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

’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 part IV 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 for Java, part II for Web and part III for native Windows applications.

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

Feedback

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 UI testing with QF-Test 1

1 Working 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 Creating your own Test-suite (Java) 18
   2.1 Starting the Application ....................................... 18
   2.2 Recording Actions .............................................. 25
   2.3 Recording Checks .............................................. 27
   2.4 Setting up a Test-suite ....................................... 27
   2.5 Stopping the Demo ............................................. 29
   2.6 Running the whole Test-suite ................................ 30

3 Writing a Procedure (Java) 32
   3.1 Identifying reusable parts ..................................... 32
   3.2 Manual creation of procedures ............................... 33
   3.3 Transforming nodes into procedures ........................ 37
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Components (Java)</td>
<td>38</td>
</tr>
<tr>
<td>4.1</td>
<td>Addressing subitems of tables, lists and trees</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>Windows and components Section</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>Using the Debugger (Java)</td>
<td>47</td>
</tr>
<tr>
<td>5.1</td>
<td>Setting a Breakpoint</td>
<td>48</td>
</tr>
<tr>
<td>5.2</td>
<td>Stepping Through a Test or Sequence</td>
<td>49</td>
</tr>
<tr>
<td>5.3</td>
<td>Skipping Execution of Nodes</td>
<td>51</td>
</tr>
<tr>
<td>5.4</td>
<td>Error or Exception triggering Debugging Mode</td>
<td>52</td>
</tr>
<tr>
<td>5.5</td>
<td>Resolving Errors directly from the Run-log</td>
<td>54</td>
</tr>
<tr>
<td>5.6</td>
<td>Pause Execution</td>
<td>56</td>
</tr>
<tr>
<td>6</td>
<td>Variables and Procedure Parameters (Java)</td>
<td>57</td>
</tr>
<tr>
<td>6.1</td>
<td>Procedure using a variable</td>
<td>57</td>
</tr>
<tr>
<td>6.2</td>
<td>The Variable Bindings table</td>
<td>61</td>
</tr>
<tr>
<td>6.3</td>
<td>Advanced debugging of variable bindings</td>
<td>65</td>
</tr>
<tr>
<td>6.4</td>
<td>Setting Variables</td>
<td>68</td>
</tr>
<tr>
<td>6.5</td>
<td>Variable binding levels</td>
<td>70</td>
</tr>
<tr>
<td>7</td>
<td>The Standard Library (Java)</td>
<td>73</td>
</tr>
<tr>
<td>7.1</td>
<td>Inspecting the Standard Library</td>
<td>74</td>
</tr>
<tr>
<td>7.2</td>
<td>Selected Packages and Procedures</td>
<td>75</td>
</tr>
<tr>
<td>7.2.1</td>
<td>The Checkbox Packages</td>
<td>75</td>
</tr>
<tr>
<td>7.2.2</td>
<td>The Combobox/Combo Packages</td>
<td>75</td>
</tr>
<tr>
<td>7.2.3</td>
<td>The General Packages</td>
<td>76</td>
</tr>
<tr>
<td>7.2.4</td>
<td>The List Packages</td>
<td>76</td>
</tr>
<tr>
<td>7.2.5</td>
<td>The Menu Packages</td>
<td>76</td>
</tr>
<tr>
<td>7.2.6</td>
<td>The Table Packages</td>
<td>76</td>
</tr>
<tr>
<td>7.2.7</td>
<td>The Tree Packages</td>
<td>77</td>
</tr>
<tr>
<td>7.2.8</td>
<td>The Cleanup Packages</td>
<td>77</td>
</tr>
<tr>
<td>7.2.9</td>
<td>The Run-log Package</td>
<td>78</td>
</tr>
<tr>
<td>7.2.10</td>
<td>The Run-log.Screenshots Package</td>
<td>78</td>
</tr>
<tr>
<td>7.2.11</td>
<td>The Shellutils Package</td>
<td>78</td>
</tr>
</tbody>
</table>
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2.12</td>
<td>The Utils Package</td>
<td>79</td>
</tr>
<tr>
<td>7.2.13</td>
<td>The Database Package</td>
<td>79</td>
</tr>
<tr>
<td>7.2.14</td>
<td>The Check Package</td>
<td>80</td>
</tr>
<tr>
<td>7.2.15</td>
<td>The Databinder Package</td>
<td>80</td>
</tr>
<tr>
<td>8</td>
<td>Control structures (Java)</td>
<td>81</td>
</tr>
<tr>
<td>8.1</td>
<td>If - else</td>
<td>81</td>
</tr>
<tr>
<td>8.2</td>
<td>Loops</td>
<td>84</td>
</tr>
<tr>
<td>9</td>
<td>It’s time to start your own Application (Java)</td>
<td>92</td>
</tr>
<tr>
<td>II</td>
<td>Web UI testing with QF-Test</td>
<td>93</td>
</tr>
<tr>
<td>10</td>
<td>Working with a Sample Test-suite (Web)</td>
<td>95</td>
</tr>
<tr>
<td>10.1</td>
<td>Loading the Test-suite</td>
<td>95</td>
</tr>
<tr>
<td>10.2</td>
<td>Starting the Browser</td>
<td>97</td>
</tr>
<tr>
<td>10.3</td>
<td>First Test-case</td>
<td>99</td>
</tr>
<tr>
<td>10.4</td>
<td>Second Test-case - with Error</td>
<td>101</td>
</tr>
<tr>
<td>10.5</td>
<td>The Run-log for Error Diagnosis</td>
<td>102</td>
</tr>
<tr>
<td>10.6</td>
<td>Getting Help</td>
<td>105</td>
</tr>
<tr>
<td>10.7</td>
<td>Stopping the Application</td>
<td>106</td>
</tr>
<tr>
<td>10.8</td>
<td>A full Test Run</td>
<td>107</td>
</tr>
<tr>
<td>10.9</td>
<td>Report Generation</td>
<td>107</td>
</tr>
<tr>
<td>11</td>
<td>Creating your own Test-suite (Web)</td>
<td>110</td>
</tr>
<tr>
<td>11.1</td>
<td>Creating the Setup Sequence</td>
<td>110</td>
</tr>
<tr>
<td>11.2</td>
<td>Recording Actions</td>
<td>117</td>
</tr>
<tr>
<td>11.3</td>
<td>Recording Checks</td>
<td>119</td>
</tr>
<tr>
<td>11.4</td>
<td>Setting up a Test-suite</td>
<td>119</td>
</tr>
<tr>
<td>11.5</td>
<td>Stopping the Demo</td>
<td>121</td>
</tr>
<tr>
<td>11.6</td>
<td>Running the whole Test-suite</td>
<td>122</td>
</tr>
<tr>
<td>11.7</td>
<td>Improved Web Component Recognition</td>
<td>123</td>
</tr>
</tbody>
</table>
12 Writing a Procedure (Web) 125
  12.1 Identifying reusable parts ........................................... 125
  12.2 Manual creation of procedures ..................................... 126
  12.3 Transforming nodes into procedures ............................... 130

13 Components (Web) 131
  13.1 Addressing subitems of tables, lists and trees ..................... 131
  13.2 Windows and components Section .................................. 133

14 Using the Debugger (Web) 140
  14.1 Setting a Breakpoint ................................................... 141
  14.2 Stepping Through a Test or Sequence ............................. 142
  14.3 Skipping Execution of Nodes ......................................... 144
  14.4 Error or Exception triggering Debugging Mode ..................... 145
  14.5 Resolving Errors directly from the Run-log ....................... 147
  14.6 Pause Execution ...................................................... 149

15 Variables and Procedure Parameters (Web) 150
  15.1 Procedure using a variable .......................................... 150
  15.2 The Variable Bindings table ........................................ 154
  15.3 Advanced debugging of variable bindings .......................... 158
  15.4 Setting Variables .................................................... 161
  15.5 Variable binding levels ............................................. 163

16 The Standard Library (Web) 166
  16.1 Inspecting the Standard Library ..................................... 167
  16.2 Selected Packages and Procedures ................................. 168
    16.2.1 The Checkbox Package ........................................ 168
    16.2.2 The Select Package ........................................... 168
    16.2.3 The General Package ......................................... 169
    16.2.4 The Table Package ........................................... 169
    16.2.5 The Run-log Package ......................................... 169
    16.2.6 The Run-log.Screenshots Package ............................. 169
CONTENTS

16.2.7 The Shellutils Package ........................................... 170
16.2.8 The Utils Package .................................................. 170
16.2.9 The Database Package ............................................. 171
16.2.10 The Check Package ............................................... 171
16.2.11 The Databinder Package ........................................... 172

17 Control structures (Web) .............................................. 173
  17.1 If - else .............................................................. 173
  17.2 Loops ............................................................... 176

18 It’s time to start your own Application (Web) ....................... 184

III Native Windows UI testing with QF-Test .......................... 185

19 Working with a Sample Test-suite (Win) ............................ 187
  19.1 Loading the Test-suite ............................................ 187
  19.2 Starting the Application ......................................... 189
  19.3 First Test-case ..................................................... 191
  19.4 Second Test-case - with Error ................................... 193
  19.5 The Run-log for Error Diagnosis ................................ 194
  19.6 Getting Help ....................................................... 197
  19.7 Stopping the Application ......................................... 198
  19.8 A full Test Run .................................................... 199
  19.9 Report Generation ................................................ 199

20 Creating your own Test-suite (Win) .................................. 202
  20.1 Starting the Application ......................................... 202
  20.2 Recording Actions ................................................ 208
  20.3 Recording Checks ................................................ 210
  20.4 Setting up a Test-suite ........................................... 211
  20.5 Stopping the Demo ................................................ 213
  20.6 Running the whole Test-suite ................................... 213
CONTENTS

