# **QF-Test - The Tutorial**

Version 7.1.3

Quality First Software GmbH<sup>1</sup> Copyright © 2002-2024 Quality First Software GmbH

April 23, 2024

<sup>1</sup>https://www.qfs.de

# Preface

# **QF-Test**

QF-Test is a professional tool for automated testing of Java, Web and native Windows applications with a graphical user interface (UI).

QF-Test tests the system as a whole through the UI. It can also be used for integration tests checking the overall workflow and the interaction of single systems. The main use cases for QF-Test are automated regression tests. You can use it as well for load testing and input of mass data via the graphical user interface.

QF-Test is designed for the use of testers and developers alike. It has an intuitive UI. Tests can easily be build with the recording function. On the other hand tests can be set up and structured like any other software. If you need to test some functionality for which QF-Test provides no standard test elements you can almost always implement it via a script.

During test execution QF-Test writes a run log intended for post-mortem error analysis. Additionally you can create configurable HTML reports (also XML or JUnit format) presenting the test results in an overview and with graphics.

QF-Test can be run platform independently on Windows, Linux, Unix and macOS. It supports the Java technologies Swing, JavaFX and SWT and the most common web browsers, some even headless. From QF-Test version 5 on you can also test native Windows applications. For a detailed list of the supported platforms, Java and browser technologies please refer to the System requirements section of the user manual.

#### The video

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-42.html

# **Tutorial**

This tutorial is meant as a hands-on introduction to QF-Test.

In the base part we will show the main functions of QF-Test and guide you through the necessary steps to set up your own test suite. You will learn how to analyze your test results, step through your test by use of the debugger and generate an overview report. Further topics are the concept of modularization by help of procedures and component recognition, which is of central meaning in UI Testing.

With <u>part V<sup>(294)</sup></u> some more advanced QF-Test features come on stage, like data driven testing, ensuring of test case prerequisites and automatic generation of basic procedures. Those apply for Java, web and native Windows testing.

The way you write tests for Java, Web or native Windows applications is the same for all three technologies. Only with Web applications you might have to have a look at component recognition before you start with the tests, and with native Windows applications you need to be a bit more patient when recording or replaying the tests. Now you might wonder why we provided three different base parts. This is because the demo applications we use look slightly different for each technology. You also get different setup sequences and different components. So, in order as not to confuse you with a demo not matching the exact description of the tutorial we decided to provide three base parts, part  $I^{(2)}$  for Java, part  $II^{(103)}$  for Web and part  $III^{(203)}$  for native Windows applications.

This tutorial is also available as HTML online version at https://www.qfs.de/en/qftest/tutorial.html.

As an alternative to private study, QFS offers training courses for QF-Test. Details can be found at https://www.qfs.de/en/qftest/training.html.

The following notations are used throughout the tutorial:

- Menu-Submenu represents a menu or menu item.
- (Modifier-Key) stands for a keystroke, where the modifier is one (or a combination) of [Shift]/①, [Control]/①, [Alt]/①, or [第].
- 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, <u>references<sup>(ii)</sup></u> in the PDF version are underlined and show the target page number in small braces.

# Feedback

Note This tutorial has been rewritten from scratch for QF-Test 4.2. We hope you'll like it and look forward to your feedback - be it positive or negative.

Please send all comments, bug reports, wishes etc. to qftest@qfs.de.

# Contents

I	Java	a UI testing with QF-Test	1
1	Worl	king with a Sample Test suite (Java)	3
	1.1	Loading the Test suite	3
	1.2	Starting the Application	5
	1.3	First Test case	7
	1.4	Second Test case - with Error	9
	1.5	The Run log for Error Diagnosis	10
	1.6	Getting Help	13
	1.7	Stopping the Application	14
	1.8	A full Test Run	15
	1.9	Report Generation	15
2	Crea	ting your own test suite (Java)	19
	2.1	Starting the Application	19
	2.2	Recording Actions	26
	2.3	Recording Checks	28
	2.4	Setting up a test suite	29
	2.4 2.5	Setting up a test suite	
			31
3	2.5 2.6	Stopping the Demo	31
3	2.5 2.6	Stopping the Demo	31 32 <b>34</b>
3	2.5 2.6 Writi	Stopping the Demo          Running the whole test suite          ng a Procedure (Java)	31 32 <b>34</b> 34

4	Com	onents (Java)	42
	4.1	Addressing subitems of tables, lists and trees	. 42
	4.2	Windows and components Section	. 44
	4.3	SmartIDs - Addressing components directly	. 50
5	Usin	the Debugger (Java)	53
	5.1	Setting a Breakpoint	. 54
	5.2	Stepping Through a Test or Sequence	. 55
	5.3	Skipping Execution of Nodes	. 57
	5.4	Error or Exception triggering Debugging Mode	. 59
	5.5	Resolving Errors directly from the Run log	. 60
	5.6	Pause Execution	. 62
6	Varia	oles and Procedure Parameters (Java)	64
	6.1	Procedure using a variable	. 64
	6.2	The Variable Bindings table	. 68
	6.3	Advanced debugging of variable bindings	. 72
	6.4	Setting Variables	. 75
	6.5	Variable binding levels	. 77
7	The	tandard Library (Java)	80
	7.1	Inspecting the Standard Library	. 81
	7.2	Selected Packages and Procedures	. 82
		7.2.1 The Checkbox Packages	. 82
		7.2.2 The Combobox/Combo Packages	. 83
		7.2.3 The General Packages	. 83
		7.2.4 The List Packages	. 83
		7.2.5 The Menu Packages	. 83
		7.2.6 The Table Packages	. 84
		7.2.7 The Tree Packages	. 84
		7.2.8 The Cleanup Packages	. 84
		7.2.9 The Run log Package	. 85
		7.2.10 The Run log.Screenshots Package	. 85

V

		7.2.11 The Shellutils Package	. 86
		7.2.12 The Utils Package	. 86
		7.2.13 The Database Package	. 87
		7.2.14 The Check Package	. 87
		7.2.15 The Databinder Package	. 87
8	Cont	rol structures (Java)	89
	8.1	If - else	. 89
	8.2	Loops	92
9	lt's ti	me to start your own Application (Java)	101
II	We	b UI testing with QF-Test	102
10	Work	ing with a Sample Test suite (Web)	104
	10.1	Loading the Test suite	. 104
	10.2	Starting the Browser	106
	10.3	First Test case	108
	10.4	Second Test case - with Error	110
	10.5	The Run log for Error Diagnosis	. 111
	10.6	Getting Help	. 114
	10.7	Stopping the Application	115
	10.8	A full Test Run	116
	10.9	Report Generation	116
11	Crea	ting your own test suite (Web)	120
	11.1	Creating the Setup Sequence	120
	11.2	Recording Actions	. 127
	11.3	Recording Checks	129
	11.4	Setting up a test suite	130
	11.5	Stopping the Demo	132
	11.6	Running the whole test suite	133

12	Writi	ng a Procedure (Web)	134
	12.1	Identifying reusable parts	134
	12.2	Manual creation of procedures	135
	12.3	Transforming nodes into procedures	140
13	Com	ponents (Web)	142
	13.1	Addressing subitems of tables, lists and trees	142
	13.2	Web Component Recognition	144
	13.3	Windows and components Section	146
	13.4	SmartIDs - Addressing components directly	152
14	Using	g the Debugger (Web)	155
	14.1	Setting a Breakpoint	156
	14.2	Stepping Through a Test or Sequence	157
	14.3	Skipping Execution of Nodes	159
	14.4	Error or Exception triggering Debugging Mode	161
	14.5	Resolving Errors directly from the Run log	162
	14.6	Pause Execution	164
15	Varia	bles and Procedure Parameters (Web)	166
	15.1	Procedure using a variable	166
	15.2	The Variable Bindings table	170
	15.3	Advanced debugging of variable bindings	174
	15.4	Setting Variables	177
	15.5	Variable binding levels	179
16	The S	Standard Library (Web)	182
	16.1	Inspecting the Standard Library	183
	16.2	Selected Packages and Procedures	184
		16.2.1 The Checkbox Package	184
		16.2.2 The Select Package	184
		16.2.3 The General Package	185
		16.2.4 The Table Package	185

		16.2.5 The Run log Package	. 185
		16.2.6 The Run log.Screenshots Package	. 186
		16.2.7 The Shellutils Package	. 186
		16.2.8 The Utils Package	. 186
		16.2.9 The Database Package	. 187
		16.2.10 The Check Package	. 187
		16.2.11 The Databinder Package	. 188
17	Cont	rol structures (Web)	189
	17.1	If - else	. 189
	17.2	Loops	. 192
18	lt's ti	me to start your own Application (Web)	201
	Na	tive Windows UI testing with QF-Test	202
19	Work	ing with a Sample Test suite (Win)	204
	19.1	Loading the Test suite	004
			. 204
		Starting the Application	
	19.2		. 206
	19.2 19.3	Starting the Application	. 206 . 208
	19.2 19.3	Starting the Application	. 206 . 208 . 210
	19.2 19.3 19.4 19.5	Starting the ApplicationFirst Test caseSecond Test case - with Error	. 206 . 208 . 210 . 211
	19.2 19.3 19.4 19.5 19.6	Starting the Application	. 206 . 208 . 210 . 211 . 214
	19.2 19.3 19.4 19.5 19.6 19.7	Starting the Application	206 208 210 211 211 214 215
	19.2 19.3 19.4 19.5 19.6 19.7 19.8	Starting the Application	206 208 210 211 214 214 215 216
20	19.2 19.3 19.4 19.5 19.6 19.7 19.8 19.9	Starting the Application	206 208 210 211 214 214 215 216
20	19.2 19.3 19.4 19.5 19.6 19.7 19.8 19.9	Starting the Application	<ul> <li>206</li> <li>208</li> <li>210</li> <li>211</li> <li>214</li> <li>215</li> <li>216</li> <li>216</li> <li>220</li> </ul>
20	19.2 19.3 19.4 19.5 19.6 19.7 19.8 19.9 <b>Crea</b> 20.1	Starting the Application	<ul> <li>206</li> <li>208</li> <li>210</li> <li>211</li> <li>214</li> <li>215</li> <li>216</li> <li>216</li> <li>220</li> <li>220</li> </ul>
20	19.2 19.3 19.4 19.5 19.6 19.7 19.8 19.9 <b>Crea</b> 20.1 20.2	Starting the Application	<ul> <li>206</li> <li>208</li> <li>210</li> <li>211</li> <li>214</li> <li>215</li> <li>216</li> <li>216</li> <li>220</li> <li>220</li> <li>226</li> </ul>
20	<ul> <li>19.2</li> <li>19.3</li> <li>19.4</li> <li>19.5</li> <li>19.6</li> <li>19.7</li> <li>19.8</li> <li>19.9</li> <li>Creat</li> <li>20.1</li> <li>20.2</li> <li>20.3</li> </ul>	Starting the Application   First Test case   Second Test case - with Error   The Run log for Error Diagnosis   Getting Help   Stopping the Application   A full Test Run   Report Generation   Starting the Application   Starting the Application   Recording Actions	<ul> <li>206</li> <li>208</li> <li>210</li> <li>211</li> <li>214</li> <li>215</li> <li>216</li> <li>216</li> <li>216</li> <li>220</li> <li>220</li> <li>226</li> <li>228</li> </ul>

	20.6	Running the whole test suite	232
21	Writi	ng a Procedure (Win)	234
	21.1	Identifying reusable parts	234
	21.2	Manual creation of procedures	235
	21.3	Transforming nodes into procedures	240
22	Com	ponents (Win)	242
	22.1	Addressing subitems of tables, lists and trees	242
23	Using	g the Debugger (Win)	245
	23.1	Setting a Breakpoint	246
	23.2	Stepping Through a Test or Sequence	247
	23.3	Skipping Execution of Nodes	249
	23.4	Error or Exception triggering Debugging Mode	251
	23.5	Resolving Errors directly from the Run log	252
	23.6	Pause Execution	254
24	Varia	bles and Procedure Parameters (Win)	256
	24.1	Procedure using a variable	256
	24.2	The Variable Bindings table	260
	24.3	Advanced debugging of variable bindings	264
	24.4	Setting Variables	267
	24.5	Variable binding levels	269
25	The S	Standard Library (Win)	272
	25.1	Inspecting the Standard Library	273
	25.2	Selected Packages and Procedures	274
		25.2.1 The Run log Package	274
		25.2.2 The Run log.Screenshots Package	274
		25.2.3 The Shellutils Package	275
		25.2.4 The Utils Package	275
		25.2.5 The Database Package	276

		25.2.6The Check Package	
26	26.1	rol structures (Win) If - else	
27	lt's ti	me to start your own Application (Win)	290
IV	An	droid Apps testing with QF-Test	291
V	Adv	vanced features of QF-Test	293
28	Data	driven Testing: Running one Test case with different test data	295
	28.1	Situation	295
	28.2	Traditional way of implementing data driven testing	296
	28.3	Data driver concept	297
	28.4	Summary	303
29	-	endencies: Automatically ensuring correct prerequisites for each Te	
	case		304
	29.1	General	
		Ensuring prerequisites	
	29.4	Error and exception handling	
		29.4.1 Error handling	
		29.4.2 Exception handling	
	00 F	29.4.3 Summary	
	29.5	More about dependencies	. 320
30	Auto	mated creation of basic procedures	322
	30.1	General	322
	30.2	Automated creation of procedures	325

X

30.3	Configu	uration of the automated creation
	30.3.1	Introduction
	30.3.2	First example
	30.3.3	Using the current text for checking
	30.3.4	Creating container procedures
	30.3.5	Using the current value of child components
	30.3.6	More configuration capabilities

# **List of Figures**

1.1	The Test suite FirstJavaTests.qft	4
1.2	The "Test set: Simple Tests" Node	5
1.3	The Setup Node	5
1.4	The Sequence to start the SUT	6
1.5	The CarConfigurator Demo	7
1.6	The "First" Test case	7
1.7	Details of the first Test case	8
1.8	The result view in the status line	8
1.9	The Second Test case	9
1.10	Details of the second Test case	9
1.11	Error in the second test case	10
1.12	Runlog for the second test case	11
1.13	Error in the second test case	12
1.14	Screenshot node showing the error situation	13
1.15	The Cleanup Sequence	14
1.16	Runlog for the Completed Test set	15
1.17	Report Generation Properties	16
1.18	An HTML Report	17
2.1	The Quickstart Wizard	20
2.2	Type of Application	21
2.3	Type of Executable	22
2.4	Executable file selection.	23
2.5	Final Information	24

2.6	Generated Setup Sequence	25
2.7	The CarConfigurator Demo Window	25
2.8	Recording actions on the CarConfigurator Demo	26
2.9	The Recorded Sequence	27
2.10	The Renamed Sequence	27
2.11	The recorded check node	28
2.12	Start organizing the test suite	30
2.13	The Organization of your test suite	31
2.14	The simple cleanup sequence	32
2.15	The Run log of the test suite	32
3.1	Two identical test steps	35
3.2	Create a procedure node	36
3.3	Fill in the procedure content	37
3.4	Insert a procedure call	38
3.5	Select a procedure	39
3.6	Test-suite with procedure	40
4.1	Addressing a table cell	42
4.2	Locating a component	46
4.3	Component tree	47
4.4	Details of the Component node	48
5.1	Set break point	54
5.2	Start test run	54
5.3	Remove break point	55
5.4	Stepping into a node	56
5.5	Stepping over a node	56
5.6	Stepping out of a node	56
5.7	Pause execution at first node of the second test case	57
5.8	Skip over a node	58
5.9	Skip out of a node	58
5.10	Set debugger options to pause on error	59

5.11 5.12 5.13 5.14	Test run paused by error	
6.1	Two almost identical test steps	65
6.2	Procedure with hard coded value	66
6.3	The Details of the 'Procedure' node	67
6.4	'Check text' node	68
6.5	Second procedure calls 'checkFinalPrice'	69
6.6	Variable bindings	70
6.7	Popup menu for 'Additional node operations'	71
6.8	Variable bindings stack showing incorrect value	72
6.9	Continue Execution from here	74
6.10	Details of the Set variable node	76
6.11	Procedure returning a value	77
7.1	The Standard Library	81
7.1 8.1	The Standard Library       Setup Sequence with if-else structures	81 89
	-	
8.1	Setup Sequence with if-else structures	89
8.1 8.2	Setup Sequence with if-else structures	89 90 91
8.1 8.2 8.3	Setup Sequence with if-else structures	89 90 91 93
8.1 8.2 8.3 8.4	Setup Sequence with if-else structures	89 90 91 93
8.1 8.2 8.3 8.4 8.5	Setup Sequence with if-else structures	89 90 91 93 94
8.1 8.2 8.3 8.4 8.5 8.6	Setup Sequence with if-else structures	89 90 91 93 94 96 98
8.1 8.2 8.3 8.4 8.5 8.6 8.7	Setup Sequence with if-else structures	89 90 91 93 94 96 98 99
8.1 8.2 8.3 8.4 8.5 8.6 8.7 8.8	Setup Sequence with if-else structures	89 90 91 93 94 96 98 99
8.1 8.2 8.3 8.4 8.5 8.6 8.7 8.8 10.1	Setup Sequence with if-else structures	89 90 91 93 94 96 98 99 105
8.1 8.2 8.3 8.4 8.5 8.6 8.7 8.8 10.1 10.2 10.3	Setup Sequence with if-else structures	89 90 91 93 94 96 98 99 105 106

10.6 The "First" Test case
10.7 Details of the first Test case
10.8 The result view in the status line
10.9 The Second Test case
10.10 Details of the second Test case
10.11 Error in the second test case
10.12 Run log for the second test case
10.13 Error in the second test case
10.14 Screenshot node showing the error situation
10.15 The Cleanup Sequence
10.16 Run log for the Completed Test set
10.17 Report Generation Properties
10.18 An HTML Report
11.1 The Quickstart Wizard
11.2 Type of Application
11.3 Specification of the URL
11.4 Final Information
11.5 Generated Setup Sequence
11.6 The "CarConfigurator Web" demo page in the browser
11.7 Recording actions on the "CarConfigurator Web" demo
11.8 The Recorded Sequence
11.9 The Renamed Sequence
11.10 The recorded check node
11.11 Start organizing the test suite
11.12 The Organization of your test suite
11.13 The simple cleanup sequence
11.14 The Run log of the test suite
12.1 Two identical test steps
12.2 Create a procedure node
12.3 Fill in the procedure content

	Insert a procedure call138Select a procedure139Test-suite with procedure140
13.1	Addressing a table cell
13.2	Web resolvers as installed in the FirstWebTests.qft
13.3	Locating a component
13.4	Component tree
13.5	Details of the Component node
14.1	Set break point
14.2	Start test run
14.3	Remove break point
14.4	Stepping into a node
14.5	Stepping over a node
14.6	Stepping out of a node
14.7	Pause execution at first node of the second test case
14.8	Skip over a node
14.9	Skip out of a node
14.10	Set debugger options to pause on error
14.11	Test run stopped by error
14.12	Error Dialog
14.13	Update check node with current data
14.14	Corrected check node
15.1	Two almost identical test steps
15.2	Procedure with hard coded value
15.3	The Details of the 'Procedure' node
15.4	'Check text' node
15.5	Second procedure calls 'checkFinalPrice'
15.6	Variable bindings
15.7	Popup menu for 'Additional node operations'
15.8	Variable bindings stack showing incorrect value

15.9	Continue Execution from here	176
15.10	Details of the Set variable node	178
15.11	Procedure returning a value	179
16.1	The Standard Library	183
17.1	Setup Sequence with if-else structures	189
17.2	Wait for client to connect writes the result into the variable "isSUTRunning"	190
17.3	If node evaluates the variable	191
17.4	Transform a node into another one	193
17.5	Pack a node into another one	194
17.6	Details of a Loop node	196
17.7	The new test case	198
17.8	Details of the Check items node	199
19.1	The Test suite FirstWinTests.qft	205
-	The "Test set: Simple Tests" Node	
19.3	The Setup Node	
19.4	The Sequence to start the SUT	
19.5	The Windows CarConfigurator Demo	
19.6	The "First" Test case	
19.7	Details of the first Test case	
19.8	The result view in the status line	
	The Second Test case	
	Details of the second Test case	
	Error in the second test case	
	Run Log for the second test case	
	Screenshot node showing the error situation	
	The Cleanup Sequence	
	Run Log for the Completed Test set	
	'Report Generation Properties	
19.18	An HTML Report	218

### xvii

20.1	The Quickstart Wizard
20.2	Type of Application
20.3	Windows executable file selection
20.4	Final Information
20.5	Generated Setup Sequence
20.6	The Windows CarConfigurator Demo
20.7	Recording actions on the CarConfiguratorForms Demo
20.8	The Recorded Sequence
20.9	The Renamed Sequence
20.10	The recorded check node
20.11	Start organizing the test suite
20.12	The Organization of your test suite
20.13	The simple cleanup sequence
20.14	The Run log of the test suite
21.1	Two identical test steps
21.1	Create a procedure node
21.2	Fill in the procedure content
21.3	Insert a procedure call
21.4	Select a procedure
21.5	Test-suite with procedure
21.0	
22.1	Addressing a table cell
23.1	Set break point
23.2	Start test run         246
-	Remove break point
	Stepping into a node
23.5	Stepping over a node
	Stepping out of a node
23.7	Pause execution at first node of the second test case
23.7	Skip over a node
	•
23.9	Skip out of a node

~ ~ ~ ~		<b>0-</b> 1
23.10	Set debugger options to pause on error	251
23.11	Test run stopped by error	252
23.12	Perror Dialog	252
23.13	Update check node with current data	253
23.14	Corrected check node	254
24.1	Two almost identical test steps	257
24.2	Procedure with hard coded value	258
24.3	The Details of the 'Procedure' node	259
24.4	'Check text' node	260
24.5	Second procedure calls 'checkFinalPrice'	261
24.6	Variable bindings	262
24.7	Popup menu for 'Additional node operations'	263
24.8	Variable bindings stack showing incorrect value	264
24.9	Continue Execution from here	266
24.10	Details of the Set variable node	268
24.11	Procedure returning a value	269
25.1	The Standard Library	273
26.1	Setup Sequence with if-else structures	278
26.2	Wait for client to connect writes the result into the variable "isSUTRunning"	279
26.3	If node evaluates the variable	280
26.4	Transform a node into another one	282
26.5	Pack a node into another one	283
26.6	Details of a Loop node	285
26.7	The new test case	287
26.8	Details of the Check items node	288
28.1	Traditional way of data driven testing	296
28.2	Traditional way with a nested Test set	297
28.3		
	Data table dialog	298

28.5	Test set with Data driver
28.6	Using the \$ (discount) parameter
28.7	Full data table
28.8	Name for run log and report attribute
28.9	Run log with different names per Test case
29.1	First Test set of dependencies_work.qft
29.2	First Test set of dependencies_work.qft
29.3	Sample test suite with the first Dependency
29.4	The run log of the execution
29.5	Procedure startStop.startApplication
29.6	The test suite with a Dependency reference
29.7	Ensuring prerequisites for Test case 'Discount 15'
29.8	'vehicles dialog opened' Dependency
29.9	Implementation of specified test cases
29.10	Run log of nested Dependencies
29.11	Test-suite of Error handler
29.12	Dependency with Error handler
29.13	Run log for Dependency with Error handler
29.14	Try-catch nodes in each Test case
29.15	Test-suite with Catch
29.16	Run log of a Dependency with Catch
30.1	Screenshot of test suite
30.2	The test suite automated_procedures_work.qft
30.3	The recorded procedures
30.4	The test suite containing the procedures
30.5	The procedures for all panels
30.6	The current configuration
30.7	The own configuration file
30.8	The checkText procedure
30.9	The checkText procedure with parameters

30.10 Using the <compid> place holder</compid>
30.11 Your first automatically created procedures
30.12 The configuration for the current text
30.13 The created procedures with the current text
30.14 The template for container procedures
30.15 Usage of @FORCHILDREN tag
30.16 The created container procedures
30.17 Configuration of <ccurrentvalue></ccurrentvalue>
30.18 Test-suite using <ccurrentvalue></ccurrentvalue>
30.19 Parameters for container procedures
30.20 Parameters for container procedures in test suite

xxi

# Part I

# Java UI testing with QF-Test

This first part of the tutorial is meant to help you learn the basic features and workflows of QF-Test. It focuses on the test of Java applications and its specifics.

For testing Web applications please go to part II<sup>(103)</sup> or part III<sup>(203)</sup> for native Windows programs, as those parts use the same scenarios but with different systems under test.

Within part  $V^{^{(294)}}$  more advanced QF-Test features are explained, applicable for all supported UI technologies.

# Chapter 1

# Working with a Sample Test suite (Java)

In this first chapter, we will have a look at a simple test suite, explain its major elements, execute it and evaluate the result.

Video This chapter is also available as a video tutorial at



"Working with a Sample Test suite" https://www.qfs.de/en/yt/tutorial-1.html

# 1.1 Loading the Test suite

Note On first startup of QF-Test and/or the System Under Test (SUT) via QF-Test you might get a security warning from the firewall asking whether to block the Java network communication 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.

After starting up QF-Test, you can immediately bring up our first example test suite.

Action • Press the 🗁 toolbar button to bring up the file open dialog

- Navigate to the subdirectory **qftest-7.1.3/doc/tutorial** of your QF-Test installation
- There select the file FirstJavaTests.qft

QF-Test will then load the indicated test suite which should look as follows:

FirstJavaTests.qft] QF-Test	- 0 3	×
<u>File Edit View Insert Operation</u>	ons <u>R</u> ecord R <u>u</u> n De <u>b</u> ugger <u>C</u> lients E <u>x</u> tras <u>H</u> elp	
	> 🥕 🜑 🖩 🗸 🗆 🗈 () 🕨 🖿 🗮 🎫 🖘 🖸 どうどうの	•
FirstJavaTests.qft ×		
Test suite FirstJavaTests.qft	Test suite	
Q Test suite	Name	^
> D Test set Simple Tests		
<ul> <li>Procedures</li> <li>Other Extras</li> </ul>	+ / × + Include files	
> C Windows and components	File	
	qfs.qft	
	+ / 🔺 🕂 Dependencies (reverse includes)	
	File	
	+ / × + Variable definitions	
	Name Value	
	Execution timeout (ms)	~
	OK Cancel	
	Terminal	
	ierminal	^
		~

Figure 1.1: The Test suite FirstJavaTests.qft

The **left part** of the main window contains the test suite, organized in a tree structure. The **right side** shows the details of a selected tree node.

At **bottom right** you'll see the terminal displaying messages sent by QF-Test and the application you are testing.

In the tree structure of the main window you can navigate and select individual nodes of the test suite.

### • Double click the node Test set: Simple Tests to expand it.

You'll find the test set contains two test case nodes enclosed by a "Setup"/"Cleanup" pair.

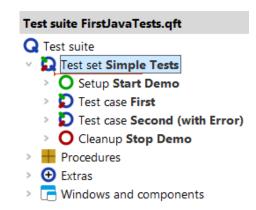


Figure 1.2: The "Test set: Simple Tests" Node

In the following sections we'll describe the purpose and function of the individual nodes.

# 1.2 Starting the Application

Our first step is to examine the "Setup" node:

#### • Expand the Setup: Start Demo node now.



Figure 1.3: The Setup Node

In the "Setup" node you'll see two child nodes:

- 1. Set variable set the variable 'client' to the connection name for the SUT, which will be needed for every action replayed to the application.
- 2. Sequence: Start client if necessary dependent on OS starts the System Under Test (SUT) for the respective operating system in case it is not already running.

Let's also have a brief look inside the **Sequence: Start client if necessary and dependent on OS**:

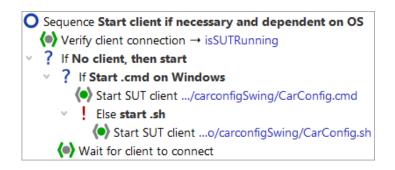


Figure 1.4: The Sequence to start the SUT

QF-Test allows you to test applications on Microsoft Windows, macOS as well as on Unix systems. The test suite described in the tutorial can be replayed on any of them. The only node where we need to differentiate between the operating systems is the startup of the application. This is done by means of an **If - Else** structure running a .bat file on Windows, respectively a .sh file otherwise (on macOS or Linux).

The **Start SUT client** node starts the application (SUT) and sets up a link between *qftest* and the SUT. In order to be independent of the actual installation we use a relative path, starting from the QF-Test version directory, contained in the QF-Test variable f(qftest:dir.version) (see manual chapter Variables).

Please also note that the application is only started in case the client has not yet been connected.

At this point, we're ready to actually start the SUT:

Action

- Click on the **O** Setup: Start Demo" so it is selected and still expanded (the child nodes stay visible).
  - Click the Start test run toolbar button. This button causes the selected node to be executed.

During execution QF-Test marks the active step by use of an arrow pointer ->.

When the setup sequence is completed, our demo application "CarConfigurator" is going to appear on the screen. As QF-Test gets back the focus after the replay action, thus being raised to the foreground, the demo application might be hiding behind it.

#### 1.3. First Test case

👄 CarConfigurator Swing	_	
<u>File</u> <u>Options</u> <u>Purchase</u> order	<u>H</u> elp	
Vehicles Specials Accessories		
Vehicles		
Model	ID	Price
Hydro2 Vovage	M1 M2	\$79,000.00 \$56,500.00
Voyage Hybrid	M3	\$56,500.00
Roadster-E	M4 M5	\$45,900.00 \$29,000.00
		\$23,000.00
Base price		\$0.00
Specials price		\$0.00
Accessories price		\$0.00
Discount		0% -5%
Final price		\$0.00

Figure 1.5: The CarConfigurator Demo

# 1.3 First Test case

Let's check out now what test case "First" contains. There are four test steps inside:



Figure 1.6: The "First" Test case

**Reset** - performs a reset by use of the File->Reset menu action.

Select Model 15 - chooses the last model 15 within the vehicles table.

**Select Special Model Jazz** - switches to the Specials tab and choose the Jazz option. **Check Final Price** - checks that the calculated final price field located at bottom right equals a given value.

#### 1.3. First Test case

Test steps are used to group the nodes and to document what is being done. This will prove very useful when it comes to error analysis or test adaptations.

• Expand the four test step nodes.

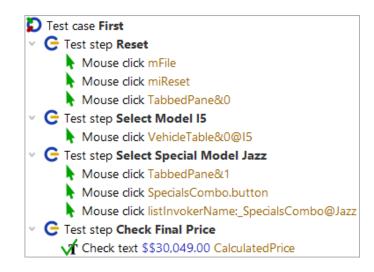


Figure 1.7: Details of the first Test case

You can see Mouse clicks and Checks, which have been grouped in test step nodes for better readability of the test case. The action nodes display the action type (Mouse click, Check, ...) and the component targeted, i.e. where the action goes to. When writing a test you can use the QF-Test recording function to create them. Recording will be explained in the next chapter <u>Creating your own test suite (Java)</u><sup>(19)</sup>.

- Please select the test case "First" node
  - Click the replay button ▶.

Action

The test steps will then be replayed in the SUT, which will happen very quickly.

The test result is indicated during and after the test run in the status line at the bottom of the QF-Test main window and should read now 'Finished: No errors'. Next to it there are counters for the numbers and results of the test cases executed. In our case it was just one, error-free, which means a success rate of 100%.

Finished: No errors	<b>#</b> 1	θ	1 🧕	0 100
---------------------	------------	---	-----	-------

Figure 1.8: The result view in the status line

Each counter icon has a descriptive tool tip. A list of all counters can be found in the chapter 'Capture and replay' of the manual.

#### 1.4. Second Test case - with Error

# 1.4 Second Test case - with Error

The second test case will show us what happens when an error occurs during test execution.

• Expand the test case "Second (with Error)".

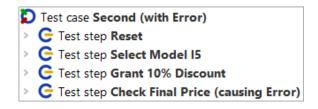


Figure 1.9: The Second Test case

Apart from the third test step it is identical to the first test case. So what does this one do?

Test step: Grant 10% discount - Writes the value 10 into the discount field.

The 'Input' node is another basic action node and can be created directly via the QF-Test recording function.

• Expand the Test step: Grant 10% Discount.



Figure 1.10: Details of the second Test case

Let's execute the second test case:

- Select the node Test case: "Second (with Error)".
  - Click the replay button ▶.

Action

This time a dialog shows up telling us that an error occurred.

#### 1.5. The Run log for Error Diagnosis



Figure 1.11: Error in the second test case

What happened? To find out we'll open the QF-Test run log for error analysis.

An alternative approach for error analysis would be to execute the test case again using the debugger. This will be explained in chapter Using the Debugger  $(Java)^{(53)}$ .

# 1.5 The Run log for Error Diagnosis

QF-Test logs detailed information for every test execution.

Action

- Please open the latest run log by one of the following options:
  - either by pressing the Show run log button of the error dialog

or in case you have already closed the dialog

- by pressing toolbar button ⇒ or
- by pressing Ctrl-L.
- Note The most recent run logs are also listed at the bottom of the Run menu of the main window.

The run log comes up in a separate window displaying the logged actions of the test case you've just executed:

Action

Q				
(240417161724.qzp] Run log - 17/04/2024, 1	6:17 - 0 exceptions, 1 error ar	id 0 warnings	- 0	×
<u>F</u> ile <u>E</u> dit <u>V</u> iew <u>H</u> elp				
		n 🔍 🙆 💻		
Run log	Run log			
Run log (detailed)	Test run ID			
Test case Second (with Error)	240417161724			
	Time and date	Duration	Real time spent	
	16:17:24.241 2024-0		1.617 s	
		1.4205		
	Result			
	Exceptions	Errors	Warnings	
	0	1	0	
	Environment			
	Hostname	Operating system	Executed by	
	w10t2	amd64-Windows 10-		
	WIOLZ	ando4-windows to	qis	
	Java version	QF-Test v	ersion	
	17.0.10+7	7.1.3-dev	(30107)	
	OK			
	OK		Cancel	

Figure 1.12: Runlog for the second test case

The run log reflects the tree structure of the test suite view you are already familiar with. When you click on one of the nodes on the left side, the properties of the event including time stamp and duration will be displayed on the right.

In the tree left you will notice nodes surrounded by a red border. These are indicators showing where a problem occurred in a child node. If you keep expanding the red nodes, you'll eventually come to the actual error node.

Please use an easier way to find the error source by pressing the Find next error
 toolbar button or the [Ctrl-N] key shortcut.

All nodes with red highlighting have been expanded and the actual error node has been selected:

Run log	Log for failed Check text	
Run log (detailed)	Time and date Duration Real time sper	nt 🐴
Test case Second (with Error)	16:17:25.308 0 ms 0 ms	
> C Test step Reset		
> C Test step Select Model I5	Message	
> C Test step Grant 10% Discount	Mismatch	
Test step Check Final Price (causing Error)	Expected: '\$29,000.00'	
Check text \$29,000.00 CalculatedPrice > CarConfig	Got: '\$26,100.00'.	
16:17:24.793 Expansion of text: '\$\$29,000.00' → '\$29,000.00'	Annotation	
\$10:17:24.794 Expansion of client: '\$(client)' → 'CarConfig'		
TextField CalculatedPrice		
Failed: Check text \$26,100.00 CalculatedPrice > CarConfig		
Stack trace		
Creenshot of screen 1	Current values for failed Check text	
Screenshot of window: CarConfigurator Swing [CarConfig]	Client	
16:17:25.854 Logged information from client CarConfig:	\$(client)	
	QF-Test component ID	
	OK Cancel	

Figure 1.13: Error in the second test case

The error message on the right says that the expected value of the final price field differs from the actual one. Of course this error is there by intention as the second test case is supposed to show us how to analyze an error.

Another helper for error analysis is the **Screenshot** of the SUT taken at the time when the error occurred (four nodes down from the red node). Being able to see the state of the SUT at that moment often proves useful for determining the cause of the error. The following image shows a screenshot node:

Run log	Screenshot	
Run log (detailed)	Time and date	^
Test case Second (with Error)	16:17:25.501 2024-04-17	
> G Test step Reset		
> C Test step Select Model I5	🔍 🔍 📙 💽 Screenshot (1680x1050) [1:1]	
> C Test step Grant 10% Discount	GerConfigurator Swing − □ ×	
<ul> <li>C Test step Check Final Price (causing Error)</li> <li>C Check text \$29,000.00 CalculatedPrice &gt; CarConfi</li> <li>16:17:24.793 Expansion of text: '\$\$29,000.00'</li> <li>16:17:24.794 Expansion of client: '\$(client)' →</li> <li>TextField CalculatedPrice</li> <li>Failed: Check text \$26,100.00 CalculatedPrice</li> <li>Stack trace</li> <li>Screenshot of screen 1</li> <li>Screenshot of window: CarConfigurator Swing</li> <li>16:17:25.854 Logged information from client (</li> </ul>	Eile       Options       Purchase order       Help         Vehicles       Specials       Accessories       rd       Rui         Vehicles       Model       ID       Price         Hydro2       M1       \$79,000,00       Voyage         Voyage       M2       \$56,500,00         Voyage Hybrid       M3       \$56,500,00	
	Roadster-E         M4         \$45,900.00           I5         M5         \$29,000.00             >	*

Figure 1.14: Screenshot node showing the error situation

In addition to screenshots of all monitors, QF-Test also saved images of the client windows at the time of the error. This allows you to analyze the contents even if they are covered by other dialogs or windows.

Note The information gathered in a long test run accumulates and can eat up enormous amounts of memory. This is the reason why QF-Test is configured by default to create a compact run log, keeping only the relevant information for report generation and error diagnosis.

This functionality can be configured via the option "Create compact run log" within  $\boxed{\text{Edit} \rightarrow \text{Options} \rightarrow \text{Run logs} \rightarrow \text{Content}}$ . The root node of the run log tells you whether it is a compact or detailed run log. You can also configure the number of screenshots to be saved.

# 1.6 Getting Help

We take a short break in this section to give you a few tips that might prove helpful as you continue with the tutorial.

There are different places where you can look for help or information:

The comprehensive be achieved via most search can navigates you to the search functionality ||Help $\rightarrow$ Online search...|. This homepage throughout available on our and allows querying all

### 1.7. Stopping the Application

**documentation** (manual https://www.qfs.de/en/qf-test-manual.html, tutorial https://www.qfs.de/en/qf-test-tutorial.html, standard library https://www.qfs.de/en/qf-test-support/documentation/standard-library.html,

blog https://www.qfs.de/en/blog.html and our videos https://www.qfs.de/en/get-started-with-qf-test/videos.html). Search results can be filtered as needed.

In case you work **offline** and want to search for a certain topic, the **PDF versions of the manual or tutorial** available via the **Help** menu can be used. The Offline HTML version doesn't have a content search option. However, there is a link to the PDF on every HTML page in the top and bottom lines so switching is easy.

QF-Test also offers a **context sensitive help** for tree node types and their details. To use it, simply press the **right mouse button** on an arbitrary tree node or attribute in the details pane. From the context menu select **What's this?**. This will directly bring you to the reference explanation of this item in the manual.

Beside getting help from the documentation you also have the option to contact our support team. During your evaluation phase or after that as customer with a valid maintenance contract you may issue your questions directly to our support experts (send an email to support@qfs.de or use the support form from the QF-Test help menu Contact the support team or call us +49 8171 - 38648 20). Contact details can be found at https://www.qfs.de/en/qf-test-support/product-support.html.

# **1.7 Stopping the Application**

We haven't inspected the cleanup sequence, so let's have a look at it now:

### • Expand the Cleanup: Stop Demo node.



Figure 1.15: The Cleanup Sequence

The Cleanup sequence stops the client process in a hard way and waits until it fully terminates. This is a very simple approach and shall suffice for the moment.

Action **Execute** the **cleanup** to see the CarConfigurator vanish.

### 1.8 A full Test Run

After we have seen how the single elements of the test set work, let us have a look at the functionality provided by the test set node.

Action

- First, close the "CarConfigurator" in case it is still running.
- Then select the "Test set: Simple Tests" node.
- Execute it with the replay button .

The result dialog will come up after test execution, informing us about the error caused by the second test case.

Action • Open up the run log again by  $\Rightarrow$  to take a closer look:

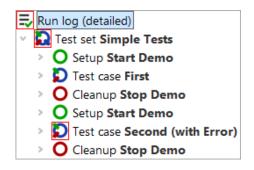


Figure 1.16: Runlog for the Completed Test set

This shows the special behavior of Setup / Cleanup nodes in a test set: They are executed before and after **each test case** to help achieving a proper starting state for each test case.

Note Stopping the SUT after each test case is not the smartest way to ensure a clean state. There are more elegant ways for setup and cleanup that will be explained with the advanced features in this tutorial (chapter 29<sup>(304)</sup>).

### 1.9 Report Generation

In the world of quality assurance documenting the test results is pretty important. To this end, QF-Test offers an automated report generation feature. Since you've just done a complete test run, we're at a good point to show you this feature.

#### 1.9. Report Generation

- Make sure the **run log of the test run** is open.
  - In the **run log window** select **File**→**Create report...** to bring up the dialog for the report parameters.

	×
Save to directory C:\Users\qfs\.qftest\ErsteWinTest	s 🕞
✓ Create HTML report	Create XML report
Create JUnit report	
✓ List test steps	List checks
✓ List exceptions	✓ List errors
List warnings	Show icons for nodes
Pass HTML tags through	Use doctag extensions
Ignore skipped nodes	Ignore not implemented nodes
Embed pie chart	Include testsuite name
Embed thumbnails	
Scale factor for thumbnails (%) 20	
✓ Show report in HTML browser	when finished
<u>O</u> K	<u>C</u> ancel

Figure 1.17: Report Generation Properties

In the first field, you can specify the filename of the report. Following this, you can decide what type of report you want. QF-Test offers three kinds of reports, HTML, XML and JUnit format. An XML report is useful if you want to process the data further, e.g. if you have written your own XSLT stylesheets to shape the report. JUnit reports prove useful when you need to import results into build or test management tools.

Let's generate an HTML report from the results of the last test run.

• So just leave the report **options unchanged**.

• Start generation by pressing the **OK** button.

The report will then be generated and presented to you in a browser window:

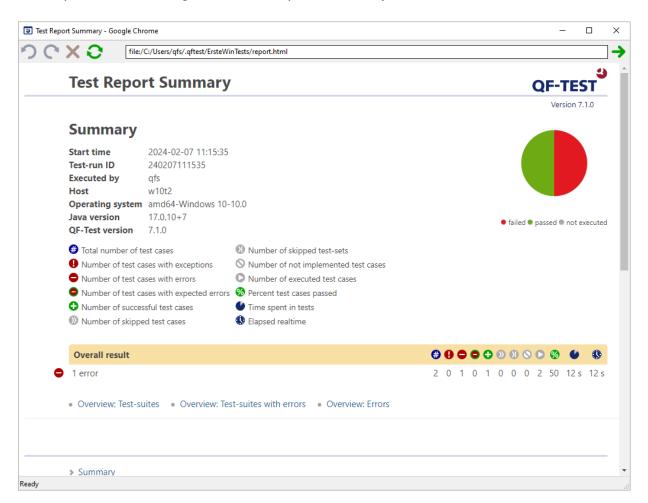


Figure 1.18: An HTML Report

The report begins with a summary containing informational data from your system on the top left side, a legend describing the meaning of icons used in the report on the top right side, an overview pie chart in the middle and the overall test result below. In our case, the result we see are the error-free first test case and the second with the well-known error, leading to a success rate of 50%.

Following the summary there are three overview sections:

- 1. Test suites that have been executed within the test run.
- 2. Test suites in which errors occurred.

3. Errors including their exact position and detailed message.

The report generator is very useful for creating an overview document for presentation and archiving purposes.

# **Chapter 2**

## Creating your own test suite (Java)

In the second chapter of the Java tutorial we will create our own sequences for starting and stopping an SUT from QF-Test. Furthermore we are going to record actions and checks and use those to build up a simple test case.

Video This chapter is also available as a video tutorial at



"Creating your own test suite" https://www.qfs.de/en/yt/tutorial-2.html

## 2.1 Starting the Application

To begin, you need to launch the application from *qftest*. There is a **Quickstart Wizard** to help you in creating the respective setup sequence.

Action

- Open a new test suite via the menu item File New test suite...
  - To open the Quickstart Wizard please use the Extras→Quickstart Wizard...
     menu.

The Wizard starts up with a welcome message and some further information.

Action • After saying a short hello please press the **Next** button to begin.

### 2.1. Starting the Application

Quickstart Wizard	
Pages	Welcome
First steps Welcome Type of application Application specifics Optional Conclusion Summary	<ul> <li>Welcome to the Quickstart Wizard!</li> <li>In general your application needs to be started by QF-Test in a specific way in order to allow recording and replay of user actions as well as verification of results.</li> <li>This applies to both stand-alone Java programs and web applications (defined by a URL).</li> <li>This wizard will assist you in creating a startup sequence for your application.</li> <li>Just press Next to begin.</li> <li>Further information can be found in the <u>Quickstart chapter</u> of the manual.</li> </ul>
	Back Einish Cance

Figure 2.1: The Quickstart Wizard

In step two you can choose the type of application to be tested.

- Please keep the first option **A Java application**.
  - Press Next.

🕹 Quickstart Wizard		$\times$
Pages	Type of application	
First steps Welcome Type of application Application specifics Optional Conclusion Summary	<ul> <li>Please select the type of your application from the following list.</li> <li>A Java application</li> <li>A web application in a browser (specified via URL)</li> <li>A web application in an emulated mobile browser</li> <li>A native Windows application</li> <li>An Android application</li> <li>An Electron application</li> <li>A PDF document</li> </ul>	
	<u>B</u> ack <u>N</u> ext Einish <u>C</u> ar	ncel

Figure 2.2: Type of Application

In step three you can choose the type of application executable to be tested.

- Please keep the first option "A stand-alone executable (launched via .exe, .cmd, .bat, .app,...)"
  - Continue with **Next**.

🕙 Quickstart Wizard		×	
Pages	Type of application executable		
First steps Welcome Type of application Application specifics	<ul> <li>Please select the type of your Java application executable from the following list.</li> <li>A stand-alone executable (launched via .exe, .cmd, .bat, .app, .sh,)</li> </ul>		
Type of application executable	○ A Java archive (java –jar <archive>)</archive>		
Optional Conclusion	○ A Java class (java <class>)</class>		
Summary			
	<u>B</u> ack <u>N</u> ext <u>Finish</u>	cel	

Figure 2.3: Type of Executable

The next step specifies the executable file.

- For this please use the select executable button on the right.
  - Navigate to the subdirectory **qftest-7.1.3/demo/carconfigSwing/** of the QF-Test installation.
  - Select the CarConfig.cmd file (or CarConfig.sh on macOS/Linux).
- Note In the figure below we used the QF-Test variable \${qftest:dir.version} to address the version specific directory of the QF-Test installation, which you have already come across in the previous chapter. (Details on special QF-Test variables can be found in the manual chapter Variables).

### 2.1. Starting the Application

Action

• Please press the **Finish** button, as we do not need to go to the further optional steps for our simple demo.

Pages	Executable or start script
Pages First steps Welcome Type of application Application specifics Type of application executable Executable or start script Optional Working directory SWT instrumentation Client name Conclusion Summary	Executable or start script Please specify the executable file (e.g. a .exe or .app) or start script (e.g. a .cmd or .bat) of your application either directly or via the file explorer on the right. \${qftest:dir.version}\demo\carconfigSwing\CarConfig.cmd Note: If you want to test a Java/SWT or Eclipse-based RCP application, please take care about the 'SWT instrumentation' page.
	Back Next Finish Cancel

Figure 2.4: Executable file selection.

We directly reach the final summary that explains what will happen after closing the wizard and how to continue.

• Please press **Finish** in order to end the wizard.

Quickstart Wizard	×
Pages	Summary
First steps Welcome Type of application Application specifics Type of application executable Executable or start script Optional Working directory SWT instrumentation Client name Conclusion Summary	When pressing the "Finish" button, the 'Setup' sequence for your configured application (SUT) will be created within your test suite. You will find that sequence under the Extras node.         Select this 'Setup' sequence and press the 'Start run' button in the toolbar.         Your application should then be started and you will be ready for recording.         Entered data:         Type of application executable:         Java/App         Executable or start script:       \${qftest:dir.version}\demo\carconfigSwing\CarCo         Working directory:       \${qftest:dir.version}\demo\carconfigSwing         SWT instrumentation:       No SWT application         Client name:       CarConfig
	< >
	Back Next Einish Cancel



The generated setup sequence **Launch CarConfig** appears in the "Extras" section of your test suite and contains three parts:

- Set variable defines the global client variable used throughout the test suite.
- Wait for client to connect checks whether the client is already running.
- Launch SUT if not running starts the System Under Test as client if it is not already running by us of a "Start SUT client" and wait for its start.
- Note The information whether the client is already running is stored into a variable "isSUTRunning" in the first "Wait for client to connect" node and evaluated by the subsequent "If" condition. You can find this in the respective node details. This kind of conditional execution will be explained later in detail.

#### 2.1. Starting the Application



Figure 2.6: Generated Setup Sequence

Now we want to see some action:

### • Please ensure the **Setup: Launch CarConfig** node is **selected**.

• Then **click** or simply hit "Enter" (Return).

You should see the CarConfigurator application appear on your screen soon. As the focus changes back to QF-Test after the execution, the Demo might be covered by the test suite window.

👄 CarConfigurator Swing	_		×
<u>File</u> <u>Options</u> <u>Purchase</u> orde	r <u>H</u> elp		
Vehicles Specials Accessories	]		
Vehicles			
Model	ID	Price	
Hydro2 Voyage	M1 M2	\$79,00 \$56,50	0.00
Voyage Hybrid Roadster-E	M3 M4	\$56,50	
IS	M5	\$45,90 \$29,00	
Base price		\$0.00	
Specials price		\$0.00	
Accessories price		\$0.00	
Discount		0%	-5%
Final price		\$0.00	

Figure 2.7: The CarConfigurator Demo Window

At the end of this section let's save our test suite.

- Action
- Press the <sup>B</sup> toolbar button or use the <u>File→Save</u> menu option with its short-cut <u>Ctrl-S</u>.
  - In the file explorer navigate to an appropriate directory where you have write access e.g. Documents in your user home directory.
  - Provide a name e.g. MyFirstTests.qft.
  - Finish the saving action by pressing on Save.

## 2.2 Recording Actions

You're now ready to record some actions for our demo:

👄 CarConfigurator Swing	_		×
<u>File</u> <u>Options</u> <u>Purchase</u> order	<u>H</u> elp		
Vehicles Specials Accessories			
Vehicles			
Model	ID	Price	
Hydro2 Voyage Voyage Hybrid	M1 M2 M3	\$79,00 \$56,50 \$56,50	0.00
Roadster-E I5	M4 M5	\$45,90 \$29,00	
Base price		\$0.00	
Specials price		\$0.00	
Accessories price		\$0.00	
Discount		0%	-5%
Final price		\$0.00	

Figure 2.8: Recording actions on the CarConfigurator Demo

- Action Please press the Record button.
  - Switch to the SUT application window. From now on every mouse and keyboard action performed within the SUT window will be recorded.
  - Click to the table cell 15 in the last row.

- Change to the **Specials** tab of the tabbed pane.
- Choose the special **Jazz** from the combo box.
- Finally switch back to the Vehicles tab in the tabbed pane.
- Stop the recording by use of the **b**utton.

You'll find the recorded sequence placed in the "Extras" section:

Ð	Extras
>	O Setup Start Demo
>	O Sequence Recorded 04/17/24 04:19 PM

Figure 2.9: The Recorded Sequence

The recorded sequence has a default name with date and time of the recording. You can change this name as you see fit by simply clicking on the node and changing its properties in the details view on the right.

- Action
- Please rename the recorded sequence to "Select Model I5 Jazz".
  - Then **open** the sequence node to see its content. There should be the expected mouse clicks. You should even be able to interpret where they go to.

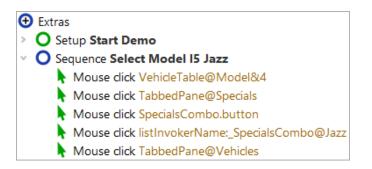


Figure 2.10: The Renamed Sequence

Now, let us replay the recorded sequence.

- Select the Select Model I5 Jazz sequence node.
  - Press the play button

#### 2.3. Recording Checks

You should now see exactly the same sequence of mouse events executed in the SUT as you recorded before.

The sequence is supposed to replay (even multiple times) without errors. You should see "Finished: No error" in the bottom right corner of your test suite window.

### 2.3 Recording Checks

To verify the client's behavior we use check nodes, which query certain states and properties of elements within the SUT. Also checks can be recorded.

Action

Action

- Click the "Record a check" ✓ button.
  - Switch to the SUT window. When moving the mouse over the components you will notice a blue border indicating the current selection.
  - **Right-click** the value field of "**Final price**". In the popup menu you are offered a choice of standard checks for a text field component.
  - Select the first option "Text" for a check on the textual value of the field.
  - Stop the recording by using the **Stop button**.

Again, the newly recorded sequence appears in the "Extras" section.

