

QF-Test - The Manual

Version 4.0.8

Gregor Schmid

Quality First Software GmbH¹

Copyright © 1999-2015 Quality First Software GmbH²

November 4, 2015

¹<http://www.qfs.de>

²<http://www.qfs.de>

Preface

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

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

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

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

How to use this manual

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

The PDF version of the manual is located at

qftest-4.0.8/doc/manual_en.pdf, the entry page of the HTML version is at `qftest-4.0.8/doc/manual/en/manual.html`.

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

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

User manual⁽²⁾

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

Reference manual⁽²³⁷⁾

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

Technical reference⁽⁵⁶⁷⁾

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

A learning-by-doing tutorial is also available in HTML and PDF versions. The HTML version, which is directly accessible from the **Help→Tutorial (HTML)...** menu item, is located at `qftest-4.0.8/doc/tutorial/en/tutorial.html`. The PDF version is to be found at `qftest-4.0.8/doc/tutorial/tutorial_en.pdf` and is also directly accessible through **Help→Tutorial (PDF)...**.

The following notations are used throughout the manual:

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

- `[Modifier-Key]` stands for a keystroke, where the modifier is one (or a combination) of `[Shift]`, `[Control]` or `[Alt]`.
- `Monospaced font` is used for names of directories and files, user input and program output.
- In order to transfer at least part of the convenience of cross-linking to the paper version, references⁽ⁱ⁾ in the PDF version are underlined and show the target page number in small braces.

Contents

I	User manual	1
1	Installation and startup	2
1.1	System requirements	2
1.1.1	Prime precondition	2
1.1.2	Supported technology versions	3
1.2	Linux/Unix Installation	4
1.3	Windows Installation	5
1.4	The license file	6
1.5	The configuration files	7
1.6	Starting QF-Test	7
1.7	Firewall Security Warning	8
2	The user interface	9
2.1	The test-suite	9
2.2	Basic editing	11
2.2.1	Navigating the tree	11
2.2.2	Insertion marker	12
2.2.3	Moving nodes	12
2.2.4	Transforming nodes	13
2.2.5	Tables	13
2.2.6	Packing and Unpacking	15
2.2.7	Sorting Nodes	15
2.3	Advanced editing	15
2.3.1	Searching	16

2.3.2	Replacing	20
2.3.3	Complex searches and replace operations	22
2.3.4	Multiple views	23
3	Quickstart your application	24
3.1	Quickstart Wizard	25
3.1.1	Setup sequence creation	25
3.1.2	Executing the setup sequence	27
3.1.3	Troubleshooting	27
3.2	Program output and the Clients menu	29
3.3	An application started indirectly from an already connected SUT	29
4	Capture and replay	30
4.1	Recording sequences	30
4.2	Running tests	31
4.3	Recording checks	33
4.4	Recording components	34
4.5	Recording of HTTP Requests (GET/POST)	35
5	Components	37
5.1	GUI component hierarchy	37
5.2	Components in QF-Test	39
5.3	QF-Test IDs	40
5.4	Component class	41
5.4.1	Generic classes	42
5.4.2	Class hierarchy for web applications	42
5.4.3	Settings for class recording	43
5.5	Component inspection	44
5.6	The importance of naming components	44
5.7	Considerations for setting names	46
5.8	Updating Components	47
5.9	Troubleshooting component recognition problems	49
5.10	Accessing hidden fields on a web page	51

6	Sub-items of complex GUI components	52
6.1	Item nodes	52
6.2	The QF-Test ID of an Item	54
6.3	Special QF-Test ID-syntax for direct element access	54
6.4	Multi-level sub-items and QPath	55
6.5	Activating item recording for web tables	56
7	Variables	57
7.1	Variable lookup	57
7.2	Defining variables	58
7.3	Variable example	60
7.4	Fetching data from the GUI	61
7.5	External data and special groups	62
7.6	Expressions	66
7.7	Immediate and lazy binding	66
8	Problem analysis and debugging	68
8.1	The run-log	68
8.1.1	Error states	69
8.1.2	Navigating the run-log tree	71
8.1.3	Accepting values of failed checks as good	71
8.1.4	Split run-logs	72
8.1.5	Run-log options	73
8.1.6	Creating a test-suite from the run-log	73
8.2	The debugger	74
8.2.1	Entering the debugger	74
8.2.2	The debugger window	75
8.2.3	Debugger commands	76
8.2.4	Manipulating breakpoints	77
9	Organizing the test-suite	78
9.1	Sequence and Test nodes	78
9.2	Test management with Test-set and Test-case nodes	80

9.2.1	Concepts	80
9.2.2	Variables and special attributes	81
9.2.3	Migration and backwards compatibility	81
9.3	Procedures and Packages	82
9.3.1	Local Procedures and Packages	83
9.3.2	Relative Procedures	83
9.3.3	Inserting Procedure call nodes	84
9.3.4	Parameterizing nodes	84
9.3.5	Transforming a Sequence into a Procedure	84
9.4	Documenting test-suites	85
10	Projects	86
11	The standard library	88
12	Dependencies	90
12.1	Dependency basics	90
12.2	The dependency stack	91
12.3	Error escalation	95
12.4	Special variables	96
12.5	Error handling	97
13	Scripting	99
13.1	Fundamentals of the Jython integration	99
13.2	The run-context <code>rc</code>	102
13.3	Scripting examples	102
13.3.1	Logging messages	102
13.3.2	Performing checks	103
13.3.3	Accessing variables	103
13.3.4	Setting variables	105
13.3.5	Global Jython variables	105
13.3.6	Exchanging Jython variables between interpreters	106
13.3.7	Accessing the SUT's GUI components	106

13.3.8	Calling Procedures	107
13.3.9	Setting options	108
13.3.10	A complex example	109
13.3.11	Override components	110
13.4	Modules	111
13.5	Post-mortem debugging of Jython scripts	113
13.6	Differences between Jython version 2.2.1 and 2.5	113
13.6.1	Boolean type	114
13.6.2	Character encodings	114
13.6.3	Getting the name of a Java class	115
13.7	Scripting with Groovy	115
13.7.1	Accessing variables in Groovy	118
13.7.2	Global Groovy variables	119
13.7.3	Groovy packages	119
14	Data-driven testing	121
14.1	Data-driver examples	121
14.2	General use of Data drivers	125
14.3	Examples for Data drivers	126
14.4	Advanced use	126
15	Reports and test documentation	129
15.1	Reports	130
15.1.1	Report concepts	131
15.1.2	Report contents	131
15.1.3	Creating reports	132
15.2	Testdoc documentation for Test-sets and Test-cases	134
15.3	Pkgdoc documentation for Packages, Procedures and Dependencies	135
16	Test execution	138
16.1	Test execution in batch mode	138
16.1.1	Command line usage	139
16.1.2	Windows batch script	141

16.1.3	Groovy	142
16.2	Executing tests in daemon mode	144
16.2.1	Launching the daemon	144
16.2.2	Controlling a daemon from QF-Test's command line	145
16.2.3	Controlling a daemon with the daemon API	146
17	Distributed test development	150
17.1	Referencing nodes in another test-suite	150
17.2	Managing Components	152
17.3	Merging test-suites	153
17.3.1	Importing Components	153
17.3.2	Importing Procedures and Testcases	153
17.4	Strategies for distributed development	153
17.5	Static validation of test-suites	155
17.5.1	Avoiding invalid references	156
17.5.2	Unused procedures	157
18	Automated Creation of Basic Procedures	159
18.1	Introduction	159
18.2	How to use the Procedure Builder	159
18.3	Configuration of the Procedure Builder	161
18.3.1	The Procedure Builder definition file	161
19	Interaction with Test Management Tools	163
19.1	HP ALM - Quality Center	163
19.1.1	Introduction	163
19.1.2	Step-by-step integration guide	165
19.1.3	Troubleshooting	175
19.2	Imbus TestBench	177
19.2.1	Introduction	177
19.2.2	Creating QF-Test template from interactions	177
19.2.3	Importing test-execution results	177
19.3	QMetry	178

19.3.1	Introduction	178
19.3.2	Sample Configuration	180
19.4	Klaros	181
19.4.1	Introduction	181
19.4.2	Importing QF-Test results into Klaros	181
19.5	SQS-TEST/Professional	182
19.5.1	Introduction	182
19.5.2	Integration	182
19.6	TestLink	182
19.6.1	Introduction	182
19.6.2	Generating template test-suites for QF-Test from test-cases . . .	183
19.6.3	Execution of test-cases	185
19.6.4	Importing QF-Test results into TestLink	185
20	Integration with Build Tools	187
20.1	Ant	187
20.2	Maven	188
20.3	Jenkins/Hudson	190
20.3.1	Install and start Jenkins	190
20.3.2	Requirements for GUI tests	191
20.3.3	Install QF-Test Plugin	192
20.3.4	Configure a job for QF-Test	193
21	Performing GUI-based load tests	197
21.1	Background and comparison with other techniques	197
21.2	Load tests with QF-Test	198
21.2.1	Running tests with parallel threads	198
21.2.2	Synchronization	199
21.2.3	Coordinating tests on multiple machines	199
22	Executing Manual Tests in QF-Test	201
22.1	Introduction	201
22.2	Step-by-step Guide	202

22.3	Structure of the Excel file	203
22.4	The ManualTestRunner test-suite	204
22.5	Results	205
II	Best Practices	206
23	Introduction	207
24	How to start a testing project	208
24.1	Infrastructure and testing environment	208
24.2	Location of files	210
24.2.1	Network installation	211
24.3	Component Recognition	211
25	Organizing test-suites	213
25.1	Organizing tests	213
25.2	Modularization	214
25.3	Parameterization	215
25.4	Working in multiple test-suites	215
25.5	Roles and responsibilities	217
25.6	Managing components at different levels	219
25.7	Reverse includes	219
26	Efficient working techniques	220
26.1	Using QF-Test projects	220
26.2	Creating test-suites from scratch	220
26.3	The standard library qfs.qft	221
26.4	Component storage	221
26.5	Extending test-suites	222
26.6	Working in the script editor	223
27	How to achieve robust component recognition	224
27.1	Using the default configuration	224

27.2	Using the 'Name overrides everything' mode	226
27.3	Using regular expressions for working with dynamic window titles	228
27.4	Influencing the names by implementing a NameResolver	229
27.5	Handling classes of components	230
27.5.1	Handling class changes due to re-factoring by using 'Record system-classes only'	231
27.6	Avoiding recording every component or using generic components . . .	232
27.7	Switching component recognition settings dynamically	233
28	Test execution	234
28.1	Dependencies	234
28.2	Timeout vs. delay	234
28.3	What to do if the run-log contains an error	235
III	Reference manual	236
29	Options	237
29.1	General options	239
29.1.1	Projects	241
29.1.2	Editing	242
29.1.3	Bookmarks	246
29.1.4	External tools	246
29.1.5	Backup files	249
29.1.6	Library path	252
29.1.7	License	253
29.2	Recording options	254
29.2.1	Events to record	255
29.2.2	Events to pack	257
29.2.3	Components	261
29.2.4	Recording sub-items	267
29.2.5	Recording procedures	269
29.3	Replay options	270

29.3.1	Client options	273
29.3.2	Terminal options	276
29.3.3	Event handling	280
29.3.4	Component recognition	284
29.3.5	Delays	288
29.3.6	Timeouts	290
29.4	Web options	294
29.5	Debugger options	297
29.6	Run-log options	299
29.6.1	General run-log options	300
29.6.2	Options determining run-log content	305
29.6.3	Options for mapping between directories with test-suites	308
29.7	Variables	309
30	Elements of a test-suite	311
30.1	The test-suite and its structure	311
30.1.1	Test-suite	311
30.2	Test and Sequence nodes	314
30.2.1	Test-case	314
30.2.2	Test-set	321
30.2.3	Test call	327
30.2.4	Test	330
30.2.5	Sequence	334
30.2.6	Test-step	336
30.2.7	Sequence with time limit	339
30.2.8	Extras	342
30.3	Dependencies	343
30.3.1	Dependency	344
30.3.2	Dependency reference	347
30.3.3	Setup	350
30.3.4	Cleanup	352
30.3.5	Error handler	355

30.4	Data driver	357
30.4.1	Data driver	357
30.4.2	Data table	360
30.4.3	Database	363
30.4.4	Excel data file	368
30.4.5	CSV data file	372
30.4.6	Data loop	376
30.5	Procedures	378
30.5.1	Procedure	378
30.5.2	Procedure call	381
30.5.3	Return	384
30.5.4	Package	385
30.5.5	Procedures	387
30.6	Control structures	388
30.6.1	Loop	388
30.6.2	While	391
30.6.3	Break	394
30.6.4	If	396
30.6.5	Elseif	399
30.6.6	Else	403
30.6.7	Try	405
30.6.8	Catch	408
30.6.9	Finally	411
30.6.10	Throw	413
30.6.11	Rethrow	415
30.6.12	Server script	416
30.6.13	SUT script	419
30.7	Processes	421
30.7.1	Start Java SUT client	422
30.7.2	Start SUT client	426
30.7.3	Start browser	428

30.7.4	Start process	433
30.7.5	Execute shell command	436
30.7.6	Wait for client to connect	438
30.7.7	Stop client	441
30.7.8	Wait for process to terminate	443
30.8	Events	446
30.8.1	Mouse event	447
30.8.2	Key event	450
30.8.3	Text input	454
30.8.4	Window event	457
30.8.5	Component event	459
30.8.6	Selection	462
30.8.7	File selection	466
30.9	Checks	469
30.9.1	Check text	470
30.9.2	Boolean check	476
30.9.3	Check items	480
30.9.4	Check selectable items	484
30.9.5	Check image	488
30.9.6	Check geometry	494
30.10	Queries	498
30.10.1	Fetch text	499
30.10.2	Fetch index	502
30.10.3	Fetch geometry	505
30.11	Miscellaneous	509
30.11.1	Set variable	509
30.11.2	Wait for component to appear	512
30.11.3	Wait for document to load	516
30.11.4	Wait for download to finish	520
30.11.5	Load resources	523
30.11.6	Load properties	526

30.12 HTTP Requests	528
30.12.1 Server HTTP request	528
30.12.2 Browser HTTP request	532
30.13 Windows, Components and Items	535
30.13.1 Window	535
30.13.2 Web page	540
30.13.3 Component	545
30.13.4 Item	551
30.13.5 Window group	554
30.13.6 Component group	555
30.13.7 Windows and components	556
31 Exceptions	558
IV Technical reference	566
32 Command line arguments and exit codes	567
32.1 Call syntax	567
32.2 Command line arguments	571
32.2.1 Arguments for the starter script	571
32.2.2 Arguments for the Java VM	572
32.2.3 Arguments for QF-Test	572
32.2.4 Placeholders in the filename parameter for run-log and report . .	587
32.3 Exit codes for QF-Test	588
33 GUI engines	590
34 Running an application from QF-Test	592
34.1 Various methods to start the SUT	592
34.1.1 A standalone script or executable file	593
34.1.2 An application launched through Java WebStart	594
34.1.3 An applet in a web browser	595
34.1.4 An application started with <code>java -jar <archive></code>	598

34.1.5	An application started with <code>java -classpath <classpath> <class></code>	600
34.1.6	A web application in a browser	601
35	JDK and SWT instrumentation	604
35.1	JDK instrumentation	604
35.2	JDK instrumentation background	607
35.2.1	Modifications to the JDK	607
35.2.2	Effects of the instrumentation	608
35.3	SWT instrumentation	608
35.3.1	Preparation for manual SWT instrumentation	609
35.3.2	Manual SWT instrumentation for eclipse based applications . . .	609
35.3.3	Manual instrumentation for standalone SWT applications	610
36	Technical details about miscellaneous issues	611
36.1	Drag&Drop	611
36.2	Component recognition	612
36.3	Timing	614
36.4	Regular expressions	614
36.5	Line breaks under Unix and Windows	616
36.6	Quoting and escaping special characters	616
36.7	Include file resolution	617
37	Jython and Groovy scripting	619
37.1	Module load-path (Jython)	619
37.2	The plugin directory	620
37.3	The package cache (Jython)	620
37.4	Initialization (Jython)	621
37.5	Namespace environment for script execution (Jython)	621
37.6	Run-context API	622
37.7	The <code>qf</code> module	639
37.8	Image API	642
37.8.1	The <code>ImageWrapper</code> class	642

37.9	Exception handling	645
37.10	Debugging scripts (Jython)	645
38	Special features for web applications	647
38.1	Pseudo DOM API	647
38.1.1	The abstract <code>Node</code> class	647
38.1.2	The <code>DocumentNode</code> class	652
38.1.3	The <code>FrameNode</code> class	654
38.1.4	The <code>DomNode</code> class	655
38.1.5	The <code>DialogNode</code> class	656
38.2	Special support for various AJAX toolkits	657
38.2.1	AJAX resolver concepts	658
38.2.2	Supported AJAX toolkits	659
38.2.3	Setting unique IDs	664
38.3	Customizing QF-Test for your AJAX toolkit	666
38.3.1	Determining specifics of your web page	666
38.3.2	Example for WebCarConfigurator	669
39	Extension APIs	678
39.1	Resolver hooks	678
39.1.1	Overview over the supported kinds of resolvers	679
39.1.2	The <code>ClassNameResolver</code> interface	680
39.1.3	The <code>GenericClassNameResolver</code> interface	681
39.1.4	The <code>NameResolver2</code> interface	681
39.1.5	The <code>IdResolver</code> interface	682
39.1.6	The <code>FeatureResolver2</code> interface	683
39.1.7	The <code>ExtraFeatureResolver</code> interface	683
39.1.8	The <code>ItemNameResolver2</code> interface	686
39.1.9	The <code>ItemValueResolver2</code> interface	687
39.1.10	The <code>TreeTableResolver</code> interface	688
39.1.11	The <code>InterestingParentResolver</code> interface	689
39.1.12	Simplified resolver creation with the <code>resolvers</code> module	689

39.1.13	The <code>ResolverRegistry</code>	697
39.1.14	Error handling	703
39.2	Implementing custom item types with the <code>ItemResolver</code> interface . .	703
39.2.1	<code>ItemResolver</code> concepts	704
39.2.2	The <code>ItemResolver</code> interface	705
39.2.3	The class <code>SubItemIndex</code>	709
39.2.4	The <code>ItemRegistry</code>	710
39.2.5	Default item representations	713
39.3	Implementing custom checks with the <code>Checker</code> interface	715
39.3.1	The <code>Checker</code> interface	715
39.3.2	The class <code>Pair</code>	717
39.3.3	The <code>CheckType</code> interface and its implementation <code>DefaultCheckType</code>	718
39.3.4	The class <code>CheckDataType</code>	718
39.3.5	The class <code>CheckData</code> and its subclasses	719
39.3.6	The <code>CheckerRegistry</code>	721
39.3.7	Custom checker example	722
39.4	Working with the Eclipse Graphical Editing Framework (GEF)	724
39.4.1	Recording GEF items	724
39.4.2	Implementing a GEF <code>ItemNameResolver2</code>	726
39.4.3	Implementing a GEF <code>ItemValueResolver2</code>	728
39.5	Test-run listeners	728
39.5.1	The <code>TestRunListener</code> interface	729
39.5.2	The class <code>TestRunEvent</code>	730
39.5.3	The class <code>TestSuiteNode</code>	731
39.6	<code>ResetListener</code>	732
39.7	DOM processors	733
39.7.1	The <code>DOMProcessor</code> interface	734
39.7.2	The <code>DOMProcessorRegistry</code>	735
39.7.3	Error handling	736
39.8	Image API extensions	737
39.8.1	The <code>ImageRep</code> class	737

39.8.2 The <code>ImageComparator</code> interface	740
40 Daemon mode	741
40.1 Daemon concepts	741
40.2 Daemon API	742
40.2.1 The <code>DaemonLocator</code>	743
40.2.2 The <code>Daemon</code>	743
40.2.3 The <code>TestRunDaemon</code>	745
40.2.4 The <code>DaemonRunContext</code>	748
40.2.5 The <code>DaemonTestRunListener</code>	755
40.3 Daemon security considerations	755
40.3.1 Creating your own keystore	756
40.3.2 Specifying the keystore	757
40.3.3 Specifying the keystore on the client side	757
41 The Procedure Builder definition file	758
41.1 Placeholders	758
41.2 Conditions for Package and Procedure Definition	760
41.3 Interpretation of the Component Hierarchy	761
41.4 Details about the <code>@CONDITION</code> tag	762
42 The <code>ManualStepDialog</code>	764
42.1 The <code>ManualStepDialog</code> API	764
43 Details about transforming nodes	766
43.1 Introduction	766
43.2 Transformation with type changes	766
43.3 Additional transformations below the <code>Extras</code> node	767
43.3.1 Transformations without side-effects	767
43.3.2 Transformations with side-effects	768
44 Details about the algorithm for image comparison	769
44.1 Introduction	769

44.2	Description of algorithms	770
44.2.1	Classic image check	770
44.2.2	Pixel-based identity check	771
44.2.3	Pixel-based similarity check	772
44.2.4	Block-based identity check	774
44.2.5	Block-based similarity check	775
44.2.6	Histogram check	777
44.2.7	Analysis with Discrete Cosine Transformation	778
44.2.8	Block-based analysis with Discrete Cosine Transformation	780
44.2.9	Bilinear Filter	782
44.3	Description of special functions	783
44.3.1	Image-in-image search	783
45	Result lists	786
45.1	Introduction	786
45.2	Specific list actions	788
45.2.1	All types of lists	788
45.2.2	Replacing	788
45.2.3	Error list	789
45.3	Exporting and loading results	789
46	Generic classes	790
46.1	Overview generic classes	790
46.2	Mapping of complex components	795
46.2.1	Mapping of Lists	795
46.2.2	Mapping of TabPanels and accordions	795
46.2.3	Mapping of Tables	796
46.2.4	Mapping of Trees	797
46.3	Properties of generic classes	798
46.3.1	Accordion	798
46.3.2	BusyPane	798
46.3.3	Button	799

46.3.4 Calendar	799
46.3.5 ColorPicker	799
46.3.6 CheckBox	800
46.3.7 Closer	800
46.3.8 Divider	801
46.3.9 ComboBox	801
46.3.10 Expander	802
46.3.11 FileChooser	802
46.3.12 Graphics	802
46.3.13 Icon	803
46.3.14 Indicator	803
46.3.15 Item	804
46.3.16 Label	804
46.3.17 Link	805
46.3.18 List	805
46.3.19 Maximizer	806
46.3.20 Menu	806
46.3.21 MenuItem	807
46.3.22 Minimizer	807
46.3.23 Popup	808
46.3.24 RadioButton	808
46.3.25 Restore	809
46.3.26 Panel	809
46.3.27 ProgressBar	810
46.3.28 ScrollBar	810
46.3.29 Separator	811
46.3.30 Sizer	811
46.3.31 Slider	811
46.3.32 Spacer	812
46.3.33 Spinner	812
46.3.34 SplitPanel	813

46.3.35 Table	813
46.3.36 TableCell	814
46.3.37 TableFooter	814
46.3.38 TableHeader	814
46.3.39 TableHeaderCell	815
46.3.40 TabPanel	815
46.3.41 TableRow	816
46.3.42 Text	816
46.3.43 TextArea	817
46.3.44 TextField	817
46.3.45 Thumb	818
46.3.46 ToggleButton	818
46.3.47 ToolBar	818
46.3.48 ToolBarItem	819
46.3.49 ToolTip	819
46.3.50 Tree	819
46.3.51 TreeNode	820
46.3.52 TreeTable	821
46.3.53 Window	821
A FAQ - Frequently Asked Questions	822
B Release notes	830
B.1 Version 4.0.8 - November 4, 2015	830
B.2 Version 4.0.7 - August 27, 2015	831
B.3 Version 4.0.6 - July 16, 2015	832
B.4 Version 4.0.5 - May 12, 2015	833
B.5 Version 4.0.4 - March 5, 2015	834
B.6 Version 4.0.3 - January 29, 2015	836
B.7 Version 4.0.2-p2	838
B.8 Version 4.0.2-p1	839
B.9 Version 4.0.2 - October 22, 2014	839

B.10	Version 4.0.1 - August 28, 2014	841
B.11	Main new features in version 4	843
B.12	Version 4.0.0 - July 31, 2014	845
B.13	Version 4.0.0-rc1	846
B.14	Version 4.0.0-beta3	848
B.15	Version 4.0.0-beta2	849
B.16	Version 4.0.0-beta1	850
B.17	Version 3.5.5	855
B.18	Version 3.5.4	856
B.19	Version 3.5.3	856
B.20	Version 3.5.2	857
B.21	Version 3.5.1	859
B.22	Main new features in version 3.5	860
B.23	Version 3.5.0	860
B.24	Version 3.5M3	863
B.25	Version 3.5M2	864
B.26	Version 3.5M1	865
B.27	Version 3.4.11	869
B.28	Version 3.4.10	870
B.29	Version 3.4.9	870
B.30	Version 3.4.8	871
B.31	Version 3.4.7	872
B.32	Version 3.4.6	872
B.33	Version 3.4.4	873
B.34	Version 3.4.3	874
B.35	Version 3.4.2	875
B.36	Version 3.4.1	876
B.37	Main new features in version 3.4	878
B.38	Version 3.4.0	879
B.39	Version 3.4M2	881
B.40	Version 3.4M1	883

B.41	Version 3.3.0	885
B.42	Version 3.2.2	888
B.43	Version 3.2.1	888
B.44	New features in QF-Test version 3.2	889
B.45	Version 3.2.0	890
B.46	Version 3.1.4	891
B.47	Version 3.1.3	893
B.48	Version 3.1.2	894
B.49	Version 3.1.1	896
B.50	Version 3.1.0	898
B.51	Version 3.0.2	900
B.52	Version 3.0.1	901
B.53	New features in QF-Test version 3	902
C	Keyboard shortcuts	904
C.1	Navigation and editing	904
C.2	Record and replay functions	906
C.3	Keyboard helper	907
D	Glossary	909
E	Third party software	910

List of Figures

2.1	Structure of a test-suite	10
2.2	Insertion marker	12
2.3	Example table	13
2.4	The simple search dialog	16
2.5	The advanced search dialog	17
2.6	Result list for 'Locate references'	18
2.7	Incremental search	19
2.8	The replace dialog	21
2.9	The replace query dialog	22
3.1	Quickstart Wizard	26
3.2	Startup sequence created by the Quickstart Wizard	26
4.1	Disabled and enabled Record button	30
5.1	Components of a GUI	38
5.2	Component hierarchy of a Swing SUT	40
5.3	Pseudo class hierarchy for web elements	43
5.4	Update components dialog	48
6.1	An Item for a table cell	53
7.1	Direct and fallback bindings	58
7.2	System variables	58
7.3	Variable example	60
7.4	Variable bindings example	61

8.1	A simple test and its run-log	69
8.2	Error states in a run-log	70
9.1	Sequences and Tests	79
9.2	Packages and Procedures	82
10.1	The project view	87
11.1	Standard library <code>qfs.qft</code>	89
12.1	Dependency stack A-B-C-D	92
12.2	Dependency C has "forced cleanup"	93
12.3	Rollback and build up stack to A-E	94
12.4	Exception in forced cleanup sequence of C causes B to clean up	95
12.5	Change in characteristic variable causes cleanup of A	96
12.6	Execution of Catch and Error handler nodes	98
13.1	Detail view of a Server script with help window for <code>rc</code> methods	100
14.1	A simple data-driven test	121
14.2	Data table example	122
14.3	Run-log of a data-driven test	123
14.4	Data-driven test with nested loops	123
14.5	Second data table example	124
14.6	Run-log of a data-driven test with nested loops	125
15.1	Example report	130
17.1	Result of analyzing references	157
18.1	Recorded procedures	160
18.2	The Procedure Builder definition file	162
19.1	Integration with ALM - QualityCenter	164
19.2	QF-Test VAPI-XP-TEST test-case in HP ALM - QualityCenter	165
19.3	In Test plan create new Test-set	166

19.4	Create new test of type VAPI-XP-TEST	167
19.5	HP VAPI-XP Wizard	168
19.6	Test details	169
19.7	Copy template content to script text area	170
19.8	New test-set in Test lab section	171
19.9	Add test to execution grid	172
19.10	Run the test	173
19.11	Test result	174
19.12	Uploaded run-log	175
19.13	Script debug run	176
19.14	QF-Test run-log in QMetry	180
20.1	Jenkins after start-up.	191
20.2	Install QF-Test Plugin.	192
20.3	Set custom workspace.	193
20.4	Add build step to run QF-Test	194
20.5	Configure build step advanced options.	194
20.6	Configure post build steps.	195
22.1	Example for a ManualStepDialog	202
25.1	Structure of multiple test-suites	216
25.2	Including test-suites of level 1	217
25.3	Structure of different test-suites with roles	218
27.1	Default configuration for component recording	225
27.2	Default configuration for component recognition	226
27.3	'Name overrides everything' configuration for component recording	227
27.4	'Name overrides everything' configuration for component recognition	228
27.5	Using a regular expression in the Feature attribute	229
27.6	Option to configure recording of system classes only	231
29.1	Options tree	238
29.2	General options	239

29.3	Projects	241
29.4	Editing	243
29.5	Bookmarks	246
29.6	External tools options	247
29.7	Backup file options	250
29.8	Library path option	252
29.9	License options	253
29.10	Recording options	254
29.11	Options for events to record	256
29.12	Options for events to pack	257
29.13	Dragging to a sub-menu	259
29.14	Options for recording components	261
29.15	Popup menu for recording components	262
29.16	Options for recording sub-items	268
29.17	Procedure Builder options	269
29.18	Replay options	270
29.19	Client options	273
29.20	Terminal options	277
29.21	Event handling options	280
29.22	Component recognition options	285
29.23	Delay options	288
29.24	Timeout options	290
29.25	Web options	294
29.26	Debugger options	298
29.27	General run-log options	300
29.28	Options determining run-log content	305
29.29	Options for mapping between directories with test-suites	308
29.30	Variable options	309
30.1	Test-suite attributes	312
30.2	Test-case attributes	316
30.3	Test-set attributes	323

30.4	Test call Attributes	328
30.5	Test attributes	331
30.6	Sequence attributes	335
30.7	Test-step attributes	337
30.8	Sequence with time limit attributes	340
30.9	Extras attributes	343
30.10	Dependency attributes	345
30.11	Dependency reference attributes	348
30.12	Setup attributes	351
30.13	Cleanup attributes	353
30.14	Error handler attributes	355
30.15	Data driver attributes	358
30.16	Data table attributes	361
30.17	Database attributes	364
30.18	Excel data file attributes	369
30.19	CSV data file attributes	373
30.20	Data loop attributes	376
30.21	Procedure Attributes	379
30.22	Procedure call Attributes	382
30.23	Return Attributes	384
30.24	Package Attributes	386
30.25	Procedures Attributes	387
30.26	Loop attributes	389
30.27	While attributes	392
30.28	Break attributes	395
30.29	If attributes	397
30.30	Elseif attributes	400
30.31	Else attributes	403
30.32	Try attributes	406
30.33	Catch attributes	409
30.34	Finally attributes	412

30.35 Throw attributes	414
30.36 Rethrow attributes	415
30.37 Server script attributes	417
30.38 SUT script attributes	419
30.39 Start Java SUT client attributes	423
30.40 Start SUT client attributes	426
30.41 Start browser attributes	430
30.42 Start process attributes	434
30.43 Execute shell command attributes	437
30.44 Wait for client to connect attributes	439
30.45 Stop client attributes	442
30.46 Wait for process to terminate attributes	444
30.47 Mouse event attributes	447
30.48 Key event attributes	451
30.49 Text input attributes	455
30.50 Window event attributes	458
30.51 Component event attributes	460
30.52 Selection attributes	465
30.53 File selection attributes	467
30.54 Check text attributes	472
30.55 Boolean check attributes	477
30.56 Check items attributes	481
30.57 Check selectable items attributes	485
30.58 Check image attributes	490
30.59 Check geometry attributes	495
30.60 Fetch text attributes	500
30.61 Fetch index attributes	503
30.62 Fetch geometry attributes	506
30.63 Set variable attributes	510
30.64 Wait for component to appear attributes	513
30.65 Wait for document to load attributes	517

30.66 Wait for download to finish attributes	521
30.67 Load resources attributes	524
30.68 Load properties attributes	527
30.69 Server HTTP request Attribute	530
30.70 Browser HTTP request Attribute	533
30.71 Window attributes	536
30.72 Web page attributes	541
30.73 Component attributes	546
30.74 Item attributes	552
30.75 Window group attributes	554
30.76 Component group attributes	555
30.77 Windows and components attributes	556
34.1 Starting the SUT from a script or executable	593
34.2 Starting the SUT through Java WebStart	594
34.3 Starting the SUT as an applet in a web browser	596
34.4 Starting the SUT from a jar archive	599
34.5 Starting the SUT via the main class	600
34.6 Starting a web-based SUT in a browser	602
35.1 Dialog for JDK instrumentation	605
38.1 HTML code inspection in Firefox	668
38.2 WebCarConfigurator	669
38.3 Recording of '-5%' button in WebCarConfigurator	670
38.4 Recording with goodCssClasses in WebCarConfigurator	671
38.5 Recording with genericClasses in WebCarConfigurator	672
38.6 Recording of SPAN textfields	673
38.7 Recording TextFields in WebCarConfigurator	674
38.8 Recording of table in WebCarConfigurator	675
38.9 Recording of resolved table item in WebCarConfigurator	677
44.1 Original image	770

44.2	Classic image check	771
44.3	Pixel-based identity check	772
44.4	Pixel-based similarity check	773
44.5	Block-based identity check	774
44.6	Block-based similarity check	776
44.7	Histogram	777
44.8	Analysis with Discrete Cosine Transformation	779
44.9	Block-based analysis with Discrete Cosine Transformation	781
44.10	Bilinear Filter	782
44.11	Image-in-image search: Expected image	784
44.12	Image-in-image search: Got image	784
45.1	Sample result list for 'Locate references'	787
A.1	Set browser maximum memory	828
C.1	Keyboard helper	908

List of Tables

1.1	Supported technology versions	3
4.1	Test result counter in the status line	33
6.1	Separators and index formats for element access	54
7.1	Definitions in the special group <code>qftest</code>	66
9.1	Relative procedure calls	83
22.1	Description of the Excel file for the definition of manual tests	203
22.2	Description of the Excel file with the results of manual tests	204
22.3	Description of the global variables in the <code>ManualTestRunner</code> test-suite . .	204
22.4	States of manual test-execution	205
26.1	List of variables with auto-completion.	223
30.1	Placeholders for the Name for separate run-log attribute	318
30.2	Placeholders for the Name for separate run-log attribute	324
30.3	Placeholders for the Name for separate run-log attribute	329
30.4	Placeholders for the Name for separate run-log attribute	332
30.5	Placeholders for the Name for separate run-log attribute	359
30.6	Iteration range examples	362
30.7	Iteration range examples	365
30.8	Database drivers	366
30.9	Database connection strings	367
30.10	Iteration range examples	370

30.11 Iteration range examples	374
30.12 Iteration range examples	377
30.13 Condition examples	393
30.14 Condition examples	398
30.15 Condition examples	401
30.16 Modifier values	449
30.17 Modifier values	453
30.18 Supported SWT widgets for a Selection event	463
30.19 Supported DOM nodes for a Selection event	464
30.20 Provided Check types of Boolean check	470
30.21 Components supported by Fetch text	499
30.22 Components supported by Fetch geometry	505
30.23 Extra features assigned by QF-Test	539
30.24 Extra features assigned by QF-Test	544
30.25 Extra features assigned by QF-Test	549
30.26 Sub-items of complex Swing components	551
32.1 Placeholders in filename parameters	587
32.2 Exit codes for QF-Test	588
32.3 <code>calldaemon</code> exit codes for QF-Test	589
38.1 Supported AJAX toolkits	658
38.2 Ext JS resolver versions	660
38.3 GWT resolver versions	660
38.4 ICEfaces resolver versions	661
38.5 jQuery UI resolver versions	661
38.6 jQuery EasyUI resolver versions	661
38.7 PrimeFaces resolver versions	662
38.8 Qooxdoo resolver versions	662
38.9 RAP resolver versions	662
38.10 RichFaces resolver versions	663
38.11 Vaadin resolver versions	663

38.12 ZK resolver versions	664
39.1 Internal item representations for JavaFX GUI elements	713
39.2 Internal item representations for Swing GUI elements	714
39.3 Internal item representations for SWT GUI elements	714
39.4 Internal item representations for DOM nodes	714
40.1 The run state	749
41.1 Placeholders for component procedures	759
41.2 Additional placeholders for container procedures	760
41.3 Comment attributes for procedure creation	761
41.4 Hierarchy placeholders	762
41.5 Samples for the @CONDITION tag	763
46.1 Generic classes for input and output	791
46.2 Generic classes for interactions	792
46.3 Generic classes for container components	793
46.4 Generic classes for data containers	794
46.5 Generic classes for data	794
46.6 Generic classes for specific components	795
46.7 Mapping of Lists	795
46.8 Mapping of TabPanels	796
46.9 Mapping of Tables	797
46.10 Mapping of trees	797
46.11 Checktypes for Accordion	798
46.12 Special qfs:type values for Buttons	799
46.13 Special qfs:type values for CheckBoxes	800
46.14 Checktypes for Checkbox	800
46.15 Special qfs:type values for Closer	801
46.16 Checktypes for ComboBox	801
46.17 Special qfs:type values for Expander	802
46.18 Special qfs:type values for Icon	803

46.19 Special qfs:type values for Indicator	803
46.20 Special qfs:type values for Item	804
46.21 Checktypes for Item	804
46.22 Special qfs:type values for Labels	805
46.23 Special qfs:type values for Links	805
46.24 Special qfs:type values for List	806
46.25 Checktypes for List	806
46.26 Special qfs:type values for Maximizer	806
46.27 Special qfs:type values for Menu	807
46.28 Special qfs:type values for Minimizer	807
46.29 Special qfs:type values for Popup	808
46.30 Special qfs:type values for RadioButtons	808
46.31 Checktypes for RadioButton	809
46.32 Special qfs:type values for Restore	809
46.33 Special qfs:type values for Panel	810
46.34 Checktypes for ProgressBar	810
46.35 Special qfs:type values for Sizer	811
46.36 Checktypes for Slider	812
46.37 Special qfs:type values for Spacer	812
46.38 Checktypes for Spinner	813
46.39 Checktypes for Table	813
46.40 Checktypes for TableCell	814
46.41 Checktypes for TableHeader	815
46.42 Checktypes for TableHeaderCell	815
46.43 Checktypes for TabPanel	816
46.44 Special qfs:type values for Text	816
46.45 Checktypes for TextArea	817
46.46 Special qfs:type values for TextField	817
46.47 Checktypes for TextField	818
46.48 Checktypes for ToggleButton	818
46.49 Checktypes for Tree	820

46.50	Checktypes for TreeNode	820
46.51	Special qfs:type values for Window	821
B.1	Highlights in QF-Test version 4	843
B.2	New features in QF-Test version 3.5	860
B.3	New features in QF-Test version 3.4	879
B.4	New features in QF-Test version 3.2	890
B.5	New features in QF-Test version 3	903
C.1	Shortcuts for navigation and editing	906
C.2	Shortcuts for special record and replay functions	907

Part I

User manual

Chapter 1

Installation and startup

QF-Test is available in three installation options: The `qftest-4.0.8.tar.gz` archive for Linux and Unix only, the executable setup file `qftest-4.0.8.exe` for Windows only and the `qftest-4.0.8.zip` archive for Linux, Unix and Windows.

Basically it is possible to have different versions of QF-Test installed in parallel. Existing configuration files will not be overwritten in this process.

In [section 24.2^{\(210\)}](#) you can find best practices about the QF-Test installation.

1.1 System requirements

1.1.1 Prime precondition

QF-Test needs Java 6 or higher (JRE or JDK, Java 8 recommended) installed in order to work. To check for an adequate Java being installed on your system please open a command window e.g. `cmd.exe` on Windows and use the command:

```
java -version
```

If Java is installed you should get something like:

```
java version "1.8.0_45"  
Java(TM) SE Runtime Environment (build 1.8.0_45-b14)  
Java HotSpot(TM) Server VM (build 25.45-b02, mixed mode)
```

Versions greater or equal 1.6.0 are fine as 1.6 means Java 6.

If you get an error or a version less 1.6.0, please install a current JDK or JRE from <http://www.java.com>. QF-Test can work with both 32bit and 64bit Java versions.

Note

If your system under test (SUT) requires to still use Java 5, that should work as well, but is not officially supported anymore. The Java command for the SUT can be configured separately when creating the setup sequence for your SUT.

1.1.2 Supported technology versions

The following table summarizes the officially supported versions of operating systems and required software for this QF-Test version 4.0.8. Support for additional systems and versions may be available on request but is not owed by QFS. Another option to get support for older software can be to use one of the older QF-Test versions that are still available for download at <http://www.qfs.de/en/qftest/download.html>.

Technology	Version restriction	Comment
Operating system		
Windows	Vista, 7, 8, 10, Server 2008 + 2012	
Linux		No known restrictions.
Unix		Swing and JavaFX testing possible on Solaris, HP-UX, AIX,...
MAC OS X		Not officially supported but Swing and JavaFX testing possible.
Java		
JDK/JRE	1.6 (6) - 1.8 (8)	1.5 for SUT also possible but not officially supported.
Swing		All platforms.
JavaFX	2.2 - 8 (2.1 with limitations)	All platforms.
SWT	3.7 - 4.5	Windows and Linux GTK only. Older versions available on request.
Web Browser		
Internet Explorer	8 - 11	
Firefox 32bit	24esr or 26 - 42	Older versions available on request.
Firefox 64bit	31esr or 32 - 42	Linux only.
Chrome	Current stable Chromium version (33), part of the QF-Test distribution	Windows only.
AJAX Toolkits		
See section 38.2 ⁽⁶⁵⁷⁾		

Table 1.1: Supported technology versions

1.2 Linux/Unix Installation

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

Unpack either the `.zip` archive with `unzip qftest-4.0.8.zip` or the `.tar.gz` archive with `tar xfvz qftest-4.0.8.tar.gz`. This will create a directory named `qftest`, which we will refer to as the main or root directory of QF-Test. When the *qftest* product family grows, this directory will become the home for other products as well.

After unpacking a QF-Test archive for the first time, QF-Test's root directory will hold only the version-specific subdirectory `qftest-4.0.8`. When upgrading, a new subdirectory for the current version will be added.

To finish the installation, change to the specific directory for the current QF-Test version with `cd qftest/qftest-4.0.8` and run one of the two setup scripts provided. `setup.sh` is a plain Bourne shell script while `setup.ksh` is written for the Korn shell. On Solaris you must run `./setup.ksh`. On other Unix systems both scripts should work equally well but the Bourne shell version `./setup.sh` is preferred. Note that if you need to use the Korn shell scripts and want to create you own link or start QF-Test directly from its `bin` directory, you need to use the `qftest.ksh` start script instead of the plain `qftest` script.

These setup scripts will create the directories `log` and `jython` under QF-Test's root directory if they don't already exist. Additionally they will offer to create a symbolic link from the `/usr/local/bin` directory (or `/usr/bin` if there is no `/usr/local/bin`) to the respective Bourne shell or Korn shell run script for the `qftest` command. You need to have write permission to the `/usr/local/bin` directory for the link to be created.

Subsequently the default `java` program for QF-Test is determined. This can be overridden later with the `-java <executable>(572)` argument. The setup script searches `PATH` and proposes to use the first `java` program it detects. If you want to use a different program or if none was found, you can enter one. The script determines the JDK version automatically.

Next setting to perform is the maximum amount of memory to be used by QF-Test. As default 512 MB are taken. Alternatively QF-Test can be started with the `-J-XmxZZZm` command line argument, where `ZZZ` defines the memory in MB.

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

Those of the above settings that differ from the default are written to the file

`launcher.cfg` in QF-Test's root directory. This file is read by the `qftest` launch-script and also evaluated during an update of QF-Test.

1.3 Windows Installation

On Windows you can install QF-Test using either the Windows specific `qftest-4.0.8.exe` or the archive `qftest-4.0.8.zip`.

Installing QF-Test from `qftest-4.0.8.exe` follows the standard setup procedures for Windows. You will be asked to accept the license agreement and select the root directory for QF-Test. When upgrading to a new version, make sure you select the same directory again (this will be the default).

To install QF-Test from the `qftest-4.0.8.zip` archive, first extract the archive in a convenient place. The default location is `C:\Program Files\qfs`. This will create a directory called `qftest` which we will refer to as the root directory of QF-Test. When the *qftest* product family grows, this directory will become the home for other products as well. After extracting the archive, run the executable file called `setup.exe` in the version-specific directory `qftest-4.0.8` under QF-Test's root directory. This will take you through the same setup steps as `qftest-4.0.8.exe`, except that no files will be extracted. If you later want to uninstall QF-Test you'll have to remove its files by hand after running the uninstaller.

Note

Make sure that you select QF-Test's root directory as the target directory for the setup.

In both cases, setup will create a menu shortcut and an optional desktop icon for QF-Test and create a few registry entries to associate `*.qft`, `*.qzp`, `*.qrz` and `*.qrl` files with QF-Test. It will also copy the `qftest.exe` executable to the Windows system directory and store the location of QF-Test's root directory and some version information in the registry, so `qftest.exe` can be run from the console or batch scripts without further ado. Information about the QF-Test version will also be fetched from the Windows registry.

When finished, the setup program will offer to configure the Java program for QF-Test. This can be done with the help of a small dialog in which you can select the Java program for QF-Test.

Furthermore, the dialog let you adjust the maximum amount of memory to be used by QF-Test. 512 MB are taken by default.

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

The values above are stored in the Windows registry from where they are read by the `qftest` start program. You can run the configuration program any time from the system

menu to change these settings. To remove them, clear the respective field or set is to system standard respectively and select "OK".

1.4 The license file

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

4.0+

Since QF-Test 4.0 it is possible to activate or update your QF-Test license directly by use of [Help→Update license...](#).

Of course the traditional way as described below is also still valid.

Place the license file in QF-Test's installation root directory, e.g. `C:\Program Files (x86)\qfs\qftest`.

Make sure it is named `license` **with no extension**. Some mail clients try to guess the file type and add an extension on their own. When upgrading to a new version you can simply keep the license file provided that it is valid for the new version.

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

- Place the file `license.new` in the same directory as the current license. Make sure that this directory and the file `license` are writable by you.
- Start QF-Test in interactive mode. QF-Test will detect the license update, verify its validity and offer to upgrade your license file.
- If you agree, the current license will be renamed to `license.old` and the new, combined license will be written to `license`. When you are satisfied that everything is OK, you can remove the files `license.old` and `license.new`.
- If QF-Test doesn't seem to recognize the license upgrade, make sure that the timestamp of the file `license.new` is newer than that of the file `license`. Also make sure that no other instance of QF-Test is running on your computer.

In case you need to specify a special name or location for the license file or work with more than one license, this can be achieved with help of the `-license <file>`⁽⁵⁷⁷⁾ argument as described in [chapter 32](#)⁽⁵⁶⁷⁾.

1.5 The configuration files

QF-Test saves all of its window configuration and those global options that represent personal preferences together in one file which is usually called `config` and placed in the directory `.qftest` under your home directory (typically called `Personal Files` or `C:\Users\<username>` or `$USERPROFILE%` under Windows). This file is not read when QF-Test is run in batch mode (see [section 1.6^{\(7\)}](#)).

System specific options that need to be shared between users are saved in a file called `qftest.cfg` in the root directory of QF-Test. The locations of the configuration files can be changed with the command line arguments `-usercfg <file>`⁽⁵⁸⁶⁾ and `-systemcfg <file>`⁽⁵⁸⁴⁾ and you will have to make use of the latter if you don't have write permission for QF-Test's root directory.

1.6 Starting QF-Test

QF-Test can be run in two modes. In normal mode QF-Test is the editor for test-suites and run-logs and the control center for running programs, capturing events and executing tests. When run with the `-batch`⁽⁵⁷¹⁾ argument, QF-Test goes into "batch" mode. Instead of opening an editor window, the test-suites given on the command line are loaded and executed automatically without the need for supervision. The result of the test is reflected in QF-Test's `exit code`⁽⁵⁸⁸⁾, optional run-logs (see [section 8.1^{\(68\)}](#)) and reports (see [chapter 15^{\(129\)}](#)).

The setup script for Linux/Unix offers to create a symbolic link from `/usr/local/bin` to the `qftest` start script in the `qftest-4.0.8/bin` directory under QF-Test's root directory. That way you can simply enter `qftest` at the shell prompt to launch the application.

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

When run from the command line, QF-Test offers a wide range of arguments for customization, like selecting the Java VM to use. These are explained in detail in [chapter 32^{\(567\)}](#).

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

1.7 Firewall Security Warning

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

Chapter 2

The user interface

This chapter explains the structure of QF-Test's main window. When you are done reading, it might be a good idea to run QF-Test and try things out. In the **Help** menu there is an entry labeled **Tutorial** that should bring up your web browser with a hands-on, learning-by-doing tutorial. Should this fail because QF-Test cannot determine your system's standard browser, the tutorial can be found in the directory `qftest-4.0.8/doc/tutorial`, where a PDF version is also available.

2.1 The test-suite

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

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

There are more than 60 different kinds of nodes all of which are explained in detail in the reference manual⁽³¹¹⁾. Some nodes are used as containers for data while others control the execution of a test. All of them have their own unique set of attributes.

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

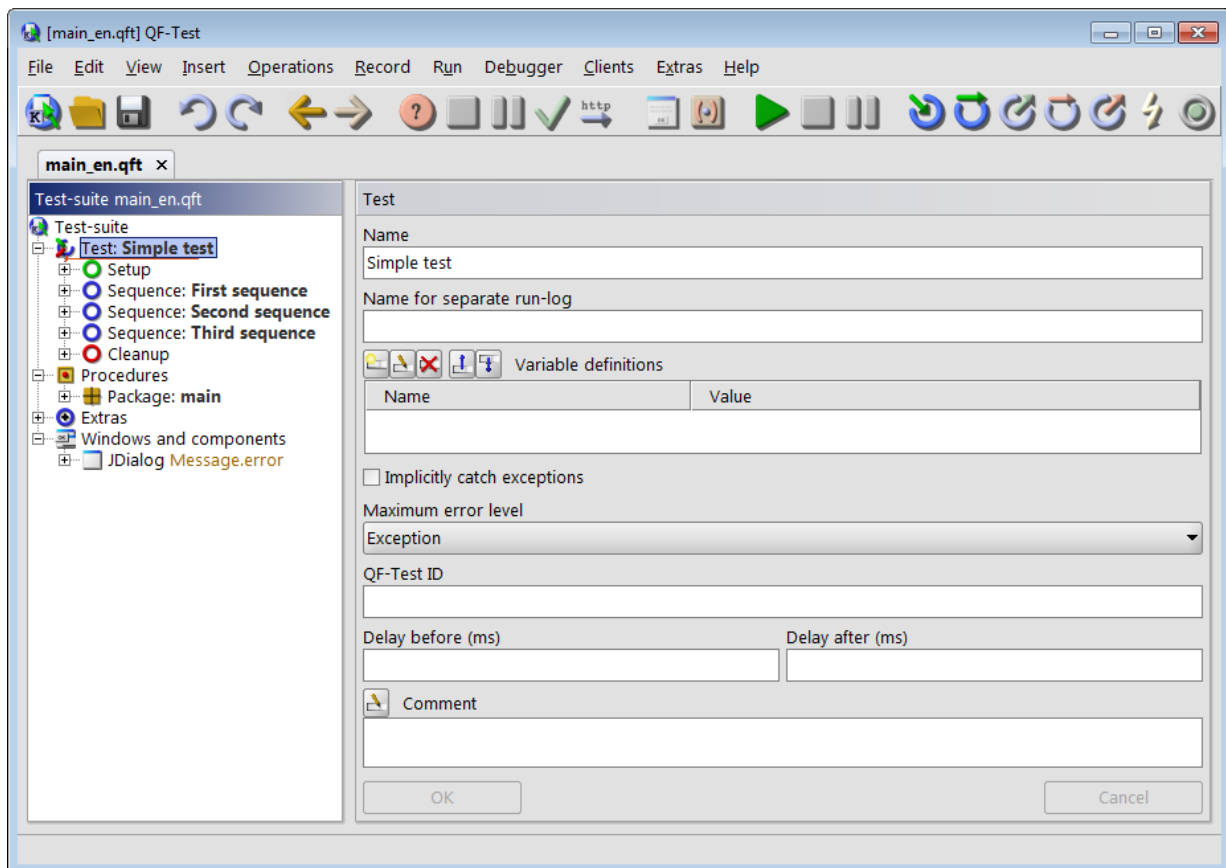


Figure 2.1: Structure of a test-suite

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

The basic structure of a test-suite and thus the child nodes of the Test-suite⁽³¹¹⁾ root node is fixed. An arbitrary number of Test-set⁽³²¹⁾, Test-case⁽³¹⁴⁾ or Test⁽³³⁰⁾ nodes are followed by the Procedures⁽³⁸⁷⁾, Extras⁽³⁴²⁾ and Windows and components⁽⁵⁵⁶⁾ nodes. The Procedures node holds Packages⁽³⁸⁵⁾ and Procedures⁽³⁷⁸⁾ which are explained further in [section 9.3](#)⁽⁸²⁾. The Extras node is a kind of playground or clipboard where all kinds of nodes can be added for experimentation or temporary storage. The windows and components of the SUT's user interface are represented as Window⁽⁵³⁵⁾ and Component⁽⁵⁴⁵⁾ nodes which are located below the Windows and components node.

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

2.2 Basic editing

Editing a test-suite falls into two categories: operations like Cut/Copy/Paste on the tree's nodes and changing the attributes of a node. The latter can be done either by editing the fields in the detail view and selecting *OK* or pressing **Return**, or by bringing up a dialog for the selected node with **Edit→Properties** or **Alt-Return** and changing the values there. If you change some values in the detail view and forget to press *OK* before moving the selection to another node, QF-Test will pop up a dialog with the changed values, asking you to either confirm your changes or discard them. This feature can be turned off with the option *Ask before implicitly accepting detail modifications*⁽²⁴³⁾.

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

An extremely useful feature is the **Edit→Undo** function (**Ctrl-Z**) which will take back any kind of change made to the test-suite, including recordings or use of the replace dialog. Changes are undone step by step. If you find you went too far and undid more than you wanted, you can use **Edit→Redo** (**Ctrl-Y**) to undo the undone. The number of steps that can taken back are limited only by available memory and can be configured with the option *Number of undo levels per suite*⁽²⁴⁴⁾ (default 30).

2.2.1 Navigating the tree

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

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

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

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

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

2.2.2 Insertion marker

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

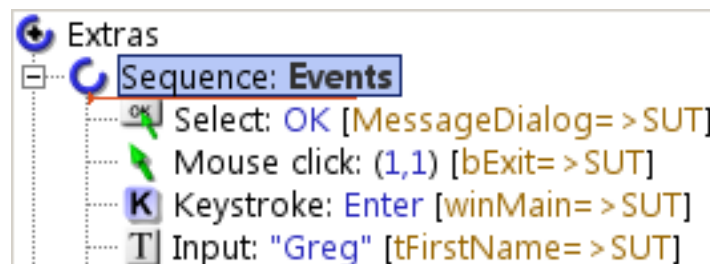


Figure 2.2: Insertion marker

Nodes are always inserted after the selected node. If the selected node is expanded, the new node is inserted as the first child of the selected node, otherwise it becomes a sibling of same. This behavior will take a little to get used to, especially for long-time users of the Windows explorer. However, there is no other way to insert a node at a definite position. In the example shown in [figure 2.2^{\(12\)}](#) above, a new node would be inserted as the first child of the sequence called "Events", just before the Mouse event⁽⁴⁴⁷⁾.

2.2.3 Moving nodes

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

2.0+

Alternatively, nodes can be moved using standard Drag&Drop operations. The default operation will move the selected node(s). If the **CTRL** key is held down during the drop, the nodes are copied instead.

While dragging the mouse over the tree of a test-suite, the insertion marker shows where the nodes will be dropped when the mouse button is released and whether the operation is allowed. A green marker signals a legal operation, a red marker an illegal one. Nothing will happen if the nodes are dropped on an illegal target position.

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

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

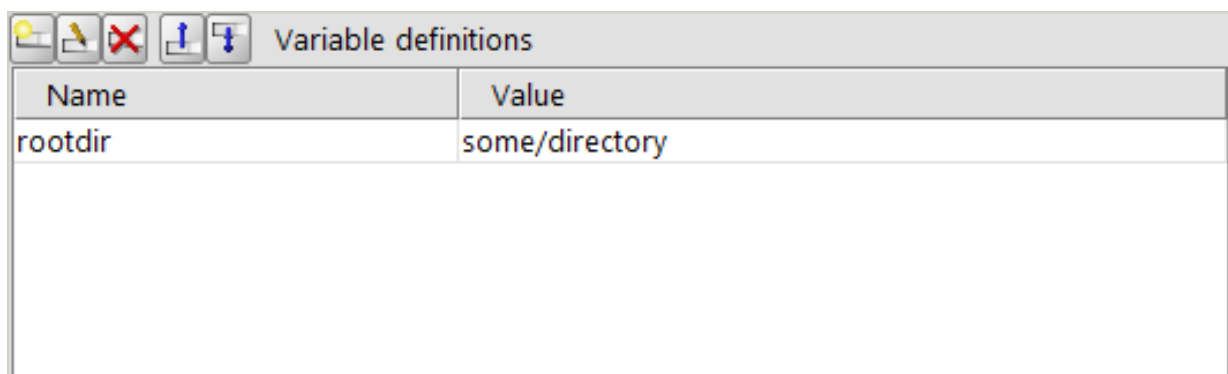
2.2.4 Transforming nodes

Some nodes can be transformed into different node types, which is a lot more convenient than first creating the desired target node and then copying over the required attributes. Examples of interchangeable nodes are [Sequence](#)⁽³³⁴⁾ and [Test](#)⁽³³⁰⁾ or [Server script](#)⁽⁴¹⁶⁾ and [SUT script](#)⁽⁴¹⁹⁾. The transformation of a node is possible only if its childnodes and its current position in the tree are also valid for the desired target node. The potential transformation targets can be selected from the entry **Transform node into** in the context menu. If the entry is not available there are no valid target nodes. In that case, moving the node to the [Extras](#)⁽³⁴²⁾ node first may help.

You can find more details about the conversion mechanism under [Details about transforming nodes](#)⁽⁷⁶⁶⁾.

2.2.5 Tables

In various places QF-Test employs tables to view and edit a set of values, e.g. when defining [variables](#)⁽⁵⁷⁾ or for [checks](#)⁽⁴⁶⁹⁾ of multiple elements.



Name	Value
rootdir	some/directory

Figure 2.3: Example table

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

**Shift-Insert**

Insert a new row.

**Shift-Return**, **Alt-Return**

Edit a row. Opens a dialog with fields for every cell of the selected row.

**Shift-Delete**

Delete the selected row.

**Shift-Ctrl-Up**

Move the selected row up by one.

**Shift-Ctrl-Down**

Move the selected row down by one.

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



Insert a new column.



Delete the selected column.



Edit the title of the selected column.

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

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

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

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

2.2.6 Packing and Unpacking

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

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

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

Note

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

2.2.7 Sorting Nodes

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

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

Note

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

2.3 Advanced editing

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

2.3.1 Searching

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

General search

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

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

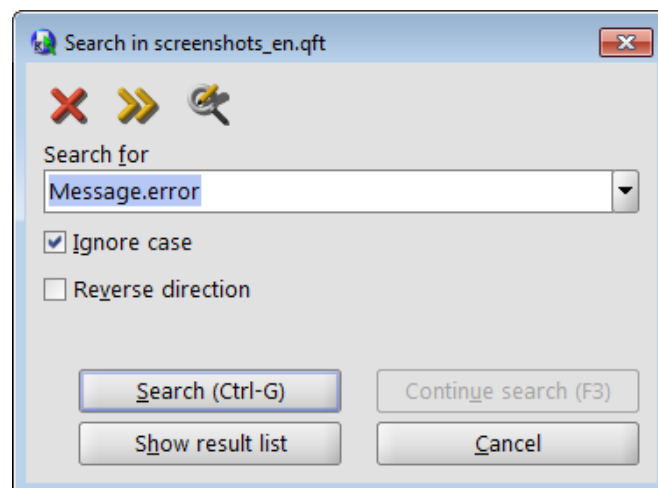


Figure 2.4: The simple search dialog

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

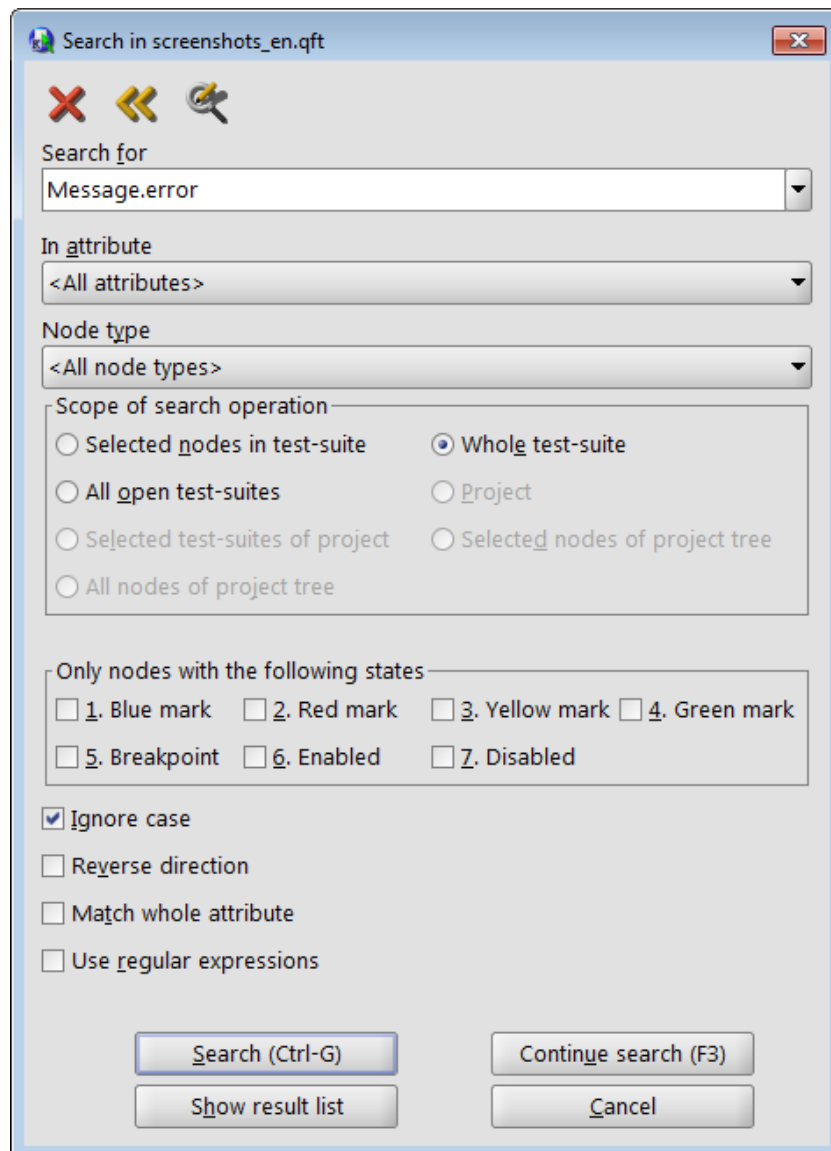


Figure 2.5: The advanced search dialog

By default QF-Test will search all attributes in all kinds of nodes for the requested string. Use the "In attribute" option to limit the search to a specific attribute and/or the "Node type" option to limit it to nodes of a specific kind. The other options should be self-explanatory. Regular expressions are explained in [section 36.4^{\(614\)}](#). If "Match whole attribute" is selected, a search for the word "tree", for example, will not match an attribute value of "treeNode". Activating options in "Only nodes with the following states" limits the search to nodes that have all of the activated states, e.g. a green mark and a breakpoint. The option "Scope of operation" tells QF-Test where to search for the given expression, below the selected node(s), in the current test-suite or in all currently opened suites.

Note

To search for values of boolean attributes like `Implicitly catch exceptions(333)`, use "true" or "false" (no quotes). If you want to search for an empty value you have to check "Match whole attribute".

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

3.4+

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

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

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

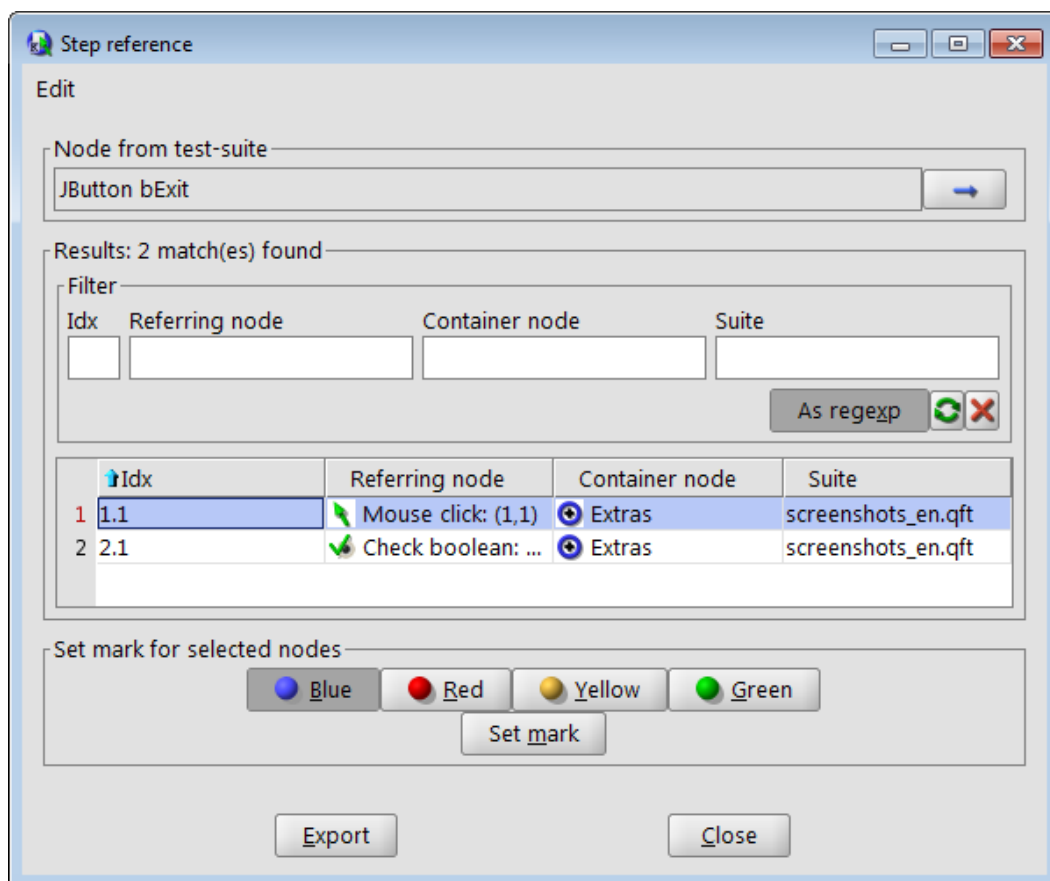


Figure 2.6: Result list for 'Locate references'

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

Incremental text search

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

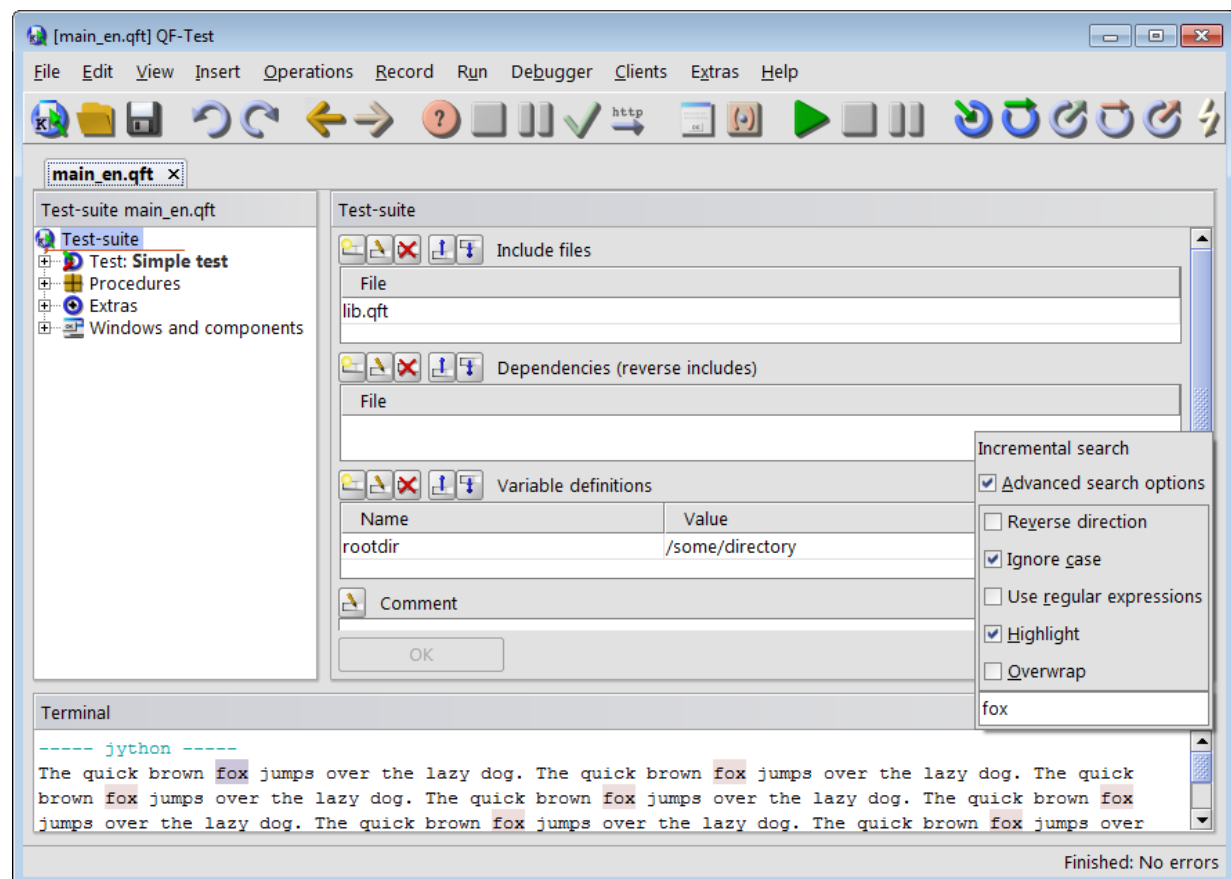


Figure 2.7: Incremental search

The search can be limited to a certain part of the contained text by selecting the region

to be searched and invoking **Search in selection...** from the component's context menu or pressing **Ctrl-Shift-F**.

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

2.3.2 Replacing

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

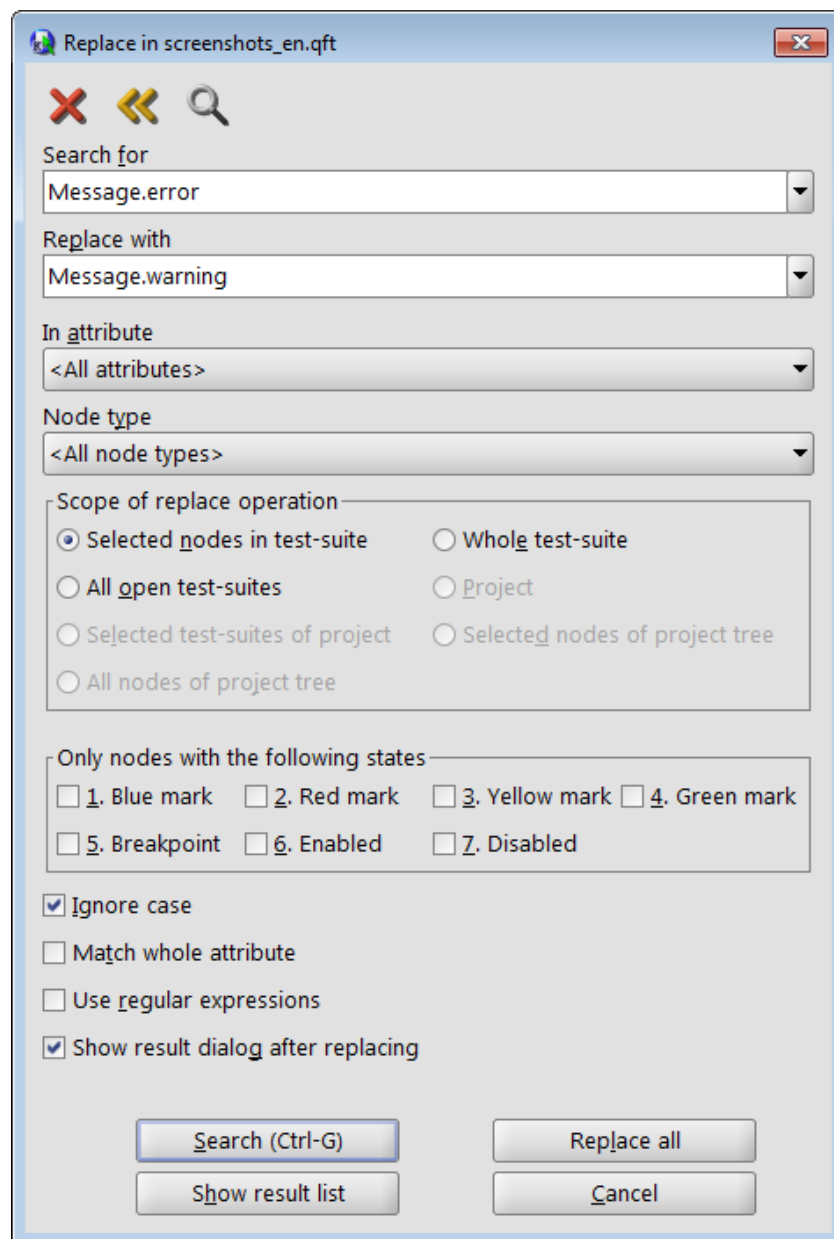


Figure 2.8: The replace dialog

The options are identical to the ones for searching. When "Match whole attribute" is turned off, multiple replacements within one attribute are possible. When replacing "a" with "b" for example, "banana" would change to "bbnbnb". Be sure to read [section 36.4^{\(614\)}](#) about how to use regular expressions for replacing.

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

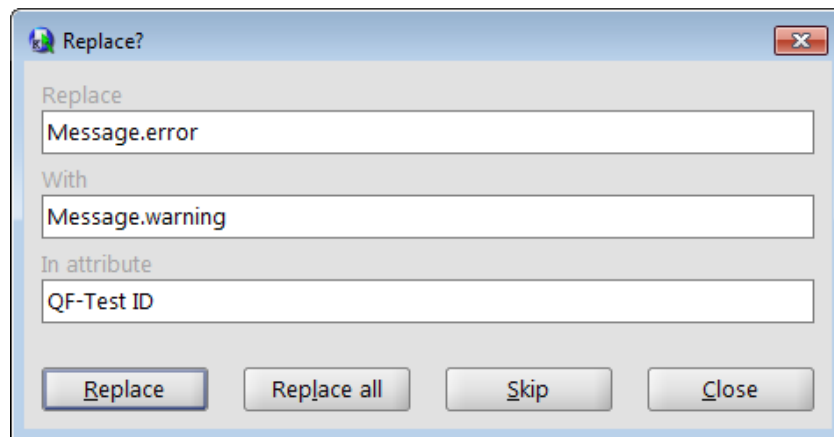


Figure 2.9: The replace query dialog

It offers the following choices:

- When *Replace* is selected, the attribute's value is changed and the search carries on, showing the query again for the next match.
- *Replace all* means change this value and all the rest of the matches in one go without asking again.
- *Skip* leaves this attribute unchanged. The search continues and the query dialog is shown again in case of another match.
- Obviously *Cancel* ends the replace operation.

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

3.1+

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

Note

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

2.3.3 Complex searches and replace operations

3.3+

Sometimes a simple search is not enough. Imagine, for example, that you want to set a Timeout of 3000 milliseconds for all text checks on a certain component. You know the

component's QF-Test ID, but if you search for that QF-Test ID you will also find events and other kinds of checks referencing it. And if you search for the node text 'Check text' you will find all Check text nodes, not just those for the given component.

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

2.3.4 Multiple views

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

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

Chapter 3

Quickstart your application

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

In order to make you application recognized by QF-Test as SUT it basically needs to be started out of QF-Test. There are a number of special process nodes available within the `Insert→Process nodes` to perform this task but the straight forward way is to use the Quickstart Wizard as described below. For those with an aversion to wizard dialogs, the manual way is explained at [section 34.1^{\(592\)}](#).

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

Swing

For Swing/JavaFX or combined Swing/JavaFX and SWT applications QF-Test hooks into the Java's JVM Tool Interface. Normally QF-Test can do this directly. Only for some non-standard JDKs it may be necessary to instrument those first. See [JDK instrumentation^{\(604\)}](#) for details if necessary.

JavaFX

For JavaFX applications and respective combinations the connection works exclusively via the QF-Test agent. Please ensure the option [Connect via QF-Test agent^{\(275\)}](#) is activated.

SWT

For Eclipse/SWT based applications, an instrumentation of the SWT library is necessary. The Quickstart Wizard, which is described below, will support you in the task to add the necessary step to the setup sequence. Background information can be found in [section 35.3^{\(608\)}](#).

Web

Web application testing does not require instrumentation but there are some restrictions to notice. On Windows QF-Test only supports only 32bit browsers which need to run on a 32bit JDK. If QF-Test itself does not run on a 32bit JDK you can specify an alternative one for starting the browser. Internet Explorer and Google Chrome are automatically run in 32bit, for Firefox please check your installed version. On Linux 32bit and 64bit versions of Firefox are supported, you just need to make sure that the JDK and the Browser are of the same architecture.

32bit Firefox releases can be downloaded from <http://releases.mozilla.org/pub/mozilla.org/firefox/releases/>.

32bit JDKs can be downloaded from <http://www.java.com>. Chapter [Installation and startup](#)⁽²⁾ describes how to configure the JDK to be used by QF-Test.

3.1 Quickstart Wizard

3.1.1 Setup sequence creation

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

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

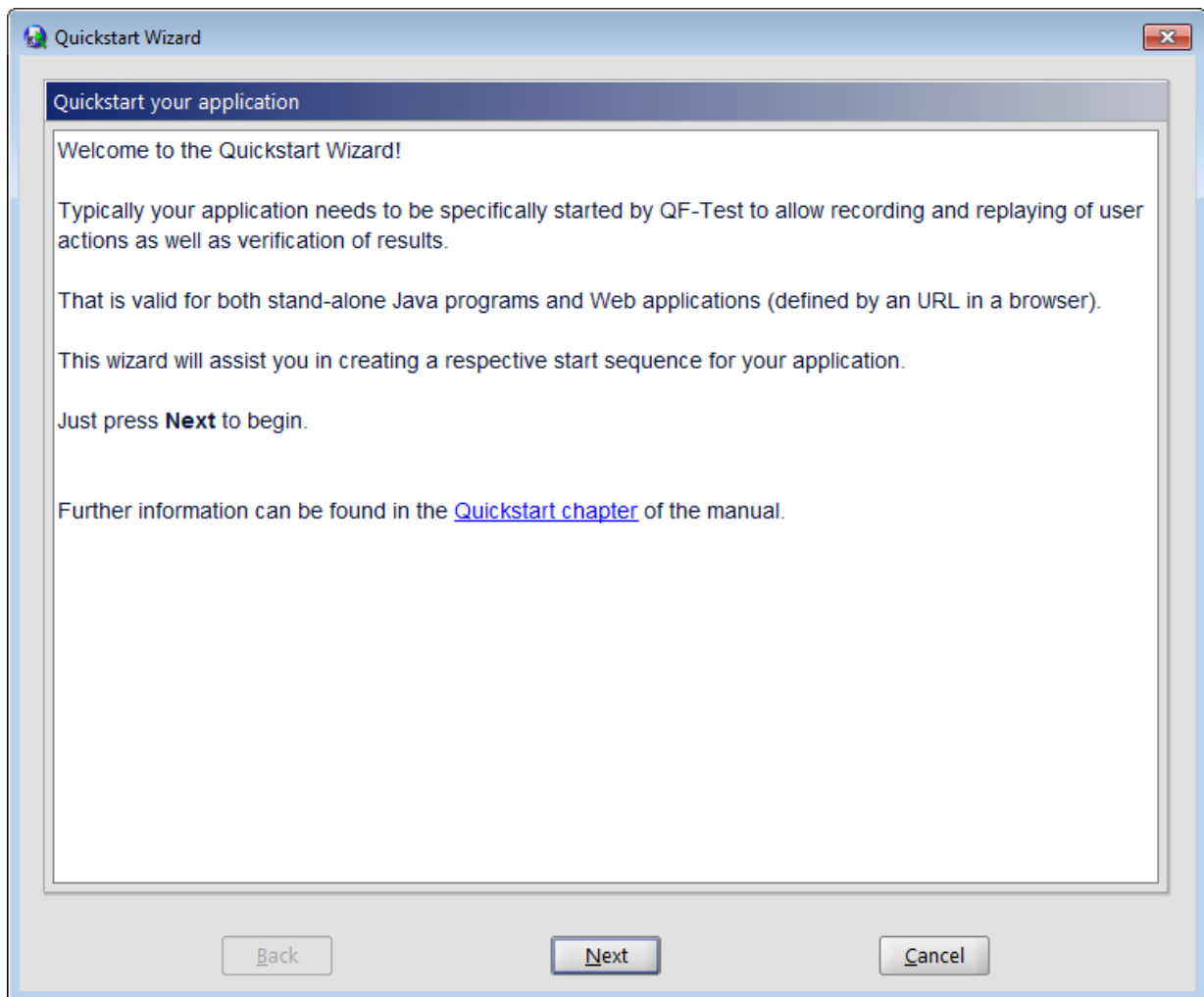


Figure 3.1: Quickstart Wizard

As result the Wizard delivers a startup sequence under the "Extras", as shown in the following figure:

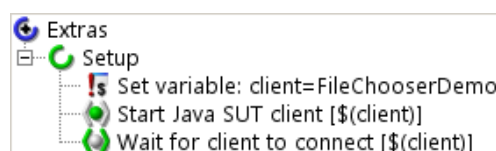


Figure 3.2: Startup sequence created by the Quickstart Wizard

The setup sequence contains a start node that may vary in type and attributes dependent on your application. The start node is followed by a "Wait for client" node that

ensures QF-Test connects to your application during the startup procedure. (Details about the different start node types and attributes can be found in [section 34.1^{\(592\)}](#).)