21 Writing a Procedure (Win)  215
  21.1 Identifying reusable parts . . . . . . . . . . . . . . . . . . . . . . . . .  215
  21.2 Manual creation of procedures . . . . . . . . . . . . . . . . . . . . . . .  216
  21.3 Transforming nodes into procedures . . . . . . . . . . . . . . . . . . . . .  220

22 Components (Win)  221
  22.1 Addressing subitems of tables, lists and trees . . . . . . . . . . . . . . . . .  221

23 Using the Debugger (Win)  224
  23.1 Setting a Breakpoint . . . . . . . . . . . . . . . . . . . . . . . . . . . . .  225
  23.2 Stepping Through a Test or Sequence . . . . . . . . . . . . . . . . . . . .  226
  23.3 Skipping Execution of Nodes . . . . . . . . . . . . . . . . . . . . . . . . .  228
  23.4 Error or Exception triggering Debugging Mode . . . . . . . . . . . . . . . . .  229
  23.5 Resolving Errors directly from the Run-log . . . . . . . . . . . . . . . . . .  231
  23.6 Pause Execution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .  233

24 Variables and Procedure Parameters (Win)  234
  24.1 Procedure using a variable . . . . . . . . . . . . . . . . . . . . . . . . . .  234
  24.2 The Variable Bindings table . . . . . . . . . . . . . . . . . . . . . . . . .  238
  24.3 Advanced debugging of variable bindings . . . . . . . . . . . . . . . . . . .  242
  24.4 Setting Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .  245
  24.5 Variable binding levels . . . . . . . . . . . . . . . . . . . . . . . . . . .  247

25 The Standard Library (Win)  250
  25.1 Inspecting the Standard Library . . . . . . . . . . . . . . . . . . . . . . . .  251
  25.2 Selected Packages and Procedures . . . . . . . . . . . . . . . . . . . . . .  252
    25.2.1 The Run-log Package . . . . . . . . . . . . . . . . . . . . . . . . . .  252
    25.2.2 The Run-log.Screenshots Package . . . . . . . . . . . . . . . . . . . .  252
    25.2.3 The Shellutils Package . . . . . . . . . . . . . . . . . . . . . . . . .  252
    25.2.4 The Utils Package . . . . . . . . . . . . . . . . . . . . . . . . . . .  253
    25.2.5 The Database Package . . . . . . . . . . . . . . . . . . . . . . . . .  253
    25.2.6 The Check Package . . . . . . . . . . . . . . . . . . . . . . . . . . .  254
    25.2.7 The Databinder Package . . . . . . . . . . . . . . . . . . . . . . . . .  254
26 Control structures (Win) 255

26.1 If - else ................................................................. 255
26.2 Loops ................................................................. 258

27 It’s time to start your own Application (Win) 266

IV Advanced features of QF-Test 267

28 Data driven Testing: Running one Test-case with different test data 269

28.1 Situation ................................................................. 269
28.2 Traditional way of implementing data driven testing ....................... 270
28.3 Data driver concept .................................................. 271
28.4 Summary ................................................................. 277

29 Dependencies: Automatically ensuring correct prerequisites for each Test-case 279

29.1 General ................................................................. 279
29.2 Ensuring prerequisites ................................................ 281
29.3 Combining dependencies ............................................. 284
29.4 Error and exception handling ....................................... 288
   29.4.1 Error handling ................................................ 288
   29.4.2 Exception handling ............................................ 290
   29.4.3 Summary ....................................................... 293
29.5 More about dependencies ........................................... 293

30 Automated creation of basic procedures 295

30.1 General ................................................................. 295
30.2 Automated creation of procedures ................................... 297
30.3 Configuration of the automated creation ................................ 302
   30.3.1 Introduction .................................................. 302
   30.3.2 First example ................................................. 303
   30.3.3 Using the current text for checking ........................... 307
   30.3.4 Creating container procedures ................................. 309
30.3.5 Using the current value of child components ............... 312
30.3.6 More configuration capabilities ........................................... 316
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 JCarConfigurator 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 Run-Log 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 Run-Log for the Completed Test-set .................. 15
1.17 Report Generation Properties ......................... 16
1.18 An HTML Report ..................................... 17

2.1 The Quickstart Wizard .................................. 19
2.2 Type of Application .................................... 20
2.3 Type of Executable ..................................... 21
2.4 Executable file selection. ............................. 22
2.5 Final Information ....................................... 23
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6</td>
<td>Generated Setup Sequence</td>
<td>24</td>
</tr>
<tr>
<td>2.7</td>
<td>The JCarConfigurator Demo Window</td>
<td>24</td>
</tr>
<tr>
<td>2.8</td>
<td>Recording actions on the JCarConfigurator Demo</td>
<td>25</td>
</tr>
<tr>
<td>2.9</td>
<td>The Recorded Sequence</td>
<td>26</td>
</tr>
<tr>
<td>2.10</td>
<td>The Renamed Sequence</td>
<td>26</td>
</tr>
<tr>
<td>2.11</td>
<td>The recorded check node</td>
<td>27</td>
</tr>
<tr>
<td>2.12</td>
<td>Start organizing the Test-suite</td>
<td>28</td>
</tr>
<tr>
<td>2.13</td>
<td>The Organization of your Test-suite</td>
<td>29</td>
</tr>
<tr>
<td>2.14</td>
<td>The simple cleanup sequence</td>
<td>30</td>
</tr>
<tr>
<td>2.15</td>
<td>The Run-Log of the Test-suite</td>
<td>31</td>
</tr>
<tr>
<td>3.1</td>
<td>Two identical test-steps</td>
<td>33</td>
</tr>
<tr>
<td>3.2</td>
<td>Create a procedure node</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>Fill in the procedure content</td>
<td>34</td>
</tr>
<tr>
<td>3.4</td>
<td>Insert a procedure call</td>
<td>35</td>
</tr>
<tr>
<td>3.5</td>
<td>Select a procedure</td>
<td>36</td>
</tr>
<tr>
<td>3.6</td>
<td>Test-suite with procedure</td>
<td>37</td>
</tr>
<tr>
<td>4.1</td>
<td>Addressing a table cell</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>Locating a component</td>
<td>42</td>
</tr>
<tr>
<td>4.3</td>
<td>Component tree</td>
<td>43</td>
</tr>
<tr>
<td>4.4</td>
<td>Details of the Component node</td>
<td>44</td>
</tr>
<tr>
<td>5.1</td>
<td>Set break point</td>
<td>48</td>
</tr>
<tr>
<td>5.2</td>
<td>Start test run</td>
<td>48</td>
</tr>
<tr>
<td>5.3</td>
<td>Remove break point</td>
<td>49</td>
</tr>
<tr>
<td>5.4</td>
<td>Stepping into a node</td>
<td>49</td>
</tr>
<tr>
<td>5.5</td>
<td>Stepping over a node</td>
<td>50</td>
</tr>
<tr>
<td>5.6</td>
<td>Stepping out of a node</td>
<td>50</td>
</tr>
<tr>
<td>5.7</td>
<td>Pause execution at first node of the second test-case</td>
<td>51</td>
</tr>
<tr>
<td>5.8</td>
<td>Skip over a node</td>
<td>51</td>
</tr>
<tr>
<td>5.9</td>
<td>Skip out of a node</td>
<td>52</td>
</tr>
<tr>
<td>5.10</td>
<td>Set debugger options to pause on error</td>
<td>53</td>
</tr>
</tbody>
</table>
10.6 The "First" Test-case ......................................................... 99
10.7 Details of the first Test-case ............................................. 100
10.8 The result view in the status line ....................................... 100
10.9 The Second Test-case ..................................................... 101
10.10 Details of the second Test-case ......................................... 101
10.11 Error in the second test-case ........................................... 102
10.12 Run-Log for the second test-case ..................................... 103
10.13 Error in the second test-case ........................................... 104
10.14 Screenshot node showing the error situation ...................... 105
10.15 The Cleanup Sequence .................................................. 106
10.16 Run-Log for the Completed Test-set ................................ 107
10.17 Report Generation Properties ......................................... 108
10.18 An HTML Report ........................................................ 109

11.1 The Quickstart Wizard .................................................. 111
11.2 Type of Application ........................................................ 112
11.3 Specification of the URL ............................................... 113
11.4 Final Information .......................................................... 114
11.5 Generated Setup Sequence ............................................. 115
11.6 The WebCarConfigurator demo page in the browser .............. 116
11.7 Recording actions on the WebCarConfigurator Demo ............. 117
11.8 The Recorded Sequence .................................................. 118
11.9 The Renamed Sequence .................................................. 118
11.10 The recorded check node ............................................... 119
11.11 Start organizing the Test-suite ....................................... 120
11.12 The Organization of your Test-suite ................................ 121
11.13 The simple cleanup sequence ........................................ 122
11.14 The Run-Log of the Test-suite ....................................... 123
11.15 Web resolvers as installed in the FirstWebTests.qft .......... 124