- Please rename the sequence to "Check final price"
  - Expand it to see the check node.

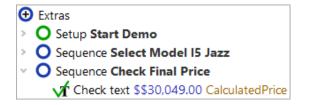


Figure 2.11: The recorded check node

• Feel free to run this sequence, too, verifying it is working properly.

As the next step, we want to create a test case from the two sequences.

## 2.4 Setting up a test suite

The top-level nodes of the test suite define its basic structure:

- Right after the "Test-suite" node an arbitrary number of "Test set" and "Test case" nodes may be added specifying the functional tests.
- "Procedures" contains reusable sequences (procedures) that can be organized into packages.
- "Extras" is the scratch pad for recording and experimentation.
- "Windows and components" contains the all-important elements of the test suite: registered elements of the SUT, e.g. windows, menus and buttons. The details of each element in the "Windows and components" section contains the properties of the recorded UI element required by QF-Test to find the component when replaying a test.

Functional test cases are represented by "Test case" nodes and can be grouped and structured with the help of "Test set" nodes.

"Setup" and "Cleanup" nodes are intended for test steps ensuring a well-defined state before and after a test case.

- Let's start by renaming the top-level test set node from "unnamed" to "Demo Tests".
  - If a dialog pops up asking us whether to update references we can simply confirm with "Yes".
  - The second step is to move the "Setup" node generated by the Quickstart Wizard from the "Extras" node into the "Test set" node right before the "Test case" node. Moving the "Setup" node can be done via mouse (Drag&Drop), contextmenu (right mouse-button copy/paste) or by Ctrl-X and Ctrl-V keyboard commands.

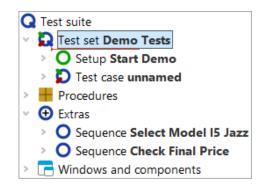


Figure 2.12: Start organizing the test suite

The next step is to make a test case of the two sequences previously recorded.

- Please rename the test case node from "unnamed" to "First".
  - **Open** the test case node by clicking the '+' symbol.
  - Move the two sequences from "Extras" into the test case.

You need to open the test case node because otherwise QF-Test would try to place the sequence nodes after the test case node on the same level, which is not a valid option.

QF-Test always records sequence nodes. They have the same functionality as test step nodes, only they do not show up in the report. So, just to show you, we will transform the two sequence nodes into test step nodes.

- Please open the context menu for the first of the two sequence nodes by a rightclick.
  - Choose Transform node into...→Test step
  - Repeat this for the **second** sequence node.

Your test suite should now look like this:

Action

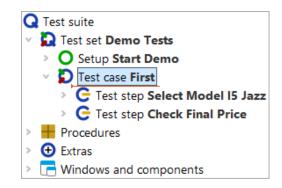


Figure 2.13: The Organization of your test suite

### 2.5 Stopping the Demo

The only thing missing now is a cleanup sequence closing down the SUT.

There are usually various ways to terminate an SUT, e.g. clicking the close button of the application window, pressing ALT-F4 or via the menu action  $\boxed{\text{File} \rightarrow \text{Exit}}$ . All these options can be directly recorded and then used in the cleanup sequence.

Let's use the last one. So please perform the following steps.

Action

### Start recoding

- Record the menu SUT menu action File→Exit.
- You see the demo window close.
- Stop recording
- Rename the recorded sequence to "Stop demo".
- Open the **context menu** for the recorded sequence and select **Convert into...** -> Cleanup.
- Finally move the cleanup node up to be the last node in the test set.
- Note The cleanup node can only be dragged and dropped to the test set if the test set's last child node is collapsed. To expand or collapse a node during a drag and drop operation, hold your cursor over the triangle next to the node.

You should end up with the following:

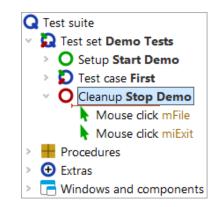


Figure 2.14: The simple cleanup sequence

By this we have finished with the basic steps of structuring our suite.

### 2.6 Running the whole test suite

Finally let's execute our newly created suite:

Action

- Terminate the SUT client in case it is running.
  - Select the root node of the test suite.
  - Run it by pressing "Replay" ► or typing Return).

The SUT is expected to appear, the test case will be executed and finally the SUT will be terminated.

We know the test run details can be looked up in the run log.

Action • It can be opened by clicking the = toolbar button or via the Run $\rightarrow$ 1.... menu option with its short-cut Ctrl-L



Figure 2.15: The Run log of the test suite

In the first tutorial chapter we've already learned how to use the run log for error analysis.

# **Chapter 3**

# Writing a Procedure (Java)

Video This chapter is also available as a video tutorial at



Action

"Writing a Procedure" https://www.qfs.de/en/yt/tutorial-3.html

In chapter one and two you learned how to start an application from QF-Test thus being able to record mouse and keyboard actions, add checks and organize the result in a test case. This approach is fine and sufficient as long as your tests are simple and you have just a few of them. But as soon as the number of tests increases it is important to make use of so called 'procedures'.

Procedures make sequences reusable and therefore avoid duplicated identical parts. This is important for easy and on the long run efficient maintenance of your tests.

Procedures can be organized into packages = . Procedures and packages are the basis for modularizing your tests.

## 3.1 Identifying reusable parts

In this chapter we will work with the test suite <code>FirstJavaTests.qft</code> you already know from chapter one.

- Copy FirstJavaTests.qft from the subdirectory qftest-7.1.3/doc/tutorial
  of the QF-Test installation to a working directory and
  - open FirstJavaTests.qft.
  - If you want to keep the changes you will be making to the demo test suite **save it** in a working directory as described at the end of section 2.1<sup>(19)</sup>.

Please have a look at the test step "Reset" in the two test cases. They are exactly the same.

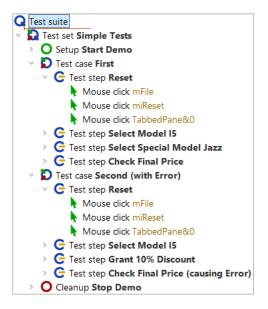


Figure 3.1: Two identical test steps

Following above concepts it would be a good idea to turn them into a procedure.

## 3.2 Manual creation of procedures

There is more than one way to create procedures and insert procedure calls. We will start with the manual one by inserting an (empty) procedure node and moving the respective actions into it. Then we will create the respective procedure call.

It is good to know those basic steps but there is a second more elegant way of creating procedures, which we will explain afterwards.

Okay, let's do it by hand. We will start with creating a procedure node and naming it appropriately.

Open the Procedures node and keep it selected.

- Chose Insert→Procedure nodes→Procedure.
- Name it 'reset'. The other fields can be left empty.
- Press the **OK button** to finalize the creation of the procedure.

### 3.2. Manual creation of procedures

Action

• Open the newly created 'reset' procedure node.

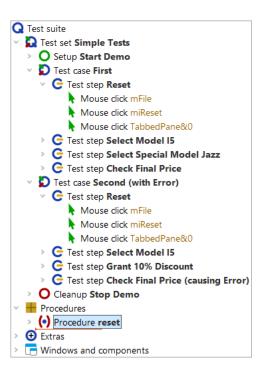


Figure 3.2: Create a procedure node

The second step is to fill the procedure with the respective reusable actions.

- Select the three 'Mouse click' nodes in the test step. To select more than one node at once you can select the first one, then press the Shift key and, while keeping it pressed, click the last node you want to select.
  - Move them down into the procedure e.g. by mouse (drag&drop) or cut/paste from the Edit or context menu.

#### 3.2. Manual creation of procedures

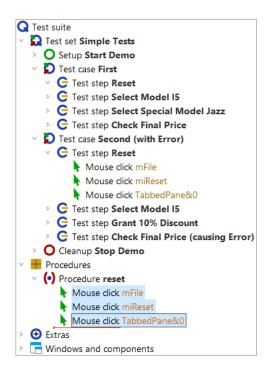


Figure 3.3: Fill in the procedure content

The third step is to add a procedure call to the place of the three 'Mouse clicks' you moved.

- Action
- Select the test step 'Reset', which is still open.
  - Select Insert $\rightarrow$  Procedure nodes $\rightarrow$  Procedure call or use the Ctrl-A shortcut.

### 3.2. Manual creation of procedures

Procedure call Variable for return value Local variable Local variable Variable definitions Name Value QF-Test ID Delay after (ms) OK Cancel			
Variable for return value  Local variable  Variable definitions  Name Value  QF-Test ID  Delay before (ms) Delay after (ms)  Comment	Procedure call		$\times$
Local variable  Variable definitions  Name Value  QF-Test ID  Delay before (ms) Delay after (ms)  Comment	(•) Procedure name		^
Local variable  Variable definitions  Name Value  QF-Test ID  Delay before (ms) Delay after (ms)  Comment			
+     ×     +     Variable definitions       Name     Value       QF-Test ID       Delay before (ms)     Delay after (ms)        Comment	Variable for return value		
+     ×     +     Variable definitions       Name     Value       QF-Test ID       Delay before (ms)     Delay after (ms)        Comment			
Name     Value       QF-Test ID       Delay before (ms)       Delay after (ms)       ✓ Comment	Local variable		
QF-Test ID Delay before (ms) Delay after (ms) Comment	🛨 🧹 🗶 🕂 🕂 Variak	ole definitions	
Delay before (ms) Delay after (ms)	Name Value		
Delay before (ms) Delay after (ms)			
Comment	QF-Test ID		
Comment			
	Delay before (ms)	Delay after (ms)	
OK Cancel	Comment		
OK Cancel			~
	ОК	Cancel	

Figure 3.4: Insert a procedure call

- On the dialog **press** the procedure selection button (•) left of the label 'Procedure name'.
  - **Select 'reset'** from your test suites procedures. Other fields can be left as they are.
  - Press the **OK** button on both dialogs to finalize the creation of the procedure call.

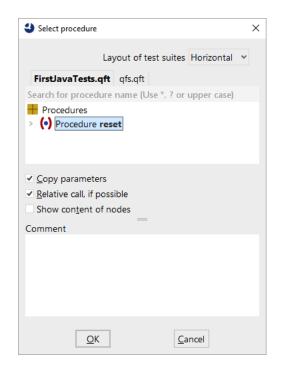


Figure 3.5: Select a procedure

In order get a real benefit from the procedure, of course, we also need to replace the content of reset test step in the second test case by the 'reset' procedure call, too.

You can do it the same way as before or use the following **alternative steps** to create a procedure call:

- **Open** the 'Reset' test step of the second test case.
  - Remove the three 'Mouse click' action nodes therein.
  - Select the 'reset' procedure node.
  - **Move** it via drag&drop into the 'Reset' test step (copy/paste action can be used alternatively). This does not actually move it but create a respective procedure call.

The test suite should look like this:

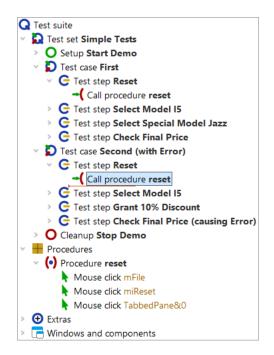


Figure 3.6: Test-suite with procedure

When now executing the test cases the reset will work exactly like it did before. Hence, in the run log you will see the same executed node as before, only preceded by the procedure call.

### 3.3 Transforming nodes into procedures

As already mentioned at the beginning of the last section, QF-Test offers an alternative, much faster way to create a procedure:

Action

- Select the test step or sequence node that contains the reusable steps to be transformed into a procedure.
  - Select menu item **Operations**→**Transform node into**→**Procedure** or use the Ctrl-Shift-P shortcut.

You will find that the test step, respectively sequence node disappeared and there is a procedure call in its place. Moreover, a procedure was created in the Procedures section containing the child nodes of the former test step / sequence node and named the same.

It is good practice with QF-Test to record a sequence, give it a name and immediately turn it into a procedure via <u>Ctrl-Shift-P</u> if you suspect it to be of use somewhere else, too.

# **Chapter 4**

# **Components (Java)**

Let us have a look at the last main section in the test suite window, the Windows and components node. When talking about components we also want to show you how to address subitems of components like tables, trees and lists.

Video This chapter is also available as a video tutorial at



"Components" https://www.qfs.de/en/yt/tutorial-4.html

## 4.1 Addressing subitems of tables, lists and trees

Subcomponents of tables, lists and trees can be addressed by indices. There are two main types: textual and numeric indices. Let's record a mouse click to a table cell and analyze the recorded QF-Test component ID.

Action

- Start the CarConfig application, in case it has not been started already, by executing the Setup node of the test suite.
  - Activate the recording mode by pressing the toolbar button igodot .
  - Click a table cell, e.g. the first model.
  - Stop the recording by pressing

You will see the recorded mouse click in the Extras section.

O Sequence Recorded 17/04/2024, 16:23
Mouse click VehicleTable@Model&0

Figure 4.1: Addressing a table cell

The QF-Test component ID recorded is VehicleTable@Model&0. The single parts are:

- VehicleTable is the QF-Test component ID of the table component itself.
- @ and & separate the single parts from each other and at the same time indicate the type of index that follows: @ for a textual index and & for a numeric index.
- Model is the textual index for the table column with the title 'Model'.
- 0 is the numeric index for the first table row.

#### Note Numeric indices always start at 0.

You can use any index type for column or row. It is just important that the separator and the index type match.

Action • Change the QF-Test component ID so that the third price value in the table will be clicked, using numeric indices only.

The solution is to type VehicleTable&1&2 in the QF-Test component ID attribute of the Mouse event node.

In order to address the model 'I5' using textual indices only, you would have to enter VehicleTable@Model@I5. Using numeric indices you would write VehicleTable&O&4 and for mixed indices VehicleTable&O@I5 or VehicleTable@Model&4.

The third type of index QF-Test supports is a textual index containing a regular expression. Regular expressions can be used to replace a string by an expression that can match for more than only one string. For a detailed explanation of regular expressions please refer to the manual. So you could also address the cell for the model 'I5' using VehicleTable@Model%I.\*.

Lists are addressed the same way as tables are, just with one index only.

A tree has only one index. This is the path down the tree to the node to be specified. The path consists of the respective nodes of the tree, separated by slashes ('/'). Let's record a mouse click to a tree:

- Action Start the CarConfig application, in case it has not been started already, by executing the Setup node of the test suite.
  - Navigate to the tree: In the CarConfig application select the menu item Options -> Specials...], select a model and press 'Details'
  - Activate recording mode by pressing the toolbar button
  - Click a tree node, e.g. 'Description'.

#### 4.2. Windows and components Section

### • Stop the recording by pressing

For the tree node 'Description' the recorded QF-Test component ID would be DetailsTree@/Information/Description. The single parts are:

- DetailsTree is the QF-Test component ID of the tree component itself.
- @ separates the QF-Test component ID of the tree from the index and at the same time indicates the type of index that follows: @ for a textual index.
- /Information/Description is the textual index for the tree path to 'Description'.

If you wanted to address the node using a numeric index you would have to use  $DetailsTree \sqrt[6]{0/1}$ .

### 4.2 Windows and components Section

The topic 'components' is covered by several videos:

Video



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:

The video

The video



'ComponentNotFoundException - simple case'

https://www.qfs.de/en/yt/componentnotfoundexception-simple-40.html

shows a simple case.

A more complex case is discussed in the video



'ComponentNotFoundException - complex case'

https://www.qfs.de/en/yt/componentnotfoundexception-complex-40.html

Live recording of the special webinar

#### 4.2. Windows and components Section



'Component recognition'

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

QF-Test stores the information about how to find a component in the UI of the SUT in the Windows and components section. It analyses the component information when recording actions to the SUT and saves the information for the components the user interacted with in the details of the Component nodes.

Since Java Swing, JavaFX and SWT have a clear concept of how a certain component has to be technically implemented, you rarely have to bother about the information QF-Test stores for a certain component. In most cases QF-Test will recognize the component it has to replay some event to. Only when the UI of your application changed significantly in a way QF-Test cannot compensate, you will have to think about Component nodes.

Note For a detailed instruction about what to do then, please refer to the manual chapter Troubleshooting component recognition problems. There are also links to videos showing respective samples.

This section is meant to give you an idea about which kind of information is stored in the Component nodes and how QF-Test uses it to recognize a component in the UI.

Let's have a look at the details of a 'TextField' component node.

- Start the CarConfig application, in case it has not been started already, by executing the Setup node of the test suite.
  - Open the procedure 'Check final price'.
  - Navigate to the text field Component node by right-clicking the text check node and selecting Locate component or using the Ctrl-W shortcut.

[FirstJavaTests.qft] * QF-Test			- 0 X
	and Burn Dalaura	en Oliente Estere Hele	
Eile       Edit       View       Insert       Operations       Re $\bigcirc$ $ \bigcirc$ $ \bigcirc$ $ \bigcirc$ $ \bigcirc$ $ \bigcirc$ $ \bigcirc$			<b>⇒ ಶರ</b> ೮ರ
*FirstJavaTests.qft ×			
Test suite *FirstJavaTests.qft		Check text	
Test suite Test set Simple Tests Test set Simple Tests Test set Simple Tests		Client \$(client)	
<ul> <li>(•) Procedure checkFinalPrice</li> <li>(•) Procedure reset</li> <li>(•) Procedure reset</li> <li>(•) Extras</li> <li>(•) Windows and components</li> </ul>	C <u>u</u> t <u>C</u> opy <u>P</u> aste Delete Edit	Ctrl-X t ID Ctrl-C Ctrl-V Delete Alt-Enter	
	<u>M</u> arks	>	
	Start run <u>T</u> oggle disabled sta <u>B</u> reakpoint on/off	te Ctrl+Shift-D Ctrl-F8	
	Locate component	Ctrl-W	
	Highlight compone Show i <u>n</u> Inspector Add to boo <u>k</u> marks	nt Ctrl-D	
	Trans <u>f</u> orm node int Pack n <u>o</u> des	• •	
	Additional nod <u>e</u> op	erations >	
	<u>I</u> nsert node	>	
	Clean tree	Ctrl-Period	Cancel
	What's this?		Finished: No erro

Figure 4.2: Locating a component

This is going to take you directly to the 'TextField CalculatedPrice' node in the Windows and components section.

ile <u>E</u> dit <u>V</u> iew Insert <u>O</u> perations <u>R</u> ecord R <u>u</u> n De <u>bug</u> ▶ 🕞 🛱 🎝 🤇 🔶 → 🥕 🛑 🔳	॥✓ □(•) ► ■ ॥ ≕ ७ ℃ ৫	C
*FirstJavaTests.qft ×	Component	
<ul> <li>Test suite</li> <li>Test set Simple Tests</li> <li>Procedures</li> <li>Procedure checkFinalPrice</li> <li>Check text \$(price) CalculatedPrice</li> <li>Procedure reset</li> <li>Extras</li> <li>Windows and components</li> <li>Window JCarConfigurator</li> <li>MenuBar MenuBar</li> <li>TabPanel TabbedPane</li> <li>Popup popupInvokerName: SpecialsCombo</li> <li>Panel CalculatedPrice</li> <li>TextField CalculatedPrice</li> <li>TextField DiscountValue</li> <li>Dialog dialogCarConfiguratorError</li> </ul>	QF-Test ID CalculatedPrice Class name TextField Name CalculatedPrice Feature Label: Final price \$ As regexp +	

Figure 4.3: Component tree

Let's have a look at the properties of this component node used to identify the UI element.

Component							
QF-Test ID							
CalculatedPrice							
Class name TextField							
Name							
CalculatedPrice							
Feature Label: Final price							
\$ As regexp							
+ / × + Extra features							
State				Nai	me	Value	
Ignore	×			qfs:class		javax.swing.JTextField	
Ignore	×			qfs:genericclass		TextField	
Should match	×			qfs:label		Endpreis	
Ignore	×			qfs:system	class	javax.swing.JTextField	
٢							>
Structure							
Class index					Class count		
3					4		
Geometry X					Y		
167					86		
Width 140					Height 17		
140					17		
Comment							

Figure 4.4: Details of the Component node

The topmost attribute is **QF-Test ID**, which provides the 'handle' to the component to be used in the test cases and procedures. All other attributes refer to the component in the UI.

The next attribute is **Class name**. In our case it is 'TextField'. For component recognition it is essential to know the class of a component. In fact this class is a generalized value of the Java class or Java system class. This is helpful to achieve recognition independently of the specific implementation and allows an easy porting of tests e.g. from Java Swing to its successor JavaFX. However, QF-Test saves the specific and the system class in the Extra features table as Extra features named qfs:class and qfs:systemclass. By default, they are not used for component recognition.

Other examples for classes would be 'Panel', 'Dialog' or 'Button'.

#### 4.2. Windows and components Section

The **Name** attribute is the name or id given to the UI component by the programmer. If there is a Name then this, together with the class, is all QF-Test needs to identify a component in the UI.

If the programmer did not set a name or id for the UI component and the Name attribute therefore remained empty QF-Test needs other criteria like a certain text associated with the component, index information and geometry.

A feature associated with a button for example would be the text shown on the button. QF-Test saves a text directly belonging to the UI component in the Feature attribute. Texts in the vicinity of a component that could be the label of the component are saved in the Feature attribute with the prefix Label: and in the Extra features table with the Extra features Name qfs:label.

The **Structure** information refers to all UI components of the respective class. The total number of UI components of that class is saved in the Class count attribute, the index of the component itself in the Class index attribute.

Last there is the **Geometry** information. It is the one having the lowest weight within the recognition algorithm. It can be of value if no other helpful information is available for a component.

In case you are interested in further details of the component recognition you will find such in the Component recognition chapter of the technical reference of the manual.

If you want to get a feeling for the component recognition you could play around with the attribute values and see what you need to change to make QF-Test not recognize it anymore or even recognize a wrong component in the UI. You will find that you need to change quite a few attributes before QF-Test recognizes a different component. This means that component recognition with QF-Test is very robust. With regression tests a significant part of the UI component criteria need to change before QF-Test will not recognize the component anymore even if the component has no name or id.

When you click the Component node QF-Test will highlight the recognized component in the UI by outlining it with a dark blue border.

- Action
- **Delete CalculatedPrice from the Name attribute**. Because as long as there is an entry in the Name attribute QF-Test will not consider the attributes below.
  - Change the Feature attribute from Label: Final price to Label: xxx.
  - Click the 'TextField' node and check QF-Test still highlights the Final Price field in the UI.
  - Change the Feature attribute back to Label: Final price, either via the respective toolbar button or by typing Ctrl-Z.
  - Change the qfs:label value in the Extra features table from Final price to Discount.

#### 4.3. SmartIDs - Addressing components directly

- Click the 'TextField' node and check QF-Test still highlights the Final Price field in the UI.
- Change the qfs:label value in the Extra features table back to Final price.
- Change all Structure and Geometry attributes to a different value and check that QF-Test still highlights the Final Price field in the UI.
- Change the Feature attribute from Label: Final price to Label: Discount.
- Check that now QF-Test highlights the Discount field.

This is just to give you a bit of a feeling for component recognition. In the above mentioned chapters (and some more) of the manual you will find detailed information about what to do when you have trouble with component recognition.

### 4.3 SmartIDs - Addressing components directly

Since version 7.0 QF-Test officially supports SmartIDs allowing you to reference a component without having to record a Component node.

For some applications SmartIDs may reduce the time required for management and maintenance of the component information considerably.

SmartIDs may also have a positive effect on readablity and maintainability of the tests themselves.

It also allows you to write tests without use of the recording functionality, for example when a component or even the whole application does not yet exist, and you want to go ahead with tests ("test first" approach). SmartIDs then allow you to specify the future recognition critiera directly.

The recognition criteria available are the component class, the name or a label and the index. The values are the same as in the recorded Component node. Nested SmartIDs even reflect the component hierarchy.

The SmartID is used in place of the QF-Test ID of the component. SmartIDs start with the hash symbol # as the first character, in the simplest form followed by the name or the label of the component, for example:

- #btnOK, "btnOK" being the name of the component or
- #First name, "First name" being the label of the component.

#### 4.3. SmartIDs - Addressing components directly

The disadvantage of this simple form of the SmartID is performance at replay, as QF-Test has to search all components for the given criteria. For this reason it is better to specify the class of the component, too. Above SmartIDs would then be:

- #Button:btnOK
- #TextField:First name

Currently, the recording of SmartIDs is not activated by default. You can activate it directly via the menu.

Now, please record some SmartIDs by

- Action
- opening the menu **Record** and selecting **Record SmartID**.
- Activate recording mode via the button "Start recording"
- Click to a text field, for example the one for the discount percentage, then
- click to a table cell, for example the first model.
- Stop the recording via "Stop recording"

You will find the recorded mouse clicks in the Extras section.

The SmartID for the input field is *#TextField:name=DiscountValue*, because QF-Test preferably uses the name for the SmartID value when there are a name and a label for the component.

The second mouse click shows the SmartID for the table component, followed by @ and the text index of the column and & with the numeric index of the row, just like you saw it in the one but last section: #Table:name=VehicleTable@Modell&O.

- Action
- When you do not want to record SmartIDs any more you can deactivate it via the "Record" menu.

Of course, you can continue working with SmartIDs. When recording more SmartIDs you will come across more expressions used in SmartIDs. We now want to show you some of them and give you an explanation why they are used.

For labels ending with a colon you will find a back slash in front of the colon:

• #TextField:First name\:

The explanation is the colon defines the end of the name of the class. Therefore, colons in the component name or the label have to be escaped by a back slash. Other characters like @, & and % have to be escaped as well because they are used as separators before indices.

Labels will be prefixed by a qualifier, here left=:

• #TextField:left=First name

Explanation: a SmartID consisting of class and name of the component reaches the same performance at replay as does component recognition via a recorded Component node. Labels, however, are a different case. There are various options for what might be the best label for a component. QF-Test searches all the available labels of a component for the best one. For performance it is good when you specify the type of label expected.

left= indicates the label to the left of the component. Other SmartID qualifiers indicating the position of the label are right=, top=, topleft= or bottom=. When the text of the component serves as label the qualifier is text=, for the tooltip tooltip=.

When you get several components with the same name or label on a display you might see a compound SmartID consisting of to single SmartIDs concatinated by @:

- #TitledPanel:Customer address@#TextField:left=First name
- #TitledPanel:title=Invoice address@#TextField:left=First name

The example shows the SmartIDs for the label "First name", both on the panel "Customer address" and on the panel "Invoice address".

The second sample is a bit more performant than the first one because of the qualifier title= in the SmartID for the panel. With the first one readablity is a bit better. It depends on the application whether you have to pay attention to performance. With web applications with many UI elements it is usually an issue. However, with Java applications you can opt for readablity in most cases.

Long labels can be shortened by the use of regular expressions:

• #%Dialog:Information.\*@#Button:OK

The percent sign directly after the hash sympol indicates the name or the label to be a regular expression. In above sample the regular expression shortens the title. It may also be used to identify an "OK" button in any dialog with the title starting by "Information". For more details about regular expressions please see the manual - regular expressions. You will find more information about components and SmartIDs in manual - components.

# **Chapter 5**

# **Using the Debugger (Java)**

In this chapter we will learn how to run a test suite with QF-Test's built-in and intuitive debugger. If you are familiar with debugging from other IDEs like Eclipse, you will find this debugger similar in function and usefulness.

Video This chapter is also available as a video tutorial at



"Using the Debugger" https://www.qfs.de/en/yt/tutorial-5.html

By the end of this chapter you will be familiar with the following debugger functionality:

- Setting a Breakpoint<sup>(54)</sup> e.g. via Ctrl-F8 (①雅-B) on macOS).
- Pausing a test run at any time and resuming operation using the debugger button
   or the pause test run hotkey (<u>Alt-F12</u>).
- Stepping Through a Test or Sequence<sup>(55)</sup> using the debugger buttons 'Single step'

   <sup>3</sup> , 'Step over' <sup>3</sup> and 'Step out' <sup>3</sup>.
- Skipping Execution of Nodes<sup>(57)</sup> using the debugger buttons 'Skip over' <sup>♥</sup> and 'Skip out' <sup>♥</sup>.
- Error or Exception triggering Debugging Mode<sup>(59)</sup>.
- Resolving Errors directly from the Run log<sup>(60)</sup>.
- Jump directly to the current error in the run log via Ctrl-J. (Jump to run log in chapter section 5.5<sup>(60)</sup>).
- Note Instead of the debugger buttons you can also enter the commands via the QF-Test menu and most by keyboard shortcuts as well. You'll find the shortcut listed beside the menu option, if available. For a complete list, refer to the Keyboard shortcuts section of the

#### 5.1. Setting a Breakpoint

user manual. You can also find a little helper there for attaching to your keyboard which shows the function key assignment of QF-Test.

There are some more functions related to the debugger that we will come to in later chapters:

- Locating the current node using the debugger button 

   (Locate the current node in chapter section 6.3<sup>(72)</sup>).
- "Continue execution from here" via the popup menu of the respective node (figure  $6.9^{(74)}$ ).
- Rethrowing exceptions using the toolbar button  $\oint$  .
- The variable bindings table (section  $6.3^{(72)}$ ).

# 5.1 Setting a Breakpoint

First of all we need to activate debugging mode. There are several ways to do so. One of them is to set a breakpoint at the node where we want to have a closer look. When the test is being executed and QF-Test comes to the break point it will then pause and switch into debugging mode. The pause button III will then be activated.

• Select a node and press Ctrl-F8 (企業-B) on macOS). The breakpoint is indicated by a B.



Figure 5.1: Set break point

• Select the Test suite node and **press Enter** to start the test run.



Figure 5.2: Start test run

Action

• Remove the breakpoint by pressing Ctrl-F8 (①希B on macOS) again.

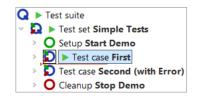


Figure 5.3: Remove break point

Instead of using the keyboard shortcut Ctrl-F8 ( $\bigcirc \Re$ -B on macOS) you may also set or unset a breakpoint by clicking the node and selecting the Debugger $\rightarrow$ Breakpoint on/off menu item, or alternatively right-clicking the node and selecting the Breakpoint on/off context menu item.

Again, you can see the little arrow, which now marks the next node to be executed, called the **current node**. When entering debugging mode QF-Test also navigates to the current node, in case it had not been visible, and selects it, highlighting it blue.

The menu option Debugger $\rightarrow$ Clear all breakpoints is useful to remove all breakpoints set in your test suite.

There is no limit to the number of breakpoints you can set in your test suite, but note that breakpoints are not saved with the test suite.

# 5.2 Stepping Through a Test or Sequence

Now let's step through the test case we set up in the previous section.

Action

Please try out the debugger buttons Single step <sup>b</sup>, Step over <sup>c</sup> and Step out

You will find that **Single step v** opens a node containing child nodes and makes the first child node the active node. Continuing from where we left the test suite at the end of the last section, i.e. in debugging mode, with 'Test case: First' being the current node, the test suite would now look like this:

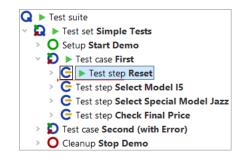


Figure 5.4: Stepping into a node

In the case of leaf nodes (nodes without child nodes), the effect of  $\mathfrak{D}$  is the same as the following button's.

**Step over**  $\bigcirc$  runs the current node including all children. Execution pauses at the next node of the same level to be executed, which then becomes the active one.

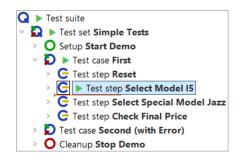


Figure 5.5: Stepping over a node

**Step out**  $\checkmark$  runs the remaining nodes at the same level including their child nodes. Execution pauses when a node that is higher in the hierarchical structure is found, which then becomes the active one.

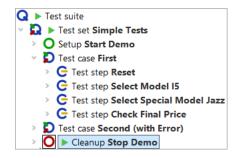


Figure 5.6: Stepping out of a node

#### 5.3. Skipping Execution of Nodes

In the given example the node higher in the hierarchical structure where execution stops is the Cleanup node. As explained in the first chapter <u>A full Test Run<sup>(15)</sup></u> this shows the special behavior of Setup / Cleanup nodes in a test set: They are executed before and after **each test case** to help achieving a proper starting state for each test case.

- Note You will only find this behavior when you started the whole test suite or test set and are in debugging mode. If you just selected the test case and did a step-over action then QF-Test will execute the test case and then select the next test case node.
- Run the Cleanup and Setup nodes by stepping over them, using the debugger button **Step over** <sup>♥</sup> and then **Step in** the second test case via <sup>♥</sup> to get ready for the next section where you will learn about the skip functionality.
- Note Please be aware that menus or comboboxes tend to close when the application looses the focus, as will happen when activating the debugging mode. In such a case you should not stop test execution between the node opening the menu or combobox and the node performing the selection. One way to do achieve this is to set a break point **B** after the node performing the selection and to activate normal test execution by releasing the pause button **II** when you reach the node opening the menu or combobox.

# 5.3 Skipping Execution of Nodes

The "skip" functions expand the QF-Test debugger's capabilities 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 them. This may be helpful for various reasons, e.g. to quickly navigate to a certain position in your test run or to skip a node which currently leads to an error.

With the end of the last section, the test suite should have reached the following state:

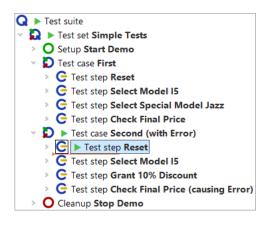


Figure 5.7: Pause execution at first node of the second test case

#### 5.3. Skipping Execution of Nodes

Action • Now press the Skip over <sup>♥</sup> button. QF-Test simply jumps over the active node without executing any child nodes. The active node now is the next node to be executed on the same level.

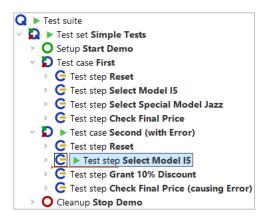


Figure 5.8: Skip over a node

• At last, press the **Skip out** <sup>𝒞</sup> button. You see that QF-Test skips all nodes on the same (or lower) level and directly jumps to the next node one level up in hierarchy.

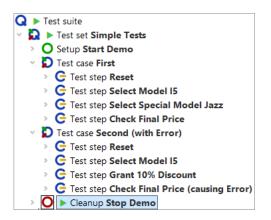


Figure 5.9: Skip out of a node

Note Use "Skip over" and "Skip out" cautiously as skipping out of a sequence before it is completed can leave the SUT in an unknown state that other sequences or tests in your test suite cannot react to.

Action

# 5.4 Error or Exception triggering Debugging Mode

When debugging a test you may want run it until it encounters an error, an exception or sometimes even a warning and then have it pause in debugging mode.

In this section you will see how this can be done while debugging the second test case.

- Please open the debugger menu and change the default options as follows:

  - Also activate the Debugger→Options→Break on error menu item.

Afterwards, when you open the debugger menu and options submenu it should look like this:

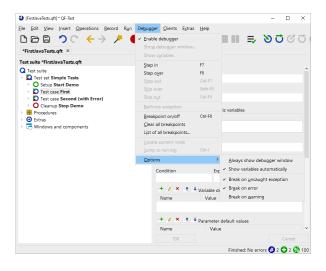


Figure 5.10: Set debugger options to pause on error

We changed the debugger options because with default settings QF-Test will not pause on exceptions or errors, as you saw earlier on.

Action • Select the "Test-suite" node and start test execution via "Start test run" 🕨 .

QF-Test will pause at the faulty node and enter debugging mode:

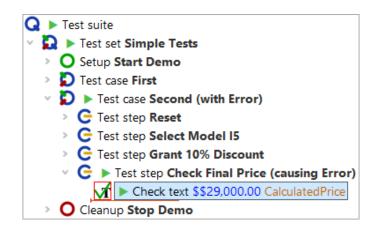
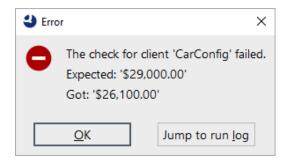
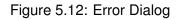


Figure 5.11: Test run paused by error

The node which halted test execution will be indicated by an arrow and its symbol will be surrounded by a red square. Also, an error dialog will inform you about the failed check.

As always the run log is the key to resolving errors, so let's open it and find out how to resolve the error in the next section.





Action

• Click the Jump to run log button in the error dialog.

# 5.5 Resolving Errors directly from the Run log

The **Jump to run log** button from the dialog in <u>figure 5.12<sup>(60)</sup></u> will not only open the run log but takes us directly to the node that holds the error details. Apart from the actual error message you will find screenshots and a copy of the variable bindings table (stack trace), which we will introduce later on (The Variable Bindings table<sup>(68)</sup>).

The error details tell you that the expected value does not match with the one shown in

#### 5.5. Resolving Errors directly from the Run log

the application. As the one in the application is correct we want to update the expected value with the one from the application. This can easily be achieved as follows:

- Action
- Right-click the red-bordered node "Failed: Check text: default ..." indicating the actual error
  - Select Update check node with current data from the context menu.

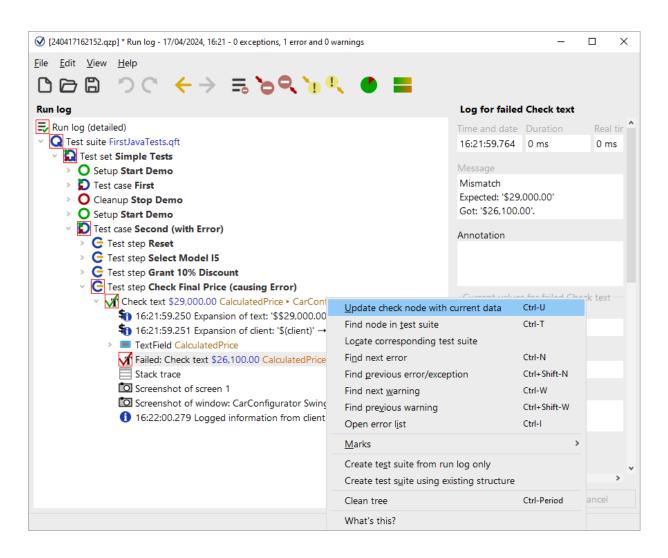


Figure 5.13: Update check node with current data

This locates the corresponding Check text node in the test suite and updates the expected value of the Text attribute with the value got as indicated by the run log.

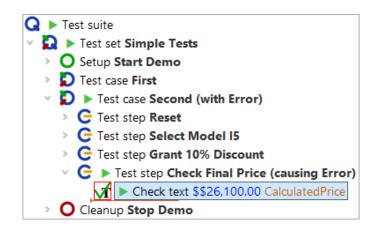


Figure 5.14: Corrected check node

The previously faulty node still is highlighted with a red border since we have not run it again.

#### Action Now continue execution by releasing the pause button II.

QF-Test runs the rest of the test suite, i.e. the Check text and Cleanup nodes, and informs you at the end of the run that there was one error, which you have already fixed.

Since the error has been fixed and we will continue using the test cases as examples you could rename the second test case and delete '(with error)' in its name as well as '(causing Error)' in the name of the test step.

**Jump into run log:** You do not have to wait for an error dialog to open the run log at the current point of execution (or close by). Whenever you are in debugging mode, select the **Debugger** $\rightarrow$ **Jump to Run log** menu option, or use the **Ctrl-J** shortcut. If you just want to open the run log without jumping to the current point of execution you can use **Ctrl-L**. This will work after the test run finished, too.

## 5.6 Pause Execution

When a test is being executed and you want to enter debugging mode you can quickly set a breakpoint at some node not yet executed. Or you can just hit the toolbar button "Pause" III and QF-Test will directly enter debugging mode.

In order to resume execution just release the pause button **II**. This is completely independent of the way you entered debugging mode.

Depending on how focus demanding the SUT is, it may be difficult to focus QF-Test long enough to hit the pause button. But you can still rely on the "Don't Panic" short-

cut <u>Alt-F12</u>. It will pause all running tests immediately. To continue, press the same combination again.

# **Chapter 6**

# Variables and Procedure Parameters (Java)

In this chapter you are going to learn how to use a procedure to perform the same action on various input data. You will also learn about variables - how to use and how to debug them.

This chapter is also available as a video tutorial at



"Variables and Procedure Parameters" https://www.qfs.de/en/yt/tutorial-6.html

#### 6.1 Procedure using a variable

Have a look at the last test step 'Check final price' of our two test cases.

Video

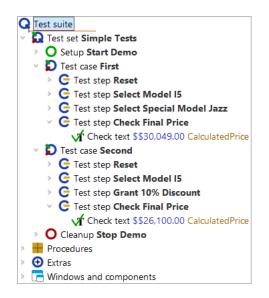


Figure 6.1: Two almost identical test steps

They perform the same action, however, with different data. Even though it is only one node, it makes sense to pack it into a procedure. We may want to adapt the hard coded values \$\$30,049.00 and \$\$26,100.00 to a different format so that the check will also work when the format of the price field changes to a different currency. And we do not want to implement the same algorithm twice. (If you wonder about the two dollar signs: This is what QF-Test records for a dollar sign. For replay it does not matter whether you use one or two dollar signs.)

- Action
- Select the 'Check text' node in the first test case.
  - Use **Operations**→**Pack nodes**→**Sequence** or use the <u>Ctrl-Shift-S</u> shortcut to pack it into a sequence.
  - Name the sequence checkFinalPrice. The procedure name follows the Java convention to run the words together and start the single words with capital letters. On the other hand QF-Test allows the use of spaces in procedure names, so you are free to name it as you like.
  - Press **Ctrl-Shift-P** for the quickest way to transform the 'Sequence' node into a procedure, as you learned at the end of the last chapter. You see the sequence is replaced by a call to the 'checkFinalPrice' procedure.
  - **Double click** the procedure call node to jump to the procedure definition.
  - **Open** the procedure node to see its content.

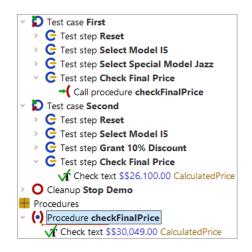


Figure 6.2: Procedure with hard coded value

As expected, the check is now located in this procedure. However, it is valid for one price only, i.e. \$30,049.00. Since we want to use the same procedure for the second test case as well we need to make the price a variable and pass its value as a parameter from the test case to the procedure.

In the next example we will start by defining a parameter for the procedure. Additionally, we will set a default value for the parameter. Default values are most useful when the parameter usually has that value and you do not want to specify it every time you call the procedure. Even though this does not hold true for the price it is a good example to demonstrate to you how a default parameter works and how to overwrite it with another value.

Let's define the parameter and add a default value:

#### Action • Select the procedure 'checkFinalPrice'

- **Press** the <sup>d</sup> "Add new row" button belonging to the table 'Parameter default values'.
- Enter price as name for the parameter.
- Enter \$30,049.00 in the value field.
- Click the 'OK' button.

Procedure			
Name			
checkFinalPrice			
-+ -/ -× -+ -+	Parameter	default values	
Name	Value		
price	\$30,049.00	0	
Exception QF-Test ID			~
Delay before (ms)		Delay after (ms)	
Comment			

Figure 6.3: The Details of the 'Procedure' node

The next step is to replace the value of the Text attribute of the Check text node by a reference to the variable.

Note Variable syntax: When working with variables you need to bear in mind that in some places you need to tell QF-Test the name of the variable and in others you want to refer to the value of the variable. In the Name column of the Parameter default values table of the Procedure node QF-Test expects the name of a variable. It is price, which is why you typed the word price.

In the Text attribute of the Check text node details QF-Test expects a character string for comparison with the text of the UI element. As we want to use a value stored in a variable, we have to tell QF-Test not to use the entered string as plain text, but to interpret it as a reference to the value stored in a variable. In QF-Test this is done by enclosing the variable name in (). In our case the variable reference is (price). If you did not put the variable name into (), QF-Test would compare the price (a number) shown in the UI element to the string price, which is obviously nonsense.

- Action Select the Check text node in the procedure 'checkFinalPrice'.
  - **Type \$ (price)** in the Text attribute of the Check text node details.
  - Click the 'OK' button of the node details.

Check text	
Client	
\$(client)	
C QF-Test component ID	
CalculatedPrice	
Text	
\$(price)	
\$ 🗌 As regexp	
\$ Negate	
Check type identifier default	
Timeout	
Result handling	
Variable for result	
Local variable	
Error level of message	
Error	×
\$ Throw exception on faile	ure
Name	
QF-Test ID	
Delay before (ms)	Delay after (ms)
Comment	

Figure 6.4: 'Check text' node

### Action • • Run the first test case.

It should execute without an error.

# 6.2 The Variable Bindings table

The next step is to make use of the procedure call in the second test case as well.

#### 6.2. The Variable Bindings table

Action • Replace the Check text node of the second test case by a procedure call to checkFinalPrice. You can simply copy the respective node from the first test case or add the procedure call as learned in the previous chapter.

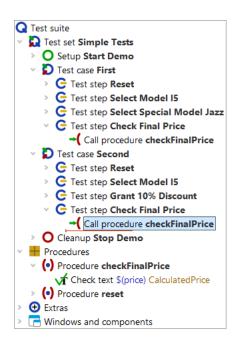


Figure 6.5: Second procedure calls 'checkFinalPrice'

- Note If you added the procedure call by a copy or drag and drop operation from the procedure itself you will find the price in the Variable definitions table of the procedure call. This is what we are eventually aiming at. However, at this stage we want to explain the default value. So, if you want to follow the tutorial exactly, please delete the default value by pressing the red X above the table.
- Verify **QF-Test is configured to pause at errors** as shown in <u>Set debugger</u> options to pause on error<sup>(59)</sup>.
  - Select the 'Test case: Second' node.
  - Execute it by pressing or Enter.

An error message shows up indicating different values for the price expected and the price got. What went wrong? Let's go hunting. Typically we use the run log for this but there is another view worth to know of.

• So click OK to close the error message.

#### 6.2. The Variable Bindings table

In debugging mode you will find an additional bottom right section of the QF-Test window showing a list of nodes with variables bound to those nodes.

- Action
- You might want to resize the variable bindings table in case is too small to see all its content.

Variable bindings				
Node	Test suite	Bindings	+ / ×	d definition
(•) Procedure checkFinalPrice	FirstJavaTests.qft	0	Name	Value
Call procedure checkFinalPr	FirstJavaTests.qft	0	price	\$30,049.00
C Test step Check Final Price	FirstJavaTests.qft	0		-
Test case Second	FirstJavaTests.qft	0		
Globals		1		
Command line		3		
<b>Q</b> Test suite	FirstJavaTests.qft	0		
Fallback stack		0		
• Procedure checkFinalPrice	FirstJavaTests.qft	1		
System		0		

Figure 6.6: Variable bindings

The variable bindings table is very useful for debugging. It comes in quite handy, too, when working with procedures and trying to understand the way QF-Test figures out which variable value to use. It shows the current values of the variables.

**Note** QF-Test always checks the variable bindings table from top to bottom.

You can see that the first rows of the table have no bindings at all. Then there is a binding at the level 'Globals' and another one in the fallback stack for the procedure 'checkFinalPrice'. The global variable is used for the client connection, which has been set when starting the application (cf <u>Starting the Application</u><sup>(5)</sup>). The other variable is more interesting to us - only it has the wrong value.

The default value is intended to be used for the parameter if no value has been defined elsewhere. This is why we added the parameter to the 'Parameter default values' table of the procedure node.

To do things correctly we need to pass the proper value when calling the procedure. Again, there are several ways to do it. One is to add a new row to the variable definitions table of the 'Call procedure' nodes similarly to the way you did at the 'Procedure' node in the last section.

If the procedure is called multiple times within the test suite, there is a better way:

- Action
- Stop the current test execution clicking the toolbar button 🔳 .
  - Right-click the 'Procedure' node and select
     Additional node operations→Update parameters of references
     from the popup menu.

[FirstJavaTests.qft] * QF-Test				– 🗆 X
				~
<u>File Edit View Insert Opera</u>	ations <u>R</u> ecord R <u>u</u> n De <u>b</u> u	gger <u>C</u> lients		
	←→ 🥕 🛑 🔳			0000
*FirstJavaTests.qft ×				
Test suite *FirstJavaTests.qft		Procedur	e	
<b>Q</b> Test suite		Name		
Test set Simple Tests		checkFinal	Price	
Setup Start Demo				
<ul> <li>D Test case First</li> <li>C Test step Reset</li> </ul>	C <u>u</u> t	Ctrl-X	1 Parameter default values	
C Test step Reset	<u>С</u> ору	Ctrl-C	Value	
C Test step Select :	<u>P</u> aste	Ctrl-V	Anal <u>y</u> ze references	
C Test step Check I	Delete	Delete	Analyze <u>d</u> uplicates	
Call procedure	E <u>d</u> it	Alt-Enter	Ana <u>l</u> yze invalid characters	
Test case Second Fest step Reset	<u>M</u> arks	>	Analyz <u>e</u> empty nodes	
C Test step Reset     C Test step Select I	Start run		Make calls relati <u>v</u> e	
G Test step Grant 1	Toggle disabled state	Ctrl+Shift-D	Make calls absolute	~
Centre Check I	Breakpoint on/off	Ctrl-F8	Make references explicit	
→ Call procedure	Locate references	Ctrl-I	Make references implicit	
Cleanup Stop Demc	Add to boo <u>k</u> marks	Ctrl-D	Update parameters of references	
Procedures     O     Procedure checkFine	Pack n <u>o</u> des	>	Para <u>m</u> eterize	
Check text \$(price	Additional nod <u>e</u> operations	>	Fill <u>c</u> omment	
> (•) Procedure reset	Insert node	>	nt	
> 🔁 Extras	New window			
Windows and componer –	Expand all	Alt-Right		
	Collapse all	Alt-Left		
	Clean tree	Ctrl-Period		Cancel
	What's this?			
	what's this?		Finished: 0 exceptions, 1 error, 0 warnin	gs 册 1 👽 1 🎲 100

Figure 6.7: Popup menu for 'Additional node operations'

• In the following dialog, please check the tick mark for **Add missing parameters to callers** is set and **click OK**.

In the 'Call procedure' nodes QF-Test adds a row each for every default parameter to the variable definition table of the procedure call. It also copies the default value of each default parameter. In our case it is the parameter price with the value \$30,049.00.

You might notice that the numerical value of the price variable is still wrong in the second case, regardless of whether it is defined implicitly as a default value or explicitly via a parameter. For now we want to keep that error to show you additional means of debugging.

• Close the 'Updated nodes' dialog QF-Test opened to inform you about the updated nodes.

# 6.3 Advanced debugging of variable bindings

Next, we want to explore the variable bindings table and see how it can be used for debugging purposes. For this reason do not correct the faulty value of the procedure call we added in the last section, but let us find out more about debugging.

In the next steps we want to have QF-Test pause the test execution at the procedure call. Then we will step into the procedure and, while doing so, have a look at the variable bindings table. Finally, we will navigate directly from the variable bindings table to the faulty procedure call and correct the parameter value.

- Add a breakpoint to the 'Call procedure: checkFinalPrice' node of the second test case.
  - Run the second test case again.

Action

 When QF-Test stops at the breakpoint step into the procedure via <sup>3</sup> and watch the table of the variable bindings as you do so.

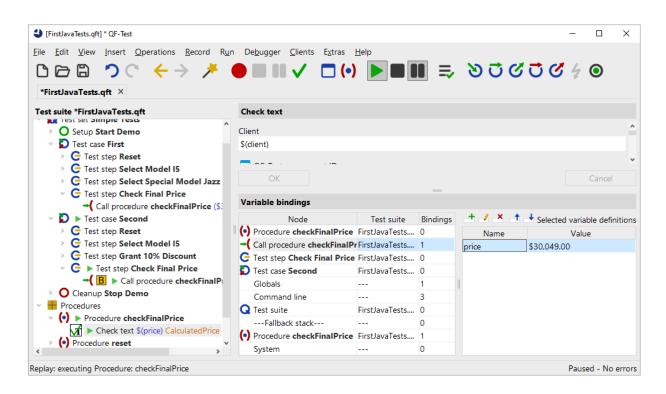


Figure 6.8: Variable bindings stack showing incorrect value

As you step into the procedure, first the row 'Call procedure: checkFinalPrice' and with the next step the row 'Procedure: checkFinalPrice' appear at the top of the table.

#### 6.3. Advanced debugging of variable bindings

Now the variable price shows on two different levels of the variable bindings table: In the 'Call procedure: checkFinalPrice' row and the 'Procedure: checkFinalPrice' row on the fallback stack. Since we did not adapt the value of the parameter passed to the procedure, neither of the two values bound to the price variable is correct.

QF-Test lets you change the values of the variables interactively in the variable bindings table when you are in debugging mode. You can even add new variables or delete them. However, changes to variable values in the variable bindings table are not persistent. They only last as long as the variable is on the stack (variable bindings table). In our case, if we changed the price value, as long as the procedure is being executed.

The parameter value in the procedure call will not be altered by changing the current value of the variable in the variable bindings table. To do so you have to navigate to the Procedure call node and change it there.

To get there quickly, you just double-click the procedure call in the variable binding table (second row). This feature is particularly useful when debugging more complex tests where the node you want to jump to is not directly visible in the test suite window. You can invoke it via right-clicking the row and selecting Jump to node in test suite from the pop-up menu, too.

