event[681], Key event[685] or Text input[689] is not enabled. In that case the event would be silently ignored, very likely leading to unexpected results during further execution of the test.

For backwards compatibility this kind of exception can be suppressed by deactivating the option Throw DisabledComponentException[469].

DisabledComponentStepException

This exception is thrown when the target component for a Mouse event[681], Key event[685] or Text input[689] is disabled under Windows and components.

BadItemException

This exception is thrown by ItemResolvers if element and item types do not match.

ExecutionTimeoutExpiredException

This exception is thrown when the execution timeout of a node has been exceeded.

BusyPaneException

This exception is thrown when the target component for a Mouse event[681], Key event[685] or Text input[689] is covered by a GlassPane with a "busy" mouse cursor. The option Wait for 'busy' GlassPane (ms)[481] determines how long QF-Test will wait for the GlassPane to disappear before the exception is actually triggered.

InvisibleDnDTargetException

This exception is thrown when the location in the target component for a Mouse event[681] of type `DRAG_FROM`, `DRAG_OVER` or `DROP_TO` for a Drag&Drop operation is invisible and cannot be made visible by scrolling the target component.

InvisibleTargetComponentException

This exception is thrown when the target component is invisible.

InvisibleTargetItemException

This exception is thrown if the surrounding target component is visible, but the target element in it is invisible.

DeadlockTimeoutException

This exception is thrown when the SUT does not react for a given amount of time which is defined in the option Deadlock detection (s)[479].

DownloadNotCompleteException

This exception is thrown, when waiting for a download that is not complete.

DownloadStillActiveException

This exception is thrown, when trying to download to a file that is still blocked by another download.

NoSuchDownloadException

This exception is thrown, when waiting for a download that was never started.

VariableException

This exception is never thrown itself but is the base class for exceptions thrown in the context of variable expansion.

BadVariableSyntaxException

This exception is thrown when a value that is to be expanded doesn't follow a proper variable syntax (e.g. no closing brace).

MissingPropertiesException

This exception is thrown when no properties or ResourceBundle are available for the group name of a property or resource looked up with ${group:prop} (see Load resources[(785)] and Load properties[(788)]).

MissingPropertyException

This exception is thrown when a property looked up with ${id:property} is not available (see Load resources[(785)] and Load properties[(788)]).

RecursiveVariableException

This exception is thrown when the expansion of a variable expression leads to recursive variable lookup, e.g. if you set the variable named x to $(y) and the variable named y to $(x) and then try to expand the value $(x).

UnboundVariableException

This exception is thrown when a variable for a value that is to be expanded doesn't exist.

VariableNumberException

    This exception is thrown when a variable expansion for a numeric attributes results in something other than a number.

BadExpressionException

    This exception is thrown when evaluating a $[...] expression fails (see section 6.6[114]).

BadTestException

    This exception is thrown when evaluating the Condition[606] of an If[605] or Elseif[609] node fails.

BadRegexpException

    This exception is thrown whenever converting a String to a regular expression (see section 47.3[909]) fails, e.g. for Item[829] or Check text[708] nodes.

BadRangeException

    This exception is thrown when the syntax of the Iteration ranges for a Data binder node has invalid syntax or specifies an index outside the valid data range.

CannotExecuteException

    This exception is thrown when execution of a process from a Start SUT client[639], Start Java SUT client[635] or Start process[642] node fails.

InvalidDirectoryException

    This exception is thrown when the Directory[640] attribute of a Start SUT client[639] node refers to a non-existent directory.

CheckFailedException

    This exception is thrown when a check[707] node with activated Throw exception on failure[712] attribute fails.

CheckNotSupportedException

As explained in the section about checks[707], each check can handle only a limited set of target components. This exception is thrown when the target component is not suitable for a check.

OperationNotSupportedException

This exception is thrown when an operation like Fetch text[740] is not supported for the designated target component.

BadComponentException

This exception is thrown, when a component for an event[681] is not suitable, i.e. a non-window for a Window event[692].

IndexFormatException

This exception is thrown when an invalid index format for a sub-item is encountered (see section 5.9.1[84]).

IndexFoundException

This exception is thrown when a sub-item is found during execution of a Wait for component to appear[771] node that looks for the absence of the item.

IndexNotFoundException

This exception is thrown when no sub-item can be located for a given index.

IndexRequiredException

This exception is thrown when no sub-item index is provided for an operation that requires one, e.g. a Check text[708] on a `JTree`.

UnexpectedIndexException

This exception is thrown when a sub-item index is provided for an operation that does not require one, e.g. a Check items[719] on a `JTree`.

ClientNotConnectedException

This exception is thrown when the target client for an operation is not connected. It differs from a `NoSuchClientException`[856] in that there is an active process for that name but no RMI connection.

CannotAttachException

This exception is thrown when attaching to a Windows application failed.

ConnectionFailureException

This exception is thrown when the connection to a client failed.

NoSuchClientException

This exception is thrown when the target client for an operation does not exist.

NoSuchEngineException

This exception is thrown when an engine is referenced that does not exist or when an attempt is made to use an engine that has not (yet) been connected to QF-Test.

DuplicateClientException

This exception is thrown if an attempt is made to run more than one client simultaneously under the same name.

UnexpectedClientException

This exception is thrown when an unexpected exception is thrown in the SUT during the replay of an event[681]. Unless it is due to a bug in QF-Test, it indicates a problem in the SUT.

ExtensionException

This exception is thrown when a client throws an unexpected exception.

ClientNotTerminatedException

This exception is thrown when a Wait for process to terminate[677] node is executed and the process doesn't terminate.

UnexpectedExitCodeException

This exception is thrown when the exit code of a terminated client doesn't match the expected value in a Wait for process to terminate[677] node's Expected exit code[679] attribute.

BadExitCodeException

This exception is thrown when the Expected exit code[(679)] attribute of a Wait for process to terminate[(677)] node doesn't match the specification and cannot be parsed.

ComponentIdMismatchException

This exception is thrown when the QF-Test component ID[(682)] attribute of a node points to a node that is not a Window[(812)], Component[(823)] or Item[(829)].

UnresolvedComponentIdException

This exception is thrown when the target of the QF-Test component ID[(682)] attribute of a node cannot be determined.

TestNotFoundException

This exception is thrown when the Test case[(518)] or Test set[(525)] for a Test call[(531)] cannot be determined.

DependencyNotFoundException

This exception is thrown when the Dependency[(548)] for a Dependency reference[(551)] cannot be determined.

InconsistentDependenciesException

This exception is thrown when the Dependency[(548)] cannot be linearized due to inconsistent references.

RecursiveDependencyReferenceException

This exception is thrown when the Dependency[(548)] cannot be linearized due to recursive references.

ProcedureNotFoundException

This exception is thrown when the Procedure[(586)] for a Procedure call[(589)] cannot be determined.

StackOverflowException

This exception is thrown when the nesting of Procedure calls[(589)] gets too deep, hinting to a problem with endless recursion. See also the option Call stack size[(457)].

UserException

This exception is thrown explicitly by a Throw[(625)] node.

CannotRethrowException

This exception is thrown when an attempt is made to rethrow an exception with a Rethrow[(627)] node but no exception was caught by a Catch[(619)] node.

ScriptException

This exception is thrown when the execution of a script from a Server script[(628)] or SUT script[(631)] fails.

AndroidSdkException

This is a base class. In order to test Android Applications, a valid Android SDK needs to be installed. In case QF-Test detects any problem with the Android SDK, an exception derived from this base class will be thrown.

InvalidAndroidSdkPathException

This exception will be thrown if the specified path does not point to a valid Android SDK installation.

AndroidSdkNotFoundException

This exception will be thrown in case when QF-Test is unable to determine the Android SDK location.

InvalidAndroidAdbPathException

This exception will be thrown in case a given ADB class path is invalid.

AndroidAdbNotFoundException

This exception will be thrown when no Android Adb was found on the computer.

InvalidAndroidEmulatorPathException

This exception will be thrown in case a given Android emulator path is invalid.

AndroidEmulatorNotFoundException

This exception will be thrown when no Android emulator was found on the computer.

AndroidVirtualDeviceException

This is a base class. In order to test Android Applications, a valid android emulator needs to be installed. The emulator can then run an android virtual device (AVD). If QF-Test is not able to find the android virtual device or if testing this device is not supported, an exception derived from this base class will be thrown.

NoAndroidVirtualDeviceException

An exception of this type will be thrown if no avd was found on which the tests may get executed.

NoSupportedAndroidVirtualDeviceException

An exception of this type will be thrown if no supported avd was found on which the tests may get executed.

AndroidVirtualDeviceNotFoundException

An exception of this type will be thrown if QF-Test was unable to find the specified avd.

AndroidVirtualDeviceNotSupportedException

An exception of this type will be thrown if QF-Test is not able to test the specified avd.

AndroidVirtualDeviceParsingException

An exception of this type will be thrown if QF-Test fails to parse the string specifying the android emulator to use.

BreakException

> This is not a standard TestException and cannot be caught by a Catch[619] node. It is thrown by a Break[604] node in order to break out of a loop. From a script, raising a `BreakException` will have the same effect. If thrown outside of a loop, a `BreakException` will cause the error below.

ReturnException

> This is not a standard TestException and cannot be caught by a Catch[619] node. It is thrown by a Return[592] node in order to return from a Procedure[586]. From a script, raising a `ReturnException` will have the same effect. If thrown outside of a Procedure, a `ReturnException` will cause the error below.

TestOutOfMemoryException

> This is a special exception that is thrown when QF-Test determines that it is running out of memory during test execution. The exception causes the test to abort immediately and cannot be caught because once QF-Test has run out of memory there is little it can do to handle it. QF-Test tries to keep a little reserve memory so it will at least try to save the run log.

Part IV

Technical reference

Chapter 42

Command line arguments and exit codes

42.1 Call syntax

The call syntax for interactive and batch mode varies widely since some command line arguments are specific to interactive mode or batch mode or even sub-modes of batch mode. Note that all of the arguments have sensible default values which you only need to override for special cases. In most cases you'll only need to execute either `qftest [<suite> | <run log>]*` to run QF-Test in interactive mode, or `qftest -batch [-runlog [<file>]] [-report <directory>] <suite>` to execute a test in batch mode.

5.2+ For maximum flexibility the names of all QF-Test arguments are case-insensitive and embedded '-', '_', '.' and ':' characters are ignored, so `-report.html` is equivalent to `-reportHtml` or `-report-html`. The latter is the officially documented form because it avoids conflicts with Windows PowerShell.

Windows The program `qftest.exe` is a Windows GUI application. When started from a command shell, it will not wait for QF-Test to terminate but return immediately. Thus, when executing a test in batch mode, you cannot see whether QF-Test has finished or not (you may put the command into a `.bat` file to deal with this behaviour). Furthermore you won't see any output from QF-Test in the console window when using `qftest.exe`. For both reasons you may prefer to utilize the `qftestc.exe` Console application when launching QF-Test from a command shell: It waits for QF-Test to terminate and print output from <u>Server scripts</u>⁽⁶²⁸⁾ will be displayed in the console window. Apart from that, everything said about `qftest.exe` in this chapter holds true for `qftestc.exe` too.

Mac In case the macOS App is used those parameters can be defined directly in QF-Test via Edit→Options under General->Startup (please also see the macOS specific note

under Starting QF-Test[11]).

Interactive mode

The full call syntax for interactive mode is:

```
qftest [-dbg(867)] [-java <executable> (deprecated)(868)]
[-noconsole(868)] [-J<java-argument>]*
[-allow-shutdown [<shutdown ID>](868)] [-daemon(870)]
[-daemonhost <host>(871)] [-daemonport <port>(871)]
[-daemonrmiport <port>(871)] [-dontkillprocesses(871)]
[-engine <engine>(871)] [-groovydir <directory>(872)]
[-help(872)] [-ipv6(872)] [-javascriptdir <directory>(873)]
[-jythondir <directory>(873)] [-jythonport <number>(873)]
[-keybindings <value>(873)] [-keystore <keystore file>(873)]
[-keypass <keystore password>(873)] [-libpath <path>(873)]
[-license <file>(873)] [-license-waitfor <seconds>(873)]
[-logdir <directory>(874)] [-noupdatecheck(875)]
[-option <name>=<value>(875)] [-options <file>(875)]
[-plugindir <directory>(876)] [-port <number>(876)] [-reuse(878)]
[-run(878)] [-runlogdir <directory>(879)] [-runtime(879)]
[-serverhost <host>(880)] [-shell <executable>(879)]
[-shellarg <argument>(879)] [-suitesfile <file>(881)]
[-systemcfg <file>(881)] [-systemdir <directory>(881)]
[-tempdir <directory>(881)] [-test <n>|<ID>(881)]*
[-usercfg <file>(883)] [-userdir <directory>(883)]
[-variable <name>=<value>(883)]* [-version(883)] [<suite> | <run
log>]*
```

There are several sub-modes for running QF-Test in batch mode. The default is to execute one or more test suites. Alternatively QF-Test can be invoked to create test documentation from test suites or reports from run logs. QF-Test can also be run in daemon mode where it sits in the background waiting for calls from the outside telling it what to do (see chapter 53[1139] for further information about the daemon mode). Finally, showing help or version information can also be seen as separate sub-modes.

Test execution

To execute one or more test suites and create a run log and/or report as a result, use: qftest -batch [-run(878)] [-dbg(867)]
```
[-java <executable> (deprecated)(868)] [-noconsole(868)]
[-J<java-argument>]* [-allow-shutdown [<shutdown ID>](868)]
[-clearglobals(870)] [-compact(870)] [-engine <engine>(871)]
[-exitcode-ignore-exception(871)]
[-exitcode-ignore-error(871)] [-exitcode-ignore-warning(872)]
[-groovydir <directory>(872)] [-ipv6(872)]
```

[-javascriptdir <directory>[(873)]] [-jythondir <directory>[(873)]]
[-jythonport <number>[(873)]] [-keystore <keystore file>[(873)]]
[-keypass <keystore password>[(873)]] [-libpath <path>[(873)]]
[-license <file>[(873)]] [-license-waitfor <seconds>[(873)]]
[-logdir <directory>[(874)]] [-nolog[(874)]]
[-nomessagewindow[(874)]] [-option <name>=<value>[(875)]]
[-options <file>[(875)]] [-plugindir <directory>[(876)]]
[-port <number>[(876)]] [-report <directory>[(876)]]
[-report-checks[(876)]] [-report-customdir <directory>[(876)]]
[-report-doctags[(876)]] [-report-errors[(877)]]
[-report-exceptions[(877)]] [-report-html <directory>[(877)]]
[-report-ignorenotimplemented[(877)]] [-report-ignoreskipped[(877)]]
[-report-junit <directory>[(877)]] [-report-name <name>[(877)]]
[-report-nodeicons[(877)]] [-report-passhtml[(877)]]
[-report-piechart[(877)]] [-report-include-suitename[(877)]]
[-report-scale-thumbnails <percent>[(878)]]
[-report-teststeps[(878)]] [-report-thumbnails[(878)]]
[-report-warnings[(878)]] [-report-xml <directory>[(878)]]
[-runid <ID>[(878)]] [-runlogdir <directory>[(879)]]
[-runlog [<file>][(878)]] [-runtime[(879)]] [-serverhost <host>[(880)]]
[-shell <executable>[(879)]] [-shellarg <argument>[(879)]]
[-sourcedir <directory>[(880)]] [-suitesfile <file>[(881)]]
[-splitlog[(880)]] [-systemcfg <file>[(881)]]
[-systemdir <directory>[(881)]] [-test <n>|<ID>[(881)]]*
[-threads <number>[(883)]] [-userdir <directory>[(883)]]
[-variable <name>=<value>[(883)]]* [-verbose [<level>][(883)]]
<suite>+

Test execution via QF-Test daemon
The following parameters can be specified when executing a test case by calling a daemon: qftest -batch -calldaemon[(870)]
[-cleanup[(870)]] [-clearglobals[(870)]] [-dbg[(867)]]
[-java <executable> (deprecated)[(868)]] [-noconsole[(868)]]
[-J<java-argument>]* [-daemonhost <host>[(871)]]
[-daemonport <port>[(871)]] [-exitcode-ignore-exception[(871)]]
[-exitcode-ignore-error[(871)]] [-exitcode-ignore-warning[(872)]]
[-ipv6[(872)]] [-keystore <keystore file>[(873)]]
[-keypass <keystore password>[(873)]] [-nomessagewindow[(874)]]
[-ping[(875)]] [-option <name>=<value>[(875)]] [-options <file>[(875)]]
[-runid <ID>[(878)]] [-runlogdir <directory>[(879)]]
[-runlog [<file>][(878)]] [-startclean[(880)]] [-startsut[(880)]]
[-stopclean[(880)]] [-stoprun[(880)]] [-systemdir <directory>[(881)]]
[-suitedir <dir>[(880)]] [-systemdir <directory>[(881)]]

[-terminate⁽⁸⁸¹⁾] [-timeout <milliseconds>⁽⁸⁸³⁾]
[-userdir <directory>⁽⁸⁸³⁾] [-variable <name>=<value>⁽⁸⁸³⁾]*
[-verbose [<level>]⁽⁸⁸³⁾] <suite#test case>

Change the XML format of existing test-suites

qftest -batch -convertxml⁽⁸⁷⁰⁾
[-convertxml-indent <number>⁽⁸⁷⁰⁾]
[-convertxml-linelength <number>⁽⁸⁷⁰⁾]
[-convertxml-utf8 <true|false>⁽⁸⁷⁰⁾] (<suite> | <directory>)+

Create test documentation

Package or test case documentation can be create for one or more test suites or whole directories. This is described further in chapter 22⁽²⁷¹⁾. The command line syntax is: qftest -batch -gendoc⁽⁸⁷²⁾
[-dbg⁽⁸⁶⁷⁾] [-java <executable> (deprecated)⁽⁸⁶⁸⁾]
[-noconsole⁽⁸⁶⁸⁾] [-J<java-argument>]* [-license <file>⁽⁸⁷³⁾]
[-license-waitfor <seconds>⁽⁸⁷³⁾] [-nomessagewindow⁽⁸⁷⁴⁾]
[-option <name>=<value>⁽⁸⁷⁵⁾] [-options <file>⁽⁸⁷⁵⁾]
[-pkgdoc <directory>⁽⁸⁷⁵⁾] [-pkgdoc-dependencies⁽⁸⁷⁵⁾]
[-pkgdoc-doctags⁽⁸⁷⁵⁾] [-pkgdoc-html <directory>⁽⁸⁷⁵⁾]
[-pkgdoc-includelocal⁽⁸⁷⁵⁾] [-pkgdoc-nodeicons⁽⁸⁷⁵⁾]
[-pkgdoc-passhtml⁽⁸⁷⁵⁾] [-pkgdoc-sortpackages⁽⁸⁷⁶⁾]
[-pkgdoc-sortprocedures⁽⁸⁷⁶⁾] [-pkgdoc-xml <directory>⁽⁸⁷⁶⁾]
[-sourcedir <directory>⁽⁸⁸⁰⁾] [-systemdir <directory>⁽⁸⁸¹⁾]
[-testdoc <directory>⁽⁸⁸²⁾] [-testdoc-doctags⁽⁸⁸²⁾]
[-testdoc-followcalls⁽⁸⁸²⁾] [-testdoc-html <directory>⁽⁸⁸²⁾]
[-testdoc-nodeicons⁽⁸⁸²⁾] [-testdoc-passhtml⁽⁸⁸²⁾]
[-testdoc-sorttestcases⁽⁸⁸²⁾] [-testdoc-sorttestsets⁽⁸⁸²⁾]
[-testdoc-teststeps⁽⁸⁸²⁾] [-testdoc-xml <directory>⁽⁸⁸³⁾]
(<suite> | <directory>)+

Create a report from run logs

To create a report from one or more run logs or whole directories use:
qftest -batch -genreport⁽⁸⁷²⁾ [-dbg⁽⁸⁶⁷⁾] [-java <executable>
(deprecated)⁽⁸⁶⁸⁾] [-noconsole⁽⁸⁶⁸⁾] [-J<java-argument>]*
[-license <file>⁽⁸⁷³⁾] [-license-waitfor <seconds>⁽⁸⁷³⁾]
[-nomessagewindow⁽⁸⁷⁴⁾] [-option <name>=<value>⁽⁸⁷⁵⁾]
[-options <file>⁽⁸⁷⁵⁾] [-report <directory>⁽⁸⁷⁶⁾]
[-report-checks⁽⁸⁷⁶⁾] [-report-customdir <directory>⁽⁸⁷⁶⁾]
[-report-doctags⁽⁸⁷⁶⁾] [-report-errors⁽⁸⁷⁷⁾]
[-report-exceptions⁽⁸⁷⁷⁾] [-report-html <directory>⁽⁸⁷⁷⁾]
[-report-ignorenotimplemented⁽⁸⁷⁷⁾] [-report-ignoreskipped⁽⁸⁷⁷⁾]
[-report-junit <directory>⁽⁸⁷⁷⁾] [-report-name <name>⁽⁸⁷⁷⁾]
[-report-nodeicons⁽⁸⁷⁷⁾] [-report-passhtml⁽⁸⁷⁷⁾]

[-report-piechart[(877)]] [-report-include-suitename[(877)]]
[-report-scale-thumbnails <percent>[(878)]]
[-report-teststeps[(878)]] [-report-thumbnails[(878)]]
[-report-warnings[(878)]] [-report-xml <directory>[(878)]]
[-runlogdir <directory>[(879)]] (<run log> | <directory>)+

Daemon mode

To run QF-Test in daemon mode as described chapter 53[(1139)] use: qftest -batch -daemon[(870)] [-dbg[(867)]]
[-java <executable> (deprecated)[(868)]] [-noconsole[(868)]]
[-J<java-argument>]* [-daemonhost <host>[(871)]]
[-daemonport <port>[(871)]] [-daemonrmiport <port>[(871)]]
[-engine <engine>[(871)]] [-groovydir <directory>[(872)]] [-ipv6[(872)]]
[-javascriptdir <directory>[(873)]] [-jythondir <directory>[(873)]]
[-jythonport <number>[(873)]] [-keystore <keystore file>[(873)]]
[-keypass <keystore password>[(873)]] [-libpath <path>[(873)]]
[-license <file>[(873)]] [-license-waitfor <seconds>[(873)]]
[-logdir <directory>[(874)]] [-nolog[(874)]] [-nomessagewindow[(874)]]
[-option <name>=<value>[(875)]] [-options <file>[(875)]]
[-plugindir <directory>[(876)]] [-port <number>[(876)]] [-runtime[(879)]]
[-serverhost <host>[(880)]] [-shell <executable>[(879)]]
[-shellarg <argument>[(879)]] [-splitlog[(880)]]
[-systemcfg <file>[(881)]] [-systemdir <directory>[(881)]]
[-userdir <directory>[(883)]] [-variable <name>=<value>[(883)]]*

Import one test suite into another

qftest -batch -import[(872)] [-import-from <test suite>[(872)]]
[-import-into <test suite>[(872)]] [-import-components[(872)]]
[-import-procedures[(872)]] [-import-tests[(872)]]

Analyze references of a test suite

qftest -batch -analyze[(869)] [-analyze-target <directory>[(869)]]
[-suitedir <dir>[(880)]] [-analyze-references[(869)]]
[-analyze-duplicates[(869)]] [-analyze-invalidchar[(869)]]
[-analyze-emptynodes[(869)]] [-analyze-components[(869)]]
[-analyze-procedures[(869)]] [-analyze-dependencies[(869)]]
[-analyze-tests[(869)]] [-analyze-packages[(869)]]
[-remove-unused-callables[(869)]] [-remove-unused-components[(869)]]
[-analyze-transitive[(869)]] [-analyze-followincludes[(870)]]
(<suite> | <directory>)+

Merging run logs

qftest -batch -mergelogs[(874)] [-mergelogs-mode [<mode>][(874)]]
[-mergelogs-usefqn[(874)]] [-mergelogs-resultlog [<file>][(874)]]

[`-mergelogs-masterlog [<file>]`[(874)]] (`<run log>` |
`<directory>`)+

Get version information
qftest -batch `-version`[(883)]

Cleanly terminate one specific QF-Test instance running on the local system (see `-allow-shutdown [<shutdown ID>]`[(868)])
qftest -batch `-shutdown <ID>`[(879)] 4711

Pause the current test run on the local system
qftest -batch `-interrupt-running-instances`[(872)]
[`-timeout <milliseconds>`[(883)]]

Compress images in an existing test suite
qftest -batch `-compress`[(870)] `<suite>`+

Get help
qftest -batch `-help`[(872)]

42.2   Command line arguments

Command line arguments for QF-Test fall in three categories. They can be mixed freely.

42.2.1   Arguments for the starter script

These arguments are evaluated directly by the `qftest` shell script or executable and override settings determined during installation. On Unix these settings are stored in the file `launcher.cfg` in QF-Test's system directory, on Windows the file is called `launcherwin.cfg`.

-batch
Run QF-Test in batch mode. This causes QF-Test to load and execute a test suite directly and finish with an exit code that represents the result of the test run.

-dbg
Turn on debugging output for the starter script. The same effect is achieved by setting the environment variable `QFTEST_DEBUG` to a non-empty value. On Windows this causes QF-Test to open a console window to display the output that would otherwise be invisible unless the argument `-noconsole`[(868)] is also given. This also turns on debugging output for the `qfclient` and `java` helper programs when using the old connection mechanism (see chapter 44[(889)]).

–java <executable> (deprecated)
>   The Java executable used to run QF-Test. The default is `java` on Unix and `javaw.exe` on Windows, unless a different value was set during installation. This argument will be removed in a future version of QF-Test.

–noconsole (Windows only)
>   On Windows this argument suppresses the console window that would otherwise be opened in case `–dbg`[867] is specified.

42.2.2   Arguments for the Java VM

You can pass arguments to the Java VM through the starter script by prepending them with `–J`, e.g. `–J-Duser.language=en` to set a system property. To set the classpath, prepend `–J` only to the `–cp` or `–classpath` argument, not to the actual value, e.g. `–J-classpath myclasses.jar`. When setting the classpath this way, QF-Test's own jar archives need not to be taken into account.

42.2.3   Arguments for QF-Test

The rest of the arguments are handled by QF-Test itself when it is executed by the Java virtual machine. These arguments can also be placed in a file using the syntax `<name>=<value>` for arguments with parameters or `<name>=true` or `<name>=false` to turn a simple argument on or off. By default this file is called `qftest.options`, located in the `bin` directory of QF-Test and used only for internal logging purposes. If you change anything in this file, you can move it to QF-Test's system directory so your changes will still apply after upgrading. Alternatively you can use the `–options <file>`[875] argument to specify a different file. Arguments given on the command line override arguments from an option file except for those which can be given multiple times. In the latter case the arguments are merged.

–allowkilling
>   Deprecated, use `–allow-shutdown` without shutdown ID instead.

–allow-shutdown [<shutdown ID>]
>   Explicitly specifies that this QF-Test instance can be cleanly terminated via a batch call with the `–shutdown <ID>`[879] argument. An optional string argument can be provided as shutdown ID which allows selective process termination without knowing the process ID. The shutdown ID must contain at least one non-numeric character to be distinguishable from the numerical process ID. If `–allow-shutdown [<shutdown ID>]`[868] has not been specified on the command line, a running QF-Test process can only be terminated cleanly by

means of the process ID. The argument `-allow-shutdown false` prohibits all clean termination of the QF-Test instance, even if `-shutdown <ID>`[(879)] is used with the correct process ID.

`-analyze` (batch mode only)
> Run the static validation of test suites. Analyzing results are stored to a given file.

`-analyze-target <directory>` (batch mode only)
> The target folder, where the result file should be created.

`-analyze-references` (batch mode only)
> Switch for analyzing references the given test suite.

`-analyze-duplicates` (batch mode only)
> Switch for analyzing duplicates of the given test suite.

`-analyze-invalidchar` (batch mode only)
> Checks the given test suite for invalid characters in node names.

`-analyze-emptynodes` (batch mode only)
> Checks the given test suite for empty nodes.

`-analyze-components` (batch mode only)
> Switch for analyzing components of the given test suite.

`-analyze-dependencies` (batch mode only)
> Switch for analyzing dependency references of the given test suite.

`-analyze-procedures` (batch mode only)
> Switch for analyzing procedure calls of the given test suite.

`-analyze-tests` (batch mode only)
> Switch for analyzing test calls of the given test suite.

`-analyze-packages` (batch mode only)
> Switch for analyzing packages of the given test suite.

`-remove-unused-callables` (batch mode only)
> Switch to remove unused callable nodes.

`-remove-unused-components` (batch mode only)
> Switch to remove unused components.

`-analyze-transitive` (batch mode only)
> Switch for analyzing references and calls transitively, i.e. follow the calls and analyze its content also.

–analyze–followincludes (batch mode only)
Switch for analyzing all included test suites of the given test suite.

–calldaemon (batch mode only)
Connect to a running QF-Test daemon to execute a test case.

–cleanup (calldaemon mode only)
With this argument all TestRunDaemons belonging to a daemon are cleaned up and all clients killed before running the test.

–clearglobals (batch mode and calldaemon mode only)
If more than one test suite is specified for batch execution, clear global variables and resources before the execution of each test suite. Can also be used in calldaemon mode to clear globals before assigning variables from the command line and starting the test and, if used in combination with $-\texttt{stopclean}^{(880)}$, to clear globals after the test.

–compact (batch mode only)
Create a compact run log which retains only those branches and nodes that contain warnings, errors, exceptions or other information that is relevant to the report. This is equivalent to activating the option Create compact run log[510] in interactive mode. Ignored if log file creation is suppressed with $-\texttt{nolog}^{(874)}$.

–compress (batch mode only)
Loss-free compresses the images in an existing test suite.

–convertxml (batch mode only)
Run in batch mode to automatically convert the XML file format of test suites according to the following arguments or options.

–convertxml–indent <number> (convertxml mode only)
Number of blank characters to use per level of indentation. If not specified the value of the option Number of blanks for indentation when saving test suites[424] is used.

–convertxml–linelength <number> (convertxml mode only)
Maximum length of lines containing XML attributes. If not specified the value of the option Line length for saving test suites[424] is used.

–convertxml–utf8 <true|false> (convertxml mode only)
Whether to save the converted suites with UTF-8 (value true) or ISO-8859-1 encoding (value false). If not specified the value of the option Use UTF-8 encoding for saving test suites[424] is used.

–daemon
Run QF-Test in daemon mode. Further information is provided in chapter 53[1139].

–daemonhost <host> (daemon or calldaemon mode only)

   In calldaemon mode, specify the host where to locate the QF-Test daemon. The default is localhost. When starting a daemon, either in interactive or in batch mode, this parameter defines the hostname or IP address that daemon objects use at RMI level. The default in this case is chosen by Java, typically the IP address of the primary local network interface.

–daemonport <port>

   Specify the registry port for the QF-Test daemon to listen on and to connect to respectively. The default is 3543 or the port defined with –port <number>[(876)].

–daemonrmiport <port>

   Specify the port that the QF-Test daemon should use for RMI communication. Useful only when running the daemon behind a firewall. When running unprotected without SSL (see section 53.3[(1154)]) this can be identical to the daemon port specified with –daemonport <port>[(871)]. If using SSL, two different ports are required.

–dontkillprocesses (batch mode only)

   When finishing batch execution, don't explicitly kill processes started by QF-Test as part of the tests. However, whether or not a sub-process of QF-Test survives QF-Test's exit is system-dependent.

–engine <engine>

   Specify which engine license(s) to use. This option is only useful in case the QF-Test license contains a mix of GUI engine(s) with different numbers of engine licenses. In that case it may be necessary to specify the engine license(s) to use in order to prevent license conflicts with colleagues using the same license. Possible values are "all" to use all supported licenses, "ask" to bring up a dialog for engine selection or any combination of "awt" for AWT/Swing, "fx" for JavaFX, "swt" for SWT or "web" for Web, e.g. "awt,web". This is explained in more detail in section 39.1.9[(437)].

–exitcode-ignore-exception (batch and calldaemon mode)

   For exit code calculation exceptions, errors and warnings are ignored. This means if exceptions, errors or warnings occurred during a test run the exit code will be 0. This option is helpful when integrating QF-Test with build tools that rate a build as failed dependent on the the exit code.

–exitcode-ignore-error (batch and calldaemon mode)

   For exit code calculation errors and warnings are ignored. This means if just errors and warnings occurred during a test run the exit code will be 0. This option is helpful when integrating QF-Test with build tools that rate a build as failed dependent on the the exit code.

–exitcode–ignore–warning (batch and calldaemon mode)
> For exit code calculation warnings are ignored. This means if just warnings occurred during a test run the exit code will be 0. This option is helpful when integrating QF-Test with build tools that rate a build as failed dependent on the the exit code.

–gendoc (batch mode only)
> Tell QF-Test that this batch run serves to create test documentation from test suites.

–genreport (batch mode only)
> Tell QF-Test that this batch run serves to create a report from run logs.

–groovydir <directory>
> This argument overrides the default location of the directory for additional Groovy modules. The default directory called groovy is located under QF-Test's system directory.

–help
> Show help about available command line arguments.

–import (batch mode only)
> Import a given test suite into another one. This mode can be used to merge two test suites.

–import–from <test suite> (batch mode only)
> The source test suite, which should be imported into another one.

–import–into <test suite> (batch mode only)
> The target tests-suite for importing.

–import–components (batch mode only)
> Switch for merging components of two given test suites.

–import–procedures (batch mode only)
> Switch for merging packages and procedures of two given test suites.

–import–tests (batch mode only)
> Switch for merging test cases and test sets of two given test suites.

–interrupt–running–instances (batch mode only)
> Interrupt a test run on the current system and brings up a dialog which allows to pause or to interrupt the current test run.

–ipv6
> QF-Test uses only IPv4 communication, so by default IPv6 support is disabled at Java level which can reduce startup time significantly. In case you need to enable

IPv6 support in QF-Test, for example for use within a plugin, you can use this argument.

-javascriptdir <directory>
   This argument overrides the default location of the directory for additional JavaScript modules. The default directory called `javascript` is located under QF-Test's system directory.

-jythondir <directory>
   This argument overrides the default location of the directory for additional Jython modules. The default directory called `jython` is located under QF-Test's system directory.

-jythonport <number>
   Tell the embedded Jython interpreter to listen for TCP connections at the specified port. You can then use `telnet` to connect to that port and get an interactive Jython command line.

-keybindings <value> (interactive mode only)
   Currently used only for macOS to switch between the new default Mac bindings (value system) or the old, Windows oriented QF-Test bindings (value classic).

-keystore <keystore file>
   An alternative keystore file to use for securing daemon communication with SSL. See section 53.3[(1154)] for details. To disable SSL by specifying no keystore, use this argument in the form `-keystore=`.

-keypass <keystore password>
   The password for the keystore file used for securing daemon communication with SSL. See section 53.3[(1154)] for details.

-kill-kunning-instances
   Deprecated, use `-shutdown all` instead.

-libpath <path>
   Override the library path option (Directories holding test suite libraries[(436)]). The directories of the library path should be separated by the standard path separator character for the system, i.e. ';' for Windows and ':' for Unix. QF-Test's `include` directory will automatically be appended to the path.

-license <file>
   Set the location of the license file (see section 1.5[(9)]).

-license-waitfor <seconds>;
   Specify an interval in seconds to wait during QF-Test startup in case a license is not immediately available. This timeout is also in effect when renewing a lease from the QF-Test license server and the server is temporarily unavailable.

–logdir <directory>
> This argument overrides the default location of the directory into which QF-Test saves its internal log files. The default log directory is called `log` and located under QF-Test's system directory.

–mergelogs (batch mode only)
> Tell QF-Test that this batch run serves to merge several run logs. You can find a detailed description at section 7.1.9[124].

–mergelogs–masterlog [<file>] (batch mode only)
> The path to the run log which will act as master run log for log mering, if the run log should be patched with newer results. Use this switch, if you want to replace individual test cases in that run log with results from a rerun.

–mergelogs–mode [<mode>] (batch mode only)
> Specifies the mode how run logs should be merged based in the main run log, specified with the `–mergelogs–masterlog [<file>]`[874] switch. You can specify the modes "replace", "merge" and "append". "replace" takes the new results and overwrites the existing test cases from the main run log. "merge" adds the new test cases to main run log and "append" simply adds the new run log to the main run log.

–mergelogs–resultlog [<file>] (batch mode only)
> The path to the run log which will contain the merged results of the master run log and the new run logs with updates. So, this will be the new and clean run log file.

–mergelogs–usefqn (batch mode only)
> Use that switch, if the full qualified name of test cases should be used in case of merging run logs. Otherwise only the name of test cases will be used without test set names.

–nolog (batch mode only)
> Suppress the automatic creation of a run log. If any of `–runlog [<file>]`[878], `–report <directory>`[876], `–report–html <directory>`[877], `–report–xml <directory>`[878] or `–report–junit <directory>`[877] is given, this argument is ignored. This option is retained for backwards compatibility only. To keep memory use manageable, split run logs should be used instead (see `–splitlog`[880]).

–nomessagewindow (batch mode only)
> In case of a fatal error in batch mode QF-Test prints an error message to the console and for improved visibility also brings up an error dialog for about 30 seconds. That dialog can be suppressed with the help of this argument. Batch

commands that don't require an actual display, i.e. all batch commands that do
not execute tests, will run in AWT headless mode if this argument is specified.

–noupdatecheck

Using this argument disables the automatic update check. This overrides the
update options (see section 39.1.10[438]).

–option <name>=<value>

Specifies options. `-option <name>=<value>` sets a value of the option with a
name `<name>` to `<value>`. This argument can be given more than once to set
several options.

–options <file>

Override the location of the file used to specify additional command line
arguments. This argument can be given more than once to use several sources
of command line arguments.

–ping (calldaemon mode only)

Use this option, if you want to check whether a daemon is up and running.

–pkgdoc <directory> (batch mode only)

With this argument QF-Test creates both HTML and XML pkgdoc documentation.
If no directory is given, it is created from the basename of the suite.

–pkgdoc–dependencies (batch mode only)

Whether to list dependencies when creating the pkgdoc documentation. Default
is true, use `-pkgdoc-dependencies=false` to disable.

–pkgdoc–doctags (batch mode only)

Whether to use the QFS doctag extensions when creating the pkgdoc
documentation. Default is true, use `-pkgdoc-doctags=false` to disable.

–pkgdoc–html <directory> (batch mode only)

With this argument QF-Test creates HTML pkgdoc documentation. If no directory
is given, it is created from the basename of the suite.

–pkgdoc–includelocal (batch mode only)

Whether to include local packages and procedures (those whose names begin
with an '_'). Default is false.

–pkgdoc–nodeicons (batch mode only)

Whether to show icons for nodes in the pkgdoc documentation. Default is true,
use `-pkgdoc-nodeicons=false` to disable.

–pkgdoc–passhtml (batch mode only)

Whether to pass HTML tags in comments through to the HTML pkgdoc. Default
is true, use `-pkgdoc-passhtml=false` to disable.

–pkgdoc–sortpackages (batch mode only)
> Whether to sort packages alphabetically. Default is true, use `–pkgdoc–sortpackages=false` to disable.

–pkgdoc–sortprocedures (batch mode only)
> Whether to sort procedures alphabetically. Default is true, use `–pkgdoc–sortprocedures=false` to disable.

–pkgdoc–splitparagraph (batch mode only)
> Specifies whether comments are splitted into paragraphs by using empty lines. Default is true, with `–pkgdoc–splitparagraph=false` it is possible to disable this option.

–pkgdoc–stylesheet <file> (batch mode only)
> Optional XSLT Stylesheet for the second step of the transformation.

–pkgdoc–xml <directory> (batch mode only)
> With this argument QF-Test creates XML pkgdoc documentation. If no directory is given, it is created from the basename of the suite.

–plugindir <directory>
> This argument overrides the default location of the directory for plugins, jar files that should be made accessible to scripts. The default directory called `plugin` is located under QF-Test's system directory. See section 48.2[(916)] for more information about plugins.

–port <number>
> The TCP port on which QF-Test communicates with the SUT. By default QF-Test uses an arbitrary dynamic port where it creates its own RMI registry. A specific port should only be requested if it must be hard-coded when starting the SUT.

–report <directory> (batch mode only)
> Create a combined XML/HTML report. The directory name may contain placeholders as explained in section 42.2.4[(884)].

–report–checks (batch mode only)
> Whether to list checks in the report. Default is false. Please note: the argument refers only to checks with default result handling, i.e. just logging to the run log, not setting a variable or throwing an exception. For more information please see section 22.1.2[(273)].

–report–customdir <directory> (batch mode only)
> Directory containing css stylesheets and icons for custom reports.

–report–doctags (batch mode only)
> Whether to use the QFS doctag extensions when creating the report. Default is true, use `–report–doctags=false` to disable.

–report–errors (batch mode only)
> Whether to list errors in the report. Default is true, use `–report–errors=false` to disable.

–report–exceptions (batch mode only)
> Whether to list exceptions in the report. Default is true, use `–report–exceptions=false` to disable.

–report–html <directory> (batch mode only)
> Create an HTML report. The directory name may contain placeholders as explained in section 42.2.4[(884)].

–report–ignorenotimplemented (batch mode only)
> Whether to ignore nodes that are not implemented in the report in which case the legend and respective columns for not implemented tests are also not shown. Default is false.

–report–ignoreskipped (batch mode only)
> Whether to ignore skipped nodes in the report in which case the legend and respective columns for skipped tests are also not shown. Default is false.

–report–junit <directory> (batch mode only)
> Create a report in JUnit XML format as understood by many Continuous Integration Tools. The directory name may contain placeholders as explained in section 42.2.4[(884)].

–report–name <name> (batch mode only)
> Specify the name for the report, meaning its identifier, not a file name. Default is the runid. The name may contain placeholders as explained in section 42.2.4[(884)].

–report–nodeicons (batch mode only)
> Whether to show icons for nodes in the report. Default is true, use `–report–nodeicons=false` to disable.

–report–passhtml (batch mode only)
> Whether to pass HTML tags in comments through to the HTML report. Default is true, `–report–passhtml=false` to disable.

–report–piechart (batch mode only)
> Whether to create a pie chart in the top part of the HTML report. Default is true, `–report–piechart=false` to disable.

–report–include–suitename (batch mode only)
> Whether to use the value of the Name[(516)] attribute of the Test suite[(515)] step as label of a testsuite in the HTML report. Default is true, `–report–include–suitename=false` to use the file name instead.

–report–scale–thumbnails <percent> (batch mode only)
How to scale thumbnail images for screenshots in the error listings of the report, an integer value interpreted as percent of the original image size. Default is 20.

–report–teststeps (batch mode only)
Whether to list test steps in the report. Default is true, use `–report-teststeps=false` to disable.

–report–thumbnails (batch mode only)
Whether to display thumbnail images for screenshots in the error listings of the report. Default is false.

–report–warnings (batch mode only)
Whether to show warning information in the report. Default is true.

–report–xml <directory> (batch mode only)
Create an XML report. The directory name may contain placeholders as explained in section 42.2.4[(884)].

–reuse (interactive mode only)
This argument is used mainly when launching QF-Test from a desktop icon or the Windows explorer through a file association. It tells the newly started QF-Test instance to search for an already running version of QF-Test and ask that to open the given file(s). If another instance can be reused in that way, the newly started program will terminate immediately and new windows for the file(s) will be opened by the old instance.

–run (interactive and batch mode)
If this parameter is set for the interactive mode, it will directly start the specified test suites or tests after launching QF-Test. Using it in batch mode explicitly tells QF-Test that this batch run is for actual test execution as opposed to generating documentation or a report. As this is the default operation for batch mode this argument can be omitted.

–runid <ID> (batch and calldaemon mode)
Specify an ID for the test run. The ID may contain placeholders as explained in section 42.2.4[(884)] and will itself serve as a replacement for the placeholder `%i/+i`.

–runlog [<file>] (batch and calldaemon mode)
Save the run log in the given file. The optional filename may contain placeholders as explained in section 42.2.4[(884)]. If no filename is given, it is composed of the basename of the suite and a timestamp. If missing, the extension `.qrz` is added automatically and the run log is saved compressed. Otherwise the extension `.qrl` or `.qrz` determines compression. Even without this argument a run log is created unless suppressed with `–nolog`[(874)] or when a

report is generated. The default value is `%p%b.qrz`. In calldaemon mode, a run log will be stored only if a (local) filename is specified.

-runlogdir <directory>
In interactive mode this argument overrides the option Directory for run logs[(503)] at a special layer for command line arguments. If specified, interactively changing the option has no effect whereas changing it at script level is still possible. In batch mode this directory serves as the target base directory for saving run logs unless the filename of the run logs specified with `-runlog [<file>]`[(878)] is an absolute path. If this argument is given and a report is generated, the files in the report will be laid out according to the structure of the run logs relative to this directory. The directory name may contain placeholders as explained in section 42.2.4[(884)].

-runtime
Use a runtime license only. In batch mode, QF-Test will normally use a runtime license (or multiple runtime licenses if `-threads <number>`[(883)] is given). If not enough free runtime licenses are available, full development licenses will be used instead unless `-runtime` is given in which case no development license is used and QF-Test will fail with an error message. In interactive mode, if `-runtime` is given, QF-Test will use a runtime license instead of a full development license. In that mode, any test suite can be loaded and tests can be run interactively as usual, including debugging support. Saving of test suites will be disabled, however, though test suites can be modified for temporary experiments.

-shell <executable>
The shell to use when executing a Execute shell command[(652)] node. Default for Unix is `/bin/sh`, for Windows `COMMAND.COM` or `cmd.exe`.

-shellarg <argument>
The argument that causes the shell specified with `-shell <executable>`[(879)] to execute the following argument and then exit. For Unix shells this is typically `-c`, `COMMAND.COM` and `cmd.exe` expect `/c`. If you have Unix tools installed on Windows and specify `sh` or `bash` as the shell to use, don't forget to change `-shellarg <argument>` to `-c`.

-shutdown <ID> (batch mode only)
Cleanly terminates the QF-Test instance with the given process ID (only digits) or shutdown ID on the local system, if allowed (see `-allow-shutdown [<shutdown ID>]`[(868)]). In batch mode this means that a running test is stopped, connected clients are terminated, the run log is saved, and the QF-Test instance terminates with the exit code -12. If QF-Test is running interactively, all open testsuites are closed without saving modifications. In special cases this might be useful, especially if auto-save is configured in a

meaningful way (see Auto-save interval (s)⁽⁴³⁵⁾). The special shutdown ID `all` terminates all running QF-Test processes that were started with the `-allow-shutdown [<shutdown ID>]`⁽⁸⁶⁸⁾ argument to explicitly allow that.

–serverhost <host>

    Set the host name or IP address for communication between QF-Test and the SUT. You may need to do this when running QF-Test and the SUT on different machines or if you experience troubles with reverse name lookup. The default is to use the loopback interface. To use the primary network interface of the local host, specify `-serverhost=` with an empty value.

–sourcedir <directory> (batch mode only)

    If this argument is given and a report is generated, the files in the report will be laid out according to the structure of the test suites relative to this directory unless `-runlogdir <directory>`⁽⁸⁷⁹⁾ is also specified. In any case, the directory of a test suite listed in the report will only be listed if this argument is specified and the test suite is located below this directory.

–splitlog (batch mode only)

    In batch mode split run logs (see section 7.1.6⁽¹²²⁾) are enabled by default and can be turned off via `-splitlog=false`. If `-splitlog` is explicitly specified without parameter, the default extension for run logs is changed from `.qrz` to `.qzp` so as to create split run logs in ZIP format. The same can be achieved by specifying the name of the run log explicitly with the desired extension.

–startclean (calldaemon mode only)

    With this argument all contexts of the shared TestRunDaemon are cleaned up and released before running the test.

–startsut (only for internal use)

    This argument is used to start a client application on a remote host. You should not work with this argument directly. The standard library `qfs.qft` contains a procedure `qfs.daemon.startRemoteSUT` which can be use for this purpose.

–stopclean (calldaemon mode only)

    With this argument all contexts of the shared TestRunDaemon are cleaned up and released after running the test.

–stoprun (calldaemon mode only)

    Stop a running test executed by the Daemon on the given host and port. This argument can be combined with `-cleanup`⁽⁸⁷⁰⁾ or `-stopclean`⁽⁸⁸⁰⁾.

–suitedir <dir> (calldaemon mode only)

    Specify the (remote) directory where the QF-Test Daemon looks for test suites. Use an absolute path when specifying the test to execute, if this argument is not given.

–suitesfile <file> (interactive and batch mode)
    Specify a text file containing test suites or test cases to be executed. You should
    specify one test suite per line. Individual test cases can be added as for the the
    `–test <n>|<ID>` argument. You can find some samples in the table below.

Entries in file	Description
path/suite1.qft path/suite2.qft	Both test suites will be executed.
path/suite1.qft path/suite2.qft#id-tc1	Test suite suite1.qft will be executed completely and test case 'id-tc1' of suite2.qft will be executed.
path/suite1.qft -test tc1 -test tc2	Test cases tc1 and tc2 of suite1.qft will be executed.

Table 42.1: Samples `–suitesfile <file>`

–systemcfg <file>
    Set the location of the system configuration file (see section 1.6[(10)]).

–systemdir <directory>
    Override the location of the directory holding the system configuration files (see
    section 1.6[(10)]) including optional plugins and scripting modules. If
    any of `–systemcfg <file>`[(881)], `–plugindir <directory>`[(876)],
    `–jythondir <directory>`[(873)], `–groovydir <directory>`[(872)] or
    `–javascriptdir <directory>`[(873)] are also specified they have precedence.

–tempdir <directory> (interactive mode only)
    Can be used to specify a temporary directory which is needed on Windows only
    for the context sensitive help system. By default the values of the environment
    variables `TEMP` and `TMP` are tried.

–terminate (calldaemon mode only)
    Use this option, if you want to terminate a running QF-Test daemon.

–test <n>|<ID> (interactive and batch mode)
    Without this argument all the top-level tests of the suite are executed one after
    the other. Using `–test <n>|<ID>`, you can select specific tests. An arbitrary
    node located anywhere in a test suite can be accessed by its QF-Test ID[(524)].
    Test case[(518)] or Test set[(525)] nodes can be referenced also by their qualified name.
    Top-level tests can also be selected by their index, the index for the first test being
    0. You can use this argument multiple times, even for the same test.

–testdoc <directory> (batch mode only)
 With this argument QF-Test creates both HTML and XML testdoc documentation.
 If no directory is given, it is created from the basename of the suite.

–testdoc–doctags (batch mode only)
 Whether to use the QFS doctag extensions when creating the testdoc
 documentation. Default is true, use `–testdoc-doctags=false` to disable.

–testdoc–followcalls (batch mode only)
 By default QF-Test ignores Test call[(531)] nodes during testdoc creation. With this
 argument the target Test case, Test set or whole test suite are processed as if they
 were part of the original test suite. Thus it is possible to create a subset testdoc
 documentation by creating a dedicated test suite with Test calls to the required
 parts.

–testdoc–html <directory> (batch mode only)
 With this argument QF-Test creates HTML testdoc documentation. If no directory
 is given, it is created from the basename of the suite.

–testdoc–nodeicons (batch mode only)
 Whether to show icons for nodes in the testdoc documentation. Default is true,
 use `–testdoc-nodeicons=false` to disable.

–testdoc–passhtml (batch mode only)
 Whether to pass HTML tags in comments through to the HTML testdoc. Default
 is true, use `–testdoc-passhtml=false` to disable.

–testdoc–sorttestcases (batch mode only)
 Whether to sort test cases alphabetically. Default is true, use
 `–testdoc-sorttestcases=false` to disable.

–testdoc–sorttestsets (batch mode only)
 Whether to sort test sets alphabetically. Default is true, use
 `–testdoc-sorttestsets=false` to disable.

–testdoc–splitparagraph (batch mode only)
 Specifies whether comments are splitted into paragraphs by using empty lines.
 Default is true, with `–testdoc-splitparagraph=false` it is possible to
 disable this option.

–testdoc–stylesheet <file> (batch mode only)
 Optional XSLT Stylesheet for the second step of the transformation.

–testdoc–teststeps (batch mode only)
 Whether to list test steps in the testdoc documentation. Default is true, use
 `–testdoc-teststeps=false` to disable.

–testdoc–xml <directory> (batch mode only)
> With this argument QF-Test creates XML testdoc documentation. If no directory is given, it is created from the basename of the suite.

–threads <number> (batch mode only)
> Run the same test suite in a number of parallel threads. Typically used for the purpose of load testing. One license is required per thread, so normally –runtime[(879)] should be specified as well. See chapter 31[(375)] for more information about load tests.

–timeout <milliseconds> (batch or calldaemon mode only)
> Give a timeout value in milliseconds for the test run when executing a test in batch mode or through the QF-Test daemon. Default is infinite.

–usercfg <file> (interactive mode only)
> Set the location of the user configuration file (see section 1.6[(10)]).

–userdir <directory>
> Override the location of the directory holding the user configuration files (see section 1.6[(10)]). If –usercfg <file>[(883)] or –runlogdir <directory>[(879)] are also specified they have precedence.

–variable <name>=<value>
> To override a system or suite variable definition (see chapter 6[(103)]) use this argument to set the variable named <name> to the value <value>. Using this argument multiple times you can define more than one variable.

–verbose [<level>]
> Print progress and status information during a test run to the console. This is in particular useful when driving a test via –calldaemon, because the actual test usually runs on a different host where you may not be able to observe it. Note that on Windows you need to use qftestc.exe (instead of qftest.exe) to see the output. Specifying a level of verbosity is optional, possible values are all (print all nodes) and tests (default, only Test set and Test case nodes are printed out). Each level can be combined with errors (print error and exception messages) like tests,errors.

–version
> Print version information and exit.

42.2.4 Placeholders in the filename parameter for run log and report

The filename given in any of the command line arguments <u>-runid <ID></u>[878], <u>-runlog [<file>]</u>[878], <u>-runlogdir <directory></u>[879], <u>-report <directory></u>[876], <u>-report-html <directory></u>[877], <u>-report-name <name></u>[877], <u>-report-xml <directory></u>[878] or <u>-report-junit <directory></u>[877], may contain placeholders of the form %X or +X (the latter must be used on Windows where '%' is a special character) where X may be any of the characters listed in the table below. QF-Test will fill in the respective value when creating the run log or report. All time values refer to the time the test was started.

Note When executing multiple test suites, be sure to include the base name of the suite in the filename by specifying %b. Otherwise only a single run log or report may be written that represents only the test run of the last test suite.

Character	Replacement
%	Literal '%' character.
+	Literal '+' character.
i	The current runid as specified with <u>-runid <ID></u>[878].
p	The directory of the test suite relative to <u>-sourcedir <directory></u>[880]. Expands to the absolute directory in case -sourcedir <directory> is unspecified and is empty if -sourcedir <directory> is specified but the test suite is not located below it.
P	The absolute directory of the test suite. May only be given at the beginning.
b	The basename of the test suite, exclusive directory and .qft extension.
r	The return value (or "exit code") of the test run (-runlog only).
w	The number of warnings in the test run (-runlog only).
e	The number of errors in the test run (-runlog only).
x	The number of exceptions in the test run (-runlog only).
y	The current year (2 digits).
Y	The current year (4 digits).
M	The current month (2 digits).
d	The current day (2 digits).
h	The current hour (2 digits).
m	The current minute (2 digits).
s	The current second (2 digits).

Table 42.2: Placeholders in filename parameters

So if, for example, you want to save the run log in a subdirectory of your test suite directory called `logs` and want to include a timestamp and the exit code, use

```
-runlog %p/logs/%b-%y%M%d-%h%m%s-%r.qrl
```

Note   It is possible to use %b, %p and %P for collective parameters like runid or report. This makes sense only when processing a single test suite. When processing multiple test suites, the name of the first test suite is used.

42.3   Exit codes for QF-Test

When run in interactive mode, the exit code of QF-Test is not very useful. It is either negative if QF-Test fails to start or 0.

In batch mode however the exit code expresses the result of the test run. Negative values represent fatal errors that prevent the test from being executed, while positive values stand for errors during the test run. Note that many systems only support exit codes between 0 and 255, so every exit code may have to be calculated modulo 256, i.e. -1=255, -2=254 and so on.

The following exit codes are currently defined:

Value	Meaning
0	Everything OK
1	Warnings occurred during the test run
2	Errors occurred during the test run
3	Exceptions occurred during the test run
-1	Unexpected Exception
-2	Bad command line arguments
-3	Missing or invalid license
-4	Errors while setting up the RMI connection
-5	Errors while loading the test suite
-6	The test suite doesn't contain any tests
-12	The process was terminated from outside via the `-shutdown <ID>`[879] batch command

Table 42.3: Exit codes for QF-Test

Besides, there are special exit codes when running QF-Test with the `-calldaemon` argument:

Value	Meaning
-7	Daemon could not be found
-8	Failed to get or create a TestRunDaemon
-9	Failed to get or create a run context
-10	The test could not be started
-11	The test has not ended within the given timeout

Table 42.4: `calldaemon` exit codes for QF-Test

Chapter 43

GUI engines

Swing, JavaFX, SWT can be combined together in a single application not only by using top-level windows of different technologies but also by embedding components of one technology into windows of another. QF-Test supports testing such kinds of applications.

4+ Also web pages can be integrated into Java applications by use of embedded browsers, e.g. JavaFX's WebView component of JxBrowser. QF-Test provides support for a number of such hybrid combinations.

To that end, the concept of a *GUI engine* was introduced. One GUI engine is responsible for handling recording and replay for one GUI toolkit thread. Normal applications have only one such thread. As explained above, combinations of Swing, JavaFX and SWT are possible that have one thread each and will thus require two GUI engines to operate in parallel. In theory it is also possible to have multiple GUI engines of the same kind, e.g. by creating multiple instances of the SWT Display class.

Note The first GUI engine created for an SUT is called the default engine. It is used in all cases where no GUI engine is explicitly specified, most notably SUT script[(631)] nodes with an empty GUI engine[(633)] attribute.

Each QF-Test GUI engine is identified by a token for the GUI toolkit and a number. `awt0`, `fx0` and `swt0` are the primary GUI engine for AWT/Swing, JavaFX and SWT. Unless you have a *very* special application you will never need to concern yourself with the number of the engine, as there will never be an engine called `awt1`, `fx1` or `swt1` and the alias `awt`,`fx` or `swt` is sufficient. When recording, QF-Test always uses the latter.

Note If your application uses only the default engine, engine names can be left empty. Alternatively the token `default` can be used to explicitly address the default engine.

Typically engine identifiers are automatically set correctly during replay. Only when inserted by hand they need to be considered. In a test suite, engine identifiers are now stored in the following places.

- Wait for client to connect[(670)] nodes. Only required if your application combines AWT/Swing, JavaFX and/or SWT. By specifying the engine attribute you can wait for the respective GUI engine to become initialized.

- Window[(812)] nodes. The engine of a Window node marks the window and all its child nodes as being either AWT/Swing, JavaFX or SWT. Embedded components of the other kind will be moved to a node for a pseudo window.

- SUT script[(631)] nodes. An SUT script is executed on the event dispatch thread of the SUT, so for combined AWT/Swing, JavaFX and/or SWT applications the engine is required to specify whether the script should be run on the AWT/Swing, JavaFX or the SWT thread. Thus, an SUT script node can only retrieve and interact with components of one kind.

- File selection[(704)] nodes. For Swing applications, the File selection node is rarely used because the Swing `JFileChooser` is implemented in Java and can be fully controlled by QF-Test. The SWT `FileDialog` on the other hand is similar to the AWT `FileChooser`. Both are implemented natively and QF-Test has no control over the individual controls. Also the JavaFX `FileChooser` needs special handling. Thus file selection must be replayed using the File selection node. Because this node is not explicitly associated with a Component or Window node, the engine has to be specified within the node.

Chapter 44

Running an application from QF-Test

The Setup sequence creation[29] is the recommended tool to set up your SUT for testing. It results in an advanced setup sequence already prepared for later requirements.

This chapter contains some details in case you want to create a setup sequence yourself.

44.1 Various methods to start the SUT

With the Quickstart Wizard QF-Test offers a utility to guide you through the steps of creating a start sequence for your SUT. Please refer to chapter 3[28] for more information about the Quickstart Wizard.

Nevertheless we also want to describe how to create a start sequence for your application manually. There are basically two ways to start a Java application as an SUT from QF-Test. The first one is based on the standard `java ...` command line with its two variants for either starting a class or a jar file. The alternative is running a script or executable file which will then start the Java program. Indirect methods like launching the SUT through `ant` also fall into this category, as do Java WebStart.

The following examples show some typical setups. To get more detailed information about the required attributes, please follow the respective links to the reference manual. The tutorial also includes a number of examples.

Independent of how the SUT is started, the respective node should typically be followed immediately by a Wait for client to connect[670] node with an identical Client attribute. Again, see the reference manual for further details.

44.1.1   A standalone script or executable file

If your application is started through a script or a binary executable, create a
<u>Start SUT client</u>⁽⁶³⁹⁾ as follows:

Figure 44.1: Starting the SUT from a script or executable

- Create a <u>Start SUT client</u>⁽⁶³⁹⁾ node.

- Assign a name to the client in the <u>Client</u>⁽⁶⁴⁰⁾ attribute.

- Set the <u>Executable</u>⁽⁶⁴⁰⁾ to the script or executable that starts your application. If the
  program is not located in a directory on the `PATH`, the full path is required.

- Set the <u>Directory</u>⁽⁶⁴⁰⁾ attribute to the working directory for your application.

- One thing to watch out for in scripts is redirection of the standard output and error
  streams (e.g. `>myapp.log`) which you may want to remove so that the output
  of the SUT reaches QF-Test and is captured in the run log. Similarly, the `start`
  command in Windows batch files causes the SUT to detach and keeps the output
  away from QF-Test.

44.1.2 An application launched through Java WebStart

Using the new connection mechanism, an application launched through Java WebStart can be started directly from QF-Test without the need to modify any JNLP files (**so do not use** Extras→Create WebStart SUT client starter...). Instead create a Start SUT client[(639)] node as follows:

Start SUT client
Client
SUT

▯ Executable
javaws

▭ Directory

＋ ✎ ✖ ↑ ↓ Executable parameters
Parameter
http://java.sun.com/products/javawebstart/apps/draw.jnlp

QF-Test ID

Delay before (ms)   Delay after (ms)

✎ Comment
Start SUT through Java WebStart

Figure 44.2: Starting the SUT through Java WebStart

- Create a Start SUT client[(639)] node.

- Assign a name to the client in the Client[(640)] attribute.

- Set the Executable[(640)] attribute to the Java WebStart executable which is typically called `javaws` and located somewhere inside the JDK or JRE. You'll probably have to specify the full path.

- For Java WebStart the Directory[(640)] attribute typically is of no consequence except that Java WebStart is looking in that directory for a file named `.javaws` which can contain settings like debug levels.

- Create an entry in the Executable parameters[(641)] for the executable to specify the URL for the application's JNLP descriptor.

44.1.3 An application started with `java -jar <archive>`

If your application is normally launched through a command of the form `java -jar <archive>`, create a Start Java SUT client[(635)] node as follows:

Figure 44.3: Starting the SUT from a jar archive

- Create a Start Java SUT client[(635)] node.

- Assign a name to the client in the Client[(636)] attribute.

- If necessary, change the Executable[(636)] attribute.   Its default value `${qftest:java}` is the `java` executable that QF-Test was started with.

- Set the Directory[(637)] attribute to the working directory for your application.

- Create two entries in the Executable parameters[(637)] table for the executable. Set the first to `-jar` and the second to the name of the archive. Unless the archive is located in the Directory[(637)] selected above, its full path is required.

44.1.4   An   application   started   with   `java -classpath <classpath> <class>`

If your application is normally launched through a command of the form `java -classpath <classpath> <class>`, create a Start Java SUT client[(635)] node as follows:

Figure 44.4: Starting the SUT via the main class

- Create a <u>Start Java SUT client</u>[(635)] node.

- Assign a name to the client in the <u>Client</u>[(636)] attribute.

- If necessary, change the <u>Executable</u>[(636)] attribute.   Its default value `${qftest:java}` is the `java` executable that QF-Test was started with.

- Set the <u>Directory</u>[(637)] attribute to the working directory for your application.

- Set the <u>Class name</u>[(637)] attribute to the fully qualified name of the application's starter class (the class with the `main()` method), just like for `java`.

- Create two entries in the Executable parameters[(637)] table for the executable. Set the first to `-classpath` and the second to the list of jar files and directories that constitute the classpath. The full path is required for jar archives not located in the Directory[(637)] selected above. This argument can get very long and hard to edit directly in the table. See section 2.2.5[(17)] about how to pop up a dialog for more convenient editing.

44.1.5 A web application in a browser

Like Swing, JavaFX or SWT clients, a web-based SUT - i.e. a browser - is started as a separate process from within QF-Test. In order to gain access to the internals of the browser and the web page shown with its *Document Object Model* (DOM), QF-Test embeds a standard browser like Chrome in a special wrapper application. The technology for embedding and accessing those standard browsers enables efficient access to the DOM beyond the browsers' standard interfaces and a unified interface that hides browser differences and enables QF-Test - and thus you - to focus on test automation with a single set of tests for all supported browsers on multiple platforms.

A Start web engine[(645)] node can be used to launch a browser.

Figure 44.5: Launch the browser process

Browser windows can be opened via a Open browser window node in an already running process.

Figure 44.6: Open the web site in the browser

When setting up the startup sequence with the <u>Setup sequence creation</u>[(29)] or defining your own <u>Directory of browser installation</u>[(647)] attribute, try pointing QF-Test to a current Firefox installation. On Linux, the standard browser for your distribution may be installed in various places.

44.1.6   Opening a PDF Document

4.2+ QF-Test allows to verify PDF documents. Therefore, a client is started as separate process within QF-Test. In order to gain access to the internals of the PDF document and its components QF-Test analyzes the document in its own viewer.

Figure 44.7: Opening a PDF Document

A Start PDF client[654] node can be used to launch the viewer and to open the PDF document.

More information can be found in chapter 17[239].

Chapter 45

JRE and SWT instrumentation

45.1 JRE deinstrumentation

JRE instrumentation has been obsolete for a long time now and can even cause problems. The following section remains only to provide background information and to show how to remove an existing JRE instrumentation without reinstalling the JRE.

To remove instrumentation of a JRE you need write permission for some of its directories, so you may have to work with administrator privileges to perform this step.

When instrumenting the JRE QF-Test made use of the official accessibility interface which is provided by Java for just this purpose. It can be used by accessibility and capture replay tools to interact with Java applications without those applications knowing about it and without requiring any changes to those applications.

To activate this interface, QF-Test created or modified the file `.../lib/accessibility.properties` in the JRE installation and added the class `de.qfs.apps.qftest.start.Connector` to the property "assistive_technologies". This has the effect that this class will be instantiated whenever the AWT toolkit is initialized in any Java application that is run with this JRE.

To make sure that this class can always be found without modifications to the `CLASSPATH`, the file `qfconnect.jar`, which contains the Connector class, was placed in the Java extensions directory `.../lib/ext`.

To deinstrument the JRE, first remove the entry for `.../lib/accessibility.properties` from the "assistive_technologies" property in `.../lib/accessibility.properties` so that the Connector class will no longer be used. Next delete the file `qfconnect.jar` from the `.../lib/ext` directory of the JRE which is possible only when no Java program is currently being run with this JRE.

45.2 SWT instrumentation

Some special setup is required for testing SWT based applications with QF-Test/swt. Because SWT was not written with testability in mind, applications need to be run with a few slightly modified SWT classes in which we have added the necessary hooks for event filtering and component tracking to enable testing. The changes are transparent so that the behavior of an application is not changed, regardless of whether it is run inside or outside of QF-Test.

4.5+ If the SUT is run with the QF-Test agent and the option Connect without SWT instrumentation[(497)] enabled, the required classes are exchanged by the agent during startup of the SUT. This works for all SWT versions on Windows and for SWT 4.8 and higher on Linux. Older versions on Linux still need to be instrumented as described below. It is generally a good idea to include a call to the SWT instrumentation procedure into your startup sequence with the parameter `forceInstrumentation` set to `false`. That way QF-Test can determine dynamically based on options settings and SWT version, whether instrumentation is required or the agent can do its job.

If you use QF-Test's Quickstart Wizard to create the setup sequence for your SUT (see chapter 3[(28)]), it will take care of SWT instrumentation as well. For those with an aversion to wizard dialogs, the manual way is described next.

The standard library `qfs.qft`, which is part of the QF-Test distribution and described in detail in the tutorial, contains a Procedure[(586)] with which to perform the SWT instrumentation. It is named `setup` and located in the Package[(594)] `qfs.swt.instrument`. Insert a Procedure call[(589)] node before the start node for your SUT in your setup sequence. Set its Procedure name[(590)] attribute to `qfs.qft#qfs.swt.instrument.setup` and in the Variable definitions[(591)] set the parameter `sutdir` to the installation directory of your application. The plugin parameter can be left empty except when you are testing an Eclipse/RCP application that does not follow the standard plugin directory layout. In that case you can specify the plugin file to instrument directly via the `plugin` parameter. That's all. In case you want to know what goes on behind the scenes, all manual steps are described further on in this section.

45.2.1 Preparation for manual SWT instrumentation

Supported architectures for SWT testing are 32 and 64 bit Windows and 32 and 64 bit Linux with Gtk. The instrumentation files are provided in directories called `.../qftest-7.1.2/swt/$ARCH/$VERSION` where `$ARCH` is either `win32`, `win32-64`, `linux-gtk` or `linux-gtk-64` and `$VERSION` is one of the supported SWT versions.

First you need to determine whether your application is a standalone SWT application or is based on eclipse. To do so, simply take a look at the directory structure of your

application. If you find a directory called `plugins` containing a file called `org.eclipse.swt.win32.win32.x86_X.Y.Z.jar` (on Windows) or `org.eclipse.swt.gtk.linux.x86_X.Y.Z.jar` (on Linux), with `X.Y.Z` representing a version number like `3.2.0`, your application is based on eclipse. For a standalone SWT application you should find a file called `swt.jar`, typically inside a directory called `lib`.

45.2.2 Manual SWT instrumentation for eclipse based applications

Simply replace the SWT plugin jar with one instrumented by QF-Test. To create the instrumented plugin you must run the Procedure `qfs.qft#qfs.swt.instrument.setup` described above once with your original plugin (or a copy thereof) specified in the `plugin` parameter. QF-Test will create a backup copy of the original jar named `_org.eclipse.swt....jar.orig`. Next copy the instrumented plugin to the `plugin` directory of your application.

Finally, start your application once from the command line with the `-clean` command line argument to have it rebuild its plugin cache, e.g.

```
eclipse -clean
```

Your application's binary name may be different from `eclipse`, but all eclipse based applications should support the `-clean` argument.

45.2.3 Manual instrumentation for standalone SWT applications

For standalone SWT applications, replace the `swt.jar` file with the one provided with QF-Test. You may want to create a backup of the original first.

Note   If you are launching the client application by means of a Start Java SUT client[635] node, you can set the classpath to point to the corresponding `.../qftest-7.1.2/swt/$ARCH/$VERSION/swt.jar` archive and leave your original file untouched.

Chapter 46

Technical information about components

46.1 Weighting of recognition features for recorded components

When searching for a component, QF-Test calculates the probability with which each component in the SUT corresponds to the searched component. The component with the highest probability is then used, as long as that this probability is above a freely configurable threshold. First, the probabilities of the windows in the SUT are examined. Then, the search is continued in the window with sufficiently high probability.

The same procedure is followed level by level, i.e. for each direct and indirect parent node of the searched Component[(823)] node, but from top to bottom. At each level, the components matching the attribute Class name[(825)] are determined and their probability is calculated. Invisible components are not considered.

At each level, the probability of a component is calculated in several stages:

- Every calculation starts with a value of 99 percent, which is first reduced by deviations from geometry specifications. This serves as the base probability for the next stages.

- The following three stages can either result in a "match", "no match", or be skipped. If no value is specified for a stage, it is skipped; the probability remains unchanged. Each of the three steps has a freely configurable bonus in case of a match or a penalty in case of a deviation. A bonus in effect increases the probability score to more than its value, a penalty reduces it to below its value.

- First, the structure of the Components[(823)] is checked, (not of Windows[(812)], which do not have this information). All components of the currently evaluated container

component whose class matches the given Class name[(825)] or a derived class are collected in a list (including invisible components). For a match, the amount of previously identified components with the matching class as well as the index of the component in this list must match.

- Then, the Feature[(825)] and possible Extra features[(825)] are checked. If the test of an Extra feature with status "must match" fails, the component is discarded.

- Finally, the Name[(825)] of the component is checked. If a Name[(825)] exists, this is the deciding check since bonus and penalty have the highest values here.

For dialogs, there is another step that checks the modality of the dialog. Normally, a dialog is either modal or non-modal, so a mismatch would prevent detection. However, one and the same dialog could be presented modally or non-modally depending on context. If your SUT contains such a dialog, you must set "Modal penalty" to a value above the minimum probability.

If the calculated probability does not reach a minimum value, the component is discarded. The component with the highest probability is used. If there is a discrepancy in the component's structure, feature, or name, a message is written to the log, as this may indicate that it is not the correct component after all. Most of the time, however, this just indicates that the SUT has changed slightly. The component should then be updated before the changes accumulate and the component is no longer recognized.

Even though the search for the name already dominates this process, you can increase its importance even more by setting the options Name override mode (replay)[(472)] and Name override mode (record)[(450)] to "Override everything". Then QF-Test will simplify the search for a component if it has a name. Instead of, as explained above, working through all parent containers from the outside in, they are skipped and the window is directly searched for a component matching the name and class. This increases the independence from the GUI structure; The component will even be recognized if a new hierarchy level between window and component is introduced or removed. This method requires that, if a name is given, it is unique at least among the visible components of the same class in one window.

If such uniqueness is not given, "Hierarchical resolution" is the next best setting for the two options. It requires that two components with the same name have at least differently named parent containers. This setting preserves most of the benefits and flexibility of names. However, it will fail recognition if a named component is moved from its parent container.

46.2 Generating the component QF-Test ID

QF-Test uses the following algorithm designed for the best possible assignability of GUI objects when creating QF-Test IDs[(824)]:

1. The Name[(825)] attribute of a Component has a value: The value is used for the QF-Test ID.

2. No Name, but a Feature[(825)] is available: This value is used for the QF-Test ID and prefixed with the value of the Class name[(825)] in lowercase letters.

3. No Name or Feature[(825)] is available, but an Extra features[(825)] named `qfs:label` is: This value is used for the QF-Test ID and prefixed with the value of the Class name[(825)] in lowercase letters.

4. Neither name nor description is available: The class name in lowercase letters is used.

If the QF-Test ID derived this way is not unique, a running number is attached.

Finally a prefix can be added to the QF-Test ID. This depends on the following options:

- Prepend QF-Test ID of window parent to component QF-Test ID[(452)]

- Prepend parent QF-Test ID to component QF-Test ID[(452)]

You can find these settings in the options menu in section Recording→Components. Open the options menu via Edit→Options

Find examples in How to judge robust component recognition[(50)].

Note
Since the QF-Test ID attribute only serves to link the test nodes to recorded components, it can be nice to change it afterwards for better readability. If you choose a value that is already in use, QF-Test will output a warning. If you have already recorded events referring to this component, QF-Test will offer to automatically adjust their QF-Test component ID attribute. This automatic feature does not work for references including variables in the QF-Test component ID attribute.

46.3 SmartIDs - general syntax

The various SmartID features can be combined with each other. The following is the general syntax for combining SmartID features. Square brackets mark optional elements, while uppercase text signifies a placeholder:

`#[%][noscope:][ENGINE:][CLASS:][VALUE][<INDEX>]`

A SmartID consists of the following parts in the following order:

1. `#` always indicates the beginning of a SmartID.

2. `%` (optional) can mark VALUE as a regular expression (see <u>Regular expressions</u>[(909)]).

3. `noscope:` (optional) can be used to indicate that the SmartID also applies outside of the current <u>Scope</u>[(80)].

4. `ENGINE:` (optional) specifies the UI engine to which the SmartID applies to. This is only needed if QF-Test is connected to multiple applications using different UI technologies at the same time. Valid values are: `awt:`, `fx:`, `swt:`, `web:`, and `win:`, whereby letter casing does not matter.

5. `CLASS:` (optional) specifies the generic class or class of the component (see <u>chapter 59</u>[(1186)]), for example `Label:`, `CheckBox:ComboBoxListItemCheckBox:` or `CheckBox\:MyCheckBox:`.

6. `VALUE` (optional if CLASS is given) specifies the value the component must match, e.g. a label or name or the value of another criterium specified by MATCH.

7. `<INDEX>` (optional) specifies a numerical index, starting at 0.

Examples:

- `#Button:OK`

- `#TextField:Postal code<1>`

- `#TextField:name=address<2>`

- `#Postal code<1>`

- `#noscope:SWT:Label:Postal code<1>`

46.4 SmartIDs - special characters

The special characters `:`, `@`, `&` and `%` have special meanings in SmartIDs. ":" terminates a component class or UI engine. A `%` at the start of a SmartID signals the use of a regular expression (see <u>section 47.3</u>[(909)]). The other characters mark the beginning of a sub-item, like a table cell. If these characters should appear in a SmartID with their literal meaning, they must be escaped by prefixing them with `\`.

Example: A dialog title containing an email address is to be used as SmartID. The `@` inside must be escaped like this: `#abc\@qfs.de`.

A `%` at the beginning of a SmartID, directly after the `#`, can not be escaped. In this case it is better to place a prefix or the class between `#` and `%`. Now, `%` can be escaped with a `\`.

Example: A button has the label `%`. As SmartID, you can use `#Button:\%` or `#Label=\%`.

A ":" character that is part of a built-in generic class (see chapter 59[(1186)]), like `Panel:TitledPanel`, does not need to be escaped.

46.5   Android - list of trivial component identifiers

The following component identifiers are not transferred to the 'Name' attribute by default. This can be controlled via a NameResolver (see section 52.1.7[(1023)]).

- `button`
- `canvas`
- `checkbox`
- `choice`
- `container`
- `content`
- `dialog`
- `dock`
- `drawer`
- `filedlg`
- `frame`
- `label`
- `list`
- `menu`
- `menubar`

- menuitem

- pager

- panel

- popup

- row

- scrollbar

- scrollpane

- tabs

- textfield

- toolbar

- title

- text

- win

Chapter 47

Technical details about miscellaneous issues

47.1 Drag&Drop

From the beginning Drag&Drop has been hard to implement in Java. JDK 1.1 had no Drag&Drop support at all and the first steps taken with JDK 1.2 were far from satisfying. In JDK 1.3 Drag&Drop has matured, but there are still problems with stability. That Drag&Drop is so hard to get right is partly due to the fundamental difference in the way Drag&Drop is implemented by the underlying systems. Another reason is the goal to support Drag&Drop between Java and the underlying system and not just between Java applications.

As a result Drag&Drop for Java is implemented "native", meaning at the level of the operating system, so the events involved can be neither interpreted, nor generated by QF-Test. Hence Drag&Drop cannot be replayed directly.

Even so QF-Test has support for Drag&Drop, at least for JDK 1.3 and above. Direct recording of Drag&Drop is supported with JDK 1.4 and above. Replay of Drag&Drop is achieved through the special Mouse events[681] DRAG_FROM and DROP_TO as well as the optional DRAG_OVER. To replay these events, QF-Test makes use of the AWT Robot that was added to Java with version 1.3. The AWT Robot makes it possible to generate "hard" events at system level. These "hard" events actually move the mouse cursor across the screen and can trigger a Drag&Drop operation.

Using Drag&Drop together with the Shift and or Control key is also supported. To simulate a Drag&Drop with the Control key pressed, change the Modifiers[683] attributes of the DRAG_FROM, DRAG_OVER and DROP_TO events to include the Control modifier by adding 2 to the current value. It is even possible to simulate the Shift or Control key being pressed or released during the drag operation by changing the Modifiers of only some of the events.

As mentioned above, Drag&Drop used to be a bit unstable on some systems. In some cases with older JDKs the use of the AWT Robot to simulate Drag&Drop could crash the SUT or even the whole system. Nowadays the situation is much better. The introduction of mouse movement interpolation helped improve the reliability of Drag&Drop replay a lot. Please see section 39.3.5[(476)] for details.

Note On Windows simulating Drag&Drop may conflict with some mouse drivers. In case of problems reduce the speed of the mouse cursor to 50% in the mouse settings of the control panel.

47.2 Timing

Besides component recognition, timing is an inherently difficult problem in automatic testing. No two runs of a Java program are identical when it comes to timing. Too much depends on things like system load or memory usage. This can lead to a situation where a target component for an event is not available, because the VM is still busy popping up the dialog window that contains it.

To avoid needless failures of tests, QF-Test combines several tactics:

- Events are synchronized with the AWT event queue, meaning that after each event sent to the SUT, QF-Test waits until the events generated as side effects have been processed, before it sends another event.

- In some cases, especially with asynchronous updates, this doesn't suffice, so whenever a component is not available, QF-Test waits for a certain amount of time to give it a chance to appear. This delay can be customized through the option Wait for non-existent component (ms)[(480)].

- An additional timeout defined by the option Wait for non-existent item (ms)[(480)] applies when looking for sub-items of a component, e.g. a tree node.

- These default timeouts should be kept rather short, a few seconds at most, so in addition you can set individual timeouts and delays wherever applicable.

47.3 Regular expressions

The regular expressions that you can use in the search and replace dialogs and in places like the Feature[(825)] attributes, the Primary index[(830)] of an Item[(829)] node or in checks[(707)] all use standard Java regexp syntax. Before QF-Test version 3.1 the default was the GNU regexp package (see also appendix F[(1409)]). It can still be activated via the option Use old-style GNU regexps (before 3.1)[(483)]. One major difference between the

two is that '{' and '}' are special characters that need to be escaped with '\' for Java regexps, but normal characters for GNU:

Detailed regexp documentation with links to further information and even a whole book about regular expressions are provided in the Java documentation for the class `java.util.regex.Pattern` at http://download.oracle.com/javase/1.5.0/docs/api/java/util/regex/Pattern.html. It's also worth to have a look at the Wikipedia article about regular expressions.
Following is a short summary of the basics:

- A '.' stands for any one character except line breaks. With the new Java regexps you can start your regexp with the embedded flag '(?s)' to treat multi-line text like a single line so '.' will match everything. The old GNU regexps have no such flag, so you need to use '(.|\n)' to match everything. Unfortunately this expression causes a `StackOverflowException` with Java regexps, so if QF-Test finds this expression in any regular expression it will treat it as a GNU regexp regardless of the option setting.

- Character in between '[' and ']' match any one of these characters.

- A '?' says the preceding element is optional, i.e. it may appear 0 or 1 times.

- '+' means at least one of the preceding element.

- '*' means 0 or more of the preceding element.

- A group is created with '(' and ')'. A '?', '+' or '*' after the closing brace refers to the whole group. All groups in a regexp are numbered in the order of their opening brace. The first group has the number 1, 0 stands for the whole regexp. For search and replace, `$n` in the replacement string expands to the part of the original value matched by the nth group. Example: To change the extension of file names starting with `/tmp/` from `.foo` to `.bar`, search for `(/tmp/.*)\.foo` and replace with `$1.bar`.

- A '|' separates alternatives in a group.

- '\' quotes to suppress the special meaning of the following character or introduces special characters, e.g. '\n' for LineFeed (a line break), '\r' for CarriageReturn (not needed for QF-Test, see) or '\t' for Tab.

Examples:

- `.*` describes a sequence of arbitrary characters, which is optional.

- `.+` describes a sequence of arbitrary characters, but there must be at least one character, i.e. some mandatory characters.

- `[0-9]` describes one arbitrary cipher.

- `[0-9]+` describes a sequence of arbitrary ciphers, but there must be at least one cipher.

- `[0-9]{1,3}` describes a sequence of arbitrary ciphers, but there must be at least one cipher and at maximum three ciphers.

- To match any text that contains the word 'tree' use `'.*tree.*'`.

- To match arbitrary text possibly including line breaks: `'(?s).*'` for Java regexps and `'(.|\n)*'` for GNU.

- To replace 'tree' in arbitrary text with 'node' use `'(.*)tree(.*)'` to search and `$1node$2` to replace. In the replace dialog simply replace `tree` with `node` and disable the "Match whole string" check box to achieve the same effect.

- To search for 'name' or 'names': `'names?'`

- To search for 'tree' or 'node': `'(tree|node)'`

- An arbitrary word consisting of letters and numbers: [0-9a-zA-Z]+

- ...

4.0+ QF-Test allows you to use the context menu item ⌾Escape text for regular expressions⌾ on all attributes which allow regular expressions in order to escape special characters of regular expressions correctly with '\'.

47.4   Line breaks under Unix and Windows

The difference in the treatment of line breaks between Unix and Windows is a well-known problem. While Unix uses a single LineFeed character ('\n', hex 0x0A) as line separator, Windows uses the combination CarriageReturn/LineFeed ('\r\n', hex 0x0D0A). Java automatically converts text as needed which generally works well.

However, the XML standard specifies that an XML parser has to convert line breaks of any type into LineFeed only, regardless of the system under which it is running. This could lead to trouble, for example when checking a multi line text field. QF-Test works around the problem by converting all text strings read from the SUT to the Unix version with LineFeed only. This has the added benefit that tests created on one system will run unchanged on the other.

47.5 Quoting and escaping special characters

A common problem for most complex systems is the treatment of characters with a special meaning. A typical example are blanks in filenames. To specify such filenames on the command line, they need to be protected either by using double quotes or by escaping the blanks with a backslash character ('\').

Since QF-Test makes use of special characters in various contexts while reading arbitrary strings from the SUT that may contain any character, some kind of quoting mechanism is unavoidable. That QF-Test runs on various operating systems and makes use of regular expressions[909] which have their own set of special characters doesn't make things any easier. However, QF-Test attempts to keep things as simple as possible by restricting quoting to the places where it can't be avoided and by quoting all strings read from the SUT during recording correctly.

The most prominent special character for QF-Test is the '$' sign used for variable expansion. Variable syntax is applicable in almost all attributes. If you need a literal '$' character in an attribute value you have to double it.

Example: To verify with a Check text[708] node that a text field contains the string "4 US$", set the Text[710] attribute to "4 US$$".

Other special characters are used only in a few places and must be quoted only there. These are the '#' character used for Procedure[586] and Component[823] access across suites and the characters '@', '&' and '%' for the special syntax for sub-item access[84]. Since these are used as separators they cannot be escaped by doubling them, so QF-Test follows the convention to use the backslash '\' as escape character which turns the backslash itself into another special character. To avoid quoting-hell with Windows filenames, QF-Test only uses quoting where the above characters are used and even there a single backslash that is not followed by a special character is interpreted literally.

To be precise, you have to escape the characters '#' and '\' in the Procedure name[590] attribute of a Procedure call[589] and the characters '#', '\', '@', '&' and '%' in the attributes QF-Test component ID[682] of events and checks as well as the Primary index[830] and Secondary index[830] of an Item[829]. Remember that the backslash is also used as the escape character for regular expressions[909], so to get a literal '\' into a regexp for a sub-item, you first need to escape it for the regexp itself, i.e. "\\", then escape these for QF-Test leading to "\\\\".

There is one more special case that requires a special character and corresponding quoting. This is the '/' character used as separator for tree path sub-items of a `JTree` component. Thus the '/' must be quoted if and only if you need a literal / in a sub-item of a `JTree` component. Sub-items of other components don't require special handling.

47.6 Include file resolution

This is a section you should hopefully never need to read. It explains in detail how implicit Procedure and Component references are resolved during test replay. If you need to read it, your test suite include hierarchy probably is too complicated and you should consider simplifying your includes.

There are basically two scenarios in which QF-Test must implicitly resolve a Procedure or Component reference to another suite when the requested Procedure or Component cannot be found in the current (or explicitly referenced) suite:

- The current suite includes other suites (by defining them in the Include files[516] attribute of the Test suite[515] root node). In this case, QF-Test searches all included suites in the given order.

- The current suite (or rather one of its Procedures) was called by another suite. Here QF-Test searches the calling suite for the requested node.

The whole thing gets complicated, when (possibly indirect) Procedure calls across test suite boundaries and (possibly indirect, maybe even recursive) includes are combined. Following are detailed explanations of the search algorithm that will hopefully enable you to debug and resolve any include-file related problems.

- Whenever execution leaves the current suite to continue with some Procedure or to retrieve a Component, the other suite becomes the current suite. This process is complemented by two things: the old suite is pushed onto the so-called *call-stack* and the variable bindings of the new current suite are pushed on top of the *fallback bindings stack* (see chapter 6[103]), so they override the bindings of the old suite. In the run log this process is documented by adding a *Suite-change* node which holds all of the run log nodes for the execution that takes place outside the old suite.

- Any search through test suites starts with the current suite, then continues top-down through the call-stack. So if, for example, A calls B which calls C, then C is searched first, followed by B and finally A.

- Includes are considered stronger bindings than the call-stack. This means that during the search through the current suite and the suites on the call-stack, at each step the included test suites are searched before moving to the next suite on the call-stack. For example, if A calls B which includes C, A is on the call-stack and B is the current Suite, then B will be searched first, then C, and lastly A.

- In case of multiple, possibly indirect includes, the search is always conducted depth-first in the oder in which the include files are listed. This means that if A

includes B and C, and B includes D, first A is searched, followed by B, then D and then C.

- If a PROCEDURE is found (possibly indirectly) in an included test suite (as opposed to the current suite, an explicitly referenced suite or a suite on the call-stack), the change from the old current suite to the new current suite doesn't take place in one step. This has to be illustrated with an example right from the start or we'll get totally lost: Let's say A calls B and that A includes C. B calls a PROCEDURE which is found in C by way of A. Instead of changing suites directly from B to C, A will first become the current suite and then C. As a consequence, A gets pushed onto the call-stack again on top of B and its variable bindings are also pushed again on top of B's bindings on the fallback bindings stack. The reasoning behind this is that C, which is now the current suite, is "closer to" A, which includes C, than it is to B, which only happened to be called by A. One could also say that inclusion creates a kind of union, so that to B, A and C will always appear as a single test suite as long as B doesn't call C explicitly.

- That's it, except for one thing: During a search QF-Test never searches a suite twice. This would be useless in any case, but it is more than an optimization, since it prevents trouble with recursive includes if A includes B and B includes A.

If you really have a problem determining how, why or why not a certain PROCEDURE or COMPONENT was retrieved, first take a look at the run log. It shows exactly which suites were used and which variable expansions took place.

Chapter 48

Scripting (Jython, Groovy and JavaScript)

This section explains technical details about the Jython integration in QF-Test and serves as a reference for the whole API exposed by QF-Test for use in Jython, Groovy and JavaScript scripts. For a more gentle introduction including examples please take a look at chapter 11[161].

48.1 Module load-path

The load-path for scripting modules is assembled from various sources in the following order:

- the script directory in the <u>user configuration directory</u>[10]

- the directory `qftest/qftest-7.1.2/<scriptlanguage>`

In addition, during Server script or SUT script node execution, the directory of the containing test suite is prepended to the path.

The directory `qftest/qftest-7.1.2/<scriptlanguage>` contains internal modules of the specific script language. You should not modify these files, since they may change in later versions of QF-Test.

The script directory in the <u>user configuration directory</u>[10] is the place to put your own shared modules. These will be left untouched during an update of QF-Test. You can locate the rescpective script directory via ⃞Help⃞ under "System info" as "dir.<scriptlanguage>".

Modules that are specific to a test suite can also be placed in the same directory as the test suite. The file extension for all modules must be `.py`.

In Jython you can add additional directories to the load-path by defining the `python.path` system property.

48.2   The plugin directory

The script languages can also be used to access Java classes and methods beyond the scope of QF-Test by simply importing such classes, e.g.

```
from java.util import Date
from java.text import SimpleDateFormat
print SimpleDateFormat("yyyy-MM-dd").format(Date())
```

Example 48.1: Accessing Java classes from Jython

The classes available for import are those on the `CLASSPATH` during startup of QF-Test or the SUT respectively, all classes of the standard Java API and QF-Test's own classes. For the SUT things also depend on the ClassLoader concept in use. WebStart and Eclipse/RCP in particular make it difficult to import classes directly from the SUT.

Additionally, there are plugin directories into which you can simply drop a jar file to make it available to scripts. QF-Test searches for a directory called `plugin`. You can locate the currently used plugin directory via ‖Help‖ under "System info" as "dir.plugin". The location of the plugin directory can be overridden with the command line argument `-plugindir <directory>`[(876)].

Jar files in the main plugin directory are available to both Server script and SUT script nodes. To make a jar available solely to Server scripts or solely to SUT scripts, drop it in the respective sub-directory called `qftest` or `sut` instead.

48.3   Initialization (Jython)

During QF-Test and SUT startup an embedded Jython interpreter is created. For QF-Test, the module named `qftest` is imported, for the SUT the module named `qfclient`. Both are based on `qfcommon` which contains shared code. These modules are required to provide the run context interface and to set up the global namespace.

Next the load-path `sys.path` is searched for your personal initialization files. For QF-Test initialization, the file called `qfserver.py` is loaded, the file called `qfsut.py` is used for the SUT. In both cases `execfile` is used to execute the contents of these files directly in the global namespace instead of loading them as modules. This is much more

convenient for an initialization file because everything defined and all modules imported will be directly available to Server scripts and SUT scripts. Note that at initialization time no run context is available and no test suite-specific directory is added to `sys.path`.

48.4 Namespace environment for script execution (Jython)

The environments in which Server scripts or SUT scripts are executed are defined by the global and local namespaces in effect during execution. Namespaces in Jython are dictionaries which serve as containers for global and local variable bindings.

The global namespace is shared between all scripts run in the same Jython interpreter. Initially it will contain the classes `TestException` and `UserException`, the module `qftest` or `qfclient` for QF-Test or the SUT respectively, and everything defined in or imported by `qfserver.py` or `qfsut.py`. When assigning a value to a variable declared to be global with the `global` statement, that variable is added to the global namespace and available to scripts run consecutively. Additionally, QF-Test ensures that all modules imported during script execution are globally available.

The local namespace is unique for each script and its lifetime is limited to the script's execution. Upon invocation the local namespace contains `rc`, the interface to QF-Test's run context, and `true` and `false` bound to `1` and `0` respectively for better integration with QF-Test.

Accessing or setting global variables in a different Jython interpreter is enabled through the methods `fromServer`, `fromSUT`, `toServer` and `toSUT`.

48.5 Run context API

The run context object `rc` is an interface to the execution state of the currently running test in QF-Test. Providing this wrapper instead of directly exposing QF-Test's Java API leaves us free to change the implementation of QF-Test without affecting the interface for scripts.

Following is a list of the methods of the run context object `rc` in alphabetical order. The syntax used is a bit of a mixture of Java and Python. Python doesn't support static typing, but the parameters are passed on to Java, so they must be of the correct type to avoid triggering exceptions. If a parameter is followed by an '=' character and a value, that value is the default and the parameter is optional.

Note   Please note that the Groovy syntax for keyword parameters is different from Jython and requires a ':' instead of '='. The tricky bit is that, for example, `rc.logMessage("bla"`,

`report=true)` is perfectly legal Groovy code yet doesn't have the desired effect. The '=' here is an assignment resulting in the value true, which is simply passed as the second parameter, thus the above is equal to `rc.logMessage("bla", true)` and the true is passed to `dontcompactify` instead of `report`. The correct Groovy version is `rc.logMessage("bla", report:true)`.

---

void addDaemonLog(byte[] data, String name=None, String comment=None, String externalizename=None)

Add a run log retrieved from a DaemonRunContext to the current run log.

Parameters

data	The byte array retrieved via DaemonRunContext.getRunLog().
name	An optional name for the daemon log node. If unspecified the ID of the Daemon is used.
comment	An optional comment for the daemon log node.
externalizename	An optional name to externalize the daemon log and save it as a partial log of a split run log.

---

void addResetListener(ResetListener listener)

Server only. Register a `ResetListener` within the current run context.

Parameters

listener	The Listener that should be added. The listener should implement the interface de.qfs.apps.qftest.extensions.qftest.ResetListener.

---

void addTestRunListener(TestRunListener listener)

Register a `TestRunListener` with the current run context. In interactive mode and batch mode there is a single, shared run context, so the listener will remain in effect until it gets removed via `removeTestRunListener` or `clearTestRunListeners`. In daemon mode, each `DaemonRunContext` has its own set of listeners. See section 52.7[(1087)] for details about the `TestRunListener` API.

Parameters

listener	The listener to register.

`String callProcedure(String name, dictionary parameters=None)`

Call a Procedure[586] in a test suite.

As a convenience, this method can also be called from an SUT script. Care should be taken however, because the script is executed inside the AWT event dispatch thread, so weird side-effects are possible, though QF-Test does its best to avoid these. If possible, call Procedures from a Server script instead.