Swing
JavaFX For Swing or JavaFX based applications the setup sequence typically is straight forward except for Applet testing in a browser which uses a special procedure from the [standard library^{\(88\)}](#).

SWT For SWT based applications an additional procedure call node for [SWT instrumentation^{\(608\)}](#) is added.

Web The standard startup sequence for web includes some additional nodes for setting variables, initializing browser cache and cookie settings and possibly install an AJAX toolkit resolver.

3.1.2 Executing the setup sequence

The setup sequence is either directly started by the Quickstart Wizard after creation or can be executed by selecting the green setup sequence node in the tree and pressing "Replay" toolbar button .

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

Now you are able to record and replay your first test sequences as described in [chapter 4^{\(30\)}](#). There is also a learning by doing tutorial available from the QF-Test help menu which guides you through all features of QF-Test.

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

3.1.3 Troubleshooting

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

- The error dialog QF-Test typically displays should provide a first indication.
- Please look for error messages in the terminal window. If there is no terminal window visible in the bottom area of QF-Test, it can be activated through the menu item [View→Terminal→Show](#). Additional information about program output can be found in [section 3.2^{\(29\)}](#).

- Be sure to double-check the attribute values in the setup sequence nodes are correct. Possibly a typo has crept in somewhere. Details about the different start node types and attributes can be found in [section 34.1^{\(592\)}](#).

Swing

- If you are testing an Applet (possibly on Windows 8) please see [section 34.1.3^{\(595\)}](#) for additional information.

Web

- As browser development cycles i.e. those of Firefox tend to shorten, be sure the installed browser is supported by the QF-Test version you are using. The terminal output should show a respective error message. See [release notes^{\(830\)}](#) in appendix B for latest supported browser versions. Possibly you need to update QF-Test to a later version or temporarily use another browser.

Web

- If the error dialog indicates that no 32bit browser and/or 32bit JDK is used, please refer to the beginning of this chapter.

If the SUT gets visible but QF-Test is not able to connect to the client (`ClientNotConnectedException(562)`):

- Please double-check the terminal output content (see also above) for possible error messages.
- In case the case the red record button in the toolbar gets activated after the error message occurred, the timeout value in the [Wait for client to connect^{\(438\)}](#) node needs to be increased.

Swing

- Please ensure the option [Connect without JDK instrumentation \(Swing\)^{\(275\)}](#) is activated. If you are using a non-standard JDK please check out the [section 35.1^{\(604\)}](#) chapter.

Swing

- If you are testing an Applet, please see [section 34.1.3^{\(595\)}](#) for additional information. There are a number of special cases to consider.

JavaFX

- For JavaFX applications please ensure the option [Connect via QF-Test agent^{\(275\)}](#) is activated.

SWT

- For an Eclipse/SWT application first make sure that you specified the correct application directory. You may want to take a look at the run-log (see [section 8.1^{\(68\)}](#)) to see if any warnings or errors were logged during execution of the Procedure `qfs.swt.instrument.setup`.
- Check the run-log in general for possible additional error indications (see [section 8.1^{\(68\)}](#)).

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

3.2 Program output and the Clients menu

The standard output and error streams of all processes started by QF-Test are captured and stored in the run-log under the node that represents the respective starter node. In this QF-Test does not distinguish between SUT clients and arbitrary processes or shell scripts started with a Start process⁽⁴³³⁾ or Execute shell command⁽⁴³⁶⁾ node.

The main window contains a shared terminal view that shows the output of all processes started by a test that was run from this window. The View→Terminal sub-menu holds items to configure whether this terminal is visible, whether long lines are wrapped and whether it is automatically scrolled to the end when new output arrives. Other items let you clear the terminal or save its contents to a file. The maximum amount of text that the terminal holds is configurable in the option Maximum size of shared terminal (kB)⁽²⁷⁷⁾.

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

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

The number of terminated clients that are kept in the Clients menu is set with the option Number of terminated clients in menu⁽²⁷⁴⁾. If your processes generate lots of output and you are low on memory you may want to reduce that number.

3.3 An application started indirectly from an already connected SUT

If a second Java VM is started from an already connected SUT, QF-Test will recognize this as an indirect connection attempt from a child process of the first SUT and automatically assign an artificial client name to this new SUT. The name is created by appending ':2' to the client name of the parent SUT, signifying that this is the second process for this client. Yet another Java VM started by either of these SUTs would get ':3' appended to the original client name unless the second process was already terminated so the ':2' was again free for use.

To summarize, the sequence for connecting to an indirectly started SUT typically consists of an event node that triggers something like a button click or menu selection, causing the SUT to launch the second SUT, followed by a Wait for client to connect⁽⁴³⁸⁾ node for the ':2' extended client name.

Chapter 4

Capture and replay

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

4.1 Recording sequences

In order to record a sequence of events in the SUT, the SUT must have been run from QF-Test (see [chapter 3^{\(24\)}](#)) and the connection between QF-Test and the SUT must be established. A visual indicator of this is the color of the record button which turns red when it is enabled.



Figure 4.1: Disabled and enabled Record button

To record a sequence, simply start recording by pressing the record button  or selecting **Record→Start**. Then switch to the SUT, execute a few commands, switch back to QF-Test and stop the recording with the stop button  or **Record→Stop**. The recorded events will be added to the test-suite, either directly at the position indicated by the [insertion marker^{\(12\)}](#) or as a new [Sequence^{\(334\)}](#) under the [Extras^{\(342\)}](#) node, depending on the setting of the [Insert recording at current selection^{\(254\)}](#) option. Pause the recording with **Record→Pause** or the pause button  if you need to execute some steps in the SUT that should not be recorded and you don't want to stop and restart the recording.

Recording mode can be started and stopped directly in the SUT by use of the Hotkey for recording⁽²⁵⁴⁾. Default key is **[F11]**.

Any Components⁽⁵⁴⁵⁾ referred to by the newly recorded events are added automatically to the Windows and components⁽⁵⁵⁶⁾ node if they are not there already.

There are many options that influence the way QF-Test records events and how it treats the components of the GUI. All of these are explained in detail in section 29.2⁽²⁵⁴⁾ of the reference manual. Once you are familiar with QF-Test you should take the time to skim through it.

Here's some general advice for recording:

- Record short sequences at a time.
- After recording, take a look at the sequence, try to understand what you got and whether it represents the actions you took.
- Edit the sequence to remove unnecessary events, especially those at the beginning and end caused by switching windows. QF-Test has excellent filters that should catch nearly all of these, but some might remain and have to be removed manually.
- Finally, try out the new sequence to see whether it replays OK. Then you can cut/copy/paste as needed to integrate it into larger parts.

Mac

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

4.2 Running tests

To run some tests, select the node or nodes to execute and press **[Return]** or the play button  or select **[Run→Start]**. QF-Test will mark each node with a small arrow as it is executed and also show progress messages in the status bar. This can slow down execution a little and can be turned off with the options Mark nodes during replay⁽²⁷¹⁾ and Show replay messages in status line⁽²⁷¹⁾.

When the test is finished, the result is shown in the status bar. If things are fine you should see "No errors", otherwise the number of warnings, errors and exceptions is

shown. Additionally, a message dialog is shown in case of errors or exceptions to make sure you don't miss these.

As for recording there are many options that influence the replay of tests. Some of these are only for convenience while others have a major impact on the outcome of the tests. Be sure to read [section 29.3^{\(270\)}](#) some time to familiarize yourself with these.

To abort execution before the test is finished, press the stop button  or select [Run→Stop](#). You can also suspend execution temporarily via the pause button  or by selecting [Run→Pause](#). This will also enable the debugger (see [chapter 8^{\(68\)}](#)). To continue, press pause again.

While a test is run at full speed it can be tricky to stop or interrupt it, especially when the mouse cursor is actually moved across the screen or the SUT's windows are raised on every event. To regain control, press the (Don't) Panic key⁽²⁷⁰⁾ (the default is [Alt-F12](#)). This will pause all running tests immediately. To continue, press the same combination again.

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

The current error state during replay as well as the final result is shown in the status line at the bottom of the QF-Test main window. The visibility of this status line can be controlled via [View→Show status line](#).

In case [Test-set^{\(321\)}](#) or [Test-case^{\(314\)}](#) nodes ([section 9.2^{\(80\)}](#) describes their usage) are executed the status line also contains relevant result counters from the following list.

Counter Icon	Description
#	Total number of test-cases. This counter value starts with a '>' symbol in case there are skipped test-sets.
!	Number of test-cases with exceptions.
-	Number of test-cases with errors.
⊖	Number of test-cases with expected errors. <u>Expected to fail if...</u> ⁽³¹⁸⁾ marks a test-case expected to fail.
+	Number of successful test-cases.
»	Number of skipped test-cases. A test-case is skipped when its (optional) <u>Condition</u> ⁽³¹⁸⁾ fails. This counter value starts with a '>' symbol in case there are skipped test-sets.
»	Number of skipped test-sets. A test-set is skipped when its (optional) <u>Condition</u> ⁽³²⁵⁾ fails.
⊘	Number of not implemented test-cases. A test-case is not implemented when it doesn't contain nodes that were executed during the test-run.
▶	Number of executed test-cases.
%	Percent test-cases passed.

Table 4.1: Test result counter in the status line

The final test result counts also appear in the report which can be created for any test run. Reports are discussed in [chapter 15](#)⁽¹²⁹⁾.

Note

The counter values above can also be accessed as variables⁽⁶²⁾ during the test-run. A TestRunListener⁽⁷²⁸⁾ can help to keep track of counter values and trigger dependent actions.

4.3 Recording checks

Though it can be quite entertaining to record sequences and watch the magic dance of the SUT as they are played back, the task at hand is to find out whether the SUT actually works as expected. This is where checks⁽⁴⁶⁹⁾ come into play. The most common check, the Check text⁽⁴⁷⁰⁾ node, reads the text displayed by a component, e.g. a text field, and compares it to a given value. If the values differ, an error is signaled.

How checks work is explained in detail in the Reference manual⁽⁴⁶⁹⁾. There is a range available from simple text check to advance Check image⁽⁴⁸⁸⁾ and even custom check types⁽⁷¹⁵⁾ can be implemented. Here we are going to concentrate on the most convenient way to create checks, which is to record them.

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

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

There are different kinds of checks⁽⁴⁶⁹⁾ that can be performed. Which kinds of checks are applicable depends on the selected component. Some components don't display any text, so a *Check text*⁽⁴⁷⁰⁾ node doesn't make sense for, say, a scroll bar. Clicking on a component with the right mouse button while in check mode brings up a menu of applicable checks for this component. Select one of the items to create the respective check node. Clicking with the left mouse button always records the default check, which is the topmost one in the popup menu.

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

Checks integrate well with events and you'll soon develop a recording style a'la *click, click, type, click, check, click, click, check...* Having to switch back and forth between QF-Test and the SUT every time you want to create a check can be a real pain. That is where the *Hotkey for checks*⁽²⁵⁵⁾ option comes into play. It defines a key which toggles the SUT between record mode and check mode. The default value is **[F12]**, but if this key has some defined meaning for your application you can change it to whatever you like. To record a sequence of interspersed events and checks, simply start recording, switch to the SUT and record the sequence. Whenever you want to add some checks, just press **[F12]** (or whatever you defined), record the checks, then switch back to record mode by pressing **[F12]** again and continue. This way you can work with the SUT for the whole sequence and need to switch back to QF-Test only to stop the recording.

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

4.4 Recording components

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

To activate *component recording mode* you simply need to press the record components button  or select **[Record→Record components]** from the menu. Then switch to the SUT window where you will notice a special behavior in which the component below the mouse cursor is highlighted.

Clicking with the left mouse button on a component will record the single component while pushing the right button instead will pop up a menu with choices to record the nested components as well or all components in the window. Multiple components can be captured in this way. Then switch back to QF-Test and release the record components button  or deactivate **Record→Record components**. Now the recorded component information is stored in shape of respective Component⁽⁵⁴⁵⁾ nodes under the Windows and components⁽⁵⁵⁶⁾ node.

Component recording mode can be alternatively controlled by a configurable hotkey. Default binding is **F11**, the option that configures it is Hotkey for components⁽²⁶¹⁾.

Pressing **F11** (default setting) *in the SUT window* starts component recording. Further details can be found in the documentation for the option Hotkey for components⁽²⁶¹⁾.

Note

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

The component recording feature can also be used to quickly locate a component independent of whether it has been recorded before. When creating event or checks by hand or changing the target component, the QF-Test component ID⁽⁴⁴⁸⁾ attribute needs to be specified. When a component is recorded, its QF-Test ID is saved in the clipboard and can be pasted directly into the QF-Test component ID field with **Ctrl-V**. You can also jump directly to the Component node with **Shift-Ctrl-Backspace** or by choosing the **Edit→Select next node** menu item or clicking the respective toolbar button.

The popup menu which appears when clicking with the right mouse button on a component also contains an entry **Show methods** which allows of some kind of component inspection (see section 5.5⁽⁴⁴⁾).

Components play a central role in the structure of a test-suite which is explained further in chapter 5⁽³⁷⁾.

4.5 Recording of HTTP Requests (GET/POST)

Web

In order to record a (GET/POST) request sent by the SUT, the SUT must have been launched from QF-Test (see chapter 3⁽²⁴⁾) and the connection between QF-Test and the SUT must be established.

While recording, the SUT is in *record mode*, which means that all events are collected and sent to QF-Test. With the help of the HTTP-request button  or by selecting

Record→Record HTTP Requests you can bring it into *request recording mode*. In contrast to Recording sequences⁽³⁰⁾ all GET/POST-request send by the web browser are saved as http-request nodes in this special recording mode. To get back to record mode, select the HTTP-request button or menu item again.

In section Web options⁽²⁹⁴⁾ the ability to change the type of the recorded request is described. By default Browser HTTP request⁽⁵³²⁾ is recorded. This Request type is likely used to automate large web form inputs, the use of separate input nodes will be avoided. The form data will be submitted within the browser, so that the response will be shown as well. At this point the test execution could be continued. In opposition to this the Server HTTP request⁽⁵²⁸⁾ will be directly submitted through QF-Test without the need of a running browser. The response is also only available in QF-Test and a eventually running browser will not be affected.

All attributes of an accordingly recorded request node as well as the parametrization of requests are explained in detail in the HTTP Requests⁽⁵²⁸⁾ section of the reference part of this manual.

Chapter 5

Components

Though they often go unnoticed, at least until the first `ComponentNotFoundException`⁽⁵⁵⁸⁾ occurs, the `Component`⁽⁵⁴⁵⁾ nodes are the heart of a test-suite. Everything else revolves around them. Explaining this requires a little side-tracking:

5.1 GUI component hierarchy

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

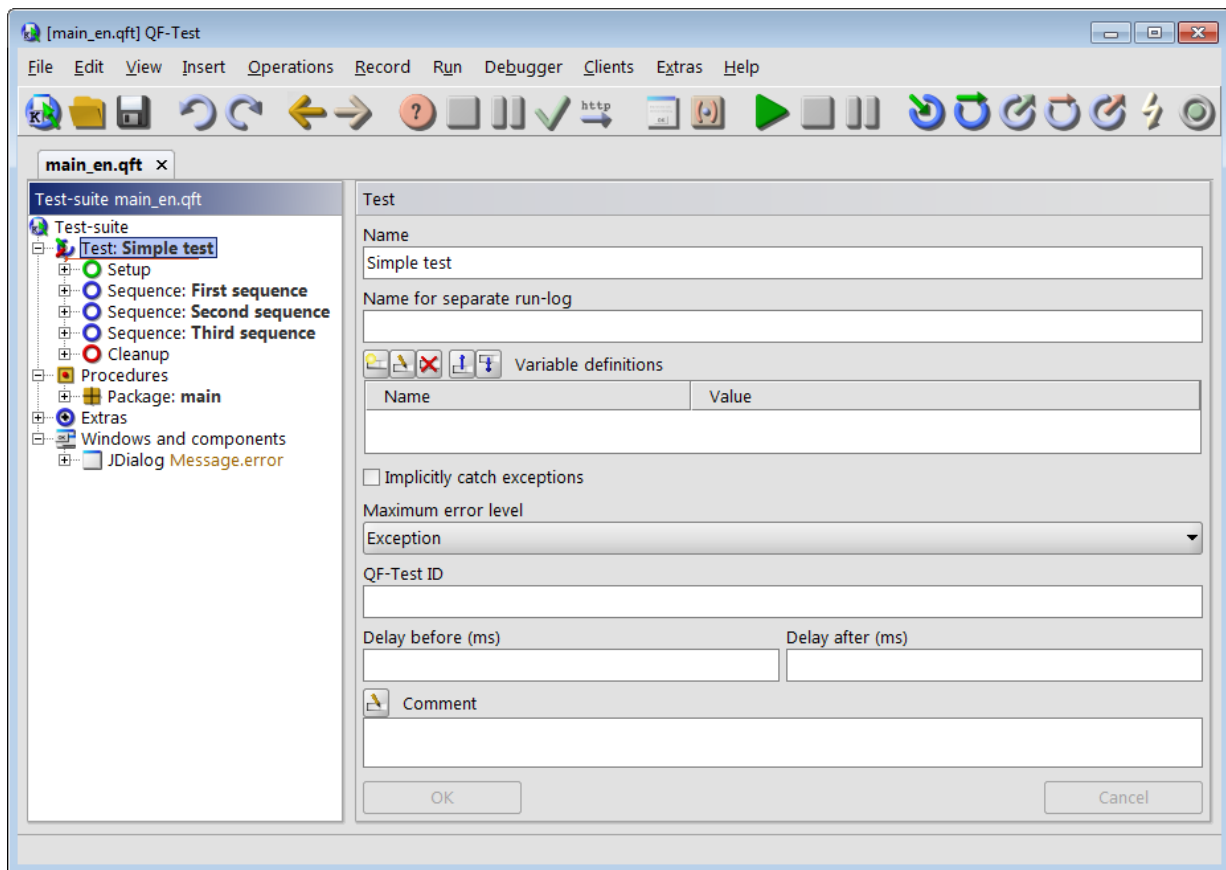


Figure 5.1: Components of a GUI

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

SWT

In SWT the main GUI components are called `Control`, `Widget` or `Item`. Unless explicitly stated otherwise the term "component", as used in this manual, also applies to these and not only to AWT/Swing/JavaFX `Components`.

JavaFX

The same is valid for JavaFX components called `Nodes` that build up the component hierarchy denominated as `Scene graph`.

Web

The internal representation of an HTML page is based on the *Document Object Model*

(DOM) as defined by the W3C, a tree structure consisting of nodes. The root node, a `Document` can contain `Frame` nodes with further `Document` nodes and/or a root `Element` with a tree structure of further `Element` nodes. Though an HTML page with its DOM is quite different from a Swing, JavaFX or SWT interface, the abstractions QF-Test uses work just as well and the general term "component" also applies to DOM nodes.

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

This may sound trivial and obvious, but component recognition is actually the most complex part of QF-Test. The reason for this is the necessity to allow for change. QF-Test is a tool designed for regression testing, so when a new version of the SUT is released, tests should continue to run, ideally unchanged. So when the GUI of the SUT changes, QF-Test needs to adapt. If, for example, the "OK" and "Cancel" buttons were moved from the bottom of the detail view to its top, QF-Test would still be able to replay events for these buttons correctly. The extent to which QF-Test is able to adapt varies and depends on the willingness of developers to plan ahead and assist a little bit in making the SUT well-suited to automated testing. But more on that later ([section 5.6^{\(44\)}](#) and [section 5.7^{\(46\)}](#)).

5.2 Components in QF-Test

The recorded components are transformed into `Window(535)` and `Component(545)` nodes which form a hierarchy that represents the actual structure of the GUI. These nodes are located under the `Windows and components(556)` node. The following image shows part of the Components representing QF-Test's main window.

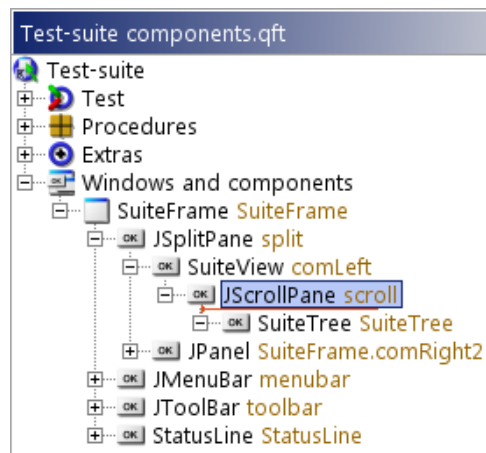


Figure 5.2: Component hierarchy of a Swing SUT

Instead of a Window, the root Document of a web page is represented as a Web page⁽⁵⁴⁰⁾ node. Nested Documents inside Frames are represented as Component nodes.

Every time a sequence is recorded, nodes are generated for components that are not yet represented. When the sequence is discarded later on, the Components remain, hence Component nodes have a tendency to proliferate. The popup menu (right button click) for Window and Component nodes has two items, Mark unused components... and Remove unused components, which will mark or remove those Component nodes that are no longer being referred to. Be careful though if you are referencing Components across test-suite boundaries or use variable values in QF-Test component ID attributes as these are not taken into account unless the test-suites belong to the same project or the Dependencies (reverse includes)⁽³¹²⁾ attribute of the Test-suite root node is set correctly .

Besides this way of representing components as nodes it is also possible to address components as multilevel sub-items with an XPath-like syntax called QPath as explained in section 6.4⁽⁵⁵⁾

The attributes of Components and the algorithm for component recognition are explained in detail in section 36.2⁽⁶¹²⁾. Here we will concentrate on the association between Component nodes and the rest of the test-suite.

5.3 QF-Test IDs

Every node of the test suite has a QF-Test ID attribute which is secondary for most kinds of nodes. For Component nodes however, the QF-Test ID⁽⁵⁴⁶⁾ has an important function. It is the unique identifier for the Component node by which events, checks and other nodes that have a target component refer to it. Such nodes have a QF-Test component ID

Web

Note
4.0+

attribute which is set to the Component's QF-Test ID. This level of indirection is important. If the GUI of the SUT changes in a way that QF-Test cannot adapt to automatically, only the Component nodes for the unrecognized components need to be updated to reflect the change and the test will run again.

It is essential to understand that the Component's QF-Test ID is an artificial concept for QF-Test's internal use and should not be confused with the Name attribute, which serves for identifying components in the SUT and is explained in detail in the following section. The actual value of the QF-Test ID is completely irrelevant, except for the requirement to be unique, and it bears no relation whatever to the actual component in the GUI of the SUT. However, the QF-Test ID of the Component is shown in the tree of the test-suite, for Component nodes as well as for events and other nodes that refer to a Component. For this reason, Components should have expressive QF-Test IDs that allude to the actual GUI component.

When creating a Component node, QF-Test has to assign a QF-Test ID automatically. It does its best to create an expressive value from the information available. The option Prepend parent QF-Test ID to component QF-Test ID⁽²⁶⁶⁾ controls part of this process. If the generated QF-Test ID doesn't suit you, you can change it. QF-Test will warn you if you try to assign a QF-Test ID that is not unique and if you have already recorded events that refer to the Component, it will change their QF-Test component ID attribute to reflect the change. Note that this will not cover references with a variable QF-Test component ID attribute.

Note

A common mistake is changing the QF-Test component ID attribute of an event instead of the QF-Test ID itself. This will break the association between the event and the Component, leading to an UnresolvedComponentIdException⁽⁵⁶³⁾. Therefore you should not do this unless you want to change the actual target component of the event.

Experienced testers with a well-structured concept for automated testing will find the component recording feature described in section 4.4⁽³⁴⁾ useful. It can be used to record the component hierarchy first in order to get an overview over the structure of the GUI and to assign QF-Test IDs that suit you. Then you can continue to record the sequences and build the test-suite around the components.

5.4 Component class

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

Besides its role in component identification, the class of a component is also important for registering various kinds of resolvers that can have great influence on the way QF-Test handles components. Resolvers are explained in detail in [section 39.1^{\(678\)}](#).

5.4.1 Generic classes

4.0+

Each toolkit defines its own system-specific classes for components like Buttons or Tables. In case of Buttons, that definition could be `javax.swing.JButton` for Java Swing or `org.eclipse.swt.widgets.Button` for Java SWT or `javafx.scene.control.ButtonBase` for JavaFX or `INPUT:SUBMIT` for web applications. In order to allow your tests to run independently of the actually utilized technology QF-Test unifies those classes via so-called generic classes, e.g. all buttons are simply called `Button` now. This approach provides a certain degree of independence from the dedicated technical classes and will allow you to create tests without taking care about the specific technology. You can find a detailed description of generic classes at [chapter 46^{\(790\)}](#). In addition to generic classes QF-Test records system-specific classes as [Extra features^{\(547\)}](#) with the state "Ignore". In case of component recognition problems due to too many similar components these can be activated to have a stricter component recognition at the expense of flexibility.

Another reason for generic classes is that dedicated technical classes could get changed during development, e.g. due to introduction of a new base framework or even another technology. In such cases QF-Test needs to be quite flexible in order to recognize a proper class. Here the concept of generic classes allows you to be able to cope with those changes and for the most part to re-use existing tests. You can find more details at [section 5.4.3^{\(43\)}](#).

5.4.2 Class hierarchy for web applications

Web

For Swing, FX and SWT QF-Test works with the actual Java GUI classes whereas a pseudo class hierarchy is used for web applications as follows:

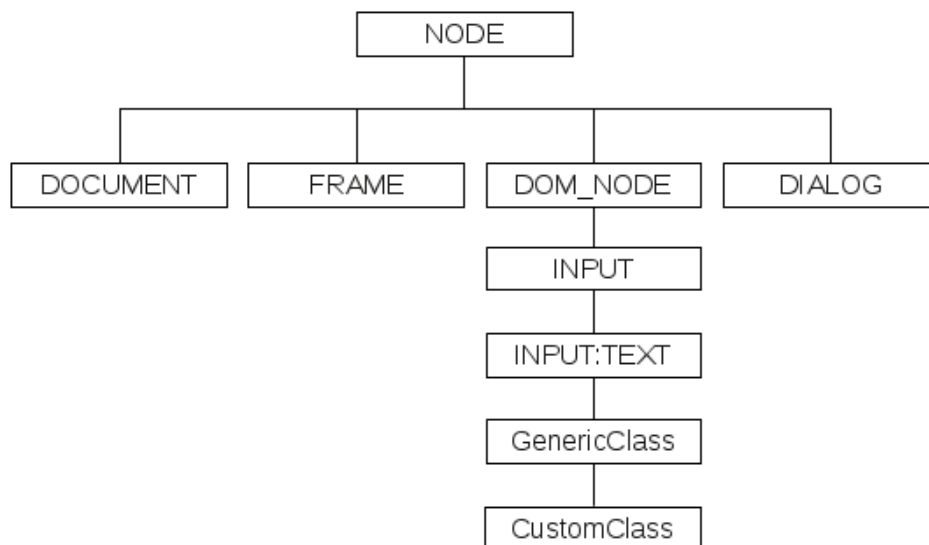


Figure 5.3: Pseudo class hierarchy for web elements

As shown, "NODE" is at the root of the pseudo class hierarchy. It matches any kind of element in the DOM. Derived from "NODE" are "DOCUMENT", "FRAME", "DOM_NODE" and "DIALOG", the types of nodes implementing the pseudo DOM API explained in section 38.1⁽⁶⁴⁷⁾. "DOM_NODE" is further sub-classed according to the tag name of the node, e.g. "H1", "A" or "INPUT" where some tags have an additional subclass like "INPUT:TEXT".

5.4.3 Settings for class recording

QF-Test can record classes of component in various ways, therefore it organizes component classes in various categories. Those categories are called as the specific class, the technology-specific system class, the generic class and the dedicated type of the generic class. Each category is recorded at Extra features⁽⁵⁴⁷⁾.

The option Record generic class names for components⁽²⁶³⁾ is checked by default. Using this option allows you to record generic classes in order to share and re-use your tests when testing a different technology with just minor changes to the existing tests.

In case you work with one Java engine only and you prefer to work with the "real" Java classes, you could also work without the generic class recording. But in this case you should consider to check the option Record system class only⁽²⁶³⁾. This option makes QF-Test to record the technology-specific system class instead of the derived class. If you switch off this option you will get the derived class which enables you to make a very well targeted recognition but could cause maintenance efforts in case of changes coming from refactoring.

Web

In web applications QF-Test records classes as described in the previous chapter section 5.4.2⁽⁴²⁾. In case you have to work with a supported AJAX toolkit (see section 38.2⁽⁶⁵⁷⁾), QF-Test records generic classes as well. You shouldn't modify the default options for this technology.

5.5 Component inspection

3.1+

Depending on its class a component has a set of (public) methods and fields which can be used in an SUT script⁽⁴¹⁹⁾ once you have a reference to the object (see section 13.3.7⁽¹⁰⁶⁾). Select the entry **Show the component's methods...** from the context menu of a node under the Windows and components⁽⁵⁵⁶⁾ branch to display the methods and fields of the corresponding class or right click on a component in the SUT while you are in component recording mode (see section 4.4⁽³⁴⁾).

Web

The methods and fields displayed for (HTML) elements in a browser cannot be used directly with an object returned by `rc.getComponent()`. These are at Javascript level and require a wrapping of the method calls into `evalJS` (cf. section 38.1⁽⁶⁴⁷⁾).

5.6 The importance of naming components

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

Note

Please note that recorded names are stored in the Name⁽⁵⁴⁷⁾ attribute of Component nodes. Because they also serve as the basis for the QF-Test ID for the same node, Name and QF-Test ID are often identical. But always keep in mind that the QF-Test ID is used solely within QF-Test and that the Name is playing the critical part in identifying the component in the SUT. If the name of a component changes, it is the Name attribute that must be updated, there is no need to touch the QF-Test ID.

The technique to use for setting names during development depends on the kind of SUT:

Swing

All AWT and Swing components are derived from the AWT class `Component`, so its method `setName` is the natural standard for Swing SUTs and some developers make good use of it even without test automation in mind, which is a great help.

JavaFX

For JavaFX `setId` is the pendant of Swing's `setName` method to set identifiers for components (called 'Nodes'). Alternatively IDs can be set via the FXML attribute `fx:id`. While the ID of a 'Node' should be unique within the scene graph, this uniqueness is not enforced. This is analogous to the 'ID' attribute on an HTML element.

SWT

Unfortunately SWT has no inherent concept for naming components. An accepted standard convention is to use the method `setData(String key, Object value)` with the String `"name"` as the key and the designated name as the value. If present, QF-Test will retrieve that data and use it as the name for the component. Obviously, with no default naming standard, very few SWT applications today have names in place, including Eclipse itself.

Fortunately QF-Test can derive names for the major components of Eclipse/RCP based applications from the underlying models with good results - provided that IDs were specified for those models. See the [Automatic component names for Eclipse/RCP applications](#)⁽²⁶⁵⁾ option for more details.

Web

The natural candidate for naming the DOM nodes of a web application is the 'ID' attribute of a DOM node - not to be confused with the QF-Test ID attribute of QF-Test's Component nodes. Unfortunately the HTML standard does not enforce IDs to be unique. Besides, 'ID' attributes are a double-edged sword because they can play a major role in the internal Javascript operations of a web application. Thus there is a good chance that 'ID' attributes are defined, but they cannot be defined as freely as the names in a Swing, JavaFX or SWT application. Worse, many DHTML and Ajax frameworks need to generate 'ID' attributes automatically, which can make them unsuited for naming. The option [Turn 'ID' attribute into name where "unique enough"](#)⁽²⁹⁴⁾ determines whether QF-Test uses 'ID' attributes as names.

Web

In case you want to test a web application using a supported AJAX toolkit, please take a look at [section 38.2.3](#)⁽⁶⁶⁴⁾ for details about assigning IDs.

If developers have implemented some other consistent naming scheme not based on the above methods, those names can still be made accessible to QF-Test by implementing a `NameResolver` as described in [section 39.1](#)⁽⁶⁷⁸⁾.

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

changes are significant or small changes have accumulated, component recognition will fail and manual intervention will be required to update the test-suite.

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

5.7 Considerations for setting names

There is one critical requirement for names: They must not change over time, not from one version of the SUT to another, not from one invocation of the SUT to the next and not while the SUT executes, for example when a component is destroyed and later created anew. Once a name is set it must be persistent. Unfortunately there is no scheme for setting names automatically that fulfills this requirement. Such schemes typically create names based on the class of a component and an incrementing counter and invariably fail because the result depends on the order of creation of the components. Because names play such a central role in component identification, non-persistent names, specifically automatically generated ones, can cause a lot of trouble. If development cannot be convinced to replace them with a consistent scheme or at least drop them, such names can be suppressed with the help of a `NameResolver` as described in [section 39.1^{\(678\)}](#).

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

Global uniqueness of names is also not required. Each class of components has its own namespace, so there is no conflict if a button and a text field have the same name. Besides, only the names of components contained within the same window should be unique because this gives the highest tolerance to change. If your component names are unique on a per-window basis, set the options `Name override mode (replay)(285)` and `Name override mode (record)(264)` to "Override everything". If names are not unique per window but identically named components are at least located inside differently named ancestors, "Hierarchical resolution" is the next best choice for those options.

Two questions remain: Which components should have names assigned and which names to use? As a rule of thumb, all components that a user directly interacts with should have a name, for example buttons, menus, text fields, etc. Components that are not created directly, but are automatically generated as children of complex components

don't need a name, for example the scroll bars of a `JScrollPane`, or the list of a `JComboBox`. The component itself should have a name, however.

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

Since QF-Test "knows" the components for which `setName` is most useful, it comes with a feature to locate and report these components. QF-Test even suggests names to assign, though these aren't necessarily useful. This feature is similar to component recording and is explained in the documentation for the option Hotkey for components⁽²⁶¹⁾.

The suggested names for DOM nodes are currently not very useful.

5.8 Updating Components

Unavoidably the components of the SUT are going to change over time. If names are used consistently this is not really a problem, since in that case QF-Test can cope with just about any kind of change.

Without names however, changes tend to accumulate and may reach a point where component recognition fails. To avoid that kind of problem, QF-Test's representation of the SUT's components should be updated every now and then to reflect the current state of affairs. This can be done with the help of the **Update component(s)** menu-item in the context menu that you get by right-clicking on any node under the Windows and components⁽⁵⁵⁶⁾ node.

This function can change a lot of information in your test-suite at once and it may be difficult to tell whether everything went fine or whether some components have been misidentified. To avoid problems, always create a backup file before updating multiple components. Don't update too many components at once, take things Window by Window. Make sure that the components you are trying to update are visible except for the menu-items. After each step, make sure that your tests still run fine.

Provided that you are connected to the SUT, this function will bring up the following dialog:

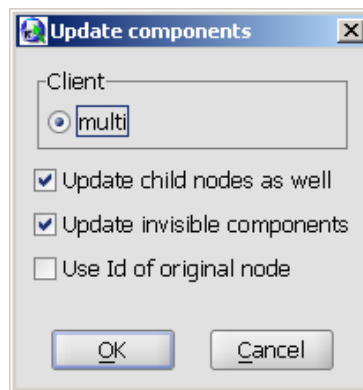


Figure 5.4: Update components dialog

If you are connected to multiple SUT clients, you must choose one to update the components for.

Select whether you only want to update the selected Component⁽⁵⁴⁵⁾ node or all its child nodes as well.

You can choose to include components that are not currently visible in the SUT. This is mostly useful for menu-items.

The QF-Test ID⁽⁵⁴⁶⁾ for an updated node is left unchanged if "Use QF-Test component ID of original node" is selected. Otherwise, updated nodes will receive a QF-Test ID generated by QF-Test. If the QF-Test ID of a node is changed, all nodes referring to that node via their QF-Test component ID attribute will be updated accordingly. QF-Test also checks for references to the component in all suites of the same project and in those suites that are listed in the Dependencies (reverse includes)⁽³¹²⁾ attribute of the Test-suite⁽³¹¹⁾ node. Those suites are loaded automatically and indirect dependencies are resolved as well.

Note In this case, QF-Test will open modified test-suites automatically, so you can save the changes or undo them.

After pressing "OK", QF-Test will try to locate the selected components in the SUT and fetch current information for them. Components that are not found are skipped. The Component nodes are then updated according to the current structure of the SUT's GUI, which may include moving nodes to different parents.

Note For large component hierarchies this very complex operation can take a while, in extreme cases even a few minutes.

This function is especially useful when names have been set for the first time in the SUT. If you have already generated substantial test-suites before convincing the developers to add names, you can use this function to update your Components to include the new names and update their QF-Test IDs accordingly. This will work best if you can get hold

of an SUT version that is identical to the previous one except for the added names.

Note

Very important note: When updating whole windows or component hierarchies of significant size you may try to update components that are not currently visible or available. In that case it is very important to avoid false-positive matches for those components. You may want to temporarily adjust the bonus and penalty options for component recognition described in [section 29.3.4^{\(284\)}](#) to prevent this. Specifically, set the 'Feature penalty' to a value below the 'Minimum probability', i.e. to 49 if you have not changed the default settings. Don't forget to restore the original value afterwards.

If you need to change the setting of the options [Name override mode \(replay\)^{\(285\)}](#) and [Name override mode \(record\)^{\(264\)}](#) because, for example, component names turned out not to be unique after all, change only the setting for the recording options before updating the components. When finished, change the replay option accordingly.

5.9 Troubleshooting component recognition problems

If your SUT has changed in a way that makes it impossible for QF-Test to locate a component, your test will fail with a [ComponentNotFoundException^{\(558\)}](#). This should not be confused with an [UnresolvedComponentIdException^{\(563\)}](#) which is caused by removing a Component node from the test-suite or changing the QF-Test component ID attribute of an Event node to a non-existing QF-Test ID.

When you get a [ComponentNotFoundException](#), rerun the test with QF-Test's debugger activated so that the test gets suspended and you can look at the node that caused the problem. Here it pays if your QF-Test ID attributes are expressive because you need to understand which component the test tried to access. If you cannot figure out what this node is supposed to do, try to deactivate it and rerun the test to see if it runs through now. It could be a stray event that was not filtered during recording. In general your tests should only contain the minimum of nodes required to achieve the desired effect.

If the node needs to be retained, take a look at the SUT to see if the target component is currently visible. If not, you need to modify your test to take that situation into account. If the component is visible, ensure that it was already showing at the time of replay by checking the screenshot in the run-log and try to re-execute the failed node by single-stepping. If execution now works you have a timing problem that you need to handle by either modifying the options for default delays ([section 29.3.5^{\(288\)}](#)) or with the help of a [Wait for component to appear^{\(512\)}](#) node or a Check node with a Timeout. As a last resort you can work with a fixed delay.

If the component is visible and replay fails consistently, the cause is indeed a change in the component or one of its parent components. The next step is identifying what changed and where. To do so, re-record a click on the component, then look at the old

and new Component node in the hierarchy under Windows and components⁽⁵⁵⁶⁾.

Note

You can jump directly from the Event node to the corresponding Component node by pressing **Ctrl-W** or right-clicking and selecting **Locate component**. You can jump back via **Ctrl-Backspace** or **Edit→Select previous node**. A clever trick is to mark the Component nodes to compare by setting breakpoints on them to make them easier to spot.

The crucial point is where the hierarchy for those two components branches. If they are located in different Window nodes, the difference is in the Window itself. Otherwise the old and new Component have a common ancestor just above the branching point and the crucial difference is in the respective nodes directly below that branch. When you have located those nodes, examine their attributes top-to-bottom and look for differences.

Note

You can open a second QF-Test window via **View→New window...** so as to place the detail views of the nodes to compare side to side.

The only differences that will always cause recognition failures are Class name and Name. Differences in Feature, structure or geometry attributes can usually be compensated unless they accumulate.

A change in the Class name attribute can be caused by refactoring done by development, in which case you need to update your Class name attribute(s) to reflect the change(s). Another possible cause is obfuscation, a technique for making the names of the application classes illegible for protection against prying eyes. This poses a problem because the class names can then change with each version. You can prevent both refactoring and obfuscation problems by activating the option Record system class only⁽²⁶³⁾.

If the Name has changed things get more difficult. If the change is apparently intentional, e.g. a typo was fixed, you can update the Name attribute accordingly. More likely the cause is some automatically generated name that may change again anytime. As explained in the previous section, your options in this case are discussing things with development or suppressing such names with the help of a `NameResolver` as described in section 39.1⁽⁶⁷⁸⁾.

Changes to the Feature attribute are common for Window nodes, where the Feature represents the window title. When combined with a significant change in geometry such a change can cause recognition to break. This can be fixed by updating the Feature to match the new title or, preferably, by turning it into a regular expression that matches all variants.

Depending on the kind and amount of changes to accommodate there are two ways to deal with the situation:

- Update the attributes of the old node and remove the newly recorded ones.
- Keep the new node and remove the old one, first taking care that events referring to the old node are updated to the new QF-Test ID. There's a clever trick to achieve that: Edit the QF-Test ID of the old Component node to match the new one. QF-Test

will complain about the duplicate QF-Test ID, which you can ignore, and offer to update all references, which you should accept. Then you can safely remove the old node.

Note

Automatic updates for references from other test-suites require that the suites belong to the same project or the correct setting the Dependencies (reverse includes)⁽³¹²⁾ attribute of the Test-suite root node.

5.10 Accessing hidden fields on a web page

Web

Hidden fields are not captured by default and therefore not stored under the Windows and components⁽⁵⁵⁶⁾ node.

In case you frequently need to access hidden fields you can deactivate the Take visibility of DOM nodes into account⁽²⁹⁶⁾ option.

Another way to get hidden fields recorded is the following:

- Activate the record components⁽³⁴⁾ mode, navigate the mouse cursor to the parent item containing the hidden field (most likely a FORM element).
- Press right mouse button and select Component and children from the popup menu.
- Deactivate the record components⁽³⁴⁾ mode.
- A search⁽¹⁶⁾ within Windows and components⁽⁵⁵⁶⁾ for e.g. 'HIDDEN' should get you to your destination component quickly.

To access a hidden field's attributes (e.g. the 'value' attribute) you can create a simple SUT script⁽⁴¹⁹⁾ as shown below. Details on scripting in general, the used methods and parameters can be found in Scripting⁽⁹⁹⁾, Run-context API⁽⁶²²⁾ and Pseudo DOM API⁽⁶⁴⁷⁾ respectively.

```
node = rc.getComponent('id of the hidden component', hidden=1)
node.getAttribute('value')
```

Example 5.1: Accessing the value attribute of a hidden field

Chapter 6

Sub-items of complex GUI components

Some components of a rich GUI, trees or tables most notably, are rather complex and can contain an arbitrary number of elements like tree-nodes or table-cells. Depending on the kind of application, these elements may themselves be GUI components (Nodes in JavaFX, DOM nodes in a web application) or just a graphical representation of some data (Swing renderer concept, SWT Tree and Table items). This is a purely technical distinction and because it makes sense, from the tester's point of view, to treat these elements uniformly as separate entities and possible targets for events. QF-Test represents such sub-items in one of two ways, either as an Item⁽⁵⁵¹⁾ node or with a special syntax.

6.1 Item nodes

An `Item` is defined by two parts, the component it belongs to and an index into this component. The component is identified by the parent node of the `Item` node. The index can be either numeric or textual. Numeric indexes start with 0. In a `List`, for example, the element with index 1 is the second element from the top. Plain numeric indexes are not very useful for tree-nodes, because only the visible nodes are counted. Expanding or collapsing a tree node changes the numeric indexes of all nodes below it.

A textual index refers to the element by the text that is shown in the GUI. A list-item shown as "item1" in a `List` component would be recorded with a textual index of "item1". The textual representation is more flexible but can cause problems if the names of the elements in a component are not unique. A textual index can also be given as a regular expression (see section 36.4⁽⁶¹⁴⁾). In this case, the first element found that matches the regular expression will be used.

The option Sub-item format⁽²⁶⁸⁾ controls the format for recording indexes.

Almost all kinds of Items have only one index. The exception is the table-cell in a `Table` component. Since tables are two-dimensional structures, two indexes are required to refer to a table-cell. The first one, the Primary index⁽⁵⁵²⁾, defines the column of the table and the Secondary index⁽⁵⁵²⁾ the row.

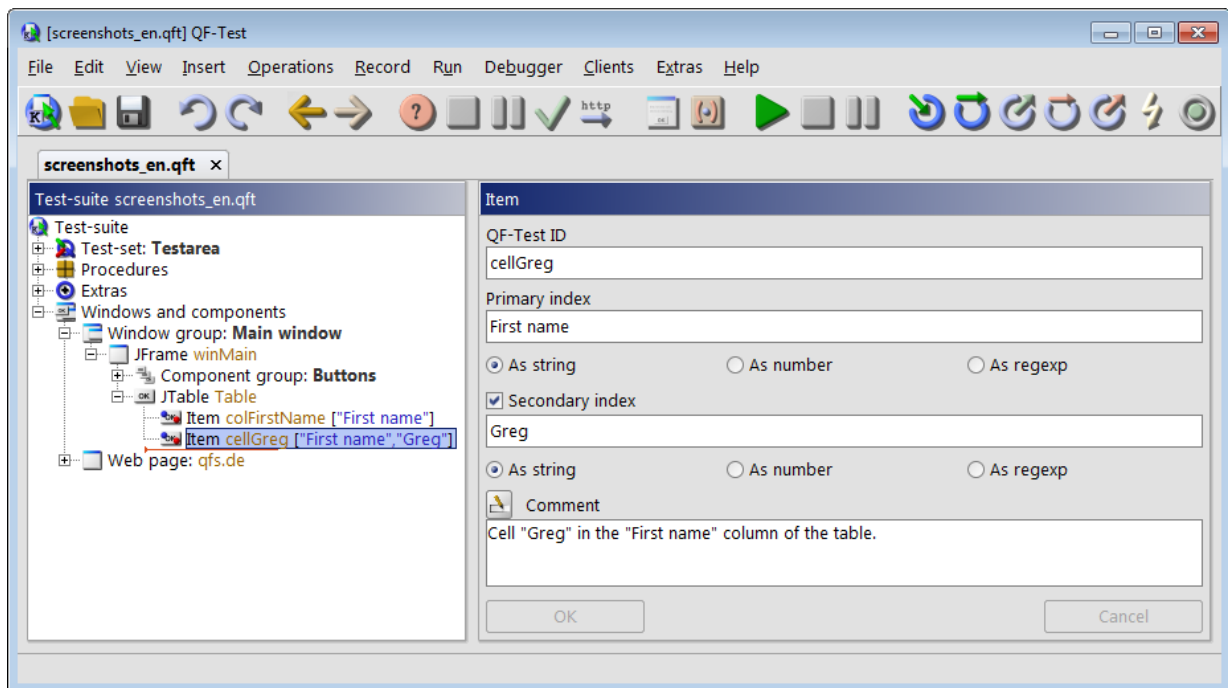


Figure 6.1: An Item for a table cell

Tree-nodes are also special. The hierarchic structure of a tree doesn't map well to a linear one. One reason is the dependence of the numeric index on the current expansion state of the nodes. Another reason is that many kinds of trees have nodes with non-unique names. However, when the direct and indirect parent-nodes are taken into account to work with paths instead of names, uniqueness is often restored.

QF-Test uses a special syntax to represent tree nodes. An index that starts with a '/' character is considered to be a path index. Imagine a tree that shows a file-system hierarchy. The file `"/tmp/file1"` could be represented flat as just `"file1"`, possibly colliding with another `"file1"` in some other directory, or as the full, unique path `"/tmp/file1"`. QF-Test also supports numeric indexes with this syntax. As a numeric index `"/2/3"` designates the fourth child node of the third child of the root. A mixed-format index like `"/tmp/2"` to get the third child of the node named `"tmp"` is currently not possible.

Note

This special syntax makes `'/'` a special character for elements of tree components, so it needs to be escaped when a literal `'/'` character is required. See section 36.6⁽⁶¹⁶⁾ for

details about escaping and quoting special characters.

6.2 The QF-Test ID of an Item

Everything that was said about the QF-Test ID⁽⁵⁴⁶⁾ attribute of Components in [section 5.3](#)⁽⁴⁰⁾ also applies to an Item's QF-Test ID⁽⁵⁵²⁾. It has to be unique and can be referred to by events and similar kinds of nodes that require a component target.

When assigning an Item node's QF-Test ID automatically, QF-Test creates it by concatenating the containing Component's QF-Test ID and the index (or indexes) of the element. These kinds of QF-Test IDs are expressive and typically easy to recognize. However, they are also the source of a common misunderstanding: If you want to change the index of a recorded Item to point to some other element, you **must not** change the QF-Test component ID attribute of the node that refers to it (unless you use the special syntax described in the following section). Change the Primary index of the Item node instead.

6.3 Special QF-Test ID-syntax for direct element access

There is an alternative way to target an event on an element of a complex component that does not require creation of an Item node. By using a special syntax you can specify the index of the element directly in the QF-Test component ID attribute of the respective node. Instead of the QF-Test ID of an Item node, specify the QF-Test ID of the containing Component and follow it with a special separator character and the index of the element. The character that separates the QF-Test ID and the index designates the format for the index:

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

Table 6.1: Separators and index formats for element access

To address a table-cell with Primary index and Secondary index simply append another separator character and the second index. That way, the Primary index and Secondary index can be of different format.

Example: If your test-suite contains a Component node for a `Table` component with the QF-Test ID `mytable` and you want to simulate a click on table-cell in the 6th row of

the column with the title "Name", create a Mouse event⁽⁴⁴⁷⁾ node and set its QF-Test component ID⁽⁴⁴⁸⁾ attribute to `mytable@Name&5`.

Almost all internal `ItemResolver` implementations (cf. section 39.2⁽⁷⁰³⁾) allow of negative indices to start counting from the end. For example, `mytable&-1&-1` addresses the last cell in the rightmost table column.

Whether it is preferable to use an Item⁽⁵⁵¹⁾ node or the special syntax depends on the situation. You can use either method as you see fit and mix them as you like. As a rule of thumb, `Item` nodes are better suited to components with a few more or less constant elements, the columns of a `Table` for example, or the tabs of a `TabbedPane`. The special syntax is more flexible and generally preferable for indexes with variable values and for elements with varying or editable values. The option Sub-item type⁽²⁶⁸⁾ determines whether `Item` nodes are created or the special syntax is used when recording. With the default value "Intelligent" QF-Test will follow the rules outlined above.

The special meaning of the separator characters implies that literal '@', '&' or '%' characters in element indexes need to be escaped. See section 36.6⁽⁶¹⁶⁾ for details about escaping and quoting special characters.

6.4 Multi-level sub-items and QPath

In JavaFX or web applications it is possible for items like a table cell to contain arbitrary components. Typical examples are `CheckBoxes` or `TextFields`, but it is even possible to nest tables. Such inner components can be addressed using a special multi-level item syntax called `QPath` that is similar to `XPath`. `QPath` is not limited to sub-items, it can be used to address child components of any component and thus extend or replace the component hierarchy in a test-suite.

Any QF-Test component ID used in an event or check node, regardless of whether it already contains sub-item syntax or not, can be affixed with one or more special sub-item indexes of the form `@:ClassName<idx>`, where `<idx>` is optional. This instructs QF-Test to first resolve the target component (and possibly item) for the part of the QF-Test component ID before the '@:' and therein search for visible components of class `ClassName`. If `<idx>` is specified, it is treated as a 0-based index into the list of visible candidate components. No `<idx>` is equivalent to `<0>`.

Why introduce `QPath` instead of simply using `XPath`? Well, all indexes in QF-Test are 0-based, especially numerical item indexes and the Class index⁽⁵⁴⁹⁾ attribute for the class structure of `Component` nodes, so it would have been inconsistent to use a 1-based index in this case, but `XPath` is 1-based. And using `XPath` syntax of the form `[idx]` with a 0-based index would have been even more confusing.

6.5 Activating item recording for web tables

Web

Within web pages, tables are used both for 'real' visible tables and for invisible layout purpose. Also nested tables are possible. QF-Test typically cannot know where a real table starts, and therefore item based recording is not possible without further information.

Hence this information can be provided by the user. By use of the procedure `qfs.web.table.recording.activateItemRecording` from the standard library⁽⁸⁸⁾ item recording can be activate for a given web table.

In case you don't know which QF-Test component ID to provide to the procedure, simply record a click to an arbitrary table cell. Jump to the respective component node by pressing **Ctrl-W** or right-clicking and selecting **Locate component** and look for the next node with the class name `Table` upwards in the component hierarchy. That should be the one to use.

Note

When using one of the AJAX toolkit resolvers⁽⁶⁵⁷⁾ you typically don't have to do any separate activation. AJAX tables are handled by the resolver.

Chapter 7

Variables

Variables are the primary means to add flexibility to a test-suite. Though they are used mainly as parameters for Procedures⁽³⁷⁸⁾, they are also useful in many other cases.

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

- `$(varname)` expands to the value of some previously defined variable.
- `${group:name}` accesses external data⁽⁶²⁾ from a resource bundle or property file. The groups `system` and `qftest` are always defined and have a special meaning (see section 7.5⁽⁶²⁾).
- `$(expression)` evaluates some mathematical expressions⁽⁶⁶⁾.

7.1 Variable lookup

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

Each set of variable definitions, called bindings, is placed on one of two stacks of bindings. One stack is used for actual - or direct - definitions and one for fallback bindings or default values. When a variable's value is requested via `$(...)`, QF-Test first searches the stack of direct bindings from top to bottom, then the stack of fallbacks, also top down. The first value found is used. If there is no binding at all for a name, an UnboundVariableException⁽⁵⁶⁰⁾ is thrown unless you use the special syntax `${default:varname:defaultvalue}` to provide a default value for this case as described in section 7.5⁽⁶²⁾.

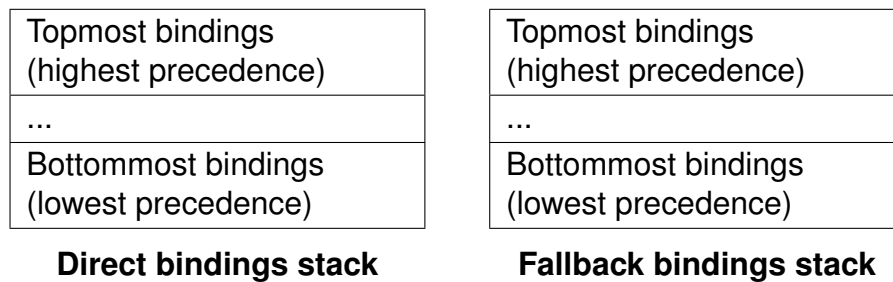


Figure 7.1: Direct and fallback bindings

This mechanism supports recursive or self-referencing variable definitions. For example, setting a variable named `classpath` to the value `some/path/archive.jar:${classpath}` will extend a binding for `classpath` with lower precedence. If no such binding exists, a `RecursiveVariableException`⁽⁵⁶⁰⁾ is thrown.

7.2 Defining variables

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

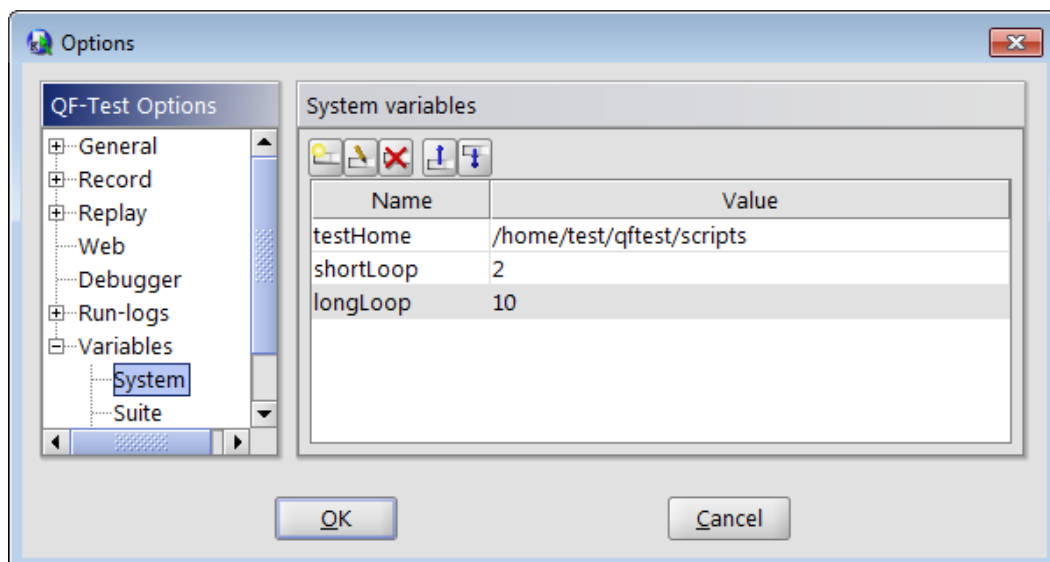


Figure 7.2: System variables

Four sets of variable definitions are accessible from the global options⁽³⁰⁹⁾:

System variables

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

Suite variables

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

Variables from the command line

You can define variables on the command line when starting QF-Test with the -variable <name>=<value>⁽⁵⁸⁶⁾ argument which can be used multiple times. These bindings are placed above the suite bindings on the direct bindings stack, so they override the bindings of the suite. For example, a variable `count=1` can be defined for the suite and used as `$(count)` for the Number of iterations⁽³⁹⁰⁾ of a Loop⁽³⁸⁸⁾ node for quick trial runs. Then you can use `qftest -batch -variable count=100 ...` for the actual testing. The command line bindings are accessible mainly for your information, but you can also edit them for experimentation.

Global variables

Placed above the command line variables on the direct call stack, these bindings are a means to pass values between unrelated parts of a test run. Values read from the SUT by a Fetch text⁽⁴⁹⁹⁾ or a Fetch index⁽⁵⁰²⁾ node are bound here, as well as definitions from a Set variable⁽⁵⁰⁹⁾ node. The globals are not cleared in between test runs as a convenience and you can edit them for the same reason. To simulate a real test run you should clear them via the Run→Clear global variables menu first. When run in batch mode (see section 1.6⁽⁷⁾) QF-Test clears the global variables before the execution of each test given with the -test <n>|<ID>⁽⁵⁸⁴⁾ command line argument.

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

- Each sequence type node defines a set of bindings that are placed on top of the direct bindings stack when the test run enters the node and removed when the node is exited.
- A Procedure call's⁽³⁸¹⁾ bindings are placed on top of the direct bindings stack before executing the Procedure⁽³⁷⁸⁾. The bindings defined by a Procedure are placed on top of the fallback stack during execution of the Procedure. Think of them as default values for the expected parameters. Once the procedure returns, both sets of bindings are removed.

7.3 Variable example

Consider the following example:

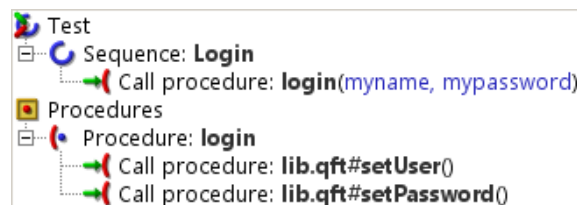


Figure 7.3: Variable example

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

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

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

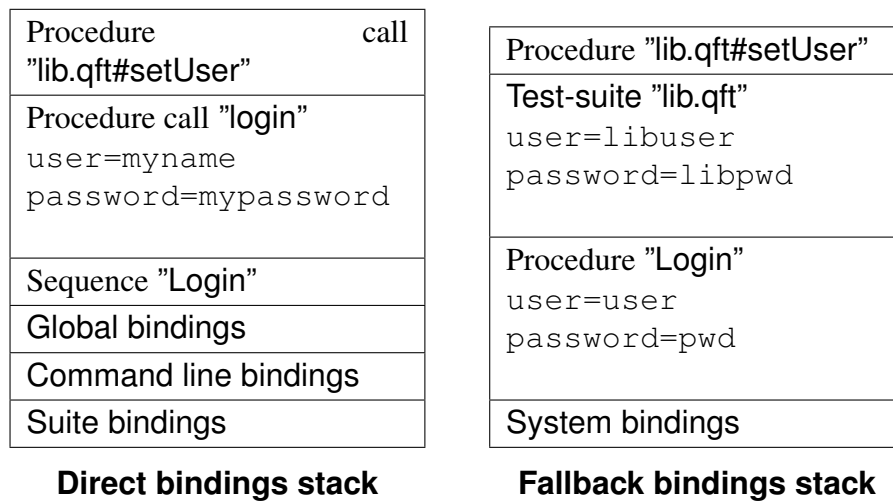


Figure 7.4: Variable bindings example

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

7.4 Fetching data from the GUI

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

QF-Test offers a special set of query nodes⁽⁴⁹⁸⁾ for this kind of task, available via Insert→Miscellaneous:

- Fetch text⁽⁴⁹⁹⁾ to fetch a component or item text,
- Fetch index⁽⁵⁰²⁾ to fetch the index of an components item,
- Fetch geometry⁽⁵⁰⁵⁾ to fetch a component or items geometry.

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

Instead of inserting fetch nodes by hand they can easily be created by first recording a mouse click node the the respective component and then use the transform operation⁽¹³⁾ to convert the same to the fetch node you need.

7.5 External data and special groups

External data is made available by the Load resources⁽⁵²³⁾ and Load properties⁽⁵²⁶⁾ nodes. These assign a group name to a set of definitions from a resource bundle or properties file. To access the value of the definition for *name*, use the syntax `${group:name}`.

When run in batch mode (see section 1.6⁽⁷⁾) QF-Test clears the resources and properties before the execution of each test given with the `-test <n>|<ID>`⁽⁵⁸⁴⁾ command line argument. In normal mode QF-Test keeps them around to ease building a suite, but for a true trial run you should clear them via the Run→Clear resources and properties menu first.

Some special group names are predefined and always available:

system

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

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

env

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

default

3.4+

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

id

3.1+

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

quoteitem

4.0+

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

quoteregex, quoteregexp

4.0+

The group `quoteregex` with its alias `quoteregexp` can be used to escape characters with special meaning in regular expressions. This is often useful when building regular expressions dynamically.

qftest

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

Name	Meaning
32 or 32bit	<i>true</i> if QF-Test is running in a 32bit Java VM - which is not the same as running on a 32bit Operating System - <i>false</i> otherwise.
64 or 64bit	<i>true</i> if QF-Test is running in a 64bit Java VM, <i>false</i> otherwise.
batch	<i>true</i> if QF-Test is running in batch mode, <i>false</i> for interactive mode.
client.exitcode.<name>	The exit-code of the last process started with the <u>Client</u> ⁽⁴²³⁾ attribute set to <name>. In case the process is still alive the result is the empty string.
client.output.<name>	The output of the last process started with the <u>Client</u> ⁽⁴²³⁾ attribute set to <name>. The maximum size for buffered output is defined by the option <u>Maximum size of client terminal (kB)</u> ⁽²⁷⁴⁾ .
clients	A list of the names of all active process clients, separated by a newline.
clients.all	A list of the names of all process clients, separated by a newline. This includes live clients as well as the recent dead clients similar to those listed in the "Clients" menu.

<code>count.exceptions</code>	Number of exceptions in the current test-run.
<code>count.errors</code>	Number of errors in the current test-run.
<code>count.warnings</code>	Number of warnings in the current test-run.
<code>count.testcases</code>	Total number of total test-cases (run and skipped) in the current test-run.
<code>count.testcases.exception</code>	Number of test-cases with exceptions in the current test-run.
<code>count.testcases.error</code>	Number of test-cases with errors in the current test-run.
<code>count.testcases.expectedtofail</code>	Number of test-cases expected to fail in the current test-run.
<code>count.testcases.ok</code>	Number of successful test-cases in the current test-run.
<code>count.testcases.ok.percentage</code>	Percentage of successful test-cases in the current test-run.
<code>count.testcases.skipped</code>	Number of skipped test-cases in the current test-run.
<code>count.testcases.notimplemented</code>	Number of not implemented test-cases in the current test-run.
<code>count.testcases.run</code>	Number of run test-cases in the current test-run.
<code>count.testsets.skipped</code>	Number of skipped test-sets in the current test-run.
<code>dir.root</code>	QF-Test's root directory
<code>dir.version</code>	Version-specific directory of QF-Test
<code>executable</code>	The <code>qftest</code> executable matching the currently running QF-Test version, including the full path to its <code>bin</code> directory and with <code>.exe</code> appended on Windows. Useful if you need to run QF-Test from QF-Test for example to call a daemon or create reports.
<code>java</code>	Standard Java program (<code>javaw</code> under Windows, <code>java</code> under Linux/Unix) or the explicit Java argument if QF-Test is started with <code>-java <executable></code> ⁽⁵⁷²⁾
<code>return</code>	The most recent value returned from a <u>Procedure</u> ⁽³⁷⁸⁾ through a <u>Return</u> ⁽³⁸⁴⁾ node.
<code>runid</code>	The runid of the current test-run. See <u>section 15.1</u> ⁽¹³⁰⁾ for further information about the runid.
Name	Meaning
<code>screen.height</code>	Screen height in pixels
<code>screen.width</code>	Screen width in pixels

skipnode	This magic value is not for the casual user. It causes QF-Test to skip execution of the current node. Its primary use is as the value for a variable defined in the <u>Text</u> ⁽⁴⁵⁶⁾ attribute of a <u>Text input</u> ⁽⁴⁵⁴⁾ node which also has its <u>Clear target component first</u> ⁽⁴⁵⁶⁾ attribute set. An empty value would clear the field whereas <code>\$_{qftest:skipnode}</code> leaves the field unchanged. But skipnode is also applicable for fine-grained execution control by placing a variable in the comment of a node and selectively passing <code>\$_{qftest:skipnode}</code> to that variable. Please note that you almost always want to use lazy syntax <code>'\$_'</code> with this variable. Otherwise its expansion as the parameter in a Procedure call node would cause skipping the whole call.
suite.dir	Directory of the current suite
suite.file	File name of the current suite without directory
suite.path	File name of the current suite including directory
testcase.name	The name of the current Test-case, empty if no Test-case is currently being executed.
testcase.qname	The qualified name of the current Test-case, including the names of its parent Test-sets. Empty if no Test-case is currently being executed.
testcase.reportname	The expanded report name of the current Test-case, empty if no Test-case is currently being executed.
testcase.splitlogname	The qualified name of the current Test-case converted to a filename, including the names of its parent Test-sets as directories. Empty if no Test-case is currently being executed.
testset.name	The name of the current Test-set, empty if no Test-set is currently being executed.
testset.qname	The qualified name of the current Test-set, including the names of its parent Test-sets. Empty if no Test-set is currently being executed.
testset.reportname	The expanded report name of the current Test-set, empty if no Test-set is currently being executed.
testset.splitlogname	The qualified name of the current Test-set converted to a filename, including the names of its parent Test-sets as directories. Empty if no Test-set is currently being executed.
teststep.name	The name of the current Test-step, empty if no Test-step is currently being executed.
teststep.qname	The qualified name of the current Test-step, including the names of its parent Test-steps, but not including Test-cases or Test-sets. Empty if no Test-step is currently being executed.
teststep.reportname	The expanded report name of the current Test-step, empty if no Test-step is currently being executed.

<code>thread</code>	The index of the current thread. Always 0 except if QF-Test is started with the argument <code>-threads <number></code> ⁽⁵⁸⁶⁾ .
<code>threads</code>	The number of parallel threads. Always 1 except if QF-Test is started with the argument <code>-threads <number></code> ⁽⁵⁸⁶⁾ .
<code>version</code>	QF-Test version
<code>windows</code>	"true" under Windows, "false" under Linux/Unix

Table 7.1: Definitions in the special group `qftest`

7.6 Expressions

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

In fact these expressions are far more powerful, since they are evaluated by the Jython interpreter. You can use arbitrary constructs that are legal syntax for the Jython method `eval`. See [chapter 13](#)⁽⁹⁹⁾ for details about Jython scripting.

Note

Accessing QF-Test variables in an expression follows the same rules as in Jython scripts (see [section 13.3.3](#)⁽¹⁰³⁾). You can use the standard QF-Test syntax `$(...)` and `${...:...}` for numeric or Boolean values. String values should be accessed with `rc.lookup(...)`.

7.7 Immediate and lazy binding

3.0+

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

The difference, of course, is the time and thus the context in which a variable is expanded. In most cases there is no difference at all, but there are situations where it is

essential to use either lazy or immediate binding. Consider the following two examples:

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

Imagine another utility test-suite with a procedure to copy a file. Two parameters called 'source' and 'dest' specify the source file and destination directory. The caller of the procedure wants to copy a file called 'data.csv' from the same directory as the calling test-suite to some other place. The natural idea is to bind the variable 'source' to the value '\${qftest:suite.dir}/data.csv' in the procedure call. With immediate binding, '\${qftest:suite.dir}' will indeed expand to the directory in which the calling suite resides. However, if lazy binding were used, the actual expansion would take place inside the procedure. In that case, '\${qftest:suite.dir}' would expand to the directory of the utility suite, which most likely is not what the caller intended.

In versions of QF-Test up to and including 2.2 all variable expansion was lazy. As the examples above show, both variants are sometimes necessary. Since immediate binding is more intuitive it is now the default. This can be changed with the option When binding variables, expand values immediately⁽³⁰⁹⁾. The option Fall back to lazy binding if immediate binding fails⁽³⁰⁹⁾ complements this and helps to ease migration of old test-suites to the use of Immediate Binding. The warnings issued in this context help locating the few spots where you should use explicit lazy binding as described below. Except for very rare cases where lazy binding is required but immediate binding also works so that the fallback is not triggered, all tests should work out of the box.

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

Note

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

Chapter 8

Problem analysis and debugging

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

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

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

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

8.1 The run-log

During test replay QF-Test creates a run-log that records everything that is going on. The run-logs for recent tests are accessible from the **Run** menu, the current or most recent run-log can also be opened by typing **Ctrl-L**. See [section 8.1.5^{\(73\)}](#) for information about options influencing run-log creation.

The structure of a run-log is similar to that of a test-suite, but there are subtle differences. Nodes are added to the run-log when they are executed. `Setup`⁽³⁵⁰⁾ and `Cleanup`⁽³⁵²⁾ nodes, for example, are typically executed more than once, in which case multiple copies will be recored in the run-log as shown below:

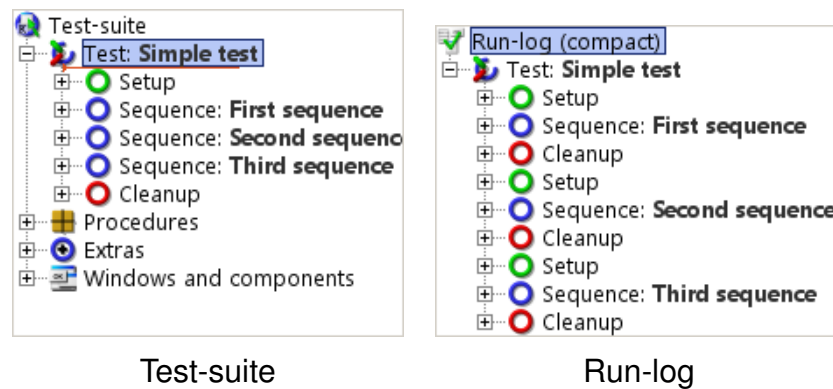


Figure 8.1: A simple test and its run-log

A run-log is the essential resource for determining **what** went wrong in a test, **where** it went wrong and maybe even get an idea about **why** it went wrong. Therefore the emphasis is on completeness of information rather than readability and a run-log should not be confused with a report or summary. Report generation is covered in [chapter 15](#)⁽¹²⁹⁾.

In addition to the nodes copied from the test-suite, a run-log contains failure information, optional annotations, information about variable expansion, various kinds of messages and, not to forget, timestamps and durations.

Note

The difference between the values of "Duration" and "Real time spent" are explicit delays introduced in nodes via the 'Delay before/after' attribute or user interrupts.

The information gathered from a long test-run accumulates and can eat up enormous amounts of memory, so QF-Test normally creates a special form of compact run-log in which only the information relevant for report generation and for error diagnosis is retained. This functionality is configurable through the option `Create compact run-log`⁽³⁰⁷⁾ and the command line argument `-compact`⁽⁵⁷⁴⁾. This should not be mixed up with the option to save *compressed* run-logs as `*.qrz` files which simply uses plain gzip compression to reduce the size of the log file, but doesn't have any impact on memory usage. Another alternative to save memory is to create split run-logs as described in [section 8.1.4](#)⁽⁷²⁾. Compact and split run-logs can be combined for maximum efficiency.

8.1.1 Error states

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

Warnings

Warnings indicate problems that are typically not serious, but might lead to trouble in the future and may need looking at. For example, QF-Test issues a warning, if the best match for a component barely meets the requirements and differs in some significant way.

Errors

Errors are considered to be serious defects that require closer inspection. They indicate that the SUT does not fulfill some requirement. A typical cause for an error is a mismatch in a Check text⁽⁴⁷⁰⁾ node.

Exceptions

Exceptions are the most serious kinds of errors. An exception is thrown when a situation occurs in which QF-Test cannot simply continue with the execution of the test. Most exceptions indicate problems with the logic of the test, though they can just as well be caused by the SUT. A ComponentNotFoundException⁽⁵⁵⁸⁾, for example, is thrown when no component in the SUT matches the intended target for an event. A list of all possible exceptions is available in chapter 31⁽⁵⁵⁸⁾.

Each node of a run-log has an associated state which can be one of *normal*, *warning*, *error* or *exception*. This state is visually represented by a frame around the node's icon which is orange for *warning*, red for *error* and bold red for *exception*.

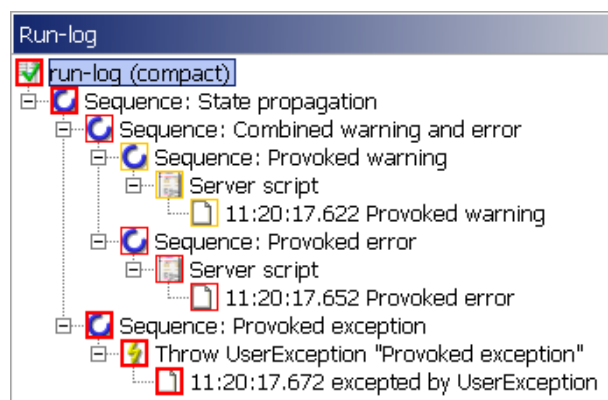


Figure 8.2: Error states in a run-log

As shown in the (somewhat reduced) screenshot above, error states propagate from bottom to top. The *exception* state takes precedence over the *error* state, which in turn overrides *warning*. The most severe kind of error that propagates to the top of the tree determines the final result of a test and QF-Test's exit code when run in batch mode (see [section 32.3^{\(588\)}](#)).

If necessary, the propagation of errors can be restricted for all kinds of [Sequence^{\(334\)}](#) nodes with the help of the [Maximum error level^{\(335\)}](#) attribute. This can be useful for sequences which are known to contain errors that should not be taken into account just yet. Exceptions can be handled with the help of the [Try^{\(405\)}](#) and [Catch^{\(408\)}](#) nodes. The [Maximum error level^{\(410\)}](#) attribute of the Catch node determines the state to propagate for a caught exception.

8.1.2 Navigating the run-log tree

All of the basic editing methods for a run-log are similar to those for a test-suite. One significant difference is that can neither add or remove any nodes nor edit the attributes of the nodes copied from the test-suite. You can add annotations though, for example to document the reason for an error if it is known.

The first question to answer when looking at a run-log is "What happened"? The function [Edit→Find next error](#), [Ctrl-N](#) for short, moves the selection to the next place at which an error actually occurred and [Edit→Find previous error](#) ([Ctrl-P](#)) moves backwards. The options [Find next error function locates^{\(302\)}](#) and [Skip suppressed errors^{\(302\)}](#) determine the kinds of errors to look for and whether to ignore those that didn't propagate up to the root node. There's a menu item shortcut [Edit→Skip suppressed errors](#) to quickly toggle the latter option.

The next question might be "Where did this happen"? Though a run-log is similar in many ways to a test-suite, the connection isn't always obvious. The function [Edit→Find node in test-suite](#) ([Ctrl-T](#)) will take you to the exact node in the test-suite that is represented by the selected node in the run-log, always provided that the test-suite can be located and hasn't changed in a way that prevents this. If the run-log is loaded from a file, the corresponding test-suite may not be located at the same place as when the test was executed. If the test-suite cannot be found, a dialog will pop up that lets you select a different file. In case you select a wrong file or some other test-suite is found instead of the one the run-log was created from, you may end up at some totally different node, or none at all. In that case you can use the menu item [Edit→Locate corresponding test-suite](#) to explicitly change the test-suite.

8.1.3 Accepting values of failed checks as good

A noteworthy feature of QF-Test's run-log is the ability to quickly accept the actual values of a failed Check node as good. When QF-Test logs a failed Check it includes the complete current state of the Check node's target component in the SUT. This is much more useful than the failure message alone, which, for example, might just tell you that a table column has 10 rows instead of the expected 9, but not what its contents are.

If you are analyzing a failed Check and see that the value in the SUT was actually correct and the expected value stored in the test-suite wrong, you can press **Ctrl-U** or select **Update check node with current data** from the context menu to accept the data from the failed Check as the new correct value for the Check node.

Warning: QF-Test currently doesn't preserve regular expressions in Check text⁽⁴⁷⁰⁾ or Check items⁽⁴⁸⁰⁾ nodes, they will simply get overwritten.

8.1.4 Split run-logs

Run-logs for long-running tests can get very large and consume an enormous amount of memory, even more so in case many screenshots are kept. Compact run-logs help, but not enough to make tests that run for days on end possible without turning off the run-log entirely. The best way to overcome the memory problem are split run-logs.

For split run-logs, whenever a certain part of a test has finished, QF-Test takes the run-log for that part, removes it from the main run-log, saves it to a separate file and replaces it with a single node that references that file. The partial logs are complete run-logs in themselves and can be viewed and archived independently, though normally they are accessed through the main run-log. When navigating the main run-log or when creating reports, QF-Test transparently loads the partial run-logs from the separate files as required and removes them from memory when no longer needed. This makes it possible to navigate huge run-logs while still retaining a relatively small memory footprint. Of course operations like searching or report creation that need to traverse the whole run-log become slower, but jumping from error to error remains quite fast and loading the main run-log is sped up drastically.

There are two ways for storing a main run-log and its partial logs: All combined together in a single ZIP file with the extension `.qzp` or with the partial logs in a separate directory. The latter is named after the main run-log with the extension `.qrl` or `.qrz` removed and the suffix `_logs` appended. Inside a `.qzp` ZIP file the same layout is used so that it is possible to zip or unzip files manually without breaking the internal references in the run-log. This compatibility is also the reason why by default partial logs inside the ZIP are stored compressed with the extension `.qrz`. This is slightly less efficient than storing uncompressed `.qrl` files, but that way a `.qzp` run-log can be unzipped without its overall size exploding.

To make use of split run-logs you can explicitly define points at which a run-log is broken and split into parts. This is done via the Name for separate run-log⁽³⁵⁹⁾ attribute of Data driver⁽³⁵⁷⁾, Test-case⁽³¹⁴⁾, Test-set⁽³²¹⁾, Test call⁽³²⁷⁾ or Test⁽³³⁰⁾ nodes. When used with a Data driver, the logs for each iteration are saved separately, in the other cases the node with the Name for separate run-log attribute is split off. Otherwise partial run-logs are split off automatically when they reach a certain size. This functionality can be configured via the option Minimum size for automatic splitting (kB)⁽³⁰³⁾.

Split run-logs are also very handy for tracking the progress of a test in batch mode. In that context it is extremely useful that the file names for the partial logs can be created using the same placeholders as when specifying the name of the main run-log on the command line. In particular the error state of the partial log can be made part of its filename. Please see the documentation for the attribute Name for separate run-log⁽³⁵⁹⁾ for details.