12.1 Two identical test-steps ................................................ 126
12.2 Create a procedure node ............................................... 127
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.3</td>
<td>Fill in the procedure content</td>
</tr>
<tr>
<td>12.4</td>
<td>Insert a procedure call</td>
</tr>
<tr>
<td>12.5</td>
<td>Select a procedure</td>
</tr>
<tr>
<td>12.6</td>
<td>Test-suite with procedure</td>
</tr>
<tr>
<td>13.1</td>
<td>Addressing a table cell</td>
</tr>
<tr>
<td>13.2</td>
<td>Locating a component</td>
</tr>
<tr>
<td>13.3</td>
<td>Component tree</td>
</tr>
<tr>
<td>13.4</td>
<td>Details of the Component node</td>
</tr>
<tr>
<td>14.1</td>
<td>Set break point</td>
</tr>
<tr>
<td>14.2</td>
<td>Start test run</td>
</tr>
<tr>
<td>14.3</td>
<td>Remove break point</td>
</tr>
<tr>
<td>14.4</td>
<td>Stepping into a node</td>
</tr>
<tr>
<td>14.5</td>
<td>Stepping over a node</td>
</tr>
<tr>
<td>14.6</td>
<td>Stepping out of a node</td>
</tr>
<tr>
<td>14.7</td>
<td>Pause execution at first node of the second test-case</td>
</tr>
<tr>
<td>14.8</td>
<td>Skip over a node</td>
</tr>
<tr>
<td>14.9</td>
<td>Skip out of a node</td>
</tr>
<tr>
<td>14.10</td>
<td>Set debugger options to pause on error</td>
</tr>
<tr>
<td>14.11</td>
<td>Test run stopped by error</td>
</tr>
<tr>
<td>14.12</td>
<td>Error Dialog</td>
</tr>
<tr>
<td>14.13</td>
<td>Update check node with current data</td>
</tr>
<tr>
<td>14.14</td>
<td>Corrected check node</td>
</tr>
<tr>
<td>15.1</td>
<td>Two almost identical test-steps</td>
</tr>
<tr>
<td>15.2</td>
<td>Procedure with hard coded value</td>
</tr>
<tr>
<td>15.3</td>
<td>The Details of the 'Procedure' node</td>
</tr>
<tr>
<td>15.4</td>
<td>'Check text' node</td>
</tr>
<tr>
<td>15.5</td>
<td>Second procedure calls 'checkFinalPrice'</td>
</tr>
<tr>
<td>15.6</td>
<td>Variable bindings</td>
</tr>
<tr>
<td>15.7</td>
<td>Popup menu for 'Additional node operations'</td>
</tr>
<tr>
<td>15.8</td>
<td>Variable bindings stack showing incorrect value</td>
</tr>
</tbody>
</table>
15.9 Continue Execution from here ........................................... 160
15.10 Details of the Set variable node ........................................ 162
15.11 Procedure returning a value ........................................... 163
16.1 The Standard Library ..................................................... 167
17.1 Setup Sequence with if-else structures ............................... 173
17.2 Wait for client to connect writes the result into the variable "isSUTRunning" ........................................... 174
17.3 If node evaluates the variable ......................................... 175
17.4 Transform a node into another one ................................... 177
17.5 Pack a node into another one ......................................... 178
17.6 Details of a Loop node .................................................. 180
17.7 The new test-case ......................................................... 182
17.8 Details of the Check items node ..................................... 183
19.1 The Test-suite FirstWinTests.qft ..................................... 188
19.2 The "Test-set: Simple Tests" Node ................................... 189
19.3 The Setup Node .......................................................... 189
19.4 The Sequence to start the SUT ....................................... 190
19.5 The Windows CarConfigurator Demo ................................ 191
19.6 The "First" Test-case ..................................................... 191
19.7 Details of the first Test-case .......................................... 192
19.8 The result view in the status line .................................... 192
19.9 The Second Test-case ................................................... 193
19.10 Details of the second Test-case ..................................... 193
19.11 Error in the second test-case ....................................... 194
19.12 Run-Log for the second test-case ................................... 195
19.13 Error in the second test-case ....................................... 196
19.14 Screenshot node showing the error situation ..................... 197
19.15 The Cleanup Sequence ................................................ 198
19.16 Run-Log for the Completed Test-set ................................ 199
19.17 Report Generation Properties ....................................... 200
19.18 An HTML Report ....................................................... 201
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.1</td>
<td>The Quickstart Wizard</td>
<td>203</td>
</tr>
<tr>
<td>20.2</td>
<td>Type of Application</td>
<td>204</td>
</tr>
<tr>
<td>20.3</td>
<td>Windows executable file selection.</td>
<td>205</td>
</tr>
<tr>
<td>20.4</td>
<td>Final Information</td>
<td>206</td>
</tr>
<tr>
<td>20.5</td>
<td>Generated Setup Sequence</td>
<td>207</td>
</tr>
<tr>
<td>20.6</td>
<td>The Windows CarConfigurator Demo</td>
<td>207</td>
</tr>
<tr>
<td>20.7</td>
<td>Recording actions on the CarConfiguratorForms Demo</td>
<td>208</td>
</tr>
<tr>
<td>20.8</td>
<td>The Recorded Sequence</td>
<td>209</td>
</tr>
<tr>
<td>20.9</td>
<td>The Renamed Sequence</td>
<td>209</td>
</tr>
<tr>
<td>20.10</td>
<td>The recorded check node</td>
<td>210</td>
</tr>
<tr>
<td>20.11</td>
<td>Start organizing the Test-suite</td>
<td>212</td>
</tr>
<tr>
<td>20.12</td>
<td>The Organization of your Test-suite</td>
<td>212</td>
</tr>
<tr>
<td>20.13</td>
<td>The simple cleanup sequence</td>
<td>213</td>
</tr>
<tr>
<td>20.14</td>
<td>The Run-Log of the Test-suite</td>
<td>214</td>
</tr>
<tr>
<td>21.1</td>
<td>Two identical test-steps</td>
<td>216</td>
</tr>
<tr>
<td>21.2</td>
<td>Create a procedure node</td>
<td>217</td>
</tr>
<tr>
<td>21.3</td>
<td>Fill in the procedure content</td>
<td>217</td>
</tr>
<tr>
<td>21.4</td>
<td>Insert a procedure call</td>
<td>218</td>
</tr>
<tr>
<td>21.5</td>
<td>Select a procedure</td>
<td>219</td>
</tr>
<tr>
<td>21.6</td>
<td>Test-suite with procedure</td>
<td>220</td>
</tr>
<tr>
<td>22.1</td>
<td>Addressing a table cell</td>
<td>221</td>
</tr>
<tr>
<td>23.1</td>
<td>Set break point</td>
<td>225</td>
</tr>
<tr>
<td>23.2</td>
<td>Start test run</td>
<td>225</td>
</tr>
<tr>
<td>23.3</td>
<td>Remove break point</td>
<td>226</td>
</tr>
<tr>
<td>23.4</td>
<td>Stepping into a node</td>
<td>226</td>
</tr>
<tr>
<td>23.5</td>
<td>Stepping over a node</td>
<td>227</td>
</tr>
<tr>
<td>23.6</td>
<td>Stepping out of a node</td>
<td>227</td>
</tr>
<tr>
<td>23.7</td>
<td>Pause execution at first node of the second test-case</td>
<td>228</td>
</tr>
<tr>
<td>23.8</td>
<td>Skip over a node</td>
<td>228</td>
</tr>
<tr>
<td>23.9</td>
<td>Skip out of a node</td>
<td>229</td>
</tr>
</tbody>
</table>
23.10 Set debugger options to pause on error ................................. 230
23.11 Test run stopped by error .................................................. 230
23.12 Error Dialog ...................................................................... 231
23.13 Update check node with current data .................................... 232
23.14 Corrected check node ......................................................... 233

24.1 Two almost identical test-steps ............................................. 235
24.2 Procedure with hard coded value .......................................... 236
24.3 The Details of the 'Procedure' node ....................................... 237
24.4 'Check text' node ................................................................. 238
24.5 Second procedure calls 'checkFinalPrice' ............................... 239
24.6 Variable bindings ............................................................... 240
24.7 Popup menu for 'Additional node operations' ......................... 241
24.8 Variable bindings stack showing incorrect value ..................... 242
24.9 Continue Execution from here .............................................. 244
24.10 Details of the Set variable node .......................................... 246
24.11 Procedure returning a value ................................................ 247

25.1 The Standard Library ......................................................... 251

26.1 Setup Sequence with if-else structures ................................ 255
26.2 Wait for client to connect writes the result into the variable "isSUTRunning" 256
26.3 If node evaluates the variable .............................................. 257
26.4 Transform a node into another one ...................................... 259
26.5 Pack a node into another one .............................................. 260
26.6 Details of a Loop node ....................................................... 262
26.7 The new test-case ............................................................. 264
26.8 Details of the Check items node ......................................... 265