Parameters

name	The fully qualified name of the Procedure.
parameters	The parameters for the Procedure. This should be a dictionary. Its keys and values can be arbitrary values. They are converted to strings for the call.
Returns	The value returned from the Procedure through an optional Return[592] node.

`int callTest(String name, dictionary parameters=None)`

Server only. Call a Test case[518] or Test set[525] in a test suite or an entire test suite.

Parameters

name	The fully qualified name of the Test case or Test set.
parameters	The parameters for the Test case or Test set. This should be a dictionary. Its keys and values can be arbitrary values. They are converted to strings for the call.
Returns	The final state of the execution. Either rc.OK, rc.WARNING, rc.ERROR, rc.EXCEPTION, rc.SKIPPED or rc.NOT_IMPLEMENTED.

`int callTestAsProcedure(String name, dictionary parameters=None)`

Server only. Call a Test case[518] or Test set[525] in a test suite or an entire test suite but treat it as a procedure call so that an uncaught exception terminates the entire call instead of just the currently executing Test case.

Parameters

name	The fully qualified name of the Test case or Test set.
parameters	The parameters for the Test case or Test set. This should be a dictionary. Its keys and values can be arbitrary values. They are converted to strings for the call.
Returns	The final state of the execution. Either rc.OK, rc.WARNING, rc.ERROR, rc.EXCEPTION, rc.SKIPPED or rc.NOT_IMPLEMENTED.

Boolean check(boolean condition, String message, int
level=rc.ERROR, boolean report=true, boolean nowrap=false)
Check or "assert" that a condition is true and log a message according to the result.

Parameters

condition	The condition to evaluate.
message	The message to log. It will be preceded by "Check OK: " or "Check failed: " depending on the result. For the old-style XML or HTML report the message will be treated like a Check node if it starts with an '!' character.
level	The error level in case of failure. The following constants are defined in the run context:

> - rc.OK
>
> - rc.WARNING
>
> - rc.ERROR
>
> - rc.EXCEPTION
>
> If the level is rc.EXCEPTION, a UserException[(858)] will be thrown if the check fails.

report	If true, the check will appear in the report. Only applicable if level <= rc.WARNING.
nowrap	If true, lines of the message will not be wrapped in the report. Use for potentially long messages.
Returns	The result of the check.

Boolean checkEqual(Object actual, Object expected, String message, int level=rc.ERROR, boolean report=true, boolean nowrap=false)
Check or "assert" that an object matches a given value and log a message according to the result. Comparison is done using the == operator.
Parameters

actual	The actual value.
expected	The expected value.
message	The message to log. It will be preceded by "Check OK: " or "Check failed: " depending on the result. In case of failure, the expected and actual values will also be logged.
level	The error level in case of failure. The following constants are defined in the run context:

- `rc.OK`

- `rc.WARNING`

- `rc.ERROR`

- `rc.EXCEPTION`

If the level is `rc.EXCEPTION`, a <u>UserException</u>[858] will be thrown if the check fails.

report	If true, the check will appear in the report. Only applicable if level <= rc.WARNING.
nowrap	If true, lines of the message will not be wrapped in the report. Use for potentially long messages.
Returns	The result of the check.

Boolean checkImage(ImageRep actual, ImageRep expected, String message, int level=rc.ERROR, boolean report=true, boolean nowrap=false)

Check or "assert" two given `ImageRep` (see section 52.10.1[(1096)]) objects for equality and log a message according to the result. Comparison is done using the `equals` method of the `ImageComparator` (see section 52.10.2[(1099)]) of the expected object.

Parameters

actual	The actual value `ImageRep` object.
expected	The expected `ImageRep` object.
message	The message to log. It will be preceded by "Check OK: " or "Check failed: " depending on the result. In case of failure, the expected and actual values will also be logged. For the old-style XML or HTML report the message will be treated like a Check node if it starts with an '!' character.
level	The error level in case of failure. The following constants are defined in the run context: • `rc.OK` • `rc.WARNING` • `rc.ERROR` • `rc.EXCEPTION` If the level is `rc.EXCEPTION`, a UserException[(858)] will be thrown if the check fails.
report	If true, the check will appear in the report. Only applicable if level <= rc.WARNING.
nowrap	If true, lines of the message will not be wrapped in the report. Use for potentially long messages.
Returns	The result of the check.

```
Object[] checkImageAdvanced(ImageRep actual, ImageRep expected,
String message, String algorithm, int level=rc.ERROR, boolean
report=true, boolean nowrap=false)
```
Check or "assert" two given `ImageRep` (see section 52.10.1[(1096)]) objects for equality and log a message according to the result. Comparison is done using the specified algorithm.

Parameters

actual	The actual value `ImageRep` object.
expected	The expected `ImageRep` object.
message	The message to log. It will be preceded by "Check OK: " or "Check failed: " depending on the result. In case of failure, the expected and actual values will also be logged. For the old-style XML or HTML report the message will be treated like a Check node if it starts with an '!' character.
algorithm	The algorithm to use for the comparison as described in chapter 57[(1167)].
level	The error level in case of failure. The following constants are defined in the run context:
	• `rc.OK`
	• `rc.WARNING`
	• `rc.ERROR`
	• `rc.EXCEPTION`
	If the level is `rc.EXCEPTION`, a `UserException`[(858)] will be thrown if the check fails.
report	If true, the check will appear in the report. Only applicable if level <= rc.WARNING.
nowrap	If true, lines of the message will not be wrapped in the report. Use for potentially long messages.
Returns	An array with following content: The result of the check as Boolean. The result of the check as probability. The transformed image of the expected image as `ImageRep`, depending on the algorithm. The transformed image of the actual image as `ImageRep`, depending on the algorithm. Further information where appropriate.

---

void clearGlobals()

Server only. Undefine all global variables.

void clearProperties(String group)

Server only. Delete a given set of loaded properties or resources.

Parameters

group	The group name of the properties or resources.

void clearTestRunListeners()

Remove all `TestRunListeners` from the current run context.

String expand(String text)

Expand a string using standard QF-Test variable expansion for `$(...)` or `${...:...}` syntax.

Remember to double the '$' signs to avoid expansion before the script is executed (see section 47.5[(912)]).

Parameters

text	The string to expand.
Returns	The expanded string.

Object fromServer(String name)

SUT only. Retrieve the value of a global variable in the Jython or Groovy interpreter of QF-Test. If the variable is undefined, a KeyError is raised.

Parameters

name	The name of the variable.
Returns	The value of the variable.

Object fromSUT(String client, String name)

Server only. Retrieve the value of a global variable in the Jython or Groovy interpreter of the SUT. If the variable is undefined, a KeyError is raised.

Parameters

client	The name of the SUT client.
name	The name of the variable.
Returns	The value of the variable.

Boolean getBool(String varname)

Look up the value of a QF-Test variable, similar to `lookup()`, but treat it as a boolean.

Parameters

varname	The name of the variable.
Returns	The value of the variable.

`Boolean getBool(String group, String name)`
Look up the value of a QF-Test resource or property, similar to `lookup()`, but treat it as a boolean.
Parameters

`group`	The name of the group.
`name`	The name of the resource or property.
Returns	The value of the resource or property.

`Exception getCaughtException()`
Server only. If the script is run inside a Catch[(619)] node, the exception that was caught is returned. In all other cases, `None` is returned.

Returns	The caught exception.

`Component getComponent(String id, int timeout=0, boolean hidden=false)`
SUT only. Find a component or a component's sub-item using QF-Test's component recognition mechanism.
Parameters

`id`	The QF-Test ID[(824)] of the Component[(823)] node that represents the component in the test suite.
`timeout`	This parameter is ignored and always 0 for SUT scripts that are running on the event dispatch thread of the respective GUI engine because it is not possible to free this thread in a safe way in order to wait for the respective component.
`hidden`	If true, find invisible components as well. Useful for menu items.
Returns	The actual Java component. For sub-items, a pair of the form `(component, index)` is returned, where the type of `index` depends on the type of the item. For tree nodes it is a `javax.swing.tree.TreePath` object, for table-cells a pair of the form `(row, column)` and an integer for all other kinds of items. Column indexes returned are always given in table coordinates, not in model coordinates.

`List getConnectedClients()`
Get the names of the currently connected SUT clients.

Returns	A list with the names of the currently connected SUT clients, an empty list in case there are none.

`Properties getGlobals()`

Get the global variables bound in the current context.

| **Returns** | The global variables of the current context. |

`Integer getInt(String varname)`

Look up the value of a QF-Test variable, similar to `lookup()`, but treat it as an integer.

Parameters

| **varname** | The name of the variable. |
| **Returns** | The value of the variable. |

`Integer getInt(String group, String name)`

Look up the value of a QF-Test resource or property, similar to `lookup()`, but treat it as an integer.

Parameters

group	The name of the group.
name	The name of the resource or property.
Returns	The value of the resource or property.

`Object getLastComponent()`

SUT only. Get the last component that was addressed by QF-Test for replaying some event, check or miscellaneous operation. Calls to `rc.getComponent()` have no impact.

| **Returns** | The last component addressed by QF-Test. |

`Exception getLastException()`

Server only. Get the last exception (caught or uncaught) that was thrown during the test run. In most cases `getCaughtException` is probably more useful.

| **Returns** | The most recent exception that was thrown. |

`Object getLastItem()`

SUT only. Get the last item that was addressed by QF-Test for replaying some event, check or miscellaneous operation. Calls to `rc.getComponent()` have no impact.

| **Returns** | The last item addressed by QF-Test. |

`Properties getLocals(nonEmpty=false)`

Get the innermost local bindings of the context. Mostly useful within a procedure to get the parameters of the procedure call and implement something similar to keyword arguments in Jython or Groovy.

Parameters

| **nonEmpty** | True to get the first non-empty set of bindings, false to get the innermost bindings even when empty. |
| **Returns** | The innermost local variable bindings of the current context. |

Number getNum(String varname)
Look up the value of a QF-Test variable, similar to `lookup()`, but treat it as a number, i.e. as int or float for Jython and as Integer or BigDecimal for Groovy.
Parameters

varname	The name of the variable.
Returns	The value of the variable.

Number getNum(String group, String name)
Look up the value of a QF-Test resource or property, similar to `lookup()`, but treat it as a number, i.e. as int or float for Jython and as Integer or BigDecimal for Groovy.
Parameters

group	The name of the group.
name	The name of the resource or property.
Returns	The value of the resource or property.

Object getOption(String name)
Get an option value at run time. This method is provided more for the sake of completeness, you will probably not need it. For the obvious use case of restoring the value of an option to its previous value after a change with `setOption` you should use `unsetOption` instead because values set at script level hide values set interactively in the options dialog. For temporary changes to an option best use `pushOption` / `popOption`.
Parameters

name	The name of the option, a constant from the `Options` class which is automatically imported in Jython and Groovy scripts. The names of the options that can be read in this way are documented in chapter 39[(417)].
Returns	The current value of the option.

Object getOverrideElement(String id)
SUT only. Get the overridden target GUI element for the given ID.
Parameters

id	The QF-Test ID or SmartID previously used to override the GUI element.
Returns	The GUI element previously registered for the given ID. None/null if no GUI element was registered or the element is no longer valid.

Pattern getPattern(String varname)

Look up the value of a QF-Test variable, similar to `lookup()`, but treat it as a regular expression.

Parameters

varname	The name of the variable.
Returns	A Java Pattern-Object with the value of the variable as regular expression.

Pattern getPattern(String group, String name)

Look up the value of a QF-Test resource or property, similar to `lookup()`, but treat it as a regular expression.

Parameters

group	The name of the group.
name	The name of the resource or property.
Returns	A Java Pattern-Object with the value of the resource or property as regular expression.

Properties getProperties(String group)

Get a set of loaded properties or resources.

Parameters

group	The group name of the properties or resources.
Returns	The variables bound for the given group or null if no such group exists.

String getPropertyGroupNames()

List all available property group names defined by the user. Names are returned in alphabetic order.

Returns	A string listing all the names of all user defined property groups. Names are sorted alphabetically and separated by newlines.

`String getStr(String varname, boolean expand=true)`
Look up the value of a QF-Test variable, similar to `lookup()`.
Parameters

`varname`	The name of the variable.
`expand`	Whether to expand the value of the variable recursively. If, for example, the value of `$(varname)` is the literal string `"$(othervar)"`, this method will return the expanded value of `$(othervar)` if `expand` is true and `"$(othervar)"` itself if `expand` is false. Note that if you want to set this parameter, you must use Python keyword syntax to avoid conflicts with `getStr(String group, String name)`, i.e. `rc.getStr("var", expand=0)` instead of `rc.getStr("var", 0)`.
Returns	The value of the variable.

`String getStr(String group, String name, boolean expand=true)`
Look up the value of a QF-Test resource or property, similar to `lookup()`.
Parameters

`group`	The name of the group.
`name`	The name of the resource or property.
`expand`	Whether to expand the value of recursively. If, for example, the value of `${group:name}` is the literal string `"$(othervar)"`, this method will return the expanded value of `$(othervar)` if `expand` is true and `"$(othervar)"` itself if `expand` is false.
Returns	The value of the resource or property.

`String id(String id)`
Return the QF-Test ID of a specified component. This method should be used to take care that this QF-Test component ID becomes updated when moving or changing the QF-Test ID of the referenced component.
Parameters

`id`	The QF-Test component ID.
Returns	The QF-Test component ID.

boolean isOptionSet(String name)

Test whether an option has been set at script level.

Parameters

name	The name of the option, a constant from the `Options` class which is automatically imported in Jython and Groovy scripts. The names of the options that can be read in this way are documented in <u>chapter 39</u>[(417)].
Returns	True if the option has been set, false otherwise.

boolean isResetListenerRegistered(ResetListener listener)

Server only. Checks if a `ResetListener` is registered.

Parameters

listener	The `ResetListener` to check, if it is registered.
Returns	True if the ResetListener has been registered, otherwise False.

void logDiagnostics(String client)

Server only. Adds event information stored in the SUT for possible error diagnosis to the run log.

Parameters

client	The name of the SUT client from which to get the information.

void logError(String msg, boolean nowrap=false)

Add a user-defined error message to the run log.

Parameters

msg	The message to log.
nowrap	If true, lines of the message will not be wrapped in the report. Use for potentially long messages.

void logImage(ImageRep image, String title=None, boolean dontcompactify=false, boolean report=false)

Add an `ImageRep` (see <u>section 52.10.1</u>[(1096)]) object to the run log.

Parameters

image	The `ImageRep` object to log.
title	An optional title for the image.
dontcompactify	If true, the message will never be removed from a compact run log.
report	True to log the image in the report (implies dontcompactify).

**void logMessage(String msg, boolean dontcompactify=false,
boolean report=false, boolean nowrap=false)**

Add a plain message to the run log.

Parameters

msg	The message to log.
dontcompactify	If true, the message will never be removed from a compact run log.
report	If true, the message will appear in the report.
nowrap	If true, lines of the message will not be wrapped in the report. Use for potentially long messages.

**void logWarning(String msg, boolean report=true, boolean
nowrap=false)**

Add a user-defined warning message to the run log.

Parameters

msg	The message to log.
report	If true (the default), the warning will be listed in the report. Set this to false to exclude this specific warning from the report.
nowrap	If true, lines of the message will not be wrapped in the report. Use for potentially long messages.

String lookup(String varname, boolean expand=true)

Look up the value of a QF-Test variable, similar to `$(varname)`.

Parameters

varname	The name of the variable.
expand	Whether to expand the value of the variable recursively. If, for example, the value of `$(varname)` is the literal string `"$(othervar)"`, this method will return the expanded value of `$(othervar)` if expand is true and `"$(othervar)"` itself if expand is false. Note that if you want to set this parameter, you must use Python keyword syntax to avoid conflicts with `lookup(String group, String name)`, i.e. `rc.lookup("var", expand=0)` instead of `rc.lookup("var", 0)`.
Returns	The value of the variable.

String lookup(String group, String name, boolean expand=true)

Look up the value of a QF-Test resource or property, similar to `${group:name}`.

Parameters

group	The name of the group.
name	The name of the resource or property.
expand	Whether to expand the value of recursively. If, for example, the value of `${group:name}` is the literal string `"$(othervar)"`, this method will return the expanded value of `$(othervar)` if `expand` is true and `"$(othervar)"` itself if `expand` is false.
Returns	The value of the resource or property.

void overrideElement(String id, Component com)

SUT only. Override the target GUI element for component recognition for an element with the given ID. When that QF-Test ID or SmartID is referenced, QF-Test ignores all associated information and directly returns the given element.

Invalidated components are unregistered automatically.

Parameters

id	The QF-Test ID or SmartID of the GUI element to override.
com	The GUI element to return as the resolved target. None/null to revert to the default mechanism.

void popOption(String name)

Negates a preceding call to `pushOption`.

Parameters

name	The name of the option to unset, a constant from the `Options` class which is automatically imported in Jython and Groovy scripts. The constants for options that can be set in this way are documented in chapter 39[(417)].

void pushOption(String name, object value)

Set an option value at runtime, similar to `setOption`. In contast to the latter, the preceding value is saved for each nested call and can be restored via `popOption`. The `pushOption` and `popOption` calls, which are best placed into a Try[616] / Finally[623] combination, are ideal for temporarily changing an option value without negating a preceding `setOption` call.

Parameters

name	The name of the option, a constant from the `Options` class which is automatically imported in Jython and Groovy scripts. The names of the options that can be set in this way are documented in chapter 39[417].
value	The value to set, typically a boolean, a number or a constant from the `Options` class for options edited via a drop-down list. For hotkey options like the hotkey for pausing test run ("Don't Panic" key) this value should be a string like "F12" or "Shift-F6". Supported modifiers are "Shift", "Control" or "Ctrl", "Alt" and "Meta" and combinations thereof. Key specifiers are prepended with "VK_" and then looked up in the class `java.awt.event.KeyEvent`. Case is irrelevant for both, so "shift-alt-enter" will work as well.

void removeResetListener(ResetListener listener)

Server only. Remove a `ResetListener`.

Parameters

listener	The `ResetListener` to remove.

void removeTestRunListener(TestRunListener listener)

Remove a `TestRunListener` from the current run context.

Parameters

listener	The listener to remove.

void resetDependencies(String namespace=None)

Completely reset the dependency stack without executing any cleanup.

Parameters

namespace	An optional namespace to reset the dependencies for.

void resolveDependency(String dependency, String namespace=None, dictionary parameters=None)

Resolve a Dependency[(548)].

Parameters

dependency	The fully qualified name of the Dependency to resolve.
namespace	An optional namespace to resolve the Dependency in.
parameters	The parameters for the Dependency. This should be a dictionary. Its keys and values can be arbitrary values. They are converted to strings for the call.

void rollbackAllDependencies()

Unroll the dependency stacks in all namespaces. This is done in reverse order of their initialization, except for the one in the general name space, which will always be unrolled last.

void rollbackDependencies(String namespace=None)

Unroll the dependency stack.

Parameters

namespace	An optional namespace to unroll the dependencies in.

void setGlobal(String name, object value)

Define a global QF-Test variable.

Parameters

name	The name of the variable.
value	An arbitrary value for the variable. It is automatically converted to a string. A value of None unsets the variable.

void setLocal(String name, object value)

Define a local QF-Test variable.

Parameters

name	The name of the variable.
value	An arbitrary value for the variable. It is automatically converted to a string. A value of None unsets the variable.

void setOption(String name, object value)
Set an option value at run time. Any value thus set overrides the value read from the system configuration file or set via the option dialog, but is never shown in the option dialog or saved to a configuration file. The default value can be restored via `unsetOption`. The value of a possibly preceding call to `setOption` gets overwritten. In case that value should be restored, `pushOption` / `popOption` must be used instead.

Parameters

name	The name of the option, a constant from the `Options` class which is automatically imported in Jython and Groovy scripts. The names of the options that can be set in this way are documented in chapter 39[417].
value	The value to set, typically a boolean, a number or a constant from the `Options` class for options edited via a drop-down list. For hotkey options like the hotkey for pausing test run ("Don't Panic" key) this value should be a string like "F12" or "Shift-F6". Supported modifiers are "Shift", "Control" or "Ctrl", "Alt" and "Meta" and combinations thereof. Key specifiers are prepended with "VK_" and then looked up in the class `java.awt.event.KeyEvent`. Case is irrelevant for both, so "shift-alt-enter" will work as well.

void setProperty(String group, String name, object value)
Set the value of a resource or property in a group.

Parameters

group	The name of the group. A new group is created automatically if necessary.
name	The name of the resource or property.
value	An arbitrary value for the property. It is automatically converted to a string. A value of `None` unsets the property. This method also works for the special groups 'system' and 'env' and can be used as a means to set environment variables and system properties. Values in the special group 'qftest' cannot be set or changed that way.

void skipTestCase()
Stop the execution of the current test case and mark it as skipped.

void skipTestSet()
Stop the execution of the current test set and mark it as skipped.

void stopTest()
Terminate the current test run.

`void stopTestCase(boolean expectedFail=false)`

Stop the execution of the current test case.

Parameters

`expectedFail`	If true, mark possible errors in this test case as expected failures.

`void stopTestSet()`

Stop the execution of the current test set.

`void syncThreads(String name, int timeout, int count=-1,`
`boolean throw=true, int remote=0)`

Server only. Synchronize a number of parallel threads for load testing. The current thread is blocked until all threads have reached this synchronization point or the timeout is exceeded. In the latter case, a <u>TestException</u>[850] is thrown or an error logged.

Parameters

`name`	An identifier for the synchronization point.
`timeout`	The maximum time to wait in milliseconds.
`count`	The number of threads to wait for. Default value -1 means all threads in the current QF-Test instance.
`throw`	Whether to throw an exception (default) or just log an error if the timeout is exceeded without all threads reaching the synchronization point.
`remote`	The number of QF-Test instances - potentially running on different machines - to synchronize. Default 0 means don't do remote synchronization.

`void toServer(...)`

SUT only. Set some global variables in the Jython or Groovy interpreter of QF-Test. Each argument can be any of:

A string

> This is treated as the name of a global variable in the local interpreter. The variable by the same name in QF-Test's interpreter is set to its value.

A dictionary with string keys

> For each key in the dictionary, a global variable by that name is set to the corresponding value from the dictionary.

A keyword argument in the form `name=value`

> The global variable named `name` is set to `value`.

`void toSUT(String client, ...)`
Server only. Set some global variables in the Jython or Groovy interpreter of the SUT. Except for `client`, each argument can be any of:

A string
> This is treated as the name of a global variable in the local interpreter. The variable by the same name in SUT's interpreter is set to its value.

A dictionary with string keys
> For each key in the dictionary, a global variable by that name is set to the corresponding value from the dictionary.

A keyword argument in the form `name=value`
> The global variable named `name` is set to `value`.

Parameters

`client`	The name of the SUT client.

`void unsetOption(String name)`
Restore an option value by removing a possible override from a previous call to `setOption`.
Parameters

`name`	The name of the option to unset, a constant from the `Options` class which is automatically imported in Jython and Groovy scripts. The constants for options that can be set in this way are documented in chapter 39[417].

48.6 The `qf` module

In some cases there is no run context available, especially when implementing some of the extension interfaces described in the following sections. The module `qf` enables logging in those cases and also provides some generally useful methods that can be used without depending on a run context. Following is a list of the methods of the `qf` module in alphabetical order. Unless mentioned otherwise, methods are available in Groovy and Jython and for both Server script and SUT script nodes.

Note   Please note that the Groovy syntax for keyword parameters is different from Jython and requires a ':' instead of '='. The tricky bit is that, for example, `qf.logMessage("bla", report=true)` is perfectly legal Groovy code yet doesn't have the desired effect. The '=' here is an assignment resulting in the value true, which is simply passed as the second parameter, thus the above is equal to `qf.logMessage("bla", true)` and the true is passed to `dontcompactify` instead of `report`. The correct Groovy version is `qf.logMessage("bla", report:true)`.

Pattern asPattern(String regexp)

This method interprets the input as regular expression and returns the corresponding Java Pattern object. Valid input values are defined in the Java API of the Pattern object.

Parameters

regexp	The regular expression
Returns	A Pattern object, which can be used for string comparisons.

String getClassName(Object objectOrClass)

Get the fully qualified name of the Class of a Java object, or of a Java class itself. Mostly useful for Jython where getting the name of a class can become a real hassle.

Parameters

objectOrClass	The Java object or class to get the class name for.
Returns	The class name or None in case something non-Java is passed in.

Object getProperty(Object object, String name)

Get a property for an object that was previously set via setProperty.

Parameters

object	The object to get the property for.
name	The name of the property.
Returns	The property value.

boolean isInstance(Object object, String className)

This is a simple alternative to instanceof in Groovy and isinstance() in Jython that deliberately compares class and instance names only so conflicts with differing class loaders are avoided.

Parameters

object	The object to check.
className	The name of the class or interface to test for.
Returns	True if the object is an instance of the given class or implements the given interface.

void logError(String msg, boolean nowrap=false)

Add a user-defined error message to the run log. If a run context is available it is used and logging takes effect immediately. Otherwise the message is buffered and logged at the next opportunity.

Parameters

msg	The message to log.
nowrap	If true, lines of the message will not be wrapped in the report. Use for potentially long messages. This parameter has no effect if the message needs to be buffered.

---

void logMessage(String msg, boolean dontcompactify=false, boolean report=false, boolean nowrap=false)

Add a plain message to the run log. If a run context is available it is used and logging takes effect immediately. Otherwise the message is buffered and logged at the next opportunity.

Parameters

msg	The message to log.
dontcompactify	If true, the message will never be removed from a compact run log.
report	If true, the message will appear in the report.
nowrap	If true, lines of the message will not be wrapped in the report. Use for potentially long messages. This parameter has no effect if the message needs to be buffered.

---

void logWarning(String msg, boolean report=true, boolean nowrap=false)

Add a user-defined warning message to the run log. If a run context is available it is used and logging takes effect immediately. Otherwise the message is buffered and logged at the next opportunity.

Parameters

msg	The message to log.
report	If true (the default), the warning will be listed in the report. Set this to false to exclude this specific warning from the report.
nowrap	If true, lines of the message will not be wrapped in the report. Use for potentially long messages. This parameter has no effect if the message needs to be buffered.

---

void print(Object object, ...)

Prints a string or the string representation of an object to the terminal. If more than one object is specified there representations are joint with a space character. In contrast to a simple `print` statement, the text is not transferred using the standard output stream.

Parameters

object	The object, which should be printed.

---

void println(Object object)

Prints a string or the string representation of an object to the terminal, and starts a new line. If more than one object is specified there representations are joint with a space character. In contrast to a simple `println` statement, the text is not transferred using the standard output stream.

Parameters

object	The object, which should be printed.

---

void setProperty(Object object, String name, Object value)

Set an arbitrary property for an object. For Swing, SWT or web components the value is stored in the respective user data via `putClientProperty`, `setData` or `setProperty` respectively. For everything else a `WeakHashMap` is used. Either way the property will not prevent garbage collection of the object.

Parameters

object	The object to set the property for.
name	The name of the property.
value	The value to set. Null to remove the property.

String toString(Object object, String nullValue)

Get the string representation of an object. Mostly useful for Jython but sometimes also useful for Groovy thanks to the default conversion of null to the empty string.

Parameters

object	The object to get the string representation for.
nullValue	The value to return if object is None, the empty string by default.
Returns	Jython 8-bit or Unicode strings are returned unchanged, Java objects are turned into a string via `toString`. In Jython, everything else is converted into an 8-bit Jython string.

48.7 Image API

3.0+

The Image API provides classes and interfaces to take screenshots, to save or load images or for own image comparisons. The image API is designed so that the different methods in general do not throw any exception. Instead, the different methods are logging warnings.

48.7.1 The `ImageWrapper` class

For taking screenshots you can use the Jython class ImageWrapper, located in the module `imagewrapper.py`, which comes with the QF-Test installation.

Here is a short sample Jython script demonstrating the usage of the Image API:

```
from imagewrapper import ImageWrapper
#create ImageWrapper instance
iw = ImageWrapper(rc)
#take screenshot of the whole screen
currentScreenshot = iw.grabScreenshot()
#save screenshot to a file
iw.savePng("/tmp/screenshot.png", currentScreenshot)
```

Example 48.2: Image API in Jython

And the same in Groovy:

```
import de.qfs.ImageWrapper
def iw = new ImageWrapper(rc)
def currentScreenshot = iw.grabScreenshot()
iw.savePng("/tmp/screenshot.png", currentScreenshot)
```

Example 48.3: Image API in Groovy

Following is a list of the methods of the `ImageWrapper` class in alphabetical order. The syntax used is a bit of a mixture of Java and Python. Python doesn't support static typing, but the parameters are passed on to Java, so they must be of the correct type to avoid triggering exceptions. If a parameter is followed by an '=' character and a value, that value is the default and the parameter is optional.

ImageWrapper ImageWrapper(RunContext rc)

Constructor method of the `ImageWrapper` class.

Parameters

rc	The current run context of QF-Test.

---

int getMonitorCount()

Return the number of monitors.

Returns	The total number of monitors.

ImageRep grabImage(Object com, int x=None, int y=None, int width=None, int height=None)
Take screenshot of a given component. If you use the parameters x, y, width and height, you can take a screenshot of a specific region of the component.
Parameters

com	The QF-Test ID of the component to take a screenshot from.
x	The X coordinate of the left upper corner of the region to take the screenshot.
y	The Y coordinate of the left upper corner of the region to take the screenshot.
width	The width of the region to take the screenshot.
height	The height of the region to take the screenshot.
Returns	An `ImageRep` object containing the actual screenshot.

ImageRep grabScreenshot(int x=None, int y=None, int width=None, int height=None)
Take screenshot of the whole screen. If you use the parameters x, y, width and height, you can take a screenshot of a specific region of the screen.
Parameters

x	The X coordinate of the left upper corner of the region to take the screenshot.
y	The Y coordinate of the left upper corner of the region to take the screenshot.
width	The width of the region to take the screenshot.
height	The height of the region to take the screenshot.
Returns	An `ImageRep` object containing the actual screenshot.

ImageRep[] grabScreenshots(int monitor=None)
Take screenshots of all available screens. This procedure might be useful, if you work with more than one screen.
If you want to take a screenshot of one specific screen, you can also use this procedure.
Parameters

monitor	Index of the monitor to take the screenshot from. The first monitor has 0, the second 1 and so on.
Returns	An array of `ImageRep` objects of all screenshots or the specific `ImageRep` object, if the `monitor` parameter has been used.

`ImageRep loadPng(String filename)`
Load an image from a given file return an `ImageRep` object containing this image. The file has to contain the image in PNG format.
Parameters

`filename`	The path to the file, where the image is stored.
Returns	An `ImageRep` object containing loaded image.

`void savePng(String filename, ImageRep image)`
Save the given `ImageRep` object to a file. The file will be in PNG format.
Parameters

`filename`	The path to the file, where the image should be stored to.
`image`	The `ImageRep` object to store.

48.8 Exception handling

All QF-Test exceptions listed in chapter 41[(850)] are automatically imported in Jython scripts and can be used for `try/except` clauses like

```
try:
    com = rc.getComponent("someId")
except ComponentNotFoundException:
    ...
```

When working with Groovy you must first import the exception:

```
import de.qfs.apps.qftest.shared.exceptions.
    ComponentNotFoundException
try {
    com = rc.getComponent("someId")
} catch (ComponentNotFoundException) {
    ...
}
```

Only the following exceptions should be raised explicitly from script code (with `raise` or `throw new` respectively):

- `UserException("Some message here...")` should be used to signal exceptional error conditions.

- `BreakException()` or `raise BreakException("loopId")` can be used to break out of a Loop[(597)] or While[(600)] node, either without parameters to break out of

the innermost loop or with the QF-Test loop ID parameter to break out of a specific loop with the respective QF-Test ID.

- `ReturnException()` or `raise ReturnException("value")` can be used to return - with or without a value - from a Procedure[(586)] node, similar to executing a Return[(592)] node.

48.9 Debugging scripts (Jython)

When working with Jython modules you don't have to restart QF-Test or the SUT after you made changes. You can simply use `reload(<modulename>)` to load the module anew.

Debugging scripts in an embedded Jython interpreter can be tedious. To simplify this task, QF-Test offers an active console window for communicating with each interpreter. For more information please see the last part of section 11.1[(162)].

Alternatively, a network connection can be established to talk remotely to the Jython interpreter - in QF-Test as well as within the SUT - and get an interactive command line. To enable this feature you must use the command line argument `-jythonport <number>`[(873)] to set the port number that the Jython interpreter should listen on. For the SUT `-jythonport=<port>` can be defined in the "Extra" Executable parameters[(637)] of the Start Java SUT client[(635)] or Start SUT client[(639)] node. You can then connect to the Jython interpreter, for example with

```
telnet localhost <port>
```

Combined with Jython's ability to access the full Java API, this is not only useful for debugging scripts but can also be used to debug the SUT itself.

Chapter 49

Web

This chapter addresses topics that are only relevant when testing web applications in a browser.

49.1 Improving component recognition with a `CustomWebResolver`

Video:

'CustomWebResolver in QF-Test'
https://www.qfs.de/en/yt/no-rocket-science-special-webinar.html

HTML is a very flexible language for describing the content of web pages. But unfortunately there are no real standards with regard to components which should be used to draw a button, a text field or a table showing data. As a result nearly every framework implements its own way of drawing such components. This means the HTML structure (the so-called DOM tree) looks different for every web framework. In order to allow QF-Test to identify the components as buttons or data tables etc. we need some kind of dictionary. The dictionary should work as a translator for the properties of the HTML components to the QF-Test vocabulary.

QF-Test offers a generic component resolver to be configured freely, the `CustomWebResolver`, allowing you to adapt the component recognition of QF-Test without too great an effort to the specific needs of your web page.

Before starting to set up test cases you should check the component recognition and maybe improve it. We recommend the following approach:

1. Record GUI objects you want to interact with in the test on different web pages of the application.

2. Check the recorded QF-Test components

   - whether they were mapped to Generic classes[(1186)],

   - if they have sufficient recognition criteria (Name and Feature attributes, `qfs:label` in Extra features),

   - for the depth of the component hierarchy,

   - whether complex components such as tables, lists, trees etc. have been mapped as such and the sub-items were addressed via index. You will find detailed information on the standard recording of HTML elements and criteria for deciding whether it is suffcient in Recognition of web components and toolkits[(202)].

3. In case you identified weak points, check the respective GUI objects of different web pages trying to identify their characteristic attributes to be mapped to generic QF-Test classes, as well as 'good' attributes for the Name and Feature of the QF-Test component. You can use the UI Inspector[(97)] to analyze the web elements.

4. Configure the component mapping as described in The Install CustomWebResolver node[(949)].

5. In case tests already exist: Update existing Component[(823)] nodes, see Update Components[(94)].

49.1.1   General configuration

The Install CustomWebResolver node[(949)] offers the following functionality:

Mapping of HTML objects to functional components
You can map functional GUI objects as buttons, text fiels, data tables etc. to QF-Test components of a certain generic class. Advantages:
- recording of additional criteria for recognition,
- class specific checks,
- indexing of sub-items during recording,
- recording of generic class types,
- better component recognition via specific generic QF-Test classes vs. common HTML classes.

The data recorded for a certain object depends on its generic class as described in Generic classes[(1186)].

Reduction of the recorded component hierarchy
You can tell the recording algorith to ignore certain HTML components in the

component's hierarchy. This reduces the complexity of a recorded web page from the full HTML structure to a component hierarchy relevant for recognition or structuring. The video

'Dealing with the explosion of complexity in web test automation' https://www.qfs.de/en/yt/web-test-automation-40.html

gives you a good idea of how QF-Test handles a deeply nested DOM structure.

| Current recording | Simplified recording |
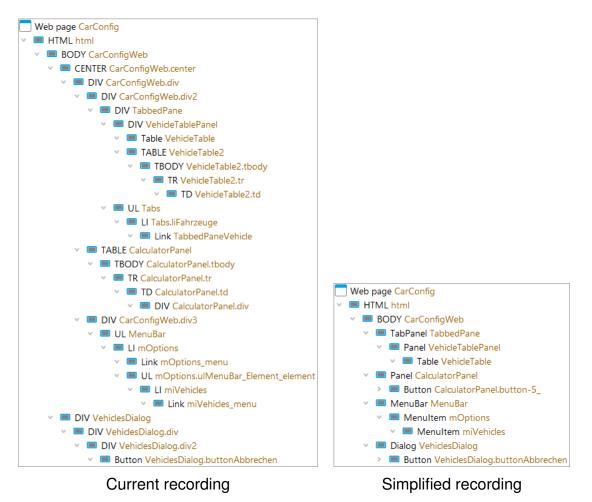
Figure 49.1: Reduction of complexity for "CarConfigurator Web" demo

Using alternative attributes for ids

By default QF-Test uses the HTML attributes `id` or `name` to identify a component and saves them in the Name attribute of the Component node. You can configure Install CustomWebResolver[(796)] to use a different HTML attribute as id.

Specification of further attributes for recognition   You can specify an attribute useful for component recognition, the value of which QF-Test

will save to the Feature attribute of the component.

You can use the following HTML features to identify a GUI object:

- the `class` attribute,

- any other attribute,

- the HTML tag.

The mapping of the HTML attributes can be subject to certain conditions. QF-Test offers the following options to set conditions for single mappings. It is possible to combine them.

- The use of regular expressions.

- Map only if the object is a child component (at a given depth) of another object of a given class.

- Map only if the object has a certain HTML tag, additonally to the other criteria.

A functional component often may consist of nested layers of elements. Some of the layers may have attributes useful for component recognition, others not. For recording and replay it does not matter which layer you map. The main thing is to has attributes for component recognition. QF-Test will also check nested components for further attributes and save them with the mapped QF-Test component. Example: CustomWebResolver – TabPanel and Accordion[(973)]

In addition you get functional components such as combo boxes, lists, tables and trees which have to be implemented in HTML via several objects, named 'complex components' in QF-Test, i.e. a list, where you need to tell QF-Test which HTML object will be the list container and which HTML objects will be the list items.

The following sections provide a list of mandatory and optional HTML elements which need to be mapped for a complex component to be recognized as such, each containing a comprehensive example.

- CustomWebResolver – Combo boxes[(971)]

- CustomWebResolver – Lists[(969)]

- CustomWebResolver – Tables[(961)]

- CustomWebResolver – TabPanel and Accordion[(973)]

- CustomWebResolver – Tree[(964)]

In most cases the HTML attribute `class` is significant for component recognition and provides information about the functional type of the component, sometimes it is other attributes. There are also some frameworks where you can only access this information through special JavaScript methods. For adapting QF-Test to those frameworks you need to implement other resolvers in addition to a `CustomWebResolver`. In this chapter, the focus is on the first two. For adapting QF-Test to more complicated cases, please get in touch with our support team.

49.1.2   The Install CustomWebResolver node

The mapping of HTML objects to Generic classes[(1186)] is usually done via the Install CustomWebResolver[(796)] node.

7.0+   Before QF-Test version 7, this mapping was achieved by a call to the Procedure `installCustomWebResolver`[(841)] and `updateCustomWebResolverProperties` from the `qfs.qft` standard library. These procedure calls should now be converted to a Install CustomWebResolver node. Before conversion of the procedure call, the contained parameters are automatically checked to uncover possible invalid assignments and to facilitate the switch to the Install CustomWebResolver node. If your procedure call contains variables, you must provide a run log during conversion which contains the desired variable values. If your procedure call contains invalid entries as comments, you may have to remove them before conversion or put them back in the desired location after conversion.

If you used the Quickstart Wizard from the Extras menu to create the setup sequence for your web application, as recommended, you will find the Install CustomWebResolver node in the last Sequence node. You should configure this node as required for your application.

Figure 49.2: Installing the CustomWebResolver in the Setup node of the Quickstart Wizard

Please find general information about component recognition in <u>Recognition of web components and toolkits</u>[(202)] as well as in <u>General configuration</u>[(946)].

In the following chapters, the syntax used by the Install CustomWebResolver node and the available configuration categories are explained.

Note

Please note changes in the Install CustomWebResolver node are likely to also change the recognition criteria for a GUI element. Thus they may deviate from the recognition criteria of <u>Component</u>[(823)] nodes already recorded. Therefore, you should update existing Component nodes as described in <u>Update Components</u>[(94)]. Ideally, the configuration of the Install CustomWebResolver node should be done before setting up the tests. Nodes recorded during the configuration phase should be deleted altogether.

Install CustomWebResolver node – Syntax

The Install CustomWebResolver node is configured in a text area using the YAML syntax. Knowledge of the basic functionality of YAML is neccessary, but it will be explained where neccessary below.

On the top level there are the configuration categories (see the following sections). These are written on their own line and are followed by a colon (dictionary keys). On the second level, the entries each start on a new line, beginning with a hyphen (list items). Further levels are indicated via indentation.

To facilitate working with the YAML configuration, various templates can be inserted via the toolbar above the editor.

Client

$(client)

Figure 49.3: CustomWebResolver configuration templates

The menu, which can be opened via the edit button next to the line numbers, is context-sensitive. It contains any available actions for the respective line of configuration. If you work with this menu, you will always have the full overview of available actions and automatically achieve the correct syntax.

If you followed Quickstart your application[28] to create the startup sequence and left the framework selection on the default setting, you will receive a configuration with two categories and two entries on the second level, plus a few explanatory comments:

Figure 49.4: CustomWebResolver with a template for `genericClasses`

In the following configuration, the line after the category `genericClasses` was selected and then the edit menu to the left of the line numbers was used to insert a template for a generic class (comments removed for brevity).

Figure 49.5: CustomWebResolver with two generic classes
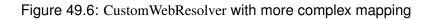
In the next step, the generic class `List` was entered, as well as the CSS class `datalist`. HTML elements with this CSS class will now be assigned this generic class during component recognition. This process was repeated for the generic class

`Item:ListItem`. It will be assigned to each GUI element with the HTML tag `LI`. Normally, only elements with this tag that are inside a `List` component should be considered. Therefore, the next step is to use the edit button ✏ and choose the entry "Add ancestor". As you can see, the syntax for the entry changes: As soon as more than one characteristic is needed for the mapping, the first mapping is moved to the next level with the appropriate prefix, and the additional characteristic is added to the same level.

Figure 49.6: CustomWebResolver with more complex mapping

CustomWebResolver configuration categories

Every CustomWebResolver is based on a globally defined default configuration with a set of generally applicable rules. You can inspect part of these defaults in the included library `qftest-7.1.2/include/qfs-resolvers.qft` in the procedure `qfs.web.cwr.helpers.default`.

You can find a complete list of all available configuration categories in QF-Test in the Install CustomWebResolver node behind the "New mapping" button.

The following sections briefly introduce the most important configuration categories. Please keep in mind that all functionality of the different categories can be explored via the context-sensitive edit button ✏. Not every possible permutation of the syntax is described here.

Configuration category `base`

Contains the short name of the base resolver which serves as the foundation of the configuration:

- `autodetect`: automatic detection of the framework used. Falls back to `custom`

if no supported framework was detected.

- `custom`: No framework should be used as a basis for the configuration. In this case, `base` can also be omitted completely.

- The short name of the framework, for example `vaadin`:
  The mappings shipped with QF-Test in `qftest-7.1.2/include/qfs-resolvers.qft` for the respective framework are used. In your configuration, these can be supplemented with your own mappings.

  You can find the short names of each supported framework in the table table 49.7[(989)]. If you create a setup sequence with the Quickstart Wizard and choose a framework, its short name is inserted here.

  You can also use the full name of a procedure as a short name. This procedure should contain you own base resolver for the application under test. It is also possible to create multiple procedures and orchestrate them freely in one Install CustomWebResolver node. You can even combine them with a base resolver predefined by QF-Test, but the predefined resolver must usually be at the start of the list. The configurations will then be applied in the given order.

```
base:
- vaadin
- myResolvers.Panels
- myResolvers.otherClasses
```

Example 49.1: List of base resolvers

Configuration category `genericClasses`

In this category, the recognition criteria are defined, on the basis of which a specific generic class is assigned to a GUI element. The respective properties of the generic classes are explained in chapter 59[(1186)].

Generic classes can receive a type extension. It is used for the mapping of some HTML elements. For example, `Item:ListItem` refers to a list item, `Button:ComboBoxButton` refers to a button that opens a combo box. Type extensions are also interesting because you can define you own types. The example in CustomWebResolver – Tables[(961)] uses this technique.

Note   When using a class with a type extension in a SmartID, the colon before any custom type extension must be escaped with a backslash, see SmartID syntax for Class name[(76)].

The given entries are evaluated from top to bottom, and for each HTML element the topmost matching generic class is used. The only exception to this rule are entries with `ancestor`, which are always evaluated first.

The tag name and the attributes of an HTML element are the most basic elements of component recognition. The `class` attribute has a special role. It contains the CSS classes that influence the display of the GUI element in the browser and are thus often characteristic of a particular GUI element class.

Install CustomWebResolver offers ways to create mappings for each of these cases:

CSS class

The CSS class refers to an entry in the attribute `class` of the GUI element. Please note that multiple classes can be separated by spaces in the attribute, but only individual classes are considered here.

Simple mapping: The CSS class is entered on the same line as the generic class.

```
genericClasses:
- Button: btn
```

Example 49.2: Simple mapping of a CSS class to a generic class

In the example, only HTML elements with the CSS class `btn` receive the generic class `Button`.

Mapping with multiple criteria: The CSS class is indented in a line below the generic class and prefixed with `css:`.

```
genericClasses:
- Button:
    css: btn
    tag: DIV
```

Example 49.3: Mapping CSS class and tag name to a generic class

In the example, only HTML elements with the CSS class `btn` and the HTML tag name `DIV` receive the generic class `Button`.

HTML tag name

Simple mapping: The tag name is added in angled brackets after the generic class.

```
genericClasses:
- TableCell: <TD>
```

Example 49.4: Simple mapping of a tag name to a generic class

In the example only HTML elements with the tag `TD` receive the generic class `TableCell`.

Mapping with multiple criteria: The tag name is indented in a line below the generic class and prefixed with `tag:`.

```
genericClasses:
- TableCell:
    tag: TD
    ancestor: TableRow
```

Example 49.5: Mapping a tag name with an ancestor to a generic class

In the example, only HTML elements with the tag `TD` receive the generic class `Button` if they are inside a GUI element with the class `TableRow`.

HTML attribute

Simple mapping: The attribute name, an equals sign and the attribute value are added after the generic class.

```
genericClasses:
- TableRow: role=datarow
```

Example 49.6: Simple mapping of an attribute to a generic class

In the example, only HTML elements with the attribute `role` and the value `datarow` are assigned the generic class `TableRow`.

Mapping with multiple criteria: Indented below the generic class a line is added for the attribute name with the prefix `attribute:` and for the attribute value with the prefix `attributeValue:`.

```
genericClasses:
- TableRow:
    attribute: role
    attributeValue: datarow
```

Example 49.7: Mapping an attribute value to a generic class

In the example, only HTML elements with the attribute `role` and the value `datarow` receive the generic class `TableRow`.

Note   The `class` attribute can also be used here. However, then the entire value of the attribute must match for the mapping to apply. For example, if two CSS classes must be present and the others are to be ignored, a regular expression can be used. This is also an example for an additional level of indentation.

```
genericClasses:
- TableRow:
    attribute: class
    attributeValue:
        value: (^|.*\s)btn(\s.*|$)
        regex: true
```

<div align="center">Example 49.8: Mapping an attribute value to a generic class</div>

Ancestor

To make a mapping additionally dependent on the existence of a specific parent element, `ancestor:` is used.

Simple mapping: The class of the container is added with the prefix `ancestor:`.

```
genericClasses:
- TableRow:
    tag: TR
    ancestor: Table
```

<div align="center">Example 49.9: Simple ancestor mapping</div>

In the example, only HTML elements with the HTML tag `TR` receive the generic class `TableRow` if they lie anywhere within an element with the generic class "Table".

Mapping with multiple criteria: Indented below the prefix `ancestor:` follow one or more of the attributes `type:`, `level:` and `className:`.

Available values for `type:` are `ancestor` (arbitrary nesting), `parent` (direct parent element) and `interestingparent` (directly inside the QF-Test parent element determined by `node.getInterestingParent()`).

```
genericClasses:
- TableRow:
    tag: TR
    ancestor:
        type: ancestor
        level: 2
        className: TableRow
```

<div align="center">Example 49.10: Complex ancestor mapping</div>

In the example, only HTML elements with the HTML tag `TR` receive the generic class `TableRow` which are two levels deep inside an element with the generic class "Table".

The attributes `css:` and `attributeName:` can be combined with `tag:`, but not with each other.

If HTML elements with different recognition criteria should receive the same generic class, two entries for that class need to be added:

```
genericClasses:
- TableRow:
    attribute: role
    attributeValue: datarow
- TableRow:
    tag: TR
    ancestor: Table
```

Example 49.11: Same generic class for different HTML elements

`ancestor:` is also available in many other configuration parameters. Check the edit menu ✎ for the entry "Add ancestor".

Configuration category `ignoreTags`

A list of class names or tags for which to ignore nodes when creating the parent hierarchy of a node. To distinguish tags from class names, tags must be written in uppercase letters or between angle brackets.

In the following example, all `DIV` and `TBODY` nodes not mapped to a generic class and not interacted with directly will be ignored.

```
ignoreTags:
- <DIV>
- <TBODY>
```

Example 49.12: ignoreTags

Configuration category `ignoreByAttributes`

A list of HTML attributes and values for which to ignore nodes when creating the parent hierarchy of a node:

```
ignoreByAttributes:
- id: container
```

Example 49.13: ignoreByAttributes

Configuration category `autoIdPatterns`

A list of patterns specifying ids generated automatically by a framework. If the id attribute matches the pattern the value will not be used for the Name attribute of the component:

```
autoIdPatterns:
- myAutoId
- value: auto.*
  regex: true
```

Example 49.14: autoIdPatterns

Configuration category `customIdAttributes`

A list of attribute names which can act as id for a component. Keep in mind that you need to include the attribute "id" here if you only want to augment the default QF-Test behavior.

The following example will make the attribute myid be used for ID resolution.

```
customIdAttributes:
- myid
```

Example 49.15: customIdAttributes

Configuration category `interestingByAttributes`

A list of attribute names and values telling QF-Test to create a node in the component tree for the respective GUI object.

```
interestingByAttributes:
- id: container
- id: header
```

Example 49.16: interestingByAttributes

Configuration category `attributesToQftFeature`

A list of attributes where the values will be used for the Feature attribute of the QF-Test component.

Configuration category `redirectClasses`

In this category you can configure for individual generic classes if events should be redirected to elements of that class or if an ancestor element should be recorded instead. You can also define multiple rules to achieve different behavior depending on the class of the parent element.

Entries are evaluated from top to bottom, and only the first matching entry is applied.

Use carefully. When in doubt, contact the QF-Test support team.

Configuration category `documentJS`

Define JavaScript code to be injected into the web page. Can be used to inject custom JavaScript functions or run certain code after every page load.

In the following example, pay attention to the syntax for multiline strings in YAML. Injected JavaScript code should not contain any empty lines to avoid conflicts with the YAML syntax.

```
documentJS: |-
  window.hello = function() {
      console.log("Hello World");
  }
  hello();
```

Example 49.17: documentJS

Configuration category `attributesToQftName`

A list of attributes which will be used for the Name attribute of components.

Configuration category `nonTrivialClasses`

A list of CSS classes of HTML-Elements which shouldn't be ignored by QF-Test. Trivial nodes are usually I, FONT, BOLD etc. If you want to keep them, you need to activate them here specifying a proper CSS class.

Use carefully. When in doubt, contact the QF-Test support team.

Configuration category `browserHardClickClasses`

A list of QF-Test classes whose components should always receive hard or semi-hard events during playback. For example, the entry `Button` will play back hard clicks on buttons. Can also be limited to certain browsers.

Configuration category `treeResolver`

This category bundles configuration options which control how QF-Test handles tree nodes in `Tree` and `TreeTable`. Use this category if QF-Test has trouble differentiating levels of hierarchy in trees, expanding individual tree nodes or reading the correct text content of tree nodes in your application.

Configuration category `treetableResolver`

This category bundles configuration options which control how QF-Test handles tree nodes in `TreeTable` components. You can for example define the index of table columns in your application a tree can be located, in case QF-Test cannot find it automatically.

49.1.3   CustomWebResolver – Tables

In order to resolve Table components correctly, it is necessary to map the component containing all entries, i.e. the table itself as well as the components which represent the individual rows of a table and the individual table cell entries. Furthermore, the row containing all headings as well as the specific headings need to be mapped to generic classes.

Class	Required components and sub-items
Table	Represents the Table component, contains all rows and cells.
TableRow	Represents a table row.
TableCell	Represents a table cell.
TableHeader	Represents a row of the table headers.
TableHeaderCell	Represents a header cell.
	Optional sub-items
CheckBox:TableCellCheckBox	(Optional) Represents a CheckBox in table cell.
Icon:TableCellIcon	(Optional) Represents an Icon in a table cell.
CheckBox:TableCellHeaderCheckBox	(Optional) Represents a CheckBox of a table header cell.
Icon:TableCellHeaderIcon	(Optional) Represents an Icon of a table header cell.

Table 49.1: Mapping of Tables

In addition to the following example you will find a detailed instruction for mapping a table in <u>Mapping of complex components like data tables</u>[982].

Example:

The following HTML code defines two tables, one as a data table and the other one for the layout of buttons:

```
<div role="datatablecontainer">
    <table>
        <th type="header">
            <td class="datacell">Form</td>
            <td class="datacell">Color</td>
        </th>
        <tr>
            <td class="datacell">Square</td>
            <td>Red</td>
        </tr>
        <tr>
            <td class="datacell">Diamond</td>
            <td class="datacell">Blue</td>
        </tr>
    </table>
</div>
<table>
     <tr>
        <td>
            <div class="button">Save</div>
            <div class="button">Cancel</div>
        </td>
    </tr>
</table>
```

Example 49.18: HTML Table

The following configuration for the Install CustomWebResolver[(796)] node only maps the data table to a QF-Test table component.

```
genericClasses:
- Button: button
- TableCell:
    css: datacell
    ancestor: TableRow
- Panel:myTablePanel: role=datatablecontainer
- TableHeader:
    attribute: type
    attributeValue: header
    tag: th
- Table:
    tag: table
    ancestor:
        type: parent
        className: Panel:myTablePanel
- TableHeaderCell:
    tag: td
    ancestor: TableHeader
- TableRow:
    tag: tr
    ancestor: Table
ignoreTags:
- <DIV>
- <SPAN>
- <TABLE>
```

Example 49.19: HTML table

In the mapping of `Panel:myTablePanel`, the class type `myTablePanel` was freely "invented" to distinguish the `DIV` element containing the data table from other `DIV`s. This allows us to use `parent:   Panel:myTablePanel` in the mapping of the table.

The mapping for the column title `TableHeader` reads as follows: The attribute `type` with the value `header` will be mapped to the generic class `TableHeader` only if the HTML tag name is `TH`.

To make sure the mappings will only affect HTML elements with the tags `TR` and `TD` that are part of a data table we add `ancestor:` to each.

We do not want to record HTML elements with the HTML tag `TABLE` that are not mapped to a QF-Test Table component. Therefore, we add `<TABLE>` in the category `ignoreTags` in addition to the default entries `<DIV>` and `<SPAN>` which, in turn, make sure that unmapped `DIV` and `SPAN` elements will not be recorded.

49.1.4   CustomWebResolver – Tree

In order to resolve Tree components correctly, it's necessary to map the component containing all entries, which is the tree itself, as well as the components which represent

the individual tree entries. Furthermore, you need to map the toggle button which opens and closes a tree node.

Class	Required components and sub-items
Tree	Represents the Tree component, contains all tree nodes.
TreeNode	Represents a tree node.
Expander:TreeNodeExpander	Represents the toggling component used to open and close the tree node.
	Optional sub-items
Spacer:TreeNodeSpacer	(Optional) Represents the spacing object used to create the indentation of the tree node.
CheckBox:TreeNodeCheckBox	(Optional) Represents a CheckBox of a tree node.
Icon:TreeNodeIcon	(Optional) Represents an Icon in a tree node.

Table 49.2: Mapping of trees

Example:

The "CarConfigurator Web" demo (`qftest-7.1.2/demo/carconfigWeb/carconfigWeb_de.qft`) contains a tree. Please open the specials dialog via the menu ⟦Options→Specials...⟧, select a model and click the Button 'Details'.

When you have a look at the CSS classes recorded by QF-Test or analyze the web page with the section 5.12.2[97] you will find the following HTML structure (slightly simplifies and shortened):

```
<table id="DetailsTree" class="tree">
  <tbody>
    <tr class="treenode even">
      <td>Information</td>
      <td></td>
    </tr>
    <tr class="treenode odd>
      <td>
        <span>
          <span class=treenodespacer">  </span>
          <span class="treetoggler">
            <img src="include/collapse.png" onclick="javascript: ..."></img>
          </span>
          <span>Zubehör</span>
        </span>
      </td>
    </tr>
    <tr class="treenode odd>
      <td>
        <span>
          <span class=treenodespacer">      </span>
          <span>Zentralverriegelung</span>
        </span>
      </td>
    </tr>
  </tbody>
</table>
```

Example 49.20: HTML Tree

In the CarConfigurator demo test suite you will find the mapping for the tree in the
Install CustomWebResolver[(796)] node. It is located in the procedure `startStop.start` in
the last sequence, `Install CustomWebResolver`.
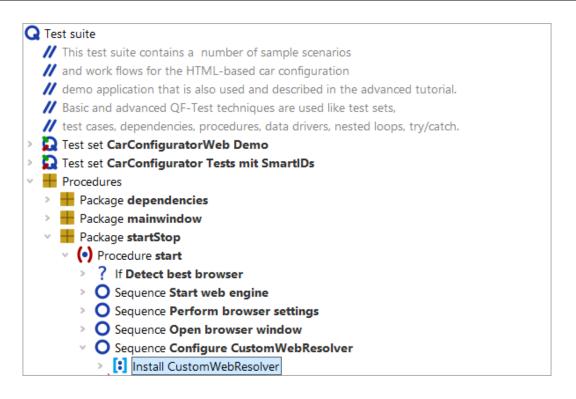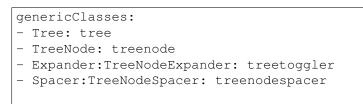
Figure 49.7: CarConfigurator Web

The following CustomWebResolver configuration entries map the tree:

```
genericClasses:
- Tree: tree
- TreeNode: treenode
- Expander:TreeNodeExpander: treetoggler
- Spacer:TreeNodeSpacer: treenodespacer
```

Example 49.21: HTML tree

49.1.5 CustomWebResolver – TreeTable

Tree tables are a combination of table and tree. For them, you map `TreeTable` the same way you would a normal table. Additionally, you need to map the open/close buttons of the tree nodes.

QF-Test assumes that the tree components are located in the first table column. You can adjust this behavior with the configuration category "treetableResolver" of the Install CustomWebResolver.

Class	Required components and sub-items
TreeTable	Represents the TreeTable component, contains all rows and cells.
TableRow	Represents a table row and tree node.
TableCell	Represents a table cell.
TableHeader	Represents a row of the table headers.
TableHeaderCell	Represents a header cell.
Expander:TreeNodeExpander	Represents the toggling component used to open and close the tree node.
	Optional sub-items
TreeNode	(Optional) Represents a tree node, must be located inside a TableCell.
Spacer:TreeNodeSpacer	(Optional) Represents the spacing object used to create the indentation of the tree node.
CheckBox:TreeNodeCheckBox	(Optional) Represents a CheckBox of a tree node.
Icon:TreeNodeIcon	(Optional) Represents an Icon in a tree node.

Table 49.3: Mapping of TreeTables

Example:

A TreeTable could look like this in HTML code:

```
<table id="DetailsTreeTable" class="treetable">
  <thead>
    <tr>
      <th>Information</th>
      <th>Details</th>
    </tr>
  </thead>
  <tbody>
    <tr class="treenode odd>
      <td>
        <span class="treetoggler"></span>
        <span>Item 1</span>
      </td>
      <td>Lorem ipsum</td>
    </tr>
    <tr class="treenode odd>
      <td>
        <span class=treenodespacer">      </span>
        <span>Item 1.1</span>
      </td>
      <td>Dolor sit amet</td>
    </tr>
  </tbody>
</table>
```

Example 49.22: HTML TreeTable

The matching configuration of the `Install CustomWebResolver` node could look like this:

```
genericClasses:
- TreeTable: treetable
- TableRow: <TR>
- TableCell: <TD>
- TableHeader: <THEAD>
- TableHeaderCell: <TH>
- Expander:TreeNodeExpander: treetoggler
```

Example 49.23: TreeTable CWR configuration

49.1.6 CustomWebResolver – Lists

In order to resolve List components correctly, it is necessary to map the component containing all list entries, i.e. the list itself, as well as the components which represent the individual list entries.

Class	Required components and sub-items
List	Represents the List component, contains all list entries.
Item:ListItem	Represents the individual list entry.
	Optional sub-items
CheckBox:ListItemCheckBox	(Optional) Represents a checkbox inside the list entry.
Icon:ListItemIcon	(Optional) Represents an icon inside the list entry.

Table 49.4: Mapping of Lists

In the case of combo boxes you can also map the specific `ComboBoxList:` `List:ComboBoxList` and `Item:ComboBoxListItem`.

Sample:

The following HTML code represents a list:

```
<ul class="datalist">
  <li class="list-item">Entry A</li>
  <li class="list-item">Entry B</li>
  <li class="list-item">Entry C</li>
  <li class="list-item">Entry D</li>
  <li class="list-item">Entry E</li>
</ul>
```

Example 49.24: HTML list

This HTML code offers the choice whether to map the HTML tags or the CSS classes. We highly recommend to use the CSS classes, as this will be a lot more precise. `datalist` strongly hints that the element refers to a list in the sense of QF-Test. With `list-item` we can add `ancestor:  List` to be sure not to map sub-items of other complex components (e.g. with table cells not shown here, which very well could have the CSS class `list-item`)

If we used the HTML tags, this might easily lead to wrong mappings with other components. If such a thing should happen, and you find it difficult to know why a mapping is not working, you can use the procedure `qfs.web.cwr.dumpConfiguration` to display the currently configured mappings.

```
genericClasses:
- List: datalist
- Item:ListItem:
    class: list-item
    ancestor: List
```

Example 49.25: HTML list

49.1.7 CustomWebResolver – Combo boxes

Combo boxes mostly consist of text fields or buttons. If the user clicks that component, a list of items is shown. The user can then select an item. To resolve a combo box correctly, you need to map the combo box with the text field and the components for the list and its items.

HTML `SELECT` nodes will be mapped to combo boxes automatically. The selection of the list item will be recorded as a Selection node.

Class	Required components and sub-items
ComboBox	Container component which contains a text field and the button.
List:ComboBoxList	Represents the List component, contains all list entries.
Item:ComboBoxListItem	Represents the individual list entry.
	Optional sub-items
Button:ComboBoxButton	(Optional) Represents the button opening the selection list.
TextField:ComboBoxTextField	(Optional) Represents the text field receiving text input and showing the selected item.
CheckBox:ComboBoxListItemCheckBox	(Optional) Represents a checkbox inside the list entry.
Icon:ComboBoxListItemIcon	(Optional) Represents an icon inside the list entry.

Table 49.5: Mapping of ComboBoxes

It is sufficient to use the standard mappings for list items for the selection list. See section 49.1.6[(969)].

Sample:

HTML code for a combo box:

```
<div class="combobox-wrapper">
  <div role="combobox" aria-expanded="true" aria-owns="ex1-listbox"
    aria-haspopup="listbox" id="ex1-combobox">
    <input type="text" aria-autocomplete="list" aria-controls="ex1-listbox"
      id="ex1-input" aria-activedescendant="">
  </div>
  <ul aria-labelledby="ex1-label" role="listbox" id="ex1-listbox"
    class="listbox">
    <li class="result" role="option" id="result-item-0">Leek</li>
    <li class="result" role="option" id="result-item-1">Lemon</li>
  </ul>
</div>
```

Example 49.26: HTML combo box

A `ComboBox` consists of a container element showing the current selection and a list of the selectable items. In the example the `DIV` object contains the list plus the `ComboBox` itself. Often the list is defined somewhere completely different in the DOM. You can find it by opening the list and checking the developer tool of the browser. Alternatively, you can get the CSS classes of the list object and the list entries by recording a click to a list item and checking the `qfs:class` entry of the Extra features table of the recorded components.

```
genericClasses:
- List:ComboBoxList: listbox
- Item:ComboBoxListItem:
    class: result
    ancestor: List:ComboBoxList
- ComboBox: role=combobox
```

Example 49.27: HTML combo box

With this `ComboBox` we have several options for the mapping. When you have contact to the developers of the application it is best to ask them which attributes define a certain component class. If not, you have to decide which attributes best represent a certain class. You should check this with other GUI elements of the same type. Also, you should check similar GUI elements for the same attributes. In this case maybe there is another, hopefully unique, attribute, or you need to make it more specific by adding `ancestor:…`

In the example above, `role=combobox` should be precise enough for the ComboBox itself, just as `role=listbox` or alternatively `css:  listbox` for the list. It probably will not make a big difference which one you choose. If normal lists use the same attribute you can omit the class type, i.e. you would just map `List:  listbox` and `Item:ListItem:`. This does not affect the functionality the combo box within QF-Test.

However, `css: result` and `role=option`, used for the list items, could very well be used with other elements, too. Therefore, we will add `ancestor: List:ComboBoxList` to be sure.

49.1.8   CustomWebResolver – TabPanel and Accordion

In order to resolve tab panels or accordions correctly, it is necessary to map the component containing all entries, i.e. the tab panel itself as well as the components which represent the individual tab panel entries.

Class	Required components and sub-items
TabPanel	Represents the tab panel component, contains all tabs.
Item:TabPanelItem	Represents an individual tab.
	Optional sub-items
Panel:TabPanelContent	(Optional) Represents a panel containing the actual content of a selecting tab.
Closer:TabPanelCloser	(Optional) Represents the closer button of one tab.
CheckBox:TabPanelCheckBox	(Optional) Represents the checkbox of one tab.
Icon:TabPanelIcon	(Optional) Represents the icon of one tab.

Table 49.6: Mapping of tab panels

In case of accordion components you can map the classes to `Accordion`, `Item:AccordionItem` etc.

Sample:

The following HTML code represents a tab panel. As with many implementations of web pages there are a lot of additional nodes, irrelevant for the semantics of the tabs. Some of the nodes have attributes which could be used for the mapping. Therefore, we will show you below two different ways to configure the Install CustomWebResolver[796] node, leading to exactly the same result.

```
<div role="tab-container">
  <div class="tabs">
    <div>
      <div>
        <ul>
          <div class="tab-bar">
            <div>
              <li>
                <div type="tab">Tab1</div>
              </li>
              <li>
                <div type="tab">Tab2</div>
              </li>
              <li>
                <div type="tab">Tab3</div>
              </li>
            </div>
          </div>
        </ul>
        <div class="content">
          <div role="tab-content" id="Tab1">
            <label>The first tab panel</label>
          </div>
        </div>
        <div class="content">
          <div role="tab-content" id="Tab2">
            <label>The second tab panel</label>
          </div>
        </div>
        <div class="content">
          <div role="tab-content" id="Tab3">
            <label>The third tab panel</label>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
```

Example 49.28: HTML tab panel

```
genericClasses:
- TabPanel: tabs
- Panel:TabPanelContent: content
- Item:TabPanelItem: type=tab
ignoreTags:
- <DIV>
- <SPAN>
- <UL>
- <LI>
```

Example 49.29: HTML tab panel variant 1

The second configuration is using a different approach. QF-Test will record the same component as above.

```
genericClasses:
- TabPanel: role=tab-container
- Panel:TabPanelContent: role=tab-content
- Item:TabPanelItem:
tag: li
ancestor: TabPanel
ignoreTags:
- <DIV>
- <SPAN>
- <UL>
```

Example 49.30: HTML tab panel variant 2

49.1.9 Example for "CarConfigurator Web" demo

As this approach is quite difficult to understand just by reading, we describe a sample implementation using the "CarConfigurator Web" demo in this section. You can find the "CarConfigurator Web" test suite at `qftest-7.1.2/demo/carconfigWeb/carconfigWeb_en.qft`.
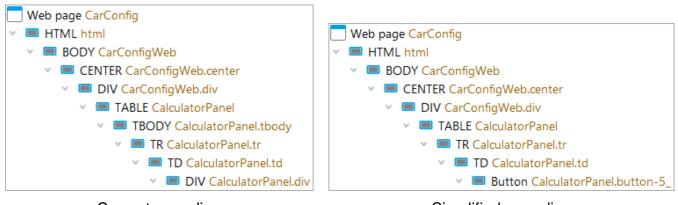
Figure 49.8: CarConfigurator Web

As stated in the previous section we need to figure out which attribute provides the required information. This information will then be used to point to a generic class of QF-Test.

Simple class mapping

To begin with the example, we resolve the recognition of the '-5%' button in the right bottom corner. The figure below shows our goal. On the left we find the current recording without simplification steps, on the right we see the desired recording.

| Current recording | Simplified recording |

Figure 49.9: Simplification due to simple class mapping

First you should record a simple text check or a mouse click to that button. Then jump to the recorded components via ⟦Locate component⟧. There you can see that you got a component of the class `DIV` and an empty name. The other attributes don't provide anything useful. Please note that QF-Test didn't record the actual text of '-5%' in any attribute. This means QF-Test has no good information for recognizing that component. There is just the geometry and the structure information. Now let us make this component more readable and the component recognition more robust.

Figure 49.10: Recording of '-5%' button in "CarConfigurator Web" demo

When analyzing the recorded component more in detail we discover that there is an extra feature `class` with the value `button`. Now we can assume that a button in our project will have that particular attribute. Especially after verifying the assumption for further buttons.

So, please insert a Install CustomWebResolver[(796)] node below the Extras node. As we found out previously the `class` attribute contains the class information for QF-Test. Knowing this, we can add `Button:  button` to the category `genericClasses`. The expression `Button:  button` signifies that any component with the CSS class `button` will be assigned the generic class `Button`. This will make QF-Test record the default features for buttons when we re-record the components. Run the Install CustomWebResolver[(796)] node and re-record the component. You will get the following recording:

Figure 49.11: Recording with genericClasses in "CarConfigurator Web"

As you can see you got a click on component `button-5_`. When you jump to the recorded component you can see the class `Button` has been assigned the '-5%' for the feature as well and we even got the extra feature `qfs:label` with that text. This component will now be treated as button by QF-Test. Of course, you should advise the development team to assign a dedicated ID to that button as well.

This simple assignment of one value can be sufficient for lots of cases, especially for buttons, menu items or checkboxes. If your web page doesn't contain that information in the attribute `class`, but in the attribute `role`, then you need to add `Button:` `role=button` to the category `genericClasses`. In some cases the information about the specific class will not be part of the leaf component, but in a parent component. The next section shows how to deal with this challenge.

Advanced class mapping

After the simple case in the previous section we will take a look at a more complex scenario now. Let's analyze how the text fields displaying the selected price information are treated, e.g. the final price text field. Like in the previous section we need to record some mouse clicks or text checks on those text fields. Then we need to navigate to the recorded components and analyze them. The figure below shows the current situation and our goal.

Current recording   Simplified recording

Figure 49.12: Simplification due to advanced class mapping

We got some `SPAN` nodes recorded. Here we have no `class` attribute, but an `id` attribute in the HTML. So, we can conclude that the `id` is very specific to the particular field. When you select its parent component, which is a `TD` node, you will find a `class` attribute with the value `textfield`, which corresponds to the actual component class. When you select that component, QF-Test also highlights the entire text field on the web page. So we can assume that a component with the value `textfield` for the `class` attribute represents a text field semantically.

Figure 49.13: Recording of `SPAN` text fields

So, now let's extend the configuration of our Install CustomWebResolver[(796)] node. We need to map the `textfield` value as generic class `TextField`. Therefore, we extend the category `genericClasses` by `TextField: textfield`.

When you delete the previously recorded component, rerun the Install CustomWebResolver[(796)] node, reload the web page and re-record the component, you will get the following recording:

Figure 49.14: Recording text fields in "CarConfigurator Web"

The text fields will be recorded as expected, and we even get rid of one level in the component hierarchy. In addition, the text fields have QF-Test specific attributes like the extra feature `qfs:label` assigned.
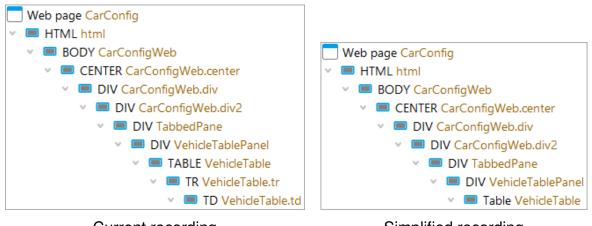
The next section show a translation for components which contain data and how to access that data afterward. Such components represent data tables, trees or lists and are handled as complex components by QF-Test.

Mapping of complex components like data tables

The previously described approach will work for most standard components as buttons or checkboxes. But besides those components there are also complex components in our GUI. Those components represent data, and we would like to address their content by the sub-item syntax provided by QF-Test. Those components could be tables, trees

or lists. For those components we need to map the dedicated class as well as the sub-item class. You will find the details in the following sections: CustomWebResolver – Combo boxes[971], CustomWebResolver – Lists[969], CustomWebResolver – Tables[961], CustomWebResolver – TabPanel and Accordion[973], CustomWebResolver – Tree[964].

Current recording   Simplified recording

Figure 49.15: Simplification for complex components

Our example is the table showing the cars of the "CarConfigurator Web". Again, we need to record some clicks on the shown cars and analyze the recording. The standard recording looks like this:

Figure 49.16: Recording of a table in "CarConfigurator Web"

The click was recorded on a `TD` component, which is the child of a `TR` component, which is part of a `TABLE` component. The recorded `TD` component contains an extra feature `class` with the value `dataCell`. The `TR` component has the value `dataRow`, and the `TABLE` has the value `dataTable` for that attribute.

When we select the nodes and observe the component highlighting in the SUT we notice the following:

A `TD` node represents a cell, a `TR` node represents a row of a table and a `TABLE` node represents an entire table. Exactly those nodes need to be investigated now in order to create a good mapping to generic classes. QF-Test requires those three classes plus the header row and a header cell to resolve a table, see CustomWebResolver – Tables[(961)] for details.

Let's start with the `TABLE` node. This node has a `class` attribute with the value `dataTable`. This seems to be a clear sign that any `dataTable` represents a ta-

ble. So we select the Install CustomWebResolver[(796)] node again and extend the category `genericClasses` by `Table:  dataTable`.

The next step is the row of the table. After selecting the `TR` node we can see that there is another `class` attribute with the value `dataRow`. This seems to be a clean-cut case. Now we need to add that value to the Install CustomWebResolver[(796)] node again and extend the category `genericClasses` with `TableRow:  dataRow`.

Next we need to analyze the `TD` node. Again, we find the `class` attribute, this time with the value `dataCell`. So, let's add this to our node as before: Add `TableCell: dataCell` to `genericClasses`.

Also, we would like QF-Test to recognize the column headers, so it can use them as text index for the column when we record a table cell. This time the `class` attribute is `headerRow` for the header row and `headerCell` for each column header. So we complete the `genericClasses` category. It now reads:

```
genericClasses:
- TableHeader: headerRow
- TableHeaderCell: headerCell
- TableCell: dataCell
- TableRow: dataRow
- Table: dataTable
- TextField: textfield
- Button: button
```

Example 49.31: Category `genericClasses`

We will now delete the previously recorded components, run the Install CustomWebResolver[(796)] node, re-load the web page and re-record a click again.

As result we get a click with the typical QF-Test item syntax on a component like `VehicleTable@Model&0` (or any other row, depending what you clicked on). In the recorded components area you will just see the `Table` object and no child component anymore as they are now treated by QF-Test as items of a table.

Figure 49.17: Recording of resolved table item in "CarConfigurator Web"

After resolving this complex component we can proceed to the next section for the next steps.

Next steps

As shown in the previous sections our first task for testing web-projects is to figure out how QF-Test recognizes the components and to create a corresponding dictionary. This task looks rather difficult at the first glance, but its result will drastically reduce the maintenance work due to component changes or hierarchy changes in later stages of your project. This is because QF-Test uses the relevant properties of your HTML components only, and not any information available.

chapter 59[(1186)] shows a full list of all generic classes for standard components and complex components like lists or trees. We recommend that you map only those components

which are really required and not every existing component. It's rather simple to extend the mechanism later.

Repeating the steps of the previous example we would now continue to map other components like menu items or tabs.   As the mapping would be too much for this manual we provided a full sample in the demo test suite `qftest-7.1.2/demo/carconfigWeb/carconfigWeb_en.qft` in the procedure `startStop.start` in the last sequence, `Install CustomWebResolver`.

In order to use the created dictionary at every start of your web application you should move the Install CustomWebResolver[796] node into your Setup node directly after launching the browser. In case you created the Setup node via the Quickstart Wizard, you will find the node in the sequence `Install CustomWebResolver` that can be configured accordingly.

Final steps

Besides the pure translation of web page specifics into QF-Test classes, it's also possible to ignore certain components during recording. This is done via the categories `ignoreTags` and `ignoreByAttributes`. However, you should do this only after you mapped most of the business components.

Finally, we would like to show the differences in the recording of the component tree as it was originally and after implementing the CustomWebResolver as in the demo test case of the demo test suite. The figure below shows the recording without any resolvers on the left and the simplified tree on the left.

```
Web page CarConfig
  HTML html
    BODY CarConfigWeb
      CENTER CarConfigWeb.center
        DIV CarConfigWeb.div
          DIV CarConfigWeb.div2
            DIV TabbedPane
              DIV VehicleTablePanel
                Table VehicleTable
                TABLE VehicleTable2
                  TBODY VehicleTable2.tbody
                    TR VehicleTable2.tr
                      TD VehicleTable2.td
              UL Tabs
                LI Tabs.liFahrzeuge
                  Link TabbedPaneVehicle
            TABLE CalculatorPanel
              TBODY CalculatorPanel.tbody
                TR CalculatorPanel.tr
                  TD CalculatorPanel.td
                    DIV CalculatorPanel.div
          DIV CarConfigWeb.div3
            UL MenuBar
              LI mOptions
                Link mOptions_menu
                UL mOptions.ulMenuBar_Element_element
                  LI miVehicles
                    Link miVehicles_menu
      DIV VehiclesDialog
        DIV VehiclesDialog.div
          DIV VehiclesDialog.div2
            Button VehiclesDialog.buttonAbbrechen
```

```
Web page CarConfig
  HTML html
    BODY CarConfigWeb
      TabPanel TabbedPane
        Panel VehicleTablePanel
          Table VehicleTable
      Panel CalculatorPanel
        Button CalculatorPanel.button-5_
      MenuBar MenuBar
        MenuItem mOptions
          MenuItem miVehicles
      Dialog VehiclesDialog
        Button VehiclesDialog.buttonAbbrechen
```

| Current recording | Simplified recording |

Figure 49.18: Simplification of the "CarConfigurator Web" demo

49.2 Special support for various web frameworks

Modern web applications are generally very interactive and their look and feel is comparable to desktop applications. Behind these applications is a whole zoo of web frameworks that drive them, each with a different focus and unique set of widgets. Such frameworks pose a problem for QF-Test and in fact any automated testing tool for several reasons:

- The actual component hierarchy is created automatically from abstract widgets like buttons or lists. Often each widget is implemented as a number of DIV nodes. This leads to very deeply nested hierarchies with very little structure.

- IDs are either not assigned at all or automatically created and thus worse than

useless for regression testing.

- The asynchronous communication with the web server and dynamic creation of DOM nodes may cause timing-related problems.

There is no panacea to address these problems in a generic way. In most cases QF-Test can interact with web frameworks out-of-the-box, but component recognition and performance are not ideal. Optimal testability can only be achieved with special case handling that exactly fits a given framework and takes advantage of its peculiarities.

The video

> ▷ 'Dealing with the explosion of complexity in web test automation' https://www.qfs.de/en/yt/web-test-automation-40.html

gives you a good idea of how QF-Test handles a deeply nested DOM structure.

QF-Test ships with optimized CustomWebResolver configuration for a number of frameworks:

Framework name	Homepage	Short name
Angular Material	material.angular.io	`angular`
Ext JS	sencha.com/products/extjs	`extjs`
Fluent UI React	developer.microsoft.com/en-us/fluentui#/	`fluentui`
Flutter Web	flutter.dev/multi-platform/web	`flutter`
Google Web Toolkit (GWT)	gwtproject.org	`gwt`
ICEfaces	icesoft.org	`icefaces`
jQuery UI	jqueryui.com	`jqueryui`
jQuery EasyUI	jeasyui.com	`jeasyui`
Kendo UI for jQuery	www.telerik.com/kendo-jquery-ui	`kendoui`
Prime Faces	primefaces.org	`primefaces`
Qooxdoo	qooxdoo.org	`qooxdoo`
Rich Ajax Platform (RAP)	eclipse.org/rap	`rap`
RichFaces	jboss.org/richfaces	`richfaces`
Smart GWT	smartclient.com	`smartgwt`
Vaadin	vaadin.com	`vaadin`
W3C ARIA	w3.org/WAI/ARIA/apg	`aria`
ZK	zkoss.org	`zk`

Table 49.7: Supported web frameworks

The given short name can be used in the Install CustomWebResolver node category `base`, see section 49.1.2[(953)]. QF-Test is even able to automatically detect whether one of those frameworks is used in your web application and to install the respective resolver. The short name `autodetect` activates this mechanism.

49.2.1 Web framework resolver concepts

A web framework resolver is a set of resolvers and other methods implemented specifically for a given web framework. Most notably QF-Test tries to assign individual classes matching the high-level widgets to DOM nodes and remove intermediate nodes that are just an implementation detail. Name, Feature and Extra feature attributes are determined in a way suitable for the framework and events are simulated on the correct DOM node in a way that most closely matches user interaction. These measures drastically reduce the component hierarchy and increase the reliability and performance of component recognition and replay. Timing and synchronization are also addressed.

As a necessary consequence the components and events recorded for a given web application vary drastically with and without an active web framework resolver and are not compatible with each other. Thus, the decision whether to use a web framework resolver should be made as early as possible, otherwise tests will either need to be reimplemented after activating the resolver or tests with and without resolver must be cleanly separated. If a resolver is available for your application you should practically always use it. The only exception is if the existing test base is already too large, mostly complete and stable.

Implementing web framework resolvers is an ongoing process. As development of a web framework continues, the associated CustomWebResolver may also have to be updated. Therefore, the built-in CustomWebResolvers are marked with a version number that corresponds to the version number of the framework for which the resolver was originally designed. As long as there are no incompatible changes, this CustomWebResolver can also be used for newer versions of the framework.

Older CustomWebResolvers in QF-Test still use a versioning scheme that is independent of the framework version. These will be updated to the new versioning scheme on their next update.

Web framework resolvers are activated via the <u>Install CustomWebResolver</u>⁽⁷⁹⁶⁾ node where you can provide the version to use. You can choose to specify only the major version, in which case QF-Test uses the latest medium.minor version available for this major version. This is normally the best option and used in the SUT startup sequences created with QF-Test's Quickstart Wizard (see <u>chapter 3</u>⁽²⁸⁾). Alternatively you can specify major.medium version or even major.medium.minor to use an exact version and thus run your tests with the resolver version with which they were created.

49.2.2 Setting unique IDs

Any web framework has its custom way of setting unique IDs. Please find details about the supported ones in the following chapter:

Angular Material

The simplest solution is to set the 'ID' attribute `<div id="myId"/>` for any required component.

Ext JS

You can set IDs like
`var container = Ext.create('Ext.container.Container', {`
`id: 'MyContainerId',`
`... });`.
As alternative you can also call `container.getEl().set({ 'qfs-id': 'myId' });`. In this case you will need to implement a NameResolver for reading 'qfs-id' as name for QF-Test.

GWT

The simplest way is calling the method `widget.getElement().setId("myId");` for the required widgets.

As an alternative you can also call `widget.ensureDebugId("myId")`. But if you want to use that method you need to modify your `xxx.gwt.xml` file to enable debug IDs. Add `<inherits name="com.google.gwt.user.Debug"/>` to the file.

It's also possible to set a custom identifier which can then be used via a NameResolver.For example call `setAttribute("qfs-id", "myId")` to set an 'qfs-id' attribute.

ICEfaces

The simplest solution is to set the 'ID' attribute `<p:inputText id="myId"/>` for any required component in the xhtml definition.

jQuery UI

The simplest solution is to set the 'ID' attribute `<p:inputText id="myId"/>` for any required component. Additionally, you can give an existing element an id with: `$(element).attr("id","myId");`

jQuery EasyUI

The simplest solution is to set the 'ID' attribute `<p:inputText id="myId"/>` for any required component.

Kendo UI for jQuery

You need to set the 'ID' attribute in your source code or graphical editor.

PrimeFaces

The simplest solution is to set the 'ID' attribute `<p:inputText id="myId"/>` for any required component in the xhtml definition.

Qooxdoo

There is no default mechanism. You can either apply a custom attribute to the generated DOM nodes or add a custom attribute to the `setData` method of the widget. You can evaluate those attributes in a resolver.

RAP

Starting with RAP version 2.2 a name set via `widget.setData("name", "myId")` is retrieved automatically by QF-Test, just as for SWT. This field can only be used if it is registered with `WidgetUtil.registerDataKeys("name");` before.

For RAP versions older than 3.0.0 the following technique is also available. It is discouraged because it deviates from the SWT standard and requires additional settings for the webserver:

Call the method `widget.setData(WidgetUtil.CUSTOM_WIDGET_ID, "myId");` for the required widgets.

After applying IDs to components you need to modify your webserver environment and specify the following parameter `-Dorg.eclipse.rap.rwt.enableUITests=true` before launching the webserver.

Note: The VM argument was renamed in RAP 2.0. For RAP versions older than 2.0 its name is `-Dorg.eclipse.rap.rwt.enableUITests=true`.

RichFaces

You need to set the 'ID' attribute in your source code or graphical editor.

Smart GWT

The simplest solution is to call `widget.setID("id")` for any required component.

Vaadin

The simplest solution is to call `widget.setId("id")` (`widget.setDebugId("id")` for Vaadin version < 7) for any required component.

You can also set a custom stylesheet class, which you could read with a NameResolver. Therefore call `widget.setStyleName("qfs-id=myId")`.

ZK

QF-Test uses the widget ID, which is also used in the `zul` files, so you should get meaningful IDs for most of the objects.

The ZK framework also offers a custom IDGenerator to set such IDs. But implementing this could be quite exhaustive. In this case it might be a better choice to rely on the default mechanism of QF-Test.

49.3   Browser connection mode

QF-Test has three different modes to gain access to a browser. This section describes these modes in details.

4.1+   Given that the QF-Driver approach using embedding is not maintained anymore by some browser vendors or is not supported at all, a new mechanism was implemented for QF-Test 4.1 to support future browsers and browser versions. This mechanism uses Selenium WebDriver as a bridge between the browser and QF-Test.

5.3+   For the browsers based on Chromium there is a more effective alternative to Selenium WebDriver - CDP-Driver.

The following table lists browsers and the respective connection mode. QF-Test determines the correct mode automatically by default. However, you can override the choice via the attribute Browser connection mode[647] in the Start web engine[645] node

Browser	Connection mode	Comment
Chrome	QF-Driver	A current stable Chromium version is part of the QF-Test distribution (Windows only)
Chrome	CDP-Driver	Experimental support for newer versions also (see Supported technologies[3])
Chrome	WebDriver	The ChromeDriver shipped supports various versions, further versions via the automatic ChromeDriver Download. For version number see Supported technologies[3]
Chrome (headless)	CDP-Driver	
Chrome (headless)	WebDriver	
Firefox	WebDriver	Supported by the GeckoDriver shipped, currently 102esr and higher
Firefox (headless)	WebDriver	
Microsoft Edge	CDP-Driver	see Chrome (CDP-Driver)
Microsoft Edge	WebDriver	
Microsoft Edge (headless)	CDP-Driver	
Microsoft Edge (headless)	WebDriver	
Opera	CDP-Driver	see Chrome (CDP-Driver)
Opera	WebDriver	`deprecated`
Safari	WebDriver	

Table 49.8: Connection mode for browsers

49.3.1 QF-Driver connection mode

This approach integrates the locally installed browser into a wrapper-window. This approach is also called embedding. QF-Test natively embeds the browser into its own window, thus gaining access to its automation-interfaces. By using these interfaces QF-Test can listen for events from the browser and is also able to inject events into the browser.

49.3.2 CDP-Driver connection mode

5.3+

Chrome DevTools Protocol is an API that is available for testing and debugging of browsers based on Chromium (Google Chrome, Microsoft Edge and Opera) and is used

for example in embedded development tools. Since version 5.3 QF-Test uses this interface to connect and communicate with a browser. Unfortunately, Mozilla does not provide a full implementation of such an interface for Firefox. Such an implementation does not exist also for Safari.

49.3.3   WebDriver in general

WebDriver is evolving into a W3C-standard for interacting with web browsers. (http://www.w3.org/TR/webdriver). WebDriver is a remote control interface that enables introspection and control of browsers, based on a platform and language-neutral wire protocol.

The various browser vendors have agreed on this quasi-standard, so that the WebDriver integration is partly implemented directly by the vendors themselves. Partially the integration is based on plugins, some vendors already include the integration in the default setup of their browsers.

QF-Test uses the WebDriver interfaces to interact with the browser. Since the WebDriver approach only partially fits the concepts of QF-Test, its web engine was extended so that most of QF-Test's functionality is also available via WebDriver, including the added benefits like synchronization, abstraction of components etc.

Note   Selenium WebDriver requires Java version 8 or higher.

49.3.4   Known limitations of the WebDriver mode

The WebDriver connection mode is under active development. Due to this, some features known from QF-Driver connection mode are not yet available, mostly due to restrictions of the WebDriver specification.

- No support of file downloads and HTTP authentication.

- It is not possible to record or replay HTTP requests directly.

- wd.getComponent(WebElement) does not work currently on elements in inner frames.

- Events triggering a page load are sometimes not recorded.

- Event-Synchronization is in some cases delayed.

49.4 Web – Pseudo Attributes

Web

The idea of pseudo attributes is to simplify resolvers using JavaScript to retrieve values from a browser. You can register a pseudo attribute in QF-Test which receives its value as result of some JavaScript code execution.

In a SUT script you can fetch attribute values of HTML elements via the method `getAttribute()` (This is also the way the CustomWebResolver (see section 49.1.2[(949)]) evaluates its `genericClasses` category). Pseudo attributes values are fetched with the same mechanism, behaving just like normal attributes in the "eyes" of QF-Test. Only, they are not defined via the HTML source code or explicitly set via `node.setAttribute()`, but execute a piece of JavaScript code.

When you define a pseudo attribute you can mark it as "cacheable". In that case the pseudo attribute will be evaluated the first time referenced and then its value will be saved until the next complete scan of the page. This improves the overall testing performance. Pseudo attributes should not be cached for values subject to change (e.g. the status of a check box), because then a change of the value would not be "visible" to QF-Test. Uncached pseudo attributes will not save their value, but it will be fetched (and kept shortly for processing) each time an event will be recorded, like for example the single events "moved", "pressed", "released" and "clicked" of a mouse click, or for the identification of a component during replay.

The following example defines a pseudo attribute for all HTML elements with the tag "ICON" or "IMAGE" evaluating the value of `iconname` via JavaScript (Technically speaking, it calls the sample method `inspect` defined by the framework for the HTML node).

```
import de.qfs.apps.qftest.client.web.dom.DomNodeAttributes
import de.qfs.apps.qftest.client.web.dom.FunctionalPseudoAttribute
def attr = new FunctionalPseudoAttribute("js_icon",
                    "try {return _qf_node.inspect('iconname')} catch(e){}", true)
DomNodeAttributes.registerPseudoAttributes("ICON", attr)
DomNodeAttributes.registerPseudoAttributes("IMAGE", attr)
```

Example 49.32: Groovy SUT script registering a pseudo attribute
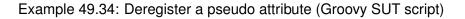
The pseudo attribute can then be used in a feature resolver. In comparison to a direct call of `node.callJS()` in the script this method takes advantage of the internal caching mechanisms of QF-Test:

```
def getFeature(node, feature):
    iconname = node.getAttribute("js_icon")
    return iconname
resolvers.addResolver("iconFeature", getFeature, "ICON", "IMAGE")
```

Example 49.33: Using a pseudo attribute in a resolver (Jython SUT script)

The following script deregisters the pseudo attribute.

```
import de.qfs.apps.qftest.client.web.dom.DomNodeAttributes
DomNodeAttributes.unregisterPseudoAttributes("ICON", "js_icon")
DomNodeAttributes.unregisterPseudoAttributes("IMAGE", "js_icon")
```

Example 49.34: Deregister a pseudo attribute (Groovy SUT script)

A pseudo attribute has to be defined via the following method from the module `de.qfs.apps.qftest.client.web.dom.pseudo attributes`:

PseudoAttribute FunctionalPseudoAttribute(String name, String javaScriptFunction, Boolean cached)

Defines a pseudo attribute.

Parameters

name	The name for the pseudo attribute.
javaScriptFunction	The JavaScript code to be executed within a function when referencing the pseudo attribute. Use `_qf_node` as the reference for the HTML element. The execution is equal to a call of `DomNode.callJS`.
cached	`true`, when you want to cache the value after the first reference to the pseudo attribute, otherwise `false`.
Returns	A pseudo attribute you can then register.

PseudoAttribute PseudoAttribute(String name, String javaScriptCode, Boolean cached)

Defines a pseudo attribute.

Parameters

name	The name for the pseudo attribute.
javaScriptCode	The JavaScript code to be evaluated when referencing the pseudo attribute. Use `_qf_node` as the reference for the HTML element. The execution is equal to a call of `DomNode.evallJS`.
cached	`true`, when you want to cache the value after the first reference to the pseudo attribute, otherwise `false`.
Returns	A pseudo attribute you can then register.

Having defined the pseudo attribute you need to register it via the following method from module `de.qfs.apps.qftest.client.web.dom.DomNodeAttributes`:

---

void registerPseudoAttributes(String tag, PseudoAttribute pseudoAttribute)

Registers a pseudo attribute for HTML elements with the given tag.

Parameters

tag	The tag of the HTML elements to be registered for. When you want to register the pseudo attribute for HTML elements with different tags you need to do it for each one in turn. When you want to register it for all HTML elements, use the tag ″`<QF_ALL>`″.
pseudoAttribute	The pseudo attribute previously defined.

---

void unregisterPseudoAttributes(String tag, String name)

Deregisters the pseudo attribute for HTML elements with the given tag. You do not need to deregister a pseudo attribute. When stopping QF-Test it will be done automatically.

Parameters

tag	The tag of the HTML elements for which to deregister the pseudo attribute.
name	The name of the pseudo attribute.

49.5   Accessing hidden fields on a web page

Web

Hidden fields are not captured by default and therefore not stored under the Windows and components[835] node.

In case you frequently need to access hidden fields you can deactivate the Take visibility of DOM nodes into account[493] option.

Another way to get hidden fields recorded is the following:

- Activate the record components[39] mode.

- Navigate the mouse cursor to the parent item containing the hidden field (most likely a FORM element).

- Press right mouse button and select ‖Component and children‖ from the popup menu.

- Deactivate the record components[39] mode.

- A search[19] within Windows and components[835] for e.g. 'HIDDEN' should get you to your destination component quickly.

To access a hidden field's attributes (e.g. the 'value' attribute) you can create a simple SUT script[(631)] as shown below. Details on scripting in general, the used methods and parameters can be found in Scripting[(161)], Run context API[(917)] and Pseudo DOM API[(1118)] respectively.

```
node = rc.getComponent('id of the hidden component', hidden=1)
node.getAttribute('value')
```

Example 49.35: Accessing the value attribute of a hidden field

49.6 WebDriver with Safari

To run tests with Safari, macOS and Safari 12 or newer are required. In addition, a setup has to be performed to oactivate the browser automation: Open the Safari Preferences and select "Show Develop menu in menu bar". In this menu, activate the "Allow Remote Automation" option. After that, open a Terminal window and execute once the command `/usr/bin/safaridriver -p 0` to authorize the driver.

Note   Due to special security restrictions imposed by the Apple SafariDriver, the interaction between QF-Test and Safari is limited in the following ways:

- Tests can only be replayed but not recorded

- No hard events are possible

- Only one browser instance is allowed

Chapter 50

Controlling native Windows applications via the UIAuto module - without the QF-Test `win` engine

The `win` engine is described in a separate chapter <u>Testing native Windows applications</u>[207].