8.1.5 Run-log options

There are several options that influence the creation of run-logs and their content. These are explained in detailed in section 29.6⁽²⁹⁹⁾. Following is a short summary of the most important options:

- Unless Automatically save run-logs⁽³⁰¹⁾ is turned off, recent run-logs are saved in the directory `.qftest` in the user's home directory to save memory and to make them persistent between sessions.
- Number of run-logs in menu⁽³⁰¹⁾ sets the number of run-logs to keep around after a test is run. Unless you turn off automatically saved run-logs there is no need to reduce this number.
- Log level for the SUT⁽³⁰⁵⁾ determines which kinds of messages are logged. Logging all messages uses up lots of memory.
- If Create compact run-log⁽³⁰⁷⁾ is set, only nodes containing relevant error or report information are retained. This is a very useful setting in most cases.
- By turning Create split run-logs⁽³⁰³⁾ off, split run-logs can be disabled. Normally the Name for separate run-log attributes of various nodes determine if and where a run-log is split.
- With Don't create run-log⁽³⁰⁷⁾ creation of run-logs can be suppressed entirely. This is dangerous, since no information will be available in case of errors. Since split run-logs have become available this option is no longer required to make long-running demos or stress tests possible, but it is retained for backwards compatibility.

8.1.6 Creating a test-suite from the run-log

If several people are involved in the test development process, it might be useful to generate a test-suite from the run-log directly. The generated test-suite could be used to reproduce a test-run on-the-fly without having the entire structure of test-suites.

You can create a test-suite from the run-log via performing a right mouse click at any node in the run-log and selecting **Create test-suite** from the context menu.

QF-Test creates a new file containing all executed steps of the respective tests under Extras⁽³⁴²⁾ as well as the used components.

QF-Test only adds the executed steps to the new test-suite. Variables will be expanded immediately, so you can only see their value in the new file. Organizational nodes like procedures or control structures will not become created.

You have to set a couple of options in order to get this feature properly working (Under Run-log -> Content):

- Create compact run-log⁽³⁰⁷⁾ needs to be switched off.
- Log variable expansion⁽³⁰⁵⁾ needs to be switched on.
- Log parent nodes of components⁽³⁰⁵⁾ needs to be switched on.

If you have access to all test-suites, you can use also use information from them for creating the new one. Therefore select **Create test-suite from existing structure** from the context menu. In contrast to the approach described above, it is not required to switch on the option Log parent nodes of components⁽³⁰⁵⁾.

8.2 The debugger

As in any development environment, at some point the need will arise to debug problems introduced into a test-suite which cannot readily be solved by a straight-forward analysis of the elements and structure of the suite. To this end, QF-Test includes an intuitive debugger. Those of you familiar with debugging programs in Java or other programming languages will find this debugger similar in function and usefulness.

8.2.1 Entering the debugger

The QF-Test debugger can be started directly by selecting a node (or some nodes) to execute and pressing the step-in  or step-over  buttons, or by using the menu

operations **Debugger→Step in** and **Debugger→Step over** or the keyboard shortcuts **F7** and **F8**. See [section 8.2.3^{\(76\)}](#) for a detailed explanation of these operations.

If you are running tests on your test-suite and use the play button to start execution (see [section 4.2^{\(31\)}](#)), the debugger will normally not be entered. However, the debugger will be activated automatically when any one of the following occur:

- A user-defined breakpoint is reached (see [section 8.2.4^{\(77\)}](#) on turning on/off breakpoints).
- Execution is interrupted manually by pressing the pause button or **F9** or selecting the **Run→Pause** menu item.
- A caught or uncaught exception is thrown, an error happens or a warning is logged and the respective option to break under that condition is set (see option [Automatic breaks^{\(298\)}](#)).

When the debugger suspends execution of the test, the node about to be executed will be shown with a colored frame around its icon that reflects the cause for the break. If the debugger is stopped due to manual intervention, a user breakpoint or when stepping, the frame will be black. When stopping due to a warning, error or exception the frame will be orange, red or thick red respectively, exactly like the error indicators in the run-log.

Note

When the debugger is entered due to a warning, error or exception it will move execution back to the beginning of the node that caused it, giving you a chance to fix the cause of the problem and re-execute that node again. If this is undesirable or impossible you can simply skip the node (see [section 8.2.3^{\(76\)}](#)).

8.2.2 The debugger window

The debugger can be run either from within the normal test-suite view, or by opening a dedicated debugger window by selecting **Debugger→Show debugger window** once the debugger has been entered.

You can also cause the debugger window to open automatically whenever the debugger is entered by setting the option [Always open debugger window^{\(298\)}](#) in the global options dialog or under the **Debugger→Options** menu. If you open or close the debugger window explicitly, this is considered a "manual override" and this option will be ignored for the rest of the test-run.

The debugger window is similar to a normal test-suite window. You can select nodes and edit their attributes, but you cannot delete or insert nodes, there are no file operations and no recorder. For the more complex operations you can quickly jump from the debugger window to the same node in the respective test-suite window by pressing

Ctrl-T or selecting **Find node in test-suite** from the **Edit** menu or the context popup menu.

The lower half of the debugger window shows the nodes that are binding variables on the primary and fallback variable stacks (see [chapter 7^{\(57\)}](#)). For the primary stack all nodes are shown, even if they are not binding any variables. This is useful because it serves as a kind of stack-trace for the current point of execution of the test-run. You can double-click on any node to quickly navigate to the node in its associated test-suite.

A single click on a node brings up its variable bindings in the right half of the variable display. There the variable values can be edited, new variables can be added or existing ones removed. These changes immediately affect the current test-run, but are of a temporary nature. They are not propagated to the nodes in which the variables were bound in the first place.

8.2.3 Debugger commands

Most of the debugger commands are similar to those of any other debugger. However, some additional commands are included that deal with special situations.

Step-wise debugging of a test-suite is available through three operations:

- The step-in button  (**F7**, **Debugger→Step in**) executes the currently selected node and will set the execution mark to the next deepest node, regardless of how deep that node may lie in the node structure. This operation is useful, for example, to step into and debug a Procedure or Sequence.
- The step-over button  (**F8**, **Debugger→Step over**) executes the currently selected node as well as any child nodes that lie under it and then sets the execution mark to the next node at the same level. This is helpful for being able to execute an entire Procedure or Sequence without stepping through each step individually.
- The step-out button  (**Ctrl-F7**, **Debugger→Step out**) executes the currently selected node as well as any other nodes at the same level (including any child nodes of these nodes) and then sets the execution mark to the next node at the next higher level. This type of operation is useful when, for example, you are debugging a Procedure or Sequence and don't want to step through the rest of the nodes in it. By simply using step-out, you can execute the rest of the nodes and return.

The *skip* functions expand the QF-Test debugger in a powerful way which is not typically possible for a debugger in a standard programming environment. In short, they allow you to jump over one or more nodes without having to execute those nodes at all.

- The skip-over button  ((Shift-F9), `Debugger→Skip over`) jumps over the current node without executing it, moving the execution mark to the next node.
- The skip-out button  ((Ctrl-F9), `Debugger→Skip out`) ends the execution of the current Procedure or Sequence and jumps to the next node at the next higher level.

Even more powerful is the ability to continue the test-run at any arbitrary node, even in a completely different test-suite. QF-Test will keep as much of the current execution context as possible, including variable bindings. The closer the new target location is to the current point of execution, the more information can be salvaged.

You can switch execution to a different node by pressing `Ctrl-,` or by selecting the menu item `Run→Continue execution from here` or the respective item in the context menu. When you do so, execution will not continue immediately, only the next node to be executed will change. You can continue the test as usual by single-stepping or resuming the test-run.

The following additional commands are available:

- The rethrow-exception button  (`Debugger→Rethrow exception`) is only active when the debugger was entered due to an exception. It lets you rethrow the exception to be handled by the test-suite just as if the debugger had never caught it in the first place. This is especially useful when the "Break on caught exception" option is set.
- The locate-current-node button  (`Debugger→Locate current node`) quickly moves the selection in the tree-view to the node that is about to be executed. It is a useful shortcut in case you get lost while moving around the test-suite.

8.2.4 Manipulating breakpoints

Setting a breakpoint on a node will tell the debugger to suspend execution just before it enters that node. Breakpoints are displayed in the tree-view by prepending "(B)" to the name of a node.

Breakpoints can be set or removed individually with `Ctrl-F8` or with the `Debugger→Breakpoint on/off` menu item. After finishing a debugging session you can use `Debugger→Clear all breakpoints` to remove any breakpoints that might have been left hanging around. This command will remove all breakpoints from *all* test-suites.

Breakpoints are transient and will not be saved with the test-suite.

Note

Chapter 9

Organizing the test-suite

Creating useful, reliable tests requires more than just recording sequences and playing them back. You can fill a test-suite with lots of sequences in a short time, but you are bound to lose track of what you've got sooner or later. You need to give structure to your tests to make sure that:

- You have a good idea of what you are testing.
- You are testing the right thing.
- Your tests are reliable, repeatable and easy to maintain in case the SUT changes in a way that QF-Test can't adapt to automatically.
- The results of your tests are conclusive.

The essential prerequisite of getting the components right has been discussed in [chapter 5^{\(37\)}](#). Here we are going to concentrate on structuring the actual sequences, events, checks, etc.

9.1 Sequence and Test nodes

The primary building block of a test is the Sequence⁽³³⁴⁾ node which executes its child nodes one by one in the order they appear. The Test⁽³³⁰⁾ node is very similar but adds semantics for setup and cleanup. Before QF-Test version 2 it was the prime representative of a test-case. In that role it has been replaced by the new Test-case⁽³¹⁴⁾ and Test-set⁽³²¹⁾ nodes which are described in the following [section 9.2^{\(80\)}](#). However, Sequence and Test nodes are still very useful low-level building blocks. Their difference lies in their semantics.

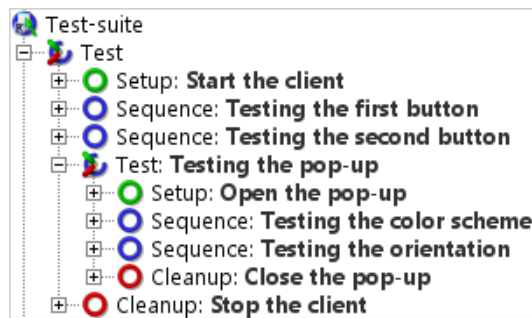


Figure 9.1: Sequences and Tests

The concept behind a Sequence node is that its child nodes must be run in exactly that order and that each node depends on the outcome of the previous node to work correctly. A sequence of recorded events is a very good example for what constitutes a Sequence node. Imagine what would happen if the events were rearranged in a random order.

The semantic contract of the Test node on the other hand is that each of its child nodes is independent of all of the other children, so the order in which they are executed shouldn't really matter. Achieving such a state of independence requires that each child of a Test node can rely on the same set of preconditions. It is the Test node's job to ensure that these conditions are met. To that end, a Test node can have two special additional child nodes, the Setup⁽³⁵⁰⁾ and the Cleanup⁽³⁵²⁾ nodes.

When a Test node is executed, its child nodes are executed one by one in the order they appear. We said this before, but it is a half truth only. Every time it executes one of its normal children, the Test node first runs the Setup node which is responsible for providing the settings that the child node requires. Whenever a child node is done, the Cleanup node is executed to clean up any mess in the SUT the child node may have caused.

This may sound needlessly complex, but implementing all the preconditions of a test-case and isolating test-cases so that they are completely independent are so important that in QF-Test version 2 Setup and Cleanup nodes have essentially been replaced by an even more powerful mechanism based on the new Dependency⁽³⁴⁴⁾ nodes which are described in chapter 12⁽⁹⁰⁾. But as Tests still have their use at lower levels, so do Setup and Cleanup nodes. A Test node with well-matched Setup and Cleanup nodes has the following important qualities:

- You can add child nodes to the Test at any time and any position without disturbing the other child nodes.
- When you want to extend a test or just try out some sequence, you don't need to execute all the previous steps first to get the SUT into the proper state.
- You can run any sub-set of a Test node if you don't want to run the full test.

- If one child of a Test node fails for whatever reason, the rest of its children can still be executed, provided that the Setup and Cleanup node can be relied on to provide the proper settings under these circumstances. In a Sequence, where each node depends on the outcome of its predecessor, the failure of one node will invalidate the whole sequence.

The Implicitly catch exceptions⁽³³³⁾ attribute of a Test node helps to facilitate error handling of the kind described above.

9.2 Test management with Test-set and Test-case nodes

The Test-set⁽³²¹⁾ and Test-case⁽³¹⁴⁾ nodes provide a small-scale, pragmatic form of test management right inside QF-Test. Their main feature is the smart dependency management described in the following chapter 12⁽⁹⁰⁾ that allows Test-cases to be implemented completely independent from each other. With properly written Dependencies, cleanup of the SUT from previously executed tests is handled automatically along with the setup for the next test and all error handling.

9.2.1 Concepts

Conceptually a Test-case node represents a single elementary test case. As such it is the main link between test planning, execution and result analysis. With the help of Dependencies, Test-cases can be isolated from each other so that they can be run in any arbitrary order. QF-Test automatically takes care of the necessary test setup. Cleanup is also automatic and will be performed only when necessary in order to minimize overhead in the transition from one test to the next. This enables things like running subsets of functional test-suites as build tests or retesting only failed Test-cases.

Test-sets basically are bundles of Test-cases that belong together and typically have similar requirements for setup and cleanup. Test-sets can be nested. The whole structure of Test-sets and Test-cases is very similar to Package and Procedure nodes. The Test-suite root node can be considered a special kind of Test-set.

Test-suite, Test-set and Test-case nodes can be called from other places using a Test call node. That way, tests that run only a subset of other tests can easily be created and managed. Test call nodes are allowed everywhere, but should not be executed from inside a Test-case node because that would break the atomicity of a Test-case from the report's point of view. A warning is issued if Test-case execution is nested.

9.2.2 Variables and special attributes

As both Test-sets and Test-cases can be called from a Test call node they each have a set of default parameters similar to those of a Procedure⁽³⁷⁸⁾. These will be bound on the fallback stack and can be overridden in the Test call node. A Test-case has an additional set of variable bindings similar to those of a Test⁽³³⁰⁾ node. These are hard bindings for the primary variable stack that will be defined during the execution of the Test-case and cannot be overridden in a Test call node.

The list of Characteristic variables is a set of names of variables that are part of the characteristics of the test for data-driven testing. Each execution of the Test-case with a different set of values for these variables is considered a separate test case. The expanded values of these variables are shown in the run-log and report for improved error analysis.

Another useful attribute is the Condition which is similar to the Condition⁽³⁹⁷⁾ of an If⁽³⁹⁶⁾ node. If the Condition is not empty, the test will only be executed if the expression evaluates to true. Otherwise the test will be reported as skipped.

Sometimes a Test-case is expected to fail for a certain period of time e.g. when it is created prior to the implementation of the respective feature or before a bug-fix is available in the SUT. The Expected to fail if... attribute allows marking such Test-cases so they are counted separately and don't influence the percentage error statistics.

9.2.3 Migration and backwards compatibility

For backwards compatibility and to ease transition from old-style Test nodes to Test-set and Test-case nodes QF-Test treats nodes as a Test-set or Test-case for documentation and report if their place in the hierarchy allows it.

Old test-suites with a structure based on Test nodes can be migrated to make use of the new features of Test-sets and Test-cases. To this end, right-click on a Test node to bring up the context menu. If a transformation is allowed, QF-Test will offer to transform the Test node into a Test-set or Test-case node.

3.0+

It is possible to convert a whole hierarchy of Test nodes to a hierarchy of Test-set and Test-case nodes by selecting the recursive conversion option in the popup menu.

Note

Both Test-set and Test-case nodes may contain Setup or Cleanup nodes for backwards compatibility. In a Test-set, these work just as in a Test: Setup and Cleanup are executed for each test contained in the Test-set. In a Test-case however, Setup and Cleanup are only run once at the beginning and end of its execution. If a Test-set or Test-case has both a Dependency and Setup/Cleanup nodes, the Dependency will be executed first. Setup and Cleanup will have no impact on the dependency stack described in section 12.2⁽⁹¹⁾.

9.3 Procedures and Packages

In a way, writing good tests is a little like programming. After mastering the initial steps, tests and source code alike tend to proliferate. Things work fine until some building block that was taken for granted changes. Without a proper structure, programs as well as tests tend to collapse back upon themselves at this point as the effort of adapting them to the new situation is greater than the one needed for recreating them from scratch.

The key to avoiding this kind of problem is reuse or avoidance of redundancy. Generating redundancy is one of the main dangers of relying too much on recording alone. To give an example, imagine you are recording various sequences to interact with the components in a dialog. To keep these sequences independent of each other, you start each one by opening the dialog and finish it by closing the dialog again. This is good thinking, but it creates redundancy because multiple copies of the events needed to open and close the dialog are contained in these sequences. Imagine what happens if the SUT changes in a way that invalidates these sequences. Let's say a little confirmation window is suddenly shown before the dialog is actually closed. Now you need to go through the whole suite, locate all of the sequences that close the dialog and change them accommodate the confirmation window. Pure horror.

To stress the analogy again, this kind of programming style is called *Spaghetti Programming* and it leads to the same kind of maintenance problems. These can be avoided by collecting the identical pieces in one place and referring to them wherever they are needed. Then the modifications required to adapt to a change like the one described above are restricted to this place only.

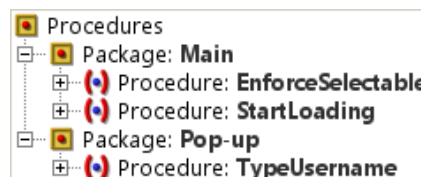


Figure 9.2: Packages and Procedures

QF-Test comes with a set of nodes that help to achieve this kind of modularization, namely the Procedure⁽³⁷⁸⁾, Procedure call⁽³⁸¹⁾ and Package⁽³⁸⁵⁾ nodes. A Procedure is similar to a Sequence⁽³³⁴⁾ except that its Name⁽³⁷⁹⁾ attribute is a handle by which a Procedure call node can refer to it. When a Procedure call is executed, the Procedure it refers to is looked up and execution continues there. Once the last child node of the Procedure has finished, the Procedure call has completed as well.

Packages are just a way to give even more structure to Procedures. A hierarchy of Packages and Procedures, rooted at the special Procedures⁽³⁸⁷⁾ node, is used to group sets of Procedures with a common context together and to separate them from other Procedures

used in different areas.

A Procedure that always does exactly the same, no matter where it is called from, is only marginally useful. To expand on the above example, let's say we want to extend the Procedure that opens the dialog to also set some initial values in some of its fields. Of course we don't want to have these initial values hard-coded in the Procedure node, but want to specify them when we call the Procedure to get different values in different contexts. To that end, parameters can be defined for the Procedure. When the Procedure call is executed, it specifies the actual values for these parameters during this run. How all of this works is explained in [Variables](#)⁽⁵⁷⁾. Also please take a look at the detailed explanation for the [Procedure](#)⁽³⁷⁸⁾ and [Procedure call](#)⁽³⁸¹⁾ nodes for a better understanding of how these complement each other.

A test-suite library with a set of commonly useful Procedures is provided with QF-Test under the name `qfs.qft`. An entire chapter of the Tutorial is devoted to this library and [section 17.1](#)⁽¹⁵⁰⁾ explains how to include it in your test-suites.

9.3.1 Local Procedures and Packages

If you work with several test-suite libraries you might face a situation, where you define reusable test-steps or sequences, which you only want to use within a dedicated test-suite. If you want to create such local Procedures, you can put a `'_'` as first sign of the procedure's name. This marks a Procedure as test-suite local.

A call of a local Procedure can only be inserted within the test-suite, where it is defined. You can use the same concept for local Packages.

9.3.2 Relative Procedures

If you call Procedures from other Procedures, it could be convenient not to specify the full procedure name all the time.

So called 'relative' procedure calls can only be added to a Package, which has the Border for relative calls (see [Border for relative calls](#)⁽³⁸⁶⁾) attribute specified. The structure of that call follows the concept below:

Level	Call
Procedures of the same level	.Name of Procedure
Procedures one level higher	..Name of Procedure
Procedures one level deeper	.Name of Package.Name of Procedure

Table 9.1: Relative procedure calls

As you can see each dot stands for one level. So calling a Procedure two levels higher requires three dots (Current level also requires a dot.)

9.3.3 Inserting Procedure call nodes

As you should organize your tests in separate test steps, which are ideally the same like QF-Test's procedures, QF-Test offers several ways to insert those Procedure call nodes:

1. Via the menu `Insert→Procedures→ProcedureCall`
2. Via right mouse click and selecting `Insert node→Procedures→ProcedureCall`
3. Copy a Procedure node and insert it at the location of the Procedure call using the normal Copy/Paste actions
4. Via Drag&Drop operation, i.e. dragging the Procedure node to its target node
5. Via the keyboard shortcut `Ctrl-A`

This approach is also valid for inserting Dependency reference nodes, except the keyboard shortcut.

9.3.4 Parameterizing nodes

You can create parameters for a Procedure, Dependency or Test-case automatically via the menu `Operations→Parameterize node`.

The parameter details dialog allows you to define for which actions you want to create parameters, e.g. only text-inputs or check nodes.

9.3.5 Transforming a Sequence into a Procedure

This transformation is very useful for developing procedures immediately after recording! Under Extras you can convert a recorded Sequence node into a Procedure and move that to the Procedures node.

If you transform a Sequence under Test-cases QF-Test automatically creates a Procedure node and inserts a Procedure call to the previous location of the transformed node.

9.4 Documenting test-suites

Like with any programming-related task it is important for successful test-automation to properly document your efforts. Otherwise there is a good chance (some might say a certainty) that you will lose the overview over what you have done so far and start re-implementing things or miss out tests that should have been automated. Proper documentation will be invaluable when working through a run-log, trying to understand the cause of a failed test. It will also greatly improve the readability of test reports.

Based on the Comment⁽³²⁶⁾ attributes of Test-set⁽³²¹⁾, Test-case⁽³¹⁴⁾, Package⁽³⁸⁵⁾ and Procedure⁽³⁷⁸⁾ nodes, QF-Test can create a set of comprehensive HTML documents that will make all required information readily available. The various kinds of documents and the methods to create them are explained in detail in chapter 15⁽¹²⁹⁾.

Chapter 10

Projects

3.5+

Projects provide a better overview, improve navigation between test-suites and expand the scope for search and replace operations. Also, QF-Test automatically manages dependencies resulting from includes or absolute references between test-suites that belong to the same project (see section 17.1⁽¹⁵⁰⁾). Many other features have already been implemented or are under development.

Technically a QF-Test project is a set of test-suites located in one or more directories with a common root. There is a 1:1 relation between the project and its directory and the name of the directory automatically becomes the name of the project.

To create a new project, select the menu item File→New project... and choose the directory. QF-Test then creates a file called `qftest.qpj` in that directory which identifies it as a project. All test-suites located below that directory, except those specified in the option Project files and directories to exclude⁽²⁴²⁾ automatically belong to this project. Please see section 29.1.1⁽²⁴¹⁾ for options affecting projects, including the exclusion list.

A sub-project is sub-directory of a project which is itself a project. Test-suites within a sub-project also belong to all outer projects containing the sub-project. *The* project of a test-suite is the innermost sub-project it belongs to. Automatic dependency resolution always covers the whole outermost project of a suite including all sub-projects.

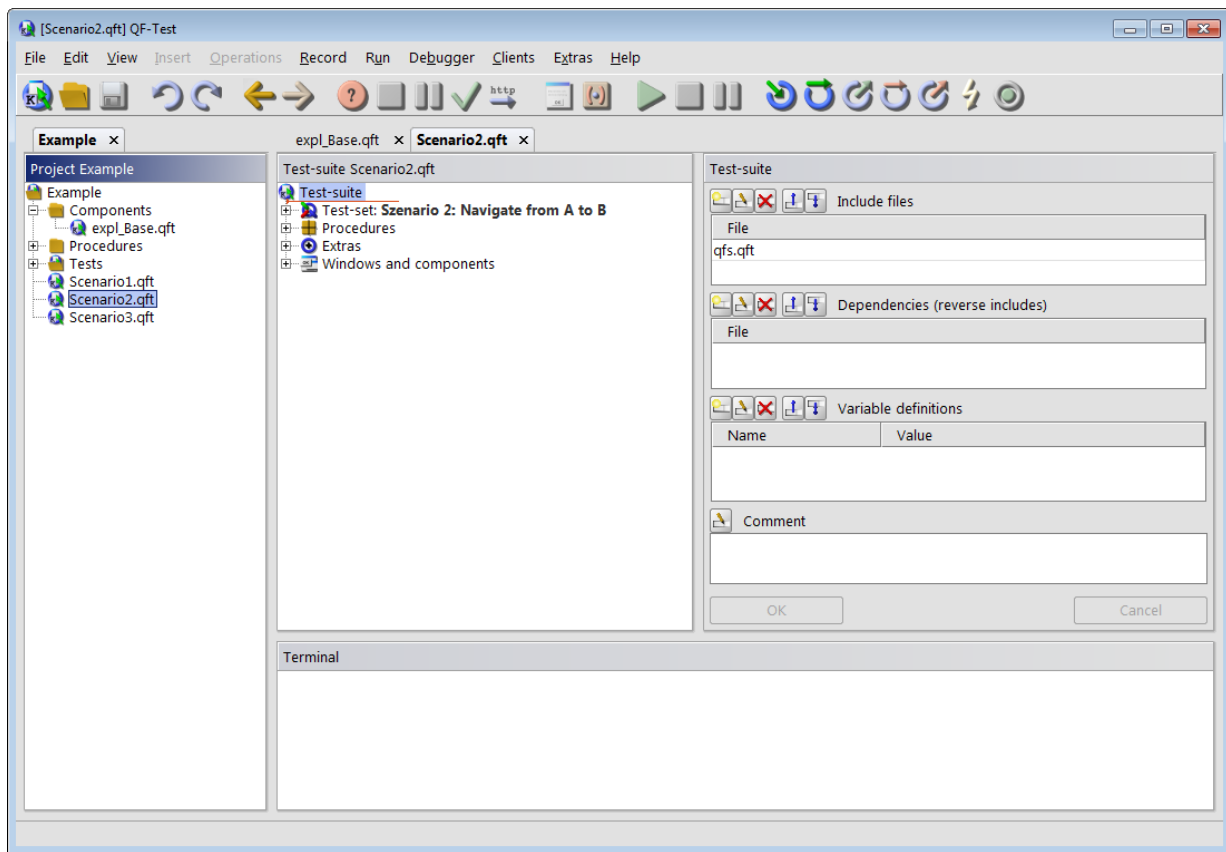


Figure 10.1: The project view

The project view with one or more projects can be turned on or off via the menu item **View→Show projects**. The project tree shows the hierarchy of directories and test-suites starting from the project root. Double clicking a test-suite opens it, as does pressing the **Return** key. You can select several files or directories to be opened in one go, including all test-suites located below the selected directories.

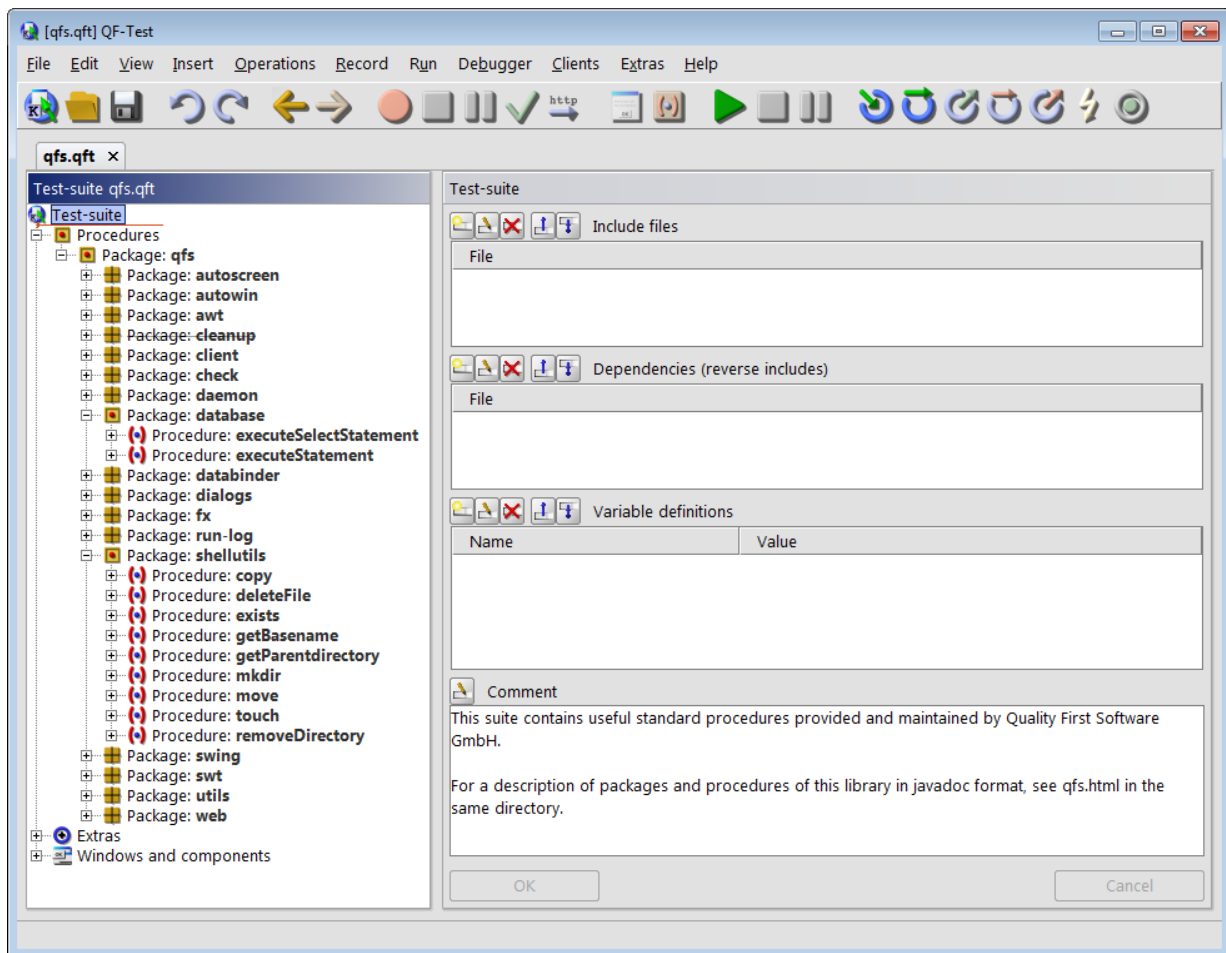
The hierarchy is refreshed automatically at intervals defined in the option **Project refresh interval (s)**⁽²⁴¹⁾. You can refresh a directory including its complete hierarchy at any time by selecting it and pressing **F5**. For a more thorough rescan that does not rely on modification times but may take significantly longer for large projects, press **Shift-F5** instead.

To switch keyboard focus back and forth between the test-suite and the project view, press **F6**. Via **Shift-F6** you can navigate to the node representing the current test-suite in the project tree. If necessary, project view and project are automatically shown.

Chapter 11

The standard library

The standard library `qfs.qft`, a test-suite that is part of the QF-Test distribution, contains many useful procedures for a diverse set of tasks.

Figure 11.1: Standard library `qfs.qft`

Among others there are procedures for accessing and checking components (AWT, Swing, JavaFX, SWT, Web) in a generic manner, file system and database access, logging messages or screenshots to the run-log and report and for performing cleanup.

A complete description of all packages and procedures including parameters and return values is provided in the library's HTML documentation, also accessible from the QF-Test **Help** menu. The latest version is also available online.

`qfs.qft` is included by default in every newly created test-suite. As its directory is on the library path⁽²⁵²⁾, specifying just `qfs.qft` in the Include files⁽³¹²⁾ of the Test-suite node is sufficient.

Note

All procedures referring to an SUT use the generic variable `$(client)` as an implicit parameter. You must make sure that this variable is set correctly either globally or locally or specified as an explicit parameter in the procedure call.

Chapter 12

Dependencies

Dependencies⁽³⁴⁴⁾ are a very advanced feature, albeit a little complex. You should have a reasonably good grasp of QF-Test, especially for things like control flow, variable binding and error handling, before you start using them. However, when properly implemented, Dependencies will feel almost like magic when you run several non-related Test-cases and all setup and cleanup is handled automatically. Dependencies are also crucial for running tests in a QF-Test Daemon as described in chapter 40⁽⁷⁴¹⁾.

To illustrate the following sections, two test-suites `doc/tutorial/dependencies.qft` and `demo/carconfig/carconfig_en.qft` are provided. The former illustrates the various concepts for Dependencies and the latter is a simple real-world example. The SWT demo test-suite `swt_addressbook.qft` and the data driver `demo_datadrivers.qft` also make use of Dependencies. Single-stepping through these suites in the debugger, looking at the variable bindings and examining the run-logs should help you to familiarize yourself with this feature.

12.1 Dependency basics

Conceptually a Dependency describes a set of preconditions that have to be met before a test can be executed. These preconditions are often dependent on each other, for example parameter settings may have to be read which are required for starting the SUT. The target Window for a specific test-case cannot be opened before the SUT is running, and so on.

Dependencies can be defined in two places: Generic Dependencies that will often be reused or serve as the basis for other Dependencies can be implemented just like a Procedure and placed below the Procedures⁽³⁸⁷⁾ node, for example inside a Package⁽³⁸⁵⁾ node called "Dependencies". Their fully qualified name is built analogous to a Procedure name and Dependencies can be referenced using a Dependency reference almost exactly like a

Procedure call⁽³⁸¹⁾.

Alternatively, Dependencies can be implemented at the beginning of a Test-suite, Test-set or Test-case node. In addition to having their own Dependency, Test-cases and Test-sets can inherit the Dependency from their parent node.

Each Dependency should always take care of only one specific precondition and refer to other Dependencies to ensure that the more basic preconditions on which it depends are taken care of. This is done implicitly by inheriting the Dependency of a parent node or explicitly with a Dependency reference.

The actual implementation is done in Setup and Cleanup nodes inside the Dependency. The Setup node should always be implemented in a way that ensures the minimum of effort to fulfill the requirement. For example, the Setup to start the SUT should first check whether the SUT is already connected or not and execute the SUT startup sequence only in the latter case. The reason for this will be explained below. Setup and Cleanup nodes should always be prepared to handle errors that do not affect the outcome. For example, the Cleanup node that terminates the SUT should terminate cleanly if the SUT is already disconnected.

12.2 The dependency stack

When a Dependency is executed, QF-Test linearizes the Dependency and its ancestors to create a dependency stack. For example, if Dependency D depends on Dependencies B and C which in turn depend on Dependency A, the stack will look like [A,B,C,D]. If this is the first Dependency to be executed, there will be no previous dependency stack and QF-Test will simply execute the Setup nodes of the Dependencies from most basic to most specific, i.e. A-B-C-D.

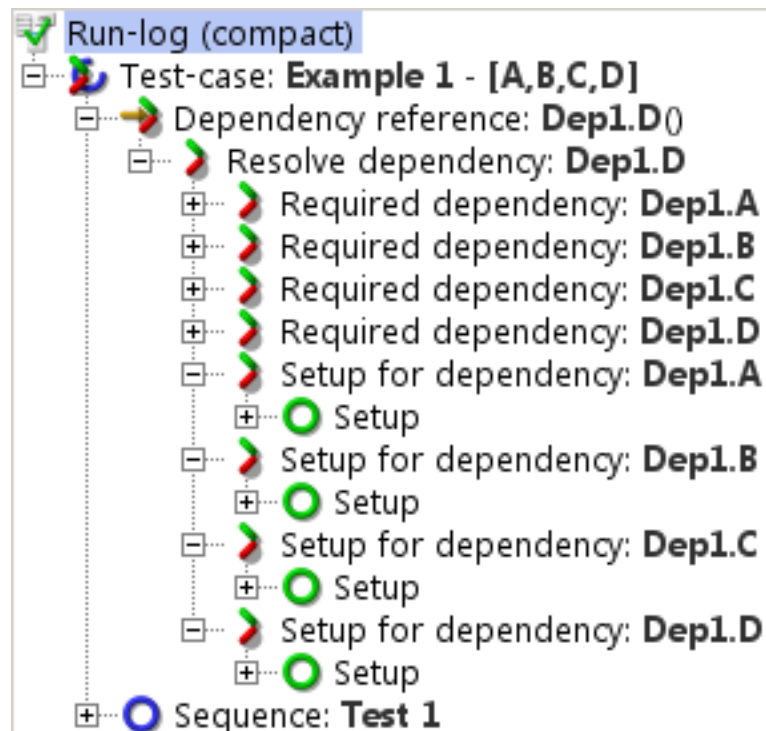


Figure 12.1: Dependency stack A-B-C-D

After the Test-case that required the Dependency has been executed, part of the dependency stack may be rolled back. If any of the Dependencies on the stack has the Forced cleanup attribute set, it and all of the following Dependencies will have their Cleanup sequences executed, this time in the opposite order. If, in the example above, C has Forced cleanup set, first D's Cleanup node will be run, then C's Cleanup node. After that, the remaining dependency stack will look like [A,B].

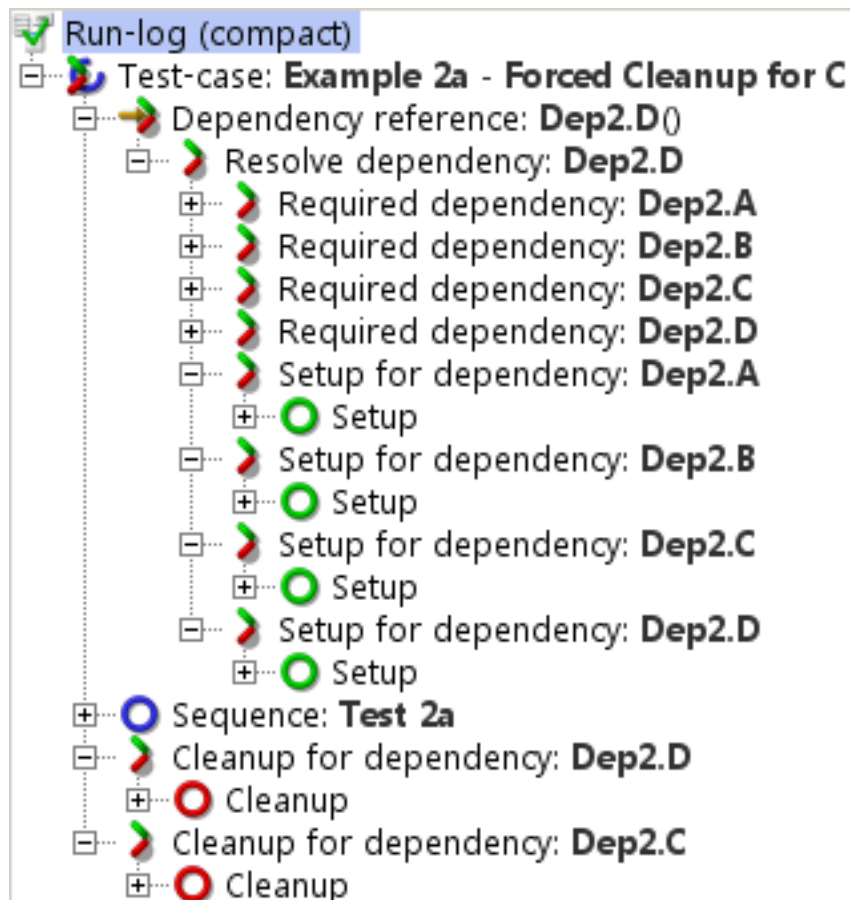


Figure 12.2: Dependency C has "forced cleanup"

Now let's say another Test-case is executed that depends on a Dependency E which in turn depends on A, so the new target dependency stack is [A,E]. Now QF-Test compares the current dependency stack [A,B] to the new target stack [A,E] and finds the first point of difference, which is B vs. E in this case. Next the old dependency stack will be unrolled to this point from right to left, so B's Cleanup node is executed. The current stack now looks like [A]. Then dependency E is added which results in [A,E].

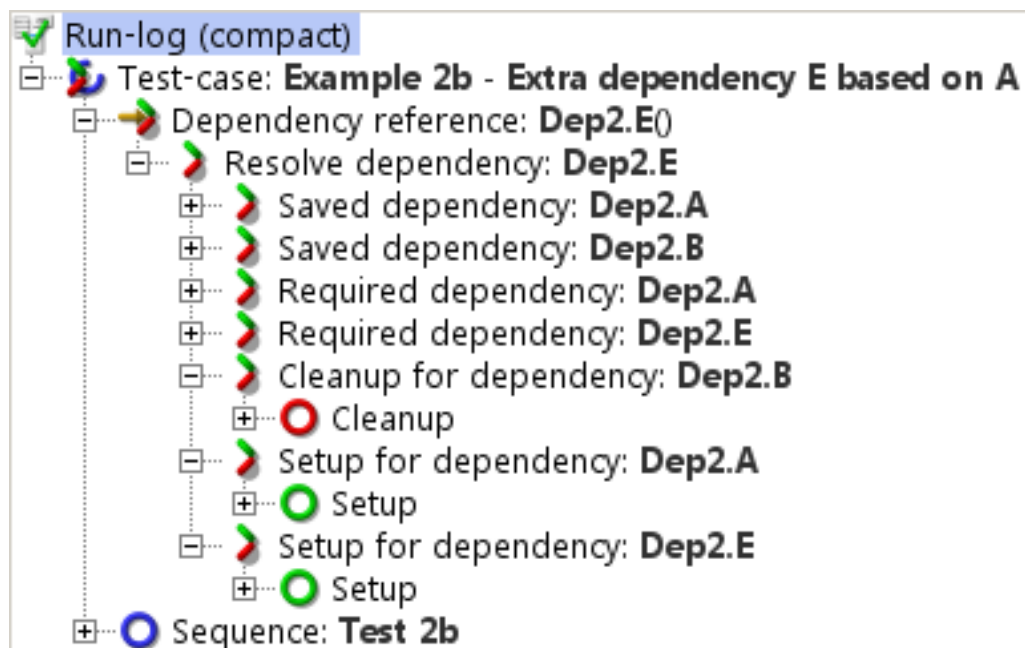


Figure 12.3: Rollback and build up stack to A-E

The next point is crucial: To get from the cleaned up old dependency stack [A] to the new target stack [A,E], QF-Test doesn't simply execute the Setups of the rest of the new dependency stack - E in this case. Instead, the Setup nodes of the whole new target stack are executed, here first A's Setup node and then E's. This repeated execution is the main reason why Setup nodes should always be implemented to do as little as possible.

Why such a weird asymmetry? Let's assume that A is the Dependency responsible for bringing up the SUT. In the normal case for the previous example, the SUT will already be running and if A's Setup is properly implemented it will notice that and there will be very little overhead. However, QF-Test has no control over what has happened since the last time A's setup was run. You might have closed the SUT manually or it might have been closed as a side effect of the previous test or even crashed. In this case, if A's setup were not run again, E's precondition could never be satisfied and the whole Test-case would fail along with all subsequent tests that also rely directly or indirectly on Dependency A. Thus, always executing all Setup nodes on the target dependency stack is the only way to ensure proper setup in all circumstances.

If get stuck or lost you can clear the dependency stack manually in two ways: By selecting the **Run→Rollback dependencies** or the **Run→Reset dependencies** menu item. The former will completely roll back the dependency stack, executing the Cleanup nodes of all Dependencies that were left on the stack, while the latter will simply clear the stack without executing anything.

12.3 Error escalation

Another thing that is just grand about Dependencies is the convenient way that errors can be escalated without any additional effort. Let's again consider the example from the previous section after the first dependency stack has been initialized to [A,B,C,D] and the Setups have been run. Now what happens if the SUT has a really bad fault, like going into a deadlock and not reacting to user input any longer?

When a Cleanup node fails during rollback of the dependency stack, QF-Test will roll back an additional Dependency and another one if that fails again and so on until the stack has been cleared. Similarly, if one of the Setups fails, an additional Dependency is rolled back and the execution of the Setups started from scratch. In the example above, the deadlocked client would cause failures until the rollback of the SUT Dependency terminates the SUT and causes a new one to be started in the next round.

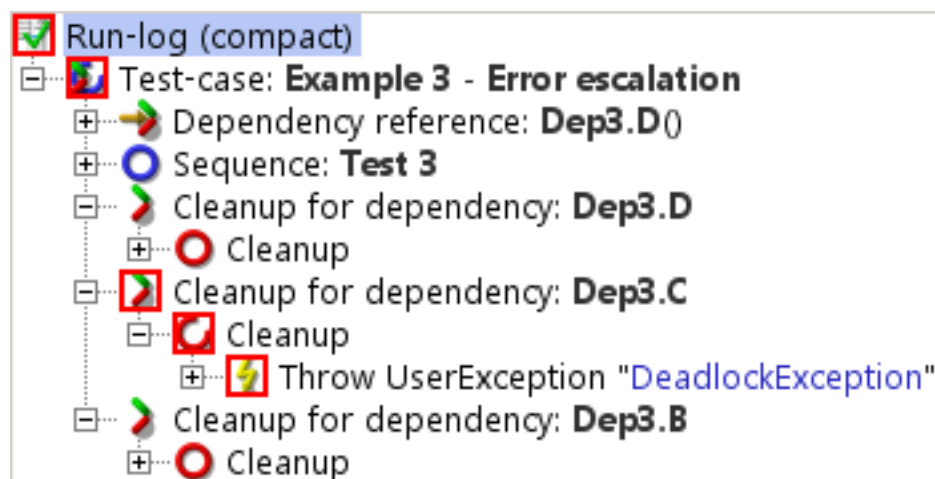


Figure 12.4: Exception in forced cleanup sequence of C causes B to clean up

For this to work it is very important to write Cleanup sequences in a way that ensures that either the desired state is reached or that an exception is thrown and that there is a more basic dependency with a more encompassing Cleanup. For example, if the Cleanup node for the SUT Dependency just tries to cleanly shut down the SUT through its File->Exit menu without exception handling and further safeguards, an exception in that sequence will prevent the SUT from being terminated and possibly interfere with all subsequent tests. Instead, the shutdown should be wrapped in a Try/Catch with a Finally node that checks that the SUT is really dead and if not, kills the process as a last resort.

With good error handling in place, Test-cases will rarely interfere with each other even in case of really bad errors. This helps avoid losing a whole night's worth of test-runs just because of a single error.

12.4 Special variables

A Dependency can in turn depend on the values of certain variables. For example, you may want to test your SUT with different JDK versions. Thus, the JDK would be a characteristic variable for a Dependency that brings up the SUT. If you have a current dependency stack [A] and the target stack is also [A], but last time A was executed with one JDK value and now some other JDK is requested, QF-Test must roll back the Dependency to terminate the current SUT and ensure that a new one with the proper JDK is started.

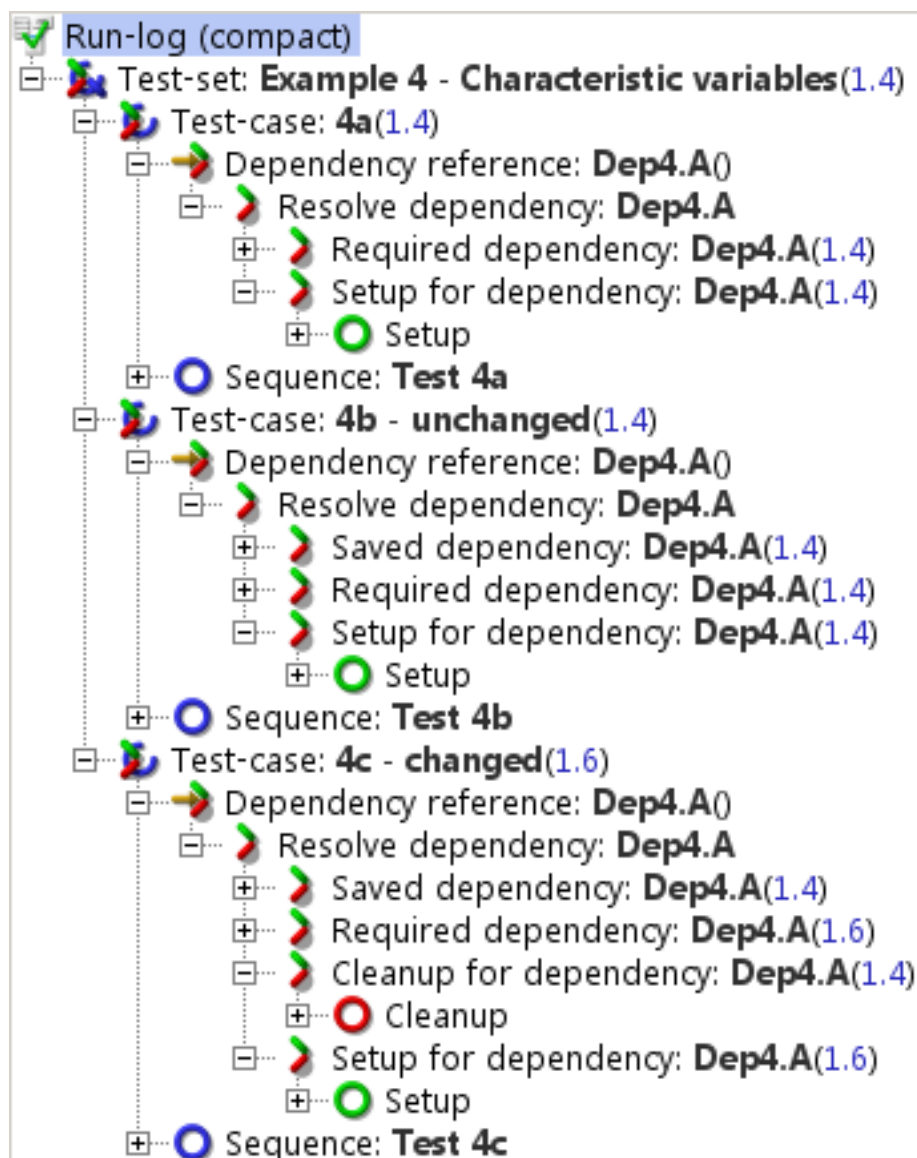


Figure 12.5: Change in characteristic variable causes cleanup of A

All this is handled fully automatically if you add the JDK variable to the list of Characteristic variables of the Dependency. The values of the Characteristic variables are always taken into account when comparing dependency stacks and two Dependencies on the stack are only considered identical if the values of all Characteristic variables from the previous and the current run are equivalent. Consequently it is also possible for a Dependency to directly or indirectly refer to the same base Dependency with different values for its Characteristic variables. In that case the base Dependency will appear multiple times in the linearized dependency stack.

Furthermore, QF-Test stores the values of the Characteristic variables during execution of the Setup of a Dependency. When the Dependency is rolled back, i.e. its Cleanup node is executed, QF-Test will ensure that these variables are bound to the same value as during execution of the Setup. This ensures that a completely unrelated Test-case with conflicting variable definitions can be executed without interfering with the execution of the Cleanup nodes during Dependency rollback. Consider for example the commonly used "client" variable for the name of an SUT client. If a set of tests for one SUT has been run and the next test will need a different SUT with a different name, the "client" variable will be changed. However, the Cleanup node for the previous SUT must still refer to the old value of "client", otherwise it wouldn't be able to terminate the SUT client. This is taken care of automatically as long as "client" was added to the list of Characteristic variables.

12.5 Error handling

Besides supporting automatic escalation of errors a Dependency can also act as an error or exception handler for the tests that depend on it. Catch nodes, which can be placed at the end of a Dependency, are used to catch and handle exceptions thrown during a test. Exceptions thus caught will still be reported as exceptions in the run-log and the report, but they will not interfere with subsequent tests or even abort the whole test-run. This is very similar to the Implicitly catch exceptions⁽³³³⁾ attribute of Test⁽³³⁰⁾ nodes, but more specific.

An Error handler node is another special node that may be added to a Dependency after the Cleanup and before the Catch nodes. It will be executed whenever the result of a Test-case is "error". In case of an exception, the Error handler node is not executed automatically because that might only cause more problems and even interfere with the exception handling, depending on the kind of exception. To do similar things for errors and exception, implement the actual handler as a Procedure and call it from the Error handler and the Catch node. Error handlers are useful for capturing and saving miscellaneous states that are not automatically provided by QF-Test. For example, you may want to create copies of temporary files created during execution of your SUT that may hold information pertaining to the error.

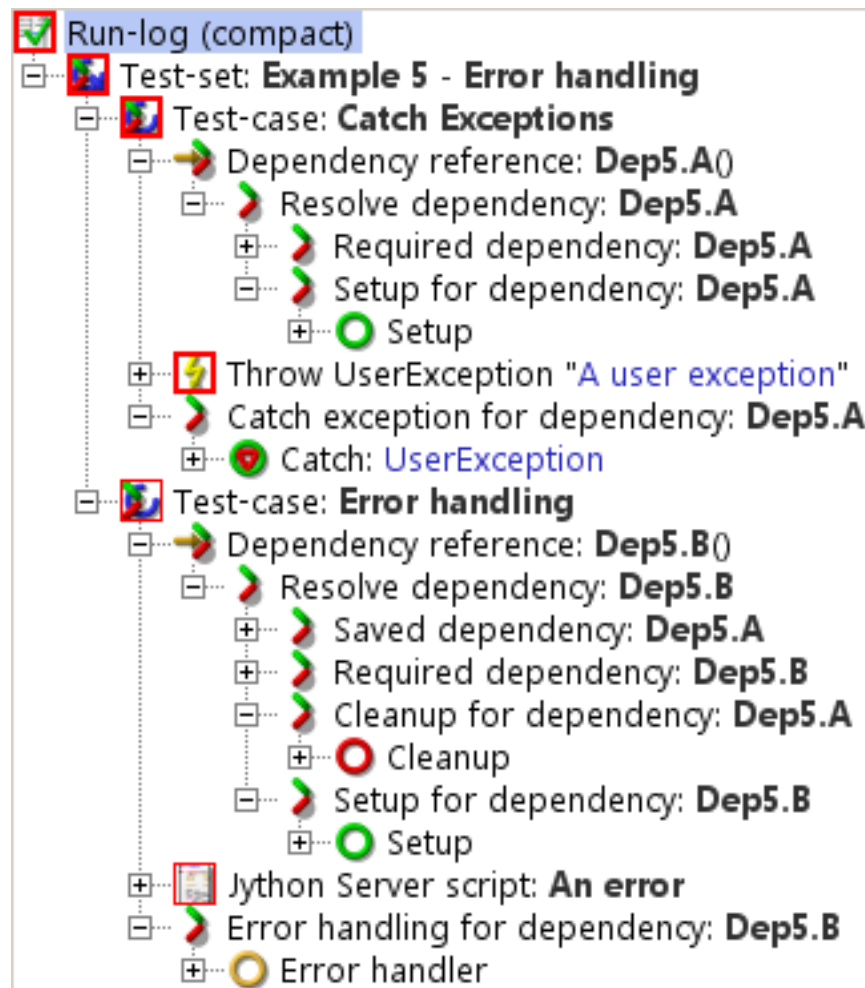


Figure 12.6: Execution of Catch and Error handler nodes

Only the topmost Error handler that is found on the dependency stack is executed, i.e. if in a dependency stack of [A,B,C,D] both A and C have Error handlers, only C's Error handler is run. Otherwise it would be difficult to modify the error handling of the more basic Dependency A in the more specialized Dependency C. To reuse A's error handling code in C, implement it as a Procedure.

Chapter 13

Scripting

One of QF-Test's benefits is that complex tests can be created without writing a single line of code. However, there are limits to what can be achieved with a GUI alone. When testing a program which writes to a database, for example, one might want to verify that the actual values written to the database are correct; or one might want to read values from a database or a file and use these to drive a test. All this and more is possible with the help of powerful scripting languages like Jython and Groovy.

3.0+

While Jython is supported since the beginning of QF-Test, Groovy has found its way into QF-Test a bit later (QF-Test version 3). This language might be more convenient than Jython for those who are familiar with Java. It's mainly a matter of individual preference whether to utilize Jython or Groovy scripting inside QF-Test.

In this chapter the basics of the scripting integration are explained in detail with Jython. Most of that also holds true for Groovy, especially QF-Test's *run-context* methods are the same for both languages. The peculiarities of Groovy will be explained in the section Scripting with Groovy⁽¹¹⁵⁾.

3.0+

The scripting language to use for a given Server script⁽⁴¹⁶⁾ or SUT script⁽⁴¹⁹⁾ node is determined by its Script language⁽⁴¹⁸⁾ attribute, so you can mix both languages inside a test-suite. The default language to use for newly created script nodes can be set via the option Default script language for script nodes⁽²⁴⁰⁾.

13.1 Fundamentals of the Jython integration

Python is an excellent, object oriented scripting language written in C by Guido van Rossum. A wealth of information including an excellent Python tutorial is available at <http://www.python.org>. Python is a standard language that has been around for years with extensive freely accessible documentation. Therefore this manual only explains how Jython is integrated into QF-Test, not the language itself. Python is a very natural

language. Its greatest strength is the readability of Python scripts, so you should have no problems following the examples.

Jython (formerly called JPython) is a Java implementation of the language Python. It has the same syntax as Python and almost the same set of features. The object systems of Java and Jython are very similar and Jython can be integrated seamlessly into applications like QF-Test. This makes it an invaluable tool for Java scripting. Jython has its own web page at <http://www.jython.org>. There is also an extensive tutorial available which may help you getting started with this scripting language.

QF-Test uses Jython version 2.5 which supports a large majority of the standard Python library.

The approach to scripting in QF-Test is inverse from that of other GUI test tools. Instead of driving the whole test from a script, QF-Test embeds scripts into its own test-suite. This is achieved with the two nodes Server script⁽⁴¹⁶⁾ and SUT script⁽⁴¹⁹⁾.

Both nodes have a Script⁽⁴¹⁷⁾ attribute for the actual code.

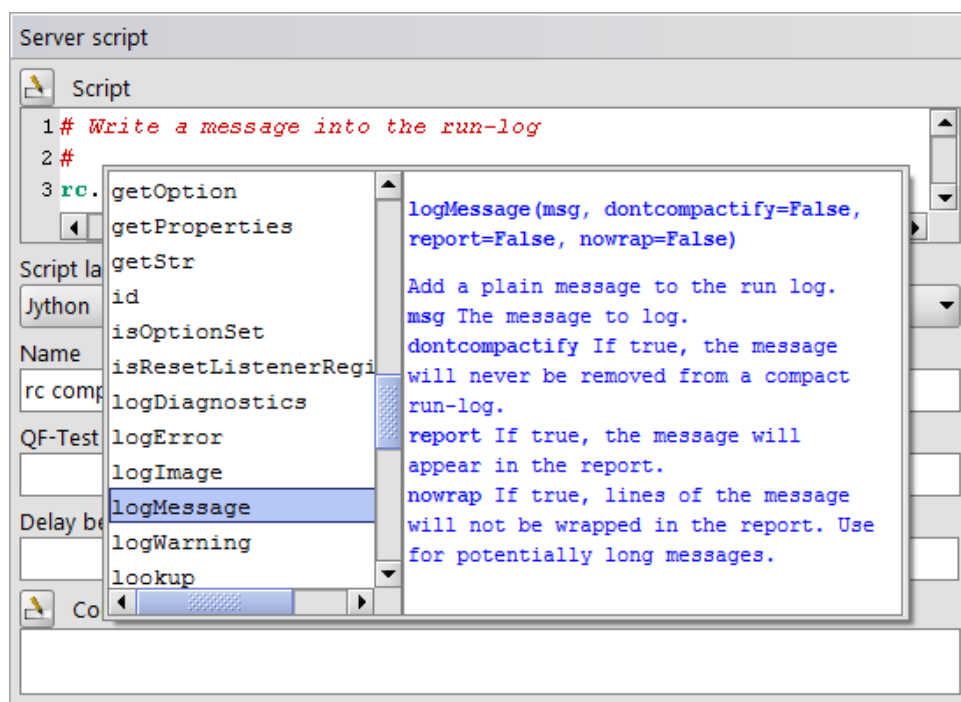


Figure 13.1: Detail view of a Server script with help window for `rc` methods

3.0+

The internal script editor has some useful features to ease the typing of code. Reserved key words, built-in functions, standard types, literals and comments are highlighted. Indentation is handled automatically inside of code blocks. With **[TAB]** and **[Shift-TAB]** respectively several selected lines can be indented manually.

However, the probably most useful feature - at least for the QF-Test newbie - might be the input assistance for many built-in methods. Type, for example, `rc.` and maybe some initial letters of a method name. Then press **Ctrl-Space** to make a popup window displaying the appropriate methods and descriptions of QF-Test's *run-context* (cf. chapter 37⁽⁶¹⁹⁾). Select one of the methods and confirm with **Return** to insert it into the script code. To get a list of all objects equipped with help, just press **Ctrl-Space** with the mouse cursor positioned after whitespace.

Server scripts are useful for tasks like calculating the values of variables or reading and parsing data from a file and using that to drive a test. SUT scripts on the other hand give full access to the components of the SUT and to every Java API that the SUT exposes. An SUT script might be used to retrieve or check values in the SUT to which QF-Test doesn't have access. The SUT script node additionally requires a `Client`⁽⁴¹⁹⁾ attribute for the name of the SUT client to run in.

Server scripts are run in a Jython interpreter embedded in QF-Test itself, while SUT scripts are run in a Jython interpreter embedded in the SUT. These interpreters are independent of each other and do not share any state. However, QF-Test uses the RMI connection between itself and the SUT for seamless integration of SUT scripts into the execution of a test.

Through the menu **Extras→Jython terminal...** you can open a window with an interactive command prompt for the Jython interpreter embedded into QF-Test. You can use this terminal to experiment with Jython scripts, get a feel for the language, but also to try out some sophisticated stuff like setting up database connections. The keystrokes **Ctrl-Up** and **Ctrl-Down** let you cycle through previous input and you can also edit any other line or mark a region in the terminal and simply press **Return** to send it to the Jython interpreter. In that case QF-Test will filter the `»>` and `'...'` prompts from previous interpreter output.

Similar Jython terminals are available for each SUT client. The respective menu items are located below the **Clients** menu.

Note

When working in a Jython terminal, there's one thing you need to be aware of: The commands issued to the interpreter are not executed on the event dispatch thread. This may not mean anything to you and most of the time it doesn't cause any problems, but it may deadlock your application if you access any Swing or SWT components or invoke their methods. To avoid that, QF-Test provides the global method `runAWT` (and `runSWT` respectively) that executes arbitrary code on the event dispatch thread. For example, to get the number of visible nodes in a `JTree` component named `tree`, use `runAWT("tree.getRowCount()")` (or `runAWT { tree.getRowCount() }` in Groovy) to be on the safe side.

13.2 The run-context `rc`

When executing Server scripts and SUT scripts, QF-Test provides a special environment in which a local variable named `rc` is bound. This variable represents the *run-context* which encapsulates the current state of the execution of the test. It provides an interface (fully documented in [section 37.6^{\(622\)}](#)) for accessing QF-Test's variables, for calling QF-Test Procedures and can be used to add messages to the run-log. To SUT scripts it also provides access to the actual Java components of the SUT's GUI.

For those cases where no run-context is available, i.e. Resolvers, TestRunListeners, code executing in a background thread etc. QF-Test also provides a module called `qf` with useful generic methods for logging and other things. Please see [section 37.7^{\(639\)}](#) for details.

13.3 Scripting examples

Probably the best way to learn about Jython and QF-Test is through examples, so we're going to provide a few here. Full technical background and a comprehensive API reference are available in [chapter 37^{\(619\)}](#).

Working examples are also provided in the test-suite `doc/tutorial/demo-script.qft`.

13.3.1 Logging messages

One thing the run-context can be used for is to add arbitrary messages to the run-log that QF-Test generates for each test-run. These messages may also be flagged as warnings or errors.

```
rc.logMessage("This is a plain message")
rc.logWarning("This is a warning")
rc.logError("This is an error")
```

Example 13.1: Logging messages from scripts

When working with compact run-logs (which is strongly encouraged, see the option `Create compact run-log(307)`), plain messages may be removed from the run-log to preserve memory. When an error happens, the most recent 100 or so nodes in the run-log are kept even in a compact run-log, so in general this is not a problem. If you really need to make sure that a message will definitely be kept in the run-log you can enforce this by specifying the optional second parameter `dontcompactify`, e.g.

```
rc.logMessage("This message will not be removed", dontcompactify=true)
# or simply
rc.logMessage("This message will not be removed", 1)
```

Example 13.2: Logging messages that will not get removed in compact run-logs

Note

Only the `logMessage` method has that extra parameter. Warnings and errors are never removed from the run-log, so it doesn't apply to `logWarning` and `logError`.

13.3.2 Performing checks

Most of the time logging messages is tied to evaluating some condition. In that case, it is often desirable to get a result in the HTML or XML report equivalent to that of a Check node. The methods `rc.check` and `rc.checkEqual` will do just that:

```
var = 0
rc.check(var == 0, "!Value of var is 0")
rc.checkEqual('${system:user.language}', 'en', "English locale required",
              rc.EXCEPTION)
```

Example 13.3: Performing checks

For the old-style report the message is treated like a Check if it starts with an '!'. The optional last argument changes the error level in case of failure.

13.3.3 Accessing variables

Using QF-Test variables in Jython scripts is not difficult, but there are two viable ways to do so and it is important to understand the difference and which method to apply in which case.

First, standard QF-Test variable expansion takes place before the script is parsed and executed, so you can use `$(var)` or `${group:name}` style variables. This is very handy if you know that the variable values are either numbers or Boolean values, because Jython will recognize these without quoting:

```
if ${qftest:batch}:
    rc.logMessage("We are running in batch mode")
else:
    rc.logMessage("We are running in interactive mode")
```

Example 13.4: QF-Test variable expansion

The example above will work fine, because `${qftest:batch}` will expand to either `true` or `false`. Though standard Jython does not recognize these as Boolean values, the special environment QF-Test provides for the scripts makes this work. The following example also works well, provided `$(i)` is a numeric value, for example a `Loop(388)` index.

```
# log some value
rc.logMessage("data[$(i)]:" + data[$(i)])
```

Example 13.5: More variable expansion

It gets a bit more complicated when using QF-Test variables with arbitrary string values. Strings need to be quoted for Jython, using either single quotes `'` or double quotes `"`.

```
rc.logMessage("$ (someText) ")
```

Example 13.6: Simple text expansion

The code above will work very well unless `$(someText)` expands to a value that contains line-breaks or double quote characters. In that case, the script is not valid Jython code and a `ScriptException(564)` is thrown.

To avoid that kind of problem you should make it a habit to use the run-context's `lookup` method (see [section 37.6^{\(622\)}](#) for API reference) instead of `$(...)` or `${...:...}` whenever you want to access a QF-Test value as a string. That way you'll never have to worry about quoting.

```
# access a simple variable
rc.logMessage(rc.lookup("someText"))
# access a property or resource
rc.logMessage(rc.lookup("qftest", "version"))
```

Example 13.7: Using `rc.lookup` to access string variables

If you want to combine multiple variables in one string, it is easier to use `rc.expand` instead of `rc.lookup`. Note that you must escape the `'$'` characters by doubling them to prevent QF-Test from expanding the values itself (see [section 36.6^{\(616\)}](#)).

```
rc.logMessage("The resource is" +
    rc.expand("${${group}:${name}}"))
```

Example 13.8: Using `rc.expand` for complex variable access**Note**

Let us again stress the difference between the `'$'` and the `rc.lookup` methods for

accessing variables: '\$' expressions are expanded **before** the script is passed to the Jython interpreter. That means the text "\$ (var) " in the script is replaced by a verbatim copy of the value of the variable `var`. The method `rc.lookup` however returns the value of `var` **during** the processing of the script and, as explained above, is recommended for accessing string values.

13.3.4 Setting variables

To make the results of a Jython script available during further test execution, values can be stored in global or local variables. The effect is identical to that of a Set variable⁽⁵⁰⁹⁾ node. The corresponding methods in the run-context are `rc.setGlobal` and `rc.setLocal`.

```
# Test if the file /tmp/somefile exists
from java.io import File
rc.setGlobal("fileExists", File("/tmp/somefile").exists())
```

Example 13.9: Using `rc.setGlobal`

After executing the above example `$(fileExists)` will expand to 1 if the file `/tmp/somefile` exists and to 0 if it doesn't.

To clear a variable, set it to `None`, to clear all global variables use `rc.clearGlobals()` from a Server script.

13.3.5 Global Jython variables

Sometimes it is helpful to have a Jython variable available in several scripting nodes. If the value of the variable is not a simple string or integer, it is normally not sufficient to use `rc.setGlobal(...)` to store it in a global QF-Test variable because the value will be converted to a string in the process. Instead, such a variable should be declared `global` as shown in the following example.

```
global globalVar
globalVar = 10000
```

Example 13.10: Global Jython variable

The `globalVar` is now accessible within all further scripting nodes of the same type (Server scripts or SUT scripts of the same client). For changing the value of `globalVar` within another script, the `global` declaration is necessary again. Otherwise a new local

variable is created instead of accessing the existing global. Use the `del` statement to remove a global Jython variable:

```
global globalVar
del globalVar
```

Example 13.11: Delete a global Jython variable

13.3.6 Exchanging Jython variables between interpreters

Sometimes one would like to use variable values that have been defined in one Jython interpreter in a different interpreter. For example, an SUT script might have been used to create a list of items displayed in a table. Later we want to iterate over that list in a Server script.

To simplify such tasks, the run-context provides a symmetrical set of methods to access or set global variables in a different interpreter. For SUT scripts these methods are named `toServer` and `fromServer`. The corresponding Server script methods are `toSUT` and `fromSUT`.

The following example illustrates how an SUT script can set a global variable in QF-Test's interpreter:

```
cellValues = []
table = rc.lookup("idOfTable")
for i in range(table.getRowCount()):
    cellValues.append(table.getValueAt(i, 0))
rc.toServer(tableCells=cellValues)
```

Example 13.12: Setting a server variable from an SUT script

After the above script is run, the global variable named "tableCells" in QF-Test's interpreter will hold the array of cell values.

Note

The cell values in the above example are not necessarily strings. They could be numbers, date values, anything. Unfortunately Jython's *pickle* mechanism isn't smart enough to transport instances of Java classes (not even serializable ones), so the whole exchange mechanism is limited to primitive types like strings and numbers, along with Jython objects and structures like arrays and dictionaries.

13.3.7 Accessing the SUT's GUI components

For SUT scripts the run-context provides an additional method that is extremely useful. Calling `rc.getComponent("componentId")` will retrieve the information of the

Component⁽⁵⁴⁵⁾ node in the test-suite with the QF-Test ID⁽⁵⁴⁶⁾ "componentId" and pass that to QF-Test's component recognition mechanism. The whole process is basically the same as when simulating an event, including the possible exceptions if the component cannot be found.

If the component is located, it will be passed to Jython, not as some abstract data but as the actual Java object. All methods exposed by the Java API for the component's class can now be invoked to retrieve information or achieve effects which are not possible through the GUI alone. To get a list of a component's method see section 5.5⁽⁴⁴⁾.

```
# get the custom password field
field = rc.getComponent("tfPassword")
# read its crypted value
passwd = field.getCryptedText()
rc.setGlobal("passwd", passwd)
# get the table component
table = rc.getComponent("tabAddresses")
# get the number of rows
rows = table.getRowCount()
rc.setGlobal("tableRows", rows)
```

Example 13.13: Accessing components with `rc.getComponent`

You can also access sub-items this way. If the `componentId` parameter references an item, the result of the `getComponent` call is a pair, the component and the item's index. The index can be used to retrieve the actual value. The following example shows how to get the value of a table cell. Note the convenient way Jython supports sequence unpacking during assignment.

```
# first get the table and index
table, (row, column) = rc.getComponent("tableAddresses@Name@Greg")
# then get the value of the table cell
cell = table.getValueAt(row, column)
```

Example 13.14: Accessing sub-items with `rc.getComponent`

13.3.8 Calling Procedures

The run-context can also be used to call back into QF-Test and execute a Procedure⁽³⁷⁸⁾ node. Jython is perfect for reading and parsing data from a database or from a file, so this feature can be used to run data-driven tests.

Parameters are passed from a Jython script to a QF-Test Procedure in a Jython dictionary. The keys and values of the dictionary can be any kind of Jython object. They are converted to strings before they are passed to QF-Test.

```
rc.callProcedure("text.clearField",  
                {"component": "nameField"})
```

Example 13.15: Simple procedure call

In the example above the Procedure named "clearField" in the Package⁽³⁸⁵⁾ named "text" will be called. The single parameter for the call named "component" is set to the value "nameField".

The value returned by the Procedure through a Return⁽³⁸⁴⁾ node is returned as the result of the `rc.callProcedure` call.

Note

Great care must be taken when using `rc.callProcedure(...)` in SUT script nodes. Only short-running Procedures should be called that won't trigger overly complex actions in the SUT. Otherwise a DeadlockTimeoutException⁽⁵⁵⁹⁾ might be caused. For data-driven tests where for some reason the data must be determined in the SUT, use `rc.toServer(...)` to transfer the values to QF-Test's interpreter, then drive the test from a Server script node where these restrictions do not apply.

13.3.9 Setting options

3.1+

Many of the options described in chapter 29⁽²³⁷⁾ can also be set at runtime via `rc.setOption`. Constants for option names are predefined in the class `Options` which is automatically available for Jython and Groovy scripts.

A real-life example where this might be useful is if you want to replay an event on a disabled component, so you need to temporarily disable QF-Test's check for the enabled/disabled state:

```
rc.setOption(Options.OPT_PLAY_THROW_DISABLED_EXCEPTION, false)
```

Example 13.16: Jython/Groovy example for setOption

After replaying this special event, the original value read from the configuration file or set in the option dialog can be restored by unsetting the option as the following example shows:

```
rc.unsetOption(Options.OPT_PLAY_THROW_DISABLED_EXCEPTION)
```

Example 13.17: Jython/Groovy example for unsetOption

Note

Be sure to set QF-Test options in a Server script node and SUT options in an SUT script node, otherwise the setting will have no effect. The option documentation in chapter 29⁽²³⁷⁾ shows which one to use.

13.3.10 A complex example

We are going to close this section with a complex example, combining features from Jython and QF-Test to execute a data-driven test. For the example we assume that a simple table with the three columns "Name", "Age" and "Address" should be filled with values read from a file. The file is assumed to be in "comma-separated-values" format with "|" as the separator character, one line per table-row, e.g.:

```
John Smith|45|Some street, some town  
Julia Black|35|Another street, same town
```

To verify the SUT's functionality in creating new table rows, a QF-Test Procedure should be created that takes three parameters "name", "age", and "address", creates a new table-row and fills it with these values. Then we can use Jython to read and parse the data from the file, iterate over the data-sets and call back to QF-Test for each table-row to be created. The name of the file to read is passed in a QF-Test variable named "filename". When we have finished filling the table, we compare the state of the actual table component with the data read from the file to make sure everything is OK.


```
import string
data = []
# read the data from the file
fd = open(rc.lookup("filename"), "r")
line = fd.readline()
while line:
    # remove whitespace
    line = string.strip(line)
    # split the line into separate fields
    # and add them to the data array
    if len(line) > 0:
        data.append(string.split(line, "|"))
        line = fd.readline()
# now iterate over the rows
for row in data:
    # call a qftest procedure to create
    # one new table row
    rc.callProcedure("table.createRow",
                     {'name': row[0], 'age': row[1],
                      'address': row[2]})
# verify that the table-rows have been filled correctly
table = rc.getComponent("tabAddresses")
# check the number of rows
rc.check(table.getRowCount() == len(data), "Row count")
if table.getRowCount() == len(data):
    # check each row
    for i in range(len(data)):
        rc.check(table.getValueAt(i, 0) == data[i][0],
                  "Name in row " + `i`)
        rc.check(table.getValueAt(i, 1) == data[i][1],
                  "Age in row " + `i`)
        rc.check(table.getValueAt(i, 2) == data[i][2],
                  "Address in row " + `i`)
```

Example 13.18: Executing a data-driven test

Of course, the example above serves only as illustration. It is too complex to be edited comfortably in QF-Test and too much is hard-coded, so it is not easily reusable. For real use, the code to read and parse the file should be parameterized and moved to a module, as should the code that verifies the table. This topic is covered in the following section.

13.3.11 Override components

You might face a situation where you want to work with a component, which you have to search before working with it. Sometimes recording all required components can be exhaustive or might be too complicated. For such cases you can use the method

`rc.overrideElement` to set the found component (either by generic components or via scripting) to a QF-Test component. Now you can work with the assigned component and use all available QF-Test nodes.

Let's imagine that we have a panel and we want to work with the first textfield, but because of changing textfields we cannot rely on the standard way of the recognition. Now we can implement a script, which looks for the first textfield and assigns that textfield to the `PriorityAwtSwingComponent` from the standard library `qfs.qft`. Once we have executed that script we can work with any QF-Test nodes using the `PriorityAwtSwingComponent`, which actually performs all actions on the found textfield.

```
from de.qfs.apps.qftest.extensions import ResolverRegistry
panel = rc.getComponent("myPanel")
for component in panel.getComponents():
    if ResolverRegistry.instance().isInstance(component, \
        "javax.swing.JTextField"):
        rc.overrideElement("PriorityAwtSwingComponent", component)
        break
```

Example 13.19: Using `rc.overrideElement`

This concept is very useful if you know an algorithm to determine the target component of your test-steps.

You can find such priority-components for all engines in the standard library `qfs.qft`. You can also find an illustrative example in the provided demo test-suite `carconfig_en.qft`, located in the directory `demo/carconfig` in your QF-Test installation.

13.4 Modules

Modules for Jython in QF-Test are just like standard Python modules. You can import these modules into QF-Test scripts and call their methods, which simplifies the development of complex scripts and increases maintainability since modules are available across test-suites.

Modules intended to be shared between test-suites should be placed in the directory `jython` under QF-Test's root directory. Modules written specifically for one test-suite can also be placed in the test-suite's directory. The version-specific directory `qftest-4.0.8/jython/Lib` is reserved for modules provided by Quality First Software GmbH. Jython modules must have the file extension `.py`.

To improve [example 13.18^{\(110\)}](#) you could write a module `csvtable.py` with methods `loadTable` to read the data from the file and `verifyTable` to verify the results. An example module is provided in `qftest-4.0.8/doc/tutorial/csvtable.py`. Following is a simplified version:

```
import string
def loadTable(file, separator=','):
    data = []
    fd = open(file, "r")
    line = fd.readline()
    while line:
        line = string.strip(line)
        if len(line) > 0:
            data.append(string.split(line, separator))
            line = fd.readline()
    return data
def verifyTable(rc, table, data):
    ret = 1
    # check the number of rows
    if table.getRowCount() != len(data):
        if rc:
            rc.logError("Row count mismatch")
            return 0
    # check each row
    for i in range(len(data)):
        row = data[i]
        # check the number of columns
        if table.getModel().getColumnCount() != len(row):
            if rc:
                rc.logError("Column count mismatch " +
                           "in row " + `i`)
            ret = 0
        else:
            # check each cell
            for j in range(len(row)):
                val = table.getModel().getValueAt(i, j)
                if str(val) != row[j]:
                    if rc:
                        rc.logError("Mismatch in row " +
                                   `i` + " column " +
                                   `j`)
            ret = 0
    return ret
```

Example 13.20: Writing a module

The code above should look familiar. It is an improved version of parts of [example 13.18^{\(110\)}](#). With that module in place, the code that has to be written in QF-Test is reduced to:

```
import csvtable
# load the data
data = csvtable.loadTable(rc.lookup("filename"))
# now iterate over the rows
for row in data:
    # call a qftest procedure to create
    # one new table row
    rc.callProcedure("table.createRow",
                     {'name': row[0], 'age': row[1],
                      'address': row[2]})
# verify that the table-rows have been filled correctly
table = rc.getComponent("tabAddresses")
csvtable.verifyTable(rc, table, data)
```

Example 13.21: Calling methods in a module

For more complex import of data QF-Test can be extended with existing Python modules. For example, at <http://python-dsv.sourceforge.net/> an excellent module for very flexible CSV import is freely available.

13.5 Post-mortem debugging of Jython scripts

Python comes with a simple line-oriented debugger called `pdb`. Among its useful features is the ability for post-mortem debugging, i.e. analyzing why a script failed with an exception. In Python you can simply import the `pdb` package and run `pdb.pm()` after an exception. This will put you in a debugger environment where you can examine the variable bindings in effect at the time of failure and also navigate up to the call stack to examine the variables there. It is somewhat similar to analyzing a core dump of a C application.

Though Jython comes with `pdb`, the debugger doesn't work very well inside QF-Test for various reasons. But at least post-mortem debugging of Jython scripts is supported from the Jython terminals (see [section 13.1^{\(99\)}](#)). After a [Server script^{\(416\)}](#) node fails, open QF-Test's Jython terminal, for a failed [SUT script^{\(419\)}](#) node open the respective SUT Jython terminal, then just execute `debug()`. This should have a similar effect as `pdb.pm()` described above. For further information about the Python debugger please see the documentation for `pdb` in Python version 2.5 at <http://www.python.org/doc/2.5/lib/module-pdb.html>.

13.6 Differences between Jython version 2.2.1 and 2.5

Jython version 2.5 was a major rewrite of the Java version of Python. Though most of the changes are backwards compatible, there are some subtle differences resulting from changes in the Java integration as well as the Python language itself.