- Action Double-click the second row with the procedure call in the variable bindings table.
  - Set the value of the parameter to the correct value, i.e. \$26, 100.00.

When checking the variable bindings table you will notice that the current value has not changed. This is hardly surprising as we have not yet executed the procedure call again. Only, test execution is already past the procedure call. Fortunately, QF-Test has another very useful debugging feature to set back (or forward) test execution to some node: **Continue execution from here**, which can be invoked either via the popup menu of the node you want to make the current node or by pressing Ctrl-, after selecting the node.

In order to try out the newly set value:

Action

- Right-click the 'Call procedure: checkFinalPrice' node of the second procedure.
  - Select 'Continue execution from here' in the popup menu.

Image: Second Secon	32	= >> =						tions <u>R</u> ecord R <u>u</u> n De <u>b</u> ugge	e Edit View Insert Opera
Test suite *FirstJavaTests.qft       Procedure call         ist suite       (•) Procedure name         Test set Simple Tests       (•) Procedure name         Setup Start Demo       Variable for return value         Test step Reset       Cut         Test step Select Model       Cut         Test step Select Model       Cut         Test step Select Specia       Cut         Test step Check Final P       Paste         Call procedure check       Delete         Delete       Delete         Egit       Alt-Enter         Value       Cancel         Marks       Start run         Locate procedure       Ctrl-P         Test step Check Final       Dogle disabled state         Ctrl-P       Transform node into         Transform node into       Transform node into         Pack ngdes		=> 00			(•)	_			
st suite       (•) Procedure name         I est set Simple Tests       (•) Procedure name         O Setup Start Demo       Variable for return value         I rest case First       Variable for return value         I rest case First       Variable for return value         I rest case Second       Cut         I rest step Select Model       Copy         I rest case Second       Edit         I rest step Select Model       Delete         I rest step Select Model       Delete         I rest step Select Model       Delete         I rest step Select Model       Start run         I rest step Check Final       I run         I run       I ru									FirstJavaTests.qft ×
Test set Simple Tests       checkFinalPrice         O Setup Start Demo       Variable for return value         O Test case First       Variable for return value         O Test step Reset       Cut       Ctrl-X         O Test step Select Model       Gopy       Ctrl-C         O Test step Select Model       Delete       Delete         O Test step Select Model       Delete       Delete         O Test step Select Model       Edit       Alt-Enter         Variable definitions       Edit       Alt-Enter         Value       Marks       Start run         Ioggle disabled state       Ctrl-Piss       Start run         Ioggle disabled state       Ctrl-Pisst       Name       Value         Procedures       Procedure       Ctrl-Pisst       Name       Value         Procedure seet       Add to boogmarks       Ctrl-D       irst0       Name       Value         Pack ngdes        1       Additional nodg operations        3       irst0					I .	lure cal	P		st suite *FirstJavaTests.qft
O Setup Start Demo       checkFinalPrice         Variable for return value         C Test step Reset         C Test step Select Model         C Test step Select Specia         C Test step Select Specia         C Test step Check Final P         C Test step Select Model         D Test case Second         E Test step Reset         C Test step Select Model         S Test step Select Model         C Test step Select Model         S Test step Select Model         S Test step Check Final         I oggle disabled state         C Test step Check Final         I oggle disabled state         Locate procedure         C Test step Select Model         S Test step Select Model         S Test step Check Final         I oggle disabled state         Locate procedure         C Tri-P         Marks         C Procedures         Procedures         Pack ngdes					ame	cedure i	(		
Test case First       Variable for return value         C Test step Reset       Cut       Ctrl-X         C Test step Select Model       Copy       Ctrl-C         Paste       Ctrl-V       Paste       Variable definitions         C Test step Check Final P       Paste       Ctrl-V         C Test step Check Final P       Delete       Delete       Value         C Test step Reset       Egit       Alt-Enter       Value         C Test step Select Model       Start run       Start run       Ioggle disabled state       Ctrl-F8         C Test step Check Final P       Ioggle disabled state       Ctrl-F8       Isrst 0       Isrst 0         C Test step Check Final P       Ioggle disabled state       Ctrl-F8       Isrst 0       Isrst 0         C Test step Check Final P       Ioggle disabled state       Ctrl-P8       Isrst 0       Isrst 0         Procedures       Cancel Procedure       Ctrl-P       Isrst 0       Isrst 0       Isrst 0         Procedure reset       Add to boogmarks       Ctrl-D       Isrst 0       Isrst 0       Isrst 0         Pack ngdes						nalPrice	c		
C Test step Reset       Cut       Cut-X         C Test step Select Model       Copy       Ctrl-X         C Test step Select Specia       Copy       Ctrl-C         P C Test step Check Final P       Pelete       Delete         C Test step Reset       Cut.       Alt-Enter         Variable definitions       Edit       Alt-Enter         Value       Marks       Start run         C Test step Select Model       Start run       Ioggle disabled state       Ctrl-F8         C Test step Check Final       Toggle disabled state       Ctrl-F8       Eest         C Test step Check Final       Toggle disabled state       Ctrl-F8       Eest         C Cleanup Stop Demo       Breakpoint on/off       Ctrl-F8       Eest       Name       Value         Procedures       Cocate procedure       Ctrl-P       irst 0       Transform node into       irst 0       Price       \$30,049,00       S30,049,00         Procedure reset       Additional nodg operations									
<ul> <li>G Test step Select Model</li> <li>G Test step Select Specia</li> <li>G Test step Select Specia</li> <li>G Test step Check Final P</li> <li>( Call procedure check</li> <li>C Test step Reset</li> <li>C Test step Reset</li> <li>E Test step Select Model</li> <li>G Test step Check Final P</li> <li>C Cleanup Stop Demo</li> <li>Procedure CheckFinalP</li> <li>Add to boogmarks</li> <li>C Transform node into</li> <li>Fransform node into</li></ul>				e	rn value	for retu		Cut	
<ul> <li>C Test step Select Specia</li> <li>C Test step Check Final P</li> <li>C Test step Check Final P</li> <li>C Call procedure check</li> <li>C Test step Reset</li> <li>C Test step Select Model</li> <li>C Test step Select Model</li> <li>C Test step Select Model</li> <li>C Test step Check Final P</li> <li>C Cleanup Stop Demo</li> <li>Procedures</li> <li>C Check text S(price) C</li> <li>P Procedure checkFinalP</li> <li>Add to boogmarks</li> <li>C Transform node into</li> <li>Fransform node into</li></ul>								-	C Test step Select Model
<ul> <li>C Test step Check Final P</li> <li>C Test case Second</li> <li>C Test step Reset</li> <li>C Test step Select Model</li> <li>C Test step Select Model</li> <li>C Test step Check Final</li> <li>C Cancel</li> <li>Marks</li> <li>Start run</li> <li>I oggle disabled state</li> <li>C trl-P</li> <li>B Call procedure</li> <li>C trl-P</li> <li>Add to book marks</li> <li>C trl-D</li> <li>Transform node into</li> <li>Tra</li></ul>									
Call procedure check       Delete       Delete <t< td=""><td></td><td></td><td>ble definitions</td><td>rial</td><td>+ va</td><td></td><td></td><td>-</td><td></td></t<>			ble definitions	rial	+ va			-	
Image: Second of the construction						ər			
> C Test step Select Model         > C Test step Select Model         > C Test step Grant 10% Di         > C Test step Grant 10% Di         > C Test step Check Fina         → ( B) > Call procedure         > C Cleanup Stop Demo         Procedures         (•) > Procedure checkFinalP         → C Procedure checkFinalP         → C Procedure reset         Extras         Add to bookmarks         Ctrl-D         Transform node into         Pack nodes         Additional node operations         > 1         Windows and components									
Image: Step Grant 10% Direct Step Grant 10% Direct Step Grant 10% Direct Step Check Finat Grant 10% Direct Step Check Finat Grant 10% Direct Step Check Final Procedures       Image: Ima	ncel		_			,			- I
• (B) Call procedure         O Cleanup Stop Demo         Procedures         (•) Procedure checkFinalP					ıgs				
O Cleanup Stop Demo       Locate procedure       Ctrl-P       irst 0       irst 1         Procedure checkFinalP       Add to bookmarks       Ctrl-D       irst 0       irst 0         Image: Check text S(price) C       Procedure reset       Add to bookmarks       Ctrl-D       irst 0         Extras       Additional node operations       1       irst 0       irst 0         Windows and components       Continue execution from here       Ctrl-Comma       irst 0	definitio	Selected variable	+ / × + +		est	ift-D			
Procedures       Add to book marks       Ctrl-D       irst 0         Image: Add to book marks       Continue execution from here       Ctrl-Comma	uemmuo								
(•) Procedure checkFinalP       Irransform node into       Irrst       0         (•) Procedure reset       Transform node into       First       0         Pack nodes       >        1         Extras       Additional node operations        3         Windows and components       Continue execution from here       Ctrl-Comma       First       0		\$30,049.00	price \$	1	irst 1				
(•) Procedure reset     Pack nodes      1       Extras     Additional node operations      3       Windows and components     Continue execution from here     Ctrl-Comma     irst				)	First 0				Procedure checkFinalP
Extras Additional node operations > 3 Windows and components Continue execution from here Ctrl-Comma						>		-	► Check text \$(price) C
Windows and components Continue execution from here Ctrl-Comma						>		—	•
Continue execution non nere continue					-	>	ns	Additional nod <u>e</u> operations	
						mma	here (	Continue execution from here	Windows and components
Insert node				1	- 0	>		Insert node	
Clean tree Ctrl-Period 0				1	irst 1			Insert node	
hanges applied What's this? Paused - No						riod	(	-	

6.3. Advanced debugging of variable bindings

Figure 6.9: Continue Execution from here

When checking the variable bindings table you will find that the top two rows have disappeared. This is because now we exited the procedure (even though 'backwards') and therefore the procedure call and all its variable bindings was taken off the call stack.

Action • Release the pause button II.

The test run should finish without error.

Note As the variable bindings table is very useful when looking for incorrect variable values you will also find a copy of it in the run log whenever an error or exception is logged. It is written to the subnode 'Stack trace' of the node causing the error, showing the variable values at the time the error occurred.

**Locate the current Node:** Sometimes during debugging you will navigate far away from the current node where execution stopped and eventually want to get back to it again.

The easiest way to do so is by pressing the "Locate Current Node"  $\bigcirc$  button or select the Debugger $\rightarrow$ Locate Current Node menu option to cause the debugger to "select" the current node.

# 6.4 Setting Variables

In addition to the methods you have already seen, variables can also be set as follows:

- Via the Set variable node,
- as the return value of a procedure,
- as the result value of QF-Test nodes like the Fetch text node, the Fetch geometry node, the Fetch index node and the Check node,
- in the 'Variable definitions' table of the 'Test-suite' node, the 'Test case' node, the 'Test step' node, the 'Sequence' node and others like the 'If' node or 'Loop' node,
- via command line parameters.

For information about the best place where to define a variable please refer to the next section.

You can insert a Set variable node via the menu item  $\boxed{\text{Insert} \rightarrow \text{Miscellaneous} \rightarrow \text{Set variable}}$ . In its details you can specify whether the value should be bound as a local or a global variable.

The following figure shows the details of a Set variable node. (You can find it as the first node of the Setup node.) It defines a variable named client. It is a global variable as the Local variable attribute has not been checked.

Set variable		
Variable name		
client		
Local variable		
Default value		
CarConfig		
\$ Interactive		
Description		
Timeout		
OF-Test ID		
Delay before (ms)	Delay after (ms)	
beidy before (ind)	being arter (mb)	
Comment		
Comment		

Figure 6.10: Details of the Set variable node

When you want to set a variable as the result of a procedure call you need to specify the variable name in the 'Variable for return value' attribute of the procedure call. Within the procedure itself you have to add a Return node with the value to be returned as the last node to be executed.

The next figure shows a theoretical example of a procedure which returns a value. The procedure fetches the discount value displayed in the SUT and returns it to the calling test case. There, the receiving variable is named Discount and declared as a local variable.

④ [FirstJavaTests.qft] * QF-Test	- 🗆 X
<u>File Edit View Insert Operations Record Run</u>	De <u>b</u> ugger <u>C</u> lients E <u>x</u> tras <u>H</u> elp
*FirstJavaTests.qft ×	
· ·	
Test suite *FirstJavaTests.qft	Procedure call
Q Test suite	Procedure name
V 🔂 Test set Simple Tests	getDiscount
> O Setup Start Demo	5
> D Test case First	Variable for return value
D Test case Second (with Error)	Discount
Test case Third	I
← Call procedure getDiscount → Discount	<ul> <li>Local variable</li> </ul>
> O Cleanup Stop Demo	🛨 🏒 🔺 🕂 Variable definitions
<ul> <li>Procedures</li> <li>Procedure getDiscount</li> </ul>	Name Value
	Nume Value
Fetch text DiscountValue → returnValue	
→ Return \$(returnValue) > ⊕ Extras	05 T 110
<ul> <li>Extras</li> <li>Windows and components</li> </ul>	OK Cancel
<ul> <li>Windows and components</li> </ul>	OK Cancel
Changes applied	

Figure 6.11: Procedure returning a value

The Fetch text node of above example is one of the QF-Test nodes directly setting a variable value. You need to specify the variable name in the attribute called accordingly. Again, you have the choice whether to make it local or global.

Quite a number of nodes have a 'Variable definitions' table where you may define variables local to a procedure or a test case. If the respective node is part of a procedure the variable will be local to the procedure. Otherwise it will be local to the respective test case. Variables bound to the test suite node can be accessed from all nodes within the test suite.

In debugging mode, all nodes you can bind a variable to will show up on the variable bindings table when entered.

You can enter variables in the command line via the parameter -variable. For details please refer to the manual, chapter 'Command line arguments and exit codes'.

## 6.5 Variable binding levels

Note This section may be difficult to understand when you are a programming beginner. Then it may be better to come back to it when you started writing procedures for your own tests.

In QF-Test there are many places where you can set a variable:

- The test suite node,
- in test cases and procedures as default or local variables,
- as a parameter in a procedure call,
- as a global variable and
- as a command line parameter.

Now the question is: Which is the correct place for defining a variable?

And the answer is: It depends on the use case.

Each level of the variable bindings has its own use case:

#### **Procedure parameters**

When you call the same procedure several times in a test case and with different values each time you should set the values via a parameter at a procedure call. This is done by adding a row to the variable definitions table of a Procedure call node specifying the respective variable name and value.

#### Local variables in a procedure

They are created in the procedure and are deleted when the execution of the procedure finishes. Use a local variable when you do not need it outside of the current procedure. In procedures they are predestined for intermediate results.

#### Local variables in a test case

Variables local to a test case are either created during execution of a test case or defined in the respective table in the details of the test case node. They will be deleted from the variable bindings table once the test case finishes. Use them for values you want to refer to in several nodes of a test case and you do not need outside of it.

#### **Global variables**

Global variables are created at some point of the test execution and exist until they are explicitly deleted or QF-Test is stopped. They survive stops and restarts of test execution. Use them for values that need to be accessible by all test cases. A typical example is the variable client created in the 'Setup' node when starting the application. To get rid of them you need to either exit QF-Test or select the menu item Run $\rightarrow$ Clear global variables.

#### **Command line parameters**

In batch mode you may want to run test with various parameters. They are valid throughout the batch run. A typical example would be the browser to be used.

#### 6.5. Variable binding levels

#### **Test-suite variables**

test suite variables can be referred to by all test cases. Their usage is the same as that of global variables. Only that they can be overridden at batch execution by command line parameters, whereas global variables cannot.

#### Default values (Fallback stack)

You can define default values for variables of procedures, test cases and test sets. In case you do not define a variable of the same name on a higher level QF-Test will use the default value set.

# **Chapter 7**

# The Standard Library (Java)

#### Video This chapter is also available as a video tutorial at



The Standard Library"

https://www.qfs.de/en/yt/tutorial-7.html

QF-Test provides a certain number of node types. If you need additional functionality you can implement it in a script node. To make life easier for you QF-Test comes with a set of procedures implementing the most commonly needed additional functions. You will find them in the standard procedures library.

When you cannot solve a problem using the provided node types it is a good idea to have a look in the standard library whether there is a solution to your problem. If you find a similar solution you can copy the procedure and adapt it to your needs. For information about scripting please refer to the manual, chapter 12 'Scripting'.

The file qfs.qft holds the standard procedures library. As it is constantly being enhanced and distributed with every new version of QF-Test you should not make any changes to procedures in that file, but copy the procedure to your own test suite if required and then adapt it.

- Note To make use of <code>qfs.qft</code> it needs to be included in your test suite's root node. With a newly created test suite the file <code>qfs.qft</code> is added automatically to the list of included files.
- Select the 'Test-suite' root node of your test suite.
  - Verify the <code>qfs.qft</code> is available within the table for "Include files".
  - Add  ${\tt qfs.qft}$  to this list, if it's not already there.
- Note Path information is not necessary for qfs.qft as the include directory of QF-Test is contained in the library path (see also Reference part of the manual).

Action

• Add a procedure call to an arbitrary procedure from the qfs.qft standard library. In the procedure chooser don't miss to switch to the respective tab.

In addition to the description provided in this tutorial you can find the full HTML documentation of the standard library available via Help-Standard Library qfs.qft...

# 7.1 Inspecting the Standard Library

In addition to inserting procedure calls from the Standard Library, it also can be helpful to sometimes have a look how certain things have been implemented.

Action

• Locate and load the test suite file <code>qfs.qft</code>, which is located in the <code>qftest-7.1.3/include</code> directory of your QF-Test installation.

🕹 [qfs.qft] QF-Test	- 🗆 X
<u>File Edit View Insert Operations</u>	<u>R</u> ecord R <u>u</u> n De <u>b</u> ugger <u>C</u> lients E <u>x</u> tras <u>H</u> elp
CBB 2C ←→	> 🗡 ●■■✔ 🗖 (•) ▶■■ ☴ 📎  Ơ ℧ Ơ ϟ ⊙
qfs.qft ×	
Test suite qfs.qft	Test suite
Q Test suite	Name
Procedures	
Package qfs	
Package android	+ / × Include files
Package automac	File
Package autoscreen	
> 🕂 Package autowin	
> 🕂 Package awt	🕂 🏒 📩 🕂 Dependencies (reverse includes)
> 🕂 Package cleanup	
Package client	File
Package check	
> 🕂 Package daemon	
> 🕂 Package database	🕂 🧭 📩 🕂 Variable definitions
Package databinder	Name Value
Package dialogs	
> 🕂 Package <b>fx</b>	
> 🗕 Package jib	
> Package pdf	Execution timeout (ms)
Package run-log	
Package shellutils	
Y 🖶 Package swing	Comment
> Package cleanup	This suite contains useful standard procedures provided and maintained by Quality First Software GmbH.
> Package checkbox	
> Package combobox	For a description of packages and procedures of this library in javadoc format, see qfs.html in the same directory.
Package dialogs	
Package filechooser v	OK

Figure 7.1: The Standard Library

#### 7.2. Selected Packages and Procedures

You can see there is one main package qfs that contains further specific packages. The qfs package helps to easily identify the packages belonging to the standard library.

The specific packages cover very different areas of utility procedures described in more detail further below.

Note

Within nearly all of the procedures of this library, you'll notice that the variable  $\client$ ) is referenced. This is the standard mechanism for creating independence from a specific SUT. Here, the library assumes that the test suite which uses the library will set a value for  $\client$ ) prior to using any procedures.

## 7.2 Selected Packages and Procedures

We will now have a closer look at a number of selected packages and procedures from the standard library.

We will start with packages for accessing components dependent on the UI technology they are implemented with, say JavaFX, Java Swing, Eclipse/SWT and Web components.

### 7.2.1 The Checkbox Packages

We begin by looking at the qfs.fx.checkbox qfs.swing.checkbox, qfs.swt.checkbox or qfs.web.checkbox package.

Some important procedures within this package are:

- **select** Selects (checks) a checkbox. If the checkbox is already selected, then no action is taken.
- **deselect** Deselects (un-checks) a checkbox. If the checkbox is already deselected, then no action is taken.
- set Sets a checkbox to a given state (true or false).

For each of these procedures, you pass the QF-Test ID of a checkbox component as a variable argument. The library handles verification of whether or not the checkbox state was properly set as expected.

Usage of the other procedures in this package follows the general model seen here.

### 7.2.2 The Combobox/Combo Packages

The packages qfs.fx.combobox, qfs.swing.combobox, qfs.swt.combo or qfs.web.select contain procedures to select a value in a combobox. Some procedures within this package are:

- setValue Select a value in the list of the combobox.
- getItemCount Return the number of entries in the select box.

## 7.2.3 The General Packages

The packages qfs.fx.general, qfs.swing.general, qfs.swt.general or qfs.web.general contain useful procedures to work with components.

Some procedures within this package are:

- setLocation Set the location of a given component.
- setSize Set the size of a given component.

### 7.2.4 The List Packages

The packages qfs.fx.list, qfs.swing.list, qfs.swt.list or qfs.web.list contain useful procedures to work with lists.

Some procedures within this package are:

• getItemCount Return the number of items of the list.

### 7.2.5 The Menu Packages

The qfs.fx.menu, qfs.swing.menu, qfs.swt.menu or qfs.web.menu packages allow you to easily select items and checkbox items from menus or sub-menus. The procedures visible after expanding the package node are:

- **selectItem** Selects an item from a menu.
- **selectSubItem** Selects an item from a sub-menu.

For each of these procedures, you must pass the QF-Test ID of the menus as well as the item and/or sub-item to select or check; the usage varies slightly depending on the nature of the procedure.

### 7.2.6 The Table Packages

The packages <code>qfs.fx.table</code>, <code>qfs.swing.table</code>, <code>qfs.swt.table</code> and <code>qfs.web.table</code> provide utility procedures for tables, like:

- **getRowCount** Return the number of rows of a table. It uses technology specific methods to gather the row number.
- **getColumnCount** Return the number of columns of a table. It uses technology specific methods to gather the column number.
- **selectCell** Select a given table-cell.

### 7.2.7 The Tree Packages

We've provided some simple access procedures for manipulating trees within the packages <code>qfs.fx.tree</code>, <code>qfs.swing.tree</code>, <code>qfs.swt.tree</code> and <code>qfs.web.tree</code>. These include:

- **collapseNode** Collapse a node of a tree using separate parameters.
- **expandNode** Expand a node of a tree using separate parameters.
- **selectNode** Select a given tree node.

For any of these procedures, you simply pass the QF-Test ID of the node you wish to manipulate.

## 7.2.8 The Cleanup Packages

The packages <code>qfs.fx.cleanup</code>, <code>qfs.swing.cleanup</code> and <code>qfs.swt.cleanup</code> are useful for generic cleanup of the SUT environment after an unexpected exception occurs. Imagine, for example, that an exception is thrown while attempting to manipulate a menu in the SUT. The exception will cause the execution path within your test suite to be re-directed to an exception handler of a Catch node or of a dependency, or an "implicit" exception handler. This means that the normal flow of execution, which would have properly closed the open menu, has now been interrupted. Without proper action, that menu could be left open and thus block other events directed to the SUT.

Some important procedures within this package are:

• closeAllModalDialogs Ensure that modal dialog windows of the SUT are closed. only available for Swing and FX!

- closeAllDialogsAndModalShells Ensure that dialog and modal shells of the SUT are closed. only available for Eclipse/SWT!
- **closeAllMenus** Close all menus of the SUT unconditionally.

The concept of implicit exception handling is an important one as an exception in a single test case is not meant to stop the whole test run. Just the current test case needs to be aborted and then continue with the next test case.

Therefore, an exception raised during the course of a Test case will be caught at that level and not propagated to prevent aborting the whole test run. The error state is duly noted in run log and report however.

If the Test case uses a Dependency the exception is passed to the Catch node of the same, if available. This kind of exception (and error) handling is described in the Dependency chapter of the manual.

## 7.2.9 The Run log Package

The <code>qfs.run-log</code> package contains procedures, which writes specified messages into the run log. This package has been introduced to give testers without scripting-knowledge the opportunity to write messages into the run log.

Here is the list of the most important procedures within this package:

- logError Write a given error message into the run log.
- **logWarning** Write a given warning message into the run log.
- **logMessage** Write a given message into the run log.

## 7.2.10 The Run log.Screenshots Package

The <code>qfs.run-log.screenshots</code> package contains procedures, which write images into the run log and some helper methods.

Some important procedures within this package are:

- getMonitorCount Return the total number of monitors.
- **logScreenshot** Write a screenshot of the whole screen into the run log.
- **logImageOfComponent** Write an image of a given component into the run log.
- **logScreenshotOfMonitor** Write a screenshot of a given monitor into the run log.

### 7.2.11 The Shellutils Package

The  ${\tt qfs.shellutils}$  package contains procedures to support most common shell-commands.

Some important procedures within this package are:

- **copy** Copy a given file or directory to a specified target.
- deleteFile Delete a given file.
- exists Check for existence of a given file or directory.
- getBasename Return only the file name of a full file name.
- getParentdirectory Return only the directory name of the full file name.
- mkdir Create a given directory. It also creates non-existing directories in path.
- move Move a file of directory.
- touch Create a specified file.
- removeDirectory Remove a specified directory.

### 7.2.12 The Utils Package

The <code>qfs.utils</code> package contains procedures, which covers common helper-functionality during test-development.

Some important procedures within this package are:

- **getDate** Return a string containing the date. Default is the current date. (Other dates can be configured.)
- **getTime** Return a string containing the time. Default is the current time. (Other timestamps can be configured.)
- logMemory Log current memory use.
- printVariable Print the content of a given variable to the console.
- printMessage Print a given message to the console.
- writeMessageIntoFile Write a given string into a given file.

## 7.2.13 The Database Package

The <code>qfs.database</code> package contains procedures to execute SQL commands on a database.

Please note, that the class of the database-driver must be in QF-Test's plugin directory or in the CLASSPATH before QF-Test startup.

To get more information about the connection-mechanism to your database, please ask your developers or see www.connectionstrings.com.

Some important procedures within this package are:

- executeSelectStatement Execute a given SQL-Select-Statement. It stores the result in a global variable "resultRows" on the Jython variable stack and thus accessible from Jython scripts. Additionally, it stores the result in a group variable with the default name 'resultGroup', which can be accessed directy by QF-Test nodes.
- **executeStatement** Execute a given SQL-command. Here any SQL command can be specified.

## 7.2.14 The Check Package

The qfs.check package contains procedures to do checks.

Some important procedures within this package are:

- **checkEnabledStatus** Check, whether a component is enabled or disabled. It writes an error into the run log, if failing.
- **checkSelectedStatus** Check, whether a component is selected or not. It writes an error into the run log, if failing.
- **checkText** Check the text of a component. It writes an error into the run log, if failing.

## 7.2.15 The Databinder Package

The <code>qfs.databinder</code> package contains procedures for execution within a "Data driver" node which bind data for iteration.

Some important procedures within this package are:

• **bindList** Create and register a databinder that binds a list of values to a variable. Variables are separated by whitespace or by a given separator character.

#### 7.2. Selected Packages and Procedures

• **bindSets** Create and register a databinder that binds a list of value-sets to a set of variables. Value-sets are separated by line breaks. Variables within a value-set are separated by whitespace or by a given separator character.

# **Chapter 8**

# **Control structures (Java)**

The two most important control structures of QF-Test are loops and the conditional execution of nodes. Loops can be implemented by two different kinds of nodes: While and Loop nodes. If, Elseif and Else nodes are available to implement conditional execution.

eo This chapter is also available as a video tutorial at



"Control Structures" https://www.qfs.de/en/yt/tutorial-8.html

# 8.1 If - else

You already came across If and Else nodes in the Setup sequence in the chapter <u>Starting</u> the Application<sup>(5)</sup>. Let's have a closer look at the details of the nodes.

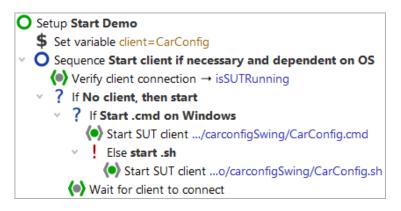


Figure 8.1: Setup Sequence with if-else structures

By means of an If node you can control whether certain nodes will be executed or not.

Video

In our case whether to start the SUT application. First, we need to find out if the client is already running. This is the job of the Wait for client to connect node, which writes the result of its inquiry, either true or false, into a variable named isSUTRunning.

Wait for client to connect		
Client		^
\$(client)		
Timeout		
0		
GUI engine		
Result handling		
Variable for result isSUTRunning		
<ul> <li>Local variable</li> </ul>		
Error level of message Error	~	
\$ Throw exception on failure		
QF-Test ID		
Delay before (ms) Del	ay after (ms)	
	-) ()	
Comment		
This node checks whether the SU result of this check will be stored isSUTRunning. The variable itself of	in the variable	
already running or false if SUT is r		~

Figure 8.2: Wait for client to connect writes the result into the variable "isSUTRunning"

The If node has a Condition attribute where you'll find an expression evaluating the result variable isSUTRunning. As we want to refer to its value we need to use the syntax () (see also note on variable syntax in chapter section 6.1<sup>(64)</sup>).

If				
Condition			Script langua	ge
<b>not</b> \$(isSUTRu	nning)		Jython	~
Name				
No client, then st	art			
+ / × +	↓ Variable de	efinitions		
Name	Value			
Maximum error le	evel			
Exception				~
QF-Test ID				
Delay before (ms)	)	Delay after (ms	)	
Comment				

Figure 8.3: If node evaluates the variable

Depending on whether the client is already running or not QF-Test will execute the nodes nested in the If node.

• Stop the client in case it is running.

Action

#### • Single-step through the Setup node.

• Leave the client running and single-step through the Setup node a second time.

If you like you can check the value of the variable <code>isSUTRunning</code> in the variable bindings table. The first time it will have the value <code>false</code> so that the condition <code>not</code> <code>\$(isSUTRunning)</code> will become true and the SUT will be started. The second time it will be <code>true</code> and the if-condition will fail. The nodes nested in the If node will be skipped.

Within the first If node there is a second one checking the type of operation system - either Windows or else a different one (Unix or Mac). This is where the Else node comes in, which will be entered if the condition test of the If node fails (in our case: the operation system is not Windows).

For checking the operating system you can directly resort to a QF-Test variable: QF-Test stores the information about the operation system in a group variable where the group is called 'qftest' and the variables 'linux', 'macos' or 'windows', respectively. The syntax for accessing group variables is group:varname, e.g. group:varname.

# 8.2 Loops

QF-Test provides two different kinds of nodes loops:

- Loop nodes execute their child nodes for a certain number of times. However, you can leave the loop any time using a Break node.
- While nodes execute their child nodes until a certain condition becomes false. Again, you can leave it any time using a Break node.
- Note Loop nodes will always stop after the given number of times. In the case of While nodes, however, you need to make sure that the condition will become false at some point. Otherwise you would have an infinite loop. In interactive mode you can always stop execution by hitting the pause button II. In batch mode you would have to kill the QF-Test process. (You start QF-Test in batch mode using the command line parameter -batch. Then QF-Test does not start its UI and just executes the given test suite.)

In the following exercise we want to implement a test case checking whether a certain row is displayed in the table of the CarConfig application.

The actions of the test case will be:

Action

- Determine the number of rows the table has.
- Loop over all rows and check if it is the row we are looking for.
- Break the loop when a match was found.
- Write an error to the run log if the row was not found.

Please start with recording a check on the row of interest:

- Activate the check recording mode by clicking the toolbar button  $\checkmark$ 
  - **Right-click a row** in the CarConfig application and select the menu item Row from the popup menu.
  - Stop the recording by pressing

- · Change the name of the recorded sequence to e.g. 'Check row'
- Turn the recorded sequence into a test case by right-clicking it and selecting the submenu item Transform node into→Test case from the popup menu.

[FirstJavaTests.qft] * QF-Test				– 🗆 X
	unting Decend Due Debu	Climate	Fortune - U.S.	
<u>File Edit View</u> Insert Ope	erations <u>R</u> ecord R <u>u</u> n De <u>b</u> u	gger <u>C</u> lients		$\rightarrow c \pi \rightarrow$
	$\leftarrow \rightarrow \nearrow \blacksquare$			S C C C
*FirstJavaTests.qft ×				
Test suite *FirstJavaTests.qft		Sequence	•	
<b>Q</b> Test suite		Name		
Test set Simple Tests		Charle Boy		
Procedures	C <u>u</u> t	Ctrl-X		
<ul> <li>         • Extras         • O Sequence Check Rov         •     </li> </ul>	<u>С</u> ору	Ctrl-C	T Variable definitions	
Check items: row	<u>P</u> aste	Ctrl-V	Value	
Sequence Recorded	Delete	Delete		
Windows and componer	E <u>d</u> it	Alt-Enter		
	<u>M</u> arks	>		
	Start run			
	Toggle disabled state	Ctrl+Shift-D	rror level	
	<u>B</u> reakpoint on/off	Ctrl-F8		`
	Add to boo <u>k</u> marks	Ctrl-D		
	Trans <u>f</u> orm node into	>	O S <u>e</u> tup	
	Pack n <u>o</u> des	>	O <u>C</u> leanup	
	Unpack node <u>s</u>		(•) Procedure Ctrl+Shift	t-P
	Additional nod <u>e</u> operations	>	0 <u>L</u> oop	
	Insert node	>	Sequence with time limit	
	<u>N</u> ew window		D Test c <u>a</u> se	
	Expand all	Alt-Right	C Test step	
	Collapse all	Alt-Left	🔇 <u>W</u> hile	
	Clean tree	Ctrl-Period		Cancel
Changes applied	What's this?			Finished: No error

Figure 8.4: Transform a node into another one

In general, QF-Test lets you add nodes very efficiently by packing one node into another one:

• Open the test case node and **pack the recorded** Check node into a loop by rightclicking it and selecting the submenu item Pack nodes→Loop from the popup menu.

4 [FirstJavaTests.qft] * QF-Test				- 🗆 ×
Eile Edit View Insert Operations Re		ger <u>C</u> lients E <u>x</u> tras		ರ ರ <b>ರ ೮</b> ರ
Test suite *FirstJavaTests.qft		Check items		
Test suite     Test set Simple Tests     Procedures		Client \$(client)		
<ul> <li>Extras</li> <li>Test case Check Row</li> <li>Check items: row (2) VehicleT</li> <li>Sequence Recorded 17/04/202</li> <li>Windows and components</li> </ul>	C <u>u</u> t <u>C</u> opy <u>P</u> aste Delete Edit	Ctrl-X Ctrl-C Ctrl-V Delete Alt-Enter	t ID 1 ms Text	Regexp
	<u>M</u> arks		>	
	Start run <u>T</u> oggle disabled sta	ate Ctrl+Shift-D	Seguence	Ctrl+Shift-S
	<u>B</u> reakpoint on/off Locate component	Ctrl-F8 Ctrl-W	? <u>I</u> f O <u>L</u> oop	Ctrl+Shift-I
	Highlight compone Show i <u>n</u> Inspector		C Test step Sequence with time limit	Ctrl+Shift-T t
	Add to boo <u>k</u> marks		<ul> <li>▼ Iry</li> <li>&gt; O Setup</li> </ul>	Ctrl+Shift-Y
	Pack n <u>o</u> des		> 🔇 <u>W</u> hile	
	Additional nod <u>e</u> op	perations	>	~
	Clean tree	Ctrl-Period		Cancel
	What's this?			Finished: No erro

Figure 8.5: Pack a node into another one

QF-Test evaluates dynamically which nodes may be packed into one another and only presents the appropriate ones. So, in case you do not find the 'Loop' submenu item make sure you have right-clicked the correct node. The same holds true for the 'Transform node into' and 'Insert node' methods.

In the next series of actions we want to set the value for the Number of iterations attribute of the Loop node. In order to do so we need to find out how many rows the table has. There is no simple node that you could use. However, in the last chapter we learned that the standard library provides a lot of extended functionality. So let's insert the procedure getRowCount from the package qfs.swing.table in the standard library.

#### Action • Select the Test case node and press Ctrl-A

• Press the 'Select procedure' button (•) left to 'Procedure name'.

94

- Click the tab 'qfs.qft' in the 'Select procedure' dialog.
- Navigate to 'getRowCount' in the package 'qfs.swing.table'
- Click 'OK' to select it.
- Click 'OK' in the 'Procedure call' dialog.

Adding a procedure via <u>Ctrl-A</u> was described in <u>Manual creation of procedures</u><sup>(35)</sup>. If you would like to check with the screenshots please have a look there.

#### • Enter the variable name rows in the Variable for return value attribute.

- Change the default value for the id in the variable definitions table to the QF-Test component ID of the table, i.e. VehicleTable.
- Click the **OK** button.
- Select the Loop node.
- Enter a reference to the variable \$ (rows) in the Number of iterations attribute of the Loop node.
- Enter the name of an iteration counter, e.g. i in the respective attribute of the Loop node.
- Click the **OK** button.

Loop		
Name		
Number of Description		the set of the second set
Number of iterati	ions	Iteration counter
\$(rows)		i
+ / × +	🚽 Variable d	efinitions
Name	Value	
Maximum error l	evel	
Exception		~
QF-Test ID		
QUEICSCID		
Delay before (ms	)	Delay after (ms)
🗸 Comment		

Figure 8.6: Details of a Loop node

In the next series of actions we will change the recorded row index to the iteration counter and add a variable for the result to the details of the Check node. Then we will add an If node after the Check node evaluating the result, with a Break node within to quit the loop when the row was found.

#### Action

#### Open the Loop node.

- Select the Check node.
- Change the recorded row index of the QF-Test component ID to the iteration counter \$(i). The QF-Test component ID should now read VehicleTable@Model&\$(i)
- Enter the variable name checkSucceeded in the 'Variable for result' attribute and click the OK button.
- **Right-click** the Check node and the submenu item Insert node →Control structures→Break from the popup menu.

Action

- Click 'OK' in the 'Break' dialog.
- Pack the Break node into an If node by pressing Ctrl-Shift-I) (Of course you can also pack it via the menu).
- Type \$ (checkSucceeded) in the 'Condition' attribute of the 'If' node and click the OK button.

The variable checkSucceeded will be set to either true or false by the Check node so that the reference to the variable  $\$  (checkSucceeded) is all we need to enter in the 'Condition' attribute of the If node.

In the next series of actions let's add an Else node as the last node in the Loop node. It will only be entered if all repetitions of the loop were executed, which in our case means that the row was not found and the check never became true.

- **Collapse the If node** if it is open. This is important because otherwise the Else node would belong to the If node and not to the Loop node.
  - Right-click the If node and select the submenu item Insert node  $\rightarrow$  Control structures  $\rightarrow$  Else.
  - Click 'OK' in the 'Else' dialog.
  - Open the Else node.
  - From the standard library insert the procedure logError contained in the package qfs.run-log as described above.
  - Type Row not found in the value field of message in the Variable definitions table.
  - Change the value of withScreenshots in the Variable definitions table from false to true.
  - Click 'OK' in the 'Break' dialog.

When you run tests in batch mode screenshots are a great help for analyzing errors. On the other hand a great number of screenshots lead to a big log-file. This is why the default value for withScreenshots is false.

Last, let's complete the test case with Setup and Cleanup nodes and move it into the top part of the test suite.

Action • Copy the Setup and Cleanup nodes of 'Test set: Simple Tests' into the new test case as the first and last node.

• Move the test case from the Extras section into the top section of the test suite after the 'Test set: Simple Tests' node.

This is what the new test case would look like:

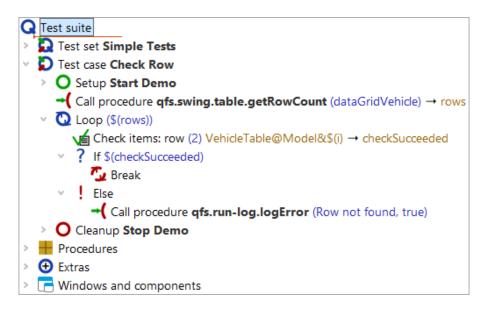


Figure 8.7: The new test case

#### • Execute the new test case.

It should run without error.

• Then modify a value in the details of the Check items node, e.g. change the name of the car to Wrong value.

Check it	tems		
Client			
\$(client)	)		
📑 QF-	Test component ID		
Vehicle	Table@Model&\$(i)		
+ /	🔺 📑 🚽 Items		
	Te	ĸt	Regexp
	rong value 15,000.00		
Check ty row	ype identifier		
Timeout			
nneout			
Variab	handling le for result Succeeded		
Loca	al variable		
	evel of message		
Error			~
\$ 1	Throw exception on fa	ilure	
Name			
QF-Test	ID		
Delay be	efore (ms)	Delay after (ms)	
Con	nment		

Figure 8.8: Details of the Check items node

### • Execute the new test case again.

This time the Else node should be entered and you should get an error message.

# **Chapter 9**

# It's time to start your own Application (Java)

After having spent a lot of time with all those example programs, you are now ready to start on your own application (if you really haven't already done so).

Video This chapter is also available as a video tutorial at



"It's time to start your own Application" https://www.qfs.de/en/yt/tutorial-9.html

The Quickstart Wizard available via the menu  $\boxed{\text{Extras} \rightarrow \text{Quickstart Wizard...}}$  helps you to achieve this. Simply follow the wizard steps to generate the setup sequence. Please refer also to chapter 3 "Quickstart" in the user manual.

Then go ahead with what you have learned in this tutorial - record small sequences of events and checks, turn them into procedures which go into your test library, then set up the test cases using procedure calls.

Finally, we reached the end of the basic tutorial part.

# Part II Web UI testing with QF-Test

This part II of the tutorial is meant to help you learn the basic features and workflows of QF-Test. It focuses on the test of web applications and its specifics.

For testing Java applications please go to part  $I^{(2)}$  or part  $III^{(203)}$  for native Windows programs, as those parts use the same scenarios but with different systems under test.

In case you already have worked through one of the other parts but web testing is relevant for you as well, you might not want to work again through the all same scenarios. Hence you should have at least a look at the sections <u>Creating the Setup Sequence<sup>(120)</sup></u>, <u>Web Component Recognition<sup>(144)</sup></u> and <u>Windows and components Section<sup>(146)</sup></u> that contain web specific parts.

Within part  $V^{^{(294)}}$  more advanced QF-Test features are explained, applicable for all supported UI technologies.

# Chapter 10

# Working with a Sample Test suite (Web)

In this first chapter, we will have a look at a simple test suite, explain its major elements, execute it and evaluate the result.

Video This chapter is also available as a video tutorial at



"Working with a Sample Test suite" https://www.qfs.de/en/yt/tutorial-1.html

# 10.1 Loading the Test suite

Note On first startup of QF-Test and/or the System Under Test (SUT) via QF-Test you might get a security warning from the firewall asking whether to block the Java network communication 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.

After starting up QF-Test, you can immediately bring up our first example test suite.

Action

- Press the boundary button to bring up the file open dialog
  - Navigate to the subdirectory **qftest-7.1.3/doc/tutorial** of your QF-Test installation
  - There select the file FirstWebTests.qft

QF-Test will then load the indicated test suite which should look as follows:

[FirstWebTests.qft] QF-Test				- 0	×
File Edit View Insert Operatio	ons <u>R</u> ecord R <u>u</u> n De <u>b</u> ugger <u>C</u> lients B	- 			
		] (•) ▶ ■ 88 ₹	<b>b C</b> C (	504	۲
Test suite FirstWebTests.qft	Test suite				
Q Test suite	Name				~
> D Test set Simple Tests	Name				
> 🕂 Procedures					
<ul> <li>→ Extras</li> <li>→ Windows and components</li> </ul>	+ ✓ × ↑ + Include files				
vindows and components	qfs.qft				
	1				
	🕂 🗹 📩 🕂 Dependencies (reverse	includes)			
	File				
	🛨 🏒 📩 🕂 Variable definitions				
	Name	Value			
	browsername				
	browserdir	D ( 05 D )			
	connectionMode	Prefer QF-Driver			
	Execution timeout (ms)				
	🧹 Comment				~
	OK			Cancel	
	Terminal				
					^
					~

Figure 10.1: The Test suite FirstWebTests.qft

The **left part** of the main window contains the test suite, organized in a tree structure. The **right side** shows the details of a selected tree node.

At **bottom right** you'll see the terminal displaying messages sent by QF-Test and the application you are testing.

In the tree structure of the main window you can navigate and select individual nodes of the test suite.

#### • Double click the node Test set: Simple Tests to expand it.

You'll find the test set contains two test case nodes enclosed by a "Setup"/"Cleanup" pair.

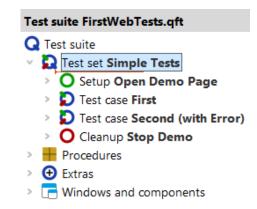


Figure 10.2: The "Test set: Simple Tests" Node

In the following sections we'll describe the purpose and function of the individual nodes.

## 10.2 Starting the Browser

Our first step is to examine the "Setup" node:

#### Action • Expand the Setup: Open Demo Page node now.



Figure 10.3: The Setup Node

In the "Setup" node you'll see two child Sequence nodes:

- Set global variables dependent on OS defines the global client variable used throughout the test suite and variables for the browser to be used dependent on the operating system (Chrome on Windows and macOS, Firefox on Linux).
- Start browser if necessary starts the respective browser if it is already not running and loads the demo web page.

Let's also have a brief look inside the Sequence: Start browser if necessary:

Figure 10.4: The Sequence to start the Browser

First you see a "Wait for client" node to double-check whether the client is already running. Only if it is not, it will be started.

The browser start itself happens in four steps:

- 1. Start web engine starts the browser process to allow configuration.
- 2. **Perform browser settings** configure the browser (e.g. cache, cookies, proxy settings, ...).
- 3. **Open browser window** open the browser window and wait for the web page to be loaded.
- Install CustomWebResolver configure component mapping, so QF-Test will recognize the functionality of components (text field, button, data table etc). Section <u>Web Component Recognition<sup>(144)</sup></u> explains the reasons why this is advisable, along with information on the configuration.

These four steps are automatically generated when you use the Quickstart Wizard, that will be explained in the next tutorial chapter 11<sup>(120)</sup>.

At this point, we're ready to actually start the Browser:

Action

- Click on the **O** Setup: Open Demo Page" so it is selected but still expanded (the child nodes stay visible).
  - Click the Start test run toolbar button. This button causes the selected node to be executed.

During execution QF-Test marks the active step by use of an arrow pointer "->".

When the setup sequence is completed, the browser with the "CarConfigurator" demo page is going appear on the screen.

CarConfigurator Web -	· Google Chrome			- L	
C X O	file:/T:/ins tall/qftest/qftes	t-branch/demo/carconfi	gWeb/html/CarConfig.htm?lar	ig=en	
	File Options	Purchase order	Help		
	Vehicles S	pecials Acce	ssories		
	Model	ID	Price		
	Hydro2	M1	\$79,000.00		
	Voyage	M2	\$56,500.00		
	Voyage Hybrid	M3	\$56,500.00		
	Roadster-E	M4	\$45,900.00		
	15	M5	\$29,000.00		
	Base price	\$(	0.00		
	Specials price		0.00		
	Accessories price		0.00		
	Discount	C	% -5%		
	Final price	so	.00		

Figure 10.5: The CarConfigurator web demo page

# 10.3 First Test case

Let's check out now what **test case** "First" contains. There are four test steps inside:

Ð	Test case First
>	C Test step Reset
>	C Test step Select Model I5
>	C Test step Select Special Model Jazz
>	C Test step Check Final Price

Figure 10.6: The "First" Test case

#### 10.3. First Test case

**Reset** - performs a reset by use of the File->Reset menu action. **Select Model I5** - chooses the last model I5 within the vehicles table. **Select Special Model Jazz** - switches to the Specials tab and choose the Jazz option. **Check Final Price** - checks that the calculated final price field located at bottom right equals a given value.

Test steps are used to group the nodes and to document what is being done. This will prove very useful when it comes to error analysis or test adaptations.

#### • Expand the four test step nodes.

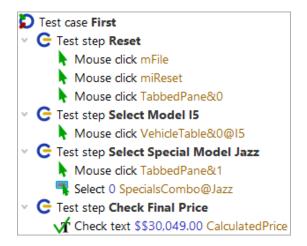


Figure 10.7: Details of the first Test case

You can see Mouse clicks and Checks, which have been grouped in test step nodes for better readability of the test case. The action nodes display the action type (Mouse click, Check, ...) and the component targeted, i.e. where the action goes to. When writing a test you can use the QF-Test recording function to create them. Recording will be explained in the next chapter 11<sup>(120)</sup>.

- Please select the test case "First" node
  - Click the replay button 🕨 .

Action

The test steps will then be replayed in the SUT, which will happen very quickly.

The test result is indicated during and after the test run in the status line at the bottom of the QF-Test main window and should read now 'Finished: No errors'. Next to it there are counters for the numbers and results of the test cases executed. In our case it was just one, error-free, which means a success rate of 100%.

Finished: No errors 🌐 1 🛟 1 🧐 100

Figure 10.8: The result view in the status line

Each counter icon has a descriptive tool tip. A list of all counters can be found in the chapter 'Capture and replay' of the manual.

## **10.4 Second Test case - with Error**

The second test case will show us what happens when an error occurs during test execution.

#### • Expand the test case "Second (with Error)".



Figure 10.9: The Second Test case

Apart from the third test step it is identical to the first test case. So what does this one do?

Test step: Grant 10% discount - Writes the value 10 into the discount field.

The 'Input' node is another basic action node and can be created directly via the QF-Test recording function.

• Expand the Test step: Grant 10% Discount.



Figure 10.10: Details of the second Test case

Let's execute the second test case:

#### Action

• Select the node Test case: "Second (with Error)".

Click the replay button ▶.

This time a dialog shows up telling us that an error occurred.



Figure 10.11: Error in the second test case

What happened? To find out we'll open the QF-Test run log for error analysis.

An alternative approach for error analysis would be to execute the test case again using the debugger. This will be explained in chapter Using the Debugger (Web)<sup>(155)</sup>.

# **10.5** The Run log for Error Diagnosis

QF-Test logs detailed information for every test execution.

Action • Please open the latest run log by one of the following options:

- either by pressing the Show run log button of the error dialog

or in case you have already closed the dialog

- by pressing **toolbar button** ⇒ or
- by pressing Ctrl-L.
- Note The most recent run logs are also listed at the bottom of the Run menu of the main window.

The run log comes up in a separate window displaying the logged actions of the test case you've just executed:

Action

ile Edit View Help	> ☴ `əQ`	0 🔍 🌒 🚍		
Run log	Run log			
Run log (detailed)	Test run ID			
Test case Second (with Error)	240417162847			
	Time and date	Duration	Real time spent	
	16:28:47.685 2024-		4.547 s	
	Exceptions	Errors 1	Warnings 0	
	Environment			
	Hostname	Operating system	Executed by	
	w10t2	amd64-Windows 10-		
	Java version	QF-Test v	ersion	
	17.0.10+7	7.1.3-dev	(30107)	
	ОК		Cancel	

Figure 10.12: Run log for the second test case

The run log reflects the tree structure of the test suite view you are already familiar with. When you click on one of the nodes on the left side, the properties of the event including time stamp and duration will be displayed on the right.

In the tree left you will notice nodes surrounded by a red border. These are indicators showing where a problem occurred in a child node. If you keep expanding the red nodes, you'll eventually come to the actual error node.

Please use an easier way to find the error source by pressing the Find next error
 toolbar button or the [Ctrl-N] key shortcut.

All nodes with red highlighting have been expanded and the actual error node has been selected:

Run log	Log for failed Check to	ext		
in log (detailed)	Time and date	Duration	Real time spent	^
Test case Second (with Error)	16:28:50.496 2024-04	0 ms	0 ms	
C Test step Reset				
C Test step Select Modell I5	Message			
C Test step Grant 10% Discount	Mismatch			
C Test step Check Final Price (causing Error)	Expected: '\$29,000.00'			
✓ ✓ Check text \$29,000.00 CalculatedPrice ► CarCo	Got: '\$26,100.00'.			
16:28:49.958 Expansion of text: '\$\$29,000.0	Annotation			
16:28:49.958 Expansion of client: '\$(client)' ·	Annotation			
TextField CalculatedPrice				
Failed: Check text \$26,100.00 CalculatedPric				
Stack trace				
Screenshot of screen 1	Current values for failed	d Check text		
Screenshot of window: CarConfig.htm [CarC	Client			
16:28:52.227 Logged information from clier	\$(client)			
	QF-Test componen	nt ID		~
< >	OK		Cancel	

Figure 10.13: Error in the second test case

The error message on the right says that the expected value of the final price field differs from the actual one. Of course this error is there by intention as the second test case is supposed to show us how to analyze an error.