Generally you should use the `win` engine of QF-Test for testing and controlling native Windows applications. This requires a respective license. In case you do not have a license for the `win` engine of QF-Test and you require just a little test or need to do some simple controlling of a Windows application, e.g. during testing of process flow across Java, Web and Windows applications, you can use the library described in the current chapter.

The module allows control of native Windows GUI elements via Microsoft's UI Automation interface. It can trigger actions and check certain values. The interface is described on https://en.wikipedia.org/wiki/Microsoft_UI_Automation.

Procedures in the <u>standard library</u>[158] `qfs.qft` wrap the methods of the module most frequently required.

It is not possible to record actions or checks directly (capturing). The parameters identifying a GUI object need to be determined and then be passed to the respective procedure.

Actions are replayed via 'hard' system events mainly. This results in a different replay behavior than you are used to with QF-Test for Java or web applications.

Despite these restrictions the module can be very helpful for simple testing and controlling tasks on native Windows applications.

50.1 Proceeding

The Microsoft UI Automation is an Accessibility and Test Framework allowing programs to control the GUI elements of native Windows applications. With QF-Test you can use the framework in script nodes via the Jython module `uiauto` (alternatively `de.qfs.UIAuto` for Groovy, `uiauto` for Javascript).

QF-Test provides a package in the standard library with procedures for the most commonly needed interactions with GUI elements for direct and easy use of the API for test development. This chapter describes the standard library package.

You will find the procedures relevant for control of native Windows elements in the package `qfs.autowin`. You can see several procedures marked as deprecated. They have been replaced by normal nodes of the `win` engine of QF-Test, and the procedures in the `qfs.autowin` package will not be further maintained. However, you can continue using them as they are. If you encounter problems, for example with scaled windows applications, it is advised to switch to the `win` engine.
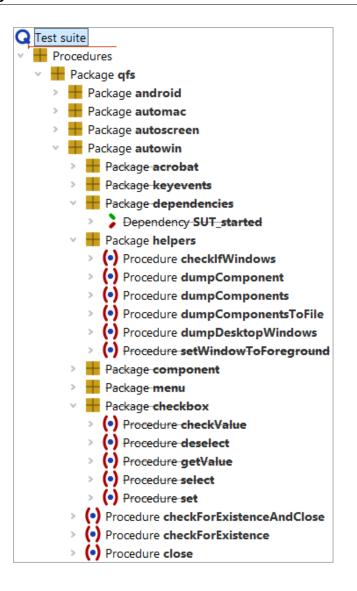
Figure 50.1: UI Automation procedures in the standard library

When developing tests for native windows applications you generally need to perform the following steps:

- Start the application

- Determine the identifying parameters for the GUI elements

- Set up the tests calling the respective procedures specifying the GUI elements via the identifiers.

50.1.1 Starting the application

The application to be tested may but does not necessarily have to be started via QF-Test.

When starting the application via QF-Test the client process started is listed in the QF-Test menu ⟦Clients⟧ and can also be stopped via QF-Test.

Please use the procedure `qfs.autowin.checkForExistence` to check whether the application was started.

You will find an example for the start of an application in section 50.2.1[(1007)].

50.1.2 Listing the GUI elements of a window

Before you can set up a test you need to get an overview of the GUI elements of the application. You may either use the procedure `qfs.autowin.helpers.dumpComponents` to print the GUI elements to the QF-Test terminal or `qfs.autowin.helpers.dumpComponentsToFile` to write them to a file.

The procedure `qfs.autowin.helpers.DumpDesktopWindows` allows you to list the titles of all open windows of the desktop.

`qfs.autowin.helpers.dumpComponents` prints the name (Name), the class (ClassName), the compontent type (ControlType) and the Id (AutomationId) of the GUI elements, provided they were implemented for the respective GUI element.

All the GUI elements visible on the Windows desktop are organized in a tree structure with the desktop as the root element. When calling the dump procedure you need to specify the window for which to list the GUI elements. Nesting of the components is represented via indentation.

Note   The procedure dumpComponents() prints its output to the QF-Test terminal displayed in the botton part of the QF-Test window. The output is not displayed in the terminal or consoles which can be opened separately (client terminal and scripting consoles).

Please find an example in section 50.2.2[(1008)].

50.1.3 Information on single GUI elements

The procedure `qfs.autowin.helpers.dumpComponent` allows you to print further information for single GUI elements, including a list of the methods available for the element as well as attribute values.

50.1.4 Identifiers for GUI elements

All procedures of the standard library package preforming actions on native Windows applications need to determine the respective GUI element as the first step and then perform the action in a second step. You find the procedures in the package `qfs.autowin.component`. Because all procedures use `qfs.autowin.component.getControl` to identify the GUI element, the parameters of this helper procedure are valid for all the procedures performing an action on a GUI element.

The following parameters (and combinations) are valid (in the order of evaluation):

- AutomationId

- ControlType and name

- ControlType and index

- ClassName and name

- ClassName and index

- Name

AutomationId
    The AutomationId is a unique identifier for the GUI element within a window. It has to be set explicitly during application development, which unfortunately does not always happen.

Name
    The name usually corresponds to the text displayed. Names do not have to be unique and you may have to specify the ControlType or the ClassName of the GUI element additionally. Names will be evaluated as regular expressions. For more information about regular expressions please see Regular expressions[909].

ControlType
    The ControlType is a value from a predefined list of component types, e.g. Button, CheckBox, ComboBox, DataGrid, Edit, List, Tab, Text. The procedure `qfs.autowin.helpers.dumpComponent` shows the name and the numeric value of the respective ControlType. In order to identify a GUI element via its ControlType you usually need to specify its name or its index (relative to the GUI elements in the window of that ControlType), too, except there is just the one GUI element of that ControlType in the window.

ClassName

ClassNames are framework specific. Additionally to the ClassName you may specify the name or the index (relative to the GUI elements in the window of that ClassName) of the GUI element.

50.1.5 Actions on GUI elements

You will find procedures in the package `qfs.autowin.component` of the standard library for the most common actions. You are free to enhance the package. We recommend to use a separate test suite for the enhancement and not to change the qfs.qft since we continuously update the standard library and ship a new version with every QF-Test release.

Mouse click

Procedure: `qfs.autowin.component.click`

The procedure tries to replay a click event to the GUI element. In case this is not implemented the procedure replays a hard mouse click to the position of the GUI element.

Wait for component

Procedure: `qfs.autowin.component.waitForComponent`

The procedure waits for the given component and returns control to the calling node as soon as it finds the component. The given timeout (in milliseconds) is the maximum time to wait. It throws an exception if the component is not found within the given time.

Wait for window

Procedure: `qfs.autowin.checkForExistence`

The procedure waits for the given window and returns control to the calling node as soon as it finds the window. The given timeout (in milliseconds) is the maximum time to wait. The parameter 'errorLevel' specifies whether to log a message, warning, error or an exception in case the the window is not found within the given time.

Text input

Procedure: `qfs.autowin.component.setText`

The procedure uses the method setText() of the IUIAutomationElement interface to enter a text to the given component. In case the setText() method has not been implemented for the component please use setValue().

Keyboard events
The Package `qfs.autowin.keyevents` provides procedures for replaying the keyboard events ENTER, TAB and DELETE. The procedure `qfs.autowin.keyevents.sendKey` lets you replay any key like a single letter, a digit, a function key, etc, also combined with the modifiers SHIFT, CTRL and ALT. The event is replayed to the component with the focus in the given window.

Fetch text
Procedure: `qfs.autowin.component.getText`

The text of a GUI element cannot be accessed directly. The name of a GUI elements usually corresponds to the text displayed. Some elements have a value corresponding to the text, independently of the name.The procedure getText() first tries to determine the value of the element and, in case this fails or the value is an empty string, returns the name of the element.

The procedures getName() and getValue() are provided additionally.

Fetch geometry
Procedure: `qfs.autowin.component.getGeometry`

Check text
Procedure: `qfs.autowin.component.checkText`

The procedure fetches the text of the GUI elements via the procedure getText() and compares the value returned with the given text.

The procedures checkName() and checkValue() are provided additionally.

Check geometry
Procedure: `qfs.autowin.component.checkGeometry`

The procedure fetches the geometry data via getGeometry() and compares them to the given values.

Image check
Procedure: `qfs.autowin.component.checkImage`

The procedure relies on a file with a reference image. The file needs to have a `png` format. The procedure determines the screen coordinates of the element via

qfs.autowin.component.getGeometry. The actual comparison is done via the procedure getPositionOfImage() of the qfs.autoscreen package of the standard library.

Select an item in a menu

Procedure: qfs.autowin.menu.selectItem

Especially when single-stepping through the test when debugging it is useful to have a procedure which clicks to a menu and its menu item which can be executed in one step. Thus the application will not loose the focus between steps which might cause the menu to close.

50.2   Example

Please find the sample test suites in the subdirectories demo/carconfigWpf and demo/carconfigForms of the QF-Test installation directory.

50.2.1   Starting the application

The sample test suites use the dependency SUT_started in the package qfs.autowin.dependencies of the standard library to start the demo application. The node

- Execute shell command

actually starts the demo. The

- Procedure call: qfs.autowin.checkForExistence

waits for the demo application to appear before passing on control to the test case itself.

The demo test suite makes use of the dependency functionality which enables you to manage the set up and clean up requirements of your tests very efficiently. In short, in a test case the dependency functionality runs the setup node before passing control to the test case, thus making sure that the requirements implemented in the setup node are met before running the test case. At the end of the test case, the cleanup node is not executed by default. Only in the next test case it may be executed if required by the dependency of that test case, either because it calls a different dependency or the

characteristic variables of the dependency have changed. For more information about dependencies please refer to the tutorial or the manual, chapter <u>Dependency nodes</u>⁽¹³⁸⁾.

For general information on the start of a native windows application please refer to <u>section 50.1.1</u>⁽¹⁰⁰³⁾.

50.2.2 GUI element information

The next step after starting the SUT is to get an overview of the GUI elements of the application. In this example we will use the procedure `qfs.autowin.helpers.dumpComponents`. The output will be written to the QF-Test terminal.

In the following we will have a look at the output of dumpComponents() for some of the GUI elements of the WPF demo application.

Figure 50.2: The WPF demo application

Window (1)
> Name: CarConfiguratorNet WPF
> ClassName: Window
> ControlType: Window (#50032)
> This shows a typical set of identifiers for the window itself. You can address it via its name, which seems to be unique.

Menu (2)
    Name: Help
ClassName: MenuItem
ControlType: MenuItem (#50011)
AutomationId: mHelp

This GUI element has an AutomationId. Use this to identify the GUI element unambiguously. Of course, you may also use its name 'Help' or the ControlType (or ClassName) 'MenuItem' along with the index '4'.

Table cell (3)
    Name: Rolo
ClassName: DataGridCell
ControlType: Custom (#50025)

The table cell can either be addressed by its name or by the ControlType along with the name. If you want to address the table cell via an index it would make sense to use the ClassName and not the ControlType as the same ControlType 'Custom' is used for non table cells as well. An AutomationId has not been implemented.

Input field (4)
    ClassName: TextBox
ControlType: Edit (#50004)
AutomationId: textBoxDiscountPrice

The input field can be addressed via the AutomationId or the ControlType or ClassName along with the index 0.

Text display field (5)
    Name: $ 12,300.00
ClassName: Text
ControlType: Text (#50020)
AutomationId: labelCalculatedPriceOutput

Use the AutomationId to identify the GUI element as the name varies according to the text displayed.

Please find general information in <u>section 50.1.2</u>⁽¹⁰⁰³⁾.

Chapter 51

Controlling and testing native MacOS applications

At the moment QF-Test primarily supports functional testing of Java and Web applications. We are working on a comparable module for testing native MacOS applications with equivalent processes and features. The module described in this chapter provides a temporary solution when simple control of native MacOS applications is required during testing of process flow across Java, Web and MacOS applications.

The module allows control of native MacOS GUI elements via the MacOS Accessibility interface. It can trigger actions and check certain values. Further information about this interface can be found in the corresponding documentation.

Procedures in the <u>standard library</u>⁽¹⁵⁸⁾ `qfs.qft` wrap the methods of the module most frequently required.

It is not possible to record actions or checks directly (capturing). The parameters identifying a GUI object need to be determined and then be passed to the respective procedure.

Actions are replayed via 'hard' system events mainly. This results in a different replay behavior than you are used to with QF-Test for Java or web applications.

Despite these restrictions the module can be very helpful for simple testing and controlling tasks on native MacOS applications.

51.1   Proceeding

The MacOS Accessibility Interface is allowing programs to control the GUI elements of native MacOS applications. With QF-Test you can use the framework in script nodes via the Jython module `automac` (alternatively `de.qfs.automac` for Groovy, `automac` for

Javascript).

QF-Test provides a package in the standard library with procedures for the most commonly needed interactions with GUI elements for direct and easy use of the API for test development. This chapter describes the standard library package.

You will find the procedures relevant for control of native MacOS elements in the package `qfs.automac`.

When developing tests for native MacOS applications you generally need to perform the following steps:

- Start the application

- Determine the identifying parameters for the GUI elements

- Set up the tests calling the respective procedures specifying the GUI elements via the identifiers.

51.1.1 Starting the application

The application to be tested may but does not necessarily have to be started via QF-Test. In any case you need to establish a "connection" to the application via the procedure `qfs.automac.app.connect`. It stores a handle to the accessibility interface of the application in a QF-Test Jython variable. It has the following parameters for searching, respectively starting, the application.

You can specify `bundleId` to identify the application via the unique bundle id, e.g. com.apple.Calculator. In case the application is already running QF-Test just stores the handle, otherwise it starts the application as well.

You can specify `bundleFile` to identify the application via the bundle file where the application is stored, e.g. /Applications/Calculator.app. In case the application is already running QF-Test just stores the handle, otherwise it starts the application as well. In that case a QF-Test process will also appear in the list of the QF-Test clients menu Clients By stopping that process the application will also be terminated.

You can specify `title` to identify the application via the window title. The application has to be started beforehand, e.g. via the node Execute shell command.

You can specify `processId` to identify the application via the process identification number (PID). The application has to be started beforehand, e.g. via the node Execute shell command.

The procedure `qfs.automac.helpers.DumpDesktopWindows` lists title, PID, bundle id and bundle file of all running applications in the terminal.

51.1.2 Listing the GUI elements of a window

Before you can set up a test you need to get an overview of the GUI elements of the application. You may either use the procedure `qfs.automac.helpers.dumpComponents` to print the GUI elements to the QF-Test terminal or `qfs.automac.helpers.dumpComponentsToFile` to write them to a file.

`qfs.automac.helpers.dumpComponents` prints label, title, role, subrole, type and identifier of the GUI elements, provided the attribute was implemented for the respective GUI element.

All the GUI elements visible on the desktop are organized in a tree structure with the desktop as the root element. The nesting of the components is represented via indentation. The procedure lists the components of the application specified in the procedure call or of the one already connected.

Note   The procedure dumpComponents() prints its output to the QF-Test terminal displayed in the botton part of the QF-Test window. The output is not displayed in the terminal or consoles which can be opened separately (client terminal and scripting consoles).

51.1.3 Information on single GUI elements

The procedure `qfs.automac.helpers.dumpComponent` allows you to print further information for single GUI elements, including a list of the methods available for the element as well as attribute values.

51.1.4 Identifiers for GUI elements

All procedures of the standard library package performing actions on native MacOS applications need to determine the respective GUI element as the first step and then perform the action in a second step. You find the procedures in the package `qfs.automac.component`. Because all procedures use `qfs.automac.component.getControl` to identify the GUI element, the parameters of this helper procedure are valid for all the procedures performing an action on a GUI element.

Valid parameters:

- label

- title

- identifier

- role

- roleType

- subrole

- index

If you specify more than one parameter the procedure looks for the GUI element for which all values match.

`identifier`
> Specify `identifier` for the unique identifier of the GUI element within the window. It has to be set explicitly during application development, which unfortunately does not always happen.

`label`
> The label usually corresponds to the text displayed. It does not have to be unique and you may have to specify other parameters additionally. In case you use the Accessibility Inspector for the analysis of the GUI elements the attribute is called either `Label` or `AXDescription`.

`title`
> The title usually corresponds to the text displayed. It does not have to be unique and you may have to specify other parameters additionally.

`role`
> The role is a value from a predefined list of component types, e.g. AXButton.

`roleType`
> `roleType` specifies the type of the GUI element. In case you use the Accessibility Inspector for the analysis of the GUI elements the attribute is called either `Type` or `AXRoleDescription`.

`subrole`
> `subrole` is an additional specification to `role`.

`index`
> When there are more than one GUI element matching the given parameters the index specifies the one to pick. `index` starts at 0.

51.1.5 Actions on GUI elements

You will find procedures in the package `qfs.automac.component` of the standard library for the most common actions. You are free to enhance the package. We recommend to use a separate test suite for the enhancement and not to change the qfs.qft

since we continuously update the standard library and ship a new version with every QF-Test release.

Mouse click
> Procedure: `qfs.automac.component.click`

The procedure waits for the given component and the replays a click event to the GUI element.

Wait for component
> Procedure: `qfs.automac.component.waitForComponent`

The procedure waits for the given component and returns control to the calling node as soon as it finds the component. The given timeout (in milliseconds) is the maximum time to wait. It throws an exception if the component is not found within the given time.

Text input
> Procedure: `qfs.automac.component.setValue`

The procedure waits for the given component and then enters the value. It uses the method setValue() of the Accessibility interface.

Keyboard events
> Procedure: `qfs.automac.sendKey`

The procedure lets you replay any key like a single letter, a digit, a function key, etc, also combined with the modifiers SHIFT, CTRL and ALT. The event is replayed to the component with the focus in application.

Fetch text
> Procedure: `qfs.automac.component.getValue`

The procedure waits for the given component and then fetches the value via the method getValue().

Fetch geometry
> Procedure: `qfs.automac.component.getGeometry`

The procedure fetches and returns the x and y screen coordinates of the upper left corner of the component as well as the width and height. They are returned as comma-separated text.

Check text
> Procedure: `qfs.automac.component.checkValue`

The procedure fetches the text of the GUI elements via the procedure getValue() and compares the value returned with the given text.

Check geometry

Procedure: `qfs.automac.component.checkGeometry`

The procedure fetches the geometry data via getGeometry() and compares them to the given values.

Image check

Procedure: `qfs.automac.component.checkImage`

The procedure relies on a file with a reference image. The file needs to have a `png` format. The procedure determines the screen coordinates of the element via `qfs.automac.component.getGeometry`. The actual comparison is done via the procedure getPositionOfImage() of the `qfs.autoscreen` package of the standard library.

Select an item in a menu

Procedure: `qfs.automac.menu.selectItem`

Especially when single-stepping through the test when debugging it is useful to have a procedure which clicks to a menu and its menu item which can be executed in one step. Thus the application will not loose the focus between steps which might cause the menu to close.

Chapter 52

Extension APIs

QF-Test provides some extension APIs that let you extend its functionality. The interfaces can conveniently be implemented in Jython, Groovy or or Java. In the latter case, the compiled classes should be put in a jar file and placed in the plugin directory (see section 48.2⁽⁹¹⁶⁾) so Jython or Groovy can be used to glue things together.

52.1 The `resolvers` module

This extension API lets you install hooks that can modify the way QF-Test recognizes and records components and items. This is a very powerful feature that gives you fine-grained control over the QF-Test component management.

Video:

> ▷ 'Resolvers in QF-Test'
> https://www.qfs.de/en/yt/resolvers-46.html

52.1.1 Usage

When registering resolvers it is important to specify the correct GUI engine⁽⁶³³⁾ attribute in the SUT script⁽⁶³¹⁾. If the wrong engine is specified, the resolver simply will not work. If no engine is specified the resolver applies to all engines which can cause confusion and break replay in engines for which the resolver was not intended.

We will start with a short description of how QF-Test does component recognition to give you an idea where resolvers come into play. It consists roughly of four steps:

1. Get the component object from the GUI.

2. Extract the data for each component: e.g. component class, id, coordinates, component text.

3. Analyze the relationship between components: e.g. structure information (index), find the label belonging to the component (qfs:label).

4. Recoding: Create a Component node and save the retrieved data in the details of the node.
   Replay: Compare the retrieved data with the details of the node that is the target of the replay action.

QF-Test uses resolvers for steps 2 and 3. Via the API you can overwrite resolver methods and thus manipulate component recognition.

A resolver is the only way to influence component recognition during recording. During replay you also have other options (e.g. a script or a regular expression in the details of the Component node) to get at component data.

Web For web applications QF-Test offers a specialized interface providing most of the functionality of the resolvers described below, which is a lot easier to handle. See Improving component recognition with a `CustomWebResolver`[(945)]. The Install CustomWebResolver[(796)] node has been optimized for web elements thus providing a better performance than resolvers of this section. The use of below resolvers for web components should be limited to very special cases.

A list of available `resolvers`:

- `NameResolver` section 52.1.7[(1023)]

- `GenericClassNameResolver` section 52.1.8[(1026)]

- `ClassNameResolver` section 52.1.9[(1026)]

- `FeatureResolver` section 52.1.10[(1027)]

- `ExtraFeatureResolver` section 52.1.11[(1028)]

- `ItemNameResolver` section 52.1.12[(1034)]

- `ItemValueResolver` section 52.1.13[(1035)]

- `TreeTableResolver` section 52.1.14[(1036)]

- `InterestingParentResolver` section 52.1.15[(1038)]

- `TooltipResolver` section 52.1.16[(1039)]

- `IdResolver` section 52.1.17[(1039)]

- EnabledResolver section 52.1.18[(1040)]

- VisibilityResolver section 52.1.19[(1041)]

- MainTextResolver section 52.1.20[(1042)]

- WholeTextResolver section 52.1.21[(1043)]

- BusyPaneResolver section 52.1.22[(1043)]

- GlassPaneResolver section 52.1.23[(1044)]

- EventSynchronizer section 52.1.24[(1045)]

- BusyApplicationDetector section 52.1.25[(1045)]

- ExtraFeatureMatcher section 52.1.26[(1046)]

52.1.2 Implementation

The following two steps are required to implement a resolver:

1. Implementation of the resolver interface.

2. Registration of the interface for the desired component class(es).

In most cases the interface consists of only one method. A typical example would be (Jython-Skript):

```
def getName(menuItem, name):
    if not name:
        return menuItem.getLabel()
resolvers.addResolver("menuItems", getName, "MenuItem")
```

Example 52.1: Simple `NameResolver` (Jython) for `MenuItems`

The first three lines are the method of the resolver interface. The name of the method defines the resolver type. Each resolver type manipulates a certain value in the Component node data. In our case the method is `getName` thus defining a name resolver. The fourth line calls the function `addResolver` of the `resolvers` module and registers the resolver.

Most resolver methods only have two parameters: the first is the component for which component recognition is done at that moment. The second is the value or object to be handled by the method. With a name resolver it is the name determined by the QF-Test

standard name resolver. With a feature resolver the feature determined by QF-Test and so on. You will find a detailed description of the resolvers interfaces in chapters section 52.1.7[(1023)] to section 52.1.25[(1045)].

Each resolver needs to be given a name at registration time. The name has to be unique. It will be used when the resolver needs to be updated or uninstalled explicitly via `resolvers.removeResolver("resolver name")` (see section 52.1.4[(1022)]).   The names of all registered resolvers can be listed via the function `resolvers.listNames()` (see section 52.1.5[(1023)]).

After changing the contents of a resolver script it needs to be executed again in order to register the updated resolver. As long as the name of the resolver remains unchanged there is no need to first deregister the old version first.

All resolver types can be registered either for single components, specific classes or Generic classes[(56)]. Resolvers registered for a single component are only called when exactly that component is being handled. Resolvers registered for a certain class are called for all components of this type and derived classes.

A resolver may be registered for one or several components and/or classes. If no parameter is specified the resolver will be called for components of all classes. For example, a `NameResolver` or a `FeatureResolver` registered globally will be called for each and every name or feature. This is similar to but more efficient than registering them on the `java.lang.Object` class in the case of Java applications.

You may set up resolvers for various tasks and register them at run time. In order to install a resolver permanently, put the SUT script node for the resolver directly after the Wait for client to connect[(670)] node in the start sequence of the SUT.

If multiple resolvers are registered globally or registered on the same object or class, the resolver added last will be called first. The first resolver returning a non-null value determines the outcome.

Since a resolver will be called for each instance of the component, respectively class, displayed in the GUI you should implement time-saving algorithms for the resolvers. For example, in a Jython script the execution of `string[0:3] == "abc"` is faster than `string.startswith("abc")`.

All exceptions thrown inside a name resolver will be caught and handled by the `ResolverRegistry`. However, instead of dumping a stack trace, the registry will only print a short message like "Exception inside NameResolver" because some resolvers may be called very often, and a buggy resolver printing a stack trace for every error would flood the net and the client terminal. Therefore name resolvers should include their own error handling. This can still generate a lot of output in some cases, but the output will be more useful than a Java stack trace.

The `resolvers` module is always automatically available in all SUT script nodes.

Most examples in the manual are implemented as Jython scripts. In section 52.1.7[(1023)]

you will find examples for Groovy SUT script nodes.

52.1.3 `addResolver`

The generic function `addResolver` has a central role in the `resolvers` module. Given the name of the defined method and its parameters it identifies the respective object and its specific function for registering the resolver.

---

 void addResolver(String resolverName, Method method, Object target=None, ...)

Register the `resolver` determined by the given method for the given target(s). If another resolver was previously registered under the given name, deregister that first.

Parameters

name	The name under which to register the resolver.
method	The method implementing the resolver method. The name of the method defines the type of the registered resolver, i.e. for Groovy this has to be a MethodClosure. Valid names are e.g.: `getName`, `getClassName`, `getGenericClassName`, `getFeature`, `getExtraFeatures`, `getItemName`, `getItemValue`, `getItemNameByIndex`, `getTree` and `getTreeColumn`, `isInterestingParent`, `getTooltip`, `getId`, `isEnabled`, `isVisible`, `getMainText`, `matchExtraFeature`, `isBusy`, `isGlassPaneFor`, `sync` and `applicationIsBusy`.
target	One or more optional targets to register the resolver for. Each can be any of the following:

- An individual component

- The fully qualified name of a class

If no target is given a global resolver for all components is registered.

`void addResolver(String resolverName, Object object, Object target=None, ...)`

Register the `resolver` determined by the given method(s) of the object for the given targets. If another resolver was previously registered under the given name, deregister that first.

Parameters

`name`	The name under which to register the resolver.
`object`	An object or a class providing one or more resolver methods. Depending on the method names the respective resolver is registered. For valid method names see the description of `addResolver` above.
`target`	One or more optional targets to register the resolver for. Each can be any of the following: • An individual component • The fully qualified name of a class If no target is given a global resolver for all components is registered.

`History`

Resolvers have quite a history in QF-Test. Up to QF-Test version 4.1 you had to call a function specific to each resolver interface in order to register a certain resolver type. You may continue to use those functions. However, they are no longer described in the manual. The flexible `addResolver` function replaces the following functions, among others, of the `resolvers` module:

- `addNameResolver2(String name, Method method, Object target=None, ...)`

- `addClassNameResolver(String name, Method method, Object target=None, ...)`

- `addGenericClassNameResolver(String name, Method method, Object target=None, ...)`

- `addFeatureResolver2(String name, Method method, Object target=None, ...)`

- `addExtraFeatureResolver(String name, Method method, Object target=None, ...)`

- `addItemNameResolver2(String name, Method method, Object target=None, ...)`

- `addItemValueResolver2(String name, Method method, Object target=None, ...)`

- `addTreeTableResolver(String name, Method getTable, Method getColumn=None, Object target=None)`

- `addTooltipResolver(String name, Method method, Object target=None, ...)`

- `addIdResolver(String name, Method method, Object target=None,...)`

52.1.4 `removeResolver`

The function `removeResolver` may be used to deregister resolvers installed via the `resolvers` module.

Often, resolvers are registered directly after the start of the application and remain active during the full time of test execution. In some cases, however, resolvers are required only for handling a certain component and then need be to removed, either due to performance issues or because the effect of the resolver is not desirable for other components.

There are two functions for deregistration. The first, `removeResolver` deregisters a single resolver, the second, `removeAll`, removes all resolvers registered by the user.

---

void removeAll()

Deregister all resolvers registered via the `resolvers` module from all targets they were registered for.

---

void removeResolver(String name)

Deregister a resolver from all the targets it was registered for.

Parameters

name	The name the resolver was registered under.

The example first removes a resolver registered under the name "menuItems", then deregister all resolvers registered via the `resolvers` module.

```
resolvers.removeResolver("menuItems")
resolvers.removeAll()
```

Example 52.2: SUT script deregistering a resolver

52.1.5 `listNames`

Return a list of resolver names registered via the `resolvers` module.

`List<String> listNames()`

List the registered resolvers.

The example checks whether a certain resolver has been registered. If not, an error message is written to the run log.

```
if (! resolvers.listNames().contains("specialNames")) {
    rc.logError("Special names resolver not registered!")
}
```

Example 52.3: Groovy SUT script searching for a certain resolver registered via the `resolvers` module

52.1.6 Accessing 'Best label'

When you want to access the Best label[69] from within a resolver, you can use the method `rc.engine.helper.getBestLabel()`.

`String getBestLabel(Component node)`

Returns the Best label[69].

Parameters

node	The component for with to fetch the best label.
Returns	The best label for the component.

The example shows a name resolver transferring the best label to the Name attribute.

```
_h = rc.engine.helper
def getName(node, name):
    label = _h.getBestLabel(node)
        return label
resolvers.addResolver("labelAsName", getName, "TextField", "TextArea")
```

Example 52.4: NameResolver using the best label

52.1.7 The `NameResolver` Interface

3.1+

The `NameResolver` works on the Name attribute value of a Component node.

After QF-Test determined the name of a GUI element the registered `NameResolvers` get a chance to override or suppress this name. The first resolver that returns a non-null value determines the outcome. If no resolvers are registered or all of them return null the original name is used.

A `NameResolver` can change (or provide) the name of a GUI element as set with `setName()` for AWT/Swing, `setId()` or the `fx:id` attribute for JavaFX, `setData(name, ...)` for SWT or via the 'ID' attribute of a DOM node for web applications. It can be very useful when setting names in the source code is not an option, like for third-party code or when child components of complex components are not readily accessible. For example, QF-Test provides a name resolver for the Java Swing `JFileChooser` dialog, which you can read more about in the Tutorial chapter 'The Standard Library'.

In some cases it may be desirable to suppress an element's name, for example for names which are not unique or which - even worse - vary depending on the situation. To do so, getName should return the empty string.

Technologies: AWT/Swing, JavaFX, SWT. For web applications please use Install CustomWebResolver[796] node as described in Improving component recognition with a `CustomWebResolver`[945]. It was optimized for web elements and is more performant. Just in case the functionality provided there is insufficient make use of the `NameResolver`.

A `NameResolver` needs to implement the following method:

String getName(Object element, String name)

Determine the name of a GUI element.

Parameters

element	The element to determine the name for.
name	The original name QF-Test would use without a resolver.
Returns	The name to use or null if the element is not handled by the resolver. Returning an empty string suppresses the original name.

The first example is a `NameResolver` returning the text of the menu item as name for components of the generic class `MenuItems` for which the QF-Test standard resolver could not determine a name.

```
def getName(menuItem, name):
    if not name:
        return menuItem.getLabel()
resolvers.addResolver("menuItems", getName, "MenuItem")
```

Example 52.5: Jython name resolver for `MenuItems`

Give it a try. Copy the example above into a <u>SUT script</u>[(631)] node and execute it. If your application is based on SWT instead of Swing, replace `getLabel()` with `getText()`. Then record some menu actions into a new, empty test suite. You'll find that all recorded menu item components without name will now have names set according to their labels. If `setName` is not used in your application and the labels of menu items are more or less static while the structure of the items often changes, this can be a very useful feature.

The second example is a name resolver assigning a defined name ('Serial number') to a component which would otherwise have a partially dynamic name (e.g. 'Serial no: 100347'). It is registered for a specific Java Swing class.

```
def getName(menuItem, name):
    if name and name[0:10] == "Serial no:":
        return "Serial number"
resolvers.addResolver("lfdNr", getName, "javax.swing.JMenuItem")
```

Example 52.6: Jython `NameResolver` for a specific class

The following Groovy example returns the text of the menu item as the name for a component the QF-Test standard resolver did not find a name for.

```
def getName(def menuItem, def name) {
    if (name == null) {
        return menuItem.getLabel()
    }
}
resolvers.addResolver("menuItems", this.&getName, "MenuItem")
// You could also code it shorter:
//   resolvers.addResolver("menuItems", this, "MenuItem")
// since every Groovy script represents an object
// and addResolver(...) for objects registers
// all methods of the object as a resolver if possible.
```

Example 52.7: Groovy `resolver` for `MenuItems`

A resolver can be registered for multiple component classes at once:

```
def getName(com, name):
    return com.getText()
resolvers.addResolver("labels", getName, "Label", "Button")
```

Example 52.8: Register a `Resolver` for multiple classes

52.1.8   The **`GenericClassNameResolver`** Interface

A `GenericClassNameResolver` can assign generic classes (chapter 59[(1186)]) to arbitrary components. It can be used to make recorded components more readable and to register additional resolvers for the newly created classes.

Technologies: all

You should only use this resolver with a web application if the Install CustomWebResolver[(796)] node is insufficient.

After QF-Test determined the generic class name of a GUI element the registered `GenericClassNameResolvers` get a chance to override this generic class name. The first resolver that returns a non-null value determines the outcome. If no resolvers are registered or all of them return null the original generic class name is used.

For performance reasons classes are cached so the resolver will only be called once at the most for each element. If you change your resolver you need to re-load or to close and re-open the area which shows the component.

If a generic class name was already assigned to an element via a CustomWebResolver, no GenericClassNameResolvers will be called for that element.

A `GenericClassNameResolver` needs to implement the following method:

`String getGenericClassName(Object element, String name)`
Determine the name of the generic class of a GUI element.
Parameters

`element`	The element to determine the generic class name for.
`name`	The original generic class name QF-Test would use without a resolver. May be null.
Returns	The generic class name to use or null if the element is not handled by this resolver. An empty string to suppress the generic class determined by QF-Test.

52.1.9   The **`ClassNameResolver`** Interface

The `ClassNameResolver` can control the class QF-Test records for a component. It can be used to make recorded components more readable and to register additional resolvers for the newly created classes. However, we generally recommend the use of Generic classes[(1186)] instead. To register generic classes you should use the `GenericClassNameResolver` (section 52.1.8[(1026)]).

Technologies: all

You should only use this resolver with a web application if the

Install CustomWebResolver[(796)] node is insufficient.

A `ClassNameResolver` needs to implement the following method:

String getClassName(Object element, String name)

Determine the name of the class of a GUI element.

Parameters

element	The element to determine the class name for.
name	The original class name QF-Test would use without a resolver.
Returns	The class name to use or null if the element is not handled by this resolver.

After QF-Test determined the class name of a GUI element the registered `ClassNameResolvers` get a chance to override this class name. The first resolver that returns a non-null value determines the outcome. If no resolvers are registered or all of them return null the original class name is used. The resolver is free to return any arbitrary class name. Those names will be treated as normal classes in QF-Test internal methods.

For performance reasons classes are cached so the resolver will only be called once at the most for each element. If you change your resolver you need to re-load or to close and re-open the area which shows the component.

52.1.10 The `FeatureResolver` Interface

A `FeatureResolver` can provide a feature for a GUI element.

After QF-Test determined the feature of a GUI element the registered `FeatureResolvers` get a chance to override or suppress this feature. The first resolver that returns a non-null value determines the outcome. If no resolvers are registered or all of them return null the original feature is used.

To suppress an element's feature `getFeature` should return the empty string.

Technologies: AWT/Swing, JavaFX, SWT. For web applications please use the Install CustomWebResolver[(796)] node as described in Improving component recognition with a `CustomWebResolver`[(945)]. It was optimized for web elements and is more performant. Just in case the functionality provided there is insufficient make use of the `FeatureResolver`.

A `FeatureResolver` needs to implement the following method:

String getFeature(Object element, String feature)

Determine the feature of a GUI element.

Parameters

element	The element to determine the feature for.
feature	The original feature QF-Test would use without a resolver.
Returns	The feature to use or null if the element is not handled by this resolver. Returning an empty string suppresses the original feature.

The following example implements a feature resolver returning the title of the panel border as feature for Java/Swing panels.

```
def getFeature(com, feature):
    try:
        title = com.getBorder().getInsideBorder().getTitle()
        if title != None:
            return title
    except:
        pass
resolvers.addResolver("paneltitle", getFeature, "Panel")
```

Example 52.9: A `FeatureResoler` for Java/Swing `Panels`

52.1.11 The `ExtraFeatureResolver` Interface

An `ExtraFeatureResolver` can add, change or delete an Extra feature in the Extra features table for a GUI element. For this purpose the interface provides a number of methods.

Instances of the class `de.qfs.apps.qftest.shared.data.ExtraFeature` represent one Extra feature for a GUI element, comprising its name and value along with information about whether the feature is expected to match, whether it is a regular expression and whether the match should be negated. For possible states the class defines the constants STATE_IGNORE, STATE_SHOULD_MATCH and STATE_MUST_MATCH.

After QF-Test determined the Extra features for a GUI element, the registered `ExtraFeatureResolvers` get a chance to override these features. In contrast to other resolvers, QF-Test does not stop when the first resolver returns a non-null value. Instead it passes its result as input to the next resolver which makes it possible to register several `ExtraFeatureResolvers` that handle different Extra features. If no resolvers are registered or all of them return null, QF-Test will proceed to use the original set.

Of course, in order to be able to implement the `getExtraFeatures` method properly, you need to know the details for the API of the classes involved, namely `ExtraFeature` and `ExtraFeatureSet` described below - after the examples.

Technologies: AWT/Swing, JavaFX, SWT. For web applications please use the Install CustomWebResolver[796] described in Improving component recognition with a CustomWebResolver[945]. It was optimized for web elements and is more performant. Only if the functionality provided there is insufficient should you use the `ExtraFeatureResolver`.

To ensure consistency when capturing and replaying `qfs:label*` Extra feature variants as well as mapping them to and from SmartID, some constraints should be maintained. They do not apply to the legacy `qfs:label` Extra features, which stands on its own.

When handling `qfs:label*` variants in an `ExtraFeatureResolver` you have to make sure that the whole set of the variants remains consistent. This means

- There should be at most one `qfs:label*` variant with "Should match", the rest should be "Ignore".

- `qfs:labelBest` should either be the "Should match" variant or it should have the same value as the "Should match" variant.

QF-Test itself maintains those constraints when determining associated labels. To make it easier for `ExtraFeatureResolvers` to do so as well, the `ExtraFeatureSet` class manages those constraints when called from an `ExtraFeatureResolver`:

- If the value of `qfs:labelBest` is changed and it has the state "Ignore", the value of the "Should match" variant is automatically changed as well.

- If the value of the "Should match" variant is changed, the value of `qfs:labelBest` is automatically changed as well.

- If a `qfs:label*` variant is set to "Should match" all others are changed to ignore and `qfs:labelBest` is updated accordingly.

Note   If there are `qfs:label*` variants with the status "Must match" or with a regular expression constraint handling is immediately deactivated.

A `ExtraFeatureResolver` needs to implement the following method:

**ExtraFeatureSet getExtraFeatures(Object element,
ExtraFeatureSet features)**

Determine extra features for a GUI element.

Parameters

element	The element to determine the extra features for.
features	The extra features determined by QF-Test itself, an empty set in case there are none. These can be modified in place or ignored and a different ExtraFeatureSet returned.
Returns	The original features modified in place or a different set, null if the element is not handled by this resolver. To suppress all the element's original extra features return an empty ExtraFeatureSet.
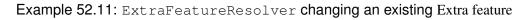
The first example implements an `ExtraFeatureResolver` adding the title of a Java/Swing dialog as an Extra feature with the status "must match" (STATE_MUST_MATCH). This comes in handy when component recognition depends on the correct title of the dialog.

```
def getExtraFeatures(node, features):
    try:
        title = node.getTitle()
        features.add(resolvers.STATE_MUST_MATCH,"dialog.title", title)
        return features
    except:
        pass
resolvers.addResolver("dialog title", getExtraFeatures,"Dialog")
```

Example 52.10: `ExtraFeatureResolver` adding an Extra feature for Java/Swing dialogs

The following example shows how to change an existing Extra feature. The example handles `qfs:labelBest`, which triggers special treatment of the `qfs:label*` variants as described above: if they adhere to the constraints, QF-Test will update the respective `qfs:label*` variants so that the whole set will remain consistent.

```
def getExtraFeatures(node, features):
    label = features.get("qfs:labelBest")
    if label and label.getValue() == "unwanted":
        label.setValue("wanted")
        return features
resolvers.addResolver("change label", getExtraFeatures)
```

Example 52.11: `ExtraFeatureResolver` changing an existing Extra feature

The next example shows how to change the state of the `qfs:label*` variant with the state 'should match' to 'ignore':

```
def getExtraFeatures(node, features) {
    def labelFeature = features.getShouldMatchLabel()
    if (labelFeature) {
        labelFeature.setState(resolvers.STATE_IGNORE)
        return features
    }
}
resolvers.addResolver("get label example", this)
```

Example 52.12: An `ExtraFeatureResolver` (Groovy) changing the state of the Extra feature

Thanks to the constraints described above, a simple `ExtraFeatureResolver` that was formerly written as

```
def getExtraFeatures(node, features):
    label = features.get("qfs:label")
    if label and label.getValue() == "unwanted":
        label.setValue("wanted")
        return features
resolvers.addResolver("change label", getExtraFeatures)
```

Example 52.13: `ExtraFeatureResolver` changing an existing Extra feature

can simply be updated to example `ExtraFeatureResolver` changing an existing Extra feature [1030] above.

In the following you will find the description of the APIs of the classes `ExtraFeature` and `ExtraFeatureSet`.

ExtraFeature ExtraFeature(String name, String value)

Create a new ExtraFeature with the default state STATE_IGNORE.

Parameters

name	The name of the ExtraFeature.
value	The value of the ExtraFeature.

`ExtraFeature ExtraFeature(int state, String name, String value)`

Create a new ExtraFeature with a given state.

Parameters

`state`	The state of the ExtraFeature. Values allowed are STATE_IGNORE, STATE_SHOULD_MATCH, STATE_MUST_MATCH.
`name`	The name of the ExtraFeature.
`value`	The value of the ExtraFeature.

`String getName()`

Get the name of the ExtraFeature.

Returns	The name of the ExtraFeature.

`boolean getNegate()`

Get the negate state of the ExtraFeature.

Returns	The negate state of the ExtraFeature.

`boolean getRegexp()`

Get the regexp state of the ExtraFeature.

Returns	Whether the ExtraFeature's value is a regular expression.

`int getState()`

Get the state of the ExtraFeature.

Returns	The state of the ExtraFeature.

`String getValue()`

Get the value of the ExtraFeature.

Returns	The value of the ExtraFeature.

`void setName(String name)`

Set the name of the ExtraFeature.

Parameters

`name`	The name to set.

`void setNegate(boolean negate)`

Set the negate state of the ExtraFeature.

Parameters

`negate`	The negate state to set.

`void setRegexp(boolean regexp)`

Set the regexp state of the ExtraFeature.

Parameters

`regexp`	The regexp state to set.

`void setState(int state)`

Set the state of the ExtraFeature.

Parameters

`state`	The state to set.

`void setValue(String value)`

Set the value of the ExtraFeature.

Parameters

`value`	The value to set.

The class `de.qfs.apps.qftest.shared.data.ExtraFeatureSet` collects `ExtraFeatures` into set:

`ExtraFeatureSet ExtraFeatureSet()`

Create a new, empty ExtraFeatureSet.

`void add(ExtraFeature extraFeature)`

Add an ExtraFeature to the set, potentially replacing a feature with the same name.

Parameters

`extraFeature`	The ExtraFeature to add.

`void add(String name, String value)`

Add a new extra feature to the set with the state STATE_IGNORE, potentially replacing a feature with the same name.

Parameters

`name`	The name of the ExtraFeature.
`value`	The value of the ExtraFeature.

`void add(int state, String name, String value)`

Add a new extra feature with the given state to the set, potentially replacing a feature with the same name.

Parameters

`state`	The state of the ExtraFeature. Values allowed are resolvers.STATE_IGNORE, resolvers.STATE_SHOULD_MATCH, resolvers.STATE_MUST_MATCH.
`name`	The name of the ExtraFeature.
`value`	The value of the ExtraFeature.

`ExtraFeature get(String name)`

Get an ExtraFeature from the set.

Parameters

`name`	The name of the ExtraFeature to get.
Returns	The ExtraFeature or null if no feature by that name is stored in the set.

`ExtraFeature getShouldMatchLabel()`

Get the `qfs:label*` ExtraFeature variant with the state "Should match" from the set. In case there are more than one, the first one will be returned.

Returns	The ExtraFeature or null if there is no `qfs:label*` ExtraFeature variant with the state "Should match".

`ExtraFeature remove(String name)`

Remove an extraFeature from the set.

Parameters

`name`	The name of the ExtraFeature to remove.
Returns	The ExtraFeature that was removed or null if no feature by that name was stored in the set.

`ExtraFeature[] toArray()`

Get all ExtraFeatures in the set.

Returns	An array of the contained ExtraFeatures, sorted by name.

52.1.12   The `ItemNameResolver` Interface

3.1+

An `ItemNameResolver` can change (or provide) the textual representation of the index for addressing a sub-item of a complex component.

After QF-Test determined the name for an item's index the registered `ItemNameResolvers` get a chance to override. The first resolver that returns a non-null value determines the outcome. If no resolvers are registered or all of them return null the original name is used.

Technologies: AWT/Swing, JavaFX, SWT. For web applications please use the Install CustomWebResolver[(796)] node described in Improving component recognition with a `CustomWebResolver`[(945)].   It was optimized for web elements and is more performant. Only if the functionality provided there is insufficient should you use the `ItemNameResolver`.

An `ItemNameResolver` needs to implement the following method:

String getItemName(Object element, Object item, String name)
Determine the name of an item of a complex GUI element that will be used for the textual representation of the item's index.

Parameters

element	The GUI element to which the item belongs.
item	The item to get the name for. Its type depends on the GUI element and the registered `ItemResolvers` as described in section 52.4.5[(1071)].
name	The original name that QF-Test would use without a resolver.
Returns	The name to use or null if the resolver does not handle this element or item.

The example implements an `ItemNameResolver` making the ID of a `JTable` available as index:

```
def getItemName(tableHeader, item, name):
    id = tableHeader.getColumnModel().getColumn(item).getIdentifier()
    if id:
        return str(id)
resolvers.addResolver("tableColumnId", getItemName,
                      "javax.swing.table.JTableHeader")
```

Example 52.14: An `ItemNameResolver` for `JTableHeader`

52.1.13 The `ItemValueResolver` Interface

3.1+

The `ItemValueResolver` is used to improve the textual check of elements.

An `ItemValueResolver` can change (or provide) the textual representation of the value a sub-item of a complex component as used by a Check text[(708)] node or retrieved via a Fetch text[(740)] node.

After QF-Test determined the value for an item's index the registered `ItemValueResolvers` get a chance to override. The first resolver that returns a non-null value determines the outcome. If no resolvers are registered or all of them return null the original value is used.

Technologies: AWT/Swing, JavaFX, SWT. For web applications please use the Install CustomWebResolver[(796)] as described in Improving component recognition with a CustomWebResolver[(945)]. It was optimized for web elements and is more performant. Just in case the functionality provided there is insufficient use the `ItemValueResolver`.

An `ItemValueResolver` needs to implement the following method:

`String getItemValue(Object element, Object item, String value)`
Determine the value of an item of a complex GUI element that will be used for its textual representation in a Check text[708] or a Fetch text[740] node.
Parameters

`element`	The GUI element to which the item belongs.
`item`	The item to get the value for. Its type depends on the GUI element and the registered `ItemResolvers` as described in section 52.4.5[1071].
`value`	The original value QF-Test would use without a resolver.
Returns	The value to use or null if the resolver does not handle this element or item.

52.1.14 The `TreeTableResolver` Interface

A `TreeTableResolver` helps QF-Test recognize TreeTable components. A TreeTable is a mixture between a table and a tree. It is not a standard Swing component, but most TreeTables are implemented alike using a tree as the renderer component for one column of the table. Once QF-Test recognizes a TreeTable as such, it treats the row indexes of all table cells as tree indexes, which is a lot more useful in that context than standard table row indexes. In addition, geometry information for cells in the tree column is based on tree nodes instead of table cells.

Technologies: AWT/Swing

Note   The interface is only relevant for AWT/Swing. For SWT and JavaFX multi-column trees are support by QF-Test automatically. For web frameworks the TreeTable is defined by the (custom) web resolver (see Improving component recognition with a `CustomWebResolver`[945]).

A `TreeTableResolver` needs to implement to following two methods:

`JTree getTree(JTable table)`
Determine the tree component used to implement a TreeTable.
Parameters

`table`	The JTable component to determine the tree for.
Returns	The tree or null if the JTable is a plain table and not a TreeTable.

---

int getTreeColumn(JTable table)
Determine the column index of the tree component in a TreeTable. Most implementations place the tree in the first column, in which case the index is 0.
Parameters

table	The JTable component to determine the tree's column index for.
Returns	The column index or -1 if the JTable is a plain table and not a TreeTable. The column index must always be given in the table's model coordinates, not in view coordinates.

Most `TreeTableResolvers` are trivial to implement. The following Jython example works well for the `org.openide.explorer.view.TreeTable` component used in the popular netBeans IDE, provided that the resolver is registered for the TreeTable class:

```
def getTreeMethod(table):
        return table.getCellRenderer(0,0)
    def getColumn(table):
        return 0
    resolvers.addResolver("treetableResolver", getTreeMethod, \
    getColumn, "org.openide.explorer.view.TreeTable")
```

Example 52.15: `TreeTableResolver` for netBeans IDE

The following example shows a typical `TreeTableResolver`.

```
def getTree(table):
    return table.getTree()
def getColumn(table):
    return 0
resolvers.addResolver("treeTable", getTree, getColumn,
                          "my.package.TreeTable")
```

Example 52.16: `TreeTableResolver` for Swing TreeTable with optional `getColumn` method

As practically all TreeTables implement the tree in the first column of the table the `getColumn` method is optional. When none is passed QF-Test automatically creates a default implementation for the first column:

```
def getTree(table):
    return table.getTree()
resolvers.addResolver("treeTable", getTree, None,
                          "my.package.TreeTable")
```

Example 52.17: Simplified `TreeTableResolver`

If no dedicated `getTree` method is available, the cell renderer of the column containing the tree (typically 0) might work, as it is typically derived from `JTree`.

```
def getTree(table):
    return table.getCellRenderer(0,0)
resolvers.addResolver("treeTable", getTree,
                          "my.package.TreeTable")
```

Example 52.18: Simplified `TreeTableResolver` using the method `getCellRenderer`

52.1.15 The `InterestingParentResolver` Interface

An `InterestingParentResolver` influences which components will be treated as interesting or ignorable by QF-Test recording. This, in turn, determines whether a Component node will be created for a component.

Technologies: AWT/Swing, JavaFX, SWT. For web applications please use the Install CustomWebResolver[796] node as described in Improving component recognition with a `CustomWebResolver`[945]. It was optimized for web elements and is more performant. Just in case the functionality provided there is insufficient make use of the `InterestingParentResolver`.

An `InterestingParentResolver` needs to implement the following method:

Boolean isInterestingParent(Object parent, boolean interesting)
Determine whether a parent element is interesting.
Parameters

parent	The (direct or indirect) parent element.
interesting	Whether QF-Test considers the parent interesting without resolver.
Returns	Boolean.TRUE if interesting, Boolean.FALSE if not, null if this resolver cannot tell either way.

52.1.16 The `TooltipResolver` Interface

A `TooltipResolver` can provide a tooltip for a component. A tooltip is one of the texts considered for the 'qfs:label' Extra feature.

Technologies: AWT/Swing, JavaFX, SWT. For web applications please use the Install CustomWebResolver[(796)] node as described in Improving component recognition with a `CustomWebResolver`[(945)]. It was optimized for web elements and is more performant. Just in case the functionality provided there is insufficient make use of the `TooltipResolver`.

A `TooltipResolver` needs to implement the following method:

String getTooltip(Object element, String tooltip)

Determine the tooltip of a GUI element.

Parameters

element	The element to determine the tooltip for.
tooltip	The original tooltip QF-Test would use without a resolver.
Returns	The tooltip to use or null if the element is not handled by this resolver. Returning an empty string suppresses the original tooltip.

52.1.17 The `IdResolver` interface

An `IdResolver` allows modifying or even removing the 'ID' attribute of a DomNode. When QF-Test registers the DOM nodes of a web page it also caches the "id" attribute of those nodes. Depending on the option Turn 'ID' attribute into name where "unique enough"[(491)] the value of the "id" attribute will even be taken as name for the component. As many web pages or component libraries generate such IDs automatically it's a very common requirement to modify that ID in order to get stable and reliable component recognition.

There are three possibilities to deal with such automatically generated IDs:

- The simplest method influencing the IDs can be achieved by using the Install CustomWebResolver[(796)] node. There you should configure the category `autoIdPatterns`. This parameter allows to specify dedicated values to ignore like `myAutoId` or even regular expressions like `auto.*`, which ignores any ID beginning with `auto`.

- In case you have introduced a custom attribute, which should act as id instead of the original 'ID' attribute, you should also use Install CustomWebResolver[(796)]. There you should configure the category `customIdAttributes`. It allows to specify custom attributes which will be used for determining the 'ID'.

- You can activate the option <u>Eliminate all numerals from 'ID' attributes</u>⁽⁴⁹¹⁾ to ignore any numerals from the ID.

- In case you would like to implement a complex algorithm you need to implement an `IdResolver`.

The options mentioned above can also be combined and don't exclude each other. In case you decide to implement a custom algorithm you should always use an `IdResolver`. You should take care that the 'ID' attribute of a node can show up in multiple places. The most notably place is the attribute <u>Name</u>⁽⁸²⁵⁾ of the node (depending on the option <u>Turn 'ID' attribute into name where "unique enough"</u>⁽⁴⁹¹⁾), its <u>Feature</u>⁽⁸²⁵⁾ and its <u>Extra feature</u>⁽⁸²⁵⁾. Because of that many locations you should prefer implementing an `IdResolver` over implementing individual `Name-`, `Feature-` and `ExtraFeatureResolvers`. More importantly, changing a node's 'ID' attribute can have a major impact on whether the attribute is unique and QF-Test's mechanism for using an ID as a Name takes uniqueness into account, so an `IdResolver` is allowed to return non-unique IDs whereas a `NameResolver2` is not.

Technologies: Web

The `de.qfs.apps.qftest.extensions.IdResolver` interface consists of a single method:

`String getId(DomNode node, String id)`
Determine the ID of a `DomNode`. The resolved ID will be cached and can later be retrieved via `node.getId()`, whereas `node.getAttribute("id")` always returns the original, unmodified 'ID' attribute.
Parameters

`node`	The `DomNode` to determine the ID for.
`id`	The ID that QF-Test has determined, possibly with suppressed numerals, depending on the setting of the option <u>Eliminate all numerals from 'ID' attributes</u>⁽⁴⁹¹⁾. To implement the resolver based on the original 'ID' attribute, simply fetch this with `node.getAttribute("id")`.
Returns	The ID or null if the element is not handled by this resolver. Returning an empty string will suppress or hide the node's actual ID.

52.1.18   The `EnabledResolver` Interface

An `EnabledResolver` provides information about when to consider a component active or inactive. AWT/Swing Components have a respective attribute. Web and

JavaFX, however, have special stylesheet classes that need to be evaluated via the `EnabledResolver`.

Technologies: JavaFX, Web

An `EnabledResolver` needs to implement the following method:

Boolean isEnabled(Object element, boolean enabled)

Determine whether a GUI element is regarded as active or inactive.

Parameters

element	The element to determine the enabled state for.
enabled	The original state QF-Test would use without a resolver.
Returns	True or false. Null if the element was not handled by the resolver.

The example determines the enabled state of a web node via the css class `v-disabled`.

```
def isEnabled(element):
    try:
        return not element.hasCSSClass("v-disabled")
    except:
        return True
resolvers.addResolver("vEnabledResolver",isEnabled, \
                    "DOM_NODE")
```

Example 52.19: An `EnabledResolver`

52.1.19 The `VisibilityResolver` Interface

4.1+

A `VisibilityResolver` influences whether to consider a web element to be visible.

Technologies: Web

A `VisibilityResolver` needs to implement the following method:

Boolean isVisible(Object element, boolean visible)

Determine whether a GUI element is regarded visible.

Parameters

element	The web element to determine the visible state for.
visible	The original state QF-Test would use without a resolver.
Returns	True or false. Null if the element was not handled by the resolver.

The resolver in the example below returns false for the visibility state of the web element in case it is opaque.

```
import re
def getOpacity(element):
    style = element.getAttribute("style")
    if not style:
        return 1
    m = re.search("opacity:\s*([\d\.]+)", style)
    if m:
        return float(m.group(1)) == 0.4
    else:
        return 1
def isVisible(element,visible):
    while visible and element:
        visible = getOpacity(element) > 0
        element = element.getParent()
    return visible
resolvers.addResolver("opacityResolver",isVisible)
```

Example 52.20: A `VisibilityResolver`

52.1.20 The `MainTextResolver` Interface

A `MainTextResolver` determines the primary line of text of a component, which then may be used for the Feature[(825)], the `qfs:label*` variants[(67)] etc.

Technologies: AWT/Swing, JavaFX, SWT, Web

A `MainTextResolver` needs to implement the following method:

String getMainText(Object element, String text)

Determine the 'main' text of a component

Parameters

element	The GUI element to determine the text for.
text	The original text QF-Test would use without a resolver.
Returns	The 'main' text. Null if the element was not handled by the resolver. Returning an empty string suppresses the original text.

The resolver in the example removes the string TO-DO from the 'main' text of all components.

```
def getMainText(element,text):
    if text:
        return text.replace("TO-DO","")
resolvers.addResolver("removeMarkFromText", getMainText)
```

Example 52.21: A `MainTextResolver`

52.1.21 The `WholeTextResolver` Interface

4.1+

A `WholeTextResolver` determines the 'whole' text of a component, i.e. what should be used for checks, etc.

Technologies: AWT/Swing, JavaFX, SWT, Web

A `WholeTextResolver` needs to implement the following method:

String getWholeText(Object element, String text)

Determine the 'whole' text of a component

Parameters

element	The GUI element to determine the text for.
text	The original text QF-Test would use without a resolver.
Returns	The 'whole' text. Null if the element was not handled by the resolver. Returning an empty string suppresses the original text.

The resolver in the example removes the string `TO-DO` from the texts used for example for checks of TextFields and TextAreas.

```
def getWholeText(element,text):
    if text:
        return text.replace("TO-DO","")
resolvers.addResolver("removeMarkFromText", getWholeText, "TextField", "TextArea")
```

Example 52.22: A `WholeTextResolver`

52.1.22 The `BusyPaneResolver` Interfaces

4.1+

At text execution, QF-Test waits for BusyPanes covering other components to disappear before resuming in a determined state. A `BusyPaneResolver` influences whether to consider a component as being covered.

Technologies: AWT/Swing, JavaFX

A `BusyPaneResolver` needs to implement the following method:

Boolean isBusy(Object element)
Determine whether a component is currently being covered by a BusyPane or comparable component.
Parameters

element	The GUI element to determine the text state for.
Returns	True if the element cannot currently be accessed because of a BusyPane or similar.
	False otherwise.
	Null if the element was not handled by the resolver.

The resolver in the example below deactivates recognition of BusyPanes for components of the type "my.special.Component".

```
def isBusy():
    return false
resolvers.addResolver("neverBusyResolver",isBusy,"my.special.Component")
```

Example 52.23: A `BusyPaneResolver`

52.1.23 The `GlassPaneResolver` Interfaces

4.1+

When components (e.g. transparent ones) hide others components you can use a `GlassPaneResolver` to inform QF-Test of this relationship and thus redirect events to the correct component.

Technology: AWT/Swing

A `GlassPaneResolver` needs to implement the following method:

Object isGlassPaneFor(Object element, Object target)
Determine the relationship between an overlaying component and the actual target component.
Parameters

element	The GUI element events are received for.
target	The GUI element QF-Test would pass the event on to without a resolver.
Returns	The component to which to pass the events on to. Null if the element was not handled by the resolver.

The resolver in the example below deactivates passing on events through GlassPanes.

```
def isGlassPaneFor(element):
    return element
resolvers.addResolver("noGlassPaneResolver", isGlassPaneFor)
```

Example 52.24: A `GlassPaneResolver`

52.1.24 The `EventSynchronizer` Interface

4.1+

After replaying an event to the SUT QF-Test waits for synchronization with the respective Event Dispatch Thread. Via an `EventSynchronizer` you can tell QF-Test when the SUT is ready to accept the next event. It ought to be used when the SUT has a non-standard event synchronization.

Technologies: AWT/Swing, JavaFX, SWT, Web

An `EventSynchronizer` needs to implement the following method:

---

void sync(Object context)

Synchronization with the Event Dispatch Thread of the SUT.

Parameters

context	The context specified when registering the resolver.

The resolver in the following example stops execution on the Dispatch Thread until the next full second.

```
import time
def sync():
    t = time.time()
    full = int(t)
    delta = t - full
    time.sleep(delta)
resolvers.addResolver("timeSynchronizer",sync)
```

Example 52.25: An `EventSynchronizer`

52.1.25 The `BusyApplicationDetector` Interface

4.1+

Using a `BusyApplicationDetector` can tell QF-Test when to consider an application to be currently 'busy' and not in grade of accepting events.

Technologies: AWT/Swing, JavaFX, SWT, Web

A `BusyApplicationDetector` needs to implement the following method:

Boolean applicationIsBusy

Determine whether an application is currently 'busy'.

| **Returns** | True if the application is 'busy', false otherwise. |

The resolver in the example uses a SUT specific method to tell QF-Test it is 'busy':

```
def applicationIsBusy():
    return my.app.App.instance().isDoingDbSynchronization()
resolvers.addResolver("dbAccessDetector",applicationIsBusy)
```

Example 52.26: A `BusyApplicationDetector`

52.1.26 `Matcher`

The difference between a `matcher` and a `resolver` is that `matchers` are relevant for replay only. They have no effect on recordings. However, they are registered via the `resolvers` module as well.

A `matcher` can become useful when you are working with generic components or for keyword driven testing, if you do not record components.

4.1+

The `ExtraFeatureMatcher` Interface

An `ExtraFeatureMatcher` influences whether to consider an Extra feature QF-Test registered for the component as 'suitable'.

Technologies: AWT/Swing, JavaFX, SWT, Web

An `ExtraFeatureMatcher` needs to implement the following method:

Boolean matchExtraFeature(Object element, String name, String value, boolean regexp, boolean negate)

Check the Extra feature of a component.

Parameters

element	The GUI element to check the Extra feature for.
name	The name of the Extra feature.
value	The value of the Extra feature.
regexp	True, if `value` is a regular expression.
negate	True, if the check is to be negated.
Returns	True, if the ExtraFeature is suitable, false otherwise. Null if the element was not handled by the resolver.

The matcher in the example below checks the value of the Extra feature `my:label` against the `my-label` attribute of the web element.

```
import re
def matchExtraFeature(element, name, value, regexp, negate):
    if not name == "my:label":
        return None
    label = element.getAttribute("my-label")
    if label:
        if regexp:
            match = re.match(value,label)
        else:
            match = (value == label)
    else:
        match = False
    return (match and not negate) or (not match and negate)
resolvers.addResolver("myLabelResolver", matchExtraFeature)
```

Example 52.27: An `ExtraFeatureMatcher`

The resolver method call can be limited to a specific feature name by means of the special resolvers method `addSpecificExtraFeatureMatcher`:

```
import re
def matchExtraFeature(element, name, value, regexp, negate):
    label = element.getAttribute("my-label")
    if label:
        if regexp:
            match = re.match(value,label)
        else:
            match = (value == label)
    else:
        match = False
    return (match and not negate) or (not match and negate)
resolvers.addSpecificExtraFeatureMatcher("myLabelResolver", \
                                   matchExtraFeature, "my:label")
```

Example 52.28: Using `addSpecificExtraFeatureMatcher`

52.1.27   External Implementation

As an alternative to directly implementing a resolver in an SUT-script, it is possible to provide them as Java classes inside a JAR file in the plugin folder. In doing so, it is helpful to implement the aforementioned resolver interfaces (Basically, QF-Test is able to detect resolvers by their implemented method names).

To implement the interfaces provided by QF-Test, the file `qfsut.jar` has to be added to the development classpath. Most of the interfaces reside in the `de.qfs.apps.qftest.extensions` package, and the names of the interfaces which have two method parameters are suffixed with a "2". All Interfaces named `Item...` reside in the package `de.qfs.apps.qftest.extensions.items`. When calling `resolvers.addResolver(...)` in an SUT script, provide an instance of the implemented resolver class as argument.

52.2 The `webResolvers` module

For most use cases, registering a resolver using the `resolvers` module as described in the last chapter section 52.1[(1016)] is the best way to adapt the component recognition of QF-Test. If the custom resolvers are used for a *web* context and heavily interacting with the browser content directly, the `webResolvers` module provides an option to run the resolver code directly in the browser.

52.2.1 General Information

QF-Test internally caches the state of the displayed web page and runs most of the commands, which are usually invoked in a resolver, on the cached representation of the web page. This tremendously improves resolving time. Resolvers registered on the `webResolver` module are always executed directly in the browser, which might slow down component recognition. On the other hand, resolvers registered using the `webResolvers` module are convenient if it requires several downcalls to the browser, i.e. if using the `getAttributeIfSpecified` method or `evalJS(...)`.

In general, the `webResolvers` module mimics the `resolvers` module, so its methods and their usage is similar to those described in the last chapter. In difference to those, the methods have to be written as JavaScript functions, wrapped in Strings. The resolver code is then directly executed in the browser, so its function arguments are also Browser objects and JavaScript objects.

If you write your SUT script in JavaScript, you can use a function directly as argument for the register call. The given function will then be stringified and registered as resolver in the browser. Be aware that this resolver function is executed in the browser, not in the QF-Test SUT script environment, and variables are not shared between the two contexts.

The `webResolvers` module is limited to these resolvers types:

- `NameResolver` as described in section 52.1.7[(1023)]

- `ClassNameResolver` as described in section 52.1.9[(1026)]

- `GenericClassNameResolver` as described in <u>section 52.1.8</u>[(1026)]

- `FeatureResolver` as described in <u>section 52.1.10</u>[(1027)]

- `ExtraFeatureResolver` as described in <u>section 52.1.11</u>[(1028)]

- `ItemNameResolver` as described in <u>section 52.1.12</u>[(1034)]

- `ItemValueResolver` as described in <u>section 52.1.13</u>[(1035)]

- `InterestingParentResolver` as described in <u>section 52.1.15</u>[(1038)]

- `TooltipResolver` as described in <u>section 52.1.16</u>[(1039)]

- `IdResolver` as described in <u>section 52.1.17</u>[(1039)]

- `EnabledResolver` as described in <u>section 52.1.18</u>[(1040)]

- `VisibilityResolver` as described in <u>section 52.1.19</u>[(1041)]

- `MainTextResolver` as described in <u>section 52.1.20</u>[(1042)]

- `WholeTextResolver` as described in <u>section 52.1.21</u>[(1043)]

- `BusyApplicationDetector` as described in <u>section 52.1.25</u>[(1045)]

- `ExtraFeatureMatcher` as described in <u>section 52.1.26</u>[(1046)]

52.2.2 `addResolver`

The generic method `addResolver` has a central role in the `webResolvers` module. The method registers a new resolver. It identifies the type of the resolver using the name of the JavaScript function which not always is equal to the name of the variable which references the function) and its parameter count.

 `addResolver(String resolverName, String function, Object target=None, ...):`
Register the `webResolver` determined by the given JavaScript function for the given target(s). If another resolver was previously registered under the given name, deregister that first.
Parameters

`name`	The name under which to register the resolver.
`function`	The JavaScript function implementing the resolver. The name of the function defines the type of the registered resolver. Valid names are e.g.: `getName`, `getClassName`, `getGenericClassName`, `getFeature`, `getExtraFeatures`, `getItemName`, `getItemValue`, `isInterestingParent`, `getTooltip`, `getId`, `isEnabled`, `isVisible`, `getMainText`, `matchExtraFeature`, **and** `applicationIsBusy`.
`target`	One or more optional targets to register the resolver for. Each can be any of the following: • An individual component • The fully qualified tag name of a class If no target is given a global resolver for all components is registered.

52.2.3 `removeResolver`

The function `removeResolver` may be used to deregister resolvers installed via the `webResolvers` module.

 `void removeAll()`
Deregister all resolvers registered via the `webResolvers` module from all targets they were registered for.

 `void removeResolver(String name)`
Deregister a resolver from all the targets it was registered for.
Parameters

`name`	The name the resolver was registered under.

The example first removes a resolver registered under the name "myName", then deregisters all resolvers registered via the `webResolvers` module.

```
webResolvers.removeResolver("myName")
webResolvers.removeAll()
```

Example 52.29: SUT script deregistering webResolvers

52.2.4 `listNames`

Return a list of resolver names registered via the `webResolvers` module.

 List<String> listNames()

List the registered resolvers.

The example checks whether a certain resolver has been registered. If not, an error message is written to the run log.

```
if (! webResolvers.listNames().contains("specialNames")) {
    rc.logError("Special names resolver not registered!")
}
```

Example 52.30: Groovy SUT script searching for a certain resolver registered via the `webResolvers` module

52.2.5 Examples

Since the role and usage of *webResolvers* is equal to those registered using the `resolvers` module, we refer to the last chapter for a detailed description of the specific resolvers. Here, we demonstrate the usage of *webResolvers* in selected examples:

```
getNameFunc = """
function getName(node, name) {
    if (! name) return node.getAttribute("__name__");
}
"""
webResolvers.addResolver("myNames", getNameFunc, "DIV")
```

Example 52.31: Registering a `webNameResolver` in Jython

```
def getFeatureFunc = """
function getFeature(node, feature) {
    try {
        var title = node.firstChild.textContent;
        if (title) return title;
    } catch (e) {}
}
"""
webResolvers.addResolver("paneltitle", getFeatureFunc, "Panel")
```

Example 52.32: A `webFeatureResoler` registered using a Groovy SUT script

```
function getExtraFeatures(node, features) {
    try {
        const myId = node.getAttribute("myId");
        if (myId) {
            features.add(ExtraFeature.STATE_MUST_MATCH,"myId", myId);
            return features;
        }
    } catch (e) {}
}
webResolvers.addResolver("myId", getExtraFeatures);
```

Example 52.33: `webExtraFeatureResolver` adding an Extra feature for a special attribute using a JavaScript SUT script

```
function getExtraFeatures(node, features) {
    const labelFeature == features.get("qfs:label");
    if (labelFeature && labelFeature.value == "bad name") {
        labelFeature.value = "good name";
        return features;
    }
}
webResolvers.addResolver("get label example", getExtraFeatures,"TextField");
```

Example 52.34: `webExtraFeatureResolver` changing an existing Extra feature

```
getColumnIdFunc = """
function getColumnId(table, item, name) {
    var id;
    if (item[1] < 0 ) { // whole column addressed
        id = table.getElementsByTagName("TH")[0].id;
    }
    if (id) return id;
}
"""
webResolvers.addResolver("tableColumnId", getColumnIdFunc, "TableColumn")
```

Example 52.35: A `webItemNameResolver` registered using a Jython SUT script

```
def getTooltipFunc = """
function getTooltip(node, tooltip) {
    if (tooltip) {
        var userLang = navigator.language || navigator.userLanguage;
        if (userLang) {
            return tooltip + "-" + userLang;
        }
    }
}
"""
webResolvers.addResolver("tootip-lang", getTooltipFunc)
```

Example 52.36: A `webTooltipResoler` registered using a Groovy SUT script

```
function isEnabled(node) {
    try {
        return ! node.className.split(' ').contains("v-disabled");
    } catch (e) {
        return true;
    }
}
webResolvers.addResolver("vEnabledResolver",isEnabledFunc);
```

Example 52.37: A `webEnabledResolver`, registered using a JavaScript SUT script

```
function isVisible(element, visible) {
    const getOpacity = function(el) { // inner function, so it is part
                                      // of the registered resolver
        const style = window.getComputedStyle(el);
        const opacity = style.getPropertyValue("opacity")
        if (! opacity) return 1;
        return parseFloat(opacity);
    }
    return visible && getOpacity(element) > 0;
}
webResolvers.addResolver("opacityResolver",isVisible);
```

Example 52.38: A `webVisibilityResolver`

```
getMainTextFunc = """
function getMainText(element,text) {
    if (text) return text.replace("TO-DO","")
}
"""
webResolvers.addResolver("removeMarkFromText",getMainTextFunc)
```

Example 52.39: A `webMainTextResolver`, registered in a Jython SUT script

```
function isBusy(element) {
    const overlay = element.getElementsByClassName("overlay")[0];
    return !!overlay;
}
webResolvers.addResolver("overlayBusyResolver",isBusy);
```

Example 52.40: A `webBusyPaneResolver`

```
function applicationIsBusy() {
    const tasks = $specialFramework.waitingTasks();
    return tasks && tasks.length() > 0;
}
webResolvers.addResolver("taskChecker",applicationIsBusy);
```

Example 52.41: A `webBusyApplicationDetector`

```
function matchExtraFeature(element, name, value, regexp, negate) {
    if (name != "my:label") return;
    const label = element.getAttribute("my-label");
    var match = false;
    if (label) {
        if (regexp) {
            match = !! label.match(value);
        } else {
            match = (value == label);
        }
    }
    return negate ? ! match : match;
}
webResolvers.addResolver("myLabelResolver", matchExtraFeature)
```

Example 52.42: A `webExtraFeatureMatcher`

```
function matchExtraFeature(element, name, value, regexp, negate) {
    const label = element.getAttribute("my-label");
    var match = false;
    if (label) {
        if (regexp) {
            match = !! label.match(value);
        } else {
            match = (value == label);
        }
    }
    return negate ? ! match : match;
}
webResolvers.addSpecificExtraFeatureMatcher("myLabelResolver",
                                matchExtraFeature, "my:label");
```

Example 52.43:   Using   `addSpecificExtraFeatureMatcher`   for   a
`webExtraFeatureMatcher`

52.3 The `ResolverRegistry`

Resolvers of all kinds can be implemented via the `resolvers` module as Jython or Groovy scripts as described in section 52.1[(1016)]. Unless you want to understand how the `resolvers` module itself works or want to implement a resolver in Java you may skip the following sections.

This section describes how to implement a resolver directly by Java classes. However, this will make the code of your application depend on QF-Test classes. The preferred alternative is to implement the resolver interfaces in Jython or Groovy. That way the

whole mechanism can be strictly separated from the SUT and will not interfere with the build process.

Additionally to the resolver registration described in this section you need to implement the resolver interfaces described from <u>section 52.1.7</u>[(1023)] and following.

The need for describing such a method as an interface that has to implemented by a class makes resolvers difficult at Java level. Then an instance of that class has to be created and registered for use by QF-Test. If you don't get it right on first try, that instance has to be deregistered before a new instance from a new class can be created and instance thereof registered, otherwise there may be interference between the two versions of the resolver. Add some code for error handling and you've got many times more glue code than actual "flesh".

As stated in section <u>Implementation</u>[(1018)] all exceptions thrown inside a name resolver will be caught and handled by the `ResolverRegistry`. However, instead of dumping a stack trace, the registry will only print a short message like "Exception inside NameResolver" because some resolvers may be called very often, and a buggy resolver printing a stack trace for every error would flood the net and the client terminal. Therefore name resolvers should include their own error handling. This can still generate a lot of output in some cases, but the output will be more useful than a Java stack trace.

Before implementing a `resolver` in Java please have a look at sections <u>section 52.1.1</u>[(1016)] and <u>section 52.1.2</u>[(1018)], where everything not specific to scripts and the `resolvers module` itself holds true for resolvers implemented in Java as well.

The singleton class `de.qfs.apps.qftest.extensions.ResolverRegistry` is the central agent for registering and removing name resolvers.

The `ResolverRegistry` API is pretty straightforward:

`static String getElementName(Object element)`
This static method should be used instead of `com.getName()` or `widget.getData()` by resolvers that need to operate based on existing names of elements, except for a `NameResolver2` that gets just the original name passed to its `getName` method. This method suppresses trivial or AWT/Swing specific component names that QF-Test handles specially.
Parameters

`element`	The element to get the name for.
Returns	The name of the element or `null` if a trivial or special name is suppressed.

`static ResolverRegistry instance()`
There can only ever be one `ResolverRegistry` object and this is the method to get hold of this singleton instance.

Returns	The singleton `ResolverRegistry` instance.

---

static boolean isInstance(Object object, String className)
This static method should be used instead of instanceof (or isinstance() in Jython). It checks whether an object is an instance of a given class. The check is performed by name instead of through reflection, so conflicts with differing class loaders are prevented and there is no need to import the class to check against.
Parameters

object	The object to check.
className	The name of the class to test for.
Returns	True if the object is an instance of the given class.

void registerExtraFeatureResolver(ExtraFeatureResolver resolver)
Register a generic or global extra feature resolver.
Parameters

resolver	The resolver to register.

void registerExtraFeatureResolver(Object element, ExtraFeatureResolver resolver)
Register an extra feature resolver for an individual GUI element. The resolver does not prevent garbage collection and will be removed automatically when the element becomes unreachable.
Parameters

element	The GUI element to register for.
resolver	The resolver to register.

void registerExtraFeatureResolver(String clazz, ExtraFeatureResolver resolver)
Register an extra feature resolver for a GUI element class.
Parameters

clazz	The name of the class to register for.
resolver	The resolver to register.

void registerFeatureResolver2(FeatureResolver2 resolver)
Register a generic or global feature resolver.
Parameters

resolver	The resolver to register.

void registerFeatureResolver2(Object element, FeatureResolver2 resolver)

Register a feature resolver for an individual GUI element. The resolver does not prevent garbage collection and will be removed automatically when the element becomes unreachable.

Parameters

element	The GUI element to register for.
resolver	The resolver to register.

void registerFeatureResolver2(String clazz, FeatureResolver2 resolver)

Register a feature resolver for a GUI element class.

Parameters

clazz	The name of the class to register for.
resolver	The resolver to register.

void registerIdResolver(IdResolver resolver)

Register a generic or global ID resolver.

Parameters

resolver	The resolver to register.

void registerIdResolver(Object element, IdResolver resolver)

Register an ID resolver for an individual GUI element. The resolver does not prevent garbage collection and will be removed automatically when the element becomes unreachable.

Parameters

element	The GUI element to register for.
resolver	The resolver to register.

void registerIdResolver(String clazz, IdResolver resolver)

Register an ID resolver for a GUI element class.

Parameters

clazz	The name of the class to register for.
resolver	The resolver to register.

void registerNameResolver2(NameResolver2 resolver)

Register a generic or global name resolver.

Parameters

resolver	The resolver to register.

`void registerNameResolver2(Object element, NameResolver2 resolver)`

Register a name resolver for an individual GUI element. The resolver does not prevent garbage collection and will be removed automatically when the element becomes unreachable.

Parameters

`element`	The GUI element to register for.
`resolver`	The resolver to register.

`void registerNameResolver2(String clazz, NameResolver2 resolver)`

Register a name resolver for a GUI element class.

Parameters

`clazz`	The name of the class to register for.
`resolver`	The resolver to register.

`void registerTreeTableResolver(TreeTableResolver resolver)`

Register a generic or global TreeTable resolver.

Parameters

`resolver`	The resolver to register.

`void registerTreeTableResolver(Object com, TreeTableResolver resolver)`

Register a TreeTable resolver for a specific component. The resolver will be removed automatically if the component becomes invisible.

Parameters

`com`	The component to register for.
`resolver`	The resolver to register.

`void registerTreeTableResolver(String clazz, TreeTableResolver resolver)`

Register a TreeTable resolver for a component class.

Parameters

`clazz`	The name of the class to register for.
`resolver`	The resolver to register.

`void unregisterExtraFeatureResolver(ExtraFeatureResolver resolver)`

Unregister a generic or global extra feature resolver.

Parameters

`resolver`	The resolver to unregister.

`void unregisterExtraFeatureResolver(Object element,`
`ExtraFeatureResolver resolver)`

Unregister an extra feature resolver for an individual GUI element.

Parameters

`element`	The GUI element to unregister for.
`resolver`	The resolver to unregister.

`void unregisterExtraFeatureResolver(String clazz,`
`ExtraFeatureResolver resolver)`

Unregister an extra feature resolver for a GUI element class.

Parameters

`clazz`	The name of the class to unregister for.
`resolver`	The resolver to unregister.

`void unregisterFeatureResolver2(FeatureResolver2 resolver)`

Unregister a generic or global feature resolver.

Parameters

`resolver`	The resolver to unregister.

`void unregisterFeatureResolver2(Object element,`
`FeatureResolver2 resolver)`

Unregister a feature resolver for an individual GUI element.

Parameters

`element`	The GUI element to unregister for.
`resolver`	The resolver to unregister.

`void unregisterFeatureResolver2(String clazz, FeatureResolver2`
`resolver)`

Unregister a feature resolver for a GUI element class.

Parameters

`clazz`	The name of the class to unregister for.
`resolver`	The resolver to unregister.

`void unregisterIdResolver(IdResolver resolver)`

Unregister a generic or global ID resolver.

Parameters

`resolver`	The resolver to unregister.

`void unregisterIdResolver(Object element, IdResolver resolver)`

Unregister an ID resolver for an individual GUI element.

Parameters

`element`	The GUI element to unregister for.
`resolver`	The resolver to unregister.

`void unregisterIdResolver(String clazz, IdResolver resolver)`

Unregister an ID resolver for a GUI element class.

Parameters

`clazz`	The name of the class to unregister for.
`resolver`	The resolver to unregister.

`void unregisterNameResolver2(NameResolver2 resolver)`

Unregister a generic or global name resolver.

Parameters

`resolver`	The resolver to unregister.

`void unregisterNameResolver2(Object element, NameResolver2 resolver)`

Unregister a name resolver for an individual GUI element.

Parameters

`element`	The GUI element to unregister for.
`resolver`	The resolver to unregister.

`void unregisterNameResolver2(String clazz, NameResolver2 resolver)`

Unregister a name resolver for a GUI element class.

Parameters

`clazz`	The name of the class to unregister for.
`resolver`	The resolver to unregister.

`void unregisterResolvers(Object element)`

Unregister all resolvers for a specific GUI element.

Parameters

`element`	The element to unregister for.

`void unregisterResolvers(String clazz)`

Unregister all resolvers for the class of a GUI element, value or renderer.

Parameters

`clazz`	The name of the class to unregister for.

---

void unregisterTreeTableResolver(TreeTableResolver resolver)

Unregister a generic or global TreeTable resolver.

Parameters

resolver	The resolver to unregister.

---

void unregisterTreeTableResolver(Object com, TreeTableResolver resolver)

Unregister a TreeTable resolver for a specific component.

Parameters

com	The component to unregister for.
resolver	The resolver to unregister.

---

void unregisterTreeTableResolver(String clazz, TreeTableResolver resolver)

Unregister a TreeTable resolver for a component class.

Parameters

clazz	The name of the class to unregister for.
resolver	The resolver to unregister.

52.4   Implementing custom item types with the `ItemResolver` interface

3.1+

As described in section 5.9[82], QF-Test is able to dig below the given structure of GUI elements and work with sub-items that are not GUI components themselves, like the cells of a table, nodes in a tree or drawings on a canvas. Such items are implemented via the `ItemResolver` mechanism which can be used to implement your own custom items.

The `ItemResolver` interface is more complex than the simple `NameResolver2` or `FeatureResolver2` described in the previous section and cannot be implemented without solid programming skills and a thorough understanding of the underlying concepts. Also, `ItemResolvers` and `Checkers`, described in the next section, are closely related and need to be implemented together if you want to be able to perform checks for your items.

On the upside the whole mechanism is very powerful and once implemented and registered, your `ItemResolvers` will integrate smoothly with QF-Test so that there is no distinction between standard and custom items, so don't let yourself be deterred.

You can find demo implementations in the directory `qftest-7.1.2/Jython/Lib` under QF-Test's root directory. The respective demo files are `ktable.py` and `gef.py`. Both resolvers are for SWT specific tables, but the concept is the same for all engines.

52.4.1   `ItemResolver` **concepts**

Before you start implementing an `ItemResolver` you need to determine the kinds of items that your GUI element might hold. There might be more than one kind, though the decision is arbitrary. For example, we implemented items for all standard tables of Swing, SWT and JavaFX so that an item can either be a table cell or a complete column. The latter is useful because it makes it possible to implement a check that checks a whole table column at once.

First you need to decide how to represent your items internally. You can use any kind of `Object` because QF-Test doesn't ever examine your internal representation itself, it just passes it to the methods of your `ItemResolver`. What works best depends on the API of the GUI element. If it already comes with its own concept for sub-items it may be best to reuse those classes.

The most important decision to make is how to represent the item to the user of QF-Test. As described in section 5.9[(82)], the user can address an item via a numerical index or a textual one which can also be matched by a regular expression. You need to be able to provide a two-way mapping between an item and its index(es), i.e. you need to be able to answer the following two questions:

- Given an item, what is its numerical and its textual index?

- Given a numerical or textual index, which item matches that index best?

The issues involved in naming sub-items are the same as those for setting component names. Please take a thorough look at section 5.4.2[(59)] and section 5.4.2[(61)] before continuing.

An single numerical or textual index is represented by a `SubItemIndex` object. The current item concept supports addressing an item like a table cell via a primary and a secondary index, but in the future we hope to support indexes to any depth, so instead of using a path for a tree node it could be addressed with a mixed-type index in a form like "tree@Root&1%Name:.*". Therefore the complete index is represented as an array of `SubItemIndex` objects, though currently limited to single or two-element arrays.

Most items have a geometry, i.e. a location and a size. The coordinates for an item are always calculated relative to the true upper left origin of the element, regardless of whether it is scrolled, so they are independent of the current scroll position of the element. For items where geometry is not applicable or cannot be determined, coordinates can be ignored and the methods `getItemLocation` and `getItemSize` should simply return [0,0].

52.4.2 The `ItemResolver` interface

The methods of the interface `de.qfs.apps.qftest.extensions.items.ItemResolver` fall into three categories: Retrieving an item, mapping between an item and its index and retrieving miscellaneous information for - or performing actions on - an item.

`Object getItem(Object element, int x, int y)`
Get a sub-item for a GUI element at a given location. For items where geometry is not applicable or cannot be determined, the coordinates can be ignored and the item determined based on some other means like selection.
Parameters

`element`	The GUI element to get the item for.
`x`	The X coordinate relative to element.
`y`	The Y coordinate relative to element.
Returns	An arbitrary object representing an item or null if there is no item at the given location.

`int getItemCount(Object element, Object item)`
Get the number of items in a GUI element or at the next item level. Though this method is currently unused it should be implemented if possible. In the future it may be used for a 'Fetch count' node similar to Fetch index[744] or Fetch text[740].
Parameters

`element`	The GUI element for which to get the item count.
`item`	Null to get the number items at the top level of the element. An item to get the number of its sub-items.
Returns	The number of items or -1 if there is no further sub-item level.

`Object getItemForIndex(Object element, SubItemIndex[] idx)`
Get an item for a given sub-item index.
At the end of this procedure a call of `setIndexesResolved` might be required in case more than one index is resolved.
Parameters

`element`	The GUI element to get the item for.
`idx`	The sub-item index(es) for the item.
Returns	The item that best matches the given index.
Throws	
`IndexNotFoundException`	If no item matches the given index. Use the constructor `de.qfs.apps.qftest.shared.exceptions.IndexNotFoundException(SubItemIndex)` for this case.

`SubItemIndex[] getItemIndex(Object element, Object item, int type)`

Get the `SubItemIndex(es)` for a sub-item of a GUI element.

Parameters

`element`	The GUI element to which the item belongs.
`item`	The item to get the index for.
`type`	The type of index to get. Possible values are `INTELLIGENT`, `AS_STRING` and `AS_NUMBER`, all defined in the `SubItemIndex` class. Unless only one kind of index is supported, a textual index should be returned for `AS_STRING` and a numerical index for `AS_NUMBER`. If the type is `INTELLIGENT` you are free to return whatever best represents the given item, even a mixed index like a column title and a row index for a table cell.
Returns	An array of `SutItemIndex` objects. Currently only single or two-element arrays are allowed.

`int[] getItemLocation(Object element, Object item)`

Get the location of a sub-item relative to its parent element.

Parameters

`element`	The GUI element to which the item belongs.
`item`	The item whose location to get.
Returns	The item's location as a two-element int array [x,y]. The location returned must always be relative to the upper left corner of the whole element, even if that corner is not currently visible, for example because it has been scrolled outside the visible area. For items without geometry simply return [0,0].

`int[] getItemSize(Object element, Object item)`

Get the size of a sub-item relative to its parent element.

Parameters

`element`	The GUI element to which the item belongs.
`item`	The item whose size to get.
Returns	The item's size as a two-element int array [width,height]. For items without geometry simply return [0,0].

`String getItemValue(Object element, Object item)`

Get the value of a sub-item to be used for a text check.

Parameters

`element`	The GUI element to which the item belongs.
`item`	The item whose value to get.
Returns	A string representing the value of the item, i.e. its contents, label, whatever.

`Boolean repositionMouseEvent(Object element, Object item, int[] pos)`

Change the coordinates for a mouse event on a sub-item of a GUI element to a default location - typically the center - if the coordinates may be safely ignored during replay. This method is called only if the option Record MouseEvents without coordinates where possible [(441)] is active. Whether or not it is safe to override the event coordinates may depend on the original coordinates. For example, QF-Test repositions events on a tree node with positive coordinates pointing inside the node to its center whereas negative coordinates indicate a click on the expansion toggle and are left unchanged.

Parameters

`element`	The GUI element to which the item belongs.
`item`	The target item for the event.
`pos`	A two-element int array of the form [x,y] with the coordinates of the event relative to the item. Its values can be modified in place. You can either set them to a specific coordinate or to [Integer.MAX_VALUE,Integer.MAX_VALUE] to ignore coordinates for this event so that it will later be replayed on the center of the target.
Returns	Boolean.TRUE if the position was modified, Boolean.FALSE if it was left unchanged. Null to signal that this resolver does not handle the element.

Throws

`BadItemException`	If element and item types don't match up (should never happen).

`Boolean scrollItemVisible(Object element, Object item, int x, int y)`

Scroll a GUI element so that an item becomes visible. If possible, the item should be made fully visible. In case the item doesn't fit into the visible region of the element, at least the given position should be shown. In most cases you can simple return null to let QF-Test handle scrolling. Sometimes however, scrolling cannot be implemented generically based on the item's geometry, because some items need special compensation, for example an SWT Table with a visible header. For GUI elements that cannot be scrolled, this method should simply return Boolean.FALSE. If Boolean.TRUE is returned, QF-Test will call the method again after a short interval because SWT sometimes interferes with scrolling. In the ideal case the second call should return FALSE because the position is already OK. After three attempts that return TRUE QF-Test gives up and signals an error.

Parameters

`element`	The GUI element to which the item belongs.
`item`	The item that must be made visible.
`x`	The X-coordinate of the position relative to the item that must always become visible.
`y`	The Y-coordinate of the position relative to the item that must always become visible.
Returns	Boolean.TRUE if the scroll position of the element change, Boolean.FALSE if the position is unchanged scrolled or if the element cannot be scrolled. Null to signal that QF-Test should use its default mechanism and try to scroll the item itself.

`void setIndexesResolved(int num)`

Notify the registry about how many item indexes were resolved during item resolution in `getItemForIndex`. If this method is not called at the end of `getItemForIndex`, the registry assumes one index.

Parameters

`num`	The number of indexes resolved.

52.4.3 The class `SubItemIndex`

As explained in the previous section, a `de.qfs.apps.qftest.shared.data.SubItemIndex` represents a (partial) index for a sub-item of a complex GUI element. This class defines some constants with the following meanings:

STRING
　　　This is a textual index

NUMBER
> This is a numerical index

REGEXP
> This is a regular expression to match a textual index

INTELLIGENT
> When retrieving an index, use whichever type best suits the item

AS_STRING
> Retrieve a textual index

AS_NUMBER
> Retrieve a numerical index

It also provides the following methods:

`SubItemIndex SubItemIndex(String index)`

Create a new `SubItemIndex` of type `STRING`.

Parameters

`index`	The textual index.

`SubItemIndex SubItemIndex(int index)`

Create a new `SubItemIndex` of type `NUMBER`.

Parameters

`index`	The numerical index.

`int asNumber()`

Get the index as a number.

Returns	The numerical index.

Throws

`IndexFormatException`	If the index is not of type `NUMBER` or cannot be parsed as an integer.

`String getIndex()`

Get the index as a String.

Returns	The index converted to a String.

`String getType()`

Get the type of the index.

Returns	The type of the index, one of `STRING`, `NUMBER` or `REGEXP`.

---

boolean matches(String name)

Test whether the index matches a given item name.

Parameters

name	The name to match.
Returns	True if the index is not numerical and matches the given name.

Throws

IndexFormatException	If the index contains a malformed regular expression.

52.4.4 The `ItemRegistry`

Once implemented and instantiated, your `ItemResolver` must be registered with the `ItemRegistry`. The class `de.qfs.apps.qftest.extensions.items.ItemRegistry` has the following interface:

---

static ItemRegistry instance()

There can only ever be one `ItemRegistry` object and this is the method to get hold of this singleton instance.

Returns	The singleton `ItemRegistry` instance.

---

void registerItemNameResolver2(Object element, ItemNameResolver2 resolver)

Register an `ItemNameResolver2` for an individual GUI element.

Parameters

element	The GUI element to register for. The resolver does not prevent garbage collection and will be removed automatically when the element becomes unreachable.
resolver	The `ItemNameResolver2` to register.

---

void registerItemNameResolver2(String clazz, ItemNameResolver2 resolver)

Register an `ItemNameResolver2` for a class of GUI elements.

Parameters

clazz	The class of GUI element to register for.
resolver	The `ItemNameResolver2` to register.

`void registerItemResolver(Object element, ItemResolver resolver)`

Register an `ItemResolver` for an individual GUI element.

Parameters

`element`	The GUI element to register for. The resolver does not prevent garbage collection and will be removed automatically when the element becomes unreachable.
`resolver`	The `ItemResolver` to register.

`void registerItemResolver(String clazz, ItemResolver resolver)`

Register an `ItemResolver` for a class of GUI elements.

Parameters

`clazz`	The class of GUI element to register for.
`resolver`	The `ItemResolver` to register.

`void registerItemValueResolver2(Object element, ItemValueResolver2 resolver)`

Register an `ItemValueResolver2` for an individual GUI element.

Parameters

`element`	The GUI element to register for. The resolver does not prevent garbage collection and will be removed automatically when the element becomes unreachable.
`resolver`	The `ItemValueResolver2` to register.

`void registerItemValueResolver2(String clazz, ItemValueResolver2 resolver)`

Register an `ItemValueResolver2` for a class of GUI elements.

Parameters

`clazz`	The class of GUI element to register for.
`resolver`	The `ItemValueResolver2` to register.

`void unregisterItemNameResolver2(Object element, ItemNameResolver2 resolver)`

Unregister an `ItemNameResolver2` for an individual GUI element.

Parameters

`element`	The GUI element to unregister for.
`resolver`	The `ItemNameResolver2` to unregister.

`void unregisterItemNameResolver2(String clazz,`
`ItemNameResolver2 resolver)`

Unregister an `ItemNameResolver2` for a class of GUI elements.

Parameters

`clazz`	The class of GUI element to unregister for.
`resolver`	The `ItemNameResolver2` to unregister.

`void unregisterItemResolver(Object element, ItemResolver`
`resolver)`

Unregister an `ItemResolver` for an individual GUI element.

Parameters

`element`	The GUI element to unregister for.
`resolver`	The `ItemResolver` to unregister.

`void unregisterItemResolver(String clazz, ItemResolver`
`resolver)`

Unregister an `ItemResolver` for a class of GUI elements.

Parameters

`clazz`	The class of GUI element to unregister for.
`resolver`	The `ItemResolver` to unregister.

`void unregisterItemValueResolver2(Object element,`
`ItemValueResolver2 resolver)`

Unregister an `ItemValueResolver2` for an individual GUI element.

Parameters

`element`	The GUI element to unregister for.
`resolver`	The `ItemValueResolver2` to unregister.

`void unregisterItemValueResolver2(String clazz,`
`ItemValueResolver2 resolver)`

Unregister an `ItemValueResolver2` for a class of GUI elements.

Parameters

`clazz`	The class of GUI element to unregister for.
`resolver`	The `ItemValueResolver2` to unregister.