13.6.1 Boolean type

Jython now has a real boolean type with values `True` and `False` whereas in older versions integer values 0 and 1 served as boolean values. This can cause problems if boolean results from calls like `file.exists()` are assigned to a QF-Test variable, e.g. "fileExists" and later checked in a Condition⁽³⁹⁷⁾ attribute in the form `$(fileExists) == 1`. Such conditions generally be written as just `$(fileExists)` which works well with all Jython versions.

13.6.2 Character encodings

All Java strings are sequences of 16-bit characters. Python's original strings are made of 8-bit characters. Later, unicode strings with 16-bit characters were added. Jython literal strings like `"abc"` are 8-bit, prepending 'u' for `u"abc"` turns them into unicode strings.

In Jython 2.2, Java strings were converted to 8-bit Python strings based on the default encoding of the Java VM, typically ISO-8859-1 (also known as latin-1) in western countries. In Jython 2.5, every Java string is now interpreted as a unicode Jython string. This results in a lot more implicit conversion between 8-bit and unicode strings, for example when concatenating a Java string - now converted to unicode - and a literal string like `rc.lookup("path") + "/file"`. Most of the time this works well, but if the literal string contains characters outside the 7-bit ASCII character-set, things start to get messy. The default encoding for 8-bit Jython characters can be specified in the option Default character encoding for Jython⁽²⁴⁰⁾ with a default of latin-1 for maximum backwards compatibility. On the upside it is now possible to have default encodings other than latin-1 and to specify literal strings of characters in international character sets.

One thing to watch out for is existing code of the form

```
import types
if type(somevar) == types.StringType:
    ...
```

The type `types.StringType` is the 8-bit string. It does not match unicode strings. To

test whether some variable is a Jython string, regardless of whether it's 8-bit or unicode, change that to

```
import types
if type(somevar) in types.StringTypes:
    ...
```

One new requirement - coming from newer Python versions - is that Python module files containing characters outside the 7-bit ASCII character must specify the character encoding to be used in a comment line close to the top of the file, e.g.

```
# coding: latin-1
```

Please see <http://www.python.org/peps/pep-0263.html> for details.

13.6.3 Getting the name of a Java class

This simple operation is surprisingly difficult in Jython. Given a Java object you would expect to simply write `obj.getClass().getName()`. For some objects this works fine, for others it fails with a cryptic message. This can be very frustrating. Things go wrong whenever there is another `getName` method defined by the class, which is the case for `AWT Component`, so getting the class name this way fails for all AWT/Swing component classes.

In Jython 2.2.1 the accepted workaround was to use the Python idiom `obj.__class__.__name__`. This no longer works in Jython 2.5 because it no longer returns the fully qualified class name, only the last part. Instead of `java.lang.String` you now get just `String`. The only solution that reliably works for version 2.5 is:

```
from java.lang import Class
Class.getName(obj.getClass())
```

This also works for 2.2, but it is not nice, so we initiated a new convenience module with utility methods called `qf` that gets imported automatically. As a result you can now simply write

```
qf.getClassName(obj).
```

13.7 Scripting with Groovy

Groovy is another established scripting language for the Java Platform. It was invented by James Strachan and Bob McWhirter in 2003. All you need for doing Groovy is a Java Runtime Environment (JRE) and the `groovy-all.jar` file. This library contains a compiler to create Java class files and provides the runtime when using that classes in the Java Virtual Machine (JVM). You may think of Groovy as being Java with an additional `.jar` file. In contrast to Java, Groovy is a dynamic language, meaning that the behaviour of an object is determined at runtime. Groovy also allows to load classes from sources without creating class files. Finally, it is easy to embed Groovy scripts into Java applications like QF-Test.

The Groovy syntax is similar to Java, maybe more expressive and easier to read. When coming from Java you can embrace the Groovy style step by step. Of course we cannot explain all aspects of the Groovy language here. For in-depth information, please take a look at the Groovy home page at <http://groovy.codehaus.org> or read the excellent book "Groovy in Action" by Dierk Koenig and others. Perhaps the following tips may help a Java programmer getting started with Groovy.

- The semicolon is optional as long as a line contains only one statement.
- Parentheses are sometimes optional, e. g. `println 'hello qfs'` means the same as `println('hello qfs')`.
- Use `for (i in 0..<len) { ... }` instead of `for (int i = 0; i < len; i++) { ... }`.
- The following imports are made by default: `java.lang.*`, `java.util.*`, `java.io.*`, `java.net.*`, `groovy.lang.*`, `groovy.util.*`, `java.math.BigInteger`, `java.math.BigDecimal`.
- Everything is an object, even integers like `'1'` or booleans like `'true'`.
- Instead of using getter and setter methods like `obj.getXxx()`, you can simply write `obj.xxx` to access a property.
- The operator `==` checks for equality, not identity, so you can write `if (somevar == "somestring")` instead of `if (somevar.equals("somestring"))`. The method `is()` checks for identity.
- Variables have a dynamic type when being defined with the `def` keyword. Using `def x = 1` allows for example to assign a `String` value to the variable `x` later in the script.

- Arrays are defined differently from Java, e. g. `int[] a = [1, 2, 3]` or `def a = [1, 2, 3] as int[]`. With `def a = [1, 2, 3]` you define a `List` in Groovy.
- Groovy extends the Java library by defining a set of extra methods for many classes. Thus you can for example apply an `isInteger()` method to any `String` object in a Groovy script. That's what is called *GDK* (according to the *JDK* in Java). To get a list of those methods for an arbitrary object `obj`, you can simply invoke `obj.class.metaClass.metaMethods.name` or use the following example:

```
import groovy.inspect.Inspector
def s = 'abc'
def inspector = new Inspector(s)
def mm = inspector.getMetaMethods().toList().sort() {
    it[Inspector.MEMBER_NAME_IDX] }
for (m in mm) {
    println(m[Inspector.MEMBER_TYPE_IDX] + ' ' +
            m[Inspector.MEMBER_NAME_IDX] +
            '(' + m[Inspector.MEMBER_PARAMS_IDX] + ')')
}
```

Example 13.22: GDK methods for a `String` object

- Inner classes are not supported, in most cases you can use *Closures* instead. A *Closure* is an object which represents a piece of code. It can take parameters and return a value. Like a block, a *Closure* is defined with curly braces `{ ... }`. Blocks only exists in context with a class, an interface, static or object initializers, method bodies, `if`, `else`, `synchronized`, `for`, `while`, `switch`, `try`, `catch`, and `finally`. Every other occurrence of `{ ... }` is a *Closure*. As an example let's take a look at the `eachFileMatch` GDK method of the `File` class. It takes two parameters, a filter (e. g. a `Pattern`) and a *Closure*. That *Closure* takes itself a parameter, a `File` object for the current file.

```
def dir = rc.lookup('qftest', 'suite.dir')
def pattern = ~/.*\.qft/
def files = []
new File(dir).eachFileMatch(pattern) { file ->
    files.add(file.name)
}
files.each {
    // A single Closure argument can also be referred with "it"
    rc.logMessage(it)
}
```

Example 13.23: Closures

- Working with `Lists` and `Maps` is simpler than in Java.

```
def myList = [1, 2, 3]
assert myList.size() == 3
assert myList[0] == 1
myList.add(4)
def myMap = [a:1, b:2, c:3]
assert myMap['a'] == 1
myMap.each {
    this.println it.value
}
```

Example 13.24: Working with lists and maps

13.7.1 Accessing variables in Groovy

In QF-Test Groovy scripts we decided not to support the `$`-Expansion for QF-Test variables. It only takes place in Jython scripts (see [section 13.3.3^{\(103\)}](#)). The reason is that Groovy already uses the dollar sign to dereference script variables and evaluate expressions within a `GString`.

```
def x = 3
assert "$x" == 3
assert "${2 * x}" == 6
```

Example 13.25: `GString` expansion

Values of QF-Test variables can be obtained at runtime by means of several `rc` methods:

- `String lookup(varname)` **or** `String lookup(group, varname)`
- `String getStr(varname)` **or** `String getStr(group, varname)`
- `Integer getNum(varname)` **or** `Integer getNum(group, varname)`
- `Boolean getBool(varname)` **or** `Boolean getBool(group, varname)`

```
rc.setGlobal('fileExists', new File('c:/tmp/somefile.foo').exists())
assert rc.lookup('fileExists') == 'false'
assert rc.getStr('fileExists') == 'false'
assert ! rc.getBool('fileExists')
rc.setGlobal('myvar', '3')
assert rc.getNum('myvar') == 3
```

Example 13.26: Lookup for QF-Test variables in a Groovy script

13.7.2 Global Groovy variables

Exchanging variables between several script nodes of the same type (Server scripts or SUT scripts of the same client) is even easier than in Jython. The rule is that undeclared variables are assumed to be defined in the binding of the script. If they are not, they will be added automatically to the list of binding variables.

```
myGlobal = 'global'
```

Example 13.27: Define a global Groovy variable

```
assert myGlobal == 'global'
def globals = binding.variables
assert globals['myGlobal'] == 'global'
globals.remove('myGlobal')
assert globals.find { it == 'myGlobal' } == null
```

Example 13.28: Use and delete a global Groovy variable

Predefined global variables are the QF-Test run-context `rc` and the `PrintWriter` `out`, which is used for the script's `println` method.

13.7.3 Groovy packages

Just like Java classes, Groovy source files (`.groovy`) can be organized in packages. Those intended to be shared between test-suites should be placed in the directory `groovy` under QF-Test's root directory. Others that are written specifically for one test-suite can also be placed in the test-suite's directory. The version-specific directory `qftest-4.0.8/groovy` is reserved for Groovy files provided by Quality First Software GmbH.

```
package my
class MyModule
{
    public static int add(int a, int b)
    {
        return a + b
    }
}
```

Example 13.29: `MyModule.groovy`

The file `MyModule.groovy` could be saved in a subdirectory `my` below the suite directory. Then you can use the `add` method from `MyModule` as follows:

```
import my.MyModule as MyLib
assert MyLib.add(2, 3) == 5
```

Example 13.30: Using `MyModule`

This code also shows another groovy feature: *Type aliasing*. By using `import` and `as` in combination you can reference a class by a name of your choice.

Chapter 14

Data-driven testing

Data-driven testing is a very important aspect of test automation. In short, the goal is to run a given test or set of tests multiple times with different sets of input data and expected results. QF-Test has various means to store data or load external data for use in data-driven tests. The most convenient is based on a Data driver node which sets up an environment for iterating over the sets of data combined with one or more Data binder nodes to bind the variables for test execution. Note that there is no Data binder node as such. The name serves as a generic term for the specific nodes like a Data table or a CSV data file. This is best explained through some examples. A demo test-suite with simple and advanced examples named `datadriver.qft` is provided in the directory `doc/tutorial` below QF-Test's root directory.

14.1 Data-driver examples

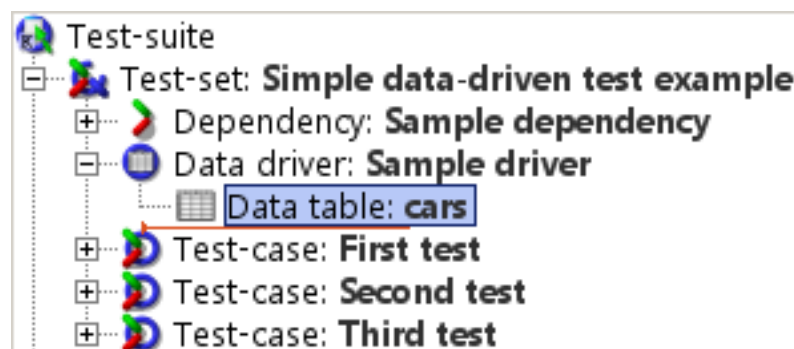


Figure 14.1: A simple data-driven test

The image above shows a Test-set with a Data driver node that contains a single Data binder in the form of a Data table node. The contents of the Data table are as follows:

Data table

Name: cars Iteration counter:

Iteration ranges:

Data bindings:

	Model	Variant	Price
0	Rolo	None	19000
1	I5	Family	33000
2	Minigolf	Jazz	28000

QF-Test ID:

Delay before (ms): Delay after (ms):

Comment:

Figure 14.2: Data table example

When the Test-set is executed, it will iterate over the rows of the Data table shown above. For each of the three iterations the values of the respective row in the table are bound to the variable named in the matching column header. Thus, during the first iteration the variable named "Model" is bound to "Rolo", "Variant" to "None" and "Price" to "19000". During the second iteration, "Model" is set to "I5" and to "Minigolf" during the third and last iteration. For each iteration, all Test-case child nodes of the Test-set are run.

The following image shows a run-log for the above Test-set.

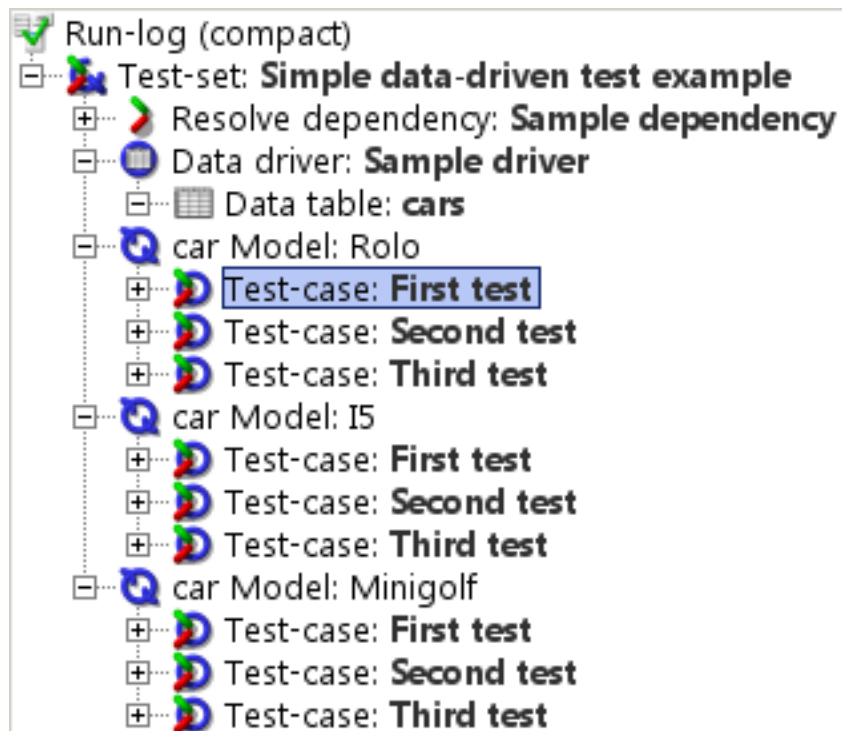


Figure 14.3: Run-log of a data-driven test

The next example shows that data-driven testing is not limited to a single loop:

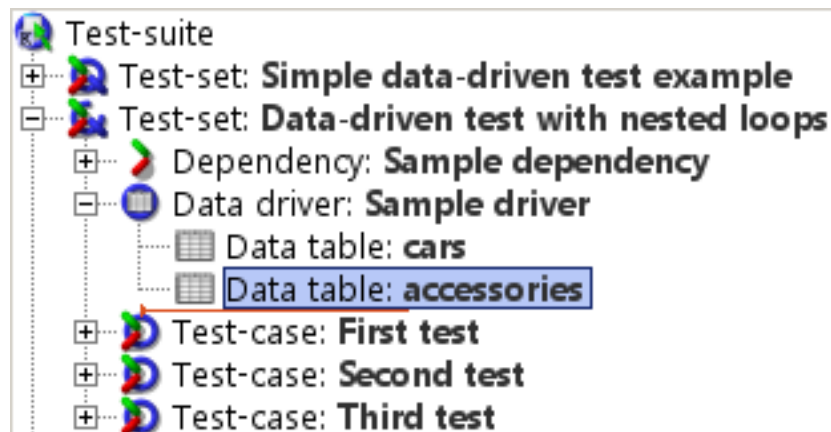


Figure 14.4: Data-driven test with nested loops

The Data driver now contains a second Data table node with the following contents:

Data table

Name

accessories

Iteration counter

Iteration ranges

Data bindings

	Name	APrice
0	Alloy Rims	900
1	Radio with CD	400

QF-Test ID

Delay before (ms)

Delay after (ms)

Comment

Figure 14.5: Second data table example

The Test-set will now go through a total of six iterations because for each of the three iterations of the outer loop "cars", both iterations of the inner loop "accessories" will be run as shown in the following run-log.

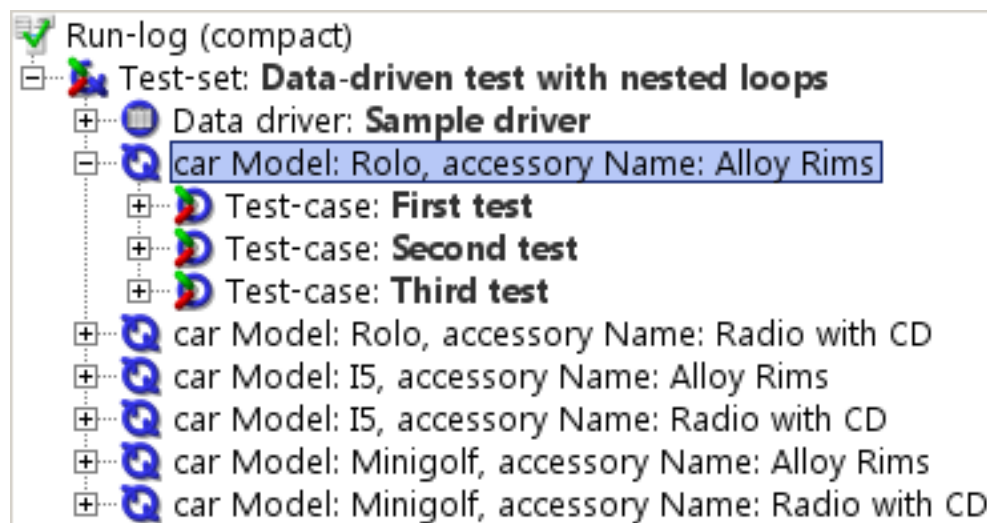


Figure 14.6: Run-log of a data-driven test with nested loops

Note

The extremely useful dynamic names of the loop nodes in the run-logs are obtained by setting the attribute Name for loop pass in the run-log to the value "car Model: \$(Model)" in the first and to "car Model: \$(Model), accessory Name: \$(Accessory)" in the second example. As you can see, that name is expanded individually for each iteration, so you can make use of the variables bound for that iteration.

14.2 General use of Data drivers

As seen in the example above the Data driver node must be placed in a Test-set node, between the optional Dependency and Setup nodes. When the Test-set is executed it will check for Data driver and run it. The contents of the Data driver node are not limited to Data binders. Like a normal Sequence the Data driver node can hold any executable node to be able to perform any setup that may be required to retrieve the data. Thus it is also possible to share Data binders by putting them inside a Procedure and calling the Procedure from inside the Data driver.

Conceptually, a Data binder represents a loop where a different set of variables is bound for each iteration. A Data binder must be registered with a name in the Data driver context of a Test-set. This ensures that the loop can be interrupted by a Break⁽³⁹⁴⁾ node with the same name. Once the Test-set has run the Data driver node, it will iterate over the registered data loops and perform the tests.

In case of nested loops the Data binder that was registered first represents the outermost loop. Its variables are bound first and have lesser precedence than the variables from the inner loop(s).

14.3 Examples for Data drivers

We provide a couple of examples for reading CSV or Excel files in the test-suite `doc/tutorial/datadrivers.qft`.

14.4 Advanced use

Besides the `Data table`⁽³⁶⁰⁾ node there are various other means for binding data in a data driver. The `Excel data file`⁽³⁶⁸⁾, `CSV data file`⁽³⁷²⁾, `Database`⁽³⁶³⁾ and `Data loop`⁽³⁷⁶⁾ nodes are all explained in detail in [section 30.4](#)⁽³⁵⁷⁾.

It is also possible to bind data by calling the Procedures `qfs.databinder.bindList` or `qfs.databinder.bindSets` in the standard library `qfs.qft`. These take as parameters strings with lists or sets of values to split and iterate over. Please see tutorial chapter 8 for information about the standard library.

And finally, data can be bound directly from Jython (and analogous from Groovy) with the help of the `databinder` module, which offers the following methods:

```
bindList(Object rc, String loopname, String varname, Object
values, String separator=None, String counter=None, String
intervals=None)
```

Create and register a databinder that binds a list of values to a variable.

Parameters

rc	The current run-context.
loopname	The name under which to bind the data, equivalent to the Name attribute of a Data binder node.
varname	The name of the variable to bind to.
values	The values to bind. Either a sequence type or a string to split.
separator	Optional separator character to split the values at in case they're a string. Default is whitespace.
counter	An optional variable name for the iteration counter.
intervals	Optional ranges of indices, separated by comma, e.g. "0,2-3".

```
bindSets(Object rc, String loopname, Object varnames, Object  
values, String separator=None, String counter=None, String  
intervals=None)
```

Create and register a databinder that binds a list of value-set to a set of variables.

Parameters

rc	The current run-context.
loopname	The name under which to bind the data, equivalent to the Name attribute of a Data binder node.
varnames	The names of the variables to bind to. Either a sequence type or a string to split.
values	The value-sets to bind. Either a sequence of sequences - each inner sequence being one set of data to bind - or a string to split.
separator	Optional separator character to split the varnames and the values of a value-set at in case they're a string. Default is whitespace. Value-sets are separated by line-breaks.
counter	An optional variable name for the iteration counter.
intervals	Optional ranges of indices, separated by comma, e.g. "0,2-3".

```
bindDict(Object rc, String loopname, dictionary dict, String  
counter=None, String intervals=None)
```

Create and register a databinder that binds data from a dictionary. The keys of the dictionary are the names of the variables and the values are sequences of values to be bound.

Parameters

rc	The current run-context.
loopname	The name under which to bind the data, equivalent to the Name attribute of a Data binder node.
dict	The dictionary to bind.
counter	An optional variable name for the iteration counter.
intervals	Optional ranges of indices, separated by comma, e.g. "0,2-3".

Some examples:

```
import databinder
# Three iterations with the values "spam", "bacon" and "eggs"
# bound to the variable named "ingredient"
databinder.bindList(rc, "meal", "ingredient", ["spam", "bacon", "eggs"])
# Same with string values
databinder.bindList(rc, "meal", "ingredient", "spam bacon eggs")
# Same with string values and special separator
databinder.bindList(rc, "meal", "ingredient", "spam|bacon|eggs", "|")
# Two iterations, the first with item="apple" and number="5",
# the second with item="orange" and number="3"
databinder.bindSets(rc, "fruit", ["item", "number"],
                    [["apple",5], ["orange",3]])
# Same with string values, note the linebreak
databinder.bindSets(rc, "fruit", "item number", """apple 5
orange 3""")
# Same as before with the data stored in a dict
databinder.bindDict(rc, "fruit",
                   {"item": ["apple", "orange"],
                    "number": [5,3]})
```

Example 14.1: Examples for use of the databinder module

Chapter 15

Reports and test documentation

Besides test-suites and run-logs QF-Test can create a number of additional documents. Most important of these is the report, which summarizes the overall results of a test-run along with an overview over the test-suites executed and their individual results. The report is easy to read and understand without further knowledge about QF-Test and thus complements the run-log which is geared towards error analysis and requires some QF-Test experience to fully understand.

Following is an example of a report summary:

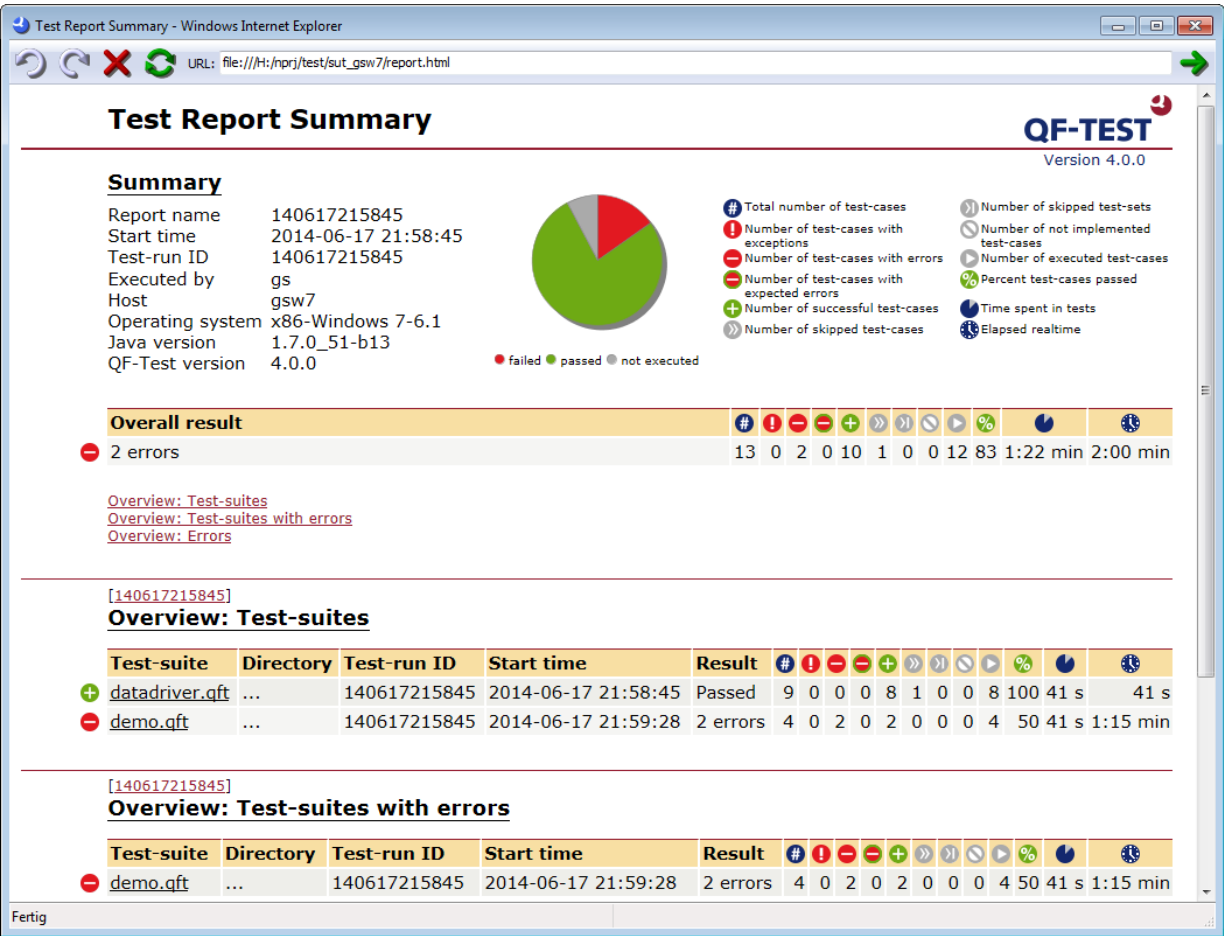


Figure 15.1: Example report

The other documents are more static in nature, describing the content of test-suites instead of test-run results. The *testdoc* document gives an overview over the structure of *Test-set*⁽³²¹⁾ and *Test-case*⁽³¹⁴⁾ nodes in a given set of test-suites. It is intended for the test architect or QA project manager and documents the current state of test development. The *pkgdoc* documentation is similar, but focuses on *Package*⁽³⁸⁵⁾, *Procedure*⁽³⁷⁸⁾ and *Dependency*⁽³⁴⁴⁾ nodes instead. The result is a library reference comparable to Java's *JavaDoc*. The target audience for *pkgdoc* are test implementers requiring information about already existing procedures and their parameters.

15.1 Reports

Before we can start explaining how to create reports, some basic terms and concepts need to be defined.

15.1.1 Report concepts

A report represents the results of one or more *test-runs*. A test-run comprises the execution of either a single test-suite or multiple test-suites, typically executed together in one batch-run. A test-run is identified by a *runid*. It is possible to execute a test-run in several steps by assigning the same runid to the resulting run-logs.

A report is identified by a *report name*. For a report that covers a single test-run, the report name is usually the same as the runid. For reports summarizing the results of several test-runs a distinct report name can be specified.

Reports can be created in multiple variants: XML, HTML and JUnit. Most users will probably use the HTML variant which can be viewed in a browser, printed and archived. The XML variant can serve as the basis for collecting the results of a test-run for further processing, for example to collect test-results in a database or to create customized HTML reports. We suggest that you always create both HTML and XML reports unless you have a good reason to do otherwise. JUnit reports base on the JUnit XML format as created by Apache Ant by use of its JUnitReport task. This format is not as pretty and detailed as the first two report variants QF-Test offers but it is directly understood by many continuous integration tools and may prove useful for a quick integration with those.

A report consists of one summary document, plus one document per run-log. These files are collected together with complementary files like icons, stylesheets and screenshot images in a directory. At the file level, this directory represents the report.

The layout of the files inside the report directory depends on some command line options explained below. Basically there are two ways to lay out the files: Based on the file structure of the original test-suites or based on the file structure of the run-logs.

15.1.2 Report contents

In advance to the overall test result, a report as shown above starts with a summary containing informational system data and a legend describing the meaning of counter icons used in the report (see [Running tests](#)⁽³¹⁾).

Note

The difference between "Time spent in tests" and "Elapsed time" are explicit delays introduced in nodes via the 'Delay before/after' attribute or user interrupts.

The contents of a report are based on the original structure of the executed test-suites. The main structure is created from [Test-set](#)⁽³²¹⁾ and [Test-case](#)⁽³¹⁴⁾ nodes. The [Comment](#)⁽³²⁶⁾ attributes of the root node as well as the Test-set and Test-case nodes share the doctags with testdoc documents as explained in [section 15.2](#)⁽¹³⁴⁾. In addition to those doctags the '@title' doctag can be specified in the comment of the root node to set a title for the report document created for the respective test-suite.

If `-report.teststeps`⁽⁵⁸¹⁾ is specified in batch mode (true by default) or the respective option is active in the interactive dialog, Test-cases can be further broken down into steps with the help of `Test-step`⁽³³⁶⁾ nodes. In addition to explicitly wrapping steps into a Test-step, any node can be turned into a test-step by specifying the doctag '@teststep' in its Comment, followed by an optional name for the step. For Test-step nodes the '@author', '@version' and '@since' doctags are also applicable. The names, comments and tag values of the various nodes can contain variables that will be expanded at execution time so that the expanded value is shown in the report. This is especially useful for test-steps within a procedure.

If listing of test-steps is active, Setup, Cleanup and Dependency nodes are also listed and checks, screenshots and messages, including warnings, errors and exceptions are properly integrated into the nested steps. If the test-suites are set up properly the resulting report can serve as a very readable summary of what was going on during the execution of a test.

Whether warnings and checks are listed is determined by the command line arguments `-report.warnings`⁽⁵⁸¹⁾ and `-report.checks`⁽⁵⁷⁹⁾ or the respective interactive options. Warnings from component recognition are never listed because they are too technical and could easily flood the report. For checks one must distinguish between checks that represent an actual verification step and those that are used solely for control flow, for example to check whether a checkbox is already selected and click it only in case it is not. By default QF-Test lists those Check nodes in the report that have the default result settings, i.e. the Error level of message is 'Error', no exception is thrown and no result variable bound. All others are treated as helpers for control flow and not listed in the report. For cases where this default treatment is not appropriate, you can force a Check into the report via the doctag '@report' in its Comment attribute or prevent its listing via '@noreport'. Of course failed checks are treated as warnings, errors or exceptions (depending on their Error level of message) and cannot be excluded from the report if messages at the respective level are shown.

Additional messages, checks and screenshots can be added to the report by scripts via the methods `rc.logMessage`, `rc.logImage` and `rc.check` and its variants, which have an optional `report` parameter. For details, please see the run-context API documentation in [section 37.6](#)⁽⁶²²⁾.

15.1.3 Creating reports

There are three ways to create reports:

- Interactively from a run-log through the menu item .
- In batch mode as the result of a test-run.

- In batch mode by transforming already existing run-logs.

The interactive variant is easy to use. Just select the target directory for the report and whether you want the XML and/or the HTML variant.

For report creation in batch mode there are a number of command line options which are listed and explained in [section 32.2](#)⁽⁵⁷¹⁾. Let's look at the variant of creating reports as the result of a test-run first:

The command line syntax for plain test-execution in batch mode is `qftest -batch <test-suite> [<test-suite>...]`

To create a combined XML and HTML report, use `-report <directory>`⁽⁵⁷⁹⁾. To create only one version or to separate the XML, HTML variants, use `-report.xml <directory>`⁽⁵⁸¹⁾ and/or `-report.html <directory>`⁽⁵⁸⁰⁾. For JUnit reports `-report.junit <directory>`⁽⁵⁸⁰⁾ works respectively.

The runid of a test-run is specified with `-runid <ID>`⁽⁵⁸²⁾, the name of the report with `-report.name <name>`⁽⁵⁸⁰⁾. If the report name is unspecified it will default to the runid.

To lay out the files in the report directory according to the file structure of the test-suites, use `-sourcedir <directory>`⁽⁵⁸³⁾. To use the file structure of the run-log as the basis, use `-runlogdir <directory>`⁽⁵⁸²⁾.

The following is a typical example of a command line for a batch run making use of the placeholders explained in [section 32.2.4](#)⁽⁵⁸⁷⁾:

```
qftest -batch -runid +M+d -runlog logs/+i -report report_+i
      -sourcedir . suite1.qft subdir/suite2.qft
```

Example 15.1: Creating a report as the result of a test-run

Creating a report as a separate step by transforming a set of run-logs is similar in many respects. The run-logs to transform have to be specified instead of the test-suites to execute and the `-runid <ID>` and `-sourcedir <directory>` command line options have no effect. The following is an example for how to create a weekly summary report based on the assumption that you have collected all run-logs below the directory named `logdir`, possibly in subdirectorys thereof:

```
qftest -batch -genreport -report report_+M+d
      -report.name week_of_+y+M+d logdir
```

Example 15.2: Creating a weekly summary report

15.2 Testdoc documentation for Test-sets and Test-cases

The type of test documents called *testdoc* provide overview and detailed information over the Test-set⁽³²¹⁾ and Test-case⁽³¹⁴⁾ nodes of one or more test-suites. They are a valuable tool for QA project managers to keep track of the current state of test development. Similar to reports, testdoc documents are laid out as directories with one summary file and one detailed file per test-suite.

A testdoc document for a single suite can be created interactively from a test-suite by selecting **Create HMTL/XML testdoc...** from the **File** menu. This is very useful during test development to quickly check whether all tests are properly documented.

For actual use as a reference it is preferable to create complete sets of documents spanning multiple test-suites for a whole project. This can be done by running QF-Test in batch mode with the `-gendoc`⁽⁵⁷⁶⁾ command line argument. In its simplest form, a call to create testdoc documentation for a whole directory tree would look as follows:

```
qftest -batch -gendoc -testdoc test_documentation
      directory/with/test-suites
```

Example 15.3: Creating testdoc documentation

Please see chapter 32⁽⁵⁶⁷⁾ for detailed information about the available command line arguments.

To get optimal results you can use HTML markup in the Comment⁽³²⁶⁾ attributes of Test-set and Test-case nodes and also make use of doctags. A *doctag* is a keyword beginning with '@', sometimes followed by a name and always by a description. This is a proven concept in JavaDoc, the standard documentation format for Java programs (see <http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#tag>).

The following doctags are supported for Test-set and Test-case nodes:

@deprecated

If a Test-set or Test-case is no longer to be used, this description should explain when and why the node was deprecated and especially which replacement should be used.

@condition

Non-formal explanation of the condition under which the node is executed or skipped.

@param

Description for a parameter. Following are the name of the parameter and its description.

@charvar

Description for a characteristic variable. Following are the name of the variable and its description.

@author

Author of the Test-set or Test-case.

@version

Version of the Test-case or Test-case.

@since

The version since which this Test-set or Test-case has been available.

In addition to the doctags described above, the doctag '@title' in the comment of the root node can be used to specify a title for the testdoc document created for the respective test-suite.

15.3 Pkgdoc documentation for Packages, Procedures and Dependencies

The concepts of and methods for creation of *pkgdoc* documents are nearly identical to testdoc, so this section is brief. Instead of Test-set and Test-case nodes, pkgdoc documents cover Package⁽³⁸⁵⁾, Procedure⁽³⁷⁸⁾ and Dependency⁽³⁴⁴⁾ nodes. They are intended for the test developer to keep track of the procedures available for use in implementing tests.

Please refer to the standard library `qfs.qft` as a good example how a pkgdoc may look like.

A pkgdoc document can also either be created interactively using File→Create HMTL/XML pkgdoc... or in batch mode. Again, please see chapter 32⁽⁵⁶⁷⁾ for detailed information about the available command line arguments.

As the following example shows, testdoc and pkgdoc can even be created together in a single batch run:

```
qftest -batch -gendoc -testdoc tests -pkgdoc procedures
      directory/with/test-suites
```

Example 15.4: Creating testdoc and pkgdoc documentation in a single run

Of course pkgdoc also supports HTML markup and doctags. The following doctags are supported for Package, Procedure and Dependency nodes:

@deprecated

If a Procedure, Dependency or Package is no longer to be used, this description should explain when and why the node was deprecated and especially which replacement should be used.

@param (Procedure and Dependency only)

A parameter of a Procedure or Dependency. Following are the name of the parameter and its description.

@charvar (Dependency only)

Description for a characteristic variable of a Dependency. Following are the name of the variable and its description.

@return (Procedure only)

The return value of the procedure.

@result (Procedure and Dependency only)

Can be used to document side-effects of the Procedure or Dependency like setting a global variable.

@throws (Procedure only)

Expected exception. Following are the name of the exception and a description of its cause.

@catches (Dependency only)

An exception being caught by the Dependency. Following are the name of the exception and a description of the handler.

@author

Author of the Package, Procedure or Dependency.

@version

Version of the Package, Procedure or Dependency.

@since

The version since which this Package, Procedure or Dependency is available.

In addition to the doctags described above, the doctag '@title' in the comment of the root node can be used to specify a title for the pkgdoc document created for the respective test-suite.

Chapter 16

Test execution

When talking about test execution, there are two aspects to be considered. On one hand you need to run tests while they are developed to check them for proper operation. This situation has already been described in [section 4.2^{\(31\)}](#). Basically all you have to do to run a test interactively is invoking **Run→Start** from the main menu.

On the other hand you want to run your tests periodically to ensure the stability of the system under test, for example in nightly regression tests. Instead of launching QF-Test, loading the test-suite and running it from the graphical user interface, it is much more convenient here to execute tests from the command line in batch mode. This kind of running tests is explained in the first section of this chapter ([Test execution in batch mode^{\(138\)}](#)).

Sometimes, for instance when you want to run the test on a remote computer, a second variant comes into play: the daemon mode. This type of test execution, which uses a running QF-Test instance to execute tests, is the topic of the second section ([Executing tests in daemon mode^{\(144\)}](#)).

For integration of QF-Test with build tools like `ant`, `maven` or Hudson/Jenkins, please refer to [chapter 20^{\(187\)}](#).

16.1 Test execution in batch mode

There are a lot of command line arguments when running QF-Test in batch mode; an overview can be found in [chapter 32^{\(567\)}](#). Here we will present examples showing the most important of them.

The examples are written for the Windows operating system, but you may easily adapt them for the Linux platform. What is different is the path specification and also the syntax for placeholders ([section 32.2.4^{\(587\)}](#)): On Linux you can use `+X` as well as `%X`. On Windows there's a separate console application `qftestc.exe`. In contrast to its

GUI variant `qftest.exe`, it waits until the execution of QF-Test has terminated and also displays print output from a [Server script](#)⁽⁴¹⁶⁾. You can use `qftestc.exe` in place of `qftest.exe` wherever you'll find it convenient.

16.1.1 Command line usage

Let's start with the most simple QF-Test command to execute a test:

```
qftest -batch -run c:\mysuites\suiteA.qft
```

Example 16.1: Test execution from the command line

The argument `-batch` makes QF-Test start without a graphical user interface. The second argument, `-run`, is the specifier for test execution. Finally, at the end of the command line, you find the test-suite to be executed.

Note

The argument `-run` is optional, i. e. the test execution is defined as default for the batch mode.

When running the above command, all top-level Test, Test-case and Test-set nodes of `suiteA.qft` will be executed one after another. After the test-run you will find a run-log file in the current directory; it has the same name as the test-suite (except from the extension, which can be `.qrl`, `.qrz` or `.qzp`). The run-log file shows the result of the test-run.

By specifying `-nolog` you can suppress the creation of a run-log. Probably this only makes sense, if you have extended your test by your own log output (written to a file). Otherwise you'd have to check the result code of QF-Test, whereas 0 means that everything is alright. A positive value in contrast indicates that warnings, errors or exceptions occurred during the test-run (see [section 32.3](#)⁽⁵⁸⁸⁾). That's why in most situations you'll probably prefer to create a run-log and save it at a fixed place in the file system. This can be achieved with the parameter `-runlog`:

```
qftest -batch -compact -runlog c:\mylogs\+b c:\mysuites\suiteA.qft
```

Example 16.2: Test execution with run-log creation

A run-log file `suiteA.qrz` will now be created in the specified directory `c:\mylogs`. The placeholder `+b` is responsible for its name being identical with that of the test-suite. The additional switch `-compact` prevents the run-log from growing too large: Only the nodes needed for a report and those immediately before an error or an exception are kept in the run-log. Especially in case of very long test-runs this may help to reduce the

amount of required memory. In the meantime however, QF-Test offers split run-logs for the same reason. For more information about this topic see [section 8.1^{\(68\)}](#).

Note

Whether the file is indeed created as compressed run-log (to be distinguished from the above "compact") with extension `.qrz`, depends on the system settings. To force the creation of a particular format you can set the file extension explicitly. With `-runlog c:\mylogs\+b.qrl`, for example, an uncompressed XML file will be produced.

Sometimes you may want to execute not the whole test-suite but only parts of it. By using the parameter `-test` you can run a specific node of the test-suite:

```
qftest -batch -runlog c:\mylogs\+b -test "My test-case" c:\mysuites\suiteA.qft
```

Example 16.3: Executing a specified node

The parameter `-test` expects the QF-Test ID attribute of the node to follow or the qualified name of a Test-case or Test-set. If you want to execute several nodes, you can define `-test <ID>` multiple times. Apart from the node's QF-Test ID, `-test` accepts also the numerical index of a top-level node. For example, `-test 0` will run the first child of the Test-suite node.

The run-log provides a rather technical view of the test-run; it is helpful mainly when analyzing errors (cf. [section 8.1^{\(68\)}](#)). The report in contrast contains a summary of the executed test-cases and errors (cf. [chapter 15^{\(129\)}](#)) in XML or HTML format. It is created from the run-log either in a separate step after running the test or automatically with the test-run:

```
qftest -batch -runlog c:\mylogs\+b  
-report c:\mylogs\rep_+b_+y+M+d+h+m  
c:\mysuites\suiteA.qft
```

Example 16.4: Creating a report

In this example the XML and HTML files are saved in a directory which name consists of the test-suite and a timestamp like `c:\mylogs\rep_suiteA_0806042152`. When replacing the argument `-report` with `-report.xml` or `-report.html` respectively, only an XML or HTML report will be created.

Test-cases often uses variables to control the execution of the test. For example, you may have defined the variable `myvar` in the [Test-suite^{\(311\)}](#) node of the suite. You can overwrite its default value when running the test-suite from the command line:

```
qftest -batch -variable myvar="Value from command line"  
-runlog c:\mylogs\+b c:\mysuites\suiteA.qft
```

Example 16.5: Test execution with variables

If needed, you can specify `-variable <name>=<wert>` multiple times to set values for different variables.

16.1.2 Windows batch script

Running tests from the command line is fundamental for integrating QF-Test in test management systems (see [Interaction with Test Management Tools^{\(163\)}](#)). Otherwise, living without such a tool, you may find it convenient to embed the command for the test execution into a script. A simple Windows batch script (`qfbatch.bat`) looks like this:

```
@echo off
setlocal
if "%1" == "" (
    echo Usage: qfbatch Testsuite
    goto end
) else (
    set suite=%~f1
)
set logdir=c:\mylogs
pushd c:\programs\qftest\qftest-4.0.8\bin
@echo on
.\qftest -batch -compact -runlog %logdir%\+b %suite%
@echo off
if %errorlevel% equ 0 (
    echo Test terminated successfully
    goto end
)
if %errorlevel% equ 1 (
    echo Test terminated with warnings
    goto end
)
if %errorlevel% equ 2 (
    echo Test terminated with errors
    goto end
)
if %errorlevel% equ 3 (
    echo Test terminated with exceptions
    goto end
)
if %errorlevel% leq -1 (
    echo Error %errorlevel%
    goto end
)
:end
popd
```

Example 16.6: Batch script `qfbatch.bat` to execute a test-suite

Now you can simply run that script with only the file name of the test-suite as parameter. Everything else is done automatically: The test-suite will be executed, the run-log file stored in `logdir` and finally the script will print out the state of the test-run (depending on the QF-Test result code).

16.1.3 Groovy

3.0+

Since version QF-Test 3 the language Groovy is part of the release (cf. [chapter 13^{\(99\)}](#)). It is meant mainly for scripting inside QF-Test (Server and SUT scripts), but it can, like Jython, also be used outside of QF-Test. Groovy is probably well suited to create a little test execution management system by yourself. By the way, Groovy simplifies working with Ant, too: Instead of dealing with bulky XML files, which makes it hard to define conditions, you can work with the Groovy `AntBuilder`. However, that's out of scope here, the following example doesn't rely on Ant but only on the basic Groovy features:

```
def suite = ''
if (args.size() == 0) {
    println 'Usage: groovy QfExec Testsuite'
    return
}
else {
    suite = args[0]
}
def qftestdir = 'c:\\programs\\qfs\\qftest\\qftest-4.0.8'
def qftest = qftestdir + '\\bin\\qftest.exe'
def command = "$qftest -batch -compact -runlog c:\\mylogs\\+b \"$suite\""
def printStream = { stream ->
    while (true) {
        try {
            stream.eachLine { println it }
        } catch (IOException) {
            break
        }
    }
}
println "Running command: $command"
def proc = command.execute()
new Thread().start() { printStream(proc.in) }
new Thread().start() { printStream(proc.err) }
proc.waitFor()
switch (proc.exitValue()) {
    case '0': println 'Test terminated successfully'; break
    case '1': println 'Test terminated with warnings'; break
    case '2': println 'Test terminated with errors'; break
    case '3': println 'Test terminated with exceptions'; break
    default: println "Error ${proc.exitValue()}"
}
```

Example 16.7: Groovy script QfExec.groovy to execute a test-suite

If you have Groovy installed on your computer independently of QF-Test, you can run the example test-suite simply via `groovy QfExec c:\mysuites\suiteA.qft`. Otherwise you can use the Groovy jar file from the QF-Test installation, preferably again with help of a batch script:

```
@echo off
setlocal
if "%1" == "" (
    echo Usage: qfexec Testsuite
    goto end
)
set qftestdir=c:\programs\qftest\qftest-4.0.8
set scriptfile=QfExec.groovy
java -cp %qftestdir%/lib/groovy-all.jar groovy.ui.GroovyMain %scriptfile% %*
:end
```

Example 16.8: Batch script `qfexec.bat` to run a Groovy script (here: `QfExec.groovy`)

Now execute the test-suite with `qfexec c:\mysuites\suiteA.qft`.

16.2 Executing tests in daemon mode

In daemon mode QF-Test listens to RMI connections and provides an interface for remote test execution. This is useful for simplifying test execution in a distributed load-testing scenario ([chapter 21^{\(197\)}](#)), but also for integration with existing test-management or test-execution tools ([chapter 19^{\(163\)}](#)).

16.2.1 Launching the daemon

!!! Warning !!!

Anybody with access to the QF-Test daemon can start any program on the machine running the daemon with the rights of the user account that the daemon is running under, so access should be granted only to trusted users.

If you are not running the daemon in a secure environment where every user is trusted or if you are creating your own library to connect to the QF-Test daemon, you definitely should **read [section 40.3^{\(755\)}](#)** about how to secure daemon communication with SSL.

To work with a daemon, you must first launch it on any computer in your network (of course, this host can also be `localhost`):

```
qftest -batch -daemon -daemonport 12345
```

Example 16.9: Launching a QF-Test daemon

Note

Important compatibility note:

3.5+

Starting with QF-Test version 3.5, SSL is used for daemon communication by default. To interact with a QF-Test version older than 3.5 you must start the daemon with an empty `-keystore <keystore file>`⁽⁵⁷⁷⁾ argument in the form:

```
qftest -batch -keystore= -daemon -daemonport 12345
```

Example 16.10: Launching a QF-Test daemon without SSL

If you omit the argument `-daemonport`, the daemon will listen on QF-Test's standard port 3543. You may check whether the daemon is running by means of the `netstat` utility:

Windows

```
netstat -a -p tcp -n | findstr "12345"
```

Linux

```
netstat -a --tcp --numeric-ports | grep 12345
```

If you want to launch a daemon on a remote host, you may use for instance `ssh` or `VNC`. Your network administrator knows whether and how this works. To follow the examples below, a local daemon will be sufficient.

16.2.2 Controlling a daemon from QF-Test's command line

3.0+

The easiest way to get in touch with a daemon is running QF-Test from the command line in the *calldaemon* mode. The following example checks if a daemon is listening at the specified host and port:

```
qftestc -batch -calldaemon -daemonhost localhost -daemonport 12345 -ping
```

Example 16.11: Pinging a QF-Test daemon

In contrast to the `netstat` command from above `-ping` also works between different computers (if you check the daemon on your local computer, you can omit the argument `-daemonhost`).

What you actually want from a daemon is executing your test-case(s) and getting back a run-log file. It sounds and indeed looks quite similar to what you have seen before when running a test in batch mode:

```
qftest -batch -calldaemon -daemonhost somehost -daemonport 12345
        -runlog c:\mylogs\+b
        -suitedir c:\mysuites
        suiteA.qft#"My test-case"
```

Example 16.12: Running a test-case with the QF-Test daemon

Note

In contrast to the batch mode, a Test-case or a Test-set node is always referenced here by its qualified name, for instance "My Test-set.My Test-case" (just to remember: `-test <ID>` may be used in batch mode). To execute the complete suite `suiteA.qft`, you can simply omit the test-case or write `suiteA.qft#..`.

If the daemon is running on a remote host, you have to specify it explicitly via `-daemonhost` (default is `-daemonhost localhost`). Note that the parameter `-suitedir` refers to the remote host (where the daemon is running) while `-runlog` defines a local file.

3.4+

In case you cannot easily observe the test running on a remote host, you may find it convenient to add the argument `-verbose` to get status output in the console (on Windows, use `qftestc` to see the output).

A running daemon, no matter whether local or remote, can be terminated with the *calldaemon* command `-terminate`:

```
qftest -batch -calldaemon -daemonport 12345 -daemonhost localhost -terminate
```

Example 16.13: Terminating a QF-Test daemon

A complete list of the *calldaemon* parameters can be found in the chapter Command line arguments and exit codes⁽⁵⁶⁷⁾.

16.2.3 Controlling a daemon with the daemon API

Using the QF-Test command line to control a daemon was quite easy. On the other hand, to get all capabilities of a daemon, you have to deal with the daemon API. In this section we will concentrate on some basic examples, the whole interface is described in chapter 40⁽⁷⁴¹⁾.

To get started with the daemon API, insert a Server script node with the following code:

```

from de.qfs.apps.qftest.daemon import DaemonRunContext
from de.qfs.apps.qftest.daemon import DaemonLocator
host = "localhost"
port = 12345
# Leading r means raw string to allow normal backslashes in the path string.
testcase = r"c:\mysuites\suiteA.qft#My test-case"
timeout = 60 * 1000
def calldaemon(host, port, testcase, timeout=0):
    daemon = DaemonLocator.instance().locateDaemon(host, port)
    trd = daemon.createTestRunDaemon()
    context = trd.createContext()
    context.runTest(testcase)
    if not context.waitForRunState(DaemonRunContext.STATE_FINISHED, timeout):
        # Run did not finish, terminate it
        context.stopRun()
        if not context.waitForRunState(DaemonRunContext.STATE_FINISHED, 5000):
            # Context is deadlocked
            raise UserException("No reply from daemon RunContext.")
        rc.logError("Daemon call did not terminate and had to be stopped.")
    result = context.getResult()
    log = context.getRunLog()
    rc.addDaemonLog(log)
    context.release()
    return result
result = calldaemon(host, port, testcase, timeout)
rc.logMessage("Result from daemon: %d" %result)

```

Example 16.14: Daemon API in a Server script

The script shows the basic mechanisms to control a daemon:

- First find a running daemon with `locateDaemon`.
- Provide an environment for test-runs by calling `createTestRunDaemon`.
- To run a test, you need a context object (`createContext`). The creation of that object requires a QF-Test run-time license.
- Now the context enables you to start a test-run (`runTest`) and to query about its current state. `waitForRunState` waits during the defined timeout (in milliseconds) until the specified state has occurred. In the example above, we wait for the test to terminate within one minute.
- Finally, when the test-run has terminated, the context can query the test result with the method `getResult` (cf. [Exit codes for QF-Test^{\(588\)}](#)).
- Moreover, you can use the context to get the run-log of the daemon test-run. It can be included in the local run-log by means of the `rc` method `addDaemonLog`.

Note To keep it small and simple, the example script does not contain any error handling. However, particularly when working with a daemon, you should check every method call.

Note Driving a daemon from a Server script has the disadvantage of consuming an additional QF-Test license to run the script node interactively or in batch mode. However, this doesn't apply nor for the above-mentioned *calldaemon* mode neither for the case when controlling a daemon outside QF-Test (see below).

The usage of the daemon API is not restricted to Server scripts. Outside QF-Test a daemon can be contacted by means of a Java program or, more easily, a Groovy script. The following Groovy script works with several running daemons and may serve as a starting point for load tests. Suppose we have started some daemons in our network, each on a separate machine. We want to execute a test-case simultaneously by all of the daemons and we want to save a run-log for every single test-run (*daemon1.qrl*, ..., *daemonN.qrl*). The test-suite containing the test-case to be executed may be available for all daemon instances via the network drive *z:*).

```
import de.qfs.apps.qftest.daemon.DaemonLocator
import de.qfs.apps.qftest.daemon.DaemonRunContext
def testcase = "z:\\mysuites\\suiteA.qft#My test-case"
def logfile = "c:\\mylogs\\daemon"
def timeout = 120 * 1000
def daemons = DaemonLocator.instance().locateDaemons(10000)
def contexts = []
// Start tests
for (daemon in daemons) {
    def trd = daemon.createTestRunDaemon()
    trd.setGlobal('machines', daemons.size().toString())
    def context = trd.createContext()
    contexts << context
    context.runTest(testcase)
}
// Wait for tests to terminate
for (i in 0..<contexts.size()) {
    def context = contexts[i]
    context.waitForRunState(DaemonRunContext.STATE_FINISHED, timeout)
    byte[] runlog = context.getRunLog()
    def fos = new FileOutputStream("$logfile${i + 1}.qrl")
    fos.write(runlog)
    fos.close()
    context.release()
}
```

Example 16.15: Groovy daemon script *CallDaemon.groovy*

To run that Groovy script, you need the QF-Test libraries *qftest.jar*, *qfshared.jar*, and *qflib.jar* as well as the Groovy library, which is also part of

the QF-Test installation. The following batch script shows how it works:

```
@echo off
setlocal
set qftestdir=c:\programs\qftest\qftest-4.0.8
set qflibdir=%qftestdir%\qflib
set classpath=%qftestdir%\lib\groovy-all.jar
set classpath=%classpath%;%qflibdir%\qftest.jar;%qflibdir%\qfshared.jar;
    %qflibdir%\qflib.jar
java -cp %classpath% groovy.ui.GroovyMain CallDaemon
```

Example 16.16: Batch script `calldaemon.bat` to run `Calldaemon.groovy`

To make the daemon example a load test (cf. [chapter 21^{\(197\)}](#)), you have to synchronize the test-runs inside of "My test-case" (e. g. after starting the SUT). This can be done by means of the `rc` method `syncThreads`:

```
def machines = rc.getNum('machines')
rc.syncThreads('startup', 60000, -1, machines)
```

Example 16.17: Groovy Server script node to synchronize the test-runs

The variable `machines` denotes the number of hosts with a daemon running on them. Best define it in the Test-suite node of the test-suite with a default value of 1. When running the Groovy script, it will be overwritten with the correct value.

Chapter 17

Distributed test development

The previous chapters all focused on creating and organizing sets of reliable tests in a single test-suite. However, when testing a large application, a single suite may not be enough. There are at least two scenarios where splitting tests into multiple test-suites becomes essential:

- Multiple developers are creating and editing tests. To avoid redundancy and duplication of code, separately developed tests should use common Procedures and Components where possible, but only one person can edit a test-suite at a time.
- Tests are simply getting too large. Run-logs for extensive test-runs may cause the system to run out of memory and organizing a large number of tests in a single suite is difficult. Things may become unwieldy. It may also be desirable to be able to run some of the tests as part of the whole test as well as standalone.

QF-Test provides a set of advanced features that make it possible to split and arrange tests across a set of test-suites. Multiple developers can work on separate parts of a test, then coordinate their efforts, merge the Components of their suites and create libraries of shared Procedures.

This chapter first explains the various mechanisms for distributed test development and how they interact. The final section then summarizes these in concise step-by-step instructions on how to approach large testing efforts with QF-Test.

17.1 Referencing nodes in another test-suite

It is possible to reference Procedures⁽³⁷⁸⁾ and Components⁽⁵⁴⁵⁾ in a test-suite other than the current one. These references can be explicit or implicit through included files:

- Explicit references use a syntax similar to the one used in URLs to specify an item inside a web page. The referenced suite must be prepended to the Procedure name⁽³⁸²⁾ attribute of a Procedure call⁽³⁸¹⁾ of the QF-Test component ID attribute of a Component dependent node, separated by a '#' character. The usual *packagepath.procedure* becomes *suite#packagepath.procedure*.
- Implicit references make use of the Include files⁽³¹²⁾ attribute of the Test-suite⁽³¹¹⁾ node. Whenever a node is not found in the current suite, QF-Test will search for a matching Procedure or Component within all the suite's directly or indirectly included files (a file is considered indirectly included by a suite if it is found as an included file within one of the suite's own included files; for example, if suite A includes B, and suite B includes C, then C is indirectly included by A).

A test-suite that references a node in another test-suite becomes dependent on that suite. If the Name of a Procedure or the QF-Test ID of a Component in the referenced suite changes, the suite with the reference must get updated, otherwise the link is broken and the suite will no longer work correctly. In such cases QF-Test will automatically update references if it knows about them. The best way to ensure that is to have both test-suites in a common project because QF-Test automatically tracks all includes and all explicit references within a project. Alternatively you can list the calling suite in the Dependencies (reverse includes)⁽³¹²⁾ attribute of Test-suite root node of the referenced suite.

While implicit references are more convenient in most cases, they can make tests harder to understand because it is not immediately obvious where the Procedure or Component referenced by some node is actually located. One possibility to find out is to select "Locate procedure" (**Ctrl-P**) or "Locate component" (**Ctrl-W**) from the context menu. Additionally, QF-Test provides the menu items **Operations→Make references explicit** and **Operations→Make references implicit** which let you toggle quickly between the two modes without changing the actually referenced nodes.

In both cases, the referenced suite can either be given a relative or absolute filename. Relative filenames will be resolved relatively to the directory of current suite, or - if that fails - relatively to the directories on the library path (see option Directories holding test-suite libraries⁽²⁵²⁾). Always use the forward '/' as the directory separator, even under Windows. QF-Test will map it to the correct character for the system it runs on. This keeps your test-suites independent from the operating system.

Note

Your Package and Procedure names as well as Component QF-Test IDs should not contain any '\ ' or '#' characters. If they do, you need to include an escape character in the Procedure call or the QF-Test component ID attribute. See section 36.6⁽⁶¹⁶⁾ for details about escaping and quoting special characters.

When choosing the Procedure for a Procedure call or the Component for some event in the dialog, QF-Test offers a selection of all currently opened test-suites. If a Procedure or Component from another test-suite is selected, QF-Test automatically creates the correct

reference, taking included suites into account. When the test is run at a later time, the referenced test-suite is loaded automatically if necessary.

During execution QF-Test keeps a stack of currently executing suites. Whenever a Procedure is called in another suite, the called suite is pushed on to the top of this stack and removed when execution returns to the calling suite. Whenever during the execution of a Procedure a Window or Component is referenced by its QF-Test ID, QF-Test searches through this stack of suites from the top to the bottom, i.e. first in the test-suite of the called Procedure and then in the calling suite, always checking any included files along the way. This process is quite complicated and you should take care to keep your include hierarchies simple. In case you encounter problems anyway, a detailed explanation is given in [section 36.7^{\(617\)}](#).

17.2 Managing Components

As we have emphasized in [chapter 5^{\(37\)}](#), the Components are the essential part of a test-suite. If the SUT changes between releases, these will likely be affected most. If changes are so massive that QF-Test cannot adapt automatically, the Components will have to be updated manually. This is why you should try to avoid redundancy in the Component hierarchy of your tests more than in any other part.

Therefore, when splitting your tests across multiple test-suites you should try to keep the Components together in one central test-suite and include this suite from the other suites. For very large applications you may want to split the Component hierarchy into parts, each related to a separate part of the SUT's GUI.

Maintaining this central Component library is not trivial. The problems that will arise can be resolved with QF-Test as follows:

- When multiple test developers are recording new Components simultaneously, they cannot be integrated immediately into the central suite, because only one user can edit the central suite at a time. Instead, Components must be integrated later by importing them into the central suite when the new tests have stabilized. This is explained in the following section.
- When the SUT changes, Components in the central suite may need to be updated. If this involves changing any Component QF-Test IDs, this will break any references to these Components from other suites. To avoid that, QF-Test must update those references and it will do so, provided that the suites that depend on the central suite are currently loaded, belong to the same project or are listed in the [Dependencies \(reverse includes\)^{\(312\)}](#) attribute of the [Test-suite^{\(311\)}](#) node of the central suite.

17.3 Merging test-suites

Test-suites can be merged by importing one test-suite into another with the File→Import... menu item.

You can select the areas of the test-suite, which should be imported.

You have to take care about a correct Include/Reverse-Include of your test-suites to ensure, that all calls and component references are still valid. See [chapter 25^{\(213\)}](#) for details.

17.3.1 Importing Components

During import, all Windows and Components of the imported test-suite are integrated into the component hierarchy of the importing suite. Components that already exist are not copied. An QF-Test ID conflict (identical components with different QF-Test IDs or differing components with identical QF-Test IDs) is resolved automatically by changing the QF-Test ID of the imported component.

Afterwards, all Windows and Components are removed from the imported suite. All nodes in the imported suite that referred to these Components are updated accordingly. Ideally, the imported suite should include the importing suite so no explicit suite references will have to be created.

17.3.2 Importing Procedures and Testcases

As you can import Components QF-Test also allows to import Procedures, Packages, Dependencies and Test-cases as well as Test-sets by choosing 'Procedures' or 'Tests' in the import dialog. You should take care about keeping all calls consistent, e.g. in most cases it does not make sense to import Procedures without their required Components.

In case you only want to import one dedicated Procedure or Test-case you can use the button 'Detailimport' on the importdialog. Here you can choose any node you want to import separately.

17.4 Strategies for distributed development

There is no single *best way* of test development or organization, but one approach that works well is the following:

- Start with a central test-suite that has the functionality needed to start and stop the

SUT and a basic set of Tests and Procedures. This will become your master suite which will contain all Components.

- Make sure that your developers have understood the importance of assigning names with `setName()` and that unique names are assigned consistently where needed. Where `setName()` is not an option, try to implement `ComponentNameResolvers` to achieve this (see [section 39.1^{\(678\)}](#)). You should be able to record and replay sequences without much ado and without "polluting" the Component hierarchy after trivial changes in the user interface.
- Move as much functionality as possible into Procedures, especially commonly-used stuff and the setup and cleanup routines for the SUT.
- To create new tests, start with an empty test-suite. Include the master test-suite by editing the `Include files(312)` attribute of the `Test-suite(311)` node of the new suite. Create the Setup and Cleanup nodes to start and stop the SUT by calling the respective Procedures in the master suite.
- Create your tests as required. When recording sequences, the Components of the master-suite will be used if possible. New Components are added to the new suite, so the master suite will not be modified at this stage.
- Where possible, call Procedures in the master suite for common operations.
- When your new set of tests is complete and you are satisfied that they work well, import any required nodes of your new test-suite into the master suite. You have to ensure that all new Component nodes that you recorded are imported into the master suite's Component hierarchy in any case. The master suite's existing Components will not be affected by this, so other suites that depend on the master suite will not need to be modified.
- After importing Components you can import all or only the required Procedures into the master suite.
- You now have various options of how to arrange the actual sequences of events and checks that form your tests. In any case it is a good idea to move everything to Procedures and Packages structured after your test-plan. Then the top-level Tests nodes of the master suite and your new suite will only contain the required hierarchy of Test and Sequence nodes filled with Procedure calls to the actual test-cases. Such an arrangement has several advantages:
 - All your tests are structured cleanly.
 - You can easily create different sets of tests with varying complexity and run-time.

- You have the option to keep the test-cases in separate test-suites and have the master suite call them. These “test-case-libraries” must include the master-suite, so they need not contain any Components themselves. You can organize your tests so that the master-suite will run the whole set of tests, while each separate suite can also be run standalone.
- The tests can be maintained by several developers as long as modifications to the master suite are coordinated.
- If you decide to keep your new tests in the newly created test-suite instead of moving them to the master suite, modify the master suite to tell QF-Test that there is a new test-suite that depends on it. To do so, either ensure that both test-suites belong to the same project or add the new test-suite to the Dependencies⁽³¹²⁾ attribute of the master suite’s Test-suite node.
- If you need to modify or extend the new test-suite later, proceed as before. You can record new sequences as needed. When you are done, merge any newly created Components back into the master suite.
- If your SUT changes in a way that requires updates or adaptations to the master-suite’s Component hierarchy, you must coordinate your test developers. Before you start updating the Components, make sure that all suites that directly or indirectly include the master suite belong to the same project as the master suite or are listed in the Dependencies attribute of the master suite’s Test-suite node. If modifying the Components of the master suite involves any QF-Test component ID changes, QF-Test will update the depending test-suites accordingly, so they should not be edited simultaneously by others.
- The file format for QF-Test test-suites is XML and thus plain text. As a result, test-suites can be managed very well by version control systems like CVS. Changes to some QF-Test component ID attributes of the depending suites can typically be merged with other changes without conflicts, alleviating the need for coordination.

Of course, the above scheme can be extended to have several master suites for testing different parts or aspects of an application. It may be a good idea to ensure that the component hierarchies in these suites don’t overlap too much though. This will save you the effort of maintaining all these hierarchies in case the user interface of the SUT changes significantly.

17.5 Static validation of test-suites

Working in a project over time will cause modifications, refactoring or deletion of steps in your test-suite structure, e.g. you may consider renaming Procedures or simply removing them once they are not required anymore.

17.5.1 Avoiding invalid references

In such cases it is quite important that you adapt all references of the according Procedure in order to guarantee that the remaining tests keep running. For this purpose QF-Test automatically updates all references during the process of renaming or moving elements on demand.

If you want to ensure that your created test structure doesn't contain any call of non-existing Procedures anymore, you can also use the "Analyze references" command of QF-Test in order to perform a static validation of your test-suite. This command will open a dialog showing all references and whether they are still okay or something is missing.

You can trigger the analysis via a right mouse-click and selecting **Additional node operations→Analyze references...** or selecting the according entry from the main menu under **Operations**. This method is also available in batch mode.

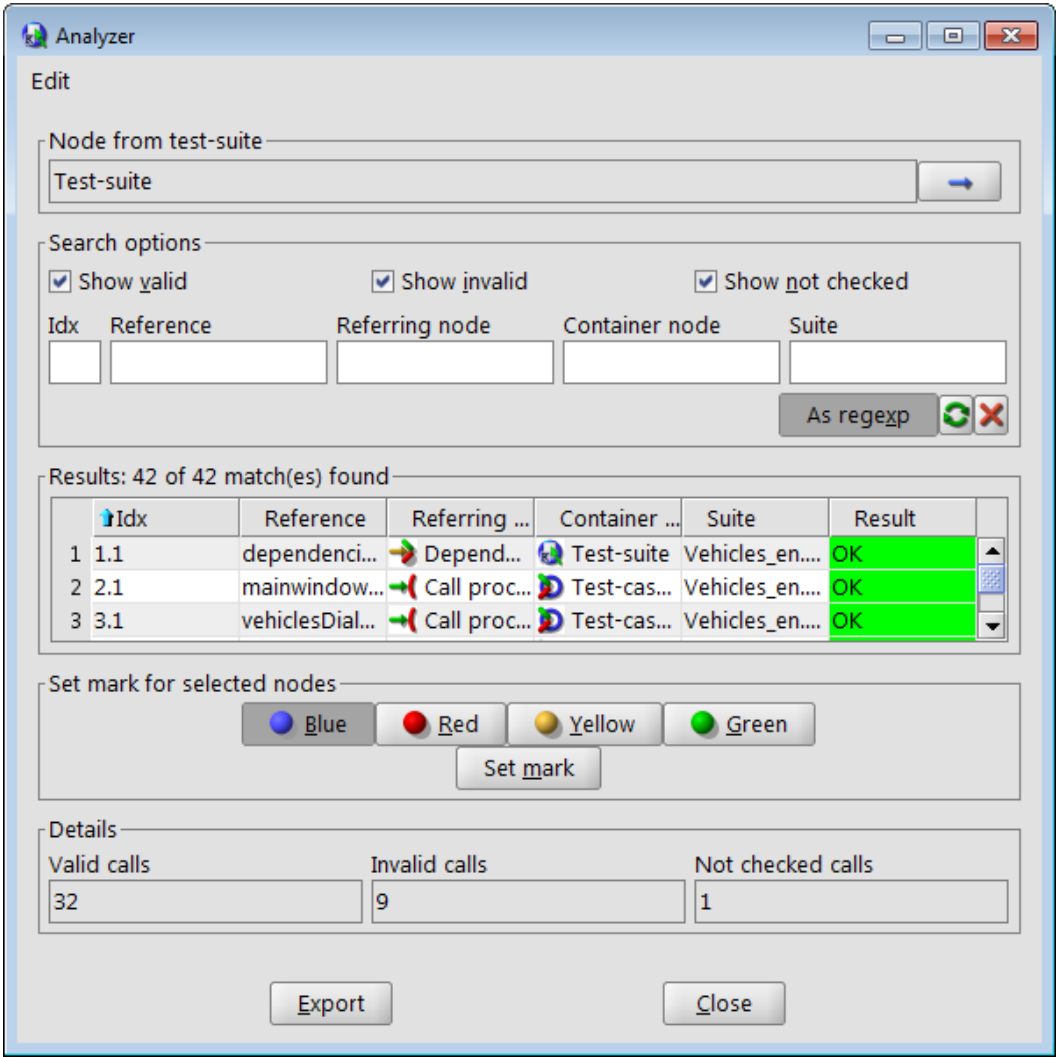


Figure 17.1: Result of analyzing references

3.5+

QF-Test also provides features to search through your test-suites for duplicate nodes, empty Packages or Procedures or to analyze for nodes having invalid characters in their names.

This kind of static validation is available for Procedures, Dependencies, Test-cases, Test-sets and Components and their references.

17.5.2 Unused procedures

4.0.3+

During test development it could happen that procedures, which were used in the first version of your tests will not be used in newer versions due to re-factoring of tests.

If those procedures won't get deleted immediately they will stay in the test-suite and the test-suite will grow and grow. Sometimes you could get the feeling that you have too many procedures or that you have lost the overview of your procedures. In order to check for such unused procedures or dependencies in your test-suite you can open the context menu via a right mouse click at Test-suite or Procedures and select **Additional node operations→Find unused callables...**. This operation creates a report showing any procedures or dependencies which had been created but haven't been used yet. Now you could decide what you want to do with those.

Sometimes you might simply remove all of those unused nodes immediately via **Additional node operations→Remove unused callables**.

Chapter 18

Automated Creation of Basic Procedures

3.0+

18.1 Introduction

At the beginning of a typical QF-Test project the tester records the first tests and starts them. After a couple of such recordings and first success stories he notices that only recording or performing copy/paste bares some hidden pitfalls in maintaining the tests. Just think about a possible workflow change in a main panel, then the tester might have to adapt all test-cases. That's why we recommend to make use of the modularization concept using procedures and variables as early as possible in a project. For more information about the modularization concept, please see [section 9.3^{\(82\)}](#).

In projects containing a lot of dialogs and graphical components it might be sufficient to split those procedures into component-specific ones, e.g. "press button ok" and separate workflow procedures, e.g. "create a vehicle" combining the component-specific steps together. This approach enables the tester to create new test-cases very fast. However, he has to put a lot of efforts into creating those basic procedures first.

QF-Test comes with a Procedure Builder, which will create those basic procedures for you. Using the Procedure Builder will drastically decrease the efforts of recording and creating procedures for graphical components. So the tester can solely concentrate on his main focus, i.e. designing the workflow of the test itself and the according test-data.

18.2 How to use the Procedure Builder

For creating the basic procedure automatically, you have to perform following steps:

- Start the SUT from QF-Test.

- Navigate to the window or frame you want to create procedures for in the SUT.
- Press the 'Record Procedures' button, select the according menu-item in QF-Test or use the Hotkey for procedure recording⁽²⁶⁹⁾.
- Perform a right mouse-click on the respective component in the SUT.
- Select the according recording-mode.
- Stop procedure recording by releasing the 'Record Procedures' button, by deselecting the according menu-item in QF-Test or using the Hotkey for procedure recording⁽²⁶⁹⁾.

Now you should be able to find a newly created package in the Procedures⁽³⁸⁷⁾ node of the test-suite where you stopped the recording, containing the created procedures for the components. By default this is called `procbuilder`. If the package `procbuilder` already exists, a package `procbuilder1` will be created and so on. When you open the package you will see a set of packages for functions like `check`, `get`, `select`, `wait` and so on. These contain the procedures created for the components you selected when running the procedure builder, apart from the `check` package, where another level with packages for the various check modes is inserted. This is the default structure, which you can adapt to your needs by modifying the definition file for the procedure builder as described in the following section.

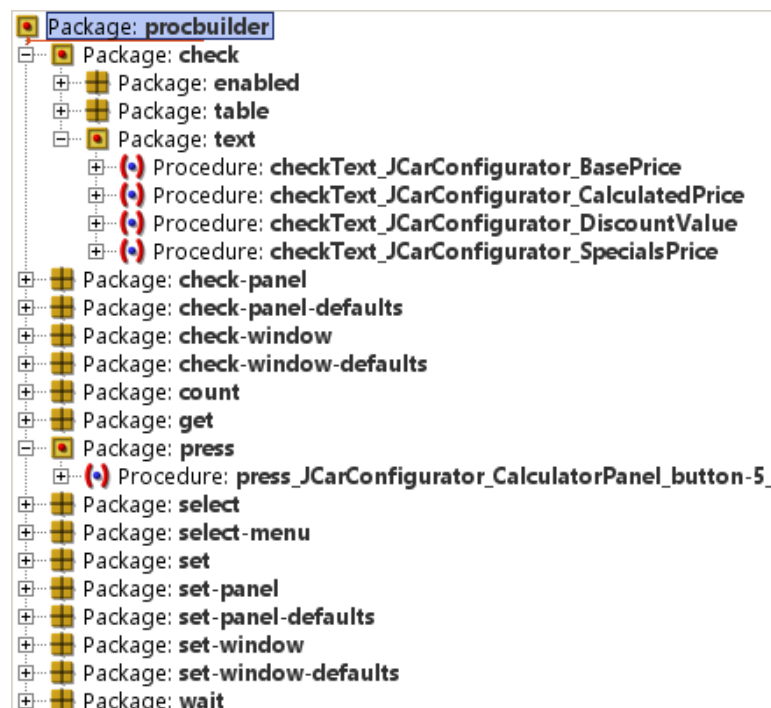


Figure 18.1: Recorded procedures

18.3 Configuration of the Procedure Builder

The act of building procedures is controlled by a template suite, which is located at `qftest-4.0.8/include/procbuilderdef.qft`. This file should be copied to any project-specific location, if you want to adapt it to your project. You can define its location in the options at [Configuration file for recorded procedures](#)⁽²⁷⁰⁾.

The template suite contains procedures for the most common GUI elements and actions. If you require other test-steps, you can add the according procedure to this test-suite.

The file itself is a test-suite with a dedicated structure. You can find a detailed explanation of this structure in the subsection [section 18.3.1](#)⁽¹⁶¹⁾. The definition file allows the tester to define procedures for components of dedicated classes or to define procedures for working with all components of one certain window.

You can find some demo configurations at `qftest-4.0.8/demo/procbuilder`.

18.3.1 The Procedure Builder definition file

The automated creation of basic procedures delivers different procedures depending on the components. A text-field requires a setter procedure for setting its text, a button requires a press procedure for pressing it or a window could require a setter which calls the setter procedures of all text-fields or combo-boxes on that window to call just one procedure for using the window etc..

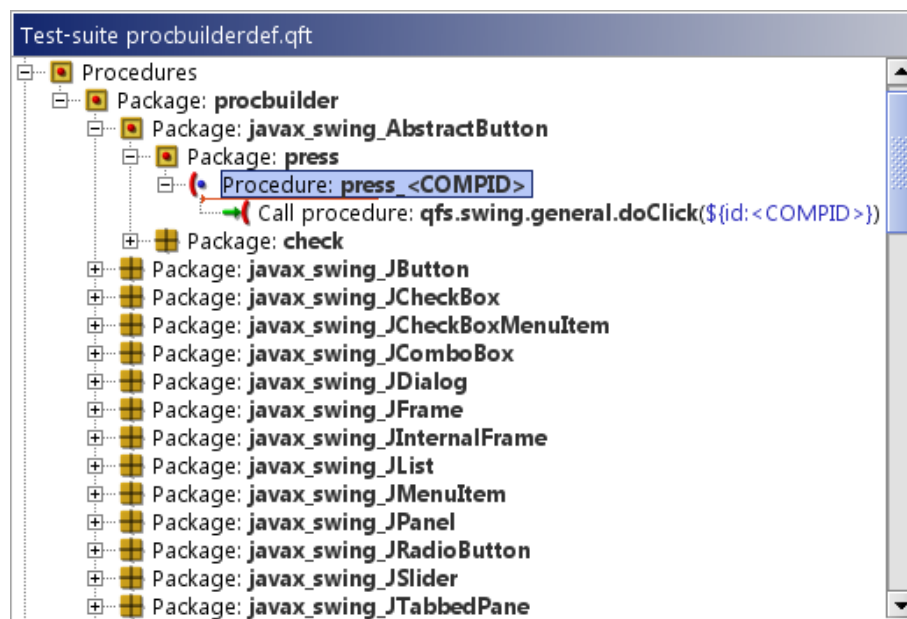


Figure 18.2: The Procedure Builder definition file

The topmost package in the Procedures⁽³⁸⁷⁾ node is the name of the target package for the newly created packages. By default this is called `procbuilder`. This package will be inserted below the Procedures⁽³⁸⁷⁾, when you have finished recording procedures. If the package `procbuilder` already exists, a package `procbuilder1` will be created and so on.

The next level is the class level. Here you can define a package per class. The package name represents the full class name, but with `'_'` as separators instead of `'.'`. That's because `'.'` is not allowed in package names. The Procedure Builder creates the procedures also for descendants of specified classes. In case the names of your classes contain a `'_'`, you have to mark this via `'\_'`.

The following levels can be chosen freely because those levels are intended to structure the procedures.

At the last level you have to define the steps of the procedure itself.

Of course there are a lot of variable data in that definition, e.g. like `<COMPID>`.

Using those you can specify variables for the procedure names, like the QF-Test ID of the current component or the component-name. You can also record the current value of the text-field or the current selected status of a checkbox. It's even possible to keep the package structure variable. For an overview of all possible variables, please see chapter 41⁽⁷⁵⁸⁾.

Chapter 19

Interaction with Test Management Tools

3.0+

QF-Test contains some pragmatic test management approaches, like creating a test-case overview or documenting test-cases within QF-Test. In bigger projects it might be necessary to make use of an own dedicated test management system to track the development status of test-cases or to link test-cases and their results to defects, use-cases or features. Besides support for planning of test-cases and tracking their results a test management system could also contain a test execution engine, which supervises the occupation of test-systems during different test-runs.

As QF-Test doesn't come with all of those features though continuously improving in that area, it is very easy to integrate QF-Test with such a test management or test execution system using the QF-Test Batch mode or the QF-Test Daemon mode. For more information about the Batch mode or the Daemon mode, please see [chapter 16^{\(138\)}](#).

The following chapters describe some exemplary solutions which we provide for established test management systems. If you cannot find your test management system in that list, please contact us or our mailing-list to get hints about a possible integration approach.

19.1 HP ALM - Quality Center

19.1.1 Introduction

The current integration of QF-Test and HP ALM - Quality Center utilizes the built-in VAPI-XP-TEST type of Quality Center.