Another helper for error analysis is the **Screenshot** of the SUT taken at the time when the error occurred (four nodes down from the red node). Being able to see the state of the SUT at that moment often proves useful for determining the cause of the error. The following image shows a screenshot node:

Run log	Screenshot	
in log (detailed)	Time and date	^
Test case Second (with Error)	16:28:51.042 2024-04-17	
C Test step Reset		
C Test step Select Modell 15	🔍 🔍 📙 💽 Screenshot (1680x1050) [1:1]	
C Test step Grant 10% Discount		
C Test step Check Final Price (causing Error)	CarConfigurator Web - Google Chrome	
Check text \$29,000.00 CalculatedPrice + CarCo	file:/T:/ins tall/qftest/qftest-branch/demo/car	
16:28:49.958 Expansion of text: '\$\$29,000.0		
16:28:49.958 Expansion of client: '\$(client)'	< II.	
TextField CalculatedPrice		
Failed: Check text \$26,100.00 CalculatedPric		
Stack trace	File Options	
Screenshot of screen 1		
Screenshot of window: CarConfig.htm [CarC	Mahialaa	
<ol> <li>16:28:52.227 Logged information from clier</li> </ol>	Venicies	
	<	~
< >	OK Cancel	

Figure 10.14: Screenshot node showing the error situation

In addition to screenshots of all monitors, QF-Test also saved images of the client windows at the time of the error. This allows you to analyze the contents even if they are covered by other dialogs or windows.

Note The information gathered in a long test run accumulates and can eat up enormous amounts of memory. This is the reason why QF-Test is configured by default to create a compact run log, keeping only the relevant information for report generation and error diagnosis.

This functionality can be configured via the option "Create compact run log" within  $\boxed{\text{Edit} \rightarrow \text{Options} \rightarrow \text{Run logs} \rightarrow \text{Content}}$ . The root node of the run log tells you whether it is a compact or detailed run log. You can also configure the number of screenshots to be saved.

# 10.6 Getting Help

We take a short break in this section to give you a few tips that might prove helpful as you continue with the tutorial.

There are different places where you can look for help or information:

The comprehensive be achieved via most search can ||Help $\rightarrow$ Online search...|. This to the search functionality navigates you homepage available on our and allows querying throughout all

#### 10.7. Stopping the Application

**documentation** (manual https://www.qfs.de/en/qf-test-manual.html, tutorial https://www.qfs.de/en/qf-test-tutorial.html, standard library https://www.qfs.de/en/qf-test-support/documentation/standard-library.html,

blog https://www.qfs.de/en/blog.html and our videos https://www.qfs.de/en/get-started-with-qf-test/videos.html). Search results can be filtered as needed.

In case you work **offline** and want to search for a certain topic, the **PDF versions of the manual or tutorial** available via the **Help** menu can be used. The Offline HTML version doesn't have a content search option. However, there is a link to the PDF on every HTML page in the top and bottom lines so switching is easy.

QF-Test also offers a **context sensitive help** for tree node types and their details. To use it, simply press the **right mouse button** on an arbitrary tree node or attribute in the details pane. From the context menu select **What's this?**. This will directly bring you to the reference explanation of this item in the manual.

Beside getting help from the documentation you also have the option to contact our support team. During your evaluation phase or after that as customer with a valid maintenance contract you may issue your questions directly to our support experts (send an email to support@qfs.de or use the support form from the QF-Test help menu Contact the support team or call us +49 8171 - 38648 20). Contact details can be found at https://www.qfs.de/en/qf-test-support/product-support.html.

# 10.7 Stopping the Application

We haven't inspected the cleanup sequence, so let's have a look at it now:

### • Expand the Cleanup: Stop Demo node.



Figure 10.15: The Cleanup Sequence

The Cleanup sequence stops the client process in a hard way and waits until it fully terminates. This is a very simple approach and shall suffice for the moment.

Action **Execute** the **cleanup** to see the Browser with the CarConfigurator demo vanish.

Action

## 10.8 A full Test Run

After we have seen how the single elements of the test set work, let us have a look at the functionality provided by the test set node.

- First, close the browser window with the CarConfigurator demo in case it is still running.
  - Then select the "Test set: Simple Tests" node.
  - Execute it with the replay button .

The result dialog will come up after test execution, informing us about the error caused by the second test case.

• Open up the run log again by ⇒ to take a closer look:

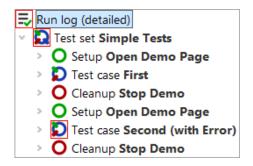


Figure 10.16: Run log for the Completed Test set

This shows the special behavior of Setup / Cleanup nodes in a test set: They are executed before and after **each test case** to help achieving a proper starting state for each test case.

Note Stopping the SUT after each test case is not the smartest way to ensure a clean state. There are more elegant ways for setup and cleanup that will be explained with the advanced features in this tutorial (chapter 29<sup>(304)</sup>).

# 10.9 Report Generation

In the world of quality assurance documenting the test results is pretty important. To this end, QF-Test offers an automated report-generation feature. Since you've just done a complete test run, we're at a good point to show you this feature.

#### 10.9. Report Generation

- Make sure the **run log of the test run** is open.
  - In the **run log window** select **File**→**Create report...** to bring up the dialog for the report parameters.

	×		
Save to directory C:\Users\qfs\.qftest\FirstWebTest	s 🕞		
Create HTML report	Create XML report		
Create JUnit report			
✓ List test steps	List checks		
<ul> <li>List exceptions</li> </ul>	✓ List errors		
List warnings	Show icons for nodes		
Pass HTML tags through	<ul> <li>Use doctag extensions</li> </ul>		
Ignore skipped nodes Ignore not implemented no			
Embed pie chart	<ul> <li>Include testsuite name</li> </ul>		
Embed thumbnails			
Scale factor for thumbnails (%) 20			
Show report in HTML browser when finished			
<u>O</u> K	Cancel		

Figure 10.17: Report Generation Properties

In the first field, you can specify the file name of the report. Following this, you can decide what type of report you want. QF-Test offers three kinds of reports, HTML, XML and JUnit format. An XML report is useful if you want to process the data further, e.g. if you have written your own XSLT stylesheets to shape the report. JUnit reports prove useful when you need to import results into build or test management tools.

Let's generate an HTML report from the results of the last test run.

• So just leave the report **options unchanged**.

Action

• Start generation by pressing the **OK** button.

The report will then be generated and presented to you in a browser window:

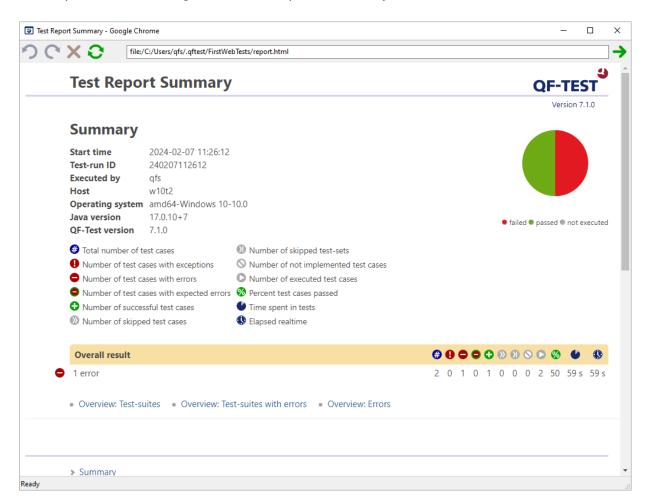


Figure 10.18: An HTML Report

The report begins with a summary containing informational data from your system on the top left side, a legend describing the meaning of icons used in the report on the top right side, an overview pie chart in the middle and the overall test result below. In our case, the result we see are the error-free first test case and the second with the well-known error, leading to a success rate of 50%.

Following the summary there are three overview sections:

- 1. Test suites that have been executed within the test run.
- 2. Test suites in which errors occurred.

3. Errors including their exact position and detailed message.

The report generator is very useful for creating an overview document for presentation and archiving purposes.

# Chapter 11

# Creating your own test suite (Web)

In the second chapter of the Web tutorial we will create our own sequences for starting and stopping an SUT from QF-Test. Furthermore we are going to record actions and checks and use those to build up a simple test case.

Video This chapter is also available as a video tutorial at



"Creating your own test suite" https://www.qfs.de/en/yt/tutorial-2.html

# 11.1 Creating the Setup Sequence

To begin, you need to launch the application from *qftest*. There is a **Quickstart Wizard** to help you in creating the respective setup sequence.

Action

- Open a new test suite via the menu item File New test suite.....
  - To open the Quickstart Wizard please use the Extras-Quickstart Wizard...
     menu.

The Wizard starts up with a welcome message and some further information.

Action • After saying a short hello please press the **Next** button to begin.

🕹 Quickstart Wizard	>
Pages	Welcome
First steps Welcome Type of application Application specifics Optional Conclusion Summary	<ul> <li>Welcome to the Quickstart Wizard!</li> <li>In general your application needs to be started by QF-Test in a specific way in order to allow recording and replay of user actions as well as verification of results.</li> <li>This applies to both stand-alone Java programs and web applications (defined by a URL).</li> <li>This wizard will assist you in creating a startup sequence for your application.</li> <li>Just press Next to begin.</li> <li>Further information can be found in the <u>Quickstart chapter</u> of the manual.</li> </ul>
	<u>B</u> ack <u>N</u> ext <u>Finish</u> <u>Cancel</u>

Figure 11.1: The Quickstart Wizard

In step two you can choose the type of application to be tested.

- Please select the second option **A web application in a browser**.
  - Press Next.

Action

🕘 Quickstart Wizard		×
Pages	Type of application	
First steps Welcome Type of application	Please select the type of your application from the following list. A Java application	
Application specifics		
Optional Conclusion Summary	<ul> <li>A web application in a browser (specified via URL)</li> <li>A web application in an emulated mobile browser</li> <li>A native Windows application</li> <li>An Android application</li> <li>An Electron application</li> <li>A PDF document</li> </ul>	
	<u>B</u> ack <u>N</u> ext <u>Finish</u>	<u>C</u> ancel

Figure 11.2: Type of Application

In step three you are asked for the URL of the web page to be tested.

Typically a http based URL is given here, but we will use the local CarConfig webdemo.

- Action
- Press the by toolbar button to bring up the file open dialog
- Navigate to the directory .../qftest-7.1.3/demo/carconfigWeb/html from your QF-Test installation
- There select the file CarConfig.htm.
- Conclude the file selection.

Note In the figure below we used the QF-Test variable \${qftest:dir.version} to ad-

in the manual chapter Variables).

실 Quickstart Wizard Х URL of the web page Pages First steps Please specify the URL for the web page to be tested, either directly on the net using the 'http://...' format or by selecting the respective HTML file. Welcome Type of application B \${qftest:dir.version}/demo/carconfigWeb/html/CarConfig\_en.htm Application specifics URL of the web page Optional Browser Web Framework Java executable Browser settings Proxy settings Browser window Client name Conclusion Summary Next <u>F</u>inish <u>B</u>ack Cancel

dress the version specific directory of the QF-Test installation, which you have already come across in the previous chapter. (Details on special QF-Test variables can be found

Figure 11.3: Specification of the URL.

- Press the **Finish** button, as we do not need to go to the further optional steps for our simple demo.
- Note Concerning the browser to be used for testing this means the wizard default will be selected (Chrome on Windows and macOS, Firefox on Linux). Just in case this is not possible for any reason, please checkout the optional next wizard steps to select a different browser.

We directly reach the final summary that explains what will happen after closing the wizard and how to continue.

### 11.1. Creating the Setup Sequence

#### Action

### • Please press **Finish** in order to end the wizard.

ages		Summary					
First steps Welcome Type of application		When pressing the "Finish" button, the 'Setup' sequence for your configured application (SUT) will be created within your test suite. You will find that sequence under the Extras node.					
Application specifics URL of the web page		Select this 'Setup' sequence and press the 'Start run' button in the toolbar.					
Optional Browser Web Framework		Your application should then be s	started and you will be ready for recording.				
Java executable		URL of the web page:	\${gftest:dir.version}/demo/carconfigWeb/html/Ca				
Browser settings		Browser:	Google Chrome				
Proxy settings Browser window		Web Framework:	Auto detection				
Client name		Java executable:	\${qftest:java}				
Conclusion		Empty cache at browser start:	true				
Summary		Allow cookies:	true				
		Delete cookies at browser start:	true				
		Locale:	en				
		Proxy settings:	No proxy				
		Browser window:					
		Client name:	CarConfig_en				
		<	>				

Figure 11.4: Final Information

The generated setup sequence **Launch CarConfig\_en** appears in the "Extras" section of your test suite and looks equivalent to the one we already know for the last chapter (section  $10.2^{(106)}$ ).

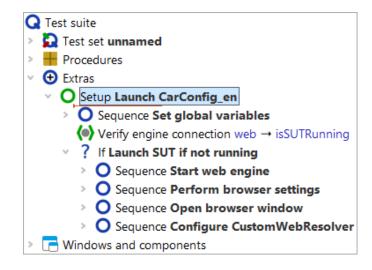


Figure 11.5: Generated Setup Sequence

Now we want to see some action:

- Please ensure the Setup: Launch CarConfig\_en node is selected.
  - Then **click** or simply hit "Enter" Return.

You should see the browser appear on your screen soon. First it shows a page that redirects you to the CarConfig demo. As the focus changes back to QF-Test after the execution, the browser might get covered by the test suite window.

CarConfigurator Web - 0	Google Chrome			- 0				
0 C X O	file:/T:/ins tall/qftest/qftest-bra	file:/T:/ins tall/qftest/qftest-branch/demo/carconfigWeb/html/CarConfig.htm?lang=en						
	File Options F	urchase order	Help					
	Vehicles Spec	ials Acce	ssories					
	Model	ID	Price					
	Hydro2	M1	\$79,000.00					
	Voyage	M2	\$56,500.00					
	Voyage Hybrid	M3	\$56,500.00					
	Roadster-E	M4	\$45,900.00					
	15	M5	\$29,000.00					
	Base price	\$0	0.00					
	Specials price	\$0	0.00					
	Accessories price	\$0	.00					
	Discount	0	% -5%					
	Final price	\$0	.00					

### 11.1. Creating the Setup Sequence

Ready

Figure 11.6: The "CarConfigurator Web" demo page in the browser

At the end of this section let's save our test suite.

- Action
- Press the toolbar button or use the File→Save menu option with its short-cut Ctrl-S.
  - In the file explorer navigate to an appropriate directory where you have write access e.g. Documents in your user home directory.
  - Provide a name e.g. MyFirstTests.qft.
  - Finish the saving action by pressing on Save.

## 11.2 Recording Actions

You're now ready to record some actions for our demo:

```
CarConfigurator Web - Google Chrome
                                                                                             _
                                                                                                   \times
  OCXO
                                                                                                          →
                         file:/T:/ins tall/qftest/qftest-branch/demo/carconfigWeb/html/CarConfig.htm?lang=en
                            File
                                    Options
                                               Purchase order
                                                                  Help
                             Vehicles
                                            Specials
                                                          Accessories
                            Model
                                                     ID
                                                                          Price
                           Hydro2
                                                     M1
                                                                    $79,000.00
                                                                    $56,500.00
                           Voyage
                                                     M2
                           Voyage Hybrid
                                                     M3
                                                                    $56,500.00
                           Roadster-E
                                                                    $45,900.00
                                                     Μ4
                            15
                                                     M5
                                                                    $29,000.00
                                                              $0.00
                           Base price
                           Specials price
                                                              $0.00
                           Accessories price
                                                              $0.00
                                                              0 %
                                                                           -5%
                           Discount
                           Final price
                                                             $0.00
Ready
```

Figure 11.7: Recording actions on the "CarConfigurator Web" demo

- Action
- Please press the **Record button**.
- Switch to the "CarConfigurator Web" window. From now on every mouse and keyboard action performed within the SUT window will be recorded.
- Click to the table cell 15 in the last row.
- Change to the **Specials** tab of the tabbed pane.
- Choose the special Jazz from the combo box.

Action

- Finally switch back to the **Vehicles** tab in the tabbed pane.
- Stop the recording by use of the **b**utton.

You'll find the recorded sequence placed in the "Extras" section:



Figure 11.8: The Recorded Sequence

The recorded sequence has a default name with date and time of the recording. You can change this name as you see fit by simply clicking on the node and changing its properties in the details view on the right.

- Please **rename** the recorded sequence to "Select Model I5 Jazz".
  - Then **open** the sequence node to see its content. There should be the expected mouse clicks. You should even be able to interpret where they go to.



Figure 11.9: The Renamed Sequence

Now, let us replay the recorded sequence.

- Select the **Select Model I5 Jazz** sequence node.
  - Press the play button

You should now see exactly the same sequence of mouse events executed in the SUT as you recorded before.

The sequence is supposed to replay (even multiple times) without errors. You should see "Finished: No error" in the bottom right corner of your test suite window.

Note While recording, QF-Test recognized the functional entities like the data table, the tab and the combo box, and subsequently addresses the subitems by index. For details see <u>Addressing subitems of tables, lists and trees<sup>(142)</sup></u>. This was possible due to the correct configuration of the component mapping. For further information see <u>Web Component</u> Recognition<sup>(144)</sup>.

## 11.3 Recording Checks

To verify the client's behavior we use check nodes, which query certain states and properties of elements within the SUT. Also checks can be recorded.

Action

Action

- Click the "Record a check" ✓ button.
  - Switch to the SUT window. When moving the mouse over the components you will notice a blue border indicating the current selection.
  - **Right-click** the value field of "**Final price**". In the popup menu you are offered a choice of standard checks for a text field component.
  - Select the first option "Text" for a check on the textual value of the field.
  - Stop the recording by using the **Stop button**.

Again, the newly recorded sequence appears in the "Extras" section.

- Please rename the sequence to "Check final price"
  - **Expand** it to see the check node.

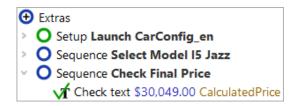


Figure 11.10: The recorded check node

• Feel free to run this sequence, too, verifying it is working properly.

As the next step, we want to create a test case from the two sequences.

## 11.4 Setting up a test suite

The top-level nodes of the test suite define its basic structure:

- Right after the "Test-suite" node an arbitrary number of "Test set" and "Test case" nodes may be added specifying the functional tests.
- "Procedures" contains reusable sequences (procedures) that can be organized into packages.
- "Extras" is the scratch pad for recording and experimentation.
- "Windows and components" contains the all-important elements of the test suite: registered elements of the SUT, e.g. windows, menus and buttons. The details of each element in the "Windows and components" section contains the properties of the recorded UI element required by QF-Test to find the component when replaying a test.

Functional test cases are represented by "Test case" nodes and can be grouped and structured with the help of "Test set" nodes.

"Setup" and "Cleanup" nodes are intended for test steps ensuring a well-defined state before and after a test case.

- Let's start by renaming the top-level test set node from "unnamed" to "Demo Tests".
  - If a dialog pops up asking us whether to update references we can simply confirm with "Yes".
  - The second step is to move the "Setup" node generated by the Quickstart Wizard from the "Extras" node into the "Test set" node right before the "Test case" node. Moving the "Setup" node can be done via mouse (Drag&Drop), contextmenu (right mouse-button copy/paste) or by Ctrl-X and Ctrl-V keyboard commands.

Action

Action

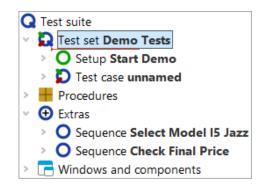


Figure 11.11: Start organizing the test suite

The next step is to make a test case of the two sequences previously recorded.

- Please **rename** the test case node from "unnamed" to "First".
  - **Open** the test case node by clicking the '+' symbol.
  - Move the two sequences from "Extras" into the test case.

You need to open the test case node because otherwise QF-Test would try to place the sequence nodes after the test case node on the same level, which is not a valid option.

QF-Test always records sequence nodes. They have the same functionality as test step nodes, only they do not show up in the report. So, just to show you, we will transform the two sequence nodes into test step nodes.

- Please open the context menu for the first of the two sequence nodes by a rightclick.
  - Choose Transform node into...→Test step
  - Repeat this for the **second** sequence node.

Your test suite should now look like this:

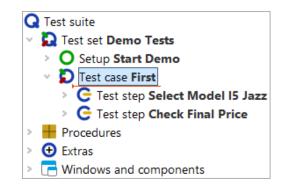


Figure 11.12: The Organization of your test suite

Now you've completed the most important steps of structuring a test suite.

### 11.5 Stopping the Demo

The only thing missing now is a cleanup sequence closing down the SUT.

There are usually various ways to terminate an SUT, e.g. clicking the close button of the application window, pressing ALT-F4 or via the menu action  $\boxed{\text{File} \rightarrow \text{Exit}}$ . All these options can be directly recorded and then used in the cleanup sequence.

Let's use the first one.

- Action
- Start recoding
- Click the window closing button of the browser.
- You see the demo window close.
- Stop recording
- Rename the recorded sequence to "Stop demo".
- Open the **context menu** for the recorded sequence and select **Convert into...** -> Cleanup.
- Finally move the cleanup node up to be the last node in the test set.
- Note The cleanup node can only be dragged and dropped to the test set if the test set's last child node is collapsed. To expand or collapse a node during a drag and drop operation, hold your cursor over the triangle next to the node.

You should end up with the following:

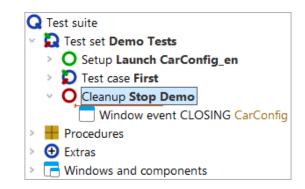


Figure 11.13: The simple cleanup sequence

By this we have finished with the basic steps of structuring our suite.

## 11.6 Running the whole test suite

Finally let's execute our newly created suite:

- Action
- Terminate the SUT client in case it is running.
  - Select the root node of the test suite.
  - Run it by pressing "Replay" ▶ or typing Return.

The SUT is expected to appear, the test case will be executed and finally the SUT will be terminated.

We know the test run details can be looked up in the run log.



Figure 11.14: The Run log of the test suite

In the first tutorial chapter we've already learned how to use the run log for error analysis.

# Chapter 12

# Writing a Procedure (Web)

### Video This chapter is also available as a video tutorial at



Action

"Writing a Procedure" https://www.qfs.de/en/yt/tutorial-3.html

In chapter one and two you learned how to start an application from QF-Test thus being able to record mouse and keyboard actions, add checks and organize the result in a test case. This approach is fine and sufficient as long as your tests are simple and you have just a few of them. But as soon as the number of tests increases it is important to make use of so called 'procedures'.

Procedures make sequences reusable and therefore avoid duplicated identical parts. This is important for easy and on the long run efficient maintenance of your tests.

Procedures can be organized into packages = . Procedures and packages are the basis for modularizing your tests.

## 12.1 Identifying reusable parts

In this chapter we will work with the test suite FirstJavaTests.qft you already know from chapter one.

- Copy FirstWebTests.qft from the subdirectory qftest-7.1.3/doc/tutorial
  of the QF-Test installation to a working directory and
  - open FirstWebTests.qft.
  - If you want to keep the changes you will be making to the demo test suite **save it** in a working directory as described at the end of section 11.1<sup>(120)</sup>.

Please have a look at the test step "Reset" in the two test cases. They are exactly the same.

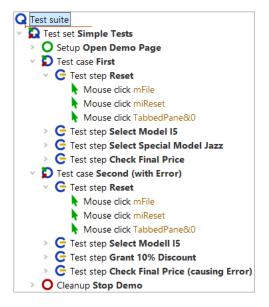


Figure 12.1: Two identical test steps

Following above concepts it would be a good idea to turn them into a procedure.

# 12.2 Manual creation of procedures

There is more than one way to create procedures and insert procedure calls. We will start with the manual one by inserting an (empty) procedure node and moving the respective actions into it. Then we will create the respective procedure call.

It is good to know those basic steps but there is a second more elegant way of creating procedures, which we will explain afterwards.

Okay, let's do it by hand. We will start with creating a procedure node and naming it appropriately.

Open the Procedures node and keep it selected.

Action

- Chose Insert $\rightarrow$  Procedure nodes $\rightarrow$  Procedure.
- Name it 'reset'. The other fields can be left empty.
- Press the **OK button** to finalize the creation of the procedure.

Action

· Open the newly created 'reset' procedure node.

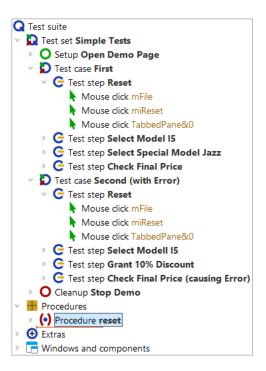


Figure 12.2: Create a procedure node

The second step is to fill the procedure with the respective reusable actions.

- Select the three 'Mouse click' nodes in the test step. To select more than one node at once you can select the first one, then press the Shift key and, while keeping it pressed, click the last node you want to select.
  - Move them down into the procedure e.g. by mouse (drag&drop) or cut/paste from the Edit or context menu.

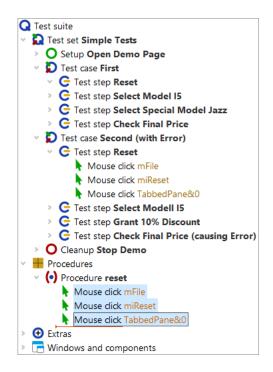


Figure 12.3: Fill in the procedure content

The third step is to add a procedure call to the place of the three 'Mouse clicks' you moved.

- Action
- Select the test step 'Reset', which is still open.
  - Select Insert $\rightarrow$  Procedure nodes $\rightarrow$  Procedure call or use the Ctrl-A shortcut.

Procedure call		х
(•) Procedure name		^
Variable for return value		
Local variable		
🛨 🦯 🗙 🕂 🕂 Variab	le definitions	
Name Value		
QF-Test ID		
Delay before (ms)	Delay after (ms)	
Comment		
		~
ОК	Cancel	

Figure 12.4: Insert a procedure call

- On the dialog **press** the procedure selection button (•) left of the label 'Procedure name'.
  - Select 'reset' from your test suites procedures. Other fields can be left as they are.
  - Press the **OK** button on both dialogs to finalize the creation of the procedure call.

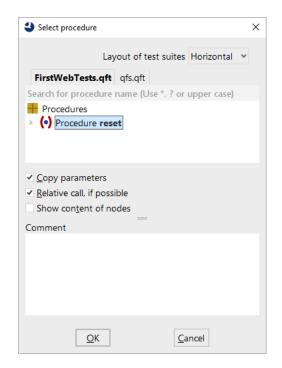


Figure 12.5: Select a procedure

In order get a real benefit from the procedure, of course, we also need to replace the content of reset test step in the second test case by the 'reset' procedure call, too.

You can do it the same way as before or use the following **alternative steps** to create a procedure call:

- **Open** the 'Reset' test step of the second test case.
  - Remove the three 'Mouse click' action nodes therein.
  - Select the 'reset' procedure node.
  - **Move** it via drag&drop into the 'Reset' test step (copy/paste action can be used alternatively). This does not actually move it but create a respective procedure call.

The test suite should look like this:

Action

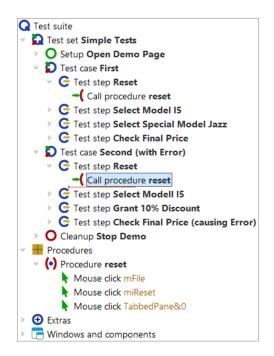


Figure 12.6: Test-suite with procedure

When now executing the test cases the reset in meant to still work like before. Hence, in the run log you will see the same executed node as before, only preceded by the procedure call.

## 12.3 Transforming nodes into procedures

As already mentioned at the beginning of the last section, QF-Test offers an alternative, much faster way to create a procedure:

Action

- Select the test step or sequence node that contains the reusable steps to be transformed into a procedure.
  - Select menu item **Operations**→**Transform node into**→**Procedure** or use the Ctrl-Shift-P shortcut.

You will find that the test step, respectively sequence node disappeared and there is a procedure call in its place. Moreover, a procedure was created in the Procedures section containing the child nodes of the former test step / sequence node and named the same.

It is good practice with QF-Test to record a sequence, give it a name and immediately turn it into a procedure via <u>Ctrl-Shift-P</u> if you suspect it to be of use somewhere else, too.

# Chapter 13

# **Components (Web)**

Let us have a look at the last main section in the test suite window, the Windows and components node. When talking about components we also want to show you how to address subitems of components like tables, trees and lists.

Video This chapter is also available as a video tutorial at



"Components" https://www.gfs.de/en/yt/tutorial-4.html

## 13.1 Addressing subitems of tables, lists and trees

Subcomponents of tables, lists and trees can be addressed by indices. There are two main types: textual and numeric indices. Let's record a mouse click to a table cell and analyze the recorded QF-Test component ID.

Action

- Start the CarConfig application, in case it has not been started already, by executing the Setup node of the test suite.
- Activate the recording mode by pressing the toolbar button 🛡 .
- Click a table cell, e.g. the first model.
- Stop the recording by pressing

You will see the recorded mouse click in the Extras section.

Sequence Recorded 17/04/2024, 16:37
Mouse click VehicleTable@Model&0

Figure 13.1: Addressing a table cell

The QF-Test component ID recorded is VehicleTable@Model&0. The single parts are:

- VehicleTable is the QF-Test component ID of the table component itself.
- @ and & separate the single parts from each other and at the same time indicate the type of index that follows: @ for a textual index and & for a numeric index.
- Model is the textual index for the table column with the title 'Model'.
- 0 is the numeric index for the first table row.

#### Note Numeric indices always start at 0.

You can use any index type for column or row. It is just important that the separator and the index type match.

Action • Change the QF-Test component ID so that the third price value in the table will be clicked, using numeric indices only.

The solution is to type <code>VehicleTable&1&2</code> in the QF-Test component ID attribute of the Mouse event node.

In order to address the model 'I5' using textual indices only, you would have to enter VehicleTable@Model@I5. Using numeric indices you would write VehicleTable&O&4 and for mixed indices VehicleTable&O@I5 or VehicleTable@Model&4.

The third type of index QF-Test supports is a textual index containing a regular expression. Regular expressions can be used to replace a string by an expression that can match for more than only one string. For a detailed explanation of regular expressions please refer to the manual. So you could also address the cell for the model 'I5' using VehicleTable@Model%I.\*.

Lists are addressed the same way as tables are, just with one index only.

A tree has only one index. This is the path down the tree to the node to be specified. The path consists of the respective nodes of the tree, separated by slashes ('/'). Let's record a mouse click to a tree:

- Action Start the CarConfig application, in case it has not been started already, by executing the Setup node of the test suite.
  - Navigate to the tree: In the CarConfig application select the menu item Options -> Specials...], select a model and press 'Details'
  - Activate recording mode by pressing the toolbar button
  - Click a tree node, e.g. 'Description'.

### • Stop the recording by pressing

For the tree node 'Description' the recorded QF-Test component ID would be DetailsTree@/Information/Description. The single parts are:

- DetailsTree is the QF-Test component ID of the tree component itself.
- @ separates the QF-Test component ID of the tree from the index and at the same time indicates the type of index that follows: @ for a textual index.
- /Information/Description is the textual index for the tree path to 'Description'.

If you wanted to address the node using a numeric index you would have to use  $DetailsTree \sqrt[6]{0/1}$ .

### 13.2 Web Component Recognition

Addressing subitems of complex components like lists, tables or trees via index, as described in the previous section, is only possible because of a suitable component mapping in the Setup. That way, QF-Test knows in what way certain functionalities like text fields, buttons, checkboxes, data tables, trees etc. have been implemented in the HTML code of the application.

Knowing the functionality of a component at recording time has the following advantages:

- Subitems will be addressed via index, as you already know. They will not be recorded as a separate component.
- With mouse clicks, the position where to replay the click to later on, will be optimized: the best position for a click to a button is its center, whereas with text fields it is important to click at the same position as during recording, since maybe you want to insert text at exactly that place.
- For check recording QF-Test provides a context sensitive popup menu offering various checks, depending on the functionality of the component: for data tables you will be offered checks for the whole row or column, for example. For text fields, on the other hand, you will find a check whether it is editable.
- QF-Test will save additional criteria for component recognition when it knows the funcionality of the component: with a button, for example, its text would be a relevant information for recognition. With text fields, however, the text contained would be quite useless in that respect. So QF-Test will look for a suitable label component and save that text.

#### 13.2. Web Component Recognition

Last but not least, the additional information QF-Test is able to gather, knowing the functionality of the component, will contribute towards the stability of the tests.

Configuration of the component recognition is required because Web applications have a limited set of basic components, which can be combined in flexible ways in order to implement complex components and designs. HTML tables, for example, are used, both, for just controlling the page layout and for displaying logical tables, like the VehicleTable. Without additional information, QF-Test cannot know where exactly the layout table ends and the logical table starts. This is why, without mapping, QF-Test would record something like VehicleTable.td for a table cell. The "td" coming from the HTML table data element "TD" and indicating a table cell, but nothing more. QF-Test would not able to assign a logical table row or column.

In the Setup of the demo suite FirstWebTests.qft of the first chapter or the one created via the quickstart wizard in the second chapter the component mapping is done right after the start of the browser. The following figure shows the resolver registration node in the FirstWebTests.qft.



Figure 13.2: Web resolvers as installed in the FirstWebTests.qft

The node inform QF-Test which HTML element contains the data table und which HTML elements would be the data row and the data cell, among other mappings. The configuration was done specifically for the "CarConfigurator Web".

Note At this stage it is just important for you to know about the option to improve component recognition by the use of resolvers.

QF-Test automatically analyses the structure of web applications, detects the most common web UI toolkits (like Angular Material, ExtJS, GWT, SmartGWT, Vaadin,...) and installs the respective predefined resolvers.

Obviously, not all web applications have been developed using a toolkit, some use extended toolkits and not all existent toolkits have predefined resolvers with QF-Test. Hence you may come across custom component implementations.

So when you start testing your own web application and encounter issues with components not being accessed as reliably as expected, there might be the time to at least think about a resolver to improve the recognition behavior. Please get in touch with our support (support@qfs.de) then for assistance.

There is also a chapter in the technical reference part of the manual with more in-depth information about CustomWebResolvers and how to start an implementation on your own if you wish to do so.

### **13.3** Windows and components Section

The topic 'components' is covered by several videos:

Video



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:

The video



'ComponentNotFoundException - simple case'

https://www.qfs.de/en/yt/componentnotfoundexception-simple-40.html

shows a simple case.

A more complex case is discussed in the video



'ComponentNotFoundException - complex case' https://www.gfs.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

QF-Test stores the information about how to find a component in the UI of the SUT in the Windows and components section. It analyzes the component information during the recording of actions to the SUT and saves the information for the components the user interacted with in the details of the Component nodes.

This section is meant to give you an idea about which kind of information is stored in the Component nodes and how QF-Test uses it to recognize a component in the UI. There are two cases where you will want to have a look at components and where it will be useful to have a basic idea of the concepts of component recognition.

The first one is when the UI of your application changed significantly between when you

#### **13.3.** Windows and components Section

first recorded the component to when you want to replay the test, e.g. because of a new release of the application. QF-Test uses several algorithms to evaluate whether a certain UI element currently displayed matches the criteria of the UI element that was once recorded. However, when too many of the criteria no longer fit then you will have to have a look at the components. For a detailed instruction about what to do then, please refer to the manual chapter Troubleshooting component recognition problems. There are also links to videos showing respective samples.

The second one is when you want to improve component recognition. As you already know from the chapter <u>Web Component Recognition<sup>(144)</sup></u> a certain functional component like a checkbox or table, even an input field, can be implemented in quite a number of different ways from the basic HTML components available. Even though QF-Test provides component recognition for a number of component libraries there may always be some components that are not recognized for what they really are. For details about how to implement additional component recognition please have a look at the chapter 'CustomWebResolver' in the technical reference part of the manual.

Let's have a look at a 'TextField' component node and see which criteria were recorded and saved in the details of the Component node.

- Start the CarConfig application, in case it has not been started already, by executing the Setup node of the test suite.
  - Open the procedure 'Check final price'.
  - Navigate to the text field Component node by right-clicking the text check node and selecting Locate component or using the Ctrl-W shortcut.

[FirstWebTests.qft] * QF-Test			X
Eile <u>E</u> dit <u>V</u> iew <u>I</u> nsert <u>O</u> perations <u>R</u> e	cord Run Debugg	er Clients Extras Help	
		🗸 🗖 (•) 🕨 🗏	ವ್ ನಿ <b>ಲೆ</b> ೮ ರೆ
*FirstWebTests.qft ×			
Test suite *FirstWebTests.qft		Check text	
Test suite     Test set Simple Tests     Procedures		Client \$(client)	
<ul> <li>(•) Procedure checkFinalPrice</li> <li>(•) Procedure reset</li> <li>(•) Procedure reset</li> <li>(•) Extras</li> <li>(•) Windows and components</li> </ul>	C <u>u</u> t <u>C</u> opy <u>P</u> aste Delete Edit	Ctrl-X t ID Ctrl-C Ctrl-V Delete Alt-Enter	
	<u>M</u> arks	>	
	Start run <u>T</u> oggle disabled sta <u>B</u> reakpoint on/off	te Ctrl+Shift-D Ctrl-F8	
	Locate component	Ctrl-W	
	Highlight compone Show i <u>n</u> Inspector Add to boo <u>k</u> marks	nt Ctrl-D	
	Trans <u>f</u> orm node int Pack nodes	o > >	
	_ Additional nod <u>e</u> op	erations >	
	Insert node	>	
	Clean tree	Ctrl-Period	Cancel
	What's this?		Finished: No erro

Figure 13.3: Locating a component

This is going to take you directly to the 'TextField CalculatedPrice' node in the Windows and components section.

e <u>E</u> dit <u>V</u> iew <u>I</u> nsert <u>O</u> perations <u>R</u> ecord R <u>u</u> n <b>D D D D D O O O O O O O O O O O O O O O</b>						II ⇒ 8	<b>0 0 0</b> 0	3			
est suite *FirstWebTests.qft	c	omponent									
Test suite	Q	F-Test ID									
Test set Simple Tests     Procedures	C	CalculatedPrice									
<ul> <li>Procedure checkFinalPrice</li> <li>Check text \$(price) CalculatedPrice</li> </ul>	-	Class name TextField									
<ul> <li>(•) Procedure reset</li> <li>• Extras</li> </ul>		ame									
windows and components	C	alculatedPrice									
<ul> <li>Web page CarConfig</li> <li>HTML CarConfig.html</li> </ul>		eature									
<ul> <li>BODY CarConfigWeb</li> <li>MenuBar MenuBar</li> </ul>		CalculatedPrice									
See Menubar Menubar     See TabPanel TabbedPane     See Panel CalculatorPanel		S regexp ★ ✓ ★ ★ Extra features									
TextField CalculatedPrice		State	F	Regexp	Negate	Name					
> TextField DiscountValue		Ignore	۷			class	textfield				
> 🔄 Dialog dialog		Should match	۷			id	CalculatedP				
		Ignore	~			qfs:class	textfield				
		Ignore	~			qfs:genericclass	TextField				
		Should match	~			qfs:label	Final Price				
		Ignore	~			qfs:originalid	CalculatedP				
		Ignore	~			qfs:systemclass	textfield				
		OK					Cancel				

Figure 13.4: Component tree

Above component tree has only a few levels. This is because of the custom web resolver registered after the start of the application as explained in <u>Web Component</u> <u>Recognition<sup>(144)</sup></u>. When you have a look at the component tree recorded for the test suite you set up in <u>chapter 11<sup>(120)</sup></u> you will notice that there are a lot more DIV levels. These have been defined on the HTML page. They are not relevant for component recognition and make the tree rather big and confusing. Therefore, in the demo test suite, we suppress their recording with the CustomWebResolver.

Let's have a look at the properties of this component node used to identify the UI element.

Component									
QF-Test ID									
CalculatedPrice									
Cl									
Class name TextField									
rextrieid									
Name									
CalculatedPrice									
Feature									
CalculatedPrice									
¢									
S As regexp									
- <b>+</b> - <b>X</b> - <b>†</b>	+	Extra fe	atures						
State		Regexp	Negate	Nai	me		Value		
Ignore	~			class		textfield			
Should match	~			id		CalculatedPrice			
Ignore	~			qfs:class		textfield			
Ignore	~			qfs:generio	class	TextField			
Should match	~			qfs:label		Final Price			
Ignore	~			qfs:origina	lid	CalculatedPrice			
Ignore	~			qfs:systemclass		textfield			
Ignore	~			tag		TD			
٢							>		
Structure									
Class index					Class cour	nt			
5			6						
Geometry									
X			Y						
157					81				
Width					Height				
109					29				
-									
Comment									

Figure 13.5: Details of the Component node

The topmost attribute is **QF-Test ID**, which provides the 'handle' to the component to be used in the test cases and procedures. All other attributes refer to the component in the UI.

The next attribute is **Class name**. In our case it is 'TextField'. For component recognition it is essential to know the class of a component. In fact this class is a generalized value of the class used by the developer. This is helpful to achieve component recognition independent of the specific implementation and allows easy porting of tests. However, QF-Test saves the specific values in the Extra features table. By default, they are not

used for component recognition.

Action

Other examples for generic classes would be 'Panel', 'Dialog' or 'Button'.

The Name attribute is the name or id given to the UI component by the programmer. If there is a Name then this, together with the class, is all QF-Test needs to identify a component in the UI.

If the programmer did not set a name or id for the UI component and the Name attribute therefore remained empty QF-Test needs other criteria like a certain text associated with the component, index information and geometry.

A feature associated with a button for example would be the text shown on the button. QF-Test saves a text directly belonging to the UI component in the Feature attribute. If the web UI element has not text but an id QF-Test saves the id in the Feature attribute. If none of the two exists and there is a text in the vicinity of the component that could be the label then this will be saved in the Feature. In any case the label will be saved in the Extra features Name <code>qfs:label</code>.

The **Structure** information refers to all UI components of the respective class. The total number of UI components of that class is saved in the Class count attribute, the index of the component itself in the Class index attribute.

Last there is the **Geometry** information. It is the one having the lowest weight within the recognition algorithm. It can be of value if no other helpful information is available for a component.

In case you are interested in further details of the component recognition you will find such in the Component recognition chapter of the technical reference of the manual.

If you want to get a feeling for the component recognition you could play around with the attribute values and see what you need to change to make QF-Test not recognize it anymore or even recognize a wrong component in the UI. You will find that you need to change quite a few attributes before QF-Test recognizes a different component. This means that component recognition with QF-Test is very robust. With regression tests a significant part of the UI component criteria need to change before QF-Test will not recognize the component anymore even if the component has no name or id.

When you click the Component node QF-Test will highlight the recognized component in the UI by outlining it with a dark blue border.

- **Delete the CalculatedPrice from the Name attribute**. Because as long as there is an entry in the Name attribute QF-Test will not consider the attributes below.
  - In the Extra features table, set the state of the extra feature with the name id from Should match to Ignore.
  - Change the Feature attribute from CalculatedPrice to xxx.

#### 13.4. SmartIDs - Addressing components directly

- Click the TextField node and check QF-Test still highlights the Final Price field in the UI.
- Change the Feature attribute back to CalculatedPrice, either via the respective toolbar button or by typing <u>Ctrl-Z</u>.
- Change the qfs:label value in the Extra features table from Final Price to Accessories Price.
- Click the 'TextField' node and check QF-Test still highlights the Final Price field in the UI.
- Change the qfs:label value in the Extra features table back to Final price.
- Change all Structure and Geometry attributes to a different value and check that QF-Test still highlights the Final Price field in the UI.
- Delete the Feature attribute and change the qfs:label value in the Extra features table from Final Price to Accessories Price.
- Check that now QF-Test highlights the Accessories Price field.

This is just to give you a bit of a feeling for component recognition. In the above mentioned chapters (and some more) of the manual you will find detailed information about what to do when you have trouble with component recognition.

## **13.4 SmartIDs - Addressing components directly**

Since version 7.0 QF-Test officially supports SmartIDs allowing you to reference a component without having to record a Component node.

For some applications SmartIDs may reduce the time required for management and maintenance of the component information considerably.

SmartIDs may also have a positive effect on readablity and maintainability of the tests themselves.

It also allows you to write tests without use of the recording functionality, for example when a component or even the whole application does not yet exist, and you want to go ahead with tests ("test first" approach). SmartIDs then allow you to specify the future recognition critiera directly.

The recognition criteria available are the component class, the name or a label and the index. The values are the same as in the recorded Component node. Nested SmartIDs even reflect the component hierarchy.

The SmartID is used in place of the QF-Test ID of the component. SmartIDs start with the hash symbol # as the first character, in the simplest form followed by the name or the label of the component, for example:

- #btnOK, "btnOK" being the name of the component or
- #First name, "First name" being the label of the component.

The disadvantage of this simple form of the SmartID is performance at replay, as QF-Test has to search all components for the given criteria. For this reason it is better to specify the class of the component, too. Above SmartIDs would then be:

- #Button:btnOK
- #TextField:First name

Currently, the recording of SmartIDs is not activated by default. You can activate it directly via the menu.

Now, please record some SmartIDs by

- Action
- opening the menu **Record** and selecting **Record SmartID**.
  - Activate recording mode via the button "Start recording"
  - Click to a text field, for example the one for the discount percentage, then
  - click to a table cell, for example the first model.
  - Stop the recording via "Stop recording"

You will find the recorded mouse clicks in the Extras section.

The SmartID for the input field is *#TextField:name=DiscountValue*, because QF-Test preferably uses the name for the SmartID value when there are a name and a label for the component.

The second mouse click shows the SmartID for the table component, followed by @ and the text index of the column and & with the numeric index of the row, just like you saw it in the one but last section: #Table:name=VehicleTable@Modell&O.

- Action
- When you do not want to record SmartIDs any more you can deactivate it via the "Record" menu.

Of course, you can continue working with SmartIDs. When recording more SmartIDs you will come across more expressions used in SmartIDs. We now want to show you some of them and give you an explanation why they are used.

For labels ending with a colon you will find a back slash in front of the colon:

• #TextField:First name\:

The explanation is the colon defines the end of the name of the class. Therefore, colons in the component name or the label have to be escaped by a back slash. Other characters like @, & and % have to be escaped as well because they are used as separators before indices.

Labels will be prefixed by a qualifier, here left=:

• #TextField:left=First name

Explanation: a SmartID consisting of class and name of the component reaches the same performance at replay as does component recognition via a recorded Component node. Labels, however, are a different case. There are various options for what might be the best label for a component. QF-Test searches all the available labels of a component for the best one. For performance it is good when you specify the type of label expected.

left= indicates the label to the left of the component. Other SmartID qualifiers indicating the position of the label are right=, top=, topleft= or bottom=. When the text of the component serves as label the qualifier is text=, for the tooltip tooltip=.

When you get several components with the same name or label on a display you might see a compound SmartID consisting of to single SmartIDs concatinated by @:

- #TitledPanel:Customer address@#TextField:left=First name
- #TitledPanel:title=Invoice address@#TextField:left=First name

The example shows the SmartIDs for the label "First name", both on the panel "Customer address" and on the panel "Invoice address".

The second sample is a bit more performant than the first one because of the qualifier title= in the SmartID for the panel. With the first one readablity is a bit better. It depends on the application whether you have to pay attention to performance. With web applications with many UI elements it is usually an issue. However, with Java applications you can opt for readablity in most cases.

Long labels can be shortened by the use of regular expressions:

• #%Dialog:Information.\*@#Button:OK

The percent sign directly after the hash sympol indicates the name or the label to be a regular expression. In above sample the regular expression shortens the title. It may also be used to identify an "OK" button in any dialog with the title starting by "Information". For more details about regular expressions please see the manual - regular expressions. You will find more information about components and SmartIDs in manual - components.

# Chapter 14

# **Using the Debugger (Web)**

In this chapter we will learn how to run a test suite with QF-Test's built-in and intuitive debugger. If you are familiar with debugging from other IDEs like Eclipse, you will find this debugger similar in function and usefulness.

Video This chapter is also available as a video tutorial at



"Using the Debugger" https://www.qfs.de/en/yt/tutorial-5.html

By the end of this chapter you will be familiar with the following debugger functionality:

- Setting a Breakpoint<sup>(156)</sup> e.g. via Ctrl-F8.
- Pausing a test run at any time and resuming operation using the debugger button
   or the "Don't Panic" key (Alt-F12).
- Stepping Through a Test or Sequence<sup>(157)</sup> using the debugger buttons 'Single step'
   Step over' <sup>O</sup> and 'Step out' <sup>O</sup>.
- <u>Skipping Execution of Nodes<sup>(159)</sup></u> using the debugger buttons 'Skip over' <sup>♥</sup> and 'Skip out' <sup>♥</sup>.
- Error or Exception triggering Debugging Mode<sup>(161)</sup>.
- Resolving Errors directly from the Run log<sup>(162)</sup>.
- Jump directly to the current error in the run log via Ctrl-J. (Jump to run log in chapter section 14.5<sup>(162)</sup>).
- Note Instead of the debugger buttons you can also enter the commands via the QF-Test menu and most by keyboard shortcuts as well. You'll find the the shortcut listed beside the menu option, if available. For a complete list, refer to the Keyboard shortcuts section

of the user manual. You can also find a little helper there for attaching to your keyboard which shows the function key assignment of QF-Test.

There are some more functions related to the debugger that we will come to in later chapters:

- Locating the current node using the debugger button 

   Icocate the current node in chapter section 15.3<sup>(174)</sup>.
- "Continue execution from here" via the popup menu of the respective node (figure  $15.9^{(176)}$ ).
- Rethrowing exceptions using the toolbar button  $\checkmark$  .
- The variable bindings table (section 15.3<sup>(174)</sup>).

## 14.1 Setting a Breakpoint

First of all we need to activate debugging mode. There are several ways to do so. One of them is to set a breakpoint at the node where we want to have a closer look. When the test is being executed and QF-Test comes to the break point it will then pause and switch into debugging mode. The pause button III will then be activated.

• Select a node and **press Ctrl-F8**. The breakpoint is indicated by a **B**.



Figure 14.1: Set break point

• Select the Test suite node and **press Enter** to start the test run.



Figure 14.2: Start test run

- Action
- Remove the breakpoint by pressing Ctrl-F8 again.



Figure 14.3: Remove break point

Instead of using the keyboard shortcut Ctrl-F8 you may also set or unset a breakpoint by clicking the node and selecting the Debugger $\rightarrow$ Breakpoint on/off menu item, or alternatively right-clicking the node and selecting the Breakpoint on/off context menu item.

Again, you can see the little arrow, which now marks the next node to be executed, called the **current node**. When entering debugging mode QF-Test also navigates to the current node, in case it had not been visible, and selects it, highlighting it blue.

The menu option Debugger $\rightarrow$ Clear all breakpoints is useful to remove all breakpoints set in your test suite.

There is no limit to the number of breakpoints you can set in your test suite, but note that breakpoints are not saved with the test suite.

## 14.2 Stepping Through a Test or Sequence

Now let's step through the test case we set up in the previous section.

Action

Please try out the debugger buttons Single step <sup>™</sup> , Step over <sup>™</sup> and Step out <sup>™</sup> .

You will find that **Single step**  $\mathfrak{D}$  opens a node containing child nodes and makes the first child node the active node. Continuing from where we left the test suite at the end of the last section, i.e. in debugging mode, with 'Test case: First' being the current node, the test suite would now look like this:

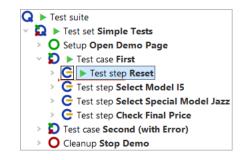


Figure 14.4: Stepping into a node

In the case of leaf nodes (nodes without child nodes), the effect of  $\mathfrak{D}$  is the same as the following button's.

Step over  $\bigcirc$  runs the current node including all children. Execution pauses at the next node of the same level to be executed, which then becomes the active one.

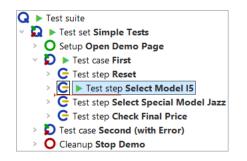


Figure 14.5: Stepping over a node

**Step out**  $\checkmark$  runs the remaining nodes at the same level including their child nodes. Execution pauses when a node that is higher in the hierarchical structure is found, which then becomes the active one.

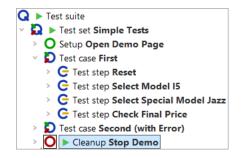


Figure 14.6: Stepping out of a node

In the given example the node higher in the hierarchical structure where execution stops is the Cleanup node. As explained in the chapter <u>A full Test Run<sup>(116)</sup></u> this shows the special behavior of Setup / Cleanup nodes in a test set: They are executed before and after **each test case** to help achieving a proper starting state for each test case.

- Note You will only find this behavior when you started the whole test suite or test set and are in debugging mode. If you just selected the test case and did a step-over action then QF-Test will execute the test case and then select the next test case node.
- Run the Cleanup and Setup nodes by stepping over them, using the debugger button **Step over** <sup>♥</sup> and then **Step in** the second test case via <sup>♥</sup> to get ready for the next section where you will learn about the skip functionality.
- Note Please be aware that menus or comboboxes tend to close when the application looses the focus, as will happen when activating the debugging mode. In such a case you should not stop test execution between the node opening the menu or combobox and the node performing the selection. One way to do achieve this is to set a break point **B** after the node performing the selection and to activate normal test execution by releasing the pause button **II** when you reach the node opening the menu or combobox.

# 14.3 Skipping Execution of Nodes

The "skip" functions expand the QF-Test debugger's capabilities 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 them. This may be helpful for various reasons, e.g. to quickly navigate to a certain position in your test run or to skip a node which currently leads to an error.