28.1 Traditional way of data driven testing ................................. 270
28.2 Traditional way with a nested Test-set ................................. 270
28.3 Data table dialog .............................................................. 272
28.4 The filled data table .......................................................... 273
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.5</td>
<td>Test-set with Data driver</td>
<td>273</td>
</tr>
<tr>
<td>28.6</td>
<td>Using the $\text{(discount)}$ parameter</td>
<td>274</td>
</tr>
<tr>
<td>28.7</td>
<td>Full data table</td>
<td>275</td>
</tr>
<tr>
<td>28.8</td>
<td>Name for run-log and report attribute</td>
<td>276</td>
</tr>
<tr>
<td>28.9</td>
<td>Run-log with different names per Test-case</td>
<td>277</td>
</tr>
<tr>
<td>29.1</td>
<td>First Test-set of dependencies_work.qft</td>
<td>280</td>
</tr>
<tr>
<td>29.2</td>
<td>First Test-set of dependencies_work.qft</td>
<td>281</td>
</tr>
<tr>
<td>29.3</td>
<td>Sample test-suite with the first Dependency</td>
<td>281</td>
</tr>
<tr>
<td>29.4</td>
<td>The run-log of the execution</td>
<td>282</td>
</tr>
<tr>
<td>29.5</td>
<td>Procedure startStop.startApplication</td>
<td>283</td>
</tr>
<tr>
<td>29.6</td>
<td>The test-suite with a Dependency reference</td>
<td>283</td>
</tr>
<tr>
<td>29.7</td>
<td>Ensuring prerequisites for Test-case 'Discount 15'</td>
<td>284</td>
</tr>
<tr>
<td>29.8</td>
<td>'vehicles dialog opened' Dependency</td>
<td>286</td>
</tr>
<tr>
<td>29.9</td>
<td>Implementation of specified test-cases</td>
<td>287</td>
</tr>
<tr>
<td>29.10</td>
<td>Run-log of nested Dependencies</td>
<td>288</td>
</tr>
<tr>
<td>29.11</td>
<td>Test-suite of Error handler</td>
<td>289</td>
</tr>
<tr>
<td>29.12</td>
<td>Dependency with Error handler</td>
<td>289</td>
</tr>
<tr>
<td>29.13</td>
<td>Run-log for Dependency with Error handler</td>
<td>290</td>
</tr>
<tr>
<td>29.14</td>
<td>Try-catch nodes in each Test-case</td>
<td>291</td>
</tr>
<tr>
<td>29.15</td>
<td>Test-suite with Catch</td>
<td>292</td>
</tr>
<tr>
<td>29.16</td>
<td>Run-log of a Dependency with Catch</td>
<td>293</td>
</tr>
<tr>
<td>30.1</td>
<td>Screenshot of test-suite</td>
<td>297</td>
</tr>
<tr>
<td>30.2</td>
<td>The test-suite automated_procedures_work.qft</td>
<td>298</td>
</tr>
<tr>
<td>30.3</td>
<td>The recorded procedures</td>
<td>299</td>
</tr>
<tr>
<td>30.4</td>
<td>The test-suite containing the procedures</td>
<td>301</td>
</tr>
<tr>
<td>30.5</td>
<td>The procedures for all panels</td>
<td>301</td>
</tr>
<tr>
<td>30.6</td>
<td>The current configuration</td>
<td>302</td>
</tr>
<tr>
<td>30.7</td>
<td>The own configuration file</td>
<td>303</td>
</tr>
<tr>
<td>30.8</td>
<td>The checkText procedure</td>
<td>304</td>
</tr>
<tr>
<td>30.9</td>
<td>The checkText procedure with parameters</td>
<td>305</td>
</tr>
</tbody>
</table>
30.10 Using the `<COMPID>` place holder ........................................ 306
30.11 Your first automatically created procedures .......................... 307
30.12 The configuration for the current text .................................. 308
30.13 The created procedures with the current text ......................... 309
30.14 The template for container procedures ................................. 310
30.15 Usage of `@FORCHILDREN` tag ...................................... 311
30.16 The created container procedures .................................. 312
30.17 Configuration of `<CURRENTVALUE>` ............................... 313
30.18 Test-suite using `<CURRENTVALUE>` ............................... 314
30.19 Parameters for container procedures ................................. 315
30.20 Parameters for container procedures in test-suite .................. 316
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\textsuperscript{84} or part III\textsuperscript{186} for native Windows programs, as those parts use the same scenarios but with different systems under test. Within part IV\textsuperscript{268} more advanced QF-Test features are explained, applicable for all supported UI technologies.
Chapter 1

Working with a Sample Test-suite (Java)

The video 'Tutorial Java 1'

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

1.1 Loading the Test-suite

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.

- Press the toolbar button to bring up the file open dialog
- Navigate to the subdirectory qftest-5.1.2/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:
1.1. Loading the Test-suite

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.
1.2 Starting the Application

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

- **Expand** the **Setup: Start Demo** node now.

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 and 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**:
1.2. Starting the Application

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 `$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:

- **Click** on the **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 "JCarConfigurator" 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

Let’s check out now what test-case “First” contains. There are four test-steps inside:

- **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.
1.3. First Test-case

- **Expand** the four **test-step nodes**.

![Test-case: First](image)

**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 [Creating your own Test-suite (Java)](18).

- **Please select** the **test-case "First"** node

- **Click** the replay button ➤.

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](image)

**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

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

- **Expand** the test-case "Second (with Error)".

![Image of test-case](image)

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**.

![Image of test-step](image)

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 ➤.

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

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)**.
1.5. The Run-log for Error Diagnosis

![Figure 1.12: Run-Log for the second test-case](image)

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:
1.5. The Run-log for Error Diagnosis

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:
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 Edit→Options→Run-logs→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 most comprehensive search can be achieved via Help→Online search... This navigates you to the search functionality on our homepage and allows querying throughout all available...
1.7. Stopping the Application

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](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:

**Action**

- Expand the **Cleanup: Stop Demo** node.

![Figure 1.15: The Cleanup Sequence](https://www.qfs.de/en/qf-test-manual.html)