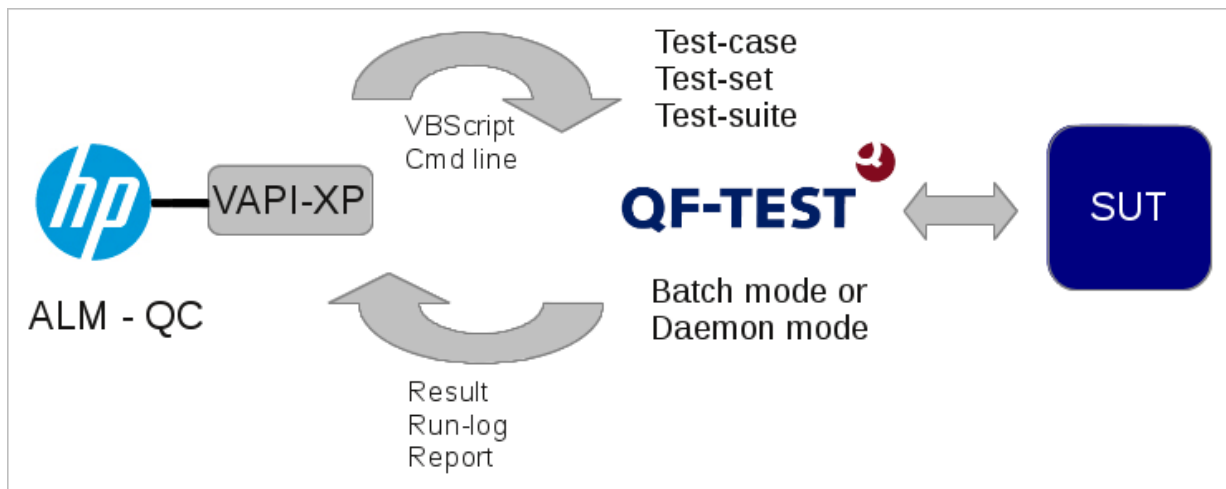


Figure 19.1: Integration with ALM - QualityCenter

The VAPI-XP-TEST type is intended to be an automated test-case for any test-tool. QF-Test comes with a template file for the VAPI-XP-TEST script, which is `qcVapiXPTemplate.txt`, see `qftest-4.0.8/ext/qualitycenter`. This script can be used as template for all QF-Test tests in Quality Center. Please see [section 19.1.2^{\(165\)}](#) for a detailed step-by-step description.

The QF-Test VAPI-XP-TEST template script employs an external worker VBScript script, called `qcTemplate.vbs`. This script is also part of the QF-Test distribution (see `qftest-4.0.8/ext/qualitycenter`) and has to be adapted to your specific needs. So we encourage you to copy that file to a project specific location and adapt it according to your needs.

The worker script launches QF-Test in batch mode on each test system locally, i.e. it has to be accessible for each test system. As the test suite files and the configuration files have to be available on the test system too, we recommend to put all those files on a shared network drive or into the version management system.

After the execution of the test the run-log of QF-Test will be appended to the test instance as well as the status of the test will be set to the result.

You can also change the worker script to make use of a daemon call (for details about the daemon mode, please see [chapter 40^{\(741\)}](#)). In this case QF-Test will establish the network connection to the test system and launch the test by itself. In case of the normal batch call Quality Center establishes the connection to the test-system and triggers the local QF-Test installation to perform the test. If you make use of the daemon call, the worker script has to be located on the Quality Center system, but the test suite still needs to be accessible on each test system.

If you do not make use of VBScript in your project, feel free to port the QF-Test demo

scripts to JScript or any other supported language.

The following figure shows the VAPI-XP-TEST test-case in Quality Center:

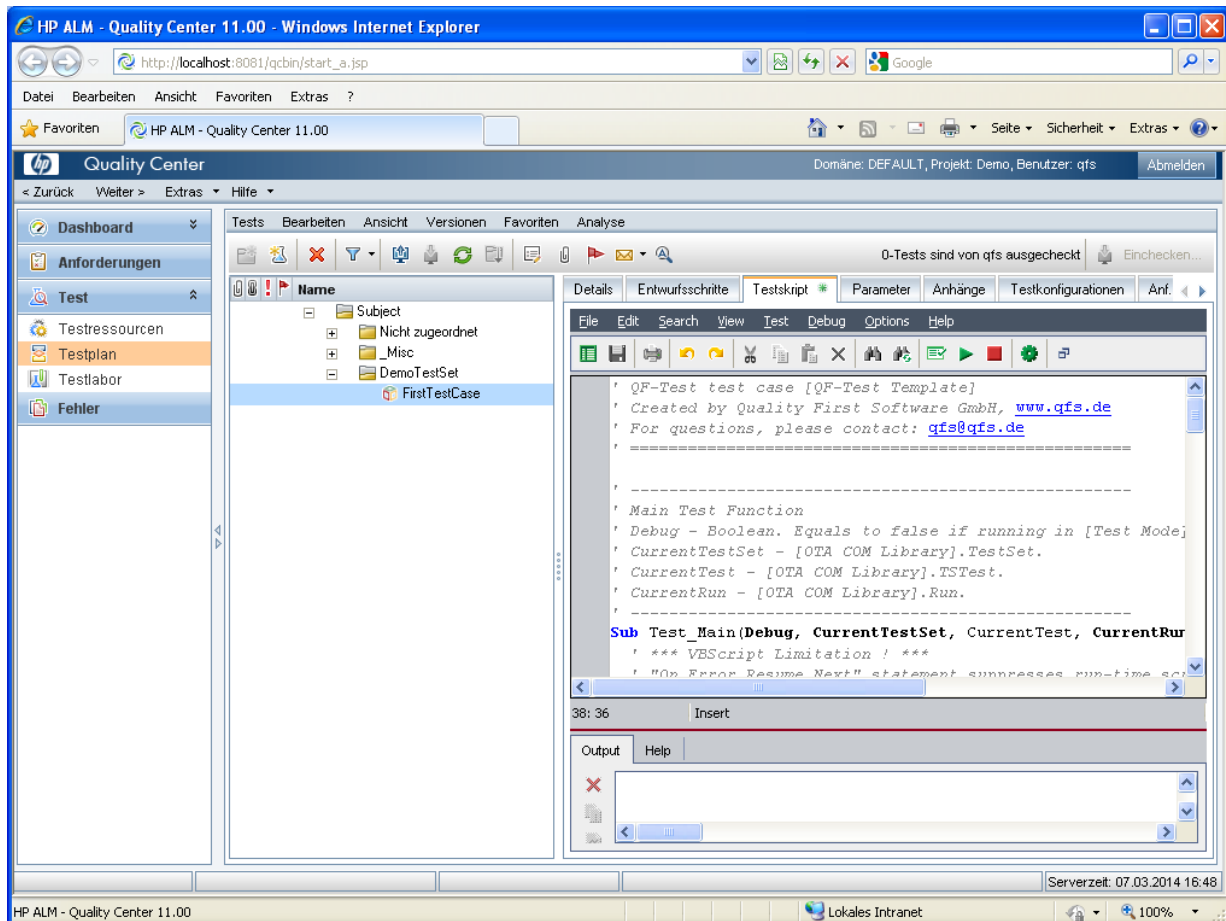


Figure 19.2: QF-Test VAPI-XP-TEST test-case in HP ALM - QualityCenter

19.1.2 Step-by-step integration guide

General steps to be performed on the the test system:

1. Copy the template worker script from `qftest-4.0.8/ext/qualitycenter/qcTemplate.vbs` to your project location and rename it to a proper name. We recommend to use the same path on all test-systems. Perhaps you should use a shared network drive.
2. Within the worker script you can define certain default option, e.g. whether the batch or daemon mode should be used as default and what should be the name

for the default run-log file. This also can be done at a later stage, which might be recommendable when initially starting with the integration process to keep things simple.

Steps in Quality Center to create a test-case:

1. Start Quality Center and log in to your project.
2. You might want to create a new test-set e.g. called "DemoTestSet" in the "Test plan" area.

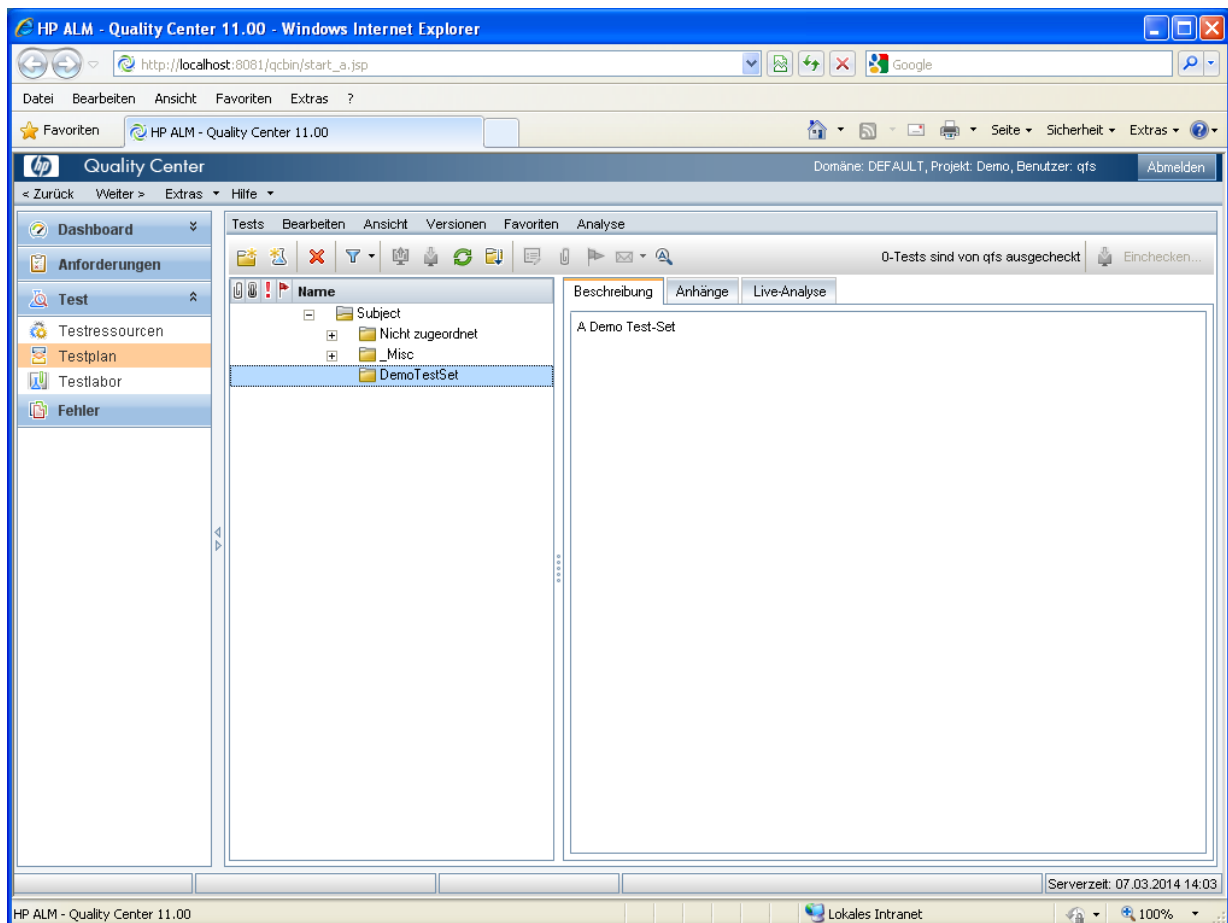


Figure 19.3: In Test plan create new Test-set

3. In this test-set create a new test with type VAPI-XP-TEST.

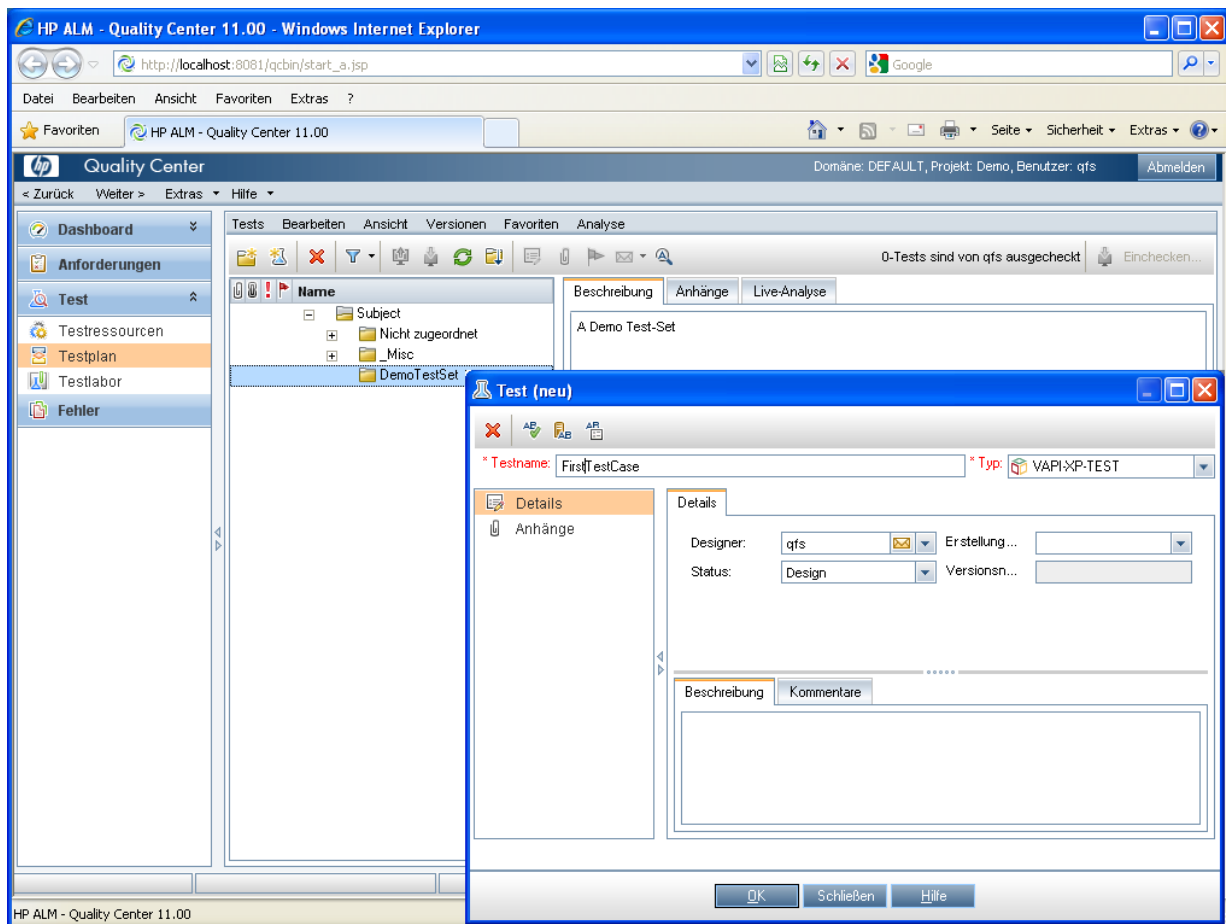


Figure 19.4: Create new test of type VAPI-XP-TEST

Note

4. On the HP VAPI-XP Wizard window just press finish without any modifications. (That means you have VBScript as script language and COM/DCOM Server Test as test type).

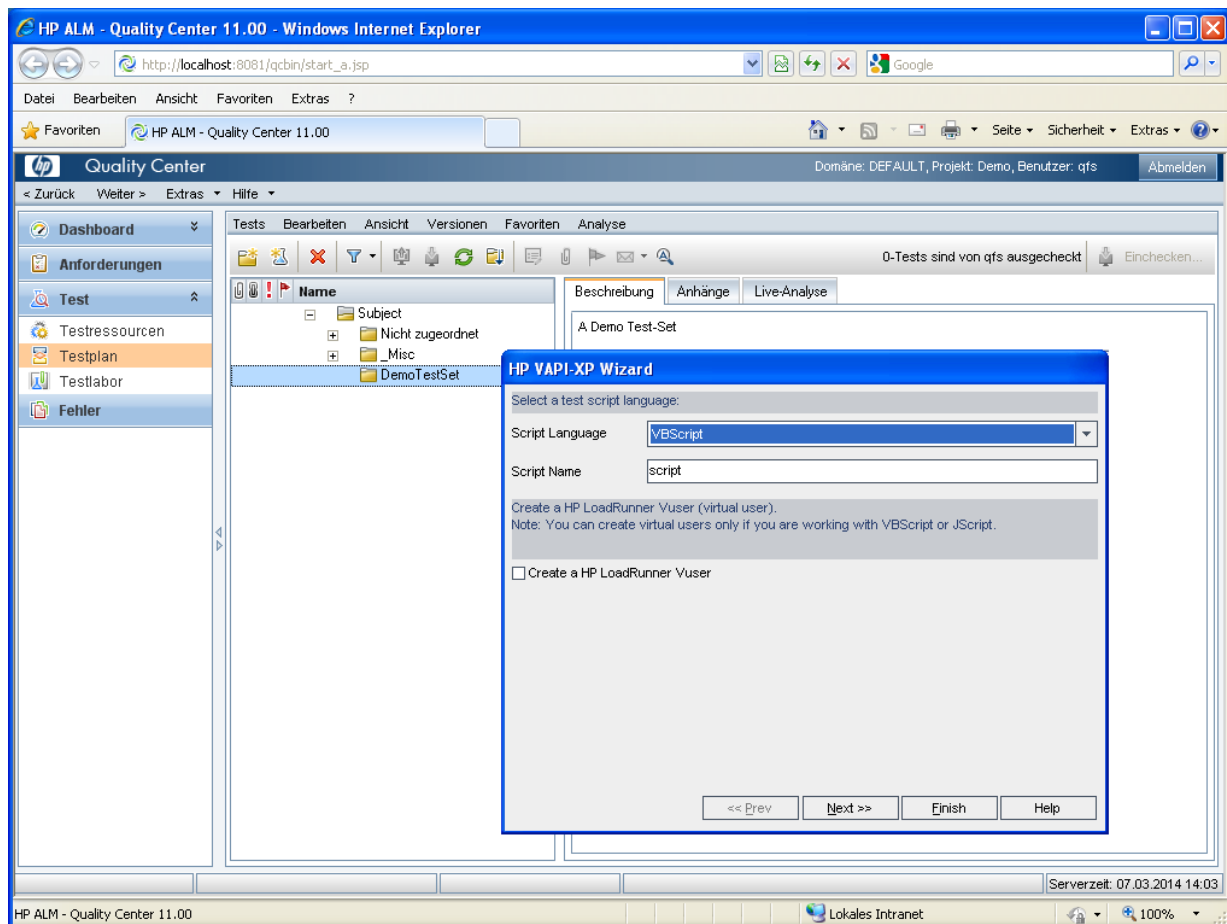


Figure 19.5: HP VAPI-XP Wizard

5. You will then get a new test as shown below.

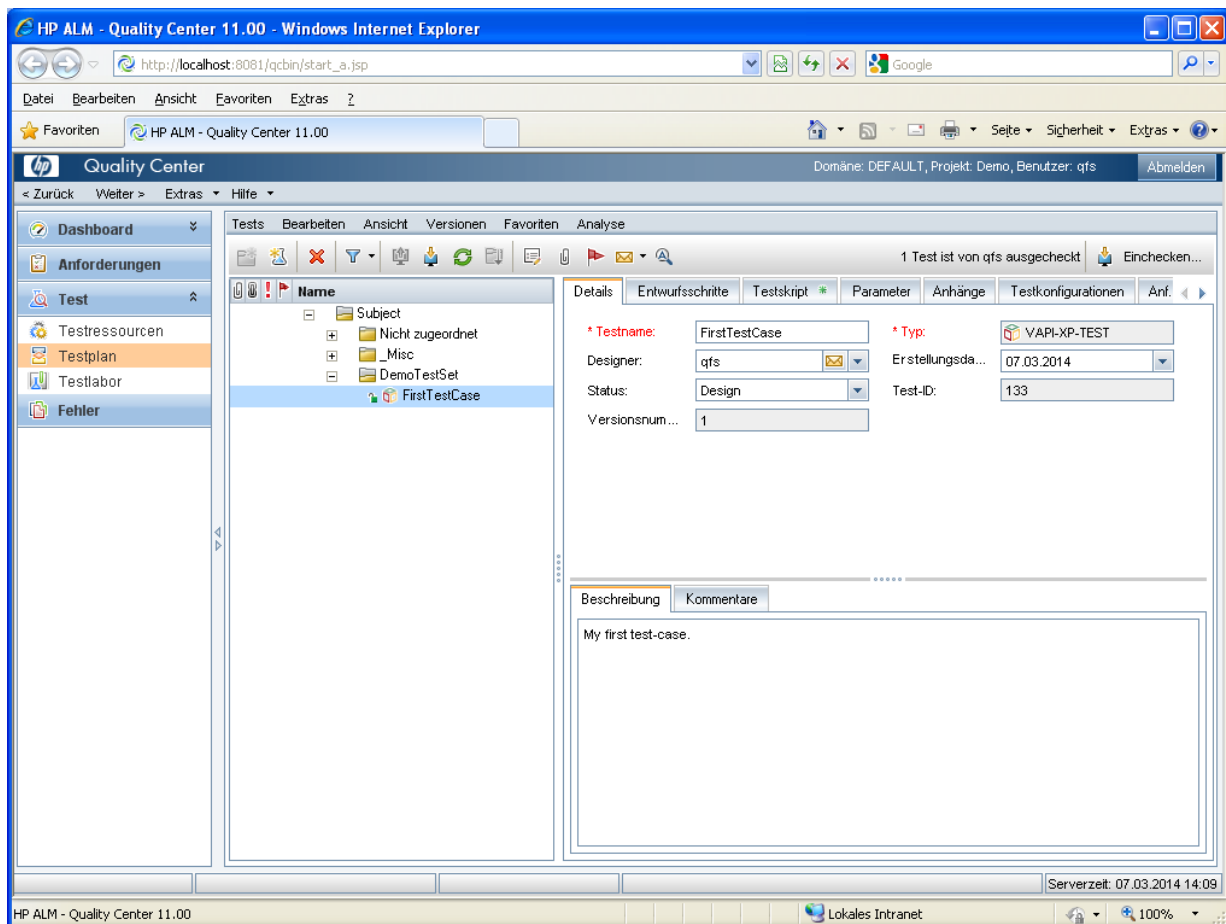


Figure 19.6: Test details

6. Change to the 'Test script' tab of the test and copy the content of the template file `qftest-4.0.8/ext/qualitycenter/qcVapiXPTemplate.txt` into the Script Viewer's text area.

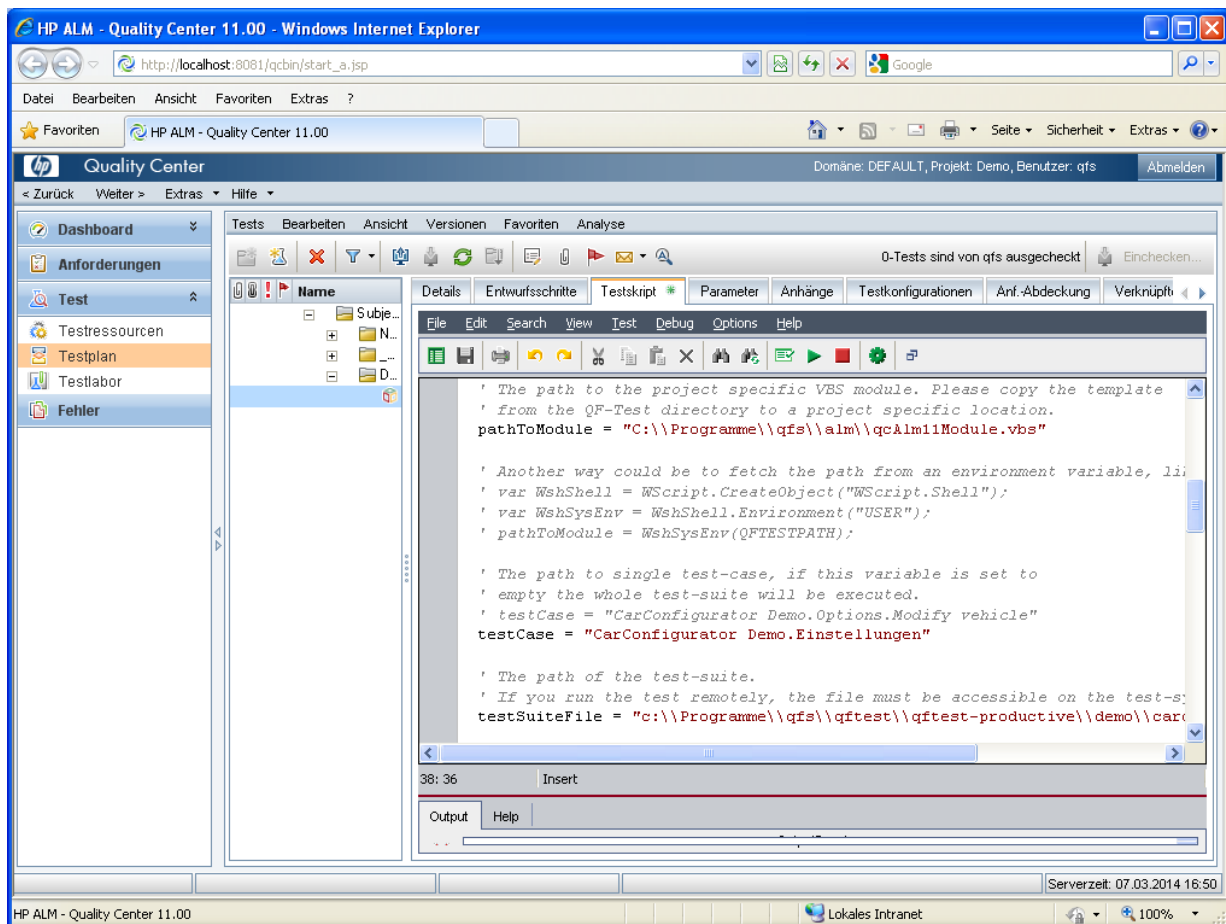


Figure 19.7: Copy template content to script text area

7. Within the script please do following adaptations:

- Change the `pathToModule` variable to the location you have copied the worker script `qcTemplate.vbs` to.
- Change the `testSuiteFile` variable to your desired test-suite file.
- If you want to execute one specific test, you can also change the `testCase` variable to the desired test-case name.

Please read the comments in the script carefully, because you can also use test-case specific settings optionally.

Steps to be performed to run the sample test-case

1. Change to the "Test lab" section in Quality Center.

2. You might want to create a new sample test-set.

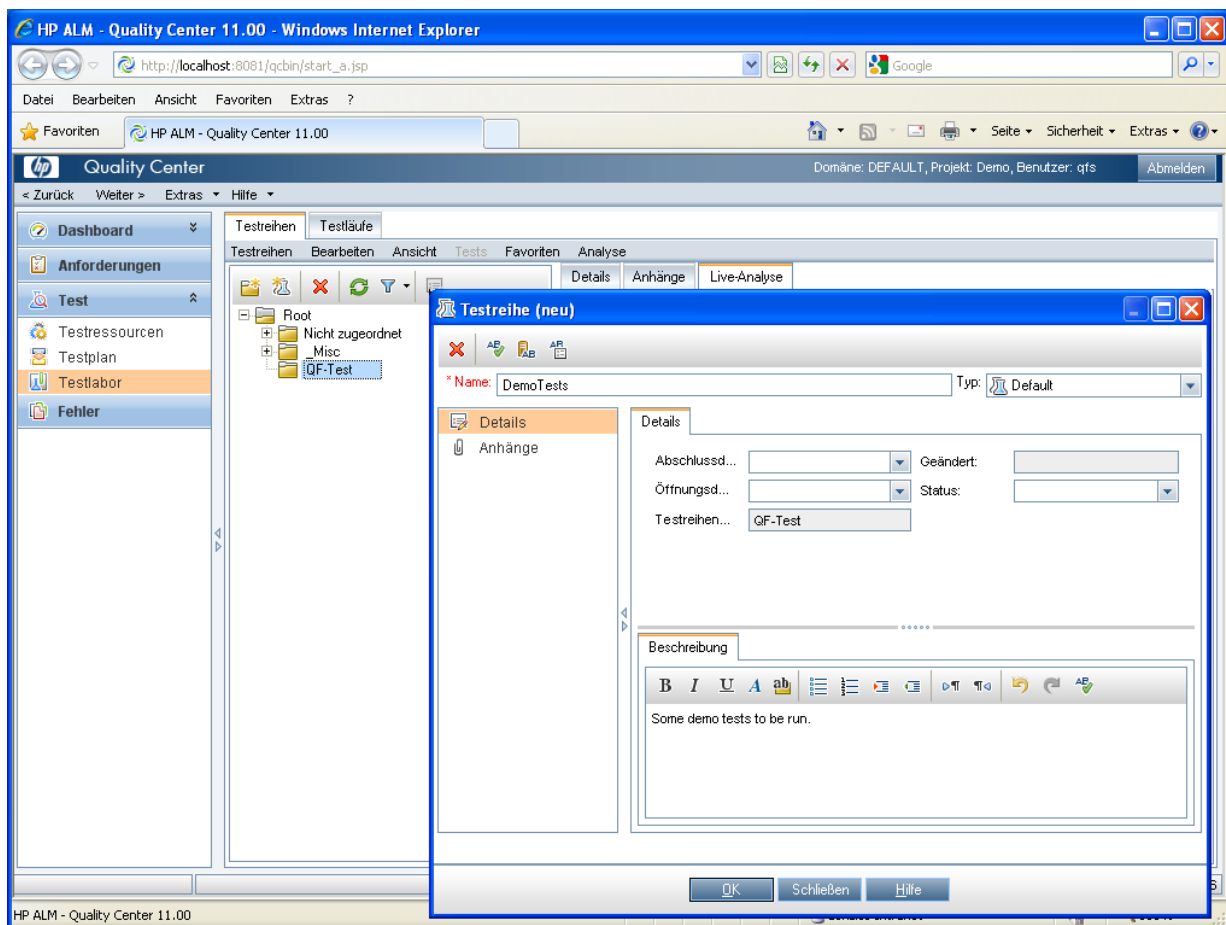


Figure 19.8: New test-set in Test lab section

3. Add the test-case to the new test-sets' execution grid by choosing it from the test plan structure.

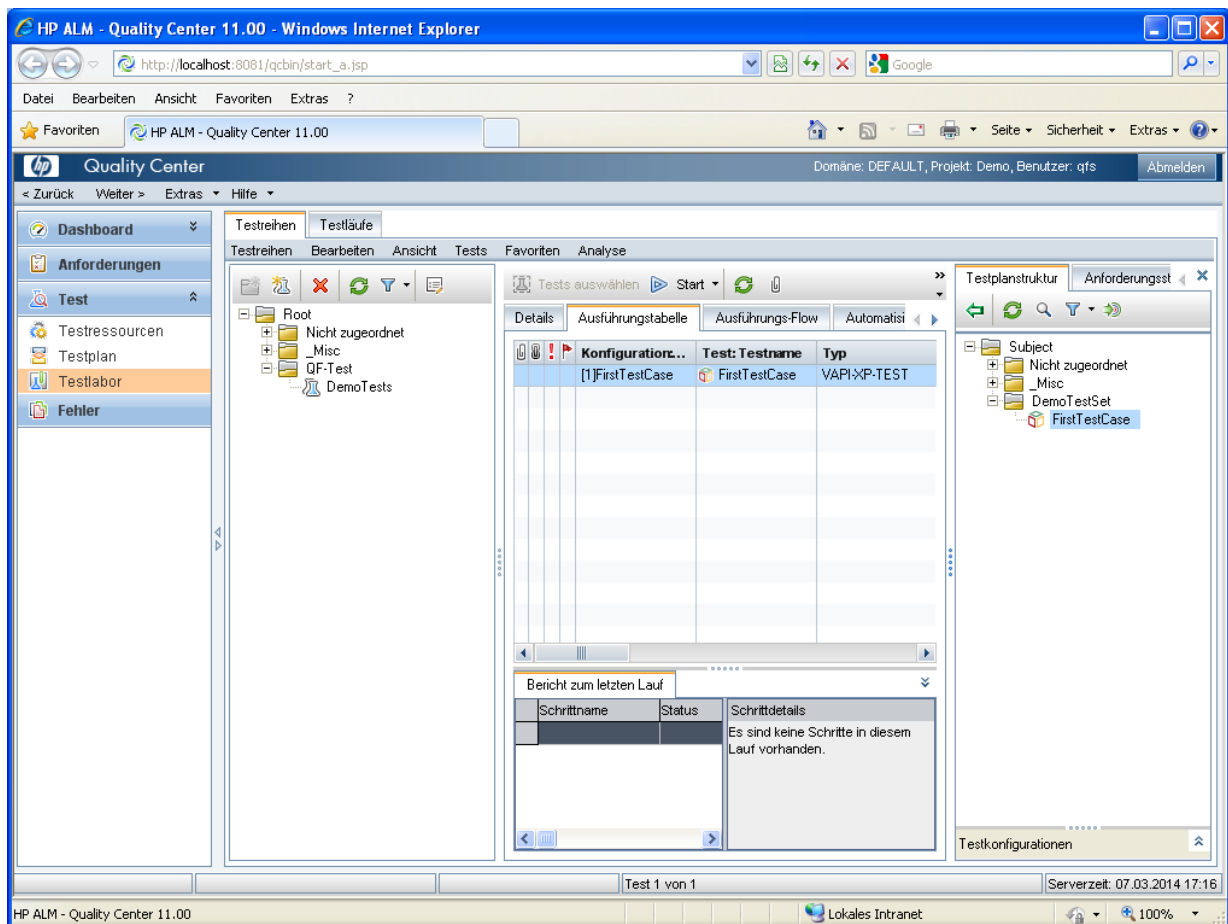


Figure 19.9: Add test to execution grid

4. Now you can launch the test-case. Ensure the "Run all tests locally" checkbox is activated unless you really have a remote system with a QF-Test environment already set up.

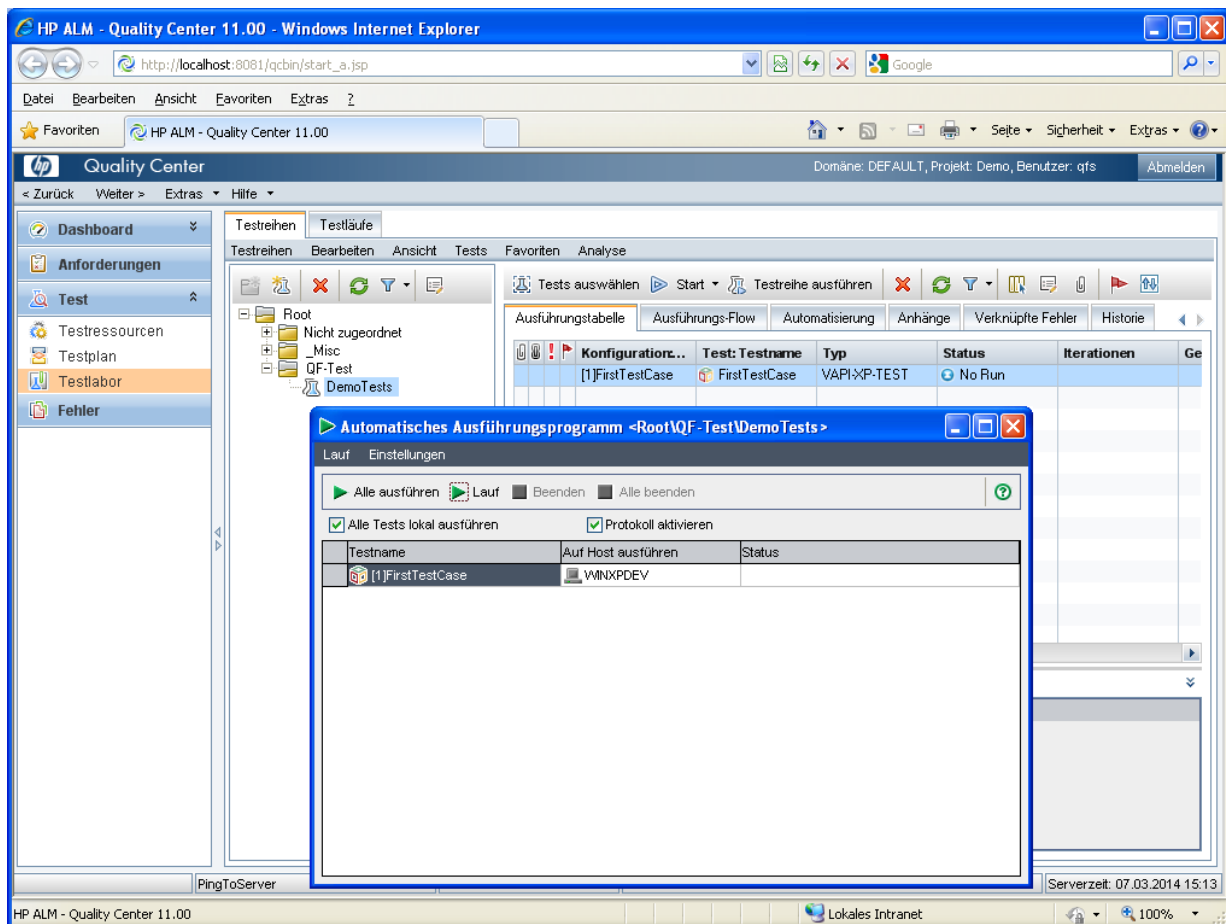


Figure 19.10: Run the test

5. Now Quality Center should start the test run - possibly on your machine, then you should see the SUT coming up after some time, actions being performed and closed at the end. When the run has finished, the result is noted down with the test: Passed or Failed.

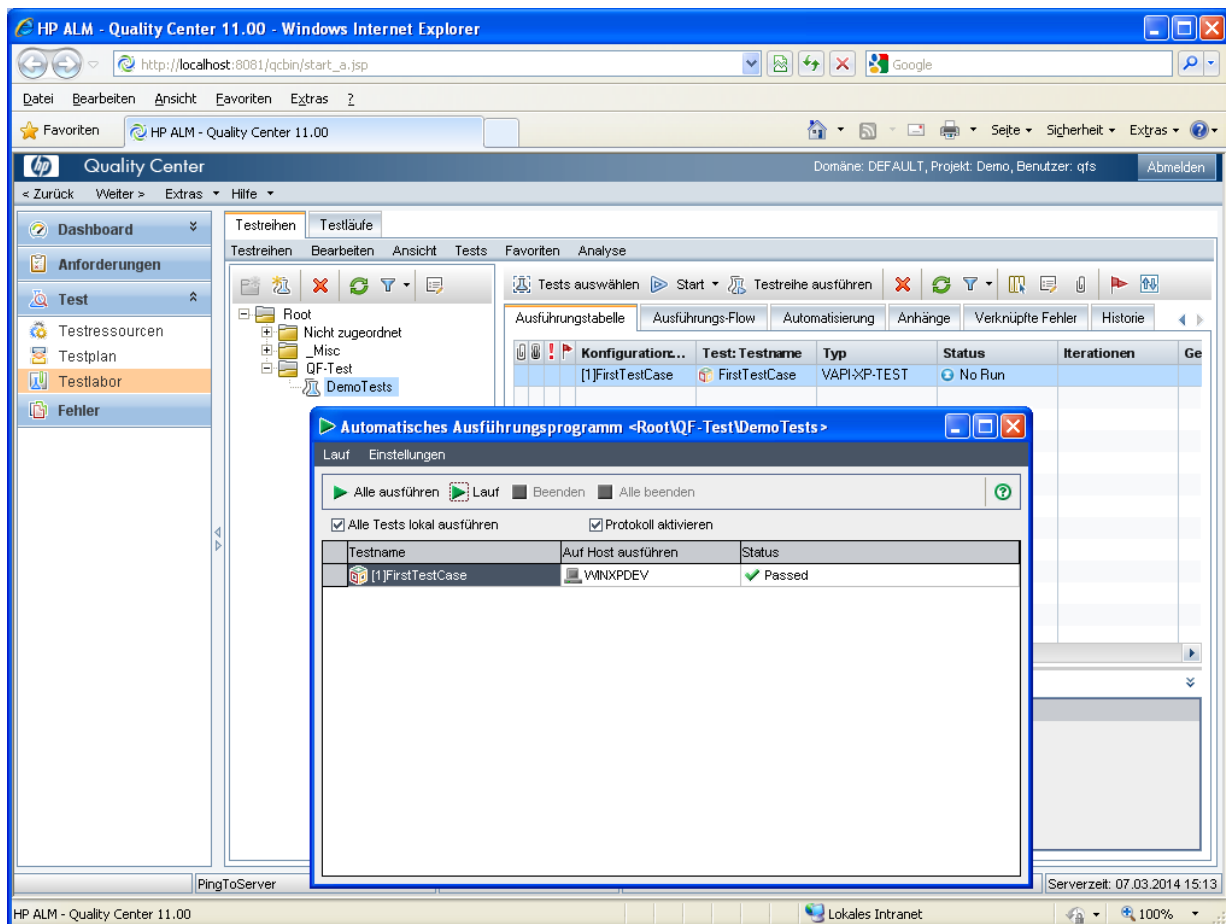


Figure 19.11: Test result

- After the test has terminated, in addition to the result the run-log of the test will be uploaded as attachment to the test instance. To view the run-log, please double-click to the test in the execution grid, then change to "Runs" and again double-click at the paper-clip attachments symbol for the respective test-run.

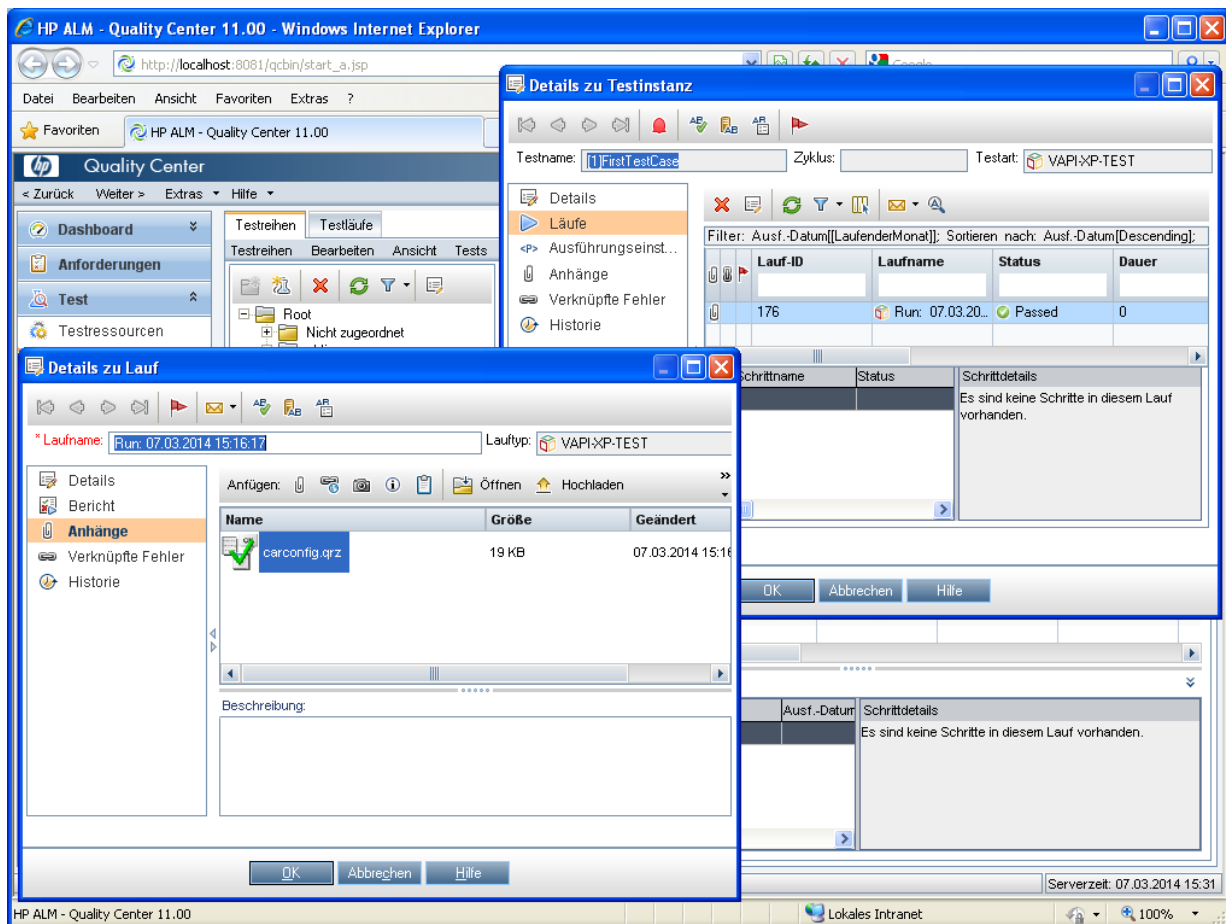


Figure 19.12: Uploaded run-log

19.1.3 Troubleshooting

First of all we need to state that we are not QualityCenter experts. Therefore there might be better and advance options for troubleshooting. Hence we want to at least provide some hints we used so far.

Unfortunately the process output during the test execution in QualityCenter is only visible for a fraction of time, not allowing a direct analysis. Therefore we need to find a work around.

The text editor of the VAPI-XP-TEST node in the "Test plan" area allows to directly execute the script. Then in the output area below the output gets visible permanently showing possibly something helpful.

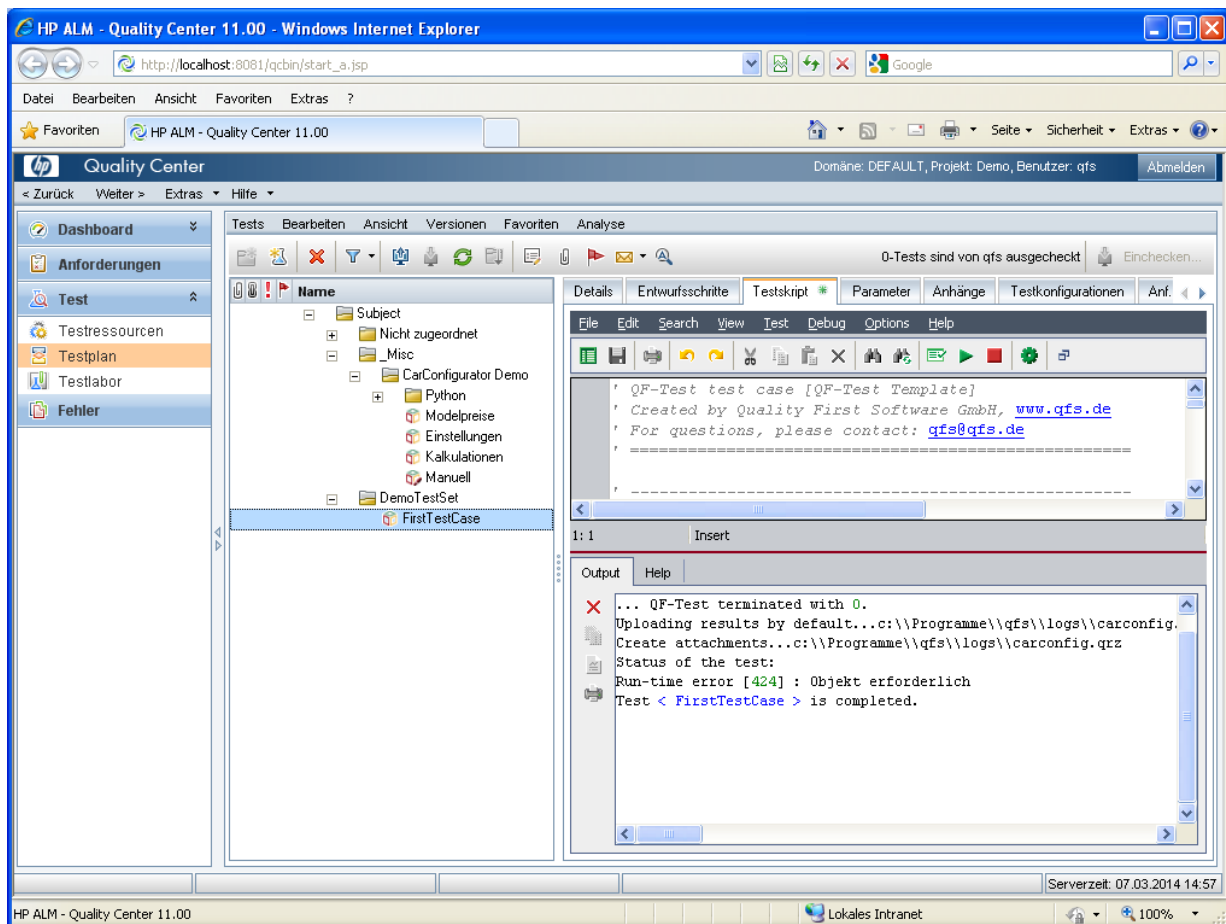


Figure 19.13: Script debug run

But the direct execution from the script node needs to be handled with care. Of course it is not considered as a real test run, so no run-log can be uploaded which results in a respective Run-time error "Object required" in the output. Don't get confused by that!

For more debugging, additional statements like `TDOutput.Print "Some text"` can be added to both the test script and the worker script. By this you can see how far the script runs until a possible error occurs.

The text script editor has a "Syntax check" button which is helpful for validation after every change.

19.2 Imbus TestBench

19.2.1 Introduction

The current integration of QF-Test and the TestBench consists of two parts:

- Creating a QF-Test template file using the TestBench interactions.
- Importing QF-Test results into TestBench.

You can find all required libraries and test-suites in the folder `qftest-4.0.8/ext/testbench/Version_1.1_TestBench_2.3`.

The following section provides a short overview about the integration concept.

19.2.2 Creating QF-Test template from interactions

After planning your tests and designing the interactions in the TestBench, you can create a template QF-Test file using the QF-Test export plug-in for interactions. Imbus will provide all required information, how to install this plugin.

After exporting the interactions you will find all interactions as procedures and their structure as packages in the QF-Test file. Now you can start recording the respective steps in QF-Test and fill the empty procedures.

The completed file has to be saved in your project-specific test-suite folder, because this file should be used as input file for the test execution later. We recommend to use a project-specific location, perhaps a shared network drive or the version management system.

19.2.3 Importing test-execution results

For running tests you need specific test-cases and procedures, which can be found in the provided test-suites in the folder `qftest-4.0.8/ext/testbench/Version_1.1_TestBench_2.3/suite`.

You can also find a demo implementation for the JCarConfigurator in the folder `qftest-4.0.8/ext/testbench/Version_1.1_TestBench_2.3/suite/demo`. You have to take care that you need to include the test-suite `TestBench_Automation.qft` to your test-suite. If you have created procedures via the iTEP exporting as described in chapter [section 19.2.2^{\(177\)}](#) you will need to include that test-suite as well.

The next step is to adapt the configuration for the output files. This can be achieved by modifying the files `testaut.properties` and `user.properties`.

Now you are ready to call the test-case Standalone test executor from `TestBench_Automation.qft`.

When the test-run has been completed, you can import all those results using the iTEP or iTORX import plug-in into the TestBench. The single QF-Test run-logs will then be attached to the test-instances.

19.3 QMetry

19.3.1 Introduction

The current integration between QF-Test and QMetry relies on planning the tests and its steps within QMetry and forwarding the actual test-execution to QF-Test. Once the test-run terminates the QF-Test run-log and its HTML report will be automatically uploaded to QMetry to the respective result area as well as the state of the test-case in QMetry will be set to the according result.

You need to prepare your test-system in order to run QF-Test tests. Please perform the following steps:

- In the 'Admin' area of the QMetry Testmanagement view install a test-execution agent at the 'Agent' view.
- Download the required agent and configuration files to install the QMetry execution agent on your test-system.
- Install the respective QF-Test QMetry Launcher at your test-agent.
- Install and set-up a platform at the 'Platform' view, which is also located in the 'Admin' area of QMetry.
- Configure `QMetryAgent.properties` correctly to use the required environment variables of QMetry's QF-Test wrapper.
- Configure `QMetryConfig.properties` correctly to show to the right QF-Test executable.
- Configure additional parameters for the QF-Test call in `QMetryConfig.properties`, see next section for details.
- Launch the QMetry agent. Please do not launch the agent as Windows-Service to avoid running GUI-Tests within the service-session. If you launch the agent as

service you should run the QF-Test tests via the QF-Test daemon, which shouldn't run in a service session then.

After setting up the agent and launcher, you need to plan the test-execution. QMetry supports several ways of integrating QF-Test test-cases. You can find all supported integrations in QMetry's integration guide document. Please perform following steps for a simple integration:

- In the Testmanagement view change to 'Test Cases' and plan the test-cases there.
- At the individual test-case you have to set the value 'Test Script Name' to the path of the required QF-Test test-suite holding the actual implementation of the test-case.
- The name of the test-case must be exactly the same as the specified value for the QF-Test ID attribute in QF-Test.
- Add the test-case to an executable test-suite in the 'Test Suites' view.

Now you are ready to run the test-cases:

- Open the 'Test Suites' view and select the required test-suite for execution.
- Select the 'Execute TestSuite' tab.
- Run or schedule a test-run via assigning an agent to the 'Automation' column.
- The next time when the local QMetry agent is polling the QMetry server it will get the necessary information to run the test-case.
- Once the test-run terminates you will find the run-log of QF-Test and its HTML report attached to the 'Execution History' of the executed test-suite. The state of the test-case will also be updated accordingly.

The following figure shows the 'Execution History' tab in 'Test Suites' holding the run-log:

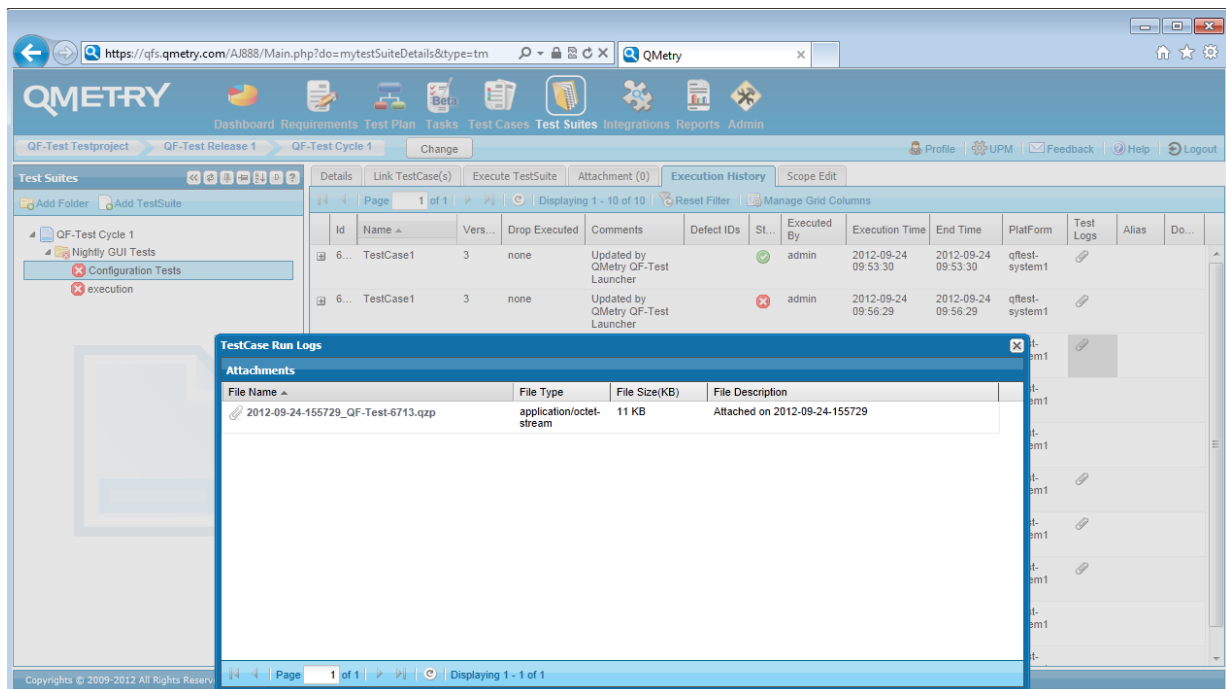


Figure 19.14: QF-Test run-log in QMetry

You will find a more detailed description of how to setup QMetry in the manual of QMetry and in QMetry's integration guide document.

19.3.2 Sample Configuration

It's recommended to set following values in the configuration file `QMetryConfig.properties`:

- Set the value of `generic.adapter.success.code` to `0,1`.
- Set `qftest.additional.arguments` to `-test ${QMTestCaseName}` in case of local test execution.
- In case of using QF-Test's daemon set `qftest.additional.arguments` to `-test ${QMTestCaseName} -calldaemon -daemonhost <testsystem> -daemonport <daemonport> .`

As already mentioned in the previous section, you need to use the same name for the test-case within QMetry and for the value of the QF-Test ID attribute within QF-Test.

Further ways for integrating QMetry and QF-Test can be found in the integration guide document provided by QMetry.

19.4 Klaros

19.4.1 Introduction

Klaros is a test management tool developed and supported by verit Informationssysteme GmbH, Kaiserslautern, Germany.

Klaros is available in two kinds of editions, a free community edition and an enterprise edition with an extended set of functionality, individual configuration options and full customer support.

The current integration of QF-Test with Klaros comprises:

- Import of QF-Test results into Klaros.

19.4.2 Importing QF-Test results into Klaros

After creating the XML report file as described in [chapter 15^{\(129\)}](#), you can upload the results to Klaros. An example for a QF-Test import URL may look like this, where the result file is contained in the HTTP request body.

```
http://localhost:18080/klaros-web/seam/resource/rest/importer?
    config=P00001&env=ENV00001&sut=SUT00001&type=qftest&
    time=05.02.2013_12:00&username=me&password=secret
```

Example 19.1: Importing test results into Klaros

The `curl` command line tool can be used on Linux or Windows/Cygwin to trigger an import in a single command line.

```
curl -v -H "Content-Type: text/xml" -T "my_qftest_report.xml" \
"http://localhost:18080/klaros-web/seam/resource/rest/importer\
?config=P00001&env=ENV00001&sut=SUT00001&type=qftest\
&time=05.02.2013_12:00&user=me&password=secret"
```

Example 19.2: Using `curl` command to import test results into Klaros

Further information can be found within the Klaros online manual at <http://www.klaros-testmanagement.com/files/doc/html/User-Manual.Import-Export.TestResultImport.html>.

19.5 SQS-TEST/Professional

19.5.1 Introduction

SQS is one of the leading international specialists for software quality with about 3000 employees world-wide.

With SQS-TEST/Professional Suite SQS offers an extensive test management framework, covering everything from requirement management through test-case development and execution to integration of test-results into a comprehensive report system.

QF-Test seamlessly interoperates with this system without special requirements for the integration.

19.5.2 Integration

QF-Test execution is triggered by SQS-TEST/Professional and results are led back, processed and administrated again in SQS-TEST/Professional.

The integration with QF-Test is performed via a public API interface of SQS-TEST/Professional. Only the SQS-pilot has to be installed.

That given, scripts can be transferred to the target system, either manually or directly addressed within the SQS tool.

Please ask SQS for more details on their SQS-TEST/Professional platform. Also for technical questions on the integration with QF-Test contact the SQS support via support@sqz.com or by telephone +49 2203 9154 -11 (SQS test suite support).

19.6 TestLink

19.6.1 Introduction

The current integration of QF-Test with the open-source tool TestLink consists of two parts:

- Generating template test-suites for QF-Test from the planned test-cases of TestLink.
- Importing QF-Test results into TestLink.

3.5.1+

If you use TestLink 1.9.4 or newer you can use the TestLink API for interacting with

TestLink. The TestLink API requires a valid development key. Therefore open TestLink and go to 'My Settings'. In the settings you can generate a development key by pressing 'Generate key' under the 'API interface' section.

For TestLink 1.9.3 or older versions the integration mechanism accesses the database of TestLink directly. This approach requires a JDBC database driver to use the provided scripts. You can download those drivers from the web page of their providers.

Exporting the planned test-cases including its test-steps from TestLink to QF-Test supports the test-creator to implement the test-cases exactly as planned.

Importing the test results into TestLink provides a better overview over all executed manual and automated tests-cases in one tool.

Note

Test results can also be uploaded to TestLink without exporting them before. Therefore you have to take care, that the ID of the test-case from TestLink is part of the test-case's name in QF-Test. The name has to be called like this: `<TestLink-ID>: Name of the test-case.`

19.6.2 Generating template test-suites for QF-Test from test-cases

QF-Test offers the capability to generate template test-suites following the same structure as the planned tests in TestLink to guarantee a synchronized structure of automated tests and test planning.

In the QF-Test file you can find one Test-case node per test-case and one Test-set node per suite from TestLink. If you have specified the fields "Steps" and "Expected Results" of a test-case, the generating-script will also create an empty Test-step for each test-step in the according test-case. The expected result will be shown in the Comment attribute of the Test-step node.

Now the template test-suite has to be filled by the test automation engineer with the according steps by adding QF-Test steps to the generated Test-step nodes.

3.5.1+

In case you use TestLink 1.9.4 or newer you need to perform following steps:

1. Take care that test automation is enabled in TestLink. Therefore set the respective `enable_test_automation` key to `ENABLED` in the configuration file `config.inc.php`.
2. Copy the folder `qftest-4.0.8/ext/testlink/api` to a project-specific location.
3. Open the launcher script you want to use with a text editor. The launcher scripts are `exportTests.bat` for Windows and `exportTests.sh` for Unix.
4. Adapt the paths of the variables `JAVA`, `QFTDIR` and `TESTLINKINTEGRATOR`.

5. Open the file `TestLinkUserSpecifics.py` with a text editor.
6. Adjust the variables `serverurl` and `devkey`.
7. If you want to export custom fields from TestLink, also adjust the variable `custom_fields`.
8. Run the adapted export script, like shown below.

```
exportTests.bat --testproject projectname
                --targetsuite /path/to/testsuite.qft
```

Example 19.3: Sample call of exporting test-cases from 1.9.4

If you use TestLink 1.9.3 or older, please perform those steps:

1. Copy the folder `qftest-4.0.8/ext/testlink/export` to a project-specific location.
2. Open the launcher script you want to use with a text editor. The launcher scripts are `exportTestLinkToQFT.bat` for Windows and `exportTestLinkToQFT.sh` for Unix.
3. Adapt the paths of the variables `JAVA`, `QFTDIR` and `TESTLINKINTEGRATOR`.
4. Open the file `TestLinkDBIntegrator.py` with a text editor.
5. Adjust the variables `dbdriver`, `connctionstr`, `dbuser` and `dbpass` according to your database connection.
6. If you want to export custom fields from TestLink, also adjust the variable `custom_fields`.
7. Run the adapted export script, like shown below.

```
exportTestLinkToQFT.bat --testproject projectname
                      --targetsuite /path/to/testsuite.qft
```

Example 19.4: Sample call of exporting test-cases till 1.9.3

19.6.3 Execution of test-cases

Executing the QF-Test tests can be performed as usual. But you should create a XML-report at the end of the test-run, because the import mechanism is using this report. Therefore you have to use the '-report.xml' parameter during test execution. If you create the reports via the GUI, you have to check the checkbox 'Create XML report'.

Note In case you did not export test-cases from TestLink the ID of the test-case from TestLink has to be part of the test-case's name in QF-Test. The name has to be called like this:
<TestLink-ID>: Name of the test-case.

```
qftest -batch -report.xml reportFolder testsuite.qft
```

Example 19.5: Sample execution to create a XML report

19.6.4 Importing QF-Test results into TestLink

After creating the XML report file, you can upload the results to TestLink.

Per default the import mechanism creates a new build for every test-run. The build number of TestLink will be created by the run-ID of the QF-Test report. You can change the run-ID, by setting the parameter '-runid' when launching the tests with QF-Test. But you can also set the '-build' parameter during import to specify a custom build name.

3.5.1+ In case you use TestLink 1.9.4 or newer you need to perform following steps:

1. Take care that test automation is enabled in TestLink. Therefore set the respective `enable_test_automation` key to `ENABLED` in the configuration file `config.inc.php`.
2. Copy the folder `qftest-4.0.8/ext/testlink/api` to a project-specific location. (If you have copied them already for exporting you can use the same files.)
3. Open the launcher script you want to use with a text editor. The launcher scripts are `importResults.bat` for Windows and `importResults.sh` for Unix.
4. Adapt the paths of the variables `JAVA`, `QFTDIR` and `TESTLINKINTEGRATOR`.
5. Open the file `TestLinkUserSpecifics.py` with a text editor.
6. Adjust the variables `serverurl` and `devkey`. (If you have adapted them already for exporting you can use the same values.)
7. Run the adapted import script, like shown below.

```
importResults.bat --testproject projectname
                  --resultfile qftestReport.xml --testplan testplanname
                  --platform system1
```

Example 19.6: Importing test results into TestLink from 1.9.4

If you want to overwrite the build name you can use the '-build' parameter.>

```
importResults.bat --testproject projectname
                  --resultfile qftestReport.xml --testplan testplanname
                  --platform system1 --build myBuild
```

Example 19.7: Importing test results into TestLink from 1.9.4 with custom build

If you use TestLink 1.9.3 or an older version, please perform following steps:

1. Copy the folder `qftest-4.0.8/ext/testlink/import` to a project-specific location.
2. Open the launcher script you want to use with a text editor. The launcher scripts are `importToTestLink.bat` for Windows and `importToTestLink.sh` for Unix.
3. Adapt the paths of the variables `JAVA`, `QFTDIR` and `TESTLINKINTEGRATOR`.
4. Open the file `ReportParser.py` with a text editor.
5. Adjust the variables `dbdriver`, `connctionstr`, `dbuser` and `dbpass` according to your database connection.
6. If you want to export custom fields from TestLink, also adjust the variable `custom_fields`.
7. Run the adapted import script, like shown below.

```
importToTestLink.bat --testproject projectname
                    --resultfile qftestReport.xml --testplan testplanname
                    --tester tester
```

Example 19.8: Importing test results into TestLink till 1.9.3

Chapter 20

Integration with Build Tools

Automated GUI testing is just one step in the build process of a software development project. The basic need for coordinating all those steps of compiling, testing, documenting and deploying of software has lead to a variety of build tools (e.g. `make`, `ant`, `maven`) and so called continuous integration tools (e.g. `Jenkins/Hudson`, `Cruise Control`, `Continuum`).

In general, by use of QF-Test's command line interface as documented in chapter 16⁽¹³⁸⁾ and chapter 32⁽⁵⁶⁷⁾ a straight forward integration with those tools should be possible.

The following sections contain examples for integrations with a some of the tools mentioned above.

20.1 Ant

People who are using Apache Ant (<http://ant.apache.org>) as build system may easily integrate QF-Test in their build file:

```

<project name="QF-Test" default="runtest">
  <property name="qftest"
    location="c:\Program Files\qfs\qftest\qftest-4.0.8\bin\qftest.exe" />
  <property name="logdir" value="c:\mylogs" />
  <target name="runtest" description="Run a test in batchmode">
    <echo message="Running ${suite} ..." />
    <exec executable="${qftest}" failonerror="false"
      resultproperty="returncode">
      <arg value="-batch" />
      <arg value="-compact" />
      <arg value="-runlog" />
      <arg value="${logdir}\+b" />
      <arg value="${suite}" />
    </exec>
    <condition property="result"
      value="Test terminated successfully.">
      <equals arg1="${returncode}" arg2="0" />
    </condition>
    <condition property="result"
      value="Test terminated with warnings.">
      <equals arg1="${returncode}" arg2="1" />
    </condition>
    <condition property="result"
      value="Test terminated with errors.">
      <equals arg1="${returncode}" arg2="2" />
    </condition>
    <condition property="result"
      value="Test terminated with exceptions.">
      <equals arg1="${returncode}" arg2="3" />
    </condition>
    <echo message="${result}" />
  </target>
</project>

```

Example 20.1: Ant build file build.xml to execute a test-suite

The above example assumes the test-suite to be defined as property when running ant: `ant -Dsuite="c:\Program Files\qfs\qftest\qftest-4.0.8\demo\carconfig\carconfig_en.qft"`.

20.2 Maven

People who are using Apache Maven (<http://maven.apache.org>) as build system may easily integrate QF-Test in their build. This can be achieved by using the antrun plugin of Maven. A demo pom.xml file, where QF-Tests tests are executed in the test phase

could look like this:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <artifactId>testant</artifactId>
  <packaging>jar</packaging>
  <name>testant</name>
  <groupId>de.qfs</groupId>
  <version>1</version>
  <properties>
    <qf.exe>"C:\Program Files\qfs\qftest\qftest-4.0.8\bin\qftest.exe"</qf.exe>
    <qf.reportfolder>qftest</qf.reportfolder>
    <qf.log>logfile.qrz</qf.log>
    <qf.suite>"c:\path\to\testsuite.qft"</qf.suite>
  </properties>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-antrun-plugin</artifactId>
        <executions>
          <execution>
            <phase>test</phase>
            <configuration>
              <tasks>
                <exec executable="${qf.exe}">
                  <arg value="-batch"/>
                  <arg value="-report"/>
                  <arg value="${qf.reportfolder}"/>
                  <arg value="-runlog"/>
                  <arg value="${qf.log}"/>
                  <arg value="${qf.suite}"/>
                </exec>
              </tasks>
            </configuration>
            <goals>
              <goal>run</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Example 20.2: Maven build file pom.xml to execute a test-suite

In your project it might become required to run the tests during another build phase, than the configured `test` phase in the example. In this case you have to configure the plugin accordingly, like described in the Maven documentation.

20.3 Jenkins/Hudson

Jenkins/Hudson (<http://jenkins-ci.org> or <http://hudson-ci.org>) are continuous integration build tools. They are meant to control and monitor the build process within a software project. One important step in this build process is automated testing.

There are number of benefits to be gained when integrating QF-Test with Jenkins/Hudson:

- In case Jenkins/Hudson is already used for the continuous integration process, integration of automated GUI tests can be easily achieved.
- Easy-to-use administration of scheduled test runs and notification of results via email or RSS.
- Jenkins/Hudson's web based GUI provides good overview and control of test results.
- By use of the HTML Publisher Plugin it is possible to embed QF-Test's HTML reports directly into the Jenkins/Hudson GUI.
- Results generated during the test run such as run-logs and reports can be archived automatically. Therefore maintaining an own directory structure is not needed anymore.

As Jenkins and Hudson share a common history, following sections will use Jenkins as representative.

20.3.1 Install and start Jenkins

To install Jenkins download the `war` Archive (which can be found [here](#)) and start it via `java -jar jenkins.war`.

As soon as Jenkins is started its web interface can be accessed via `http://localhost:8080`. It should look like the following:

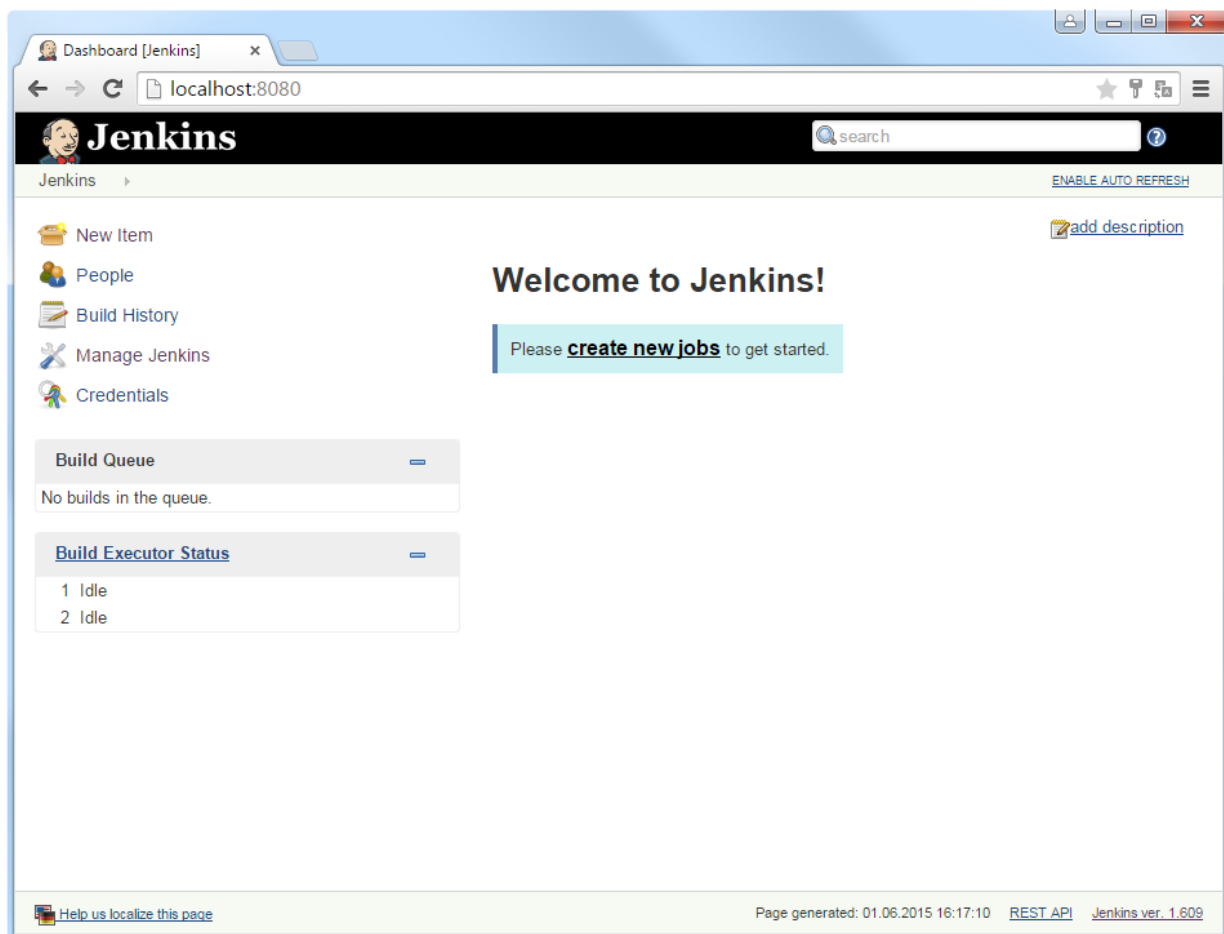


Figure 20.1: Jenkins after start-up.

20.3.2 Requirements for GUI tests

GUI testing requires an unlocked, active desktop. That is the only way to ensure that the SUT behaves the same as if a normal user interacts with it.

Jenkins allows execution of tasks on remote machines. This is of course also relevant for GUI testing. Due to its nature GUI tests are typically not intended to run on the central build server. In addition, tests might need to be executed for different environments, operating systems and SUT versions.

On a remote machine, a Jenkins slave agent needs to be launched in order to connect to the Jenkins server and wait for jobs to be processed. As described in the Jenkins documentation, there are several options to launch this agent, but for the GUI tests to properly work the only possible launch method is to use Java Web Start.

For GUI tests it is vital to have an active, unlocked user session. Therefore it is not

possible to start the agent via a windows service but a real (test) user must be logged in (e.g. via auto login) using Windows Autostart to launch the Jenkins agent. Furthermore screen locking needs to be disabled.

Note Please see also FAQ 14 for more technical background details.

20.3.3 Install QF-Test Plugin

The QF-Test Plugin enables QF-Test to interact with Jenkins. To install the plugin open the Jenkins dashboard and navigate via "Manage Jenkins" to "Manage Plugins". Select the QF-Test Plugin from the "Available" tab. When installing the QF-Test Plugin the JUnit and HTML-Publisher Plugin will also be downloaded automatically, in case they were not already installed. Finally restart Jenkins to complete the installation. Now the QF-Test Plugin will show up under the Installed tab, as shown in Figure 20.2.

Note For an in-depth look at the plugin and all of its available features, please have a look at the Youtube Video about the installation and configuration of the plugin.

Note Jenkins will automatically use the latest installed version of QF-Test. In case you want to use a different version, you can provide its path under the QF-Test section in the Jenkins configuration (Manage Jenkins -> Configure System).

Updates Available Installed Advanced						
Enabled	Name ↓	Version	Previously installed version	Pinned	Uninstall	
☒	Ant Plugin This plugin adds Apache Ant support to Jenkins.	1.2		Unpin		
☒	Credentials Plugin This plugin allows you to store credentials in Jenkins.	1.18				
☒	CVS Plugin Integrates Jenkins with CVS version control system using a modified version of the Netbeans cvsclient.	2.11				
☒	External Monitor Job Type Plugin Adds the ability to monitor the result of externally executed jobs.	1.4				
☒	HTML Publisher Plugin This plugin publishes HTML reports.	1.3		Unpin	Uninstall	
☒	Javadoc Plugin This plugin adds Javadoc support to Jenkins.	1.0		Unpin		
☒	JUnit Plugin Allows JUnit-format test results to be published.	1.5		Unpin		
☒	QF-Test Plugin QF-Test is a cross-platform software tool for the GUI test automation specialized on Java and Web applications.	1.0			Uninstall	

Figure 20.2: Install QF-Test Plugin.

20.3.4 Configure a job for QF-Test

To create a new job open the Jenkins dashboard and click "New Item". The standard project type is "Freestyle project". Once the job is created it has to be configured.

The first thing that needs to be done is to change the workspace of this job to the directory that contains your test-suites. It can be changed in the "Advanced Project Options" under "Advanced..." where you check "Use custom workspace" and specify the test-suites' directory.

In the "Display Name" field you have the option to provide a name to be shown instead of your job name, which is typically left empty.



Figure 20.3: Set custom workspace.

Now proceed to the "Build" section and add the "Run QF-Test" build step by choosing it from the "Add build step" drop down. In the left field you can either enter the name of a test-suite or a folder containing test-suites (e.g. your workspace directory being "C:\MySuites\" and the path of the test-suite you want to run being "C:\MySuites\qftest\mysuite.qft", simply enter `qftest/mysuite.qft` in the first field). In case you enter a directory, every test-suite in this directory will be run. In the right field you can enter additional Command line arguments⁽⁵⁷¹⁾.

Build

Run QF-Test ?

Name of test-suite(s) or parent folder Additional Command Line Arguments

testsuite.qft

Delete

Some/Directory/Containing/Testsuites

Delete

Add

Advanced...

Delete

Add build step ▼

Figure 20.4: Add build step to run QF-Test

The advanced options of the "Run QF-Test" build step are optional.

The first option allows you to provide a dedicated QF-Test version for this job.

The next option lets you change the directory for the temporary reports. The QF-Test Plugin will create a folder called "qftestJenkinsReports" in the selected workspace and save the reports and logs for the post-build actions.

Finally, there is an option to run your test-suites on a daemon. You need to enter the daemonhost and port to do so. To learn more about the daemon mode please see [section 16.2^{\(144\)}](#) of the manual.

☒ Use specific QF-Test version ?

Path for qftest executable C:\Program Files (x86)\qfs\qftest\qftest-4.0.2

☒ Change temporary directory for reports ?

Directory for temporary reports Temporary/Reports

☒ Run test-suites on daemon ?

Daemonhost localhost

Daemonport 1234

Figure 20.5: Configure build step advanced options.

Next the post build actions need to be configured to process test results and archive QF-Test run-logs and HTML reports.

To archive the run-logs add the post-build action "Archive the artifacts" and for "Files to archive" enter this path: `qftestJenkinsReports/$JOB_NAME/$BUILD_NUMBER/logs/*.q*`

To publish the HTML Reports add the post-build action "Publish HTML reports". In the HTML directory enter this path: `qftestJenkinsReports/$JOB_NAME/$BUILD_NUMBER/html/`. Also change the index page from `index.html` to `report.html`. You can keep past HTML reports if you check the respective box.

Finally add the post-build action "Publish JUnit test result report" and for "Test report XMLs" enter this path: `qftestJenkinsReports/$JOB_NAME/$BUILD_NUMBER/junit/report_junit.xml`

Note

There will be warnings that the path you entered doesn't match anything, because of the environment variables in the path. These warnings can be ignored.

The screenshot shows the Jenkins configuration page for post-build steps. It contains three sections:

- Archive the artifacts:** The "Files to archive" field contains `qftestJenkinsReports/$JOB_NAME/$BUILD_NUMBER/logs/*.q*`. A red warning icon and message state: "'qftestJenkinsReports/\$JOB_NAME/\$BUILD_NUMBER/logs/*.q*' doesn't match anything: even 'qftestJenkinsReports' doesn't exist". There is an "Advanced..." button and a "Delete" button.
- Publish HTML reports:** The "HTML directory to archive" field contains `qftestJenkinsReports/$J`, the "Index page[s]" field contains `report.html`, and the "Report title" field contains `HTML Report`. The "Keep past HTML reports" checkbox is checked. There is an "Add" button and a "Delete" button.
- Publish JUnit test result report:** The "Test report XMLs" field contains `qftestJenkinsReports/$JOB_NAME/$BUILD_NUMBER/junit/report_junit.xml`. A red warning icon and message state: "'qftestJenkinsReports/\$JOB_NAME/\$BUILD_NUMBER/junit/report_junit.xml' doesn't match anything: even 'qftestJenkinsReports' doesn't exist". Below this, a link explains the "Fileset 'includes' setting" and a checkbox for "Retain long standard output/error" is present. The "Health report amplification factor" is set to `1.0`. A "Delete" button is at the bottom right.

Figure 20.6: Configure post build steps.

Note This plugin supports concurrent builds, as long as your licence allows enough QF-Test instances.