With the end of the last section, the test suite should have reached the following state:

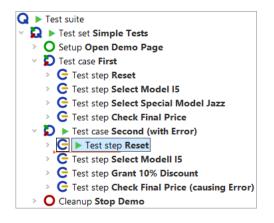


Figure 14.7: Pause execution at first node of the second test case

### 14.3. Skipping Execution of Nodes

Action • Now press the Skip over <sup>♥</sup> button. QF-Test simply jumps over the active node without executing any child nodes. The active node now is the next node to be executed on the same level.

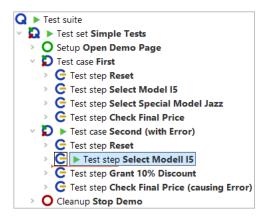


Figure 14.8: Skip over a node

• At last, press the **Skip out** <sup>𝒞</sup> button. You see that QF-Test skips all nodes on the same (or lower) level and directly jumps to the next node one level up in hierarchy.

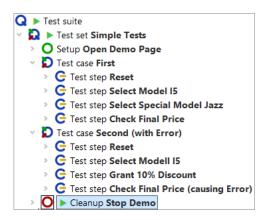


Figure 14.9: Skip out of a node

Note Use "Skip over" and "Skip out" cautiously as skipping out of a sequence before it is completed can leave the SUT in an unknown state that other sequences or tests in your test suite cannot react to.

Action

# 14.4 Error or Exception triggering Debugging Mode

When debugging a test you may want run it until it encounters an error, an exception or sometimes even a warning and then have it pause in debugging mode.

In this section you will see how this can be done while debugging the second test case.

- Please open the debugger menu and change the default options as follows:

  - Also activate the Debugger→Options→Break on error menu item.

Afterwards, when you open the debugger menu and options submenu it should look like this:

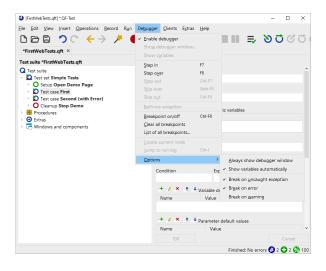


Figure 14.10: Set debugger options to pause on error

We changed the debugger options because with default settings QF-Test will not pause on exceptions or errors, as you saw earlier on.

Action • Select the "Test-suite" node and start test execution via "Start test run" 🕨 .

QF-Test will pause at the faulty node and enter debugging mode:

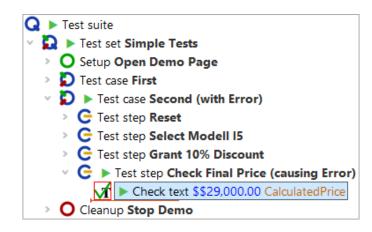
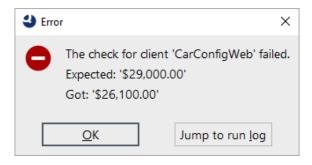
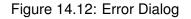


Figure 14.11: Test run stopped by error

The node which halted test execution will be indicated by an arrow and its symbol will be surrounded by a red square. Also, an error dialog will inform you about the failed check.

As always the run log is the key to resolving errors, so let's open it and find out how to resolve the error in the next section.





Action

• Click the Jump to run log button in the error dialog.

# 14.5 Resolving Errors directly from the Run log

The **Jump to run log** button from the dialog in <u>figure 14.12<sup>(162)</sup></u> will not only open the run log but takes us directly to the node that holds the error details. Apart from the actual error message you will find screenshots and a copy of the variable bindings table (stack trace), which we will introduce later on (The Variable Bindings table<sup>(170)</sup>).

The error details tell you that the expected value does not match with the one shown in

### 14.5. Resolving Errors directly from the Run log

the application. As the one in the application is correct we want to update the expected value with the one from the application. This can easily be achieved as follows:

- Action
- Right-click the red-bordered node "Failed: Check text: default ..." indicating the actual error
  - Select Update check node with current data from the context menu.

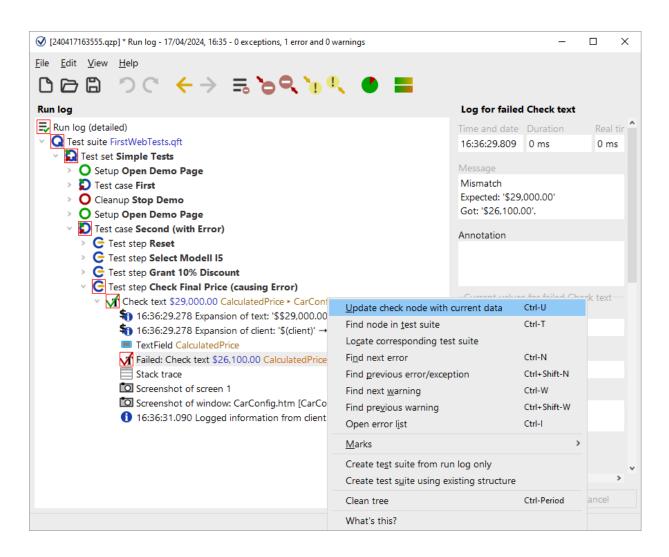


Figure 14.13: Update check node with current data

This locates the corresponding Check text node in the test suite and updates the expected value of the Text attribute with the value got as indicated by the run log.

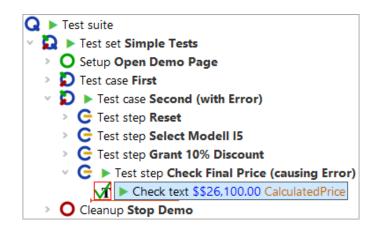


Figure 14.14: Corrected check node

The previously faulty node still is highlighted with a red border since we have not run it again.

### • Now continue execution by releasing the pause button II.

QF-Test runs the rest of the test suite, i.e. the Check text and Cleanup nodes, and informs you at the end of the run that there was one error, which you have already fixed.

Since the error has been fixed and we will continue using the test cases as examples you could rename the second test case and delete '(with error)' in its name as well as '(causing Error)' in the name of the test step.

**Jump into run log:** You do not have to wait for an error dialog to open the run log at the current point of execution (or close by). Whenever you are in debugging mode, select the **Debugger** $\rightarrow$ **Jump to Run log** menu option, or use the **Ctrl-J** shortcut. If you just want to open the run log without jumping to the current point of execution you can use **Ctrl-L**. This will work after the test run finished, too.

# 14.6 Pause Execution

When a test is being executed and you want to enter debugging mode you can quickly set a breakpoint at some node not yet executed. Or you can just hit the toolbar button "Pause" III and QF-Test will directly enter debugging mode.

In order to resume execution just release the pause button **II**. This is completely independent of the way you entered debugging mode.

Depending on how focus demanding the SUT is, it may be difficult to focus QF-Test long enough to hit the pause button. But you can still rely on the "Don't Panic" short-

cut <u>Alt-F12</u>. It will pause all running tests immediately. To continue, press the same combination again.

# **Chapter 15**

# Variables and Procedure Parameters (Web)

In this chapter you are going to learn how to use a procedure to perform the same action on various input data. You will also learn about variables - how to use and how to debug them.

Video This chapter is also available as a video tutorial at



. The video uses a Java application for demonstration. Variables are used in the same way with web applications

### **15.1 Procedure using a variable**

Have a look at the last test step 'Check final price' of our two test cases.

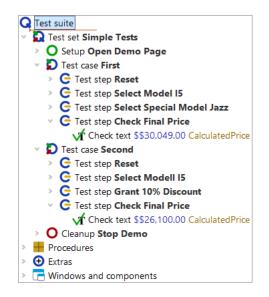


Figure 15.1: Two almost identical test steps

They perform the same action, however, with different data. Even though it is only one node, it makes sense to pack it into a procedure. We may want to adapt the hard coded values  $30,049.00 \in$  and  $26,100.00 \in$  to a different format so that the check will also work when the format of the price field changes to a different currency. And we do not want to implement the same algorithm twice.

• Select the 'Check text' node in the first test case.

Action

- Use **Operations**→**Pack nodes**→**Sequence** or use the <u>Ctrl-Shift-S</u> shortcut to pack it into a sequence.
- Name the sequence checkFinalPrice. The procedure name follows the Java convention to run the words together and start the single words with capital letters. On the other hand QF-Test allows the use of spaces in procedure names, so you are free to name it as you like.
- Press **Ctrl-Shift-P** for the quickest way to transform the 'Sequence' node into a procedure, as you learned at the end of the last chapter. You see the sequence is replaced by a call to the 'checkFinalPrice' procedure.
- **Double click** the procedure call node to jump to the procedure definition.
- **Open** the procedure node to see its content.

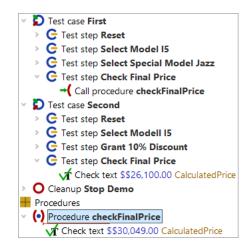


Figure 15.2: Procedure with hard coded value

As expected, the check is now located in this procedure. However, it is valid for one price only, i.e. 30,049.00  $\in$ . Since we want to use the same procedure for the second test case as well we need to make the price a variable and pass its value as a parameter from the test case to the procedure.

In the next example we will start by defining a parameter for the procedure. Additionally, we will set a default value for the parameter. Default values are most useful when the parameter usually has that value and you do not want to specify it every time you call the procedure. Even though this does not hold true for the price it is a good example to demonstrate to you how a default parameter works and how to overwrite it with another value.

Let's define the parameter and add a default value:

### Action • Select the procedure 'checkFinalPrice'

- **Press** the <sup>d</sup> "Add new row" button belonging to the table 'Parameter default values'.
- Enter price as name for the parameter.
- Enter 30, 049.00 € in the value field.
- Click the 'OK' button.

Procedure			
Name			
checkFinalPrice			
+ / × + +	Parameter	default values	
Name	Value		
price	\$30,049.0	0	
Maximum error leve Exception QF-Test ID	2		~
Delay before (ms)		Delay after (ms)	
Comment			

Figure 15.3: The Details of the 'Procedure' node

The next step is to replace the value of the Text attribute of the Check text node by a reference to the variable.

Note Variable syntax: When working with variables you need to bear in mind that in some places you need to tell QF-Test the name of the variable and in others you want to refer to the value of the variable. In the Name column of the Parameter default values table of the Procedure node QF-Test expects the name of a variable. It is price, which is why you typed the word price.

In the Text attribute of the Check text node details QF-Test expects a character string for comparison with the text of the UI element. As we want to use a value stored in a variable, we have to tell QF-Test not to use the entered string as plain text, but to interpret it as a reference to the value stored in a variable. In QF-Test this is done by enclosing the variable name in \$(). In our case the variable reference is \$(price). If you did not put the variable name into \$(), QF-Test would compare the price (a number) shown in the UI element to the string price, which is obviously nonsense.

- Select the Check text node in the procedure 'checkFinalPrice'.
  - **Type \$ (price)** in the Text attribute of the Check text node details.
  - Click the 'OK' button of the node details.

Action

Check text	
Client	
\$(client)	
C QF-Test component ID	
CalculatedPrice	
Text	
\$(price)	
\$ As regexp	
Segate	
Check type identifier default	
Timeout	
Result handling	
Variable for result	
Local variable	
Error level of message	
Error	~
\$	ure
Name	
QF-Test ID	
Delay before (ms)	Delay after (ms)
Comment	
Comment	

Figure 15.4: 'Check text' node

### Action • • Run the first test case.

It should execute without an error.

# 15.2 The Variable Bindings table

The next step is to make use of the procedure call in the second test case as well.

### 15.2. The Variable Bindings table

Replace the Check text node of the second test case by a procedure call to checkFinalPrice. You can simply copy the respective node from the first test case or add the procedure call as learned in the previous chapter.

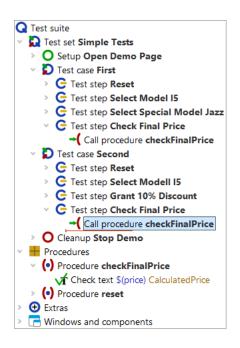


Figure 15.5: Second procedure calls 'checkFinalPrice'

- Note If you added the procedure call by a copy or drag and drop operation from the procedure itself you will find the price in the Variable definitions table of the procedure call. This is what we are eventually aiming at. However, at this stage we want to explain the default value. So, if you want to follow the tutorial exactly, please delete the default value by pressing the red X above the table.
- Verify **QF-Test is configured to pause at errors** as shown in <u>Set debugger</u> options to pause on error<sup>(161)</sup>.
  - Select the 'Test case: Second' node.
  - Execute it by pressing or Enter.

An error message shows up indicating different values for the price expected and the price got. What went wrong? Let's go hunting. Typically we use the run log for this but there is another view worth to know of.

• So click OK to close the error message.

### 15.2. The Variable Bindings table

In debugging mode you will find an additional bottom right section of the QF-Test window showing a list of nodes with variables bound to those nodes.

- Action
- You might want to resize the variable bindings table in case it is too small to see all its content.

Variable bindings				
Node	Test suite	Bindings	+ / ×	
(•) Procedure checkFinalPrice	FirstWebTests.qft	0	Name	Value
Call procedure checkFinalPi	FirstWebTests.qft	0	price	\$30,049.00
G Test step Check Final Price	FirstWebTests.qft	0		-
D Test case Second	FirstWebTests.qft	0		
Globals		3		
Command line		3		
🔾 Test suite	FirstWebTests.qft	3		
Fallback stack		0		
• Procedure checkFinalPrice	FirstWebTests.qft	1		
System		0		

Figure 15.6: Variable bindings

The variable bindings table is very useful for debugging. It comes in quite handy, too, when working with procedures and trying to understand the way QF-Test figures out which variable value to use. It shows the current values of the variables.

**Note** QF-Test always checks the variable bindings table from top to bottom.

You can see that the first rows of the table have no bindings at all. Then there is a binding at the level 'Globals' and another one in the fallback stack for the procedure 'checkFinalPrice'. The global variable is used for the client connection, which has been set when starting the application (cf <u>Starting the Browser<sup>(106)</sup></u>). The other variable is more interesting to us - only it has the wrong value.

The default value is intended to be used for the parameter if no value has been defined elsewhere. This is why we added the parameter to the 'Parameter default values' table of the procedure node.

To do things correctly we need to pass the proper value when calling the procedure. Again, there are several ways to do it. One is to add a new row to the variable definitions table of the 'Call procedure' nodes similarly to the way you did at the 'Procedure' node in the last section.

If the procedure is called multiple times within the test suite, there is a better way:

- Action
- Stop the current test execution clicking the toolbar button 🔳 .
  - Right-click the 'Procedure' node and select
     Additional node operations→Update parameters of references
     from the popup menu.

[FirstWebTests.qft] * QF-Test				– 🗆 🗙
	ations <u>R</u> ecord R <u>u</u> n De <u>b</u> u	gger <u>C</u> lients	Extras Help	
	$\leftarrow \rightarrow \not > \bigcirc \blacksquare$			000000
*FirstWebTests.qft ×				
Test suite *FirstWebTests.qft		Procedur	e	
Test suite     Test set Simple Tests     O Setup Open Demo Pa	ae	Name checkFinal	Price	
<ul> <li>Detrop open open open open open open open op</li></ul>	Cut Copy Paste Delete Edit Marks Start run Toggle disabled state Breakpoint on/off Locate references Add to bookmarks Pack nodes Additional node operations Insert node New window Expand all Collapse all	Ctrl-X Ctrl-C Ctrl-V Delete Alt-Enter Ctrl+Shift-D Ctrl-F8 Ctrl-I Ctrl-D > Alt-Right Alt-Right	Make calls relative Make calls absolute Make references explicit Make references implicit Update parameters of references Parameterize	
	Clean tree	Ctrl-Period		Cancel
	What's this?		Finished: 0 exceptions, 1 error, 0 warnir	ngs 🤫 1 🛟 1 🧭 10

Figure 15.7: Popup menu for 'Additional node operations'

• In the following dialog, please check the tick mark for **Add missing parameters to callers** is set and **click OK**.

In the 'Call procedure' nodes QF-Test adds a row each for every default parameter to the variable definition table of the procedure call. It also copies the default value of each default parameter. In our case it is the parameter price with the value 30,049.00  $\in$ .

You might notice that the numerical value of the price variable is still wrong in the second case, regardless of whether it is defined implicitly as a default value or explicitly via a parameter. For now we want to keep that error to show you additional means of debugging.

 Close the 'Updated nodes' dialog QF-Test opened to inform you about the updated nodes.

# **15.3** Advanced debugging of variable bindings

Next, we want to explore the variable bindings table and see how it can be used for debugging purposes. For this reason do not correct the faulty value of the procedure call we added in the last section, but let us find out more about debugging.

In the next steps we want to have QF-Test pause the test execution at the procedure call. Then we will step into the procedure and, while doing so, have a look at the variable bindings table. Finally, we will navigate directly from the variable bindings table to the faulty procedure call and correct the parameter value.

- Action Add a breakpoint to the 'Call procedure: checkFinalPrice' node of the second test case.
  - Run the second test case again.
  - When QF-Test stops at the breakpoint step into the procedure via <sup>™</sup> and watch the table of the variable bindings as you do so.

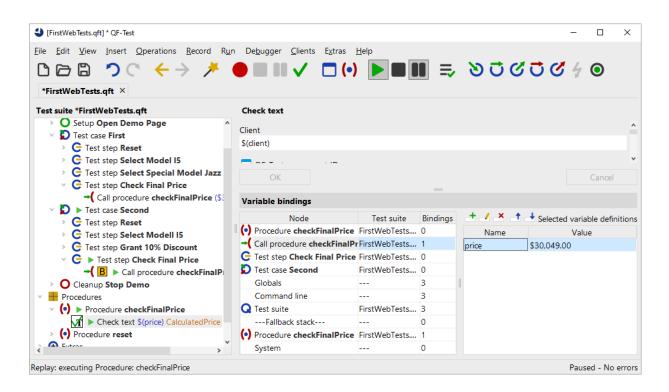


Figure 15.8: Variable bindings stack showing incorrect value

As you step into the procedure, first the row 'Call procedure: checkFinalPrice' and with the next step the row 'Procedure: checkFinalPrice' appear at the top of the table.

### 15.3. Advanced debugging of variable bindings

Now the variable price shows on two different levels of the variable bindings table: In the 'Call procedure: checkFinalPrice' row and the 'Procedure: checkFinalPrice' row on the fallback stack. Since we did not adapt the value of the parameter passed to the procedure, neither of the two values bound to the price variable is correct.

QF-Test lets you change the values of the variables interactively in the variable bindings table when you are in debugging mode. You can even add new variables or delete them. However, changes to variable values in the variable bindings table are not persistent. They only last as long as the variable is on the stack (variable bindings table). In our case, if we changed the price value, as long as the procedure is being executed.

The parameter value in the procedure call will not be altered by changing the current value of the variable in the variable bindings table. To do so you have to navigate to the Procedure call node and change it there.

To get there quickly, you just double-click the procedure call in the variable binding table (second row). This feature is particularly useful when debugging more complex tests where the node you want to jump to is not directly visible in the test suite window. You can invoke it via right-clicking the row and selecting Jump to node in test suite from the pop-up menu, too.

- Action Double-click the second row with the procedure call in the variable bindings table.
  - Set the value of the parameter to the correct value, i.e. 26, 100.00  $\in$ .

When checking the variable bindings table you will notice that the current value has not changed. This is hardly surprising as we have not yet executed the procedure call again. Only, test execution is already past the procedure call. Fortunately, QF-Test has another very useful debugging feature to set back (or forward) test execution to some node: **Continue execution from here**, which can be invoked either via the popup menu of the node you want to make the current node or by pressing **Ctrl**-, after selecting the node.

In order to try out the newly set value:

Action

- Right-click the 'Call procedure: checkFinalPrice' node of the second procedure.
  - Select 'Continue execution from here' in the popup menu.

[FirstWebTests.qft] * QF-Test							-		×
	tions <u>R</u> ecord R <u>u</u> n De <u>b</u> ugge	_	E <u>x</u> tr	_		. ⇒	৯ ট	5 C	Ū
Test suite *FirstWebTests.qft		Procedure	call	I					
est suite Test set Simple Tests O Setup Open Demo Page D Test case First		(•) Procedur checkFinalPr		ame					
<ul> <li>G Test step Reset</li> <li>G Test step Select Model</li> <li>G Test step Select Specia</li> <li>G Test step Check Final P</li> <li>Call procedure check</li> <li>Test case Second</li> <li>G Test step Reset</li> </ul>	C <u>u</u> t <u>C</u> opy <u>P</u> aste Delete E <u>d</u> it <u>M</u> arks	Variable for r Ctrl-X Ctrl-C Ctrl-V Delete Alt-Enter	>	🕂 Varia	able definitions /alue			Cancel	
<ul> <li>G Test step Select Model</li> <li>G Test step Grant 10% Di</li> <li>G &gt; Test step Check Fina</li> <li>Call procedure</li> <li>Cleanup Stop Demo</li> <li>Procedures</li> </ul>	Start run Toggle disabled state Breakpoint on/off Locate procedure Add to boo <u>k</u> marks	Ctrl+Shift-D Ctrl-F8 Ctrl-P Ctrl-D		<b>1gs</b> <sup>•</sup> est <sup>•</sup> irst 0 <sup>•</sup> irst 1 <sup>•</sup> irst 0	+ / × Name price	↑ → Sele \$30,0	Valu		nitior
<ul> <li>Procedure checkFinalP(</li> <li>M ► Check text \$(price) C</li> <li>Procedure reset</li> <li>Extras</li> <li>Windows and components</li> </ul>	Trans <u>f</u> orm node into Pack n <u>o</u> des Additional nod <u>e</u> operations		> > >	First 0 First 0 3 First 3	1				
windows and components	Continue execution from <u>h</u> ere Insert node Clean tree	Ctrl-Comma Ctrl-Period	,	0 First 1 0					
hanges applied	What's this?						Davia	ed - No	orro

Figure 15.9: Continue Execution from here

When checking the variable bindings table you will find that the top two rows have disappeared. This is because now we exited the procedure (even though 'backwards') and therefore the procedure call and all its variable bindings was taken off the call stack.

Action • Release the pause button II.

The test run should finish without error.

Note As the variable bindings table is very useful when looking for incorrect variable values you will also find a copy of it in the run log whenever an error or exception is logged. It is written to the subnode 'Stack trace' of the node causing the error, showing the variable values at the time the error occurred.

**Locate the current Node:** Sometimes during debugging you will navigate far away from the current node where execution stopped and eventually want to get back to it again.

The easiest way to do so is by pressing the "Locate Current Node"  $\bigcirc$  button or select the Debugger $\rightarrow$ Locate Current Node menu option to cause the debugger to "select" the current node.

# 15.4 Setting Variables

In addition to the methods you have already seen, variables can also be set as follows:

- Via the Set variable node,
- as the return value of a procedure,
- as the result value of QF-Test nodes like the Fetch text node, the Fetch geometry node, the Fetch index node and the Check node,
- in the 'Variable definitions' table of the 'Test-suite' node, the 'Test case' node, the 'Test step' node, the 'Sequence' node and others like the 'If' node or 'Loop' node,
- via command line parameters.

For information about the best place where to define a variable please refer to the next section.

You can insert a Set variable node via the menu item  $\boxed{\text{Insert} \rightarrow \text{Miscellaneous} \rightarrow \text{Set variable}}$ . In its details you can specify whether the value should be bound as a local or a global variable.

The following figure shows the details of a Set variable node. (You can find it as the first node of the Setup node.) It defines a variable named client. It is a global variable as the Local variable attribute has not been checked.

Set variable		
Variable name		
client		
Local variable		
Default value		
CarConfigWeb		
\$ Interactive		
Description		
Timeout		
lineout		
QF-Test ID		
QF-Test ID		
Delay before (ms)	Delay after (ms)	
Comment		

Figure 15.10: Details of the Set variable node

When you want to set a variable as the result of a procedure call you need to specify the variable name in the 'Variable for return value' attribute of the procedure call. Within the procedure itself you have to add a Return node with the value to be returned as the last node to be executed.

The next figure shows a theoretical example of a procedure which returns a value. The procedure fetches the discount value displayed in the SUT and returns it to the calling test case. There, the receiving variable is named Discount and declared as a local variable.

④ [FirstWebTests.qft] * QF-Test	- 🗆 X
Eile       Edit       View       Insert       Operations       Record       Run  <b< th=""><th></th></b<>	
· Test suite *FirstWebTests.qft	Procedure call
<ul> <li>✓ Test suite</li> <li>✓ Test set Simple Tests</li> <li>✓ Setup Open Demo Page</li> <li>✓ Test case First</li> <li>✓ Test case Second (with Error)</li> <li>✓ Test case Third</li> <li>✓ Call procedure getDiscount → Discount</li> <li>✓ Cleanup Stop Demo</li> <li>✓ Procedures</li> <li>✓ Procedure getDiscount</li> <li>✓ Fetch text DiscountValue → returnValue</li> <li>✓ Return \$(returnValue)</li> <li>✓ Extras</li> <li>✓ Windows and components</li> </ul>	Procedure name getDiscount    Variable for return value   Discount    Local variable     I Local variable     Variable definitions    Name   Value      OK   Cancel

Figure 15.11: Procedure returning a value

The Fetch text node of above example is one of the QF-Test nodes directly setting a variable value. You need to specify the variable name in the attribute called accordingly. Again, you have the choice whether to make it local or global.

Quite a number of nodes have a 'Variable definitions' table where you may define variables local to a procedure or a test case. If the respective node is part of a procedure the variable will be local to the procedure. Otherwise it will be local to the respective test case. Variables bound to the test suite node can be accessed from all nodes within the test suite.

In debugging mode, all nodes you can bind a variable to will show up on the variable bindings table when entered.

You can enter variables in the command line via the parameter -variable. For details please refer to the manual, chapter 'Command line arguments and exit codes'.

### 15.5 Variable binding levels

Note This section may be difficult to understand when you are a programming beginner. Then it may be better to come back to it when you started writing procedures for your own tests.

In QF-Test there are many places where you can set a variable:

- The test suite node,
- in test cases and procedures as default or local variables,
- as a parameter in a procedure call,
- as a global variable and
- as a command line parameter.

Now the question is: Which is the correct place for defining a variable?

And the answer is: It depends on the use case.

Each level of the variable bindings has its own use case:

#### **Procedure parameters**

When you call the same procedure several times in a test case and with different values each time you should set the values via a parameter at a procedure call. This is done by adding a row to the variable definitions table of a Procedure call node specifying the respective variable name and value.

#### Local variables in a procedure

They are created in the procedure and are deleted when the execution of the procedure finishes. Use a local variable when you do not need it outside of the current procedure. In procedures they are predestined for intermediate results.

### Local variables in a test case

Variables local to a test case are either created during execution of a test case or defined in the respective table in the details of the test case node. They will be deleted from the variable bindings table once the test case finishes. Use them for values you want to refer to in several nodes of a test case and you do not need outside of it.

### **Global variables**

Global variables are created at some point of the test execution and exist until they are explicitly deleted or QF-Test is stopped. They survive stops and restarts of test execution. Use them for values that need to be accessible by all test cases. A typical example is the variable client created in the 'Setup' node when starting the application. To get rid of them you need to either exit QF-Test or select the menu item Run $\rightarrow$ Clear global variables.

#### **Command line parameters**

In batch mode you may want to run test with various parameters. They are valid throughout the batch run. A typical example would be the browser to be used.

### **Test-suite variables**

test suite variables can be referred to by all test cases. Their usage is the same as that of global variables. Only that they can be overridden at batch execution by command line parameters, whereas global variables cannot.

### Default values (Fallback stack)

You can define default values for variables of procedures, test cases and test sets. In case you do not define a variable of the same name on a higher level QF-Test will use the default value set.

# **Chapter 16**

# **The Standard Library (Web)**

### Video This chapter is also available as a video tutorial at



The Standard Library"

https://www.qfs.de/en/yt/tutorial-7.html

QF-Test provides a certain number of node types. If you need additional functionality you can implement it in a script node. To make life easier for you QF-Test comes with a set of procedures implementing the most commonly needed additional functions. You will find them in the standard procedures library.

When you cannot solve a problem using the provided node types it is a good idea to have a look in the standard library whether there is a solution to your problem. If you find a similar solution you can copy the procedure and adapt it to your needs. For information about scripting please refer the the manual, chapter 12 'Scripting'.

The file qfs.qft holds the standard procedures library. As it is constantly being enhanced and distributed with every new version of QF-Test you should not make any changes to procedures in that file, but copy the procedure to your own test suite if required and then adapt it.

- Note To make use of <code>qfs.qft</code> it needs to be included in your test suite's root node. With a newly created test suite the file <code>qfs.qft</code> is added automatically to the list of included files.
- Select the 'Test-suite' root node of your test suite.
  - Verify the <code>qfs.qft</code> is available within the table for "Include files".
  - Add <code>qfs.qft</code> to this list, if it's not already there.
- Note Path information is not necessary for qfs.qft as the include directory of QF-Test is contained in the library path (see also Reference part of the manual).

Action

• Add a procedure call to an arbitrary procedure from the qfs.qft standard library. In the procedure chooser don't miss to switch to the respective tab.

In addition to the description provided in this tutorial you can find the full HTML documentation of the standard library available via Help-Standard Library qfs.qft...

### 16.1 Inspecting the Standard Library

In addition to inserting procedure calls from the Standard Library, it also can be helpful to sometimes have a look how certain things have been implemented.

Action

• Locate and load the test suite file <code>qfs.qft</code>, which is located in the <code>qftest-7.1.3/include</code> directory of your QF-Test installation.

🕹 [qfs.qft] QF-Test	- 🗆 X
<u>File Edit View Insert Operations</u>	<u>R</u> ecord R <u>u</u> n De <u>b</u> ugger <u>C</u> lients E <u>x</u> tras <u>H</u> elp
	> 🗡 ●■Ⅱ✔ □() ▶■Ⅱ ☴ ᠔づびづびϟ◎
qfs.qft ×	
Test suite qfs.qft	Test suite
Q Test suite	Name
Procedures	
Package qfs	
> 🕂 Package android	🕂 🧹 🐣 📫 Include files
> 🕂 Package automac	File
Package autoscreen	
> 🗕 Package autowin	
> 🕂 Package awt	🕂 🏒 📩 📩 Dependencies (reverse includes)
> Package cleanup	File
> 🗕 Package client	File
> Package check	
> Package daemon	
> Package database	🛨 🧹 📩 🕂 Variable definitions
Package databinder	Name Value
Package dialogs	
> 🗕 Package <b>fx</b>	
> 🗕 Package jib	
> Package pdf	Execution timeout (ms)
> Package run-log	
Package shellutils	
V Package swing	Comment
> Package cleanup	This suite contains useful standard procedures provided and maintained by Quality First Software GmbH.
> Package checkbox	
> Package combobox	For a description of packages and procedures of this library in javadoc format, see qfs.html in the same directory.
Package dialogs	
Package filechooser	OK Cancel

Figure 16.1: The Standard Library

### 16.2. Selected Packages and Procedures

You can see there is one main package qfs that contains further specific packages. The qfs package helps to easily identify the packages belonging to the standard library.

The specific packages cover very different areas of utility procedures described in more detail further below.

Note Within nearly all of the procedures of this library, you'll notice that the variable s(client) is referenced. This is the standard mechanism for creating independence from a specific SUT. Here, the library assumes that the test suite which uses the library will set a value for s(client) prior to using any procedures.

### 16.2 Selected Packages and Procedures

We will now have a closer look at a number of selected packages and procedures from the standard library.

We will start with packages for accessing components dependent on the UI technology they are implemented for, here Web.

### 16.2.1 The Checkbox Package

We begin by looking at the qfs.web.checkbox package.

Some important procedures within this package are:

- **select** Selects (checks) a checkbox. If the checkbox is already selected, then no action is taken.
- **deselect** Deselects (un-checks) a checkbox. If the checkbox is already deselected, then no action is taken.
- set Sets a checkbox to a given state (true or false).

For each of these procedures, you pass the QF-Test ID of a checkbox component as a variable argument. The library handles verification of whether or not the checkbox state was properly set as expected.

Usage of the other procedures in this package follows the general model seen here.

### 16.2.2 The Select Package

The package qfs.web.select contains procedures to select a value in a combobox. Some procedures within this package are:

- setValue Select a value in the list of the combobox.
- getItemCount Return the number of entries in the select box.

### 16.2.3 The General Package

The package qfs.web.general contains useful procedures to work with components. Some procedures within this package are:

- setLocation Set the location of a given component.
- setSize Set the size of a given component.

### 16.2.4 The Table Package

The package qfs.web.table provides utility procedures for tables, like:

- getRowCount Return the number of rows of a table. It uses technology specific methods to gather the row number.
- **getColumnCount** Return the number of columns of a table. It uses technology specific methods to gather the column number.
- selectCell Select a given table-cell.

### 16.2.5 The Run log Package

The qfs.run-log package contains procedures, which writes specified messages into the run log. This package has been introduced to give testers without scripting-knowledge the opportunity to write messages into the run log.

Here is the list of the most important procedures within this package:

- **logError** Write a given error message into the run log.
- logWarning Write a given warning message into the run log.
- logMessage Write a given message into the run log.

### 16.2.6 The Run log.Screenshots Package

The <code>qfs.run-log.screenshots</code> package contains procedures, which write images into the run log and some helper methods.

Some important procedures within this package are:

- getMonitorCount Return the total number of monitors.
- **logScreenshot** Write a screenshot of the whole screen into the run log.
- **logImageOfComponent** Write an image of a given component into the run log.
- **logScreenshotOfMonitor** Write a screenshot of a given monitor into the run log.

### 16.2.7 The Shellutils Package

The  ${\tt qfs.shellutils}$  package contains procedures to support most common shell-commands.

Some important procedures within this package are:

- copy Copy a given file or directory to a specified target.
- deleteFile Delete a given file.
- exists Check for existence of a given file or directory.
- getBasename Return only the file name of a full file name.
- getParentdirectory Return only the directory name of the full file name.
- mkdir Create a given directory. It also creates non-existing directories in path.
- move Move a file of directory.
- touch Create a specified file.
- removeDirectory Remove a specified directory.

### 16.2.8 The Utils Package

The <code>qfs.utils</code> package contains procedures, which covers common helper-functionality during test-development.

Some important procedures within this package are:

### 16.2. Selected Packages and Procedures

- **getDate** Return a string containing the date. Default is the current date. (Other dates can be configured.)
- **getTime** Return a string containing the time. Default is the current time. (Other timestamps can be configured.)
- **logMemory** Log current memory use.
- **printVariable** Print the content of a given variable to the console.
- printMessage Print a given message to the console.
- writeMessageIntoFile Write a given string into a given file.

### 16.2.9 The Database Package

The <code>qfs.database</code> package contains procedures to execute SQL commands on a database.

Please note, that the class of the database-driver must be in QF-Test's plugin directory or in the CLASSPATH before QF-Test startup.

To get more information about the connection-mechanism to your database, please ask your developers or see www.connectionstrings.com.

Some important procedures within this package are:

- **executeSelectStatement** Execute a given SQL-Select-Statement. It stores the result in a global variable "resultRows" on the Jython variable stack and thus accessible from Jython scripts. Additionally, it stores the result in a group variable with the default name 'resultGroup', which can be accessed directy by QF-Test nodes.
- **executeStatement** Execute a given SQL-command. Here any SQL command can be specified.

### 16.2.10 The Check Package

The qfs.check package contains procedures to do checks.

Some important procedures within this package are:

• **checkEnabledStatus** Check, whether a component is enabled or disabled. It writes an error into the run log, if failing.

- **checkSelectedStatus** Check, whether a component is selected or not. It writes an error into the run log, if failing.
- **checkText** Check the text of a component. It writes an error into the run log, if failing.

### 16.2.11 The Databinder Package

The <code>qfs.databinder</code> package contains procedures for execution within a "Data driver" node which bind data for iteration.

Some important procedures within this package are:

- **bindList** Create and register a databinder that binds a list of values to a variable. Variables are separated by whitespace or by a given separator character.
- **bindSets** Create and register a databinder that binds a list of value-sets to a set of variables. Value-sets are separated by line breaks. Variables within a value-set are separated by whitespace or by a given separator character.

# Chapter 17

# **Control structures (Web)**

The two most important control structures of QF-Test are loops and the conditional execution of nodes. Loops can be implemented by two different kinds of nodes: While and Loop nodes. If, Elseif and Else nodes are available to implement conditional execution.

leo This chapter is also available as a video tutorial at



"Control Structures" https://www.qfs.de/en/yt/tutorial-8.html

# 17.1 If - else

You already came across If nodes in the Setup sequence in the chapter <u>Starting the</u> Browser<sup>(106)</sup>. Let's have a closer look at the details of the node.

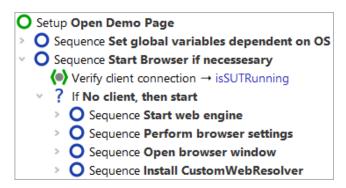


Figure 17.1: Setup Sequence with if-else structures

By means of an If node you can control whether certain nodes will be executed or not. In our case whether to start the SUT application. First, we need to find out if the client

Video

is already running. This is the job of the Wait for client to connect node, which writes the result of its inquiry, either true or false, into a variable named isSUTRunning.

Wait for client to connect	
Client \$(client)	^
Timeout 0	
GUI engine	
Result handling Variable for result isSUTRunning	
Local variable Error level of message	
Error	~
Throw exception on failure QF-Test ID	
Delay before (ms) Delay after (ms)	
Comment	
This node checks whether the SUT is already or result of this check will be stored in the variable isSUTRunning. The variable itself can contain the already running or false if SUT is not running.	ble true if SUT is

Figure 17.2: Wait for client to connect writes the result into the variable "isSUTRunning"

The If node has a Condition attribute where you'll find an expression evaluating the result variable isSUTRunning. As we want to refer to its value we need to use the syntax () (see also note on variable syntax in chapter section 15.1<sup>(166)</sup>).

lf			
Condition <b>not</b> \$(isSUTRu	nning)		Script language Jython 🗸
Name No client, then st	art		
+ / × ↑ Name	↓ Variable de Value	efinitions	
Maximum error le Exception QF-Test ID	evel		~
Delay before (ms)	1	Delay after (ms	)
Comment			

Figure 17.3: If node evaluates the variable

Depending on whether the client is already running or not QF-Test will execute the nodes nested in the If node.

### • Stop the client in case it is running.

Action

### • Single-step through the Setup node.

• Leave the client running and single-step through the Setup node a second time.

If you like you can check the value of the variable <code>isSUTRunning</code> in the variable bindings table. The first time it will have the value <code>false</code> so that the condition <code>not</code> <code>\$(isSUTRunning)</code> will become true and the SUT will be started. The second time it will be <code>true</code> and the if-condition will fail. The nodes nested in the If node will be skipped.

Note Within the first node of the setup sequence you will find more If nodes. They are used to set global variables determining which web browser to use, depending on the operating system. For better readability only If nodes have been used. You may just as well use Elseif and Else to implement that functionality. The child nodes of an Elseif node would

be executed if the If condition were false and the Elseif condition were true. The child nodes of an an Else node would be executed if the If condition and all Elseif conditions were false.

For checking the operating system you can directly resort to a QF-Test variable: QF-Test stores the information about the operation system in a group variable where the group is called 'qftest' and the variables 'linux', 'macos' or 'windows', respectively. The syntax for accessing group variables is \${group:varname}, e.g. \${qftest:windows}.

# 17.2 Loops

QF-Test provides two different kinds of nodes loops:

- Loop nodes execute their child nodes for a certain number of times. However, you can leave the loop any time using a Break node.
- While nodes execute their child nodes until a certain condition becomes false. Again, you can leave it any time using a Break node.
- Note Loop nodes will always stop after the given number of times. In the case of While nodes, however, you need to make sure that the condition will become false at some point. Otherwise you would have an infinite loop. In interactive mode you can always stop execution by hitting the pause button ■. In batch mode you would have to kill the QF-Test process. (You start QF-Test in batch mode using the command line parameter -batch. Then QF-Test does not start its UI and just executes the given test suite.)

In the following exercise we want to implement a test case checking whether a certain row is displayed in the table of the CarConfig application.

The actions of the test case will be:

- Determine the number of rows the table has.
- Loop over all rows and check if it is the row we are looking for.
- Break the loop when a match was found.
- Write an error to the run log if the row was not found.

Please start with recording a check on the row of interest:

- Action
- Activate the check recording mode by clicking the toolbar button
  - **Right-click a row** in the CarConfig application and select the menu item Row from the popup menu.

- Stop the recording by pressing
- · Change the name of the recorded sequence to e.g. 'Check row'
- Turn the recorded sequence into a test case by right-clicking it and selecting the submenu item Transform node into→Test case from the popup menu.

[FirstWebTests.qft] * QF-Test				- 🗆 ×
	ations <u>R</u> ecord R <u>u</u> n De <u>b</u> uy $\leftarrow \rightarrow  earrow equation (A = A = A = A = A = A = A = A = A = A $	gger <u>C</u> lients	E <u>x</u> tras <u>H</u> elp	<b>७ ७ ७ ७</b>
Test suite *FirstWebTests.qft		Sequence	2	
Test suite     Test set Simple Tests     Procedures	Cut	Name Charle Por	"	
	 Copy Paste Delete E <u>d</u> it	Ctrl-C Ctrl-V Delete Alt-Enter	Value	
	<u>M</u> arks Start run	>		
	Ioggle disabled state <u>B</u> reakpoint on/off Add to boo <u>k</u> marks	Ctrl+Shift-D Ctrl-F8 Ctrl-D	rror level	
	Trans <u>form node into</u> Pack n <u>o</u> des Unpack node <u>s</u> Additional nod <u>e</u> operations	>	Setup Cleanup Procedure Ctrl+Shift Loop	t-P
	Insert node		O Sequence with time limit Test case	
	Expand all Collapse all	Alt-Right Alt-Left	G Test ste <u>p</u> O <u>W</u> hile	Const
hanges applied	Clean tree What's this?	Ctrl-Period		Cancel

Figure 17.4: Transform a node into another one

In general, QF-Test lets you add nodes very efficiently by packing one node into another one:

• Open the test case node and **pack the recorded Check node into a loop** by rightclicking it and selecting the submenu item Pack nodes→Loop from the popup menu.

[FirstWebTests.qft] * QF-Test				- 🗆 ×
Eile Edit View Insert Operations Real C C C C C C C C C C C C C C C C C C C		er <u>C</u> lients E <u>x</u> tras		<b>ಶ ರ</b> ೮ ರ
Test suite *FirstWebTests.qft		Check items		
Test suite     Test set Simple Tests     Procedures		Client \$(client)		
Extras     Extras     Extras     Check ace Check Row     Check items: row (2) VehicleT     Sequence Recorded 17/04/202     Windows and components	C <u>u</u> t <u>C</u> opy <u>P</u> aste Delete Edit	Ctrl-X Ctrl-C Ctrl-V Delete Alt-Enter	t ID 1 ms Text	Regexp
	<u>M</u> arks		>	
	Start run <u>T</u> oggle disabled sta		Seguence	Ctrl+Shift-S
	<u>Breakpoint on/off</u> Locate component	Ctrl-F8 Ctrl-W	? <u>I</u> f O Loop	Ctrl+Shift-I
	Highlight compone Show i <u>n</u> Inspector	ent	C Test step Sequence with time limit	Ctrl+Shift-T
	Add to boo <u>k</u> marks Trans <u>f</u> orm node int		▼ Iry	Ctrl+Shift-Y
	Pack n <u>o</u> des		> 🔕 <u>W</u> hile	
	Additional nod <u>e</u> op	erations	>	
	lnsert node		>	~
	Clean tree	Ctrl-Period		Cancel
	What's this?			Finished: No erro

Figure 17.5: Pack a node into another one

QF-Test evaluates dynamically which nodes may be packed into one another and only presents the appropriate ones. So, in case you do not find the 'Loop' submenu item make sure you have right-clicked the correct node. The same holds true for the 'Transform node into' and 'Insert node' methods.

In the next series of actions we want to set the value for the Number of iterations attribute of the Loop node. In order to do so we need to find out how many rows the table has. There is no simple node that you could use. However, in the last chapter we learned that the standard library provides a lot of extended functionality. So let's insert the procedure getRowCount from the package qfs.web.table in the standard library.

### Action • Select the Test case node and press Ctrl-A

• Press the 'Select procedure' button (•) left to 'Procedure name'.

- Click the tab 'qfs.qft' in the 'Select procedure' dialog.
- Navigate to 'getRowCount' in the package 'qfs.web.table'
- Click 'OK' to select it.
- Click 'OK' in the 'Procedure call' dialog.

Adding a procedure via <u>Ctrl-A</u> was described in <u>Manual creation of procedures<sup>(135)</sup></u>. If you would like to check with the screenshots please have a look there.

#### • Enter the variable name rows in the Variable for return value attribute.

- Change the default value for the id in the variable definitions table to the QF-Test component ID of the table, i.e. VehicleTable.
- Click the **OK** button.
- Select the Loop node.
- Enter a reference to the variable \$ (rows) in the Number of iterations attribute of the Loop node.
- Enter the name of an iteration counter, e.g. i in the respective attribute of the Loop node.
- Click the **OK** button.

Loop			
Name			
Number of iteratio	ons	Iteration counter	
\$(rows)	0115	i	
+ / × +	1		
+ / × +		efinitions	
Name	Value		
Maximum error le			
	evel		
Exception		~	
QF-Test ID			
Delay before (ms)		Delay after (ms)	
Delay before (IIIs)		Delay after (IIIs)	
🧹 Comment			

Figure 17.6: Details of a Loop node

In the next series of actions we will change the recorded row index to the iteration counter and add a variable for the result to the details of the Check node. Then we will add an If node after the Check node evaluating the result, with a Break node within to quit the loop when the row was found.

#### Action

### Open the Loop node.

- Select the Check node.
- Change the recorded row index of the QF-Test component ID to the iteration counter \$(i). The QF-Test component ID should now read VehicleTable@Model&\$(i)
- Enter the variable name checkSucceeded in the 'Variable for result' attribute and click the OK button.
- **Right-click** the Check node and the submenu item Insert node →Control structures→Break from the popup menu.

Action

- Click 'OK' in the 'Break' dialog.
- Pack the Break node into an If node by pressing Ctrl-Shift-I) (Of course you can also pack it via the menu).
- Type \$ (checkSucceeded) in the 'Condition' attribute of the 'If' node and click the OK button.

The variable checkSucceeded will be set to either true or false by the Check node so that the reference to the variable  $\$  (checkSucceeded) is all we need to enter in the 'Condition' attribute of the If node.

In the next series of actions let's add an Else node as the last node in the Loop node. It will only be entered if all repetitions of the loop were executed, which in our case means that the row was not found and the check never became true.

- **Collapse the If node** if it is open. This is important because otherwise the Else node would belong to the If node and not to the Loop node.
  - Right-click the If node and select the submenu item Insert node  $\rightarrow$  Control structures  $\rightarrow$  Else.
  - Click 'OK' in the 'Else' dialog.
  - Open the Else node.
  - From the standard library insert the procedure logError contained in the package qfs.run-log as described above.
  - Type Row not found in the value field of message in the Variable definitions table.
  - Change the value of withScreenshots in the Variable definitions table from false to true.
  - Click 'OK' in the 'Break' dialog.

When you run tests in batch mode screenshots are a great help for analyzing errors. On the other hand a great number of screenshots lead to a big log-file. This is why the default value for withScreenshots is false.

Last, let's complete the test case with Setup and Cleanup nodes and move it into the top part of the test suite.

Action • Copy the Setup and Cleanup nodes of 'Test set: Simple Tests' into the new test case as the first and last node.

• Move the test case from the Extras section into the top section of the test suite after the 'Test set: Simple Tests' node.

This is what the new test case would look like:

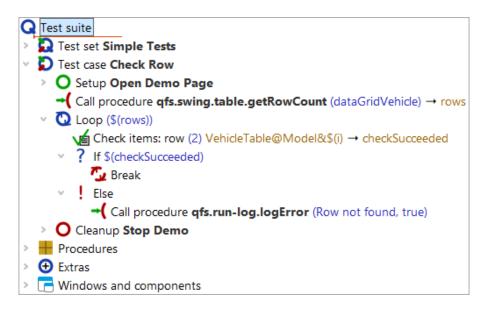


Figure 17.7: The new test case

#### • Execute the new test case.

It should run without error.

• Then modify a value in the details of the Check items node, e.g. change the name of the car to Wrong value.

Check items		
Client \$(client)		
C QF-Test component ID VehicleTable@Model&\$(i)		
t d tems		D
O Wrong value     \$15,000.00		Regexp
Check type identifier		
Timeout		
Result handling Variable for result checkSucceeded		
Local variable		
Error level of message Error		~
\$ Throw exception on faile	ure	
Name		
QF-Test ID		
Delay before (ms)	Delay after (ms)	
Comment		

Figure 17.8: Details of the Check items node

## • Execute the new test case again.

This time the Else node should be entered and you should get an error message.

# **Chapter 18**

# It's time to start your own Application (Web)

After having spent a lot of time with all those example programs, you are now ready to start on your own application (if you really haven't already done so).

Video This chapter is also available as a video tutorial at



"It's time to start your own Application" https://www.qfs.de/en/yt/tutorial-9.html

The Quickstart Wizard available via the menu  $\boxed{\text{Extras} \rightarrow \text{Quickstart Wizard...}}$  helps you to achieve this. Simply follow the wizard steps to generate the setup sequence. Please refer also to chapter 3 "Quickstart" in the user manual.

Then go ahead with what you have learned in this tutorial - record small sequences of events and checks, turn them into procedures which go into your test library, then set up the test cases using procedure calls.

Finally, we reached the end of the basic tutorial part.

# Part III

# Native Windows UI testing with QF-Test

This part III of the tutorial is meant to help you learn the basic features and workflows of QF-Test. It focuses on the test of native Windows applications and its specifics.

For testing Java programs please go to part  $I^{(2)}$  or part  $I^{(103)}$  for Web applications, as those parts use the same scenarios but with different systems under test.

In case you already have worked through part I or II but win testing is relevant for you as well, you might not want to work again through the all same scenarios.

Within part  $V^{^{(294)}}$  more advanced QF-Test features are explained, applicable for all supported UI technologies.

# **Chapter 19**

# Working with a Sample Test suite (Win)

In this first chapter, we will have a look at a simple test suite, explain its major elements, execute it and evaluate the result.

#### Video This chapter is also available as a video tutorial at



"Working with a Sample Test suite" https://www.qfs.de/en/yt/tutorial-1.html

## 19.1 Loading the Test suite

Note On first startup of QF-Test and/or the System Under Test (SUT) via QF-Test you might get a security warning from the firewall asking whether to block the Java network communication 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.

After starting up QF-Test, you can immediately bring up our first example test suite.

Action

- Press the boundary button to bring up the file open dialog
  - Navigate to the subdirectory **qftest-7.1.3/doc/tutorial** of your QF-Test installation
  - There select the file FirstWinTests.qft

QF-Test will then load the indicated test suite which should look as follows:

[FirstWinTests.qft] QF-Test		– 🗆 X
<u>F</u> ile <u>E</u> dit <u>V</u> iew <u>I</u> nsert <u>O</u> peration	ions <u>R</u> ecord R <u>u</u> n De <u>b</u> ugger <u>C</u> lients E <u>x</u> tras <u>H</u> elp	
000 0C 4	÷→ ≯ ●■■↓ ⊂ () ▶■Ⅱ ☴ `	<b>3 7</b> 6 7 7 6 4 0
FirstWinTests.qft ×		
Test suite FirstWinTests.qft	Test suite	
Q Test suite	Name	~
> D Test set Simple Tests	Name	
Procedures	+ / × 1 + Indude files	
<ul> <li>         • Extras     </li> <li>         • Windows and components     </li> </ul>	File	
<ul> <li>Windows and components</li> </ul>	qfs.qft	
	🛨 🗹 🔺 🕂 Dependencies (reverse includes)	
	File	
	+ / × + Variable definitions	
	Name Value	
	Execution timeout (ms)	×
	OK	Cancel
	Terminal	
		^
		~

Figure 19.1: The Test suite FirstWinTests.qft

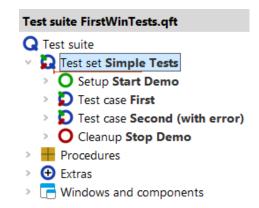
The **left part** of the main window contains the test suite, organized in a tree structure. The **right side** shows the details of a selected tree node.

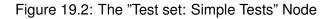
At **bottom right** you'll see the terminal displaying messages sent by QF-Test and the application you are testing.

In the tree structure of the main window you can navigate and select individual nodes of the test suite.

#### • Double click the node Test set: Simple Tests to expand it.

You'll find the test set contains two test case nodes enclosed by a "Setup"/"Cleanup" pair.





In the following sections we'll describe the purpose and function of the individual nodes.