52.4.5   Default item representations

For the implementation of the `ItemNameResolver2, ItemValueResolver2` and
`Checker` interfaces it is important to know which kind of Object is used for the internal
representation of an item. This internal representation will be passed to the methods

`getItemName`, `getItemValue`, `getCheckData` and `getCheckDataAndItem`.

JavaFX   The following table lists the complex GUI elements and the default internal item representation used by QF-Test standard `ItemResolvers` for JavaFX.

GUI element class	Item type
Accordion	Integer index
ChoiceBox	Integer index
ComboBox	Integer index
ListView	Integer index
TabPane	Integer index
TableView	int array [column,row] with row < 0 to represent a whole column
TableHeaderRow	Integer column index
TextArea	Integer line
TreeView	TreeItem object

Table 52.1: Internal item representations for JavaFX GUI elements

Swing   The following table lists the complex GUI elements and the default internal item representation used by QF-Test standard `ItemResolvers` for Swing.

GUI element class	Item type
JComboBox	Integer index
JList	Integer index
JTabbedPane	Integer index
JTable	int array [column,row] with row < 0 to represent a whole column
JTableHeader	Integer column index
JTextArea	Integer line
JTree	TreePath path

Table 52.2: Internal item representations for Swing GUI elements

SWT   The following table lists the complex GUI elements and the default internal item representation used by QF-Test standard `ItemResolvers` for SWT.

GUI element class	Item type
CCombo	Integer index
Combo	Integer index
CTabFolder	Integer index
List	Integer index
StyledText	Integer line
TabFolder	Integer index
Table	int array [column,row] or just Integer column to represent a whole column
Text	Integer line
Tree	Object array [Integer column,TreeItem row] or just Integer column to represent a whole column

Table 52.3: Internal item representations for SWT GUI elements

The following table lists the complex GUI elements and the default internal item representation used by QF-Test standard `ItemResolvers` for Web.

GUI element class	Item type
SELECT node	OPTION node
TEXTAREA node	Integer line

Table 52.4: Internal item representations for DOM nodes

52.5   Implementing custom checks with the `Checker` interface

Checks are one of QF-Test's most useful features. Test automation would be mostly useless without the ability to verify the results of simulated actions. However, the default set of Checks available in QF-Test is naturally limited to checking the most common attributes of standard components. For special attributes or custom components you can resort to read the value in an SUT script[631] and use the method `rc.checkEqual()` to compare it against the expected value. Such an SUT script is perfectly fine, it performs and integrates well, is flexible and can be modularized by placing it inside a Procedure[586]. It has two major disadvantages however: It cannot be recorded and it is daunting for non-programmers.

With the help of the API described in this section the default set of Checks in QF-Test can be extended. In fact, QF-Test's own new-style checks are implemented exactly this

way. By implementing and registering a `Checker` for a given type of GUI element and possibly item you can create your own checks that can be recorded and replayed just like the standard ones.

To make this as simple as possible, QF-Test handles everything from showing the check in the check popup menu, fetching the check data, recording the respective Check node to store that data, sending the data back to the SUT upon replay, fetching the then current check data, comparing it to the expected value and reporting success or mismatch. All that is left for you to do is tell QF-Test which checks your `Checker` implements and for each of these provide the check data on request.

Illustrative examples are provided at the end of the chapter and in the test suite `carconfigSwing_en.qft`, located in the directory `demo/carconfigSwing` in your QF-Test installation.

52.5.1   The `Checker` interface

The interface `de.qfs.apps.qftest.extensions.checks.Checker` must be implemented in order to add custom checks for your application. The associated helper classes and interfaces are documented in the subsequent sections.

CheckData getCheckData(Object element, Object item, CheckType type)
Get the check data for the current state of the GUI element or item. This method is used during replay.
Parameters

element	The GUI element for which to get the check data.
item	An optional item within the element to check. Its type depends on the GUI element and the registered `ItemResolvers` as described in section 52.4.5[1071].
type	The type of check to perform.
Returns	The check data for the current state of the GUI element itself, in case item is null, or the given item within the element. The kind of check to perform is specified via the type parameter, which should normally be one of those formerly returned by `getSupportedCheckTypes`. If you cannot perform the requested check for the given type and target, return null.

`Pair getCheckDataAndItem(Object element, Object item, CheckType type)`

Get the check data for the current state of the GUI element or item and also the item that the check actually applies to. This method is used during recording, where QF-Test not only needs to know, which data to record, but also whether to record the check for the GUI element as a whole or for an item. To implement this method without duplicating any code, call your own `getCheckData` method to retrieve the check data.

Parameters

`element`	The GUI element for which to get the check data.
`item`	An optional item within the element to check. Its type depends on the GUI element and the registered `ItemResolvers` as described in section 52.4.5[(1071)].
`type`	The type of check to perform.
Returns	A `Pair` with the check data for the current state of the GUI element, and the item this check should be performed on, which may be null.

`CheckType[] getSupportedCheckTypes(Object element, Object item)`

Get the types of checks supported for the given GUI element and optional item.

Parameters

`element`	The GUI element for which to get the available checks.
`item`	An optional item within the element to check. Its type depends on the GUI element and the registered `ItemResolvers` as described in section 52.4.5[(1071)].
Returns	An array of `CheckTypes` supported by your checker. The first element is the default for recording a check with a left-click. If the item is null, return only those kinds of checks that can be applied to the whole GUI element. Otherwise it is best to provide all available checks because even though the user may have right-clicked on an item, he may still want to record a check on the whole GUI element.

52.5.2 The class `Pair`

The class `de.qfs.lib.util.Pair` for the return value of `getCheckDataAndItem` is a simple utility class that often comes in handy for grouping two values. You'll only need its constructor, but of course you can also read its values:

`Pair Pair(Object first, Object second)`

Create a new `Pair`.

Parameters

`first`	The first object. May be null.
`second`	The second object. May be null.

`Object getFirst()`

Get the first object of the `Pair`.

Returns	The first object.

`Object getSecond()`

Get the second object of the `Pair`.

Returns	The second object.

52.5.3   The `CheckType` interface and its implementation `DefaultCheckType`

A   `de.qfs.apps.qftest.extensions.checks.CheckType`   encapsulates
information for a specific kind of check. It combines a `CheckDataType` with an
identifier and provides a user-friendly representation of the check for the check popup
menu.   Unless you need to provide multi-lingual representations of the check
you should never implement this interface yourself, but simply instantiate a
`de.qfs.apps.qftest.extensions.checks.DefaultCheckType` instead:

`DefaultCheckType(String identifier, CheckDataType dataType,`
`String description)`

Create a new `DefaultCheckType`.

Parameters

`identifier`	The identifier for the check. The QF-Test standard checks all use lower case identifiers. To prevent conflicts, simply start your custom check identifiers with a capital letter.
`dataType`	The `CheckDataType` required for your check data.
`description`	The description to show in the check popup menu for this check.

For completeness sake, following are the methods of the `CheckType` interface:

`CheckDataType getDataType()`

Get the `CheckDataType` for the check type.

Returns	The data type for the check type.

`String getDescription()`

Get the localized description to show for this check type in the check popup menu.

| **Returns** | The description for the check type. |

`String getIdentifier()`

Get the identifier for the check type.

| **Returns** | The identifier for the check type. |

52.5.4 The class `CheckDataType`

The class `de.qfs.apps.qftest.extensions.checks.CheckDataType` is similar to an `Enum`. It defines a number of constant `CheckDataType` instances that simply serve to identify the kind of data that a check operates on. Each constant corresponds to one or more of the available Check nodes of QF-Test.

Besides serving as a constant identifier, a `CheckDataType` has no public attributes or methods and you cannot add any new `CheckDataTypes`. If you want to implement a check of a kind that does not fit the existing data types you'll need to convert your data so that it does, for example by using a string representation. The following `CheckDataType` constants are defined:

STRING
  A single string. Used by the Check text[708] node.

STRING_LIST
  A list of string items, like the cells in a table column. Used by the Check items[719] node.

SELECTABLE_STRING_LIST
  A list of selectable string items, like the elements of a list. Used by the Check selectable items[724] node.

BOOLEAN
  A boolean state, either true of false. Used by the Boolean check[713] node.

GEOMETRY
  A set of four integer values for X and Y coordinates, width and height. Not all have to be defined. Used by the Check geometry[734] node.

IMAGE
  An image of a whole component or item or a sub-region thereof. Used by the Check image[729] node.

52.5.5   The class `CheckData` and its subclasses

The class `de.qfs.apps.qftest.shared.data.check.CheckData` and its subclasses, all from the same package, complete the `Checker` API. A `CheckData` encapsulates the actual data for a check, must be returned from `Checker.getCheckData()` and is used to exchange this check data between the SUT and QF-Test. There is one concrete `CheckData` subclass corresponding to each `CheckDataType`. You'll only ever need to use their constructors, so that's what we'll list here. Only two of these classes are publicly available so far:

BooleanCheckData BooleanCheckData(String identifier, boolean value)

Create a new `BooleanCheckData`.

Parameters

identifier	The identifier for the check type. Should normally match the identifier of the `type` argument passed to `Checker.getCheckData`.
value	The actual value for the check, a boolean state.

GeometryCheckData GeometryCheckData(String identifier, int x, int y, int width, int height)

Create a new `GeometryCheckData`.

Parameters

identifier	The identifier for the check type. Should normally match the identifier of the `type` argument passed to `Checker.getCheckData`.
x	The x-coordinate for the check.
y	The y-coordinate for the check.
width	The width for the check.
height	The height for the check.

`ImageCheckData ImageCheckData(String identifier, ImageRep image, int xOffset, int yOffset, int subX, int subY, int subWidth, int subHeight)`

Create a new `ImageCheckData`.

Parameters

`identifier`	The identifier for the check type. Should normally match the identifier of the `type` argument passed to `Checker.getCheckData`.
`image`	The image for the check. See section 52.10[(1096)].
`xOffset`	An optional x-offset.
`yOffset`	An optional y-offset.
`subX`	The X-coordinate of an optional check region.
`subY`	The Y-coordinate of an optional check region.
`subWidth`	The Width of an optional check region.
`subHeight`	The Height of an optional check region.

`SelectableItemsCheckData SelectableItemsCheckData(String identifier, Object[][] values)`

Create a new `SelectableItemsCheckData`.

Parameters

`identifier`	The identifier for the check type. Should normally match the identifier of the `type` argument passed to `Checker.getCheckData`.
`values`	The actual value for the check, an array of arrays with a String, a Boolean for the regexp flag and a Boolean for the selected flag.

`StringCheckData StringCheckData(String identifier, String value)`

Create a new `StringCheckData`.

Parameters

`identifier`	The identifier for the check type. Should normally match the identifier of the `type` argument passed to `Checker.getCheckData`.
`value`	The actual value for the check, a String.

StringItemsCheckData StringItemsCheckData(String identifier, String[] values)

Create a new `StringItemsCheckData`.

Parameters

identifier	The identifier for the check type. Should normally match the identifier of the `type` argument passed to `Checker.getCheckData`.
values	The actual value for the check, an array of Strings.

Furthermore you can define an optional algorithm for an ImageCheckData.

---

void setAlgorithm(String algorithm)

Sets an algorithm. A detailed description can be found in <u>Details about the algorithm for image comparison</u>[(1167)].

52.5.6 The `CheckerRegistry`

Once implemented and instantiated, your `Checker` must be registered with the `CheckerRegistry`. The class `de.qfs.apps.qftest.extensions.checks.CheckerRegistry` has the following interface:

---

static CheckerRegistry instance()

There can only ever be one `CheckerRegistry` object and this is the method to get hold of this singleton instance.

Returns	The singleton `CheckerRegistry` instance.

---

void registerChecker(Object element, Checker checker)

Register a `Checker` for an individual GUI element.

Parameters

element	The GUI element to register for. The checker does not prevent garbage collection and will be removed automatically when the element becomes unreachable.
checker	The `Checker` to register.

---

void registerChecker(String clazz, Checker checker)

Register a `Checker` for a class of GUI elements.

Parameters

clazz	The class of GUI element to register for.
checker	The `Checker` to register.

---

void unregisterChecker(Object element, Checker checker)

Unregister a `Checker` for an individual GUI element.

Parameters

element	The GUI element to unregister for.
checker	The `Checker` to unregister.

---

void unregisterChecker(String clazz, Checker checker)

Unregister a `Checker` for a class of GUI elements.

Parameters

clazz	The class of GUI element to unregister for.
checker	The `Checker` to unregister.

52.5.7   Custom checker example

The following Jython SUT script illustrates how to put everything together. Let's say you have a Java Swing application and want to check all labels which reside in a panel at once. To this end, your custom checker needs to iterate over all components contained in the panel and its children respectively, identify the labels and generate a list of all their text contents.   In QF-Test notation, this means you need to create a `CheckDataType.STRING_LIST` check type and return the data in an `StringItemsCheckData` object:

```
from de.qfs.apps.qftest.extensions import ResolverRegistry
from de.qfs.apps.qftest.extensions.checks import CheckerRegistry, \
    Checker, DefaultCheckType, CheckDataType
from de.qfs.apps.qftest.extensions.items import ItemRegistry
from de.qfs.apps.qftest.shared.data.check import StringItemsCheckData
from de.qfs.lib.util import Pair
from java.lang import String
import jarray
componentClass = "javax.swing.JPanel"
allLabelsCheckType = DefaultCheckType("AllLabels",
    CheckDataType.STRING_LIST,
    "All labels in the panel")
class AllLabelsChecker(Checker):
    def __init__(self):
        pass
    def getSupportedCheckTypes(self, com, item):
        return jarray.array([allLabelsCheckType], DefaultCheckType)
    def getCheckData(self, com, item, checkType):
        if allLabelsCheckType.getIdentifier() == checkType.getIdentifier():
            labels = self._findLabels(com)
            labels = map(lambda l: l.getText(), labels)
            values = jarray.array(labels, String)
            return StringItemsCheckData(checkType.getIdentifier(), values)
        return None
    def getCheckDataAndItem(self, com, item, checkType):
        data = self.getCheckData(com, item, checkType)
        if data is None:
            return None
        return Pair(data, None)
    def _findLabels(self, com, labels=None):
        if labels is None:
            labels = []
        if ResolverRegistry.instance().isInstance(com, "javax.swing.JLabel"):
            labels.append(com)
        for c in com.getComponents():
            self._findLabels(c, labels)
        return labels
def unregister():
    try:
        CheckerRegistry.instance().unregisterChecker(
            componentClass, allLabelsChecker)
    except:
        pass
def register():
    unregister()
    global allLabelsChecker
    allLabelsChecker = AllLabelsChecker()
    CheckerRegistry.instance().registerChecker(
        componentClass, allLabelsChecker)
register()
```

Example 52.44: Check all labels in a panel

After running that script once, you'll find a new entry "All labels in the panel" among the entries in the check type menu as soon as you right click on a `JPanel` component while being in recording mode (cf. section 4.3⁽³⁸⁾). If you want to use the `allLabelsChecker` all over your client application, you can put the above SUT script behind your Wait for client to connect node in the Setup sequence. Otherwise, you may register the checker only when it is actually needed as shown above and remove it afterwards by means of another SUT script:

```
from de.qfs.apps.qftest.extensions.checks import CheckerRegistry
global allLabelsChecker
def unregister():
    try:
        CheckerRegistry.instance().unregisterChecker(
            "javax.swing.JPanel", allLabelsChecker)
    except:
        pass
unregister()
```

Example 52.45: Remove the label checker

52.6   Working with the Eclipse Graphical Editing Framework (GEF)

_{3.2+}

The Graphical Editing Framework (GEF) is a set of Eclipse plugins for creating editors that support visual editing of arbitrary models. This framework is very popular and QF-Test has supported recording and playback of GEF items for a long time (since about version 2.2). It is also a good example for the power of the `ItemResolver` concept (see section 52.4⁽¹⁰⁶²⁾), because the `gef` Jython module contains an implementation of just that interface.

The `gef` module can deal with GEF editors at a generic level and even support several editors at once. Though reasonable item names are provided out of the box also for GMF applications, there are limits to what can be determined automatically. Depending on the underlying model classes, there might still remain some work for you: Implementing custom resolvers to provide useful names and values for your items.

52.6.1   Recording GEF items

The actual GEF component is the `FigureCanvas`. This control displays `Figures` which represent `EditParts`. When recording a mouse click on such an element, QF-Test does not register a pure Mouse event node for the canvas component with the cor-

responding (x,y) position but tries to recognize the object under the mouse cursor. For example, the recorded QF-Test component ID may look like

```
 canvas@/Diagram/My ship/ShipLargeCargo (wine)
canvas@Connection-2
canvas@/Diagram/Rectangle 16329001
```

where "canvas" is the QF-Test ID of the `FigureCanvas` component, followed by the item index of the recognized `EditPart` (see section 5.9[82]). `EditParts` reside in a tree like hierarchy which is reflected in the index by a path separator '/'. The names of the individual items are generated as follows:

- The item name is getModel().toString() unless it contains a hash value (e.g. NodeImpl@2d862).

- QF-Test tries to extract a name for the item from the model ("My ship" in the above examples).

- The class name along with a description gets recorded, e.g. "ShipLargeCargo (wine)".

- If there's no description, an index is appended to the class name when there's more than one item of that class, e.g. "Connection-2" for the third connection.

- The root `EditPart` always reads "Diagram".

As one can imagine, those generated item names may not always be useful. For example, items might be deleted so that the recorded index is not longer valid. Or the generated item name is unstable as "Rectangle 16329001" in the GEF Shapes example: The number is random and when restarting the application a different one will be created. Three options exist to overcome the problem:

- Instead of working with a textual index, you can try to go with a numerical one. To this end, open the recording options and set the 'Sub-Item format' to 'Number' (see section 39.2.4[453]). This is probably not satisfying because a numerical index like `/0/1` tells nothing about an item.

- Get in touch with your developers and convince them to provide a useful implementation of the `toString()` method of the item's model. It would make live easy for you, but only if the developers are cooperative.

- Write an `ItemNameResolver2`. This is the tough course but unfortunately the most likely scenario. It is covered in the next section.

52.6.2   Implementing a GEF ItemNameResolver2

As stated in section 52.1[(1016)], an `ItemNameResolver2` is the hook to change or provide names for items. To get started, insert a new Jython SUT script[(631)] in the Extras[(547)] node with the following code:

```
def getItemName(canvas, item, name):
    print "name: %s" %name
    print "item: %s" %(item.__class__)
    model = item.getModel()
    print "model: %s" %(model.__class__)
resolvers.addItemNameResolver2("myGefItemNames", getItemName,
    "org.eclipse.draw2d.FigureCanvas")
```

Example 52.46: Get started with a GEF ItemNameResolver2

To ease the installation of the resolver we use the `resolvers` module described in section 52.1[(1016)]. The resolver gets registered for the `FigureCanvas` class where the items reside. The default item name provided by QF-Test is supplied as the last argument to our function `getItemName()`. Now run the script, press the record button and then simply move the mouse over your figures on the canvas - supposing you have created some of them previously. Note that this first resolver implementation does nothing but print out out some information into the terminal, something like

```
 name:  Rectangle 16329001
item:  org.eclipse.gef.examples.shapes.parts.ShapeEditPart
model:  org.eclipse.gef.examples.shapes.model.RectangularShape
```

The question is now: Does the model of the GEF `EditPart` provide any property that might be used as name for the item? The answer in the case of the GEF Shapes example is "No", and hopefully you are in a better situation with your application. To find out insert a line

```
 print dir(model)
```

in the `getItemName()` function and run the script again. Now you will also see the methods of the model when moving the mouse over the items in record mode. With a bit of luck you will find methods like `getId()` or `getLabel()` and can create a resolver like this:

```
def getItemName(canvas, item, name):
    model = item.getModel()
    return model.getId()
resolvers.addItemNameResolver2("myGefItemNames", getItemName,
    "org.eclipse.draw2d.FigureCanvas")
```

Example 52.47: A simple ItemNameResolver2

Let's go back to the GEF Shapes example where we don't have such useful methods. Only geometry information is available for the shapes and that is not really helpful. At least we can distinguish between rectangles and ellipses. To make the item names unique we simply add a child index as shown in the following resolver:

```
def getItemName(canvas, item, name):
    name = None
    shapes = "org.eclipse.gef.examples.shapes"
    diagrammEditPart = shapes + ".parts.DiagramEditPart"
    shapeEditPart = shapes + ".parts.ShapeEditPart"
    connectionEditPart = shapes + ".parts.ConnectionEditPart"
    ellipticalShape = shapes + ".model.EllipticalShape"
    rectangularShape = shapes + ".model.RectangularShape"
    if qf.isInstance(item, shapeEditPart):
        siblings = item.getParent().getChildren()
        for i in range(len(siblings)):
            if (item == siblings[i]):
                if qf.isInstance(item.getModel(), ellipticalShape):
                    name = "Ellipse " + str(i)
                elif qf.isInstance(item.getModel(),
                                            rectangularShape):
                    name = "Rectangle " + str(i)
    elif qf.isInstance(item, connectionEditPart):
        source = item.getSource()
        target = item.getTarget()
        sourceName = getItemName(canvas, source, str(source.getModel()))
        targetName = getItemName(canvas, target, str(target.getModel()))
        name = "Connection " + sourceName + " " + targetName
    elif qf.isInstance(item, diagrammEditPart):
        name = "Diagram"
    return name
resolvers.addItemNameResolver2("shapesItemNames", getItemName,
    "org.eclipse.draw2d.FigureCanvas")
```

Example 52.48: An ItemNameResolver2 for GEF Shapes

With this resolver in place, the item index for a rectangle becomes

```
/Diagram/Rectangle 1
```

where the trailing number is the child index of the item. The above implementation also provides names for the connections by calling `getItemName()` recursively for the source and the target item of the connection. Checking the types with `qf.isInstance()` (see section 48.6[(937)]) will save you the need to import the GEF classes, something that is not trivial.

Once your resolver is working fine you should move the script into your Setup[(554)] sequence right behind the Wait for client to connect[(670)] node. This way the resolver will be registered automatically when the SUT starts.

52.6.3 Implementing a GEF ItemValueResolver2

Usually a GEF editor consists of two parts. Having focused so far on the canvas where you draw the figures, we now take a look at the palette where you select the kind of figure to draw (e.g. 'Rectangle', 'Ellipse' or 'Connection'). Its entries look like tool buttons but actually the palette is a `FigureCanvas` too. You will be glad to know that this one works out of the box, that is without implementing an `ItemNameResolver2`. When you click for example on the 'Rectangle' button, QF-Test recognizes a

```
        /Palette Root/Palette Container (Shapes)/Palette Entry
(Rectangle)
```

item. What will happen when you record a check (cf. section 4.3[38]) for the 'Object value' for this button? You may expect to get the button text 'Rectangle' but in fact the value of this item is

```
 Palette Entry (Rectangle)
```

The reason is that by default name and value of an item are the same. To alter this behavior and provide customized values you need to implement an `ItemValueResolver2`. This interface is very similar to the `ItemNameResolver2` above. For the palette we can code the following one:

```
def getItemValue(canvas, item, value):
    value = None
    paletteEditPart = \
        "org.eclipse.gef.ui.palette.editparts.PaletteEditPart"
    if qf.isInstance(item, paletteEditPart):
        value = item.getModel().getLabel()
    return value
resolvers.addItemValueResolver2("shapesItemValues", getItemValue,
    "org.eclipse.draw2d.FigureCanvas")
```

Example 52.49: An ItemValueResolver2 for the GEF Shapes palette

The method `getLabel()` returns the text as displayed in the palette.

52.7 Test run listeners

3.1+

Once registered with the current run context via `rc.addTestRunListener`, an implementation of the `TestRunListener` interface will get notified whenever test execution enters or exits a node and when a problem occurs. An illustrative example is provided in the test suite `TestRunListener.qft`, located in the directory `demo/runlistener` in your QF-Test installation. Best deactivate the debugger before running the whole test suite.

Note A variant of the `TestRunListener` interface called `DaemonTestRunLister` can be used to monitor a test run remotely via the daemon API. It is described in section 53.2.5[(1153)].

The run listener API consists of the following classes:

52.7.1 The `TestRunListener` interface

The interface `de.qfs.apps.qftest.extensions.qftest.TestRunListener` has to be implemented and registered with a run context via `rc.addTestRunListener()`.

Note To implement the interface you can also derive from the class `de.qfs.apps.qftest.extensions.qftest.AbstractTestRunListener` which provides empty implementations of all methods so you only need to implement those methods you are interested in.

---

void nodeEntered(TestRunEvent event)

Notify the listener that a node is being entered.

Parameters

event	The event containing the details.

---

void nodeExited(TestRunEvent event)

Notify the listener that a node was just exited.

Parameters

event	The event containing the details.

---

void problemOccurred(TestRunEvent event)

Notify the listener that a problem occurred.

Parameters

event	The event containing the details.

---

void runStarted(TestRunEvent event)

Notify the listener that a test run was started.

Parameters

event	The event containing the details, irrelevant in this case.

---

void runStopped(TestRunEvent event)

Notify the listener that a test run was stopped.

Parameters

event	The event containing the details, irrelevant in this case.

52.7.2 The class `TestRunEvent`

The class `de.qfs.apps.qftest.extensions.qftest.TestRunEvent`
holds information about the currently executed nodes and the current error
state. It defines the following constants for execution states and error levels:
`STATE_NOT_IMPLEMENTED`, `STATE_SKIPPED`, `STATE_OK`, `STATE_WARNING`,
`STATE_ERROR` and `STATE_EXCEPTION`. The first two states apply only to Test set[(525)]
and Test case[(518)] nodes.

In the `runStopped` method you can also check whether the test run has been inter-
rupted or completed normally. Therefore the constants `STATE_RUN_INTERRUPTED`
and `STATE_RUN_TERMINATED` are defined.

`JsonObject asJsonValue()`
Serializes the event object as `JsonObject`. This can be used to simplify the in-
teraction of TestRunListeners with Json-based tools like web services or databases.
The API of the Json library embedded in QF-Test is documented in the file
`doc/javadoc/json.zip` in your QF-Test installation

Returns	The object as `JsonObject`.

`int getErrors()`
Get the error count for the exited node.

Returns	The total error count for the node just exited. Only avail-able for `nodeExited`.

`int getExceptions()`
Get the exception count for the exited node.

Returns	The total exception count for the node just exited. Only available for `nodeExited`.

`int getLocalState()`
Get the execution state for the current node.

Returns	The local execution state for the current node, the high-est error level of this node and its children regardless of whether this state propagates to the top. Available only for `nodeExited` and `problemOccurred`. For Test set and Test case nodes the local state can also be one of `STATE_SKIPPED` or `STATE_NOT_IMPLEMENTED`.

`String getMessage()`
Get the current error message.

Returns	The message for the current warning, error or exception. Only available for `problemOccurred`.

TestSuiteNode getNode()

Get the current node

| **Returns** | The current node or null for `runStarted` and `runStopped`. |

TestSuiteNode[] getPath()

Get the whole tree path for the current node.

| **Returns** | The path for the current node as seen from the run log, equivalent to the execution stack, or null for `runStarted` and `runStopped`. The last node in the array is the current node. |

---

int getState()

Get the overall execution state for the current node.

| **Returns** | The overall propagating state for the current node, the highest error level of this node and its children, possibly limited by Maximum error level[(537)] attributes. Available only for `nodeExited` and `problemOccurred`. |

---

int getWarnings()

Get the warning count for the exited node.

| **Returns** | The total warning count for the node just exited. Only available for `nodeExited`. |

52.7.3 The class **TestSuiteNode**

A `de.qfs.apps.qftest.extensions.qftest.TestSuiteNode` is a representation of a QF-Test node that is currently being executed, including information about the type of node, its name, comment, etc.

JsonObject asJsonValue()

Serializes the node as `JsonObject`.

| **Returns** | The object as `JsonObject`. |

String getComment()

Get the comment of the node.

| **Returns** | Get the expanded comment of the node. |

String getComponentId()

Get the QF-Test ID of the component of the node, if available.

| **Returns** | Get the expanded QF-Test ID of the component of the node. |

String getExpandedTreeName()

Get the expanded tree name of the node.

Returns	The name of the node as displayed in the test suite tree wth expanded variables.

String getId()

Get the ID of the node.

Returns	The ID of the node.

String getName()

Get the name of the node.

Returns	The name of the node or null if that type of node does not have a 'Name' attribute.

String getReportName()

Get the report name of the node.

Returns	The expanded report name for Test set[(525)] and Test case[(518)] nodes. The normal name for other nodes or null if undefined.

String getSuite()

Get the test suite to which the node belongs.

Returns	The full path of the test suite to which the node belongs.

String getTreeName()

Get the tree name of the node.

Returns	The name of the node as displayed in the test suite tree.

String getType()

Get the type of the node.

Returns	The type of node, the last part of the name of the class implementing the node's behavior.

String getVerboseReportName()

Get the expanded report name or expanded name of the node.

Returns	The expanded report name for Test set[(525)] and Test case[(518)] nodes. As fallback, the expanded name of the node is returned.

52.8 ResetListener

4.0.3+

During test development you sometimes want to stop all connected clients and reset all dependencies as well as delete global QF-Test variables in order to establish a well

defined starting point for the next test execution.

For that purpose QF-Test offers Run→Reset everything in its main menu. To accommodate their particular needs test developers can additionally implement a ResetListener which allows to

- keep (certain) clients alive

- restore global QF-Test variables

- perform custom operations, for example deleting global Jython variables

To manage ResetListeners the QF-Test runcontext[917] provides the methods `addResetListener()`, `isResetListenerRegistered()` and `removeResetListener()`.

The ResetListener interface itself has two methods:

---

void afterReset()

This method is called when the reset was executed.

Set<String> beforeReset()

This method is called before dependencies are reset, global QF-Test variables deleted and clients terminated when invoking Run→Reset everything. Implement this method when you want to prevent certain clients from being terminated.

Returns A `java.util.Set<String>` object containing the names of all clients that should stay alive.

The following example shows the implementation and registration of a ResetListener. It restores the global QF-Test variable `client` and prevents from closing the respective SUT.

```
from java.util import HashSet
from de.qfs.apps.qftest.extensions.qftest import ResetListener
from de.qfs.apps.qftest.shared.exceptions import UnboundVariableException
class RL(ResetListener):
    def beforeReset(self):
        try:
            self.client = rc.lookup("client")
            h = HashSet()
            h.add(self.client)
            return h
        except UnboundVariableException:
            self.client = None
    def afterReset(self):
        if self.client != None:
            rc.setGlobal("client", self.client)
global resetListener
try:
    rc.removeResetListener(resetListener)
except:
    pass
resetListener = RL()
rc.addResetListener(resetListener)
```

Example 52.50: Example of a ResetListener implementation

52.9   DOM processors

When creating reports from a run log, or package documentation from a test suite, QF-Test operates in a two-step process. The first step creates an XML document which is transformed to an HTML document in the second step. Both transformations are done using XSLT stylesheets.

Note   The term DOM (for *Document Object Model*) also applies to XML documents, not only to HTML web pages. This section is all about XML and XSLT and not about the DOM of a web SUT.

However, XSLT stylesheets are not very useful when it comes to parsing plain text. The Comment fields of Test set, Test case, Test step, Package or Procedure nodes often contain some internal structure that XSLT cannot make use of. Additionally, the internal structures employed by users may vary, depending on the conventions used. A typical example is the use of JavaDoc tags to describe parameters of Procedure nodes. Here's an example Comment for the Procedure qfs.swing.menu.select from our standard library after the first step of the transformation:

```
<comment>Select an item from a menu.
For example: for the File -> Open action, the QF-Test component ID
 "File" is the menu, and the QF-Test component ID "Open" is the item.
@param  client  The name of the SUT client.
@param  menu    The QF-Test ID of the menu.
@param  item    The QF-Test ID of the menu item.</comment>
```

Example 52.51: Example Comment after first step transformation

It is very difficult to make use of the `@param` tags with XSLT alone. This is where DOM processors enter the scene. Between the first and second transformation, QF-Test can optionally run an additional transformation directly on the DOM tree of the XML document generated by the first step. During that extra transformation, QF-Test traverses the DOM tree, calling the registered DOM processors for each node to give them a chance to manipulate the DOM.

Note For JDK 1.4 the XML Document Object Model (DOM) is part of the standard API. For earlier JDK versions it is provided by XML parser xerces (from the Apache project) which QF-Test includes. The API documentation for the DOM is available at http://download.oracle.com/javase/1.5.0/docs/api/org/w3c/dom/package-summary.html.

52.9.1   The `DOMProcessor` interface

The interface that must be implemented is `de.qfs.apps.qftest.extensions.DOMProcessor`. It is quite trivial:

`Element process(Element node)`

Process one element node.

| **Returns** | An element node or null. If null is returned, the child nodes of the node are processed normally. Otherwise, the child nodes are not processed. If a node other than the original node is returned, the original node is replaced with the return value. |

In the `process` method, the processor is free to do whatever it likes, as long as it constrains itself to the node passed in and its sub-nodes. The node can be replaced simply by returning some different element node.

Note To remove an element node from the DOM, the `DOMProcessor` must be registered on an ancestor of the node, its parent node, for example. The current node may not be removed from the DOM in the `process` method.

QF-Test provides two example implementations of DOM processors. The `ParagraphProcessor` is available in the `misc` directory for illustration. It is used internally to break comments which contain empty lines into paragraphs.

Also to be found in the `misc` directory is the `DocTagProcessor` which is used to transform JavaDoc tags like `@param` or `@author` to an XML DOM sub-tree. After processing, the above example would look as follows:

```
<comment>Select an item from a menu.
For example: for the File -> Open action, the QF-Test component ID
 "File" is the menu, and the QF-Test component ID "Open" is the item.</comment>
<param name="client">The name of the SUT client.</param>
<param name="menu">The QF-Test component ID of the menu.</param>
<param name="item">The QF-Test component ID of the menu item.</param>
```

Example 52.52: Example comment after DOM processing

Transforming the above into useful HTML during the second stage transformation is now straightforward.

52.9.2 The `DOMProcessorRegistry`

Before a DOM processor can be used, it must be registered for the kind of node(s) it applies to. This is done through the `DOMProcessorRegistry`.

There is one `DOMProcessorRegistry` instance object per kind of transformation, each identified by a string. Currently these identifiers are ″`report`″ for report generation and ″`testdoc`″ and ″`pkgdoc`″ for test set and package documentation plus their variants for transforming the respective summary documents named ″`report-summary`″, ″`testdoc-summary`″ and ″`pkgdoc-summary`″. To get hold of a registry instance, use the static `instance` method:

DOMProcessorRegistry instance(String identifier)

Get hold of a registry instance.

Parameters

identifier   The identifier for the kind of transformation.

The rest of the methods consist of the typical set of register/unregister variants:

---

 void registerDOMProcessor(DOMProcessor processor)

Register a generic DOM processor that will be called for all kinds of nodes.

Parameters

processor   The processor to register.

---

void registerDOMProcessor(String node, DOMProcessor processor)

Register a DOM processor for a specific kind of node.

Parameters

name	The type of the node.
processor	The processor to register.

void unregisterDOMProcessor(DOMProcessor processor)

Unregister a generic DOM processor.

Parameters

processor	The processor to unregister.

void unregisterDOMProcessor(String node, DOMProcessor processor)

Unregister a DOM processor for a specific kind of node.

Parameters

name	The type of the node.
processor	The processor to unregister.

void unregisterDOMProcessors()

Unregister all DOM processors.

52.9.3 Error handling

Exceptions raised during DOM processing by the `process` method of a `DOMProcessor` are caught and duly reported, the transformation is stopped in that case.

52.10 Image API extensions

3.0+

The Image API of QF-Test makes use of a technology independent class for image representation `ImageRep` and it offers an interface to implement an own image compare algorithm.

52.10.1 The `ImageRep` class

The class `de.qfs.apps.qftest.shared.extensions.image.ImageRep` is the wrapper class for technology independent images.

The class holds the image representation either as ARGB data, which is an int array

or as RGB data, which is a byte array. Besides the image data, you can also specify a name, the width and the height for the `ImageRep` object.

The `ImageRep` class also contains an `equals` method for comparisons. If you want to implement your own image comparison algorithm, you have to implement your own `ImageComparator`. This implementation has to be registered at the `ImageRep` object. See for further information.

ImageRep ImageRep()

Constructor for the `ImageRep` class.

ImageRep ImageRep(String name, byte[] rgb, boolean png, int width, int height)

Constructor method for the `ImageRep` class.

Parameters

name	The name of the `ImageRep` object.
rgb	A byte array containing RGB data of the image.
png	Whether the image is already png encoded.
width	The width of the image.
height	The height of the image.

ImageRep ImageRep(String name, int[] argb, boolean png, int width, int height)

Constructor method for the `ImageRep` class.

Parameters

name	The name of the `ImageRep` object.
argb	An int array containing the ARGB data of the image.
png	Whether the image is already png encoded.
width	The width of the image.
height	The height of the image.

---

void crop(int x, int y, int width, int height)

Crop the image to a specified sub-region.

Parameters

x	The X coordinate of the left upper corner of the sub-region.
y	The Y coordinate of the left upper corner of the sub-region.
width	The width of the sub-region.
height	The height of the sub-region.

`ImageRepDrawer draw()`
Creates an ImageRepDrawer object for this image. This image allows drawing lines, rectangles and further figures on the image.

`ImageRepDrawer draw(Object obj)`
Creates an ImageRepDrawer object for this image.

Parameters

`obj`	A lambda object. The lambda object gets a java.awt.Graphics2D object as input which it can use to draw on the image.

`boolean equals(ImageRep compare)`
Return, whether the current object is equal to a given object. It uses the `equals` method of the current `ImageComparator` implementation.

Parameters

`compare`	The `ImageRep` object to compare with.
Returns	True, if images are equal, otherwise false.

`int[] getARGB()`
Get the ARGB data. If no ARGB data is set, the RGB data will be translated into the ARGB data.

Returns	The current ARGB data.

`ImageComparator getComparator()`
Get the current `ImageComparator` implementation.

Returns	The current `ImageComparator` implementation.

`int getHeight()`
Get the height.

Returns	The current height.

`String getName()`
Get the name.

Returns	The current name.

`int getPixel(int x, int y)`
Get an ARGB pixel value.

Parameters

`x`	The X coordinate of the pixel.
`y`	The Y coordinate of the pixel.
Returns	The pixel value.

---

byte[] getPng()
Get the RGB data. If no RGB data is set, the ARGB data will be translated into the RGB data.

Returns	The current RGB data.

---

int getWidth()

Get the width.

Returns	The current width.

---

void setARGB(int[] argb)

Set the ARGB data.

Parameters

argb	The new ARGB data.

---

void setComparator(ImageComparator comparator)
Set `ImageComparator` implementation, which will be used for comparing images.

Parameters

comparator	The new `ImageComparator` implementation.

---

void setHeight(int height)

Set the height.

Parameters

height	The new height.

---

void setName(String name)

Set the name.

Parameters

name	The new name.

---

void setPng(byte[] png)

Set the RGB data.

Parameters

png	The new RGB data.

---

void setWidth(int width)

Set the width.

Parameters

width	The new width.

52.10.2 The `ImageComparator` interface

The   interface
`de.qfs.apps.qftest.shared.extensions.image.ImageComparator` has to

be implemented, if you want to implement your own image comparison algorithm.

The implementation has then to be registered at the used `ImageRep` objects using the `setComparator` method.

---

boolean equals(ImageRep actual, ImageRep expected)
Return, whether the current object is equal to a given object. It uses the equals method of the current `ImageComparator` implementation.
Parameters

actual	The actual `ImageRep` object.
expected	The expected `ImageRep` object.
Returns	True, if images are equal, otherwise false.

52.10.3   The `ImageRepDrawer` class

The   class
`de.qfs.apps.qftest.shared.extensions.image.ImageRepDrawer` provides
methods to draw on an `ImageRep` object.

ImageRepDrawer arrow(int x1, int y1, int x2, int y2)
Draws an arrow.
Parameters

x1	The x-part of the first coordinate of the arrow.
y1	The y-part of the first coordinate of the arrow.
x2	The x-part of the second coordinate of the arrow.
y2	The y-part of the second coordinate of the arrow.
Returns	The ImageRepDrawer object for method concatination.

ImageRepDrawer arrow(int x1, int y1, int x2, int y2, int arrowStretch)
Draws an arrow.
Parameters

x1	The x-part of the first coordinate of the arrow.
y1	The y-part of the first coordinate of the arrow.
x2	The x-part of the second coordinate of the arrow.
y2	The y-part of the second coordinate of the arrow.
arrowStretch	The size of the arrow head.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer arrow(int x1, int y1, int x2, int y2, int arrowStretch, int strokeSize)`

Draws an arrow.

Parameters

`x1`	The x-part of the first coordinate of the arrow.
`y1`	The y-part of the first coordinate of the arrow.
`x2`	The x-part of the second coordinate of the arrow.
`y2`	The y-part of the second coordinate of the arrow.
`arrowStretch`	The size of the arrow head.
`strokeSize`	The size of the stroke.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer arrow(int x1, int y1, int x2, int y2, int arrowStretch, int strokeSize, Color strokeColor)`

Draws an arrow.

Parameters

`x1`	The x-part of the first coordinate of the arrow.
`y1`	The y-part of the first coordinate of the arrow.
`x2`	The x-part of the second coordinate of the arrow.
`y2`	The y-part of the second coordinate of the arrow.
`arrowStretch`	The size of the arrow head.
`strokeSize`	The size of the stroke.
`strokeColor`	The stroke color.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer arrow(int x1, int y1, int x2, int y2, int arrowStretch, int strokeSize, int r, int g, int b)`

Draws an arrow.

Parameters

`x1`	The x-part of the first coordinate of the arrow.
`y1`	The y-part of the first coordinate of the arrow.
`x2`	The x-part of the second coordinate of the arrow.
`y2`	The y-part of the second coordinate of the arrow.
`arrowStretch`	The size of the arrow head.
`strokeSize`	The size of the stroke.
`r`	The red part of the stroke color.
`g`	The green part of the stroke color.
`b`	The blue part of the stroke color.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer arrow(int x1, int y1, int x2, int y2, int arrowStretch, int strokeSize, int r, int g, int b, int a)`

Draws an arrow.

Parameters

`x1`	The x-part of the first coordinate of the arrow.
`y1`	The y-part of the first coordinate of the arrow.
`x2`	The x-part of the second coordinate of the arrow.
`y2`	The y-part of the second coordinate of the arrow.
`arrowStretch`	The size of the arrow head.
`strokeSize`	The size of the stroke.
`r`	The red part of the stroke color.
`g`	The green part of the stroke color.
`b`	The blue part of the stroke color.
`a`	The alpha part of the stroke color.
Returns	The ImageRepDrawer object for method concatination.

`BufferedImage asBufferedImage()`

Converts this image to a BufferedImage.

Returns	This image as BufferedImage.

`ImageRepDrawer circle(int x, int y, int r)`

Draws a circle.

Parameters

`x`	The x-coordinate of the center of the circle.
`y`	The y-coordinate of the center of the circle.
`r`	The radius of the circle.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer circle(int x, int y, int r, Color color)`

Draws a circle.

Parameters

`x`	The x-coordinate of the center of the circle.
`y`	The y-coordinate of the center of the circle.
`r`	The radius of the circle.
`color`	The color of the circle.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer cross(int x, int y)`

Draws a cross.

Parameters

`x`	The x-coordinate of the cross to draw.
`y`	The y-coordinate of the cross to draw.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer cross(int x, int y, int size)`

Draws a cross.

Parameters

`x`	The x-coordinate of the cross to draw.
`y`	The y-coordinate of the cross to draw.
`size`	The size of the cross to draw.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer draw(Object drawFunction)`

Draws on an ImageRep object.

Parameters

`drawFunction`	A lambda object. The lambda object takes a java.awt.Graphics2D object as input which can be used to draw onto the image.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer erase(int x, int y, int w, int h)`

"Erases" a particular area in the image.

Parameters

x	The x-coordinate of the area that should get erased.
y	The y-coordinate of the area that should get erased.
w	The width of the area that should get erased.
h	The height of the area that should get erased.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer fillRectangle(int x, int y, int w, int h)`

Draws a filled rectangle.

Parameters

x	The x-coordinate of the rectangle to draw.
y	The y-coordinate of the rectangle to draw.
w	The width of the rectangle to draw.
h	The height of the rectangle to draw.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer fillRectangle(int x, int y, int w, int h, Color color)`

Draws a filled rectangle.

Parameters

x	The x-coordinate of the rectangle to draw.
y	The y-coordinate of the rectangle to draw.
w	The width of the rectangle to draw.
h	The height of the rectangle to draw.
color	The color to use to fill the rectangle.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer image(ImageRep imgToDraw, int x, int y)`

Draws an image onto the already existing image.

Parameters

imgToDraw	The image to draw.
x	The x-coordinate where the image should get drawn.
y	The y-coordinate where the image should get drawn.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer image(Image img, int x, int y)`

Draws an image onto the already existing image.

Parameters

`img`	The image to draw.
`x`	The x-coordinate where the image should get drawn.
`y`	The y-coordinate where the image should get drawn.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer line(int x1, int y1, int x2, int y2)`

Draws a line.

Parameters

`x1`	The x-part of the first coordinate of the line to draw.
`y1`	The y-part of the first coordinate of the line to draw.
`x2`	The x-part of the second coordinate of the line to draw.
`y2`	The y-part of the second coordinate of the line to draw.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer line(int x1, int y1, int x2, int y2, Color strokeColor)`

Draws a line.

Parameters

`x1`	The x-part of the first coordinate of the line to draw.
`y1`	The y-part of the first coordinate of the line to draw.
`x2`	The x-part of the second coordinate of the line to draw.
`y2`	The y-part of the second coordinate of the line to draw.
`strokeColor`	The color that should get used for stroke drawing.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer line(int x1, int y1, int x2, int y2, int strokeSize)`

Draws a line.

Parameters

`x1`	The x-part of the first coordinate of the line to draw.
`y1`	The y-part of the first coordinate of the line to draw.
`x2`	The x-part of the second coordinate of the line to draw.
`y2`	The y-part of the second coordinate of the line to draw.
`strokeSize`	The stroke thickness.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer line(int x1, int y1, int x2, int y2, int strokeSize, Color strokeColor)`

Draws a line.

Parameters

`x1`	The x-part of the first coordinate of the line to draw.
`y1`	The y-part of the first coordinate of the line to draw.
`x2`	The x-part of the second coordinate of the line to draw.
`y2`	The y-part of the second coordinate of the line to draw.
`strokeSize`	The line thickness of the line to draw.
`strokeColor`	The color to use for line drawing.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer line(int x1, int y1, int x2, int y2, int r, int g, int b)`

Draws a line.

Parameters

`x1`	The x-part of the first coordinate of the line to draw.
`y1`	The y-part of the first coordinate of the line to draw.
`x2`	The x-part of the second coordinate of the line to draw.
`y2`	The y-part of the second coordinate of the line to draw.
`r`	The red value of the color that should get used for stroke drawing.
`g`	The green value of the color that should get used for stroke drawing.
`b`	The blue value of the color that should get used for stroke drawing.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer line(int x1, int y1, int x2, int y2, int r, int g, int b, int a)`

Draws a line.

Parameters

`x1`	The x-part of the first coordinate of the line to draw.
`y1`	The y-part of the first coordinate of the line to draw.
`x2`	The x-part of the second coordinate of the line to draw.
`y2`	The y-part of the second coordinate of the line to draw.
`r`	The red value of the color that should get used for stroke drawing.
`g`	The green value of the color that should get used for stroke drawing.
`b`	The blue value of the color that should get used for stroke drawing.
`a`	The alpha value of the color that should get used for stroke drawing.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer line(int x1, int y1, int x2, int y2, int strokeSize, int r, int g, int b, int a)`

Draws a line.

Parameters

`x1`	The x-part of the first coordinate of the line to draw.
`y1`	The y-part of the first coordinate of the line to draw.
`x2`	The x-part of the second coordinate of the line to draw.
`y2`	The y-part of the second coordinate of the line to draw.
`strokeSize`	The line thickness of the line to draw.
`r`	The red value of the color to use for line drawing.
`g`	The green value of the color to use for line drawing.
`b`	The blue value of the color to use for line drawing.
`a`	The alpha value of the color to use for line drawing.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer pixel(int x, int y)`
Colors a particular pixel on the image by using the color previously set via setStroke-Color. In case no color was set previously, the color black will be used.
Parameters

x	The x-coordinate of the pixel that should get colored.
y	The y-coordinate of the pixel that should get colored.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer pixel(int x, int y, java.awt.Color color)`
Colors a particular pixel on the image.
Parameters

x	The x-coordinate of the pixel that should get colored.
y	The y-coordinate of the pixel that should get colored.
c	The color object that should get used to color that particular pixel.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer pixel(int x, int y, int r, int g, int b)`
Colors a particular pixel on the image.
Parameters

x	The x-coordinate of the pixel that should get colored.
y	The y-coordinate of the pixel that should get colored.
r	The red value of the color that should get used to color a particular pixel.
g	The green value of the color that should get used to color a particular pixel.
b	The blue value of the color that should get used to color a particular pixel.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer pixel(int x, int y, int r, int g, int b, int a)`

Colors a particular pixel on the image.

Parameters

x	The x-coordinate of the pixel that should get colored.
y	The y-coordinate of the pixel that should get colored.
r	The red value of the color that should get used to color a particular pixel.
g	The green value of the color that should get used to color a particular pixel.
b	The blue value of the color that should get used to color a particular pixel.
a	The alpha value of the color that should get used to color a particular pixel.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer rectangle(int x, int y, int w, int h)`

Draws a rectangle.

Parameters

x	The x-coordinate where the rectangle should get drawn.
y	The y-coordinate where the rectangle should get drawn.
w	The width of the rectangle that should get drawn.
h	The height of the rectangle that should get drawn.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer rectangle(int x, int y, int w, int h, int b)`

Draws a rectangle.

Parameters

x	The x-coordinate where the rectangle should get drawn.
y	The y-coordinate where the rectangle should get drawn.
w	The width of the rectangle that should get drawn.
h	The height of the rectangle that should get drawn.
b	The width of the stroke of the rectangle.
Returns	The ImageRepDrawer object for method concatination.

ImageRepDrawer rectangle(int x, int y, int w, int h, int b,
Color strokeColor)
Draws a rectangle.
Parameters

x	The x-coordinate where the rectangle should get drawn.
y	The y-coordinate where the rectangle should get drawn.
w	The width of the rectangle that should get drawn.
h	The height of the rectangle that should get drawn.
b	The width of the stroke of the rectangle.
strokeColor	The color to use for the rectangle strokes.
Returns	The ImageRepDrawer object for method concatination.

ImageRepDrawer rectangle(int x, int y, int w, int h, int b,
Color strokeColor, Color fillColor)
Draws a rectangle.
Parameters

x	The x-coordinate where the rectangle should get drawn.
y	The y-coordinate where the rectangle should get drawn.
w	The width of the rectangle that should get drawn.
h	The height of the rectangle that should get drawn.
b	The width of the stroke of the rectangle.
strokeColor	The color to use for the rectangle strokes.
fillColor	The color to use to fill the rectangle.
Returns	The ImageRepDrawer object for method concatination.

ImageRepDrawer rectangle(int x, int y, int w, int h, int b, int
strokeCap, Color strokeColor)
Draws a rectangle.
Parameters

x	The x-coordinate where the rectangle should get drawn.
y	The y-coordinate where the rectangle should get drawn.
w	The width of the rectangle that should get drawn.
h	The height of the rectangle that should get drawn.
b	The width of the stroke of the rectangle.
strokeCap	The stroke caps to use.
strokeColor	The color to use for the rectangle strokes.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer rectangle(int x, int y, int w, int h, int b, int strokeCap, Color strokeColor, Color fillColor)`

Draws a rectangle.

Parameters

`x`	The x-coordinate where the rectangle should get drawn.
`y`	The y-coordinate where the rectangle should get drawn.
`w`	The width of the rectangle that should get drawn.
`h`	The height of the rectangle that should get drawn.
`b`	The width of the stroke of the rectangle.
`strokeCap`	The stroke cap to use for rectangle drawing.
`strokeColor`	The stroke color to use for rectangle drawing.
`fillColor`	The color to use to fill the rectangle.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer rectangle(int x, int y, int w, int h, int b, int strokeCap, int rstroke, int gstroke, int bstroke, int rfill, int gfill, int bfill)`

Draws a rectangle.

Parameters

`x`	The x-coordinate where the rectangle should get drawn.
`y`	The y-coordinate where the rectangle should get drawn.
`w`	The width of the rectangle that should get drawn.
`h`	The height of the rectangle that should get drawn.
`b`	The width of the stroke of the rectangle.
`strokeCap`	The stroke cap to use for rectangle drawing.
`rstroke`	The red value of the color to use for the rectangle strokes.
`gstroke`	The green value of the color to use for the rectangle strokes.
`bstroke`	The blue value of the color to use for the rectangle strokes.
`rfill`	The red value of the color to use to fill the rectangle.
`gfill`	The green value of the color to use to fill the rectangle.
`bfill`	The blue value of the color to use to fill the rectangle.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer rectangle(int x, int y, int w, int h, int b, int strokeCap, int rstroke, int gstroke, int bstroke, int astroke, int rfill, int gfill, int bfill, int afill)`

Draws a rectangle.

Parameters

`x`	The x-coordinate where the rectangle should get drawn.
`y`	The y-coordinate where the rectangle should get drawn.
`w`	The width of the rectangle that should get drawn.
`h`	The height of the rectangle that should get drawn.
`b`	The width of the stroke of the rectangle.
`strokeCap`	The stroke cap to use for rectangle drawing.
`rstroke`	The red value of the color to use for the rectangle strokes.
`gstroke`	The green value of the color to use for the rectangle strokes.
`bstroke`	The blue value of the color to use for the rectangle strokes.
`astroke`	The alpha value of the color to use for the rectangle strokes.
`rfill`	The red value of the color to use to fill the rectangle.
`gfill`	The green value of the color to use to fill the rectangle.
`bfill`	The blue value of the color to use to fill the rectangle.
`afill`	The alpha value of the color to use to fill the rectangle.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer setFillColor(java.awt.Color color)`

Sets the default fill color.

Parameters

`color`	The new fill color.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer setFillColor(int r, int g, int b)`

Sets the default fill color.

Parameters

`r`	The red component of the fill color.
`g`	The green component of the fill color.
`b`	The blue component of the fill color.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer setFillColor(int r, int g, int b, int a)`

Sets the default fill color.

Parameters

`r`	The red component of the fill color.
`g`	The green component of the fill color.
`b`	The blue component of the fill color.
`a`	The alpha component of the fill color.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer setFont(java.awt.Font font)`

Sets the default font.

Parameters

`font`	The default font to use.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer setFont(String fontName)`

Sets the default font.

Parameters

`fontName`	Der Name of the default font to use.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer setFont(String fontName, int size)`

Sets the default font.

Parameters

`fontName`	The name of the default font to use.
`size`	The size of the default font to use.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer setFont(String fontName, int size, int style)`

Sets the default font.

Parameters

`fontName`	The name of the default font to use.
`size`	The size of the default font to use.
`style`	The style of the default font to use.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer setStrokeCap(int cap)`

Set the stroke caps.

Parameters

`cap`	Can either be ImageRepDrawer.CAPS_SQUARED for strokes with squared caps or ImageRep-Drawer.CAPS_ROUND for strokes with round caps.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer setStrokeColor(java.awt.Color color)`

Sets the default stroke color.

Parameters

`color`	The new color.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer setStrokeColor(int r, int g, int b)`

Sets the default stroke color.

Parameters

`r`	The red component of the stroke color.
`g`	The green component of the stroke color.
`b`	The blue component of the stroke color.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer setStrokeColor(int r, int g, int b, int a)`

Sets the default stroke color.

Parameters

`r`	The red component of the stroke color.
`g`	The green component of the stroke color.
`b`	The blue component of the stroke color.
`a`	The alpha component of the stroke color.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer setStrokeSize(int size)`

Sets the default stroke thickness.

Parameters

`size`	The new default stroke thickness.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer text(int x, int y, String text)`

Draws a particular text onto the image.

Parameters

`x`	The x-coordinate of the position where the text should get drawn.
`y`	The y-coordinate of the position where the text should get drawn.
`text`	The text that should get drawn.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer text(int x, int y, String text, Color textColor)`

Draws a particular text onto the image.

Parameters

`x`	The x-coordinate of the Position where the text should get drawn.
`y`	The y-coordinate of the Position where the text should get drawn.
`text`	The text that should get drawn.
`textColor`	The color that should get used.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer text(int x, int y, String text, String fontName)`

Draws a particular text onto the image.

Parameters

`x`	The x-coordinate of the position where the text should get drawn.
`y`	The y-coordinate of the position where the text should get drawn.
`text`	The text that should get drawn.
`fontName`	The name of the font to use.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer text(int x, int y, String text, Color textColor, Font font)`

Draws a particular text onto the image.

Parameters

`x`	The x-coordinate of the Position where the text should get drawn.
`y`	The y-coordinate of the Position where the text should get drawn.
`text`	The text that should get drawn.
`textColor`	The color that should get used.
`font`	The font that should get used.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer text(int x, int y, String text, String fontName, int fontSize)`

Draws a particular text onto the image.

Parameters

`x`	The x-coordinate of the position where the text should get drawn.
`y`	The y-coordinate of the position where the text should get drawn.
`text`	The text that should get drawn.
`fontName`	The name of the font to use.
`fontSize`	The font size to use.
Returns	The ImageRepDrawer object for method concatination.

`ImageRepDrawer text(int x, int y, String text, int r, int g, int b)`

Draws a particular text onto the image.

Parameters

`x`	The x-coordinate of the position where the text should get drawn.
`y`	The y-coordinate of the position where the text should get drawn.
`text`	The text that should get drawn.
`r`	The red value of the color that should get used.
`g`	The green value of the color that should get used.
`b`	The blue value of the color that should get used.
Returns	The ImageRepDrawer object for method concatination.

ImageRepDrawer text(int x, int y, String text, String fontName, int fontSize, int fontStyle)
Draws a particular text onto the image.
Parameters

x	The x-coordinate of the position where the text should get drawn.
y	The y-coordinate of the position where the text should get drawn.
text	The text that should get drawn.
fontName	The name of the font to use.
fontSize	The font size to use.
fontStyle	The font style to use.
Returns	The ImageRepDrawer object for method concatination.

ImageRepDrawer text(int x, int y, String text, int r, int g, int b, int a)
Draws a particular text onto the image.
Parameters

x	The x-coordinate of the position where the text should get drawn.
y	The y-coordinate of the position where the text should get drawn.
text	The text that should get drawn.
r	The red value of the color that should get used.
g	The green value of the color that should get used.
b	The blue value of the color that should get used.
a	The alpha value of the color that should get used.
Returns	The ImageRepDrawer object for method concatination.

In the following examples shows how the ImageRepDrawer method can be used to draw onto an ImageRep object:

```
from imagewrapper import ImageWrapper
from java.awt import Color
iw = ImageWrapper(rc)
img = iw.loadPng(r"C:/temp/foobar.png")
img.draw().setStrokeSize(12).setStrokeColor(Color.RED).line(20, 20, 2000, 3500) \
    .setStrokeColor(Color.GREEN).setStrokeSize(1).rectangle(40, 45, 250, 300)
rc.logImage(img)
```

Example 52.53: Using the ImageRepDrawer object to draw a red line and a green rectangle on an image.

52.11 Pseudo DOM API

Web

To a certain extent QF-Test exposes the DOM of a web based SUT to SUT script[(631)] nodes. This API is not equivalent to working directly at the JavaScript level which can be done via the methods `toJS`, `callJS` and `evalJS` described in this chapter. With this API is possible to traverse the DOM and retrieve and set attributes of the respective nodes, but not to manipulate the structure of the DOM. Thus this API is useful for implementing `Name-` or `FeatureResolvers` as described in section 52.1[(1016)].

For Swing, FX and SWT QF-Test works with the actual Java GUI classes whereas a pseudo class hierarchy is used for web applications as follows:
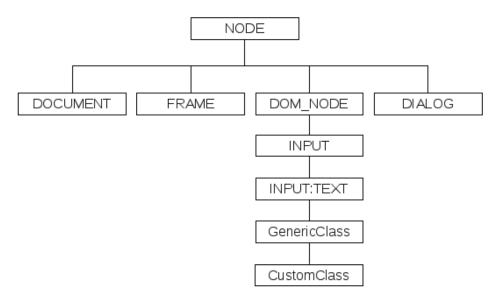
Figure 52.1: Pseudo class hierarchy for web elements

As shown, "NODE" is at the root of the pseudo class hierarchy. It matches any kind of element in the DOM. Derived from "NODE" are "DOCUMENT", "FRAME", "DOM_NODE"

and "DIALOG", the types of nodes implementing the pseudo DOM API explained in section 52.11[(1118)]. "DOM_NODE" is further sub-classed according to the tag name of the node, e.g. "H1", "A" or "INPUT" where some tags have an additional subclass like "INPUT:TEXT".

Note   The DOM can differ depending on the browser, so you should try not to rely too much on child indexes in your resolvers or scripts in case of cross-browser testing if viable.

QF-Test's DOM API comprises a hierarchy of the following five classes:

52.11.1 The abstract `Node` class

All classes of QF-Test's pseudo DOM are derived from this class and thus implement the following interface. Its is located in the package `de.qfs.apps.qftest.client.web.dom`.

`Node findCommonAncestor(Node node1, Node node2, Node topmost)`
Get the common parent of two nodes.
Parameters

`node1`	The first node.
`node2`	The second node.
`topmost`	The topmost node to stop the search.
Returns	The common parent or null.

`Node getAncestorOfClass(String clazz)`
Get the closest parent of a specified class name.
Parameters

`clazz`	The parent's class name.
Returns	The parent or null.

`Node getAncestorOfClass(String clazz, int maxDepth)`
Get the closest parent of a specified class name. You can specify a maximal search level.
Parameters

`clazz`	The parent's class name.
`maxDepth`	The maximum search level.
Returns	The parent or null.

`String getAttribute(String name)`
Get the value of an attribute of the node. For convenience this method is defined at Node level. If this node is not a `DomNode` the result will always be null. In some cases this method also returns attributes that have not been explicitly specified, like the width and height of an IMG node. Which attributes can be retrieved in that way is browser dependent.
Parameters

`name`	The name of the attribute
Returns	The value of the attribute or null if no such attribute is available for the node.

`String getAttributeIfSpecified(String name)`
Get the value of an attribute of the node if it is explicitly specified in the HTML code. For convenience this method is defined at Node level. If this node is not a `DomNode` the result will always be null.
Parameters

`name`	The name of the attribute
Returns	The value of the attribute or null if no such attribute is explicitly specified for the node.

`Node getChild(int index)`
Get the child node at the given index. Remark for Opera before version 86, Google Chrome and Microsoft Edge before version 100: Text-nodes, which are only used to format the HTML code and which have no influence on the visualisation of the web site are not transferred from the browser to QF-Test in CDP connection mode. Therefore, a different index of the child node can occure in this connection mode wth these browsers. If possible, use `getChildrenByTagName` in your scripts.
Parameters

`index`	The index of the child node, starting with 0.
Returns	The child node at the index.

Throws

`IllegalArgumentException`	if index is negative or exceeds the number of child nodes.

`int getChildCount()`
Get the number of child nodes. Remark for Opera before version 86, Google Chrome and Microsoft Edge before version 100: Text-nodes, which are only used to format the HTML code and which have no influence on the visualisation of the web site are not transferred from the browser to QF-Test in CDP connection mode. Therefore, a different number of the child nodes can occure in this connection mode with these browsers.

Returns	The number of child nodes.

Node[] getChildren()
Get the children of this node as an array. Remark for Opera before version 86, Google Chrome and Microsoft Edge before version 100: Text-nodes, which are only used to format the HTML code and which have no influence on the visualisation of the web site are not transferred from the browser to QF-Test in CDP connection mode. Therefore, less children might be reported in this connection mode with these browsers.

Returns	The child nodes.

String getClassName()
Get the most current class name of that node.

Returns	The most current class name.

String[] getClassNames()
Get all class names of that node.

Returns	An array with all class names of that node.

DocumentNode getDocument()
Get the document to which this node belongs.

Returns	The document to which this node belongs. A `DocumentNode` will return itself, a `DialogNode` will return null.

Node getElementById(String id)
Get a direct or indirect child node of this node with a given ID. The given ID is not compared to the node's original 'ID' attribute, but to its ID, which may have been modified by an `IdResolver` (see section 52.1.17[(1039)]).

Parameters

id	The ID to look for.
Returns	The child node with the given ID, an arbitrary one in case of multiple matches or null if none is found. A `DialogNode` will always return null, a `FrameNode` forwards the call to its `DocumentNode` child node and a `DocumentNode` to its <HTML> root element.

Node[] getElementsByClassName(String className)
Get all direct and indirect child nodes with a specified class.

Parameters

className	The name of the class.
Returns	An array of child nodes with the specified class name. If none are found, an empty array is returned.

Node[] getElementsById(String id)
Get all direct or indirect child nodes of this node with a given ID. The given ID is not compared to the node's original 'ID' attribute, but to its ID, which may have been modified by an `IdResolver` (see section 52.1.17[(1039)]).
Parameters

id	The ID to look for.
Returns	An array of child nodes with the given ID. If none are found, an empty array is returned. A `DialogNode` will always return an empty array, a `FrameNode` forwards the call to its `DocumentNode` child node and a `DocumentNode` to its <HTML> root element.

Node[] getElementsByIdAndTagname(String id, String tagName)
Get all direct or indirect child nodes of this node with a given ID and a given tag name. The given ID is not compared to the node's original 'ID' attribute, but to its ID, which may have been modified by an `IdResolver` (see section 52.1.17[(1039)]).
Parameters

id	The ID to look for.
tagName	The tag name to look for.
Returns	An array of child nodes with the given ID and the given tag name. If none are found, an empty array is returned. A `DialogNode` will always return an empty array, a `FrameNode` forwards the call to its `DocumentNode` child node and a `DocumentNode` to its <HTML> root element.

Node[] getElementsByTagName(String tagName)
Get all direct or indirect child nodes of this node with a given tag name.
Parameters

tagName	The tag name to look for.
Returns	An array of child nodes with the given tag name. If none are found, an empty array is returned. A `DialogNode` will always return an empty array, a `FrameNode` forwards the call to its `DocumentNode` child node and a `DocumentNode` to its <HTML> root element.

Node getFirstChild()
Get the first child node: an element node, a text node or a comment.

Returns	The first child node.

Node getFirstElementChild()
Get the first child element (text nodes and comments are skipped).

Returns	The first child element.

String getFlatText()

Get the flat text content of a Node. In case of a plain text node, return its value. Otherwise collect the values of all text nodes directly contained within this node, excluding those nested at a deeper level.

| **Returns** | The whole direct text content of a node. |

String getGenericClassName()

Get the generic class name of that node.

| **Returns** | The generic class name. |

int getIndexOfChild(Node child)

Get the index of a given child node.

Parameters

| **child** | The child node to get the index for. |
| **Returns** | The index of the child node, starting with 0, or -1 in case it is not a child of the this node. |

Node getInterestingParent()

Get the interesting parent of a node. A top-level `DocumentNode` or a `DialogNode` will return null. Everything else should have a parent unless removed from the DOM via JavaScript.

| **Returns** | The interesting parent of the node. |

Node getInterestingParent(int n)

Get the interesting parent of a node. A top-level `DocumentNode` or a `DialogNode` will return null. Everything else should have a parent unless removed from the DOM via JavaScript.

Parameters

| **n** | The interesting parent's level. |
| **Returns** | The interesting parent of the node. |

String getName()

Get the tag name of the node. The tag name is the type of node in upper case like "HTML" for an <HTML> node. Plain text nodes are represented as a `DomNode` with tag name "#text". Pseudo tag names are defined for `DocumentNodes` ("DOCUMENT") and `DialogNodes` ("DIALOG").

| **Returns** | The tag name of the node. |

Node getNextElementSibling()

Get the next element in the same tree level.

| **Returns** | The next element sibling. |

Node getNextSibling()

Get the next sibling node: An element node, a text node, or a comment node.

| **Returns** | The next sibling. |

`String getNodeType()`
Get an identifier for the type of the node. Though this method is not fully in line with pure OO doctrine, while traversing the DOM it is often necessary to find out the type of a given node and this is quite convenient.

Returns	A string specifying the type of node. The respective constants are defined in the concrete sub-classes: `DocumentNode.DOCUMENT_NODE`, `FrameNode.FRAME_NODE`, `DomNode.DOM_NODE` and `DialogNode.DIALOG_NODE`.

`Node getNthParent(int n)`
Get the n-th parent node.

Parameters

n	The parent's level.
Returns	The n-th parent or null.

`Node getParent()`
Get the parent of a node. A top-level `DocumentNode` or a `DialogNode` will return null. Everything else should have a parent unless removed from the DOM via JavaScript.

Returns	The parent of the node.

`Node getPreviousElementSibling()`
Get the previous element in the same tree level.

Returns	The previous element sibling.

`Node getPreviousSibling()`
Get the previous sibling node: An element node, a text node, or a comment node.

Returns	The previous sibling.

`Object getProperty(String name)`
Retrieve a user-defined property.

Parameters

name	The name of the property to retrieve.
Returns	The value of the specified property or null.

`String getSimpleText()`
Get the simple text content of a Node. In case of a plain text node, return its value. Otherwise traverse the DOM and collect all text nodes contained directly within this node or in "simple" nodes like <B>, but do not descend into structurally more complex nodes like <TABLE>.

Returns	The simple text content of a node.

String getText()
Get the text content of a Node. In case of a plain text node, return its value. Otherwise traverse the DOM and collect all nested text node's values. Care is taken to collapse whitespace between nodes to get as close as possible to what is displayed in the browser.

| **Returns** | The whole direct and indirect text content of a node. |

String getVisibleFlatText()
Get the flat text content of a Node. In case of a plain text node, return its value. Otherwise collect the values of all text nodes directly contained within this node, excluding invisible ones and those nested at a deeper level.

| **Returns** | The whole direct visible text content of a node. |

String getVisibleSimpleText()
Get the simple text content of a Node. In case of a plain text node, return its value. Otherwise traverse the DOM and collect all visible text nodes contained directly within this node or in "simple" nodes like <B>, but do not descend into structurally more complex nodes like <TABLE>.

| **Returns** | The visible simple text content of a node. |

String getVisibleText()
Get the text content of a Node. In case of a plain text node, return its value. Otherwise traverse the DOM and collect all nested text node's values, ignoring invisible nodes. Care is taken to collapse whitespace between nodes to get as close as possible to what is displayed in the browser.

| **Returns** | The whole direct and indirect visible text content of a node. |

boolean isAncestor(Node node)
Check whether the given node is an ancestor of the current node.

Parameters

| **node** | The possible ancestor node. |
| **Returns** | True, if node is an ancestor, otherwise false. |

boolean isAttributeSpecified(String name)
Test whether an attribute of the node is explicitly specified in the HTML code. For convenience this method is defined at Node level. If this node is not a `DomNode` the result will always be false.

Parameters

| **name** | The name of the attribute |
| **Returns** | True if an attribute is explicitly specified for the node. |

boolean isBrowserChrome()
Test whether the browser to which the node belongs is a Chrome derivative.

| **Returns** | True for Chrome derivatives, false otherwhise. |

`boolean isBrowserHeadless()`
Test whether the browser to which the node belongs is a headless browser.

Returns	True for headless browser, false otherwhise.

`boolean isBrowserMozilla()`
Test whether the browser to which the node belongs is a Mozilla derivative.

Returns	True for Mozilla derivatives, false otherwhise.

`boolean isBrowserSafari()`
Test whether the browser to which the node belongs is a Safari.

Returns	True for Safari, false otherwhise.

`boolean isMatchingClass(String className)`
Check, whether the node matches a given class name.

Parameters

`className`	The class name to check.
Returns	True, if node matches this class, otherwise false.

`void setProperty(String name, Object value)`
Set a user-defined property.

Parameters

`name`	The name of the property.
`value`	The property value or null to remove an existing property.

52.11.2 The `DocumentNode` class

Web The root `document` of a web page is not represented by a Window node, but the special node Web page[(818)]. Nested `document` nodes in `frames` correspond to Components nodes.

The `DocumentNode` class is derived from `Node` and also resides in the package `de.qfs.apps.qftest.client.web.dom`. In addition to the methods defined in the `Node` class and explained above, `DocumentNode` provides the following:

`Object callJS(String code)`
Execute some JavaScript code within a function in the context of this document. This often works even if the call of `eval()` is restricted by a Content Security Policy (CSP).
Parameters

`code`	The code to execute.
Returns	Whatever the code returns explicitly using a `return` statement, converted to the proper object type. Even returning a DOM node, frame or document works.

`Object evalJS(String script)`
Evaluate some JavaScript code in the context of this document by calling `window.eval()`. In most cases, the method `callJS` should be use, since the call of `eval()` might be restricted for the document by a Content Security Policy (CSP).
Parameters

`script`	The script to execute.
Returns	Whatever the script returns, converted to the proper object type. Even returning a DOM node, frame or document works.

`FrameNode[] getFrames()`
Get the child frames of the document.

Returns	The child frames of the document, an empty array in case there are none.

`DomNode getRootElement()`
Get the <HTML> root element of the document.

Returns	The root element of the document.

`String getSourcecode()`
Get the HTML source code of the document in its current state, which is not necessarily the same as what was loaded when the document was opened because attributes or the DOM's structure may have been changes since, e.g. via JavaScript.

Returns	The current HTML code of the document.

`String getTitle()`
Get the title of the document as defined in the <TITLE> of the <HEAD> of its root element.

Returns	The title of the document.

`String getUrl()`
Get the URL of the document.

Returns	The URL of the document.

`boolean hasParent()`
To test whether a document is a top-level document this method should be used instead of testing whether the result of `getParent()` is null. The reason is that loading of nested child documents may be completed before loading of the main document. During this time it is known that the nested document will have a parent, but the parent is not available yet.

Returns	True if the document has a parent, false if it is a top-level document.

In addition to the instance methods describe above, the `DocumentNode` class provides some static convenience methods for manipulating URLs.

static String getUrlBase(String url)
Get the base part of an URL, including host and directory but excluding possible parameters separated by '?' and excluding a possible Tomcat session ID.
Parameters

url	The URL to get the base part of.
Returns	The base part of the URL.

static String getUrlHost(String url)
Get the host part of an URL, i.e. the part between "http(s)://" and the next '/'.
Parameters

url	The URL to get the host part of.
Returns	The host part of the URL or null if the protocol is neither http nor https.

static ExtraFeatureSet getUrlParameters(String url)
Get the parameters of an URL, including those separated by '?' and a possible Tomcat session ID, in the form of an `ExtraFeatureSet` that can be used for an `ExtraFeatureResolver` as described in section 52.1.11[(1028)].
Parameters

url	The URL to get the parameters for.
Returns	The parameters of the URL.

static String normalizeUrl(String url)
Normalize an URL by replacing '\' with '/', "file://(/(/))" with "file:///", up-casing windows drive letters and decoding %.. sequences.
Parameters

url	The URL to normalize.
Returns	The normalized URL.

52.11.3   The **FrameNode** class

The `FrameNode` class is derived from `Node` and also resides in the package `de.qfs.apps.qftest.client.web.dom`. In addition to the methods defined in the `Node` class and explained above, `FrameNode` provides the following:

DocumentNode getChildDocument()
Get the child document of the frame.

Returns	The child document of the frame.

DomNode getFrameElement()

Get the `DomNode`-Object, which contains the frame, if known, i.e. an IFRAME- or FRAME-node.

| **Returns** | Der `DomNode` oder `null`. |

String getFrameName()

Get the name of the frame as defined by its "name" attribute.

| **Returns** | The name of the frame. |

String getFrameUrl()

Get the URL of the frame which should normally be the same as the URL of its child document.

| **Returns** | The URL of the frame. |

int[] getGeometry()

Get the location and size of the frame relative to its parent frame or the browser window's display area.

| **Returns** | The frame geometry in the form [x, y, width, height]. |

52.11.4   The `DomNode` class

The `DomNode` class is derived from `Node` and also resides in the package `de.qfs.apps.qftest.client.web.dom`. In addition to the methods defined in the `Node` class and explained above, `DomNode` provides the following:

Object callJS(String code)

Execute some JavaScript code in a function in the context of this node's document. This often works even if the call of `eval()` is restricted by a Content Security Policy (CSP).

Parameters

| **code** | The script to execute. Use `_qf_node` as the reference for the HTML element. |
| **Returns** | Whatever the code returns explicitly using a `return` statement, converted to the proper object type. Even returning a DOM node, frame or document works. |

`Object evalJS(String script)`
Evaluate some JavaScript code in the context of this node's document by calling `window.eval()`. In most cases, the method `callJS` should be use, since the call of `eval()` might be restricted for the document by a Content Security Policy (CSP).
Parameters

`script`	The script to execute. Use `_qf_node` as the reference for the HTML element.
Returns	Whatever the script returns, converted to the proper object type. Even returning a DOM node, frame or document works.

`DomNode[] getAllByCSS(String css)`
Get all nodes, which are determined via CSS-selectors.
Parameters

`css`	CSS-selector starting from that node.
Returns	An array of all found nodes or null.

`DomNode[] getAllByXPath(String xpath)`
Get all nodes, which are determined via XPath.
Parameters

`xpath`	XPath starting from that node.
Returns	An array of all found nodes or null.

`DomNode getByCSS(String css)`
Get a node, which is determined via a CSS-selector.
Parameters

`css`	CSS-selector starting from that node.
Returns	The found node or null.

`DomNode getByXPath(String xpath)`
Get a node, which is determined via XPath.
Parameters

`xpath`	XPath starting from that node.
Returns	The found node or null.

`Node[] getChildrenByTagName(String tagName)`
Get all direct child nodes of this node with a given tag name.
Parameters

`tagName`	The tag name to look for.
Returns	An array of child nodes with the given tag name. If none are found, an empty array is returned.

`String getId()`

Get the cached ID of the node which may have been processed by an `IdResolver` as described in section 52.1.17[(1039)]. The original, unmodified 'ID' attribute is available via `getAttribute("id")`.

Returns	The cached ID of the node.

`int[] getLocationOnScreen()`

Get the location and size of the node relative to desktop.

Returns	The node's geometry in the form [x, y, width, height].

`boolean hasCSSClass(String cl)`

Check, whether the node has a specified css-class.

Parameters

cl	The name of the css-class.
Returns	True, if the node has this css-class, otherwise false.

`boolean hasFocus()`

Test whether the node has the keyboard focus.

Returns	True if the node has the keyboard focus.

`boolean isShowing()`

Test whether the node is visible or can be made visible by scrolling.

Returns	True if the node is visible.

`void requestFocus()`

Tell the node to request the keyboard focus. This is a best-effort implementation. Whether the node actually receives the keyboard focus depends on the browser and operating system settings.

`void scrollVisible()`

Try to scroll the page or a frame so as to make the node fully visible. This is a best effort implementation. Whether the node can actually be made visible depends on the browser and operating system settings.

`void setAttribute(String name, String value)`

Set an attribute value on the node.

Parameters

name	The name of the attribute.
value	The value to set.

`void toJS(String name)`

Set a JavaScript variable in the context of this node's document to this node.

Parameters

name	The name of the variable to set.

52.11.5 The `DialogNode` class

The `DialogNode` class, also derived from `Node` is not a standard DOM class, but created solely for convenient access to dialogs within QF-Test. It also resides in the package `de.qfs.apps.qftest.client.web.dom` but has very little to do with the other `Node` classes. A `DialogNode` represents a message or error dialog which can be triggered via JavaScript. It provides the following methods:

`long getStyle()`
Get the style of the dialog.

Returns	The style of the dialog. Any of the constants `STYLE_ALERT`, `STYLE_CONFIRM` or `STYLE_AUTHENTICATE` defined in the `DialogNode` class.

`String getText()`
Get the message text of the dialog.

Returns	The message text of the dialog.

`String getTitle()`
Get the title of the dialog.

Returns	The title of the dialog.

52.12 WebDriver SUT API

4.1+

The WebDriverConnection SUT API provides classes and interfaces to enable using Selenium WebDriver Java API inside a SUT script[(631)]. With this kind of bridge you can use your existing Selenium WebDriver scripts inside a SUT script[(631)] of QF-Test. You can even combine the Pseudo DOM API (section 52.11[(1118)]) with Selenium WebDriver based scripts.

Note   This API is only usable if the browser is connected to QF-Test using connection mode "WebDriver". Calls on the returned WebDriver-object are automatically synchronized and guarded by a time-out.

```
from webdriver import WebDriverConnection
from org.openqa.selenium import By
wdc = WebDriverConnection(rc)
driver = wdc.getDriver()
# driver now of type org.openqa.selenium.WebDriver
element = driver.findElement(By.cssSelector(".myClass"))
# element is now of type org.openqa.selenium.WebElement
# You can call WebDriver-Methods directly on the element
element.click()
# Objects of type WebElement can be mapped to the QF-Test Pseudo DOM API
node = wdc.getComponent(element)
# and assigned to a component in the component tree
rc.overrideElement("Your-QF-Test-Id",node)
# Also, a QF-Test component can be translated to a WebElement object
node = rc.getComponent("QF-Test-Id-Of-Some-Textfield")
element = wdc.getElement(node)
# and interacted using WebDriver-methods
element.clear()
```

Example 52.54: WebDriver-Usage in a Jython SUT Script

Note The WebDriver-Object is extended by methods to control the automatic timeout.

```
import de.qfs.WebDriverConnection
def wdc = new WebDriverConnection(rc)
def driver = wdc.getDriver()
print sprintf("Current timeout: %d ms", driver.getCallTimeout())
driver.setCallTimeout(30000) # 30 sec
driver.get("http://www.slowpage.com") # Slow WebDriver-Action
driver.resetCallTimeout()
```

Example 52.55: WebDriver-Timeout Control (Groovy Script)

52.12.1 The `WebDriverConnection` class

Following is a list of the methods of the `WebDriverConnection` class in alphabetical order. The syntax used is a bit of a mixture of Java and Python. Python doesn't support static typing, but the parameters are passed on to Java, so they must be of the correct type to avoid triggering exceptions. If a parameter is followed by an '=' character and a value, that value is the default and the parameter is optional.

`Object getComponent(WebElement element, String windowname=None)`
Get the QF-Test component of a given WebDriver WebElement. This component could then be used with the other API's QF-Test provides.(e.g. the Pseudo DOM-API in section 52.11[(1118)])

Parameters

`element`	The WebDriver WebElement.
`windowname`	The windowname of the Browser of which the WebElement is requested.
Returns	The corresponding QF-Test component.

`WebDriver getDriver(String windowname=None)`
Get the WebDriver instance used to interact with a browser in WebDriver mode. Requires, that a web page was opened using a Start web engine[(645)] step.

Parameters

`windowname`	The windowname of the Browser of which the WebDriver is requested.
Returns	The WebDriver instance.

`WebElement getElement(Object componentOrId)`
Get the WebDriver WebElement of the given QF-Test component or the QF-Test component id.

Parameters

`componentOrId`	The QF-Test component or its QF-Test component id.
Returns	The WebDriver WebElement object of the component.

`WebDriver getUnmanagedDriver(String browserType=None, DesiredCapabilities desiredCapabilities=None)`
Get a WebDriver instance with the specified browser type. As long as no page has been opened with a Start web engine[(645)] step, the WebDriver instance is not monitored by QF-Test, so web pages and their components are not automatically detected by QF-Test. Interaction with the web page is only possible by the means of the embedded Selenium API, and checks have to be performed using `rc.check` and `rc.checkEqual` (see section 11.1[(162)])

Parameters

`browserType`	The type of browser to be tested (see Browser type[(646)]). Is this parameter empty or unset, the `browserName` capability of the *desiredCapabilities* is inspected. If this is also not set, the value of the Browser type[(646)] attribute of the Start web engine[(645)] step starting the SUT is used.
`desiredCapabilities`	The DesiredCapabilities, which should be handed over to the WebDriver instance.
Returns	The WebDriver instance.

52.13 Windows Control API

5.0+

The elments of native Windows applications are represented during a test by Java objects of the class `WinControl`. This class provides several public methods, e.g. to develop custom resolvers.

52.13.1 The `WinControl` class

Following is a list of the methods of the `WinControl` class in alphabetical order.

`WinControl getAncestorByUiaType(String typeName)`

Get the control ancestor which has the given UIAutomation type name.

Parameters

`typeName`	The type name.
Returns	The ancestor or `null`.

`WinControl getChild(int index)`

Get the control's child with the given index.

Parameters

`index`	The index.
Returns	The child.

Throws

`IllegalArgumentException`	If index is negative or exceeds the number of child nodes.

`int getChildCount()`

Get number of children of the control.

Returns	The child count.

`WinControl[] getChildren()`

Get the children of the control.

Returns	An array with the children.

`WinControl[] getChildrenByUiaClassName(String className)`

Get all children of the control which have the given UIAutomation class name.

Parameters

`className`	The class name.
Returns	An array with the children.

`WinControl[] getChildrenByUiaType(String typeName)`

Get all children of the control which have the given UIAutomation type name.

Parameters

`typeName`	The type name.
Returns	An array with the children.

`WinControl[] getElementsByClassName(String className)`

Get all descendants of the control which match the given class name.

Parameters

`className`	The class name to be matched.
Returns	An array of `WinControl` objects, which have the current object as ancestor and match the given class name.

`WinControl[] getElementsByClassName(String[] classNames)`

Get all descendants of the control which match any of the given class names.

Parameters

`classNames`	The class names to be matched.
Returns	An array of `WinControl` objects, which have the current object as ancestor and match any of the given class names.

`WinControl[] getElementsByClassName(String[] classNames, String[] stopClassNames)`

Get all descendants of the `WinControl` matching any of the given class names, but skips all controls (and their decendants) matching the given stop class names.

Parameters

`classNames`	The class names to be matched.
`stopClassNames`	The stop class names.
Returns	An array with the matching descendants.

`String[] getGenericClassNames()`

Get the generic class names of the control.

Returns	An array of Strings with the generic class names of the control.

`int getHwnd()`

Get the native window handle.

Returns	The native window handle.

`int[] getLocation()`

Get the (physical) location of the element within its parent.

Returns	An array with the X and Y coordinates.

---

int[] getLocationOnScreen()

Get the (physical) location of the element on the screen.

| **Returns** | An array with X, Y, width and height. |

WinControl getNextSibling()

Get the control's next sibling.

| **Returns** | The next sibling or `null`. |

String getPatterns()

Get all patterns in a string separated by whitespace. Prefer using `hasPattern()` for not to bother with the exact format of the return string.

| **Returns** | A string with the patterns of the element. |

---

int[] getSize()

Get the (physical) size of the element.

| **Returns** | An array with the width and the height. |

String getTextOrValue()

Get a value for an element, most of the time from the Value or Text pattern, if any. A value may also be retrieved from Text children or in special cases from the Automation Name.

| **Returns** | A value or `null`. |

WinControl getTopAncestor()

Get the top-level ancestor of the control.

| **Returns** | The top-level ancestor or `null`. |

String getUiaClassName()

Get the class name for the `WinControl`. This is the UIAutomation class name extended by a framework specific prefix to avoid confusion with QF-Test's generic class names.

| **Returns** | A String with the class name of the control. |

AutomationBase getUiaControl()

Create an `AutomationBase` control for the `WinControl`, compatible with the `uiauto` script module (chapter 50[(1000)]).

| **Returns** | The `AutomationBase` object. |

String getUiaDescription()

Get the UI Automation description of the control. If there is no full description, the accessibility description is returned as fallback.

| **Returns** | The description, if any or an empty string. |

String getUiaHelp()

Get the UI Automation help text of the control.

| **Returns** | The help text, if any or an empty string. |

String getUiaId()

Get the UI Automation identifier of the control.

| **Returns** | The ID, if any or `null`. |

String getUiaName()

Get the UI Automation name of the control.

| **Returns** | The name, if any or `null`. |

String getUiaType()

Get the type for the `WinControl`. This is the UIAutomation type name extended by a prefix `Uia.` to avoid confusion with QF-Test's generic class names.

| **Returns** | A String with the type of the control. |

---

boolean hasPattern(String pattern)

Check whether the underlying Automation Element supports the given pattern.

Parameters

| **pattern** | The pattern name, e.g, "Invoke", "ExpandCollapse", etc. |
| **Returns** | `true` if the pattern is supported by the element, `false` otherwise. |

---

boolean isMatchingClass(String className)

Check whether the control matches a given class name.

Parameters

| **className** | The class name to be checked. |
| **Returns** | `true` if the control matches the given class name, `false` otherwise. |

---

boolean isMatchingClass(String[] classNames)

Checks if any of the given classNames is part of the `WinControl`'s class names list.

Parameters

| **classNames** | The class names to be checked. |
| **Returns** | `true` if the control matches one of the given class names, `false` otherwise. |

---

boolean isShowing()

Get the visibility of an element.

| **Returns** | `true`, when the control is considered to be visible on the screen, `false` otherwise. |

Chapter 53

Daemon mode

!!! Warning !!!

Anybody with access to the QF-Test daemon can start any program on the machine running the daemon with the rights of the user account that the daemon is running under, so access should be granted only to trusted users.

If you are not running the daemon in a secure environment where every user is trusted or if you are creating your own library to connect to the QF-Test daemon, you definitely should **read section 53.3**[(1154)] below.

53.1   Daemon concepts

In daemon mode (comparable, but not equivalent to a "service" on Windows) QF-Test listens to RMI connections and providing an interface for remote test execution. This is useful for simplifying test execution in a distributed load-testing scenario, but also for integration with existing test management or test execution tools.

There are three command line arguments, all of which are available in batch and interactive mode:

- `-daemon`[(870)] - Run QF-Test in daemon mode.

- `-daemonport <port>`[(871)] - Specify the port the daemon should listen at, the default is 3543.

- `-daemonrmiport <port>`[(871)] - Specify the RMI port the daemon should use. Useful only when running the daemon behind a firewall. Default is a random free port.

When run in batch and daemon mode, QF-Test will not use a license. Licenses will be acquired during use as described below. In interactive daemon mode, QF-Test will

be fully functional and thus use a license. In addition it will accept connections from the outside and use additional licenses during use, similar to batch mode. The latter scenario is useful during development of distributed tests.

Tests can be executed on a running daemon in one of two ways:

- Use of QF-Test's batch commandline option <u>-calldaemon</u>⁽⁸⁷⁰⁾.

- Direct implementation against the daemon API, documented in <u>section 53.2</u>⁽¹¹⁴⁰⁾.

Both options are explained in more detail and accompanied by examples in <u>section 23.2</u>⁽²⁸⁶⁾.

53.2 Daemon API

The daemon API provides all methods necessary to directly control test execution via a QF-Test daemon. Typically the following relatively simple steps need to be implemented:

Note   When writing an application based on the daemon API you need to take security considerations into account and either disable security or set some RMI-specific properties as described in <u>section 53.3</u>⁽¹¹⁵⁴⁾.

- Get hold of a `Daemon` using the `DaemonLocator` service.

- Either get hold of the shared `TestRunDaemon` or have the `Daemon` create a `TestRunDaemon`. You can use the `TestRunDaemon` to define global variables and the root directory of your test suites for the upcoming test runs.

- Either get hold of the `TestRunDaemon`'s shared `DaemonRunContext` or have it create one or more `DaemonRunContext` instances. A `DaemonRunContext` represents one test execution context for functional or load testing. Each `DaemonRunContext` is independent of the others, except for a group of contexts created via `TestRunDaemon.createContexts(int threads)` which creates a group of related contexts for load-testing similar to running QF-Test with the <u>-threads <number></u>⁽⁸⁸³⁾ command line argument. Each `DaemonRunContext` requires one QF-Test development or runtime license during its lifetime.

- Now you can tell the `DaemonRunContext` to execute tests on your behalf, either whole test suites or <u>Test set</u>⁽⁵²⁵⁾ or <u>Test case</u>⁽⁵¹⁸⁾ nodes. With properly implemented <u>Dependencies</u>⁽⁵⁴⁸⁾ you can have a Daemon execute Test cases on your behalf in any arbitrary order without explicitly having to take care of any setup or cleanup tasks.

- The `DaemonRunContext` also has methods to check the current state of a test run or wait for it to finish.

- Finally you can get the run log of the test run from the Daemon. Right now, all you can do is save it, but we will open the run log API so that you will be able to integrate run logs from various daemons and test runs into a single run log.

For proper dependency management, including rollback of no longer required dependencies between Test case invocations, it is important to use the same `DaemonRunContext` for execution of each related Test case. The easiest way to achieve this is to use the shared `TestRunDaemon` and its shared `DaemonRunContext` each time you talk to a given `Daemon`.

The following sections provide a complete reference for the daemon API. Further explanations and examples are provided in .

53.2.1 The `DaemonLocator`

The singleton class `de.qfs.apps.qftest.daemon.DaemonLocator` can be used to get hold of `Daemon` instances.

---

static DaemonLocator instance()

Get the singleton instance.

Returns	The singleton instance.

Daemon locateDaemon(String host, int port)

Get a Daemon from a specific host and port.

Parameters

host	The target host, name or IP string.
port	The target port.
Returns	The daemon or null if none can be found.

Daemon[] locateDaemons(long timeout)

Get all known daemons.

Parameters

timeout	The time in milliseconds to wait for daemons to react.
Returns	The known daemons.

---

void setKeystore(String keystoreFile)

Sets the keystore, which should be used to to secure the daemon communication.

Parameters

keystoreFile	The path to the keystore file used to encrypt the daemon communication.

---

void setKeystorePassword(String password)

Sets the password for the keystore defined with `setKeystore`.

Parameters

password	The password.

---

void setTruststore(String truststoreFile)

Sets the truststore, which should be used to to secure the daemon communication. If not set, the keystore will be used, which has been set with `setKeystore`.

Parameters

truststoreFile	The path to the truststore file used to encrypt the daemon communication.

---

void setTruststorePassword(String password)

Sets the password for the trststore defined with `setTruststore`.

Parameters

password	The password.

53.2.2 The `Daemon`

The `de.qfs.apps.qftest.daemon.Daemon` interface is an envelope for various kinds of QF-Test daemons. Currently only the `TestRunDaemon` is available, but something like an `SUTClientStarterDaemon` is already planned and others may follow.

---

void cleanup()

Clean up all TestRunDaemons belonging to this Daemon and then kill all clients. The default timeout of 30 seconds is used to wait for possible dependency rollback.

Throws

RemoteException	If something RMI specific goes wrong.

---

void cleanup(long timeout)

Clean up all TestRunDaemons belonging to this Daemon and then kill all clients.

Parameters

timeout	The maximum time in milliseconds to wait for possible dependency rollback.

Throws

RemoteException	If something RMI specific goes wrong.

`TestRunDaemon createTestRunDaemon()`

Get access to a TestRunDaemon.

Returns	A TestRunDaemon.
Throws	
`RemoteException`	If something RMI specific goes wrong.

`String getHost()`

Get the host of the Daemon.

Returns	The host of the Daemon.
Throws	
`RemoteException`	If something RMI specific goes wrong.

`String getIp()`

Get the ip of the Daemon.

Returns	The ip of the Daemon.
Throws	
`RemoteException`	If something RMI specific goes wrong.

`int getPort()`

Get the port of the Daemon.

Returns	The port of the Daemon.
Throws	
`RemoteException`	If something RMI specific goes wrong.

`TestRunDaemon getSharedTestRunDaemon()`

Get the shared TestRunDaemon.

Returns	The shared TestRunDaemon.
Throws	
`RemoteException`	If something RMI specific goes wrong.

`TestRunDaemon[] getTestRunDaemons()`

Get all TestRunDaemons created by the Daemon that are still live.

Returns	The set of live TestRunDaemons created by the Daemon. Does not include the shared TestRunDaemon.
Throws	
`RemoteException`	If something RMI specific goes wrong.

`void killClients()`

Kill all clients that belong to the VM of the daemon.

Throws	
`RemoteException`	If something RMI specific goes wrong.

---

void ping()

Test whether the Daemon is still alive.

Throws

RemoteException	If something RMI specific goes wrong.

---

void terminate(int exitCode)

Terminate the daemon process by calling System.exit.

Parameters

exitCode	The exit code for the daemon.

Throws

RemoteException	If something RMI specific goes wrong.

53.2.3 The **TestRunDaemon**

The `de.qfs.apps.qftest.daemon.TestRunDaemon` is the outer interface for test execution. It is used to define the environment for associated test runs and create the `DaemonRunContext` instances that handle the actual test execution.

Miscellaneous

Daemon getDaemon()

Get the Daemon to which the TestRunDaemon belongs.

Returns	The Daemon of the TestRunDaemon.
Throws	
RemoteException	If something RMI specific goes wrong.