Note You can use the additional command line argument "-dbg" to get more debug information in the console output of your build.

Chapter 21

Performing GUI-based load tests

21.1 Background and comparison with other techniques

In addition to functional and system tests, QF-Test can also be used to perform small-to-medium sized load tests, stress tests or performance tests. The idea is to test the performance of some server by running a number of clients concurrently.

There are many different ways for setting up and performing load tests, most of which do not work with real clients. Instead they directly make use of the protocol between the client and server, e.g. by sending HTTP request or performing RMI or other kinds of remote procedure calls.

There are a number of pros and cons for protocol-based or GUI-based load testing:

- Protocol-based testing uses very little resources at the client side, so it can easily scale up to the breaking point of the server without requiring too much hardware. GUI-based tests incur the full memory and performance overhead for each client, which can be quite significant, especially in case of Swing or JavaFX based rich clients. Besides QF-Test itself, a typical PC can sustain between 5 and 20 parallel SUT clients without significant loss of performance on the client side, depending on available RAM and the size and activity of the clients.
- Rich clients typically represent a complex user interface which correlates with a certain complexity of the client/server API. Creating protocol-based tests that cover most of that API can be quite an effort. On the other hand, GUI-based tests that have already been implemented for functional testing may be available for reuse. If not, it is still much easier to automate complete use-cases with QF-Test than at the protocol level.
- With GUI-based testing, actual end-to-end response times (the time span from

user action until the result shows up) are measured, while protocol-based tests measure only the times for the server call. Either can be useful, depending on the situation.

In summary, GUI-based load testing can be very useful and efficient - especially if functional tests can be reused - provided that either the number of clients that need to be simulated is not too high, or that sufficient hardware is available for the client side.

21.2 Load tests with QF-Test

Note

Running more than one GUI test in parallel on a single desktop is problematic. Hard events and Drag&Drop need to be avoided because they would cause contention for the mouse cursor. Worse, the windows of the various SUTs will can try to steal the keyboard focus from each other, leading to subtle timing issues. For Swing QF-Test filters hard events coming from the operating system and prevents them from reaching the SUT at the wrong time, most notably when popup menus are shown or when a component must not lose keyboard focus. Still, a lot of fine-tuning is required and tests should be made as robust as possible. In general, running parallel tests on separate machines or at least in different user sessions is preferable.

21.2.1 Running tests with parallel threads

To avoid the additional overhead of having to run one QF-Test instance per SUT client, the command line argument `-threads <number>`⁽⁵⁸⁶⁾ can be used to run QF-Test in batch mode with multiple threads for testing. The same test-suite will be executed in each thread. One runtime license is required per thread. If insufficient runtime licenses are available, full development licenses will be used unless the command line argument `-runtime`⁽⁵⁸²⁾ is given which forces use of runtime licenses only. Please see the documentation for the `-runtime` command line argument in [section 32.2](#)⁽⁵⁷¹⁾ for further information about runtime licenses.

All threads are independent and each thread will create and control its own set of processes and SUT clients. To that end, QF-Test appends the thread identifier to the `Client`⁽⁴²⁶⁾ attribute of any node that needs to refer to a SUT client. Thus it is impossible to access one thread's SUT client from another thread.

Two variables have special values when running with multiple threads: `${qftest:threads}` expands to the number of threads given with `-threads <number>` (default is 1 when running single-threaded) and `${qftest:thread}` is the index of the current thread (default is 0).

Running multiple clients on one desktop can sometimes lead to delays longer than usual, especially during startup of the SUTs. Therefore, timeouts should be increased generously, especially in Wait for client to connect⁽⁴³⁸⁾ and Wait for component to appear⁽⁵¹²⁾ nodes. Also, the options under Replay->Timeouts (see section 29.3.6⁽²⁹⁰⁾) should be increased to at least twice their default values. It is a good idea to use a separate set of system options for load testing by specifying the configuration file with the command line argument -systemcfg <file>⁽⁵⁸⁴⁾.

21.2.2 Synchronization

To get consistent results, it may sometimes be necessary to coordinate the tests in the parallel threads, either to make sure that all clients access the server simultaneously, or to prevent just that.

Threads can be synchronized with the help of a Server script⁽⁴¹⁶⁾ node with the following script:

```
rc.syncThreads("identifier", timeout)
```

The identifier is a name for the synchronization point and `timeout` is the maximum time in milliseconds to wait for all threads to reach the given synchronization point. If not all threads are expected to reach that point because they follow different execution paths, the additional parameter `count` can be given to identify the number of threads to expect, e.g.

```
rc.syncThreads("case1", 120000, count=3)
```

If the timeout is exceeded without the expected number of threads reaching the synchronization point, a TestException⁽⁵⁵⁸⁾ is thrown. To log an error instead of raising an exception, set the optional parameter `throw` to 0 (default value 1), e.g.:

```
rc.syncThreads("case1", 120000, count=3, throw=0)
```

21.2.3 Coordinating tests on multiple machines

If more than one machine is required to sustain the desired number of clients, tests can be run in parallel on multiple machines. Each QF-Test instance on each machine can use multiple threads as long as sufficient licenses for the total number of threads are available.

Currently QF-Test has no special support for coordinating the start of QF-Test instances on different machines. This has to be done manually, through a script or through some

third party tool. However, QF-Test does support synchronization of threads across multiple machines. The `syncThreads` method has one additional parameter called `remote` which defaults to 0 (no remote synchronization) and represents the number of QF-Test instances to synchronize. Each instance will first synchronize its running threads internally, then use network calls to synchronize with other running instances. Only when all threads in all requested machines have reached the synchronization point the test will continue.

For example, to synchronize all threads in tests on 4 machines after SUT startup, add a Server script node at the end of the main setup sequence with something like:

```
rc.syncThreads("startup", 300000, remote=4)
```

It is a good idea not to hard-code the number of remote instances to expect. Instead, use a variable defined at test-suite or system level that defaults to 0 so the tests will run fine on a single machine. When running on multiple machines, override that variable on the command line as explained in [chapter 7^{\(57\)}](#).

Chapter 22

Executing Manual Tests in QF-Test

3.0+

22.1 Introduction

QF-Test is primarily a tool for the creation and execution of automated tests. However, it is rarely possible - or economical - to automate 100% of the required tests for a project. In most projects some manual tests need to be performed as well. One of the biggest challenges in testing a project is consolidating the different results and reports of automated and manual testing to get an overview about the execution status of all tests. To facilitate reporting the results of manual test execution along with those of automated testing, QF-Test now offers the capability of tracking manual tests from within itself.

The steps to be performed during a manual test have to be defined in an Excel file which is read by a test-suite called `ManualTestRunner.qft`. This test-suite is provided along with a sample specification file in the directory `demo/manualtester` below the QF-Test installation directory. The test designer has to specify each step in that Excel file including the expected result. After stepping through the manual tests QF-Test provides the usual results - a run-log, HTML and XML reports. Additionally, a newly created Excel file with the results of the respective test-run is created. For a detailed description please see [section 22.2^{\(202\)}](#).

The dialog used for the test execution is called `ManualStepDialog` and looks like this:

1 First Step

Step description

Open a demo file using the menu.

Expected result

The demo file should be opened without any error messages!

Result

☐ FAILED

☒ PASSED

Received result/Remark

Previous step Next step Skip step Skip test Cancel

Figure 22.1: Example for a ManualStepDialog

The title of the dialog shows the name of the test-case. The detailed step description and the expected result are shown in the first two text-boxes. After performing the test the tester has to specify whether the test succeeded or not. In case the test failed the tester also has to enter the received result which is intended to show the differences between the actual and the expected result. This dialog can also be used for your own purposes, see [section 42.1^{\(764\)}](#).

22.2 Step-by-step Guide

Please perform the following steps on your system to launch a manual test from QF-Test.

- Copy the definition Excel file from `qftest-4.0.8/demo/manualtester/SampleTestDescription.xls` to your project location and rename it to a suitable name. We recommend to use the same path on all test-systems. Perhaps you can make use of a shared network drive.

- Also copy the execution test-suite from `qftest-4.0.8/demo/manualtester/ManualTestRunner.qft` to your project location. You may want to rename it as well.
- Open the Excel file and define the test-steps.
- After saving the changes to the Excel file, open the execution test-suite and adapt the global variable `testFile` variable to target your specific Excel file.
- Turn off the QF-Test debugger. It would only interfere with the steps of the manual tester.
- Start the test-suite via selecting the test-suite node and pressing "Start test-run".
- QF-Test will now read the data from the Excel file and open a dialog containing the first test-step.
- Enter the result of the test-step and proceed with executing each test-step.
- At the end of the test execution QF-Test will write a new Excel file containing the test description and the according results. You can also store the run-log of that execution or create an HTML report.

Please read the comments in the test-suite and Excel file carefully, because you can adapt this concept according to your needs. It is even possible to start only specific tests.

22.3 Structure of the Excel file

The Excel file has a specific structure which allows you to describe the manual test-steps quite flexibly. The meaning of the columns is explained in the following table:

Column	Description
TestCase	A unique identifier for each test-case. If the step belongs to the same test-case as the previous step, just leave this column empty.
Type of Test	Optional definition of the kind or function of the test or step, e.g. a functional test or a usability test, startup, etc.
Comment	An individual comment for the test-case. This comment will be shown in the run-log of QF-Test.
Short Description	A short description about the content of the test.
Step Description	The detailed description of the manual step.
Expected Result	The description of the expected result of that test-step.

Table 22.1: Description of the Excel file for the definition of manual tests

The Excel file with the results of the manual test execution will contain two additional columns as follows:

Column	Description
Received Result	The result the tester received during test-execution. If a test-step fails, the tester must specify a received result.
State	The state of the test, i.e. PASSED, FAILED, CANCELED or SKIPPED.

Table 22.2: Description of the Excel file with the results of manual tests

22.4 The ManualTestRunner test-suite

The `ManualTestRunner.qft` test-suite contains some global variables at suite-level which provide fine-grained control over test-run. These are explained in the following table. All variables not listed here are used internally by the test-suite and should not be changed.

Global Variable	Description
testFile	The path to the test-step definition Excel file.
testSheet	The worksheet of the Excel file containing the test-steps.
resultSheet	The name of the worksheet for the results.
tests	A list of tests to be intended to execute. If this variable is empty, all tests will be executed. If you want to execute only test 5 and 6, you can specify 5, 6 or 5–6. It is even possible to specify things like: 1, 3–5, 7 to execute the tests 1, 3, 4, 5 and 7.
defaultState	The default selection of the state. You can set it either to PASSED or FAILED. All other states will be converted to FAILED.
testCaseColumn	The heading of the column containing the test-case number.
commentColumn	The heading of the column containing the comment.
shortDescColumn	The heading of the column containing the short step description.
stepDescColumn	The heading of the column containing the full step description.
expResultColumn	The heading of the column containing the expected result.
recResultColumn	The heading of the column containing the received result.
stateColumn	The heading of the column containing the state of the test.

Table 22.3: Description of the global variables in the ManualTestRunner test-suite

22.5 Results

An executed test-step can be set to one of the following states:

Result	Description
PASSED	The test-step was successful.
FAILED	The test-step failed.
CANCELED	The test-step was canceled.
SKIPPED	The test-step was skipped.

Table 22.4: States of manual test-execution

Part II

Best Practices

Chapter 23

Introduction

This part of the manual describes best practices based on lessons learned from several customer projects and user feedback. The concepts described should assist you in finding the best strategy for using QF-Test in your projects.

Note

QF-Test is a very generic tool. The hints and experiences described here are just suggestions from our point of view, which we hope will support you in working efficiently and successfully with QF-Test in your project. But they are just one way of doing things and you will have to find your own solution that works best for your specific project.

In case you are interested in how other customers use QF-Test you can post a question to our mailing list which you can reach via <http://www.qfs.de/mailman/listinfo/qftest-list>.

Chapter 24

How to start a testing project

This chapter talks about the most important aspects that should be considered **before** you start to use QF-Test widely in your testing project. It mostly raises questions and gives general answers with references to more detailed information.

The aim of this chapter is to provide hints about issues which you should take care of in order to make your GUI tests reliable, stable, repeatable and especially maintainable.

24.1 Infrastructure and testing environment

Before you start creating and running automated tests you should think about some general matters pertaining to the environment where the tests have to run. In order to make tests reliable and repeatable you have to take into account that you must be able to bring your SUT into a well-defined state, which includes the state of its backend, e.g. a server and/or a database. If you do not think about such aspects it might become very difficult and sometimes quite tricky to re-run a test or simply to analyze test-results and maintenance of tests can become a nightmare.

Please consider the following topics:

1. What is the initial state of your SUT?
 - Which user is the actual user running the tests in your SUT? Most projects work with dedicated test-users for running tests. Another approach could be to have one test-user per test engineer.
 - Which language setting of your SUT is the primary one? Is it really required to reach a full coverage of all supported languages or is it sufficient to run the bulk of the tests in one primary language and create only a few tests to specifically test localization? In most cases repeating tests in several languages just

covers the same functionality, so you gain no real new information after running them. However, unless you take precautions, the language setting will influence the component recognition of QF-Test, please see [chapter 27^{\(224\)}](#) for details.

2. What is the initial state of your database?

- Can you work with an extra test database or do you have to use a production database? Test databases contain test-data with designed and planned content whereas production databases contain real-life data. Is the latter predictable and reliable? What about the danger that tests can mess with and possibly destroy production data? If at all possible you should avoid running automated tests in a production environment.
- Can you clean up or reset the environment after one test-run for re-running the test? Is it possible to undo changes in the database or is it required to use new test-data for the next regression phase?
- How can you read or write test-data? Do you want to use standard SQL scripts or can you reuse libraries from development? Some projects even re-install the whole database before every test-run because they cannot reuse any test-data or clean the database correctly.

3. Do you want to integrate QF-Test with other tools, e.g. build tools or test management tools?

- How to integrate QF-Test with a test management tool? If you can reuse already planned test-steps you can avoid redundant work in planning tests and creating them. For the standard integration for such tools, please see [chapter 19^{\(163\)}](#).
- Should test's be launched by a build-tool? If you have created tests you can run them unattended and trigger the run by a build-system like Ant or CruiseControl. Please see [chapter 16^{\(138\)}](#) for details about test-execution.
- Should test-results be uploaded to a reporting system or into a test-management system or is it more sufficient to put the HTML reports and run-logs on a centralized HTTP-server?

4. Who will work with QF-Test?

- Do only one or two engineers work with QF-Test or do all developers and business testers participate in test development? You can find some hints about working in a team with different roles in [section 25.5^{\(217\)}](#).
- What are the skills of the engineers? It is recommended to have at least one dedicated person with a good QF-Test knowledge in the team, who is also capable of implementing scripts and understanding software development principles.

Of course there will be more issues to take care about which are specific for your project. Try to figure them out.

24.2 Location of files

You should also think about following aspects of saving or installing files:

1. Where to install QF-Test to? QF-Test can be installed locally on every system but this forces you to update every system manually whenever you need to upgrade to a new version. You can also install QF-Test on a shared network drive, if your network is reliable, see [section 24.2.1^{\(211\)}](#) for details.
2. Where to store the configuration file `qftest.cfg`? Among other things that file contains information about how QF-Test should recognize components or what should go into the run-log. These options have to be the same for every QF-Test user, otherwise you cannot share tests in your team. To ensure that you can either use a shared network installation for QF-Test or specify the config file via command-line parameters when launching QF-Test. Make sure the shared config file is write-protected unless you explicitly want to change it. For details, see [section 1.5^{\(7\)}](#).
3. Where to store the license file `license`? You should put the license file to a central place in order to update the license only once when you receive an update for it. Again, you can either have a shared network installation for QF-Test or can use command-line parameters to specify the location of that file when launching QF-Test. For details, see [section 1.4^{\(6\)}](#).
4. Where to store the test-suites? The best place to store test-suites is a version management system where you can track the changes and access any version of the files. If this is not possible you should store them on a shared network drive.
5. Where to store the test data files? Test data files are associated with test-suites so you should store them closely to the suites, i.e. either in the same version management system or on a shared network drive.
6. Where to store the HTML reports and run-logs? You should put those files in a centralized place where any engineer can take a look at them to evaluate the tests-results. Most people tend to use an HTTP server or a shared network drive for that.

24.2.1 Network installation

If you plan to install QF-Test on a shared network drive you have to take care about some specific things.

The main source of conflict is the system settings file `qftest.cfg`. It is actually a good (and necessary) thing to have all users use the same system settings, especially for the recognition options. Sharing the system settings file facilitates this. However, this file should be made read-only so that one user will not inadvertently change the system settings for everyone. If the file is read-only, QF-Test will not save the system settings upon exit. Any change to these settings will have to be made by explicitly making that file writable, then exiting QF-Test and then making it read-only again. Alternatively each user could specify a different system settings file via `-systemcfg` but that's not advisable.

If you test a Java/SWING application and you uncheck the option "Connect without JDK instrumentation", then the list of instrumented JDKs could become another source of troubles. That file is stored centrally. But as each user may have different JDK versions or directories that file could get confused. However, you can simply just let that list become a superset of all JDK directories on all systems. Whether or not a JDK is instrumented on a given machine is always determined at runtime when the JDK instrumentation dialog is opened.

The running QF-Test instances will also share the log directory (internal logging, not a problem) and the Jython package cache which can occasionally cause problems so that QF-Test cannot initialize its Jython interpreter. This doesn't happen often and can be fixed by clearing (not removing) the Jython cachedir.

For Windows, each user should also execute the `setup.exe` for the primary QF-Test version, located in the installed `qftest-x.y.z` directory, to get proper registry settings and documentation links on his machine.

In the rare case when a QF-Test patch overwrites existing jar files of QF-Test, running instances based on those jars may crash on Windows.

24.3 Component Recognition

The most important aspect of a GUI testing tool is a stable and reliable recognition of the graphical components. In that area QF-Test is very flexible and can be configured in several ways. In most cases the default configuration for the component recognition works well, but sometimes you may have to change it.

If you change the component recognition options after creating lots of test-cases, those test-cases may break. Therefore you should try to find the most appropriate settings for your project as early as possible. It is worth spending time in that area before starting

to implement a huge amount of tests because in the worst case you might have to re-record or at least update all or most of the existing test-cases after a critical change of the recognition options.

Best start by recording some demo test-cases and figure out how QF-Test recognizes your SUT's components. The recognition mechanism is described in [chapter 5^{\(37\)}](#) and [chapter 6^{\(52\)}](#). If you re-run those demo test-cases - ideally on different versions of your SUT - and run into recognition problems, you have to ask yourself following questions about those tests:

1. Are there enough synchronization points, like Wait for component to appear or Check nodes with timeouts to execute test-steps only if the SUT is ready for them?
 - (a) Sample 1: After opening a window you can only work in that window, if it is really there -> Use a Wait for component to appear node.
 - (b) Sample 2: After pressing on a search button, you can only continue with the test when the search is really over -> Use a Check node with a timeout.

Another important aspect besides synchronization points is the correct approach of recognizing components. You have to ask yourself the following questions to determine, which recognition approach might be the most appropriate one:

1. Do the developers use unique and stable names for their components? Please take a closer look at [section 30.13^{\(535\)}](#).
2. Perhaps it is sufficient to use a regular-expression for the Feature attribute of the component of the main window under the Windows and components node. Please see [section 27.3^{\(228\)}](#) for details.
3. If development did not set useful or even dynamic names it may be required to implement a NameResolver. Please take a closer look at [chapter 27^{\(224\)}](#).
4. Do any of the QF-Test recognition options need to be changed? These are described in [chapter 27^{\(224\)}](#).
5. Is it possible to use generic components? See [section 27.6^{\(232\)}](#) for details.

In some cases it is sufficient to change the default configuration. Let us assume the developers have set unique and stable names for the target components, i.e. buttons, textfields, checkboxes etc. In such cases it may be sufficient to just change the 'Name override mode' setting of QF-Test to 'Name overrides everything'. This setting tells QF-Test to ignore any changes in the component hierarchy and just work with the target components and the window directly.

Note

You have to change this option in two places: Once at 'Record' -> 'Components' -> 'Name override mode' and at 'Replay' -> 'Recognition' -> 'Name override mode'. See [chapter 27^{\(224\)}](#) for more details.

Chapter 25

Organizing test-suites

One of the most challenging tasks in a project is keeping the test-suites maintainable over a long period of time. Especially if some window or workflow changes significantly the maintenance effort should be kept to a minimum.

It is also worth thinking about how to minimize the creation efforts of tests that contain a lot of similar or even the same steps. A typical use-case is launching the SUT or the login process or a very important and basic workflow like navigating to a certain part of the SUT.

Another aspect to think about is how to efficiently organize test-suites if different people work in your testing project.

3.5+

In any case you should create your test-suites within a QF-Test project as described in [chapter 10^{\(86\)}](#). This feature provides a better overview over your test-suites and directories.

The following sections show some best practices how to keep your tests maintainable, extensible and well-organized.

25.1 Organizing tests

In chapter [section 9.2^{\(80\)}](#) we describe the concepts of Test-set and Test-case nodes. A Test-case node stands for one dedicated test-case and its test-data. A typical Test-case could be derived from a use-case, a requirement or a defect description in your environment, e.g. 'Calculate the price for vehicle xyz at 10% discount' for the JCarConfigurator application.

Test-set nodes are collections of Test-sets and Test-cases which can be used for organizing test-cases, e.g. 'Tests for calculating prices'.

Test-step nodes represent the individual test-steps of a Test-case like 'Open window' or

'Check calculation'.

If you have an external description of the Test-case or any other associated information which might be important for it, it is recommended to add an HTML link to it in the Comment attribute of the Test-case. You will see that link in the report later. It is even possible to create a separate test documentation using the menu action **File→Create HTML/XML testdoc**. More details about documentation can be found in chapter 15⁽¹²⁹⁾.

3.1+

The report and test documentation also contain the Test-step nodes which are used in a Test-case.

If a Test-case consists of lots of Procedure calls or Sequences, you should organize the single test-steps in Test-step nodes. Those Test-step nodes have the advantage that you can really see every significant step in the QF-Test window and also in the report later.

If you want to put several nodes into a Test-step you can pack the respective nodes into a Test-step via selecting them, perform a right-mouse click and selecting **Pack nodes→TestStep**.

25.2 Modularization

One of the most important concepts for effective test automation is modularization. Modularization here means placing reusable sequences in a dedicated location and calling these whenever possible. This concept enables you to create a sequence only once and reuse it as often as you require it in your tests without re-recording the same steps all the time. Changes in the SUT that require an update of the tests that rely on such a sequence, e.g. a change to some basic workflow, can then be handled by updating just the procedure in a single location instead of many identical sequences spread all over the test-suites.

The modularization concept in QF-Test is implemented via Procedure nodes. Procedures are well described in section 9.3⁽⁸²⁾.

If you have lots of test cases, it is best to have almost every test-step as a Procedure and create those procedures up front, if possible. With those procedures in place you can fill your test-cases very fast by just adding the respective Procedure call nodes.

In larger projects it is useful to have Procedures at different level, e.g. component specific procedures like 'Click OK' and workflow oriented procedures like 'Create a vehicle'.

25.3 Parameterization

The concept of modularization enables you to maintain test-steps at a single location in your test-suites. But how to use different test data for different tests?

If you have a Procedure that can be called with different test data, e.g. a typical 'Login' process with name and password or the 'Select accessory' procedure of the JCarConfigurator, you can use variables within the QF-Test nodes. Those variables should be used at places where test data is usually being accessed. In most cases variables will be used for Text input nodes or for selections of items in a list or a table or tree nodes.

If a procedure requires variables you should define the required variables in its list of Variable definitions⁽³⁸⁰⁾. This is to ensure that you get a list of all required parameters whenever you add a respective Procedure call node for that Procedure to your tests. Some customers even set dummy default values for parameters so they can recognize immediately when a parameter has not been initialized by the calling test.

The next step is to move those variables from the Procedure call either into the Variable definitions⁽³¹⁹⁾ section of the Test-case node or to put the test data into a Data driver node with a Data table or using an external data source.

The usage of variables and parameters is well described in section 9.3⁽⁸²⁾. Parameters can also be created automatically, please see section 9.3.4⁽⁸⁴⁾. You can find more details about the data driver concept for loading test data from a data source in chapter 14⁽¹²¹⁾.

25.4 Working in multiple test-suites

Up to now you have read about using the modularization and parameterization concept to avoid unnecessary and redundant work in the creation process of your tests. You should have recognized that those concepts will reduce the maintenance efforts of your test-suites by changing or updating only one single sequence instead of several ones. But we still do not know how to organize our work for different test engineers or for a very large project with a lot of GUI elements.

The answer for an effective working organization comes again from the software development area and it is to use several 'libraries' for different areas and different responsibilities.

Importing other test-suites into a test-suite enables you to follow that encapsulation approach. A typical organization of test-suites in your project could look like this:

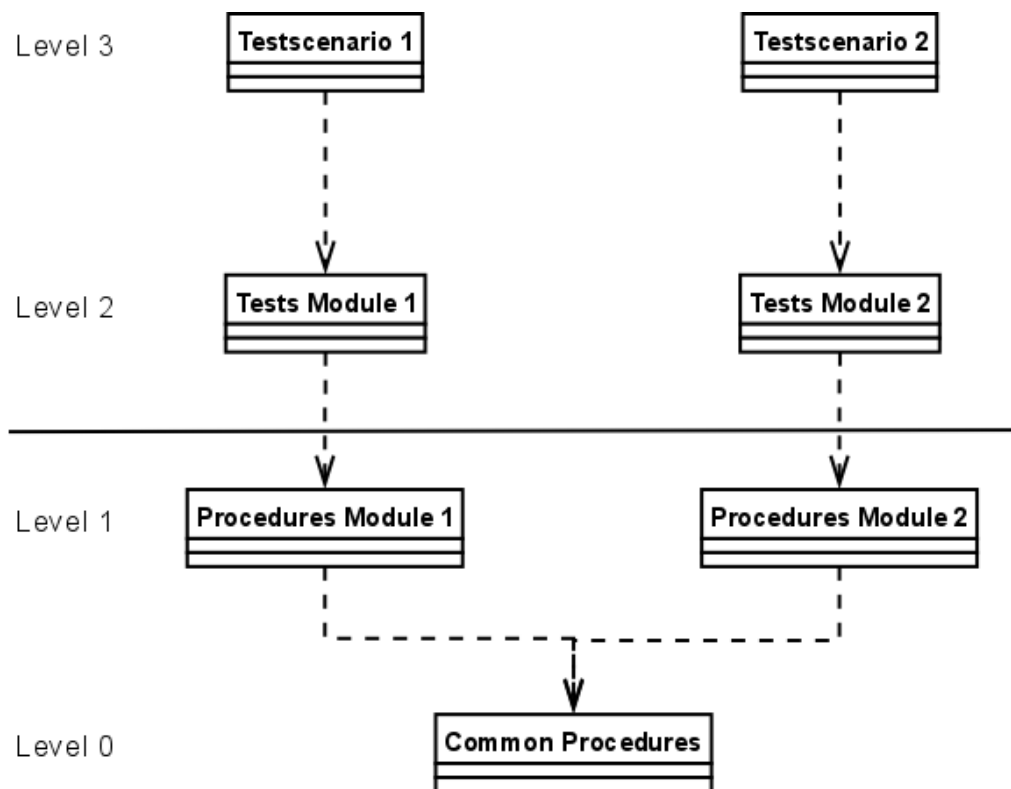


Figure 25.1: Structure of multiple test-suites

Level 0 is the level that contains test-steps (i.e. Procedures) which are required for nearly all test-cases in your projects. Such test-steps could be 'Launch SUT' or 'Perform the login'.

Level 1 contains test-steps for a specific part of the SUT. For the JCarConfigurator you can think about a test-suite 'Vehicles' containing Procedures like 'Create a vehicle', 'Remove a vehicle' and another test-suite 'Accessories', which contains Procedures like 'Create an accessory' or 'Remove an accessory'.

Level 2 is the test-case level. It contains Test-cases and Test-sets for the respective area of your software, e.g. 'Tests for vehicle creation' or 'Tests for accessory creation'. Of course you could also have a test-suite like 'Integration tests' which refers to test-steps from different test-suites at level 1 and level 0.

Level 3 is the so called scenario level. Those test-suites usually just contain Test calls to level 2 and stand for different scenarios within your test project, e.g. 'Nightly test scenario', 'Defect verification scenario' or 'Quick-test build verification'.

Note

The structure described in this document is of course just one possible solution for handling test-suites in a project and is not a strict rule you have to follow. You could also think about splitting level 1 into a GUI-element level and a workflow level or merge level

2 and level 3 to one level. Which structure you finally implement also depends on the experience and knowledge of the test-engineers working in your project.

The including area of level 1 test-suites looks like this:

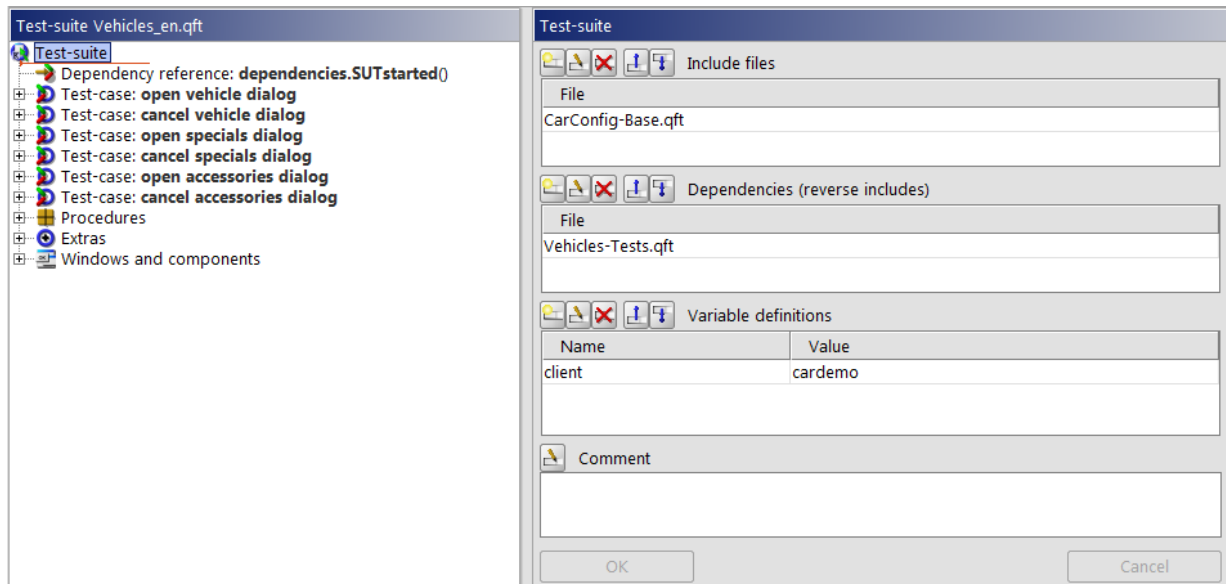


Figure 25.2: Including test-suites of level 1

You can find a detailed description of how to include test-suites in [section 17.1^{\(150\)}](#) and [section 36.7^{\(617\)}](#).

In [section 26.5^{\(222\)}](#) you can find a step-by-step description how to extend an already existing test-suite and in [section 26.4^{\(221\)}](#) you can find strategies of handling components in such a scenario.

25.5 Roles and responsibilities

If you take a closer look at the organization shown in the previous [section 25.4^{\(215\)}](#) you may recognize that it is also possible to organize your test-suites based on different knowledge levels of test engineers.

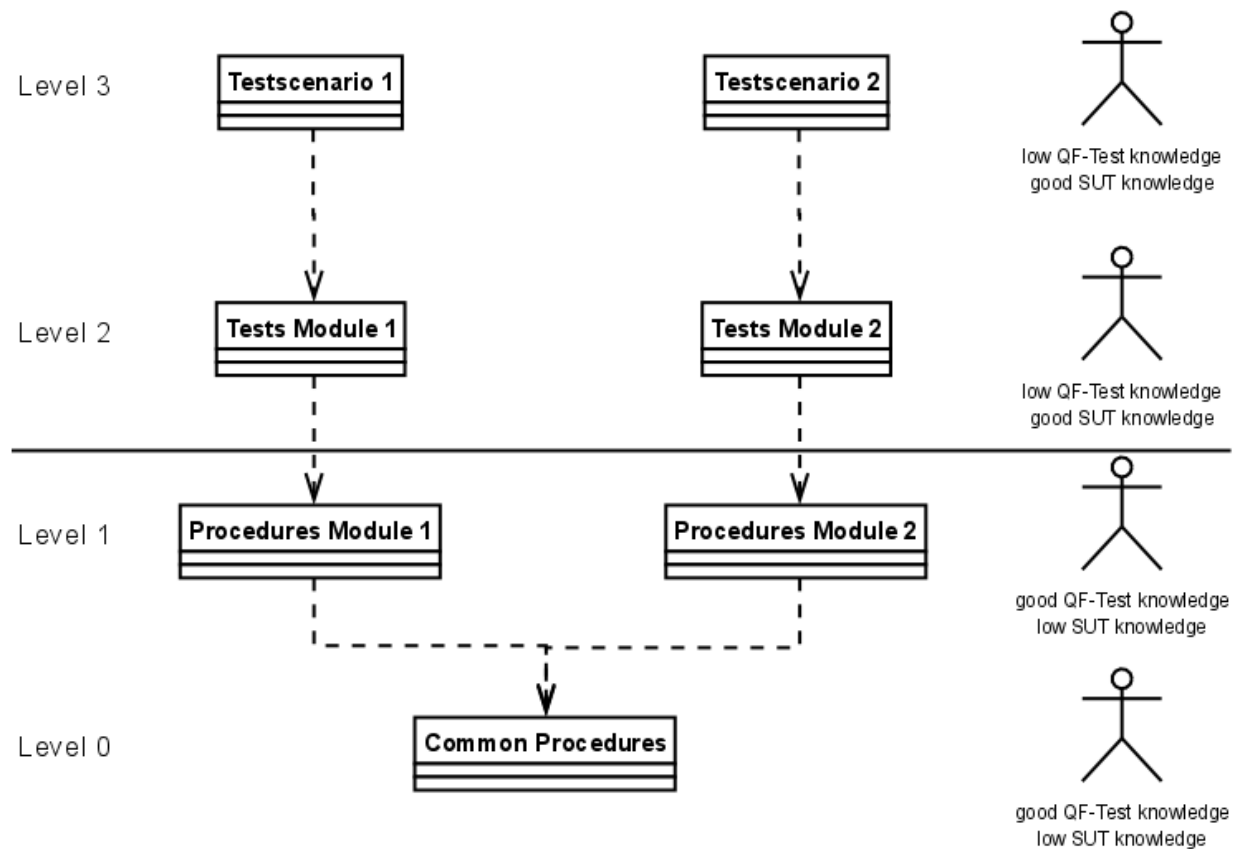


Figure 25.3: Structure of different test-suites with roles

Level 0 and level 1 require a good knowledge in working with QF-Test but not a deep knowledge of the SUT. On the other hand level 2 and level 3 require a very good knowledge of the SUT and the planned test-cases but those engineers usually do not require a very deep knowledge of QF-Test as long as they just use procedures from level 0 and level 1.

Test engineers working in level 0 and level 1 should be capable of implementing scripts or control structures like the Try/Catch concept which enables them to create strong and powerful test libraries. At least the engineers working on level 0, but also recommended for engineers working on level 1, should have a good knowledge about component recognition in QF-Test. Please see [chapter 5^{\(37\)}](#), [chapter 6^{\(52\)}](#) and [chapter 27^{\(224\)}](#).

Note

Even if you are working alone on a project it is strongly recommended to split the tests and procedures into different levels because maintenance will become easier than with everything kept in one huge test-suite.

25.6 Managing components at different levels

If you follow the approach suggested in the previous section ([section 25.4^{\(215\)}](#)) you have to define where the components belong. There are two possibilities:

1. Put all components into level 0.
2. Split the components like the procedures into several levels.

Storing all components in level 0 is the most simple solution but this could cause you to update level 0 very often, just because one single component in your project changes. You have to assign responsible persons to keep that structure cleanly.

In big projects you may consider storing the common components like the login dialog or the main frame menus, that are important for everyone, in level 0. Components specific to a certain area, e.g. a dedicated vehicle dialog, appear only in the test-suite that holds the procedures operating on those components.

The workflow for moving components between test-suites is described in [section 26.4^{\(221\)}](#) and the workflow for extending existing test-suites is described in [section 26.5^{\(222\)}](#).

25.7 Reverse includes

You don't need to care about [Dependencies^{\(312\)}](#) of test-suites belonging to a QF-Test project as described in [chapter 10^{\(86\)}](#), because QF-Test automatically resolves dependent suites. So, if you use the concept of projects, you can skip this chapter.

If you work in different test-suites in your project you might sometimes want to rename a Procedure or a Test-case. If you do that you may encounter some troubles in updating the references to that Procedure or Test-case in other test-suites. If you want to keep the other files also being updated after renaming such an element you have to maintain the [Dependencies^{\(312\)}](#) attribute of the root node of the library test-suite.

If you follow the approach described in [section 25.4^{\(215\)}](#), you should ensure that level 0 contains a reverse include to level 1, level 1 should contain one to level 2 and level 2 should contain another one to level 3. A sample from the provided demo test-suites is shown in [figure 25.2^{\(217\)}](#).

Chapter 26

Efficient working techniques

This chapter will help you in optimizing your working techniques and avoid unnecessary steps when working with QF-Test.

26.1 Using QF-Test projects

3.5+

The previous chapter describes the creation of several test-suites. As you can image the amount of test-suites will increase over time. You can get a better overview over them using a QF-Test project.

QF-Test projects show all involved test-suites in a very nice way. Furthermore projects automatically take care of propagating modifications to referring test-suites. You can find more information about projects at [chapter 10^{\(86\)}](#).

26.2 Creating test-suites from scratch

In the previous chapter we described the concept of creating maintainable test-suites by utilizing procedures and variables within QF-Test. Normally people start recording very long sequences and splitting those into smaller parts or even procedures later. It is very hard to split up long sequences as you have really to walk through the whole one to find proper boundaries. Another disadvantage is that you cannot see parts which have already been implemented in existing test-cases or procedures.

Instead of that described workflow we recommend to plan the tests and their test-steps including procedures first. Then you can start recording procedure by procedure. We came to the conclusion that anticipatory recording and creation is very helpful especially for working in bigger teams. A typical workflow for creating those procedures looks like this:

1. Plan the required procedures first.
2. Also plan the required package structure first.
3. Record every procedure as a separate sequence.
4. Rename the recorded sequence like the procedure should be called.
5. Transform the recorded sequence into a Procedure, using the Transform node into... action from the context-menu of QF-Test.
6. Move the Procedure to the correct location.
7. Replace test data with variables, either manually or by using the parameterizer (see [section 9.3.4^{\(84\)}](#))
8. Add the variables to the Variable definitions of the Procedure node, possibly specifying default values.
9. Describe the procedure in its Comment attribute, see [section 9.4^{\(85\)}](#).

An alternative approach of creating procedures is the automated creation provided by QF-Test. This concept is described in [chapter 18^{\(159\)}](#).

26.3 The standard library qfs.qft

QF-Test provides the standard library `qfs.qft` which is included per default in every test-suite.

This suite contains many useful procedures for accessing components, the file system or a database. Please take a look at that library before you begin implementing something that has already been solved by us.

26.4 Component storage

QF-Test records new components in the test-suite where you press the 'Stop recording' button. Therefore it could happen that you record components in the wrong test-suite.

If you want to move those components into another test-suite you must always use File→Import from the target suite. Take care to ensure that both suites belong to the same project or to specify correct 'Include files'/'Dependencies' relations in those test-suites. This workflow is described in detail in [section 17.2^{\(152\)}](#).

3.1+

For cleaning your component structure in a test-suite you can first import the test-suite

into itself. Then you can select the **Windows and components step**, open the context menu via a right mouse-click and click at **Mark unused components...**. You will get a list of all components which are not used in the project. If you are sure that those components can be removed, then select the **Remove unused components** in the context menu of the **Windows and components step**.

26.5 Extending test-suites

Several workflows can be followed to extend existing test-suites:

1. Simply record and extend the target test-suite directly.
2. Work with a scratch suite as described in [chapter 17^{\(150\)}](#).

If you extend testsuites directly via clicking at 'Stop recording' in that suite, then you have to care about the recorded components. In case you have changed the recorded component hierarchy under **Windows and components** the newly recorded components will be recorded in the normal hierarchy again and you have to move the new components to the optimized hierarchy. Another aspect is that it could become very difficult for moving single components into another test-suite under the right location there.

If you work in a scratch suite you can create the new test-steps temporarily in a completely new test-suite and import the recorded components and the created procedures and test-cases into the target suites together.

A detailed workflow for extending a test-suite from level 1 (from [section 25.4^{\(215\)}](#)) could look as follows:

1. Create a new test-suite.
2. Add the test-suite to be extended to the **Include files** area of the new one.
3. Save the new test-suite.
4. Ensure that both test-suites belong to the same project or add the new test-suite to the **Dependencies** area of the test-suite to extend.
5. Record the new test-steps in the scratch suite. Also create Procedures, if required.
6. Then import the components, procedures and tests into the target test-suite as described in [section 26.4^{\(221\)}](#) and [section 17.3^{\(153\)}](#).

A more detailed description of working in multiple test-suite can be found in [chapter 17^{\(150\)}](#).

26.6 Working in the script editor

The script editor of QF-Test contains some fancy features to avoid too much typing actions.

If you want to call methods of the run-context `rc`, you can simply type `rc.` and press `Ctrl-Space`, then you will get a list of all available methods.

The auto-completion is also working for several variable names, which are:

Variables	Methods
<code>doc</code>	Methods of a <code>DocumentNode</code> .
<code>frame</code>	Methods of a <code>FrameNode</code> .
<code>iw</code>	Methods of the <code>ImageWrapper</code> .
<code>node</code>	Methods of a <code>DOMNode</code> .
<code>Options</code>	The keys and values of QF-Test options to set.
<code>rc</code>	Methods of the Run-context.
<code>resolvers</code>	Methods of the Resolvers module.
Just press <code>Ctrl-Space</code> without typing anything	A list of all variables with auto-completion.

Table 26.1: List of variables with auto-completion.

Chapter 27

How to achieve robust component recognition

The most important feature of a GUI test-tool is the recognition of the graphical components. QF-Test offers a lot of configuration capabilities in that area. This chapter gives an overview over the most common strategies and settings to make the component recognition as stable as possible in a project.

We also recommend to take a closer look at [section 30.13^{\(535\)}](#) and [section 36.2^{\(612\)}](#).

Note

You have to figure out the component recognition strategy **before** you start utilizing QF-Test for a large testing area in your project. Otherwise you can get into serious troubles in maintaining your tests. Some hints for doing that are described in [chapter 24^{\(208\)}](#).

27.1 Using the default configuration

In most of the projects the default configuration works very well. The default configuration takes the name of a component, its class (type) and also its hierarchy (structure of window) into account for recognizing a component.

You could get troubles using this configuration, if

- Developers use non-unique names.
- The component hierarchy changes significantly.
- Names change significantly (see [section 27.4^{\(229\)}](#)).

If developers do not assign any names, QF-Test uses a combined probability calculation of properties, like the label, the text of a button or the geometry for recognizing a component. That propability calculation is quite good if at least a few properties stay similar

over time, e.g. the label text. But you should ensure that the windows of the SUT are always opened with or resized to the same size.

The default configuration for recording components looks like this:

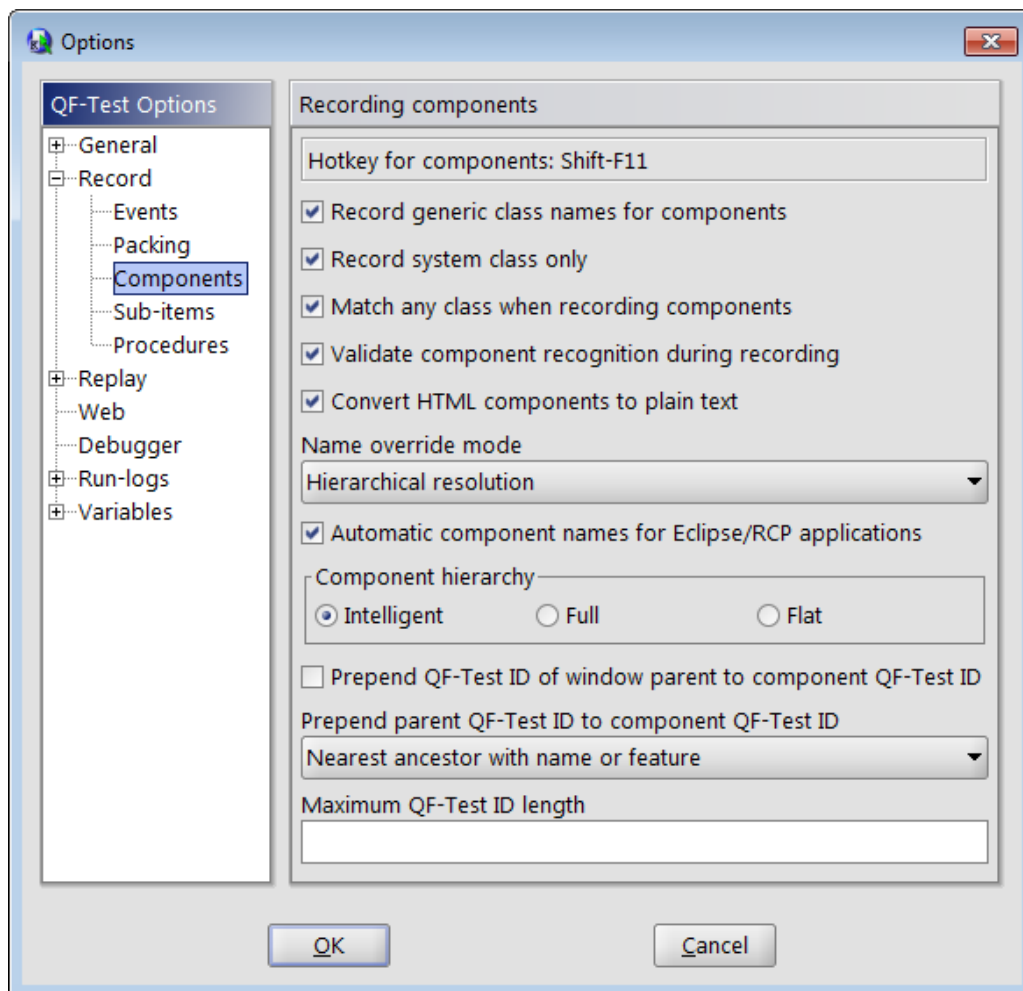


Figure 27.1: Default configuration for component recording

The default configuration for recognizing a component during playback looks like this:

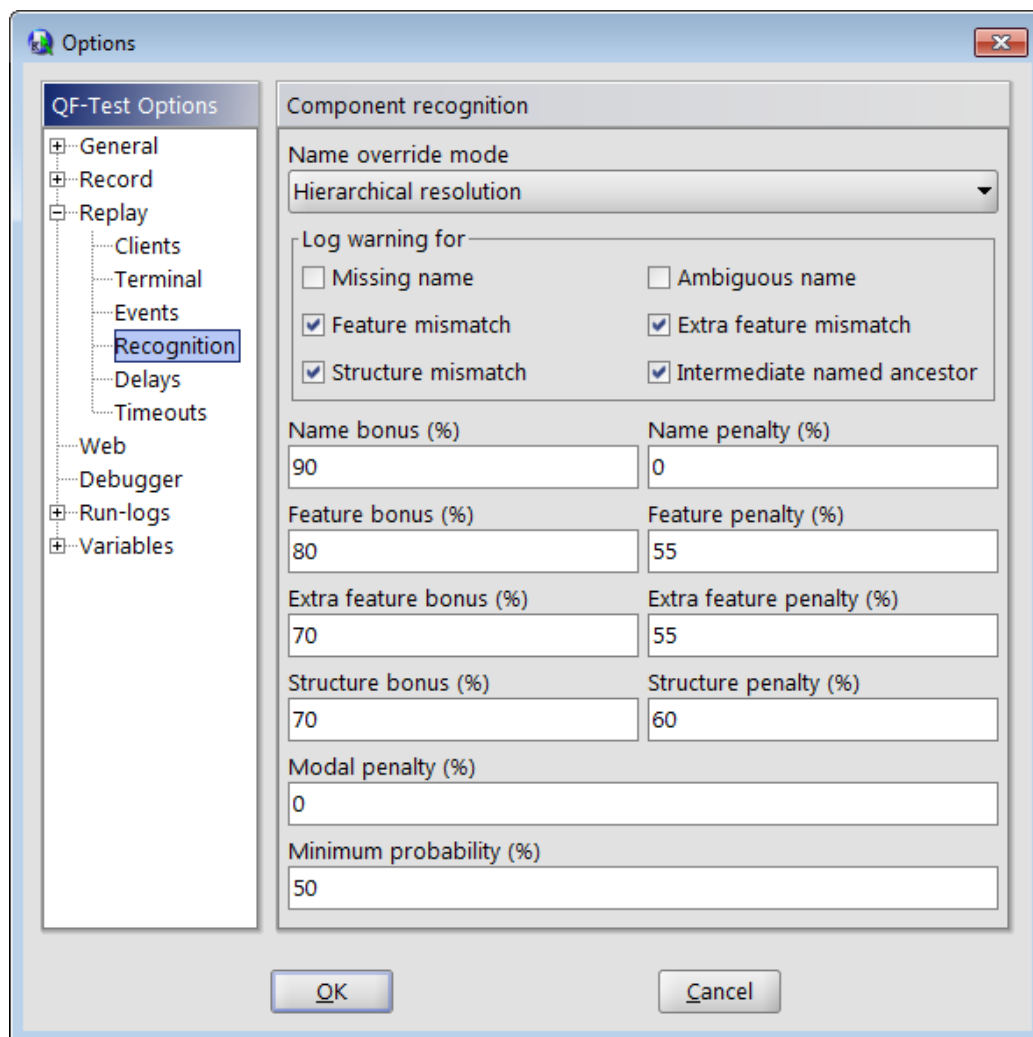


Figure 27.2: Default configuration for component recognition

27.2 Using the 'Name overrides everything' mode

If your developers use stable and unique names for all important components which can be accessed by a test, e.g. buttons, tables or text-fields, you can consider changing the configuration to the 'Name overrides everything' mode. This has the advantage that only names of those components and the window they belong to will be used for recognizing the component. Changes to the SUT's component hierarchy will have no effect on the recognition of those components based on their unique name.

This setting will cause trouble if:

- Developers use non-unique names.
- Names are not stable (see [section 27.4^{\(229\)}](#)).

The 'Name overrides everything' configuration for component recording looks like this:

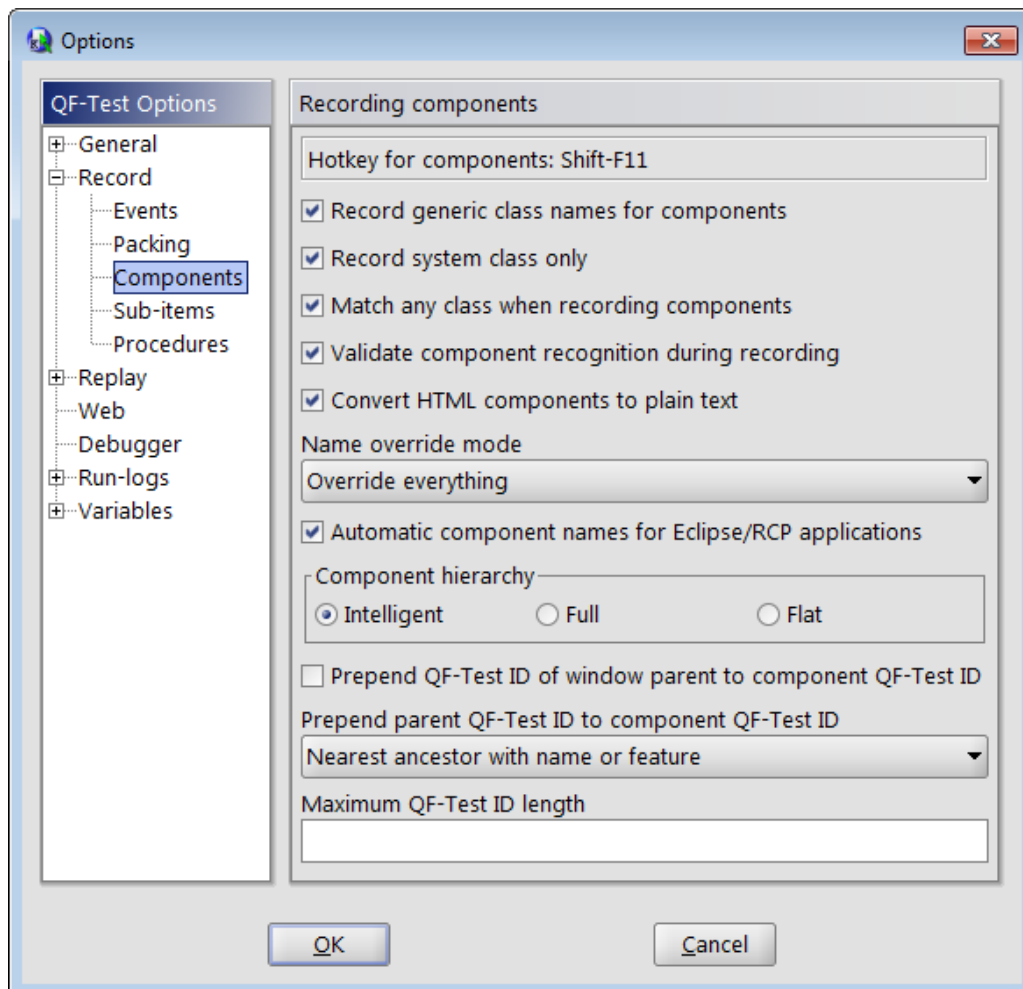


Figure 27.3: 'Name overrides everything' configuration for component recording

The 'Name overrides everything' configuration for recognizing a component during playback looks like this:

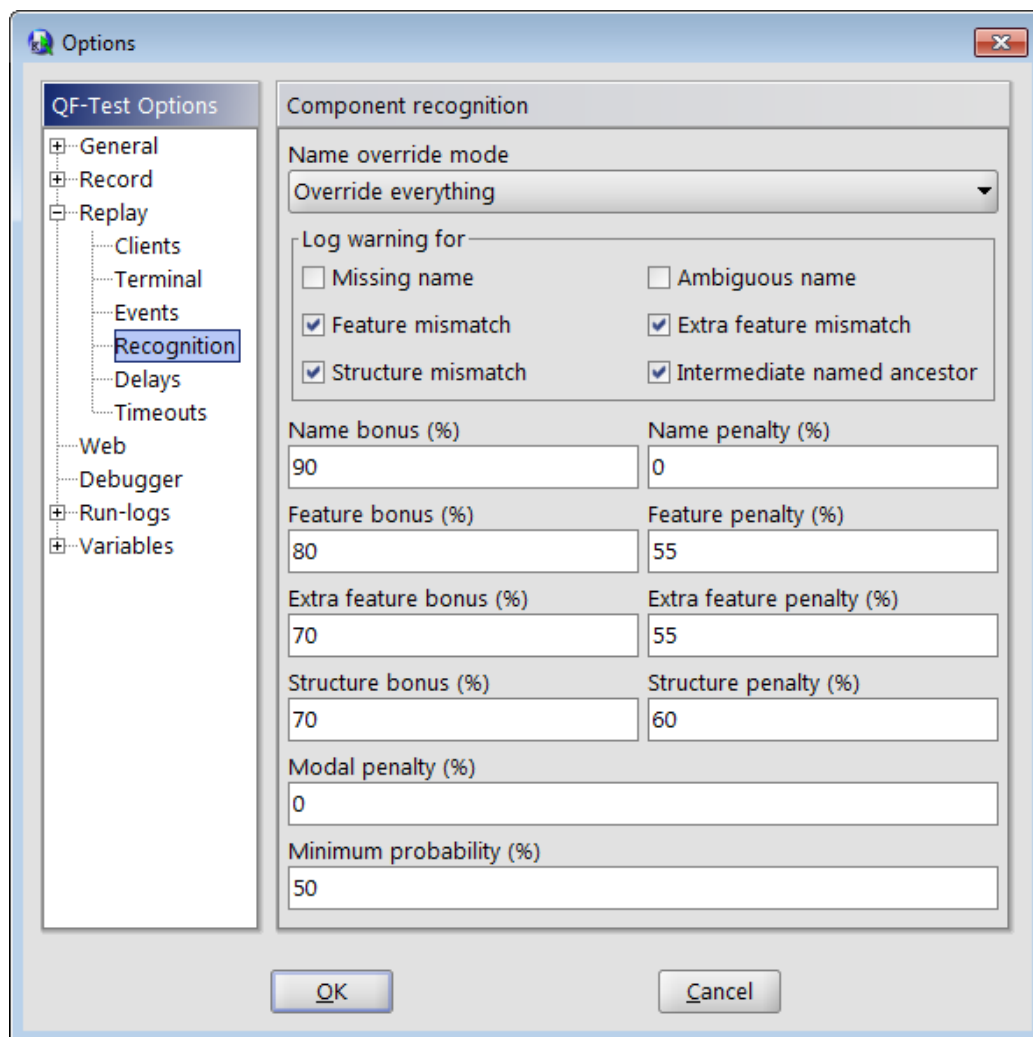


Figure 27.4: 'Name overrides everything' configuration for component recognition

27.3 Using regular expressions for working with dynamic window titles

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

In this case it is very likely that the title of the main window changes frequently, e.g. to display a version string, a user name, a file name or some other variable information.

If you want to keep your tests working and prevent recording multiple variants of this window and all its components, you have to select the respective Window node in the and edit its Feature attribute to replace the dynamic parts of that title with a regular expression. Be sure to check the 'Use regexp' box. Now your tests should work again.

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

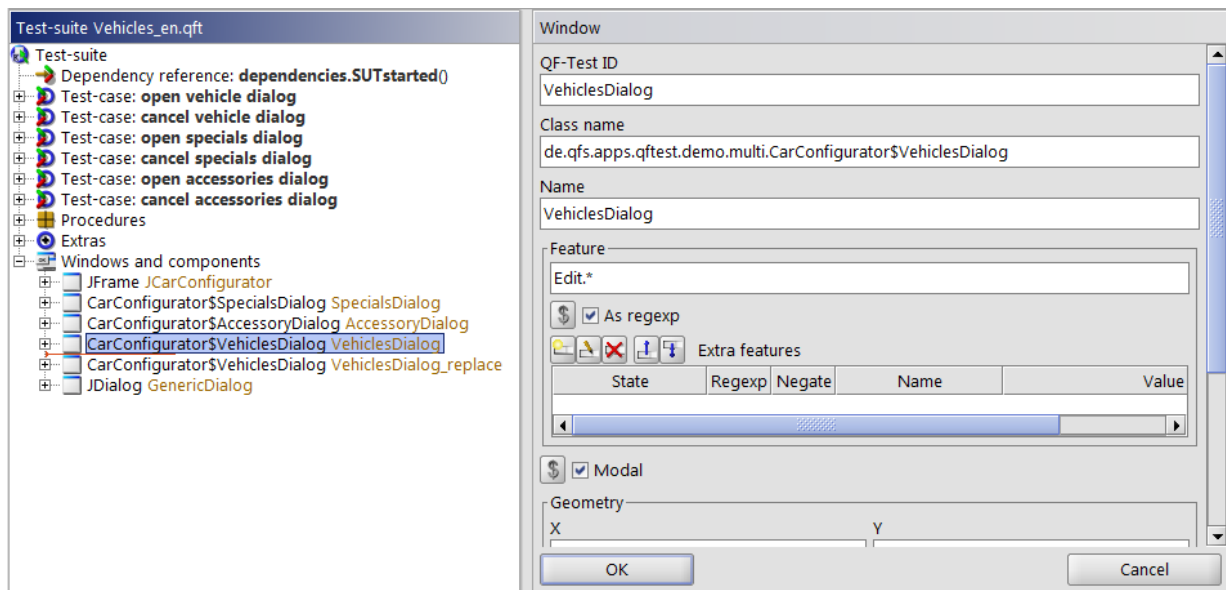


Figure 27.5: Using a regular expression in the Feature attribute

QF-Test uses regular expressions at many places. You can find detailed information at [section 36.4^{\(614\)}](#) to learn more about how to use them.

Note

QF-Test assigns the QF-Test ID of a component automatically and may use the Feature attribute to generate it like described in [section 5.3^{\(40\)}](#). The QF-Test ID attribute is an artificial concept for QF-Test's internal use to map actions onto recorded components. Thus it can be pleasing to change it for better readability afterwards. When changing the QF-Test ID QF-Test provides the opportunity to adapt all references automatically.

27.4 Influencing the names by implementing a NameResolver

In GUI testing projects you can face a lot of interesting naming concepts. Sometimes the components in an application have no names, but the testers know an algorithm how to name them reliably. Sometimes existing names change from time to time or

are completely dynamic, e.g. you can get a name 'button1' after the first recording and after the second recording you get 'button2'. Another situation could be that the current version of the application is part of the name of a dialog window.

In such cases you should take a closer look at the NameResolver interface of QF-Test.

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

You can think about utilizing NameResolvers in following cases:

- The SUT has dynamically changing names.
- You know a method to set the names uniquely.
- You want to map names to other names (e.g. due to new versions or for testing other languages.)
- You want to tune the names of components, e.g. to remove some parts and get nicer QF-Test component IDs in QF-Test.

If you can achieve per-window uniqueness of names with the help of a NameResolver you can also think about switching to the 'Name overrides everything' setting described in [section 27.2^{\(226\)}](#).

Note

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

NameResolvers are well described in [section 39.1^{\(678\)}](#).

27.5 Handling classes of components

4.0+

QF-Test records so-called generic classes, which means common class names for objects independent of their technical class.

A sample recording for Swing components leads to the generic class `Button` for any `javax.swing.JButton` even if you have extended that button class. For other engines you will get a similar recording with the respective classes of that toolkit. In addition this concept allows QF-Test to resolve obfuscated classes without any need to change the default configuration.

During replay QF-Test compares the 'class' attribute of the recorded component with any class of the objects in the SUT. Therefore QF-Test can cope with class changes as long as the standard type isn't changed.

You can find a brief explanation of the generic class concept in [section 5.4.1^{\(42\)}](#).

27.5.1 Handling class changes due to re-factoring by using 'Record system-classes only'

4.0+

This chapter typically should be relevant just for QF-Test versions older than 4.0 where no generic classes can be used.

By use of this option you can configure QF-Test to record the system base classes of components instead of custom class names. System classes are `javax.swing...` and `java.awt...` classes for Java/Swing and `org.eclipse.swt.widgets...` classes for Java/SWT applications.

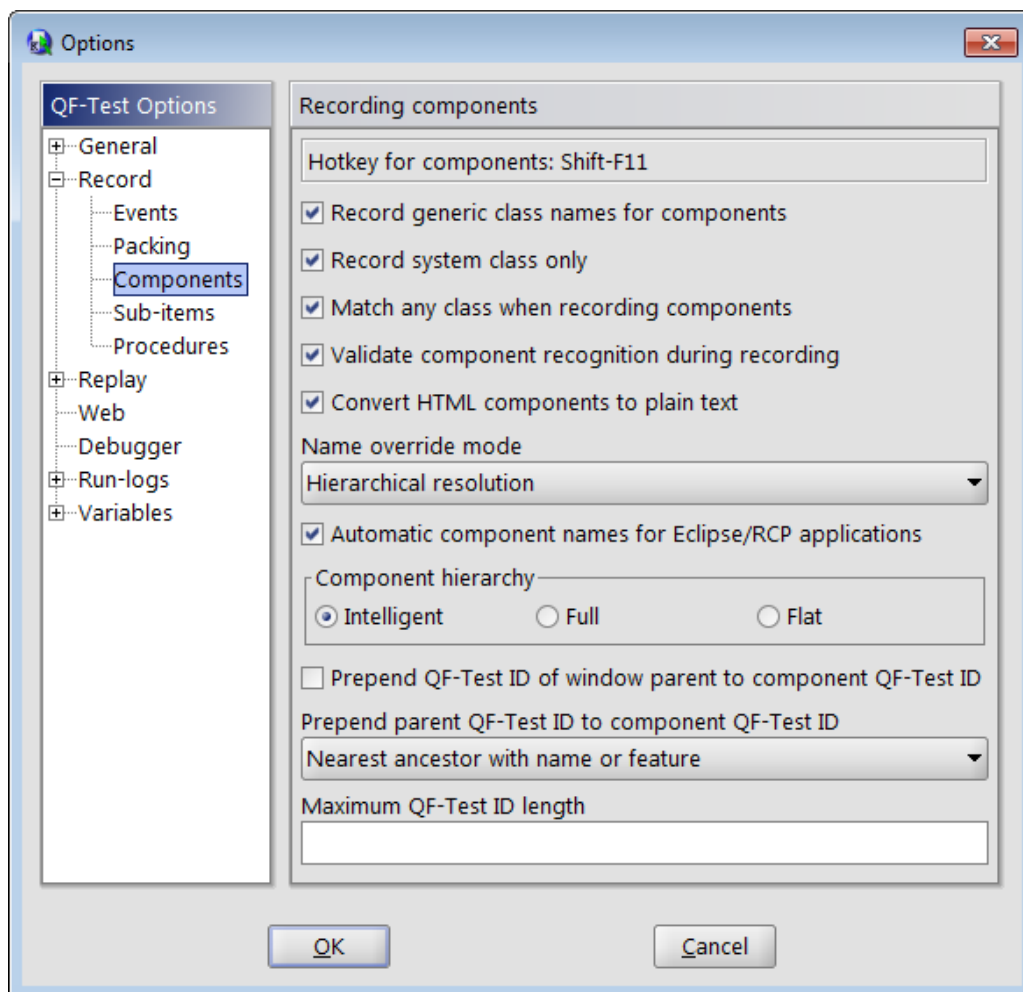


Figure 27.6: Option to configure recording of system classes only

Having just base classes, you will be safe against re-factoring steps of the developers who may change the class structure or rename classes. This option will also protect you from obfuscators which change classes names to protect them against re-engineering.

Having this option active should be helpful in most cases. Only if custom class names are vital for a proper component recognition of components, i.e. if development did not assign useful names but implemented different custom classes for specific containers you might want to consider a deactivation.

3.1+

A mixed approach is also possible by implementing a `ClassNameResolver`. This can be useful if you only want to get system classes for some components, but not all, e.g. when using a third-party library. Please see [section 39.1.2^{\(680\)}](#) for more details.

27.6 Avoiding recording every component or using generic components

By now you know how to optimize the component recognition for your needs, but you still have to record every component before you can use it in your tests.

As you can imagine, it can be tiring to record each and every component required for a test. Sometimes this may even become impossible, e.g. when you have changing identifiers or changing labels. A typical situation could be localization testing of an application.

Another situation could be the usage of a GUI framework of your developers. You could have a lot of very similar dialogs, where only some components are different. But you still have to record all parts, which are actually the same, like navigation buttons.

QF-Test offers a feature to reduce the recording of components to a minimum. This concept is called 'Generalizing components'. It consists of making use of variables in the component's details or simply to remove dynamic parts of it.

The general approach for generalizing components is as follows:

1. Record some components that you want to generalize and compare them.
2. Create one generic component with an ID, which contains 'generic' so that you can recognize it later.
3. Remove all attributes that you do not want to use in the recognition in that generic component.
4. Determine the main properties that you want to use for recognition, e.g. 'Name', 'Feature' or 'Class index'.
5. Set a variable for the respective attribute, e.g. `$(name)`.
6. To prevent false positive matches, disable recognition based on geometry by specifying '-' for the 'X' and 'Y' attributes.

7. Specify '@generic' in the Comment attribute for the generic component to prevent it from inadvertently being removed by the 'Remove unused components' action.
8. Create a procedure accessing the generic component and use the variable(s) from the previous step as parameter(s).

Note

Generic components are very handy for replaying tests, but QF-Test will not use them for recording. You will always get the real component recorded in the test-suite, but can then change the created sequence to make use of the respective generic component.

27.7 Switching component recognition settings dynamically

3.1+

Since QF-Test version 3.1 it is even possible to change component recognition options dynamically. Thus you are very flexible and you can combine different settings for your SUT.

Setting options at runtime can be done via a Server script or SUT script - depending on the option - and calling `rc.setOption(OPTION_NAME, OPTION_VALUE)`. All possible options can be found in [section 29.2^{\(254\)}](#) for recording and in [section 29.3^{\(270\)}](#) for replaying tests.

Chapter 28

Test execution

This chapter gives some hints about how to implement your tests to get stable and reliable test execution.

28.1 Dependencies

The 'Dependencies' concept of QF-Test provides functionality to guarantee that all pre-requisites for a test-case are fulfilled before running it. It is also capable of reacting to unexpected behavior, e.g. closing an error dialog, which pops up and blocks your tests.

The concept is described in [section 30.3^{\(343\)}](#) and a use-case can be found in the tutorial in the chapter 'Dependencies'.

You should at least implement a Dependency which is responsible for launching the SUT, containing a Setup for launching, a Cleanup for a normal exit and a Catch to react on any unexpected behavior.

Note

If you implement a Cleanup, try to close the SUT normally first and only if the SUT does not terminate correctly, kill it via Stop client.

For SWING and SWT applications please use the procedures `qfs.cleanup.swing.closeAllModalDialogs` and `qfs.cleanup.swt.closeAllModalDialogsAndShells` from the standard library `qfs.qft` for closing unexpected error dialogs.

28.2 Timeout vs. delay

Instead of using the 'Delay before' and 'Delay after' attributes you should try to use QF-Test's synchronization nodes to optimize test execution time.

The first kind of synchronization nodes are the 'waiter' nodes like Wait for component to appear, Wait for client to connect, Wait for document to load and Wait for process to terminate. You can specify the Timeout attribute to wait for a component, process or document. The Wait for component to appear node even provides the functionality to wait for the absence of a component.

The second kind are the 'check' nodes which allow you to specify the Timeout attribute as well. Those nodes can be used to continue the test when a GUI element of your SUT has reached a defined state.

28.3 What to do if the run-log contains an error

If the test report contains an error message or exceptions, the following steps should be performed to find the source of that failure very fast:

1. Analyze the run-log, especially the screenshots and any other messages.
2. If you cannot find the cause immediately, jump to the failing location in your test-suite by typing **Ctrl-T** in the run-log.
3. Set a breakpoint before or at the failing step.
4. Ensure that the debugger of QF-Test is enabled.
5. Run the failing test.
6. When QF-Test reaches the breakpoint and stops, open the debugger window and check the active variable bindings to see whether they contain any wrong values.
7. Perhaps at that time you can also see the error immediately in your SUT.
8. If you cannot see any source of that error, run the failing step.
9. If you still encounter errors you might have to re-debug some steps executed before the failing step. Use the 'Continue execution from here' menu entry to jump to previous steps instead of re-running the whole test again.

3.1+

Since QF-Test version 3.1 it is possible to mark nodes via the context menu-item **Set mark** or setting bookmarks for specific nodes via the menu item **Add bookmark**. These features enable you to find important nodes very fast again.

If you encounter problems with component recognition, please see [section 5.9^{\(49\)}](#) and [chapter 27^{\(224\)}](#).

Part III

Reference manual

Chapter 29

Options

Since QF-Test is a tool that is intended for a wide range of applications, the "one size fits all" approach doesn't quite work. That's why QF-Test has a great number of options that control its functionality.

There are two kinds of options for QF-Test: *user* options and *system* options. User options adjust the behavior of QF-Test's own GUI while system options influence how tests are recorded and replayed. Each user has its own set of user options whereas system options are saved in a common system file. See [section 1.5^{\(7\)}](#) for details about configuration files.

3.1+

Many options can have their value changed at run time from a script via `rc.setOption` as described in [section 37.6^{\(622\)}](#). Depending on whether the option takes effect in QF-Test itself or in the SUT, the documentation for those options shows a "Server script name" or "SUT script name" matching the constant from the `Options` class. Obviously the option has to be set in a matching [Server script^{\(416\)}](#) or [SUT script^{\(419\)}](#) node. Where the option's value can be selected from a drop-down list, the documentation also lists the constants that can be specified as the option's value.

Though the number of options may look daunting, don't let yourself be deterred by it. All options have reasonable default values, so QF-Test works well out of the box for most cases. However, if you find you need to change something or simply want to explore the range of QF-Test's abilities, this chapter is for you.

The options can be set in the dialog available through the menu item Edit→Options.... The settings are saved in two configuration files, one for personal settings and one for system-wide settings (see [section 1.5^{\(7\)}](#)).

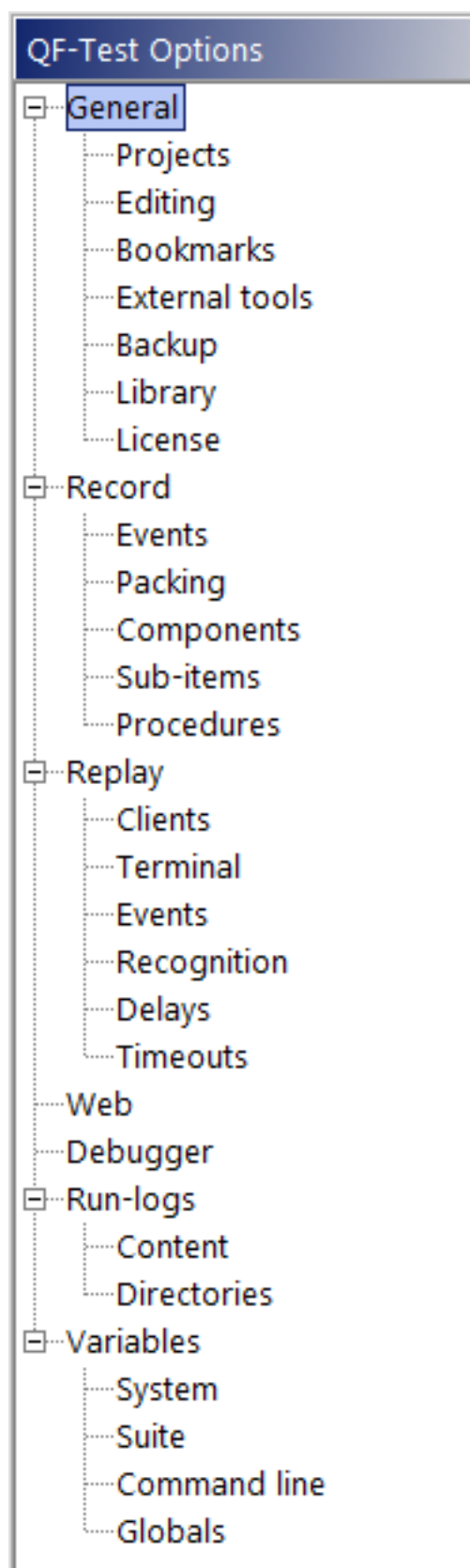


Figure 29.1: Options tree

To get at an option, first select the appropriate node of the tree. The options for that topic are then displayed in the right part of the view. When switching from one group to the other, the current values are verified but not adopted yet. This happens only after confirmation with the OK button.

29.1 General options

This is the node for general QF-Test settings.

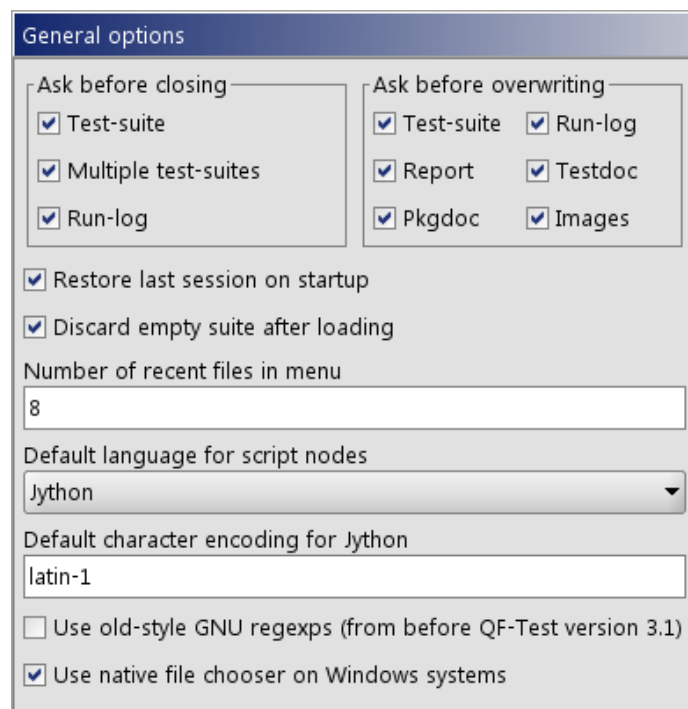


Figure 29.2: General options

Ask before closing (User)

When a test-suite or a test run-log has been modified, QF-Test asks whether it should be saved before it closes its main window. That query can be suppressed by turning off this option. Be warned that auto saving is not implemented yet, so you may lose data if you forget to save before closing.

Ask before overwriting (User)

When you try to save a test-suite or a run-log or generate a report, pgkdoc or testdoc or save the image of a Check image⁽⁴⁸⁸⁾ over an existing file or directory, QF-Test asks for confirmation unless you turn off the option for the respective type of file.

Restore last session on startup (User)

If this option is set and QF-Test is opened in the workbench view, the previous session is restored by loading previously opened test-suites and selecting the previously selected node in each suite. If one or more test-suites are specified on the command line, these are loaded in addition to the previous session and receive the initial focus on startup.

Discard empty suite after loading (User)

A common situation in daily work is that a test-suite is loaded right after starting QF-Test. In that case the initial empty test-suite is typically unwanted and even a burden. Setting this option will cause the initial suite to be closed automatically.

Number of recent files in menu (User)

The File menu offers quick access to recently used test-suites or run-logs. This option determines the maximum number of recent file entries in the menu.

Default script language for script nodes (User)

This option can be set to either "Jython" or "Groovy" and determines the default setting for the Script language⁽⁴¹⁸⁾ attribute of newly created Server script⁽⁴¹⁶⁾ or SUT script⁽⁴¹⁹⁾ nodes.

Default character encoding for Jython (System)

This option sets the default encoding for Jython scripts in Server script⁽⁴¹⁶⁾ and SUT script⁽⁴¹⁹⁾ nodes as well as the interactive Jython terminals for QF-Test and the SUT. The default encoding for standard Jython is 'ascii'. To ease migration and improve two-way-compatibility with older QF-Test versions, the default setting for QF-Test is 'latin-1'. You can also experiment with other settings that better suit your locale.

For background information please see <http://docs.python.org/library/codecs.html>.

This setting does not affect the encoding for Python modules in separate files. If you get an error that a module cannot be loaded because no encoding is declared, you probably need to add a line of the form

Note

```
# coding: latin-1
```

close to the top of that file. Please see <http://www.python.org/peps/pep-0263.html> for details.

Use old-style GNU regexps (from before QF-Test version 3.1) (System)

Server or SUT script name: OPT_USE_GNU_REGEX

Since version 3.1 QF-Test uses the standard Java regular expression syntax by default. In case this causes problems with regular expressions in older tests, you can switch back to using the GNU regexp package via this option. For further information about regular expressions please see [section 36.4^{\(614\)}](#).

Use native file chooser on Windows systems (User)

Server script name: OPT_USE_NATIVE_FILECHOOSER

On Windows systems the native file chooser is more advanced and more convenient to use than the Swing file chooser so QF-Test uses the native one by default. In case you prefer the Swing file chooser you can get it back by deactivating this option.

29.1.1 Projects

There are several options that influence the way QF-Test manages and displays projects.

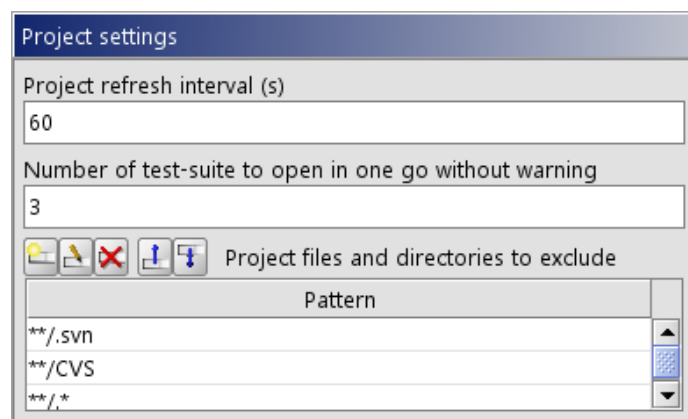


Figure 29.3: Projects

Project refresh interval (s) (User)

The interval at which a project automatically gets completely refreshed. You can refresh a directory at any time by selecting it and pressing **[F5]**. To refresh the complete hierarchy below the selected directory, press **[Shift-F5]** instead.

Number of test-suite to open in one go without warning (User)

From the project tree you can open all test-suites contained in one directory hierarchy in one go. If you accidentally select too many test-suites, QF-Test will first issue a warning with the number of test-suites, allowing you to cancel that action. This option determines the threshold for that warning.

Project files and directories to exclude (System)

In many cases a directory hierarchy holds files and directories that don't really belong to a project, most notably sub-directories created by version control systems like subversion or cvs. In this option you can specify patterns for files and directories to generally exclude from projects.