## **19.2 Starting the Application**

Our first step is to examine the "Setup" node:

#### • Expand the Setup: Start Demo node now.



Figure 19.3: The Setup Node

In the "Setup" node you'll see two child Sequence nodes:

- Set variable set the variable 'client' to the connection name for the SUT, which will be needed for every action replayed to the application.
- Sequence: Start client if necessary starts the System Under Test (SUT) in case it is not already running.

Let's also have a brief look inside the **Sequence: Start client if necessary**:



Figure 19.4: The Sequence to start the SUT

First you see a "Wait for client" node to double-check whether the client is already running. Only if it is not, it will be started.

The **Start Windows client** node starts the application (SUT) and sets up a link between *qftest* and the SUT. In order to be independent of the actual installation we use a relative path, starting from the QF-Test version directory, contained in the QF-Test variable f(qftest:dir.version) (see manual chapter Variables).

At this point, we're ready to actually start the SUT:

Action

- Click on the **O** Setup: Start Demo" so it is selected and still expanded (the child nodes stay visible).
  - Click the Start test run toolbar button. This button causes the selected node to be executed.

During execution QF-Test marks the active step by use of an arrow pointer ->.

When the setup sequence is completed, our demo application "CarConfiguratorNet Form" is going to appear on the screen. As QF-Test gets back the focus after the replay action, thus being raised to the foreground, the demo application might be hiding behind it.

CarConfigurat	or Forms		-		>
File Options	Purchase order	Help			
ehicles Specials	Accessories				
Vehicles					
Model	ID		Price		
Hydro2	M1			\$79,00	0.00
Voyage	M2			\$56,50	0.00
Voyage Hybrid	M3			\$56,50	0.00
Roadster-E	M4			\$45,90	0.00
15	M5			\$29,00	0.00
Base price		s	0.00		
Specials price		\$	0.00		
Accessories price		S	0.00		
Discount			0%	-5%	
Final price		\$	0.00		

Figure 19.5: The Windows CarConfigurator Demo

## 19.3 First Test case

Let's check out now what test case "First" contains. There are four test steps inside:

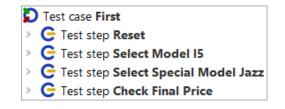


Figure 19.6: The "First" Test case

**Reset** - performs a reset by use of the File->Reset menu action and a selection of the Vehicles tab.

Select Model 15 - chooses the last model 15 within the vehicles table.

Select Special Model Jazz - switches to the Specials tab and selects the Jazz option.

#### 19.3. First Test case

**Check Final Price** - checks that the calculated final price field located at bottom right equals a given value.

Test steps are used to group the nodes and to document what is being done. This will prove very useful when it comes to error analysis or test adaptations.

### • Expand the four test step nodes.

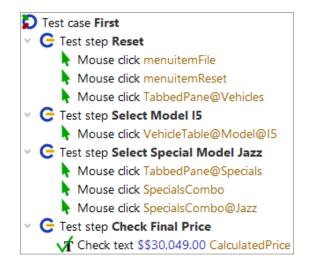


Figure 19.7: Details of the first Test case

You can see Mouse clicks and Checks, which have been grouped in test step nodes for better readability of the test case. The action nodes display the action type (Mouse click, Check, ...) and the component targeted, i.e. where the action goes to. When writing a test you can use the QF-Test recording function to create them. Recording will be explained in the next chapter 20<sup>(220)</sup>.

- Action
- Please select the test case "First" node
- Click the replay button **>**.

The test steps will then be replayed in the SUT.

The test result is indicated during and after the test run in the status line at the bottom of the QF-Test main window and should read now 'Finished: No errors'. Next to it there are counters for the numbers and results of the test cases executed. In our case it was just one, error-free, which means a success rate of 100%.



Figure 19.8: The result view in the status line

Each counter icon has a descriptive tool tip. A list of all counters can be found in the chapter 'Capture and replay' of the manual.

## **19.4 Second Test case - with Error**

The second test case will show us what happens when an error occurs during test execution.

• Expand the test case "Second (with Error)".

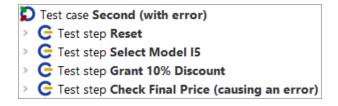


Figure 19.9: The Second Test case

Apart from the third test step it is identical to the first test case. So what does this one do?

Test step: Grant 10% discount - Writes the value 10 into the discount field.

The 'Input' node is another basic action node and can be created directly via the QF-Test recording function.

• Expand the Test step: Grant 10% Discount.

C Test step Grant 10% Discount

Figure 19.10: Details of the second Test case

Let's execute the second test case:

- Action Select the node Test case: "Second (with Error)".
  - Click the replay button ▶.

Errors during replay
 There were 0 exceptions, 1 error and 0 warnings encountered during replay.
 <u>OK</u> Show run log

Figure 19.11: Error in the second test case

What happened? To find out we'll open the QF-Test run log for error analysis.

An alternative approach for error analysis would be to execute the test case again using the debugger. This will be explained in chapter Using the Debugger (Win)<sup>(245)</sup>.

## **19.5 The Run log for Error Diagnosis**

This time a dialog shows up telling us that an error occurred.

QF-Test logs detailed information for every test execution.

• Please open the latest run log by one of the following options:

- either by pressing the Show run log button of the error dialog

or in case you have already closed the dialog

- by pressing **toolbar button** ⇒ or
- by pressing **Ctrl-L**.
- Note The most recent run logs are also listed at the bottom of the Run menu of the main window.

The run log comes up in a separate window displaying the logged actions of the test case you've just executed:

211

Action

ile <u>E</u> dit <u>V</u> iew <u>H</u> elp 🗅 🗁 🛱 🏷 (~ -	> ☴ ७९ ∖	0, 0 🚍		
Run log	Run log			
Run log (detailed)	Test run ID			
Test case Second (with error)	240417164149			
	Time and date	Duration	Real time spent	
	16:41:49.788 2024-0	7.690 s	7.937 s	
	Result Exceptions	Errors	Warnings	
	0	1	0	
	Environment			
	Hostname	Operating system	Executed by	
	w10t2	amd64-Windows 10-		
	Java version	QF-Test v	ersion	
	17.0.10+7	7.1.3-dev	(30107)	
	ОК		Cance	1

Figure 19.12: Run Log for the second test case

The run log reflects the tree structure of the test suite view you are already familiar with. When you click on one of the nodes on the left side, the properties of the event including time stamp and duration will be displayed on the right.

In the tree left you will notice nodes surrounded by a red border. These are indicators showing where a problem occurred in a child node. If you keep expanding the red nodes, you'll eventually come to the actual error node.

Please use an easier way to find the error source by pressing the Find next error
 toolbar button or the [Ctrl-N] key shortcut.

All nodes with red highlighting have been expanded and the actual error node has been selected:

Run log	Log for failed Check to	ext		
in log (detailed)	Time and date	Duration	Real time spent	^
Test case Second (with error)	16:41:57.068 2024-04	0 ms	0 ms	
C Test step Reset				
C Test step Select Model I5	Message			
C Test step Grant 10% Discount	Mismatch			
C Test step Check Final Price (causing an error)	Expected: '\$29,000.00'			
✓ M Check text \$29,000.00 CalculatedPrice ➤ CarConf	Got: '\$26,100.00'.			
16:41:56.409 Expansion of text: '\$\$29,000.00'	Annotation			
\$16:41:56.410 Expansion of client: '\$(client)' →	Annotation			
TextField CalculatedPrice				
Failed: Check text \$26,100.00 CalculatedPrice				
Stack trace				
C Screenshot of screen 1	Current values for failed	Check text		
Screenshot of window: CarConfigurator Forms	Client			
16:41:57.718 Logged information from client	\$(client)			
	QF-Test componen	t ID		~
< >	OK		Cancel	

Figure 19.13: Error in the second test case

The error message on the right says that the expected value of the final price field differs from the actual one. Of course this error is there by intention as the second test case is supposed to show us how to analyze an error.

Another helper for error analysis is the **Screenshot** of the SUT taken at the time when the error occurred (four nodes down from the red node). Being able to see the state of the SUT at that moment often proves useful for determining the cause of the error. The following image shows a screenshot node:

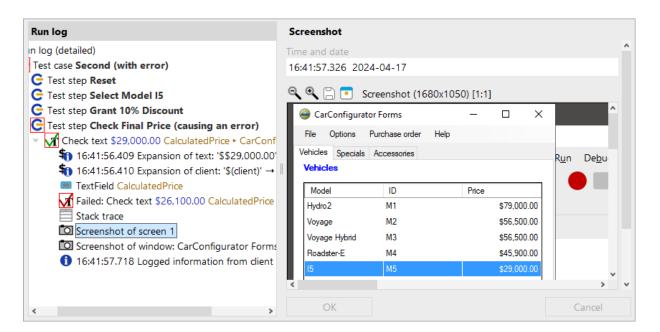


Figure 19.14: Screenshot node showing the error situation

In addition to screenshots of all monitors, QF-Test also saved images of the client windows at the time of the error. This allows you to analyze the contents even if they are covered by other dialogs or windows.

The information gathered in a long test run accumulates and can eat up enormous amounts of memory. This is the reason why QF-Test is configured by default to create a compact run log, keeping only the relevant information for report generation and error diagnosis.

This functionality can be configured via the option "Create compact run log" within  $\boxed{\text{Edit} \rightarrow \text{Options} \rightarrow \text{Run logs} \rightarrow \text{Content}}$ . The root node of the run log tells you whether it is a compact or detailed run log. You can also configure the number of screenshots to be saved.

## 19.6 Getting Help

We take a short break in this section to give you a few tips that might prove helpful as you continue with the tutorial.

There are different places where you can look for help or information:

The comprehensive achieved via most search can be you  $Help \rightarrow Online search...$ the functionality This navigates search to homepage available our and allows querying throughout all on

Note

## 19.7. Stopping the Application

**documentation** (manual https://www.qfs.de/en/qf-test-manual.html, tutorial https://www.qfs.de/en/qf-test-tutorial.html, standard library https://www.qfs.de/en/qf-test-support/documentation/standard-library.html,

blog https://www.qfs.de/en/blog.html and our videos https://www.qfs.de/en/get-started-with-qf-test/videos.html). Search results can be filtered as needed.

In case you work **offline** and want to search for a certain topic, the **PDF versions of the manual or tutorial** available via the **Help** menu can be used. The Offline HTML version doesn't have a content search option. However, there is a link to the PDF on every HTML page in the top and bottom lines so switching is easy.

QF-Test also offers a **context sensitive help** for tree node types and their details. To use it, simply press the **right mouse button** on an arbitrary tree node or attribute in the details pane. From the context menu select **What's this?**. This will directly bring you to the reference explanation of this item in the manual.

Beside getting help from the documentation you also have the option to contact our support team. During your evaluation phase or after that as customer with a valid maintenance contract you may issue your questions directly to our support experts (send an email to support@qfs.de or use the support form from the QF-Test help menu Contact the support team or call us +49 8171 - 38648 20). Contact details can be found at https://www.qfs.de/en/qf-test-support/product-support.html.

## **19.7 Stopping the Application**

We haven't inspected the cleanup sequence, so let's have a look at it now:

## • Expand the Cleanup: Stop Demo node.



Figure 19.15: The Cleanup Sequence

The Cleanup sequence stops the client process in a hard way and waits until it fully terminates. This is a very simple approach and shall suffice for the moment.

Action **Execute** the **cleanup** to see the CarConfigurator demo vanish.

## 19.8 A full Test Run

After we have seen how the single elements of the test set work, let us have a look at the functionality provided by the test set node.

Action

- First, close the CarConfigurator application in case it is still running.
- Then select the "Test set: Simple Tests" node.
- Execute it with the replay button .

The result dialog will come up after test execution, informing us about the error caused by the second test case.

Action • Open up the run log again by  $\Rightarrow$  to take a closer look:



Figure 19.16: Run Log for the Completed Test set

This shows the special behavior of Setup / Cleanup nodes in a test set: They are executed before and after **each test case** to help achieving a proper starting state for each test case.

Note Stopping the SUT after each test case is not the smartest way to ensure a clean state. There are more elegant ways for setup and cleanup that will be explained with the advanced features in this tutorial (chapter 29<sup>(304)</sup>).

## **19.9 Report Generation**

In the world of quality assurance documenting the test results is pretty important. To this end, QF-Test offers an automated report-generation feature. Since you've just done a complete test run, we're at a good point to show you this feature.

• Make sure the **run log of the test run** is open.

• In the **run log window** select **File**→**Create report...** to bring up the dialog for the report parameters.

	×
Save to directory C:\Users\qfs\.qftest\FirstWinTests	
<ul> <li>Create HTML report</li> </ul>	<ul> <li>Create XML report</li> </ul>
Create JUnit report	
<ul> <li>List test steps</li> </ul>	List checks
✓ List exceptions	✓ List errors
List warnings	✓ Show icons for nodes
Pass HTML tags through	<ul> <li>Use doctag extensions</li> </ul>
Ignore skipped nodes	Ignore not implemented nodes
Embed pie chart	<ul> <li>Include testsuite name</li> </ul>
Embed thumbnails	
Scale factor for thumbnails (%) 20	
Show report in HTML browser	when finished
<u>O</u> K	<u>C</u> ancel

Figure 19.17: Report Generation Properties

In the first field, you can specify the file name of the report. Following this, you can decide what type of report you want. QF-Test offers three kinds of reports, HTML, XML and JUnit format. An XML report is useful if you want to process the data further, e.g. if you have written your own XSLT stylesheets to shape the report. JUnit reports prove useful when you need to import results into build or test management tools.

Let's generate an HTML report from the results of the last test run.

• So just leave the report **options unchanged**.

Action

• Start generation by pressing the **OK** button.

#### 19.9. Report Generation

Test Report Summary - Google Chrome Х -<u>っ ペ X C</u> file:/C:/Users/qfs/.qftest/FirstWinTests/report.html **Test Report Summary OF-TEST** Version 7.1.0 Summary Start time 2024-02-07 11:37:40 Test-run ID 240207113740 Executed by qfs Host w10t2 Operating system amd64-Windows 10-10.0 Java version 17.0.10+7 failed passed not executed QF-Test version 7.1.0 Total number of test cases Number of skipped test-sets • Number of test cases with exceptions
• Number of not implemented test cases Number of test cases with errors Number of executed test cases Number of test cases with expected errors Percent test cases passed Number of successful test cases Time spent in tests Number of skipped test cases Elapsed realtime **Overall result (**) 2 0 1 0 1 0 0 0 2 50 33 s 33 s 1 error Overview: Test-suites
 Overview: Test-suites with errors
 Overview: Errors Summary Ready

The report will then be generated and presented to you in a browser window:

Figure 19.18: An HTML Report

The report begins with a summary containing informational data from your system on the top left side, a legend describing the meaning of icons used in the report on the top right side, an overview pie chart in the middle and the overall test result below. In our case, the result we see are the error-free first test case and the second with the well-known error, leading to a success rate of 50%.

Following the summary there are three overview sections:

- 1. Test suites that have been executed within the test run.
- 2. Test suites in which errors occurred.
- 3. Errors including their exact position and detailed message.

The report generator is very useful for creating an overview document for presentation and archiving purposes.

# **Chapter 20**

## Creating your own test suite (Win)

In the second chapter of the Win tutorial we will create our own sequences for starting and stopping an SUT from QF-Test. Furthermore we are going to record actions and checks and use those to build up a simple test case.

Video This chapter is also available as a video tutorial at



"Creating your own test suite" https://www.qfs.de/en/yt/tutorial-2.html

## 20.1 Starting the Application

To begin, you need to launch the application from *qftest*. There is a **Quickstart Wizard** to help you in creating the respective setup sequence.

Action

- Open a new test suite via the menu item File New test suite.....
  - To open the Quickstart Wizard please use the Extras→Quickstart Wizard...
     menu.

The Wizard starts up with a welcome message and some further information.

Action • After saying a short hello please press the **Next** button to begin.

Quickstart Wizard	
Pages	Welcome
First steps Welcome Type of application Application specifics Optional Conclusion Summary	<ul> <li>Welcome to the Quickstart Wizard!</li> <li>In general your application needs to be started by QF-Test in a specific way in order to allow recording and replay of user actions as well as verification of results.</li> <li>This applies to both stand-alone Java programs and web applications (defined ba URL).</li> <li>This wizard will assist you in creating a startup sequence for your application.</li> <li>Just press Next to begin.</li> <li>Further information can be found in the <u>Quickstart chapter</u> of the manual.</li> </ul>

Figure 20.1: The Quickstart Wizard

In step two you can choose the type of application to be tested.

- Please select the fourth option **A native windows application**.
  - Press Next.

Action

🕹 Quickstart Wizard		Х
Pages	Type of application	
First steps Welcome Type of application Application specifics Optional Conclusion Summary	Please select the type of your application from the following list.  A Java application A web application in a browser (specified via URL) A web application in an emulated mobile browser A native Windows application An Android application An Electron application A PDF document	
	<u>B</u> ack <u>N</u> ext <u>Finish</u>	ncel

Figure 20.2: Type of Application

The next step specifies the Windows application executable.

- For this please use the **select** button on the right.
  - Navigate to the subdirectory **qftest-7.1.3/demo/carconfigForms/** of the QF-Test installation.
  - Select the CarConfigForms.exe.

The second field can be left empty in our case hence it should be mentioned that it is also possible to connect QF-Test to an already running Windows application by help of its window title. Such can be specified in the second field. Also regular expressions can be used for specifying those window titles.

Action

#### 20.1. Starting the Application

- Note In the figure below we used the QF-Test variable \${qftest:dir.version} to address the version specific directory of the QF-Test installation, which you have already come across in the previous chapter. (Details on special QF-Test variables can be found in the manual chapter Variables).
- Please press the Finish button, as we do not need to go to the further optional steps for our simple demo.

🕙 Quickstart Wizard	×		
Pages	Windows application		
First steps Welcome	Please specify the windows application you want to test either directly or via the file explorer on the right.		
Type of application Application specifics	\${qftest:dir.version}/demo/carconfigForms/CarConfigForms.exe -lang en		
Windows application			
Optional Working directory Client name	Please specify the title of the windows application. This application may already be running or started outside of QF-Test.		
Conclusion			
Summary	The entered title contains regular expression syntax.		
	Note: You need to fill in at least one of the parameters to start/connect your windows application.		
	<u>B</u> ack <u>N</u> ext <u>Finish</u> <u>C</u> ancel		

Figure 20.3: Windows executable file selection.

We directly reach the final summary that explains what will happen after closing the wizard and how to continue.

Action • Please press **Finish** in order to end the wizard.

🕘 Quickstart Wizard		×
Pages	Summary	
<b>First steps</b> Welcome Type of application		nish" button, the 'Setup' sequence for your configured be created within your test suite. You will find that sequence
Application specifics Windows application	Select this 'Setup' seq	uence and press the 'Start run' button in the toolbar.
Optional Working directory	Your application shoul	d then be started and you will be ready for recording.
Client name	Entered data:	
Conclusion Summary	Windows application:	<pre>\${qftest:dir.version}\demo\carconfigForms\CarConfigForms -lang en</pre>
	Window title:	
	Client name:	CarConfigForms
	Working directory:	\${qftest:dir.version}\demo\carconfigForms
	<	>
		Back Next Einish Cancel

Figure 20.4: Final Information

The generated setup sequence **Launch CarConfigForms** appears in the "Extras" section of your test suite and contains three parts:

- Set variable defines the global client variable used throughout the test suite.
- Wait for client to connect checks whether the client is already running.
- Launch SUT if not running starts the System Under Test as client if it is not already running by us of a Start windows application, and wait for its start.
- Note The information whether the client is already running is stored into a variable "isSUTRunning" in the first Wait for client to connect node and evaluated by the subsequent "If" condition. You can find this in the respective node details. This kind of conditional execution will be explained later in detail.



Figure 20.5: Generated Setup Sequence

Now we want to see some action:

- Action
- Please ensure the Setup: Launch CarConfigForms node is selected.
  - Then **click** b or simply hit "Enter" (Return).

You should see the Windows CarConfigurator application appear on your screen soon. As the focus changes back to QF-Test after the execution, the Demo might be covered by the test suite window.

ဓ CarConfigu	rator Forms		_		×
File Options	Purchase order	Help			
Vehicles Specia	als Accessories				
Vehicles					
Model	ID		Price		
Hydro2	M1			\$79,00	0.00
Voyage	M2			\$56,50	0.00
Voyage Hybrid	M3			\$56,50	0.00
Roadster-E	M4			\$45,90	0.00
15	M5			\$29,00	0.00
Base price		s	0.00		
Specials price		\$	00.00		
Accessories pric	e	\$	00.00		
Discount			0%	-5%	
Final price		s	0.00		

Figure 20.6: The Windows CarConfigurator Demo

At the end of this section let's save our test suite.

- Action
- Press the toolbar button or use the File→Save menu option with its short-cut Ctrl-S.
  - In the file explorer navigate to an appropriate directory where you have write access e.g. Documents in your user home directory.
  - Provide a name e.g. MyFirstTests.qft.
  - Finish the saving action by pressing on Save.

## 20.2 Recording Actions

You're now ready to record some actions for our demo:

👄 CarConfigurat		_		×		
File Options	Purchase order	Help				
Vehicles Specials	Accessories					
Vehicles						
Model	ID		Price			
Hydro2	M1			\$79,00	0.00	
Voyage	M2			\$56,500.00		
Voyage Hybrid	M3			\$56,500.00		
Roadster-E	M4			\$45,900.00		
15	M5			\$29,00	0.00	
Base price		\$	0.00			
Specials price		\$	0.00			
Accessories price		s	0.00			
Discount			0%	-5%		
Final price		\$(	0.00			

Figure 20.7: Recording actions on the CarConfiguratorForms Demo

Action • Please press the • Record button.

#### 20.2. Recording Actions

- Switch to the SUT application window. From now on every mouse and keyboard action performed within the SUT window will be recorded. There is an indication show as red square below the mouse cursor. Just after this has disappeared you may record your action on the component.
- Click to the table cell 15 in the last row.
- Change to the **Specials** tab of the tabbed pane.
- Choose the special Jazz from the combo box.
- Finally switch back to the Vehicles tab in the tabbed pane.
- Stop the recording by use of the **b**utton.
- Note A red square below the mouse cursor indicates by its disappearing QF-Test is ready for recording actions on the component.

You'll find the recorded sequence placed in the "Extras" section:





The recorded sequence has a default name with date and time of the recording. You can change this name as you see fit by simply clicking on the node and changing its properties in the details view on the right.

- Action
- Please **rename** the recorded sequence to "Select Model I5 Jazz".
  - Then **open** the sequence node to see its content. There should be the expected mouse clicks. You should even be able to interpret where they go to.

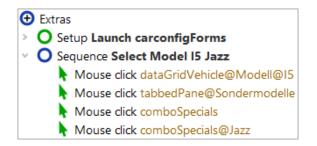


Figure 20.9: The Renamed Sequence

#### 20.3. Recording Checks

Now, let us replay the recorded sequence.

Action

Action

Select the Select Model I5 Jazz sequence node.

#### Press the play button

You should now see exactly the same sequence of mouse events executed in the SUT as you recorded before.

The sequence is supposed to replay (even multiple times) without errors. You should see "Finished: No error" in the bottom right corner of your test suite window.

## 20.3 Recording Checks

To verify the client's behavior we use check nodes, which query certain states and properties of elements within the SUT. Also checks can be recorded.

- Click the "Record a check" ✓ button.
  - Switch to the SUT window. When moving the mouse over the components you will notice a blue border indicating the current selection.
  - Move the mouse over the value field of "Final price" and wait for the check border to appear.
  - **Right-click** now on the value field. In the popup menu you are offered a choice of standard checks for a text field component.
  - Select the first option "Text" for a check on the textual value of the field.
  - Stop the recording by using the **Stop button**.
- Note Ensure the check border is shown for a component before clicking to record a check. Otherwise the check may not be recored properly.

Again, the newly recorded sequence appears in the "Extras" section.

- Please rename the sequence to "Check final price"
  - Expand it to see the check node.



Figure 20.10: The recorded check node

• Feel free to run this sequence, too, verifying it is working properly.

As the next step, we want to create a test case from the two sequences.

## 20.4 Setting up a test suite

Action

The top-level nodes of the test suite define its basic structure:

- Right after the "Test-suite" node an arbitrary number of "Test set" and "Test case" nodes may be added specifying the functional tests.
- "Procedures" contains reusable sequences (procedures) that can be organized into packages.
- "Extras" is the scratch pad for recording and experimentation.
- "Windows and components" contains the all-important elements of the test suite: registered elements of the SUT, e.g. windows, menus and buttons. The details of each element in the "Windows and components" section contains the properties of the recorded UI element required by QF-Test to find the component when replaying a test.

Functional test cases are represented by "Test case" nodes and can be grouped and structured with the help of "Test set" nodes.

"Setup" and "Cleanup" nodes are intended for test steps ensuring a well-defined state before and after a test case.

- Let's start by renaming the top-level test set node from "unnamed" to "Demo Tests".
  - If a dialog pops up asking us whether to update references we can simply confirm with "**Yes**".

#### 20.4. Setting up a test suite

The second step is to move the "Setup" node generated by the Quickstart Wizard from the "Extras" node into the "Test set" node - right before the "Test case" node. Moving the "Setup" node can be done via mouse (Drag&Drop), contextmenu (right mouse-button copy/paste) or by Ctrl-X and Ctrl-V keyboard commands.

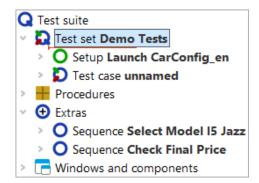


Figure 20.11: Start organizing the test suite

The next step is to make a test case of the two sequences previously recorded.

- Please **rename** the test case node from "unnamed" to "First".
  - Open the test case node by clicking the '+' symbol.
  - Move the two sequences from "Extras" into the test case.

You need to open the test case node because otherwise QF-Test would try to place the sequence nodes after the test case node on the same level, which is not a valid option.

QF-Test always records sequence nodes. They have the same functionality as test step nodes, only they do not show up in the report. So, just to show you, we will transform the two sequence nodes into test step nodes.

- Please open the context menu for the first of the two sequence nodes by a rightclick.
  - Choose Transform node into...→Test step
  - Repeat this for the **second** sequence node.

Your test suite should now look like this:

230

Action

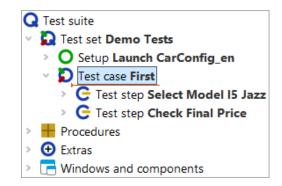


Figure 20.12: The Organization of your test suite

## 20.5 Stopping the Demo

The only thing missing now is a cleanup sequence closing down the SUT.

There are usually various ways to terminate an SUT, e.g. clicking the close button of the application window, pressing ALT-F4 or via the menu action  $\boxed{\text{File} \rightarrow \text{Exit}}$ . All these options can be directly recorded and then used in the cleanup sequence.

Let's use the last one. So please perform the following steps.

Action

#### Start recoding

- Record the menu SUT menu action File→Exit.
- You see the demo window close.
- Stop recording
- Rename the recorded sequence to "Stop demo".
- Open the **context menu** for the recorded sequence and select **Convert into...** -> Cleanup.
- Finally move the cleanup node up to be the last node in the test set.
- Note The cleanup node can only be dragged and dropped to the test set if the test set's last child node is collapsed. To expand or collapse a node during a drag and drop operation, hold your cursor over the triangle next to the node.

You should end up with the following:

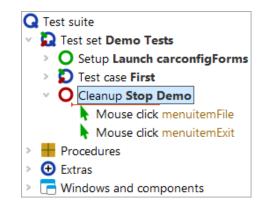


Figure 20.13: The simple cleanup sequence

By this we have finished with the basic steps of structuring our suite.

## 20.6 Running the whole test suite

Finally let's execute our newly created suite:

Action

- Terminate the SUT client in case it is running.
  - Select the root node of the test suite.
  - Run it by pressing "Replay" ► or typing Return).

The SUT is expected to appear, the test case will be executed and finally the SUT will be terminated.

We know the test run details can be looked up in the run log.

Action • It can be opened by clicking the = toolbar button or via the  $\underline{\text{Run} \rightarrow 1...}$  menu option with its short-cut  $\underline{\text{Ctrl-L}}$ 

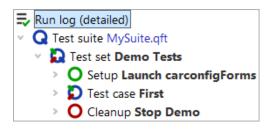


Figure 20.14: The Run log of the test suite

In the first tutorial chapter we've already learned how to use the run log for error analysis.

# **Chapter 21**

# Writing a Procedure (Win)

#### Video This chapter is also available as a video tutorial at



Action

"Writing a Procedure" https://www.qfs.de/en/yt/tutorial-3.html

In the two previous chapters you learned how to start an application from QF-Test thus being able to record mouse and keyboard actions, add checks and organize the result in a test case. This approach is fine and sufficient as long as your tests are simple and you have just a few of them. But as soon as the number of tests increases it is important to make use of so called 'procedures'.

Procedures make sequences reusable and therefore avoid duplicated identical parts. This is important for easy and on the long run efficient maintenance of your tests.

Procedures can be organized into packages = . Procedures and packages are the basis for modularizing your tests.

## 21.1 Identifying reusable parts

In this chapter we will work with the test suite FirstWinTests.qft you already know from chapter 19.

- Copy FirstWinTests.qft from the subdirectory qftest-7.1.3/doc/tutorial
  of the QF-Test installation to a working directory and
  - open FirstWinTests.qft.
  - If you want to keep the changes you will be making to the demo test suite **save it** in a working directory as described at the end of section 20.1<sup>(220)</sup>.

Please have a look at the test step "Reset" in the two test cases. They are exactly the same.

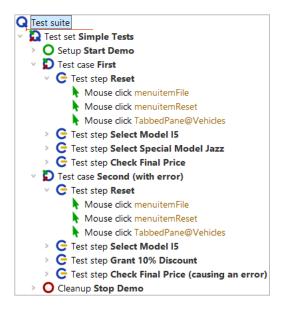


Figure 21.1: Two identical test steps

Following above concepts it would be a good idea to turn them into a procedure.

## 21.2 Manual creation of procedures

There is more than one way to create procedures and insert procedure calls. We will start with the manual one by inserting an (empty) procedure node and moving the respective actions into it. Then we will create the respective procedure call.

It is good to know those basic steps but there is a second more elegant way of creating procedures, which we will explain afterwards.

Okay, let's do it by hand. We will start with creating a procedure node and naming it appropriately.

Open the Procedures node and keep it selected.

Action

- Chose Insert-Procedure nodes-Procedure.
- Name it 'reset'. The other fields can be left empty.
- Press the **OK button** to finalize the creation of the procedure.

Action

· Open the newly created 'reset' procedure node.

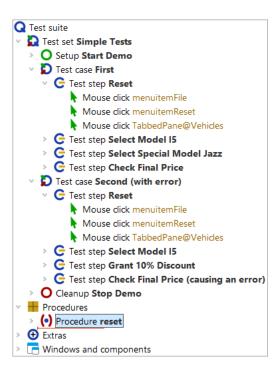


Figure 21.2: Create a procedure node

The second step is to fill the procedure with the respective reusable actions.

- Select the three 'Mouse click' nodes in the test step. To select more than one node at once you can select the first one, then press the Shift key and, while keeping it pressed, click the last node you want to select.
  - Move them down into the procedure e.g. by mouse (drag&drop) or cut/paste from the Edit or context menu.



Figure 21.3: Fill in the procedure content

The third step is to add a procedure call to the place of the three 'Mouse clicks' you moved.

- Action
- Select the test step 'Reset', which is still open.
  - Select Insert $\rightarrow$  Procedure nodes $\rightarrow$  Procedure call or use the Ctrl-A shortcut.

Procedure call		Х
(•) Procedure name		^
Variable for return value		
Local variable		
🕂 🏒 📩 🕂 🦊 Variab	le definitions	
Name Value		
QF-Test ID		
Delay before (ms)	Delay after (ms)	
Comment		
		~
OK	Cancel	

Figure 21.4: Insert a procedure call

- On the dialog **press** the procedure selection button (•) left of the label 'Procedure name'.
  - Select 'reset' from your test suites procedures. Other fields can be left as they are.
  - Press the **OK** button on both dialogs to finalize the creation of the procedure call.

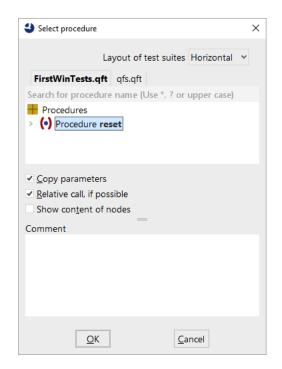


Figure 21.5: Select a procedure

In order get a real benefit from the procedure, of course, we also need to replace the content of reset test step in the second test case by the 'reset' procedure call, too.

You can do it the same way as before or use the following **alternative steps** to create a procedure call:

- **Open** the 'Reset' test step of the second test case.
  - Remove the three 'Mouse click' action nodes therein.
  - Select the 'reset' procedure node.
  - **Move** it via drag&drop into the 'Reset' test step (copy/paste action can be used alternatively). This does not actually move it but create a respective procedure call.

The test suite should look like this:

Action

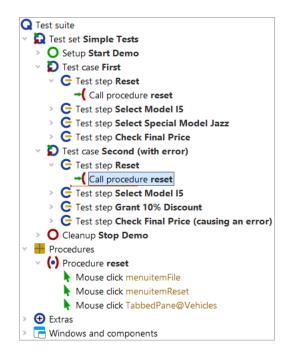


Figure 21.6: Test-suite with procedure

When now executing the test cases the reset in meant to still work like before. Hence, in the run log you will see the same executed node as before, only preceded by the procedure call.

## 21.3 Transforming nodes into procedures

As already mentioned at the beginning of the last section, QF-Test offers an alternative, much faster way to create a procedure:

Action

- Select the test step or sequence node that contains the reusable steps to be transformed into a procedure.
  - Select menu item Operations→Transform node into→Procedure or use the Ctrl-Shift-P shortcut.

You will find that the test step, respectively sequence node disappeared and there is a procedure call in its place. Moreover, a procedure was created in the Procedures section containing the child nodes of the former test step / sequence node and named the same.

It is good practice with QF-Test to record a sequence, give it a name and immediately turn it into a procedure via <u>Ctrl-Shift-P</u> if you suspect it to be of use somewhere else, too.

# **Chapter 22**

# **Components (Win)**

Let us have a look at the last main section in the test suite window, the Windows and components node. When talking about components we also want to show you how to address subitems of components like tables, trees and lists.

Video This chapter is also available as a video tutorial at



"Components" https://www.gfs.de/en/yt/tutorial-4.html

## 22.1 Addressing subitems of tables, lists and trees

Subcomponents of tables, lists and trees can be addressed by indices. There are two main types: text and numeric indices. Let's record a mouse click to a table cell and analyze the recorded QF-Test component ID.

Action

- Start the CarConfig application, in case it has not been started already, by executing the Setup node of the test suite.
  - Activate the recording mode by pressing the toolbar button lacksquare .
  - Click a table cell, e.g. the first model.
  - Stop the recording by pressing

You will see the recorded mouse click in the Extras section.



Figure 22.1: Addressing a table cell

The QF-Test component ID recorded is VehicleTable@Model&0. The single parts are:

- VehicleTable is the QF-Test component ID of the table component itself.
- @ and & separate the single parts from each other and at the same time indicate the type of index that follows: @ for a text index and & for a numeric index.
- Model is the text index for the table column with the title 'Model'.
- 0 is the numeric index for the first table row.

#### Note Numeric indices always start at 0.

You can use any index type for column or row. It is just important that the separator and the index type match.

Action • Change the QF-Test component ID so that the third price value in the table will be clicked, using numeric indices only.

The solution is to type <code>VehicleTable&1&2</code> in the QF-Test component ID attribute of the Mouse event node.

In order to address the model 'I5' using text indices only, you would have to enter VehicleTable@Model@I5. Using numeric indices you would write VehicleTable&O&4 and for mixed indices VehicleTable&O@I5 or VehicleTable@Model&4.

The third type of index QF-Test supports is a text index containing a regular expression. Regular expressions can be used to replace a string by an expression that can match more than only one string. For a detailed explanation of regular expressions please refer to the manual. So you could also address the cell for the model 'I5' using VehicleTable@Model%I.\*.

Lists are addressed the same way as tables are, just with one index only.

A tree has only one index. This is the path down the tree to the node to be specified. The path consists of the respective nodes of the tree, separated by slashes ('/'). Let's record a mouse click to a tree:

- Action Start the CarConfig application, in case it has not been started already, by executing the Setup node of the test suite.
  - Navigate to the tree: In the CarConfig application select the menu item Options -> Specials..., select a model and press 'Details'
  - Activate recording mode by pressing the toolbar button
  - Click a tree node, e.g. 'Description'.

#### 22.1. Addressing subitems of tables, lists and trees

#### • Stop the recording by pressing

For the tree node 'Description' the recorded QF-Test component ID would be DetailsTree@/Information/Description. The single parts are:

- DetailsTree is the QF-Test component ID of the tree component itself.
- @ separates the QF-Test component ID of the tree from the index and at the same time indicates the type of index that follows: @ for a text index.
- /Information/Description is the text index for the tree path to 'Description'.

If you wanted to address the node using a numeric index you would have to use DetailsTree&/0/1.

# **Chapter 23**

# **Using the Debugger (Win)**

In this chapter we will learn how to run a test suite with QF-Test's built-in and intuitive debugger. If you are familiar with debugging from other IDEs like Eclipse, you will find this debugger similar in function and usefulness.

Video This chapter is also available as a video tutorial at



"Using the Debugger" https://www.qfs.de/en/yt/tutorial-5.html

By the end of this chapter you will be familiar with the following debugger functionality:

- Setting a Breakpoint<sup>(246)</sup> e.g. via (Ctrl-F8).
- Pausing a test run at any time and resuming operation using the debugger button
   or the hotkey <u>Alt-F12</u>.
- Stepping Through a Test or Sequence<sup>(247)</sup> using the debugger buttons 'Single step'
   Note: Not
- Skipping Execution of Nodes<sup>(249)</sup> using the debugger buttons 'Skip over' <sup>♥</sup> and 'Skip out' <sup>♥</sup>.
- Error or Exception triggering Debugging Mode<sup>(251)</sup>.
- Resolving Errors directly from the Run log<sup>(252)</sup>.
- Jump directly to the current error in the run log via Ctrl-J. (Jump to run log in chapter section 23.5<sup>(252)</sup>).
- Note Instead of the debugger buttons you can also enter the commands via the QF-Test menu and most by keyboard shortcuts as well. You'll find the the shortcut listed beside the menu option, if available. For a complete list, refer to the Keyboard shortcuts section

of the user manual. You can also find a little helper there for attaching to your keyboard which shows the function key assignment of QF-Test.

There are some more functions related to the debugger that we will come to in later chapters:

- Locating the current node using the debugger button 

   Icocate the current node in chapter section 24.3<sup>(264)</sup>).
- "Continue execution from here" via the popup menu of the respective node (figure  $24.9^{(266)}$ ).
- Rethrowing exceptions using the toolbar button  $\checkmark$  .
- The variable bindings table (section 24.3<sup>(264)</sup>).

## 23.1 Setting a Breakpoint

First of all we need to activate debugging mode. There are several ways to do so. One of them is to set a breakpoint at the node where we want to have a closer look. When the test is being executed and QF-Test comes to the break point it will then pause and switch into debugging mode. The pause button III will then be activated.

• Select a node and **press Ctrl-F8**. The breakpoint is indicated by a **B**.



Figure 23.1: Set break point

• Select the Test suite node and **press Enter** to start the test run.

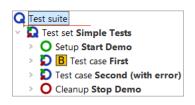


Figure 23.2: Start test run

Action

Remove the breakpoint by pressing Ctrl-F8 again.

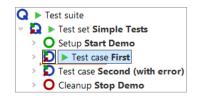


Figure 23.3: Remove break point

Instead of using the keyboard shortcut Ctrl-F8 you may also set or unset a breakpoint by clicking the node and selecting the Debugger $\rightarrow$ Breakpoint on/off menu item, or alternatively right-clicking the node and selecting the Breakpoint on/off context menu item.

Again, you can see the little arrow, which now marks the next node to be executed, called the **current node**. When entering debugging mode QF-Test also navigates to the current node, in case it had not been visible, and selects it, highlighting it blue.

The menu option Debugger $\rightarrow$ Clear all breakpoints is useful to remove all breakpoints set in your test suite.

There is no limit to the number of breakpoints you can set in your test suite, but note that breakpoints are not saved with the test suite.

## 23.2 Stepping Through a Test or Sequence

Now let's step through the test case we set up in the previous section.

Action

Please try out the debugger buttons Single step <sup>™</sup>, Step over <sup>™</sup> and Step out
 <sup>™</sup>.

You will find that **Single step**  $\mathfrak{D}$  opens a node containing child nodes and makes the first child node the active node. Continuing from where we left the test suite at the end of the last section, i.e. in debugging mode, with 'Test case: First' being the current node, the test suite would now look like this:

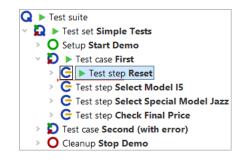


Figure 23.4: Stepping into a node

In the case of leaf nodes (nodes without child nodes), the effect of  $\mathfrak{D}$  is the same as the following button's.

**Step over**  $\bigcirc$  runs the current node including all children. Execution pauses at the next node of the same level to be executed, which then becomes the active one.

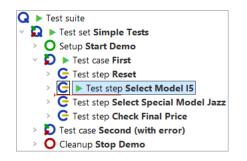


Figure 23.5: Stepping over a node

**Step out**  $\checkmark$  runs the remaining nodes at the same level including their child nodes. Execution pauses when a node that is higher in the hierarchical structure is found, which then becomes the active one.

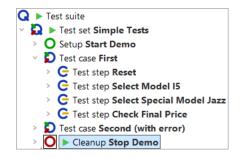


Figure 23.6: Stepping out of a node

#### 23.3. Skipping Execution of Nodes

In the given example the node higher in the hierarchical structure where execution stops is the Cleanup node. As explained in the chapter <u>A full Test Run<sup>(216)</sup></u> this shows the special behavior of Setup / Cleanup nodes in a test set: They are executed before and after **each test case** to help achieving a proper starting state for each test case.

- Note You will only find this behavior when you started the whole test suite or test set and are in debugging mode. If you just selected the test case and did a step-over action then QF-Test will execute the test case and then select the next test case node.
- Run the Cleanup and Setup nodes by stepping over them, using the debugger button **Step over** <sup>♥</sup> and then **Step in** the second test case via <sup>♥</sup> to get ready for the next section where you will learn about the skip functionality.
- Note Please be aware that menus or comboboxes tend to close when the application looses the focus, as will happen when activating the debugging mode. In such a case you should not stop test execution between the node opening the menu or combobox and the node performing the selection. One way to do achieve this is to set a break point **B** after the node performing the selection and to activate normal test execution by releasing the pause button **II** when you reach the node opening the menu or combobox.

## 23.3 Skipping Execution of Nodes

The "skip" functions expand the QF-Test debugger's capabilities 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 them. This may be helpful for various reasons, e.g. to quickly navigate to a certain position in your test run or to skip a node which currently leads to an error.

With the end of the last section, the test suite should have reached the following state:

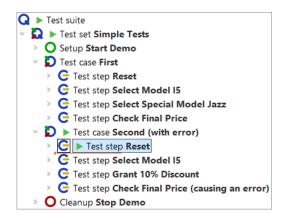


Figure 23.7: Pause execution at first node of the second test case

#### 23.3. Skipping Execution of Nodes

Action • Now press the Skip over <sup>♥</sup> button. QF-Test simply jumps over the active node without executing any child nodes. The active node now is the next node to be executed on the same level.

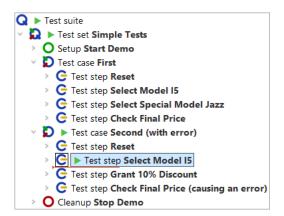


Figure 23.8: Skip over a node

• At last, press the **Skip out** <sup>𝒞</sup> button. You see that QF-Test skips all nodes on the same (or lower) level and directly jumps to the next node one level up in hierarchy.

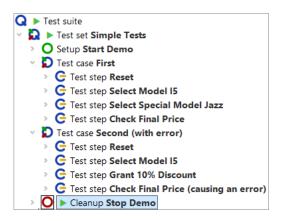


Figure 23.9: Skip out of a node

Note Use "Skip over" and "Skip out" cautiously as skipping out of a sequence before it is completed can leave the SUT in an unknown state that other sequences or tests in your test suite cannot react to.

Action

## 23.4 Error or Exception triggering Debugging Mode

When debugging a test you may want run it until it encounters an error, an exception or sometimes even a warning and then have it pause in debugging mode.

In this section you will see how this can be done while debugging the second test case.

- Please open the debugger menu and change the default options as follows:

  - Also activate the Debugger→Options→Break on error menu item.

Afterwards, when you open the debugger menu and options submenu it should look like this:

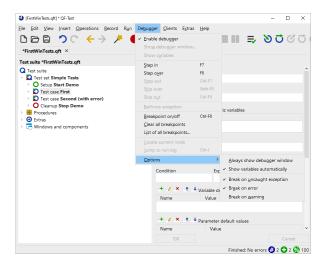


Figure 23.10: Set debugger options to pause on error

We changed the debugger options because with default settings QF-Test will not pause on exceptions or errors, as you saw earlier on.

Action • Select the "Test-suite" node and start test execution via "Start test run" ▶.

QF-Test will pause at the faulty node and enter debugging mode:

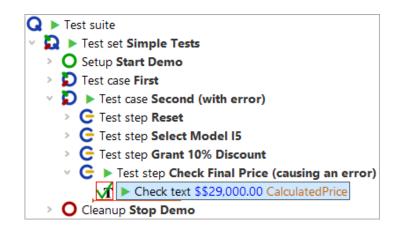
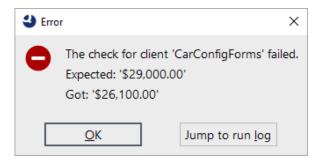
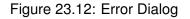


Figure 23.11: Test run stopped by error

The node which halted test execution will be indicated by an arrow and its symbol will be surrounded by a red square. Also, an error dialog will inform you about the failed check.

As always the run log is the key to resolving errors, so let's open it and find out how to resolve the error in the next section.





Action

• Click the Jump to run log button in the error dialog.

## 23.5 Resolving Errors directly from the Run log

The **Jump to run log** button from the dialog in <u>figure 23.12<sup>(252)</sup></u> will not only open the run log but takes us directly to the node that holds the error details. Apart from the actual error message you will find screenshots and a copy of the variable bindings table (stack trace), which we will introduce later on (The Variable Bindings table<sup>(260)</sup>).

The error details tell you that the expected value does not match with the one shown in

#### 23.5. Resolving Errors directly from the Run log

the application. As the one in the application is correct we want to update the expected value with the one from the application. This can easily be achieved as follows:

- Action
- Right-click the red-bordered node "Failed: Check text: default ..." indicating the actual error
  - Select Update check node with current data from the context menu.

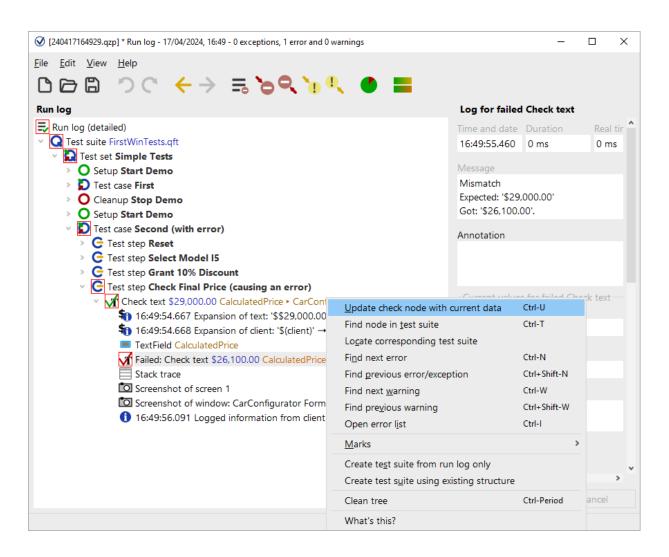


Figure 23.13: Update check node with current data

This locates the corresponding Check text node in the test suite and updates the expected value of the Text attribute with the value got as indicated by the run log.

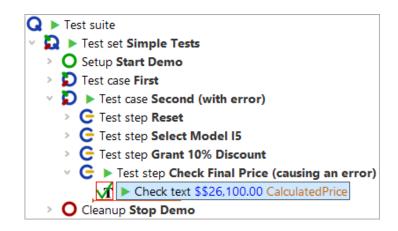


Figure 23.14: Corrected check node

The previously faulty node still is highlighted with a red border since we have not run it again.

#### • Now continue execution by releasing the pause button II.

QF-Test runs the rest of the test suite, i.e. the Check text and Cleanup nodes, and informs you at the end of the run that there was one error, which you have already fixed.

Since the error has been fixed and we will continue using the test cases as examples you could rename the second test case and delete '(with error)' in its name as well as '(causing Error)' in the name of the test step.

**Jump into run log:** You do not have to wait for an error dialog to open the run log at the current point of execution (or close by). Whenever you are in debugging mode, select the **Debugger** $\rightarrow$ **Jump to Run log** menu option, or use the **Ctrl-J** shortcut. If you just want to open the run log without jumping to the current point of execution you can use **Ctrl-L**. This will work after the test run finished, too.

### 23.6 Pause Execution

When a test is being executed and you want to enter debugging mode you can quickly set a breakpoint at some node not yet executed. Or you can just hit the toolbar button "Pause" III and QF-Test will directly enter debugging mode.

In order to resume execution just release the pause button **II**. This is completely independent of the way you entered debugging mode.

Depending on how focus demanding the SUT is, it may be difficult to focus QF-Test long enough to hit the pause button. But you can still rely on the "Don't Panic" short-

cut <u>Alt-F12</u>. It will pause all running tests immediately. To continue, press the same combination again.

# **Chapter 24**

# Variables and Procedure Parameters (Win)

In this chapter you are going to learn how to use a procedure to perform the same action on various input data. You will also learn about variables - how to use and how to debug them.

Video This chapter is also available as a video tutorial at



. The video uses a Java application for demonstration. Variables are used in the same way with web applications

### 24.1 Procedure using a variable

Have a look at the last test step 'Check final price' of our two test cases.

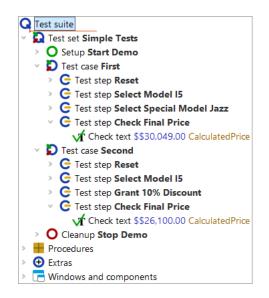


Figure 24.1: Two almost identical test steps

They perform the same action, however, with different data. Even though it is only one node, it makes sense to pack it into a procedure. We may want to adapt the hard coded values  $30,049.00 \in$  and  $26,100.00 \in$  to a different format so that the check will also work when the format of the price field changes to a different currency. And we do not want to implement the same algorithm twice. (If you wonder about the the two dollar signs: This is what QF-Test records for a dollar sign. For replay it does not matter whether you use one or two dollar signs.)

- Select the 'Check text' node in the first test case.

  - Name the sequence checkFinalPrice. The procedure name follows the Java convention to run the words together and start the single words with capital letters. On the other hand QF-Test allows the use of spaces in procedure names, so you are free to name it as you like.
  - Press **Ctrl-Shift-P** for the quickest way to transform the 'Sequence' node into a procedure, as you learned at the end of the last chapter. You see the sequence is replaced by a call to the 'checkFinalPrice' procedure.
  - **Double click** the procedure call node to jump to the procedure definition.
  - **Open** the procedure node to see its content.

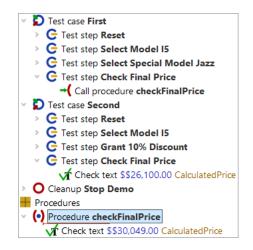


Figure 24.2: Procedure with hard coded value

As expected, the check is now located in this procedure. However, it is valid for one price only, i.e. \$30,049.00. Since we want to use the same procedure for the second test case as well we need to make the price a variable and pass its value as a parameter from the test case to the procedure.

In the next example we will start by defining a parameter for the procedure. Additionally, we will set a default value for the parameter. Default values are most useful when the parameter usually has that value and you do not want to specify it every time you call the procedure. Even though this does not hold true for the price it is a good example to demonstrate to you how a default parameter works and how to overwrite it with another value.

Let's define the parameter and add a default value:

#### Action • Select the procedure 'checkFinalPrice'

- **Press** the <sup>d</sup> "Add new row" button belonging to the table 'Parameter default values'.
- Enter price as name for the parameter.
- Enter \$ 30,049.00 in the value field.
- Click the 'OK' button.

Procedure			
Name			
checkFinalPrice			
+ / × + +	Parameter	default values	
Name	Value		
price	\$30,049.0	0	
Maximum error leve Exception QF-Test ID	21		~
Delay before (ms)		Delay after (ms)	
Comment			

Figure 24.3: The Details of the 'Procedure' node

The next step is to replace the value of the Text attribute of the Check text node by a reference to the variable.