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 JCarConfigurator 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 "JCarConfigurator" 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 to take a closer look:

```
Figure 1.16: 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 clean-up that will be explained with the advanced features in this tutorial (chapter 29[279]).

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.

**Action**
- Make sure the run-log of the test run is open.
1.9. Report Generation

- In the run-log window select File→Create report... to bring up the dialog for the report parameters.

![Report Generation Properties](image.png)

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.

**Action**

- 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:
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.

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

Figure 2.1: The Quickstart Wizard

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

**Action**

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

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

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

**Action**
- For this please use the **select executable** button on the right.
- Navigate to the subdirectory `qftest-5.1.2/demo/carconfig/` 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

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

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.
2.1. Starting the Application

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.
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 JCarConfigurator 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.

At the end of this section let's save our test-suite.
2.2. Recording Actions

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

**Action**

- Press the toolbar button or use the 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.

**2.2 Recording Actions**

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

**Figure 2.8: Recording actions on the JCarConfigurator 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 I5 in the last row.
2.2. Recording Actions

- 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 ■ button.

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

![Figure 2.9: The Recorded Sequence](image)

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](image)

Now, let us replay the recorded sequence.

**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.
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.

- 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](image)

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.
2.4. Setting up a Test-suite

- "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.

Action

- 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), context-menu (right mouse-button copy/paste) or by [Ctrl-X] and [Ctrl-V] keyboard commands.

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

Action

- Please rename the test-case node from "unnamed" to "First".
- Open the test-case node by clicking the '+' symbol.
2.5. Stopping the Demo

- **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 right-click.
- Choose **Transform node into...→Test-step**
- Repeat this for the *second* sequence node.

Your test-suite should now look like this:

![Figure 2.13: The Organization of your Test-suite](image)

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 **File→Exit**. All these options can be directly recorded and then used in the clean-up sequence.

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

- **Start recoding**.
- Record the menu SUT menu action **File→Exit**.
- You see the demo window close.
2.6. Running the whole Test-suite

- 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.

You should end up with the following:

![Figure 2.14: The simple cleanup sequence](image)

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:

- **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.

- It can be opened by clicking the toolbar button or via the menu option with its short-cut [Ctrl-L].
2.6. Running the whole Test-suite

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

Figure 2.15: The Run-Log of the Test-suite
Chapter 3

Writing a Procedure (Java)


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 FirstJavaTests.qft you already know from chapter one.

- Copy FirstJavaTests.qft from the subdirectory qftest-5.1.2/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.
Please have a look at the test-step "Reset" in the two test-cases. They are exactly the same.

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.
- **Open** the newly created ‘reset’ 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

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

- **Select** the test-step 'Reset', which is still open.

- **Select** [Insert→Procedure nodes→Procedure call] or use the [Ctrl-A] shortcut.

![Figure 3.4: Insert a procedure call](image)

- **Action**
  - 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.
3.2. Manual creation of procedures

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:
3.3 Transforming nodes into procedures

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:

- **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 [Ctrl-Shift-P] 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.

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.

- **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.

![Figure 4.1: Addressing a table cell](image)

The QF-Test component ID recorded is VehicleTable@Model&0. The single parts are:
4.1. Addressing subitems of tables, lists and trees

- `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&0&4` and for mixed indices `VehicleTable&0@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 `■`. 
4.2. Windows and components Section

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@0/1.

4.2 Windows and components Section

The topic 'components' is covered by several videos:

The video

'Component recognition'
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'
shows a simple case.

A more complex case is discussed in the video

'ComponentNotFoundException - complex case'

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.

Action

• **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.
This is going to take you directly to the 'TextField CalculatedPrice' node in the Windows and components section.
Let’s have a look at the properties of this component node used to identify the UI element.
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'.

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 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.

- **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.
4.2. Windows and components Section

- 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.
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.

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

- **Setting a Breakpoint**\(^ {(48)} \) e.g. via [Ctrl-F8].
- Pausing a test-run at any time and resuming operation using the debugger button \( \text{pause} \) or the (Don’t) Panic key [Alt-F12].
- Stepping Through a Test or Sequence\(^ {(49)} \) using the debugger buttons ‘Single step’ \( \text{step} \), ‘Step over’ \( \text{skip} \) and ‘Step out’ \( \text{jump} \).
- **Skipping Execution of Nodes**\(^ {(51)} \) using the debugger buttons ‘Skip over’ \( \text{skip} \) and ‘Skip out’ \( \text{jump} \).
- **Error or Exception triggering Debugging Mode**\(^ {(52)} \).
- **Resolving Errors directly from the Run-log**\(^ {(54)} \).
- Jump directly to the current error in the run-log via [Ctrl-J]. (Jump to run-log in chapter section 5.5\(^ {(54)} \).)

\[ \text{Note} \]

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 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:
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 will then be activated.

- Locating the current node using the debugger button (Locate the current node in chapter section 6.3(65)).
- "Continue execution from here" via the popup menu of the respective node (figure 6.9(67)).
- Rethrowing exceptions using the toolbar button .
- The variable bindings table (section 6.3(65)).

**5.1 Setting a Breakpoint**

Action
- Select a node and press Ctrl-F8. The breakpoint is indicated by a .

![Figure 5.1: Set break point](image)

Action
- Select the Test-suite node and press Enter to start the test run.

![Figure 5.2: Start test run](image)

Action
- Remove the breakpoint by pressing Ctrl-F8 again.
5.2. Stepping Through a Test or Sequence

Instead of using the keyboard shortcut [Ctrl-F8] you may also set or unset a breakpoint by clicking the node and selecting the [Debugger→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→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.

- Please try out the debugger buttons **Single step**, **Step over** and **Step out**.

You will find that **Single step** 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:
5.2. Stepping Through a Test or Sequence

In the case of leaf nodes (nodes without child nodes), the effect of Step over is the same as the following button's.

Step over 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 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

In the given example the node higher in the hierarchical structure where execution stops is the Cleanup node. As explained in the first chapter A full Test Run 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.

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.

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

Action • Run the Cleanup and Setup nodes by stepping over them, using the debugger button Step over and then Step in the second test-case via to get ready for the next section where you will learn about the skip functionality.

Note
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 after the node performing the selection and to activate normal test execution by releasing the pause button 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](image)

**Action**

- Now press the **Skip over** 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](image)
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.

**Action**

- Please open the debugger menu and change the default options as follows:
  - Activate the **Debugger → Enable debugger** menu item.
  - 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:

**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.
5.4. Error or Exception triggering Debugging Mode

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

- **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](image)

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.
5.5 Resolving Errors directly from the Run-log

The Jump to run-log button from the dialog in figure 5.12 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).

The error details tell you that the expected value does not match with the one shown in 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.
5.5. Resolving Errors directly from the Run-log

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.
5.6. Pause Execution

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 ⏸️.

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 → 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" ⏸️ and QF-Test will directly enter debugging mode.

In order to resume execution just release the pause button ⏸️. 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 shortcut [Alt-F12]. 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.

The video

Video

'Variables'

https://www.qfs.de/en/yt/variables.html

provides additional information about variables.

6.1 Procedure using a variable

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

- Select the 'Check text' node in the first test-case.
- Use Operations → Pack nodes → Sequence or use the Ctrl-Shift-S 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.
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:

- **Select the procedure 'checkFinalPrice'**
- **Press** the "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.**
6.1. Procedure using a variable

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.
6.2. The Variable Bindings table

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

Action

- Run the first test-case.

It should execute without an error.

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.
6.2. The Variable Bindings table

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.

**Note**

Verify QF-Test is configured to pause at errors as shown in Set debugger options to pause on error\(^{(53)}\).

Select the 'Test-case: Second' node.

Execute it by pressing \(\text{or [Enter]}\).

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.

**Action**

So click OK to close the error message.

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.
6.2. The Variable Bindings table

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**

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 Starting the Application\(^\text{\textregistered}\)). 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 → Complete parameters of references** from the popup menu.
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.**
- When QF-Test stops at the breakpoint **step into the procedure** via and **watch the table of the variable bindings** as you do so.

![Variable bindings stack showing incorrect value](image)

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.
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 pop-up 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.
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.

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.
6.4 Setting Variables

The easiest way to do so is by pressing the "Locate Current Node" button or select the 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. 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.
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.
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

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:
6.5. Variable binding levels

- 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 → 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)

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 qfs.qft it needs to be included in your test-suite’s root node. With a newly created test-suite the file qfs.qft is added automatically to the list of included files.

Action

• Select the 'Test-suite' root node of your test-suite.

• Verify the qfs.qft is available within the table for "Include files".

• Add 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.

- Locate and load the test-suite file \texttt{qfs.qft}, which is located in the \texttt{qftest-5.1.2/include} directory of your QF-Test installation.

![Figure 7.1: The Standard Library](image)

You can see there is one main package \texttt{qfs} that contains further specific packages. The \texttt{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.

Within nearly all of the procedures of this library, you’ll notice that the variable \texttt{$(client)$} 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 qfs.fx.table, qfs.swing.table, qfs.swt.table and qfs.web.table 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.
7.2. Selected Packages and Procedures

- **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 `qfs.fx.tree`, `qfs.swing.tree`, `qfs.swt.tree` and `qfs.web.tree`. 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 `qfs.fx.cleanup`, `qfs.swing.cleanup` and `qfs.swt.cleanup` 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 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.

7.2.10 The Run-log.Screenshots Package

The qfs.run-log.screenshots 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 qfs.shellutils package contains procedures to support most common shell commands.

Some important procedures within this package are:
7.2. Selected Packages and Procedures

- **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 or directory.
- **touch** Create a specified file.
- **removeDirectory** Remove a specified directory.

7.2.12 The Utils Package

The `qfs.utils` 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 `qfs.database` 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.
7.2. Selected Packages and Procedures

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 directly 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 *qfs.databinder* 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 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.

8.1 If - else

You already came across If and Else nodes in the Setup sequence in the chapter Starting the Application. Let’s have a closer look at the details of the nodes.

![Figure 8.1: Setup Sequence with if-else structures](image)

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.
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).
8.1. If - else

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.
- **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 `isSUTRunning` in the variable bindings table. The first time it will have the value `false` so that the condition `not $(isSUTRunning)` will become true and the SUT will be started. The second time it will be `true` 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 `${group:varname}`, e.g. `${qftest:windows}`.

![Figure 8.3: If node evaluates the variable](image-url)
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 🕳️. 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.
8.2. Loops

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 right-clicking it and selecting the submenu item **Pack nodes to Loop** from the popup menu.
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.

- **Select the Test-case node and press** [Ctrl-A]

- **Press the 'Select procedure' button** left to 'Procedure name'.
8.2. Loops

- 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 [Ctrl-A] was described in Manual creation of procedures\(^{33}\). If you would like to check with the screenshots please have a look there.

**Action**
- 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 $\{\text{rows}\}$ 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.
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.
- **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** $(\text{checkSucceeded})$ **in the 'Condition' attribute** of the 'If' node and click the **OK** button.

The variable $\text{checkSucceeded}$ will be set to either **true** or **false** by the Check node so that the reference to the variable $(\text{checkSucceeded})$ 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.

**Action**

• **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:

![Test-case Diagram]

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.**