The patterns used here are not regular expressions but a simpler form often used by development tools: An `'*'` stands for any number of characters up to the next file separator - for compatibility reasons only forward `'/'` is used - while `'**'` means 0 or more characters of any kind, including `'/'`. Every pattern is relative to the root directory of the project. Some examples:

`**/.svn`

All directories named `.svn` at any depth.

`**/*.*`

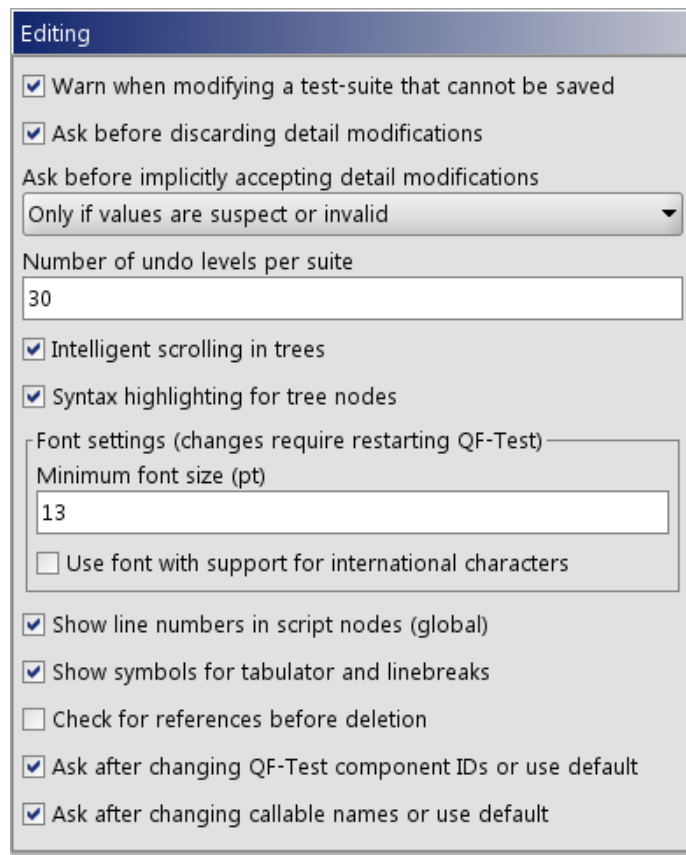
All directories starting with a `'.'` at any depth.

`deprecated`

A directory named `deprecated` directly below the project root.

29.1.2 Editing

These options are used to configure various settings regarding editing in the tree or detail view.



Editing

- ☒ Warn when modifying a test-suite that cannot be saved
- ☒ Ask before discarding detail modifications
- Ask before implicitly accepting detail modifications
 - Only if values are suspect or invalid
- Number of undo levels per suite
 - 30
- ☒ Intelligent scrolling in trees
- ☒ Syntax highlighting for tree nodes
- Font settings (changes require restarting QF-Test)
 - Minimum font size (pt)
 - 13
 - ☐ Use font with support for international characters
- ☒ Show line numbers in script nodes (global)
- ☒ Show symbols for tabulator and linebreaks
- ☐ Check for references before deletion
- ☒ Ask after changing QF-Test component IDs or use default
- ☒ Ask after changing callable names or use default

Figure 29.4: Editing

Warn when modifying a test-suite that cannot be saved (User)

If saving test-suites is prohibited, for example when working without a license, QF-Test will warn you that you will not be able to save your changes when a test-suite is modified for the first time. Deactivating this option suppresses that warning.

Ask before discarding detail modifications (User)

When you have started making changes to an existing or newly inserted node and then abort by pressing **Escape** or clicking the "Cancel" button, QF-Test asks for confirmation before discard your modifications. This dialog can be suppressed by disabling this option. In this case, please be aware that - especially in case of scripts - a lot of work may get lost in case of a mistake.

Ask before implicitly accepting detail modifications (User)

A very common mistake made while editing a test-suite is to forget pressing OK after making changes in the detail view of a node before switching to some other node, running a test, etc. If that happens QF-Test can either accept the modified values automatically or ask for confirmation by popping up a dialog with the detail view. The following options are available:

Always

Don't accept values implicitly, always ask for confirmation.

Only if values are suspect or invalid

Try to accept values implicitly as long as they are valid and not suspect. Currently "being suspect" is defined as having leading or trailing whitespace which can lead to subtle problems which are very hard to locate.

Never

Accept all valid values implicitly without asking for confirmation.

This option doesn't change the effect of explicitly discarding your modifications with the Cancel button or by pressing `Escape`.

Number of undo levels per suite (User)

This option lets you set the number of edits that can be undone in a test-suite or run-log.

Intelligent scrolling in trees (User)

The default methods for interacting with Swing trees are not ideal. Moving the selection around causes a log of unnecessary horizontal scrolling and Swing has the tendency to scroll trees to a position where little context is visible around the selected node.

Because tree navigation is essential for QF-Test, some of these methods are implemented differently to provide a more natural interface and to make sure that there is always enough context visible around the selected node. However, your mileage may vary, so if you don't like the alternative methods you can switch back to the default Swing way of things by deactivating this option.

Syntax highlighting for tree nodes (User)

This option controls activation of syntax highlighting for tree nodes within test-suites and run-logs. If active, specific text parts of nodes (e.g. node name,

parameters, client) are outlined in different colors and styles. This significantly improves readability.

Minimum font size (pt) (User)

This option lets you set the minimum font size (as point value) used within QF-Test. A change in this value becomes operative after restarting QF-Test.

Use font with support for international characters (User)

Normally QF-Test should use a font with international character support automatically, depending on your system and language settings. If this detection fails you may see "tofu" characters (small rectangles) instead of the real glyphs in nodes like Check text. In that case, try activating this option and restart QF-Test. If QF-Test still displays tofu it is a limitation of Java - the tests should work irrespective of that because the actual character values are retained.

Show line numbers in script nodes (User)

If this option is set, line numbers are shown in SUT script⁽⁴¹⁹⁾ and Server script⁽⁴¹⁶⁾ nodes.

Show symbols for tabulator and linebreaks (User)

If this option is set, QF-Test shows symbols for tabulator and linebreaks in tables and relevant textareas.

Check references before deletion (User)

If this option is set, QF-Test searches for references of nodes before nodes will be deleted. If references can be found, they will be shown in a dialog.

Ask after changing QF-Test component IDs or use default (User)

If this option is set, QF-Test asks whether the user wants to update the QF-Test component IDs of any referring node after the QF-Test ID of a component has been changed. If this option isn't set QF-Test updates all references in case of unique QF-Test component IDs.

Ask after changing callable names or use default (User)

If this option is set, QF-Test asks whether the user wants to update the callable names (i.e. procedures, packages, tests and dependencies) of any referring node

3.5+

3.4+

3.5.3+

3.5.3+

after the name of a callable node has been changed. If this option isn't set QF-Test updates all references in case of unique names.

29.1.3 Bookmarks

Here you can edit your bookmarks, a list of files and nodes that can be accessed quickly via the menu **File→Bookmarks**.

4.0+

Instead of a file you can also specify a directory. When the respective bookmark is selected, the file selection dialog is opened directly for this directory. The QF-Test ID for the node is ignored in this case.

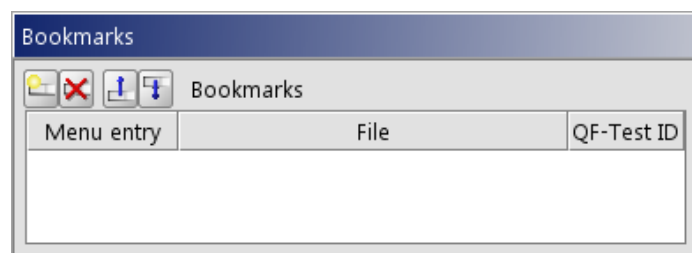


Figure 29.5: Bookmarks

Though you can also create new bookmarks manually, it is preferable to use the menu item **File→Add to bookmarks** to add a bookmark for a whole test-suite or run-log or to select **Add to bookmarks** in the context menu of a node in a test-suite to add a bookmark for this specific node.

29.1.4 External tools

The following options determine which external programs are called by QF-Test.

Figure 29.6: External tools options

External editor command (User)

Scripts can be edited in an external editor by pressing **Alt-Return** or by clicking



the button above the text area. The contents of the text area are then saved to a temporary file and the external editor is run to edit that file. It is recommended to define a name for the script before opening it in the external editor (see also Warn when running external editor without file name⁽²⁴⁸⁾). Otherwise a random number is chosen as file name, which makes it difficult to distinguish several scripts opened in the external editor.

Changes made to an external file are picked up automatically by QF-Test. Depending on your settings, you may get a warning message when this happens (see Warn when test-suite is changed by an external editor⁽²⁴⁸⁾). In case you are tempted to edit your script code parallel in the internal QF-Test editor: These changes are also saved in the temporary file. Editors like jEdit on their part are smart enough to detect the change and reload the file automatically.

This option determines the external editor command to use. There are two variants, the plain name of an executable file or a complex command including options. The latter is distinguished by the string `$(file)` which is the placeholder for the name of the temporary file. Additionally, `$(line)` may be used to pass the current line number to the editor as well.

Note

The `$(file)/$(line)` syntax is used simply to avoid yet another different convention for variable attributes. No standard QF-Test `$(...)` variable expansion is taking place.

Plain commands need never be quoted. Examples are:

- emacsclient
- notepad
- C:\Program Files\Crimson Editor\cedt.exe

Complex commands on the other hand may need to use quotes, especially on windows. QF-Test takes care of quoting the \$(file) argument itself:

- "C:\Program Files\eclipse-3.6\eclipse.exe" -launcher.openFile \$(file)
- javaw.exe -jar C:\Programs\jEdit4.2\jedit.jar -reuseview \$(file)
- "C:\Program Files\Crimson Editor\cedt.exe" \$(file)
- xterm -e vi +\$(line) \$(file)

If this option is left empty, the value of the environment variable EDITOR is used, if it is defined when QF-Test is started.

Warn when test-suite is changed by an external editor (User)

Display a warning message when changes to a script made by an external editor are picked up by QF-Test (see also External editor command⁽²⁴⁷⁾).

Warn when running external editor without file name (User)

Display a warning message when a script without name is opened in an external editor (see also External editor command⁽²⁴⁷⁾).

External imaging program (User)

The Image⁽⁴⁹¹⁾ of a Check image⁽⁴⁸⁸⁾ node can be edited in an external imaging program. The image is saved to a temporary PNG file and the external imaging program is run to edit that file. When finished editing, the file must be saved and the program exited. QF-Test will read the image back from the temporary file.

This option determines the program to use for the operation. There are two variants, the plain name of an executable file or a complex command including options. The latter is distinguished by the string \$(file) which is the placeholder for the name of the temporary file.

Note

The \$(file)/\$(line) syntax is used simply to avoid yet another different convention for variable attributes. No standard QF-Test \$(...) variable expansion is taking place.

Plain commands need never be quoted. Examples are:

- gimp
- mspaint
- C:\Windows\System32\mspaint.exe

Complex commands on the other hand may need to use quotes, especially on windows. QF-Test takes care of quoting the \$(file) argument itself:

- gimp -no-splash \$(file)
- "C:\Windows\System32\mspaint.exe" \$(file)

HTML browser (Unix only) (User)

On Unix systems this option allows you to set the HTML browser for the context sensitive help system. You can specify a complex command using '\$url' as placeholder for the URL to show, e.g.

```
netscape -remote openURL($url)
```

or just a simple command like

```
firefox
```

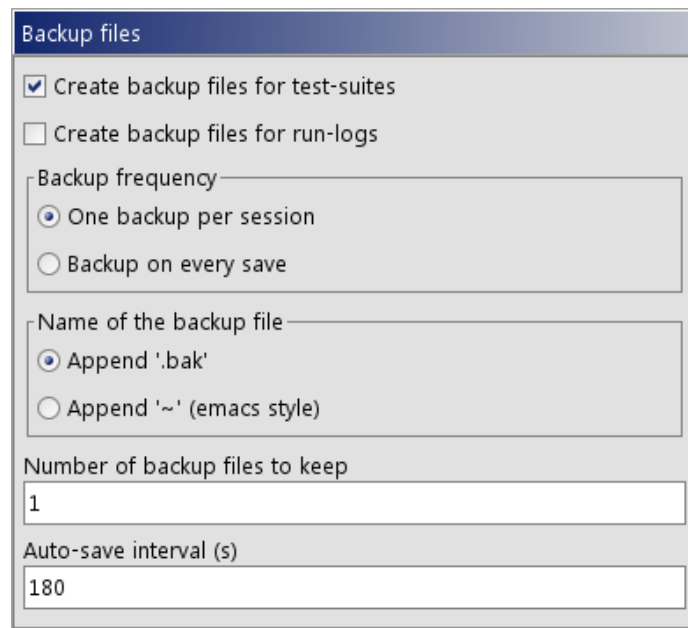
in which case the URL is passed as the last argument. On Windows the system browser is used.

PDF reader (Unix only) (User)

On Unix systems this option allows you to set the program used to display PDF files. This is necessary to open the PDF manual from the Help menu. On Windows the program associated with the .pdf extension is used.

29.1.5 Backup files

Unless told to do otherwise, QF-Test creates backups of existing files when saving test-suites or run-logs. These backup files are useful only in protecting against failures when saving a file. They are by no means a replacement for proper system backups. The following options determine if, when and how backup files are created.



Backup files

☒ Create backup files for test-suites

☐ Create backup files for run-logs

Backup frequency

☒ One backup per session

☐ Backup on every save

Name of the backup file

☒ Append '.bak'

☐ Append '~' (emacs style)

Number of backup files to keep

1

Auto-save interval (s)

180

Figure 29.7: Backup file options

Create backup files for test-suites (User)

Backup files for test-suites are created only if this option is activated. Please be careful and don't turn it off without a good reason, such as using a version control system for test-suites which obviates the need to create backups. Just think about the amount of work that goes into creating a useful test-suite and imagine the frustration if it gets destroyed accidentally.

Create backup files for test run-logs (User)

Usually a run-log is far less valuable than a test-suite, so there is a separate option that determines whether backups are created for run-logs.

Backup frequency (User)

There are two possibilities for the frequency with which backup files are created. With the first option, "One backup per session", QF-Test creates a backup file only the first time a file is saved. If you continue editing the suite or run-log and save it again, the backup file is left unchanged. Only when you edit a different file or restart QF-Test, a new backup is created. This setting is useful if you keep only one backup per test-suite.

If, on the other hand, you keep multiple backups per suite, "Backup on every save" may be the preferred choice.

Name of the backup file (User)

Like many other things, the conventions for the names of backup files differ between Unix and Windows. While the common extension for a backup file under Windows is `.bak`, there are many variants under Unix. One of the most common is appending a tilde character `'~'`.

Number of backup files to keep (User)

You can keep more than one backup file for each test-suite or run-log. If you do so, backup files are named after the scheme `.bak1`, `.bak2...` for the `.bak` naming style and `~1~`, `~2~...` for the other. The most recent backup is always numbered 1. When a new backup is created, the old number 1 is renamed to 2, 2 renamed to 3 and so on. When the maximum is reached, the oldest files are deleted.

Auto-save interval (s) (User)

Interval after which a modified test-suite is saved automatically. Setting this value to 0 will disable auto-saving. Otherwise values less than about 20 seconds are not useful. Run-logs are never saved automatically. Auto-save files are created in the same directory as the test-suite or - in the case of new suites that have never been saved - in the directory `.qftest` under the user's home directory.

29.1.6 Library path

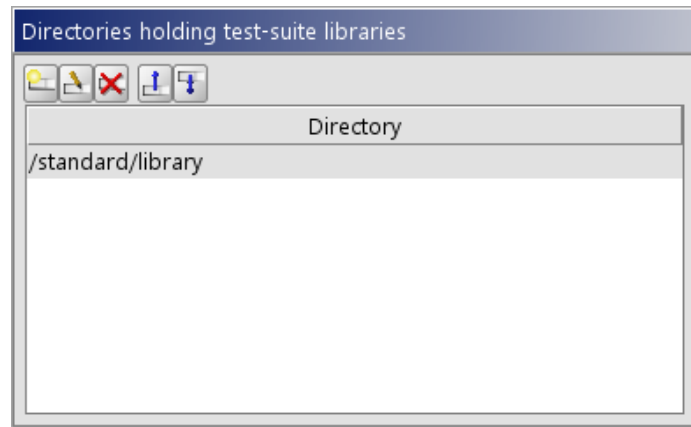


Figure 29.8: Library path option

Directories holding test-suite libraries (System)

This is a list of directories that are searched for test-suites whenever a suite reference is given as a relative path that cannot be resolved relative to the current suite. This includes direct suite references in the Procedure name⁽³⁸²⁾ attribute of a Procedure call⁽³⁸¹⁾ or a Component⁽⁵⁴⁵⁾ QF-Test ID⁽⁵⁴⁶⁾ reference as well as suites included through the Include files⁽³¹²⁾ attribute of the Test-suite⁽³¹¹⁾ node.

The `include` directory belonging to the current version of QF-Test is automatically and invisibly placed at the end of the library path. This ensures that the common library `qfs.qft` can always be included without knowing its actual location and that its version is matching the version of QF-Test at all times.

Note

If the command line argument `-libpath <path>`⁽⁵⁷⁷⁾ is given it will override the settings of this option. In interactive mode, the value of the command line argument is displayed here, but it will not be saved with the system configuration unless it is modified.

29.1.7 License

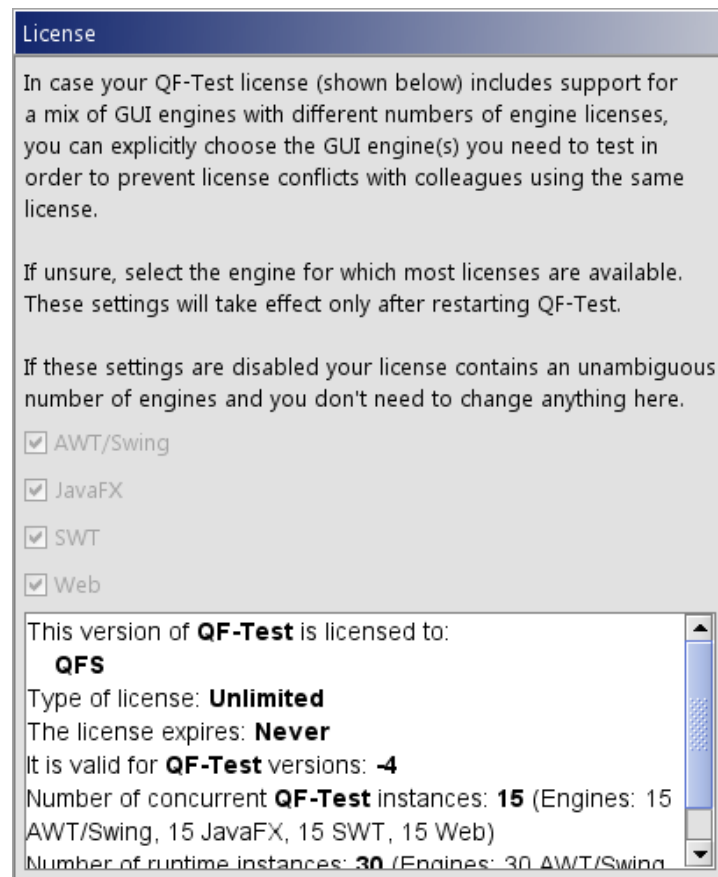


Figure 29.9: License options

Normally QF-Test license bundles contain a homogenous mix of GUI engines. For example, a bundle of QF-Test/swing licenses only supports the AWT/Swing engine, QF-Test/suite licenses support both AWT/Swing and SWT for all included licenses. For these kinds of simple licenses these license settings can be ignored.

A small problem arises in case of mixed engine licenses where some GUI engine is included only for a part of the licenses. An example for this is a license bundle that was formerly purchased for qftestJUI, upgraded to QF-Test/suite with QF-Test 2.0 and then extended with further QF-Test/swing licenses, say two licenses for QF-Test/suite and another two for QF-Test/swing. Such a license allows running four concurrent instances of QF-Test, but only two of these can make use of the SWT engine. If more than two instances are started with SWT support there will be a license conflict.

When QF-Test detects such a mixed license bundle for the first time it asks the user which engine licenses to use. The choice made then can be changed here at any

time. Besides, QF-Test can be started with the command line argument `-engine <engine>`⁽⁵⁷⁵⁾ to override the supported GUI engines for this execution.

29.2 Recording options

The following options determine which kinds of events are recorded and which filters are applied, how components are recorded and arranged, and how sub-items are handled.

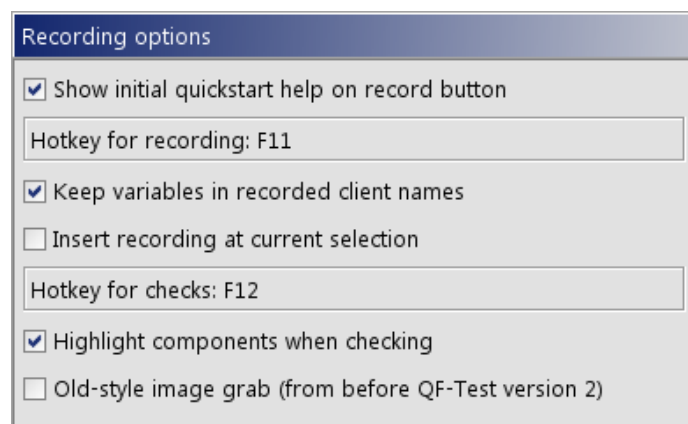


Figure 29.10: Recording options

Show initial quickstart help on record button (User)

Controls the display of an initial question mark on the record button in order to directly lead new users to the quickstart wizard.

Hotkey for recording (User)

SUT script name: `OPT_RECORD_HOTKEY`

Event recording can be directly started/stopped by pressing a key in the SUT. The key to be used for this function is set through this option. The default key is **F11**.

Keep variables in recorded client names (System)

This option is very useful if the client name assigned to your SUT contains variables (e.g. `$(client)`), which generally makes sense when creating procedures. If this option is set, the client attribute of all recorded nodes is set to the unexpanded `Client`⁽⁴²⁶⁾ value of the `Start SUT client`⁽⁴²⁶⁾ node through which the SUT was started.

Insert recording at current selection (User)

Depending on what you are currently working on, it may or may not make sense to add newly recorded sequences directly at the current insertion mark. If you deactivate this option, new recordings are placed in a new Sequence⁽³³⁴⁾ under Extras⁽³⁴²⁾.

Hotkey for checks (User)

SUT script name: OPT_RECORD_CHECK_HOTKEY

To simplify recording a sequence of events with interspersed checks, you can switch between plain recording and recording checks by pressing a key in the SUT. The key that triggers this switch is set through this option. The default key is **F12**.

Highlight components when checking (User)

SUT script name: OPT_RECORD_CHECK_HIGHLIGHT

When QF-Test is recording checks, it can give visual feedback on the component the mouse is currently over by inverting its foreground and background colors. Rarely this may have unwanted visual side effects, so you can turn that feature off with this option.

Old style image grab (from before version 2) (System)

SUT script name: OPT_RECORD_CHECK_IMAGE_OLD_STYLE

When recording images for Check image⁽⁴⁸⁸⁾ nodes, non-opaque components used to be recorded with a black background. Thus, the image could deviate from what the user was seeing. This error has been corrected and normally the background is now recorded correctly. If this option is activated, the old, broken mechanism is used instead which may be useful if many checks for non-opaque components have already been recorded using that method.

29.2.1 Events to record

These options specify which kinds of events are recorded and which aren't. You should not tinker with these unless you know what you are doing.

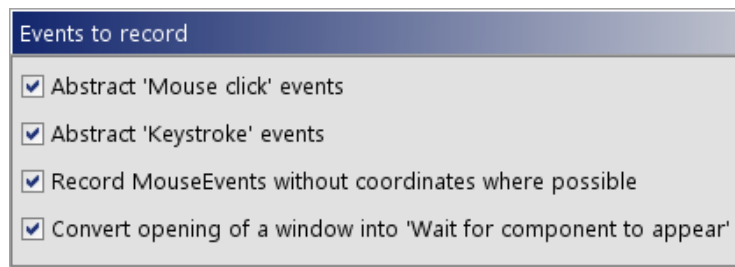


Figure 29.11: Options for events to record

Abstract 'Mouse click' events (System)

Activating this option causes a sequence of `MOUSE_MOVED`, `MOUSE_PRESSED`, `MOUSE_RELEASED` and `MOUSE_CLICKED` events to be recorded as a single 'Mouse click' pseudo event (see [section 30.8.1^{\(447\)}](#)).

Abstract 'Keystroke' events (System)

This option lets you record a sequence of `KEY_PRESSED`, `KEY_TYPED` and `KEY_RELEASED` events (or just `KEY_PRESSED` and `KEY_RELEASED` for function keys) as a single 'Keystroke' pseudo event (see [section 30.8.2^{\(450\)}](#)).

Record MouseEvents without coordinates where possible (System)

SUT script name: `OPT_RECORD_REPOSITION_MOUSE_EVENTS`

For many types of components and sub-items it doesn't matter where exactly a `MouseEvent` occurs. However, if large values are recorded for the X or Y coordinate of a [Mouse event^{\(447\)}](#), there's the danger that the event might miss its target upon replay if the component has shrunk a little due to font changes or because the window has been resized. This is also a possible source for problems when converting a recorded sequence to a [Procedure^{\(378\)}](#) with variable target components.

If this option is activated, QF-Test ignores the coordinates of recorded `MouseEvents` if it thinks that the coordinates don't matter for the target component, e.g. for all kinds of buttons, for menu items, table cells, list items and tree nodes. For the latter QF-Test distinguishes between clicks on the node itself and on the expand/collapse toggle. When `MouseEvents` without coordinates are played back, QF-Test targets the center of the respective component or item except that the X coordinate for items is limited to 5 because item bounds cannot always be calculated correctly.

Convert opening of a window into Wait for component to appear⁽⁵¹²⁾ (System)

When replaying a sequence during which a new window is opened, it may be useful to allow for a longer than usual delay until the window is opened. By activating this option, a recorded `WINDOW_OPENED` event will be turned into a Wait for component to appear⁽⁵¹²⁾ node automatically.

Web

For web clients this option causes a Wait for document to load⁽⁵¹⁶⁾ node to be inserted whenever loading of a document completes. This is important for proper synchronization when navigating to another page.

29.2.2 Events to pack

In order to keep the amount of raw event data generated during normal use of a Java GUI manageable, QF-Test employs a set of recording filters and packers. These do their best to keep everything needed for successful replay and throw away the rest. To get an impression of the actual data behind a recorded sequence, try recording a short sequence with all of these turned off, and with 'Mouse click'⁽²⁵⁶⁾ and 'Keystroke'⁽²⁵⁶⁾ pseudo events disabled.

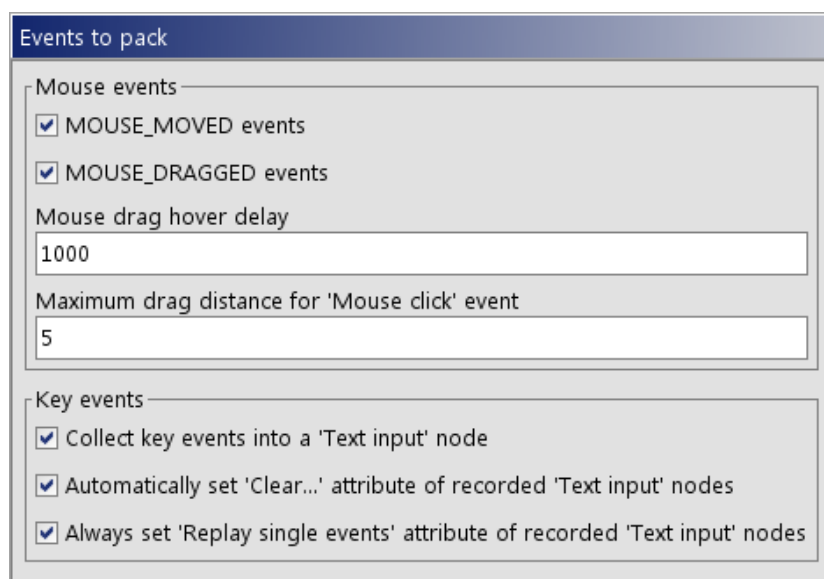


Figure 29.12: Options for events to pack

MOUSE_MOVED events (System)

SUT script name: OPT_RECORD_PACK_MOUSE_MOVED

`MOUSE_MOVED` events are especially frequent. Every mouse cursor motion generates a handful of these. Under most circumstances, only the last of a consecutive series of `MOUSE_MOVED` events is actually useful, so all events except the last one are dropped, if this option is activated. An example where this is not advisable is recording some freehand drawing in a graphics application.

Note

One might think that `MOUSE_MOVED` events are completely useless, since `MOUSE_PRESSED` or `MOUSE_RELEASED` events have their own set of coordinates, but this is not the case. Some Java components require a `MOUSE_MOVED` event before a `MOUSE_PRESSED` event is recognized.

MOUSE_DRAGGED events (System)

SUT script name: OPT_RECORD_PACK_MOUSE_DRAGGED

`MOUSE_DRAGGED` events are like `MOUSE_MOVED` events, but with one mouse button held down. Similarly, only the last of a series of `MOUSE_DRAGGED` events is recorded unless you turn off this option, except for special cases (see next option).

Mouse drag hover delay (System)

SUT script name: OPT_RECORD_MOUSE_DRAGGED_HOVER

There are situations where not only the final target of a mouse drag is of interest, but intermediate points as well. The most common is invoking a menu item in a sub-menu.

Note

As of QF-Test 1.05.2, the following no longer applies because `MOUSE_MOVED` or `MOUSE_DRAGGED` events that are required for opening sub-menus are not "optimized away" anymore. However, there may be other situations where intermediate stops are useful when recording drags.

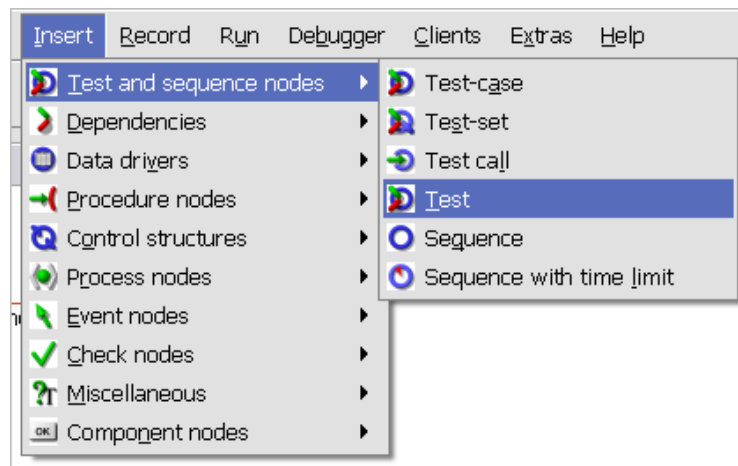


Figure 29.13: Dragging to a sub-menu

As illustrated above, creating a new Test node for a suite could be done by clicking on the **Insert** menu button, dragging the mouse to the **Test and Sequence nodes** item for the sub-menu, so the sub-menu pops up, then dragging on to the **Test** menu item and releasing the mouse button. Normally such a sequence would be reduced to a press, a drag to the final **Test** item and a release. It would fail to replay, since the sub-menu would never be popped up. To work correctly, an additional drag to the **Test and Sequence nodes** item must be recorded.

For that reason QF-Test recognizes a `MOUSE_DRAGGED` event as important if you hover over an intermediate component for a while during the drag. The delay (in milliseconds) needed to recognize such an event is set through this option.

To record the above example correctly with this option set to 1000, you'd have to click on the **Insert** menu button, drag to the **Sequences** item and keep the mouse pointer stationary for one second, then move on to the **Test** sub-menu item and release the mouse button.

Maximum drag distance for 'Mouse click' event (System)

It sometimes happens unintentionally that the mouse cursor is moved between pressing the mouse button and releasing it. This movement may be registered as a `MOUSE_DRAGGED` event, depending on the JDK version and the distance of the move. QF-Test is able to compensate for small movements and still convert the click into an abstract 'Mouse click' event. This option defines the maximum distance between pressing and releasing the mouse button that QF-Test will ignore. Every `MOUSE_DRAGGED` event above that distance will be left unchanged.

Collect key events into a Text input node (System)

Another example where a lot of events are generated is entering a short string of text into a text field. Each character typed leads to at least one `KEY_PRESSED`, one `KEY_TYPED` and one `KEY_RELEASED` event. For additional fun, the `KEY_RELEASED` events may arrive out of order or not at all, depending on operating system and JDK version.

If this option is activated, sequences of `KeyEvents` on a text component (to be exact: a component whose class is derived from `java.awt.TextField` or `javax.swing.text.JTextField`) are converted into a `Text input`⁽⁴⁵⁴⁾ node. Only true character input is packed, function or control keys or key combinations with `Control` or `Alt` are left unchanged.

When the packed sequence is replayed, only `KEY_TYPED` events are generated. `KEY_PRESSED` and `KEY_RELEASED` events cannot be generated, since the required key code is system dependent and cannot be determined from the character alone. This is not a problem however, since text components usually handle only `KEY_TYPED` events and some special keys.

Automatically set 'Clear...' attribute of recorded Text input nodes (System)

This option determines the value of the `Clear target component first`⁽⁴⁵⁶⁾ attribute of a recorded `Text input`⁽⁴⁵⁴⁾ node. If the option is not set, the attribute will not be set either. Otherwise, the `Clear target component first` attribute is set if and only if the text field or text area was empty before the input started.

Always set 'Replay single events' attribute of recorded Text input nodes (System)

This option determines the value of the `Replay single events`⁽⁴⁵⁶⁾ attribute of a recorded `Text input`⁽⁴⁵⁴⁾ node. If the option is set, the attribute will be set and vice versa. The conservative way is to keep the option set, but for a typical application that does not add its own `KeyListener`s to text fields it should be safe to turn it off so as to speed up replay of `Text input` nodes.

29.2.3 Components

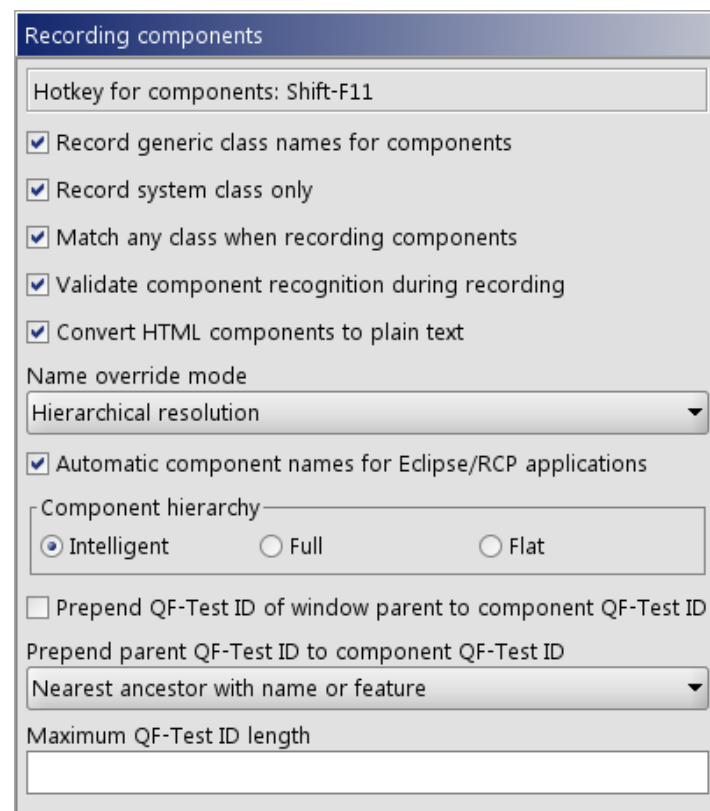


Figure 29.14: Options for recording components

Hotkey for components (User)

SUT script name: OPT_RECORD_COMPONENT_HOTKEY

This option defines the key for a very useful functionality: recording components directly from the SUT. Pressing this key in the SUT will switch it to "component recording" mode, regardless of whether event recording is currently activated or not. The default key is **Shift-F11**.

In this mode, clicking on a component with the mouse will cause the component to be recorded and added to the test-suite if it was unknown before. If more than one test-suite is currently opened, the menu item **Record→Receive components** determines the suite that will receive the components. The **QF-Test ID⁽⁵⁴⁶⁾** of the **Component⁽⁵⁴⁵⁾** is put on the clipboard so it can be pasted into any text field with **Control-V**. This latter feature is very handy when creating an event or a check from scratch.

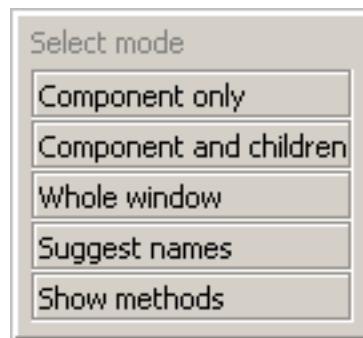


Figure 29.15: Popup menu for recording components

You can also record a whole component hierarchy at once by clicking with the right mouse button instead. This will bring up a popup menu with the following four options:

Component only

This is similar to clicking with the left mouse button. Only the selected component is recorded.

Component and children

The selected component and all components contained therein are recorded.

Whole window

Records every component in the whole window.

Suggest names

This is a special feature to improve the collaboration between testers and developers in deciding which components should have names set with `setName()`. All components in the whole window are recorded and put into a test-suite of their own. Each unnamed component for which a name will improve testability is marked with a name of the form "SUGGESTED NAME (n): suggestion". The running count in braces is just used to avoid duplicates. The suggested name is built from the component's class and other available information. It should be taken with a grain of salt.

Show methods

This is another special feature that brings up a component inspector window showing the attributes and methods of the selected component's class. See [section 5.5^{\(44\)}](#) for further information.

Normally "component recording" mode is turned off by either pressing the hotkey again or by selecting a component. If you want to record multiple single components, hold down an additional modifier key when switching to "component recording" mode. That way selecting a component will not turn off the mode, only pressing the hotkey again will.

Record generic class names for components (System)

4.0+

SUT script name: OPT_RECORD_COMPONENT_GENERIC_CLASS

Where possible QF-Test assigns generic class names to components like "Button", "Table" or "Tree" in addition to the actual Java, DOM or AJAX specific class names like "javax.swing.JButton", "javafx.scene.control.Button", "INPUT" or "X-BUTTON". These generic class names are more descriptive and robust, improve compatibility between different UIs and enable creation of generic utility procedures. Generic class names can be used for component recognition or registering resolvers. If this option is active, generic class names are recorded where available.

Record system class only (System)

SUT script name: OPT_RECORD_COMPONENT_SYSTEM_CLASS_ONLY

If this option is set, QF-Test does not record any custom classes for Components⁽⁵⁴⁵⁾. Instead it moves up the class hierarchy until it encounters a system class and records that. Set this option if the class names of your custom GUI classes tend to differ between releases.

Note

You must activate this option if you intend to obfuscate the jar files of your application or if you are using a custom class loader to load your application's GUI classes.

Web

This option does not apply to web SUTs.

Match any class when recording components (System)

4.0+

SUT script name: OPT_RECORD_TOLERANT_CLASS_MATCH

For compatibility with older QF-Test versions that did not have generic classes, QF-Test now matches against several classes of a component when recording, the concrete class, the generic class and the system class. This is very useful if you want to retain as many of your old components as possible. If you would rather get new components based on generic classes in new recordings you should deactivate this option. Components recorded for the first time will always be recorded with the class determined by the preceding two options Record generic class names for components⁽²⁶³⁾ and Record system class only⁽²⁶³⁾.

Validate component recognition during recording (System)

3.5+

SUT script name: OPT_VALIDATE_RECORDED_COMPONENTS

In case non-unique names are assigned to components QF-Test can still distinguish between these components with the help of the Extra feature⁽⁵⁴⁷⁾ `qfs:matchindex` that specifies the index of the component with the given name. If this option is set, QF-Test will check the name of the component during recording and try to assign `qfs:matchindex` correctly.

Note

You should only deactivate this option if you are sure that component names are "reasonably unique" and the component validation significantly impacts performance during recording.

Convert HTML components to plain text (System)

Swing

SUT script name: OPT_RECORD_COMPONENT_CONVERT_HTML

Swing support HTML markup in various kinds of labels, buttons and sub-elements of complex components. For component identification and validation, the HTML markup is often not useful and will clutter up things. If this option is set, QF-Test converts HTML to normal text by removing all HTML markup so only the actual text content is left.

Name override mode (record) (System)

Server or SUT script name: OPT_RECORD_COMPONENT_NAME_OVERRIDE

Possible Values: VAL_NAME_OVERRIDE_EVERYTHING,
VAL_NAME_OVERRIDE_HIERARCHY,
VAL_NAME_OVERRIDE_PLAIN

Note

There are two versions of this option which are closely related. This one is effective during recording, the other one⁽²⁸⁵⁾ during replay. Obviously, both options should always have the same value. There's one exception though: When migrating from one setting to another, QF-Test's components have to be updated. During that process, keep the replay option at the old setting and change this option to the new one. Be sure to update the replay setting after updating the components.

This option determines the weight given to the names of components for recording. Possible choices are:

Override everything

This is the most effective and adaptable way of searching components, but it requires that the names of the components are unique, at least within the same window. If that uniqueness is given, use this choice.

Web

Don't use this value for a web page with frames. Use "Hierarchical resolution"

instead.

Hierarchical resolution

This choice should be used if component names are not unique on a per-window basis, but naming is still used consistently so that two components with identical names have at least parent components or ancestors with distinct names. That way, component recognition is still tolerant to a lot of change, but if a named component is moved to a different named parent in the SUT, the test-suite will have to be updated to reflect the change.

Plain attribute

If there are components with identical names in the SUT within the same parent component you must use this setting. The name will still play an important role in component recognition, but not much more than the Feature⁽⁵⁴⁷⁾ attribute.

Automatic component names for Eclipse/RCP applications (System)

SUT script name: `OPT_RECORD_COMPONENT_AUTOMATIC_RCP_NAMES`
Eclipse and applications based on the Rich Client Platform (RCP) have a complex GUI with support for changing perspectives. Such a change causes components to be rearranged which can make it hard for QF-Test to recognize them unless names are set at least on the major components. This is further complicated by the fact that the structure of the components is not what it appears to be - the main components are all arranged in a relatively flat hierarchy within the workbench. On the upside, RCP based applications have a uniform inner structure based on `Views` and `Editors`, many of which are named.

If this option is turned on, QF-Test will do its best to automatically associate GUI elements with their RCP counterparts and assign names based on that association. This can drastically improve component recognition for such applications. However, if some names thus assigned turn out not to be reliable over time, they can also interfere. In such a case, names can be assigned to the affected components either using `setData` as described in [chapter 5](#)⁽³⁷⁾ or with the help of a `NameResolver` as described in [section 39.1](#)⁽⁶⁷⁸⁾. Both will override automatically generated names.

Component hierarchy (System)

SUT script name: `OPT_RECORD_COMPONENT_HIERARCHY`

Possible Values: `VAL_RECORD_HIERARCHY_INTELLIGENT`,
`VAL_RECORD_HIERARCHY_FULL`,
`VAL_RECORD_HIERARCHY_FLAT`

QF-Test supports three different kinds of views for the components of the SUT. For more information about their effect on component recognition, see [section 36.2](#)⁽⁶¹²⁾.

The flat view collects all components of a window as direct child nodes of the respective Window⁽⁵³⁵⁾ node. The advantage of this view is that structural changes of the component hierarchy have little effect on component recognition. This is also its greatest disadvantage: since structural information is not available, this view gives reasonable recognition quality only if `setName()` is used ubiquitously. Another drawback is the lack of clearness.

The complement to the flat view is the full hierarchy. It includes every single component of the SUT's GUI, emulating all parent/child relationships. This view can be a useful tool for developers or testers that want to gain insights into the SUT's structure, but is not very applicable for testing, since structural changes affect it too much. As long as you don't change the GUI however, it will give you excellent recognition without the help of `setName()`.

A compromise between flat and full hierarchy is available through the choice "Intelligent". For this view only the "interesting" components of the SUT are recorded. "Interesting" in this case means that either the user can interact with the component, or it is located at some important point in the hierarchy, like the children of a split pane or a tabbed pane. In some later version of QF-Test this decision may be made configurable as well.

Prepend QF-Test ID of window parent to component QF-Test ID (System)

Server script name: `OPT_RECORD_COMPONENT_PREPEND_WINDOW_ID`
If selected, QF-Test prepends the QF-Test ID of the Window⁽⁵³⁵⁾ parent of a Component⁽⁵⁴⁵⁾ to its QF-Test ID⁽⁵⁴⁶⁾ during recording. This is useful to disambiguate QF-Test IDs of components with identical names in different windows.

Prepend parent QF-Test ID to component QF-Test ID (System)

Server script name: `OPT_RECORD_COMPONENT_PREPEND_PARENT_ID`
Possible Values: `VAL_RECORD_COMPONENT_PREPEND_PARENT_ALWAYS`,
`VAL_RECORD_COMPONENT_PREPEND_PARENT_NAMED`,
`VAL_RECORD_COMPONENT_PREPEND_PARENT_FEATURE`,
`VAL_RECORD_COMPONENT_PREPEND_PARENT_NEVER`

When a Component⁽⁵⁴⁵⁾ is recorded for the first time, QF-Test assigns an automatically generated QF-Test ID⁽⁵⁴⁶⁾. The QF-Test ID of a direct or indirect parent node may be prepended to this QF-Test ID. This is useful to distinguish between similar components that don't have a name of their own.

Example: Imagine two `JScrollPane`s, one named "TreeScrollPane" and the other named "DetailScrollPane". Without this functionality, their vertical scrollbars would get the QF-Test IDs "scrollbarVertical" and "scrollbarVertical2". With this function turned on, the IDs would be "TreeScrollPane.scrollbarVertical" and "De-

tailScrollPane".scrollbarVertical". That way it is immediately obvious which component is the target of an event.

There are four possible settings:

- "Never" turns this option off.
- "Nearest named ancestor" is a useful setting, if your developers have assigned names to all major components with the Java method `setName`. A component that doesn't have a name of its own, gets the QF-Test ID of its nearest named ancestor node prepended.
- If `setName` is used sparingly or not at all, it is better to set this option to "Nearest ancestor with name or feature". That way either the name or a distinctive feature of an ancestor node will be applicable.
- "Always" is only useful if the option Component hierarchy⁽²⁶⁵⁾ is set to "Flat". With this setting, every component gets the QF-Test ID of its parent node prepended, which can lead to unusably long QF-Test IDs when components are nested deeply.

Maximum QF-Test ID length (System)

If set, this option limits the length of automatically generated IDs. This can improve clearness since QF-Test IDs can get very long, e.g. when generated from a component's feature.

29.2.4 Recording sub-items

Events on complex components like tables or trees can be recorded relative to a sub-item of the component. For further information about sub-items see the Item⁽⁵⁵¹⁾ node or section 6.3⁽⁵⁴⁾.

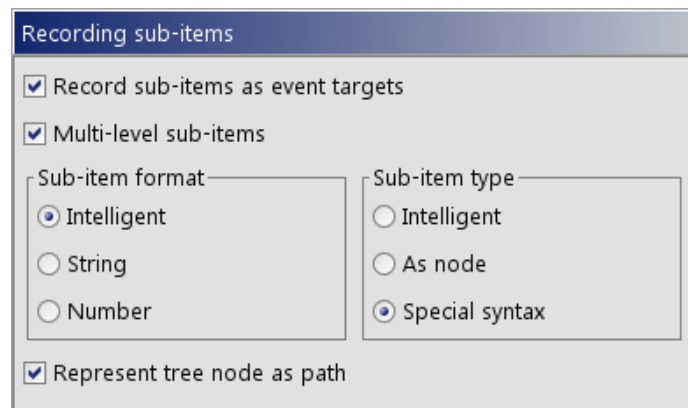


Figure 29.16: Options for recording sub-items

Record sub-items as event targets (System)

SUT script name: OPT_RECORD_SUBITEM

This option activates recording sub-items. When turned off, events on complex components are no different from events on simple components.

Multi-level sub-items (System)

Server or SUT script name: OPT_RECORD_SUBITEM_MULTILEVEL

Via this option you can completely disable multi-level sub-items (even for replay). However, you should only turn this feature off in case you are running into problems caused by test-suites that contain unquoted special characters like '@' or '%' in textual sub-item indexes. Even then it is preferable to update the test-suites with properly quoted items, possibly using the special variable syntax `${quoteitem:...}` (see [section 7.5^{\(62\)}](#)).

Sub-item format (System)

SUT script name: OPT_RECORD_SUBITEM_FORMAT

Possible Values: VAL_RECORD_SUBITEM_FORMAT_INTELLIGENT,
VAL_RECORD_SUBITEM_FORMAT_TEXT,
VAL_RECORD_SUBITEM_FORMAT_NUMBER

When recording an event for a sub-item, the Item's⁽⁵⁵¹⁾ index can be defined As string⁽⁵⁵³⁾ or As number⁽⁵⁵³⁾.

The third choice, "Intelligent", causes QF-Test to record the index in the format most appropriate for the item. If the name of the item is unique within the complex component, a string index is recorded, a numeric index otherwise.

Sub-item type (System)

Server script name: OPT_RECORD_SUBITEM_TYPE

Possible Values: VAL_RECORD_SUBITEM_TYPE_INTELLIGENT,
VAL_RECORD_SUBITEM_TYPE_NODE,
VAL_RECORD_SUBITEM_TYPE_SYNTAX

As explained in [section 6.3^{\(54\)}](#), there are two ways to address a sub-item: either through a special syntax or by creating an [Item^{\(551\)}](#) node. This option determines the method to use in recorded events.

Choosing "Intelligent" will cause an [Item^{\(551\)}](#) node to be created if the index of the sub-item is a string and if the sub-item itself is not editable. Otherwise the special syntax is used.

Represent tree node as path (System)

SUT script name: OPT_RECORD_SUBITEM_TREE_PATH

It is not uncommon that trees have identically named nodes under different parent nodes, e.g. a file system with the directories `/tmp` and `/usr/tmp`. By using a path format in the [Items^{\(551\)}](#) for tree nodes QF-Test can make full use of the hierarchical structure to distinguish between these nodes. The slash character `'/'` is used as separator.

If this option is deactivated, trees will be treated as flat lists.

29.2.5 Recording procedures

The following options determine the configuration of the Procedure Builder which is described in detail in [Automated Creation of Basic Procedures^{\(159\)}](#).

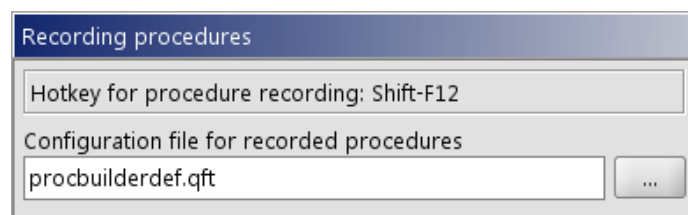


Figure 29.17: Procedure Builder options

Hotkey for procedure recording (User)

SUT script name: OPT_RECORD_PROCEDURE_HOTKEY

This option defines a key for turning on automatic procedure recording directly in the SUT. The default key is **Shift-F12**.

Configuration file for recorded procedures (System)

Here you can specify your own template file for the Procedure Builder. If a relative path is given, QF-Test looks for the definition file in the directory that QF-Test was started from and in the default include directory.

29.3 Replay options

The following settings change the way test-suites are executed.

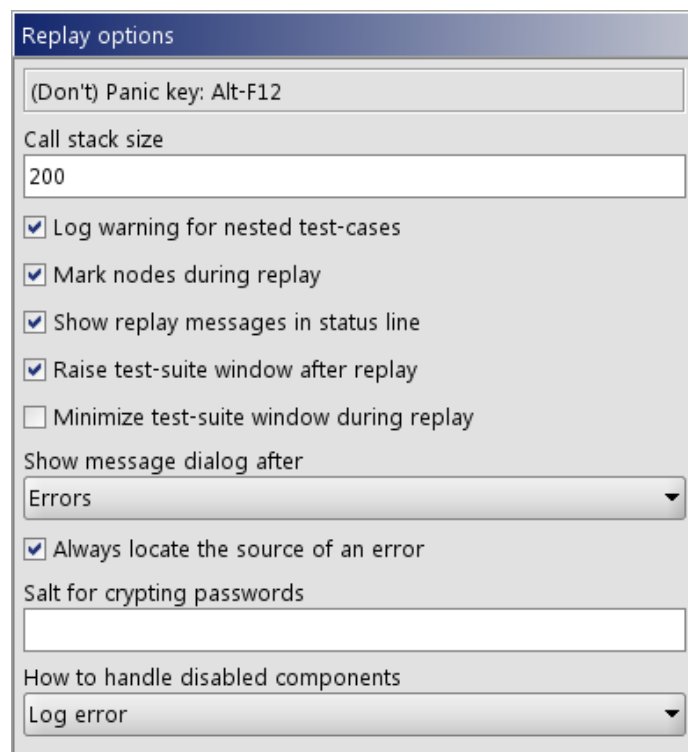


Figure 29.18: Replay options

(Don't) Panic key (User)

Server or SUT script name: OPT_PLAY_DONT_PANIC_HOTKEY

When running a test at full speed it can be rather difficult to get the focus to QF-Test's window and interrupt the test so you can do something different without having all these windows flashing around the screen. This is all the more true when the options Actually move mouse cursor⁽²⁸¹⁾ or Raise SUT windows automatically⁽²⁸⁰⁾ are activated or when running in batch mode.

This option lets you define a key combination (the default being Alt-F12) that will instantly pause all running tests if it is pressed in any SUT or QF-Test window (unless multiple QF-Test instances are run simultaneously, of course). Pressing the same key combination again will resume all tests, unless you manually resume or stop any of them. In that case its effect is automatically reset to suspend tests.

Call stack size (System)

Server script name: OPT_PLAY_CALLSTACK_SIZE

The call stack size is a limit for the nesting depth of Sequences⁽³³⁴⁾ or Procedure calls⁽³⁸¹⁾ during replay. This limit is needed to detect and handle endless recursion. When the nesting depth exceeds the call stack size, a StackOverflowException⁽⁵⁶⁴⁾ is thrown. The default value of 200 should be sufficient but can be increased for very complex tests.

Log warning for nested test-cases (System)

Server script name: OPT_PLAY_WARN_NESTED_TEST_CASE

Execution of Test-case⁽³¹⁴⁾ nodes should not be nested because such Test-cases cannot be listed properly in the report. If this option is active, a warning is logged in case a Test-case is executed within another Test-case.

Mark nodes during replay (User)

If set, tree nodes that are currently executed are marked with an arrow.

Show replay messages in status line (User)

Determines whether the name of the currently executing node is shown in the status line.

Raise test-suite window after replay (User)

This option is mainly used together with the option Raise SUT windows automatically⁽²⁸⁰⁾. It causes the window of a test-suite to be raised after a test run.

See also option Force window to the top when raising⁽²⁸⁰⁾.

Minimize test-suite window during replay (User)

If this option is set, QF-Test will minimize the window of a test-suite while its tests are being executed. The window will pop back up automatically when the test is stopped or suspended. This feature is especially useful on Windows 2000/XP systems where programs are prohibited from bringing their windows to the top so QF-Test cannot raise the windows of the SUT.

Show message dialog after (User)

After replay is finished, the status line shows the number of errors and warnings that occurred. If an uncaught exception was thrown, an error dialog is displayed. Additionally, a message dialog can be displayed in case of warnings or errors. This options sets the minimum error level that triggers such a message dialog.

Always locate the source of an error (User)

When an exception is thrown during replay, the node that caused the exception will be made visible and selected. If you don't like this, you can turn this feature off and locate the node via the Run→Find last error source... menu item instead.

Salt for crypting passwords (System)

QF-Test can store encrypted passwords in the Text⁽⁴⁵⁶⁾ attribute of a Text input⁽⁴⁵⁴⁾ node for a password field or the Detail⁽⁴⁶⁵⁾ attribute of a Selection⁽⁴⁶²⁾ used for a login dialog in a web SUT. When such passwords are en- or decrypted, QF-Test combines the key with the salt specified in this option. Without this salt, anybody with sufficient knowledge is able to decrypt your passwords to get the plain-text version.

3.0+

Note

Don't let this option give you a false sense of security. Anybody that gains access to this salt and anybody that can execute your tests can also gain access to the plain-text version of the password. However, encrypting passwords is still useful to prevent obvious plain-text passwords getting stored in test-suites and run-logs, and crypted passwords are reasonably safe from someone who only gets hold of a test-suite or run-log without access to this salt.

How to handle disabled components (System)

4.0+

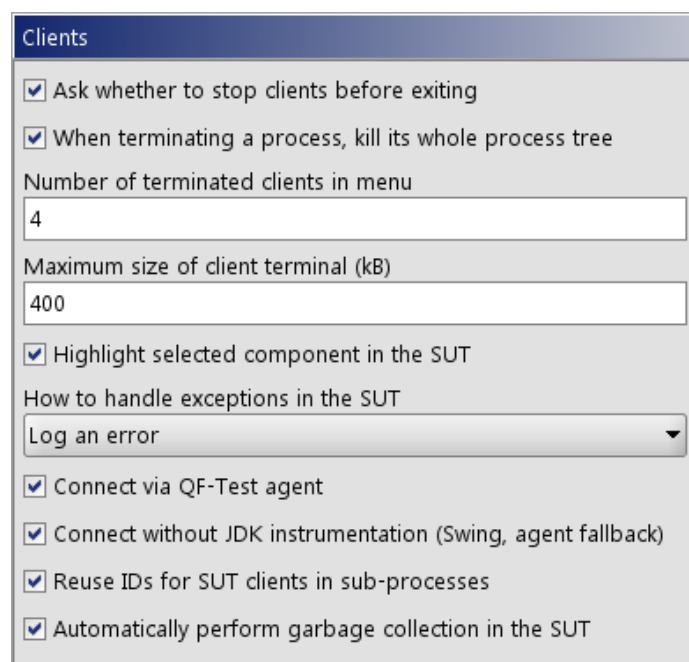
Server script name: OPT_PLAY_ERROR_STATE_DISABLED_COMPONENT
Possible Values: VAL_PLAY_DISABLED_COMPONENT_WARNING,
VAL_PLAY_DISABLED_COMPONENT_ERROR,
VAL_PLAY_DISABLED_COMPONENT_EXCEPTION

In case you replay an event on a component, which is disabled you can configure QF-Test's behavior for that case. You can

- Log a warning message
- Log an error message
- Throw a `DisabledComponentStepException`⁽⁵⁵⁹⁾

29.3.1 Client options

Various settings for process and SUT clients can be adjusted with the following options:



The screenshot shows a dialog box titled "Clients". It contains the following settings:

- ☒ Ask whether to stop clients before exiting
- ☒ When terminating a process, kill its whole process tree
- Number of terminated clients in menu: 4
- Maximum size of client terminal (kB): 400
- ☒ Highlight selected component in the SUT
- How to handle exceptions in the SUT: Log an error
- ☒ Connect via QF-Test agent
- ☒ Connect without JDK instrumentation (Swing, agent fallback)
- ☒ Reuse IDs for SUT clients in sub-processes
- ☒ Automatically perform garbage collection in the SUT

Figure 29.19: Client options

Ask whether to stop clients before exiting (User)

If there are still active clients upon exit of QF-Test, these are terminated after asking for confirmation. If this option is turned off, the clients are terminated unconditionally.

When terminating a process, kill its whole process tree (System)

Server script name: OPT_PLAY_KILL_PROCESS_TREE

The process of an SUT or a helper program started during a test can be terminated via a Stop client⁽⁴⁴¹⁾ node or manually via the **Client** menu. In case of an SUT, QF-Test first tries to communicate with it and initiate a clean `System.exit` call. Non-Java programs have to be killed. If the program has started further child processes these may or may not get terminated by a normal shutdown or kill, depending on circumstances.

It is normally undesirable to keep such processes around as they might interfere with other tests or lock files that need to be removed or overwritten. Unless this option is disabled, QF-Test will try to determine the whole process hierarchy for any program it started and make sure that the main process and all child processes get killed explicitly.

Number of terminated clients in menu (User)

Server script name: OPT_PLAY_MAX_CLIENTS

This option limits the number of menu items for terminated clients that are kept in the **Clients** menu.

Maximum size of client terminal (kB) (User)

Server script name: OPT_PLAY_TERMINAL_SIZE

The maximum amount of text (in kilobyte) that the individual client terminal will hold. If the limit is exceeded, old text will be removed when new text arrives. A value of 0 means no limit.

3.0+

Note

This option also determines the amount of output available for the special variable `${qftest:client.output.<name>}`.

Highlight selected component in the SUT (User)

If this option is set, QF-Test will highlight the associated component in the SUT whenever a Component⁽⁵⁴⁵⁾ node or a node that references a Component is selected.

How to handle exceptions in the SUT (System)

SUT script name: OPT_PLAY_SUT_EXCEPTION_LEVEL

Possible Values: VAL_PLAY_EXCEPTION_LEVEL_WARNING,
VAL_PLAY_EXCEPTION_LEVEL_ERROR,
VAL_PLAY_EXCEPTION_LEVEL_EXCEPTION

Exceptions that are thrown during event handling in the SUT are typically a sure

sign for a bug in the SUT. This option determines what to do if such an exception is caught. You can

- Log a warning message
- Log an error message
- Throw an `UnexpectedClientException`⁽⁵⁶³⁾

Connect via QF-Test agent (System)

Server script name: `OPT_PLAY_CONNECT_VIA_AGENT`

QF-Test version 4 introduces a new mechanism for connecting to an SUT, based on Java agents. It is far more powerful and flexible than the older mechanisms and should not be turned off without a very good reason.

Note

Connecting to a JavaFX based SUT is not possible without the QF-Test agent.

Connect without JDK instrumentation (Swing) (System)

Server script name: `OPT_PLAY_IMPLICIT_INSTRUMENTATION`

With this option activated there is no need to instrument standard JDKs/JREs in order to connect to an AWT/Swing SUT. In cases where automatic connections don't work - or if you deactivate this option - you need to fall back to explicit JDK instrumentation as described in [section 35.1](#)⁽⁶⁰⁴⁾ and [section 35.2](#)⁽⁶⁰⁷⁾.

Note

If this option is activated, QF-Test sets the environment variable `_JAVA_OPTIONS` (and also `IBM_JAVA_OPTIONS` for IBM JDKs) so that any JVM started directly or indirectly by QF-Test will execute QF-Test's accessibility hook when initializing the AWT toolkit. In contrast to other tools QF-Test does not set this environment variable globally and thus does not interfere with Java applications not started from QF-Test. However, if the variable is already set globally and your Java application relies on it, you must deactivate this option and use JDK instrumentation instead.

SWT

Pure Eclipse/SWT applications only require SWT instrumentation (see [section 35.3](#)⁽⁶⁰⁸⁾). Some Eclipse/RCP applications show an AWT based splash screen on startup and if this option is active that splash screen will trigger initialization of QF-Test's AWT/Swing engine, which will then become the primary engine for that SUT. If you rely on the default engine for [SUT script](#)⁽⁴¹⁹⁾ and [Wait for client to connect](#)⁽⁴³⁸⁾ nodes to be 'swt', you need to deactivate this option.

Reuse IDs for SUT clients in nested sub-processes (System)

Server script name: `OPT_PLAY_REUSE_SUT_IDS`

This is a complex option which you should hopefully never care about. When an SUT client launches another process that itself connects to QF-Test, the new

SUT client is identified by the name of the original SUT client with a ':' and a numeric ID appended. The first ID will always be 2, with increasing numbers for additional sub-processes.

When a sub-process terminates and another sub-process connects, QF-Test can either reuse the ID of the terminated process or continue incrementing to create a new ID.

In most cases it is preferable to reuse the sub-process ID. The most common case is a single sub-process that is started, terminated, then started again. By activating this option you can always address the single sub-process with the same client name.

In a more complex situation, multiple sub-processes may be launched and terminated more or less at random, depending on progression of the test-run. In such a case, always incrementing the ID for a new process is more deterministic.

In either case the ID counter will be reset when the original SUT client is started anew.

Automatically perform garbage collection in the SUT (System)

SUT script name: OPT_PLAY_SUT_GARBAGE_COLLECTION

By default QF-Test performs a full garbage collection in the SUT once every few hundred SUT script⁽⁴¹⁹⁾ executions. This is necessary due to a limitation in Java's default garbage collection mechanism that allows an OutOfMemoryError to happen for the so called PermGen space, even though the required memory could easily be reclaimed by a garbage collection.

When you are trying to analyze the memory performance of your application, these explicit garbage collections might influence the results. For that case you can disable QF-Test's garbage collection via this option.

29.3.2 Terminal options

Various settings for the shared terminal can be adjusted with the following options:

Shared Terminal

Maximum size of shared terminal (kB)
20

Regular expression to suppress display of certain text

☒ Use rich text terminal

Terminal font size (pt)
13

☒ Use monospaced font

☒ Regular expression coloring

Regular expression for red highlighting

Regular expression for orange highlighting

Regular expression for yellow highlighting

Regular expression for blue highlighting

Regular expression for green highlighting

Figure 29.20: Terminal options

Maximum size of shared terminal (kB) (User)

Server script name: OPT_PLAY_SHARED_TERMINAL_SIZE

The maximum amount of text (in kilobyte) that the shared terminal will hold. If the limit is exceeded, old text will be removed when new text arrives. A value of 0 means no limit.

Regular expression to suppress display of certain text (User)

By defining a regular expression for this option, certain text in the terminal output can be suppressed.

Default value is empty.

See also Regular expressions⁽⁶¹⁴⁾.

Use rich text terminal (User)

4.0+

If activated the rich text terminal allowing monospaced font type and coloring of given regular expressions. Deactivate this option if you want to switch back to the simple terminal as it were before QF-Test version 4.

Note

QF-Test needs to be restarted in order to make a change in this option become visible.

Terminal font size (pt) (User)

4.0+

This option lets you set the font size (as point value) used within the shared terminal.

This option only has an effect if the Use rich text terminal⁽²⁷⁷⁾ is active.

Use monospaced font (User)

4.0+

If activated the rich text terminal will use a monospaced font.

This option only has an effect if the Use rich text terminal⁽²⁷⁷⁾ is active.

Regular expression coloring (User)

4.0+

If activated the shared terminal output is processed for given regular expressions to be highlighted in different colors.

This option only has an effect if Use rich text terminal⁽²⁷⁷⁾ and Regular expression coloring⁽²⁷⁸⁾ are active.

Regular expression for red highlighting (User)

4.0+

This option allows to define a regular expression for output to be shown in red color.

This option only has an effect if Use rich text terminal⁽²⁷⁷⁾ and Regular expression coloring⁽²⁷⁸⁾ are active.

Default value is:

```
(?md).+Exception\b.*\n(?:>\n?^(?:>\sat|Caused
by:)\s.+\\n)+|.*(?:i)exception(?:>s)?\b.*
```

With this also typical Java stack traces will be highlighted.

See also Regular expressions⁽⁶¹⁴⁾.

Regular expression for orange highlighting (User)

4.0+

This option allows to define a regular expression for output to be shown in orange color. The default regular expression matches error log output and lines containing respective text.

This option only has an effect if Use rich text terminal⁽²⁷⁷⁾ and Regular expression coloring⁽²⁷⁸⁾ are active.

Default value is:

```
(?md)^[1-2] \\(\\d\\d:\\d\\d:\\d\\d\\.\\d\\d\\d\\) .*.*(?i)(?>error(?>s)?|fehler)\\b.*
```

With this also error log messages will be highlighted.

See also Regular expressions⁽⁶¹⁴⁾.

Regular expression for yellow highlighting (User)

This option allows to define a regular expression for output to be shown in yellow color. The default regular expression matches with typical stack traces and exceptions.

This option only has an effect if Use rich text terminal⁽²⁷⁷⁾ and Regular expression coloring⁽²⁷⁸⁾ are active.

Default value is:

```
(?md)^[3-4] \\(\\d\\d:\\d\\d:\\d\\d\\.\\d\\d\\d\\) .*.*(?i)(?>warning(?>s)?|warnung(?>en)?)\\b.*
```

With this also warning log messages will be highlighted.

See also Regular expressions⁽⁶¹⁴⁾.

Regular expression for blue highlighting (User)

This option allows to define a regular expression for output to be shown in blue color. The default regular expression matches with typical stack traces and exceptions.

This option only has an effect if Use rich text terminal⁽²⁷⁷⁾ and Regular expression coloring⁽²⁷⁸⁾ are active.

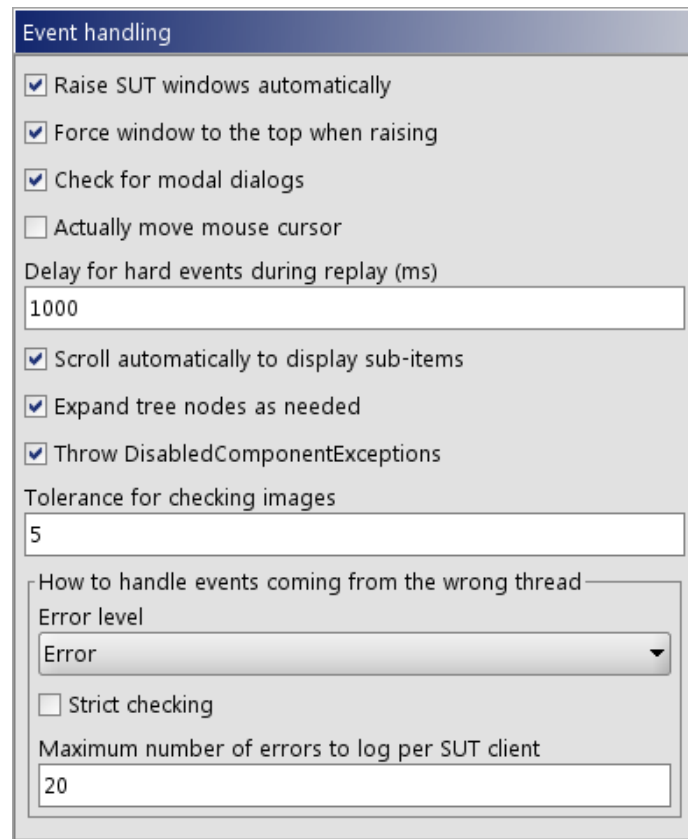
Regular expression for green highlighting (User)

This option allows to define a regular expression for output to be shown in green color. The default regular expression matches with typical stack traces and exceptions.

This option only has an effect if Use rich text terminal⁽²⁷⁷⁾ and Regular expression coloring⁽²⁷⁸⁾ are active.

29.3.3 Event handling

These options influence some details of how events are simulated in the SUT during replay.



The screenshot shows a dialog box titled "Event handling" with the following options and settings:

- ☒ Raise SUT windows automatically
- ☒ Force window to the top when raising
- ☒ Check for modal dialogs
- ☐ Actually move mouse cursor
- Delay for hard events during replay (ms): 1000
- ☒ Scroll automatically to display sub-items
- ☒ Expand tree nodes as needed
- ☒ Throw DisabledComponentExceptions
- Tolerance for checking images: 5
- How to handle events coming from the wrong thread:
 - Error level: Error (dropdown menu)
 - ☐ Strict checking
- Maximum number of errors to log per SUT client: 20

Figure 29.21: Event handling options

Raise SUT windows automatically (System)

SUT script name: OPT_PLAY_RAISE_SUT_WINDOWS

If this option is set, windows of the SUT for which a `MouseEvent` or `KeyEvent` is replayed will be raised automatically when they get activated. This eases switching between QF-Test and the SUT to visually verify that a sequence is replaying correctly.

See also options Raise test-suite window after replay⁽²⁷¹⁾ and Force window to the top when raising⁽²⁸⁰⁾.

Force window to the top when raising (System)

SUT script name: OPT_PLAY_RAISE_SUT_WINDOWS_FORCED

This option only has an effect on Windows systems.

Windows only allows an application to bring one of its own windows to the front if that application currently has the focus. This can make it difficult for QF-Test to raise SUT windows and to automatically switch between the SUT and QF-Test. If this option is activated, QF-Test temporarily sets the "always on top" attribute to force windows to the top.

See also options Raise test-suite window after replay⁽²⁷¹⁾ and Raise SUT windows automatically⁽²⁸⁰⁾.

Check for modal dialogs (System)

SUT script name: OPT_PLAY_CHECK_MODAL

A modal dialog is a window that blocks all input to other windows until it is closed. It is often used to display an error message or request user input.

Because the events simulated by QF-Test are artificial, they are not blocked by a modal dialog and can reach any window. This is normally not desirable, since the existence of a modal dialog may signal an unexpected error. Activating this option causes QF-Test to check for modal dialogs itself before replaying MouseEvents or KeyEvents. If such an event is targeted to a window that is blocked by a modal dialog, a ModalDialogException⁽⁵⁵⁸⁾ is thrown.

Actually move mouse cursor (System)

SUT script name: OPT_PLAY_MOVE_MOUSE_CURSOR

If this option is set, the mouse cursor is actually moved across the screen as MouseEvents are simulated. This feature requires a working AWT robot.

While this option is mainly intended to give visual feedback, it can have a positive impact on test reliability because it reduces side-effects through events from the underlying system that might interfere with the automated test. However, for tests where precise mouse movement is essential, for example a drawing tool, this option should be turned off.

Delay for hard events during replay (ms) (System)

SUT script name: OPT_PLAY_DELAY_HARD_EVENTS

During event replay QF-Test blocks or delays some "hard" events, i.e. events that come directly from the system, for example when the user is moving the mouse cursor. This is done to prevent them from interfering with the SUT in an unlucky moment. Popup windows, which are used for menus and combo boxes among other things, are especially sensitive to such events which can cause them to pop down accidentally. Therefore, these filters improve testing stability considerably.

Note

This option sets the maximum time that such events may be delayed. In the unlikely case that the filters have unwanted side effects with your application, you can turn them off by setting the value to 0.

Scroll automatically to display sub-items (System)

SUT script name: OPT_PLAY_SCROLL_ITEM

If this option is set, accessing a sub-item of a complex component inside a scroll pane will automatically cause the sub-item to be scrolled into view. In that case you can remove most recorded events on scroll bars or scroll buttons, which are not required for correct replay.

Expand tree nodes as needed (System)

SUT script name: OPT_PLAY_EXPAND_TREE

When accessing nodes of a tree component as hierarchic sub-items it is possible to select a node that is not visible because one of its ancestral nodes is not expanded. If this option is set, all ancestors of the node will be expanded as needed. Otherwise this situation leads to a `ComponentNotFoundException`⁽⁵⁵⁸⁾.

Throw DisabledComponentException (System)

SUT script name: OPT_PLAY_THROW_DISABLED_EXCEPTION

If QF-Test replays events on a component that is not enabled, these events are ignored silently. In most cases this indicates an error which is signaled by throwing a `DisabledComponentException`⁽⁵⁵⁹⁾.

Old test-suites may not be prepared to deal with this exception. These test-suites should be fixed, but as a quick workaround `DisabledComponentExceptions` can be suppressed by deactivating this option.

Tolerance for checking images (System)

SUT script name: OPT_PLAY_IMAGE_TOLERANCE

Initially this option was intended for SWT/Gtk only but it turned out to be universally applicable and useful.

Image rendering in Java applications and web browsers is not always fully deterministic. Even within the same session on a display with limited color depth the RGB values of an icon image can vary slightly and it becomes worse when running tests on different machines. Graphics driver, JDK version and operating system settings also play a role. This makes strict image checks almost unusable in some cases.

SWT

Note

To work around the problem, this option defines a tolerance setting for default image checks. For a pixel's red, green and blue color values a deviation from the expected value by the given amount is tolerated. Thus exact image checking for the default algorithm can be enforced by setting this value to 0 but it is preferable to use the "identity" check algorithm instead (see [Details about the algorithm for image comparison^{\(769\)}](#)). The default setting of 5 is a good compromise, allowing checks with differences that are normally not visually perceivable to succeed.

How to handle events coming from the wrong thread (System)

Swing

SUT script name: `OPT_PLAY_WRONG_THREAD_ERROR_LEVEL`

Possible Values: `VAL_PLAY_THREAD_LEVEL_WARNING`,
`VAL_PLAY_THREAD_LEVEL_ERROR`,
`VAL_PLAY_THREAD_LEVEL_EXCEPTION`

It is a rather common mistake in Swing based Java applications to access GUI components from the wrong thread. Since Swing is not thread-safe, such calls may only be made from the AWT event dispatch thread. Otherwise the potential consequences are race conditions, leading to very subtle and hard-to-debug errors, or deadlocks, freezing the application and making it unusable. Background information about this topic is available from

<http://download.oracle.com/javase/tutorial/uiswing/concurrency/index.html>, specifically the sections on "Initial Threads" and "The Event Dispatch Thread".

When QF-Test registers an event on a thread other than the AWT event dispatch thread it issues an error message including a stack trace which can be useful in fixing the problem. This set of options defines the severity of the message, whether to perform strict checking and a maximum for the number of messages to log.

The possible choices for the option "Error level" are "Error" and "Warning". We strongly suggest that you keep the default setting of "Error" and make sure that such problems are fixed sooner rather than later because they represent a serious risk.

Strict checking (System)

Swing

SUT script name: `OPT_PLAY_WRONG_THREAD_STRICT`

If strict checking for wrong thread errors is activated, error or warning messages will be issued for all kinds of events triggered from the wrong thread. Otherwise the problem will be ignored for "less relevant" events. Which events are considered more or less relevant is arbitrary, based on the fact that there's a lot of Java literature (including early Java documentation from Sun) claiming that it is safe to create components on any thread and that thread-safety only needs to be enforced once a component is made visible. A lot of code follows this pattern and the risk for causing problems in this case is indeed minimal. In the presence of

such code disabling "Strict checking" will cause error messages to be logged only for the more relevant problems. If you want to get rid of all thread violations - as we recommend - you should turn strict checking on.

Maximum number of errors to log per SUT client (System)

Swing

SUT script name: OPT_PLAY_WRONG_THREAD_MAX_ERRORS

In case an SUT contains code that violates thread safety it is possible that a very large number of events are triggered from the wrong thread. Logging all those errors can significantly impact test performance, yet logging more than the first few errors does not really contribute much. The option "Maximum number of errors to log per SUT client" limits the possible number of error messages for this case.

29.3.4 Component recognition

How component recognition works - and the impact of these options on it - is explained in section 36.2⁽⁶¹²⁾. The pre-defined values should give good results. If you experience problems with component recognition, you can try to improve it by adjusting the probabilities.

Component recognition

Name override mode
Hierarchical resolution

Log warning for

☐ Missing name	☐ Ambiguous name
☒ Feature mismatch	☒ Extra feature mismatch
☒ Structure mismatch	☒ Intermediate named ancestor

Name bonus (%) 90 Name penalty (%) 0

Feature bonus (%) 80 Feature penalty (%) 55

Extra feature bonus (%) 70 Extra feature penalty (%) 55

Structure bonus (%) 70 Structure penalty (%) 60

Modal penalty (%) 0

Minimum probability (%) 50

Figure 29.22: Component recognition options

The name of a component plays a special role. The following two options affect the impact of names:

Name override mode (replay) (System)

SUT script name: OPT_PLAY_RECOGNITION_NAME_OVERRIDE

Possible Values: VAL_NAME_OVERRIDE_EVERYTHING,
VAL_NAME_OVERRIDE_HIERARCHY,
VAL_NAME_OVERRIDE_PLAIN

Note

There are two versions of this option which are closely related. This one is effective during replay, the other one⁽²⁶⁴⁾ during recording. Obviously, both options should always have the same value. There's one exception though: When migrating from one setting to another, QF-Test's components have to be updated. During that process, keep this option at the old setting and change the record option to the new one. Be sure to update the replay setting after updating the components.

This option determines the weight given to the names of components for component recognition. Possible choices are:

Override everything

This is the most effective and adaptable way of searching components, but it requires that the names of the components are unique, at least within the same window. If that uniqueness is given, use this choice.

Don't use this value for a web page with frames. Use "Hierarchical resolution" instead.

Hierarchical resolution

This choice should be used if component names are not unique on a per-window basis, but naming is still used consistently so that two components with identical names have at least parent components or ancestors with distinct names. That way, component recognition is still tolerant to a lot of change, but if a named component is moved to a different named parent in the SUT, the test-suite will have to be updated to reflect the change.

Plain attribute

If there are components with identical names in the SUT within the same parent component you must use this setting. The name will still play an important role in component recognition, but not much more than the Feature⁽⁵⁴⁷⁾ attribute.

Log warning for missing name (System)

SUT script name: OPT_PLAY_WARN_MISSING_NAME

If this option is set, a warning will be logged whenever a component is targeted that does not have a name, but QF-Test "thinks" it should have one. A plausible name is suggested where possible.

Log warning for ambiguous name (System)

SUT script name: OPT_PLAY_WARN_AMBIGUOUS_NAME

If the option Name override mode (replay)⁽²⁸⁵⁾ is set to "Override everything" or "Hierarchical resolution", a warning will be logged whenever QF-Test encounters more than one potential target components with the same name. That warning can be suppressed with the help of this option.