Global variable handling

The TestRunDaemon has its own set of global variables which are used to initialize the globals when creating a new DaemonRunContext. The following methods have no effect on already running DaemonRunContext instances.

---

void clearGlobals()

Clear all global variables.

Throws

RemoteException	If something RMI specific goes wrong.

String getGlobal(String name)

Get a global variable value.

Parameters

name	The name of the global variable.
Returns	The value of the global variable or null if undefined.
Throws	
RemoteException	If something RMI specific goes wrong.

Properties getGlobals()

Get all global variable values.

Returns	All global variable values.
Throws	
RemoteException	If something RMI specific goes wrong.

void setGlobal(String name, String value)

Set a global variable value.

Parameters

name	The name of the global variable.
value	The value of the global variable.
Throws	
RemoteException	If something RMI specific goes wrong.

Test execution

void cleanup()

Clean up and release all contexts belonging to this TestRunDaemon. The default time-out of 30 seconds is used to wait for possible dependency rollback.

Throws	
RemoteException	If something RMI specific goes wrong.

void cleanup(long timeout)

Clean up and release all contexts belonging to this TestRunDaemon.

Parameters

timeout	The maximum time in milliseconds to wait for possible dependency rollback.
Throws	
RemoteException	If something RMI specific goes wrong.

`DaemonRunContext createContext()`

Create a single daemon run context. Needs to acquire a license.

Returns	The context or null if no license can be acquired.
Throws	
`RemoteException`	If something RMI specific goes wrong.

`DaemonRunContext[] createContexts(int threads)`

Create daemon run contexts for multiple threads. Needs to acquire one license per thread.

Parameters	
`threads`	The number of threads for the contexts.
Returns	The contexts or null if not enough licenses can be acquired.
Throws	
`RemoteException`	If something RMI specific goes wrong.

`DaemonRunContext[] getContexts()`

Get all DaemonRunContexts created by the TestRunDaemon that are still live and have not been released.

Returns	The set of live DaemonRunContexts created by the TestRunDaemon. Does not include the shared DaemonRunContext.
Throws	
`RemoteException`	If something RMI specific goes wrong.

`DaemonRunContext getSharedContext()`

Get the shared daemon run context. Needs to acquire a license if the shared context must be created first.

Returns	The shared context or null if no license can be acquired.
Throws	
`RemoteException`	If something RMI specific goes wrong.

`void setRootDirectory(String directory)`

Set the test suite root directory for new created daemon run contexts.

Parameters	
`directory`	The new root directory.
Throws	
`RemoteException`	If something RMI specific goes wrong.

Identification

String getIdentifier()

Get the identifier for the TestRunDaemon. If no identifier was previously set with setIdentifier(), a default identifier is created from the name of the Daemon, to which the TestRunDaemon belongs, and a counter.

Returns	The identifier for the TestRunDaemon.

Throws

RemoteException	If something RMI specific goes wrong.

---

void setIdentifier(String identifier)

Set an identifier for the TestRunDaemon. This can be useful in identifying a TestRunDaemon retrieved via Daemon.getTestRunDaemons().

Parameters

identifier	The identifier to set.

Throws

RemoteException	If something RMI specific goes wrong.

53.2.4 The `DaemonRunContext`

The `de.qfs.apps.qftest.daemon.DaemonRunContext` interface is in charge of the actual test execution.

The following run states are defined:

State	**Value**	**Description**
STATE_INVALID	−1	Invalid after release - cannot be reactivated.
STATE_IDLE	0	No run scheduled.
STATE_SCHEDULED	1	Run scheduled but not started.
STATE_RUNNING	2	Running.
STATE_PAUSED	3	Running but paused.
STATE_FINISHED	4	Run finished, result and run log available.

Table 53.1: The run state

6.0+ The following result codes for the `getResult()` method are the same as everywhere in QF-Test:

Result	Value	Description
RESULT_OK	0	Run OK, no warnings, errors or exceptions.
RESULT_WARNING	1	Run mostly OK, some warnings but no errors or exceptions.
RESULT_ERROR	2	Run failed with errors but no exceptions.
RESULT_EXCEPTION	3	Run failed with an exception.

Table 53.2: The result codes

---

void addTestRunListener(DaemonTestRunListener listener, boolean synchronous, long timeout)

Add a DaemonTestRunListener to the DaemonRunContext.

Parameters

listener	The listener to add.
synchronous	Whether the listener should get notified synchronously, in which case the test run will be blocked while the listener is processing the event.
timeout	Timeout in milliseconds for callbacks to the listener. If the listener does not reply within that time the listener is automatically unregistered to prevent further problems. In case of a synchronous listener the test run will then continue. A value of 0 means no timeout which is dangerous but may be useful.

---

boolean callProcedure(String procedure, Properties bindings=None)

Call a procedure in the run context.

Parameters

procedure	The procedure to run, of the form Suite#Procedure where Procedure is the fully qualified name of the Procedure.
bindings	An optional set of variable bindings. These variables have higher precedence than the globals or any bindings on the fallback stack.
Returns	True if the procedure call was started, false if suite or procedure could not be found.

Throws

RemoteException	If something RMI specific goes wrong.
IllegalStateException	If no run can be started in the current state, i.e. if the state is neither STATE_IDLE nor STATE_FINISHED.

void clearGlobals()

Clear all global variables of the DaemonRunContext.

Throws

RemoteException	If something RMI specific goes wrong.

void clearTestRunListeners()

Remove all DaemonTestRunListeners from the DaemonRunContext.

String getGlobal(String name)

Retrieve the value of a global variable from the DaemonRunContext.

Parameters

name	The name of the variable.
Returns	The value of the variable or null if undefined.

Throws

RemoteException	If something RMI specific goes wrong.

Properties getGlobals()

Retrieve all global variables from the DaemonRunContext.

Returns	The global variables.

Throws

RemoteException	If something RMI specific goes wrong.

String getIdentifier()

Get the identifier for the DaemonRunContext. If no identifier was previously set with setIdentifier(), a default identifier is created from the name of the TestRunDaemon, to which the DaemonRunContext belongs, and a counter.

Returns	The identifier for the DaemonRunContext.

Throws

RemoteException	If something RMI specific goes wrong.

String getLastTest()

Get the name of the test that is currently running or was last run on this DaemonRun-Context.

Returns	The name of the currently running or most recently executed test.

Throws

RemoteException	If something RMI specific goes wrong.

`int getNumThreads()`

Get the number of threads in the group to which the DaemonRunContext belongs.

Returns	The number of threads of the DaemonRunContext's group.
Throws	
`RemoteException`	If something RMI specific goes wrong.

`Properties getProperties(String group)`

Retrieve all properties from a property or resource group from the DaemonRunContext.

Parameters	
`name`	The name of the property or resource group.
Returns	The properties or null if no such group exists.
Throws	
`RemoteException`	If something RMI specific goes wrong.

`String getProperty(String group, String name)`

Retrieve the value of a property or resource from the DaemonRunContext.

Parameters	
`name`	The name of the property or resource group.
`name`	The name of the property.
Returns	The value of the property or null if undefined.
Throws	
`RemoteException`	If something RMI specific goes wrong.

`int getResult()`

Get the result of the test run.

Returns	The result of the run, one of RESULT_OK, RESULT_WARNING, RESULT_ERROR or RESULT_EXCEPTION.
Throws	
`RemoteException`	If something RMI specific goes wrong.
`IllegalStateException`	If the state isn't STATE_FINISHED.

`byte[] getRunLog()`

Get the run log of the test run.

Returns	The run log dumped into a byte array.
Throws	
`RemoteException`	If something RMI specific goes wrong.
`IllegalStateException`	If the state isn't STATE_FINISHED.

`int getRunState()`

Get the current run state of the context.

Returns	The current run state, one of STATE_IDLE, STATE_SCHEDULED, STATE_RUNNING, STATE_PAUSED or STATE_FINISHED.
Throws	
`RemoteException`	If something RMI specific goes wrong.

`TestRunDaemon getTestRunDaemon()`

Get the TestRunDaemon to which the DaemonRunContext belongs.

Returns	The TestRunDaemon of the DaemonRunContext.
Throws	
`RemoteException`	If something RMI specific goes wrong.

`int getThreadNum()`

Get the thread index of the DaemonRunContext.

Parameters

Returns	The thread index of the DaemonRunContext.
Throws	
`RemoteException`	If something RMI specific goes wrong.

`void release()`

Release the DaemonRunContext and return its license. If a test is running, stop it.

Throws	
`RemoteException`	If something RMI specific goes wrong.
`IllegalStateException`	If no DaemonRunContext was allocated.

`void removeTestRunListener(DaemonTestRunListener listener)`

Remove a DaemonTestRunListener from the DaemonRunContext.

Parameters

`listener`	The listener to remove.

`void rollbackDependencies()`

Roll back the dependencies for this DaemonRunContext.

Throws	
`RemoteException`	If something RMI specific goes wrong.
`IllegalStateException`	If no run can be started in the current state, i.e. if the state is neither STATE_IDLE nor STATE_FINISHED.

---

boolean runTest(String test, Properties bindings=None)

Run a test in the run context.

Parameters

test	The test to run, of the form `Suite#Test` where `#Test` is optional and `Test` is the fully qualified name of a Test set or Test case or just ".". The latter is equivalent to specifying just `Suite` and causes the whole test suite to be executed. Examples:

`MySuite`	Runs whole test suite `MySuite`.
`MySuite#.`	Runs whole test suite `MySuite`.
`MySuite#MyTestSet`	Runs test set `MyTestSet` in test suite `MySuite`.
`MySuite#MyTestSet.MyTestCase`	Runs test case `MyTestCase` located in test set `MyTestSet` in test suite `MySuite`.

bindings	An optional set of variable bindings. These variables have higher precedence than the globals or any bindings on the fallback stack.
Returns	True if the test was started, false if suite or test could not be found.

Throws

RemoteException	If something RMI specific goes wrong.
IllegalStateException	If no run can be started in the current state, i.e. if the state is neither STATE_IDLE nor STATE_FINISHED.

---

void setGlobals(Properties globals)

Set the global variables of the DaemonRunContext.

Parameters

globals	The global variables to set.

Throws

RemoteException	If something RMI specific goes wrong.

---

void setIdentifier(String identifier)

Set an identifier for the DaemonRunContext. This can be useful in identifying a DaemonRunContext retrieved via TestRunDaemon.getContexts().

Parameters

identifier   The identifier to set.

Throws

RemoteException   If something RMI specific goes wrong.

---

void setRootDirectory(String directory)

Set the test suite root directory for the next test run.

Parameters

directory   The new root directory.

Throws

RemoteException   If something RMI specific goes wrong.

---

void stopRun()

Stop the test run.

Throws

RemoteException   If something RMI specific goes wrong.

IllegalStateException If no run was scheduled.

---

boolean waitForRunState(int state, long timeout)

Wait for the context to reach a given state.

Parameters

state   The state to wait for.

timeout   Maximum time in milliseconds to wait.

Returns   True if the state was reached, false if timed out.

Throws

RemoteException   If something RMI specific goes wrong.

53.2.5 The `DaemonTestRunListener`

The   `de.qfs.apps.qftest.daemon.DaemonTestRunListener`
interface   is   identical   to   the   interface
`de.qfs.apps.qftest.extensions.qftest.TestRunListener` described in
section 52.7[(1087)], except that its methods can throw a `RemoteException` on RMI
failure.   When implementing this interface you must derive your class from
`java.rmi.server.UnicastRemoteObject`.

You can register the listener with a `DaemonRunContext` via its `addTestRunListener`

method described in the previous section.

53.3 Daemon security considerations

Anybody with access to the QF-Test daemon can start any program on the daemon machine with the rights of the user account that the daemon is running under, so care should be taken to only allow trusted users to connect to the daemon.

Of course the QF-Test daemon should always be run on a machine that is protected from outside access by a firewall. If all users that can access the machine behind the firewall are trusted, that is sufficient. If the set of trusted users needs to be limited further, please read on.

By default the QF-Test daemon uses SSL to secure the RMI connection. However, unless you take additional precautions, this only means that the network traffic between the daemon and its client is encrypted. To restrict access to certain users, one further step is required.

Setting up SSL communication can be very complex. One usually has to learn about keys, certificates, certificate authorities, chains of trust etc. Fortunately, this is a very special case, and the fact that once a user has access to the QF-Test daemon he also has control over the daemon's machine means that there is no distinction between the daemon administrator and a daemon user. Without going into details, QF-Test normally uses a single keystore file with a single self-signed certificate on both daemon and client side. More complex scenarios are possible but beyond the scope of this manual. The default keystore file is named `daemon.keystore` and provided in QF-Test's system directory or the version-specific directory. By creating your own keystore as described below you can ensure that only users that have read access to that keystore file can interact with the daemon.

53.3.1 Creating your own keystore

To create the keystore file you need a current JDK version 1.5 or higher, a JRE is not sufficient. In a shell or console window, execute the following command (you may need to prepend the path to the `keytool` program which resides in the JDK's `bin` directory):

```
keytool -keystore daemon.keystore -genkey -alias "qftest daemon"
        -keyalg DSA -validity 999999
```

Example 53.1: Creating a keystore for secure daemon communication

For further information about the `keytool` command please see

http://download.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.htm.

When asked for the password for the keystore enter `123456`. When asked for your name or organization, feel free to provide answers or not. For QF-Test these don't make any difference. Of course you can provide a secure password instead of `123456`, but that will only complicate starting the daemon and its client and not contribute much to security. You could also use a shorter validity, but in case the keystore file gets into the wrong hands, all you need to do is set up your daemon and users with a new one, so the old one becomes useless.

53.3.2   Specifying the keystore

You have several options to tell QF-Test to use your keystore instead of the default one:

- Save the file as `daemon.keystore` in QF-Test's system directory

- Save the file as `daemon.keystore` in the user configuration directory[10].

- Save the file wherever you like and provide its location to QF-Test via the command line argument `-keystore <keystore file>`[873].

In case you specified your own password for the keystore you also need to specify that via the command line argument `-keypass <keystore password>`[873].

In case you would like to start the daemon without any SSL support, for example to interact with a QF-Test version older than 3.5, either remove the file called `daemon.keystore` from QF-Test's version specific directory or use the command line argument in the form `-keystore=` to specify no keystore.

53.3.3   Specifying the keystore on the client side

If you use `qftest -batch -calldaemon` to access the daemon or script nodes from within QF-Test, the options for the client are the same as for the daemon.

To access the daemon over SSL from your own code via the daemon API you must set the System properties `javax.net.ssl.keyStore` and `javax.net.ssl.trustStore` to the keystore file and `javax.net.ssl.keyStorePassword` to the password for the keystore file. See section 53.2[1140] for details about the daemon API and section 23.2[286] for examples.

Chapter 54

The Procedure Builder definition file

For general information about the Procedure Builder, please see chapter chapter 25[(307)].

54.1 Placeholders

In the definition of procedures, packages and even in the nodes of the procedures, you can use placeholders. The following table shows all placeholders and their meaning:

Placeholder	Description
<COMPID>	The QF-Test ID of the component
<COMPNAME>	The name of the component
<COMPFEATURE>	The feature of the component
<COMPCLASS>	The recorded class of the component
<COMPTKCLASS>	The toolkit specific class of the component
<COMPSYSCLASS>	The system class of the component
<COMPGENCLASS>	The generic class of the component
<COMPEF-name-of-extra-feature>	The value of the given extra feature
<CURRENTVALUE>	The current value of the component, e.g. the text of a text-field or the current selection of a combo-box etc.
<CURRENTENABLEDSTATE>	The current enabled state of the component
<CURRENTSELECTEDSTATE>	The current selectable state of the component
<CURRENTEDITABLESTATE>	The current editable state of the component
<PCOMPID>	The QF-Test ID of the parent component
<PCOMPNAME>	The name of the parent component
<PCOMPFEATURE>	The feature of the parent component
<PCOMPCLASS>	The class of the parent component
<PCOMPEF-name-of-extrafeature>	The value of the given extra feature of the parent component
<GPCOMPID>	The QF-Test ID of the grandparent component
<GPCOMPNAME>	The name of the grandparent component
<GPCOMPFEATURE>	The feature of the grandparent component
<GPCOMPCLASS>	The class of the grandparent component
<GPCOMPEF-name-of-extrafeature>	The value of the given extra feature of the grandparent component
<ENGINE>	The engine's name, either 'awt', 'swt', 'web' or 'fx'.
<ENGINE2>	The alternative engine's name, either 'swing', 'swt', 'web' or 'fx'.

Table 54.1: Placeholders for component procedures

For procedures of container or composite components you can also use placeholders for the child components. Please see following table for those additional variables:

Variable	Description
<CCOMPID>	The QF-Test ID of the child component
<CCOMPNAME>	The name of the child component
<CCOMPFEATURE>	The feature of the child component
<CCOMPCLASS>	The recorded class of the child component
<CCOMPTKCLASS>	The toolkit class of the child component
<CCOMPSYSCLASS>	The system class of the child component
<CCOMPGENCLASS>	The generic class of the child component
<CCOMPEF-name-of-extrafeature>	The value of the given extra feature
<CCURRENTVALUE>	The current value of the child component, e.g. the text of a text-field or the current selection of a combo-box etc.
<CCURRENTENABLEDSTATE>	The current enabled state of the child component
<CCURRENTSELECTEDSTATE>	The current selectable state of the child component
<CCURRENTEDITABLESTATE>	The current editable state of the child component

Table 54.2: Additional placeholders for container procedures

54.1.1 Fallback values for placeholders

4.1.3+

In many projects procedure and parameter names will be generated using the placeholders `COMPNAME`, `COMPFEATURE` or `COMPEF-qfs:label`. Those placeholders are used because readable names or meaningful labels exist. But in some cases those placeholders cannot be filled by every component, e.g. a new implemented button has no meaningful name yet or QF-Test cannot determine a useful label for a textfield. In such cases you could consider implementing resolvers but those need to be stable and robust again. Instead you can specify so called fallback values for any placeholder now. The values of those fallback definition will be used whenever the actual placeholder has no value.

You can specify them in the 'Comment' attribute of the configured 'Package' and 'Procedure' nodes. Therefore, you need to write `@fallback_` and then the name of the placeholder. Afterwards you define the fallback placeholders. So, a placeholder for `COMPNAME` looks like `@fallback_COMPNAME COMPFEATURE`. You can see that the value of the placeholder `COMPFEATURE` will be taken into account, if no value for `COMPNAME` could be detected. You can also specify several fallback placeholders using a comma separated list like `@fallback_COMPEF-qfs:label COMPFEATURE,COMPNAME`.

54.2 Conditions for Package and Procedure Definition

You can influence the creation of packages and procedures via using the 'Comment' attribute of the 'Package' or 'Procedure' node.

Comment-Attribute	Description
@ABSOLUTECALL	This doctag can be used in procedures generating procedure calls via @FORCHILDREN when you do not want QF-Test to prefix the specified procedure call by `procbuilder`.
@CONDITION	Specifies a dedicated condition for creating packages or procedures. You can define conditions as Jython, Groovy, JavaScript or regular expression. Please see below for further details.
@EXCEPT	Specifies, whether the package or the procedure should be defined for a certain class or not. This construct might be useful, if you define a package or procedure for an abstract class and the procedures should not be created for all of its descendants.
@FORCECREATION	This doctag can be used in procedures generating procedure calls via @FORCHILDREN when you want QF-Test to create the procedure call once in the procedure, irrespective of child components.
@FORCHILDREN	When you set this doctag for Procedure nodes you can specify Procedure calls in the body of the Procedure where you prefix the Procedure name by the class name of the child components for which the Procedure call should be created. When generating the procedures, QF-Test will create the respective Procedure call for each component matching the class the Procedure call was prefixed with. Example: If you place a Procedure call with the Procedure name `TextField.setText` in a procedure flagged with @FORCHILDREN QF-Test will generate a Procedure node with Procedure calls to `setText` for each child component with the Class name `TextField`.
@NOTINHERIT	If this value is set, then this package or procedure is only defined for the exact class and not for its descendants.
@SUBITEM	This value is only valid for menu actions up-to-now. If this flag is set, then the according package or procedure will be created for sub-items on the second level of a menu only.
@SWTSTYLE	Evaluate the style attribute of a given SWT-button. This SWT specific attribute sometimes is needed because SWT distinguishes between check-boxes, buttons or radio-buttons only via the style attribute. For SWT-buttons you can define something like `@SWTSTYLE=PUSH` or `@SWTSTYLE=RADIO` for radio buttons.

Table 54.3: Comment attributes for procedure creation

54.3 Interpretation of the Component Hierarchy

It might be interesting to make use of the component-hierarchy in the package structure. This approach allows the tester to locate the component-specific procedures quite easy. If you want to create component-hierarchy packages, you can use two placeholders in the package-names:

Hierarchy-Placeholder	Description
<HIERARCHY>	Create packages for the full component-hierarchy by using the QF-Test component ID.
<HIERARCHY_NAME>	Create packages for the full component-hierarchy by using the name of the component. If a component in the hierarchy has no name, the component won't get taken into account.
<HIERARCHY_FEATURE>	Create packages for the full component-hierarchy by using the recorded feature of the component. If a component in the hierarchy has no feature, the component won't get taken into account.
<IHIERARCHY>	Create packages only for interesting components in the component-hierarchy by using the QF-Test component ID. An interesting component is a component with a feature.
<IHIERARCHY_NAME>	Create packages only for interesting components in the component-hierarchy by using the name of the component. An interesting component is a component with a feature.
<IHIERARCHY_FEATURE>	Create packages only for interesting components in the component-hierarchy by using the feature of the component. An interesting component is a component with a feature.
<MHIERARCHY>	Create packages only for menu components in the component-hierarchy by using the QF-Test component ID. It only takes components into account, which act as menu or menu item.
<MHIERARCHY_NAME>	Create packages only for menu components in the component-hierarchy by using the component-name. It only takes components into account, which act as menu or menu item. If no name is set, the component will be ignored.
<MHIERARCHY_FEATURE>	Create packages only for menu components in the component-hierarchy by using the component-feature. It only takes components into account, which act as menu or menu item. If no feature is set, the component will be ignored.

Table 54.4: Hierarchy placeholders

54.4 Details about the @CONDITION tag

3.4+

Using the `@CONDITION` tag allows you to configure, whether a dedicated node should be created or not during creation time.

Such conditions might be used to check a certain name or for appearance of a dedicated letter in the feature. If this condition is not fulfilled, the node won't be created. You can use all known placeholders, e.g. <COMPID> or <CCOMPNAME>.

Value of Condition	Meaning
@CONDITION jython "<COMPFEATURE>".startswith("abc")	Here we define a Jython condition, which will create the according node, if the feature of the current component starts with 'abc'. It is possible to use any string or comparing method of Jython.
@CONDITION groovy "<COMPFEATURE>".startsWith("abc")	Here we define a Groovy condition, which will create the according node, if the feature of the current component starts with 'abc'. It is possible to use any string or comparing method of Groovy.
@CONDITION javascript "<COMPFEATURE>".startsWith("abc")	Here we define a JavaScript condition, which will create the according node, if the feature of the current component starts with 'abc'. It is possible to use any string or comparing method of JavaScript.
@CONDITION regexp "<COMPFEATURE>" =∼ "abc.*"	Here we define a regular expression, which will create the according node, if the feature of the current component starts with 'abc'. It is possible to use all capabilities of Java regular expressions.
@CONDITION regexp "<COMPFEATURE>" !∼ "abc.*"	Here we define a regular expression condition, which will create the according node, if the feature of the current component does not start with 'abc'. It is possible to use all capabilities of Java regular expressions.

Table 54.5: Samples for the @CONDITION tag

If you need more than one row, you have to use a '\' at the end of the first row.

Chapter 55

The ManualStepDialog

The `ManualStepDialog` is a Java class delivered by QF-Test. If you want to make use of this dialog for your own tests, please see following sample script and API specification.

```
from de.qfs.apps.qftest import ManualStepDialog
#create the dialog and show it immediately
manualDialog = ManualStepDialog(None, "New Test Case Title", \
"Step Description", "Expected Test Result")
#did the test fail or succeed?
failOrSuccess = manualDialog.getResult()
#get the content of the received result
receivedResult = manualDialog.getReceivedResult()
#get the execution information, whether skipped or canceled
execInfo = manualDialog.getExecInfo()
```

Example 55.1: Example use of ManualStepDialog

55.1   The ManualStepDialog API

ManualStepDialog ManualStepDialog(Component parent, String title, String stepText, String expResult)

Create a new `ManualStepDialog`.

Parameters

parent	The parent component for the dialog.
title	The title of the dialog.
stepText	The text for the step description text-field.
expResult	The text for the expected result text-field.

`String getExecInfo()`

Get the execution information of the test step.

| **Returns** | A string containing the execution information. |

`String getReceivedResult()`

Get the received result of the test step.

| **Returns** | A string containing the received result. |

`String getResult()`

Get the result of the test step. Please, see chapter section 32.5[(391)] for all possible results.

| **Returns** | A string containing the result. |

`boolean isStatusCanceled()`

Test whether the status is CANCELED.

| **Returns** | True if the status is canceled, false otherwise. |

`boolean isStatusFailed()`

Test whether the status is FAILED.

| **Returns** | True if the status is failed, false otherwise. |

`boolean isStatusPassed()`

Test whether the status is PASSED.

| **Returns** | True if the status is passed, false otherwise. |

`boolean isStatusSkipped()`

Test whether the status is SKIPPED.

| **Returns** | True if the status is skipped, false otherwise. |

`void setExecInfo(String newExecInfo)`

Set the execution information of the test step.

Parameters

| **`newExecInfo`** | The execution information of the test step. |

`void setReceivedResult(String newRecResult)`

Set the received result of the test step.

Parameters

| **`newReceived Result`** | The received result of the test step. |

`void setResult(String newResult)`

Set the result of the test step. Please, see chapter section 32.5[(391)] for all possible results.

Parameters

| **`newResult`** | The result of the test step. |

Chapter 56

Details about transforming nodes

56.1   Introduction

The transformation mechanism allows you to turn a node into another type, e.g. a Sequence into a Procedure or a Test into a Test case. Such actions could be required for re-factoring purposes or simply to make test development for efficient.

You can transform nodes via a right mouse click and then selecting Transform node into and the desired type.

Note   QF-Test only shows transformations that are possible in the current context of the test suite, so sometimes you may not see all possible transformations.

56.2   Transformation with type changes

The following transformations also change the type of child nodes of the converted nodes:

1. Test set into Test case

   (a) Data driver into disabled Sequence

   (b) Test case into Test step

2. Test case into Test set

   (a) All children are packed into a new sub Test case node.

3. Test into Test set recursive

(a) If there are only Data driver and Test children, the Data driver is turned into a disabled Test.

(b) Otherwise all children are packed into a new sub Test case node.

4. Test into Test case

(a) Data driver into disabled Sequence

56.3   Additional transformations below the Extras node

The following transformations are allowed below the Extras node only:

56.3.1   Transformations without side-effects

1. Sequence into Cleanup

2. Sequence into Setup

3. Sequence into Test case

4. Cleanup into Procedure

5. Setup into Procedure

6. Procedure into Sequence

7. Procedure into Test case

56.3.2   Transformations with side-effects

The following transformations also change the type of some child nodes:

1. Test set into Package

(a) Cleanup into disabled Procedure

(b) Data driver into disabled Procedure

(c) Dependency reference into disabled Dependency containing the previous Dependency reference

(d) Setup into disabled Procedure

(e) Test into Procedure

(f) Test case into Procedure

(g) Test call into disabled Procedure containing the Test call

2. Test case into Procedure

   (a) Cleanup into disabled Sequence

   (b) Dependency into disabled Sequence

   (c) Dependency reference into disabled Sequence

   (d) Setup into disabled Sequence

3. Test into Procedure

   (a) Data driver into disabled Sequence

4. Test into Package

   (a) All child nodes are packed into a Procedure node.

5. Test into Sequence

   (a) Data driver into disabled Sequence

6. Package into Test set

   (a) Package into Test set

   (b) Procedure into Test case

Chapter 57

Details about the algorithm for image comparison

57.1   Introduction

The classic <u>Check image</u>[(729)] node is only minimally tolerant towards deviations. Using the default algorithm of comparing pixel by pixel it is not possible to check images that are generated in a not-quite-deterministic way or differ in size.

By using the attribute <u>Algorithm for image comparison</u>[(732)] it is possible to define a special algorithm which is tolerant towards certain image deviations. The attribute must start with the algorithm definition in the form algorithm=<algorithm> followed by all required parameters, separated by semicolons. Parameters may be defined in any order, and use of variables is possible, e.g.:

```
algorithm=<algo>;parameter1=value1;parameter2=value2;
expected=$(expected)
```

Since QF-Test 3.5.1 the definition does not need so start with algorithm=<algorithm> but can simply begin with <algorithm>.
It is also no longer necessary to define the parameter 'expected'. QF-Test uses a default value if it is not set. Please see below for more information.

A detailed description of the available algorithms and their parameters is provided in the following section. For illustration, explanations are based on their effects on the following image:

Figure 57.1: Original image

In the related run log (see section 7.1[117]) of a failed check you have the opportunity to analyze the results of the algorithm as well as the result of probability calculation.
If you activate the option Log successful advanced image checks[510] all tolerant image checks will be logged for further analysis.

57.2 Description of algorithms

57.2.1 Classic image check

Description

The classic - or default - image check compares the color value of every single expected and actual pixel. If at least one expected pixel differs from the actual pixel the check fails. The option Tolerance for checking images[469] defines the tolerance for comparing pixel values.

Purpose

This pixel based check is suitable if you expect an exact image match with minimal tolerances or any deviations. Whenever your application renders the component not fully deterministically, this algorithm is not suitable.

Example

The classic image check doesn't transform the image, thus the result looks identical to the original image.

Figure 57.2: Classic image check

The classic image check is used when the <u>Algorithm for image comparison</u>[732] attribute is empty.

57.2.2 Pixel-based identity check

Description
   This algorithm is similar to the classic algorithm, but accepts an amount of unexpected pixels. It splits every pixel in it's three sub-pixels red, green and blue. Afterwards it checks every actual color value against the expected color value. The final result is the amount of identical pixels divided by the total amount of pixels. The calculated result is checked against an expected value.

Purpose
   If your images are not rendered fully deterministic but you accept a certain percentage of unexpected pixels, this algorithm may be useful.
   But it is not suitable, if the actual images are used to have shifts or distortions.

Example
   The result image of the exemplary algorithm
   ```
   algorithm=identity;expected=0.95
   ```
   looks identical to the original image because this algorithm does not manipulate the image.

Figure 57.3: Pixel-based identity check

Parameters

algorithm=identity
> The 'Pixel based identity check' should be used.

expected (optional, but recommended)
> Defines which probability you expect.
> Valid values are between 0.0 and 1.0. If not defined, use 0.98.

resize (optional)
> Defines, if the actual image should be resized before calculation to match the size of the expected image.
> Valid values are "true" and "false".

find (optional)
> Defines an image-in-image search.
> A detailed description of this parameter can be found in .

57.2.3 Pixel-based similarity check

Description
> This algorithm splits every pixel in it's three sub-pixels red, green and blue. Afterwards it checks every actual color value against the expected color value to calculate a percental similarity. All percental deviations are added up and used for calculation. The final result is the average deviation over all color values and all pixels. The calculated result is checked against an expected value.

Purpose
> If your images are not rendered fully deterministic but you accept a certain deviation, this algorithm is a possible candidate for your purpose.
> If you accept deviations for some pixels, but the average deviation all over the image is small, this algorithm is also suitable.
> But it is not suitable, if the actual images are used to have shifts or distortions.

Example
> The result image of the exemplary algorithm
> ```
> algorithm=similarity;expected=0.95
> ```
> looks identical to the original image because this algorithm does not manipulate the image.

Figure 57.4: Pixel-based similarity check

Parameters

algorithm=similarity
> The 'Pixel based similarity check' should be used.

expected (optional, but recommended)
> Defines which probability you expect.
> Valid values are between 0.0 and 1.0. If not defined, use 0.98.

resize (optional)
> Defines, if the actual image should be resized before calculation to match the size of the expected image.
> Valid values are "true" and "false".

find (optional)
> Defines an image-in-image search.
> A detailed description of this parameter can be found in section 57.3.1[(1179)].

57.2.4 Block-based identity check

Description
> This algorithm partitions the image into quadratic blocks with a selectable size. The color value of each of these blocks is calculated as the average of the color values of the pixels the block contains. If the width or height of the image is not a multiple of the block size, the blocks at the right and bottom edge are cropped and weighted accordingly.
> The actual blocks are checked against the expected blocks. The final result is the amount of identical blocks divided by the total amount of blocks.

Purpose
> This algorithm's purpose is to check an image which only differs at some parts but is identical at the remaining parts.

Example

The exemplary algorithm
```
algorithm=block;size=10;expected=0.95
```
results in the following image:

Figure 57.5: Block-based identity check

Parameters

algorithm=block

The algorithm 'Block-based identity check' should be used.

size

Defines the size of each block.

Valid values are between 1 and the image size.

expected (optional, but recommended)

Defines the minimal match probability for the check to succeed.

Valid values are between 0.0 and 1.0. If not defined, use 0.98.

resize (optional)

Defines, if the actual image should be resized before calculation to match the size of the expected image.

Valid values are "true" and "false".

find (optional)

Defines an image-in-image search.

A detailed description of this parameter can be found in .

57.2.5 Block-based similarity check

Description

This algorithm also partitions the image in quadratic blocks with a selectable size. The color value of each of these blocks is calculated as the average of the

color values of the pixels the block contains. If the width or height of the image is not a multiple of the block size, the blocks at the right and bottom edge are cropped and weighted accordingly.

The color value of each expected block is checked against the actual block. Their color values are analyzed for percental similarity. The final result is the average similarity of all blocks with their weight taken into account.

Purpose

This algorithm is suitable for checking images with similar color variances.

Example

The exemplary algorithm

```
algorithm=blocksimilarity;size=5;expected=0.95
```

results in the following image:

Figure 57.6: Block-based similarity check

Parameters

algorithm=blocksimilarity

The algorithm 'Block-based similarity check' should be used.

size

Defines the size of each block.

Valid values are between 1 and the image size.

expected (optional, but recommended)

Defines the minimal match probability for the check to succeed.

Valid values are between 0.0 and 1.0. If not defined, use 0.98.

resize (optional)

Defines, if the actual image should be resized before calculation to match the size of the expected image.

Valid values are "true" and "false".

find (optional)

Defines an image-in-image search.

A detailed description of this parameter can be found in <u>section 57.3.1</u>[1179].

57.2.6 Histogram check

Description

To create a histogram, an image is first broken into its three base colors red, green and blue. Then the color values for each pixel are analyzed to partition them into a definable amount of categories (known as buckets when talking about histograms). The actual fill level of each bucket is compared to the expected level. The result of the algorithm is a comparison of the relative frequencies of color categories.

Purpose

Histograms are used for many scenarios. For example it is possible to check for color tendencies or do brightness analyses.

However, histograms are not suitable for checking rather plain-colored images.

Example

The exemplary algorithm

```
algorithm=histogram;buckets=64;expected=0.95
```

results in the following image:

Figure 57.7: Histogram

Parameters

algorithm=histogram

A 'Histogram' should be used for this image check.

buckets

Defines how many buckets to use.

Valid values are a power of 2 between 2 and 256.

expected (optional, but recommended)

> Defines the minimal match probability for the check to succeed.
> Valid values are between 0.0 and 1.0. If not defined, use 0.98.

resize (optional)

> Defines, if the actual image should be resized before calculation to match the size of the expected image.
> Valid values are "true" and "false".

find (optional)

> Defines an image-in-image search.
> A detailed description of this parameter can be found in .

57.2.7 Analysis with Discrete Cosine Transformation

Description

> The Discrete Cosine Transformation is a real-valued, discrete, linear, orthogonal transformation which transforms the discrete signal from local range to frequency range.
> After transforming an image, you can eliminate low-order (fast oscillating) frequencies. The remaining high-order (slow oscillating) frequencies with the steady component (zeroth frequency = 0*cos(x) + y) can now be analyzed. You can define how many frequencies per basic color should be used for this image check. You can also specify a tolerance to accept cosine-oscillations as identical which actually differ. Low-order frequencies get weighted less than high-order frequencies when calculating the result.

Purpose

> The Discrete Cosine Transformation is suitable for many kinds of image checks, which require certain tolerances. The more frequencies are used for analysis the sharper the image check is.

Example

> The exemplary algorithm
> `algorithm=dct;frequencies=20;tolerance=0.1;expected=0.95`
> results in the following image:

Figure 57.8: Analysis with Discrete Cosine Transformation

Parameters

algorithm=dct

'Analysis with Discrete Cosine Transformation' should be used for this image check.

frequencies

Defines how many frequencies to analyze.

Valid values are between 0 (steady component only) and the area of the image.

The less frequencies are analyzed the more tolerant the check is. The tolerance is also dependent on the size of the image.

tolerance

Defines the (non-linear) tolerance for accepting different cosine-oscillations as identical.

Valid values are between 0.0 and 1.0.

The value 1.0 means every image matches every other image, because the maximum difference of each frequency is tolerated. A value of 0.0 means frequencies only match if they are exactly the same. A value of 0.1 is a good starting point because only quite similar frequencies are accepted as identical.

expected (optional, but recommended)

Defines the minimal match probability for the check to succeed.

Valid values are between 0.0 and 1.0. If not defined, use 0.98.

resize (optional)

Defines, if the actual image should be resized before calculation to match the size of the expected image.

Valid values are "true" and "false".

find (optional)

Defines an image-in-image search.

A detailed description of this parameter can be found in section 57.3.1[1179].

57.2.8 Block-based analysis with Discrete Cosine Transformation

Description

When using this algorithm the image is first partitioned into quadratic blocks with a selectable size (see section 57.2.4[1171]). Afterwards every partition is analyzed using a Discrete Cosine Transformation (see section 57.2.7[1175]). The final result is the average of all results of these Discrete Cosine Transformations with consideration of the blocks and their weight.

Purpose

The Discrete Cosine Transformation used on the whole image deviates strongly in case of significant brightness differences occurring in the middle of the image because then the steady component (zeroth frequency), which is the highest weighted part, varies strongly. The partitioning circumvents this behavior because now only the affected partitions result in intense deviations while the other partitions stay untouched.

Example

The exemplary algorithm
```
algorithm=dctblock;size=32;frequencies=4;tolerance=0.1;
expected=0.95
```
results in the following image:

Figure 57.9: Block-based analysis with Discrete Cosine Transformation

Parameters

algorithm=dctblock

'Blocks for analysis with Discrete Cosine Transformation' should be used for this image check.

size

Defines the size of each block.
Valid values are between 1 and the image size.

frequencies

Defines how many frequencies to analyze.

Valid values are between 0 (steady component only) and the area of a block. The less frequencies are analyzed the more tolerant the check is. The tolerance is also dependent on the size of the image.

tolerance

Defines the (non-linear) tolerance for accepting different cosine-oscillations as identical.

Valid values are between 0.0 and 1.0.

The value 1.0 means every image matches every other image, because the maximum difference of each frequency is tolerated. A value of 0.0 means frequencies only match if they are exactly the same. A value of 0.1 is a good starting point because only quite similar frequencies are accepted as identical.

expected (optional, but recommended)

Defines the minimal match probability for the check to succeed.

Valid values are between 0.0 and 1.0. If not defined, use 0.98.

resize (optional)

Defines, if the actual image should be resized before calculation to match the size of the expected image.

Valid values are "true" and "false".

find (optional)

Defines an image-in-image search.

A detailed description of this parameter can be found in <u>section 57.3.1</u>[1179].

57.2.9 Bilinear Filter

Description

This algorithm shrinks the image to a chooseable percental size. Afterwards the image gets resized to its original size by use of a bilinear filter. This filter effects a blurring, because every color value is calculated by use of neighbor pixels.

The final result is the average deviation over all color values and all pixels of this transformed images.

Purpose

Depending on the chosen sharpness the images loose any desired image information. Thus this algorithm is valuable for nearly any scenario.

Example

The exemplary algorithm

```
algorithm=bilinear;sharpness=0.2;expected=0.95
```

results in the following image:

Figure 57.10: Bilinear Filter

Parameters

algorithm=bilinear

A 'Bilinear Filter' should be used for this image check.

sharpness

Defines the sharpness of this bilinear filter.
Valid values are between 0.0 (complete loss of information) and 1.0 (no loss of information).
The sharpness is a linear parameter. This means a value of 0.5 eliminates exactly half (plus minus rounding to entire pixels) of information.

expected (optional, but recommended)

Defines the minimal match probability for the check to succeed.
Valid values are between 0.0 and 1.0. If not defined, use 0.98.

resize (optional)

Defines, if the actual image should be resized before calculation to match the size of the expected image.
Valid values are "true" and "false".

find (optional)

Defines an image-in-image search.
A detailed description of this parameter can be found in .

57.3 Description of special functions

57.3.1 Image-in-image search

Description

The image-in-image search allows to find an expected image within a (larger) image. The check is successful when the expected image can be found

anywhere using the defined algorithm. Furthermore, you can determine the position of the match.

The following combination of parameters are valid: `find=best` or `find=anywhere`

`find=best(resultX, resultY)` **or** `find=anywhere(resultX, resultY)`

Purpose

The image-in-image search allows to compare images if you don't know the exact position and thus cannot define an offset. The search can be combined with any algorithm and is thus valuable for any purpose.

Example

The exemplary algorithm

`algorithm=similarity;expected=0.95;find=best(resultX,resultY)`
uses pixel-based similarity check (see section 57.2.3[1170]) to find an image of Q as part of the full image. The got image with highlighted region can be found within the run log. Besides, the variables resultX and resultY are set to the location of the found image.

Figure 57.11: Image-in-image search: Expected image

Figure 57.12: Image-in-image search: Got image

Parameters

find=best
Defines that the best match should be used.

find=anyhwere

Defines that the first match which exceeds the expected match probability should be used.

The image-in-image search uses multiple threads and thus finding anywhere is non-deterministic.

resultX

`resultX` is the name of a QF-Test variable which holds the x-position of the found image.

If a variable for the x-position is defined, a variable for the y-position has to be defined as well (see syntax above).

resultY

`resultY` is the name of a QF-Test variable which holds the y-position of the found image. If a variable for the y-position is defined, a variable for the x-position has to be defined as well (see syntax above).

Chapter 58

Result lists

58.1 Introduction

Search operations like locating a reference or searching for a particular value can produce large result sets, as can operations like renaming components or procedures. In order to provide a better overview of all affected nodes QF-Test shows a mass result list at the end of such operations. This list holds all nodes that have been touched by the respective operation. Besides providing an overview that list allows you to apply mass operations on all nodes quite easily. Such mass operations could be toggling a mark at all found nodes or deleting all of them from the test suites. Those operations can be triggered by the ‖Edit‖ menu or a the table's context menu.

Figure 58.1: Sample result list for 'Locate references'

The table comes with a context menu as well. This context menu allows you to perform several actions on the shown nodes. If you select entries in the result list and perform a right mouse-click you can raise some very useful actions from that menu like jumping to the node or disabling it. Depending on the type of the list there might be specific actions which only make sense in that particular context. A list of possible actions is provided in section 58.2⁽¹¹⁸⁴⁾.

Such result lists are available for the following actions:

- Once you press *Show result list* on the Search or Replace dialog.

- As result of the Locate reference action for callable nodes or components.

- As result of any command which updates several nodes.

- As result of any replacement action from the Replace dialog.

- As result of several analyzing actions under the $\boxed{\text{Additional node operations}}$ menu.

- Once you open a list of breakpoints under $\boxed{\text{Debugger}\rightarrow\text{List of all breakpoints...}}$.

- Once a duplicate QF-Test component ID gets inserted or a suite with duplicate components get opened.

- Once a node transformation changes the type of some child nodes.

- Once you press $\boxed{\text{Edit}\rightarrow\text{Open error list}}$ in the run log.

- If you update components via $\boxed{\text{Update components}}$ all errors will be shown in such a list.

If you have lots of entries in the table you can also apply a filter at the top for limiting the nodes to the given filter values. Once you reset the filter you will see all nodes again.

58.2 Specific list actions

58.2.1 All types of lists

Following actions are available on all types of lists:

- Toggle mark

- Toggle disabled state

- Jump to node in test suite

- Set a breakpoint

- Delete a node from the list only

- Delete a node from test suite

- Store the result in a `.qcv` file, for details see section 58.3[(1185)]

- If the according action affects only one node you can also jump back to this.

58.2.2 Replacing

The replacement list provides the capability of replacing only the selected values.

58.2.3   Error list

The error list of the run log provides capabilities to update failing checks in a bunch. It doesn't allow to delete a node from the run log of course.

58.3   Exporting and loading results

3.5+

In case you have a very large result list that you need to work on within more than one session, you can store its results in an external file using the menu action ⟦Extras→Export⟧. The export process creates a `.qcv` file, which is more or less a CSV file holding the information in a QF-Test internal format.   To proceed with your work at a later time, simply load the previously exported `.qcv` file via ⟦Operations→Load result list...⟧ into QF-Test.

Chapter 59

Generic classes

QF-Test abstract recorded class from the framework specific classes in order to get classes of common use. Those classes are called generic classes. This concept provides a better readability and clearer understanding of components. Furthermore already created tests can be re-used once the target technology is switched or if you want to maintain tests in various technologies in parallel.

Beside those generic classes QF-Test records generic types as well. Those types give a more detailed specification of the target component. A typical use case are password fields. Those fields have the generic class `TextField`, but they have that specific characteristic to enter passwords, so they get an additional generic type `TextField:PasswordField`. Using those types makes the recognition of generic classes for certain categories stricter and more appropriate.

A great advantage of these generic classes, especially when testing web applications is that a user can freely assign these generic classes to components. Later on these mappings may get reduced onto other generic classes. An example how this can be done may be found in the manual chapter $\underline{\text{CustomWebResolver} - \text{Tables}}^{(961)}$.

Using generic classes has the following advantages:

- It is possible to record additional component recognition features, depending on the generic class. Often, depending on the generic class, it makes sense to change the recorded Feature and/or the recorded Extra features, e.g. the `qfs:label`.

- Depending on the generic class, class specific checks may be provided, e.g. the check to check a complete table row if the generic class equals `Table`.

- The indexing of sub-elements during recording, this means that e.g. during click recording on a table cell only a Component for the table gets recorded, while the exact cell is referenced via indices.

- The recording of the generic type, as far as reasonable.

- With generic classes it becomes decidable whether the exact position or the most suitable position should get record for mouse clicks.

- Simply by assigning a recorded component to a generic class, component recognition gets sharpened compared to non-specific HTML classes.

The following sections list which information is saved in particular.

59.1 Accordion

Can be used as navigator between components. Components can be expanded and collapsed.

For the HTML mapping of accordions please refer to section 49.1.8[(973)].

Kind: Component

Coordinates for mouse click: Sub-items or exact co-ordinates

Feature: None; for web components see Feature for web components[(65)]

qfs:label*: Associated label, Label close to it, Tooltip

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
All items	All items of the accordion	items	All
Selected item	Currently selected item	current_item	All
All items with selection	All items of the accordion including their selection state	items_with_selection	At the moment not in web

Table 59.1: Checktypes for Accordion

59.2 BusyPane

Is drawn over other components to lock them.

Kind: Component SmartID: Class must be included

Coordinates for mouse click: Exact co-ordinates

Feature: None; for web components see Feature for web components[(65)]

qfs:label*: None

qfs:type: None

Additional checks: None

59.3   Button

Raises an action, once it is clicked. It normally doesn't have a defined state.

Kind: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text, tooltip; for web components see Feature for web components[(65)]

qfs:label*: Associated label, Own text, Tooltip, Description of icon, Label close to it

qfs:type:

qfs:type	Description
Button:CalendarButton	Button inside a Calendar
Button:ComboBoxButton	Button inside a ComboBox
Button:PaginatorButton	Button to switch pages like in aPaginator
Button:ScrollBarButton	Button to modify the current scrolling value

Table 59.2: Special qfs:type values for Buttons

Additional checks: None

59.4   Calendar

Can be used to select a date or time.

Kind: Component

Coordinates for mouse click: Exact co-ordinates

Feature: None; for web components see Feature for web components[(65)]

qfs:label*: Associated label, Label close to it, Tooltip

qfs:type: None

Additional checks: None

59.5 CheckBox

Has a dedicated state. Usually you can check and un-check those components.

Kind: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip; for web components see Feature for web components[(65)]

qfs:label*: Associated label, Own text, Label close to it, Tooltip, Description of icon

qfs:type:

qfs:type	Description
CheckBox:ComboListItemCheckBox	CheckBox inside a ComboListItem
CheckBox:ListItemCheckBox	CheckBox inside a ListItem
CheckBox:MenuItemCheckBox	CheckBox inside a MenuItem
CheckBox:TableCellCheckBox	CheckBox inside a TableCell
CheckBox:TreeNodeCheckBox	CheckBox inside a TreeNode

Table 59.3: Special qfs:type values for CheckBoxes

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Checked	Check the current selection of that checkbox.	checked	All

Table 59.4: Checktypes for Checkbox

59.6 Closer

Closes a component, e.g. the 'X' button of a window.

Kind: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip; for web components see Feature for web components[(65)]

qfs:label*: Associated label, Own text, Label close to it, Tooltip, Description of icon

qfs:type:

qfs:type	Description
Closer:AccordionCloser	Closer for accordion items
Closer:TabPanelCloser	Closer for tabs of a TabPanel
Closer:WindowCloser	Closer for window components

Table 59.5: Special qfs:type values for Closer

Additional checks: None

59.7 ColorPicker

Can be used to select a certain color.

Kind: Component

Coordinates for mouse click: Exact co-ordinates

Feature: None; for web components see Feature for web components[65]

qfs:label*: Associated label, Label close to it, Tooltip

qfs:type: None

Additional checks: None

59.8 ComboBox

Consists of a textfield and a list of selectable options.

Web   For the HTML mapping of a combobox please refer to section 49.1.7[971].

Kind: Component

Coordinates for mouse click: Exact co-ordinates

Feature: Associated label, tooltip; for web components see Feature for web components[65]

qfs:label*: Associated label, Label close to it, Tooltip, Prompt

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Current value	The current value of that combobox	value	All
Available values	All available vales of that combobox	value	All

Table 59.6: Checktypes for ComboBox

59.9 Divider

Splits areas of certain components. It can be moved.

Kind: Component SmartID: Class must be included

Coordinates for mouse click: Exact co-ordinates

Feature: None; for web components see Feature for web components[65]

qfs:label*: None

qfs:type: None

Additional checks: None

59.10 Expander

Is used to expand or collapse a component, e.g. for a node of a tree.

Kind: Not recorded SmartID: Class must be included

Coordinates for mouse click: No recorded

Feature: None; for web components see Feature for web components[65]

qfs:label*: None

qfs:type:

qfs:type	Description
Expander:TreeNodeExpander	Toggle of expanded/collapsed state of a tree node.

Table 59.7: Special qfs:type values for Expander

Additional checks: None

59.11   FileChooser

Can be used to select a file. Usually it consists of a textfield, a list of files and a button to choose.

Kind: Component SmartID: Class must be included

Coordinates for mouse click: Exact co-ordinates

Feature: None; for web components see Feature for web components[(65)]

qfs:label*: None

qfs:type: None

Additional checks: None

59.12   Graphics

Shows a graphics or a diagram. Clicks could raise an action, but that's not mandatory.

Kind: Component

Coordinates for mouse click: Exact co-ordinates

Feature: None; for web components see Feature for web components[(65)]

qfs:label*: Associated label, Label close to it, Tooltip, Description of icon

qfs:type: None

Additional checks: None

59.13   Icon

Shows an image. Clicks could raise an action, but that's not mandatory.

Kind: Component

Coordinates for mouse click: Exact co-ordinates

Feature: Own text or tooltip; for web components see Feature for web components[(65)]

qfs:label*: Associated label, Label close to it, Tooltip, Description of icon

qfs:type:

qfs:type	Description
Icon:ComboListItemIcon	Icon of a ComboListItem
Icon:IndicatorIcon	Icon of an Indicator
Icon:ListItemIcon	Icon of a ListItem
Icon:MenuItemIcon	Icon of a MenuItem
Icon:TableCellIcon	Icon of a TableCell
Icon:TreeNodeIcon	Icon of a TreeNode

Table 59.8: Special qfs:type values for Icon

Additional checks: None

59.14   Indicator

Shows a message after an input event. Typically they appear after entering something into textfields, could also contain an icon.

Kind: Component

Coordinates for mouse click: Exact co-ordinates

Feature: None; for web components see Feature for web components[65]

qfs:label*: Associated label, Own text, Label close to it, Tooltip, Description of icon

qfs:type:

qfs:type	Description
Indicator:ErrorIndicator	Shows an error
Indicator:InfoIndicator	Shows an information
Indicator:WarningIndicator	Shows a warning

Table 59.9: Special qfs:type values for Indicator

Additional checks: None

59.15   Item

Sub-item of a list, can be selectable.

Kind: Item or syntax SmartID: Class must be included

Coordinates for mouse click: Most appropriate position or center

Feature: None; for web components see Feature for web components[(65)]

qfs:label*: None

qfs:type:

qfs:type	Description
Item:AccordionItem	Selectable tab of an accordion
Item:ComboBoxListItem	Item of a list of a combobox
Item:ListItem	Item of a list
Item:TabPanelItem	Selectable tab of a TabPanel.

Table 59.10: Special qfs:type values for Item

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Item Text	Check for the shown text	item	All
Item visible	Check, whether item is visible	item_visible	All
The item's selected state	Check, whether item is selected	item_selected	All
The item's checked state	Check, whether item is checked	item_checked	All
Item image	Check for the image of that item	item_image	Not for web

Table 59.11: Checktypes for Item

59.16 Label

Shows some text, like a caption. Clicks could raise an action, but that's not necessary.

Kind: Component

Coordinates for mouse click: Exact co-ordinates

Feature: Own text or tooltip; for web components see Feature for web components[(65)]

qfs:label*: Associated label, Own text, Label close to it, Tooltip, Description of icon

qfs:type:

qfs:type	Description
Label:CalendarLabel	Inside a Calendar
Label:Caption	A caption of any figure or component
Label:PaginatorLabel	Shows the current page inside a paginator
Label:PanelTitle	A title of a panel
Label:WindowTitle	A title of a window

Table 59.12: Special qfs:type values for Labels

Additional checks: None

59.17   Link

Allows the user to navigate to another area of the application.

Kind: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip; for web components see Feature for web components[65]

qfs:label*: Associated label, Own text, Label close to it, Tooltip, Description of icon

qfs:type:

qfs:type	Description
Link:BreadcrumbLink	Special links for navigating inside a Breadcrumb navigation area.

Table 59.13: Special qfs:type values for Links

Additional checks: None

59.18   List

Shows multiple values. Values could be selectable as well.

Web For the HTML mapping of a list please refer to section 49.1.6[969].

Kind: Component

Coordinates for mouse click: Sub-items or exact co-ordinates

Feature: None; for web components see Feature for web components[65]

qfs:label*: Associated label, Label close to it, Tooltip

qfs:type:

qfs:type	Description
List:ComboBoxList	List of a ComboBox

Table 59.14: Special qfs:type values for List

Additional checks:

Name in Popup	Description	Name of checktype	Engine
All items	All items of the list	items	All
All items with selection	All items of the list including their selection state	items_with_selection	At the moment not in web
Selected item	Currently selected item	current_item	All

Table 59.15: Checktypes for List

59.19 LoadingComponent

Is used to show that your application is busy, e.g. if something is getting loaded by your application.

Kind: Component SmartID: Class must be included

Coordinates for mouse click: Exact co-ordinates

Feature: None; for web components see Feature for web components[65]

qfs:label*: None

qfs:type: None

Additional checks: None

59.20 Maximizer

Maximizes the size of a component, e.g. of a window.

Kind: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip; for web components see Feature for web components[65]

qfs:label*: Associated label, Label close to it, Tooltip, Description of icon

qfs:type:

qfs:type	Description
Maximizer:WindowMaximizer	Maximizer button for a window

Table 59.16: Special qfs:type values for Maximizer

Additional checks: None

59.21 Menu

Contains multiple menu items.

Kind: Component

Coordinates for mouse click: Exact co-ordinates

Feature: None; for web components see Feature for web components[65]

qfs:label*: None

qfs:type:

qfs:type	Description
Menu:MenuBar	A menu bar
Menu:PopupMenu	A menu is popping up and disappearing again

Table 59.17: Special qfs:type values for Menu

Additional checks: None

59.22 MenuItem

Is shown in menus. A click normally raises an action or changes the application's state.

Kind: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip; for web components see Feature for web components[65]

qfs:label*: Associated label, Own text, Label close to it, Tooltip, Description of icon

qfs:type: None

Additional checks: None

59.23 Minimizer

Minimizes the size of a component, e.g. of a window.

Kind: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip; for web components see Feature for web components[65]

qfs:label*: Associated label, Label close to it, Tooltip, Description of icon

qfs:type:

qfs:type	Description
Minimizer:WindowMinimizer	Minimizer button for window

Table 59.18: Special qfs:type values for Minimizer

Additional checks: None

59.24 Panel

Contains various components. Can be used to organize the UI.

Kind: Component SmartID: Class must be included except for Panel:TitledPanel

Coordinates for mouse click: Exact co-ordinates

Feature: Title, if existing; for web components see Feature for web components[65]

qfs:label*: Title

qfs:type:

qfs:type	Description
Panel:AccordionContent	Contains components of an Accordion.
Panel:Breadcrumb	Contains Breadcrumb-Links for quick navigation.
Panel:CollapsiblePanel	Can be expanded and collapsed.
Panel:Footer	Used to show a dedicated footer area.
Panel:Form	Used for entering values as a form.
Panel:Header	Used to show a dedicated header area.
Panel:Legend	Contains components which act as a legend of graphics.
Panel:MainPanel	Unique main panel of an application.
Panel:Paginator	Contains buttons to switch pages.
Panel:OptionGroup	Contains several RadioButtons.
Panel:ScrollPanel	Is scrollable and contains a ScrollBar.
Panel:TabPanelContent	Contains components of the selected TabPanelItem of a TabPanel.
Panel:TitledPanel	Has a dedicated title.

Table 59.19: Special qfs:type values for Panel

Additional checks: None

59.25 Popup

Shows components, which are only shown after a click on certain buttons and which belong to that button, e.g. the list after clicking on a button of a combobox.

Kind: Component SmartID: Class must be included

Coordinates for mouse click: Exact co-ordinates

Feature: None; for web components see Feature for web components[65]

qfs:label*: Title

qfs:type:

qfs:type	Description
Popup:CalendarPopup	Contains a Calendar
Popup:ColorPickerPopup	Contains a ColorPicker
Popup:ComboBoxPopup	Contains a list of a combobox

Table 59.20: Special qfs:type values for Popup

Additional checks: None

59.26 ProgressBar

Shows the current progress of an action.

Kind: Component

Coordinates for mouse click: Exact co-ordinates

Feature: Tooltip; for web components see Feature for web components[65]

qfs:label*: Associated label, Label close to it, Tooltip

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Value	The current value	value	All

Table 59.21: Checktypes for ProgressBar

59.27 RadioButton

Stands for a selectable option. It is typically used for selecting a state, if various states are allowed.

Kind: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip; for web components see Feature for web components[65]

qfs:label*: Associated label, Own text, Label close to it, Tooltip, Description of icon

qfs:type:

qfs:type	Description
RadioButton:ComboListItemRadioButton	RadioButton inside a ComboListItem
RadioButton:ListItemRadioButton	RadioButton inside a ListItem
RadioButton:MenuItemRadioButton	RadioButton inside a MenuItem
RadioButton:TableCellRadioButton	RadioButton inside a TableCell
RadioButton:TreeNoradioButton	RadioButton inside a TreeNode

Table 59.22: Special qfs:type values for RadioButtons

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Checked	Check the current selection of that radiobutton.	checked	All

Table 59.23: Checktypes for RadioButton

59.28   Restore

Re-creates the original size of a component, e.g. of a window.

Kind: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip; for web components see Feature for web components[65]

qfs:label*: Associated label, Label close to it, Tooltip, Description of icon

qfs:type:

qfs:type	Description
Restore:WindowRestore	Restore button for a window

Table 59.24: Special qfs:type values for Restore

Additional checks: None

59.29   ScrollBar

Is used for scrolling a component. Usually it contains a something like a thumb.

Kind: Component SmartID: Class must be included

Coordinates for mouse click: Exact co-ordinates

Feature: None; for web components see Feature for web components[65]

qfs:label*: None

qfs:type: None

Additional checks: None

59.30   Separator

Splits areas of certain components. It cannot be moved.

Kind: Component SmartID: Class must be included

Coordinates for mouse click: Exact co-ordinates

Feature: None; for web components see Feature for web components[65]

qfs:label*: None

qfs:type: None

Additional checks: None

59.31   Sizer

Modifies the size of a component, e.g. of a window.

Kind: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip; for web components see Feature for web components[65]

qfs:label*: Associated label, Label close to it, Tooltip, Description of icon

qfs:type:

qfs:type	Description
Sizer:WindowSizer	Sizer button of a window

Table 59.25: Special qfs:type values for Sizer

Additional checks: None

59.32   Slider

Allows the user to select a value via a thumb.

Kind: Component

Coordinates for mouse click: Exact co-ordinates

Feature: Tooltip; for web components see Feature for web components[65]

qfs:label*: Associated label, Label close to it, Tooltip

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Value	The current value	value	All

Table 59.26: Checktypes for Slider

59.33   Spacer

Acts as some kind of indentation, e.g. for nodes of a tree.

Kind: Not recorded SmartID: Class must be included

Coordinates for mouse click: Not recorded

Feature: None; for web components see Feature for web components[65]

qfs:label*: None

qfs:type:

qfs:type	Description
Spacer:TreeNodeSpacer	Spacer component of a TreeNode

Table 59.27: Special qfs:type values for Spacer

Additional checks: None

59.34 Spinner

Allows the user to select a value via arrow-buttons combined with a textfield.

Kind: Component

Coordinates for mouse click: Exact co-ordinates

Feature: Tooltip; for web components see Feature for web components[65]

qfs:label*: Associated label, Label close to it, Tooltip

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Value	The current value	value	All

Table 59.28: Checktypes for Spinner

59.35 SplitPanel

Shows components. Those components are located in resizable areas.

Kind: Component SmartID: Class must be included

Coordinates for mouse click: Exact co-ordinates

Feature: None; for web components see Feature for web components[65]

qfs:label*: None

qfs:type: None

Additional checks: None

59.36 Table

Shows multiple values. It has several dimensions columns. Values could be selectable as well.

Web For the HTML mapping of a table please refer to section 49.1.3[(961)].

Kind: Component

Coordinates for mouse click: Sub-items or exact co-ordinates

Feature: None; for web components see Feature for web components[(65)]

qfs:label*: Associated label, Label close to it, Tooltip

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Column	All values of a certain column	column	All
Column visible	Check, whether column is visible	column_visible	All
Column with selection	All values of a certain column including the cell's selected state	column_with_selection	All
Column title	Check for the column's title	header	All
Row	All values of a certain row	row	All

Table 59.29: Checktypes for Table

For the two check types `column` and `row` you can specify a subset of items to be checked. Please have a look at Check type identifier[(721)] for the respective syntax.

59.37 TableCell

A value of a table. It is identified by its column and its position in the row.

Kind: Item or syntax SmartID: Is referenced as table sub-item via syntax

Coordinates for mouse click: Most appropriate position or center

Feature: None; for web components see Feature for web components[(65)]

qfs:label*: None

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Cell	Text of that cell	item	All
The cell's visible state	Check whether table cell is visible.	item_visible	All
The cell's selected state	Check whether table cell is selected.	item_selected	All
The cell's editable state	Check whether table cell is editable.	item_editable	All
The cell's checked state	Check whether content inside the tablecell is checked, e.g a Check-Box.	item_checked	All
Cell image	Check for the image of that cell.	item_image	All

Table 59.30: Checktypes for TableCell

59.38 TableFooter

Footer row of a table.

Kind: Component SmartID: Class must be included

Coordinates for mouse click: Exact co-ordinates

Feature: None; for web components see Feature for web components[(65)]

qfs:label*: None

qfs:type: None

Additional checks: None

59.39 TableHeader

Header row of a table.

Kind: Component SmartID: Class must be included

Coordinates for mouse click: Sub-items or exact co-ordinates

Feature: None; for web components see Feature for web components[(65)]

qfs:label*: None

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
All items	All items of the table header	items	All

Table 59.31: Checktypes for TableHeader

59.40   TableHeaderCell

Name of a column.

Kind: Item or syntax SmartID: Is referenced as table sub-item via syntax

Coordinates for mouse click: Most appropriate position or center

Feature: None; for web components see Feature for web components[65]

qfs:label*: None

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Title	Text of that header cell	item	All
Title visible	Check whether header cell is visible.	item_visible	All
Title image	Check for the image of that header cell.	item_image	All

Table 59.32: Checktypes for TableHeaderCell

59.41   TableRow

Row of a table.

Kind: Not recorded SmartID: Not accessible as SmartID

Coordinates for mouse click: Sub-items or exact co-ordinates

Feature: None; for web components see Feature for web components[65]

qfs:label*: None

qfs:type: None

Additional checks: None

59.42   TabPanel

Can be used to navigate between components. Those components are shown in dedicated cards ("tabs"). Only one tab can be visible at once.

Web For the HTML mapping of a tab panel please refer to section 49.1.8[973].

Kind: Component

Coordinates for mouse click: Sub-items or exact co-ordinates

Feature: None; for web components see Feature for web components[65]

qfs:label*: Associated label, Label close to it, Tooltip

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
All tabs	All tabs of that panel	items	All
Selected tab	Currently selected tab	current_item	All

Table 59.33: Checktypes for TabPanel

59.43   Text

Shows common text with multiple rows. A user can't enter any value.

Kind: Component

Coordinates for mouse click: Exact co-ordinates

Feature: None; for web components see Feature for web components[65]

qfs:label*: Associated label, Label close to it, Tooltip, Prompt

qfs:type:

qfs:type	Description
Text:IndicatorText	Text of an Indicator
Text:ToolTipText	Text of a tooltip

Table 59.34: Special qfs:type values for Text

Additional checks: None

59.44 TextArea

Allows the user to enter text, even with multiple lines. It is able to show that text as well.

Kind: Component

Coordinates for mouse click: Sub-items or exact co-ordinates

Feature: Associated label, tooltip; for web components see Feature for web components[65]

qfs:label*: Associated label, Label close to it, Tooltip, Prompt

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
All lines	All lines of a TextArea	items	All
Editable	Check whether area could be modified	editable	All

Table 59.35: Checktypes for TextArea

59.45 TextField

Allows the user to enter single-line text. It is able to show that text as well.

Kind: Component

Coordinates for mouse click: Exact co-ordinates

Feature: Associated label, tooltip; for web components see Feature for web components[65]

qfs:label*: Associated label, Label close to it, Tooltip, Prompt

qfs:type:

qfs:type	Description
TextField:CalendarTextField	Inputfield of a Calendar
TextField:ComboBoxTextField	Inputfield of a ComboBox
TextField:PasswordField	Inputfields for passwords
TextField:SpinnerTextField	Inputfield of a Spinner

Table 59.36: Special qfs:type values for TextField

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Editable	Check whether textfield could be modified	editable	All

Table 59.37: Checktypes for TextField

59.46 Thumb

Can be used to thumb through certain values on a slider component.

Kind: Component SmartID: Class must be included

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip; for web components see Feature for web components[65]

qfs:label*: None

qfs:type: None

Additional checks: None

59.47 ToggleButton

Can be clicked like a button and has a certain state as well. Sometimes clicks also raise actions.

Kind: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip; for web components see Feature for web components[65]

qfs:label*: Associated label, Own text, Label close to it, Tooltip, Description of icon

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Checked	Check the current se-lection of that button.	checked	All

Table 59.38: Checktypes for ToggleButton

59.48 ToolBar

Stands for a toolbar. It typically contains several menu items and important buttons to raise most common actions.

Kind: Component SmartID: Class must be included

Coordinates for mouse click: Exact co-ordinates

Feature: None; for web components see Feature for web components[(65)]

qfs:label*: None

qfs:type: None

Additional checks: None

59.49 ToolBarItem

Stands for a clickable component inside a toolbar.

Kind: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip; for web components see Feature for web components[(65)]

qfs:label*: Associated label, Own text, Tooltip, Description of icon, Label close to it

qfs:type: None

Additional checks: None

59.50 ToolTip

A window, which gets opened as some kind of hint. Normally shown, if the user moves the mouse over a component.

Kind: Component

Coordinates for mouse click: Exact co-ordinates

Feature: Own text, tooltip; for web components see Feature for web components[65]

qfs:label*: Associated label, Own text, Label close to it, Tooltip

qfs:type: None

Additional checks: None

59.51 Tree

Shows content as tree. Can be used to show values in certain categories.

Web For the HTML mapping of a tree please refer to section 49.1.4[964].

Kind: Component

Coordinates for mouse click: Sub-items or exact co-ordinates

Feature: None; for web components see Feature for web components[65]

qfs:label*: Associated label, Label close to it, Tooltip

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
All visible nodes	All visible nodes in the tree	items	All
All visible nodes with selection	All visible nodes in the tree including their selection state	items_with_selection	All
All nodes with nesting	All nodes in the tree including their nesting level	nested_nodes	All
All visible nodes with nesting	All visible nodes of the tree including their nesting level	visible_nested_nodes	All

Table 59.39: Checktypes for Tree

59.52 TreeNode

Sub-item of a tree.

Kind: Item or syntax SmartID: Is referenced as table sub-item via syntax

Coordinates for mouse click: Sub-items or exact co-ordinates

Feature: None; for web components see Feature for web components[65]

qfs:label*: None

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Node	Text of the node	item	All
Node visible	Check whether node exists	item_visible	All
The node's selected state	Check whether node is selected	item_selected	All
The node's checked state	Check whether node is checked	item_checked	All
Sub-nodes with nesting	The node and all its child nodes including their nesting level	nested_nodes	All
Visible sub-nodes with nesting	The node and all its visible child nodes including their nesting level	visible_nested_nodes	All
Node image	The image of that node	item_image	All

Table 59.40: Checktypes for TreeNode

59.53 TreeTable

Shows content as tree. Can be used to show values in certain categories. Values normally consist of several columns like in a table.

Web For the HTML mapping of a tree table please refer to section 49.1.5[967].

Kind: Component

Coordinates for mouse click: Sub-items or exact co-ordinates

Feature: None; for web components see Feature for web components[65]

qfs:label*: Associated label, Label close to it, Tooltip

qfs:type: None

Additional checks:

All checks of both Table and Tree objects are possible.

59.54   Window

Stands for a usual window.

Kind: Component

Coordinates for mouse click: Exact co-ordinates

Feature: Title, if existing; for web components see Feature for web components[65]

qfs:label*: Title

qfs:type:

qfs:type	Description
Window:Dialog	An independent window, used for entering values or to confirm a message.
Window:EmbeddedWindow	Window of an external application, embedded to the current SUT.
Window:InternalWindow	Window showing content of another area of the SUT.
Window:Notification	Showing notifications

Table 59.41: Special qfs:type values for Window

Additional checks: None

Chapter 60

Doctags

Besides node attributes QF-Test also supports doctags to influence the behavior of nodes during test execution or for report generation. Doctags can be specified in the Comment attribute of a node in the form `@teststep` or `@noreport node`. They must be placed after the general description of the node.

60.1 Doctags for reporting and documentation

Doctags used for formatting test documentation of Test sets and Test cases are described in section 22.2[(276)], doctags for formatting of the documentation of Packages and Procedures in section 22.3[(278)].

The doctags described below influence the reprensation of nodes in the report.

Doctag	Nodes	Description
@teststep [name]	All nodes	If this doctag is set, the node will be treated as Test step in the report. You can specify an optional name.
@report	Check nodes, Sequence with time limit and Server HTTP request nodes	If this is set the node will be reported as check in the report in any case.
@noreport [type],[errorlevel]	All sequences like Test case, Test set or Test step, all Checks, Sequence with time-limit or request steps	If this doctag is set, the node won't be mentioned in the report. See section 60.1.1[(1216)] for details.
@link [filePath/url]	All nodes	This doctag can be used - multiple times if desired - to link to an external resource or file. The target can then be opened in an associated application or shown in the system file manager by right-clicking and selecting the respective entry in the popup menu (Open link or Show file in explorer). Files are resolved relative to the current test suite.

Table 60.1: Doctags for reporting and documentation

60.1.1 @noreport Doctag

4.2+

You can use the @noreport doctag to filter several nodes from the reports. You can make use of two optional parameters to specify the filtering. Those parameters are `type` and `errorlevel`. Example of a valid syntax: `@noreport tree;errorlevel<=WARNING`.

type

You can use either 'tree' or 'node'. 'tree' is the default in case nothing is specified. Using 'tree' filters the entire node and all children from the report. 'node' filters just that particular node from the report. The children will be in the report.

errorlevel

This parameter is only activate for sequence nodes like Test set, Test case or Test step. Using this parameter enables you to filter nodes only if dedicated error level have been reached. You can configure the error levels `EXCEPTION`, `ERROR`, `WARNING` or `MESSAGE`. For comparison you can use >,<,<= or >=.

`errorlevel<ERROR` filters the node only if no error and no exception occurred. That's the default setting. `errorlevel>=MESSAGE` filters the node in any case. That's very dangerous of course and should only be used if there are very good reasons.

60.2 Doctags for Robot Framework

The following doctags are used to designate QF-Test Procedures[(586)] or entire Packages[(594)] for use with Robot Framework. See chapter 28[(349)] for further information about the Robot Framework integration.

Doctag	Nodes	Description
@keyword [name]	Procedures	When used in a Procedure node, the name of the procedure or the optional name specified after the doctag is provided as keyword to Robot Framework. It is sufficient to provide one of the formats supported by Robot Framework which will automatically convert from other variants, so that, for example, a Procedure named `doClick` can automatically also be called via the keywords "Do Click" or "do_click". The doctag can be used multiple times in order to map the procedure to several different keywords. The implicit parameter "__keyword" always holds the current keyword being called in the form specified by QF-Test.
@keyword	Packages	In Package nodes the @keyword doctag can be used without argument to designate all directly or indireclty contained Procedures as keywords based on their name.
@tag [name]	Procedures	Names specified with the @tag doctag in Procedure nodes are passed through to Robot Framework as tags.

Table 60.2: Doctags for Robot Framework integration

60.3 Doctags for test execution

Using those doctags influences the execution of tests.

Doctag	Nodes	Description
@scope [QF-Test component ID|SmartID]	All nodes	Upon node entry the given component scope is pushed and applied to all subsequent component resolution based on SmartID until the node is exited. See section 5.7[80] for details.
@rerun [parameters]	All nodes	You can configure the instant rerun in case of errors. Please see section 23.3.2[295] for details.
@outputFilter keep [regexp]	All SUT client starter nodes	Only those lines in the output of the process started by the node that match the specified regular expression are shown in the QF-Test terminal.
@outputFilter drop [regexp]	All SUT client starter nodes	Lines in the output of the process started by the node that match the specified regular expression are not shown in the QF-Test terminal.
@dontcompactify	All nodes	Designate the node as relevant for the run log so it will not get removed during compactification (see option Create compact run log[510]).

Table 60.3: Doctags for test execution

60.4 Doctags for Editing

Using those doctags can influence the behavior of QF-Test during editing.

Doctag	Nodes	Description
@blue	All nodes	Add a blue mark when loading the test suite the next time.
@breakpoint	All nodes	Add a breakpoint when loading the test suite the next time.
@green	All nodes	Add a green mark when loading the test suite the next time.
@red	All nodes	Add a red mark when loading the test suite the next time.
@yellow	All nodes	Add a yellow mark when loading the test suite the next time.

Table 60.4: Doctags for editing

60.5   Doctags influencing the procedure builder

The doctags for use in the definition file for the procedure builder are described in chapter 54[(1156)]. In chapter 25[(307)] you will find general information on the procedure builder.

Appendix A

FAQ - Frequently Asked Questions

Evaluation and licensing

1. Is an evaluation version available for download?

Yes. Please visit www.qfs.de/en/qf-test/download.html.

2. Do I need anything else?

Normally a license file is required to run QF-Test. It will run without a license, but it will not let you save any files or load any files that were not provided by Quality First Software GmbH. This is sufficient for getting a first impression, working through the tutorial and making a first attempt at running your application under QF-Test. To go beyond that, you'll need a license file.

3. So how do I get a license?

You can obtain a free trial license valid for 4 weeks by filling in the request form at www.qfs.de/en/qf-test/free-trial.html.

4. How much does QF-Test cost?

License types and prices for QF-Test are listed at www.qfs.de/en/qf-test/pricing.html.

5. Does QF-Test need an additional license server?

No, not necessarily. QF-Test handles multi user license management for local networks by itself, provided that IP multicast works. For floating licenses across multiples sites and in case of restricted networks, a dedicated license server is available. The license server itself is free of charge and server licenses are very reasonably priced. For further information about the license server, please get in touch with QFS via <sales@qfs.de>.

Support, training and feedback

6. Where do I get help troubleshooting?

- Before asking for help, please read through this FAQ or the general FAQs

https://www.qfs.de/en/qf-test/faq.html to see if your question has already been answered.

- For beginners the learning-by-doing tutorial https://www.qfs.de/en/qf-test-tutorial/lc/tutorial-en-firsthelpweb.html or the instructions for getting started with QF-Test https://www.qfs.de/en/get-started-with-qf-test.html should prove useful. Further questions might also be answered by the manual.

- Webinars for beginners take place every Monday 16:00 CEST. Registration via webinar@qfs.de.

- Videos for beginners and advanced learners can be found at https://www.qfs.de/en/get-started-with-qf-test/videos.html.

- There is also a blog at https://www.qfs.de/en/blog.html containing lots of helpful postings (full text search is possible).

- While evaluating QF-Test you have free support via email support@qfs.de or phone +49 8171 3864820.

- Customers holding a Software Maintenance Agreement with QFS also profit from above support options. (For details see www.qfs.de/en/qf-test-support/product-support.html).

7. What about training for QF-Test?

QF-Test trainings for beginners and advanced users in German and English language take place regularly here at QFS. There is also the option for webinar-based or on-site consulting and training. Details can be found at www.qfs.de/en/qf-test-support/training-consulting.html.

8. How can I request an enhancement to QF-Test?

Enhancement requests are welcome anytime at support@qfs.de.

9. Where do I report a QF-Test bug?

Simply send an email to qftest-bug@qfs.de. and we will have a look. Please be sure to provide as much information as possible, especially test suites and run logs.

Test execution

10. Why do tests fail today that were OK yesterday though nothing changed in the meantime?

As the first step, please ensure that really nothing has changed as automatic updates of Java or browser versions may happen without being recognized.

Irrespective of changes there can be tests that fail only occasionally for no apparent reason. This may sound like a bug in QF-Test, but that's rarely the case. In complex, multi-threaded environments, many actions and interactions depend on timing. The first thing to try is to introduce delays at critical points. If that helps you can focus on minimizing the delay by using Check nodes with a timeout or Wait for component to appear nodes to wait for a certain condition.

If delays don't help you need to dig deeper and try to understand what's happening. It's not unlikely that the root cause is a bug in your application - typically a tricky one that shows only occasionally depending on timing or other circumstances. The blatant, obvious bugs are typically found earlier - these tricky ones are part of what testing is all about. The detailed logs and screenshots that QF-Test creates help analyzing such situations. Our support can help you interpret the data and isolate the relevant information to forward to development.

11. How do I run a test automatically from the command line, a test management tool or some other kind of script?

You can run QF-Test in batch mode through the command line argument -batch[(867)]. Many other command line arguments can be used to configure the test run. The exit code of QF-Test reflects the outcome of the test. See Test execution[(280)], Command line arguments[(867)], Exit codes for QF-Test[(885)] and Interaction with Test Management Tools[(312)] for details.

12. Is it possible to test two applications running at the same time in two different JVMs?

Yes, just start two SUT clients with different names. You can then control both of them.

13. I've got a long-running test and QF-Test runs out of memory. How can I prevent that?

To increase the available memory, start QF-Test with the argument -J-Xmx1280m (or an even greater value; QF-Test uses up to 1024 MB by default). On Windows you can alternatively use the QF-Test Java Configuration tool, available from the Windows system menu. On Linux/Unix rerunning the QF-Test setup script (setup.sh) also lets you adapt memory usage. Of course the amount of memory you can use depends on your computer. Please refer also to chapter 1[(2)] for further details.

There are a number of ways to reduce the memory use of QF-Test:

- Make sure that the option Create compact run log[(510)] is checked so that all irrelevant nodes are removed from run logs.

- QF-Test keeps 4 run logs accessible from the ⟦Run⟧ menu by default. Keep the option Automatically save run logs[(502)] active so that QF-Test can save these run logs to files and release their memory.

- Close run log windows that you no longer need so that the memory for these run

logs can be reclaimed.

- For long-running tests the best option is to create split run logs (which QF-Test uses as default) so QF-Test can save partial run logs to files instead of holding the entire run log in memory. See section 7.1.6[122] for details.

- If the option Don't create run log[510] is checked, no run log will be generated at all. This should also be used with caution since it can be extremely difficult to interpret what happened without the help of the run log. Use split run logs instead.

- If your SUT prints lots of output you can reduce the number of old clients that are kept around by changing the option Number of terminated clients in menu[461].

14. Hard mouse events and drag'n'drop operations do not work flawlessly, components cannot be found, the run log contains black or corrupt screenshots. What do I have to take care about for test execution?

GUI testing requires an unlocked, active desktop. That is the only way to ensure that the SUT behaves the same as if a normal user interacts with it.

To make sure that your test environment complies with this requirement, you'll probably need to tweak its setup. This holds true notably for continuous integration and build tools like Jenkins/Hudson (c.f. chapter 27[336]). Otherwise you can run into serious trouble during Test execution[280], for example black screenshots in the run log (c.f. section 7.1[117]), failing drag'n'drop operations, non-working hard mouse events or even problems during component recognition (c.f. chapter 5[42]). Java WebStart applications fail to start up. Chapter Hints on setting up test systems[410] contains useful tips and tricks to set-up your test systems.

Before running a GUI test, check whether the following conditions are met:

- QF-Test and the SUT have to run within an active, unlocked user session.

- The Test must not be executed within the Windows service session. It also must not run without an user session.

- During the Test execution[280] with Jenkins/Hudson or similar tools you have to ensure that the Jenkins/Hudson Windows node does not get started as service. It has to be started either via Windows Autostart or Windows Task Scheduler within a real user session. Please ensure that you select a valid user account at the 'Security options' and that you have 'Run with highest privileges' disabled. The selected user must be logged in during test execution, e.g. by automatically logging in, and the desktop must not be locked in any way.

- RDP connections must not be minimized or closed, that would result in a locked session. Instead of RDP you should use VNC, Teamviewer or similar tools to

observe the running tests. RDP must not be used for the initial user login and start of the related session.

On Windows 10 or Windows Server 2016 systems you can make use of RDP if you modify the Registry. Therefore navigate to `HKEY_CURRENT_USER\Software\Microsoft\Terminal Server Client` or `HKEY_LOCAL_MACHINE\Software\Microsoft\Terminal Server Client` and add a new value `RemoteDesktop_SuppressWhenMinimized` as `DWORD` having the value `2`. Once that setting has been set you are allowed to minimize RDP connections, but you have to keep the connection alive. The tests will still fail if you disconnect or close the session You can find further details about setting up your test systems in the manual at chapter 37[410] and chapter 23[280].

Technical background:
The keywords are 'session 0 isolation'. This means that every user has its own session ID, starting with 1. The session with ID 0 is reserved for services and applications without user context. It is restricted in its functionality and applications which run in this session are isolated from other sessions. Applications with GUI cannot be displayed within this session. When running a GUI application in the service session, it will not be rendered correctly and thus may not behave as expected. If you search for 'session 0 isolation' with your preferred search engine you will get extensive information, especially for Windows Vista and newer.
Due to security reasons, the well-known workaround for Windows XP and Windows 2000 via tscon.exe and redirection of session 0 is not working anymore since Windows Vista.

To get round all that problems you should consider to work with virtual machines, especially from a security point of view. If you execute the tests on a virtual machine, the above mentioned requirements apply for this virtual machine only, not for the host. You can lock the host and don't care about session management on the host.

Scripting

15. How can I access objects in my application that are not components?

You cannot get an object out of the blue, some kind of registry must exist that returns the object from a class static method. Typical examples in the standard Java API are `java.lang.Runtime.getRuntime()` or `java.awt.Toolkit.getDefaultToolkit()`.

16. Fine, but how do I use these from Jython, Groovy or JavaScript respectively?

This is standard Jython stuff: Simply import the class and call its methods, e.g.
```
from java.lang import Runtime
runtime = Runtime.getRuntime()
```
In Groovy the package `java.lang` gets imported even automatically:

```
def runtime = Runtime.getRuntime()
```
You can access any class of your application the same way, provided the class is declared public. Note that you must use an SUT script node, not a Server script node.

17. How can I access additional Java classes from a script?

To make additional Java classes available to Jython, Groovy and JavaScript, put them in a jar file and place that in QF-Test's plugin directory (see section 48.2[(916)]).

18. How can I throw an exception from a script?

There are two ways to do that:

- Jython:
  ```
  raise UserException("Some arbitrary message")
  ```
  Groovy:
  ```
  import de.qfs.apps.qftest.shared.exceptions.UserException
  throw new UserException("Some arbitrary message")
  ```
  JavaScript:
  ```
  import {UserException} from
  'de.qfs.apps.qftest.shared.exceptions'
  throw new UserException("Some arbitrary message")
  ```

- `rc.check(condition, "Message", rc.EXCEPTION)`
  will raise an exception only if the condition is false.

19. Which external editor should I use?

That's a matter of taste, to some even religion. A comprehensive list of editors for all kinds of operating systems that support Python syntax highlighting and other goodies is available at https://wiki.python.org/moin/PythonEditors. There are probably dozens of suitable editors with syntax highlighting for Jython, Groovy and JavaScript - jEdit (www.jedit.org) is only one of them.

Web

20. How do I know which web UI toolkit is used for my web application and what do I do if it is not directly supported by QF-Test?

If possible, please ask your developers about UI toolkits or JavaScript components used. Alternatively, activate the auto-detection mode in the Setup sequence creation[(29)] when creating your start sequence. Then QF-Test recognizes supported toolkits automatically and prints a respective message in the terminal.

If the toolkit is unsupported or remains unknown you might want to ask our support to take a look at your web application's HTML code. New or custom toolkits can be integrated with the help of a CustomWebResolver with little effort - either by yourself

(Improving component recognition with a `CustomWebResolver`[(945)]) or using our services.

21. Why is this small file-upload/file-download dialog showing up before the real file selection dialog gets displayed?

Before a file selection dialog is displayed the QF-Test browser is showing up a small dialog with an OK-/Cancel-button. This helper dialog is needed by QF-Test to get the data out of the file selection dialog since this dialog is created natively by the operating system. After clicking the OK-button in the helper-dialog the native file selection dialog is displayed and you can enter the filename or select the file directly.

22. Why does the replay of an already recorded file-upload-/file-download-sequence fail if I use another browser and how can I bypass that issue?

Depending on the implementation of the file-upload-/file-download on a specific page the replay might get a bit complicated and even vary between different browsers. The QF-Test standard library `qfs.qft` contains a special procedure `qfs.web.input.fileUpload` to handle this. Please use this procedure instead of your recorded sequence if you encounter problems during replay.

23. I get an error page notifying me about untrusted certificates. Unfortunately the standard dialog to add an exception is not working properly. How can I solve this problem?

This problem won't occur in QF-Test version 3.5.1 and higher, because SSL certificates are now accepted automatically. Please use the following workaround if you are using an older version of QF-Test.

There are different approaches to add a certificate using firefox.

Solution 1:

- Open the URL `chrome://pippki/content/certManager.xul` in a QF-Test browser window.

- This will show the certificate manager where you can add the certificate.

- After pressing OK the whole browser window will close.

- When loading the URL which needs a certificate again the page will load without a certificate error.

Solution 2:

- You can run a normal firefox from the command line with the following parameters `firefox -profile "[path to your userprofile]/.qftest/mozprofile"`

```
(e.g.:  firefox -profile
"c:/Users/user1/.qftest/mozprofile")
```
(Please close all other running firefox instances before executing this command.)

- Load the URL in the browser and add the certificate to the trusted certificates.

- When loading the URL which needs a certificate again the page will load without a certificate error.

24. I'm getting an OutOfMemoryError for the browser. How to increase memory for the QF-Test browser?

In general it is recommended to create the setup sequence by use of the quickstart wizard. In the resulting sequence within the step "Start browser without window" there is a "Start browser" node in which you can specify the maximum memory as part of the Java VM parameters via e.g. -Xmx384m, which means 384 MB maximum memory. The current default is 256 MB.

Figure A.1: Set browser maximum memory

25. The web-application opens a popup-window. If I try to close this window using a Window event "WINDOW_CLOSING" the main-window is closed instead of the popup-window at times. How can I ensure that the correct browser-window is closed ?

In order to distinguish those windows QF-Test requires additional information. This information specifies which window should be used for replaying events. It has to be set at the Wait for document to load node as well as for the recorded Web page in the attribute Name of the browser window. You can reach the recorded Web page quite fast via right mouse click at the Wait for document to load node and selecting Locate component .

We recommend to set that attribute to `${default:windowname:}`.

Specifying the attribute Name of the browser window allows QF-Test to evaluate the content of that attribute in addition to the web page's URL for recognizing the web page itself and all underlying components. That's why QF-Test is able to distinguish between both windows during replay. Setting that value for the attribute at the Wait for document to load node assigns a new name to that window from QF-Test's perspective. You can see that name in the titlebar of the browser as well.

Before replaying the respective Wait for document to load and events for that popup window you need to set the variable `windowname` to an arbitrary value, e.g. "popup". You could also use a dedicated Set variable node in order to set that variable value. A more convenient way might be to pack all events into a sequence and define the variable `windowname` at the "Variable definitions" table of that sequence.

Now you can replay those events.

It is recommended to reset the variable `windowname` after executing those event nodes again. Otherwise QF-Test will try to replay all subsequent events on that popup window again. Therefore you can use another Set variable node for `windowname` with an empty "Default value". In case of specifying the variable at the sequence node the variable will disappear after that sequence.

If you have already recorded some tests, you should update your test suite using a global replacement action for the attribute Name of the browser window. This action should set all values of that attribute from empty to `${default:windowname:}`. Therefore choose Edit→Replace from the menu and change to the advanced replacement mode via pressing the two golden arrows at the dialogs's toolbar. Now leave the "Search for" field blank and specify `${default:windowname:}` at the "Replace with" field. Then select Name of the browser window for "attribute" and check the checkbox "Whole attribute".

You need to repeat that replacement action for any test suite which contains event and component nodes for that web page.

Appendix B

Release notes

B.1   QF-Test version 7.1

B.1.1   Version 7.1.2 - March 14, 2024

New features:

SWT
- QF-Test now supports tests for applications based on Eclipse/SWT 4.31 alias "2024-03".

Electron
- Improved detection of new windows in Electron applications.

- For improved clarity the interactive terminal windows for the various script languages in QF-Test and the SUT are now labelled as consoles.

Bugs fixed:

- The recording button in the toolbar is now correctly displayed when changing the toolbar icon size.

B.1.2   Version 7.1.1 - February 27, 2024

The only change to this version is the removal of three executable files from the embedded cdp4j library that were suddenly flagged as malicious by various scanners. Those files were part of QF-Test since version 6.0.4 (November 2022), have never been used by QF-Test and should be harmless. Further information will be provided when we know more.

B.1.3 Changes that can affect test execution

- Support for WebStart in Oracle Java 8 is now deprecated and has been marked for removal in a future QF-Test version. This does not affect OpenWebStart.

- Support for applets has been removed from QF-Test. Internet Explorer was the last browser to support applets and support for Internet Explorer was deprecated in QF-Test version 6.0 and removed in QF-Test version 7.0.

- The start of QF-Test with a JRE other than the bundled one, especially with Java 8, is now deprecated. This has no impact on the java versions supported for the SUT.

- The embedded cdp4j library was updated to version 7. This implies a namespace change of the cdp4j classes from `io.webfolder.cdp` to `com.cdp4j`. If such classes are directly referenced in SUT scripts, the imports have to be adapted accordingly.

CustomWebResolver

After replacing the hard-to-digest `qfs.web.ajax.installCustomWebResolver` call with the Install CustomWebResolver[(796)] node in QF-Test 7.0 the underlying code has now been further optimized and cleansed. In some cases, adjustments may be required. Please contact <support@qfs.de> if you need any help with this. Specifically, the following points are affected:

Web
- The evaluation order of the CustomWebResolver categories "genericClasses" and "redirectClasses" is now well-defined: The first match wins, based primarily on the order of entries in the Install CustomWebResolver[(796)] node (see section 49.1.2[(949)] for details). For existing CWR configurations it is possible that entries apply that were not taken into account before.

Web
- The Install CustomWebResolver[(796)] step no longer supports the outdated categories "indirectFeatureClasses", "insertClassesFront", "textRedirectInFetch" and "goodClasses" as well as any previously deprecated categories including "ieHardClasses" and "ieSemiHardClasses".

B.1.4 Version 7.1.0 - February 20, 2024

New features:

- QF-Test now also provides a high contrast theme in light and dark mode.

- The visualization of steps in the trees for test suites and run logs has been freshened to a more condensed style that is highly configurable. See Display[(425)] for the various new options.

- Support was added for testing applications based on Java 22.

Web
- Web applications built with modern Vaadin frameworks (from 14 on) are now supported out of the box. Also included is support to enable basic testability for Flutter Web based applications as well as generic component recognition for web applications implementing the WCAG ARIA guidelines for accessibility.

Swing
- The visual UI Inspector is now also available for Windows and Swing/AWT applications, as well as web views embedded in Java applications (see section 5.12.2[(97)]).

- The UI inspector now provides an easy way to copy a suggested SmartID.

- Conditions in If[(605)] and other nodes can now be implemented in any scripting language, not just Jython. The default scripting language for new nodes can be defined in the option Default script language for conditions[(420)].

- The steps Error[(753)], Warning[(757)] and Message[(763)] can now optionally print the message to the QF-Test terminal.

- Empty arguments in steps starting an SUT client are now ignored by default (see option Ignore empty arguments when starting a client [(460)] for details).

Windows-Tests
- The Windows installer for QF-Test now directly supports uninstalling older QF-Test versions.

- The JRE distributed with QF-Test has been updated to Temurin OpenJDK version 17.0.10.

- Groovy was updated to version 4.0.18.

Web
- The embedded Chrome browser used for QF-Driver mode was updated to CEF version 120.

Web
- The embedded cdp4j library was updated to version 7.0.1. This also updates the Chrome Devtools Protocol API to r1245094.

Web
- The bundled GeckoDriver was updated to version 0.34.0.

- The various CarConfigurator applications used for demos and trainings have been filled with new data and updated to a new look. The demo test suites for these applications now contain further examples of SmartID usage.

Web
- Added newer mobile devices specifications to mobile emulation mode.

Web
- The attribute Method[805] in the Server HTTP request[802] node now also supports variable values and the new attribute Additional headers[805] can be used to define additional headers on a textual basis which is easier to address at script level than the Headers[805] table.

- The new package `qfs.utils.json` in the standard library `qfs.qft` provides utility procedures for comparing JSON files.

- The mail handling procedures in the package `qfs.utils.email.pop3` of the standard library `qfs.qft` now support SSL encrypted connections.

Android
- With the new package `qfs.autoscreen.android` it is now possible to create image based tests for Android applications.

- The recording button in the toolbar now indicates whether SmartID recording is active.

- The new class `ImageRepDrawer` adds support for simple drawing operations on `ImageRep` objects at script level (see section 52.10.3[1100]).

Web
- The flexibility of the CustomWebResolver configuration has been improved in the categories "redirectClasses", "abstractCoordinatesClasses", "ignoreTags", and "browserHardClickClasses", for example it is now more often possible to use regular expressions in the definitions.

Web
- Calls to the procedure `qfs.web.ajax.updateCustomWebResolverProperties` can now also be converted into Install CustomWebResolver[796] nodes.

Web
- It is now possible to use a procedure with an `installCustomWebResolver` call as "base" of an Install CustomWebResolver[796] step configuration.

Web
- The QF-Test pseudo DOM API was extended by the `callJS` method to execute JavaScript code in the web document context without implicitly calling `window.eval()`.

Bugs fixed:

- When replaying events the timeout values from the two options Wait for non-existent component (ms)[480] and Wait for non-existent item (ms)[480] are now taken into account individually and not just as a simple summation.

- It is now possible to use a 'QF-Test component ID' as String parameter for the `ImageWrapper.grabImage` method.