8.2. Loops

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).

The Quickstart Wizard available via the menu Extras → 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\(^{186}\) 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 Creating the Setup Sequence\(^{110}\), Improved Web Component Recognition\(^{123}\) and Windows and components Section\(^{133}\) that contain web specific parts.

Within part IV\(^{268}\) more advanced QF-Test features are explained, applicable for all supported UI technologies.
Chapter 10

Working with a Sample Test-suite (Web)

The video

'Tutorial Web 10'

covers the current chapter.

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

10.1 Loading the Test-suite

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.

- Press the toolbar button to bring up the file open dialog
- Navigate to the subdirectory qftest-5.1.2/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:
10.1. Loading the Test-suite

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.
10.2 Starting the Browser

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

- **Expand** the **Setup: Open Demo Page** node now.

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 (Internet Explorer on Windows, Chrome on 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**:
10.2. Starting the Browser

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.

4. **Install CustomWebResolver** - allow installation of individual resolvers for special component mapping.

These four steps are automatically generated when you use the Quickstart Wizard, that will be explained in the next tutorial chapter 11 \(^{(10)}\).

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

- **Click** on the **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 "\(\rightarrow\)".

When the setup sequence is completed, the browser with the "CarConfigurator" demo page is going appear on the screen.
10.3  First Test-case

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

Reset - performs a reset by use of the File->Reset menu action.
10.3. First Test-case

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.

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 📚.

Action

• Please select the test-case "First" node

• Click the replay button ▶️.

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%.

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](image)

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](image)

Let’s execute the second test-case:

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

- Click the replay button ▶.

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

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:
10.5. The Run-log for Error Diagnosis

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:
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:
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 Edit→Options→Run-logs→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 most comprehensive search can be achieved via Help→Online search...]. This navigates you to the search functionality on our homepage and allows querying throughout all available
10.7 Stopping the Application

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.

Besides 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 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](image)

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.

- Execute the cleanup to see the Browser with the CarConfigurator demo vanish.
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.

Action

- 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.

Action

- Open up the run-log again by ➡ to take a closer look:

![Run-log (detailed)](image)

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 clean-up that will be explained with the advanced features in this tutorial (chapter 29[279]).

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.

Action

- 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.

![Report Generation Properties](image)

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.

**Action**

- 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:
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)

The video

‘Tutorial Web 11’

covers the current chapter.

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.

11.1 Creating the Setup Sequence

To begin, you need to launch the application from qtest. 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.
11.1. Creating the Setup Sequence

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

**Action**

- Please select the second option **A web application in a browser**.
- Press **Next**.
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 toolbar button to bring up the file open dialog
- Navigate to the directory `./qftest-5.1.2/demo/carconfigWeb/html` from your QF-Test installation
- There select the file `CarConfig_en.htm`.
- Conclude the file selection.

**Note**
In the figure below we used the QF-Test variable `${qftest:dir.version}` to ad-
11.1. Creating the Setup Sequence

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 in the manual chapter Variables).

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.

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

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

![Quickstart Wizard](image)

**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**).
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.
11.1. Creating the Setup Sequence

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

- Press the toolbar button or use the menu option with its short-cut \( \text{Action} \) \{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:

- Please press the **Record button**.

- **Switch to the WebCarConfigurator window**. From now on every mouse and keyboard action performed within the SUT window will be recorded.

- Click to the table cell **I5** in the last row.

- Change to the **Specials** tab of the tabbed pane.

- Choose the special **Jazz** from the combo box.
11.2. Recording Actions

- Finally switch back to the **Vehicles** tab in the tabbed pane.

- **Stop the recording** by use of the button.

You’ll find the recorded sequence placed in the ”**Extras**” section:

![Figure 11.8: The Recorded Sequence](image)

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 11.9: The Renamed Sequence](image)

Now, let us replay the recorded sequence.

**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.
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.

- 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](image)

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.
11.4. Setting up a Test-suite

- "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), context-menu (right mouse-button copy/paste) or by Ctrl-X and Ctrl-V keyboard commands.

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.
11.5. Stopping the Demo

- **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.

**Action**
- Please open the **context menu** for the **first** of the two sequence nodes by a right-click.
- 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](image)

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 **File→Exit**. All these options can be directly recorded and then used in the clean-up sequence.

Let’s use the first one.

**Action**
- **Start recoding**.
- Click the **window closing button** of the browser.
11.6. Running the whole Test-suite

- 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.

You should end up with the following:

![Figure 11.13: The simple cleanup sequence](image)

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 \( \text{Return} \).

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 menu option with its short-cut \( \text{Ctrl-L} \).
11.7 Improved Web Component Recognition

When closely comparing our own recordings with the events of the sample test-suite from the first chapter, we will notice slight differences in the recorded components. E.g. when recording a click to the "I5" vehicle in the table we now may get a recording like Mouse click [VehicleTable.td] whereas in the demo suite it was Mouse click [VehicleTable@Model&5]. When replaying them, both trigger the same action and click the cell in the sixth line of the Model column. So, why are the recorded components so different and why is this important?

Web applications have a limited set of basic components, which can be combined in flexible ways in order to implement complex components and designs. E.g. HTML tables 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 QF-Test records the component VehicleTable.td in the newly created test-suite. The "td" is derived from the HTML table data element "TD" and indicates a table cell, but nothing more. QF-Test is not able to assign a logical table row and column.

The demo suite FirstWebTests.qft of the first chapter installs so-called "Resolvers" during the setup that generates this additional information. They tell QF-Test e.g. where the logical table starts and which special properties it has. Then QF-Test is able to determine the logical row and column for the cell as visible in VehicleTable@Model&5. The resolvers have been implemented specifically for the WebCarConfigurator.

The following figure shows the resolver registration node in the FirstWebTests.qft with some configuration parameters.
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, 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.
Chapter 12

Writing a Procedure (Web)

The video

'Tutorial Web 12'

covers the current chapter.

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-5.1.2/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[110].
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.
- **Choose** 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.
- **Open** the newly created 'reset' procedure node.
12.2. Manual creation of procedures

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 12.2: Create a procedure node

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→Procedure nodes→Procedure call] or use the [Ctrl-A] shortcut.

![Procedure call dialog](image)

**Figure 12.4: Insert a procedure call**

**Action**

- 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.
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:
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:

- **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 [Ctrl-Shift-P] 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.

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.

- **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 7/6/20 8:00 PM](Mouse click:VehicleTable@Model&0=5[client0])

Figure 13.1: Addressing a table cell

The QF-Test component ID recorded is VehicleTable@Model&0. The single parts are:
13.1. Addressing subitems of tables, lists and trees

- **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&0&4` and for mixed indices `VehicleTable&0@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&/0/1.

### 13.2 Windows and components Section

The topic 'components' is covered by several videos:

The video

- 'Component recognition'

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'

shows a simple case.

A more complex case is discussed in the video

- 'ComponentNotFoundException - complex case'

The video

- 'Dealing with the explosion of complexity in web test automation'

gives you a good idea of how QF-Test handles a deeply nested DOM structure.
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 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 Improved Web Component Recognition 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 AJAX toolkits 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.

**Action**

- **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.
This is going to take you directly to the 'TextField CalculatedPrice' node in the Windows and components section.
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 Improved Web Component Recognition \cite{123}. When you have a look at the component tree recorded for the test-suite you set up in chapter 11\cite{110} 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 custom web resolver.

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

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 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.

- **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.**

- **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 [Ctrl-Z].**

- **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.
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.

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

- **Setting a Breakpoint**\(^{(141)}\) e.g. via \([\text{Ctrl-F8}]\).
- **Pausing a test-run at any time and resuming operation using the debugger button \(\text{Pause} \) or the (Don’t) Panic key \([\text{Alt-F12}]\).**
- **Stepping Through a Test or Sequence**\(^{(142)}\) using the debugger buttons ‘Single step’ \(\text{Step over} \) and ‘Step out’ \(\text{Step out} \).
- **Skipping Execution of Nodes**\(^{(144)}\) using the debugger buttons ‘Skip over’ \(\text{Skip over} \) and ‘Skip out’ \(\text{Skip out} \).
- **Error or Exception triggering Debugging Mode**\(^{(145)}\).
- **Resolving Errors directly from the Run-log**\(^{(147)}\).
- **Jump directly to the current error in the run-log via \([\text{Ctrl-J}]\).** (Jump to run-log in chapter section 14.5\(^{(147)}\)).

**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:
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 will then be activated.

Action

- Select a node and press \texttt{Ctrl-F8}. The breakpoint is indicated by a 

Action

- Select the \texttt{Test-suite} node and press \texttt{Enter} to start the test run.

Action

- Remove the breakpoint by pressing \texttt{Ctrl-F8} again.
14.2 Stepping Through a Test or Sequence

Instead of using the keyboard shortcut Ctrl-F8 you may also set or unset a breakpoint by clicking the node and selecting the Debugger → 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 → 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.

- Please try out the debugger buttons Single step , Step over and Step out.

You will find that Single step 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
14.2. Stepping Through a Test or Sequence

In the case of leaf nodes (nodes without child nodes), the effect of \[\text{Step over}\] is the same as the following button’s.

**Step over** \[\text{Step over}\] 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](image)

**Step out** \[\text{Step out}\] 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](image)

In the given example the node higher in the hierarchical structure where execution stops is the **Cleanup** node. As explained in the chapter **A full Test Run** 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.

**Action**
- Run the **Cleanup** and **Setup** nodes by stepping over them, using the debugger button **Step over** and then **Step in** the second test-case via \[\text{Step in}\] 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 loses the focus, as will happen when activating the debugging mode. In such a case you
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:

Action

- Now press the Skip over 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 14.7: Pause execution at first node of the second test-case

Figure 14.8: Skip over a node
14.4. Error or Exception triggering Debugging Mode

At last, press the **Skip out** 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](image)

**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.

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.

**Action**
- Please open the debugger menu and change the default options as follows:
- **Activate** the **Debugger → Enable debugger** menu item.
- **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:
14.4. Error or Exception triggering Debugging Mode

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

- **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 14.10: Set debugger options to pause on error**

- **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.
14.5 Resolving Errors directly from the Run-log

The Jump to run-log button from the dialog in figure 14.12\(^{(147)}\) 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\(^{(154)}\)).

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

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 in.

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 → 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" in and QF-Test will directly enter debugging mode.

In order to resume execution just release the pause button in . 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 shortcut [Alt-F12]. 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.

The video

‘Variables’
https://www.qfs.de/en/yt/variables.html
provides additional information about variables. The video uses a Java Applikation 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.
15.1. Procedure using a variable

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.

Action

- Select the 'Check text' node in the first test-case.
- Use \texttt{Operations} \rightarrow \texttt{Pack nodes} \rightarrow \texttt{Sequence} or use the \texttt{Ctrl-Shift-S} shortcut to pack it into a sequence.
- Name the sequence \texttt{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 \texttt{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 15.1: Two almost identical test-steps
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:

- **Select the procedure 'checkFinalPrice'**
- **Press the "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.**
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.
15.2 The Variable Bindings table

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

**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.

It should execute without an error.
15.2. The Variable Bindings table

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.

**Note**

Verify QF-Test is configured to pause at errors as shown in Set debugger options to pause on error. (146)

- 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.

**Action**

- So click OK to close the error message.

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.
15.2. The Variable Bindings table

15.2.1 The Variable Bindings table

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 Starting the Browser[97]). 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 → Complete parameters of references from the popup menu.
15.2. The Variable Bindings table

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.
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.

- **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 \(\text{\includegraphics[width=20px]{qf-test-badge.png}}\) and watch the table of the variable bindings as you do so.**