Note Variable syntax: When working with variables you need to bear in mind that in some places you need to tell QF-Test the name of the variable and in others you want to refer to the value of the variable. In the Name column of the Parameter default values table of the Procedure node QF-Test expects the name of a variable. It is price, which is why you typed the word price.

In the Text attribute of the Check text node details QF-Test expects a character string for comparison with the text of the UI element. As we want to use a value stored in a variable, we have to tell QF-Test not to use the entered string as plain text, but to interpret it as a reference to the value stored in a variable. In QF-Test this is done by enclosing the variable name in \$(). In our case the variable reference is \$(price). If you did not put the variable name into \$(), QF-Test would compare the price (a number) shown in the UI element to the string price, which is obviously nonsense.

- Select the Check text node in the procedure 'checkFinalPrice'.
  - **Type \$ (price)** in the Text attribute of the Check text node details.
  - Click the 'OK' button of the node details.

Action

Check text	
Client	
\$(client)	
📑 QF-Test component ID	
CalculatedPrice	
Text	
\$(price)	
\$ As regexp	
\$ Negate	
Check type identifier default	
Timeout	
Result handling	
Variable for result	
Local variable	
Error level of message	
Error	~
\$ Throw exception on faile	ure
Name	
QF-Test ID	
Delay before (ms)	Delay after (ms)
Comment	

Figure 24.4: 'Check text' node

#### Action • • Run the first test case.

It should execute without an error.

## 24.2 The Variable Bindings table

The next step is to make use of the procedure call in the second test case as well.

#### 24.2. The Variable Bindings table

Replace the Check text node of the second test case by a procedure call to checkFinalPrice. You can simply copy the respective node from the first test case or add the procedure call as learned in the previous chapter.

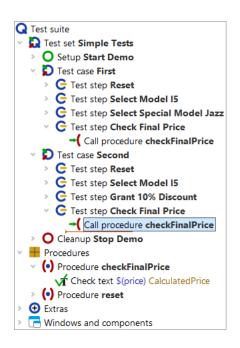


Figure 24.5: Second procedure calls 'checkFinalPrice'

- Note If you added the procedure call by a copy or drag and drop operation from the procedure itself you will find the price in the Variable definitions table of the procedure call. This is what we are eventually aiming at. However, at this stage we want to explain the default value. So, if you want to follow the tutorial exactly, please delete the default value by pressing the red X above the table.
- Verify **QF-Test is configured to pause at errors** as shown in <u>Set debugger</u> options to pause on error<sup>(251)</sup>.
  - Select the 'Test case: Second' node.
  - Execute it by pressing ▶ or Enter.

An error message shows up indicating different values for the price expected and the price got. What went wrong? Let's go hunting. Typically we use the run log for this but there is another view worth to know of.

• So click OK to close the error message.

#### 24.2. The Variable Bindings table

In debugging mode you will find an additional bottom right section of the QF-Test window showing a list of nodes with variables bound to those nodes.

- Action
- You might want to resize the variable bindings table in case it is too small to see all its content.

Variable bindings				
Node	Test suite	Bindings	_+ _/ _×	📑 🚽 Selected variable definition
(•) Procedure checkFinalPrice	FirstWinTests.qft	0	Name	Value
Call procedure checkFinalPi	FirstWinTests.qft	0	price	\$30,049.00
C Test step Check Final Price	FirstWinTests.qft	0		_
D Test case Second	FirstWinTests.qft	0		
Globals		1		
Command line		3		
Q Test suite	FirstWinTests.qft	0		
Fallback stack		0		
• Procedure checkFinalPrice	FirstWinTests.qft	1		
System		0		

Figure 24.6: Variable bindings

The variable bindings table is very useful for debugging. It comes in quite handy, too, when working with procedures and trying to understand the way QF-Test figures out which variable value to use. It shows the current values of the variables.

**Note** QF-Test always checks the variable bindings table from top to bottom.

You can see that the first rows of the table have no bindings at all. Then there is a binding at the level 'Globals' and another one in the fallback stack for the procedure 'checkFinalPrice'. The global variable is used for the client connection, which has been set when starting the application (cf <u>Starting the Application</u><sup>(206)</sup>). The other variable is more interesting to us - only it has the wrong value.

The default value is intended to be used for the parameter if no value has been defined elsewhere. This is why we added the parameter to the 'Parameter default values' table of the procedure node.

To do things correctly we need to pass the proper value when calling the procedure. Again, there are several ways to do it. One is to add a new row to the variable definitions table of the 'Call procedure' nodes similarly to the way you did at the 'Procedure' node in the last section.

If the procedure is called multiple times within the test suite, there is a better way:

- Action
- Stop the current test execution clicking the toolbar button 🔳 .
  - Right-click the 'Procedure' node and select
     Additional node operations→Update parameters of references
     from the popup menu.

(FirstWinTests.qft) * QF-Test				- 0 X
	ations <u>R</u> ecord R <u>u</u> n De <u>b</u> u	gger <u>C</u> lients	Extras Help	
				<b>७७८</b> ८०
*FirstWinTests.qft ×				
Test suite *FirstWinTests.qft		Procedu	re	
Q Test suite		Name		
<ul> <li>Dest set Simple Tests</li> <li>O Setup Start Demo</li> </ul>		checkFinal	Price	
Setup Start Demo     Setup Start Demo	Cut	Ctrl-X	↑ → Parameter default values	
> C Test step Reset	C <u>u</u> t <u>C</u> opy	Ctrl-C	Value	
> 🧲 Test step Select I	<u>Paste</u>	Ctrl-V		
> G Test step Select :	Delete	Delete	Analyze references	
Test step Check I Call procedure	Edit	Alt-Enter	Analyze <u>d</u> uplicates	
<ul> <li>Call procedure</li> <li>Test case Second</li> </ul>	Marks	>	Ana <u>l</u> yze invalid characters	
G Test step Reset	—			
> C Test step Select I	Start run		Make calls relative	
> C Test step Grant 1	Toggle disabled state	Ctrl+Shift-D	M <u>a</u> ke calls absolute	
C Test step Check I	Breakpoint on/off	Ctrl-F8	Make references e <u>x</u> plicit	
Call procedure Call procedure Cleanup Stop Demc	Locate references	Ctrl-I Ctrl-D	Make references implicit	
Procedures	Add to boo <u>k</u> marks		Update parameters of references	
<ul> <li>Procedure checkFin;</li> </ul>	Pack n <u>o</u> des	>	- ard <u></u> etenzen	
Check text \$(price	Additional node operations	>		
> (•) Procedure reset	<u>I</u> nsert node	>	nt	
<ul> <li>         • Extras     </li> <li>         • Windows and componer     </li> </ul>	<u>N</u> ew window			
- Up windows and componer -	Expand all	Alt-Right		
	Collapse all	Alt-Left		
	Clean tree	Ctrl-Period		Cancel
	What's this?		Finished: 0 exceptions, 1 error, 0 warning	as 🤀 1 🔂 1 🧖 10

Figure 24.7: Popup menu for 'Additional node operations'

• In the following dialog, please check the tick mark for **Add missing parameters to callers** is set and **click OK**.

In the 'Call procedure' nodes QF-Test adds a row each for every default parameter to the variable definition table of the procedure call. It also copies the default value of each default parameter. In our case it is the parameter price with the value \$30,049.00.

You might notice that the numerical value of the price variable is still wrong in the second case, regardless of whether it is defined implicitly as a default value or explicitly via a parameter. For now we want to keep that error to show you additional means of debugging.

 Close the 'Updated nodes' dialog QF-Test opened to inform you about the updated nodes.

## 24.3 Advanced debugging of variable bindings

Next, we want to explore the variable bindings table and see how it can be used for debugging purposes. For this reason do not correct the faulty value of the procedure call we added in the last section, but let us find out more about debugging.

In the next steps we want to have QF-Test pause the test execution at the procedure call. Then we will step into the procedure and, while doing so, have a look at the variable bindings table. Finally, we will navigate directly from the variable bindings table to the faulty procedure call and correct the parameter value.

- Add a breakpoint to the 'Call procedure: checkFinalPrice' node of the second test case.
  - Run the second test case again.

Action

• When QF-Test stops at the breakpoint step into the procedure via <sup>™</sup> and watch the table of the variable bindings as you do so.

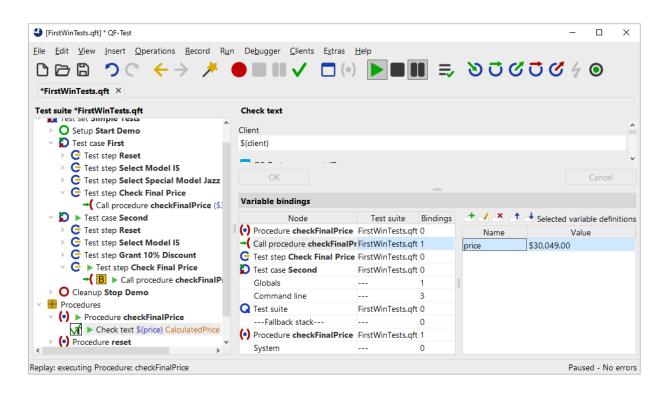


Figure 24.8: Variable bindings stack showing incorrect value

As you step into the procedure, first the row 'Call procedure: checkFinalPrice' and with the next step the row 'Procedure: checkFinalPrice' appear at the top of the table.

#### 24.3. Advanced debugging of variable bindings

Now the variable price shows on two different levels of the variable bindings table: In the 'Call procedure: checkFinalPrice' row and the 'Procedure: checkFinalPrice' row on the fallback stack. Since we did not adapt the value of the parameter passed to the procedure, neither of the two values bound to the price variable is correct.

QF-Test lets you change the values of the variables interactively in the variable bindings table when you are in debugging mode. You can even add new variables or delete them. However, changes to variable values in the variable bindings table are not persistent. They only last as long as the variable is on the stack (variable bindings table). In our case, if we changed the price value, as long as the procedure is being executed.

The parameter value in the procedure call will not be altered by changing the current value of the variable in the variable bindings table. To do so you have to navigate to the Procedure call node and change it there.

To get there quickly, you just double-click the procedure call in the variable binding table (second row). This feature is particularly useful when debugging more complex tests where the node you want to jump to is not directly visible in the test suite window. You can invoke it via right-clicking the row and selecting Jump to node in test suite from the pop-up menu, too.

- Action Double-click the second row with the procedure call in the variable bindings table.
  - Set the value of the parameter to the correct value, i.e. \$ 26,100.00.

When checking the variable bindings table you will notice that the current value has not changed. This is hardly surprising as we have not yet executed the procedure call again. Only, test execution is already past the procedure call. Fortunately, QF-Test has another very useful debugging feature to set back (or forward) test execution to some node: **Continue execution from here**, which can be invoked either via the popup menu of the node you want to make the current node or by pressing Ctrl-, after selecting the node.

In order to try out the newly set value:

Action

- Right-click the 'Call procedure: checkFinalPrice' node of the second procedure.
  - Select 'Continue execution from here' in the popup menu.

FirstWinTests.qft] * QF-Test							-		×
	ions <u>R</u> ecord R <u>u</u> n De <u>b</u> ugge		_	ras <u>H</u> elp		≡,	৯ ব	<b>S</b> (2	Ū
Test suite *FirstWinTests.qft		Procedure	call	I					
est suite Test set <b>Simple Tests</b> O Setup <b>Start Demo</b> D Test case <b>First</b>		(•) Procedur checkFinalPri	ice						
<ul> <li>Cut</li> <li>Cut</li> <li>Cut</li> <li>Copy</li> <li>Paste</li> <li>Call procedure check</li> <li>Call procedure</li> <li>Call procedu</li></ul>	Ctrl-X Ctrl-C Ctrl-V Delete Alt-Enter	_	↓ Variabl Valu	e definitions ue					
		Ctrl+Shift-D Ctrl-F8 Ctrl-P		<b>1gs</b> <sup>•</sup> est <sup>•</sup> irst 0 <sup>•</sup> irst 1	Name	Selected var		Cancel riable defini alue	
	– Trans <u>f</u> orm node into Pack n <u>o</u> des	Ctrl-D	> > >	First 0 First 0 1 3	-				
Windows and components	Continue execution from <u>h</u> ere Insert node Clean tree	Ctrl-Comma Ctrl-Period	>	First 0 0 First 1 0					
hanges applied	What's this?						Davia	ed - No	

Figure 24.9: Continue Execution from here

When checking the variable bindings table you will find that the top two rows have disappeared. This is because now we exited the procedure (even though 'backwards') and therefore the procedure call and all its variable bindings was taken off the call stack.

Action • Release the pause button II.

The test run should finish without error.

Note As the variable bindings table is very useful when looking for incorrect variable values you will also find a copy of it in the run log whenever an error or exception is logged. It is written to the subnode 'Stack trace' of the node causing the error, showing the variable values at the time the error occurred.

**Locate the current Node:** Sometimes during debugging you will navigate far away from the current node where execution stopped and eventually want to get back to it again.

The easiest way to do so is by pressing the "Locate Current Node"  $\bigcirc$  button or select the Debugger $\rightarrow$ Locate Current Node menu option to cause the debugger to "select" the current node.

## 24.4 Setting Variables

In addition to the methods you have already seen, variables can also be set as follows:

- Via the Set variable node,
- as the return value of a procedure,
- as the result value of QF-Test nodes like the Fetch text node, the Fetch geometry node, the Fetch index node and the Check node,
- in the 'Variable definitions' table of the 'Test-suite' node, the 'Test case' node, the 'Test step' node, the 'Sequence' node and others like the 'If' node or 'Loop' node,
- via command line parameters.

For information about the best place where to define a variable please refer to the next section.

You can insert a Set variable node via the menu item  $\boxed{\text{Insert} \rightarrow \text{Miscellaneous} \rightarrow \text{Set variable}}$ . In its details you can specify whether the value should be bound as a local or a global variable.

The following figure shows the details of a Set variable node. (You can find it as the first node of the Setup node.) It defines a variable named client. It is a global variable as the Local variable attribute has not been checked.

Set variable		
Variable name		
client		
Local variable		
Default value		
CarConfigForms		
\$ Interactive		
Description		
Timeout		
lineout		
QF-Test ID		
Delay before (ms)	Delay after (ms)	
Comment		

Figure 24.10: Details of the Set variable node

When you want to set a variable as the result of a procedure call you need to specify the variable name in the 'Variable for return value' attribute of the procedure call. Within the procedure itself you have to add a Return node with the value to be returned as the last node to be executed.

The next figure shows a theoretical example of a procedure which returns a value. The procedure fetches the discount value displayed in the SUT and returns it to the calling test case. There, the receiving variable is named Discount and declared as a local variable.

[FirstWinTests.qft] * QF-Test	- 🗆 X
Eile       Edit       View       Insert       Operations       Record       Run  </th <th>Debugger <u>C</u>lients Extras <u>H</u>elp</th>	Debugger <u>C</u> lients Extras <u>H</u> elp
Test suite *FirstWinTests.qft	Procedure call
<ul> <li>Q Test suite</li> <li>C Test set Simple Tests</li> <li>Setup Start Demo</li> <li>Test case First</li> <li>Test case Second (with error)</li> <li>Test case Third</li> <li>C Cleanup Stop Demo</li> <li>C Cleanup Stop Demo</li> <li>Procedure getDiscount</li> <li>ST Fetch text DiscountValue → returnValue</li> <li>Return S(returnValue)</li> <li>Extras</li> <li>Windows and components</li> </ul>	Procedure name getDiscount Variable for return value Discount I Local variable I Local variable I Variable definitions Name Value OK Cancel

Figure 24.11: Procedure returning a value

The Fetch text node of above example is one of the QF-Test nodes directly setting a variable value. You need to specify the variable name in the attribute called accordingly. Again, you have the choice whether to make it local or global.

Quite a number of nodes have a 'Variable definitions' table where you may define variables local to a procedure or a test case. If the respective node is part of a procedure the variable will be local to the procedure. Otherwise it will be local to the respective test case. Variables bound to the test suite node can be accessed from all nodes within the test suite.

In debugging mode, all nodes you can bind a variable to will show up on the variable bindings table when entered.

You can enter variables in the command line via the parameter -variable. For details please refer to the manual, chapter 'Command line arguments and exit codes'.

## 24.5 Variable binding levels

Note This section may be difficult to understand when you are a programming beginner. Then it may be better to come back to it when you started writing procedures for your own tests.

In QF-Test there are many places where you can set a variable:

- The test suite node,
- in test cases and procedures as default or local variables,
- as a parameter in a procedure call,
- · as a global variable and
- as a command line parameter.

Now the question is: Which is the correct place for defining a variable?

And the answer is: It depends on the use case.

Each level of the variable bindings has its own use case:

#### **Procedure parameters**

When you call the same procedure several times in a test case and with different values each time you should set the values via a parameter at a procedure call. This is done by adding a row to the variable definitions table of a Procedure call node specifying the respective variable name and value.

#### Local variables in a procedure

They are created in the procedure and are deleted when the execution of the procedure finishes. Use a local variable when you do not need it outside of the current procedure. In procedures they are predestined for intermediate results.

#### Local variables in a test case

Variables local to a test case are either created during execution of a test case or defined in the respective table in the details of the test case node. They will be deleted from the variable bindings table once the test case finishes. Use them for values you want to refer to in several nodes of a test case and you do not need outside of it.

#### **Global variables**

Global variables are created at some point of the test execution and exist until they are explicitly deleted or QF-Test is stopped. They survive stops and restarts of test execution. Use them for values that need to be accessible by all test cases. A typical example is the variable client created in the 'Setup' node when starting the application. To get rid of them you need to either exit QF-Test or select the menu item Run $\rightarrow$ Clear global variables.

#### **Command line parameters**

In batch mode you may want to run test with various parameters. They are valid throughout the batch run. A typical example would be the browser to be used.

### **Test-suite variables**

test suite variables can be referred to by all test cases. Their usage is the same as that of global variables. Only that they can be overridden at batch execution by command line parameters, whereas global variables cannot.

## Default values (Fallback stack)

You can define default values for variables of procedures, test cases and test sets. In case you do not define a variable of the same name on a higher level QF-Test will use the default value set.

## **Chapter 25**

# The Standard Library (Win)

## Video This chapter is also available as a video tutorial at



The Standard Library"

https://www.qfs.de/en/yt/tutorial-7.html

QF-Test provides a certain number of node types. If you need additional functionality you can implement it in a script node. To make life easier for you QF-Test comes with a set of procedures implementing the most commonly needed additional functions. You will find them in the standard procedures library.

When you cannot solve a problem using the provided node types it is a good idea to have a look in the standard library whether there is a solution to your problem. If you find a similar solution you can copy the procedure and adapt it to your needs. For information about scripting please refer the the manual, chapter 12 'Scripting'.

The file qfs.qft holds the standard procedures library. As it is constantly being enhanced and distributed with every new version of QF-Test you should not make any changes to procedures in that file, but copy the procedure to your own test suite if required and then adapt it.

- Note To make use of <code>qfs.qft</code> it needs to be included in your test suite's root node. With a newly created test suite the file <code>qfs.qft</code> is added automatically to the list of included files.
- Select the 'Test-suite' root node of your test suite.
  - Verify the <code>qfs.qft</code> is available within the table for "Include files".
  - Add  ${\tt qfs.qft}$  to this list, if it's not already there.
- Note Path information is not necessary for qfs.qft as the include directory of QF-Test is contained in the library path (see also Reference part of the manual).

Action

• Add a procedure call to an arbitrary procedure from the qfs.qft standard library. In the procedure chooser don't miss to switch to the respective tab.

In addition to the description provided in this tutorial you can find the full HTML documentation of the standard library available via Help-Standard Library qfs.qft...

## 25.1 Inspecting the Standard Library

In addition to inserting procedure calls from the Standard Library, it also can be helpful to sometimes have a look how certain things have been implemented.

- Action
- Locate and load the test suite file <code>qfs.qft</code>, which is located in the <code>qftest-7.1.3/include</code> directory of your QF-Test installation.

🕹 [qfs.qft] QF-Test	- 🗆 X
<u>F</u> ile <u>E</u> dit <u>V</u> iew <u>Insert</u> <u>O</u> perations	<u>R</u> ecord R <u>u</u> n De <u>b</u> ugger <u>C</u> lients E <u>x</u> tras <u>H</u> elp
000 00 <del>&lt; )</del>	> 🗡 🛑 🖩 🖌 ✔ 🗖 (•) 🕨 🖿 🚍 🏷 Ü Č Č Č Ć ϟ 🍥
qfs.qft ×	
Test suite qfs.qft	Test suite
Q Test suite	Name
Procedures	
Package qfs     Package android	🕂 🧹 📩 🕂 Include files
Package automac	File
Package automac     Package autoscreen	File
Package autoscreen	
> Package awt	
> Package cleanup	🛨 🛃 📩 Dependencies (reverse includes)
> + Package client	File
Package check	
> + Package daemon	
> 🕂 Package database	🕂 🟒 🐣 📑 🛃 Variable definitions
> 🕂 Package databinder	
> 🗕 Package dialogs	Name Value
> 🕂 Package <b>fx</b>	
> 🛨 Package jib	
> 🖶 Package pdf	Execution timeout (ms)
Package run-log	
Package shellutils	
V 🖶 Package swing	Comment
> Package cleanup	This suite contains useful standard procedures provided and maintained by Quality First Software GmbH.
> Package checkbox	
> Package combobox	For a description of packages and procedures of this library in javadoc format, see qfs.html in the same directory.
Package dialogs	
Package filechooser v	OK Cancel

Figure 25.1: The Standard Library

### 25.2. Selected Packages and Procedures

You can see there is one main package qfs that contains further specific packages. The qfs package helps to easily identify the packages belonging to the standard library.

The specific packages cover very different areas of utility procedures described in more detail further below.

Note Within nearly all of the procedures of this library, you'll notice that the variable s(client) is referenced. This is the standard mechanism for creating independence from a specific SUT. Here, the library assumes that the test suite which uses the library will set a value for s(client) prior to using any procedures.

## 25.2 Selected Packages and Procedures

We will now have a closer look at a number of selected packages and procedures from the standard library.

## 25.2.1 The Run log Package

The qfs.run-log package contains procedures, which writes specified messages into the run log. This package has been introduced to give testers without scripting-knowledge the opportunity to write messages into the run log.

Here is the list of the most important procedures within this package:

- **logError** Write a given error message into the run log.
- logWarning Write a given warning message into the run log.
- **logMessage** Write a given message into the run log.

## 25.2.2 The Run log.Screenshots Package

The <code>qfs.run-log.screenshots</code> package contains procedures, which write images into the run log and some helper methods.

Some important procedures within this package are:

- getMonitorCount Return the total number of monitors.
- **logScreenshot** Write a screenshot of the whole screen into the run log.
- logImageOfComponent Write an image of a given component into the run log.
- **logScreenshotOfMonitor** Write a screenshot of a given monitor into the run log.

## 25.2.3 The Shellutils Package

The  ${\tt qfs.shellutils}$  package contains procedures to support most common shell-commands.

Some important procedures within this package are:

- **copy** Copy a given file or directory to a specified target.
- deleteFile Delete a given file.
- exists Check for existence of a given file or directory.
- getBasename Return only the file name of a full file name.
- getParentdirectory Return only the directory name of the full file name.
- mkdir Create a given directory. It also creates non-existing directories in path.
- move Move a file of directory.
- touch Create a specified file.
- removeDirectory Remove a specified directory.

## 25.2.4 The Utils Package

The <code>qfs.utils</code> package contains procedures, which covers common helper-functionality during test-development.

Some important procedures within this package are:

- **getDate** Return a string containing the date. Default is the current date. (Other dates can be configured.)
- **getTime** Return a string containing the time. Default is the current time. (Other timestamps can be configured.)
- logMemory Log current memory use.
- printVariable Print the content of a given variable to the console.
- printMessage Print a given message to the console.
- writeMessageIntoFile Write a given string into a given file.

## 25.2.5 The Database Package

The <code>qfs.database</code> package contains procedures to execute SQL commands on a database.

Please note, that the class of the database-driver must be in QF-Test's plugin directory or in the CLASSPATH before QF-Test startup.

To get more information about the connection-mechanism to your database, please ask your developers or see www.connectionstrings.com.

Some important procedures within this package are:

- **executeSelectStatement** Execute a given SQL-Select-Statement. It stores the result in a global variable "resultRows" on the Jython variable stack and thus accessible from Jython scripts. Additionally, it stores the result in a group variable with the default name 'resultGroup', which can be accessed directy by QF-Test nodes.
- executeStatement Execute a given SQL-command. Here any SQL command can be specified.

## 25.2.6 The Check Package

The qfs.check package contains procedures to do checks.

Some important procedures within this package are:

- **checkEnabledStatus** Check, whether a component is enabled or disabled. It writes an error into the run log, if failing.
- **checkSelectedStatus** Check, whether a component is selected or not. It writes an error into the run log, if failing.
- **checkText** Check the text of a component. It writes an error into the run log, if failing.

## 25.2.7 The Databinder Package

The <code>qfs.databinder</code> package contains procedures for execution within a "Data driver" node which bind data for iteration.

Some important procedures within this package are:

• **bindList** Create and register a databinder that binds a list of values to a variable. Variables are separated by whitespace or by a given separator character.

## 25.2. Selected Packages and Procedures

• **bindSets** Create and register a databinder that binds a list of value-sets to a set of variables. Value-sets are separated by line breaks. Variables within a value-set are separated by whitespace or by a given separator character.

# **Chapter 26**

# **Control structures (Win)**

The two most important control structures of QF-Test are loops and the conditional execution of nodes. Loops can be implemented by two different kinds of nodes: While and Loop nodes. If, Elseif and Else nodes are available to implement conditional execution.

leo This chapter is also available as a video tutorial at



"Control Structures" https://www.qfs.de/en/yt/tutorial-8.html

## 26.1 If - else

You already came across If nodes in the Setup sequence in the chapter <u>Starting the</u> Application<sup>(206)</sup>. Let's have a closer look at the details of the node.



Figure 26.1: Setup Sequence with if-else structures

By means of an If node you can control whether certain nodes will be executed or not. In our case whether to start the SUT application. First, we need to find out if the client is already running. This is the job of the Wait for client to connect node, which writes the result of its inquiry, either true or false, into a variable named isSUTRunning.

Video

Wait for client to connect	
Client	
\$(client)	
Timeout	
0	
GUI engine	
Result handling	
Variable for result	
isSUTRunning	
Local variable	
Error level of message	
Throw exception on failure	
QF-Test ID	
Delay before (ms) Delay after (ms)	
Comment	
This node checks whether the SUT is already running. The result of this check will be stored in the variable	e
isSUTRunning. The variable itself can contain true if SUT i already running or false if SUT is not running. This variab	

Figure 26.2: Wait for client to connect writes the result into the variable "isSUTRunning"

The If node has a Condition attribute where you'll find an expression evaluating the result variable isSUTRunning. As we want to refer to its value we need to use the syntax () (see also note on variable syntax in chapter section 24.1<sup>(256)</sup>).

lf			
Condition <b>not</b> \$(isSUTRu	nning)		Script language Jython 🗸
Name No client, then st	art		
+ / × ↑ Name	↓ Variable de Value	efinitions	
Maximum error le Exception QF-Test ID	evel		~
Delay before (ms)	1	Delay after (ms	)
Comment			

Figure 26.3: If node evaluates the variable

Depending on whether the client is already running or not QF-Test will execute the nodes nested in the If node.

## • Stop the client in case it is running.

Action

## • Single-step through the Setup node.

• Leave the client running and single-step through the Setup node a second time.

If you like you can check the value of the variable <code>isSUTRunning</code> in the variable bindings table. The first time it will have the value <code>false</code> so that the condition <code>not</code> <code>\$(isSUTRunning)</code> will become true and the SUT will be started. The second time it will be <code>true</code> and the if-condition will fail. The nodes nested in the If node will be skipped.

Note Within the first node of the setup sequence you will find more If nodes. They are used to set global variables determining which win browser to use, depending on the operating system. For better readability only If nodes have been used. You may just as well use Elseif and Else to implement that functionality. The child nodes of an Elseif node would

be executed if the If condition were false and the Elseif condition were true. The child nodes of an an Else node would be executed if the If condition and all Elseif conditions were false.

For checking the operating system you can directly resort to a QF-Test variable: QF-Test stores the information about the operation system in a group variable where the group is called 'qftest' and the variables 'linux', 'macos' or 'windows', respectively. The syntax for accessing group variables is \${group:varname}, e.g. \${qftest:windows}.

## 26.2 Loops

QF-Test provides two different kinds of nodes loops:

- Loop nodes execute their child nodes for a certain number of times. However, you can leave the loop any time using a Break node.
- While nodes execute their child nodes until a certain condition becomes false. Again, you can leave it any time using a Break node.
- Note Loop nodes will always stop after the given number of times. In the case of While nodes, however, you need to make sure that the condition will become false at some point. Otherwise you would have an infinite loop. In interactive mode you can always stop execution by hitting the pause button ■. In batch mode you would have to kill the QF-Test process. (You start QF-Test in batch mode using the command line parameter -batch. Then QF-Test does not start its UI and just executes the given test suite.)

In the following exercise we want to implement a test case checking whether a certain row is displayed in the table of the CarConfig application.

The actions of the test case will be:

- Determine the number of rows the table has.
- Loop over all rows and check if it is the row we are looking for.
- Break the loop when a match was found.
- Write an error to the run log if the row was not found.

Please start with recording a check on the row of interest:

- Action
- Activate the check recording mode by clicking the toolbar button
  - **Right-click a row** in the CarConfig application and select the menu item Row from the popup menu.

- Stop the recording by pressing
- · Change the name of the recorded sequence to e.g. 'Check row'
- Turn the recorded sequence into a test case by right-clicking it and selecting the submenu item Transform node into→Test case from the popup menu.

File Edit View Insert Operations Record Run Debugger Clie   Image: Strate Simple Tests Image: Strate Simple Tests Image: Strate Simple Tests Name   Image: Strate Simple Tests Image: Strate Simple Tests Image: Strate Simple Tests   Image: Strate Simple Tests Image: Strate Simple Tests Image: Strate Simple Tests   Image: Strate Simple Tests Image: Strate Simple Tests Image: Strate Simple Tests   Image: Strate Simple Tests Image: Strate Simple Tests Image: Strate Simple Tests   Image: Strate Simple Tests Image: Strate Simple Tests Image: Strate Simple Tests   Image: Strate Simple Tests Image: Strate Simple Tests Image: Strate Simple Tests   Image: Strate Simple Tests Image: Strate Simple Tests Image: Strate Simple Tests   Image: Strate Simple Tests Image: Strate Simple Tests Image: Strate Simple Tests   Image: Strate Simple Tests Image: Strate Simple Tests Image: Strate Simple Tests   Image: Strate Simple Tests Image: Strate Simple Tests Image: Strate Simple Tests   Image: Strate Simple Tests Image: Strate Simple Tests Image: Strate Simple Tests   Image: Strate Simple Tests Image: Strate Simple Tests Image: Strate Simple Tests   Image: Strate Simple Tests Image: Strate Simple Tests Image: Strate Simple Tests   Image: Strate Simple Tests Image: Strate Simple Tests Image: Strate Simple Tests   Image: Strate Simple Tests Image: Strate Simple Tests Image: Strate	(•) ► ■ ■ = > > 0 0 0 0
Test suite        Name             Test set Simple Tests        Charle            Procedures        Cut       Ctrl-X            Extras        Copy       Ctrl-C          Sequence Check Ro        Paste       Ctrl-V          Mouse click VehicleT        Delete       Delete          Mouse click VehicleT        Egit       Alt-Enter          Windows and compone        Marks       Start run          Toggle disabled state        Ctrl-F8         Add to bookmarks       Ctrl-D          Transform node into        Pack ngdes         Unpack nodes       Additional node operations         Insert node       Insert node	nce
Test set Simple Tests       Charle         Procedures       Cut       Ctrl-X         Extras       Copy       Ctrl-C         Sequence Check Ro       Paste       Ctrl-V         Mouse click VehicleT       Delete       Delete         Sequence Recorded       Marks       Start run         Ioggle disabled state       Ctrl-F8         Add to bookmarks       Ctrl-D         Transform node into       Pack nodes         Unpack nodes       Additional node operations         Insert node       Insert node	
Add to bookmarks     Ctrl-D       Transform node into       Pack nodes       Unpack nodes       Additional node operations       Insert node	Variable definitions Value
Expand all     Alt-Right       Collapse all     Alt-Left	<ul> <li>Setup</li> <li>Cleanup</li> <li>Procedure</li> <li>Ctrl+Shift-P</li> <li>Loop</li> <li>Sequence with time limit</li> </ul>
Clean tree Ctrl-Period What's this?	Test c <u>a</u> se C Test step  Vhile Cancel

Figure 26.4: Transform a node into another one

In general, QF-Test lets you add nodes very efficiently by packing one node into another one:

• Open the test case node and **pack the recorded Check node into a loop** by rightclicking it and selecting the submenu item Pack nodes→Loop from the popup menu.

FirstWinTests.qft] * QF-Test	and Due Data	Cliente Estara	Usla	
ile Edit View Insert Operations Rec C C E C View Operations Rec +FirstWinTests.qft ×				<b>୬ ୯</b> ୯ ୯
est suite *FirstWinTests.qft		Check items		
Test suite     Test set Simple Tests     Procedures		Client \$(client)		
<ul> <li>Extras</li> <li>Test case Check Row</li> <li>Check items: row (2) VehicleT</li> <li>Mouse click VehicleTable&amp;0&amp;0</li> <li>Sequence Recorded 17/04/202</li> <li>Windows and components</li> </ul>	C <u>u</u> t <u>C</u> opy <u>P</u> aste Delete Edit	Ctrl-X Ctrl-C Ctrl-V Delete Alt-Enter	t ID 1 ms Text	Regexp
Windows and components	<u>M</u> arks		>	
	Start run <u>T</u> oggle disabled sta <u>B</u> reakpoint on/off	te Ctrl+Shift-D Ctrl-F8	Seguence Cleanup	Ctrl+Shift-S Ctrl+Shift-I
	Locate component Highlight compone Show in Inspector		C Loop C Test step	Ctrl+Shift-T
	Add to bookmarks		<ul> <li>Sequence with time lin</li> <li>Iry</li> <li>Setup</li> </ul>	nit Ctrl+Shift-Y
	Pack n <u>o</u> des		> Setup > Setup Mhile	
	Additional nod <u>e</u> op	erations	>	
	Insert node Clean tree	Ctrl-Period	>	✓ Cancel
	What's this?	Currenda		Candel

Figure 26.5: Pack a node into another one

QF-Test evaluates dynamically which nodes may be packed into one another and only presents the appropriate ones. So, in case you do not find the 'Loop' submenu item make sure you have right-clicked the correct node. The same holds true for the 'Transform node into' and 'Insert node' methods.

In the next series of actions we want to set the value for the Number of iterations attribute of the Loop node. In order to do so we need to find out how many rows the table has. There is no simple node that you could use. However, in the last chapter we learned that the standard library provides a lot of extended functionality. So let's insert the procedure getRowCount from the package qfs.win.table in the standard library.

## Action • Select the Test case node and press Ctrl-A

• Press the 'Select procedure' button (•) left to 'Procedure name'.

283

- Click the tab 'qfs.qft' in the 'Select procedure' dialog.
- Navigate to 'getRowCount' in the package 'qfs.win.table'
- Click 'OK' to select it.
- Click 'OK' in the 'Procedure call' dialog.

Adding a procedure via <u>Ctrl-A</u> was described in <u>Manual creation of procedures<sup>(235)</sup></u>. If you would like to check with the screenshots please have a look there.

### • Enter the variable name rows in the Variable for return value attribute.

- Change the default value for the id in the variable definitions table to the QF-Test component ID of the table, i.e. VehicleTable.
- Click the **OK** button.
- Select the Loop node.
- Enter a reference to the variable \$ (rows) in the Number of iterations attribute of the Loop node.
- Enter the name of an iteration counter, e.g. i in the respective attribute of the Loop node.
- Click the **OK** button.

Loop		
Name		
Number of iterati	ione	Iteration counter
\$(rows)	ions	i
_+ _ × _+	🚽 Variable d	efinitions
Name	Value	
Maria	1	
Maximum error le Exception	evei	~
Exception		
QF-Test ID		
Delay before (ms	)	Delay after (ms)
Comment		
<ul> <li>Comment</li> </ul>		

Figure 26.6: Details of a Loop node

In the next series of actions we will change the recorded row index to the iteration counter and add a variable for the result to the details of the Check node. Then we will add an If node after the Check node evaluating the result, with a Break node within to quit the loop when the row was found.

#### Action

## Open the Loop node.

- Select the Check node.
- Change the recorded row index of the QF-Test component ID to the iteration counter \$(i). The QF-Test component ID should now read VehicleTable@Model&\$(i)
- Enter the variable name checkSucceeded in the 'Variable for result' attribute and click the OK button.
- **Right-click** the Check node and the submenu item Insert node →Control structures→Break from the popup menu.

Action

- Click 'OK' in the 'Break' dialog.
- Pack the Break node into an If node by pressing Ctrl-Shift-I) (Of course you can also pack it via the menu).
- Type \$ (checkSucceeded) in the 'Condition' attribute of the 'lf' node and click the OK button.

The variable checkSucceeded will be set to either true or false by the Check node so that the reference to the variable  $\$  (checkSucceeded) is all we need to enter in the 'Condition' attribute of the If node.

In the next series of actions let's add an Else node as the last node in the Loop node. It will only be entered if all repetitions of the loop were executed, which in our case means that the row was not found and the check never became true.

- **Collapse the If node** if it is open. This is important because otherwise the Else node would belong to the If node and not to the Loop node.
  - Right-click the If node and select the submenu item Insert node  $\rightarrow$  Control structures  $\rightarrow$  Else.
  - Click 'OK' in the 'Else' dialog.
  - Open the Else node.
  - From the standard library insert the procedure logError contained in the package qfs.run-log as described above.
  - Type Row not found in the value field of message in the Variable definitions table.
  - Change the value of withScreenshots in the Variable definitions table from false to true.
  - Click 'OK' in the 'Break' dialog.

When you run tests in batch mode screenshots are a great help for analyzing errors. On the other hand a great number of screenshots lead to a big log-file. This is why the default value for withScreenshots is false.

Last, let's complete the test case with Setup and Cleanup nodes and move it into the top part of the test suite.

Action • Copy the Setup and Cleanup nodes of 'Test set: Simple Tests' into the new test case as the first and last node.

• Move the test case from the Extras section into the top section of the test suite after the 'Test set: Simple Tests' node.

This is what the new test case would look like:

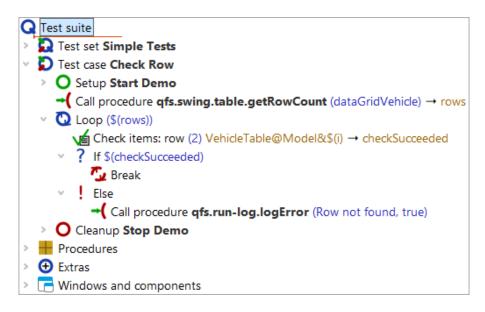


Figure 26.7: The new test case

#### • Execute the new test case.

It should run without error.

• Then modify a value in the details of the Check items node, e.g. change the name of the car to Wrong value.

Chec	k items			
Client \$(clie				
_	F-Test component II			
	cleTable@Model&\$(i)			
_ <b>_</b>	🖌 📩 📫 🕂 Items	5		
		Text		Regexp
	Wrong value \$15,000.00			
Check	k type identifier			
row				
Timed	out			
	ılt handling able for result			
che	ckSucceeded			
	ocal variable			
Erro	r level of message or			~
\$	Throw exception of	n failu	ure	
Name	2			
QF-T	est ID			
Delay	v before (ms)		Delay after (ms)	
10	Comment			

Figure 26.8: Details of the Check items node

## • Execute the new test case again.

This time the Else node should be entered and you should get an error message.

# Chapter 27

# It's time to start your own Application (Win)

After having spent a lot of time with all those example programs, you are now ready to start on your own application (if you really haven't already done so).

Video This chapter is also available as a video tutorial at



"It's time to start your own Application" https://www.qfs.de/en/yt/tutorial-9.html

The Quickstart Wizard available via the menu  $\boxed{\text{Extras} \rightarrow \text{Quickstart Wizard...}}$  helps you to achieve this. Simply follow the wizard steps to generate the setup sequence. Please refer also to chapter 3 "Quickstart" in the user manual.

Then go ahead with what you have learned in this tutorial - record small sequences of events and checks, turn them into procedures which go into your test library, then set up the test cases using procedure calls.

Finally, we reached the end of the basic tutorial part.

# Part IV

# **Android Apps testing with QF-Test**

At the moment there is no dedicated part in this tutorial for testing Android applications with QF-Test. But the respective manual chapter describes in detail all steps required to setup a system for Android testing and how to start automating your tests. Please be sure to have a look there!

The working techniques as described in the other chapters in this tutorial also basically apply for Android testing. So please feel free to have a look e.g. at the chapter <u>Java UI</u> testing with QF-Test<sup>(2)</sup> to learn step by step how to best use the QF-Test test automation features.

Within part  $V^{(294)}$  more advanced QF-Test features are explained, applicable for all supported UI technologies.

# Part V

# **Advanced features of QF-Test**

This part of the tutorial covers advanced QF-Test features.

The following chapters use the Java variant of the CarConfigurator, that you already know from the basic tutorial part. It is a Java/Swing application, but the concepts of the topics is the same for each technology.

We also provide dedicated test suite files for each chapter, so that you get the chance to take a look at the single topics separately. These files are located in qftest-7.1.3/doc/tutorial/advanced-demos/en

# **Chapter 28**

# Data driven Testing: Running one Test case with different test data

This chapter explains how data driven testing can be achieved using QF-Test.

You can find the implementation of the following examples in the demo test suite qftest-7.1.3/doc/tutorial/advanced-demos/en/datadrivenTesting.qft.

The second provided test suite <code>qftest-7.1.3/doc/tutorial/datadriver.qft</code> contains further samples like reading an Excel file or creating an iteration over test cases.

Please take care to copy all test suites to a project-related folder first and modify them there.

## 28.1 Situation

The users of the CarConfigurator can grant several discount levels to their customers. Thus the test designers have figured out three discount levels which have to be tested.

Those discount levels are 0%, 10% and 15%.

As the workflow of granting discounts is the same for each discount level, we can use the same test case for testing them. The only difference between the test cases is the input of the actual discount level and the price to check. A major advantage of using the same test case is that this avoids side effects caused by several implementations of a Test case, which could have different bugs. Additionally we can reduce the implementation effort.

The logical test case, i.e. the steps of the test, for granting a discount will look like this:

- 1. Start the SUT.
- 2. Select a vehicle.

- 3. Specify the discount.
- 4. Check whether the discount has been used for price calculation.
- 5. Stop the SUT.

The following paragraphs demonstrate how to implement such a scenario.

## 28.2 Traditional way of implementing data driven testing

The QF-Test definition of a Test case node says that a Test case is a workflow together with a specific set of test data. If you have two sets of test data, you have to use two different Test case nodes. Those nodes can be organized within a Test set node.

So the conventional way to solve the issue is to implement one Test case per discount level, like depicted below:

Г 🕰	est set Traditional data driven testing
> (	Setup start application
> 🕻	Test case Discount 0
> 🕻	Test case Discount 10
> 🕻	Test case Discount 15
> (	Cleanup stop application

Figure 28.1: Traditional way of data driven testing

Those three nodes are grouped within a Test set node. The Test set node also contains the Setup and the Cleanup sequences, which will launch the SUT before each Test case and will stop the SUT after it. This is to ensure that each test case has the same preconditions. If you do not want to re-start the SUT between the single test runs, you could add a new Test set to the Test set 'Discount levels' and move the three Test case nodes into that Test set like this:

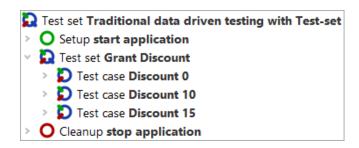


Figure 28.2: Traditional way with a nested Test set

The chapter Dependencies: Automatically ensuring correct prerequisites for each <u>Test case<sup>(304)</sup></u> will show up a more elegant and more efficient way of organizing pre-requisites of test cases.

As you can imagine this approach could be very exhausting with regards to maintaining test data, especially if we have to add or remove discount levels. Another disadvantage is that we keep the test data within QF-Test.

The next section <u>Data driver concept<sup>(297)</sup></u> demonstrates how to organize the test suite to implement the Test case only once and to keep the test data separated from the test case.

## 28.3 Data driver concept

If we want to run one test case using different test data sets, we have to define the test data in a data source first. The data source has to be part of a Data driver node. QF-Test offers built-in data source nodes for database tables, CSV files, Excel files and QF-Test data tables. A QF-Test data table will store the data in the test suite itself. This is what we will use for the following example. You can use any other type of data sources, e.g. XML files, too, by implementing your own script for reading the data.

Insert a Test set to the test suite first. You can choose whatever name you want.

A Data driver node can be inserted into a Test set via right mouse click and then selecting <u>Insert node  $\rightarrow$  Data drivers  $\rightarrow$  Data driver</u>. You have just to specify a name for this node. The actual data source will then be inserted as child node to that Data driver node. For our example we insert a 'Data table' via a right mouse click at the opened Data driver node and select <u>Insert node  $\rightarrow$  Data drivers  $\rightarrow$  Data drivers. Now we should detect this dialog:</u>

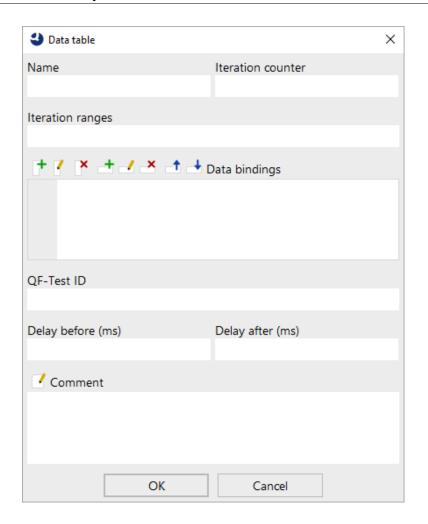


Figure 28.3: Data table dialog

First we have to specify a name for this data source. We should also define a name for the 'Iteration counter' variable. The iteration counter contains the index of the currently executed test data in the test run

The next step is to define the test data. Therefore click on the 'Insert column' button, which is the first button of the 'Data bindings' section. Then you have to define a name for the column, let us set it to 'discount'. After pressing 'OK' you will see that the column has been inserted in the 'Data bindings' area. This column heading will stand for the variable name in the tests later.

Now you can use the 'Insert row' button to insert a new row. Each row will stand for one test data set, i.e. you have to insert three rows by now, where the first row contains '0', the second '10' and the third '15'.

The table should look like this:

Test suite		Data table		
Q Test suite	^	Name	Iteration counter	^
Z Test set Traditional data driven testing		testdata	i	
O Setup start application				
> D Test case Discount 0		Iteration ranges		
D Test case Discount 10				
D Test case Discount 15		+ / x + / x	🔹 🕂 🚽 Data bindings	
O Cleanup stop application			2	
Test set Traditional data driven testing with Test-set			discount	
O Setup start application		0 0		
V 🔂 Test set Grant Discount		1 10		
> D Test case Discount 0		2 15		
> D Test case Discount 10		OF-Test ID		
> D Test case Discount 15		QF-Test ID		
O Cleanup stop application				
Z Test set data driven test with discount variable in data table		Delay before (ms)	Delay after (ms)	
👻 🤨 Data driver <b>data source</b>				
Data table <b>testdata</b>				
Setup start application		🗸 Comment		
D Test case Discount				~
O Cleanup stop application		ОК	Cancel	
🔨 🏷 Tort rot data duivan taat mith diaanmt and finalmian mulala in data tala	×	UK	Cancel	

Figure 28.4: The filled data table

The next part is the implementation of the test case. Therefore we just insert one Test case node to the Test set.

Note If you want to insert a Test case to a Test set which already contains a Data driver node, you have to select the closed Data driver node to insert it.

The Test case will contain the procedure calls for the required test steps. The Setup and Cleanup nodes of the previous example can also be copied to the Test set. The whole Test set will look like this now:

C	Test set data driven test with discount and finalprice variable in data table
v	💽 Data driver <b>data source</b>
	Data table <b>testdata</b>
>	O Setup start application
>	D Test case Discount
>	O Cleanup stop application

Figure 28.5: Test set with Data driver

The following step is to make use of the variable 'discount' which has been defined at the 'Data table' node before. We will add that variable as parameter to the 'setDiscount' procedure call. Once we have done this, we have following result:

Test suite	Procedure call	
Q Test suite	• Procedure name	^
>  Test set Traditional data driven testing	accessoryPanel.setDiscount	
Part Set Traditional data driven testing with Test-set		
Part and the set of	Variable for return value	
Test set data driven test with discount and finalprice variable in data tabl		
> 🧿 Data driver data source		
Setup start application	Local variable	
v D Test case Discount	+ / × + Variable def	initions
Call procedure menu.file.reset		Indons
Call procedure vehiclesPanel.selectCard		
Call procedure vehiclesPanel.selectVehicle (Hydro2)	discount \$(discount)	
Call procedure specialsPanel.selectCard		
Call procedure specialsPanel.selectModel (Go)	QF-Test ID	
Call procedure accessoryPanel.selectCard		
Call procedure accessoryPanel.selectAccessory (Multimedia package)	Delevis hafe en (mai)	
Call procedure accessoryPanel.selectAccessory (Parking assistant)	Delay before (ms) Del	lay after (ms)
Call procedure accessoryPanel.setDiscount (\$(discount))		
Call procedure accessoryPanel.checkFinalPrice (\$(finalprice))	Comment	
Cleanup stop application	connent	v
> 🔁 Test set data driven test with optimized procedure calls	ОК	Canaal
< · · · · · · · · · · · · · · · · · · ·	UK	Cancel

Figure 28.6: Using the \$(discount) parameter

Now we are ready to launch the Test set.

After running the tests we should get at least two errors. Those errors come from the different values of the 'Final price' text-field, but our test always verifies the same value. In our case we should also put the expected values for the 'Final price' fields to our test data via adding a second column to the 'Data table' node.

Test suite		Data ta	able				
Q Test suite	^	Name			Iteration counter	r	^
D Test set Traditional data driven testing		testdata i					
> D Test set Traditional data driven testing with Test-set							
> 🔁 Test set data driven test with discount variable in data table		Iteration	n ranges				
🔹 🔁 Test set data driven test with discount and finalprice variable in data tab							
🗸 🧓 Data driver <b>data source</b>		+ 7	x + /	× +	🚽 Data binding		
Data table <b>testdata</b>				-		S	
Setup start application			discount	_	finalprice		
V D Test case Discount		0		85,095			
→ Call procedure menu.file.reset		1 1	0	76,585	5.50		
Call procedure vehiclesPanel.selectCard		2 15 72,330.75					
Call procedure vehiclesPanel.selectVehicle (Hydro2)		OF-Test	+ 10				
Call procedure specialsPanel.selectCard		QF-Test					
Call procedure specialsPanel.selectModel (Go)							
Call procedure accessoryPanel.selectCard		Delay b	efore (ms)		Delay after (ms)		
Call procedure accessoryPanel.selectAccessory (Multimedia package)							
Call procedure accessoryPanel.selectAccessory (Parking assistant)							
Call procedure accessoryPanel.setDiscount (\$(discount))		🗹 Cor	mment				
Call procedure accessoryPanel.checkFinalPrice (\$(finalprice))							۷
Cleanup stop application	~		OK			Cancel	

Figure 28.7: Full data table

Another drawback is, that we see the same test case name in the HTML overview report and in the run logs for each test run. To avoid this we can edit the 'Name for reports and run logs' attribute of the Test case node. In that attribute we have to make use of at least one of the test case specific data, i.e. in our case 'discount'. So let us set that attribute to 'Discount \$(discount)'.

Test suite	Test case	
Q Test suite	Name	^
> D Test set Traditional data driven testing	Discount	
D Test set Traditional data driven testing with Test-set		
> D Test set data driven test with discount variable in data table	Name for run log and report	
🕐 🔀 Test set data driven test with discount and finalprice variable in data table	e Discount \$(discount)	
<ul> <li>O Data driver data source</li> <li>O Setup start application</li> </ul>	+ / × + Characteristic variables	
> D Test case Discount	Name	
Cleanup stop application		
D Test set data driven test with optimized procedure calls		
D Test set data driven test with csv file as data source	Name for separate run log	
Procedures		
<ul> <li>O Extras</li> <li>O Windows and components</li> </ul>	Inherit dependency from parent node	
	Condition Expected to fail if Script language	
	Jython 👻	
	+ / × + Variable definitions	
	Nama Valua	۷
< >	OK Cancel	

Figure 28.8: Name for run log and report attribute

If we rerun the tests now, we should get no error anymore and the run log as well as the HTML report contain three different test case names. Here you can see the created run log file:

🖶 Run log (detailed)	
Z Test set data driven test with discount and finalprice variable in data tal	ole
> 🧓 Data driver data source	
🗸 🖸 Loop pass 0	
Setup start application	
> D Test case Discount 0	
O Cleanup stop application	
🗸 🖸 Loop pass 1	
Setup start application	
D Test case Discount 10	
O Cleanup stop application	
🗸 💽 Loop pass 2	
Setup start application	
D Test case Discount 15	
O Cleanup stop application	

Figure 28.9: Run log with different names per Test case

If you want to run one Test case only without the whole Test set and its Data driver, it is

recommended to set default values for the variables as global ones at Test suite.