- After a tabulator character was pasted into a script editor field, moving the cursor could lead to an exception.

JavaFX
- A modal JavaFX window is now recorded with class name 'Dialog'.

Web
- It is now possible to send MOVED and SIZED events to the HTML component of a web page (using the Component event[(695)] step) to define the position and size of the inner browser area.

Web
- In component recording mode for web applications QF-Test now ignores invisible DOM nodes. The old behaviour can be restored via rc.setOption(Options.OPT_WEB_RECORD_INVISIBLE_ELEMENTS, true).

Web
- In some cases automatic Chromedriver download failed due to Chrome printing an unexpected error message during version detection.

Web
- Regular expressions in the CustomWebResolver category "ignoreTags" are now correctly parsed.

Web
- Elements with CSS class "visually-hidden" are now treated as invisible components in QF-Test.

Web
- In some special cases QF-Test could not connect to the Edge browser upon start.

Web
- CSS styling informationen contained in STYLE tags could mistakenly get interpreted as text content.

B.2 QF-Test version 7.0

B.2.1 Version 7.0.8 - December 5, 2023

New features:

SWT
- QF-Test now supports tests for applications based on Eclipse/SWT 4.30 alias "2023-12".

Web
- Support for JxBrowser version 7.36 was added.

Bugs fixed:

- Report nodes created with an empty @teststep doctag are now shown with expanded variable values.

Mac
- On macOS QF-Test was sometimes freezing during startup in case the WebP image compression library was not available.

Web
- In electron applications, popup menu item clicks could not be replayed unless a pulldown menu item was clicked beforehand.

Web
- The Microsoft Edge browser crashed when a new empty tab was opened manually.

Android
- The SUT client for android did not terminate automatically after closing the emulator if the recording window was open at that time.

B.2.2 Version 7.0.7 - October 11, 2023

New features:

Web
- The embedded Chrome browser used for QF-Driver mode has been updated to CEF version 117 which fixes the WebP security vulnerability.

Web
- The embedded websocket library was updated to Undertow 2.2.26.

- The script method `rc.overrideElement` now also supports overriding nested SmartIDs. It is now complemented by the new method `rc.getOverrideElement`. See section 11.2.7[(171)] and section 48.5[(917)] for details.

Bugs fixed:

- Test reports created in batch mode with QF-Test versions from 7.0.4 to 7.0.6 could show broken HTML when opened by navigating from the summary to a detail report.

B.2.3 Version 7.0.6 - September 29, 2023

Bugs fixed:

- The embedded WebP library used for image compression in testsuites and run logs was updated to version 1.3.2. In this version a severe security vulnerability (CVE-2023-4863) was fixed.

B.2.4 Version 7.0.5 - September 20, 2023

New features:

- The procedures `qfs.autowin.acrobat.savePDF` and `qfs.autowin.acrobat.saveAsText` in the standard library `qfs.qft` were updated for Acrobat Reader versions 23 and higher.

Web • Support for JxBrowser version 7.35 was added.

Web • QF-Test now supports capture and replay of clicks on popup menus in Electron applications.

Bugs fixed:

• Interactive QF-Test failed to start if a plugin contained an incompatible version of `org.w3c.css.sac` helper classes.

Web • QF-Test now uses the new headless Chrome mode also on Linux. Without that headless Chrome version 117 and higher failed to start.

B.2.5 Version 7.0.4 - August 30, 2023

New features:

• The JRE distributed with QF-Test has been updated to Temurin OpenJDK version 17.0.8.1_1.

SWT • QF-Test now supports tests for applications based on Eclipse/SWT 4.29 alias "2023-09".

Web • Support for JxBrowser version 7.34 was added.

Mac • On Apple Silicon devices, browsers connected via CDP-Driver are now launched with native ARM support, noticeably improving performance.

• Most toolbar buttons now have a "What's this?" entry in their context menu, leading to the respective documentation in the manual.

Bugs fixed:

• The `jackson.jar` library for YAML and JSON parsing no longer creates conflicts if another Jackson library is added to the QF-Test plugin directory.

• Highlighting the scope component via the context menu of the node or the @scope doctag now works correctly again.

Web • Automatic download of ChromeDriver and WebDriver binaries for Google Chrome and Microsoft Edge now works again. In both cases the respective URL and/or site layout changed.

Electron • When starting an Electron application, QF-Test occasionally waited for the full timeout even if the connection could be established quickly.

B.2.6   Version 7.0.3 - Juli 13, 2023

Bugs fixed:

- In rare cases a test run was aborted with an exception when QF-Test tried to save a split run log and encountered an incorrectly created empty screenshot.

- Recognition of components via the old qfs:label algorithm could fail in special cases where an `ExtraFeatureResolver` was registered.

- Variables in doctags of Execute shell command[(652)] nodes are now expanded correctly.

Swing
- In rare cases addressing a line as a sub item in a `JTextArea` could cause a `NullPointerException`.

Web
- When replaying a Fetch geometry node on a non-existing sub item in a web application, the geometry of the parent component was mistakenly returned. Now the correct `IndexNotFoundException` is thrown instead.

Android
- The procedure `qfs.android.adbUtils.appPackage.getCurrentPackage` in the standard library `qfs.qft` now also works on Android devices that don't provide a `grep` program.

B.2.7   Version 7.0.2 - June 22, 2023

New features:

- Support for JxBrowser version 7.33 was added.

Bugs fixed:

- When starting QF-Test version 7.0.1 to only show a run log and later opening a test suite the previous session was not restored.

- On Windows with a scaled monitor the PDF client now displays a scaled document with frames and check highlights correctly aligned. Image checks are created at 100% resolution and thus should remain compatible with image checks from a non-scaled monitor.

Web
- The installation of a CustomWebResolver lead to an exception in case it contained erroneous JavaScript commands. Now an error is logged instead.

Web
- The Install CustomWebResolver[(796)] node failed if some jar file in the plugin folder provided a conflicting Jackson library.

B.2.8 Version 7.0.1 - May 31, 2023

New features:

SWT
- QF-Test now supports tests for applications based on Eclipse/SWT 4.28 alias "2023-06".

Mac
- Support for testing clients running with an ARM Java on macOS was added.

Web
- Support for JxBrowser version 7.32 was added.

- Startup of QF-Test is now noticeably faster, in interactive mode as well as batch mode.

- The new special variable `${qftest:suite.name}` expands to the name of the test suite as defined in the root node.

Web
- New CWR node mappings and categories are now inserted at the cursor position instead of at the very end.

- When using compact run logs (see option Create compact run log[(510)]) it is now possible to exclude nodes from compactification via the doctag @dontcompactify (see chapter 60[(1215)]).

Bugs fixed:

- QF-Test now starts with the system property -Dsun.io.useCanonCaches=true to enable canonical filename caches for Java version 17 in order to avoid performance degradation caused by large projects on slow file systems. This restores the behavior of previous Java versions.

- Some procedures in the standard library failed when called with a SmartID due to use of ${qftest:engine.$(id)}. SmartIDs that don't include an explicit GUI engine now use the GUI engine "default".

- Entries in configuration files were no longer sorted if QF-Test was running with Java 17.

Web
- Occasionally check mode remained activated in a Browser even after stopping recording, thus blocking further interaction.

Web
- If a website was used as scope for a SmartID, the scope was not updated correctly when navigating.

Web
- Deleting a file via the File System Access API could throw an exception in special cases.

- Duplicated categories in a CWR configuration are no longer ignored and trigger an error instead.

- Screenshots of Swing windows are now taken with higher quality.

- On Windows with a scaled monitor the PDF client now displays a scaled document with frames and check highlights correctly aligned. Image checks are created at 100% resolution and thus should remain compatible with image checks from a non-scaled monitor.

B.2.9 Changes that can affect test execution

New Java version for QF-Test

QF-Test is now distributed with Java 17 as its own JRE. Running QF-Test with Java 8 is deprecated, meaning it is still supported for QF-Test version 7.0 but may get removed at some later point.

This change can affect Java application tests which rely on using QF-Test's Java version instead of explicitly specifying a Java binary for starting the SUT. If you run into problems due to this you have two options:

Short-term workaround: You can switch back to Java 8 via the QF-Test Java configuration or the command line.

Long-term solution: The preferred solution is to explicitly specify a dedicated Java version matching the application's requirements in the Start Java SUT client[635] node or, better yet, use a Start SUT client[639] node to launch the application via a script or executable that ensures the correct environment including the Java version.

New algorithm for determining associated labels

In most cases label resolution should continue to work out of the box. The most notable exceptions are `ExtraFeatureResolvers` working with the qfs:label `ExtraFeature`. These will need to be updated as described in section 52.1.11[1028]. For help with updates or for means to disable the new algorithm entirely, please get in touch with <support@qfs.de>.

The algorithm for determining the associated label for a component has been rewritten from scratch for better performance, clarity and increased flexibility. The new qfs:label* variants like qfs:labelLeft or qfs:labelText can be used to designate specific label variants, with qfs:labelBest as the new counterpart for the legacy extra feature qfs:label. Please see section 5.4.4[67] for detailed information about the many new options.

In order to maximize backwards compatibility, the legacy algorithm is still maintained and used to resolve the `qfs:label` extra feature so tests based on recorded Component[823] nodes should not be negatively affected. If desired, recording can be switched to the legacy algorithm and `qfs:label` via the option Recording of qfs:label* variants[486].

For replay with SmartIDs the situation is slightly different. Without explicit qualifier or with the qualifier "label=" or "qlabel=", SmartIDs are resolved based on the new algorithm with qfs:labelBest. In most cases this should work as before. In case it fails you can either re-record the affected SmartID or change its qualifier to "qfs:label=" to enforce using the old algorithm.

Further breaking changes:

<p style="margin-left: 2em;">Web</p>

- Support for Internet Explorer was deprecated in QF-Test version 6.0 and has now been removed, as it has reached End of Life. The IE-specific procedures `qfs.web.browser.settings.enableCompatibilityMode` and `qfs.web.browser.general.isIE6` were removed from the standard library `qfs.qft`.

- Support for Firefox version 43 and older using connection mode QF-Driver was deprecated in QF-Test version 6.0 and has now been removed.

- Support for 32bit software is now deprecated for removal in a future QF-Test version. This applies to the Java versions QF-Test runs on as well as all supported SUT versions. In case you still need to support tests for specific 32bit applications, please get in touch with <service@qfs.de>.

- The format of HTTP headers returned by the Server HTTP request[(802)] node was updated for easier parsing. Examples are provided in the procedures `checkHttpResponseHeader` and `getHeaderValue` in the web services demo suite, accessible via the menu item | Help→Explore sample test suites... |.

- For faster startup IPv6 is now disabled in QF-Test, where it is not needed. This has no impact on the SUT and in case IPv6 is required for a QF-Test plugin it can be reactivated via the command line argument `-ipv6`[(872)].

- When resolving a nested or scoped SmartID the already resolved parent or scope component was wrongly taken into account again, so that e.g. `#Panel:Some title@#Panel:<0>` would target the Panel "Some title" instead of its first child Panel.

- Options like SmartID recording[(484)] that have an effect in both QF-Test and the SUT can now be set in a Server script[(628)] node and will get automatically forwarded to all SUT clients.

B.2.10 Version 7.0.0 - April 27, 2023

New features:

- The new dark mode is just the most visible aspect of the streamlined UI with slightly larger fonts and icons and more spacing in general. See menu View→UI Theme and options Minimum font size (pt)[427] and Terminal font size (pt)[464].

- The development of SmartIDs has left the preview stage. Based on the reimplementation of the algorithm for finding the associated or nearest label of a component, they combine simplicity with precise and efficient component recognition in many situations. The new qfs:label* variants also apply to classic Component[823] nodes. For detailed information please see section 5.6[73] and section 5.4.4[67] as well as the various options described in section 39.4[483].

Web
- The new Install CustomWebResolver[796] node for implementing a CustomWebResolver for web applications replaces the rather cryptic Procedure call[589] to `qfs.web.ajax.installCustomWebResolver`. Existing calls will of course continue to work, but can also easily be transformed into the new node via the context menu item Transform node into as described in section 49.1.2[949].

Web
- There is now a visual UI Inspector for web and Android applications (see section 5.12.2[97]).

- With the help of the new nodes Error[753], Warning[757] and Message[763] it is now possible to directly log errors, warnings or plain messages anywhere in the test suite with the added bonus of configurable screenshots and diagnostic logs.

- QF-Test based tests can now easily be integrated into JUnit 5 tests. This simplifies the execution of QF-Test based tests from an IDE like IntelliJ or Eclipse. Also, a Gradle plugin is now available for including QF-Test based tests in build pipelines controlled by Gradle. See section 27.5[346] for details.

- The XML format for saving test suites is now configurable. For example, new test suites are now saved with UTF-8 encoding by default and with longer lines. Existing suites are not changed by default, but can be converted in one go. The new format can still be read by older QF-Test versions. Further information is provided in section 39.1.2[423] and section 42.1[862].

- XML reports are now saved with UTF-8 encoding by default.

- Support was added for testing applications based on Java 21.

- Groovy was updated to version 4.0.11.

Web
- The embedded Chrome browser used for QF-Driver mode has been updated to CEF version 108.

Web
- Support for JxBrowser version 7.31 was added.

- The embedded JUnit library was updated to version 5.9.2.

- Support was added for the web framework Fluent UI React version 8.

- QF-Test now supports accessing local files from tested web applications using the File System Access API in WebDriver and CDP-Driver connection mode so that capture and replay for this use case should now work out-of-the-box.

- Pseudo attributes can be used to simplify resolvers using JavaScript to retrieve values from a browser. In certain cases they can improve performance. For a closer look see Web – Pseudo Attributes[(996)].

- QF-Test now provides readily accessible templates for frequently used scripts like resolvers via the Templates[(633)] popup in Server script[(628)], SUT script[(631)] and Unit test[(790)] nodes. It is also possible to define additional templates that show up in the list.

- Via the new option Show step types for named tree nodes[(426)] it is possible to hide redundant parts of the tree node descriptions in test suites and run logs.

- Clicking on a line number in a script step now selects the whole line.

- If the new option Automatically open created nodes[(430)] is activated, new nodes are already expanded after insertion.

- The name of the result variable of a Procedure call[(589)] node is now shown in the tree. The result of the call can also be displayed in the run log after the name of the procedure by activating the new option Show return values of procedures[(503)].

- The buttons in the QF-Test toolbar can now be rearranged by dragging with the mouse. The original layout can be restored via the menu View→Toolbar.

- The new special variable ${qftest:language} expands to the current language of the QF-Test user interface.

- The new special variable ${qftest:project.dir} expands to the directory of the project to which the current test suite belongs.

- Tree items in the SUT can now be addressed by a combination of numerical, textual and regular expression indices.

- A Data driver[(562)] node below a Test step[(539)] node can now be packed into a nested Test step.

- A Comment[(751)] node can now be inserted above the currently selected node via the Insert menu or the keystroke Shift-Ctrl-7.

- If a Wait for client to connect[(670)] node is used to wait for a dedicated engine, that engine is now displayed in the tree.

Mac   • The QF-Test application icon for macOS was adapted to modern standards.

Web   • Many new mobile device specifications were added for mobile emulation mode.

Bugs fixed:

• An exception was thrown in the procedure `qfs.utils.xml.compareXMLFiles` if one of the two files was empty.

• Jython   scripts   that   rely   on   the   system   property `python.security.respectJavaAccessibility=false` for accessing private class members now also work with Java 9 and higher.

Web   • Check and component recording could not be properly activated in parallel in web tests with CDP-Driver or WebDriver connection mode.

Web   • Running a mobile emulation test with CDP-Driver connection mode was updated for newer Chrome versions.

Web   • In special cases the execution speed of tests for web applications can be noticeably improved via `MainTextResolver` or `WholeTextResolvers` variants without default parameter.

Web   • In rare cases, use of the @:: syntax in the `genericClasses` parameter of the procedure `qfs.web.ajax.installCustomWebResolver` could lead to a CSS class being inadvertently mapped to the class of the node.

Android   • Text input has been improved for Android devices with API level 33 and higher.

B.3   QF-Test version 6.0

B.3.1   Version 6.0.5 - March 15, 2023

New features:

• Support was added for testing applications based on Java 20.

SWT   • QF-Test now supports tests for applications based on Eclipse/SWT 4.27 alias "2023-03".

• The JRE distributed with QF-Test has been updated to Zulu OpenJDK Version 8_362.

• Groovy was updated to version 4.0.10.

- The embedded JUnit library was updated to version 5.9.2.

Web
- QF-Test now uses the "new headless mode" of Chromium based browsers.

Web
- Support for JxBrowser versions 7.29 and 7.30 was added.

Web
- The bundled GeckoDriver was updated to version 0.32.2.

Web
- QF-Test can now also intercept starting an external browser via `Desktop.open()`.

Bugs fixed:

- Jython now also works with Java 9 or higher if the system property `python.security.respectJavaAccessibility` is set to `false` in order to directly access private class members.

- The SAX variant of the procedure `qfs.utils.xml.compareXMLFiles` in the standard library `qfs.qft` now ignores leading and trailing whitespace in the `noCheck` parameter.

Web
- When using the WebDriver connection mode, alert dialogs originating from IFRAMEs might not have been displayed.

Web
- With CDP-Driver connection mode on Windows QF-Test did not switch browser tabs correctly before replaying a hard mouse event. Also, empty tabs from downloads could remain open.

Electron
- It is now possible to set the working directory for the start of an Electron application.

- The module `qf` is now also available in JUnit Groovy Scripts.

B.3.2 Version 6.0.4 - November 29, 2022

New features:

- The JRE distributed with QF-Test has been updated to Zulu OpenJDK Version 8_352.

- For the SUT QF-Test now also supports the Semeru OpenJDK from IBM.

SWT
- QF-Test now supports tests for applications based on Eclipse/SWT 4.26 alias "2022-12".

- The QF-Test manual now references the official QF-Test docker images in chapter 30[373].

- With the new methods `rc.pushOption` and `rc.popOption` it is now possible to temporarily override an option without disturbing existing settings. Procedures in the `qfs.qft` standard library now use these methods instead of the less suited `setOption`/`unsetOption`.

- When recording SmartIDs (see section 5.6[73]) the option Always record class for SmartID[484] now determines whether the class is always prepended. It is active by default. Besides readability this can have a significant impact on replay performance.

- The embedded JUnit library was updated to version 5.9.1.

Web
- The bundled cdp4j library was updated to version 6.2.0.

Web
- Support for JxBrowser version 7.28 was added.

Web
- The bundled GeckoDriver was updated to version 0.32.0.

Web
- The CustomWebResolver for Angular Material has been updated for the latest release of the framework.

Web
- In web tests with CDP-Driver connection mode it is now possible to control the print dialog with QF-Test.

Web
- The selection event of type "reload" can now be used as an alias for "refresh" to reload the displayed web page.

Electron
- The procedures in the package `qfs.web.browser.external` of the standard library `qfs.qft`, which can be used to intercept the start of an external browser process from the SUT and redirect it to QF-Test, now also work with Electron applications.

Bugs fixed:

Web
- The automated mapping of Tables and TreeTables in the web resolver for Primefaces was updated to support Primefaces version 12.0.

Web
- Chrome with QF-Driver connection mode could crash in special cases after a failed navigation.

Android
- In rare cases special characters in the text of Android components could terminate the connection to an Android device.

B.3.3 Version 6.0.3 - September 6, 2022

New features:

- The JRE distributed with QF-Test has been updated to Zulu OpenJDK version 8_345.

SWT
- QF-Test now supports tests for applications based on Eclipse/SWT 4.25 alias "2022-09".

Web
- Support for JxBrowser version 7.27 was added.

- The methods `getFirstChild`, `getNextSibling`, `getPreviousSibling`, `getFirstElementChild`,`getNextElementSibling` and `getPreviousElementSibling` were added to the QF-Test pseudo DOM API. See section 52.11.1[(1119)] for details.

Bugs fixed:

Web
- When performing an upload with CDP-Driver connection mode, a TestException is now thrown in case the specified file does not exist.

Web
- In rare cases the information about sub-items was lost when recording a mouse click in a web application.

Web
- In some cases a browser window was mistakenly closed in case of a WebDriver timeout triggered for a different frame.

Web
- Overlays in web applications driven by the Angular framework might not have been recognized correctly.

SWT
- For SWT applications on Linux, replaying a selection on a ToolItem in a vertical ToolBar could trigger the wrong item.

B.3.4 Version 6.0.2 - July 20, 2022

New features:

- Clean shutdown of running QF-Test instances - especially in batch mode - has been greatly improved. With the new command line arguments `-allow-shutdown [<shutdown ID>]`[(868)] and `-shutdown <ID>`[(879)] it is now possible to target individual QF-Test processes based on their process ID or a previously specified shutdown ID. The former command line arguments `-allowkilling`[(868)] and `-kill-kunning-instances`[(873)] still work but are now deprecated.

- The command line argument `-clearglobals`[870] now also works for calldaemon mode (see section 23.2.2[287]). To that end the `DaemonRunContext` API has two new methods, `setGlobals` and `clearGlobals` (see section 53.2[1140]).

Web
- Support for JxBrowser version 7.26 was added.

- The faster variant of the procedure `qfs.utils.xml.compareXMLFiles` in the standard library `qfs.qft` now also supports sorting.

- For SmartIDs the prefix of special features like "Tab: some tab" or "Label: some label" is now optional. See section 5.6[73] for details about SmartID syntax.

Bugs fixed:

- When logging screenshots of individual windows, QF-Test mistakenly also used to create images for embedded windows from different GUI engines.

- When editing options, assigning duplicate hot keys is no longer allowed.

Web
- The pseudo DOM method `DomNode.getElementsByTagName` now also returns slotted elements.

Web
- Text from slotted elements was not taken into account when determining the feature of web components.

Web
- In direct download mode existing files are no longer overwritten by default but saved with a unique name.

SWT
- Replay of TAB keystrokes is now significantly faster for SWT on Windows.

B.3.5 Version 6.0.1 - June 9, 2022

New features:

SWT
- QF-Test now supports tests for applications based on Eclipse/SWT 4.24 alias "2022-06".

- There is a new variable group to conveniently escape special characters in SmartIDs. `${quotesmartid:...}` deals with the item syntax characters '@', '&' and '%' as well as the SmartID special characters ':', '=', '<' and '>'.

Bugs fixed:

Android
- Replay of text input for android applications has been improved.

Web
- In web applications a GenericClassNameResolver registered on "DOM_NODE" was not called correctly.

SWT
- Replaying text input as single events for SWT applications running on Ubuntu 22 could lead to garbled text with the order of some characters changed. QF-Test now replays these events with improved synchronisation.

B.3.6 Changes that can affect test execution

Web
- Support for testing Firefox using QF-Driver connection mode, which is limited to Firefox versions 43 and lower, has been deprecated for removal in a future QF-Test version. Please consider using current Firefox versions with WebDriver connection mode instead.

Web
- Support for testing Internet Explorer, which has officially reached end-of-life, has been deprecated for removal in a future QF-Test version.

Web
- Support for testing Opera using WebDriver connection mode has been deprecated for removal in a future QF-Test version. Please use CDP-Driver connection mode instead.

Web
- When testing web applications in CDP-Driver or WebDriver connection mode, slot-table nodes are no longer referenced as direct children of the WebComponent node, but as children of their assigned slot node. This is unlikely to break existing tests, but if it does, please contact QFS support.

Swing
- The way QF-Test internally addresses table columns in a Swing JTable was changed from model-based to view-based. This has no effect if table columns or cells are addressed with a textual index @... or %... or if the order of the columns in the table view and model is identical. In case a test based on numeric column indexes &... fails you can either update the column index or restore the previous functionality with an SUT script of the form `rc.setOption(Options.OPT_SWING_TABLE_USE_VIEW_COLUMN, false)`.

Windows-Tests
- The internal API of the UI automation library has been reworked to simplify class names (e.g. "AutomationWindow" became "Window"). If you use the `uiauto` module directly in your scripts and reference class names directly, you may need to adapt the class names according to the supplied JavaDoc.

B.3.7 Version 6.0.0 - May 17, 2022

New features:

- The new Android engine adds support for testing Android applications in an emulator or a real device. See chapter 16[(217)] for further information.

- Though the embedded JRE of QF-Test is still version 8 - currently at relase 8_332 - QF-Test can now also be started with Java 17 (see command line argument `-java <executable>` (deprecated)[(868)]). This provides crisp display on scaled monitors and enables support for plugins that require newer Java versions.

- Support was added for testing applications based on Java 19.

- In order to reduce the chances of creating screenshots showing sensitive data during a test run QF-Test now takes screenshots only from relevant monitors that show a window that belongs to QF-Test or a connected SUT. While this default setting is useful for personal workstations it may be preferable to turn it off for dedicated test systems via the new option Limit screenshots to relevant screens[(509)].

- The HTML report has undergone a major overhaul. Readability is improved thanks to many subtle details with a more pleasant design and screenshots and error messages are shown as an overlay when clicked, including navigation between screenshots.

- Report creation can now also be triggered via a new toolbar button in the run log window.

- In reports, the name of a test suite, which can be specified in the Name attribute of the root node, is now used in place of the file name of the test suite. This can be configured in the report creation dialog or in batch mode via the new command line argument `-report-include-suitename`[(877)].

- Display of duration indicators for a better understanding of the run-time behaviour of a test run can be turned on for run logs via a new toolbar button and the View menu. See section 7.1.3[(120)] and the options Show relative duration indicators[(502)] and Duration indicator kind[(502)] for further information.

- Activating the new option Create screenshots for warnings[(509)] causes screenshot creation for warnings in the run log in addition to those for errors and exceptions.

- It is now possible to link nodes in a test suite to external resources or documents via the @link doctag. Via a right-click the target can then be opened in a browser or the application associate with the file type. See Doctags for reporting and documentation[(1215)] for further information.

- Groovy was updated to version 4.

- The new parameters `warningDelay` and `errorDelay` in the procedure `qfs.utils.logMemory` in the standard library `qfs.qft` are used to introduce

a short delay in case the `warningLimit` or `errorLimit` are exceeded, followed by an additional garbage collection and another check.

- The performance and memory consumption of the qfs.qft procedure `qfs.utils.xml.compareXMLFiles` have been improved.

Web
- Display and responsiveness of the highlights in check mode were significantly improved for CDP-Driver and WebDriver connection mode.

Web
- The handling of WebComponents using ShadowDOMs and slots has been improved for testing web application in CDP-Driver or WebDriver connection mode (QF-Driver support is still pending): The shadow root node is now accessible as the only child of its host node and slotted nodes are referenced as children of their assigned slot node.

Web
- Text retrieval with CDP-Driver connection mode has been greatly improved and the DOM hierarchy is now consistent with other connection modes.

Web
- Performance of image checks in headless browsers has been improved.

Web
- QF-Test now supports multiple parallel downloads in web-tests with CDP-Driver connection mode.

Web
- The new procedure `qfs.web.browser.settings.setDirectDownload` in the standard library `qfs.qft` allows to download files directly into the directory provided, suppressing the download dialog. Currently for CDP-Driver connection mode only.

Web
- The embedded Chrome browser used for QF-Driver mode has been updated to CEF version 100.

Web
- Support for JxBrowser versions 7.23 and 7.24 was added.

Web
- The bundled cdp4j library has been updated to version 5.5.0.

Web
- The bundled GeckoDriver was updated to version 0.31.0.

Web
- The method `FrameNode.getFrameElement()` was added to the pseudo-DOM API of QF-Test.

Web
- Via the parameter `consoleOutputValue` in the procedures `qfs.web.browser.settings.doStartupSettings` and `qfs.web.browser.settings.setTerminalLogs` in the standard library `qfs.qft` it is now possible to also set the type of the terminal logs.

Windows-Tests
- The embedded UI automation library has been updated to version 0.7.0.

Mac

- On macOS the occasionally showing message dialogs about Safari browser automation are now handled automatically by QF-Test and no longer block test execution.

Preview features:

The following features are not yet complete, but development has reached a point where they are already of great use and the released functionality can be relied upon without concerns about backwards compatibility.

- SmartIDs enable a flexible, easy recognition of components directly from the QF-Test component ID without recording component information first. Please see section 5.6⁽⁷³⁾ and Component nodes versus SmartID⁽⁴⁶⁾ for detailed information.

- Thanks to the new integration with Robot Framework, QF-Test procedures can be used as Robot Framework keywords.

Bugs fixed:

Windows-Tests

- The mini-installer files for Windows - `minisetup.exe` and `minisetup_admin.exe` - can now be run in silent and very-silent mode, similar to the full installation.

Mac

- QF-Test might crash when replaying hard key events on macOS.

- When using split run logs, the maximum number of screenshots for a run log was not always respected, if the option Count screenshots individually for each split log⁽⁵⁰⁹⁾ was turned off.

- The procedure `qfs.autowin.acrobat.saveAsText` in the standard library `qfs.qft` now also works for Acrobat Reader version 22.1 and up.

- A call of `rc.clearTestRunListeners()` in a server script in batch mode broke the output resulting from the command line argument `-verbose [<level>]`⁽⁸⁸³⁾.

Web

- Detection of the Chrome window for semihard clicks has been improved.

Web

- For current Opera versions in WebDriver connection mode QF-Test now supports automatic download of the required ChromeDriver version.

Electron

- The window of an electron application was inadvertently resized upon startup when tested with CDP-Driver connection mode.

Electron

- Popups were not recognized when testing an Electron application in CDP-Driver connection mode.

B.4   QF-Test Version 5.4

B.4.1   Version 5.4.3 - March 11, 2022

New features:

- QF-Test now supports tests for applications based on Eclipse/SWT 4.23 alias
  "2022-03".

- Support for JxBrowser 7.22 was added.

Bugs fixed:

- In very special cases instant rerun of a failed node, triggered via the @rerun doc-
  tag, could lead to wrong values on the variable stack.

B.4.2   Version 5.4.2 - February 18, 2022

New features:

Web
- QF-Test now supports testing with Opera 84.

Web
- The Kendo UI and Smart GWT CustomWebResolvers have been updated for the
  latest release of the respective framework.

Web
- The bundled cdp4j library has been updated to version 5.5.0.

Web
- The method `FrameNode.getFrameElement()` was added to the pseudo-DOM
  API of QF-Test.

Web
- In the procedure `qfs.web.browser.settings.setTerminalLogs` in the
  standard library `qfs.qft` it is now possible to set the type for terminal logs via
  the `consoleOutputValue`.

Bugs fixed:

Web
- Cookies were not properly reset upon browser start for Chrome and Edge version
  98 and higher on Windows.

Web
- When elements of a webpage were reordered QF-Test sometimes failed to syn-
  chronize them correctly with the new order.

- The procedure `qfs.daemon.startRemoteSUT` in the standard library `qfs.qft` did not work correctly if the QF-Test daemon was started with a keystore for securing the communication via TLS.

Swing
- Feature resolution for Swing components had a subtle bug in QF-Test versions from 5.3.4 to 5.4.1, resulting in a tool-tip having higher precedence than an explicitly associated label.

B.4.3 Version 5.4.1 - January 20, 2022

New features:

Web
- QF-Test now supports testing with Opera 83.

Web
- The bundled cdp4j library has been updated to version 5.4.1.

Bugs fixed:

Web
- When using CDP-Driver connection mode QF-Test 5.4.0 could sometimes not find elements added after page load.

Web
- It is now possible to display non-string console log data from a browser started in CDP-Driver connection mode in the QF-Test terminal.

Web
- The Chrome DevTools can now be detached when developing a web test in CDP-Driver connection mode.

Electron
- In some situations, dialogs in Electron applications were not closed properly when using WebDriver connection mode.

Electron
- Connection to an electron application on Windows failed in CDP-Driver connection mode in case the application was starting slowly.

Mac
- The command `automac.sendText` made QF-Test crash on newer macOS systems.

Mac
- `${qftest:os.version}` now returns correct values for Windows 11 as well as macOS 11 and up.

B.4.4 Changes that can affect test execution

- An error was fixed in the `qftest` launch script on Linux. While processing command line arguments with an escaped $-expression in the value of a -variable or -option argument, the $-expression was inadvertently expanded.

Swing
- Components in a Swing `JScrollPane`, most notably `JTree` and `JTable`, were assigned inconsistent qfs:label extra features.

Web
- Testing with Microsoft Edge (legacy) is no longer supported because that version of Edge is generally discontinued. This does not affect support for the current Microsoft Edge browser.

Web
- Execution of the Wait for document to load[776] step has been fixed and the check for document reload improved. This may lead to errors in places where the testsuite design relied on the malfunction. In such a case it is advisable to examine the affected Wait for document to load[776] steps and possibly disable or remove them, or replace them with a Wait for component to appear[771] step. Alternatively it is also possible to reinstate the broken version via the option Reset web-document load state during rescan (before 5.4)[497].

Web
- For web applications the attributes "aria-checked" and "aria-selected" are now automatically taken into account for Boolean check[713] nodes with check type selected or checked.

B.4.5 Version 5.4.0 - December 15, 2021

New features:

- Support was added for testing applications based on Java 18.

Electron
- Electron applications can now be tested using CDP connection mode which is far more effective and works without requiring inclusion of the problematic module @electron/remote into the electron application.

- The dialog for the option settings of QF-Test now provides search functionality.

- It is now possible to copy and paste images from and to QF-Test, most notably for Check image[729] nodes and screenshots in a run log.

- Mouse event[681] nodes with a Modifiers[683] attribute of 4, designating a right-button click, are now shown in the tree as "right-click".

Web
- The embedded Chrome browser used for QF-Driver mode has been updated to CEF version 95.

- Groovy was updated to version 3.0.9

- The JUnit library has been updated to version 5.8.1.

SWT
- QF-Test now supports tests for applications based on Eclipse/SWT 4.22 alias "2021-12".

Web • QF-Test now supports testing with Opera 80, 81 and 82.

Web • Support for JxBrowser 7.20 and 7.21 was added.

Web • For a web application the attribute <u>Check type identifier</u>⁽⁷¹⁵⁾ of a <u>Boolean check</u>⁽⁷¹³⁾ node can now be set to "attribute:<name>" to check for the boolean value of the attribute <name> in the target node.

• The new doctag @outputFilter can be used in client starter nodes in order to suppress unwanted messages in the QF-Test terminal. See <u>section 60.3</u>⁽¹²¹⁸⁾ for details.

• If the <u>Default value</u>⁽⁷⁷⁰⁾ attribute of a <u>Set variable</u>⁽⁷⁶⁸⁾ is a QF-Test component ID in the form ${id:...}, it is now possible to highlight or jump to the target component by right-clicking and selecting the respective item in the context menu.

Bugs fixed:

• The search for unused callable nodes sometimes missed certain references and thus could turn up nodes that were actually still in use.

• QF-Test now tries to avoid creating non-daemon threads in the SUT, including implicitly created threads from the RMI sub-system. These threads could prevent a process from terminating completely after closing the last window of the SUT.

• Performance and memory consumption have been improved in several places.

Swing • The title of a `JPanel` with a `TitledBorder` is now correctly retrieved as its feature.

Swing • The order of the components in a Swing `JSplitPane` seen by QF-Test could vary depending on the order of creation and replacement of those components. QF-Test now uses left->right or top->bottom order irrespective of that.

Web • QF-Test now also supports automatic ChromeDriver download for the Google Chrome variants "Dev" and "Canary".

Web • QF-Test sometimes failed to record events after frame navigation in CDP-Driver mode.

Web • An exception was fixed that could cause failures during document initialization in CDP connection mode.

Web • The "label" attribute of an OPTION element is now taken into account when determining the name of the option.

Web
- When a browser window crashes in CDP connection mode, an error is now reported and the window is automatically closed.

Web
- Handling of unload dialogs during web tests with CDP-Driver connection mode has been improved.

Web
- When running web tests on a headless browser with CDP-Driver connection mode QF-Test no longer attempts to show a temporary Swing dialog for file up- or download. As a result, headless-only web tests with CDP-Driver should now run as batch tests in a container with no X-server at all.

Web
- Synchronization with animations in web applications has been improved for CDP-driver connection mode.

Web
- By default, console output of Firefox in Webdriver connection mode was redirected to the process' standard output so that QF-Test could check it for JavaScript errors. Due to the potentially heavy load on CPU and memory this has been turned off and can be re-enabled by setting the parameter `consoleOutputValue` to 1 in the call to the procedure `qfs.web.browser.settings.doStartupSettings` in the standard library `qfs.qft`.

B.5 QF-Test Version 5.3

B.5.1 Version 5.3.4 - September 30, 2021

New features:

- The bundled GeckoDriver was updated to version 0.30.0.

Bugs fixed:

- A memory leak in QF-Test, introduced in version 5.3.3 has been fixed.

- The procedure `qfs.autowin.acrobat.saveAsText` in the standard library `qfs.qft` now also works for Acrobat Reader version 21.6 and up.

B.5.2 Version 5.3.3 - September 14, 2021

New features:

SWT
- QF-Test now supports tests for applications based on Eclipse/SWT 4.21 alias "2021-09".

• QF-Test now supports testing with Opera 78 and 79.

• Support for JxBrowser 7.17, 7.18 and 7.19 was added.

• The bundled cdp4j library was updated to version 5.4.0.

Bugs fixed:

- The dialog for editing step details is now properly shown the range of visible screens, even when a previously attached monitor gets removed or when switching a session to RDP.

B.5.3 Version 5.3.2 - July 21, 2021

Bugs fixed:

• Some websites containing custom HTML elements were not testable with CDP connection mode.

• In CDP connection mode whitespace was missing from fetched text in some cases.

• In CDP connection mode, the keyCode property of key events generated from text is now set correctly.

• In CDP connection mode the location of elements in nested IFRAMEs was calculated wrongly.

• The Name of the browser window[778] attribute of Wait for document to load[776] nodes was ignored.

B.5.4 Version 5.3.1 - June 15, 2021

New features:

- Support was added for testing applications based on Java 17.

• QF-Test now supports tests for applications based on Eclipse/SWT 4.20 alias "2021-06".

- Performance in CDP connection mode with dynamic content updates has been improved.

- Support for JxBrowser 7.16 was added.

Electron
- QF-Test now supports applications built with Electron 14 or newer, if the `@electron/remote` is bundled with the app.

- QF-Test now supports testing with Opera 77.

Bugs fixed:

Web
- Validity of text input to web applications via single events has been improved through additional explicitly defined keycodes.

B.5.5 Changes that can affect test execution

- Jython issues with character encoding have been reduced and it is now possible to treat Jython literals as 16-bit unicode string which is the natural representation for Java and thus QF-Test. For compatibility reasons the new option <u>Literal Jython strings are unicode (16-bit as in Java)</u>[(420)] is turned off by default if QF-Test encounters an existing system configuration.

  Please see <u>section 11.3.4</u>[(175)] for detailed information about why the option should be turned on and how to trouble-shoot possible issues. Chances are high that your Jython scripts will simply work and string handling will become much cleaner. If not, either undo the option change or fix the resulting incompatibilities. We had to do the latter in only a handful of places in our over 1600 test suites, some of which date back over 20 years. The section <u>Trouble shooting Jython encoding issues 11.3.4</u>[(176)] explains the most common pitfalls and of course our support is always there to help.

- The default folder name of the Firefox profile when executed in WebDriver connection mode has been renamed from `mozProfile` to `firefoxProfile` and it is now used in place instead of copying it to a temporary directory. This behaviour is now consistent with using QF-Driver but has the side-effect that preferences from one test execution are preserved for the next execution and might have to be overwritten during the next browser start. To restore the previous behavior, set the `OPT_WEBDRIVER_COPY_MOZPROFILE` option to `true` before starting the browser.

- The procedures in the package `qfs.utils.ssh` of the standard library `qfs.qft` and the underlying Jython module `ssh` have been updated to default to RSA public key authentication with the default private key file `/.ssh/id_rsa` instead of DSA which is no longer supported by most current ssh servers.

- The option <u>Create compact run log</u>[(510)] is now deactivated by default in interactive mode. Existing system configurations are not affected and the option has no effect in batch mode where compactification is controlled via the command line attribute <u>-compact</u>[(870)].

B.5.6 Version 5.3.0 - May 20, 2021

New features:

Web
- The newly added browser connection mode CDP-Driver supplements QF-Driver and WebDriver for controlling Chromium based browsers via the Chrome DevTools Protocol. By talking directly to the browser without the intervening WebDriver protocol, speed, stability and feature set of CDP-Driver are on par with QF-Driver (and that's after the QF-Driver performance boost, see below). In addition, while QF-Driver is limited to Chrome on Windows, CDP-Driver now supports Google Chrome, Microsoft Edge and Opera on Windows, Linux and macOS, so this is a real game-changer for web test automation with QF-Test.

Web
- The performance of web tests with QF-Driver for Chrome has been significantly improved. Observed speed-up ranges from 10% to over 500%.

- The user interface of QF-Test has been cleaned up and streamlined, using a uniform flat look with fewer lines and beautiful new icons that still maintain the existing image language and are immediately recognizable. The HTML manual and tutorial as well as report and test documentation have also received a face-lift.

- Use of international character sets in Jython scripts is now straightforward. If the new option Literal Jython strings are unicode (16-bit as in Java)[420] is turned on, literal strings (explicitly specified string constants like ″abc″) in Jython scripts are treated as 16-bit unicode and are thus equivalent to strings in Java and the other QF-Test scripting languages. Please see section 11.3.4[175] for detailed information.

Web
- Detection of errors in the browser console has been improved and, depending on the option How to handle errors in a web application[493], they are logged in the QF-Test run log. Besides, the new procedure `qfs.web.browser.settings.setTerminalLogs` in the standard library `qfs.qft` can be used to define if and how messages from the browser console are to be shown in the QF-Test terminal.

Web
- The embedded Chrome browser used for QF-Driver mode has been updated to CEF version 89.

Web
- The bundled GeckoDriver was updated to version 0.29.1.

Web
- Support for JxBrowser 7.14 and 7.15 was added.

Web
- QF-Test now supports testing with Opera 76.

- The JRE distributed with QF-Test has been updated to Zulu OpenJDK version 8_292.

- The quickstart wizard now has its own toolbar button. This - or any other unwanted toolbar button - can now be removed from the toolbar via a right-click popup menu.

- The 'Wait for absence' node now has a dedicated entry in the ⊞Insert⊞ menu and its logic when applied to sub-items has been simplified: Execution of the node is successful if either the parent component of the sub-item is absent or the sub-item itself.

- When running a test with the command line argument `-verbose [<level>]`[(883)], QF-Test now expands variables in node names for console output also.

- The default setting for the available Java memory for QF-Test has been increased to 1024 MB. The configuration of existing QF-Test installations is not affected.

Swing
- Sub-items of Swing `JComboBox` components can now be addressed relative to the `JComboBox` without requiring identification of the popup list.

Linux
- The ability to bring windows of the SUT to the foreground when needed and set the input focus is crucial for automated testing. On Linux QF-Test now uses an updated, more reliable method to bring a window on top regardless of desktop settings if the option Force window to the top when raising[(467)] is not deactivated.

- Exception messages in the run log or an error dialog are now displayed using word-wrap to break long lines. This can be turned off via the option Wrap lines in exception messages[(511)].

Bugs fixed:

- Installing a resolver via the generic method `resolvers.addResolver()` did not work in SUT scripts with the JavaScript language.

Web
- The browser zoom level is now reset to 100% when clearing the browser cache.

- The procedure `qfs.swing.startup.startWebstartSUT` now ensures proper quoting of the `jnlp` argument for use on the command line of Linux systems in order to avoid side-effects from special characters it might contain.

Windows-Tests
- The qfs:label extra feature for elements within a TabPanel of a native WPF application was not determined correctly.

Web
- Since QF-Test version 5.2.2 in very special cases elements in a web page were mistakenly considered to be invisible.

Mac
- QF-Test now runs again on macOS versions older than 10.14.

- Resolution of the qfs:label extra feature for a label located above the target component is now slightly more tolerant about horizontal alignment.

Swing • Fast replay of several mouse clicks onto the same location of a Webswing application could accidentally create double clicks when redirecting events through the browser.

B.6 QF-Test version 5.2

B.6.1 Version 5.2.3 - March 9, 2021

New features:

SWT • QF-Test now supports tests for applications based on Eclipse/SWT 4.19 alias "2021-03".

Bugs fixed:

- In special cases, unluckily placed comment nodes could lead to unwanted side-effects during test execution like a 'Setup' or 'Cleanup' node run just for a comment.

- Improved timing for client shutdown when ending batch mode execution.

Web • In WebDriver connection mode mutations on a website might have gone unnoticed in case the page contained too many elements.

B.6.2 Version 5.2.2 - February 12, 2021

New features:

- Support was added for testing applications based on Java 16.

Web • QF-Test now supports testing with Opera 74.

Web • The bundled GeckoDriver was updated to version 0.29.0.

- The new shortcuts Ctrl-/ or Ctrl-7 can be used to insert a new comment node in the test suite tree.

Bugs fixed:

Web • Several performance bottlenecks for web tests have been fixed, most notably for Firefox on Linux with WebDriver.

Web   • In very rare situations, Chrome was closed in case a JavaScript-execution intervened with a frame reload.

Web   • Delayed attachment of a shadow DOM is now recognized correctly.

Swing   • In special cases the improved event synchronization for Swing might have missed some events, causing slower test execution.

• If a Test case[(518)] with the Expected to fail if...[(523)] attribute set to true does not fail, it should be treated as an error. That error was incorrectly reported as an expected error itself.

• The sort order of parameters is now also automatically applied when changing the target procedure of a Procedure call[(589)] node via the chooser dialog.

B.6.3   Version 5.2.1 - December 3, 2020

New features:

SWT   • QF-Test now supports tests for applications based on Eclipse/SWT 4.18 alias "2020-12".

• Support for JxBrowser 7.12 was added.

Bugs fixed:

• Due to wrong file permissions on Linux machines, Jython scripts would fail when running QF-Test as a user other than the one that installed it.

• The JRE distributed with QF-Test has been changed back to Zulu OpenJDK. The version remains 8_275.

• In some cases, showing an alert box on a web page lead to a browser deadlock.

• Several details for the Webswing integration have been improved, including correct filtering of KeyEvents, focus handling for embedded JavaFX components and cleaner separation of client processes in the demo test suites.

• Daemon connection might have failed during handshake if different java version where used on client and server side.

B.6.4 Changes that can affect test execution

- Testing applications running on Java 7 is no longer supported.

Windows-Tests
- Many procedures in the standard library package `qfs.qft.autowin` have been deprecated in favor of the much better suited Windows engine.

- Due to the updated JRE in the QF-Test installation, graphical elements and graphs in the PDF client may be painted with slightly different anti-aliasing. This can lead to errors in <u>Check image</u>[729] nodes. Given that such problems cannot be ruled out for future JRE updates you should set the <u>Algorithm for image comparison</u>[732] attribute of affected nodes to "algorithm=similarity;expected=0.98".

B.6.5 Version 5.2.0 - November 10, 2020

New features:

- QF-Test now supports integrated testing of Swing and JavaFX applications that are displayed in a browser using the technologies Webswing or JPro. See <u>chapter 18</u>[249] for an explantion of the concepts and the demo test suite for Webswing, accessible via the menu [Help→Explore sample test suites...], entry "Webswing SwingSet Suite".

Mac
- QF-Test is now notarized by Apple and thus starts on modern macOS systems without showing a warning message.

Web
- QF-Test now also supports testing with the Microsoft Edge browser on Linux.

- The JRE distributed with QF-Test has been updated to Liberica OpenJDK version 8_275.

Web
- The embedded Chrome browser used for QF-Driver mode has been updated to CEF version 85.

- Groovy has been updated to version 3.0.6.

- Jython has been updated to version 2.7.2.

Web
- Support for JxBrowser 7.11 was added.

Web
- The embedded GeckoDriver has been updated to version 0.28.0.

Web
- QF-Test now supports testing with Opera 72.

Web
- Device specifications for many current mobile devices have been added to mobile emulation mode.

- The JUnit library has been updated to version 5.7.0.

- On Windows systems with a scaled display QF-Test now starts QF-Driver browsers in compatibility mode so that scaling is transparently handled by Windows and tests work very similar to unscaled mode except for image checks.

- It is now possible to specify options on the command line via the argument `-option <name>=<value>`[(875)].

- QF-Test command line arguments can now contain "." and "-" characters in arbitrary places and upper or lower case characters at will.

- When testing Java applications, QF-Test can now intercept calls that open a native browser window in order to launch a browser controlled by QF-Test for the given URL. An example is provided in the demo test suites "CarConfig Swing test project" and "CarConfig JavaFX test project", accessible via the menu item Help→Explore sample test suites... .

- The new procedure `qfs.utils.waitForClientOutput` in the standard library `qfs.qft` assists in synchronizing with terminal output of the SUT.

- Several more node conversions are now possible.

- The Server HTTP request[(802)] step now also supports the `PATCH` method.

- The two new procedures `qfs.utils.sendKey` and `qfs.utils.sendText` in the standard library `qfs.qft` can be used to enter text into the currently focused element of the active window.

- The 'No events were recorded' dialog can now be suppressed via the new option Show message if no events were recorded[(440)].

- When merging run logs in batch mode, the command line argument `-mergelogs-masterlog [<file>]`[(874)] can now be combined with `-mergelogs-mode [<mode>]`[(874)] set to "append". The appended run logs will be stored as externalized thus minimizing memory use both during merging and when opening the resulting run log.

- Similar to Jython, script steps for Groovy and JavaScript can now use common exceptions without an explicit import.

- When propagating the parameters of a callable node to its callers, there are now explicit choices for whether to add missing parameters, remove extraneous parameters and/or update the sort order.

Bugs fixed:

- Opening a run log with an automatic rerun still in progress could lead to an exception.

- Encrypted connections to the QF-Test daemon are now also supported by the external daemon-API.

- When generating reports, thumbnail images were created even if -report-thumbnails was not specified.

- The Unit test^(790) step now correctly supports the `self.assertEqual` call in Jython scripts.

- Text input on Swing and JavaFX components was slowed down if a browser embedded into Java was detected.

Swing
- Event synchronization under heavy load for Swing based applications has been improved.

Swing
- Text input with single events on a Swing JTextArea now handles newline characters correctly.

Windows-Tests
- Elements of Windows applications may not have been scrolled visible correctly for hard events and image checks.

Web
- With a browser in WebDriver mode a failed frame focus switch could lead to a StackOverflowException.

Web
- In some cases the MSEdgeDriver was not downloaded correctly.

Web
- Checks on elements inside a shadow DOM could not be recorded.

Web
- Soft (invisible) hyphen characters are now implicitly ignored.

Electron
- In some cases, dialog boxes from Electron where displayed empty.

JavaFX
- The visibility of JavaFX components embedded in Swing was sometimes not determined correctly.

SWT
- For SWT version 4.17 on Windows highlight rectangles on Menus were not restored correctly.

B.7   QF-Test version 5.1

B.7.1   Version 5.1.2 - September 15, 2020

New features:

SWT
- QF-Test now supports tests for applications based on Eclipse/SWT 4.17 alias "2020-09".

Bugs fixed:

- In rare cases QF-Test could crash during image compression if memory was tight.

B.7.2 Version 5.1.1 - August 26, 2020

New features:

Web
- QF-Test now supports testing with Opera 70.

Web
- The embedded GeckoDriver has been updated to version 0.27.0.

Web
- For WebDriver based tests with Chrome/Chromium, site isolation is automatically deactivated.

Web
- Support for JxBrowser 7.10 was added.

- A link to the JavaScript documentation was added to the help menu.

Bugs fixed:

- The Unit test[(790)] node now also searches for JUnit 5 (Jupiter) tests on the classpath.

Electron
- In some cases, native menu clicks on Electron applications were not properly recorded.

Web
- The cache of Chromium based browsers might not have been cleared properly.

- The option Enable 'Local variable' attribute by default[(513)] will now be taken into account when pasting a copied Procedure[(586)] node as a Procedure call[(589)], for node conversions in general and also when recording checks.

Web
- A deadlock could occur if embedded browser containers (e.g. JxBrowser) were removed and added at the same time.

- Adding Comment[(751)] nodes to a procbuilder configuration file could break procedure recording.

Mac
- On macOS, JVM options (starting with "-J-") are now handled correctly.

Windows-Tests
- Text input in Windows applications may not have worked properly when the AltGr key was involved.

• When working with dialogs in headless browser tests, sometimes the invisible dialog was not closed properly.

• A ClassNotFoundException could be triggered when an SWTBrowser was under test.

• When creating procedures via the Procbuilder using FORCECREATION the separation dots for the package structure were replaced by underscores.

• When generating procedures via the Procbuilder values from the Extra features of parent and grand parent nodes can now be used as fallback.

B.7.3   Changes that can affect test execution

• Due to the updated JRE in the QF-Test installation, graphical elements and graphs in the PDF client may be painted with slightly different anti-aliasing. This can lead to errors in Check image[(729)] nodes. Given that such problems cannot be ruled out for future JRE updates you should set the Algorithm for image comparison[(732)] attribute of affected nodes to "algorithm=similarity;expected=0.98".

The JRE update can also cause communication problems between QF-Test and the QF-Test license server in case the license server is run with a very old Java version that cannot cope with the key length required for SSL in current Java versions. In that case it is best to update the license server to the current QF-Test version and use its included JRE.

• The library `jniwrapper` is no longer loaded by default because our old `jniwrapper` version crashes QF-Test on newer JDKs. Modules with native dependencies like `autowin` have been rewritten to no longer depend on it and all references to `jniwrapper` have been removed from the standard library `qfs.qft`.

If you still have script nodes in your test suites that depend on jniwrapper you should try to reimplement these in order to remove that dependency. Please get in touch with our support if you need help.

As an interim solution you can get such scripts to work again (on older JDKs where jniwrapper does not crash) as follows:

– Copy the files from `misc/jniwrapper` in the QF-Test installation directory to `qftest` in the QF-Test plugin directory. To locate those directories, open the Help→Info dialog and look for `dir.version` and `dir.plugin` on the 'System info' tab.

– Add either a Jython server script to your startup sequence with

```
from com.jniwrapper import DefaultLibraryLoader
from java.io import File
DefaultLibraryLoader.getInstance().addPath \
  (File(rc.lookup("qftest", "dir.plugin") + "/qftest"))
```

or the following Groovy variant

```
import com.jniwrapper.DefaultLibraryLoader
DefaultLibraryLoader.getInstance().addPath
  (new File(rc.lookup("qftest", "dir.plugin") +
       "/qftest"))
```

- The ChromeDriver library for old Chrome versions (older than 72) is not bundled wit QF-Test anymore.

- Testing applications running on Java 7 ist still supported in this QF-Test version. However, support for Java 7 has been deprecated and will be removed in QF-Test version 5.2.

B.7.4   Version 5.1.0 - July 8, 2020

Video   Video:

QF-Test 5.1.0
https://www.qfs.de/en/yt/version-51-embedded-browsers-51.html

New features:

- Support was added for testing applications based on Java 15.

- The JRE distributed with QF-Test has been updated to Zulu OpenJDK version 8_252.

Web   - Recording and replay of tests for embedded browsers has been significantly improved.

- JxBrowser is now supported in version 7, embedded into Swing, JavaFX or Eclipse/SWT applications.

Electron   - Support was added for handling native dialogs in Electron applications.

Web   - QF-Test now supports testing with Opera 69.

Web   - Support for the web framework Qooxdoo has been updated for Qooxdoo version 6.

- HTML reports can now be customized via JavaScript in the form of a `user.js`. See section 22.1.4[276] for details.

- The `automac` module now provides methods for simulating keyboard and mouse events. See chapter 51[1010] for further information.

- The root node of a test suite now also has a Name[516] attribute that is shown in the tree.

- The new option Enable 'Local variable' attribute by default[513] determines, whether the attribute 'Local variable' gets pre-activated in newly created nodes.

- The procedure `qfs.utils.dragAndDrop` in the standard library `qfs.qft` has a new optional parameter `eventDelay` to control replay speed.

- It is now possible to convert a CSV data file node into an Excel data file node and vice versa.

- test suite tabs can be moved left or right using the keyboard shortcuts Shift-Ctrl-Page up and Shift-Ctrl-Page down.

Bugs fixed:

Web
- The procedure `qfs.web.browser.settings.setLocale` now also works for WebDriver connections to Chromium based browsers.

B.8   QF-Test version 5.0

B.8.1   Version 5.0.3 - June 17, 2020

New features:

SWT
- QF-Test now supports tests for applications based on Eclipse/SWT 4.16 alias "2020-06".

- The included `jsch.jar` library used by the `qfs.utils.ssh` package in the standard library `qfs.qft` has been updated to version 0.1.55 in order to add support for modern Linux systems like Ubuntu 20.

Bugs fixed:

- The included WebP image compression library was rolled back to version 1.0.0 to avoid incompatibilities.

- Component recognition might have failed when web components had non-integer sizes.

- In rare cases calling `rc.callProcedure` inside the parameters of a Procedure call[589] node could lead to the global variables in the variable stack getting lost.

- The special syntax ${qftest:engine.<componentid>} that can be used to determine the engine of a component now also works if <componentid> contains a '@', '%' or '&' character.

- Selecting a single value in a run log's error list and using "Set value as filter" twice caused an ArrayIndexOutOfBoundsException.

B.8.2   Version 5.0.2 - May 5, 2020

New features:

- The WebP image compression library has been updated to version 1.1.0.

- Keyboard event input in JX Browser is now more stable.

- Contrast of toolbar icons has been improved especially for disabled buttons.

- QF-Test now supports testing with Opera 68.

Bugs fixed:

- Angular 9 is now auto-detected correctly.

- The CSV data file[579] node now correctly handles UTF-8 encoded files with BOM that start with an encapsulated complex expression.

- The Start windows application[657] node can again attach to a client via a given class name (-class) in the Window title attribute.

- Error handling and retry for automatic downloads of WebDriver libraries has been improved.

- Recording elements with a flat hierarchy did not work.

- The PDF client is now able to check a Text component which contains only null "\u0000" characters and treats it as an empty String.

B.8.3 Version 5.0.1 - March 2, 2020

New features:

<div style="display:flex"><div style="color:#993344">**Windows-Tests**</div></div>

- A new demo test suite was added for the Windows 10 Calculator.

- QF-Test now supports Opera 67 with Operadriver 80.0.3987.100.

SWT
- QF-Test now supports tests for applications based on Eclipse/SWT 4.15 alias "2020-03".

Bugs fixed:

Windows-Tests
- When recording components for the whole window, elements within a WPF Tab-Panel were omitted.

- The `qfs.database.executeSelectStatement` procedure now works again for databases requiring an explicit `db.commit()` statement.

- When executing a Server HTTP request[802] node, the returned body was mistakenly not stored in a variable in case of a server error.

JavaFX
- Processing JavaFX images in order to calculate a hash value could cause a `NullPointerException`; to get printed to the terminal.

Windows-Tests
- Fixed a bug where a TextField in a Windows application might not get cleared before input.

Windows-Tests
- Occasionally a ModalDialogException might get incorrectly thrown in WPF Windows applications.

B.8.4 Main new features in version 5

Note
For a detailed list of new features please see the release notes below for QF-Test versions 5.0.0.

The following major new features have been implemented for QF-Test version 5:

Description	Further info
New GUI engine: Windows	Testing native Windows applications[207]
Modernized User Interface of QF-Test	QF-Test looks more modern
Tests with Java 14	Applications based on Java 14 can be tested now
test suites with comments	Comment[751] node directly in the tree of a test suite
Edge based on Chromium	Tests with the final Edge based on Chromium are now possible
File download via the Server HTTP request[802] node	Attribute Save response to file[806]

Table B.1

Changes that can affect test execution:

- The Server HTTP request[802] node now throws an exception if the status code is greater than or equal to 400. This behavior can be changed with the new attribute Error level if status code >= 400[807].

- The options Connect via QF-Test agent[514] and Instrument AWT EventQueue[514] are no longer saved in the system config file and can only be changed at runtime via a script. See section 39.11[514] for detail.

Software that is no longer supported:

Note   Please see section 1.1[3] for a detailed list of system requirements and supported technology versions.

- Testing of applications based on Java 6 is no longer supported.

B.8.5   Version 5.0.0 - February 6, 2020

New features:

- With the new Windows engine QF-Test can now test native Windows applications.

- Support was added for testing applications based on Java 14.

- The new Comment[751] node can be used to improve the structure and readability of test suites and run logs.

- It is now possible to download a file via the new Save response to file[806] attribute of the Server HTTP request[802] node.

Windows-Tests   - A package for Windows applications has been added to the standard library `qfs.qft`.

- On Windows 10 QF-Test is now correctly displayed at scaled high resolution displays.

- In the manual the chapter Web testing[200] has been revised and a section (section 49.1.2[949]) describing the procedure `qfs.web.ajax.installCustomWebResolver` of the standard library has been added.

- The option Show message dialog after[458] now has a setting to show a message dialog also when a test run finishes successfully.

- If the result dialog gets shown after a search the search dialog is now closed automatically.

- The new procedure `qfs.util.click` in the standard library `qfs.qft` can be used to click at an arbitrary position on the screen.

- You can now create an electron start sequence in the quickstart wizard that automatically detects the required ChromeDriver.

- When copying a Procedure call[589], Test call[531] or Dependency reference[551] node the name of the target node is now also copied as text to the clipboard.

- The bundled GeckoDriver has been updated to version 0.26.0.

- On Windows, tests with the Microsoft Edge 78 and newer are also possible in headless mode.

- QF-Test now supports Opera 66 with Operadriver 79.0.3945.79.

- The new variable `engine.$(componentId)` in the `qftest` special group makes it possible to find out which GUI engine a component belongs to.

- The project tree view in QF-Test now uses a natural sort order, respecting indexes and cases.

- Data for several new mobile devices was added to the Mobile Emulation setup in the quickstart wizard.

Bugs fixed:

- An image might have been removed from the run log in low-memory situations.

- Fixed a sporadic exception that could appear when creating a test suite from a run log.

- Finally[623] nodes inside a Try[616] now get executed even when an instant rerun gets triggered from within the Try[616] node.

- `ImageWrapper` methods now log warnings whenever the method fails.

- In very rare cases the Ctrl key might accidentally have stayed in pressed state after finishing replay.

B.9 QF-Test version 4

B.9.1 QF-Test version 4.7

Version 4.7.2 - December 12, 2019

New features:

- The chapters Web testing[(200)] and Web[(945)] in the manual were rewritten from scratch. The description of the central procedure `qfs.web.ajax.installCustomWebResolver` can now be found in section 49.1.2[(949)].

SWT
- QF-Test now supports tests for applications based on Eclipse/SWT 4.14 alias ”2019-12”.

Web
- QF-Test now supports tests with Opera 64 and 65.

Web
- QF-Test now also supports Microsoft Edge version 80 and newer.

Bugs fixed:

- Extending a lease from the QF-Test license server could cause delays in test execution in case of temporary networking issues.

- Fixed a problem with setting a KeyOption via the scripting API.

- Filtering for empty values in the search result dialog could cause an exception.

- The `bin` directories of QF-Test itself and the JRE that QF-Test runs with are no longer visible on the environment variable PATH that the SUT sees when started from QF-Test.

- Printing a test suite could sometimes cause an exception.

- When exporting the error list or some other search result for a run log some node names were not exported.

Web
- XPath expressions with ”-characters now work correctly.

Version 4.7.1 - September 19, 2019

New features:

SWT
- QF-Test now supports tests for applications, which have been developed using Eclipse/SWT 4.13 alias "2019-09".

Windows-Tests
- The defined user interface scaling of windows is now optionally taken into account when executing windows tests.

Electron
- If required, QF-Test now automatically downloads support for new versions of the Electron framework.

Web
- Improved testing perfomance of WebDriver controlled browsers on Windows systems.

Web
- Improved performance of the CustomWebResolver for Primefaces.

Windows-Tests
- During Windows-tests, it is now possible to check for table components a whole row at once.

- Jumping from resultlist to suite now also works with "Return".

Web
- Improved item scrolling performance with WebDriver browsers.

Web
- Updated the embedded GeckoDriver to version 0.25.0.

Web
- Supporting tests with Operadriver 76.0.3809.132.

Bugs fixed:

JavaFX
- The Java FX WebView tests had problems interacting with web frames, which are fixed now.

Windows-Tests
- A hierarchy of modal windows could falsely trigger a ModalDialogException during Windows tests.

Web
- On systems without active user session, replay of web mouse clicks could lead to a NullPointerException.

Web
- Web page load was not detected when the web page or framework undefined the JavaScript dialog methods.

- When extensively expanding variables outside a testrun, QF-Test might have used all available memory.

- Fixed a rare NullPointerException which could happen when using rc.checkImageAdvanced() in combination with an image in image search.

- The standard procedure "qfs.autowin.acrobat.savePDF" could sometimes stop with message "filename field cannot be determined" because of slow network connections when opening the save dialog and cloud targets activated.

Web
- Changed the URL for remotely loaded JavaScript in the "qfs.web.general.integrateFirebug" procedure.

Web
- The automation infobar of Google Chrome is now also hidden during webtests with Chrome version 76 and higher on Windows.

Swing
- If the QF-Test agent is activated two times by an erroneous client specification, the connection to the SUT is now established nevertheless.

- The qfs.database.executeSelectStatement procedure now also works for databases that require an explicit db.commit() statement.

Version 4.7.0 - July 26, 2019

New features:

- Testing of Java 13 based SUTs is now supported.

- Support was added for Electron 6.

- Support was added for OpenWebStart.

Web
- The embedded Chrome browser used for QF-Driver mode was updated to CEF version 73.

Web
- The Opera browser is now supported up to version 62 which uses a different scheme for the associated WebDriver.

Web
- For web testing on Windows in WebDriver mode, display scaling is now taken into account at Windows level as well as for browser-based zooming.

- The new method `rc.getPropertyGroupNames` lists the names of non-special property groups for which values are bound.

Bugs fixed:

Web
- The profile directory for testing with Microsoft Edge had the wrong name.

Web
- Closing a popup window could crash the Opera browser.

Web
- Text input for Microsoft Edge was always positioned at the end of the field.

Web • A possible deadlock in Internet Explorer when loading resources for a web page was fixed.

Web • Stopping a browser controlled via WebDriver could sometimes cause an (otherwise harmless) error message in the QF-Test terminal.

JavaFX • When using a custom ItemResolver for JavaFX clicks to the center of the item were not replayed correctly.

SWT • On Linux SWT instrumentation was performed even when not required.

B.9.2 QF-Test version 4.6

Version 4.6.0 - June 18, 2019

New features:

Web • QF-Test now supports testing web applications in Opera browser on Windows, macOS and Linux.

Web • QF-Test now also supports the new Microsoft Edge browser based on Chromium.

• Support was added for Eclipse/SWT version 4.12 alias "2019-06".

• Via the new methods `print` and `println` that were added to the `qf` module it is now possible to print messages directly to QF-Test's terminal from both Server script[(628)] and SUT script[(631)] nodes, irrespective of whether the SUT output gets redirected or not and whether there's a RunContext available or the code is part of a resolver during recording.

• The new command line argument `-license-waitfor <seconds>`[(873)] allows waiting for a QF-Test license to become available during startup.

• For better conformance with regulations for protection of personal data the `licenseserver.log` file written by the QF-Test license server now contains hashes instead of user name, host name or IP address. This can be turned off via the new command line argument -licenseserver.hashlogentries=false.

• A "mouseButton" parameter was added to the various `clickAtImage` procedures in the `qfs.autoscreen` package of the standard library `qfs.qft`.

• The new shortcut (Shift-Ctrl-Right) can be used in the parameters table of Procedure call[(589)] nodes to quickly create a forward - i.e. `x -> $(x)` - of the current variable value for the selected table rows.

- The embedded JUnit Jupiter library was updated to version 5.4.2.

Bugs fixed:

- Importing procedures from large test suites could lead to an `OutOfMemoryError`.

- Closing QF-Test in check recording mode without killing the SUT process would leave the SUT in a completely unusable state.

- The command line argument `-kill-kunning-instances`[(873)] could inadvertently terminate QF-Test processes on other machines in the local network provided those were started with `-allowkilling`[(868)].

- Auto-saving test suites could accidentally trigger re-compression of images. For test suites with many or large images this could slow down QF-Test noticeably.

- Reading the license file from a slow network drive could occasionally fail.

- Opening a test suite in Windows explorer via the context menu of the suite tab failed in case the file name contained spaces.

- In rare cases waiting for a component during SUT startup could trigger a `ClientNotConnectedException` instead of the expected `ComponentNotFoundException`.

- The procedure `qfs.autowin.acrobat.saveAsText` was updated for Acrobat version 19.12.20034.1161 and newer.

- Variables bound in the data driver of a Test step[(539)] were lost when using "continue from here" inside the Test step.

- Output of the -licenseserver.dump* batch commands was missing information for the JavaFX engine. In addition to fixing that, information about Windows engine licenses was added, the "AWT" columns were renamed to "Swing" and information for engines that are not available in the server license is no longer displayed. These changes all apply to the interactive license administration tool as well.

Web - Closing a popup window in Firefox or closing a browser tab in any WebDriver controlled browser could sometimes cause the entire SUT to terminate.

Web - Frames in a popup window were not recognized correctly if the browser was connected via WebDriver.

Web - Performance for web tests has been improved, for both QF-Driver and WebDriver connection modes. The effects are very dependent on the specific web application.

Web • A workaround for an Electron bug now allows testing with current ChromeDriver versions.

Web • Arguments passed to an Electron application were accidentally prefixed with "–".

Web • Electron applications without native menus are now handled correctly.

Web • Cleanup of browser processes has been improved even in case a browser crashes at startup.

Web • Original third party cookie settings were not always restored correctly after closing a browser.

Web • In a web application Fetch text[740], Fetch index[744] and Fetch geometry[747] nodes could fail with an internal error when trying to determine the value while the target node got dynamically removed from the DOM. Now the operation is retried instead.

Web • Navigation to an anchor within the same page in JXBrowser is now handled correctly.

JavaFX • Event replay for JavaFX is now more resilient against unexpected window focus changes on Linux, especially for menu item selections.

B.9.3 QF-Test version 4.5

Video Video about the new features in version 4.5:

[▷] 'QF-Test Features 4.5'
https://www.qfs.de/en/yt/new-features-45.html

.

Version 4.5.2 - March 20, 2019

New features:

• Support was added for Eclipse/SWT version 4.11 alias "2019-03".

• QF-Test can now automatically fetch ChromeDriver libraries in case the bundled version does not match the selected browser.

Bugs fixed:

• Procedure recording did not create the compound procedures for Panel components if the recording was started at Panel level instead of for the whole window.

- Image checks now also work if the check was recorded with image compression enabled and a SecurityManager is present in the SUT.

- An XML or XSLT library installed in QF-Test's plugin directory could interfere with report generation.

- If a non-existing file was specified on the command line QF-Test did not bring up its main window which looked like the process was hanging.

Version 4.5.1 - February 27, 2019

Bugs fixed:

- In QF-Test 4.5.0 an SUT would no longer start if the QF-Test agent was disabled via the option Connect via QF-Test agent[(514)].

- The path displayed for ${qftest:java} was incorrect if QF-Test was started with the command line argument `-java <executable>` (deprecated)[(868)] and the <executable> parameter did not specify an absolute path.

- The procedure `qfs.autowin.acrobat.saveAsText` now also works for Adobe Acrobat Reader versions 19.010.20091 and up.

- A partial `TreeTableResolver` relying on the default implementation for `getTreeColumn` could cause a stacktrace to get printed to the client console.

Web   - In headless Chrome hard or semi-hard double clicks were not replayed correctly.

Web   - In QF-Test 4.5.0 SUT scripts for web tests could not access plugin jars dropped into QF-Test's plugin directory.

- In SUT scripts the jniwrapper library and the autowin module were no longer accessible.

- On Windows 7, starting a webtest in WebDriver connection mode could trigger an exception.

Highlights of version 4.5

The following major new features have been implemented for QF-Test version 4.5:

Description	Further info
Support for Electron applications	See chapter 19[252], supported Electron versions in section 1.1.2[3]
Mobile Emulation in Quickstart Wizard with support for new devices	See section 14.6[204]
SWT without instrumentation	See option Connect without SWT instrumentation[497]
Support for Java 12	SUT only

Table B.2: Highlights in QF-Test version 4.5

Changes that can affect test execution

Web

- Setup sequences for web applications created with the Quickstart Wizard include a call to the procedure `qfs.web.ajax.installCustomWebResolver` which tries to detect the AJAX framework in use and install a matching resolver. If no pre-built resolver matches, a minimal generic resolver is installed for which the default settings were changed in QF-Test 4.5 so that `DIV` and `SPAN` nodes are now ignored and removed from the component hierarchy. This change does not affect any existing setup sequences but should be kept in mind in case the Quickstart Wizard is used to create a new setup sequence for a web application with existing tests. In such a case, simply set the parameter `ignoreTags` back to the empty value in the generated `installCustomWebResolver` procedure call.

SWT

- On Linux systems Eclipse/SWT applications with SWT versions 4.3 through 4.9 can be run either in GTK2 or GTK3 mode. In QF-Test version 4.3 support was added for GTK3 with SWT versions 4.6 and higher. Until now, however, startup sequences created by the Quickstart Wizard used to include a sequence that sets the environment variable SWT_GTK3 to 0 which causes all SWT versions that support GTK2 to fall back to that version.

  Starting with QF-Test 4.5 this environment variable setting is ignored. It has been replaced with the public option Preferred GTK version for SWT (Linux only)[498] which can be set to "2" or "3" or left empty to use the default for the respective SWT version.

  Unfortunately SWT with GTK2 has become unstable on newer Linux systems and can crash under heavy load which is not uncommon when driven by QF-Test at full speed. A fix for this is to turn on XSync, an option specific to X11 that causes X11 events to be synchronized. This can have a performance impact however, so if you need to run your SWT application with GTK2 and it has been running stably so far you can deactivate the option Activate XSync for SWT with GTK2 (Linux only)[498].

- The connection between QF-Test and the SUT is now established via the loopback interface (localhost) by default. In the rare case that QF-Test needs to control an SUT on another host the command line argument <u>-serverhost <host></u>⁽⁸⁸⁰⁾ can be used.

- Until now, web-test were wrongly started with the QF-Test classpath. Due to this, libraries like "POI" were mistakenly loaded in SUT-scripts. If you rely on these libraries in your SUT-scripts, please copy the corresponding jars explicitly into the `plugins/sut` folder.

Software that is no longer supported

Note   Please see <u>section 1.1</u>⁽³⁾ for a detailed list of system requirements and supported technology versions.

QFS is taking pains to support even very old software. QF-Test should still run, for example, under Windows XP or with an SUT based on Java 6 but for systems that are no longer supported by their own vendors we cannot provide full support.

Support for the Safari Selenium Extension has been removed because Safari does not allow extensions anymore.

Libraries that are required for Firefox versions 44-59 and Eclipse/SWT version 3.6 and older are not part of the default installation of QF-Test. They are still being maintained, albeit with limited testing. For Eclipse/SWT 3.5 - 3.6 simply download https://archive.qfs.de/pub/qftest/swt_legacy.zip and extract the contents into the `swt` directory of your QF-Test installation. For support of older browsers, please get in touch with QFS via `support@qfs.de`.

Version 4.5.0 - February 14, 2019

New features:

Web   - Via its web engine QF-Test now supports testing of Electron applications. See <u>section 1.1.2</u>⁽³⁾ about supported Electron versions and <u>chapter 19</u>⁽²⁵²⁾ for further information about setting up and running Electron tests.

Web   - The Quickstart Wizard has been extended to create setup sequences for Mobile Browser Emulation and the selection of supported mobile devices has been updated and extended.

SWT   - Eclipse/SWT applications no longer require instrumentation of the SWT plugin (except for older SWT versions on Linux) which also greatly simplifies testing of WebStart based or other non-standard SWT applications. See the option <u>Connect without SWT instrumentation</u>⁽⁴⁹⁷⁾ and <u>section 45.2</u>⁽⁹⁰⁰⁾ for further information.

- Testing of Java 12 based SUTs is now supported.

- Performance has been improved in several areas including startup of QF-Test itself and of web browsers, Jython initialization, resolvers and event replay and synchronization, the latter especially for web. The impact varies between barely noticeable and almost twice as fast depending on available CPUs and application details of the SUT.

- Images and screenshots in test suites and run logs are compressed more effectively using the WebP format. To ease the transition QF-Test 4.4 already supports reading respective files. This version makes WebP the default, configurable via the option Compress images in run logs and test suites[510]. Simply loading and saving a test suite does not change the format of embedded images so as not create too much noise in versioning systems. However, test suites can be transformed via a QF-Test batch call with the command line argument `-compress`[870].

Windows

- The ability to bring windows of the SUT to the foreground as needed and set the input focus is crucial for automated testing. On Windows QF-Test now uses a new, very reliable method to bring a window on top if the option Force window to the top when raising[467] is not deactivated.

- New entries were added to the context menu for test suite tabs in the QF-Test workbench view: Copy file path to clipboard and Show in explorer/finder/file manager .

- For processes started from QF-Test it is now possible to access output to the standard output and standard error streams individually via `${client.stdout.<name>}` and `${client.stderr.<name>}`. Combined output remains available via `${client.output.<name>}`.

- Mutual conversion operations between Start process[642] and Execute shell command[652] have been added to the context menu for tree nodes. These are especially useful because they take care of the conversion between the table with individual arguments and quoted arguments in a single command line.

- The new procedure `qfs.shellutils.exec` in the standard library `qfs.qft` is a convenient shortcut for running a command, waiting for it to finish and storing the output in a variable.

- The embedded Selenium WebDriver framework was updated to version 3.141.59, ChromeDriver to version 2.46 (with support up to Chrome version 73) and Gecko-Driver to version 0.24.0 (for Firefox 60esr and higher, currently up to version 65).

- The `qfs.utils.email.sendEmail` procedure in the standard library `qfs.qft` now supports SMTP over SSL.

- It is now possible to fetch the whole text of a PDF page via a Fetch text[(740)] node.

Bugs fixed:

- The context help menu item ⟦What's this?⟧ now works again for the Start web engine[(645)] node and its attributes.

Web
- In Firefox an unexpected dialog could lead to a NoAlertPresentException. This has been fixed for Firefox 63 and up - older Firefox versions ignore the respective WebDriver setting.

SWT
- In QF-Test version 4.4.2 on Linux systems automatically setting the input focus was broken for Eclipse/SWT applications running in GTK2 mode.

- The `rc.checkEqual` method now works correctly again when comparing numbers of a different kind, e.g. integers, doubles and BigDecimals.

- The procedure `qfs.utils.xml.compareXMLFiles` in the standard library `qfs.qft` might have missed differences in attributes.

JavaFX
- The 'enabled' check for a JavaFX MenuItem now correctly identifies the disabled state.

Web
- When switching from a browser to another application in check mode, the check highlight did not disappear reliably.

- QF-Test no longer implicitly sets the environment variable MOZ_NO_REMOTE=1 which causes Firefox to always start a new browser process instead of opening a new tab in an existing one. This legacy setting was formerly used for testing applets.

B.9.4 QF-Test version 4.4

Version 4.4.2 - December 11, 2018

New features:

- Support was added for Eclipse/SWT version 4.10 alias "2018-12".

- ChromeDriver was updated to version 2.44 (still for Chrome up to version 71).

Bugs fixed:

- The license check of QF-Test versions 4.4.0 and 4.4.1 might refuse a QF-Test 4.3 license even when that license had a valid maintenance plan.

- An SUT based on Java 9 or higher that contains references to inner classes that were not deployed with the SUT (like inner classes used only for unit tests) is now handled correctly.

- In case of a slow server response a <u>Browser HTTP request</u>[808] node could cause errors to get logged in QF-Test's terminal even when the request was executed successfully.

- When accessing the reference manual on macOS via the 'What's this?' entry in popup menus the browser did not navigate to the correct position on the target page.

Version 4.4.1 - October 26, 2018

New features:

- ChromeDriver was updated to version 2.43 (for Chrome up to version 71).

Bugs fixed:

- In QF-Test version 4.4.0 PDF tests accidentally required a Swing license. Now they work again regardless of the licensed engines.

- Running QF-Test in batch mode with multiple threads using `-threads <number>`[883] now also works for PDF and unit tests.

- The QF-Test daemon did not work when using a keystore and sharing the daemonport with the normal QF-Test port for SUT communication.

Web   - With Chrome 70 on Windows the screen location of elements was incorrectly computed after a navigation so that hard events might miss their target.

Web   - For Safari version 12 or newer the connection can no longer established via the Safari extension because it is no longer supported by Apple. SafariDriver is now used instead which only supports replay but no recording.

Web   - The print dialog control for Chrome on macOS is now handled correctly.

SWT   - Event recording was broken for Eclipse/SWT legacy versions 3.6 and older.

- WINDOW_MOVED events for the file chooser of the PDF client are now filtered reliably.

- The parameter `returnOnly` of the procedure `qfs.check.compareDates` was ignored in some cases.

Version 4.4.0 - October 11, 2018

New features:

- The new `automac` module can be used to control native macOS applications at script level. It is accompanied by procedures in the package `qfs.automac` of the standard library `qfs.qft`. See chapter 51⁽¹⁰¹⁰⁾ for further information.

SWT
- Support was added for Eclipse/SWT version 4.9 alias "2018-09".

- The Unit test⁽⁷⁹⁰⁾ node can now run unit tests using the JUnit 5 "Jupiter" engine.

- New text checks for PDF documents add the ability to check the text of an entire page with a single Check text⁽⁷⁰⁸⁾ or Check items⁽⁷¹⁹⁾ node.

Web
- GeckoDriver was updated to version 0.23.0 (for Firefox up to version 62) and ChromeDriver to version 2.42 (for Chrome up to version 69).

- The arguments of `TestRunListener` methods can now easily be converted to Json objects and for a `TestRunEvent` a call to `getNode().getComponentId()` now returns the QF-Test component ID of the current step. See section 52.7⁽¹⁰⁸⁷⁾ for details.

Bugs fixed:

- Auto-filling the comment of a procedure could accidentally drop @author information and other doctags.

- In special cases the call to `rc.toServer` in a Jython SUT script⁽⁶³¹⁾ node could cause a `NullPointerException`.

Web
- The result of a Fetch index⁽⁷⁴⁴⁾ step for first-level child nodes of a Tree or TreeTable component in a web application wasn't always correct.

Web
- With Internet Explorer a download triggered in a popup window could remain incomplete.

JavaFX
- Computing the name of an embedded Swing Frame based on the containing JavaFX SwingNode now also works with JavaFX 11.

B.9.5   QF-Test version 4.3

Version 4.3.2 - September 11, 2018

Bugs fixed:

- Raising windows on Linux systems now works more reliably.

- If an $\underline{\text{Server HTTP request}}^{(802)}$ node for a POST request contains parameters encoded directly in the URL, QF-Test now keeps the URL unchanged instead of moving those parameters into the body.

Web
- Deleting cookies after the browser window was opened now works in WebDriver mode too.

Web
- With Internet Explorer replay of a $\underline{\text{Text input}}^{(689)}$ node with very long text did not work reliably.

Version 4.3.1 - August 28, 2018

New features:

Web
- The embedded Selenium WebDriver framework was updated to version 3.14.0 and ChromeDriver to version 2.41 (with support up to Chrome version 69).

- The PDF client can now detect images with a shear mapping.

Bugs fixed:

- The calculation of the extra feature qfs:imagehash was broken for images with an alpha channel.

SWT
- With QF-Test version 4.3.0 tests for 32bit Eclipse/SWT version 4.5 did not work on Linux.

- When a filter was in effect for the project tree view and a test suite that was currently not shown in that tree got modified, display of the project tree could get corrupted.

Faster release cycle

We want to bring new features and bug fixes to you faster. Starting with QF-Test 4.3, a medium upgrade will be released whenever one or more relevant features are fully implemented and minor updates are going to focus on bug-fixes and small improvements. Our continuous self-tests have long since helped to maintain the continued stability of QF-Test and now the required conditions have been put into place:

New license format - all inclusive during maintenance period
    The QF-Test maintenance contract has always included free medium or major

upgrades within the maintenance period. However, a license update had to be requested and rolled out to all test and development machines with QF-Test installations in order to activate the new version.

With the new license file - which has to be explicitly requested for QF-Test 4.3 one more time - this extra effort will be dispensed with: The maintenance period is now encoded in the license file to allow using all new medium and major versions that are released within that period as well as the associated minor updates. When extending the maintenance contract in the future you will automatically receive a new license, valid for the extended maintenance period.

To further simplify upgrading licenses for many QF-Test installations it may be convenient to install a dedicated QF-Test license server for your network. Our sales team will gladly provide further information.

Newsletter for medium upgrades only

To prevent flooding you with emails about the more frequent updates, only medium and of course major upgrades with new functionality and features are going be announced via newsletter. If you would like to stay informed about minor updates you can subscribe to the QF-Test mailing list or our RSS-feed or activate the new automatic check for QF-Test updates described below.

Automatic check for QF-Test updates

When online, QF-Test can now check for updates automatically. You can configure this service according to your needs, for example to check only for specific kinds of updates or to deactivate it entirely.

Version 4.3.0 - July 26, 2018

New features:

- Testing of Java 11 based SUTs is now supported.

SWT
- Support was added for Eclipse/SWT version 4.8 "Photon".

SWT
- On Linux QF-Test now supports testing Eclipse/SWT applications in GTK3 mode (SWT version 4.6 and higher).

- A new demo suite was added that shows how to test REST services. It is accompanied by the new manual chapter 20[258].

- With the new option Create screenshots of all client windows upon error[509] it is now possible to always capture images of all windows of all connected SUTs regardless of where an exception or error occurred.

SWT
- The new procedure `qfs.swt.tree.expandCompleteTree` in the standard library `qfs.qft` can be used to expand all nodes in an SWT tree.

Bugs fixed:

- The default viewer for opening PDF documents on Linux was changed from `acroread` to `xdg-open`.

- The PDF client now handles XObjects even if they use a Type0 font without embedded CMap. The only exception are text checks which are not possible without a CMap.

- The wording of error messages from `rc.checkEqual` calls has been improved.

- The encoding of some German characters in scripts in the `qfs.qft` standard library has been fixed.

SWT
- Check highlight frames for SWT applications on Windows were drawn with an offset leaving visual clutter in case a window's size was reduced so far that the menu bar was broken into two lines.

B.9.6 QF-Test version 4.2

Video about the new features in version 4.2:

'QF-Test Features 4.2'
https://www.qfs.de/en/yt/new-features-42.html

Version 4.2.5 - July 11, 2018

New features:

Web
- The embedded Selenium WebDriver framework was updated to version 3.13.0 and Mozilla GeckoDriver to version 0.21.0 which adds support for Firefox 61.

Bugs fixed:

- Until now, settings for a user or system option that was newly introduced by a QF-Test update were dropped from the configuration files when working with an older QF-Test version. Starting with QF-Test version 4.2.5 such option settings are now always retained.

Web
- Errors shown by Microsoft Edge in a notification bar are now automatically dismissed.

- Placing an Oracle DB driver jar with a broken manifest in the QF-Test plugin directory could interfere with loading test suites.

- When running QF-Test in demo mode, activating a license did not enable the save menu item immediately.

Version 4.2.4 - June 14, 2018

New features:

Web
- The embedded Selenium WebDriver framework was updated to version 3.12.0 and ChromeDriver to version 2.40 (with support up to Chrome version 68). Firefox version 60 is also supported.

Web
- On Windows Insider Builds QF-Test now detects and handles the new deployment model of the Microsoft WebDriver for Edge.

Bugs fixed:

- The options for interactively selecting the GUI engine licenses were deactivated even in case of a mixed engine license.

- Data drivers for very large Excel files now need less memory.

- When exporting a Test Project from Testlink for QF-Test an existing node_order definition is now handled correctly.

- In the report expected errors inside nested test cases were counted as normal errors. Also, expected errors are now easier to identify in the run log.

- The Server HTTP request[(802)] node now supports empty values for HTTP headers.

- Stopping clients now works properly on a Windows Embedded system.

Web
- Startup of Google Chrome on Linux did not work consistently in QF-Test 4.2.3.

Web
- Trying to clear a non-existing cache folder when starting Microsoft Edge could cause an exception.

Version 4.2.3 - May 3, 2018

New features:

• The embedded Selenium WebDriver framework was updated to version 3.11.0 and ChromeDriver to version 2.38 (with support up to Chrome version 67), and Mozilla GeckoDriver to version 0.20.1 which adds support for Firefox 59. The 32bit Linux version of ChromeDriver was removed from QF-Test as it is no longer supported by the Chrome development team. If needed, the last available version (2.33) can be downloaded manually from the ChromeDriver archive.

• QF-Test now supports the Shadow DOM API and thus interaction with web elements that are embedded in an open ShadowRoot.

• To bridge the gap until a full Windows engine is available for QF-Test the `qfs.autowin` package in the standard library `qfs.qft` has been extended to make use of Windows UIAutomation. It includes several new procedures to automate simple tests for native Windows applications. Please see chapter 50⁽¹⁰⁰⁰⁾ for detailed information.

• When moving the mouse cursor over the menu entry for a connected client in the ⟦Clients⟧ menu, the main window of the respective client gets highlighted.

• The new variable group ${idlocal:...} is similar to ${id:...} with the added effect of enforcing component lookup in the test suite that is current at the time of expansion. See section 6.5⁽¹⁰⁸⁾ for further information.

• The debug information for component recognition in the run log is now more readable and - like other text fields in the run log - can be opened in an external editor via the new new keyboard shortcut ⟦Shift-Control-Return⟧.

Bugs fixed:

• The deprecation warning dialog for Java WebStart in Java 9 or higher is now handled automatically by QF-Test

• When refactoring or analyzing test suites, `rc.callProcedure` calls are now taken into account everywhere, i.e. in every variable attribute and not just in script nodes.

• A data driver iteration inside a Test step⁽⁵³⁹⁾ can now be interrupted via a Break⁽⁶⁰⁴⁾ node like any other node iteration.

• Firefox in QF-Driver mode could crash on Linux when processing certain non-UTF-8-encoded strings.

• For Chrome in QF-Driver mode right button mouse clicks did not work correctly in qooxdoo applications.

Web
- A missing "http:" protocol header in a proxy auto-config URL is now gets automatically corrected.

JavaFX
- Getting the name for an embedded Swing window from the enclosing JavaFX `SwingNode` now also works with Java update 8u171.

Version 4.2.2 - March 9, 2018

New features:

Web
- The embedded Selenium WebDriver framework was updated to version 3.10.0 and ChromeDriver to version 2.36 (with support up to Chrome version 66).

Web
- Mobile emulation mode now supports device-specific insets.

JavaFX
- Support was added for JavaFX SubScene components.

Bugs fixed:

Windows
- On Windows the wrong `qftest.exe` and `qftestc.exe` files slipped into the release bundles so that the `bin` directory of the JRE was still added to the front of the PATH environment variable which could interfere with SUT startup. Also, the command line argument <u>-plugindir <directory></u>[(876)] no longer had any effect.

Web
- When using WebDriver connection mode with QF-Test 4.2.1 in some cases the browser could no longer receive input after highlighting a component.

Web
- The webResolvers module (see <u>section 52.2</u>[(1048)]) now also supports registering a `BusyApplicationDetector`.

Version 4.2.1 - February 26, 2018

New features:

- Testing of Java 10 based SUTs is now supported.

Web
- The embedded Selenium WebDriver framework was updated to version 3.9.1, ChromeDriver to version 2.35 (with support up to Chrome version 64), and Mozilla GeckoDriver to version 0.19.1 which adds support for Firefox 58.

Web
- Chrome can now be started in mobile emulation mode via the new PROCEDURE `qfs.web.browser.general.openMobileEmulation` in the standard library `qfs.qft`.

- Proxy settings are now also supported for Firefox and Chrome with WebDriver connection.

- When nesting test cases via a Test call$^{(531)}$ node the new attribute Act like a procedure call$^{(535)}$ determines how exceptions are handled inside the call. The now obsolete backwards compatibility option "Don't treat test calls inside test cases like procedures" has been removed.

- The new Procedure `qfs.utils.waitForOneOfNComponents` in the standard library `qfs.qft` can be used to wait for one of several components to appear.

- The Procedure `qfs.utils.readTextFromFile` in the standard library `qfs.qft` can now handle different file encodings via the optional parameter `fileencoding`.

- Markers and breakpoints can now be made persistent by adding the doctags @breakpoint, @red, @green etc. See section 60.1$^{(1215)}$ for further information about doctags.

- The parameters `errorlevel` and `newerrorlevel` for the @rerun doctag are now optional. See section 23.3.2$^{(295)}$ for details.

- The Database password$^{(573)}$ attribute of the Database$^{(569)}$ node can now be encrypted.

- QF-Test now supports native full-screen mode on macOS via the menu entry $\boxed{\text{View}\rightarrow\text{Fullscreen}}$ and the keyboard shortcut $\boxed{\text{^⌘-F}}$

- On macOS the keyboard shortcut for reloading test suites was changed to $\boxed{\text{⌘-R}}$

- On Linux systems the embedded code editor now supports pasting text via the middle mouse button.

Bugs fixed:

- The PDF viewer now correctly rotates PDF documents.

- In Groovy scripts the `rc.callProcedure` call was broken when trying to use keyword argument syntax.

- In Groovy scripts the `rc.getNum` method call inadvertently returned long instead of int.

- In a few cases a broken system property set by QF-Test could interfere with SUT startup.

- The `bin` directory of the JRE used for running QF-Test is now appended to the environment variable `PATH` for the SUT and not prepended as it was in QF-Test versions before 4.2.

Highlights of version 4.2

The following major new features have been implemented for QF-Test version 4.2:

Description	Further info
Java 9 and 10 support	SUT only for now. Includes support for the new Java module system "Project Jigsaw"
Testing of PDF documents	See chapter 17[(239)]
Additional scripting language JavaScript	Equivalent to Jython and Groovy, see section 11.5[(184)]
Support for Angular	Includes UI toolkits Bootstrap and AngularJS Material, see section 49.2[(988)]
Support for Smart GWT	See section 49.2[(988)]
Dedicated nodes for static and live unit tests	See Unit test[(790)] node
Headless web-testing	WebDriver mode only, requires Chrome 60 or higher or Firefox 56 or higher
Testing of REST services	Supports GET, POST, HEAD, OPTIONS, PUT, DELETE and TRACE see Server HTTP request[(802)] node
The QF-Test tutorial was rewritten from scratch	Tutorial

Table B.3: Highlights in QF-Test version 4.2

Changes that can affect test execution

- The joint functionality in the 'Start/Open browser' node has been replaced by distinct Start web engine[(645)] and Open browser window[(649)] nodes. Conversion is automatic unless the attributes of the old 'Start/Open browser' are contradictory in which case the node remains unchanged and things will work as before. Such nodes should be manually replaced with the new nodes at some point.

- Another somewhat surprising feature for browser startup has also been deprecated: All procedures in the `qfs.web.browser.startup` package now require a running web engine. For backwards compatibility you can set the new parameter 'startBrowserIfNotRunning' to true in order to implicitly start a browser if necessary.

- The implementation of the `rc` scripting API was unified for all scripting languages. This can lead to subtle differences with Jython boolean values converted to strings, varying between "True", "true" and "1" or "False", "false" and "0". This can affect malformed boolean tests written as a String comparison, e.g. `"$(x)"` == `"false"` instead of `not $(x)`.

- Boolean attributes that can be made editable via the '$' button are now evaluated as a Jython expression, same as the Condition[606] attribute of an If[605] node. This can lead to different results or script exceptions in some cases. If this breaks existing tests in just a few places the tests should be updated as this behaviour is much more flexible and consistent. Alternatively you can activate the option Don't evaluate variable boolean attributes (before 4.2)[483] to get back the old behaviour.

- The old Test node has been deprecated and the rest of its previously unique functionality is now available elsewhere: Data drivers below test case level can now be implemented by placing a Data driver[562] node inside a Test step[539] node and the @noreport doctag can be used to prevent a Test case[518] node from showing up in the report (see chapter 60[1215] for background information about doctags).

- The ubiquitous warnings for not-quite-perfect component recognition have been demoted to message level. The warnings can be restored by activating the option Log warning instead of message[474].

Web
- When using a `CustomWebResolver` for web applications, Fetch text[740] and Check text[708] for TextField and TextArea components now both retrieve the same text. Besides, table, tree or list cells with entries implemented as TextField or TextArea now return the displayed text so some checks may need to be updated. This fix can be disabled via the option Don't fetch text for generic classes with generic class approach (before 4.2)[496].

Swing
- For Swing applications all tooltips are now considered empty in case the Swing TooltipManager is disabled.

Software that is no longer supported

Note
Please see section 1.1[3] for a detailed list of system requirements and supported technology versions.

QFS is taking pains to support even very old software. QF-Test should still run, for example, under Windows XP or with an SUT based on Java 6 but for systems that are no longer supported by their own vendors we cannot provide full support.