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.
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 pop-up 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.
When checking the variable bindings table you will find that the top two rows have dis-appeared. 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 \[\text{Pause}\].

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" button or select the 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. 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.
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.
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

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:
15.5. Variable binding levels

- 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→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.
15.5. 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 16

The Standard Library (Web)

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 qfs.qft it needs to be included in your test-suite’s root node. With a newly created test-suite the file qfs.qft is added automatically to the list of included files.

Action

• Select the 'Test-suite' root node of your test-suite.
• Verify the qfs.qft is available within the table for "Include files".
• Add 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...
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.

- Locate and load the test-suite file qfs.qft, which is located in the qftest-5.1.2/include directory of your QF-Test installation.

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.

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.

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 `qfs.run-log.screenshots` package contains procedures, which write images into the run-log and some helper methods.

Some important procedures within this package are:
16.2. Selected Packages and Procedures

- **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 `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 or directory.
- **touch** Create a specified file.
- **removeDirectory** Remove a specified directory.

16.2.8 The Utils Package

The `qfs.utils` 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.)
16.2. Selected Packages and Procedures

- **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 `qfs.database` 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](http://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 directly 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 qfs.databinder 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.

17.1 If - else

You already came across If nodes in the Setup sequence in the chapter Starting the Browser. Let’s have a closer look at the details of the node.

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.
17.1 If - else

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).
17.1. If - else

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.

**Action**

- **Stop the client** in case it is running.
- **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 *isSUTRunning* in the variable bindings table. The first time it will have the value false so that the condition not $(isSUTRunning)$ will become true and the SUT will be started. The second time it will be true 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 Elsief and Else to implement that functionality. The child nodes of an Elsief node would be executed if the If condition were false and the Elsief condition were true. The child nodes of an an Else node would be executed if the If condition and all Elsief conditions were false.
17.2 Loops

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:

- **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.

![Figure 17.4: Transform a node into another one](image)

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 right-clicking it and selecting the submenu item **Pack nodes → Loop** from the popup menu.
17.2. Loops

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.

- **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 $\text{Ctrl-A}$ was described in Manual creation of procedures\textsuperscript{(126)}. If you would like to check with the screenshots please have a look there.

\textbf{Action}

• Enter the variable name \texttt{rows} in the Variable for return value attribute.

• Change the default value for the \texttt{id} in the variable definitions table to the QF-Test component ID of the table, i.e. \texttt{VehicleTable}.

• Click the OK button.

• Select the Loop node.

• Enter a reference to the variable $\$(\texttt{rows})$ in the Number of iterations attribute of the Loop node.

• Enter the name of an iteration counter, e.g. \texttt{i} in the respective attribute of the Loop node.

• Click the OK button.
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.
- **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 $\text{(checkSucceeded)}$ 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 $\text{(checkSucceeded)}$ 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.

Action

• 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 → Control structures → 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:

![Test-case]

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.*
17.2. Loops

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).

The Quickstart Wizard available via the menu [Extras→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 or part II 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 IV 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.

19.1 Loading the Test-suite

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.

- Press the toolbar button to bring up the file open dialog
- Navigate to the subdirectory qftest-5.1.2/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:
19.1. Loading the Test-suite

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.

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**: 
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 `${qftest:dir.version}` (see manual chapter Variables).

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

- **Click** on the **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.
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 I5** - chooses the last model I5 within the vehicles table.
**Select Special Model Jazz** - switches to the Specials tab and selects the Jazz option.
**Check Final Price** - checks that the calculated final price field located at bottom right equals a given value.
19.3. 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 19.7: Details of the first Test-case](image)

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.

- **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](image)

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](image1)

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 19.10: Details of the second Test-case](image2)

Let’s execute the second test-case:

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

This time a dialog shows up telling us that an error occurred.
19.5. The Run-log for Error Diagnosis

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).

19.5 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:
19.5. The Run-log for Error Diagnosis

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:
19.5. The Run-log for Error Diagnosis

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 theScreenshot 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 19.13: Error in the second test-case
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 `Edit → Options → Run-logs → 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 most comprehensive search can be achieved via `Help → Online search...`. This navigates you to the search functionality on our homepage and allows querying throughout all available
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](image)

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 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 clean-up that will be explained with the advanced features in this tutorial (chapter 29).

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.

**Action**

- 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.

![Figure 19.17: Report Generation Properties](image)

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.

**Action**

- 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:
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.

20.1 Starting the Application

To begin, you need to launch the application from qtest. 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...

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.
20.1. Starting the Application

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

**Action**

- Please select the fourth option **A native windows application**.
- Press **Next**.
20.1. Starting the Application

The next step specifies the Windows application executable.

- For this please use the select button on the right.
- Navigate to the subdirectory qftest-5.1.2/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.
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.

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.

- Please press **Finish** in order to end the wizard.
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.
20.1. Starting the Application

Now we want to see some action:

- Please ensure the Setup: Launch CarConfigForms node is selected.
- Then click 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.
At the end of this section let’s save our test-suite.

- Press the toolbar button or use the 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:

![CarConfigurator Forms](image)

Figure 20.7: Recording actions on the CarConfiguratorForms Demo

- 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 I5 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 button.

**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:

![Figure 20.8: The Recorded Sequence](image)

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](image)

Now, let us replay the recorded sequence.
20.3. Recording Checks

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.

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.
- 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 recorded properly.

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

Action

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

![Image](image.png)

Figure 20.10: The recorded check node
20.4 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.

Action

- 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), context-menu (right mouse-button copy/paste) or by Ctrl-X and Ctrl-V keyboard commands.
20.4. Setting up a Test-suite

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

**Action**

- 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.

**Action**

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

Your test-suite should now look like this:

![Figure 20.12: The Organization of your Test-suite](image)
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 `File→Exit`. All these options can be directly recorded and then used in the clean-up sequence.

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

- **Start recording**.
- 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.

You should end up with the following:

![Figure 20.13: The simple cleanup sequence](image)

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:
20.6. Running the whole Test-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 menu option with its short-cut (Ctrl-L)

![Image of Run-Log](image)

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)

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 \[\text{Packages}\]. 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.

**Action**

- Copy `FirstWinTests.qft` from the subdirectory `qftest-5.1.2/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.

Please have a look at the test-step "Reset" in the two test-cases. They are exactly the same.
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.
- **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.
- **Open** the newly created ‘reset’ procedure node.
21.2. Manual creation of procedures

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 21.2: Create a procedure node*

*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→Procedure nodes→Procedure call] or use the [Ctrl-A] shortcut.

![Procedure call dialog](image)

**Figure 21.4: Insert a procedure call**

**Action**

- 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.
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:
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:

1. **Select the test-step or sequence node** that contains the reusable steps to be transformed into a procedure.

2. **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 [Ctrl-Shift-P] 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.

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.

- **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.

![Figure 22.1: Addressing a table cell](image)

The QF-Test component ID recorded is VehicleTable@Model&0. The single parts are:
22.1. Addressing subitems of tables, lists and trees

- **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 VehicleTable&1&2 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&0&4 and for mixed indices VehicleTable&0@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'.
- 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.

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

- **Setting a Breakpoint** 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 using the debugger buttons 'Single step', 'Step over', and 'Step out'.
- Skipping Execution of Nodes using the debugger buttons 'Skip over' and 'Skip out'.
- Error or Exception triggering Debugging Mode.
- Resolving Errors directly from the Run-log.
- Jump directly to the current error in the run-log via (Ctrl-J).

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:
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 will then be activated.

Action  
- Select a node and press Ctrl-F8. The breakpoint is indicated by a .

![Figure 23.1: Set break point](image)

Action  
- Select the Test-suite node and press Enter to start the test run.

![Figure 23.2: Start test run](image)

Action  
- Remove the breakpoint by pressing Ctrl-F8 again.
23.2 Stepping Through a Test or Sequence

Instead of using the keyboard shortcut \(\text{Ctrl-F8}\) you may also set or unset a breakpoint by clicking the node and selecting the \(\text{Debugger} \rightarrow \text{Breakpoint on/off}\) menu item, or alternatively right-clicking the node and selecting the \(\text{Breakpoint on/off}\) 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 \(\text{Debugger} \rightarrow \text{Clear all breakpoints}\) 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.

- Please try out the debugger buttons **Single step**, **Step over** and **Step out**.

You will find that **Single step** 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:
23.2. Stepping Through a Test or Sequence

In the case of leaf nodes (nodes without child nodes), the effect of \( \text{Step over} \) is the same as the following button's.

**Step over** \( \text{Step over} \) 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](image)

**Step out** \( \text{Step out} \) 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](image)

In the given example the node higher in the hierarchical structure where execution stops is the **Cleanup** node. As explained in the chapter A full Test Run\(^{(199)}\) 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.

*Action*  
- Run the **Cleanup** and **Setup** nodes by stepping over them, using the debugger button **Step over** \( \text{Step over} \) and then **Step in** in the second test-case via \( \text{Step in} \) 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
23.3 Skipping Execution of Nodes

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 after the node performing the selection and to activate normal test execution by releasing the pause button 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](image)

- Now press the **Skip over** 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](image)
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.

**Action**

- Please **open the debugger menu** and change the default options as follows:
  - **Activate** the [Debugger → Enable debugger] menu item.
  - **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:
23.4. Error or Exception triggering Debugging Mode

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

- **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:

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.
23.5 Resolving Errors directly from the Run-log

The Jump to run-log button from the dialog in figure 23.12 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).

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

- **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.
23.5. Resolving Errors directly from the Run-log

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.
23.6. Pause Execution

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 [pause].

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 → 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" [pause] and QF-Test will directly enter debugging mode.

In order to resume execution just release the pause button [pause] . 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 shortcut [Alt-F12]. 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.

The video

‘Variables’
https://www.qfs.de/en/yt/variables.html
provides additional information about variables. The video uses a Java Applikation for demonstration. Variables are used in the same way with native Windows applications.

24.1 Procedure using a variable

Have a look at the last test-step ‘Check final price’ of our two test-cases.
24.1. Procedure using a variable

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 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 \[\text{Operations} \rightarrow \text{Pack nodes} \rightarrow \text{Sequence}\] or use the [Ctrl-Shift-S] 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.
24.1. Procedure using a variable

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:

- **Select the procedure 'checkFinalPrice'**
- **Press** the “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.

![Figure 24.2: Procedure with hard coded value](image)
24.1. Procedure using a variable

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

**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.
24.2. The Variable Bindings table

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

- 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.
24.2. The Variable Bindings table

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.