Note

If the name of the variable in the Data driver is the same like the name of the procedure, you can also skip the variable definition of the Procedure This can be done because the variable of the Data driver will be put at call. QF-Test's within variable stack and SO any step the Test case can access that variable. You can see this behavior in the demo test suite gftest-7.1.3/doc/tutorial/advanced-demos/en/datadrivenTesting.gft in the Test set "data driven tests with optimized procedure calls".

Inthedemotestsuiteqftest-7.1.3/doc/tutorial/advanced-demos/en/datadrivenTesting.qftyou can also find an additional Test set using a CSV file as data source.

## 28.4 Summary

The Data driver concept of QF-Test allows the user to create logical test cases and to keep the test data separated from the test flow itself.

It is even possible to use nested Data driver nodes in a Test case. This can be achieved by creating a Test step in a Test case. This Test step can contain the nested Data driver.

You can find a more detailed explanation about data driven testing within QF-Test in the manual in the chapter Data-driven testing.

The second provided test suite <code>qftest-7.1.3/doc/tutorial/datadriver.qft</code> contains further samples like reading an Excel file or creating an iteration over test cases.

## **Chapter 29**

## **Dependencies: Automatically ensuring correct prerequisites for each** Test case

#### Video



Video:

Dependencies

https://www.qfs.de/en/yt/dependencies-basics-50.html

This chapter explains the dependency concept of QF-Test. That concept is important for creating robust test cases and for defining recovery activities, if a single test case crashes during the test run. It has been introduced to guarantee that any test case's prerequisites are fulfilled before running it.

You can find the following examples in the file qftest-7.1.3/doc/tutorial/ advanced-demos/en/dependencies.qft. There is gftest-7.1.3/doc/tutorial/ also second test suite а advanced-demos/en/dependencies\_work.qft perform your to own implementations. Please take care to copy all test suites to a project-related folder first and modify them there.

## 29.1 General

Please copy the test suite <code>qftest-7.1.3/doc/tutorial/</code> advanced-demos/en/dependencies\_work.qft to a project-related folder first and open it there. Take a look at the first Test set 'Discount Tests'. It contains three different Test case nodes and the respective Setup and Cleanup to launch and stop the SUT before each Test case. That is a typical test suite you might have created yourself.

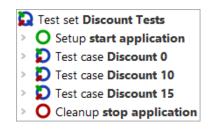


Figure 29.1: First Test set of dependencies\_work.qft

Let us assume you just want to launch one specific Test case, because that one failed or it verifies a specific defect. If you want to do so, you have either to run the whole Test set, because you have to ensure that the prerequisites are fulfilled, or you run the Setup manually and run the according Test case then.

As this is a situation which is very common but not easily solvable with the concepts we know already, QF-Test introduces the Dependencies concept. It is responsible for managing prerequisites of Test cases and allows you to run one Test case directly. In that case QF-Test will take care of the prerequisites of the Test case, e.g. launching the SUT, selecting a vehicle and so on.

A Dependency can contain a Setup, a Cleanup, an Error handler and a Catch node. The Setup of a Dependency will be executed before each Test case. This is because the correct pre-conditions of a Test case are a very important aspect of a robust test run. You could imagine a situation where one Test case closes the SUT because of a mistake. Then the next test case has to launch the SUT again. Exactly for this situation, the Dependency concept provides a very stable and fancy solution.

The second aspect of Dependencies is that they optimize the execution of tests. With the current means we have to start the SUT before each Test case and stop the SUT after each. That is no problem for a small application like the CarConfigurator, but for huge applications, like Eclipse/RCP application or any ERP system, this might become insufficient. That is why Dependencies call the Cleanup steps only if required.

Another advantage of the Dependencies concept are global Error handler and Catch for recovery management. This feature becomes quite important, if you run lots of test cases in a series and one is failing and preventing the other test cases from continuing execution because of modal error dialogs, e.g. 'OutOfMemoryException'.

In a nutshell Dependencies are

- 1. a place to define the prerequisites of a Test case.
- 2. very helpful to make test cases more independent from other ones.
- 3. a better approach of implementing Setup and Cleanup steps.
- 4. allows you to implement recovery steps in case of errors or exceptions.

- 5. optimizing test execution.
- 6. re-usable as they can be defined in the Procedures area.

The following sections demonstrate how to use Dependencies.

## 29.2 Ensuring prerequisites

Please copy the test suite qftest-7.1.3/doc/tutorial/
advanced-demos/en/dependencies\_work.qft to a project-related folder first
and open it there if you haven't done that so far.

This file contains a Test set 'Discount Tests' with three Test cases and the conventional implementation using the Setup and Cleanup. We will change this Test set to make use of a Dependency now.



Figure 29.2: First Test set of dependencies\_work.qft

First we have to insert a Dependency node. This can be achieved by a right click at the Test set and then selecting Insert node $\rightarrow$ Dependencies $\rightarrow$ Dependency. Specify a name for that Dependency, e.g. "SUT started".

The next step is to move the Setup and Cleanup nodes into that Dependency. Therefore open the Dependency node and move the mentioned nodes into it. You can do this via drag and drop or via right mouse click Cut and Paste or Ctrl-X and Ctrl-V.

The test suite should now look like this:

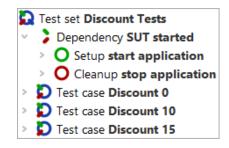


Figure 29.3: Sample test suite with the first Dependency

Let us test the Dependency now: Please stop all running clients before that, then select one Test case, e.g. 'Discount 10', and run it.

You should see that the Test case 'Discount 10' has been executed and the SUT has not been stopped at the end of the test run. Please open the run log to make a deeper analysis of what happened.

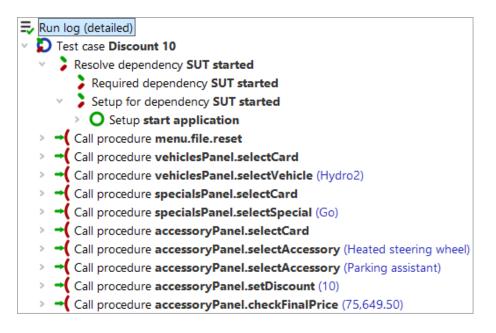


Figure 29.4: The run log of the execution

If you open the Test case in the run log, you can see a 'Resolve dependency' node. Open it and there you find two nodes. The last one shows that the Setup has been executed. The other one will be explained in following chapters.

So far we have determined that the Setup node will be automatically executed before a Test case, if the Dependency is defined at the Test set level, but the Cleanup has not been executed so far. If you start another Test case now, e.g. 'Discount 15', the already started

SUT is used.

The Setup node of a Dependency will be executed before each Test case in any case. That is to ensure that the prerequisites are fulfilled for each single Test case. The Cleanup node of a Dependency will only be executed on demand, i.e. if the steps of the Setup of that Dependency are not required anymore. In our case the Cleanup steps did not get executed, because both test cases have the same Dependency. The test execution passed because the Procedure startStop.startApplication already checks whether it's necessary to launch the application or not.

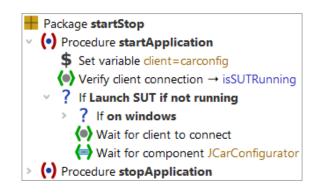


Figure 29.5: Procedure startStop.startApplication

The next step is to launch the complete Test set by clicking it and pressing 'Run test'.

All three tests should pass and the SUT should not be stopped in between. We have achieved an optimized test execution. The Cleanup will not run because all three test cases refer to the same Dependency. We save a lot of time in executing the test cases and we are closer to testing a real life behavior as users usually do not restart the SUT all the time.

The next goal is to make that Dependency available for other Test sets in our project. For this you need to move the Dependency node into the Procedures area. Then click at the Test set and insert a Dependency reference node via Insert node  $\rightarrow$  Dependencies  $\rightarrow$  Dependency reference. The dialog looks similar to the Procedure call dialog and you can select the Dependency on the same way. The test suite should now look like this:

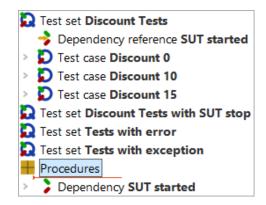


Figure 29.6: The test suite with a Dependency reference

We recommend to add all Dependencies to a separate Package called 'dependencies'.

If you run the Test set the first time after moving the Dependency, QF-Test stops the SUT before the first Test case and re-starts the SUT, because the Dependency from the Test set is different from the Dependency defined in the Procedures node. But this will be explained more detailed in following chapters.

Please take a closer look at the second Test set 'Discount Tests with SUT stop' of the demo test suite. The second Test case 'Dicount 10' stops the SUT. But the third Test case also relies on a running SUT. As we know the Dependencies concept already, we should see that the SUT will be started for the third Test case again. That is exactly the advantage of the Dependencies concept.



Figure 29.7: Ensuring prerequisites for Test case 'Discount 15'

## 29.3 Combining dependencies

Managing prerequisites can be a more complex matter than just verifying whether the SUT is up and running. In many projects several groups of test cases with different prerequisites like others exist.

#### 29.3. Combining dependencies

Let us assume you want to test a big ERP system with more views, like a 'vendor' view and an 'article' view. Then all tests for the 'vendor' view will rely on an opened 'vendor' view and all tests for the 'article' view rely on an opened 'article' view. The opening of the specific view again relies on a logged-in user and the login depends on a started SUT. So you can see something like a tree of prerequisites.

QF-Test allows the user to build such a structure of Dependency nodes via adding Dependency reference nodes to a Dependency. We will build up a small example with just two dependencies for the CarConfigurator now.

In the CarConfigurator you can open a 'vehicles' dialog via the menu 'Options' -> 'Vehicles'. We want to create certain tests for that dialog now. Later we want to create some tests on the 'accessory' dialog which can also be opened via the menu via 'Options' -> 'Accessories'.

First let us define the tests which we want to create.

Test Case 1: Create vehicle 'test1' with price '100'.

- Start SUT if necessary.
- Open vehicles dialog via the menu.
- Specify 'test1' as name and '100' for the price.
- Press 'New'.
- Close vehicles dialog via pressing 'Ok'.
- Open vehicles dialog via the menu.
- Select the newly created vehicle 'test1'.
- Close the vehicles dialog via pressing 'Cancel'.
- Stop SUT if necessary.

Test Case 2: Create vehicle 'test2' with price '99999'.

- Start SUT if necessary.
- Open vehicles dialog via the menu.
- Specify 'test2' as name and '99999' for the price.
- Press 'New'.
- Close vehicles dialog via pressing 'Ok'.
- Open vehicles dialog via the menu.

- Select the newly created vehicle 'test2'.
- Close the vehicles dialog via pressing 'Cancel'.
- Stop SUT if necessary.

Test Case 3: Create accessory 'testaccessory' with price '12'.

- Start SUT if necessary.
- Open accessories dialog via the menu.
- Specify 'testaccessory' as name and '12' for the price.
- Press 'New'.
- Press 'Ok'.
- Open accessories dialog via the menu.
- Select the newly created accessory 'testaccessory'.
- · Close the accessories dialog via pressing 'Cancel'.
- Stop SUT if necessary.

Let us take a closer look at the test steps of the tests above. We see that each test case requires a running application, so we should implement the 'Start SUT' step as prerequisite. This has to be done in the Setup of a Dependency. The 'Stop SUT' step is an optional step which can be part of the Cleanup of the Dependency. This 'SUT started' Dependency has been implemented by us in the previous examples already, so we could re-use it.

The next issue is that test case 1 and test case 2 require an opened vehicle dialog. Because we plan more tests in that area, we can create a new Dependency 'vehicles dialog opened', which will contain the opening of the dialog as Setup and the closing of the dialog via 'Cancel' as Cleanup. We are able to open this dialog only if the SUT is up and running already, so this Dependency is dependent on the 'SUT started' Dependency. The implementation of that 'open vehicle dialog' Dependency looks like this:

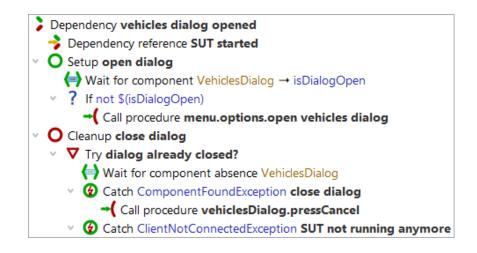


Figure 29.8: 'vehicles dialog opened' Dependency

Note In the Setup we have to check whether the dialog is already opened, because a previous test case could have forgotten to close the dialog. The 'Timeout' attribute of the Wait for component to appear node is set to '0' because the dialog is expected to be here. If the dialog is already opened, it is fine and we can continue, otherwise we have to open the dialog.

We have to create a Dependency 'accessories dialog opened' as well. This Dependency is similar to the 'vehicles dialog opened' Dependency, but it deals with the accessory dialog instead of the vehicles dialog.

After creating those dependencies we have to record the according test steps and create the test cases. The test steps have already been implemented by us and you can find them in the according Package structure in the test suite <code>qftest-7.1.3/doc/tutorial/ advanced-demos/en/dependencies.qft</code>.

The test cases should be organized in one Test set called 'Combined Dependencies Tests'. This Test set should contain two more Test sets. The first one is 'Tests for vehicles' and the second one is 'Tests for accessories'. The Test set 'Tests for vehicles' has to contain the implementation of test case 1 and test case 2 and a Dependency reference to the 'vehicles dialog opened' Dependency. The second Test set 'Tests for accessories' has to contain test case 3 and a Dependency reference to the Dependency 'accessories dialog opened'.



Figure 29.9: Implementation of specified test cases

If you run that Test set now, you will see that QF-Test stops the SUT first, because that comes from a dependency of the previous examples. Then QF-Test starts the SUT, performs the steps of test case 1 and then the steps of test case 2. Then it will perform the steps of test case 3. If you take a closer look at the beginning of test case 3 in the run log, you will see that the Cleanup of the 'vehicles dialog opened' Dependency has been executed. That happened because the 'vehicles dialog opened' Dependency was not required anymore. The following test case test case 3 required the 'accessories dialog opened' Dependency, that is why that Cleanup of the 'SUT started' Dependency has not been executed.



Figure 29.10: Run log of nested Dependencies

The ability to combine dependencies or to call the Cleanup of dependencies on demand can enable you to put a lot of preparing and cleaning steps into a Dependency. Another use case for the CarConfigurator could be to create a 'Vehicle created' Dependency, which creates a vehicle, which is required for the test and removes it afterwards. That would be an approach to ensure that the test data has been established correctly.

## 29.4 Error and exception handling

#### 29.4.1 Error handling

Please copy the test suite <code>qftest-7.1.3/doc/tutorial/</code> advanced-demos/en/dependencies\_work.qft to a project-related folder first and open it there if you haven't done that so far. You can find a Test set 'Tests with error'. The second Test case of that Test set fails.

Test set Tests with error		
>	Dependency reference SUT started	
> Ď	Test case Discount 0	
> Ď	Test case Discount 10	
> Ď	Test case Discount 15	

Figure 29.11: Test-suite of Error handler

Let us assume that we want to trigger a specific behavior after a test case, but only if it failed. In our case we could stop the SUT to guarantee that the following test case can rely on a clean environment again. We know that the Setup node is executed before each Test case and the Cleanup node will only run on demand. So how could we run specific steps only in case of an error?

The solution is the so called Error handler node for a Dependency. You have to click at the closed Cleanup node of that Dependency and insert the Error handler node via right mouse click and selecting Insert node  $\rightarrow$  Dependencies  $\rightarrow$  Error handler.

In the  $Error\ handler\ you\ can define the steps$  for stopping the SUT again. The Dependency  $SUT\ started\ should\ look\ like\ this\ now:$ 



Figure 29.12: Dependency with Error handler

Please run the whole Test set 'Tests with error' and switch into the run log after the test run has finished.

In the run log you can see that the Error handler was executed after the second Test case only.

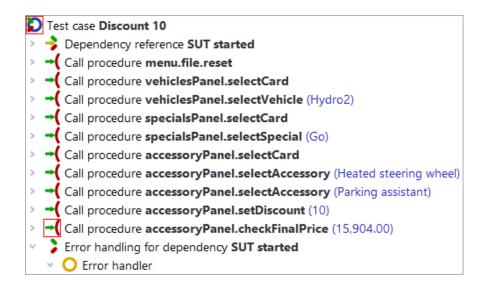


Figure 29.13: Run log for Dependency with Error handler

#### 29.4.2 Exception handling

In the previous section we have learned that Error handler nodes contain steps which are executed if the Test case fails. Besides errors also exceptions can occur during a test run. An exception is an unexpected behavior during the test run, e.g. an error dialog appears suddenly or a component could not be found anymore. How should we handle such exceptions?

You can find an example Test set 'Tests with exception' in the test suite dependencies\_work.qft.

Of course you could surround the test steps in each Test case by try-catch and implement a dedicated exception handling in each single Test case, like implemented in the demo Test set. But this approach could lead to a lot of redundancy and makes the Test cases a little bit more unreadable.

<u>Eait view insert Operations Record</u>	Nan Depagger Cherics Carras ne
dependencies_work.qft $ imes$	
st suite *dependencies_work.qft	
Call procedure vehiclesPanel.se	electCard
Call procedure vehiclesPanel.se	
Call procedure specialsPanel.se	_
-( Call procedure specialsPanel.se	
-( Call procedure accessoryPanel	•
	selectAccessory (Heated steering wheel)
-( Call procedure accessoryPanel	selectAccessory (Parking assistant)
- Call procedure accessoryPanel	setDiscount (0)
- Call procedure accessoryPanel	.checkFinalPrice (84,055.00)
<ul> <li>Getch TestException</li> </ul>	
Call procedure qfs.swing.cle	anup.closeAllModalDialogs (5)
Call procedure startStop.sto	pApplication
Test case Discount 10	
🗸 🔽 Try	
Call procedure menu.file.reset	
Call procedure vehiclesPanel.s	electCard
Call procedure vehiclesPanel.s	electVehicle (Maxigolf)
Call procedure specialsPanel.se	electCard
Call procedure specialsPanel.se	electSpecial (Go)
Call procedure accessoryPanel	selectCard
Call procedure accessoryPanel	selectAccessory (Heated steering wheel)
Call procedure accessoryPanel	selectAccessory (Parking assistant)
Call procedure accessoryPanel	setDiscount (10)
Call procedure accessoryPanel	.checkFinalPrice (75,649.50)
<ul> <li>Ocatch TestException</li> </ul>	
· · · · · · · · ·	anup.closeAllModalDialogs (5)
Call procedure startStop.sto	pApplication

Figure 29.14: Try-catch nodes in each Test case

Our goal is to reduce the redundancy and to move the exception handling - which is always the same - to one global location. This location will be our Dependency.

The first step is adding the Catch node to the Dependency via clicking at the closed Error handler node and selecting Insert node $\rightarrow$ Control structures $\rightarrow$ Catch in the menu. Then we can copy the procedure-calls for stopping the SUT into that Catch. Now the try-catch nodes in the single test cases become unnecessary. Please move the procedure-calls out of the Try and remove the empty try-catch structure.

Your test suite should look like this now:

	<u>E</u> dit	<u>V</u> iew Insert Operations <u>R</u> ecord Run Debugger <u>C</u> lients E <u>x</u> tras	
,			-
d	epen	dencies_work.qft ×	
;t	suite	*dependencies_work.qft	
		Call procedure specialsPanel.selectCard	
		Call procedure specialsPanel.selectSpecial (Go)	
		Call procedure accessoryPanel.selectCard	
		Call procedure accessoryPanel.selectAccessory (Heated steering whee	I)
		Call procedure accessoryPanel.selectAccessory (Parking assistant)	
		Call procedure accessoryPanel.setDiscount (0)	
	-	Call procedure accessoryPanel.checkFinalPrice (84,055.00)	
`	-	Test case Discount 10	
		Call procedure menu.file.reset	
		Call procedure vehiclesPanel.selectCard	
		Call procedure vehiclesPanel.selectVehicle (Maxigolf)	
		Call procedure specialsPanel.selectCard	
		Call procedure specialsPanel.selectSpecial (Go)	
		Call procedure accessoryPanel.selectCard	
		Call procedure accessoryPanel.selectAccessory (Heated steering whee	1)
		Call procedure accessoryPanel.selectAccessory (Parking assistant)	
		Call procedure accessoryPanel.setDiscount (10)	
	-	Call procedure accessoryPanel.checkFinalPrice (75,649.50)	
2	_	Test case Discount 15	
		ocedures	
1	-	Dependency SUT started	
		O Setup start application	
		Cleanup stop application	
		Error handler	
	~	Catch TestException	
		Call procedure qfs.swing.cleanup.closeAllModalDialogs	
		Call procedure startStop.stopApplication	

Figure 29.15: Test-suite with Catch

Now you can start the Test set 'Test with exception'. The second Test case throws an IndexNotFoundException, because the selected vehicle does not exist. It is intended to handle that exception in the Catch of the Dependency.

Note If the debugger is activated, QF-Test interrupts the test run at the place where the exception arises. In this case you might have to re-throw the exception via the

'Rethrow exception' button. An alternative is to deactivate the debugger via  $Debugger \rightarrow Enable debugger$ .

After the test run you can open the run log and verify what happened.

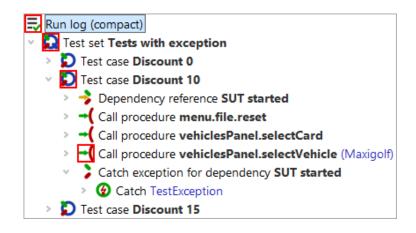


Figure 29.16: Run log of a Dependency with Catch

```
In
     а
         normal
                   project
                            vou
                                   should
                                            at
                                                 least
                                                         try
                                                              to
                                                                    catch
                                                                            а
'TestException'
                and
                       the
                             Catch
                                      should
                                               contain
                                                         а
                                                             procedure
                                                                          call
             either
of
                              qfs.swing.cleanup.closeAllModalDialogs,
gfs.fx.cleanup.closeAllModalDialogs
                                                                           or
qfs.swt.cleanup.closeAllModalDialogsAndModalShells.
                                                                       Those
procedures close all modal dialogs, e.g. dialog windows, which are blocking the
execution.
```

### 29.4.3 Summary

You have seen that you can implement a very strong recovery system for your test cases using the Error handler and, even more important, the Catch of Dependencies.

In most projects the global Catch node becomes very important, especially if ComponentNotFoundExceptions and ModalDialogExceptions appear.

## 29.5 More about dependencies

In the previous sections we have learned that we can combine several dependencies and that the Cleanup of a Dependency is only executed, if the stack of combined dependencies changes. We could also instruct a Dependency to call its Cleanup every time after a Test case. This can be achieved by checking the attribute 'Forced cleanup' of the Dependency.

There are even more interesting features about dependencies, e.g. you can use a variable instructing QF-Test to run the Cleanup. This variable is called 'Characteristic variable'. You will find more details about that feature in the manual at chapter Dependencies. This approach could be used to create one 'user login' Dependency which will only run through its Cleanup, i.e. perform the logout, if another user is required by the next Test case.

A detailed description of dependencies can be found in the manual at chapter Dependencies.

## Chapter 30

# Automated creation of basic procedures

This chapter describes how procedures for each UI component can be created automatically by QF-Test. The advantage of this approach is that you do not need to record each step of your tests manually. Furthermore you will also get a standardized structure of Packages and Procedures for testing all dialogs of your SUT.

You can find the following examples in the file <code>qftest-7.1.3/doc/tutorial/</code> advanced-demos/en/automated\_procedures.qft . There is also a second test suite <code>qftest-7.1.3/doc/tutorial/</code> advanced-demos/en/automated\_procedures\_work.qft to perform your own implementations. Please take care to copy all test suites to a project-related folder first and modify them there.

## 30.1 General

If we want to create tests for all features of the CarConfigurator, we have to record steps on each element that will be touched by a test. The CarConfigurator is a small application with five dialogs and perhaps approximately thirty graphical elements. So creating all important test cases for this application can be achieved within one or two days. But imagine a big project like an ERP system with about fifty dialogs and hundreds of elements. Creating test cases for such a huge system will definitely take longer and also the maintenance of recorded tests could become quite difficult.

As first organizational step we recommend to record each test step as a Procedure and call it from the Test cases requiring it. If you organize your tests in different test suites, you could split those test suites into two levels. The first level contains only UI component related procedures and the second level contains only test cases calling procedures of

level one.

The approach of implementing each test step as procedure brings us to a situation where we could split the work into two parts:

- 1. Creating and maintaining the procedures representing the test steps
- 2. Creating and maintaining the test cases

QF-Test brings a feature which creates that basic procedures for each UI component automatically. The usage of this feature should drastically minimize the time for creating test suites and test cases and supports you in making easier maintainable test suites.

You can find a demo suite containing some test cases created by this qftest-7.1.3/doc/tutorial/ advanced-demos/en/automated\_procedures.qft.

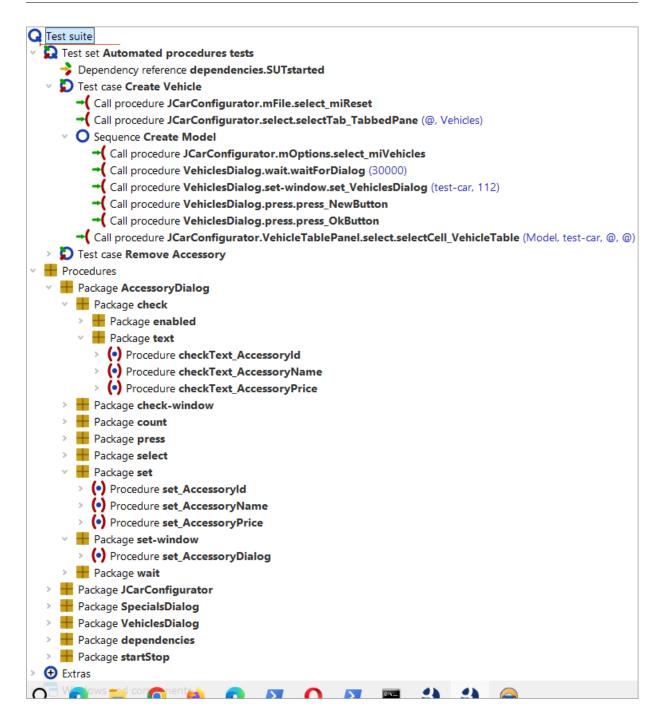


Figure 30.1: Screenshot of test suite

The following sections demonstrate how to create those procedures and organize your test cases.

## **30.2** Automated creation of procedures

Please copy the demo test suite qftest-7.1.3/doc/tutorial/
advanced-demos/en/automated\_procedures\_work.qft to a project-related
folder and open it there.

This test suite contains a Test set referring to a 'Start SUT' Dependency.



Figure 30.2: The test suite automated\_procedures\_work.qft

First we have to launch the SUT via selecting the Dependency reference and pressing 'Start test run'.

Once the SUT is up and running we are ready to record the test steps. Normally we would press the 'Start recording' button, then record the according test steps and then press 'Stop recording'. After that we would re-organize the recorded steps, i.e. generating procedures and parametrizing them. Exactly those steps can be performed automatically now.

Let us generate the basic procedures for the main window first. Before we can start, please modify the settings of QF-Test.

Open the QF-Test options via  $Edit \rightarrow Options$ . Then switch to 'Record' -> 'Procedures'. Please change the value for the 'Configuration file for recorded procedures' to the full path of the demo configuration test suite at qftest-7.1.3/demo/procbuilder/carconfig-procbuilderdef.qft. Then close the 'Options' via pressing 'OK'. You will see more details about that file in the following chapters.

After altering the settings proceed with following steps:

- Press the 'Record procedures' button.
- Perform a right mouse-click in the SUT.
- Select 'Whole window'.
- Press the 'Record procedures' button again.

Now QF-Test creates the basic procedures of the main window. You should see a new Package called procbuilder under Procedures of your test suite. This package

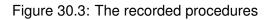
#### 30.2. Automated creation of procedures

Note

contains several packages and procedures for actions on the whole dialog and its single UI components.

The current configuration created a Package JCarConfigurator under the procbuilder Package. This Package is indicated to be the container for all actions performed on the main window.

Package <b>procbuilder</b>
Package JCarConfigurator
> 🕂 Package MenuBar
🗸 🖶 Package VehicleTablePanel
> 🖶 Package check
Record procedures e count
Procedure getColumnCount_VehicleTable
> (•) Procedure getRowCount_VehicleTable
Package select
<ul> <li>(•) Procedure selectCell_VehicleTable</li> </ul>
← Call procedure qfs.swing.table.selectCell (\${id:VehicleTable}, \$(column), \$(row), \$(columnSeparator), \$(rowSeparator))
> 🖶 Package <b>check</b>
> 🕂 Package <b>check-window</b>
> 🗕 Package get
> 🗕 Package press
> 🗕 Package select
> 🗕 Package set
> 🖶 Package set-window
> 🖶 Package wait



Please note that the involved components have also been recorded in Windows and components.

The next step is to check which procedures are useful for us and which are obsolete.

Let us take a closer look at the created Packages:

Package	Content
JCarConfigurator	This package contains all procedures for actions on components of the Car- Configurator window. It has been created because the configuration file uses the component hierarchy for creating the package structure.

Table 30.1

The Package JCarConfigurator contains several packages which are:

#### 30.2. Automated creation of procedures

Package	Content
MenuBar	This package contains procedures for clicking at the items of the menu.
VehicleTablePanel	This package contains all procedures for actions on components on the 'Ve- hicleTable' panel of the SUT. In our case there are only procedures for the 'VehicleTable' object, because that is the only component on that panel.
check	This package contains procedures for checking components.
check-window	This package contains procedures which can be called to check the compo- nents of one frame using one call. Those procedures are container proce- dures.
get	This package contains procedures for getting the current values of components, e.g. reading the text of a text-field.
select	This package contains procedures for selecting items of elements - in our case only one procedure to select a tab in the TabbedPane.
set	This package contains procedures for setting several components of the cur- rent window - in our case several setters for the text fields of the frame.
set-window	This package contains procedures setting several component of certain frames or dialogs - in our case one procedure calling all procedure for the package 'set' of the JCarConfigurator frame. Those container procedures are typical workflow procedures.
wait	This package contains all procedure waiting for several components.

#### Table 30.2

In our case all procedures might be useful. So we can move the whole 'JCarConfigurator' Package from the procbuilder Package directly under the Procedures node. Press 'Yes' on the occurring 'Update Reference' dialog.

Now the procedures can be used by test cases or other procedures.

Finally our test suite should look like this:

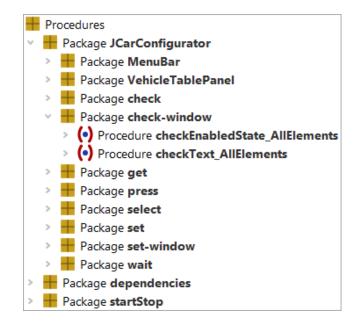


Figure 30.4: The test suite containing the procedures

Repeat that recording now for the 'Specials' and the 'Accessories' panel and just move the respective procedures or packages for the important components to the 'JCarConfigurator' package. In our case these are only the packages 'SpecialsPanel' and 'AccessoryTablePanel'.

The full test suite should then look like this:

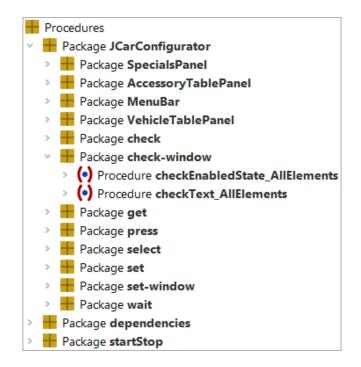


Figure 30.5: The procedures for all panels

We are ready for creating test cases using the automatically recorded test steps. You could of course also repeat that recording for all other dialogs, e.g. the 'Vehicles' dialog which can be reached via the menu 'Options' -> 'Vehicles' as well as for the 'Accessories' and 'Specials' dialogs.

You do not need to record the whole window all the time. You could also select 'Component only' to record one dedicated component or 'Component with children' to record procedures for a certain panel.

## 30.3 Configuration of the automated creation

#### 30.3.1 Introduction

In the previous example we used the file <code>qftest-7.1.3/demo/</code> procbuilder/carconfig-procbuilderdef.qft as configuration file for the automated creation. In this section we want to take a closer look at the configuration capabilities of QF-Test. So please open that file.

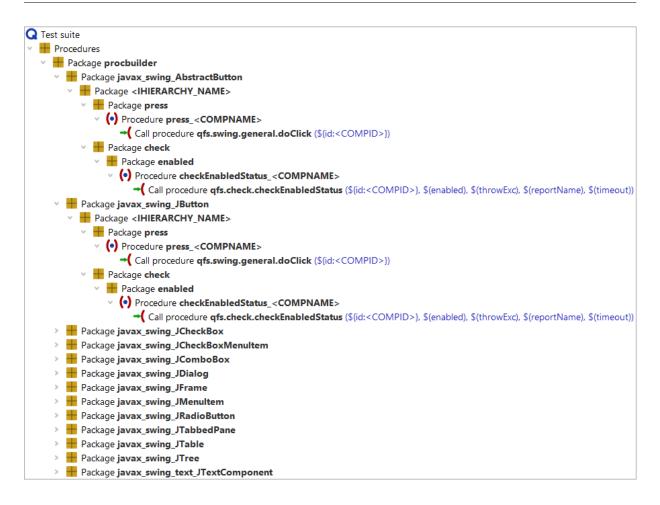


Figure 30.6: The current configuration

The package procbuilder is the root-package for all created packages. If you want to use another name for the automatically created procedures, feel free to rename the package.

If you open that package, you can see the 'class' level. This level describes the classes of the UI components which should be taken into account for creating the packages. The next levels contains information about the created package structure and procedures. You can find a detailed description of those capabilities in the manual in chapter The Procbuilder definition file.

#### 30.3.2 First example

In our first example we want to create a new settings file which we will build up step by step.

Please perform following actions at the beginning:

#### 30.3. Configuration of the automated creation

- Open a new test suite file and save it. Specify a name like 'mySettings.qft' or something similar.
- Create a new Package with the name 'myProcedures'.

The new test suite should look like this:

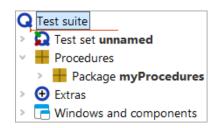


Figure 30.7: The own configuration file

Now we are ready to configure procedure templates for specific classes. Let us create procedures for the text-fields on the main panel of the CarConfigurator first. It might be interesting for our project to check the content of those text-fields, so we need to create procedures for checking the text of each text-field.

To create those procedures we have to insert a new package to the package 'myProcedures'. Please call that package 'javax\_swing\_JTextField'. 'javax.swing.JTextField' is the class of those text-fields, but we have to use '\_' in the package's name, because '.' is not allowed in that attribute. This package will instruct QF-Test to create the procedures in it, if it meets a component of the class 'javax.swing.JTextField' only. That is an important aspect, because we should realize that procedure templates can be created per classes of components.

Consecutively we want to create a template for a procedure checking the text of a component. The template procedure has to be part of another package. The name of this package should indicate its purpose, e.g. 'checkers'. After creating the 'checkers' package you can add a procedure 'checkText' to it. The procedure should contain a 'Check Text' node which is meant to check the text of components. Please insert such a 'Check text' node via right mouse click at the expanded procedure and Insert node  $\rightarrow$ Check nodes  $\rightarrow$ Check Text]. Specify (client) as client, 'dummy' as QF-Test component ID and (text) as text.

Of course, after confirming these entries, we will get a warning that a component named 'dummy' does not exist. In this case we are allowed to ignore this warning.

The test suite will then look like this:

Test suite	Check text
Q Test suite	Client
> 🛃 Test set unnamed	\$(client)
Procedures	
Package myProcedures	C QF-Test component ID
Package javax_swing_JTextField	dummy
Package checkers	
<ul> <li>Procedure checkText</li> </ul>	Text
Check text \$(text) dummy	\$(text)
> 🔁 Extras	
> 📑 Windows and components	\$ 🗌 As regexp
	\$ 🗌 Negate
	Check type identifier
	default
	Timeout
	Result handling
	OK Cancel

Figure 30.8: The checkText procedure

The procedure contains an empty default value for the parameter 'text'.

Test suite	Procedure		
C Test suite	Name		
<ul> <li>Procedures</li> <li>Package myProcedures</li> <li>Package javax_swing_JTextField</li> <li>Package checkers</li> <li>Procedure checkText</li> <li>Check text \$(text) dummy</li> </ul>	checkText       Image: Name       text         Maximum error level	eter default values Value	
<ul> <li>O Extras</li> <li>Windows and components</li> </ul>	Exception QF-Test ID		~
	Delay before (ms)	Delay after (ms)	
	Comment		
	ОК		Cancel

Figure 30.9: The <code>checkText</code> procedure with parameters

Now we have nearly completed our first procedure template, but we have to think about

#### 30.3. Configuration of the automated creation

one more issue. Each component has its specific and unique QF-Test ID, so it would be great to use that QF-Test ID attribute immediately during creation, otherwise we have to do that manually afterwards. We also want the procedure to have a component-related name instead of 'checkText' only. The place holder <COMPID> points out to QF-Test to replace it with the QF-Test ID of the current component. So we have to change the procedure name to checkText\_<COMPID> and we have to insert the <COMPID> place holder directly in the QF-Test component ID attribute of the 'Check text' node.

Test suite	Procedure		
<ul> <li>♀ Test suite</li> <li>&gt; ► Test set unnamed</li> <li>♥ Procedures</li> <li>♥ Package javax_swing_JTextField</li> <li>♥ Package checkers</li> <li>♥ Package checkText_<compid></compid></li> <li>♥ Check text \$(text) &lt; COMPID&gt;</li> <li>&gt; ● Extras</li> <li>&gt; ● Windows and components</li> </ul>	Name checkText_ <compid> + / × + Para Name text Maximum error level Exception QF-Test ID</compid>	ameter default values Value	~
	Delay before (ms)	Delay after (ms)	
	ОК		Cancel

Finally the template procedure should look like this:

Figure 30.10: Using the <COMPID> place holder

We are ready to begin now. First we have to order QF-Test to use that file. That is achieved via the options of QF-Test. Open the options dialog via  $\boxed{Edit \rightarrow Options}$  and change to 'Record' -> 'Procedures.' Specify the path of your 'mySettings.qft' for the 'Configuration file for recorded procedures' and press 'OK'.

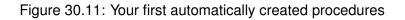
Then start the CarConfigurator again. Once it is up and running, please proceed with following steps:

- Press the 'Record procedures' button.
- Perform a right mouse-click in the SUT.
- Select 'Whole window'.
- Press the 'Record procedures' button again.

#### 30.3. Configuration of the automated creation

Test suite	Procedure	
Q Test suite	Name	
Test set Automated procedures tests	checkText_AccessoriesCalc	ulatorPanelPrice
Dependency reference dependencies.SUTstarted		
Procedures	🕂 🗹 📩 📩 Param	neter default values
> 🖶 Package dependencies	Name Value	
> 🖶 Package startStop	text	
Package myProcedures		
V 🕂 Package checkers	Maximum error level	
Procedure checkText_AccessoriesCalculatorPanelPrice	Exception	~
Check text \$(text) AccessoriesCalculatorPanelPrice		
Procedure checkText_BasePrice	QF-Test ID	
Check text \$(text) BasePrice		
<ul> <li>Procedure checkText_CalculatedPrice</li> </ul>		
Check text \$(text) CalculatedPrice	Delay before (ms)	Delay after (ms)
<ul> <li>Procedure checkText_DiscountValue</li> </ul>		
Check text \$(text) DiscountValue	Comment	
<ul> <li>Procedure checkText_SpecialsPrice</li> </ul>		
Check text \$(text) SpecialsPrice	automatically generated	
> 🕀 Extras		
> 📑 Windows and components	OK	Cancel

Congratulations, you have instructed QF-Test to record test steps for you.



#### 30.3.3 Using the current text for checking

The 'checkText' procedures have a parameter 'text' for the text to check. Up to now we have to specify that value each time we call that Procedure of the according text-field. Let us assume we have a scenario where we intend to check the default values of the CarConfigurator after its startup. In this case we would have to add the single procedure calls of all four 'checkText' procedures and specify the according values. However, QF-Test offers a place holder to set the current value of a text-field automatically to the created procedures. Therefore we have to change the default-value of the 'text' parameter to <CURRENTVALUE> in our configuration test suite. After that you have to ensure that the myProcedures package doesn't exist under the Procedures node anymore, then re-create the packages like in the previous example. Otherwise QF-Test will create a package myProcedures1 to ensure that each creation process has its own unique target package.

The configuration file looks like this:

Test suite	Procedure		
Test suite     Test set unnamed	Name checkText_ <compid></compid>		
<ul> <li>Procedures</li> <li>Package myProcedures</li> <li>Package javax_swing_JTextField</li> </ul>	+ / × ↑ → Para Name	Value	
<ul> <li>Package checkers</li> <li>Procedure checkText_<compid></compid></li> <li>Check text \$(text) <compid></compid></li> <li>Extras</li> <li>Windows and components</li> </ul>	text Maximum error level Exception QF-Test ID	<currentvalue></currentvalue>	~
	Delay before (ms)	Delay after (ms)	
	Comment OK	Cancel	

Figure 30.12: The configuration for the current text

The newly created procedures displays like below:

Test suite	Procedure
Q Test suite	Name
Test set Automated procedures tests	checkText_AccessoriesCalculatorPanelPrice
Dependency reference dependencies.SUTstarted	
Procedures	+ 🖌 🗙 📫 🕂 Parameter default values
Package dependencies	Name Value
> 🖶 Package <b>startStop</b>	text \$0.00
Package myProcedures	
V 🖶 Package checkers	Maximum error level
<ul> <li>Procedure checkText_AccessoriesCalculatorPanelPrice</li> </ul>	Exception 👻
Check text \$(text) AccessoriesCalculatorPanelPrice	
Procedure checkText_BasePrice	QF-Test ID
Procedure checkText_CalculatedPrice	
Procedure checkText_DiscountValue	
Procedure checkText_SpecialsPrice	Delay before (ms) Delay after (ms)
Extras	
> F Windows and components	Comment
	automatically generated
	OK Cancel

Figure 30.13: The created procedures with the current text

#### 30.3.4 Creating container procedures

In the previous example we have created the 'checkText' procedure for checking single text-fields. Now you should be able to create a 'set' procedure for setting the text of a text-field or a 'press' procedure for pressing a button. Those procedures are created similarly. Keep in mind that all those procedures work with one single component.

In some test scenarios it might be interesting to call one procedure for checking all components of a specific dialog or specific panel or to fill all visible text-fields. Because such procedures deal with components which act as containers, we call those procedures 'container'-procedures.

In our case we could consider one procedure calling all single 'checkText' procedures of the CarConfigurator. But how to create such a procedure like that?

First of all add a new 'class'-package to your 'mySettings.qft' file with the name 'javax\_swing\_JFrame'. The main window of the JCarConfigurator is an instance of javax.swing.JFrame, that is why we have to use that specific package name. The 'class'-package has to contain a 'type'-package which we call 'checkers-window' now. The 'type'-package has to contain a procedure checkTextOfElements\_<COMPID>, intended to perform the actual checks. We use the <COMPID> placeholder again for the procedure name to determine which dialog is affected by the created procedures.

The next step is to specify the procedure content. Let us see how we can configure this.

We have four text-fields which can be checked by using the same procedure checkText\_<COMPID>. QF-Test allows to call all four procedures with just one configured Procedure call node. Therefore insert a Procedure call for the procedure javax\_swing\_JTextField.checkers.checkText\_<CCOMPID>.

The configuration file should look like below:

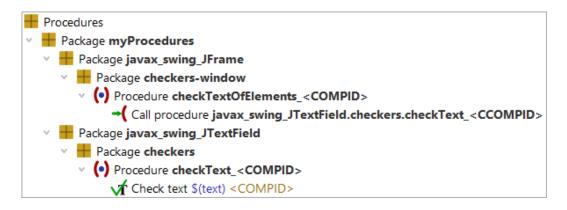


Figure 30.14: The template for container procedures

The last step is to make QF-Test to create a 'container' procedure and not a normal

Edit: Procedure  $\times$ Name checkTextOfElements\_<COMPID> + / × + Parameter default values OK Cancel

Figure 30.15: Usage of @FORCHILDREN tag

Now you can record the procedures like in the previous examples. Do not forget to remove the 'myProcedures' from Procedures before. You should then get something like this under Procedures:

'component' procedure. This can be achieved by setting @FORCHILDREN in the comment attribute of the checkTextOfElements\_<COMPID> procedure.

		ter delaan valdes	
Name	Value		
Maximum error l	evel		
Exception			~
QF-Test ID			
Delay before (ms	)	Delay after (ms)	
Delay before (ins	)	Delay after (IIIs)	
🗹 Comment			
@FORCHILDREN	I		
	-		

Test suite	Procedure			
Q Test suite	Name			
Test set Automated procedures tests	checkTextOfElements_JCarConfigurator			
Dependency reference dependencies.SUTstarted				
Procedures	🛨 🧹 🗶 🕂 🕂 Para	meter default values		
> 🖶 Package dependencies	Name Value			
> 🖶 Package startStop				
Package myProcedures				
> 🖶 Package checkers	Maximum error level			
V 🖶 Package checkers-window	Exception	~		
Procedure checkTextOfElements_JCarConfigurator				
Call procedure myProcedures.checkers.checkText_BasePrice	QF-Test ID			
Call procedure myProcedures.checkers.checkText_SpecialsPrice				
Call procedure myProcedures.checkers.checkText_AccessoriesCalculatc				
Call procedure myProcedures.checkers.checkText_DiscountValue	Delay before (ms)	Delay after (ms)		
Call procedure myProcedures.checkers.checkText_CalculatedPrice				
> 🔁 Extras	Comment			
> F Windows and components				
< >	OK	Cancel		

Figure 30.16: The created container procedures

Note QF-Test replaces the 'class' part of the procedure-calls by the top package of the configuration. In our case this is 'myProcedures'.

#### 30.3.5 Using the current value of child components

We could meet a similar situation like in the example <u>Using the current text for</u> <u>checking</u><sup>(334)</sup>, where we want to record the current values of each text-field during creation. In the example <u>Using the current text for checking</u><sup>(334)</sup> we used <CURRENTVALUE>. Now we have to specify the current text as parameter at the procedure-call in the 'container'-procedure 'checkTextElements'. Therefore we add the parameter 'text' to the procedure-call in your 'mySettings.qft' file and the value should be <CCURRENTVALUE>.



Figure 30.17: Configuration of <CCURRENTVALUE>

If you create the procedures again, you will see the current values at the procedure-calls. Please do not forget to remove the 'myProcedures' package from Procedures.

Test suite	Pr	ocedure		
Y Est set Automated procedures tests     ^     ^     ^	Nar	me		
Dependency reference dependencies.SUTstarted	che	eckTextOfEleme	nts_JCarConfic	gurator
V Procedures				
> 🖶 Package dependencies		· 🖌 📥 🖃 🖻	Parameter d	efault values
> 🖶 Package startStop	N	lame V	alue	
Package myProcedures				
V 🖶 Package checkers				
<ul> <li>Procedure checkText_AccessoriesCalculatorPanelPrice</li> </ul>				
Check text \$(text) AccessoriesCalculatorPanelPrice				
(•) Procedure checkText_BasePrice				
Check text \$(text) BasePrice				
Procedure checkText_CalculatedPrice	Ma	ximum error lev	el	
Check text \$(text) CalculatedPrice	Exc	eption		~
<ul> <li>(•) Procedure checkText_DiscountValue</li> </ul>				
Check text \$(text) DiscountValue	QF	-Test ID		
<ul> <li>(•) Procedure checkText_SpecialsPrice</li> </ul>				
Check text \$(text) SpecialsPrice	Del	ay before (ms)	Dalau	after (ms)
Package checkers-window	Dei	ay before (ms)	Delay	atter (ms)
Procedure checkTextOfElements_JCarConfigurator				
Call procedure myProcedures.checkers.checkText_BasePrice (\$0.00)	- 7	Comment		
Call procedure myProcedures.checkers.checkText_SpecialsPrice (\$0.0				
Call procedure myProcedures.checkers.checkText_AccessoriesCalcula				
Call procedure myProcedures.checkers.checkText_DiscountValue (0%				
Call procedure myProcedures.checkers.checkText_CalculatedPrice (\$ Call procedure myProcedures.checkers.checkText_CalculatedPrice (\$				
> 😧 Extras				
> Windows and components *		OK		Cancel

Figure 30.18: Test-suite using <CCURRENTVALUE>

If you take a closer look at the created procedure 'checkTextElements\_', you will see that each procedure-call gets the parameter text. Perhaps it might be convenient for test

#### 30.3. Configuration of the automated creation

development to set the according 'text' parameters as default values for the 'container'procedure. To achieve this you have to add one more parameter to the procedure call in the settings file. The name of the parameter has to be <CCOMPID> and the value will be <CCURRENTVALUE>. Then you have to change the value of the text parameter of that procedure-call to \$ (<CCOMPID>). The configuration should look like this now:

Test suite	Procedure c	all	
Q Test suite	(•) Procedure name javax_swing_JTextField.checkers.checkText_ <ccomf< th=""></ccomf<>		
V 🖶 Package myProcedures	Variable for return value		
V H Package javax_swing_JFrame			
<ul> <li>Package checkers-window</li> <li>(•) Procedure checkTextOfElements_<compid> <ul> <li>(•) Procedure javax_swing_JTextField.checkers.checkText_<ccompid> (\$(<ccompid>), <ccurrentv< li=""> </ccurrentv<></ccompid></ccompid></li></ul> </compid></li> <li>Package javax_swing_JTextField</li> <li>Package checkers</li> <li>(•) Procedure checkText_<compid></compid></li> </ul>	Name text		5
Check text \$(text) <compid>  Check text \$(text) <compid>  Check text \$(text) <compid>  Check text \$(text) &lt; COMPID&gt;  Check text \$(text) &lt; COMPID&gt; Check text \$(text) &lt; COMPID&gt; Check text \$(text) &lt; COMPID&gt; Check text</compid></compid></compid>	<ccompid> QF-Test ID</ccompid>	\$( <ccompid>)</ccompid>	
, · · · · · · · · · · · · · · · · · · ·	ОК		Cancel

Figure 30.19: Parameters for container procedures

If you create the procedures again, you will see that the 'checkTextElements' procedure has four more parameters with the current value of the elements as default values. Additionally each single procedure call uses a variable - named after the component-id - as 'text' parameter.

Test suite	Procedure	
Q Test suite ^	Name	
V D Test set Automated procedures tests	checkTextOfE	lements_JCarConfigurator
Dependency reference dependencies.SUTstarted     Procedures	+ / ×	↑ → Parameter default values
> 🕂 Package dependencies	Name	Value
> 🕂 Package startStop	BasePrice	\$(BasePrice)
Package myProcedures	SpecialsPrice	\$(SpecialsPrice)
🗸 🖶 Package checkers	AccessoriesCa	al \$(AccessoriesCalculatorPanelPrice)
<ul> <li>Procedure checkText_AccessoriesCalculatorPanelPrice</li> </ul>	DiscountValu	e \$(DiscountValue)
Check text \$(text) AccessoriesCalculatorPanelPrice	CalculatedPrid	s (CalculatedPrice)
Procedure checkText_BasePrice		
Check text \$(text) BasePrice		
Procedure checkText_CalculatedPrice	Maximum erro	or level
Check text \$(text) CalculatedPrice	Exception	×
Procedure checkText_DiscountValue	Exception	
Check text \$(text) DiscountValue	QF-Test ID	
Procedure checkText_SpecialsPrice		
Check text \$(text) SpecialsPrice		
Package checkers-window	Delay before (	ms) Delay after (ms)
Procedure checkTextOfElements_JCarConfigurator		
Call procedure myProcedures.checkers.checkText_BasePrice (\$0.00)	Comment	
Call procedure myProcedures.checkers.checkText_SpecialsPrice (\$0.0	Comment	
Call procedure myProcedures.checkers.checkText_AccessoriesCalcula		
Call procedure myProcedures.checkers.checkText_DiscountValue (0%		
Call procedure myProcedures.checkers.checkText_CalculatedPrice (\$		
> 🕁 Extras	OK	Cancel

Figure 30.20: Parameters for container procedures in test suite

### 30.3.6 More configuration capabilities

As you have seen in the previous sections there is a lot of configuration capability for the automated creation of basic procedures. But there are even more options available. Please take a look at the manual chapter The Procbuilder definition file.