Support for Java 5 has been dropped entirely. SUTs based on Java 6 still run by automatically switching back to older Jython and Groovy versions.

Libraries that are required for Firefox versions 44-51 and Eclipse/SWT version 3.6 and older are not part of the default installation of QF-Test. They are still being maintained, albeit with limited testing. For Eclipse/SWT 3.5 - 3.6 simply download https://archive.qfs.de/pub/qftest/swt_legacy.zip and extract the contents into the `swt` directory of your QF-Test installation. For support of older browsers, please get in touch with QFS via `support@qfs.de`.

Version 4.2.0 - January 10, 2018

Changes compared to QF-Test version 4.1.6:

New features:

- QF-Test can now test applications based on Java 9. This includes support for the new Java 9 module system ("Project Jigsaw") without having to resort to the "Big Kill Switch", i.e. the command line argument -illegal-access.

- Testing PDF documents like a normal application in a dedicated PDF viewer with textual and graphical checks for individual elements.

- JavaScript was added as new scripting language equal to Jython and Groovy for Server script[628] and SUT script[631] nodes. This encompasses the ECMAScript 8 standard and all QF-Test scripting modules like resolvers, rc or autowin are available in JavaScript.

Web
- Support was added for the popular AJAX framework Angular, including support for the UI toolkits Bootstrap and AngularJS Material.

Web
- Support was added for the AJAX toolkit Smart GWT.

- The new Unit test[790] node can run JUnit tests and integrate the results in the QF-Test run log and report. Tests can be implemented in Java or as scripts and run statically or dynamically in a 'live' SUT as an alternative to writing mocks.

Web
- Support was added for headless web-testing using the headless versions of Chrome and Firefox.

Web
- The Server HTTP request[802] node now supports testing REST services via the additional methods HEAD, PUT, DELETE, TRACE and CONNECT and it is now possible to specify custom headers for the HTTP request.

- The QF-Test tutorial was rewritten from scratch.

- The re-implemented Quickstart Wizard combines a concise workflow with an improved design.

- The ‖Help→Info‖ dialog has a new tab "System info" that shows many QF-Test settings and system information.

- Attributes like the Condition[606] condition of an If[605] node which are evaluated by the Jython interpreter now have syntax highlighting and support for code completion.

Web
- The embedded Chrome browser used for QF-Driver mode has been updated to CEF version 57.

- The command line argument $\underline{-run}^{(878)}$ now also works for interactive mode to directly start test execution when running QF-Test. This is complemented by the new $\underline{-suitesfile\ <file>}^{(881)}$ argument that can be used to specify a text-file containing the test suites and tests to execute.

- There are new commands available for controlling QF-Test from the command line. The new batch argument $\underline{-interrupt-running-instances}^{(872)}$ can be used to suspend tests running in the local machine. Similarly, $\underline{-kill-kunning-instances}^{(873)}$ will cleanly terminate running instances, including interactive ones, provided they were started with the argument $\underline{-allowkilling}^{(868)}$. Cleanly here means that batch tests are stopped, dependencies rolled back and run logs are saved.

Windows
- The new Windows setup program for QF-Test adheres to the current Windows conventions for placing different kinds of files in various locations. There's still an option to use the old layout with all files in a single place.

Mac
- On macOS, application arguments can now be configured from within QF-Test.

Mac
- You can now specify a macOS app directly as the $\underline{Executable}^{(640)}$ attribute of an $\underline{Start\ SUT\ client}^{(639)}$ node.

- A $\underline{Sequence}^{(536)}$ node can now also be transformed into a $\underline{Loop}^{(597)}$ or $\underline{While}^{(600)}$ node.

- Calls to deprecated procedures or test cases are now also stroke through in the tree view.

- The $\underline{Load\ properties}^{(788)}$ node now supports loading property files encoded as UTF-8 via its new attribute $\underline{File\ encoding\ is\ UTF-8}^{(789)}$.

- Components in the SUT are now highlighted automatically when clicking on a component or event node in QF-Test's tree, irrespective of whether the node was already selected or not. If automatic highlighting is deactivated via the option $\underline{Highlight\ selected\ component\ in\ the\ SUT}^{(461)}$ you can now trigger it explicitly via the context menu of component, event or check nodes as well as QF-Test component IDs in parameter tables and scripts.

- The boolean values in the $\underline{Extra\ feature}^{(825)}$ attributes of a Component or the $\underline{Items}^{(721)}$ attribute of a Check items node can now make use of variables. Simply double-click the respective table cell to edit the value.

Web
- The new `webResolvers` module enables implementing resolvers in JavaScript that run directly in the browser. See $\underline{section\ 52.2}^{(1048)}$ for detailed information and examples.

- The extra feature qfs:label now also applies to Window and Dialog components.

- There are several new (or newly documented) command line options for overriding the directories that are relevant to QF-Test, including -userdir <directory>[(883)], -systemdir <directory>[(881)], -jythondir <directory>[(873)], -groovydir <directory>[(872)], -javascriptdir <directory>[(873)], -plugindir <directory>[(876)] and -logdir <directory>[(874)].

- Several definitions for directories relevant for the QF-Test installation have been added to the qftest property group, e.g. ${qftest:dir.log} for the directory into which run logs are saved automatically. See section 6.5[(108)] for a detailed list.

- The sort order of parameter tables like the Variable definitions[(591)] attribute of a Procedure call node can now be reset to the order of the called node via the table's context menu.

Bugs fixed:

- Conversion into a regular expression used on a QF-Test component ID attribute now correctly differentiates between the ID and the sub-item syntax.

- On Ubuntu Linux taking SUT thread dumps and terminating sub-processes of the SUT did not work.

- Unicode characters outside the Basic Mulitilingual Pane (BMP) are now handled correctly.

Swing
- TooltipResolvers are now also called for non-Swing AWT components.

Web
- The procedure qfs.web.general.integrateFirebug now loads Firebug via https instead of http which prevents issues with mixed content.

Web
- In some cases the text of a MenuItem in a web application could mistakenly include the text of its sub-MenuItems as well.

Web
- Creating many embedded WebView or SWT Browser instances could lead to a memory leak.

JavaFX
- Running a JavaFX based SUT via WebStart could cause some IllegalAccessExceptions.

JavaFX
- Names for the tabs of a JavaFX TabPanel now take possible so-called Graphic elements into account.

B.9.7 QF-Test version 4.1

Video
Video about the new features in version 4.1:

'QF-Test Features 4.1'
https://www.qfs.de/en/yt/new-features-41.html

Version 4.1.6 - October 27, 2017

New features:

Web
- The embedded Selenium API was updated to version 3.6.0, ChromeDriver to version 2.33 (with support up to Chrome version 62), and Mozilla GeckoDriver to version 0.19 which adds support for Firefox 55 - 57.

Web
- Window and Panel nodes from the supported AJAX toolkits now have a dedicated title check.

Bugs fixed:

- A workaround was implemented for a bug in the Linux JRE 8_152 that could cause QF-Test to hang during startup.

Swing
- A Swing based SUT started remotely via the procedure `qfs.daemon.startRemoteSUT` did not connect if the option Connect via QF-Test agent[(514)] was deactivated in the daemon.

JavaFX
- Support was added for interaction with the JavaFX HTMLEditor component, including checks for its raw HTML code and the displayed text.

Web
- When recording HTTP requests, some special characters were not saved with the correct encoding.

SWT
- Handling of frames in the SWT embedded Browser now works again.

Version 4.1.5 - August 9, 2017

New features:

- The embedded ChromeDriver was updated to version 2.33 (with support up to Chrome version 62), and Mozilla Geckodriver to version 0.19 which includes support for Firefox 55 and higher.

Bugs fixed:

- A race condition in the load testing demo suite has been fixed.

- The code completion window for scripts was accidentally showing up in the run log.

- The procedure `qfs.utils.files.readExcelFile` in the `qfs.qft` standard library now correctly handles Excel files containing empty rows.

- A Groovy GString returned from a resolver method is now converted correctly to a Java string for further processing.

- A regression in the performance of component recognition based on qfs:label has been fixed and performance of Swing tests has improved in general.

- When executing manual tests based on the suite `ManualTestRunner.qft` the size and location of `ManualStepDialog` windows is now preserved.

JavaFX
- A modal JavaFX dialog that got opened during replay of a WINDOW_CLOSING event could lead to a DeadlockTimeoutException.

JavaFX
- The editable check was missing for JavaFX ComboBoxes.

JavaFX
- Support for the JavaFX SplitMenuButton has been improved so that clicks on the main button and the arrow button are recorded differently.

Web
- Creating many JxBrowser instances could lead to a memory leak.

Web
- Waiting twice for the same web page to load did not fail as expected in case the page was not refreshed in between.

Web
- Playback of a selection event on a list in Firefox with QF-Driver mode now triggers a change event only in case the list selection really changes.

Version 4.1.4 - June 1, 2017

New features:

SWT
- Support for Eclipse/SWT 4.7 "Oxygen" has been added.

Web
- The embedded Selenium WebDriver framework was updated to version 3.4.0, ChromeDriver to version 2.29 (with support up to Chrome version 58) and Gecko-Driver to 0.16.1 which includes support for Firefox 53 and higher.

Web
- Support for the AJAX toolkit Qooxdoo has been updated to latest generic class concept.

- When continuing test execution after a break, the bottom-right view now automatically switches from the variables to the terminal.

- Tabs in the project view now have a context menu and can be closed via a middle button click.

- The manual chapter about modifying the `CustomWebResolver` for AJAX toolkits has been updated to match the latest state of development (section 49.1[945]).

- The procedure `qfs.autowin.acrobat.saveAsText` now automatically handles the protected mode warnings by clicking "Enable all functions".

- By using the new method `resolvers.addSpecificExtraFeatureMatcher` to register an `ExtraFeatureMatcher` with the `resolvers` module, the feature name can be specified at registration time.

Bugs fixed:

- A possible `NumberFormatException` - introduced in QF-Test 4.1.3 - for batch mode execution with −verbose [<level>][883] has been fixed.

- In some cases using the advanced replace dialog to replace the regular expression .* with an empty string could lead to an exception or to table entries being deleted without confirmation.

- TestLink import scripts could have been broken in case of unicode characters in test case names of TestLink.

- When text containing double quotes got pasted from the system clipboard to variables or any other table the double quotes could get lost.

- When determining extra feature `qfs:label` for ComboBox, Spinner or Slider components, the tool tip could mistakenly get used instead of the nearest label.

- The @rerun doctag was not properly evaluated in a node with additional lines in the comment.

- The command line argument −plugindir <directory>[876] was ignored in batch mode.

- Selecting a component via the chooser might create an explicit reference even though the component was part of an included test suite.

- Explicit component references within the local test suite were not taken into account when looking for unused components.

- Analyzing references of relative test-calls didn't work correctly.

- Duration and realtime are now updated correctly when merging run logs.

- In case the fallback library for Jython version 2.5 is installed the Jython package cache now gets activated by default as it was in QF-Test version 4.0.

- If the option Intelligent scrolling in trees[429] was deactivated there were cases when the selected node was not scrolled visible in the tree at all, e.g. after a search or when entering the debugger.

Web
- In some cases custom mappings of classes were not taken into account for testing with AJAX resolvers.

Web
- The libraries required for hardware accelerated graphics in the embedded Chrome browser were removed again because they could cause a crash when opening a popup window.

Web
- Caching of non-standard attributes in the embedded Chrome browser was broken in QF-Test version 4.1.3 which could lead to component recognition issues in special cases.

Web
- Many debug messages from GeckoDriver and Mozilla Marionette are now filtered from the shared terminal.

- In some cases clearing the browser cache during startup could cause Edge to hang for up to several minutes.

JavaFX
- The entry for showing available methods was missing from the menu for JavaFX component recording.

Version 4.1.3 - March 29, 2017

Important notes:

- Selenium WebDriver for Firefox only supports the latest two Firefox versions plus the most recent ESR release. At the point of this QF-Test release that means Firefox 51, 52 and 52ESR. We try to improve on that as best we can and still support Firefox 38ESR, 45 and higher, but older Firefox versions simply don't run as well as current ones. There was a major change for Selenium WebDriver between Firefox 47 and 48 and WebDriver based tests run much faster with Firefox 48 upwards. For long-term support we currently recommend to use Firefox version 52ESR.

New features:

Web
- A major focus for this release is performance, especially for web testing. It has been drastically improved for WebDriver mode and JxBrowser and also significantly for web testing in general. The extent of some of the effects depends on the actual application and its implementation.

- Thanks to the performance improvements, recording in WebDriver mode has stabilized and should now produce the same results as in QF-Driver mode.

- Recording and replay of mouse double clicks is now more intuitive and consistent across all GUI engines. In the past, a Mouse event[681] node with event type 'Mouse click' and Click count[684] set to 2 was shown as a 'Double click' event, even though it was only the second of the two clicks required to simulate a double click. Such nodes are now called 'Followup click' and there is a new event type 'Double click' that properly simulates all the individual events for a complete double click in a single node.

- test suites can now be closed by clicking on their tab with the middle mouse button.

- The procedure `qfs.database.executeSelectStatement` in the QF-Test standard library `qfs.qft` has been improved with respect to the usage of a property group for the result set. The group is now cleared before being populated with the current result and the number of records is available via the property 'count'.

- It is now possible to configure fallback values for placeholders in the template file for recording procedures.

- QF-Test now shows the full path to the current test suite as well as the QF-Test version in the title bar of the main window. This can be reduced to just the file name via the option Show complete file path and QF-Test version in the title bar[421].

- WebDriver was updated for the latest Firefox and Chrome versions and QF-Test is now also compatible with the version of Microsoft Edge Browser shipped with the upcoming Windows 10 "Creators Update".

Bugs fixed:

- Up to now resolvers were registered globally, applying to all GUI engines, which could lead to unwanted side-effects in hybrid applications. By setting the GUI engine[633] attribute in the SUT script[631] node that registers the resolver it is now possible to restrict the resolver to the desired engine.

- QF-Test now works around an issue in Jython 2.7 that prevented QF-Test startup in case the user's home directory contained international characters.

- Groovy was updated to version 2.4.8 which fixes a leak of PermGen space in Java 7.

- When stopping recording too quickly, the last recorded events, checks, components or procedures could get lost. Synchronization with the SUT has been improved so that this should now happen only in cases of extreme load.

- When switching between projects the expansion state of the tree nodes in the project view was lost.

- Summary output of a batch run with <u>-verbose [<level>]</u>⁽⁸⁸³⁾ did not conform to the information displayed in the status bar and in the report.

Web • JxBrowser components with preloaded content are properly handled now.

Web • <u>Text input</u>⁽⁶⁸⁹⁾ nodes now also work for embedded browsers (FX WebView, JxBrowser and SWT embedded browsers) if <u>Replay single events</u>⁽⁶⁹¹⁾ is deactivated. Replay of single events also mostly works, but the events have to be simulated as hard events which is currently possible on Windows only.

Web • Some TEXT nodes in a web application were treated as disabled in WebDriver mode.

Web • After page modifications the WebDriver-API did not always assign the correct QF-Test component for a WebDriver element.

Web • Closing a secondary window of an Edge browser could inadvertently terminate the whole browser process. Also, browser shutdown in WebDriver mode has been improved for all browsers.

Web • Several issues for the embedded WebView browser in JavaFX have been fixed, including mutation handling, refresh and goto commands and return values from evalJS.

Web • Firefox settings are now only made persistent in the test profile if the system property "qftest.web.mozilla.save_profile" is set to true during browser startup.

Web • The QF-Test installation now includes libraries for the embedded Chrome browser in QF-Driver mode that are required for hardware accelerated graphics.

SWT • On Linux, tests for Eclipse with embedded JavaFX components could cause a hard deadlock at Gtk level. This has been fixed for Eclipse/SWT versions 4.3 and higher.

Version 4.1.2 - January 13, 2017

New features:

SWT • Support for Eclipse/SWT 4.6 "Neon" has been updated for Eclipse version 4.6
Note SR2. The mechanism for auto-scaling SWT-based UIs on high-DPI monitors is still deactivated by default when the SWT plugin is instrumented by QF-Test. The property to re-enable it has been renamed to `swt.autoScale` in SWT version 4.6 SR2.

- Resolvers are a panacea for all kinds of component recognition problems. To make them easier to write QF-Test 4.1 introduced an updated `resolvers` module which is now complemented by a rewrite of section 52.1.7[(1023)] which makes resolvers more accessible than ever.

- The names of all registered resolvers can now be retrieved by calling the method `resolvers.listNames()`.

- When performing automatic rerun of nodes, the beginning of an attempt is now logged as both a message and an annotation. The doctags for automatic rerun have been extended to enable catching a specific exception and to suppress log messages. See section 23.3[(292)] for further information.

- The QF-Test ID of a Test case[(518)] or Test set[(525)] node can now be retrieved at test execution time via ${qftest:testcase.id} or ${qftest:testset.id}.

- It is now possible to transform Check items[(719)] into Check selectable items[(724)] nodes and vice versa.

- If the value for the command line argument `-usercfg <file>`[(883)] or `-systemcfg <file>`[(881)] points to a directory, the configuration file is now read from or created within the specified directory.

- When editing options QF-Test now shows a warning in case a configuration file cannot be written.

Bugs fixed:

- QF-Test now logs a warning in case the engine of a component retrieved by calling `rc.getComponent` in an SUT script[(631)] node does not match the engine executing the script. Starting with the medium upgrade QF-Test 4.2 this will cause an exception. Trying to pass a component of the wrong GUI engine to `rc.overrideElement` leads to an exception right away.

- When editing the details of a node and closing the test suite without confirming the edit first, the changes were discarded.

- In case a resolver fails with an exception, other registered resolvers are executed anyway, like they were in QF-Test 4.0.

- QF-Test now removes temporary files created by WebDriver implementations.

JavaFX
- Cells in custom Table components with non-standard cell ordering are addressed correctly now.

SWT
- QF-Test could cause a deadlock in Eclipse/SWT applications with multiple Display instances.

Web • When fetching the geometry of items in a web application the values for the parent component were retrieved instead.

Web • For web applications leading and trailing whitespace is now removed from names returned from NameResolvers.

Web • Automatic scrolling for image checks on items was broken in web applications.

Version 4.1.1 - November 23, 2016

New features:

- The project view now comes with a filter at the top of the tree that matches on test suite names and limits the hierarchy accordingly, which is very useful for quickly finding test suites in large projects.

- On macOS all keyboard shortcuts should now be conforming to Mac standards and use Mac-friendly modifier keys.

Web • Selenium WebDriver was updated to version 3.0.1.

- The documentation and samples for keyword driven testing have been extended with a new example for behavior driven testing (BDT, see section 29.4[(363)]).

- The parameters for the @rerun doctag were extended to include a newerror-level=keep option and the new keepfirst parameter. See section 23.3.2[(295)] for further information.

- The Execution timeout[(542)] attribute is now also available for Test step[(539)] nodes.

Bugs fixed:

- QF-Test did not start if the Java version for QF-Test was still instrumented by an older QF-Test version (pre 4.1). Though JRE instrumentation has been deprecated for many years, such a case is still possible. QF-Test now detects that situation and helps with deinstrumentation of the JRE.

- The test for busy panes was missing when performing a check using the generic retry mechanism introduced in QF-Test 4.1.0.

- When interactively selecting a component for an event node the current target was not pre-selected in the chooser dialog.

- Trying to update failed checks from the error list after closing the originating run log caused an unexpected exception.

- When using the `databinder.bindList` API for performing data-driven testing, QF-Test now ignores empty lists instead of running a single iteration without data.

- When importing test results into TestLink, skipped test cases are now correctly reported as skipped.

- The former WebDriver API was renamed to WebDriverConnection (see section 52.12[(1132)]) to avoid confusion with Selenium's WebDriver object.

- The error message for expiration of a `-timeout <milliseconds>`[(883)] command line argument of a batch call now clearly refers to the `-timeout <milliseconds>` argument instead of the root node of the test suite.

- In large projects editing component IDs in parameter tables no longer causes delays while QF-Test is looking for the component to highlight.

- When searching through a project with an invalid regular expression many errors messages were shown instead of just one.

- Short timeouts for visible false checks on components were not honored.

Swing
- If the option Instrument AWT EventQueue[(514)] is active, QF-Test now establishes a connection with the awt:security engine in case Java WebStart shows an authentication dialog. Please note that in such a case the GUI engine[(672)] attribute of the Wait for client to connect[(670)] node must be set to awt:security.

Web
- Class resolution for AJAX toolkits was broken for some special cases.

Web
- The editable check didn't work for TextFields in the PrimeFaces toolkit.

Web
- When recording a click that triggers an alert dialog in WebDriver mode, the order of the recorded events was not always correct.

Web
- Auto-detection for AJAX toolkit resolvers has been improved. It now prints concise messages to the terminal and cleanly handles the case when no toolkit is detected.

Highlights of version 4.1

The following major new features have been implemented for QF-Test version 4.1:

Description	Further info
Additional browsers via Selenium WebDriver	Includes Firefox 44 and up, Edge and Safari as well as Chrome on Linux and macOS (Web testing[200]), includes support for executing Selenium scripts from QF-Test and for mixing both (section 52.12[1132]). Performance and functionality not yet on par with QF-Driver, requires Java 8
Dedicated QF-Test App for macOS	Support for Swing, Java FX and Web (WebDriver mode for Safari, Chrome and Firefox). No SWT support at this point.
Eclipse plugin	Special Run Configurations that combine starting the SUT, launching QF-Test and possibly running initial test steps directly from Eclipse (section 27.1[336]).
Jython update to version 2.7, Groovy to version 2.4.7	See section B.9.7[1309] for incompatibilities and fall-back to Jython 2.5 / Groovy 2.1.
License server mechanism	QF-Test licenses can be floated globally instead of just across the local network. If interested, please get in touch with QFS via `sales@qfs.de`
64 bit browser support on all systems	In QF-Driver connection mode browser and JRE architecture must match (chapter 14[200])

Table B.4: Highlights in QF-Test version 4.1 (part 1)

Description	Further info
Support for the AJAX framework Kendo UI	
Improved synchronization for web tests	QF-Test tracks HTTP requests in the browser to avoid triggering events while AJAX calls are still pending (options <u>Synchronize with HTTP requests</u>[(495)], <u>Timeout for HTTP request synchronization (ms)</u>[(495)] and <u>Extended HTTP request timeout for new documents (ms)</u>[(496)]). QF-Driver mode only.
Web: Support for component resolution via XPath and CSS-selectors	<u>Addressing via XPath and/or CSS selectors</u>[(87)]
Alternative connection mechanism for Swing	Works out-of-the-box in many of the problematic cases with Java WebStart, applets or custom EventQueues (<u>Instrument AWT EventQueue</u>[(514)])
Execution timeout	Limit the maximum execution time for individual sequences, test cases, test sets or whole batch runs. (<u>Execution timeout</u>[(524)] attribute, <u>`-timeout <milliseconds>`</u>[(883)] command line argument)
Rerun capabilities: Re-execution of failed test cases or sequences automatically or manually from the run log.	<u>Re-execution of nodes (Rerun)</u>[(292)]
Merging run logs	<u>Merging run logs</u>[(124)]
Generic retry	Smarter replay that better handles timing issues, especially components that vanish or get replaced just while QF-Test was trying to access them.
Locating procedures or components with variable names/IDs	Variable expansion is now also possible in a static context so that jumping from a procedure call like "checksFor${qftest:testcase.name}" to the correct procedure works. To that end QF-Test even caches the most recent variable values from the last execution.
Updates to the QF-Test manual include new chapters about keyword-driven testing and load-testing as well as a major rewrite of important topics and more comprehensive documentation for QF-Test's pseudo DOM API and for creating resolvers of all kinds.	<u>Organizing the test suite</u>[(129)], <u>Dependency nodes</u>[(138)], <u>Keyword-driven testing with QF-Test</u>[(352)], <u>Performing GUI-based load tests</u>[(375)], <u>Pseudo DOM API</u>[(1118)], <u>The `NameResolver` Interface</u>[(1023)]

Table B.5: Highlights in QF-Test version 4.1 (part 2)

Changes that can affect test execution

Jython 2.7

The upgrade to Jython 2.7 was necessary due to conflicts with native libraries in

the WebDriver module. Besides, Jython 2.5 was quite outdated and the Python modules coming with Jython 2.7 are more modern and mature. If the potential issues listed below are causing too much trouble, you can switch back to Jython 2.5. To do so, please download the file https://archive.qfs.de/pub/qftest/jython-2.5.1.jar and use that to replace the file `lib/jython.jar` in QF-Test's installation directory.

- Java 7 or higher required.
- No support for wildcard import (e.g. `from java.util import *`), all imported classes need to be listed explicitly.
- You can find more differences between Jython 2.5 and Jython 2.7 in the release notes of Jython 2.6 and Jython 2.7. Please find the release notes of Jython 2.6 at https://docs.python.org/3.0/whatsnew/2.6.html and the release notes of Jython 2.7 at https://docs.python.org/3.2/whatsnew/2.7.html.

Groovy 2.4
This upgrade was not strictly necessary, but it is prefereable not to fall too far behind the official Groovy Release. Groovy 2.4 still mostly works with Java 6, so the only drawback is if you still need support for a legacy application based on Java 5. In that case you can download the file https://archive.qfs.de/pub/qftest/groovy-all-2.1.0.jar and use that to replace the file `lib/groovy-all.jar` in QF-Test's installation directory.

- Java 6 or higher required.

New Swing connection mechanism
This mechanism should be able to handle even very tricky Swing SUTs that implement their own EventQueue, especially in combination with WebStart or applets. It is active by default. In case it does cause problems with your SUT it can be disabled via the option Instrument AWT EventQueue[(514)].

Chrome no longer supports Java applets
Google has completely dropped support for Java applets in Chrome. This includes the Chrome browser embedded by QF-Test which was updated to CEF version 45.

Software that is no longer supported

Note Please see section 1.1[(3)] for a detailed list of system requirements and supported technology versions.

QFS is taking pains to support even very old software. QF-Test should still run, for example, under Windows XP or with an SUT based on Java 5 or 6 but for systems that are no longer supported by their own vendors we cannot provide full support.

Libraries that are required for the Firefox and Eclipse/SWT versions listed below are no longer part of the default installation of QF-Test. The majority is still being maintained, albeit with limited testing. For Eclipse/SWT 3.5 - 3.8 simply download https://archive.qfs.de/pub/qftest/swt_legacy.zip and extract the contents into the `swt` directory of your QF-Test installation. For support of older browsers, please get in touch with QFS via `support@qfs.de`.

Official support has been dropped for Java 5, Java 6, Internet Explorer 10 and older, Firefox 37 and older as well as Eclipse/SWT 3.8 and older.

Version 4.1.0 - October 6, 2016

New features:

- The rerun capabilities are now officially available and documented in <u>section 23.3</u>[(292)].

- The QF-Test Jenkins plugin was updated for Jenkins version 2.1.

- On macOS QF-Test now uses the standard key bindings for that system. Users that prefer Windows bindings (or simply have gotten used to QF-Test's non-Mac bindings) can keep them via the command line argument <u>`-keybindings=classic`</u>[(873)].

- The QF-Test macOS App now includes a bundled JRE so manual JRE installation is no longer required.

Web
- Support for the AJAX toolkit Vaadin was updated so that the Vaadin resolver can now be customized as described in <u>section 49.1</u>[(945)].

- When debugging tests, QF-Test now shows a warning when 'Continue execution from here' is attempted on a node that is not part of the current execution stack as doing so would lose the current variable bindings.

- Chooser dialogs now include a search filter.

- It is now possible to directly highlight the target component via the popup menu for QF-Test IDs contained in variable tables or `rc.getComponent` statements in SUT scripts.

Bugs fixed:

- The QF-Test Java Configuration Tool (qfconfig) always displayed the default Java program upon startup.

- The small arrows showing tree node execution state were not always displayed correctly.

- Errors resulting from execution timeout are now shown correctly in the report.

Web - In WebDriver mode QF-Test now avoids starting two Chrome instances with the same user profile directory.

Web - Jars in the QF-Test plugin directory could interfere with the startup of a WebDriver based SUT.

Web - Alerts can now be handled on all WebDriver based browsers.

Version 4.1.0-rc2 - September 2, 2016

New features:

- Groovy was updated to version 2.4.7. QF-Test now requires at least Java 7 unless Groovy is rolled back to version 2.1.0 as explained in section B.9.7[(1309)].

Web - WebDriver support is significantly improved and performance is much better. Firefox 48 and higher now also work as Selenium was upgraded to support for Firefox Marionette.

Web - For web QF-Test now supports component resolution using XPath and CSS selectors. See Addressing via XPath and/or CSS selectors[(87)] for details.

- The procedure `qfs.utils.xml.compareXMLFiles` was improved with better filtering and the ability to ignore missing tags or attributes when specified in the "noCheck" parameter.

Web - On Windows, Text input nodes can now be replayed for JxBrowser.

- The procedure `qfs.utils.email.sendEmail` in the standard library `qfs.qft` now also supports attachments.

- Jython was modified to catch and suppress `NotImplementedErrors` caused by implementing only parts of an interface which was OK with Jython 2.5. This behavior can be modified by setting the system property `qftest.workarounds.abstractMethodsHandling` to one of the values "ignore", "warning", "error" or "exception".

Bugs fixed:

JavaFX
- Uncaught exceptions on the JavaFX application thread were correctly reported by QF-Test, but in some cases they were not passed on which might hide bugs in the SUT or at least prevent testing the SUT's own handling of uncaught exceptions.

- Occasionally a call to `DaemonLocator.locateInstance(long timeout)` might locate a daemon, but "forget" it again before returning the result after the timeout was reached.

Version 4.1.0-rc1 - July 29, 2016

Changes compared to QF-Test version 4.0.11:

New features:

Web
- The new browser connection mode based on Selenium WebDriver adds support for additional browsers including Firefox 44 and up, Edge and Safari as well as Chrome now also on Linux and macOS.

- Selenium scripts can now be run from inside QF-Test and mixing of QF-Test and Selenium tests is possible. See section 52.12[(1132)] for how to access the WebDriver API from SUT script nodes.

- QF-Test now officially supports Swing, JavaFX and Web testing on macOS. A dedicated installer is available.

- Jython was updated to version 2.7.0. QF-Test now requires at least Java 7 unless Jython is rolled back to version 2.5.1 as explained in section B.9.7[(1309)].

- By using our new license server mechanism, licenses can now be floated globally instead of just across the local network. If interested, please get in touch with QFS via `sales@qfs.de`.

Web
- 64 bit browsers are now supported on all systems.

Web
- Support was added for the AJAX framework Kendo UI.

- Many existing AJAX resolvers were updated and generalized and are now easier to customize. See Special support for various web frameworks[(988)] and Improving component recognition with a `CustomWebResolver`[(945)] for further information.

Web
- In QF-Driver mode QF-Test can now synchronize web tests based on outstanding HTTP requests. This is controlled via the options Synchronize with HTTP requests[(495)], Timeout for HTTP request synchronization (ms)[(495)] and Extended HTTP request timeout for new documents (ms)[(496)].

- The new connection mechanism for Swing works out-of-the-box in many of the problematic cases with Java WebStart, applets or custom EventQueues. It can be disabled via the option Instrument AWT EventQueue⁽⁵¹⁴⁾.

- The maximum execution time for sequences, test cases, test sets or whole batch runs can now be limited via the attribute Execution timeout⁽⁵²⁴⁾ and the command line argument `-timeout <milliseconds>`⁽⁸⁸³⁾.

- Run logs can now be merged as described in Merging run logs⁽¹²⁴⁾.

- Event replay for all engines now uses an approach we call Generic Retry that better handles timing issues, especially components that vanish or get replaced just while QF-Test was trying to access them.

- Variable expansion is now also possible in a static context so that jumping from a procedure call like checksFor${qftest:testcase.name} to the correct procedure simply works. QF-Test caches the most recent variable values from the last execution so this is useful even for dynamic bindings.

- The various data nodes like Database⁽⁵⁶⁹⁾ or Excel data file⁽⁵⁷⁴⁾ can now also be executed outside a data driver context in order to bind the values directly as variables instead of iterating over the sets. In that case the Name⁽⁵⁷⁰⁾ attribute of the node also defines the name of the properties group into which the variables are bound.

- Support for JxBrowser was updated to JxBrowser version 6.

- Depending on screen layout it may be preferable to use more space for the tree or for the terminal in QF-Test's user interface. The new menu item View→Terminal→Prefer tree over terminal toggles whether tree or terminal should occupy the lower left region.

- Improved packing of mouse events should result in cleaner recordings with fewer stray events. For special cases where individual `MOUSE_PRESSED` or `MOUSE_RELEASED` events are preferable, disable the option Simplified 'Mouse click' recording⁽⁴⁴¹⁾.

- Several chapters in the manual were added (Keyword-driven testing with QF-Test⁽³⁵²⁾, Performing GUI-based load tests⁽³⁷⁵⁾) or rewritten (Organizing the test suite⁽¹²⁹⁾, Dependency nodes⁽¹³⁸⁾ and Pseudo DOM API⁽¹¹¹⁸⁾).

- The embedded script editor now supports the keyboard shortcuts Shift-Up/Down to move one or more selected lines up or down and , Shift-Return to add a new line after the end of the current line.

- There is a new toolbar button for opening the most recent run log.

- Failed boolean checks have a more specific error message that includes the type of check.

- The `DOMProcessor` extension API now also supports modifying summary documents.

- Script APIs are being unified in preparation for additional scripting languages. Implementing and registering resolvers is now easier and more flexible and the autowin and ssh modules are now also available in Groovy.

Web

- There is now a generic string check type for DOM node attributes that can be read via `getAttribute`. Simply set the Check type identifier⁽⁷¹¹⁾ to `attribute:NAME` where `NAME` is the name of the attribute to check.

- Pressing Ctrl-C in the project tree or on the root node of a test suite copies the path of the test suite to the system clipboard.

- Locating references now also works for Package⁽⁵⁹⁴⁾ nodes. The resulting list contains references to all Procedures⁽⁵⁸⁶⁾ in the respective hierarchy.

- Pasting nodes in a test suite is more flexible. If the current location is not suitable for the kind of node to be pasted but the target is implicitly clear, the node gets automatically inserted at the correct position.

Web

- The version of the Chrome browser provided with QF-Test for QF-Driver mode web tests was update to CEF 45.

- The procedures `qfs.utils.getDate`, `qfs.utils.getTime` and `qfs.check.compareDates` in the standard library `qfs.qft` now all have parameters to specify the locale to use for formatting.

- All `getRowCount` procedures for Swing, JavaFX and SWT trees have been reworked to provide the number of visible rows. The new parameter "countOnlyVisibleNodes" limits the search to visible rows below a given node. Also added were procedures `...tree.getChildNodeCount` with a similar functionality and `...tree.check.checkChildNodeCount` for checking the number or child nodes of a node.

- The directory for exchanging files with an external editor can now be explicitly specified via the option Directory passed temporary files to external editor⁽⁴³²⁾.

- Some of the tables in QF-Test detail views were rearranged to increase visibility. Besides, all tables now support adding a new row by double-clicking in the empty area of the table.

Bugs fixed:

JavaFX
- Updates to the ID of a JavaFX node were not tracked correctly.

JavaFX
- The "imagehash" extra feature is now also implemented for JavaFX.

- Internal storage of images in test suites and run logs is now much more memory-efficient.

Web
- Tests for enabled DOM nodes now also take enclosing `FIELDSET` nodes into account.

Web
- Association of labels via the "for" attribute is now better supported.

- Using the debugger command "Continue from here" on a Procedure called from different test suite could mess up the current execution context.

B.9.8   QF-Test version 4.0

Version 4.0.11 - April 05, 2016

New features:

SWT
- Support for Eclipse/SWT 4.6 "Neon" has been added based on Eclipse version 4.6M6.

Note
The new mechanism for auto-scaling SWT-based UIs on high-DPI monitors is not supported by QF-Test yet and is deactivated by default when the SWT plugin in instrumented by QF-Test. It can be forced active by setting the system property `swt.enable.autoScale=true` but in that case test execution will be limited.

- In case of an error or exception on Windows, information about the current Windows session is now added to the diagnostic log entry in the run log. This should help understanding failures caused by a locked session as described in the FAQ " Hard mouse events and drag'n'drop operations do not work flawlessly, components cannot be found, the run log contains black or corrupt screenshots. What do I have to take care about for test execution? A[1223]".

Bugs fixed:

- QF-Test now again starts correctly with early access versions of Java 9. This was broken after changing the Java 9 versioning scheme. The value "9-ea" for the system property `java.version` does not contain the '.' character that QF-Test expected to find after the main version.

- In some cases the record button could remain active even after the last SUT client terminated.

• A few keystrokes for certain characters - e.g. '.' - were not replayed correctly as single events in QF-Test's Chrome browser.

• When trying to launch a WebStart application in QF-Test's Chrome browser, the download dialog for the JNLP file appeared instead.

• Automatically restoring a minimized SWT or web browser window before replay on Linux could trigger a Gtk error that caused the SUT to crash.

Version 4.0.10 - January 28, 2016

New features:

• Support for the AJAX toolkits ICEfaces and PrimeFaces was updated for ICEfaces version 4.1 and PrimeFaces version 5.3.

• Automatic handling of security dialogs in Java WebStart and applets now includes support for a new dialog that was added in Java 8 update 72.

• The new package `qfs.autoscreen.fx` in the standard library `qfs.qft` provides procedures for image based component resolution in a JavaFX application.

Bugs fixed:

• Exceptions in the `toString` method of a component in the SUT could prevent component recognition and lead to a `ComponentNotFoundException`.

• During QF-Test startup or when saving or loading a file, the list of recently opened files gets updated. If that list contained entries from slow or no longer valid network drives this could take an unreasonably long time during which QF-Test was blocked.

• The procedures in the package `qfs.autoscreen` now throw a `TestException` with detailed information in case an invalid algorithm syntax is specified by the caller.

• QF-Test now correctly handles images with a width or height greater than 8192 pixels and handling of large PNG images is faster.

• Occasional exceptions that were thrown when sorting a column in a result list have been fixed.

• A search in the - rarely used - separate debugger window could cause an exception.

Web   • Using `rc.overrideElement` in a web based SUT could cause a `NullPointerException` in some special cases.

Web   • Thanks to miscellaneous performance fixes, tests for large web pages should run a bit faster, most notably on Firefox.

Version 4.0.9 - December 17, 2015

New features:

Web   • Support was added for Firefox version 43.

Bugs fixed:

• QF-Test now cleanly handles exceptions caused by Excel files with functions that are not directly supported by Apache POI.

JavaFX   • When checking the image of a JavaFX component QF-Test now makes sure that the target is scrolled into view first and that geometry changes from focus borders are compensated.

JavaFX   • Synchronization with JavaFX events during replay has been further improved.

Web   • Text input in a browser could get recorded as a mix of text input and keystroke events.

Web   • For the AJAX framework ZK it is now possible to check the enabled/disabled state of TabFolder items.

Web   • Recognition of web components was broken in case of leading or trailing whitespace in the name of a FRAME node.

Web   • Setting the browser's user agent during startup now also works for Chrome.

Version 4.0.8 - November 4, 2015

New features:

Web   • Support was added for Firefox version 42.

Web   • Some web components with JavaScript code that handles events in special ways do not react to "soft" mouse events. Besides "hard" events that actually move the mouse cursor, QF-Test also implements "semi-hard" events by sending events at OS level to the browser window. The new procedure `qfs.web.general.clickIncreasinglyHard` in the standard library `qfs.qft` makes it easier to choose the best method and keep the tests robust.

Bugs fixed:

- Reading the output of a client process via `${qftest:client.output.<name>}` immediately after the process terminated could occasionally fail and return an empty value.

- The option settings for whether to record generic classes and system classes should have no impact on test replay. However, there was one special exception in case of Swing and SWT: If the 'Feature' attribute of ComboBox popups and lists was of the form 'Invoker class: ...', it was matched only against the class resulting from the current setting of the above options. Now the match is correctly performed against all classes of the invoker component.

- The extra feature 'qfs:class', which holds the most specific class name of a component, is now always recorded and not only if recording of generic classes is activated.

- When performing a successful search operation and starting to edit the highlighted attribute, the modification was lost when continuing the search without confirming the changes.

- When reading data from a custom formatted cell in an Excel sheet that references a non-existing worksheet or file, the custom format was not applied correctly. Also, handling of cells that contain errors has been improved.

JavaFX
- JavaFX Tables and TreeTables with nested column headers are now handled correctly. Also, automatic horizontal scrolling for hard events and image capturing was not quite accurate for these components.

JavaFX
- QF-Test now works around a Java issue where capturing an image of a JavaFX component can mess up the internal geometry of a JavaFX client.

JavaFX
- Recording events in an embedded WebView browser has been improved. Most notably the first event after a page navigation and in general navigation in frames were not handled correctly.

Web
- In special cases registration of the Ext JS resolver could delay startup of a web application up to 20 seconds.

Web
- Even when specifying a dedicated Java program in the Executable[648] attribute of the 'Start browser' node, QF-Test still took some definitions from its own Java version which might cause problems with applets running in the embedded browser.

Version 4.0.7 - August 27, 2015

New features:

Web
- Support was added for Firefox version 41.

Web
- Support for the AJAX toolkit Ext JS was updated for Ext JS version 6.

Bugs fixed:

- Memory usage for creating an XML or HTML report from a split run log has been further reduced.

- Step-by-step replacement of parameters or variable definitions could replace all matching values of a node instead of just the ones chosen.

JavaFX
- For some JavaFX components ItemName- and ItemValueResolver were either not separated cleanly or had no effect.

Swing
- Changes in the Java 9 API were causing problems with forced raising of windows and with replay of a file selection on the AWT FileDialog.

SWT
- Replay of a selection event on a CCombo element used inside a Tree or Table cell could fail due to focus issues.

Web
- Internet Explorer 10 and older could crash, deadlock or weirdly misbehave when running in quirks mode on pages with many nested IFRAMEs.

Web
- QF-Test's embedded Internet Explorer now treats JavaScript code in href attributes in the same way as the normal Internet Explorer.

Version 4.0.6 - July 16, 2015

New features:

- Windows 10 is now officially supported. Internet Explorer in particular was causing a number of problems, ranging from crashes to broken documentMode, userAgent or language settings which should all be fixed now.

Note
  Microsoft's new browser Edge is not supported yet.

SWT
- Support for Eclipse/SWT 4.5 has been updated for the final release of Eclipse 4.5 "Mars".

Web
- Experimental support was added for Firefox version 40.

Web
- Support for the AJAX toolkit RAP was updated for RAP version 3.0. For this version - and back to RAP 2.2 - names set by development via `widget.setData("name", ...)` can now be retrieved automatically by QF-Test, similar to handling names in SWT.

Bugs fixed:

JavaFX
- For JavaFX on Mac OS X QF-Test is now able to force the focus onto the SUT's windows, which is a prerequisite for stable and reliable test execution.

JavaFX
- JavaFX Tables and TreeTables with invisible columns are now handled correctly.

JavaFX
- Sequences recorded for editable JavaFX ComboBoxes were missing the popup Button and thus not replaying correctly. The Button and TextField inside the editable ComboBox are now automatically addressed as combo@:Button and combo@:TextField. Value checks for ComboBoxes were also improved.

JavaFX
- Custom headers in JavaFX table columns are now handled correctly.

Web
- On Windows the final release of Firefox 39 crashed with QF-Test 4.0.5 due to a change in the memory management of Firefox.

Web
- Firefox version 38.1 ESR was not recognized correctly by QF-Test 4.0.5.

SWT
- Geometry of SWT Widgets inside a titled Group was incorrect for SWT versions 4.4. and 4.5 on Linux.

Version 4.0.5 - May 12, 2015

New features:

Web
- Support was added for Firefox versions 38 and 39.

- A new standard check for tool-tips has been added.

JavaFX
- For JavaFX the synchronization of QF-Test event replay with the JavaFX toolkit and its event handling and repaint mechanism has been significantly improved, which makes test replay more robust.

- The new Procedure `qfs.utils.dumpComponents` in the standard library `qfs.qft` logs a component (sub-)hierarchy including geometry and class information which may be helpful during test development.

- Drag&Drop operations between components of different GUI engines, independent applications or the SUT and the desktop can now be performed with the help of the Procedure `qfs.utils.dragAndDrop`.

Bugs fixed:

- Creating a thread dump of the SUT is also working for 64 bit Java VMs on Windows now.

- When recording procedures with a template file that contained an empty parameter list for container procedures, the resulting procedure calls were broken so badly that the test suite could no longer be saved.

- Memory usage for creating an XML or HTML report from a run log containing externalized logs has been significantly reduced.

Swing
- Event replay for Swing, which was noticeably slower in QF-Test version 4.0.4, is now back to normal speed.

JavaFX
- In a few special cases the replay of mouse events on JavaFX menus was broken with Java 8 update 40.

JavaFX
- SUT connection with the FX engine now is being delayed until the FX toolkit has been fully initialized. This enables support for applications with embedded FX components that create the FX bridge during startup but initialize the FX toolkit later when the first FX component is shown.

JavaFX
- A possible deadlock for combined Swing and FX applications with a shared event dispatch thread has been fixed.

Web
- Class resolution despite compressed or obfuscated JavaScript has been significantly improved so that web applications based on the AJAX toolkit GWT are running much faster now.

Web
- In some cases, the check recording mode in the Chrome browser was not exited cleanly so that subsequent event recording failed.

Web
- Due to a rounding issue, image checks in the Chrome browser were sometimes one pixel off.

Web
- When checking text in web applications, combinations of NBSP and normal space were not always resolved correctly.

Web
- The QF-Test browser was leaking memory when a nested FRAME or IFRAME was unloaded during page navigation.

Version 4.0.4 - March 5, 2015

New features:

Web
- QF-Test now supports testing 64bit versions of Firefox on Linux.

Web
- Dedicated support was added for the AJAX UI framework jQuery EasyUI.

Web
- Support for Firefox has been updated to match the public release of Firefox version 36 and support for Firefox version 37 has been added.

SWT
- Support for Eclipse/SWT 4.4 "Luna" has been updated for the Eclipse 4.4 SR2 release.

Web
- The Java version for starting QF-Test's browser wrapper is no longer limited to the one QF-Test itself is using. It can now be set in the attribute Executable[(648)] of the 'Start browser' node. This makes it easier to mix 32bit and 64bit Java versions or to use the System JRE for the browser in order to support embedded Java applets.

- The special variable syntax ${qftest:32} (or its alias ${qftest:32bit}) can be used to test whether QF-Test is running in a 32bit Java VM, with the obvious counterparts ${qftest:64} and ${qftest:64bit} to check for 64bit VM.

- The new command line argument `-daemonrmiport <port>`[(871)] can be used to run the QF-Test daemon behind a firewall by specifying a dedicated port for the daemon's RMI communication in addition to the daemon lookup port that is set with `-daemonport <port>`[(871)].

- The default value of the option Wait for non-existent component (ms)[(480)] is now temporarily set to at least 30000 ms during SUT startup. Adding a Wait for component to appear[(771)] node to the SUT startup sequence is still recommended, though less emphatically.

Bugs fixed:

- When automatically updating references after refactoring, multi-level sub-item syntax was cut off after the second item level.

- The new command 'Find/Remove unused callables' treated procedures called only from dependencies as unused.

Swing
- The workaround for the Java 8 focus issues on Windows now handles the root cause instead of trying to keep the side effects under control.

Web
- The standard library procedures in the package `qfs.web.browser.settings` now explicitly check the browser parameter for validity instead of failing silently with no effect.

Web • Chrome is now able to record Server HTTP request[(802)] and Browser HTTP request[(808)] nodes.

Web • Chrome could crash when navigating to a previously visited page anew.

Web • In some cases auto-expansion of tree nodes in Chrome could cause a deadlock.

Web • When creating a new startup sequence with the quickstart wizard for the AJAX toolkit GWT the version for the GWT resolver was incorrectly set to 1 instead of 2. The same problem affected auto-detection for GWT.

Web • The AJAX toolkit jQuery UI was not handled by the auto-detection mechanism for AJAX resolvers.

Version 4.0.3 - January 29, 2015

New features:

Web • Support for Firefox has been updated to match the public release of Firefox version 35 and support for Firefox version 36 has been added.

Web • The resolver for the AJAX framework GWT has been rewritten from scratch and supports a far wider range of GWT components based on GWT version 2.7.

SWT • Support for Eclipse/SWT 4.5 "Mars" has been added based on Eclipse version 4.5M4.

JavaFX • Recording of Drag&Drop is now implemented for JavaFX also.

JavaFX • In addition to JavaFX components embedded in Swing, hybrid applications that work the other way round and embed Swing components in JavaFX are now also supported. This includes support for sharing the same event dispatch thread between JavaFX and Swing.

• The new operation "Find/Remove unused callables" has been introduced in order to get rid of procedures which are never used in your test suites.

• The new procedure `qfs.check.compareDates` in the standard library `qfs.qft` can be used to compare two date strings in arbitrary format.

• The procedure `qfs.utils.xml.compareXMLFiles` has two new parameters for stripping white space and for handling namespace definitions.

• The diagnostic information logged for every error and exception now also includes a full thread dump of the SUT as well as the SUT's Java memory usage and the current system load.

- When executing QF-Test in batch mode for report generation or other commands that don't run any tests, QF-Test now runs in AWT headless mode if the command line argument $\underline{-\texttt{nomessagewindow}}^{(874)}$ is specified. This is useful when using such commands from a continuous integration server that runs without a display.

- $\underline{\text{Server HTTP request}}^{(802)}$ nodes can now use the syntax `http://user:passwd@mydomain.com` to send requests to secured servers.

- The default Java memory setting for QF-Test is now 512 MB and the default Java memory for QF-Test's browser wrapper has been increased to 256 MB.

- The new `ResetListener` extension API can be used to influence the behavior of the command │Run→Reset everything│, e.g. to prevent a certain client process from being killed or to perform additional cleanup like deleting Jython or Groovy variables. See $\underline{\text{section 52.8}}^{(1091)}$ for details.

Bugs fixed:

- Changing the order of $\underline{\text{Catch}}^{(619)}$ nodes under a $\underline{\text{Try}}^{(616)}$ node was broken by the addition of the optional $\underline{\text{Else}}^{(613)}$ node in QF-Test version 4.0.2.

- Display of large amounts of output from the SUT in the terminal could slow down QF-Test and the test execution if rich text formatting was activated for the terminal.

- In very rare cases test execution in batch mode could hang on Windows when the SUT ran into a deadlock.

- Depending on timing, QF-Test sometimes did not kill all process clients when exiting.

- Importing test suites could be very slow if tolerant class matching was activated.

- If an Excel sheet contains too deeply nested functions, QF-Test will now throw a TestException instead of ignoring those cells. Handling such Excel files requires an increased thread stack size which can be achieved by starting QF-Test with the command line argument -J-Xss512k.

- When recording procedures, some placeholders in nested sequences in the template were not replaced correctly.

- The procedure recorder is now able to create container procedures with standard event nodes and sequences without relying on component-specific procedures.

- If an HTTP exception was thrown from a $\underline{\text{Server HTTP request}}^{(802)}$ node due to to status code > 399, the variables 'responseDate' and 'statusCode' were not set correctly.

- The horizontal scroll bar of the internal script editor was not set correctly when hard TAB characters were contained in the code.

- Waiting for the absence of a multi-level sub-item now works correctly.

- For WebStart applications QF-Test now also automatically handles the German version of a possible HTTPS certificate warning.

- HTML reports, testdoc and pkgdoc documents could get scrambled by HTML comments split across several lines if HTML pass-through was activated.

- The declaration and encoding of XML reports, testdoc and pkgdoc documents were inconsistent if the default file encoding of QF-Test's Java VM was not ISO-8859-1.

- The tool for the "QF-Test Java Configuration" could not save values to the Windows registry if setup.exe was never run before.

Swing
- When recording in Swing applications with a great number of deeply nested components, performance could suffer severely.

Swing
- In Swing applications running on Java 8, bringing up the menu for check recording could subsequently block input into text fields.

Swing
- The workaround for focus issues in Java 8 on Windows when changing the topmost state of a window has been improved to handle another special case.

JavaFX
- For hybrid JavaFX and Swing applications replaying an event on an embedded component now correctly raises and activates the surrounding window of the other toolkit which improves the reliability of such tests.

JavaFX
- Replaying a file selection in JavaFX for a save dialog now also sets the ExtensionFilter to match the chosen file's extension.

SWT
- Trying to record a check directly on the header of an SWT Table caused an exception.

Web
- Third-party plugins and extensions were not initialized correctly for Firefox 30 and above.

Web
- Resolving list items now also works for SELECT nodes and generic ComboBox elements that are located in another list.

Web
- The resolver for the AJAX framework ZK has been updated to version 1.1.1 which fixes a few minor issues and improves handling of MenuItems.

Web
- Playback of semi-hard mouse events with modifiers like Ctrl has been fixed.

SWT
- Checks for table cells in KTable components were not recorded correctly.

Version 4.0.2-p2

New features:

Web
- Support for Firefox has been updated to match the public release of Firefox version 34 and support for Firefox version 35 has been added.

Bugs fixed:

Web
- In QF-Test version 4.0.2-p1 the user agent string returned by Firefox browsers was incomplete.

Version 4.0.2-p1

Bugs fixed:

Web
- Support for Firefox version 33 has been updated to cope with the new Firefox versioning scheme introduced with Firefox 33.1.

Version 4.0.2 - October 22, 2014

New features:

Web
- Support was added for Firefox versions 33 and 34.

SWT
- Support for Eclipse/SWT 4.4 "Luna" has been updated for the Eclipse 4.4 SR1 release.

JavaFX
- Support for JavaFX has been updated for some FX changes in the Java update 8u20.

- The QF-Test Agent mechanism is now compatible with the tighter security restrictions introduced with Java updates 8u25 and 7u71/7u72.

JavaFX
- JavaFX Charts are now fully supported with improved representation of the component parts and various dedicated checks.

Web
- Support for the AJAX toolkits Ext JS and PrimeFaces has been updated to include support for Ext JS version 5 and PrimeFaces version 5.

- The backwards compatibility for recording components with generic classes into test suites with pre-QF-Test-4.0 components has been extended to include support for merging such components via the import feature.

- The Try[616]/Catch[619]/Finally[623] construct has been extended to support an Else[613] node which is executed in the case when the contents of the Try are executed cleanly without throwing an exception.

- Window[812] nodes now also support Extra feature[814] attributes, just like Component[823] and Web page[818] nodes.

- It is now possible to override the way how QF-Test determines generic classes by implementing a `GenericClassNameResolver` as described in section 52.1.8[1026].

- There is now a dedicated section in the manual explaining how to quickly implement resolvers for AJAX web toolkits that are not yet supported out-of-the-box: section 49.1[945].

Bugs fixed:

- In some cases the record button could remain active even though no SUT client was connected to QF-Test anymore.

- On a Windows machine with only a 64bit Java installed, QF-Test now explicitly searches for the 64bit Java installation in "C:\Program Files" in case it does not find a `javaw.exe` program on the default PATH.

- The procedures in the `qfs.utils.ssh` package did not use the correct default `known_hosts` file.

- The Excel data file[574] node now handles the character encoding of old-style .xls files correctly.

- The runid was not stored correctly in a run log in case a batch test was run with split run logs disabled via the command line argument -splitlog=false[880].

- Check visible tests for items could incorrectly fail with an `IndexNotFoundException` instead of reporting a passed or failed check.

- The QF-Test editor for the proc-builder configuration file could get corrupted while recording new procedures and editing the file in parallel.

- The context diff for a failed Check items[719] was incomplete in some rare cases.

- Startup for WebStart and applets has been made more robust, including a fix for a `ClassCastException` occasionally seen in applets.

- The newly introduced support for custom tab components assigned via `JTabbedPane.setTabComponent` could break replay of existing recordings.

SWT
- Cells of a Nebula Grid component can now be recorded correctly from a scrolled location and also addressed via negative indexes.

JavaFX
- Recording of events on FX controls embedded in an SWT based SUT no longer produces duplicates and checks can now be recorded normally for such FX controls.

Web
- The default compatibility mode for Internet Explorer 10 and up has been fixed and the user agent returned by IE for `document.documentMode` now always correctly reflects the current compatibility mode.

Web
- Waiting for a dynamically inserted FRAME and its document to be loaded now works correctly in all browsers.

Web
- QF-Test now uses a different method for determining the geometry of TEXT nodes in all browsers that is less invasive and more efficient. The old method could occasionally interfere with JavaScript code in web pages.

Web
- There was a tricky race condition mostly seen in connection with a special kind of download, triggered by opening a temporary web page which automatically closes immediately. QF-Test now handles such situations and the respective download cleanly.

Web
- The Chrome browser now displays a confirmation dialog when a window gets closed and an onbeforeunloadhandler is defined.

Web
- Component and procedure recording in the Chrome browser is considerably faster.

Web
- The embedded Chrome browser can now show videos.

Web
- Double click events are now correctly recorded in the Chrome browser.

Version 4.0.1 - August 28, 2014

New features:

JavaFX
- The JavaFX `TreeTableView` component, which was introduced with Java 8, is now fully supported.

Swing
- QF-Test now supports recording and checking of tabs for `JTabbedPane` components which use the `setTabComponent` method to implement some or all tabs as actual components.

- If the File encoding[581] attribute of a CSV data file[579] node is set to "UTF-8", QF-Test now tries to determine itself whether the file starts with a Byte Order Mark (BOM) or not and handle it accordingly. QF-Test also treats an File encoding attribute with "x-UTF-8-BOM" or "x-UTF-8-No-BOM" correctly, even though Java has no direct support for these encodings.

- When moving the mouse over any image stored in a test suite or run log, QF-Test now displays the (X,Y) coordinates and color values of the associated pixel.

Bugs fixed:

- The timeout value of the option Wait for non-existent item (ms)[480] was ignored in some cases, e.g. when performing checks with a timeout.

- An Extra feature using variables for a regular expression value could mistakenly trigger a BadRegexpException.

- In QF-Test version 4, leading and trailing whitespace is removed from sub-item names. Playback compatibility was provided for most cases, but not all. This should now work consistently.

- The context diff for a failed Check selectable items[724] node in the run log now includes visual representation for the expected and actual selected states and is shown correctly also in cases where there are differences only in the selected state.

- Formulas in Excel files that return boolean values or that switch cell types more than once are now handled correctly by the Excel data file[574] node and the procedure `qfs.utils.files.readExcelFile`.

Swing   - In a hybrid Swing and JavaFX application, handling of Swing `JTable` components was partly broken due to interference from the JavaFX ItemResolver for tables.

JavaFX   - Recording of items in a multi-rooted JavaFX tree (i.e. a tree with its primary root node hidden) is now correct.

SWT   - Component highlighting was broken when recording checks in an Eclipse / SWT application.

SWT   - There were still dependencies on a newer version of glibc left in the instrumentation files for 64bit versions of Eclipse, causing failures on some Linux systems with an old version of glibc - for example RHEL 6.5.

Web   - Tests with Firefox versions 27 and 28 did not work correctly.

Web   - Text checks for table headers were broken in some web tables.

Web   • A proxy auto-config (PAC) file defined in a procedure call of qfs.web.browser.settings.setProxy is now interpreted correctly.

Web   • The Chrome browser now displays a confirmation dialog when a window gets closed and an onbeforeunload handler is defined.

Web   • JavaScript errors in the Chrome browser are no longer printed to the terminal but written to the run log.

Web   • In the Chrome browser coordinates inside an IFRAME were displaced if the frame had a border defined, leading to offsets when highlighting components and to image check failures caused by insufficient automatic scrolling.

Web   • Drag&Drop via the procedure qfs.web.general.dragFromTo now also works with Chrome.

Web   • Clearing the browser cache did not work with Firefox version 32.

Web   • The procedure qfs.web.table.getRowCount could fail in the rare case when the first column of a table is invisible.

Main new features in version 4

Note   For a detailed list of new features please see the release notes for QF-Test versions 4.0.0-beta1 through version 4.0.0 below.

The following major new features have been implemented for QF-Test version 4:

Description	Further info
New GUI engine: JavaFX	Full support for JavaFX based applications
Support for Chrome browser	Currently Windows only
Improved support for Java WebStart and applets	Fast and reliable startup and connection via Java agent
Support for the AJAX framework jQuery UI	Special support for various web frameworks[(988)]
Generic classes for components	Uniform classes for all GUI engines (section 5.4.1[(56)])
Multi-level sub-item concept	QPath concept, similar to XPath (section 5.9.2[(86)])
Support for embedded browsers	JavaFX WebView and JxBrowser 4
Improved user interface with colored syntax highlighting	Option Syntax highlighting for tree nodes[(426)], Terminal options[(463)]
New package for client-side ssh connections	Standard library qfs.qft, package qfs.utils.ssh

Table B.6: Highlights in QF-Test version 4

Changes that can affect test execution:

- The Wait for component to appear[(771)] node implicitly waits for the respective SUT client to connect for the required GUI engine. In case the wait fails, a `ComponentNotFoundException`[(850)] is thrown with a message that explains whether the component was not found, the respective SUT client is not connected or whether there is no matching GUI engine available for that client. The Expected message[(620)] attribute of a Catch[(619)] node can be used to differentiate between those causes.
  In absence of a suitable client there was an inconsistency in older QF-Test versions, causing either a `ComponentNotFoundException` or a `ClientNotConnectedException`[(855)] to be thrown, depending on whether such a client was previously connected or not. In case your test still expects a `ClientNotConnectedException` you either need to change the Catch node, add an additional Catch or prepend the Wait for component to appear node with a Wait for client to connect[(670)] node.

- The new option How to handle disabled components[(458)] defines what to do when a disabled Component node is referenced during test execution. In previous versions of QF-Test the disabled state was ignored which sometimes led to confusion. The default now is to log an error. Consequently, disabled Component nodes are no longer re-used when recording.

- There is a new option called Default timeout for checks (ms)[(480)] that - as its name implies - defines a default timeout for checks that have no Timeout[(711)] attribute set. It increases general stability of check execution. If your tests include a lot of Check nodes without Timeout that are expected to fail, test execution time may increase. In this case you may want to set this option back to 0, the old default or, preferably, update the respective nodes and set their Timeout attribute to 0.

Web
- The AJAX toolkit RAP does not use the 'ID' attribute of DOM nodes to define IDs, so the option Eliminate all numerals from 'ID' attributes[(491)] had no effect for RAP. This has been fixed, so that with the option activated, numerals are now removed from IDs generated for RAP elements. In case this causes problems with existing tests, simply deactivate the option Eliminate all numerals from 'ID' attributes[(491)].

Software that is no longer supported:

Note Please see section 1.1[(3)] for a detailed list of system requirements and supported technology versions.

QFS is taking pains to support even very old software. QF-Test still runs, for example, under Windows XP or with an SUT based on Java 1.5 or 1.6 but for systems that are no longer supported by their own vendors we cannot provide full support.

Libraries that are required for the Firefox and Eclipse/SWT versions listed below are no longer part of the default installation of QF-Test. The majority is still being maintained, albeit with limited testing. If you need support for such a system, please get in touch with QFS via `support@qfs.de`.

Official support has been dropped for Java 1.5, Internet Explorer 7 and older, Firefox 25 and older (except Firefox 24 ESR) as well as Eclipse/SWT 3.6 and older.

Version 4.0.0 - July 31, 2014

New features:

Web
- Support was added for Firefox versions 31 and 32.

- A more user-friendly way of updating a QF-Test license is now available via the menu Help→Update license... .

- When opening a Jython or Groovy console for an SUT client, the command history from the most recently created Jython or Groovy console is copied. This makes it easier to try out the same scripts when repeatedly restarting the SUT.

- When importing QF-Test results into TestLink it is now possible to use the QF-Test ID for associating test cases and to omit the platform.

- After starting QF-Test the record button now has a help icon that leads to the quickstart wizard for launching an SUT. This feature is intended to assist first-time users that often wonder why they cannot start recording right away.

Web
- The procedure `qfs.web.ajax.installToolkitResolver` now allows specifying default versions for automatically detected AJAX toolkit resolvers. This prevents inadvertent updates to versions with potential incompatibilities in newer QF-Test versions.

Web
- The default severity level for JavaScript errors in web pages has been changed to 'Warning'. Current browsers simply ignore JavaScript problems by default so that treating those as an error caused too much irritation. The setting can be changed via the option How to handle errors in a web application[493].

Bugs fixed:

SWT
- On Linux systems like RHEL 6.5 with an old version of glibc, instrumented 64bit versions of Eclipse failed to start. This dependency on a newer version of glibc has been removed.

SWT
- When the QF-Test agent was enabled, an Eclipse / SWT application with embedded Swing components connected as two SUT clients instead of one client with two GUI engines.

JavaFX
- Column header cells in JavaFX tables are now addressed correctly.

JavaFX
- CheckBox elements inside JavaFX table cells or tree nodes are now recorded with the new @:CheckBox item syntax.

Web
- The check for the editable state of TextField, PasswordField and TextAreas nodes was missing for web clients.

Web
- A selection in a ComboBox that initiated a navigation to another page could cause Internet Explorer to crash.

Web
- Check recording in the browser did not work in an area covered by an invisible IFRAME.

Web
- Event synchronization for the AJAX toolkit Ext JS now handles modal dialogs correctly.

Web
- Replay of text input for INPUT elements of type "number" has been fixed.

Web
- When defining proxy settings for Internet Explorer via the `qfs.web.browser.settings.doStartupSettings` procedure, `proxy.pac` files were not handled correctly.

Swing
- In newer Java and Browser versions the startup behavior of applets has changed, sometimes causing two JRE versions to start up simultaneously. Depending on timing, this could cause the real applet to connect as $(client):2 instead of $(client). QF-Test now works around this problem by connecting to an applet only after it has become visible.

Swing
- The "All lines" check for Swing JTextArea components was not recorded correctly when clicking below the first line of text.

Swing
- There are now two different text checks for JEditorPane components that display HTML, "Text" and "HTML source". In many cases the check against the plain, displayed text is more useful than the former default check for the HTML source code. Also, the text check for JEditorPane components that display RTF has been fixed.

- When canceling an edit in the detail view or adding of a new node, QF-Test now checks whether modifications were made and asks for confirmation before discarding those. This confirmation dialog can be suppressed via the option Ask before discarding detail modifications[(428)].

- When toggling the disabled state of the selected node, modifications made but not confirmed in the detail view were silently discarded.

Version 4.0.0-rc1 - July 4, 2014

New features:

- QF-Test now supports multi-level sub-items for addressing components within an item. A typical example is a TextField within a table cell that can now be addressed as table@Header&row@:TextField. The syntax is extremely powerful with the potential to mix XPath-like syntax with QF-Test's component representation (see section 5.9.2[(86)]).

- The former 'Id' attribute of Window and Component nodes is now called QF-Test ID in order to disambiguate it from the various kinds of IDs used by the supported GUI engines.

- Generic classes are now also implemented for standard HTML nodes even without a dedicated AJAX resolver.

- Support for the AJAX toolkit RichFaces was updated to RichFaces version 5.

- Support for Eclipse/SWT has been updated for Eclipse version 4.4 "Luna".

- The Nebula Grid component is now supported by default and automatically recognized.

- It is now possible to define a directory for saving temporary run logs in interactive mode via the option Directory for run logs[(503)]. During QF-Test startup this directory can be specified on the command line via `-runlogdir <directory>`[(879)].

- The new option Directory map for test suites[(512)] can be used to map directories for test suites. This is very useful when working with a run log from a machine with a different file layout (e.g. Windows vs. Unix) because with the correct directory mapping QF-Test can locate a test suite without manual help.

- There are two new special variable groups to conveniently escape special characters. `${quoteitem:...}` deals with characters like '@' or '&' in a textual sub-item index and `${quoteregexp:...}` handles '.', '(', '[' etc. in a regular expression.

- It is now possible to specify a directory as a bookmark (see section 39.1.5[(430)]). When such a bookmark is selected, the file selection dialog is opened directly for this directory.

Bugs fixed:

Web
- Multiple instances of the Chrome browser can now run simultaneously on one machine.

Web
- Performance, stability and supported feature set for browsers embedded via JavaFX WebView and JxBrowser were improved.

Web
- The AJAX toolkit RAP does not use the 'ID' attribute of DOM nodes to define IDs, so the option Eliminate all numerals from 'ID' attributes[491] had no effect for RAP. This has been fixed, so that with the option activated, numerals are now removed from IDs generated for RAP elements. In case this causes problems with existing tests, simply deactivate the option Eliminate all numerals from 'ID' attributes[491].

- With support for JavaFX and its Swing InterOp embedding capabilities as well as support for embedded browsers, SUT clients often have more than one GUI engine. This caused problems with SUT script nodes in the standard library `qfs.qft` which were relying on the default GUI engine. All procedures with SUT scripts now have an optional parameter named `guiengine` which is predefined correctly, so that for example a Procedure in the Package `qfs.web` automatically works correctly in an embedded browser. In the exceptional case of an SUT client with more than one GUI engine of the same kind you can define the `guiengine` parameter to address a particular engine.

- After introducing generic classes, replay was broken in case an anonymous inner class was manually assigned for a component.

- During dependency resolution, Parameter default values of Dependency nodes were not correctly bound.

- Performance of test execution in interactive mode suffered from repaint issues caused by the new syntax highlighting in the tree view.

Version 4.0.0-beta3 - May 13, 2014

New features:

Web
- Support was added for Firefox version 30.

JavaFX
- Procedure recording is now fully supported for JavaFX.

Web
- For web applications based on the AJAX framework ZK QF-Test now resolves the original developer-assigned IDs and uses them as names for components.

Web

- Support for the Chrome browser is continuously improving. Up- and download are now fully supported as well as login dialogs, closing of browser windows, key and text input including modifiers and cursor position. Also, there now is a custom profile folder to separate testing from normal use with support for cookie and cache cleanup, locale and proxy settings.

Web

- Embedded browsers in JavaFX WebView or via JxBrowser are also making headway and multiple embedded browsers in one window are now handled correctly.

- Simple but very useful: The dialog shown when an error or exception occurs during a test now has an additional button for jumping directly to the respective node in the run log.

JavaFX

- A check was added for the checked state of a CheckBox in JavaFX table cells, tree nodes and list entries.

- The new procedure `qfs.web.general.dragFromTo` supports HTML5 style drag and drop operations without blocking the application, alternatively at JavaScript level or via hard events.

Bugs fixed:

SWT

- The introduction of generic classes for SWT caused a serious performance issue that has been fixed.

SWT

- For SWT QF-Test refused to replay hard events on negative coordinates which, however, are necessary to reach a window on a second monitor placed to the left of the main screen on Windows.

Web

- The print dialog is now shown correctly in Firefox.

- In very special cases running a test in a QF-Test daemon could fail due to a missing engine license even though the license was available.

Version 4.0.0-beta2 - April 11, 2014

New features:

- Generic classes are now also implemented for Swing and SWT.

- QF-Test records generic class, custom class and system class as extra features named qfs:genericclass, qfs:class and qfs:systemclass respectively. This is useful information and can be used to switch individual components to a more specific or less specific class. In case the generic class also has a more specific sub-type QF-Test also records the extra feature qfs:type, e.g. "PanelTitle" for a Label that is the title of a TitledPanel.

Web
- Supported standard functionality for Chrome now includes support for downloads, handling prompt and confirm dialogs as well as getting and setting the cursor position in text fields which is required for recording mouse events relative to a character position and for Text input nodes which insert text into the middle of existing text.

- Support for embedded browsers now includes JavaFX WebView and JxBrowser 4. Though this is still work in progress, basic capture and replay for mouse events already works.

Web
- When started from QF-Test Chrome now also automatically accepts SSL certificates for HTTPS connections.

- It is now possible to choose between different layouts of the test suites in the selection dialog for Procedures.

Bugs fixed:

JavaFX
- Thanks to improved focus management replay for JavaFX on Linux should be more reliable now, especially selection of menu items.

JavaFX
- Replaying hard mouse click for the middle or right button did not work correctly with JavaFX.

- An Excel 2010 sheet with a cross-sheet reference to a pure number is now handled correctly by the Excel data file node.

Version 4.0.0-beta1 - March 26, 2014

New features:

JavaFX
- JavaFX is the successor of Swing as the standard UI Toolkit in Java 8. QF-Test version 4.0 introduces the new product line QF-Test/fx that fully supports testing JavaFX applications. This GUI engine is nearly complete. Missing at this point is recording of procedures and support for handling the embedded WebView as a full-featured QF-Test Web engine is incomplete.

- Support for the Chrome browser on Windows is well advanced and mostly stable. As before with Internet Explorer and Firefox the goal is to hide the browser differences and enable a single test to run unchanged on all three browsers. Not implemented at this point are various browser dialogs as well as file up- and download. Another current restriction is that only one instance of Chrome can be run by QF-Test in parallel.

- The concept of generic classes, initially available for the web engine only, has been extended to support JavaFX and soon all GUI engines. It is the basis for interoperable tests that work for different GUI engines, provides a higher level of abstraction and more flexible and robust component recognition. Recording of generic classes is configured via the option <u>Record generic class names for components</u>[449].
  See the video Dealing with the explosion of complexity in web test automation for a vivid demonstration of the explosion of complexity in AJAX user interfaces. This applies just as well to JavaFX which shares the automation problems of a UI built from a large number of small GUI elements and for which QF-Test's approach with generic classes fits perfectly.

- Tolerant class matching when recording: Existing nodes in QF-Test's component hierarchy will be reused as long as they match one of the classes of the recorded component. This means that when switching to/from generic class recording, no components will be forcibly re-recorded. This has no impact on the 'Update components' feature which always updates the respective components according to the current recording option settings.

- Connection with the SUT is now implemented via the Java agent mechanism by default. It is faster, more stable - especially for Java WebStart and Applets - and much more flexible and powerful. In case of problems the Java agent mechanism can be disabled via the option <u>Connect via QF-Test agent</u>[514].

- During SUT startup Jython and Groovy are now loaded on demand, leading to faster and more reliable startup. Jython in particular was one of the main reasons for problems with Java WebStart. This also reduces the overhead and memory footprint if either of the languages is not used at all.

Web
- Support was added for the AJAX framework jQuery UI.

SWT
- Eclipse/SWT support was added for the Eclipse 4.4 "Luna".

SWT
- The Nebula NatTable component is now supported by default and automatically recognized.

- Colored syntax highlighting of tree nodes was added which significantly improves readability of test suites.

- The shared terminal now provides coloring of different message types. The new group of <u>Terminal options</u>[463] encloses a bunch of new settings to adjust text display in the shared terminal. Font size and type can be defined as well as a number of regular expressions to highlight and/or filter arbitrary text in the output.

- It is now possible to set breakpoints on <u>Component</u>[823] nodes so as to enter the debugger as soon as the node gets referenced.

- For a failed Check items[719] node both run log and report now contain a context diff of the expected and actual values so that all deviations can quickly be identified.

- Relative Test calls[531] with a leading '.' in the Test name[533] attribute are now supported.

- In places where QF-Test supports regular expressions, e.g. Text[710], Feature[825] or Items[721], normal text can now be turned into a properly quoted regular expression by right-clicking and selecting 'Escape text for regular expressions' from the context menu.

- Handling of items with multi-line text, e.g. tree nodes, table headers or cells, list items etc. has been improved.

- Search and replace operations for variables can now be limited further to cover only variable names or variable values.

- The manual chapter on integration with HP ALM - QualityCenter was rewritten from scratch: section 26.1[312].

Changes that can affect test execution:

- The Wait for component to appear[771] node implicitly waits for the respective SUT client to connect for the required GUI engine. In case the wait fails, a `ComponentNotFoundException`[850] is thrown with a message that explains whether the component was not found, the respective SUT client is not connected or whether there is no matching GUI engine available for that client. The Expected message[620] attribute of a Catch[619] node can be used to differentiate between those causes.
  In absence of a suitable client there was an inconsistency in older QF-Test versions, causing either a `ComponentNotFoundException` or a `ClientNotConnectedException`[855] to be thrown, depending on whether such a client was previously connected or not. In case your test still expects a `ClientNotConnectedException` you either need to change the Catch node, add an additional Catch or prepend the Wait for component to appear node with a Wait for client to connect[670] node.

- The new option How to handle disabled components[458] defines what to do when a disabled Component node is referenced during test execution. In previous versions of QF-Test the disabled state was ignored which sometimes led to confusion. The default now is to log an error. Consequently, disabled Component nodes are no longer re-used when recording.

- There is a new option called Default timeout for checks (ms)[480] that - as its name implies - defines a default timeout for checks that have no Timeout[711] attribute set.

It increases general stability of check execution. If your tests include a lot of CHECK nodes without TIMEOUT that are expected to fail, test execution time may increase. In this case you may want to set this option back to 0, the old default or, preferably, update the respective nodes and set their TIMEOUT attribute to 0.

New packages and procedures in the standard library `qfs.qft`

JavaFX
- The new `qfs.fx` package holds a full set of procedures for interacting with JavaFX components.

- The new `qfs.utils.ssh` package provides procedures for client-side ssh connections.

- There's a new check utility procedure called `qfs.check.checkValueInRange`.

Swing
- Various new procedures for interacting with Swing components:
  ```
  qfs.swing.checkbox.deselectWithCoordinates
  qfs.swing.checkbox.selectWithCoordinates
  qfs.swing.checkbox.setWithCoordinates
  qfs.swing.general.clickAtComponentWithDetails
  qfs.swing.general.doClickWithDetails
  qfs.swing.general.waitForComponent
  qfs.swing.general.waitForComponentForAbsence
  qfs.swing.list.clickItemWithDetails
  qfs.swing.menu.selectItemWithCoordinates
  qfs.swing.menu.selectSubItemWithCoordinates
  qfs.swing.menu.setCheckItemWithCoordinates
  qfs.swing.menu.setSubCheckItemWithCoordinates
  qfs.swing.menu.selectMenuItemInHierarchyWithCoordinates
  qfs.swing.menu.selectCheckMenuItemInHierarchyWithCoordinates
  qfs.swing.popupmenu.clickItemWithDetails
  qfs.swing.popupmenu.selectSubItemWithCoordinates
  qfs.swing.popupmenu.setCheckItemWithCoordinates
  qfs.swing.popupmenu.setSubCheckItemWithCoordinates
  qfs.swing.radiobutton.selectWithCoordinates
  qfs.swing.table.checkbox.deselectWithCoordinates
  qfs.swing.table.checkbox.selectWithCoordinates
  qfs.swing.table.checkbox.setWithCoordinates
  qfs.swing.table.clickCellWithDetails
  qfs.swing.tree.clickNodeWithDetails
  qfs.swing.tree.collapseWithCoordinates
  qfs.swing.tree.collapseNodeWithCoordinates
  qfs.swing.tree.expandWithCoordinates
  ```

```
qfs.swing.tree.expandNodeWithCoordinates
qfs.swing.tree.expandNodesWithCoordinates
```

SWT   • Various new procedures for interacting with SWT components:
```
qfs.swt.checkbox.deselectWithCoordinates
qfs.swt.checkbox.selectWithCoordinates
qfs.swt.checkbox.setWithCoordinates
qfs.swt.general.clickAtComponentWithDetails
qfs.swt.general.doClickWithDetails
qfs.swt.general.waitForComponent
qfs.swt.general.waitForComponentForAbsence
qfs.swt.list.clickItemWithDetails
qfs.swt.menu.selectItemWithCoordinates
qfs.swt.menu.selectSubItemWithCoordinates
qfs.swt.menu.setCheckItemWithCoordinates
qfs.swt.menu.setSubCheckItemWithCoordinates
qfs.swt.menu.selectMenuItemInHierarchyWithCoordinates
qfs.swt.menu.selectCheckMenuItemInHierarchyWithCoordinates
qfs.swt.popupmenu.clickItemWithDetails
qfs.swt.popupmenu.selectSubItemWithCoordinates
qfs.swt.popupmenu.setCheckItemWithCoordinates
qfs.swt.popupmenu.setSubCheckItemWithCoordinates
qfs.swt.radiobutton.selectWithCoordinates
qfs.swt.table.checkbox.deselectWithCoordinates
qfs.swt.table.checkbox.selectWithCoordinates
qfs.swt.table.checkbox.setWithCoordinates
qfs.swt.table.clickCellWithDetails
qfs.swt.tree.clickNodeWithDetails
qfs.swt.tree.collapseWithCoordinates
qfs.swt.tree.collapseNodeWithCoordinates
qfs.swt.tree.expandWithCoordinates
qfs.swt.tree.expandNodeWithCoordinates
qfs.swt.tree.expandNodesWithCoordinates
```

Web   • Various new procedures for interacting with web components:
```
qfs.web.checkbox.deselectWithCoordinates
qfs.web.checkbox.selectWithCoordinates
qfs.web.checkbox.setWithCoordinates
qfs.web.general.clickAtComponentWithDetails
qfs.web.general.doClickWithDetails
qfs.web.general.waitForComponent
qfs.web.general.waitForComponentForAbsence
qfs.web.list.clickItemWithDetails
```

```
qfs.web.radiobutton.selectWithCoordinates
qfs.web.table.clickCellWithDetails
```

Software that is no longer supported:

QFS is taking pains to support even very old software. QF-Test still runs, for example, under Windows XP or with an SUT based on Java 1.5 or 1.6 but for systems that are no longer supported by their own vendors we cannot provide full support.

Libraries that are required for the Firefox and Eclipse/SWT versions listed below are no longer part of the default installation of QF-Test. The majority is still being maintained, albeit with limited testing. If you need support for such a system, please get in touch with QFS via `support@qfs.de`.

Official support has been dropped for Java 1.5, Internet Explorer 7 and older, Firefox 25 and older (except Firefox 24 ESR) as well as Eclipse/SWT 3.7 and older.

B.10 QF-Test version 3

B.10.1 Version 3.5.7 - July 22, 2014

New features:

Web
- Support was added for Firefox versions 31 and 32.

SWT
- Eclipse/SWT support was added for Eclipse version 4.4 "Luna".

- The command line arguments `-exitcode-ignore-warning`, `-exitcode-ignore-error` and `-exitcode-ignore-exception` are now also available in calldaemon mode. These options are helpful for integration with continuous integration tools like Hudson/Jenkins or TeamCity as the exit code of a QF-Test batch run can be forced to 0 even in case of warnings, errors or exceptions.

Bugs fixed:

SWT
- On Linux systems like RHEL 6.5 with an old version of glibc, instrumented 64bit versions of Eclipse failed to start. This dependency on a newer version of glibc has been removed.

SWT
- For SWT QF-Test refused to replay hard events on negative coordinates which, however, are necessary to reach a window on a second monitor placed to the left of the main screen on Windows.

Web   • The check for the editable state of INPUT:TEXT, INPUT:PASSWORD and TEXTAREA nodes was missing for web clients.

Swing   • The "All lines" check for Swing JTextArea components was not recorded correctly when clicking below the first line of text.

• When canceling an edit in the detail view or adding a new node, QF-Test now checks whether modifications were made and asks for confirmation before discarding those.

• When toggling the disabled state of the selected node, modifications made but not confirmed in the detail view were silently discarded.

• In very special cases running a test in a QF-Test daemon could fail due to a missing engine license even though the license was available.

B.10.2   Version 3.5.6 - April 29, 2014

New features:

Web   • Support was added for Firefox versions 29 and 30.

Web   • The new Procedure `qfs.web.browser.mozilla.cleanupProfileDir` in the standard library `qfs.qft` attempts to remove redundant and broken files (most notably `places.sqlite`) in the Firefox profile directory used for testing (`<user.home>/.qftest/mozProfile` by default). The procedure is called automatically during the standard startup sequence via the call to `qfs.web.browser.settings.doStartupSettings`.

• For TestLink integration the ImportResults script now lets you specify the build name during import, overriding QF-Test's runid at test execution time is no longer necessary.

Bugs fixed:

• The warning dialog for duplicate IDs in a test suite is no longer shown in batch mode.

• An Excel 2010 sheet with a cross-sheet reference to a pure number is now handled correctly by the Excel data file node.

Web   • In some web pages content inside an IFRAME could not automatically be scrolled to the correct location for replaying a hard mouse event or performing an image check.

Web • The print dialog is now shown correctly in Firefox.

Swing • The fix for Java 8 focus handling in QF-Test version 3.5.5 could cause a very subtle issue in focus behavior for popup windows.

B.10.3 Version 3.5.5 - February 25, 2014

New features:

Web • Support for Firefox version 27 was updated and new support added for Firefox version 28.

SWT • Eclipse/SWT support was updated for service release 2 of Eclipse version 4.3.

• The driver test suites and documentation for the integration with imbus TestBench were updated.

Web • QF-Test is now less restrictive when comparing document URLs or href attributes, so a simple change of a hostname, IP address or port no longer requires working with regular expressions in the Feature attribute.

Bugs fixed:

• Focus handling for replay with Java 8 has been improved and QF-Test now works around a bug in Java 8 when setting a window topmost on Windows systems.

Web • Proxy settings for Internet Explorer are now application-local, so even after a crash the system proxy settings remain unchanged.

Web • The user agent for Internet Explorer 11 is now reported correctly by QF-Test's embedded browser.

Web • Firefox confirmation dialogs for closing or navigating away from a web page are now handled correctly and can be interacted with.

SWT • Input of Unicode characters from Text input[689] nodes with Replay single events[691] set was not replayed correctly on some Linux systems.

• Closing the QF-Test main workbench with a run log still open, then jumping from that run log to a test suite caused the saved session of opened test suites to get lost.

B.10.4 Version 3.5.4 - December 11, 2013

New features:

Web
- Support was added for Firefox versions 26 and 27.

Web
- Support for the AJAX toolkit ZK was updated for ZK version 7.

Web
- Synchronization with AJAX HTTP requests was improved for the toolkits ExtJS, ICEfaces, PrimeFaces and ZK.

Web
- On Windows the one-time installation of the QF-Test JavaScript bridge for Firefox no longer requires starting QF-Test once with administrator privileges.

Bugs fixed:

- WebStart applications are still a moving target. Various fixes for special cases were implemented to make startup of WebStart applications more reliable.

- Mouse events with variables for the X or Y coordinate that expand to empty values were not replayed to the middle of the target but to the relative coordinate 0.

- In case of deep component hierarchies QF-Test no longer logs excessive component information which, in extreme cases, could cause the SUT to run out of memory.

- A memory and thread leak in the `DaemonLocator` class has been plugged.

- The JNA library included in Jython no longer triggers stackguard warnings.

Web
- An exception caused by a special kind of HTML table with no columns in the header has been fixed.

B.10.5 Version 3.5.3 - September 17, 2013

New features:

Web
- Support was added for the AJAX toolkit ICEfaces. See section 49.2[(988)] for further information and a list of the supported AJAX toolkits and their versions.

Web
- QF-Test can now automatically detect whether a web page is based on one of the supported AJAX toolkits and installs the respective resolver.

Web
- Support was added for Firefox versions 24 and 25.

- The QF-Test demo video on the welcome dialog now links to the updated video on the QFS home page which is fully redesigned, including spoken explanations.

- It is now possible to copy and paste table cells back and forth between QF-Test tables and Excel or OpenOffice spreadsheets.

- When changing the name of a parameter in callable nodes like a Procedure QF-Test now offers to automatically update the variable name in all calling nodes.

- The standard library `qfs.qft` contains several new procedures for working with standard Swing dialogs based on the `JOptionPane` class in the package `qfs.swing.optionpane`.

Bugs fixed:

- Stability of startup of WebStart based clients and compatibility with older WebStart versions was further improved.

- Canceling a long search did not work reliably and could even lead to a deadlock in some cases.

- Replace operations for the regular expression ".*" could lead to double replacements in some cases.

- The associated label for components was sometimes not determined correctly if the label was placed to the right of the component.

B.10.6   Version 3.5.2 - July 2, 2013

Important changes:

- Handling of the security dialogs for Java WebStart and applets had to be changed again.
  Security warnings are now closed implicitly before connecting to QF-Test and it is possible to permanently accept a certificate during this step. The details for this operation can be configured via the procedure `qfs.swing.startup.setSecurityDialogSpec`.
  The default delays for WebStart and applets also changed as documented in the procedure `qfs.swing.startup.setDelays`.

New features:

SWT

- Eclipse/SWT support was updated for the final Eclipse version 4.3 "Kepler".

Web
- Support was added for Firefox version 23 and updated for version 22.

- Testlink support was updated for Testlink version 1.9.7.

- Wait for component to appear nodes now automatically handle waiting for the respective GUI engine in addition to implicitly waiting for the SUT to connect.

- Nodes for Procedures[586] and Packages[594] marked as "@deprecated" are now crossed out in the tree view.

Web
- The new procedure `qfs.web.general.integrateFirebug` in the standard library `qfs.qft` loads Firebug Lite into the running browser which can be very helpful when writing resolvers.

- The keyboard shortcut Shift-Control-I can now be used to quickly pack one or more steps into an If[605] sequence.

Bugs fixed:

- The Windows executables `qftest.exe` and `qftestc.exe` failed to read the configured Java version, memory and language from the Windows registry.

- Saving test suites and run logs and other IO operations are now significantly faster on slow network shares.

- When changing the currently executed node via 'Continue execution from here' in the test debugger, QF-Test is now smarter in keeping as much of the current state of execution as possible, including the previously broken case when continuing from a Procedure[586] node.

Web
- On Linux systems Firefox would crash when terminated while a dialog was still showing.

Web
- The recently added support for ZK and PrimeFaces was fine-tuned and fixes subtle problems with specific Trees, TreeTables and other components.

Swing
- Associated labels of Swing components, recorded as "qfs:label" extra feature, are now used for generating the QF-Test ID[824] attribute. For web and SWT components this was already the case.

- In the QF-Test daemon `DaemonTestRunListener` instances were not unregistered correctly which could slow down daemon execution after a call like
`qftest -batch -calldaemon -verbose ...`

- The SUT can now implement an RMIClassLoaderSpi without interference from QF-Test's RMI communication.

B.10.7   Version 3.5.1 - May 15, 2013

Important changes:

- The Jython version provided with QF-Test was rolled back to Jython 2.5.1 because Jython 2.5.2 caused strange but reproducible problems with some SUTs. Jython is mainly used as glue code, so stability and non-interference with the SUT are far more important than compatibility with newer Python language features and modules, so the Jython version for QF-Test will remain frozen at 2.5.1 for the foreseeable future.

New features:

Web
- New support was added for the AJAX toolkits PrimeFaces and ZK and support for Eclipse RAP was updated to RAP version 2.0. See section 49.2[(988)] for further information and a list of the supported AJAX toolkits and their versions.

SWT
- Eclipse/SWT support was added for Eclipse version 4.3 "Kepler".

Web
- Support was added for Firefox versions 21 and 22.

- The new procedure `qfs.swing.filechooser.selectFiles` can be used to select several files in a multi-select enabled Swing `JFileChooser`.

- The new Browser HTTP request[(808)] node can be used to replay HTTP GET or POST request directly in the running browser. In contrast to the existing Server HTTP request[(802)] node, which interacts directly with the server, the Browser HTTP request can be combined with event and check nodes.

- HTML reports now contain pie charts to visualize the number of successful, failed and skipped tests. A click on the chart brings up a more detailed graph. The directory structure of reports has been cleaned up by moving all images and screenshots to sub-directories.

Web
- When started from QF-Test Firefox now automatically accepts SSL certificates for HTTPS connections, limited to the current session.

- Testlink support was updated for Testlink version 1.9.4.

Bugs fixed:

- The initial connection sequence for applets and WebStart based applications had to be updated after security restraints were again tightened in recent JRE updates from Oracle. The stability of applet and WebStart startup should now be less dependent on timing and the procedure `qfs.swing.dialogs.handleSecurityWarning` is now able to handle all current security warnings for applets and WebStart.

- Exceptions caused by CSV or Excel files with inconsistent data are now handled cleanly.

- Updating components with <u>Item</u>⁽⁸²⁹⁾ nodes could cause an exception.

B.10.8   Main new features in version 3.5

Following is an overview over the main new features that have been implemented for QF-Test since the last production version 3.4.

Description	Further info
Grouping and managing test suites in projects	chapter 9⁽¹⁵⁶⁾
Support for AJAX toolkits ICEfaces, PrimeFaces, Vaadin and ZK	section 49.2⁽⁹⁸⁸⁾
Better abstraction with generic classes for web pages	section 5.4.1⁽⁵⁶⁾
Full sub-item support for several AJAX toolkits	section 49.2⁽⁹⁸⁸⁾
Secure daemon communication via SSL	section 23.2⁽²⁸⁶⁾, chapter 53⁽¹¹³⁹⁾
Improved component recognition without names thanks to associated label	Attribute Extra feature⁽⁸²⁵⁾
Integration with test management tool QMetry	section 26.3⁽³²⁷⁾
Several extensions to the standard library `qfs.qft`	Standard library
Groovy updated to version 2.1.0	

Table B.7: New features in QF-Test version 3.5

B.10.9   Version 3.5.0 - March 20, 2013

New features:

Web

- Support was added for Firefox version 20.

SWT

- Eclipse/SWT support was updated for Eclipse versions 4.2.2 and 3.8.2.

- Groovy was updated to version 2.1.0.

- Explicit references between test suites belonging to the same project are now resolved automatically provided that those test suites have been saved with QF-Test version 3.5.0 or higher.

- Multi-line values can now be edited directly in QF-Test's table cells.

- Extra mouse buttons can now be used to move backwards/forwards between recently selected nodes.

- Double-clicking on nodes that reference other nodes now navigates to the target, e.g. from Procedure call[589] to Procedure[586].

- test suite and run log files can now be dragged onto QF-Test's main window to open them.

- The new procedure `qfs.swing.dialogs.handleSecurityWarning` in the standard library `qfs.qft` closes the security dialogs displayed by current JREs during startup of applets or WebStart based applications.

- The procedure `qfs.web.browser.general.startAsNewProcess` was updated to include security dialog handling.

- Another new procedure named `qfs.swing.startup.startWebstartSUT` can be used to run WebStart and connect to the SUT, also handling the security dialog automatically.

- The new procedure `qfs.utils.files.readExcelFile` can be used to read data from an excel sheet into Jython or QF-Test variables.

Web
- The Server HTTP request[802] node now offers an optional timeout and can return the HTTP status code. Parameters are now encoded during replay instead of recording, making it easier to specify values manually. Older test suites are converted automatically during loading to reflect this change.

SWT
- The default names recorded for GEF items are now more meaningful and result in improved recognition.

- QF-Test now integrates with the test management tool Klaros. See section 26.4[330] for further information.

- The error list for a run log now includes a timestamp column and shows information about expected errors.

- When pausing a batch test via Alt-F12, input to the SUT is no longer blocked.

- If the new command line argument `-dontkillprocesses`[871] is specified in batch mode, QF-Test will not explicitly terminate processes started as part of the tests.

Bugs fixed:

- A memory leak in case of large projects with many test suites and complex includes has been plugged.

- A workaround has been implemented for a bug in recent JRE security updates that could cause batch tests to simply stop in the middle of execution.

- Variables could accidently be removed from QF-Test component IDs when fixing references automatically.

- In special cases externalized run logs could be missing from run logs saved as .qzp.

- When opening an externalized log on its own, split logs contained therein could not be accessed correctly.

- A race condition during automatic project refresh could mess up the project tree view.

- The native file chooser for Windows is now implemented based on SWT instead of XFileDialog. This fixes issues when opening directories and when changing the file filter during "Save as..." operations.

- In batch execution of multiple test suites, global variables and resources are no longer cleared before the execution of each test suite unless the new command line argument -clearglobals[(870)] is specified.

Web - Firefox on Linux no longer crashes on shutdown.

Web - For some MIME types downloading files did not work correctly with Firefox.

Web - Checks for the enabled state could fail in Internet Explorer if the state was set via JavaScript.

Web - Waiting for the absence of an element could fail if the element just became invisible instead of being removed completely.

- Variables in Data binder nodes are now expanded immediately during execution of the containing Data driver[(562)] node so that local variables bound in the Data driver are evaluated correctly.

- A Return[(592)] in a Finally[(623)] node could hide an exception thrown in the containing Try[(616)] node.

- Using Java 7 on Mac systems could cause an exception during startup of QF-Test.

- In case a process simply refuses to be killed, the Stop client[(675)] node did not remove the process from the Clients menu, thus preventing starting a new process under the same name.

- Image checks based on algorithms were not always shown correctly in the report.

- In case dependency resolution fails due to an exception during setup even after unwinding the complete dependency stack for error escalation (see section 8.6.7$^{(148)}$), QF-Test now cleans up after the final attempt so that the next test case starts with a clean slate.

- Changes to the location of the search dialog and other dialogs were often ignored instead of automatically saved to the user preferences.

B.10.10   Version 3.5M3 - January 8, 2013

New features:

Web
- Support was updated for Firefox versions 18 and 19.

- The project concept now includes support for subprojects. See chapter 9$^{(156)}$ for further information.

Web
- JavaScript error filters defined in the option Errors that should be ignored$^{(494)}$ are now applied to the complete message including the URL and line number of the source of the error.