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.

Action

- Verify QF-Test is configured to pause at errors as shown in Set debugger options to pause on error.

- 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.

Action

- So click OK to close the error message.

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.
24.2. The Variable Bindings table

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

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 Starting the Application). 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 → Complete parameters of references** from the popup menu.
24.2. The Variable Bindings table

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 \texttt{price} with the value \texttt{$30,049.00$}.

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.
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.**
- When QF-Test stops at the breakpoint **step into the procedure** via and **watch the table of the variable bindings** as you do so.

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.
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 bindings 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.

- 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 pop-up 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:

- Right-click the 'Call procedure: checkFinalPrice' node of the second procedure.

- Select 'Continue execution from here’ in the popup menu.
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.

- Release the pause button \[ \text{Pause} \].

The test run should finish without error.

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.
24.4 Setting Variables

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

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. 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.
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.
24.5 Variable binding levels

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

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:
24.5. Variable binding levels

- 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→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.
24.5. 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 25

The Standard Library (Win)

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 qfs.qft it needs to be included in your test-suite’s root node. With a newly created test-suite the file qfs.qft is added automatically to the list of included files.

Action

- Select the 'Test-suite’ root node of your test-suite.
- Verify the qfs.qft is available within the table for "Include files".
- Add 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.

- Locate and load the test-suite file qfs.qft, which is located in the qftest-5.1.2/include directory of your QF-Test installation.

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 \$\text{(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.

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 qfs.run-log.screenshots 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 qfs.shellutils package contains procedures to support most common shell-commands.

Some important procedures within this package are:
25.2. Selected Packages and Procedures

- **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 *qfs.utils* 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 *qfs.database* 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 directly 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 `qfs.databinder` 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 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.

26.1 If - else

You already came across If nodes in the Setup sequence in the chapter Starting the Application. Let’s have a closer look at the details of the node.

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.
26.1. If - else

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).
26.1. If - else

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.
- **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 `isSUTRunning` in the variable bindings table. The first time it will have the value `false` so that the condition `not $(isSUTRunning)` will become true and the SUT will be started. The second time it will be `true` 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.
26.2 Loops

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 \[\text{pause}\]. In batch mode you would have to kill the QF-Test process. (You start QF-Test in batch mode using the command line parameter \text{-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 \[\text{✓}\].
- **Right-click a row** in the CarConfig application and select the menu item \[\text{Row}\] from the popup menu.
- **Stop the recording** by pressing \[\text{■}\].
• 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.

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 right-clicking it and selecting the submenu item Pack nodes → Loop from the popup menu.
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’.
26.2. Loops

- 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 Ctrl-A was described in Manual creation of procedures. If you would like to check with the screenshots please have a look there.

**Action**

- 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.
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.

- **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 \(\rightarrow\) Control structures \(\rightarrow\) Break` from the popup menu.
- **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).
26.2. Loops

- Type \$(\text{checkSucceeded})\ in the 'Condition' attribute of the 'If' node and click the OK button.

The variable \text{checkSucceeded} will be set to either true or false by the Check node so that the reference to the variable \$(\text{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 → Control structures → 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.

- 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:

![Test-case diagram]

Figure 26.7: The new test-case

**Action**

- Execute the new test-case.

It should run without error.

**Action**

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

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).

The Quickstart Wizard available via the menu [Extras → 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

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-5.1.2/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-5.1.2/doc/tutorial/advanced-demos/en/datadrivenTesting.qft.
The second provided test-suite qftest-5.1.2/doc/tutorial/datadrivers.qft 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 JCarConfigurator 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.
28.2. Traditional way of implementing data driven testing

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:

![Figure 28.1: Traditional way of data driven testing](image)

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 pre-conditions. 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](image)
28.3. Data driver concept

The chapter Dependencies: Automatically ensuring correct prerequisites for each Test-case\(^{(279)}\) 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 Data driver concept\(^{(271)}\) 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 \([\text{Insert node} \rightarrow \text{Data drivers} \rightarrow \text{Data driver}]\). 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 \([\text{Insert node} \rightarrow \text{Data drivers} \rightarrow \text{Data table}]\). Now we should detect this 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:

![Data table dialog](image-url)
The next part is the implementation of the test-case. Therefore we just insert one Test-case node to the Test-set.

*Note* 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:

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:

![Diagram of procedure call]

Figure 28.6: Using the \( \text{\$discount} \) 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.
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)'.

Figure 28.7: Full data table
If we re-run 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:
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.

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 call. This can be done because the variable of the Data driver will be put at QF-Test's variable stack and so any step within the Test-case can access that variable. You can see this behavior in the demo test-suite qftest-5.1.2/doc/tutorial/advanced-demos/en/datadrivenTesting.qft in the Test-set "data driven tests with optimized procedure calls".

In the demo test-suite qftest-5.1.2/doc/tutorial/advanced-demos/en/datadrivenTesting.qft you 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 `qftest-5.1.2/doc/tutorial/datadrivers.qft` 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

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-5.1.2/doc/tutorial/advanced-demos/en/dependencies.qft. There is also a second test-suite qftest-5.1.2/doc/tutorial/advanced-demos/en/dependencies_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.

29.1 General

Please copy the test-suite qftest-5.1.2/doc/tutorial/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.
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 JCarConfigurator, 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-5.1.2/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](image)

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→Dependencies→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](image)
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.

![Run-log of the execution](image)

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.
29.2. Ensuring prerequisites

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 → Dependencies → 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 'Discount 10' stops the SUT. But the third Test-
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.

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

![Figure 29.7: Ensuring prerequisites for Test-case 'Discount 15']
29.3. Combining dependencies

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 JCarConfigurator now.

In the JCarConfigurator 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'.


29.3. Combining dependencies

- 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:

![Dependency: vehicles dialog opened](image)

Figure 29.8: 'vehicles dialog opened' Dependency
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 qftest-5.1.2/doc/tutorial/advanced-demos/en/dependencies.qft.

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](image)

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 runlog, 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. However, both Dependencies require the 'SUT started' Dependency, that is why that Cleanup of the 'SUT started' Dependency has not been executed.
29.4 Error and exception handling

29.4.1 Error handling

Please copy the test-suite qftest-5.1.2/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. You can find a Test-set 'Tests with error'. The second Test-case of that Test-set fails.

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 JCarConfigurator 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.
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 \textit{Insert node} → \textit{Dependencies} → \textit{Error handler}.

In the Error handler you can define the steps for stopping the SUT again. The Dependency \texttt{SUT started} should look like this now:

Please run the whole Test-set \texttt{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.
29.4. Error and exception handling

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.

Figure 29.13: Run-log for Dependency with Error handler
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 → Control structures → 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:
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.

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 → Enable debugger].

After the test-run you can open the run-log and verify what happened.
In a normal project you should at least try to catch a 'TestException' and the Catch should contain a procedure call of either  
qfs.swing.cleanup.closeAllModalDialogs, qfs.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 `qftest-5.1.2/doc/tutorial/advanced-demos/en/automated_procedures.qft`. There is also a second test-suite `qftest-5.1.2/doc/tutorial/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 JCarConfigurator, we have to record steps on each element that will be touched by a test. The JCarConfigurator 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-5.1.2/doc/tutorial/advanced-demos/en/automated_procedures.qft.
30.2 Automated creation of procedures

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 `gtest-5.1.2/doc/tutorial/advanced-demos/en/automated_procedures_work.qft` to a project-related folder and open it there.
30.2. Automated creation of procedures

This test-suite contains a Test-set referring to a 'Start SUT' Dependency.

![Test-suite](image.png)

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 → 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-5.1.2/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 contains several packages and procedures for actions on the whole dialog and its single UI components.

Note

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.
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:

<table>
<thead>
<tr>
<th>Package</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>JCarConfigurator</td>
<td>This package contains all procedures for actions on components of the JCar-Configurator window. It has been created because the configuration file uses the component hierarchy for creating the package structure.</td>
</tr>
</tbody>
</table>

Table 30.1

The Package **JCarConfigurator** contains several packages which are:
### Table 30.2

<table>
<thead>
<tr>
<th>Package</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>MenuBar</td>
<td>This package contains procedures for clicking at the items of the menu.</td>
</tr>
<tr>
<td>VehicleTablePanel</td>
<td>This package contains all procedures for actions on components on the 'VehicleTable' panel of the SUT. In our case there are only procedures for the 'VehicleTable' object, because that is the only component on that panel.</td>
</tr>
<tr>
<td>check</td>
<td>This package contains procedures for checking components.</td>
</tr>
<tr>
<td>check-window</td>
<td>This package contains procedures which can be called to check the components of one frame using one call. Those procedures are container procedures.</td>
</tr>
<tr>
<td>get</td>
<td>This package contains procedures for getting the current values of components, e.g. reading the text of a text-field.</td>
</tr>
<tr>
<td>select</td>
<td>This package contains procedures for selecting items of elements - in our case only one procedure to select a tab in the TabbedPane.</td>
</tr>
<tr>
<td>set</td>
<td>This package contains procedures for setting several components of the current window - in our case several setters for the text fields of the frame.</td>
</tr>
<tr>
<td>set-window</td>
<td>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.</td>
</tr>
<tr>
<td>wait</td>
<td>This package contains all procedure waiting for several components.</td>
</tr>
</tbody>
</table>

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:
30.2. Automated creation of procedures

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.
30.3. Configuration of the automated creation

30.3.1 Introduction

In the previous example we used the file `qftest-5.1.2/demo/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.

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

![Figure 30.6: The current configuration](image-url)
30.3. Configuration of the automated creation

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:

- 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](image)

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 JCarConfigurator 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
30.3. Configuration of the automated creation

[Insert node→Check nodes→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:

![Image of test-suite](image)

Figure 30.8: The checkText procedure

The procedure contains an empty default value for the parameter ‘text’.
Now we have nearly completed our first procedure template, but we have to think about 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 placeholder <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> placeholder directly in the QF-Test component ID attribute of the 'Check text' node.

Finally the template procedure should look like this:
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 `Edit → 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 JCarConfigurator 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.

Congratulations, you have instructed QF-Test to record test steps for you.
30.3. Configuration of the automated creation

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 JCarConfigurator 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:

![Figure 30.11: Your first automatically created procedures](image)
Figure 30.12: The configuration for the current text

The newly created procedures displays like below:
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 JCarConfigurator. 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>,

Figure 30.13: The created procedures with the current text
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](image)

The last step is to make QF-Test to create a ‘container’ procedure and not a normal ‘component’ procedure. This can be achieved by setting `@FORCHILDREN` in the comment attribute of the `checkTextOfElements_<COMPID>` procedure.
30.3. Configuration of the automated creation

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:
30.3. Configuration of the automated creation

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 Using the current text for checking, where we want to record the current values of each text-field during creation. In the example Using the current text for checking 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 <CURRENTVALUE>. 

Figure 30.16: The created container procedures
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.
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 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 \texttt{<COMPID>\texttt{ <CURRENTVALUE>}}. Then you have to change the value of the text parameter of that procedure-call to \texttt{$(<COMPID>)\texttt{ <CURRENTVALUE>}}. The configuration should look like this now:
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.
30.3. Configuration of the automated creation

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.