SWT
- Several checks were added for Eclipse JFace components like `FormText`, `Link` or `Hyperlink`.

Bugs fixed:

- The responsiveness of QF-Test severely suffered when opening many test suites.

- Refresh of large projects on slow filesystems like Windows network shares is now significantly faster.

- If several projects were opened, automatic refresh of the project hierarchy could mess up the project tree.

SWT
- Geometry of events relative to an SWT Shell now includes the window title and borders.

SWT
- Highlighting widgets inside `CTabFolders` in Eclipse 4.2 now uses the correct offset.

- Errors happening outside a test case were not listed correctly in the report if the containing test set contained several test cases.

Web
- Downloading files with Firefox could cause a JavaScript error if the file `mimeTypes.rdf` was missing from QF-Test's Mozilla profile. This file is now created on the fly if necessary.

Web • Closing the browser window of Internet Explorer 10 could cause the browser to deadlock instead of terminate.

Web • QF-Test's embedding of Internet Explorer 10 caused the browser to run in IE9 mode. Now it starts up in IE10 mode by default with the usual settings for compatibility mode enabled.

Web • Replaying [Tab] keystrokes on Firefox did not always trigger the correct focus traversal.

• Conflicts caused by an SUT using a Groovy 2 version other than the version provided with QF-Test are now resolved.

B.10.11 Version 3.5M2 - November 16, 2012

New features:

SWT • Eclipse/SWT support was updated for Eclipse versions 4.2.1 and 3.8.1.

• Daemon communication is now secured via SSL. See chapter 53[(1139)] about how to increase security by creating an individual keystore or how to disable this feature in case you need daemon interaction with older QF-Test versions.

• QF-Test generated JUnit reports now contain `<skipped>` tags, enabling build tools like Hudson/Jenkins to correctly report skipped tests.

Bugs fixed:

• License usage for idle QF-Test daemons was not determined correctly.

Swing • Replaying certain events in batch mode could switch QF-Test into recording mode, causing weird problems and significant loss of performance.

Swing • Timing for startup of Java WebStart based SUTs has been further improved to significantly increase the stability of this setup.

Web • Some focus transitions were not triggered correctly when replaying events in a browser.

Web • The filename shown in a download dialog was not always correct.

Web • Downloads in a Vaadin based SUT could cause an exception.

Web • Internet Explorer 10 now reports the correct version information.

SWT • Recording of drag and drop operations on CTabFolders in Eclipse 4.2 was broken.

- In some cases the height of SWT List items was not calculated correctly, leading to the wrong item being addressed in case of large lists with many items.

- The new Extra feature attribute `qfs:label` was sometimes not determined correctly in the presence of Swing JSlitPane or SWT SashForm components.

- Handling of an SUT with combined SWT/AWT GUI was broken. Due to an error introduced in version 3.4.11 QF-Test was treating this as two SUT clients instead of one client with two GUI engines.

B.10.12   Version 3.5M1 - October 29, 2012

New features:

- With the introduction of projects (see chapter 9[(156)]), large sets of test suites become much more manageable and updating reverse includes is now mostly a thing of the past.

- The increasingly popular AJAX toolkit Vaadin is now fully supported.

- The integration of QF-Test with various AJAX toolkits has been brought to a new level. Generic classes provide a unified view that abstracts from the technical details, focusing on typical UI elements like Tree or Table that users are familiar with. This even includes the concept of sub-items so that addressing table cells or tree nodes is now just as easy as with Swing or SWT. This level of support is currently available for Ext JS 4, RAP, RichFaces and Vaadin.

- Support was added for Firefox version 18.

- For SUTs where only few components are named - if any - component recognition has been improved by adding a new Extra feature[(825)] called `qfs:label`. It represents a best match for a label, based on various algorithms including a search for the nearest label for components that don't have one of their own. Classical forms consisting of text fields with labels next to them benefit most from it.

- Handling the opposite case, where the same component name is assigned to more than one component, is now also handled via via the new option Validate component recognition during recording[(449)].

- QF-Test now integrates with the test management tool QMetry. See section 26.3[(327)] for further information.

- Components in QF-Test's detail view are now sized dynamically depending on the contained data. Combined with improved scrolling this vastly improves creating and maintaining test suites.

- The menu item $\boxed{\text{Replay}\rightarrow\text{Reset all}}$ can be used to stop the test, reset dependencies and global variables and kill all clients with a single action.

- On Windows QF-Test now uses the native file chooser by default, configurable via the option <u>Use native file chooser on Windows or macOS systems</u>[(421)].

- There are new operations to search for various kinds of duplicates, empty tests or illegal characters. The result dialogs for these and similar operations as well as searches have more options and exported results can now be read back.

- Search and replace operations now cover whole projects and can be interrupted in case they take too long.

- Tables holding variable definitions or parameter lists can now be sorted.

- It is now possible to navigate from a QF-Test component ID specified in a variable table directly to the component by right-clicking the respective table cell.

- Parameters of procedures can now be added to or removed from the variable definitions of all callers via the context menu of the parameter table.

- Recording procedures for windows with many components now works significantly faster.

- Whitespace at the beginning or end of variable names is now generally removed to avoid confusion.

- There is a new hotkey (configurable via <u>Hotkey for recording</u>[(439)], default $\boxed{\text{F11}}$) to start and stop recording directly in the SUT. The previous default binding for $\boxed{\text{F11}}$, component recording, has been moved to $\boxed{\text{Shift-F11}}$.

- There's a new kind of check for tree nodes that can handle whole (sub-)hierarchies at once, including indentation.

- Highlighting components in the SUT for event nodes now works even if the Component node for the event is not located in the same test suite.

- Killing the SUT and other processes now works much faster.

- Leading and trailing whitespace in <u>Feature</u>[(825)] attributes is now generally ignored. This implementation is fully backwards compatible, so it doesn't hurt if features were recorded with whitespace before.

- Line numbers are now displayed for tables where this is useful, for example in <u>Data table</u>[(566)] or <u>Check items</u>[(719)] nodes.

- In places where accidentally inserted TAB or line-break characters can do harm, most notably in variable definitions or checks, they are now represented as graphical symbols. This can be turned off via the option <u>Show symbols for tab and line break characters</u>[(427)].

New packages and procedures in the standard library `qfs.qft`

- The new `qfs.autoscreen` package holds procedures for locating and interacting with components on the screen based on some previously captured image.

- Several new utility procedures:
  ```
  qfs.utils.files.compareFiles
  qfs.utils.variables.deleteGlobalVariablesExcept
  qfs.utils.variables.deletePropertyGroup
  qfs.utils.variables.deleteOnePropertyVariable
  qfs.utils.variables.listAllMembersOfPropertyGroup
  qfs.utils.xml.loadPropertiesFromXML
  qfs.utils.sleep
  ```

Swing
- Various new procedures for interacting with Swing components:
  ```
  qfs.swing.checkbox.getValue
  qfs.swing.combobox.getIndexOfItem
  qfs.swing.combobox.getValue
  qfs.swing.combobox.getValueOfItem
  qfs.swing.general.doSelect
  qfs.swing.menu.selectMenuItemInHierarchy
  qfs.swing.menu.selectCheckMenuItemInHierarchy
  qfs.swing.radiobutton
  qfs.swing.radiobutton.getValue
  qfs.swing.radiobutton.select
  qfs.swing.tabbedpane.getIndexOfTab
  qfs.swing.tabbedpane.getValueOfTab
  qfs.swing.text.getText
  ```

SWT
- Various new procedures for interacting with SWT components:
  ```
  qfs.swt.checkbox.getValue
  qfs.swt.combo.getIndexOfItem
  qfs.swt.combo.getValue
  qfs.swt.combo.getValueOfItem
  qfs.swt.general.doselect
  qfs.swt.list.getIndexOfItem
  qfs.swt.menu.selectMenuItemInHierarchy
  qfs.swt.menu.selectCheckMenuItemInHierarchy
  qfs.swt.radiobutton
  ```

```
qfs.swt.radiobutton.getValue
qfs.swt.radiobutton.select
qfs.swt.table.getIndexOfColumn
qfs.swt.table.getRowIndexOfCell
qfs.swt.tabfolder.getIndexOfTab
qfs.swt.tabfolder.getValueOfTab
qfs.swt.text.getText
qfs.swt.tree.multicolumn
qfs.swt.tree.multicolumn.getRowIndexByCellText
qfs.swt.tree.multicolumn.getIndexOfNode
```

Web
- Various new procedures for interacting with web components:
```
qfs.web.checkbox.getValue
qfs.web.radiobutton
qfs.web.radiobutton.getValue
qfs.web.radiobutton.select
qfs.web.select
qfs.web.select.getItemCount
qfs.web.select.getIndexOfItem
qfs.web.select.getValueOfItem
qfs.web.select.getValue
qfs.web.select.selectValue
qfs.web.table
qfs.web.table.checks
qfs.web.table.checks.checkNumberOfRows
qfs.web.table.checks.checkNumberOfColums
qfs.web.table.recording
qfs.web.table.recording.activateItemRecording
qfs.web.table.recording.deactivateItemRecording
qfs.web.table.getColumnCount
qfs.web.table.getIndexOfColumn
qfs.web.table.getNameOfColumn
qfs.web.table.getRowCount
qfs.web.table.getRowIndexOfCell
qfs.web.table.getValueOfCell
qfs.web.table.selectCell
qfs.web.text.getText
```

Web
- More web utilities, specifically `qfs.web.ajax.uninstallToolkitResolver`, the new package `qfs.web.checks.style` with several procedures for checking CSS style attributes of DOM nodes and a complete rewrite of `qfs.web.browser.general.startAsNewProcess`.

Bugs fixed:

- The modified connection mechanism still crashed Java WebStart on some occasions and has been further fine-tuned to prevent that.

- The option settings for Errors that should be ignored^(494) were not saved to the system configuration file.

B.10.13 Version 3.4.14 - March 20, 2013

New features:

Web
- Support was added for Firefox version 20.

SWT
- Eclipse/SWT support was updated for Eclipse versions 4.2.2 and 3.8.2.

- The new procedure `qfs.swing.dialogs.handleSecurityWarning` in the standard library `qfs.qft` closes the security dialogs displayed by current JREs during startup of applets or WebStart based applications.

- The procedure `qfs.web.browser.general.startAsNewProcess` was updated to include security dialog handling.

- Another new procedure named `qfs.swing.startup.startWebstartSUT` can be used to run WebStart and connect to the SUT, also handling the security dialog automatically.

Bugs fixed:

- A workaround has been implemented for a bug in recent JRE security updates that could cause batch tests to simply stop in the middle of execution.

Web
- Firefox on Linux no longer crashes on shutdown.

Web
- For some MIME types downloading files did not work correctly with Firefox.

Web
- Checks for the enabled state could fail in Internet Explorer if the state was set via JavaScript.

Web
- Waiting for the absence of an element could fail if the element just became invisible instead of being removed completely.

- A Return^(592) in a Finally^(623) node could hide an exception thrown in the containing Try^(616) node.

- Using Java 7 on Mac systems could cause an exception during startup of QF-Test.

- Changes to the location of the search dialog and other dialogs were often ignored instead of automatically saved to the user preferences.

B.10.14 Version 3.4.13 - January 8, 2013

New features:

Web
- Support was updated for Firefox versions 18 and 19.

Web
- JavaScript error filters defined in the option <u>Errors that should be ignored</u>[(494)] are now applied to the complete message including the URL and line number of the source of the error.

Bugs fixed:

- The responsiveness of QF-Test severely suffered when opening many test suites.

SWT
- Geometry of events relative to an SWT Shell now includes the window title and borders.

SWT
- Highlighting widgets inside `CTabFolders` in Eclipse 4.2 now uses the correct offset.

- Errors happening outside a test case were not listed correctly in the report if the containing test set contained several test cases.

Web
- Downloading files with Firefox could cause a JavaScript error if the file `mimeTypes.rdf` was missing from QF-Test's Mozilla profile. This file is now created on the fly if necessary.

Web
- Closing the browser window of Internet Explorer 10 could cause the browser to deadlock instead of terminate.

Web
- QF-Test's embedding of Internet Explorer 10 caused the browser to run in IE9 mode. Now it start up in IE10 mode by default with the usual settings for compatibility mode enabled.

Web
- Replaying [Tab] keystrokes on Firefox did not always trigger the correct focus traversal.

B.10.15 Version 3.4.12 - November 16, 2012

New features:

SWT
- Eclipse/SWT support was updated for Eclipse versions 4.2.1 and 3.8.1.

Web
- Support was added for Firefox version 18.

Bugs fixed:

- The modified connection mechanism still crashed Java WebStart on some occasions and has been further fine-tuned to prevent that.

- The option settings for Errors that should be ignored[(494)] were not saved to the system configuration file.

Swing
- Timing for startup of Java WebStart based SUTs has been further improved to significantly increase the stability of this setup.

Web
- Some focus transitions were not triggered correctly when replaying events in a browser.

Web
- The filename shown in a download dialog was not always correct.

Web
- Internet Explorer 10 now reports the correct version information.

SWT
- Recording of drag and drop operations on CTabFolders in Eclipse 4.2 was broken.

SWT
- In some cases the height of SWT List items was not calculated correctly, leading to the wrong item being addressed in case of large lists with many items.

- Handling of an SUT with a combined SWT/AWT GUI was broken. Due to an error introduced in version 3.4.11 QF-Test was treating this as two SUT clients instead of one client with two GUI engines.

B.10.16   Version 3.4.11 - October 4, 2012

New features:

- QF-Test is now compatible with Java applets running on JRE version 1.6.0_35 or higher or JRE version 1.7.0_7 or higher. A security fix from Oracle broke the existing applet connection mechanism. The new mechanism should also improve stability of Java WebStart based SUTs.

Web
- Support was added for Firefox version 17.

Web
- Via the new option Errors that should be ignored[(494)] it is now possible to ignore specific JavaScript error messages while still treating others as an error.

- When copying multiple lines of a table in the detail view of QF-Test these are now also copied to the system clipboard as a list of TAB-separated values, thus enabling pasting into a spreadsheet.

- The new procedure `qfs.utils.logNativeMemory` in the standard library `qfs.qft` can be used to get information about the native memory use of an SUT client.

- Windows executables and installers created by QFS are now signed with an official certificate for smoother integration with Windows 7 and Windows 8.

Bugs fixed:

Web   - Downloads of large files were sometimes truncated.

Web   - The new INPUT types in HTML 5 are now mapped to INPUT:TEXT so that tests remain independent of whether the browser supports them.

- The command line argument <u>-runlogdir <directory></u>[(879)] now also works for daemon calls via `qftest -batch -calldaemon`.

- Empty formatted cells in an Excel 2010 sheet are now handled correctly by the <u>Excel data file</u>[(574)] node.

Web   - Toolbar icons for QF-Test's browser wrapper are now transparent and no longer look horrible on Windows 7.

B.10.17 Version 3.4.10 - August 7, 2012

New features:

Web   - Support was added for Firefox versions 15 and 16.

- The performance of complex QF-Test operations like finding references has been significantly improved for large projects on slow filesystems like network shares.

Bugs fixed:

Web   - Crashes of IE and Firefox were further reduced.

B.10.18 Version 3.4.9 - June 29, 2012

New features:

SWT   - Eclipse/SWT support was added for Eclipse "Juno" versions 4.2 and 3.8.

Web   - Support was added for Firefox version 14.

• Several new transformation options for nodes in the test suite were introduced.

Bugs fixed:

Web
• A significant memory leak in QF-Test's browser wrapper was plugged.

Web
• Several causes for browser crashes were fixed, both for Internet Explorer and Firefox.

Web
• Component recognition now also works correctly with components recorded in a flat hierarchy, even if the web page contains FRAME nodes.

Web
• The progress bar in Internet Explorer always remained visible even when a page was completely loaded.

Web
• If the path for the Firefox installation directory contained special characters, Firefox could not be started correctly from QF-Test.

SWT
• On Windows, popup menus in an Eclipse/SWT application were sometimes mistakenly considered to be invisible.

• Files are no longer overwritten without asking when an image is saved from a Check image[729] node.

• Component group[833] and Window group[832] nodes are now handled and integrated correctly when importing components from a test suite.

• Another special case for escaping '$' characters in component names has been fixed.

B.10.19 Version 3.4.8 - June 5, 2012

New features:

Web
• Support was added for the current Firefox version 13.

SWT
• Eclipse/SWT support was added for Eclipse versions 3.8M7 through 3.8RC2 and experimental support for Eclipse version 4.2M7 through 4.2RC2.

Web
• Support for the Richfaces toolkit was improved.

• The Excel data file[574] node now also handles Excel files with the '.xlsm' extension.

Bugs fixed:

- Escaping of '$' characters in component names was broken in the component and procedure recorder.

Web
- Enabled checks did not work correctly with Internet Explorer 9 running in IE 7 compatibility mode.

Web
- Firefox could crash when closing a browser window.

Web
- When recording procedures the calls for SELECT elements were created incorrectly.

B.10.20 Version 3.4.7 - April 24, 2012

New features:

Web
- Support was added for the current Firefox version 12.

- The new Procedure `qfs.swt.ctabfolder.openViewMenu` can be used to generically open the ViewMenu of a CTabFolder based on the tab name, both for Eclipse 3.x and Eclipse 4.

Bugs fixed:

- Script errors were not formatted correctly when shown in the error dialog. In the run log those errors were shown correctly.

- Excel 2010 files used in a data driver with an Excel data file[(574)] node were not closed correctly.

Web
- There's a new workaround for the workaround for the missing MSVC manifest in a DLL from Firefox 10 and higher. Problems with another missing MSVC DLL on Windows XP and deadlocks when loading a Flash component in Firefox on Windows have been fixed.

B.10.21 Version 3.4.6 - March 27, 2012

New features:

SWT
- Eclipse/SWT support was added for Eclipse versions 3.7.2 and 3.8M6 and experimental support for Eclipse version 4.2M6.

- The Procedures `qfs.swt.ctabfolder.closeTab` and `qfs.swt.ctabfolder.checkTabIsDirty` now also support Eclipse 4.

Bugs fixed:

- The fix for SUTs that use RMI with a custom implementation of the Service Provider Interface (SPI) could cause a subtle class loading problem that has been fixed.

- Deleting the last row in a variable table in QF-Test could lead to an exception when subsequently trying to move a table row.

Web • QF-Test's handling of JavaScript errors could cause Internet Explorer to crash.

Web • If QF-Test was accidently started with the `bin` directory of an older QF-Test version as its working directory, starting a browser on Windows could fail due to conflicts between DLL versions.

Web • Use of a special API in Internet Explorer can lead to very subtle deviations in JavaScript execution. QF-Test no longer uses this interface.

Web • On some systems the workaround for the missing MSVC manifest in a DLL from Firefox 10 and higher could cause a problem with another DLL and prevent QF-Test from starting Firefox.

B.10.22 Version 3.4.5 - March 12, 2012

New features:

Web • Support was added for the current Firefox version 11.

- The new Procedure `qfs.client.sendInput` in the standard library `qfs.qft` can be used to send text to the standard input of a process started by QF-Test.

Bugs fixed:

- A race condition between termination of and receiving output from a process running on a remote machine could cause some of the process' output to get lost.

- Cancelling a long search in a large test suite did not work reliably.

- Empty variable values in the Name[825] or Feature[825] attribute of a Component[823] are now interpreted correctly as "ignore name" or "ignore feature" instead of looking for "an empty name" or "an empty feature" which was bound to fail.

- The QF-Test daemon was not registered at the default port when started without the argument `-daemonport <port>`[871].

- SUTs that use RMI with a custom implementation of the Service Provider Interface (SPI) failed to connect to QF-Test.

- The combination of running multiple threads with the argument `-threads <number>`[(883)] in batch mode and connecting to a sub-process of the SUT with $(client):2 did not work correctly.

Web
- QF-Test now works around a problem with a missing MSVC manifest in Firefox 10, causing an error message about incorrect loading of the C runtime library.

Web
- Firefox 10 crashed when navigating to about:plugins.

Web
- A download triggered at JavaScript level via a link starting with '#' caused the browser window to close.

B.10.23 Version 3.4.4 - January 30, 2012

New features:

Web
- Support was added for current versions of Firefox 9 and 10.

SWT
- Eclipse/SWT support was added for Eclipse version 3.8M5 and experimental support for Eclipse version 4.2M5.

SWT
- New checks were introduced for SWT `Decorations`, small icons typically shown next to a text field. You can now check the visibility, icon and tool tip of a `Decoration`.

- Iteration ranges can now also be used when setting up data at script level via the `databinder` module or by calling one of the Procedures `qfs.databinder.bindList` or `qfs.databinder.bindSet` in the standard library `qfs.qft`. See section 21.4[(268)] for further information.

- The thumbnail images shown for screenshots in the HTML report are now of higher quality.

Bugs fixed:

- A small change in QF-Test version 3.4.3 regarding the RMI communication between QF-Test and the SUT was undone because of possible side-effects when the SUT was using RMI itself.

- When using Check image[(729)] nodes with an advanced algorithm and a timeout QF-Test always used up the full timeout even when the image matched immediately.

- When a CHECK node with a timeout fails, QF-Test used to show the initial state of the target component in the run log. Now it shows the state at the end of the timeout period instead.

- When using Drag&Drop to move nodes within QF-Test the tree could jump erratically in some cases.

- Trying to remove the same node twice from a search result dialog could lead to an exception.

- In the standard test suite library `qfs.qft` the Procedure `qfs.shellutils.touch` now correctly updates the timestamp for existing files and the Procedure `qfs.autowin.getURLFromIE` now also works with Internet Explorer version 6.

SWT
- When recording a mouse click on an SWT ToolItem that opens a drop down menu QF-Test now records the coordinates so the event can replay correctly.

Web
- If an alert dialog was shown after closing a modal browser window, the main browser window could remain blocked after closing the alert.

Web
- JavaScript errors in Internet Explorer were not logged correctly in QF-Test's run log.

Web
- In some cases the initial loading of a web page did not succeed.

Web
- Highlighting the selected component in a browser on Linux has been improved.

Web
- Web pages with a Flash element were sometimes not displayed correctly on Linux.

Web
- The name recorded for file upload on Linux contained the Windows file separator character '\'

Web
- There were problems with browser startup on special Windows versions if QF-Test was installed below a directory with blanks in its name, e.g. "C:\Program Files (x86)"

B.10.24 Version 3.4.3 - October 27, 2011

New features:

Web
- Support was added for current versions of Firefox 8 and 9.

- The Excel data file[(574)] node now supports reading `.xslx` files in Excel 2010 format.

- The performance of some advanced image check algorithms was significantly improved.

- It is now possible to fetch option values in a script via the new method `rc.getOption`. See section 48.5[(917)] for details.

Bugs fixed:

- When switching test suites by clicking a suite tab the keyboard focus was not correctly restored.

- The keyboard shortcut Ctrl-D for setting a bookmark did not work.

- Jumping from a node in the debugger window to the same node in the test suite now works again.

- In some special cases QF-Test could kill the wrong process on Windows when trying to terminate the SUT.

B.10.25 Version 3.4.2 - September 15, 2011

New features:

Web
- Full support was added for Firefox 6 and experimental support for Firefox 7.

- For easier integration with continuous integration tools like Hudson/Jenkins or TeamCity the exit code of a batch run can be forced to 0 even in case of warnings, errors or exceptions with the help of the command line arguments `-exitcode-ignore-warning`[(872)], `-exitcode-ignore-error`[(871)] and `-exitcode-ignore-exception`[(871)].

Web
- If the option `Replace no-break space with normal space` is activated, QF-Test now represents no-break space characters in a web page as normal space characters in checks and Component[(823)] node attributes to avoid confusion.

SWT
- QF-Test now supports capture and replay of "expand" and "collapse" events for SWT ExpandItems.

Bugs fixed:

SWT
- The integrated help viewer of Eclipse uses a separate SWT Display on Windows which caused problems during replay. It is now consistently represented with GUI engine[(817)] "swt1".

Web
- Direct execution of JavaScript code now also works in Internet Explorer 9 despite stronger security restrictions.

Web
- The AJAX resolver for GWT has been updated to version 1.4.0 to fix filtering of DIV nodes from the component hierarchy. See section 49.2[(988)] for details about AJAX resolvers and their versioning scheme.

Web
- Opening a modal popup window could crash Firefox 6 on Windows.

Web
- Double clicks are now recorded correctly in Internet Explorer.

Web
- A Fetch text[(740)] node for a SELECT node now retrieves the text of the selected OPTION node and Fetch text nodes for OPTION elements represented as items are working correctly.

- Printing unicode strings in Jython could cause problems when running in batch mode.

- The content of Data table[(566)] nodes was missing in the run log.

- A Wait for client to connect[(670)] node with a very short timeout did not throw a `ClientNotConnectedException`[(855)] when single-stepping in the debugger.

- The QF-Test daemon was leaking threads when starting a remote SUT process for use with the procedure `qfs.daemon.startRemoteSUT`.

- When updating failed image checks from the run log the size of the image was not always updated correctly.

- When recording procedures, QF-Test failed to create a procedure in case the respective component had no value for a placeholder like <COMPFEATURE>.

B.10.26   Version 3.4.1 - July 15, 2011

New features:

Web
- Full support was added for Firefox 5 and experimental support for Firefox 6.

SWT
- Eclipse/SWT support was updated for Eclipse version 3.7 Indigo.

- Groovy was updated to version 1.7.10.

- Error messages for errors in Groovy scripts now show proper context and line numbers.

SWT
- QF-Test can now handle custom SWT components that hide their child Widgets by overriding `getChildren()` to return an empty array.

Web
- Support was added for ExtJS version 4 and for RichFaces version 4. See section 49.2[(988)] for details about the support AJAX toolkits and versions.

- The As regexp[(825)] attribute of Window[(812)] and Component[(823)] nodes can now be defined as a variable value.

Web
- The startup sequence created by the quickstart wizard for web applications now has a smarter layout and is easier to adjust.

- On Windows systems QF-Test now makes use of the always-on-top attribute to force SUT windows to the front even if normally prevented by Windows. To disable this feature, deactivate the option Force window to the top when raising[(467)].

- A new check was added to directly check the icon of a component instead of having to capture the whole component and define a check region.

Bugs fixed:

- When loading the test suites to restore the last session upon startup, QF-Test did not check for auto-save files.

- It was possible to start parallel tests in the same context by hitting Return at the right time. This could lead to confusing results in the run log and even to exceptions.

Swing
- Swing applications started via Java WebStart have two independent application contexts, a user context and a security context. Unless the GUI engine[(633)] attribute of a SUT script[(631)] node explicitly defines the engine, the script is now always correctly executed in the user context and not in the security context.

Swing
- Special support for JSlider components was missing.

- The special variable ${qftest:runid} did not expand to the correct value if the runid was specified explicitly via -runid <ID>[(878)] in batch mode.

Web
- A forced download triggered via a special PHP construct could cause Internet Explorer to crash.

Web
- Clearing the cache for Firefox did not always work correctly.

Web
- Trying to record a check for an XML document displayed in Internet Explorer could crash the browser.

Web
- It is now possible to record checks for applets even when they are embedded in QF-Test's browser wrapper.

- The Groovy implementation of `qf.logError` was broken.

- In a workbench view with many open test suites and some test suite tabs scrolled out to the left, coordinates for dragging test suite or node to different test suites were off, leading to surprising results.

- QF-Test was leaking resources when taking Screenshots in an Eclipse/SWT SUT.

- Calling `Display.asyncExec` from a timer could crash Eclipse on Linux Gtk with 64bit.

- The order of SWT.Activation events could show very subtle differences between manual replay and automatic replay with soft events.

- A very long search string could mess up the size of the search and replace dialog.

- Markers in run logs did not propagate beyond 'External log' nodes.

- Fallback to lazy binding did not work for not-yet-bound properties.

B.10.27   Main new features in version 3.4

Support for Java 1.4.2 was dropped by both Jython and Groovy so QF-Test now requires at least Java version 1.5 for itself and the SUT.

Following is an overview over the main new features that have been implemented for QF-Test since the last production version 3.2.

Description	Further info
Jython updated to version 2.5.1	
Groovy updated to version 1.7.6	
Support for Eclipse/SWT 3.6.2 and 3.7	
Web testing support for Firefox 4 and Internet Explorer 9	
Integration with Continuous Integration tool Jenkins/Hudson	section 27.4[343]
Fault-tolerant image checks	chapter 57[1167]
Advanced search and replace	section 2.3[19]
Iteration ranges for data drivers	Iteration ranges[567]
Direct interaction with web servers	Server HTTP request[802] node
Extended import functionality	section 24.3[301]
Debugger integrated into the workbench	
Ability to create a test suite from a run log	section 7.1.8[123]
Automatic splitting of run logs and graceful handling of OutOfMemoryErrors	Option Minimum size for automatic splitting (kB)[505].
Improved backwards/forwards navigation	
Improved pkgdoc and testdoc documentation	Command line arguments `-testdoc-teststeps`[882], `-testdoc-nodeicons`[882] and `-pkgdoc-nodeicons`[875]
Improved coordinate abstraction for mouse events	Option Record MouseEvents without coordinates where possible [441], Mouse event[681]
Variables with default values and many additional special variables	section 6.5[108]

Table B.8: New features in QF-Test version 3.4

B.10.28   Version 3.4.0 - May 26, 2011

New features:

SWT

- Support for testing Eclipse/SWT version 3.7 was updated.

- All Data binder nodes now have an attribute named Iteration ranges[567] that can be used to limit the iteration to a single element or one or more intervals, which is very convenient when implementing tests, continuing a test that had to be stopped or performing a retest with a limited subset.

- In workbench mode the current session with the currently open test suites and the positions therein is now saved when QF-Test is closed and restored upon startup,

provided the option <u>Restore last session on startup</u>⁽⁴²⁰⁾ is active. Also added to the workbench view were the ability to save all modified test suites via the ⟦File⟧ menu and switching between recently active test suites via [Control-F6].

- Moving backwards/forwards between selected nodes has been unified and should now work consistently even for special cases. After recording components you can jump directly to the recorded node by moving forwards, even if no node was actually recorded and the existing node is in a different test suite.

- Filter capabilities were added to the search/replace result and other multi result dialogs.

- Marks can now also be set for nodes in a run log, so multi-step search operations are now also available in run logs.

- The quickstart wizard has improved support for launching applets in a browser and for setting the compatibility mode in internet explorer, backed by the procedures `qfs.web.browser.general.startAsNewProcess` and `qfs.web.browser.settings.enableCompatibilityMode` in the standard library `qfs.qft`.

- When started in batch mode with the command line argument <u>-verbose [<level>]</u>⁽⁸⁸³⁾, QF-Test now displays exception, error and warning counts for each Test case.

- The new package `qfs.autowin.acrobat` in the standard library `qfs.qft` contains methods to save a PDF document showing in an Acrobat window or to copy its text content to a file via the clipboard (Windows only). These are complemented by the `qfs.utils.files` package with procedures to verify the content of a text file.

- The new methods `setProperty` and `getProperty` in the generic scripting module `qf` (see <u>section 48.6</u>⁽⁹³⁷⁾) can be used to associate arbitrary data with a GUI element, e.g. to cache some resolved feature or name values or other information that is too expensive to compute every time a resolver method is called.

- The `TestRunListener` API (see <u>section 52.7</u>⁽¹⁰⁸⁷⁾) has been extended to provide the number of exceptions, errors and warnings that occurred below a given node in the `TestRunEvent` of a call to `nodeExited`.

Web
- The AJAX resolver for GWT has been updated to version 1.2.0. It now filters irrelevant DIV nodes from the component hierarchy and has improved support for obfuscated GWT class names. See <u>section 49.2</u>⁽⁹⁸⁸⁾ for details about AJAX resolvers and their versioning scheme.

Bugs fixed:

- Early checking to handle OutOfMemoryErrors more gracefully could impact test execution speed significantly with JDK 1.5 on Windows. Tests should now run at full speed again with all JDK versions.

- Some of the advanced image check algorithms did not work correctly with JDK 1.5. All algorithms are now fully supported for JDK 1.5 and higher and return consistent results, independent of the JDK version, except for resizing, which is implemented differently in the various JDK versions and can lead to minor deviations in the resulting probability.

- The procedure `qfs.shutil.copy` in the standard library `qfs.qft` was broken. It now copies files and directories correctly, recursively if necessary and handles special cases like copying a directory into itself.

Web
- Check image[(729)] nodes now work correctly in Firefox 4.0.

Web
- Scrolling an element visible was not always possible in IE9 and some special cases with a deeply nested hierarchy.

Web
- In IE9 nodes were not highlighted properly and highlighting could interfere with recording image checks.

Web
- Browser settings for IE9 did not always take effect.

Web
- A download operation triggered from a popup window could crash the browser.

- A running test in the QF-Test daemon can now be interrupted and stopped cleanly via `-stoprun`[(880)] even if the test is currently waiting for something.

- The structure of the JUnit XML report is now compatible with Atlassian Confluence.

- In Groovy the method `rc.getProperties()` was not implemented correctly.

B.10.29   Version 3.4M2 - March 10, 2011

New features:

SWT
- Support was added for testing Eclipse/SWT versions 3.6.2 and 3.7 Milestone 5.

- The search and replace dialogs are now integrated so it is possible to start with a search and continue replacing, based on the same criteria. Also, the replace dialog is no longer modal so it can be kept open while editing.

- Integration of the debugger functionality into the workbench view is now complete. The current variable bindings are displayed automatically whenever the debugger is entered unless the option Show variable bindings automatically[(499)] is turned off.

- The keyboard shortcuts ⌊Alt-1⌋ through ⌊Alt-9⌋ navigate directly to the respective test suite tabs in the workbench view.

- Pressing ⌊F2⌋ in the test suite tree now moves focus directly to the details field that is most relevant for the display name of the node.

- The new Procedure `qfs.client.renameClient` in the standard library `qfs.qft` can be used to rename an SUT client after it connects. This is especially useful for an SUT that launches multiple sub-processes that connect as client:2, client:3 etc.

- The new special variable $_{qftest:skipnode} can be used to skip execution of a node. Please see section 6.5[108] for details and applicability.

SWT
- Hyperlinks in the Eclipse FormText widget are now recognized as sub-items.

- Advanced image checks are now also available to scripts via the new run context method `rc.checkImageAdvanced`. Please see section 48.5[917] for details.

- The various `rc.check*` methods in the run context now return the result of the evaluated condition. Please see section 48.5[917] for details.

- The image of a Check image node in the test suite and the run log as well as screenshot images can now be opened in a separate window by double-clicking the image (run log only) or clicking the respective button above the image.

- There is a new utility module for scripting called `qf`, documented in section 48.6[937]. It contains variants of commonly useful methods like `isInstance` and the ability to add messages to the run log even when no run context is available, e.g. in TestRunListener, Checker or Resolver implementations.

- Templates for procedure recording have become more powerful. Procedures can now be created conditionally by placing a @CONDITION tag with an arbitrary Jython or Groovy script in the comment. Please see chapter 54[1156] for details.

Web
- The Procedure `qfs.web.browser.settings.setProxy` in the standard library `qfs.qft` now also supports passing a URL for automatic proxy configuration.

Bugs fixed:

- Switching execution to a different node in the debugger could unbalance the parent/child relationships of the associated nodes in the run log.

- Setting and unsetting options at script level could mess up default option values.

SWT
- Improved double click replay for SWT on Windows.

SWT   • Some selection events for SWT now receive special treatment for focus handling.

SWT   • Empty lines at beginning of an SWT StyledText widget are now handled correctly on Linux.

           • Errors in nested test cases are now listed correctly in the report.

Web   • Proxy server settings for Internet Explorer 9 are now working correctly.

Web   • Image checks are now working correctly with Firefox 4.

Web   • JavaScript and authentication dialogs are now working correctly in Firefox 4.

           • QF-Test now issues a warning each time you first try to modify a write-protected test suite.

           • Doctags for Dependency nodes are now listed correctly in pkgdoc documentation.

B.10.30   Version 3.4M1 - February 3, 2011

New features:

- Groovy was updated to version 1.7.6.

- QF-Test now actively monitors the memory available in the Java VM for running tests. When available memory runs low, the test is aborted cleanly and the run log is saved. This also prevents extreme slow-down when memory gets tight.

- To prevent running out of memory in the first place, QF-Test now splits run logs automatically by default, based on the option Minimum size for automatic splitting (kB)[505].

- Managing test suites in the workbench has been improved. The tab for the active suite is now shown with a bold font, you can drag tabs to reorder test suites and double-clicking to the right of existing tabs creates a new suite.

- The search dialog now has additional options for limiting the search and it is also possible to search all loaded test suites at once. To prevent visual overload for the common case, the search dialog can now be toggled between simple and advanced mode.

- The debugger has been mostly integrated into the workbench, except for the variables view which still requires the separate debugger window.

- There is a new recording mode for web applications that tells QF-Test to ignore individual events and record entire GET or POST requests instead, which are then represented by the new Server HTTP request⁽⁸⁰²⁾ node. Upon replay QF-Test interacts directly with the HTTP/HTTPS server without requiring a browser. This can be useful for load testing or mass data entry.

- For Swing and SWT components with an icon QF-Test now records a hash value derived from the icon image as an Extra feature. As long as the icon remains stable this should lead to improved recognition, especially for toolbar buttons without a label. The default state of the image hash extra feature is 'Should match'. You can change it to 'Ignore' for components where the icon is not reliable and to 'Must match' for components where the image is the best way to identify the component.

- There is a new special variable syntax that allows specifying a default value for a variable. This is extremely useful for things like generic components or really every place where a variable has a reasonable default, because that default value is then tightly coupled with the variable lookup itself and doesn't have to be specified at Sequence or test suite level. The syntax is ${default:varname:defaultvalue}.

- Additional special variables are now available to retrieve the name of the currently executing Test set or Test step in addition to the already available Test case name. Also, counters were added for the number of test cases executed so far, failures etc. Please see section 6.5⁽¹⁰⁸⁾ for details.

- Where available, the dialog showing the methods available for a component now also displays the results of calling those methods that represent Java Bean values.

- The option Log successful advanced image checks⁽⁵¹⁰⁾ can be activated in order to store the results of a successful advanced image check in the run log. This can be useful while tuning the parameters for an advanced image check algorithm.

- The new command line option `-verbose [<level>]`⁽⁸⁸³⁾ for batch execution with or without accessing a daemon causes QF-Test to print progress information during a test run.

- The new Procedure `qfs.utils.dumpHeap` in the standard library `qfs.qft` can be used to trigger a heap dump for the Java VM of the SUT when it uses too much memory. This can be detected with the help of the Procedure `qfs.utils.logMemory` which now returns the amount of memory used.

Bugs fixed:

- When replaying mouse events without coordinates on items, QF-Test no longer targets the center of the item, but the position 5 pixels from the left, vertically centered. Apparently some events are sensitive to whether the click hits the text

of an item and not just its bounding box, so short items in a broad column could cause problems.

- Calling stored procedures in a database could fail using the default Jython bindings.

- If the debugger was turned off for normal use, debugger-stepping after hitting a breakpoint could occasionally fail so the test simply continued.

- A positive search result for the type of an extra feature could cause an exception.

- Blank characters in names for saving external run logs are now turned into an underscore because Windows doesn't handle trailing blanks in directory names gracefully.

- The input file for a CSV data file node is now closed immediately after reading the data.

- Recursive procedure calls could cause problems when importing procedures or analyzing references.

B.10.31   Version 3.3.0 - November 30, 2010

Incompatible changes:

- Support for Java 1.4.2 had to be dropped in order to update Jython and Groovy. QF-Test now requires at least Java version 1.5.

- Interaction between Jython and Java has changed slightly between Jython 2.1 and 2.5 so occasionally scripts that worked in QF-Test may now fail or behave differently.

New features:

- Jython was updated to version 2.5.1. Most scripts should continue to run without problems. Incompatibilities may arise due to the changes in character encoding and because Java strings are now uniformly treated as unicode strings in Jython.

- Groovy was updated to version 1.7.2.

SWT   - Support was added for Eclipse/SWT version 3.6.1.

Web   - Experimental support was added for Firefox 4 and Internet Explorer 9.

- QF-Test now integrates well with the increasingly popular Continuous Integration system Hudson/Jenkins. This includes JUnit as an additional target format for QF-Test reports. Please see section 27.4⁽³⁴³⁾ for further information.

- Image checks are no longer restricted to testing for exact pixel-by-pixel equality. Several flexible algorithms now allow image checks with a configurable degree of deviation from the expected image. Please see chapter 57⁽¹¹⁶⁷⁾ for details.

- Besides components you can now also import test cases and procedures from one test suite into another as described in section 24.3⁽³⁰¹⁾.

- When coordinates for mouse events are irrelevant, QF-Test now removes them altogether instead of setting them to (1,1). Upon replay the event is then aimed at the center of the target component or item. See also option Record MouseEvents without coordinates where possible ⁽⁴⁴¹⁾.

- By combining search and replace operations with marks it is now possible to narrow searches in several steps, giving very fine-grained control over where to perform replacements. Instructions and examples are provided in section 2.3.3⁽²⁷⁾.

- Moving backwards or forwards between selected nodes now works across test suites and even run logs. Thus a single mouse click or keystroke is now sufficient to return when following a procedure call to its target in a different test suite or when jumping from a run log to the respective node in the suite.

- Several new toolbar buttons were added and all the buttons now look much better. The menu structure also was cleaned up with operations on tree nodes now collected under the new Operations menu, among them many new node transformations. Also, several new keyboard shortcuts were introduced. A summary of all keyboard shortcuts is provided in appendix C⁽¹³⁹⁷⁾.

- The layout of pkgdoc and testdoc documentation now equals that of reports, including node icons and support for test steps. These can be configured via the command line arguments `-testdoc-teststeps`⁽⁸⁸²⁾, `-testdoc-nodeicons`⁽⁸⁸²⁾ and `-pkgdoc-nodeicons`⁽⁸⁷⁵⁾ or the respective options in the dialog for interactive creation of those documents.

- 'Skipped' and 'Not implemented' tests can now be ignored in the report via the command line argument `-report-ignoreskipped`⁽⁸⁷⁷⁾ or the respective option in the interactive dialog.

- Daemon tests can now be stopped at the command line via the new argument `-stoprun`⁽⁸⁸⁰⁾.

- The comment of a Procedure, Test case or Dependency node can now be filled automatically with doctags for parameters, characteristic variables and author by right-clicking the node and selecting ⟦Additional node transactions→Fill comment⟧ from the context menu.

- The `TestRunListener` API now also provides information about whether a test run was terminated normally (with or without exceptions) or aborted. See section 52.7^(1087).

- The new method `rc.callTest` in the run context can be used to call a Test case or Test set. In contrast to `callProcedure`, `callTest` can only be called from Server script nodes. It returns the exit status of the called test.

- The content of QF-Test configuration files is now sorted which significantly improves handling of those files in a version control system.

- Several procedures were added to or refined in the standard library `qfs.qft`. Its documentation is now also available online and indexed by the full text search engine on the QFS website. A link to the online text search covering all technical material is provided via the menu ⟦Help→Online search...⟧.

- It is now possible to create a test suite from a run log.

- It is now possible to pass GUI elements from an SUT script as individual targets to QF-Test's component recognition mechanism via the new run context method `overrideElement`. See also section 11.2.7^(171).

Bugs fixed:

Web
- When started from QF-Test, Internet Explorer 8 always ran in IE7 compatibility mode. This is no longer the case. Compatibility mode can now be actively selected in the quickstart wizard or via the procedure `qfs.web.browser.settings.doStartupSettings` in the standard library `qfs.qft`.

SWT
- On Linux QF-Test could sometimes crash SWT/Eclipse applications with disposed `Combo` popups.

SWT
- Test replay for Eclipse/SWT on Linux is now much more robust, especially when handling menus.

SWT
- Trailing empty newlines in `StyledText` widgets are now handled correctly on Linux.

- Creating reports with many screenshots now requires much less memory.

Web
- Downloads in Internet Explorer should now work consistently.

Web
- In case of overlapping nodes with transparent nodes on top QF-Test could pick the wrong node for check recording. It now lets the browser make the decision instead. Switching back to the old algorithm can be done via the option <u>Let the browser determine the target element for check recording</u>[(493)].

- A memory leak in QF-Test resulting from SUT clients printing a lot of output to the shared terminal in workbench view was plugged.

B.10.32   Version 3.2.2 - July 15, 2010

New features:

- Support was added for testing Eclipse/SWT version 3.6 "Helios".

SWT
- Multiple file selection in an SWT FileDialog can now be recorded and replayed.

Bugs fixed:

- Replay performance could suffer severely if the user's home directory was located on a network drive.

Swing
- A workaround for an AWT weakness introduced in QF-Test 3.2.1 could impact replay performance for AWT/Swing.

Swing
- Yet another performance problem resulting from very large hierarchies of Swing menus has also been fixed.

Web
- Several special cases for uploading files with Internet Explorer are now handled correctly.

Web
- Downloading files with Internet Explorer now works correctly in case of HTTP redirection.

B.10.33   Version 3.2.1 - May 11, 2010

Bugs fixed:

- With a 1.4 JDK in interactive mode QF-Test would only start if given a file to open.

Web
- Web-testing now works again on Windows 2000.

- The QF-Test debugger used to show duplicate entries on the variable stack for Procedure call[589] and Test call[531] nodes.

- There was a memory leak in connection with interactive editing of test suites with Check image[729] nodes for large images.

- In very special cases an import of components into their own suite could result in incorrectly updated references.

- Saving entries for zipped run logs could occasionally fail on Windows due to a problem in the underlying library. QF-Test now works around that.

- Two problems were fixed for SUTs with more than one GUI engine, e.g. a WebStart application or combined SWT/Swing. In such a case QF-Test used to log more than one screenshot of the main screen after an error and sometimes connections from the second engine could fail.

Web
- The upload dialog for web pages was not handled correctly for all browser versions and the procedure `qfs.web.input.fileUpload` in the standard library `qfs.qft` did not handle all cases correctly.

- In recent QF-Test versions there was an inofficial global Jython run context `rc` available in addition to the local one. In special cases this could cause subtle namespace problems, so the (still inofficial) global version was renamed to `__rc`.

- Several minor problems and occasional exceptions in connection with the result list dialogs were fixed.

B.10.34   New features in QF-Test version 3.2

Following is an overview over the main new features that have been implemented for QF-Test version 3.2. For detailed information about the differences between QF-Test 3.0 and 3.2, please see the following sections with the release notes for versions 3.2.0 and 3.1.x.

Description	Further info
Support for Web testing in combination with Java applets	Requires JDK 1.6.0_10 or higher
JDK instrumentation is no longer required	section 45.1[899]
Workbench view for a modern user interface	
Test step[539] nodes for detailed reports	section 22.1[272]
Refactoring support with automatic update of references	
Integration with the open source test management tool TestLink	section 26.5[331]
New 'Best Practices' part in the manual	part II[393]
Eclipse/SWT support updated for the latest versions including GEF	section 52.6[1083]
Incremental search in all of QF-Test's text fields	section 2.3.1[23]
Result list dialogs for complex operations	
Component method spy	section 5.12[96]
Options can be set at script level	section 11.2.6[170] and section 48.5[917]
Relative and test suite-local procedures	section 8.5.2[137] and section 8.5.1[137]
New-style Check nodes and support for custom Checkers	section 52.5[1073]
Static validation of references	
Automatic parameterization of procedures	section 8.5.4[138]
Persistent bookmarks and transient marks	section 39.1.5[430]
New APIs: ItemResolver, Checker, ClassNameResolver and TestRunListener	section 52.4[1062], section 52.5[1073], section 52.1.9[1026] and section 52.7[1087]

Table B.9: New features in QF-Test version 3.2

B.10.35   Version 3.2.0 - March 25, 2010

New features:

SWT
- Support for Eclipse/SWT has been brought up-to-date with the current versions 3.5.2 and 3.6M6.

Swing
- Swing components are no longer highlighted by exchanging their foreground and background color but by painting a rectangle, similar to SWT and Web. Target sub-items are now highlighted as well.

- The new Boolean check[713] node with Check type identifier attribute 'visible' is similar to Wait for component to appear[771] node, but in contrast to the latter is is listed in the report as a check and it has different default settings for the result handling.

- The most recently accessed component and possibly item in the SUT can now be retrieved in Jython and Groovy SUT script nodes via `rc.getLastComponent()` and `rc.getLastItem()`.

Web   • Selecting a file for upload in a web form can now be recorded and replayed directly on Windows. For a cross-platform implementation it is still necessary to call the procedure `qfs.web.input.fileUpload` from the standard library `qfs.qft`.

- Additional features were added to the result lists for complex operations. For example, you can now perform actual replacements in the result list for the replace dialog or you can update multiple failed checks at once from the error list in a run log. Besides, the result list for an operation that takes time is now filled continuously and the operation can be aborted if necessary.

- Recent search expressions are now saved in the user configuration.

- Dependency[548] nodes inside Packages are now listed in the pkgdoc documentation unless disabled via the command line argument -pkgdoc-dependencies[875] or the respective interactive option.

Bugs fixed:

Web   • Right-clicks for JavaScript-based popup menus no longer need to be replayed as hard events in order to trigger the popup reliably.

Web   • When recording checks the z-order of DOM nodes is now properly taken into account so that QF-Test no longer records nodes that are actually in the background, covered by other nodes.

B.10.36 Version 3.1.4 - February 9, 2010

New features:

Swing   • It is no longer necessary to instrument standard JDKs or JREs, though it doesn't hurt to keep existing instrumentation.

SWT   • Support was added for the latest Eclipse/SWT version 3.6.

Web   • QF-Test now supports Firefox 3.6.

- The new part II[393] 'Best Practices' in the manual describes recommended procedures based on lessons learned from several customer projects and user feedback that will hopefully help you to find the best strategy for using QF-Test in your own projects.

- Reports have been restructured and further extended. Check nodes now have an optional Name attribute for their representation in the report. Besides, the performance of report creation has been drastically improved and its memory impact reduced.

- The tree display in run logs now shows expanded values instead of variables. This can be toggled via the ‖View‖ menu.

- Compactification for run logs is now more efficient and special placeholder log entries show where nodes were removed during compactification.

SWT
- Support for the Eclipse Graphical Editing Framework (GEF) is now included by default and works correctly with multiple concurrent GEF editors. See section 52.6[(1083)] for further information.

Web
- The new procedure `qfs.web.input.fileUpload` in the standard library `qfs.qft` handles INPUT:FILE nodes for the various browser variants.

- TestLink integration has been simplified (section 26.5[(331)]).

- For Swing and SWT tables and multi-column trees it is now possible to check a whole row at once.

- ExtraFeatureResolvers can now also be used for SWT and Swing. These are described in section 52.1.11[(1028)].

- The standard library `qfs.qft` now includes wrapper procedures for the autowin module in the package `qfs.autowin`. These can be used to check for the existence of a window and to close it on Windows systems.

- The runid of the current test run now is now available via ${qftest:runid}.

- Special variables or properties of type group:name can now be used for Characteristic variables.

- For nodes like Test step or Test set with several tabs for variable bindings the number of bound variables is now displayed in the tab which makes it much easier to locate those bindings.

Bugs fixed:

Web
- Several causes for IE crashes were fixed.

Web
- Wait for absence now works correctly for invisible nodes.

- QF-Test doesn't get blocked as easily when the SUT generates a lot of output and the JTextArea for the shared terminal cannot keep up.

- The internal variable stack of QF-Test could get messed up in some special cases.

- The procedure builder now correctly handles '_' characters in package or class names.

B.10.37 Version 3.1.3 - December 1, 2009

New features:

- The new Test step[(539)] node can be used to provide further structure and document tests below Test cases[(518)]. Test steps are listed individually in the report unless deactivated with -report-teststeps[(878)]. Similar effects can be achieved by marking arbitrary nodes with the doctag @teststep, optionally followed by a name for the step.

- The report now also can list checks (off by default) and warnings (excluding component recognition warnings, on by default) integrated into the test step listing and also shows icons for nodes. The respective settings can be changed via -report-checks[(876)], -report-warnings[(878)] and -report-nodeicons[(877)] in batch mode or the respective options in the dialog for interactive report creation.

- In the workbench view there is now a context menu for the test suite tabs.

- All Comment attributes can now be edited in an external editor similar to scripts.

- There are new toolbar buttons in the run log window for navigating to the next/previous error or warning.

- The incremental search feature now has more intelligent scrolling for hits and also allows scrolling the component being searched via the (Up), (Down), (Page up) and (Page down) keys.

- Dependencies[(548)] can now optionally be resolved in a namespace which allows having different sets of dependencies for different parts of a test so that the dependencies don't interfere with each other. See the Dependency namespace[(553)] attribute for details.

Bugs fixed:

- Due to the JIDE libraries used, the new workbench view does not work with JDK 1.4. It now gets deactivated automatically when starting QF-Test with a 1.4 JDK.

- Several exceptions caused by the new workbench view and the multi result dialogs were fixed.

- In the new workbench view QF-Test could lose track about which suites are currently loaded, causing a suite to get loaded more than once, leading to confusing behavior and sometimes an exception.

- The new mechanism for on-the-fly SWT instrumentation only handled one SWT plugin, so that in case of a multi-platform Eclipse based application with several SWT plugins the correct plugin for the current platform could remain uninstrumented.

- Calling `stopRun` could cause the shared run context in the QF-Test daemon to become unusable.

SWT
- Text checks for SWT Group widgets were missing.

Web
- Fixed automatic scrolling of web pages for image checks and replay of hard events.

- The popup sub-menu for setting and removing marks was sometimes empty.

B.10.38 Version 3.1.2 - October 15, 2009

New features:

SWT
- Support was added for the latest Eclipse/SWT version 3.5.1.

Web
- We have started work on special support for several AJAX web toolkits, including Ext GWT, Ext JS, GWT, qooxdoo and RAP. Instead of deeply nested hierarchies of DIV nodes QF-Test now operates in terms of the actual rich interface of buttons, labels, trees etc. For information on how to activate this feature in its current state, please get in touch with QFS.

- QF-Test now comes with support for interacting with the open source test management tool TestLink. Please see section 26.5[331] for details.

- By default QF-Test now starts with a "Workbench View" where all open test suites are arranged as tabs in a single window. This feature is work in progress. In case it causes problems or if you prefer the old style, you can switch back to multiple windows via the ‖View‖ menu.

- The new function ‖Analyze references‖ in the context menu of a node can be used to statically validate the correctness of a test suite by verifying the existence of the target nodes of Procedure calls[589], Test calls[531], Dependency references[551] and references to Components[823].

- Operations that search for or result in changes to multiple nodes now bring up a mass result dialog listing the affected nodes. You can either navigate to specific nodes directly from that dialog or use it to set marks on some or all of the resulting nodes.

- Nodes in a test suite can now be marked with markers in four different colors for easier reference and quick navigation. Marks can be set either individually via the menu ⟦Edit→Marks⟧ or the context menu, or from a mass result dialog.

- All text fields, terminals etc. in QF-Test's user interface can now be searched locally by pressing ⟦Ctrl-F⟧ when the field owns the keyboard focus. Details are described in section 2.3.1[(23)].

- The new Component Method Spy retrieves and displays fields and methods of Swing or SWT components as well as the JavaScript properties of a DOM node in the web client (see section 5.12[(96)]).

- After recording a new sequence and transforming it into a Procedure[(586)] you can select ⟦Parameterize node⟧ from its context menu to tell QF-Test to automatically extract parameters for it from the contained nodes.

SWT
- The Eclipse/SWT instrumentation performed by the Procedure `qfs.qft#qfs.swt.instrument.setup` now operates on-the-fly, building a new plugin from an existing plugin and the QF-Test specific add-ons. Thus it is now possible to instrument custom SWT plugins or plugins for unofficial Eclipse versions for which QF-Test doesn't provide an exactly matching instrumented plugin.

- Event nodes can now be transformed into a Wait for component to appear[(771)] node.

- The keyboard shortcut for the 'Redo' operation was changed from ⟦Ctrl-R⟧ to ⟦Ctrl-Y⟧. Over the last years the latter has become a de facto standard.

- The warning about nested Test case[(518)] execution can now be suppressed via the option Log warning for nested test cases[(457)].

Web
- Where scrolling a web page is required to perform a hard event or an image check, QF-Test now tries to scroll a web page itself via JavaScript calls instead of relying on the browser's internal mechanism which often completely messes up the display.

- Component recognition based on geometry can now be suppressed by setting the coordinate and size attributes of a Component[(823)] to '-'.

- It is now possible to specify just the Class count[(828)] or Class index[(828)] of a Component node and ignore the other.

Bugs fixed:

- Using the replace dialog to replace a regular expression with an empty string caused an exception.

- The function to make Procedure calls or Dependency references relative did not work reliably.

- When moving several Procedures, the automatic correction of Procedure calls could fail in some special cases.

- Procedure recording now works correctly for custom classes with a special Class-Loader.

- When creating reports, warnings can be suppressed entirely by specifying `-report.warnings=false` for batch mode or deactivating the respective option in the dialog for interactive report generation.

SWT
- Recording a click on an editable table cell could cause the cell editor to get activated in the wrong table row.

SWT
- Fetch index[744] nodes were not implemented for SWT.

- Removing or marking unused components, importing and other complex component operations could fail if the test suite contained a Data table[566] node with no columns or empty table cells.

B.10.39   Version 3.1.1 - July 28, 2009

Incompatible changes:

- The changes to the check system described below are fully backwards compatible. However, once a test suite has been saved with QF-Test version 3.1.1 it cannot be run correctly with an older QF-Test version except for QF-Test version 3.0.3 which has explicit bidirectional compatibility support for this feature. Once you introduce QF-Test version 3.1.1 you can safely continue to work with QF-Test 3.0.3, but you should not use older versions like 3.1.0 or 3.0.2.

New features:

- The transition to the new check mechanism is now complete. Custom `Checkers` are now fully documented (section 52.5[1073]) and can be created for all check data types. All Check nodes now have a Check type identifier attribute and the old 'Check editable', 'Check enabled' and 'Check selected' nodes were replaced with Boolean check[713] nodes with a matching Check type identifier[715].

- It is now possible to bookmark test suites, run logs and individual nodes in a test suite so they are directly accessible via the new sub-menu ⟦File→Bookmarks⟧. Please see section 39.1.5[(430)] about how to create and manage bookmarks.

Web
- The new `ClassNameResolver` (see section 52.1.9[(1026)]) - though also useful for SWT and Swing applications - is an important milestone for AJAX web testing. With its help and the new pseudo class hierarchy for DOM nodes (see section 5.4.1[(56)]) a deeply nested hierarchy of DIV nodes can be turned into a concise component hierarchy with expressive class names that vastly improve component recognition.

- Procedures[(586)] can now automatically be parameterized via the sub-menu ⟦Parameterize node⟧ in the popup menu for the Procedure node. This feature is work in progress and will need more fine-tuning so we hope for your feedback.

- When creating testdoc documentation it is now possible to follow Test call[(531)] nodes and include the target Test case[(518)], Test set[(525)] or test suite into the generated documentation by activating the respective option in the dialog for interactive use or via the command line argument `-testdoc-followcalls`[(882)] in batch mode.

- Several more variants for converting and (un)packing nodes were added.

Bugs fixed:

- Several problems with split run logs were fixed. Log entries could get lost in compact logs, when the split run log was not saved as a zip file or when split run logs were disabled but the attribute Name for separate run log was defined.

- `DaemonTestRunListeners` did not work correctly in asynchronous mode.

- When creating a report, screenshots for failed Check image[(729)] nodes were always saved regardless of whether the failed check actually represented an error or was just a part of a conditional construct.

- Use of the new Java regular expressions in QF-Test's replace dialog was not correct so that some replacements based on regexps were not possible.

Web
- When determining uniqueness of 'ID' attributes for use as names, QF-Test was too strict so that in some cases potential names were not assigned.

- Instrumenting an OpenJDK directory now works.

Web
- Waiting for a new document did not handle the window name attribute correctly.

Web
- Waiting for a download to finish always waited for the full timeout.

SWT
- In very rare cases replaying text input could crash an SWT application on Linux.

- Option settings from `rc.setOption()` in the SUT were reset whenever options were changed in the option dialog.

SWT
- Event replay for SWT `MenuItems` with Gtk 2.4 on RedHat Enterprise Linux 4 was not reliable.

SWT
- The `ktable` and `gef` Jython modules for the respective SWT components were not correctly updated for the new `ItemResolver` and `Checker` mechanisms.

SWT
- QF-Test now properly discerns the most modal SWT `Shell` in case multiple modal `Shells` are opened simultaneously.

B.10.40 Version 3.1.0 - May 19, 2009

Incompatible changes:

- QF-Test now uses standard Java regular expressions by default which are much more powerful than the old GNU regexp package. One major difference is that '{' and '}' are special characters for Java regexps and may need to be escaped with a '\'. If necessary, you can switch back to the old GNU regexps via the option Use old-style GNU regexps (before 3.1)[(483)]. See also section 47.3[(909)].

Web
- In contrast to Mozilla, Internet Explorer reports a local navigation to an anchor on the current page as a normal navigation event, so up to now a Wait for document to load[(776)] node was recorded which failed when replaying the same test on Mozilla. QF-Test now treats this situation uniformly by ignoring the extra navigation from IE so no Wait for document to load node is recorded with either browser. Consequently, old Wait for document to load nodes for local navigation will now fail and need to be removed from your tests.

New features:

Web
- QF-Test's embedded browser for web testing now supports Java applets with JDK 1.6.0_10 and up, so QF-Test can now test the interaction of a web application with its embedded Java applet.

- Many of QF-Test's options, specifically those with an impact on test execution, can now be changed at run-time via the new run context method `rc.setOption` described in section 11.2.6[(170)] and section 48.5[(917)].

- When moving Procedure[(586)], Dependency[(548)] or Test case[(518)] nodes, all references to these nodes can now be updated automatically. Similar updates are now performed when changing the name of such nodes or QF-Test IDs of Components[(823)] via the replace dialog. As an additional feature the new special

Variable group ${id:...} (see section 6.5^(108)) and the run context method rc.id (see section 48.5^(917)) can be used to mark arbitrary text as the definition of a QF-Test component ID that will be updated whenever the referenced component changes.

- Procedure calls^(589) from one Procedure to another Procedure in the same Package hierarchy can now use a relative procedure name. Also, Procedures and Packages can now be made local to their test suite by prefixing the name with an '_'. See section 8.5.2^(137) and section 8.5.1^(137) for details.

- The selection dialog for a Procedure, Dependency or Test case node now also shows the comment of the selected node. Also, it is now possible to select multiple nodes in order to insert multiple calls at once.

- A Procedure call to a specific Procedure can now be created by copying the Procedure node and pasting it to a place where insertion of the Procedure is not allowed, which is basically everywhere a Procedure call is allowed except for the Extras^(547) node. Dragging and dropping a Procedure with the same effect is also supported.

Web
- It is now possible to interact directly with the JavaScript engine of the browser and execute arbitrary scripts via the methods DomNode.toJS, DomNode.evalJS and DocumentNode.evalJS, described in section 52.11^(1118) .

- When an error or exception occurs, the run log now contains a stacktrace node, showing all nodes on the execution stack including their variable bindings, similar to the variables view in the debugger window.

- There is a new API for various kinds of resolvers and checkers, including an ItemResolver interface for implementing your own sub-items of custom components and a Checker interface for custom checks that can be captured and replayed just like QF-Test's standard checks. See section 52.4^(1062) and section 52.5^(1073) for further information.

- The new TestRunListener API, described in section 52.7^(1087) , can be used to monitor a test run from a script. Its variant DaemonTestRunListener supports the same remotely via QF-Test's daemon API. It is documented in section 53.2.5^(1153) .

SWT
- Support for Eclipse/SWT now includes the latest Eclipse milestone 3.5M7.

Web
- The run context method rc.callProcedure now also works from an SUT script^(631) in a web application.

- Execution of a Test case or Test set can now be aborted or skipped via the run context methods rc.skipTestCase, rc.skipTestSet, rc.stopTestCase and rc.stopTestSet, e.g. to prevent wasted time and irrelevant error messages in case a dependency cannot be resolved. See section 48.5^(917) for details.

- The name of the currently executing Test case is now available via the special variables ${qftest:testcase.name}, ${qftest:testcase.qname} and ${qftest:testcase.reportname} (see <u>section 6.5</u>[(108)]).

Web
- Via the new methods `setProperty` and `getProperty` in the class `Node` from QF-Test's pseudo DOM API for web applications, arbitrary data can be associated with DOM nodes without affecting the web application.

- The new command line argument <u>–ping</u>[(875)] for `qftest –batch –calldaemon` can be used to check whether a QF-Test daemon is up and running.

Bugs fixed:

- In QF-Test version 3.0.2 hard TAB characters in Jython or Groovy scripts were replaced with blanks when showing the script, immediately causing the respective node to become modified. This change has been taken back.

Mac
- For SUTs running on Mac OS X QF-Test disables use of the screen menu bar to allow proper capture/replay of menu actions.

- With JDK 1.6 some of QF-Test's dialogs were not displayed with the correct size.

- Activating an engine license in the option dialog now works as expected.

- When recording procedures, a new root Package is always created instead of merging the procedures into a possibly existing Package named "procbuilder".

B.10.41 Version 3.0.2 - March 27, 2009

New features:

SWT
- Support was added for Eclipse/SWT versions 3.5M6 and 3.4.2.

- The method `callProcedure` was added to the class `DaemonRunContext` of QF-Test's daemon API. This method and the existing method `runTest` now also support direct passing of parameters to this call. See <u>chapter 53</u>[(1139)] for details.

Bugs fixed:

Web
- <u>Wait for document to load</u>[(776)] nodes for web clients did not work reliably in case of multiple browser windows.

Web
- Checking the enabled/disabled state of components is now also implemented for web clients.

Web • Capture and replay of MOVED and SIZED events is now also supported for browser windows of web clients.

• QF-Test now performs regular explicit garbage collections to prevent running out of memory in the PermGen Space, a special Java memory region that causes trouble for a lot of Java applications. For the SUT these extra garbage collections can be turned off via the option Automatically perform garbage collection in the SUT[462].

SWT • Disposal and recreation of an SWT `Display` is now handled correctly.

SWT • Custom SWT `Widgets` implemented as an anonymous local class were not treated correctly, possibly leading to a `ComponentNotFoundException`.

• For checks with a very long timeout deadlock detection is now adjusted accordingly.

• The demonstration test suite for integration with the Imbus TestBench was updated to the latest TestBench version and now handles split run logs correctly.

B.10.42   Version 3.0.1 - January 23, 2009

New features:

• The Jython and Groovy run context `rc` now provides the convenience methods `getBool`, `getInt`, `getNum` and `getString` for simplifying lookup of typed variable values in scripts.

• It is now possible to specify proxy settings for a web SUT directly in the quickstart wizard or via the procedure `qfs.web.browser.settings.setProxy` in the standard library test suite `qfs.qft`.

• An entry for Swing `JInternalFrame` components has been added to the template file for automatic procedure creation.

Bugs fixed:

• Web-testing with Mozilla on Linux no longer requires using a 1.4 JDK.

• Using the "Update component" feature for web pages with deeply nested tables could lead to duplicate component IDs.

• Left-to-right binding in Data table[566] and other data nodes has been fixed.

- SQL statements in Database⁽⁵⁶⁹⁾ nodes did not work with Oracle databases, due to an automatically appended ';' character. This has been fixed. If your database engine requires that SQL statements are terminated with a ';' you have to specify it yourself.

- The procedure `qfs.swt.instrument.setup` now always picks the SWT plugin with an exact version match for instrumentation, if available. This is important for SWT 3.4, where versions 3.4.0 and 3.4.1 are mutually incompatible.

- When recording procedures, a '$' character displayed in one of the fields of the SUT could lead to an exception.

- Groovy class loading and script parsing in the SUT should now be faster, in some cases - specifically applets - drastically.

- A caught exception in the Finally⁽⁶²³⁾ node of a Try⁽⁶¹⁶⁾ node could mess up the state of an exception caught earlier in a Catch⁽⁶¹⁹⁾ node.

- The demo test suite `swt_addressbook.qft` was still based on SWT 3.1, support for which is no longer distributed with QF-Test. The suite is now based on SWT 3.4.

B.10.43 New features in QF-Test version 3 - December 5, 2008

The following new features have been implemented for QF-Test version 3:

Description	Further info
Support for Web testing	Pseudo DOM API[1118] and many other places
Interoperation with HP/Mercury Quality Center and Imbus TestBench	Interaction with Test Management Tools[312]
Groovy as a second scripting language	Scripting[161], Scripting with Groovy[181]
New Jython version 2.2.1	Scripting[161]
Improved Jython and Groovy script editor with syntax-highlighting	Fundamentals of the Jython integration[172]
Automated creation of basic procedures	Automated Creation of Basic Procedures[307]
Extended and unified functionality for 'Check...' and 'Wait...' nodes	Checks[707], Wait for component to appear[771], Wait for client to connect[670] etc.
Direct support for negated checks	Checks[707]
Split run logs and IO redirection - no more OutOfMemoryErrors	Split run logs[122], Log SUT output individually[511], Maximum size of client terminal (kB)[461]
Extensions and an external driver for the daemon API	Test execution[280], Daemon mode[1139]
New concept: "Expected failure" for Test case nodes	Test case[518]
Directing and documenting manual tests	Executing Manual Tests in QF-Test[387]
Extended undo support	Self explanatory
Immediate vs. lazy variable bindings	Immediate and lazy binding[114]
Image API for creating and saving screenshots	Image API[940], Image API extensions[1096]
Crypting of text input for password fields	Salt for crypting passwords[458], Text[691]
Recursive sorting and transformation of nodes	Self explanatory
Revised search with highlighting of results	Self explanatory
Extensions to the standard library test suite `qfs.qft`	pkgdoc
New tutorial chapters for advanced features	Advanced features

Table B.10: New features in QF-Test version 3

Appendix C

Keyboard shortcuts

C.1   Navigation and editing

This table gives a useful overview of QF-Test's basic and advanced navigation and editing keyboard shortcuts:

Windows/Linux	macOS	Function
File Navigation		
Control-N	⌘-N	New
Control-O	⌘-O	Open
Control-S	⌘-S	Save
-	⇧⌘-S	Save as
Basic Editing		
Control-Z	⌘-Z	Undo last change
Control-Y	⇧⌘-Z	Redo last change
Search and Replace		
Control-F	⌘-F	Search
F3	F3	Continue previous search
Control-G	⌘-G	Search again
Control-H	^-H	Replace
Workbench View		
Control-PageDown	^-⇟	Next test suite
Control-PageUp	^-⇞	Previous test suite
Control-Shift-PageDown	Control-Shift-⇟	Change current and next suite
Control-Shift-PageUp	Control-Shift-⇞	Change current and previous suite
Alt-1, 2, ... 9	⌘-1, 2, ... 9	Switch to 1st, 2nd, ... 9th testsuite
F5	⌘-R	Refresh project directory
Shift-F5	⇧⌘-R	Rescan project directory

F6	F6	Switch focus back and forth between suite and project view
Shift-F6	⇧-F6	Select the current suite in the project tree, showing the project if necessary
Control-F6	^-F6	Switch to previously active suite. Keep Control pressed and type F6 again to move further back. Simultaneously pressing Shift reverses the direction
Control-L	^-L	Open latest run log
-	^⌘-F	Toggle Full Screen
-	⌘-,	Open Options
-	⌘-W	Close Testsuite
Alt-F4	⌘-Q	Close QF-Test
Tree View		
Up / Down / Right / Left	↑ / ↓ / → / ←	Basic navigation
Alt-Up	⌥-↑	Jump to previous sibling of node
Alt-Down	⌥-↓	Jump to next sibling of node
Alt-Right	⌥-→	Expand node recursively
Alt-Left	⌥-←	Collapse node recursively
Shift-Up	⇧-↑	Extend selection upwards
Shift-Down	⇧-↓	Extend selection downwards
Control-Up	^-↑	Move upwards without affecting selection
Control-Down	^-↓	Move downwards without affecting selection
Control-Right	^-→	Scroll tree right
Control-Left	^-←	Scroll tree left
Space	Space	Toggle selection of current node
Control-Backspace	^-⌫	Jump to last visited node
Shift-Control-Backspace	^⇧-⌫	Jump to next selected node
Control-.	^-.	Clean tree
Alt-Return	⌥-↵	Bring up node properties window
Shift-F10 / Windows context menu key	⇧-F10	Bring up node popup menu
F2	F2	Mark name or QF-Test ID of node in order to rename it
Tables		
Shift-Insert	⌥-↓	Insert new row
Shift-Return	⇧-↵	Edit selected row
Shift-Delete	⇧-⌧	Delete selected row
Shift-Control-Up	^⇧-↑	Move selected row up
Shift-Control-Down	^⇧-↓	Move selected row down
F2	F2	Edit selected value
Return	↵	Confirm changes

Escape	↵	Discard changes
Shift/Control-Up/Down	^/⇧-↑/↓	Multi selection
Control-X / C / V	⌘-X / C / V	Cut / Copy / Paste
Shift-Control-Right	^⇧-→	For variables: Forward parameters, i.e. `x ->` `$(x)`
Code Editor		
Control-Space	^-Space	Open available QF-Test variables for scripts or for dedicated nodes open a list of available methods.
Control-P	^-P	Find procedure definition (for lines which call a procedure)
Control-T	^-T	Find test definition (for lines which call a test)
Control-W	^-W	Find component (for lines which refer to a component)
Alt-Up	^⌘-↑	Move line(s) up
Alt-Down	^⌘-↓	Move line(s) down
Shift-Return	⇧-↵	Insert an empty line after the current line
Multi-line Text Elements		
Control-TAB	^-→|	Move focus to next attribute
Shift-Control-TAB	^⇧-→|	Move focus to previous attribute
Control-Return	⌘-↵	Confirm changes
For Procedure call nodes		
Control-P	^-P	Find procedure definition
For nodes with a QF-Test component ID attribute		
Control-W	^-W	Find component
For Test call nodes		
Control-P	^-P	Find called test definition
For Dependency reference nodes		
Control-P	^-P	Find dependency definition
Run log		
Control-I	^-I	Open error list dialog
Control-N	^-N	Find next error
Shift-Control-N	^⇧-N	Find previous error
Control-T	^-T	Find node in test suite
Control-W	^-W	Find next warning
Shift-Control-W	^⇧-W	Find previous warning
Shift-Control-Return	⌘⇧-↵	Show text in external editor
-	⌘-W	Close run log
Advanced Editing		
Control-7	^-7	Inserting a comment node
Shift-Control-7	^⇧-7	Inserting a comment node above the currently selected node

Control-A	^-A	Inserting a procedure call
Control-D	^-D	Add selected node to bookmarks
Shift-Control-D	^⇧-D	Toggle disabled state of selected nodes
Control-I	^-I	Find references of node Available for components, tests, dependencies and procedures
Shift-Control-I	^⇧-I	Pack selected nodes into if-sequence
Shift-Control-P	^⇧-P	Transform selected nodes into procedure Available only valid for sequence and similar nodes
Shift-Control-S	^⇧-S	Pack selected nodes into sequence
Shift-Control-T	^⇧-T	Pack selected nodes into test step
Shift-Control-Y	^⇧-Y	Pack selected nodes into try/catch

Table C.1: Shortcuts for navigation and editing

C.2   Record and replay functions

The following table contains important shortcuts for record and replay functions, which partly work also outside of QF-Test, i.e. directly in the SUT:

Windows/Linux	macOS	Function
Replay		
`Return`	`⏎`	Play (execute current node)
`F9`	`F9`	Pause
`Alt-F12` (configurable)	`⌥-F12` (configurable)	Pause test run ("Don't Panic" key)
Record		
`F11` (configurable)	`F11` (configurable)	Toggle "Record mode"
Record Checks		
`F12` (configurable)	`F12` (configurable)	Toggle "Check mode"
`Left Mouse Button`	`Primary Click`	Record default check
`Right Mouse Button`	`Secondary Click`	Show list with available checks
Record Components		
`Shift-F11` (configurable)	`⇧-F11` (configurable)	Toggle "Record single component mode"
`Control-V`	`⌘-V`	Paste recorded QF-Test component ID from clipboard
`Control-Backspace`	`^-⌫`	Jump to last recorded component
`Control-F11` (configurable)	`^-F11` (configurable)	Toggle "Record multiple components mode"
Record Procedures		
`Shift-F12` (configurable)	`⇧-F12` (configurable)	Toggle "Record procedures mode"
`Control-F12` (configurable)	`^-F12` (configurable)	Toggle "Record multiple procedures mode"
Debugger		
`F7`	`F7`	Step in
`F8`	`F8`	Step over
`Control-F7`	`^-F7`	Step out
`Shift-F9`	`⇧-F9`	Skip over
`Control-F9`	`^-F9`	Skip out
`Control-F8`	`⇧⌘-B`	Breakpoint on/off
`Control-J`	`^-J`	Jump to run log
`Control-,`	`^-,`	Continue execution from currently selected node

Table C.2: Shortcuts for special record and replay functions

C.3   Keyboard helper

The following graphic may help to easily remember the assignment of functional keys used by QF-Test. It is intended to be printed, cut out and fixed above the area of functional keys F5 to F12 of your keyboard.

QF-TEST	Reload Prj Rescan Prj	Suite ↔ Prj Suite in Prj Last Suite	Step in Step out	Step over Breakpoint		Pause Skip over Skip out	Menu Popup M	Record Rec Comp Multi Re Co	Rec Check Rec Proc Don't panic Multi Re Pr	shift alt ctrl
	F5	F6	F7	F8		F9	F10	F11	F12	

Figure C.1: Keyboard helper

Appendix D

Glossary

API

Application Programming Interface, a set of package, class and method definitions that the programmer of an application can use. The Java API refers to the interface of the standardized Java class library that is shipped with each JDK.

AWT

Abstract Windowing Toolkit, the part of the Java library responsible for the display of windows and components as well as for the dispatch of events.

GUI

Graphical User Interface. An interface between a program and the user, usually consisting of windows built from components for displaying information and receiving input.

RMI

Remote Method Invocation, communication protocol/programming interface in Java for calling a method of a remote object.

SUT

System Under Test, the application being tested with QF-Test.

VM

The Java *Virtual Machine* executes Java programs that have been compiled to Java bytecode. It is responsible for the platform independence and compatibility of Java programs across various kinds of machines and operating systems.

Appendix E

Privacy - Processing of personal and other data

E.1 Server data for version query

Since QF-Test 4.3 it is possible to check for the availability of a new QF-Test version via an encrypted HTTPS request to www.qfs.de, either automatically when starting QF-Test or via explicit user action. This version information is compared to the current QF-Test version and in case a newer version is available a notification is shown with links to to the respective QFS web pages.

For technical reasons the following data among others are stored by our provider (in so called server log files) when the latest available QF-Test version is retrieved from the QFS web server:

- Your internet protocol (IP) address

In addition the following data may be transferred with the query to select the correct update version:

- The QF-Test version currently used

- The operating system currently used

- The QF-Test language version currently used

- Type of the QF-Test license currently used

- Hash code of the QF-Test license currently used

This anonymous data is stored separately from any personal data possibly provided at a different time and does not allow drawing conclusion to a dedicated person. The data may be used for statistical purpose to optimize our web presence and offers.

The legal basis for the temporary storage of data is Art. 6 para. 1 lit. a GDPR.

The temporary storage of the IP address by the system is necessary to enable the version information to be delivered to the user's computer. For this the IP address of the user must remain stored for the duration of the session.

Data will be deleted as soon as it is no longer needed to achieve the purpose for which it was collected. In case of the collection of data for the provision of the version information, this is the case when the respective session has ended. To detect web service failures including security issues data is stored for 42 days and then automatically deleted.

The collection of data for the provision of the version information and the storage of data in log files is absolutely necessary for the operation of the web service. Consequently, there is no possibility of objection on the part of the user. However, the version information query and thus usage of the web service can be deactivated via QF-Test user options (see section 39.1.10⁽⁴³⁸⁾).

E.2 Sending support requests from within QF-Test

4.5+ QF-Test contains the option to send requests directly to the QF-Test support team. When the action is selected in the "Help" menu of the application, an encrypted web form from www.qfs.de is opened in the default web browser of the user. All information entered into the web form will be sent as email to support@qfs.de.

For technical reasons the following data among others are stored by our provider (in so called server log files) when the latest available QF-Test version is retrieved from the QFS web server:

- Your internet protocol (IP) address

- Browser type and version

- The web page you are visiting

- Date and time of your access

In addition the following data may be transferred with the query to improve the support quality and to speed up the support process:

- The QF-Test version currently used

- The operating system currently used

- The QF-Test language version currently used

- The name and company of the QF-Test license currently used

- Hash code of the QF-Test license currently used

The legal basis for the temporary storage of data is Art. 6 para. 1 lit. a GDPR.

The temporary storage of the IP address by the system is necessary to allow the transfer of support request using the browser web form. For this the IP address of the user must remain stored for the duration of the session.

Data will be deleted as soon as it is no longer needed to achieve the purpose for which it was collected. In case of the collection of data for the provision of technical support, this is the case when the business relation with the customer has ended. To detect web service failures including security issues data is stored additionally on the web server for 42 days and then automatically deleted.

The collection of data for the provision of technical support and the storage of data in log files is absolutely necessary for the operation of the web service. Consequently, there is no possibility of objection on the part of the user. However, it is possible for the user to directly sending an email to support@qfs.de instead of selecting the convenience action from the "Help" menu.

E.3   Context Information for Onine Manual

7.1+   QF-Test opens since 7.1, if the local version of the manual or tutorial is not available, an online version of the document in the browser. To provide context dependent help in QF-Test, the file `doc/context/de/context.properties` resp. `doc/context/en/context.properties` is required. If (and only if) this file does not exists, an online version of the file is requested from the QFS server. For technical reasons the following data among others are then stored by our provider (in so called server log files):

- Your internet protocol (IP) address

More information has already been described in the section "Server data for version query".

E.4   Request Data on WebDriver Download

4.5.2+   Since QF-Test 4.5.2 the webdriver file for certain browsers (e.g. Chrome) can be

downloaded and extracted automatically by QF-Test. To do so, QF-Test downloads the required files from the servers `www.qfs.de`, `github.com`, `chromedriver.storage.googleapis.com`, `developer.microsoft.com` and their content delivery peers using an encrypted connection.

For technical reasons the following data is submitted along the requests:

- Your internet protocol (IP) address

- The name of the file to download, which includes the version number of the file to download.

Please see the privacy policy of the aforementioned service providers for details on their data processing. To deactivate the automatic webdriver download, you can specify the webdriver to use explicitly in the "Start Web Engine" node by pointing the `qftest.web.webdriver.driver` Java property to the existing driver file.

E.5 Client data in QF-Test log files

The log files described below may be provided the Quality First Software GmbH support team via email or file upload in order to facilitate support services. There is no automatic transfer of such logs by QF-Test to QFS or any third party.

QF-Test run logs are a critical tool for analyzing automated test runs in order to locate errors in the test implementation of the system under test. Besides details about the system environment and executed test steps a run log may also contain textual output from the system under test, screenshots of connected monitors or virtual desktops, or the HTML code of the page tested in a browser.

Type and amount of data to be stored in run logs can be adjusted via the QF-Test run log options (see section 39.9[(500)]). Among those the number of automatically stored run logs can be specified - older run logs are automatically deleted.

The default directory where run logs are stored depends on the operating system. The current value can be retrieved via the menu ‖Help→Info→System info‖ shown as `dir.runlog`. The directory can also be explicitly defined via the command line argument `-runlogdir <directory>`[(879)].

In addition to run logs QF-Test also creates internal logs for the purpose of diagnosis of QF-Test itself or the connection to the system under test.

Internal logging of QF-Test is done in five rotating text files named `qftestN.log` with `N = [,1,2,3,4]`. Connection log data to the SUT are stored in the text file `qfconnect.log`.

The default directory for internal logs is depending on the operating system. The current value can be retrieved via the menu $\boxed{\text{Help→Info→System info}}$ shown as `dir.log`. The directory can also be explicitly defined via the command line argument `-logdir <directory>`[874]. An invalid directory parameter deactivates internal logging.

Here we'd like to emphasize that QF-Test is originally meant as tool for software test automation. There we always presume that our customers perform software tests within designated test environments with special test data because use of real personal data is not permitted according to data protection regulations, following the principles of dedication to purpose, minimization of data as well as anonymization.

Appendix F

Third party software

QF-Test makes use of the following software:

Apache Batik SVG Toolkit
Copyright © 2016 The Apache Software Foundation
Distributed under the Apache License, Version 2.0 (see
`doc/licenses/apache-2.0`).
URL: `https://xmlgraphics.apache.org/batik`

Apache Commons IO
Copyright © 2009 The Apache Software Foundation
Distributed under the Apache License, Version 2.0 (see
`doc/licenses/apache-2.0`).
URL: `http://commons.apache.org/proper/commons-io`

Apache PDFBox
Copyright © 2009-2017 The Apache Software Foundation
Distributed under the Apache License, Version 2.0 (see
`doc/licenses/apache-2.0`).
URL: `https://pdfbox.apache.org/`

Apache POI
Copyright © 2009 The Apache Software Foundation
Distributed under the Apache License, Version 2.0 (see
`doc/licenses/apache-2.0` and `doc/licenses/poi-notice`).
URL: `http://poi.apache.org`

API Guardian
Copyright © API Guardian Team
Distributed under the Apache License, Version 2.0 (see
`doc/licenses/apache-2.0`).
URL: `https://github.com/apiguardian-team/apiguardian`

ASM
Copyright © 2000-2011 INRIA, France Telecom
Distributed under individual license (see `doc/licenses/asm`).
URL: `http://asm.ow2.org`

Bean Scripting Framework (BSF)
Copyright © 1999 International Business Machines Corporation.
See the license `doc/licenses/bsf`
URL: `http://www.alphaworks.ibm.com/tech/bsf`

BasE91
Copyright © 2000-2006 Joachim Henke, 2011 Benedikt Waldvogel
See the license `doc/licenses/base91`
URL: `https://github.com/bwaldvogel/base91`

Bouncy Castle
Copyright © 2000 - 2017 The Legion of the Bouncy Castle Inc.
Part of Apache PDFBox.
Distributed under the the Bouncy Castle License (MIT analog) (see `doc/licenses/bouncy-castle.txt`).
URL: `https://www.bouncycastle.org/java.html`

Boxicons
Distributed under the Crative Commons Attribution 4.0 International Public License (see `doc/licenses/CC4.txt`).
URL: `https://boxicons.com/`

Closure Compiler
Distributed under the Apache License, Version 2.0 (see `doc/licenses/apache-2.0`).
URL: `https://github.com/google/closure-compiler`

cdp4j
Copyright © 2023 cdp4j Contributors
Distributed under the the MIT License (MIT) (see `doc/licenses/license.cdp4j`).
URL: `https://github.com/cdp4j/cdp4j`

Chromedriver
Distributed under the Modified BSD License (see `doc/licenses/chromium-bsd.txt`).
URL: `https://sites.google.com/a/chromium.org/chromedriver`

Chromium Embedded Framework
Distributed under the Modified BSD License (see

`doc/licenses/chromium-bsd.txt`).
URL: `https://bitbucket.org/chromiumembedded/cef`

ClasspathSuite
Distributed under the Apache License, Version 2.0 (see `doc/licenses/apache-2.0`).
URL: `https://github.com/takari/takari-cpsuite`

CommonJS Modules Support for Nashorn
Distributed under the the MIT License (MIT) (see `doc/licenses/nashorn-commonjs-modules`).
URL: `https://github.com/coveo/nashorn-commonjs-modules`

Cryptix library
Copyright © 1995, 1996, 1997, 1998, 1999, 2000 The Cryptix Foundation Limited. All rights reserved.
See the license `doc/licenses/cryptix`
URL: `http://www.cryptix.org`

CSS Parser
Distributed under the Apache License, Version 2.0 (see `doc/licenses/apache-2.0`).
URL: `https://sourceforge.net/projects/cssparser/`

dd-plist
Distributed under the the MIT License (MIT) (see `doc/licenses/dd-plist.txt`).
URL: `https://github.com/3breadt/dd-plist`

dom4j
Copyright © 2001-2005 MetaStuff, Ltd. All Rights Reserved
See the license `doc/licenses/dom4j`
URL: `dom4j.sourceforge.net`

Geckodriver
Distributed under the Mozilla Public License 2 (see `doc/licenses/MPL-2.html`).
URL: `https://github.com/mozilla/geckodriver`

Ghostdriver
Distributed under the Modified BSD License (see `doc/licenses/ghostdriver-bsd.txt`).
URL: `https://github.com/detro/ghostdriver`

GNU regex
Copyright © 1998-2001 Wes Biggs.

Distributed under the GNU Lesser General Public License (see `doc/licenses/LGPL`).
Source code included in `misc/gnu_src.jar`
Formerly: http://nlp.stanford.edu/nlp/javadoc/gnu-regexp-docs/index.html
Mirror URL: `https://www.qfs.de/fileadmin/Webdata/pub/gnu-regexp-docs/`

Groovy scripting language
Distributed under the Apache License, Version 2.0 (see `doc/licenses/apache-2.0`).
URL: `http://groovy-lang.org/`

InfluxDB 2.0 Java Client Library
Copyright © 2018 Influxdata, Inc.
Distributed under the the MIT License (MIT) (see `doc/licenses/influxdb-client-java.txt`).
URL: `https://github.com/influxdata/influxdb-client-java`

ini4j
Distributed under the Apache License, Version 2.0 (see `doc/licenses/apache-2.0`).
URL: `http://ini4j.sourceforge.net/`

Jackson
Copyright © 2020 FasterXML.
Distributed under the Apache License, Version 2.0 (see `doc/licenses/apache-2.0`)
URL: `https://github.com/FasterXML/jackson`

JCommon
Copyright © 2007-2012 Object Refinery Limited and Contributors.
Distributed under the GNU Lesser General Public License (see `doc/licenses/LGPL`).
URL: `http://www.jfree.org/jcommon/index.html`

jEdit Syntax Highlighting
Public domain
The jEdit 2.2.1 syntax highlighting package contains code that is Copyright © 1998-1999 Slava Pestov, Artur Biesiadowski, Clancy Malcolm, Jonathan Revusky, Juha Lindfors and Mike Dillon.
URL: `http://syntax.jedit.org`

JExcel
Copyright © 2002 Andrew Khan. All rights reserved.
Distributed under the GNU Lesser General Public License (see

`doc/licenses/LGPL`).
Source code included in `misc/jexcelapi.tar.gz`
URL: `http://www.andykhan.com/jexcelapi/index.html`

JFreeChart
Copyright © 2005-2012 Object Refinery Limited and Contributors.
Distributed under the GNU Lesser General Public License (see `doc/licenses/LGPL`).
URL: `http://www.jfree.org/jfreechart/index.html`

JGoodies Looks
Copyright © 2001-2005 JGoodies Karsten Lentzsch. All rights reserved.
See the license `doc/licenses/jgoodies-looks`
URL: `http://www.jgoodies.com`

JIDE Common Layer
Copyright © 2002 - 2009 JIDE Software, Inc, all rights reserved.
Distributed under the GNU General Public License (see `doc/licenses/GPL`) with Classpath Exception (see `doc/licenses/Classpath`).
URL: `http://www.jidesoft.com/products/oss.htm`

JNIWrapper
Copyright © 2000-2004 MIIK Ltd. All rights reserved.
URL: `http://www.jniwrapper.com`

JSch
Copyright © 2002-2012 Atsuhiko Yamanaka, JCraft,Inc. All Rights Reserved.
See the license `doc/licenses/jsch`
URL: `http://www.jcraft.com/jsch/`

JUnit
Distributed under the Eclipse Public License 1.0 (see `doc/licenses/EPL-1.0`).
URL: `https://github.com/junit-team/junit4`

Jython scripting language
Copyright © 2007 Python Software Foundation; All Rights Reserved.
See the license `doc/licenses/jython`
URL: `http://www.jython.org`

JZlib
Copyright © 2000-2011 ymnk, JCraft,Inc. All rights reserved.
See the license `doc/licenses/jzlib`
URL: `http://www.jcraft.com/jzlib/`

Hamcrest

Distributed under the BSD License (see `doc/licenses/hamcrest-bsd.txt`).
URL: `https://github.com/hamcrest/JavaHamcrest/`

Mozilla

Copyright © 1998-2006 Contributors to the Mozilla codebase.
Distributed under the Mozilla Public License (see `doc/licenses/MPL-1.1.html`).
URL: `http://www.mozilla.org/projects/embedding/`

Nashorn scripting language

Copyright © 2010, 2013, Oracle and/or its affiliates.
Distributed under the GNU General Public License version 2 with Classpath Exception (see `doc/licenses/gplv2ce`).
URL: `https://github.com/openjdk/nashorn`

Netty

Copyright © 2009 Red Hat, Inc.
Distributed under the Apache License, Version 2.0 (see `doc/licenses/apache-2.0`).
URL: `http://netty.io`

Open Telemetry

Includes okhttp and kotlin-stdlib, all distributed under the Apache License, Version 2.0 (see `doc/licenses/apache-2.0`).
URL: `https://github.com/open-telemetry/opentelemetry-java`

Operadriver

Use of the software in binary form for testing purposes is permitted.
URL: `https://github.com/operasoftware/operachromiumdriver`

PngEncoder

Copyright (c) 1999-2003 J. David Eisenberg. All rights reserved.
Distributed under the GNU Lesser General Public License (see `doc/licenses/LGPL`).
Source code included in `misc/PngEncoder.zip`
URL: `http://catcode.com/pngencoder/`

Python library as distributed with Jython

Copyright © 2001-2008 Python Software Foundation; All rights reserved.
See the license `doc/licenses/python`
URL: `http://www.python.org`

Selenium and Drivers

Distributed under the Apache License, Version 2.0 (see

`doc/licenses/apache-2.0`)
URL: `http://www.seleniumhq.org/projects/webdriver`
Indirectly used libraries are partially distributed under different licenses, these are directly referenced in the `selenium.jar`.

SendSignal.exe
Permission to use granted by the author Louis K. Thomas.
URL:
`http://www.latenighthacking.com/projects/2003/sendSignal/`

Sixlegs PNG library
Copyright © 1998, 1999, 2001 Chris Nokleberg
Distributed under the GNU Lesser General Public License (see `doc/licenses/LGPL`).
URL: `http://www.sixlegs.com`

Source Code Pro font
Copyright © 2010-2019 Adobe
Distributed under the SIL Open Font License, Version 1.1. (see `doc/licenses/SIL-open-font-license.md`).
URL: `https://github.com/adobe-fonts/source-code-pro`

STAX - Streaming API for XML
Copyright © 2003-2007 The Apache Software Foundation
Distributed under the Apache License, Version 2.0 (see `doc/licenses/apache-2.0`).
URL: `http://geronimo.apache.org`

TrueZIP Virtual File System API
Copyright © 2005-2007 Schlichtherle IT Services.
Distributed under the Apache License, Version 2.0 (see `doc/licenses/apache-2.0`).
Formerly: http://truezip.dev.java.net
New version:
`https://christian-schlichtherle.bitbucket.io/truezip`

UI-Automation
Distributed under the Apache License, Version 2.0 (see `doc/licenses/apache-2.0`).
URL: `https://github.com/mmarquee/ui-automation/`

Undertow and dependent libraries
Copyright © 2017 Red Hat, Inc.
Distributed under the Apache License, Version 2.0 (see

`doc/licenses/apache-2.0`)
URL: `http://undertow.io/`

webp-imageio

Copyright © 2018 Luciad, © 2010, Google Inc.
Distributed under the Apache License, Version 2.0 (see `doc/licenses/apache-2.0`) and the WebP license (see `doc/licenses/webp`)
URL: `https://bitbucket.org/luciad/webp-imageio`

wxActiveX - Library for hosting ActiveX controls within wxwidgets programs

Copyright © 2003 Lindsay Mathieson.
Distributed under the wxActiveX Library Licence (see `doc/licenses/wxActiveX`).
URL: `http://sourceforge.net/projects/wxactivex`

wxMozilla - Embedding Mozilla in wxWindows

Copyright © 2003-2006 Jeremiah Cornelius McCarthy.
Distributed under the wxWindows Library Licence (see `doc/licenses/wxWidgets`).
URL: `http://wxmozilla.sourceforge.net/`

wxWidgets GUI API

Copyright © 1998-2005 Julian Smart, Robert Roebling et al.
Distributed under the wxWindows Library Licence (see `doc/licenses/wxWidgets`).
URL: `http://www.wxwidgets.org/`

XML Beans

Distributed under the Apache License, Version 2.0 (see `doc/licenses/apache-2.0` and `doc/licenses/xmlbeans-notice`).
URL: `http://xmlbeans.apache.org`

Zulu OpenJDK 8

Distributed under the GNU General Public License version 2 with Classpath Exception (see `doc/licenses/gplv2ce`).
URL: `https://www.azul.com/downloads/zulu-community/`