Log warning for feature mismatch (System)

SUT script name: OPT_PLAY_WARN_FEATURE_MISMATCH

A component is considered to have a "feature mismatch" if it is determined by QF-Test as the target best suited for an event or check even though at one or more levels of the hierarchy the recorded Feature⁽⁵⁴⁷⁾ attribute did not match the current state of the component. If this option is activated, feature mismatches are

logged as warnings, notifying you that it may be a good idea to update the affected components.

Log warning for extra feature mismatch (System)

SUT script name: OPT_PLAY_WARN_EXTRA_FEATURE_MISMATCH

An "extra feature mismatch" is similar to a feature mismatch as explained above, except that it applies to extra features with status "should match". If this option is activated, extra feature mismatches are logged as warnings, notifying you that it may be a good idea to update the affected components.

Log warning for structure mismatch (System)

SUT script name: OPT_PLAY_WARN_STRUCTURE_MISMATCH

A "structure mismatch" is similar to a feature mismatch as explained above, except that instead of the feature it is the structure information represented by the attributes Class index⁽⁵⁴⁹⁾ and Class count⁽⁵⁴⁹⁾ where the mismatch occurred. If this option is activated, structure mismatches are logged as warnings, notifying you that it may be a good idea to update the affected components.

Log warning for intermediate named ancestor (System)

SUT script name: OPT_PLAY_WARN_NAMED_ANCESTOR

An "intermediate named ancestor" is a direct or indirect parent component of the target component in the SUT which is not part of the component hierarchy in QF-Test even though it has a name. If the option Name override mode (replay)⁽²⁸⁵⁾ is set to "Hierarchical resolution", this is considered as a mismatch, comparable to a feature or structure mismatch. If this option is activated, interferences through intermediate named ancestors are logged as warnings, notifying you that it may be a good idea to update the affected components.

For an explanation of the remaining options for component recognition please refer to [section 36.2](#)⁽⁶¹²⁾. The respective SUT script names for these options are:

OPT_PLAY_RECOGNITION_BONUS_NAME
OPT_PLAY_RECOGNITION_PENALTY_NAME
OPT_PLAY_RECOGNITION_BONUS_FEATURE
OPT_PLAY_RECOGNITION_PENALTY_FEATURE
OPT_PLAY_RECOGNITION_BONUS_EXTRAFEATURE
OPT_PLAY_RECOGNITION_PENALTY_EXTRAFEATURE
OPT_PLAY_RECOGNITION_BONUS_STRUCTURE
OPT_PLAY_RECOGNITION_PENALTY_STRUCTURE
OPT_PLAY_RECOGNITION_PENALTY_MODAL
OPT_PLAY_RECOGNITION_MINIMUM_PROBABILITY

29.3.5 Delays

Here values can be set for various delays.

The screenshot shows a dialog box titled "Delays". It contains two main sections. The first section, "Default delays (ms)", has two input fields: "Before" with the value 0 and "After" with the value 0. The second section is divided into two columns. The left column, titled "Drag & Drop and hard eve...", contains three input fields: "Delay (ms)" with the value 10, "Interpolation step size" with the value 1, and "Acceleration" with the value 4. Below these are two more input fields: "Acceleration threshold" with the value 6. The right column, titled "Actually move mouse cursor", contains three input fields: "Delay (ms)" with the value 5, "Interpolation step size" with the value 4, and "Acceleration" with the value 8. Below these are two more input fields: "Acceleration threshold" with the value 2.

Figure 29.23: Delay options

Default delays (ms) (System)

Server script name: OPT_PLAY_DELAY_BEFORE, OPT_PLAY_DELAY_AFTER
 These two options set the delay in milliseconds before and after the execution of a node. If a node defines its own Delay before/after⁽³³⁶⁾, its value overrides this default.

These options are useful to slow a test down so you'll be able to follow it.

Drag&Drop and interpolation of mouse movement

Simulating Drag&Drop is non-trivial. It is made possible only by generating "hard" mouse events that actually move the mouse cursor. On Windows systems, some mouse drivers can interfere with these "hard" events. See section 36.1⁽⁶¹¹⁾ for further details about Drag&Drop.

To make Drag&Drop as reliable as possible, movement of the mouse cursor is highly configurable. Since the requirements for Drag&Drop and hard mouse events differ from those for general mouse moves which only provide visual feedback, two sets of options

are provided. The settings for demo mouse moves are ignored unless the respective option Actually move mouse cursor⁽²⁸¹⁾ is activated.

Typically movements for Drag&Drop and hard events should be slower and involve more interpolation steps than those for demo moves, which could slow down a test considerably. All of the following values influence the overall speed of mouse moves. A little experimentation may be required to get the desired effect.

Delay (ms) (System)

SUT script name: OPT_PLAY_DND_DELAY, OPT_PLAY_MOVEMOUSE_DELAY
After each single movement of the mouse cursor QF-Test will wait for the specified number of milliseconds. This value should be between 2 and 20 if interpolation is enabled and between 20 and 200 if interpolation is turned off. With interpolation, 10 is a good value for Drag&Drop and 5 for demo mouse moves.

Interpolation step size (System)

SUT script name: OPT_PLAY_DND_STEP, OPT_PLAY_MOVEMOUSE_STEP
The size of the steps for interpolation of mouse movement. Set this to 0 to turn interpolation off. Good values are between 1 and 3 for Drag&Drop and between 2 and 10 for demo mouse moves.

Acceleration (System)

SUT script name: OPT_PLAY_DND_ACCELERATION, OPT_PLAY_MOVEMOUSE_ACCELERATION
To avoid needless slowdown of tests, long distance mouse movement can be accelerated. A value of 0 turns off acceleration. Useful values range from 1 for very little acceleration to 10 or more for high acceleration. Good values are between 3 and 5 for Drag&Drop and between 6 and 20 for demo mouse moves.

Acceleration threshold (System)

SUT script name: OPT_PLAY_DND_THRESHOLD, OPT_PLAY_MOVEMOUSE_THRESHOLD
To ensure that small movements as well as the start end end of each movement remain precise, acceleration is turned off for movements that require less than this threshold's number of steps. Good values are between 4 and 8 for Drag&Drop and between 0 and 6 for demo mouse moves.

29.3.6 Timeouts

These timeouts are essential for reliable replay of tests under varying conditions. They define how long QF-Test waits for a component to be in the proper state for an event before throwing an exception.

Don't make these values too small, so a little hiccup due to high load won't interrupt a test needlessly. QF-Test continues as soon as the conditions for replaying an event are met, so higher values for the timeouts won't slow down execution (except for focus, see below). On the other hand, don't set any values higher than a few seconds or you'll have to wait too long until you finally get an error message when a component is truly not found.

Timeouts	
Deadlock detection (s)	120
Wait for GUI engine (ms)	5000
Wait for non-existent component (ms)	5000
Wait for non-existent item (ms)	2000
Default timeout for checks (ms)	500
Wait for modal dialog (ms)	5000
Wait for 'busy' GlassPane (ms)	3000
Wait for button/menu enable (ms)	3000
Wait for focus (ms)	20
Poll interval for component wait (ms)	300
Sub-item poll interval (ms)	300
Retry check interval (ms)	300

Figure 29.24: Timeout options

Deadlock detection (s) (System)

Server script name: OPT_PLAY_TIMEOUT_DEADLOCK

If the SUT does not react for the given time, a `DeadlockTimeoutException`⁽⁵⁵⁹⁾ is thrown. Setting this value to 0 will suppress deadlock detection.

Wait for GUI engine (ms) (System)

SUT script name: OPT_PLAY_TIMEOUT_ENGINE

This option is useful only for multi-engine SUTs, like Eclipse with embedded AWT/Swing components. A `Wait for client to connect`⁽⁴³⁸⁾ node finishes as soon as the first GUI engine connects to QF-Test, unless its `GUI engine`⁽⁴⁴⁰⁾ attribute specifies to wait for a specific engine. To prevent a subsequent `Wait for component to appear`⁽⁵¹²⁾ node for a component of the wrong engine from failing immediately, QF-Test first waits for the time specified with this option to give the second GUI engine a chance to connect also.

Wait for non-existent component (ms) (System)

SUT script name: OPT_PLAY_TIMEOUT_COMPONENT

The maximum time in milliseconds that QF-Test waits for the target component of an event to become visible. When the connection between QF-Test and the SUT is established, this option is temporarily set to at least 30000 ms so as to allow the SUT time for its initialization.

Wait for non-existent item (ms) (System)

SUT script name: OPT_PLAY_TIMEOUT_ITEM

If an event is targeted on a sub-item of a complex component, QF-Test first waits for the component to become visible. Then it gives the SUT the chance to make the intended sub-item available before this timeout is exceeded.

Default timeout for checks (ms) (System)

Server script name: OPT_PLAY_CHECK_TIMEOUT

This option defines a default timeout for Check nodes that have no `Timeout`⁽⁴⁷⁴⁾ attribute set and that represent an actual check in the report instead of being used for test control flow, i.e. checks that don't throw an exception and don't set a result variable or that have a `@report` doctag.

If your tests include a lot of Check nodes without explicitly defined Timeout that are expected to fail - which is unlikely for the actual checks described above - you may be able speed up test execution by setting this value to 0. However,

it would be preferable to set the Timeout attribute of the respective nodes to 0 instead and leave this option unchanged because it increases general stability of check execution.

Wait for modal dialog (ms) (System)

SUT script name: OPT_PLAY_TIMEOUT_MODAL

If an event is targeted at a window that is blocked by a modal dialog, a `ModalDialogException`⁽⁵⁵⁸⁾ will be thrown. However, modal dialogs are often temporary, informing the user about some ongoing processing. If this option is set to a non-zero value, QF-Test will wait for the given time before it throws the exception. If the modal dialog disappears before the time limit is exceeded, the test will continue immediately. This greatly simplifies handling of temporary modal dialogs.

Note

If the option Convert opening of a window into Wait for component to appear^{(512) (256)} is activated, recording a sequence of events during which a temporary dialog is displayed may result in a Wait for component to appear⁽⁵¹²⁾ node for that dialog. If the dialog is displayed for a short time only, it is best to remove such nodes to avoid timing issues. If the SUT employs temporary modal dialogs often it may be best to disable the option Convert opening of a window into Wait for component to appear⁽⁵¹²⁾
(256).

Wait for 'busy' GlassPane (ms) (System)

Swing

SUT script name: OPT_PLAY_TIMEOUT_GLASSPANE

As an alternative to temporary modal dialogs some applications employ a so called GlassPane together with a 'busy' mouse cursor (typically in the form of an hourglass) to inform the user that the application is busy. The GlassPane is an invisible component that covers an entire window. If an event is delivered to such a window, the GlassPane will typically intercept it, preventing normal event processing, which can throw a test-run severely off course.

QF-Test handles this case automatically by waiting until the GlassPane disappears before delivering an event, performing a check, etc. If the timeout given in this option is exceeded and the GlassPane is still active, a `BusyPaneException`⁽⁵⁵⁹⁾ is thrown.

If the value of this option is set to 0, GlassPane checking is disabled and the event is delivered regardless. A `BusyPaneException` is never thrown in this case.

The Wait for component to appear⁽⁵¹²⁾ node is a special case. When waiting for a component (not its absence) covered by a busy GlassPane, the Timeout⁽⁵¹⁴⁾ attribute of the node overrides this option and is used to wait for both, the appearance of the component and the removal of the busy GlassPane. Thus it is possible to handle

cases where the SUT is expected to be busy for an exceptionally long time on an individual basis without changing the default timeout of this option.

Wait for button/menu enable (ms) (System)

SUT script name: OPT_PLAY_TIMEOUT_ENABLED

A MouseEvent on a button or menu item is simply ignored, if the component is not enabled. This could lead to unwanted side effects during a test, so QF-Test waits until the component is enabled or the specified timeout is exceeded. If the component does not become activated within the given time, a DisabledComponentException⁽⁵⁵⁹⁾ is thrown unless the option Throw DisabledComponentException⁽²⁸²⁾ is deactivated.

Wait for focus (ms) (System)

SUT script name: OPT_PLAY_TIMEOUT_FOCUS

If set, this timeout causes QF-Test to wait for a component to get the keyboard focus before it simulates any KeyEvents on it. This option actually can slow down a test noticeably if a component fails to get the focus, so don't set it higher than about 100. A good value is 20.

Poll interval for component wait (ms) (System)

SUT script name: OPT_PLAY_POLL_COMPONENT

When waiting for a component in the SUT to appear, QF-Test can't always rely on Java's event mechanism. Instead it has to repeatedly scan the SUT for the component. This option determines the interval between searches.

Sub-item poll interval (ms) (System)

SUT script name: OPT_PLAY_POLL_ITEM

When waiting for a non-existent sub-item in the SUT, QF-Test can't rely on the event mechanism. Instead it has to repeatedly search the complex component for the sub-item. This option determines the interval between searches.

Retry check interval (ms) (System)

SUT script name: OPT_PLAY_POLL_CHECK

If a Check⁽⁴⁶⁹⁾ fails for which a Timeout⁽⁴⁷⁴⁾ is defined, QF-Test repeatedly queries the component's state until either it matches the given values or the time is up. The interval to wait between queries is set with this option.

29.4 Web options

The following options are used specifically for web testing.

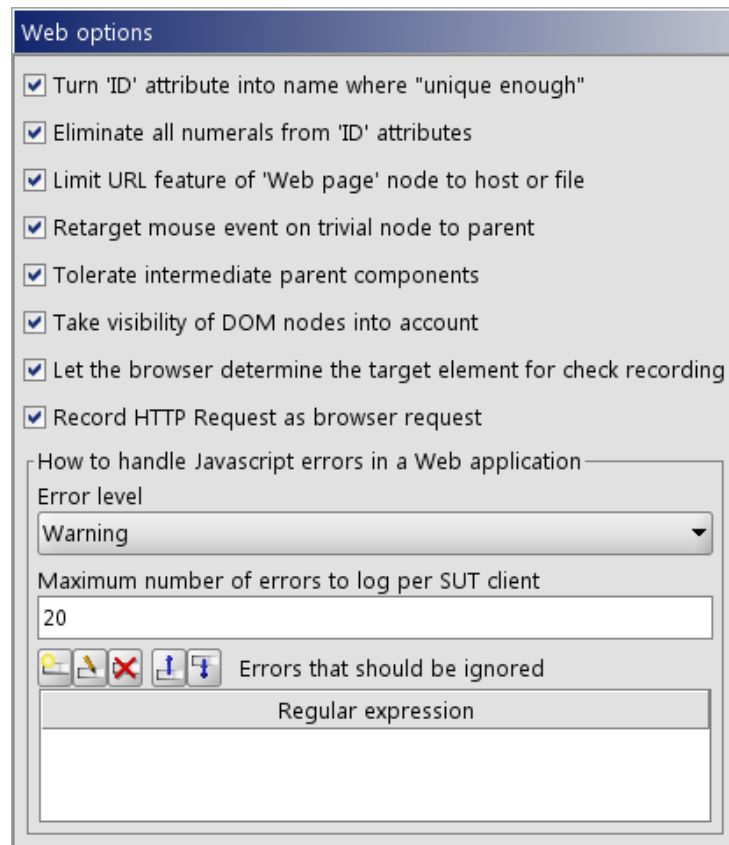


Figure 29.25: Web options

Turn 'ID' attribute into name where "unique enough" (System)

SUT script name: OPT_WEB_ID_AS_NAME

Use the ID of a DOM node as the component name, provided the ID is sufficiently unique. Uniqueness is determined per node type and depending on the setting of the options Name override mode (replay)⁽²⁸⁵⁾ and Name override mode (record)⁽²⁶⁴⁾.

Eliminate all numerals from 'ID' attributes (System)

SUT script name: OPT_WEB_SUPPRESS_NUMERALS

Note

Of course this option only changes the way QF-Test treats ID attributes. The attributes themselves are left unchanged or the application would most likely no longer work.

With this option activated QF-Test removes all numerals from 'ID' attributes to prevent problems caused by automatically generated IDs often found in Ajax frameworks like GWT. Such dynamic IDs can change after the slightest modification to a web application which causes tests to break, especially if names are based on IDs. By removing the dynamic part from such IDs they become less useful, because they are no longer unique, but also less harmful. Uniqueness of names is taken care of by QF-Test. Since IDs also serve as a basis for Feature⁽⁵⁴⁷⁾ and Extra features⁽⁵⁴⁷⁾ attributes, this option is helpful even if IDs are not used as names.

Limit URL feature of 'Web page' node to host or file (System)

SUT script name: OPT_WEB_LIMIT_URL

If this option is active, all pages coming from the same host are recorded as the same page by reducing the URL feature to the host part of the URL. This is often useful when pages share a common look and navigation structure.

For file URLs, the URL is reduced to the filename, with intermediate directories removed.

Retarget mouse event on trivial node to parent (System)

SUT script name: OPT_WEB_RETARGET_MOUSE_EVENT

When recording mouse events on DOM nodes in a web page it is often useful to ignore "trivial" nodes and concentrate on the important ones. For example, when clicking on a text hyperlink it is typically not of interest whether part of the link is formatted with a bold font. It is the link that is important.

If this option is active QF-Test does not simply record the event for the deepest DOM node under the mouse cursor. Instead it moves up the hierarchy until it finds an "interesting" node. In the example above, QF-Test would record the event on the A node with the option active and on the contained B node otherwise.

Tolerate intermediate parent components (System)

SUT script name: OPT_WEB_TOLERATE_INTERMEDIATE_PARENT

Normally QF-Test's component recognition is tolerant to changes in the component hierarchy. For web pages with deeply nested tables this can lead to performance problems because the potential variants of determining the target component grow exponentially with the nesting depth. If you experience such

problems, try to deactivate this option. It will reduce adaptability but should help with performance.

Note

The by far preferable solution is to set unique ID attributes for the different tables and other components so that QF-Test's name override mechanism can apply. This not only speeds up recognition drastically, it is also much more reliable and tolerant to change.

Take visibility of DOM nodes into account (System)

SUT script name: OPT_WEB_TEST_VISIBILITY

Similar to AWT/Swing or SWT, QF-Test normally only recognizes visible DOM nodes as target components. However, visibility of DOM nodes is not as well defined as that of components in a Java GUI. For example it is possible that an invisible DOM node has visible child nodes. Also, if a web page contains illegal HTML constructs it is possible that a DOM node is considered invisible, even though it is displayed in the browser window. If you come across such a problem you can turn off this option.

Let the browser determine the target element for check recording (System)

SUT script name: OPT_WEB_CHECK_VIA_BROWSER

When recording checks, components or procedures QF-Test needs to determine the target element under the mouse cursor. In case of overlapping nodes there are two different ways for calculating which one should be used. By default QF-Test lets the browser decide, which is usually the best choice. Since the different browsers don't always behave in the same reliable way, this option can be turned off in case of problems to use the older mechanism based on the z-order of elements instead. This option has no effect on check replay.

Record HTTP Request as browser request (System)

SUT script name: OPT_WEB_RECORD_CLIENT_REQUEST_STEP

When recording HTTP Requests a Browser HTTP request⁽⁵³²⁾ is created. This request will be submitted directly via the browser so that the response is shown afterwards and test execution can be continued directly in the browser. By deactivating this option a Server HTTP request⁽⁵²⁸⁾ will be recorded. This request will be submitted by QF-Test and doesn't have any effect on the browser. The response is only accessible in QF-Test.

How to handle Javascript errors in a Web application (System)

SUT script name: OPT_WEB_JAVASCRIPT_ERROR_LEVEL

Possible Values: VAL_WEB_JAVASCRIPT_LEVEL_WARNING,
VAL_WEB_JAVASCRIPT_LEVEL_ERROR

Dynamic HTML and AJAX are implemented via a lot of Javascript code that is executed in the Browser. If an error occurs in such a script, browsers either ignore it or show an error dialog with some details about the error, depending on user setting. Many of these errors are harmless, others can be severe. QF-Test intercepts the error message and logs an error or warning in the run-log instead. This set of options defines the severity of the message and a maximum for the number of such kinds of messages to log.

The possible choices for the option "Error level" are "Error" and "Warning". We advise that you set it to "Error" and make sure that such problems are reported to development and fixed sooner rather than later because they can represent a bug in the application you are testing. Known messages that are not going to be fixed by development can be excluded and ignored via the option Errors that should be ignored⁽²⁹⁷⁾.

Maximum number of errors to log per SUT client (System)

SUT script name: OPT_WEB_JAVASCRIPT_MAX_ERRORS

In case a web page contains erroneous code it is possible that a lot of errors are triggered. Logging all those errors can significantly impact test performance, yet logging more than the first few errors does not really contribute much. The option "Maximum number of errors to log per SUT client" limits the possible number of error messages for this case.

Errors that should be ignored (System)

SUT script name: OPT_WEB_JAVASCRIPT_ERROR_FILTERS

It is possible that some Javascript errors cannot or will not be fixed, for example when they are coming from third-party code. In such a case it is preferable to ignore the known errors while still having QF-Test report unexpected ones. When the browsers reports a Javascript error, QF-Test searches its error message for the occurrence of any of the regular expressions specified in this option. In case of a match, the error is ignored. If no exceptions are defined or none match, the error is reported in accordance with the previous options.

29.5 Debugger options

These options modify the behavior of the debugger.

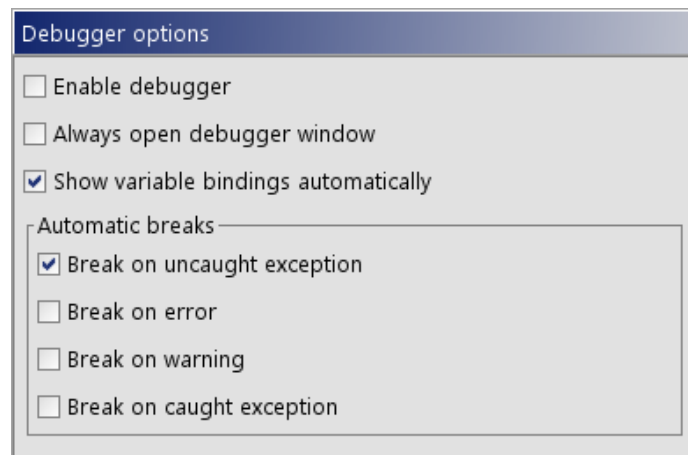


Figure 29.26: Debugger options

Enable debugger (User)

By default the debugger is disabled unless this option, which can also be modified through the menu item `Debugger→Enable debugger`, is activated.

If a test is interrupted by a breakpoint or by pressing the "Pause" button, the debugger is activated automatically. Similarly, starting a test-run with "Step in" or "Step over" will activate the debugger for the duration of that test-run.

Always open debugger window (User)

When test execution is halted and the debugger entered QF-Test can optionally open a separate window for the debugger. This option determines whether debugging should happen in a separate window or the normal test-suite window.

Show variable bindings automatically (User)

When test execution is halted and the debugger entered QF-Test can display the current variable bindings in the workbench window. If this option is active the variables are shown automatically each time a test-run first enters the debugger. Alternatively the variables can be viewed in the debugger window or shown in the workbench window via the menu `Debugger→Show variables`.

Automatic breaks (User)

These options describe the situations in which execution of a test will be suspended and the debugger entered:

Break on uncaught exception

The debugger will break if an exception is thrown that will not be handled by a Catch⁽⁴⁰⁸⁾ node.

Break on error

The debugger will break if an error occurs.

Break on warning

The debugger will break if a warning occurs.

Break on caught exception

You practically never want this! The debugger will break if an exception is thrown, even if that exception will be handled by a Catch⁽⁴⁰⁸⁾ node.

29.6 Run-log options

These options let you control which information is collected in a run-log, if and when a run-log is shown and how to locate errors.

29.6.1 General run-log options

Run-log options

Show run-log automatically

☐ At start ☐ After finish

☐ On exception ☒ Don't show

Number of run-logs in menu

4

☒ Automatically save run-logs

Directory for run-logs

☒ Show expanded variable values in tree nodes

Find next error function locates

Errors

☒ Skip suppressed errors

☒ Cleanup tree when jumping to next or previous error

☒ Save compressed run-logs (*.qrz)

Split run-logs

☒ Create split run-logs

☒ Save split run-logs as ZIP files (*.qzp)

Minimum size for automatic splitting (kB)

10000

Name for automatically split 'Test-case' run-log

`${qftest:testcase.splitlogname}`

Name for automatically split 'Test-set' run-log

`${qftest:testset.splitlogname}`

Figure 29.27: General run-log options

When to show run-logs (User)

A run-log is created for every execution of a test. A number of recent run-logs are available from the **Run** menu, the most recent run-log can also be opened by pressing **Control-L**. Additionally the run-log can be shown during execution or after an error as follows:

At start

This choice causes QF-Test to open the run-log when it begins executing a test sequence. The nodes of the log will be added as execution proceeds.

After finish

With this choice the run-log is shown after replay is finished.

On exception

The run-log is shown only if an uncaught exception is thrown.

Don't show

The run-log is not displayed automatically. You have to open it via the **Run** menu or by pressing **Control-L**.

Number of run-logs in menu (User)

A limit for the number of menu items for recent run-logs kept in the **Run** menu.

Automatically save run-logs (User)

3.0+

To prevent excessive memory use through run-logs and also to make the most recent run-logs persistent between QF-Test sessions, the recent run-logs kept in the **Run** menu are saved automatically to the directory `.qftest` in the current user's home directory or the directory defined in the option Directory for run-logs⁽³⁰¹⁾. The filename for the run-log is based on a timestamp. QF-Test uses file locks to prevent collisions and accidental removal in case of parallel sessions and automatically keeps the `.qftest` directory clean by removing unreferenced logs, so there should be no reason to disable this feature. Still, you can do so by disabling this option.

Directory for run-logs (User)

4.0+

By default, run-logs created during interactive use of QF-Test are stored in the directory `.qftest` below the current user's home directory. This option can be used to specify a different target directory.

Note

This option is interpreted by QF-Test whenever a test is started. At that point, test-suite and global variable bindings are already in place and in contrast to other options it is possible to use QF-Test variable syntax here. This includes special variables like `${env:HOME}` to look up an environment variable or even `${qftest:suite.dir}` to save the run-log next to the test-suite. If the directory is dynamic, like in the latter case, QF-Test may not be able to clean up old run-logs regularly. Errors in variable expansion are silently ignored and the default directory `.qftest` is used instead.

Show expanded variable values in tree nodes (User)

3.1+

The nodes in the tree view of a run-log can either be displayed with variables expanded to the value they had at run-time or with the original variables. Both views have their use, so you can toggle between them via this option or more quickly via the menu item **View→Show nodes expanded**.

Find next error function locates (User)

The search for an error in a run-log, accessible through the **Edit→Find next error** menu item or by pressing **Control-N**, is a very useful utility. This option determines whether the search stops at exceptions only, errors and exceptions or even warnings.

Skip suppressed errors (User)

Like the previous one, this option controls the search for errors in a run-log. If activated, warnings, errors or exceptions that have not propagated to the top, are not found. Thus exceptions caught by a `Try(405)/Catch(408)` clause or messages suppressed through the `Maximum error level(335)` attribute are skipped.

This option is also accessible through the **Edit→Skip suppressed errors** menu item.

Cleanup tree when jumping to next or previous error (User)

When repeatedly jumping to errors in a run-log the tree can easily get cluttered with many expanded nodes. If this option is activated, QF-Test will automatically clean up the tree each time you navigate to an error so that only the parents of the error node are expanded.

3.0+

Note

When viewing split run-logs, partial run-logs containing an error will remain in memory as long as their nodes are expanded. Keeping this option activated will ensure that partial run-logs will be released as soon as possible, keeping memory use manageable even viewing the errors of a very large run-logs.

Save compressed run-logs (*.qrz) (System)

Server script name: `OPT_LOG_SAVE_COMPRESSED`

Run-logs can either be saved as plain or as compressed XML files. For large run-logs without screenshots the compression factor can be as high as 10, so it's advisable to use compressed logs where possible. The only reason not to use compression is if you want to transform the XML run-log afterwards. But even then compressed run-logs are an option because the compression method used is standard gzip format, so converting to and from compressed run-logs can easily be done using `gzip`.

When saving a log-file interactively you can always switch between compressed or non-compressed format by choosing the appropriate filter or by giving the file the extension `.qrz` or `.qrl`.

In batch mode, the default run-log format is compressed. To create an uncompressed run-log, simply specify the extension `.qrl` in the parameter for the `-runlog [<file>]`⁽⁵⁸²⁾ command line argument.

Create split run-logs (User)

3.0+

Server script name: `OPT_LOG_SAVE_SPLIT`

A run-log can be split into several parts by setting the Name for separate run-log⁽³⁵⁹⁾ attribute of a Data driver⁽³⁵⁷⁾ or any of the various test nodes. By turning this option off you can temporarily disable support for split run-logs in order to get a normal, single run-log without having to modify any Name for separate run-log attributes.

See section 8.1.4⁽⁷²⁾ for further information about split run-logs.

Save split run-logs as ZIP files (*.qzp) (User)

3.0+

Server script name: `OPT_LOG_SAVE_SPLIT_ZIP`

Split run-logs can either be saved as a single ZIP file with the extension `.qzp`, containing the main run-log and all partial logs together, or as a normal `.qrl` or `.qrz` run-log that is accompanied by a directory with the same base name and the suffix `_logs`, e.g. the file `runlog.qrz` plus the directory `runlog_logs`. This option determines the format in which split run-logs are created in interactive mode. It has no effect if the option Automatically save run-logs⁽³⁰¹⁾ is turned off.

See section 8.1.4⁽⁷²⁾ for further information about split run-logs.

Minimum size for automatic splitting (kB) (System)

3.4+

Server script name: `OPT_LOG_AUTO_SPLIT_SIZE`

Split run-logs are the only reliable way to prevent running out of memory during very long running tests or when the run-log grows quickly due to screenshots or output from the SUT. They are also more efficient when transforming run-logs into reports. However, explicit setting of Name for separate run-log attributes requires an understanding of the issues involved and either making decisions about where best to split a run-log or tedious typing when trying to split into small pieces.

As a compromise, QF-Test makes a very rough calculation about the size of a run-log during executing, taking screenshots and program output into account. When execution of a Test-case or Test-set has finished and the approximate size of the run-log pertaining to that node is larger than the threshold specified in this option, the run-log is split off and saved automatically. A value of 0 prevents automatic splitting.

See [section 8.1.4^{\(72\)}](#) for further information about split run-logs.

Name for automatically split 'Test-case' run-log (System)

Server script name: OPT_LOG_AUTO_SPLIT_TESTCASE_NAME

This option specifies the name to use for an external log when it is split off automatically after execution of a Test-case has finished as described in the previous option. Variables can be used as well as the '%...' placeholders documented for the attribute [Name for separate run-log^{\(317\)}](#).

The special variable \${qftest:testcase.splitlogname} is a good base. It expands to a path name created from the name of the Test-case with possible parent Test-set nodes as directories.

See [section 8.1.4^{\(72\)}](#) for further information about split run-logs.

Name for automatically split 'Test-set' run-log (System)

Server script name: OPT_LOG_AUTO_SPLIT_TESTSET_NAME

This option specifies the name to use for an external log when it is split off automatically after execution of a Test-set has finished as described in the option [Minimum size for automatic splitting \(kB\)^{\(303\)}](#). Variables can be used as well as the '%...' placeholders documented for the attribute [Name for separate run-log^{\(324\)}](#).

The special variable \${qftest:testset.splitlogname} is a good base. It expands to a path name created from the name of the Test-set with possible parent Test-set nodes as directories.

See [section 8.1.4^{\(72\)}](#) for further information about split run-logs.

3.4+

3.4+

29.6.2 Options determining run-log content

The screenshot shows a dialog box titled "Content of run-logs". It contains the following options:

- ☒ Log variable expansion
- ☐ Log parent nodes of components
- Log level for the SUT: Warnings and errors (dropdown menu)
- Number of events to log for error diagnosis: 0 (text input)
- Maximum number of errors with screenshots per run-log: 3 (text input)
- ☒ Count screenshots individually for each split log
- ☒ Create screenshots of the whole screen
- ☒ Create screenshots of individual windows
- ☐ Log successful advanced image checks
- ☒ Create compact run-log
- ☐ Don't create run-log
- ☒ Log SUT output individually
- ☒ Compactify individually logged SUT output

Figure 29.28: Options determining run-log content

Log variable expansion (System)

Server script name: OPT_LOG_CONTENT_VARIABLE_EXPANSION

If this option is activated, every variable expansion⁽⁵⁷⁾ is logged.

Log parent nodes of components (System)

Server script name: OPT_LOG_CONTENT_PARENT_COMPONENTS

Setting this option will cause all direct and indirect parent nodes to be logged in addition to the target component node for every event, check, etc.

Log level for the SUT (System)

Server script name: OPT_LOG_CONTENT_SUT_LEVEL

Possible Values: VAL_LOG_SUT_LEVEL_MESSAGE,

VAL_LOG_SUT_LEVEL_WARNING,
VAL_LOG_SUT_LEVEL_ERROR

The level for automatically generated messages in the SUT during replay, e.g. details for component recognition. Only messages with the respective level, i.e. plain messages, warnings or errors will be logged. This option has no effect on messages created explicitly via `rc.logMessage` or `qf.logMessage`.

Number of events to log for error diagnosis (System)

SUT script name: OPT_LOG_CONTENT_DIAGNOSIS

During replay of a test QF-Test logs various events and other things going on behind the scenes. This information is quickly discarded except when an error happens. In that case the most recent events are written to a special run-log node. The information may also be useful to developers but is mostly required for error diagnosis when requesting support from Quality First Software GmbH.

This option determines the number of recent internal events to keep. Setting it to 0 disables the feature altogether. You should not set this value to less than about 400 without a good reason. Because the information is logged only for errors, the cost for gathering it is minimal.

Maximum number of errors with screenshots per run-log (System)

Server script name: OPT_LOG_CONTENT_SCREENSHOT_MAX

The maximum number of screenshots that QF-Test takes and stores in the run-log during a test-run on situations of exception or errors. Setting this value to 0 disables taking screenshots entirely, a negative value means unlimited screenshots.

Count screenshots individually for each split log (System)

Server script name: OPT_LOG_CONTENT_SCREENSHOT_PER_SPLIT_LOG

If this option is set, each partial log of a split run-log may contain the maximum number of screenshots defined above without affecting the count for the main run-log. Otherwise the limit applies for the sum of all parts belonging to the same main run-log.

See [section 8.1.4^{\(72\)}](#) for further information about split run-logs.

Create screenshots of the whole screen (System)

Server script name: OPT_LOG_CONTENT_SCREENSHOT_FULLSCREEN

Activating this option causes QF-Test to take an image of the whole screen and save it in the run-log when a screenshot is triggered by an exception or error.

Create screenshots of individual windows (System)

Server script name: OPT_LOG_CONTENT_SCREENSHOT_WINDOW

Activating this option causes QF-Test to record images of all windows and dialogs of the SUT and store them in the run-log when screenshots are triggered due to exceptions or errors. Most of the time this will work even for windows that are covered by other windows or in cases where a full screenshot is not possible, for example when a screen is locked.

Log successful advanced image checks (System)

Server script name: OPT_LOG_CONTENT_SCREENSHOT_ADVANCED_ALWAYS

If this option is activated, QF-Test will store the expected and actual images as well as the transformed images for successful advanced image checks in the run-log. Otherwise these details are kept for failed image checks only.

Activating this option can raise the size of the run-log drastically so be sure to use it in combination with compact run-logs and split run-logs.

Create compact run-log (System)

Server script name: OPT_LOG_CONTENT_COMPACT

Activating this option causes QF-Test to discard every node from a run-log that is neither relevant for error diagnosis, nor for the XML/HTML report. After an error or exception, as well as at the end of a test-run, the 100 most recent nodes are not discarded, so all the relevant information should remain available.

Large tests should almost always be run with this option enabled to reduce the risk of running out of memory due to excessively large run-logs.

This option is used only when QF-Test is run in interactive mode. It is ignored in batch mode (see [section 1.6^{\(7\)}](#)) to avoid accidental loss of information. To create a compact run-log in batch mode, use the `-compact(574)` command line argument.

Don't create run-log (System)

Server script name: OPT_LOG_CONTENT_SUPPRESS

For very-long-running tests or demos that are run in an endless loop, memory consumption of the run-log is an issue, but split run-logs are an ideal solution. Before split run-logs were available, turning run-logs off completely via this option was sometimes the only way to get long-running tests to work. Now this option is only retained for backwards compatibility.

In batch mode this option is ignored. To suppress the run-log, use the argument

3.4+

Note

`-nolog(577)`.

Log SUT output individually (System)

3.0+

SUT script name: OPT_LOG_CONTENT_SPLIT_IO

If set, any text that an SUT client prints to its stdout or stderr stream is also logged in the run-log. For each interaction with the SUT QF-Test collects text printed before the event and after the event during synchronisation. This makes it possible to associate output like an exception stacktrace that is triggered by an event with the event itself, something that is impossible if all output is kept in a single piece.

Compactify individually logged SUT output (System)

3.0+

Server script name: OPT_LOG_CONTENT_COMPACTIFY_SPLIT_IO

Output from an SUT client tends to accumulate and can consume a lot of memory. If this option is activated, individually logged SUT output for events that are no longer of interest can be removed along with the events in compact run-logs. Please see the option [Create compact run-log^{\(307\)}](#) for further information about compact run-logs.

29.6.3 Options for mapping between directories with test-suites

4.0+

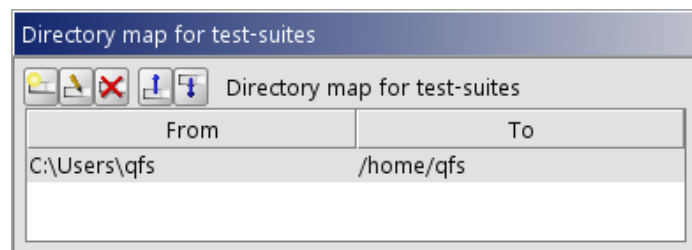


Figure 29.29: Options for mapping between directories with test-suites

Directory map for test-suites (System)

When analyzing a run-log it is often necessary to quickly switch between the run-log and the respective test-suite. However, when running automated tests on different systems like Windows and Linux, the directories from which test-suites are loaded during the test vary and there is no automatic way to map between different directory layouts.

With this option you can assist QF-Test in locating test-suites. The 'From' column is a glob pattern that must match from the beginning of the path of the test-suite stored in the run-log to the end of some directory in that path. The 'To' column is the respective replacement for the matching part and can also contain a glob pattern. When searching for the test-suite, QF-Test processes this list top to bottom, performing the replacement for every match found and the first match leading to an actual test-suite is used.

Note

A glob patterns is a simpler form of a regular expression often used by development tools: An '*' stands for any number of characters up to the next file separator while '**' means 0 or more characters of any kind, including '/'. Some examples:

****/test-suites**

All directories named `test-suites` at any depth.

T:/test/sut_*

All directories starting with `sut_` in the `T:/test` directory.

29.7 Variables

The following options pertain to variable binding.

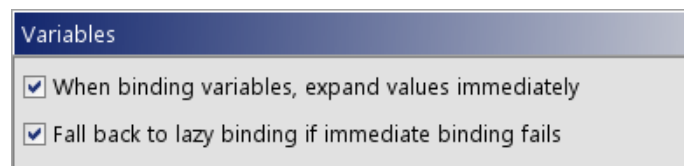


Figure 29.30: Variable options

When binding variables, expand values immediately (System)

3.0+

Server script name: `OPT_VARIABLE_IMMEDIATE_BINDING`

When a set of variable bindings is pushed onto a variable stack, any additional variable references in the values of these variables can either be expanded immediately, or the values can be left unchanged to be expanded lazily as needed. This is explained in detail in [section 7.7^{\(66\)}](#).

For immediate expansion turn this option on, for lazy expansion turn it off.

Fall back to lazy binding if immediate binding fails (System)**3.0+**

Server script name: OPT_VARIABLE_LAZY_FALLBACK

Old tests that rely on lazy variable binding may fail with the new default of immediate binding. If this option is activated, variables that cannot be bound immediately due to references to not-yet-defined variables are bound lazily instead and a warning is issued. To get rid of the warning simply change the value of the respective variable to use explicit lazy binding with '\$_'. Please see [section 7.7^{\(66\)}](#) for further information.

Furthermore, various kinds of variables can be defined here. These are explained in [chapter 7^{\(57\)}](#).

Chapter 30

Elements of a test-suite

30.1 The test-suite and its structure

There are more than 60 different kinds of nodes for a test-suite which are all listed in this chapter. Each node type has a unique set of features. The attributes of a node are displayed and edited in the detail view of the editor. The restrictions that apply to each attribute are listed as well as whether it supports variable expansion (see [chapter 7^{\(57\)}](#)).

Additional features of a node include the behavior of the node during execution of a test and the kinds of parent and child nodes allowed.

30.1.1 Test-suite



The root node of the tree represents the test-suite itself. Its basic structure is fixed. The root node contains an arbitrary number of Tests⁽³³⁰⁾, followed by the Procedures⁽³⁸⁷⁾ the Extras⁽³⁴²⁾ and the Windows and components⁽⁵⁵⁶⁾. When executed, the top-level Tests are executed one by one.

Contained in: None

Children: An arbitrary number of Test nodes, followed by the Procedures, Extras and Windows and components nodes.

Execution: The top-level Test nodes are executed one by one.

Attributes:

The screenshot shows a 'Test-suite' dialog box with the following sections:

- Include files:** A section with icons for adding, removing, and refreshing files. Below it is a text field containing 'lib.qft'.
- Dependencies (reverse includes):** A section with similar icons and an empty text field below it.
- Variable definitions:** A section with icons and a table with two columns: 'Name' and 'Value'. The table contains one row with 'rootdir' and 'some/directory'.
- Comment:** A section with an icon and an empty text area below it.

Figure 30.1: Test-suite attributes

Include files

This is a list of test-suites that are included by the suite. If a Component⁽⁵⁴⁵⁾ or Procedure⁽³⁷⁸⁾ reference cannot be resolved in the current suite, the included suites are searched for it. When recording new components, QF-Test will search the included suites for a matching Component node before creating a new one.

Relative pathnames are treated relative to the directory of the suite, or to a directory on the library path (see option Directories holding test-suite libraries⁽²⁵²⁾).

When you change anything in this attribute QF-Test will offer to update all affected nodes to compensate for the change. For example, if you add or remove a suite from the includes, QF-Test will make all references to that suite's Procedures or Components implicit or absolute so that the actual referenced nodes remain unchanged. In such a case, choose "Yes". If, on the other hand, you renamed a suite or moved it to some other directory and are simply updating the includes to reflect that, chose "No" so all former implicit references into the old suite will now point to the new one.

Variable: No

Restrictions: None

Dependencies (reverse includes)

This list of test-suites is the reverse of the Include files attribute. It has no impact

on test execution. Instead it serves a hint to tell QF-Test which test-suites depend on Components in this suite, either because they (directly or indirectly) include this suite or because they explicitly reference Components in it. This information is used when QF-Test IDs of Components are manipulated (for example after updating components, see (section 5.8⁽⁴⁷⁾)) and the QF-Test component ID attributes of nodes depending on these components have to be updated. QF-Test always checks all currently loaded suites for dependencies. Additionally, it will automatically load and check all suites listed here.

Relative pathnames are treated relative to the directory of the suite, or to a directory on the library path (see option Directories holding test-suite libraries⁽²⁵²⁾).

Note

Like the Include files, the Dependencies are also resolved indirectly. For example if suite A has the dependency B which has the dependency C, both suites B and C will be loaded and checked for references automatically when components in suite A are manipulated.

Variable: No

Restrictions: None

Variable definitions

These variables definitions are identical to the suite variable bindings accessible in the Variables⁽³⁰⁹⁾ pane of the global options. A detailed explanation of variable definition and lookup is given in chapter 7⁽⁵⁷⁾. See section 2.2.5⁽¹³⁾ about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.2 Test and Sequence nodes

Tests and sequences are the main structural elements of a test-suite. Test-case⁽³¹⁴⁾ nodes and their Test⁽³³⁰⁾ node predecessors represent logical test-cases and are implemented as specialized sequences. A sequence is a container that executes its child nodes one by one. It can define variables (see chapter 7⁽⁵⁷⁾) that remain bound during the execution of the children.

Other kinds of sequences differ either in the way their child nodes are executed, or in the restrictions they impose on their child or parent nodes.

30.2.1 Test-case



A Test-case node represents one or more logical test cases. In a sense it is the most important of all QF-Test nodes and everything else only serves to lend structure to Test-cases or to implement their logic. Functionally it is a highly specialized Sequence⁽³³⁴⁾ with a number of important extensions and special attributes.

A Test-case should focus on the actual test it supposed to perform. Setup and cleanup tasks required to ensure that the Test-case executes in the required environment and does not interfere with subsequent tests should be implemented in the form of Dependencies⁽³⁴⁴⁾ as described in chapter 12⁽⁹⁰⁾. Alternatively - or in addition to Dependencies - a Test-case can have Setup⁽³⁵⁰⁾ and Cleanup⁽³⁵²⁾ nodes to be executed before and after the Test-case.

Because a Test-case can be called from a Test call⁽³²⁷⁾ node it is also somewhat similar to a Procedure⁽³⁷⁸⁾ in that its Name⁽³¹⁶⁾ attribute is mandatory and that it has a list of Parameter default values⁽³¹⁹⁾ that can be overridden in the calling node.

Test-cases also play a central role in run-logs and test reports. During a test-run a Test-case node can be executed several times in different contexts and with different parameters. Logically these executions may represent the same or different test-cases. By defining a set of Characteristic variables⁽³¹⁹⁾ you can specify which variable values are used to differentiate between executions, thus characterizing the run-time environment of the test. The values of these variables at the time of entry to the Test-case are stored in the run-log. To emphasize the fact that each execution of a Test-case node may represent a separate logical test-case there is an alternative name attribute called Name for run-log and report⁽³¹⁷⁾. Its value may contain references to the Characteristic variables of the test. In the run-log or report the test will then be listed with this name, including the expanded run-time variable values.

Finally, there are situations in which a test cannot or should not be executed for specific variable settings. If the Condition⁽³¹⁸⁾ attribute of the Test-case is defined, the Test-case will only be executed if that expression expands to a true value. If the Test-case is not

executed due to the Condition it will be listed as *skipped* in the report.

Contained in: Test-suite⁽³¹¹⁾, Test-set⁽³²¹⁾.

Children: Optional Dependency⁽³⁴⁴⁾ or Dependency reference⁽³⁴⁷⁾ as the first element. A Setup⁽³⁵⁰⁾ may be the next and a Cleanup⁽³⁵²⁾ the last node with an arbitrary number of normal child nodes in between. A Test-case that does not contain normal child nodes will be listed as *not implemented* in the report.

Execution: First the Variable definitions⁽³¹⁹⁾ of the Test-case are bound on the primary and its Parameter default values⁽³¹⁹⁾ on the fallback variable stack (see chapter 7⁽⁵⁷⁾). With these in place the Condition⁽³¹⁸⁾ is evaluated and the node will be skipped if a non-empty Condition evaluates to false. Next the dependency of the Test-case - possibly inherited from its parent node - is determined and resolved as described in chapter 12⁽⁹⁰⁾. Then the optional Setup is executed once, followed by the normal child nodes and a single execution of the optional Cleanup. An exception raised during the course of the Test-case will be caught and passed to the Dependency for handling in a Catch⁽⁴⁰⁸⁾ node. Even if the exception is not handled by a Dependency it is not propagated beyond the Test-case to prevent aborting the whole test-run. The error state is duly noted in run-log and report however.

Attributes:

Test-case

Name
Testcase

Name for run-log and report

Name for separate run-log

☒ Inherit dependency from parent node

Condition

Expected to fail if...

Characteristic V. (0) Defaults (0) Variables (0)

Characteristic variables

Name

Maximum error level
Exception

☐ Border for relative calls

QF-Test ID

Delay before (ms) Delay after (ms)
/

Comment

Figure 30.2: Test-case attributes

Name

A Test-case is identified by its name and the names of its Test-set⁽³²¹⁾ ancestors, so you should assign a name with a meaning that is easy to recognize and remember.

Variable: No

Restrictions: Must not be empty or contain the characters '.' or '#'.

Name for run-log and report

A separate name to be used for run-log and report. This is useful to differentiate between multiple executions with potentially different values for the Characteristic variables.

Variable: Yes

Restrictions: None

Name for separate run-log

If this attribute is set it marks the node as a breaking point for split run-logs and defines the filename for the partial log. When the node finishes, the respective log entry is removed from the main run-log and saved as a separate, partial run-log. This operation is completely transparent, the main run-log retains references to the partial logs and is fully controllable. Please see [section 8.1.4^{\(72\)}](#) for further information about split run-logs.

This attribute has no effect if the option [Create split run-logs^{\(303\)}](#) is disabled or split run-logs are explicitly turned off for batch mode via the [-splitlog^{\(583\)}](#) command line argument.

There is no need to keep the filename unique. If necessary, QF-Test appends a number to the filename to avoid collisions. The filename may contain directories and, similar to specifying the name of a run-log in batch mode on the command line, the following placeholders can be used after a '%' or a '+' character:

Character	Replacement
%	Literal '%' character.
+	Literal '+' character.
i	The current runid as specified with <code>-runid <ID></code> ⁽⁵⁸²⁾ .
r	The error level of the partial log.
w	The number of warnings in the partial log.
e	The number of errors in the partial log.
x	The number of exceptions in the partial log.
t	The thread index to which the partial log belongs (for tests run with parallel threads).
y	The current year (2 digits).
Y	The current year (4 digits).
M	The current month (2 digits).
d	The current day (2 digits).
h	The current hour (2 digits).
m	The current minute (2 digits).
s	The current second (2 digits).

Table 30.1: Placeholders for the Name for separate run-log attribute

Variable: Yes

Restrictions: None, characters that are illegal for a filename will be replaced with `'_'`.

Inherit dependency from parent node

This option allows inheriting the Dependency⁽³⁴⁴⁾ from the parent node as a replacement for or in addition to specifying a Dependency for the node itself.

Variable: No

Restrictions: None

Condition

If the condition is non-empty it will be evaluated and if the result is false the execution of the current node is aborted. In this case the node will be reported as skipped.

Like the Condition⁽³⁹⁷⁾ of an If⁽³⁹⁶⁾ node, the Condition is evaluated by the Jython interpreter, so the same rules apply.

Variable: Yes

Restrictions: Valid syntax

Expected to fail if...

This attribute allows specifying a condition under which the Test-case is expected to fail. This is helpful to distinguish new problems from already known ones. Of course the latter should be fixed, but if and when that happens may be outside the tester's sphere of influence.

Most of the time it is sufficient to simply set this attribute to 'true' to mark the Test-case as an expected failure. But if the Test-case is executed multiple times, e.g. within a Data driver⁽³⁵⁷⁾ or on multiple systems and fails only in specific cases, the condition should be written so that it evaluates to true for exactly those cases, e.g. `{qftest:windows}` for a test that fails on Windows but runs fine on other systems.

If this attribute evaluates to true and the Test-case fails with an error, it will be listed separately in the run-log, report and on the status line. It still means that there is an error in the application, so the overall percentage of successful tests is not changed.

It is treated as an error if this attribute evaluates to true and the Test-case does not fail because this means that either the test is not able to reproduce the problem reliably or that the problem has been fixed and the Test-case must be updated accordingly.

Variable: Yes

Restrictions: Valid syntax

Characteristic variables

These variables are part of the characteristics of a Test-set or Test-case. Two executions of a Test-case are considered to represent the same logical test-case if the run-time values of all Characteristic variables are identical. The run-time values of the Characteristic variables are stored in the run-log.

Variable: No

Restrictions: None

Parameter default values

Here you can define default or fallback values for the Test-case's parameters (see chapter 7⁽⁵⁷⁾). Defining these values also serves as documentation and is a valuable time-saver when using the dialog to select the Test-case for the Test name⁽³²⁸⁾ attribute of a Test call⁽³²⁷⁾. See section 2.2.5⁽¹³⁾ about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Variable definitions

These variables are bound on the direct bindings stack (see chapter 7⁽⁵⁷⁾). They remain valid during the execution of the Test-case's child nodes and cannot be

overridden by a Test call⁽³²⁷⁾ node. See section 2.2.5⁽¹³⁾ about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run-log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run-log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run-log node for a sequence will propagate, can be limited.

Note

This value has no effect on the way exceptions are handled. It only affects the error states of the run-log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.6⁽⁷⁾). It also has no effect on the creation of compact run-logs (see command line argument -compact⁽⁵⁷⁴⁾). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

Border for relative calls

This flag determines whether relative procedure calls, test calls or dependency references are allowed within that certain node. Relative calls passing that border are not allowed. If that attribute is not specified in the hierarchy, no relative calls are allowed.

Variable: No

Restrictions: None

QF-Test ID

When using the command line argument -test <n>|<ID>⁽⁵⁸⁴⁾ for execution in batch mode you can specify the QF-Test ID of the node as an alternative to its qualified name.

Variable: Nein

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.2.2 Test-set



The main purpose of a Test-set is to give structure to the Test-cases⁽³¹⁴⁾ of a test-suite. Test-sets can be nested. The Name⁽³²³⁾ of a Test-set is part of the Test-case's fully qualified name to which a Test call⁽³²⁷⁾ node refers. Test-sets are also callable themselves and thus have a set of Parameter default values⁽³²⁵⁾ that can be overridden in the Test call.

One way of structuring Test-cases is to collect those with similar setup and cleanup requirements in the same Test-set which can provide a Dependency⁽³⁴⁴⁾ or Dependency reference⁽³⁴⁷⁾ node to be inherited by the Test-case or nested Test-set child nodes. If the Inherit dependency from parent node⁽³²⁵⁾ attribute of the Test-set is set, the Dependency nodes of the Test-set's parent(s) are also inherited. See chapter 12⁽⁹⁰⁾ for information about QF-Test's dependency mechanism. Alternative or additional setup and cleanup is available in the form of Setup⁽³⁵⁰⁾ and Cleanup⁽³⁵²⁾ nodes which will be executed before and after each of the Test-set and Test-case child nodes.

Another important feature of a Test-set is data-driven testing. This can be achieved by adding a Data driver⁽³⁵⁷⁾ node with one or more Data binders as described in chapter 14⁽¹²¹⁾.

Like a Test-case, a Test-set plays an important role for reports. As it can also be executed several times with different parameter settings, it has a set of Characteristic variables⁽³²⁵⁾ and an alternative Name for run-log and report⁽³²³⁾ that work just like for a Test-case. The same is true for the Condition⁽³²⁵⁾ which can be used to skip an entire Test-set depending on current variable values.

Contained in: Test-suite⁽³¹¹⁾, Test-set⁽³²¹⁾.

Children: Optional Dependency⁽³⁴⁴⁾ or Dependency reference⁽³⁴⁷⁾ as the first child, followed by an optional Data driver⁽³⁵⁷⁾. A Setup⁽³⁵⁰⁾ may come next and a Cleanup⁽³⁵²⁾ at the end with an arbitrary number of Test-set⁽³²¹⁾, Test-case⁽³¹⁴⁾ and Test call⁽³²⁷⁾ nodes in between.

Execution: First the Parameter default values⁽³²⁵⁾ of the Test-set are bound on the fallback variable stack (see chapter 7⁽⁵⁷⁾). With these in place the Condition⁽³²⁵⁾ is evaluated and the node will be skipped if a non-empty Condition evaluates to false. The Dependency⁽³⁴⁴⁾ or Dependency reference⁽³⁴⁷⁾ node will not be resolved at this time unless its Always execute, even in test-suite and test-set nodes⁽³⁴⁵⁾ attribute is set. If there is a Data driver⁽³⁵⁷⁾ node, it is executed to create a data driving context, bind one or more Data binders and then repeatedly execute the other child nodes as described in chapter 14⁽¹²¹⁾. For each loop iteration - or once in case no Data driver is present - the Test-set executes each of its Test-set or Test-case child nodes. If an optional Setup⁽³⁵⁰⁾ or Cleanup⁽³⁵²⁾ node are present, they are executed before and after each of these child nodes respectively, just like in a Test⁽³³⁰⁾ node.

Attributes:

The screenshot shows a 'Test-set' dialog box with the following fields and options:

- Name:** A text field containing 'Testarea'.
- Name for run-log and report:** An empty text field.
- Name for separate run-log:** An empty text field.
- Inherit dependency from parent node:** A checked checkbox.
- Condition:** An empty text field.
- Characteristic V. (0) Defaults (0):** A section containing:
 - Characteristic variables:** A list box with a 'Name' header and an empty text area below it.
- Maximum error level:** A dropdown menu currently showing 'Exception'.
- Border for relative calls:** An unchecked checkbox.
- QF-Test ID:** An empty text field.
- Delay before (ms):** An empty text field.
- Delay after (ms):** An empty text field.
- Comment:** A text area with a small icon to its left.

Figure 30.3: Test-set attributes

Name

The name of a Test-set is part of its own identification and of that of the Test-case⁽³¹⁴⁾ and Test-set nodes it contains, so you should assign a name with a meaning that is easy to recognize and remember.

Variable: No

Restrictions: Must not be empty or contain the characters '.' or '#'.

Name for run-log and report

A separate name to be used for run-log and report. This is useful to differentiate between multiple executions with potentially different values for the Characteristic variables.

Variable: Yes

Restrictions: None

Name for separate run-log

If this attribute is set it marks the node as a breaking point for split run-logs and defines the filename for the partial log. When the node finishes, the respective log entry is removed from the main run-log and saved as a separate, partial run-log. This operation is completely transparent, the main run-log retains references to the partial logs and is fully controllable. Please see [section 8.1.4^{\(72\)}](#) for further information about split run-logs.

This attribute has no effect if the option [Create split run-logs^{\(303\)}](#) is disabled or split run-logs are explicitly turned off for batch mode via the [-splitlog^{\(583\)}](#) command line argument.

There is no need to keep the filename unique. If necessary, QF-Test appends a number to the filename to avoid collisions. The filename may contain directories and, similar to specifying the name of a run-log in batch mode on the command line, the following placeholders can be used after a '%' or a '+' character:

Character	Replacement
%	Literal '%' character.
+	Literal '+' character.
i	The current runid as specified with -runid <ID>⁽⁵⁸²⁾ .
r	The error level of the partial log.
w	The number of warnings in the partial log.
e	The number of errors in the partial log.
x	The number of exceptions in the partial log.
t	The thread index to which the partial log belongs (for tests run with parallel threads).
y	The current year (2 digits).
Y	The current year (4 digits).
M	The current month (2 digits).
d	The current day (2 digits).
h	The current hour (2 digits).
m	The current minute (2 digits).
s	The current second (2 digits).

Table 30.2: Placeholders for the Name for separate run-log attribute

Variable: Yes

Restrictions: None, characters that are illegal for a filename will be replaced with `'_'`.

Inherit dependency from parent node

This option allows inheriting the Dependency⁽³⁴⁴⁾ from the parent node as a replacement for or in addition to specifying a Dependency for the node itself.

Variable: No

Restrictions: None

Condition

If the condition is non-empty it will be evaluated and if the result is false the execution of the current node is aborted. In this case the node will be reported as skipped.

Like the Condition⁽³⁹⁷⁾ of an If⁽³⁹⁶⁾ node, the Condition is evaluated by the Jython interpreter, so the same rules apply.

Variable: Yes

Restrictions: Valid syntax

Characteristic variables

These variables are part of the characteristics of a Test-set or Test-case. Two executions of a Test-case are considered to represent the same logical test-case if the run-time values of all Characteristic variables are identical. The run-time values of the Characteristic variables are stored in the run-log.

Variable: No

Restrictions: None

Parameter default values

Here you can define default or fallback values for the Test-set's parameters (see chapter 7⁽⁵⁷⁾). Defining these values also serves as documentation and is a valuable time-saver when using the dialog to select the Test-set for the Test name⁽³²⁸⁾ attribute of a Test call⁽³²⁷⁾. See section 2.2.5⁽¹³⁾ about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run-log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run-log node represents the worst of its child nodes' states. Using this attribute,

the maximum error state that the run-log node for a sequence will propagate, can be limited.

Note

This value has no effect on the way exceptions are handled. It only affects the error states of the run-log nodes and by that the exit code of QF-Test when run in batch mode (see. [section 1.6^{\(7\)}](#)). It also has no effect on the creation of compact run-logs (see command line argument `-compact(574)`). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

Border for relative calls

This flag determines whether relative procedure calls, test calls or dependency references are allowed within that certain node. Relative calls passing that border are not allowed. If that attribute is not specified in the hierarchy, no relative calls are allowed.

Variable: No

Restrictions: None

QF-Test ID

When using the command line argument `-test <n>|<ID>(584)` for execution in batch mode you can specify the QF-Test ID of the node as an alternative to its qualified name.

Variable: Nein

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets

you define an external editor in which comments can be edited conveniently by

pressing **[Alt-Return]** or by clicking the  button.

Variable: Yes

Restrictions: None

30.2.3 Test call



With this node a call to some other executable test node can be made. Possible targets are Test-suite⁽³¹¹⁾, Test-set⁽³²¹⁾ and Test-case⁽³¹⁴⁾ nodes in the same or a different test-suite. Execution will continue in the called node and when finished return to the Test call and thus to its parent node.

The name of a Test-case or Test-set to call is determined by its Name⁽³¹⁶⁾ and the Names⁽³²³⁾ of its Test-set ancestors. These are concatenated with a dot ('.') as separator, starting with the outermost Test-set and ending in the Test-case's name. Thus to call a Test-case named `nodeTest` in a Test-set named `Tree` that is itself a child of a Test-set named `Main`, the Test name⁽³²⁸⁾ attribute would be set to `'Main.Tree.nodeTest'`. A node in a different test-suite is addressed by prepending the filename of the test-suite followed by a '#' to the Test name. A Test-suite node is addressed by a single '.', so calling a whole test-suite is done with a Test name attribute of the form `'suiteName.qft#.'`. It is generally easiest to pick the target node interactively from a dialog by clicking on the button above the Test name attribute.

See also section 17.1⁽¹⁵⁰⁾ for further information about cross-suite calls.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The Variable definitions⁽³²⁹⁾ of the Test call are bound, the target node is determined and execution is passed to it. After the Test call returns, the Test call's variables are unbound again.

Attributes:

The 'Test call' dialog box contains the following fields and sections:

- Test name:** A text field containing 'Testarea.Testcase' with a selection icon to its left.
- Name for separate run-log:** An empty text field.
- Variable definitions:** A section with icons for adding, deleting, and copying variables, followed by a table with two columns: 'Name' and 'Value'.
- QF-Test ID:** An empty text field.
- Delay before (ms):** An empty text field.
- Delay after (ms):** An empty text field.
- Comment:** A text area with a comment icon to its left.

Figure 30.4: Test call Attributes

Name

In case of a Test-case or Test-set it is the full name, created from the Name⁽³²³⁾ of its Test-set parents and its own Name⁽³¹⁶⁾, joined by a dot. A Test-suite node is addressed by a single '.' character. The button above the attribute brings up a dialog in which you can select the target node interactively. You can also get to this dialog by pressing **[Shift-Return]** or **[Alt-Return]** when the focus is in the text field. By selecting the "Copy parameters" checkbox you can adopt the Test-case's or Test-set's Parameter default values as parameters for the Test call node to save typing.

Variable: Yes

Restrictions: Must not be empty.

Name for separate run-log

If this attribute is set it marks the node as a breaking point for split run-logs and defines the filename for the partial log. When the node finishes, the respective log entry is removed from the main run-log and saved as a separate, partial run-log. This operation is completely transparent, the main run-log retains references to the partial logs and is fully controllable. Please see section 8.1.4⁽⁷²⁾ for further information about split run-logs.

This attribute has no effect if the option `Create split run-logs`⁽³⁰³⁾ is disabled or split run-logs are explicitly turned off for batch mode via the `-splitlog`⁽⁵⁸³⁾ command line argument.

There is no need to keep the filename unique. If necessary, QF-Test appends a number to the filename to avoid collisions. The filename may contain directories and, similar to specifying the name of a run-log in batch mode on the command line, the following placeholders can be used after a '%' or a '+' character:

Character	Replacement
%	Literal '%' character.
+	Literal '+' character.
i	The current runid as specified with <code>-runid <ID></code> ⁽⁵⁸²⁾ .
r	The error level of the partial log.
w	The number of warnings in the partial log.
e	The number of errors in the partial log.
x	The number of exceptions in the partial log.
t	The thread index to which the partial log belongs (for tests run with parallel threads).
y	The current year (2 digits).
Y	The current year (4 digits).
M	The current month (2 digits).
d	The current day (2 digits).
h	The current hour (2 digits).
m	The current minute (2 digits).
s	The current second (2 digits).

Table 30.3: Placeholders for the Name for separate run-log attribute

Variable: Yes

Restrictions: None, characters that are illegal for a filename will be replaced with ' ', '_'.

Variable definitions

This is where you define the parameter values for the target node. These variables are bound on the primary variable stack (see chapter 7⁽⁵⁷⁾) so they override any Parameter default values. See section 2.2.5⁽¹³⁾ about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.2.4 Test

Note



Before QF-Test version 2 the Test node was one of the main building blocks of a test-suite. It had a few shortcomings however: Its setup/cleanup structure was too linear and inflexible for complex scenarios and it was never clear whether a Test node represented a logical test-case or was just used to implement some sequence. Thus Test nodes have been replaced with Test-set⁽³²¹⁾ and Test-case⁽³¹⁴⁾ nodes. Tests are still available for backwards compatibility and because they can be useful at lower levels or in situations where Test-set and Test-case nodes are not applicable.

A Test is a special Sequence⁽³³⁴⁾ that executes extra setup and cleanup code before and after the execution of its child nodes to ensure that each of the children runs under similar conditions and to prevent unwanted side effects between the execution of one child and the next. To do so, a Test has two special, optional child nodes besides its normal children, a Setup⁽³⁵⁰⁾ as the first and a Cleanup⁽³⁵²⁾ as the last node.

With its Implicitly catch exceptions⁽³³³⁾ attribute a Test also offers special exception handling to prevent exceptions in one Test from aborting a whole test-run.

For special cases of data driven testing a Test may also contain a Data driver⁽³⁵⁷⁾, whereas such is typically done in combination with Test-sets⁽³²¹⁾ as described in chapter 14⁽¹²¹⁾.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: Optional Data driver⁽³⁵⁷⁾ followed by an optional Setup⁽³⁵⁰⁾ at the beginning, then any kind of executable nodes and an optional Cleanup⁽³⁵²⁾ as last node.

Execution: The Variable definitions⁽³³³⁾ of the Test are bound. If there is a Data driver⁽³⁵⁷⁾ node, it is executed to create a data driving context and bind one or more Data binders for iteration over the determined data sets as described in chapter 14⁽¹²¹⁾. For each of its normal child nodes, the Setup⁽³⁵⁰⁾ is executed, then the child and then the Cleanup⁽³⁵²⁾. After the last execution of the Cleanup is complete, the variables are unbound again.

Attributes:

The screenshot shows a 'Test' dialog box with the following fields and options:

- Name:** Table functionality
- Name for separate run-log:** (empty field)
- Variable definitions:** A table with two columns: 'Name' and 'Value'. The table is currently empty.
- ☒ **Implicitly catch exceptions**
- Maximum error level:** Exception (selected from a dropdown menu)
- QF-Test ID:** (empty field)
- Delay before (ms):** (empty field)
- Delay after (ms):** (empty field)
- Comment:** Test all input and modification methods of the table

Figure 30.5: Test attributes

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and tell something about the function of the sequence.

Variable: No

Restrictions: None

Name for separate run-log

If this attribute is set it marks the node as a breaking point for split run-logs and defines the filename for the partial log. When the node finishes, the respective log entry is removed from the main run-log and saved as a separate, partial run-log. This operation is completely transparent, the main run-log retains references to the partial logs and is fully controllable. Please see [section 8.1.4^{\(72\)}](#) for further information about split run-logs.

This attribute has no effect if the option [Create split run-logs^{\(303\)}](#) is disabled or split run-logs are explicitly turned off for batch mode via the [-splitlog^{\(583\)}](#) command line argument.

There is no need to keep the filename unique. If necessary, QF-Test appends a number to the filename to avoid collisions. The filename may contain directories and, similar to specifying the name of a run-log in batch mode on the command line, the following placeholders can be used after a '%' or a '+' character:

Character	Replacement
%	Literal '%' character.
+	Literal '+' character.
i	The current runid as specified with -runid <ID>⁽⁵⁸²⁾ .
r	The error level of the partial log.
w	The number of warnings in the partial log.
e	The number of errors in the partial log.
x	The number of exceptions in the partial log.
t	The thread index to which the partial log belongs (for tests run with parallel threads).
y	The current year (2 digits).
Y	The current year (4 digits).
M	The current month (2 digits).
d	The current day (2 digits).
h	The current hour (2 digits).
m	The current minute (2 digits).
s	The current second (2 digits).

Table 30.4: Placeholders for the Name for separate run-log attribute

Variable: Yes

Restrictions: None, characters that are illegal for a filename will be replaced with `'_'`.

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see [chapter 7^{\(57\)}](#)). See [section 2.2.5^{\(13\)}](#) about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Implicitly catch exceptions

When an exception is thrown during the execution of one of the Test's normal child nodes, the Test is usually terminated prematurely. This may not be what you want, since no information is gained from the execution of the rest of the child nodes.

If the Setup and Cleanup of the test are set up so you can guarantee the same initial conditions for each child node even in the case of an exception, you can set this attribute to make the Test catch the exception implicitly. That way, if an exception is caught from a normal child node, the exception is logged and the execution of that child node is stopped. Then the Test continues with the Cleanup as if nothing had happened.

Exceptions thrown during the execution of either the Setup or the Cleanup cannot be caught that way and will always terminate the Test

Variable: No

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run-log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run-log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run-log node for a sequence will propagate, can be limited.

Note

This value has no effect on the way exceptions are handled. It only affects the error states of the run-log nodes and by that the exit code of QF-Test when run in batch mode (see [section 1.6^{\(7\)}](#)). It also has no effect on the creation of compact run-logs (see command line argument `-compact(574)`). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

When using the command line argument `-test <n>|<ID>`⁽⁵⁸⁴⁾ for execution in batch mode you can specify the QF-Test ID of the node as an alternative to its qualified name.

Variable: Nein

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing `Alt-Return` or by clicking the  button.

Variable: Yes

Restrictions: None

30.2.5 Sequence

This is the most general kind of sequence. Its children are executed one by one and their number or type is not limited in any way.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: Any

Execution: The Variable definitions⁽³³⁵⁾ of the Sequence are bound and its child nodes executed one by one. After the execution of the last child is complete, the variables are unbound again.

Attributes:

The screenshot shows a 'Sequence' dialog box with the following sections:

- Name:** A text field containing 'Insert first name'.
- Variable definitions:** A table with two columns: 'Name' and 'Value'.

Name	Value
client	SUT
name	Greg
- Maximum error level:** A dropdown menu currently set to 'Exception'.
- QF-Test ID:** An empty text field.
- Delay before (ms):** An empty text field.
- Delay after (ms):** An empty text field.
- Comment:** A text area containing the text: 'Insert the name "Greg" into the "First name" text field and check whether it is stored correctly in the table'.

Figure 30.6: Sequence attributes

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and tell something about the function of the sequence.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see [chapter 7^{\(57\)}](#)). See [section 2.2.5^{\(13\)}](#) about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run-log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run-log node represents the worst of its child nodes' states. Using this attribute,

the maximum error state that the run-log node for a sequence will propagate, can be limited.

Note

This value has no effect on the way exceptions are handled. It only affects the error states of the run-log nodes and by that the exit code of QF-Test when run in batch mode (see. [section 1.6^{\(7\)}](#)). It also has no effect on the creation of compact run-logs (see command line argument `-compact(574)`). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the [Default delay^{\(288\)}](#) from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option [External editor command^{\(247\)}](#) lets you define an external editor in which comments can be edited conveniently by

pressing `[Alt-Return]` or by clicking the  button.

Variable: Yes

Restrictions: None

30.2.6 Test-step



A Test-step is a special Sequence that serves to divide a Test-case into steps that can be documented individually and will show up in the report or testdoc documentation. In contrast to Test-cases, which should not be nested, Test-steps can be nested to any depth.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: Any

Execution: The Variable definitions⁽³³⁸⁾ of the Test-step are bound and its child nodes executed one by one. After the execution of the last child is complete, the variables are unbound again.

Attributes:

Test-step	
Name	
First step	
Name for run-log and report	
First step: Enter \$(name)	
Variable definitions	
Name	Value
Maximum error level	
Exception	
QF-Test ID	
Delay before (ms)	Delay after (ms)
Comment	
Inserts the first name.	

Figure 30.7: Test-step attributes

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and tell something about the function of the sequence.

Variable: No

Restrictions: None

Name for run-log and report

A separate name to be used for run-log and report. This is useful to differentiate between multiple executions of the surrounding Test-case with potentially different values for its Characteristic variables.

Variable: Yes

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see [chapter 7^{\(57\)}](#)). See [section 2.2.5^{\(13\)}](#) about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run-log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run-log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run-log node for a sequence will propagate, can be limited.

Note

This value has no effect on the way exceptions are handled. It only affects the error states of the run-log nodes and by that the exit code of QF-Test when run in batch mode (see. [section 1.6^{\(7\)}](#)). It also has no effect on the creation of compact run-logs (see command line argument `-compact(574)`). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the [Default delay^{\(288\)}](#) from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.2.7 Sequence with time limit



This node extends the basic Sequence⁽³³⁴⁾ node with time-constraint checking. Child nodes are executed as usual, but upon completion of the sequence the elapsed time is compared to the time-limit. Exceeding the time limit will result in a warning, error or exception, depending on the value of the attribute Error level if time limit exceeded⁽³⁴¹⁾. Explicit delays like Delay before/after⁽³⁴²⁾ or user interrupts are deducted from the duration before the constraints are checked, unless Check realtime⁽³⁴⁰⁾ is activated.

For report generation, time-constraints are treated like checks. If the Comment⁽³⁴²⁾ attribute starts with an '!' character, the result will be logged in the report.

Note

The function of this node is to check time constraints in the user-acceptance range, i.e. between a few hundred milliseconds and a few seconds. Real-time constraints of a few milliseconds or less are beyond its limits.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: Any

Execution: The Variable definitions⁽³⁴¹⁾ of the Sequence with time limit are bound and its child nodes executed one by one. After the execution of the last child is complete, the variables are unbound again. The elapsed time is compared to the given time limit.

Attributes:

Sequence with time limit

Name
Check response

Time limit for execution (ms)
200

☐ Check realtime

Error level if time limit exceeded
Error

Variable definitions

Name	Value

Maximum error level
Exception

QF-Test ID

Delay before (ms) Delay after (ms)

Comment
Ensure that the following window appears within reasonable time.

Figure 30.8: Sequence with time limit attributes

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and tell something about the function of the sequence.

Variable: No

Restrictions: None

Time limit

The time (in milliseconds) allowed for the execution of the sequence.

Variable: Yes

Restrictions: Must not be negative.

Check realtime

Normally explicit delays like Delay before/after⁽³⁴²⁾ or user interrupts are deducted from the duration, before the time constraints are checked. To prevent this deduction and therefore check the real-time, this attribute can be activated.

Variable: No

Restrictions: None

Error level if time limit exceeded

This attribute determines what happens in case the time limit is exceeded. If set to "exception", a CheckFailedException⁽⁵⁶¹⁾ will be thrown. Otherwise a message with the respective error-level will be logged in the run-log.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see chapter 7⁽⁵⁷⁾). See section 2.2.5⁽¹³⁾ about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run-log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run-log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run-log node for a sequence will propagate, can be limited.

Note

This value has no effect on the way exceptions are handled. It only affects the error states of the run-log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.6⁽⁷⁾). It also has no effect on the creation of compact run-logs (see command line argument -compact⁽⁵⁷⁴⁾). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.2.8 Extras



This node is a kind of clipboard or playground where the usual restrictions on the parent of a node don't apply. You can add any kind of node here to assemble and try out some test sequences.

Contained in: Root node

Children: Any

Execution: Cannot be executed

Attributes:

Note

Figure 30.9: Extras attributes

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing **Alt-Return** or by clicking the  button.

Variable: Yes

Restrictions: None

30.3 Dependencies

Dependencies are an advanced feature for handling setup and cleanup requirements for test-cases automatically. The goal is to isolate test-cases so that each one can be run independently without interfering with others. This is a very important requirement for things like testing arbitrary sub-sets of test-cases, for example to re-test only failed tests, or during test-development where it must be possible to quickly run and debug any given test-case.

30.3.1 Dependency



Dependency nodes are used to implement advanced, automatic handling of setup and cleanup requirements for Test-sets⁽³²¹⁾ and Test-cases⁽³¹⁴⁾. A detailed description of the Dependency mechanism is given in chapter 12⁽⁹⁰⁾. This section focuses on formal requirements for the Dependency node and its attributes.

As Dependencies are complex, they should be reused as much as possible. This can be done by grouping Test-cases with identical dependencies in a Test-set and have them inherit the Dependency of the Test-set. However, this mechanism alone is not flexible enough, so a Dependency can also be implemented just like a Procedure⁽³⁷⁸⁾ and placed among the Procedures⁽³⁸⁷⁾ of a test-suite to be referenced from a Dependency reference node. For this to work, the Name⁽³¹⁶⁾ attribute is mandatory and it also has a list of Parameter default values⁽³⁴⁶⁾ that can be overridden in the referencing node.

The Characteristic variables⁽³⁴⁶⁾ of a Dependency are part of its identity and play an important role in the dependency resolution mechanism.

Contained in: Test-suite⁽³¹¹⁾, Test-set⁽³²¹⁾, Test-case⁽³¹⁴⁾, Procedures⁽³⁸⁷⁾, Package⁽³⁸⁵⁾.

Children: Zero or more Dependency references⁽³⁴⁷⁾ on which the Dependency is based, optional Setup⁽³⁵⁰⁾ and Cleanup⁽³⁵²⁾ nodes and an optional Error handler⁽³⁵⁵⁾ followed by zero or more Catch⁽⁴⁰⁸⁾ nodes.

Execution: Normally Dependencies are executed only indirectly in the setup phase of a Test-set or Test-case. If a Dependency node is executed interactively, the dependency stack is resolved as described in chapter 12⁽⁹⁰⁾.

Attributes:

Figure 30.10: Dependency attributes

Name

A Dependency is identified by its name, so you should assign a name with a meaning that is easy to recognize and remember.

Variable: No

Restrictions: Must not be empty or contain the characters '.' or '#'.

Name for run-log and report

A separate name to be used for run-log and report. This is useful to differentiate between multiple executions with potentially different values for the Characteristic variables.

Variable: Yes

Restrictions: None

Always execute, even in test-suite and test-set nodes

Normally a Dependency is only executed if it belongs to a Test-case node. Dependencies in Test-suite or Test-set nodes are simply inherited by the Test-case descendants of these nodes. However, in some cases it is useful to resolve a Dependency early, for example when the Dependency provides parameters for a test-run that are required to evaluate a Condition⁽³¹⁸⁾ of a subsequent Test-case. This can be achieved by activating this option.

Variable: No

Restrictions: None

Forced cleanup

Normally Dependencies are only rolled back and their cleanup code executed as required by the dependency resolution mechanism described in chapter 12⁽⁹⁰⁾. In some cases it makes sense to force partial cleanup of the dependency stack immediately after a Test-case finishes. This is what the Forced cleanup attribute is for. If this option is activated, the dependency stack will be rolled back at least up to and including this Dependency .

Variable: No

Restrictions: None

Characteristic variables

These variables are part of a Dependency's identity. During Dependency resolution as described in chapter 12⁽⁹⁰⁾ two Dependencies are considered equal only if they are one and the same node and the run-time values of all their Characteristic variables are identical. Additionally, the values of the Characteristic variables are stored during the setup phase of the Dependency. Later, when the Dependency is rolled back, these settings will be temporarily restored for the cleanup phase.

Variable: No

Restrictions: None

Parameter default values

Here you can define default or fallback values for the Dependency's parameters (see chapter 7⁽⁵⁷⁾). Defining these values also serves as documentation and is a valuable time-saver when using the dialog to select the Dependency for the Referenced dependency⁽³⁴⁸⁾ attribute of a Dependency reference⁽³⁴⁷⁾. See section 2.2.5⁽¹³⁾ about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.3.2 Dependency reference

A Dependency reference is simply a stand-in for a Dependency⁽³⁴⁴⁾ defined in some other place, typically a Package⁽³⁸⁵⁾. The name of the referenced Dependency is determined by its Name⁽³⁴⁵⁾ and the Names⁽³⁸⁶⁾ of its Package parents. These are concatenated with a dot ('.') as separator, starting with the outermost Package and ending in the Dependency's name. Thus to reference a Dependency named `demoStarted` in a Package named `Demo` that is itself a child of a Package named `Main`, set the Referenced dependency⁽³⁴⁸⁾ attribute to `'Main.Demo.demoStarted'`.

See section 17.1⁽¹⁵⁰⁾ about how to reference a Dependency in a different test-suite.

Contained in: Test-suite⁽³¹¹⁾, Test-set⁽³²¹⁾, Test-case⁽³¹⁴⁾ and Dependency⁽³⁴⁴⁾

Children: None

Execution: Normally Dependency references are executed only indirectly in the setup phase of a Test-set or Test-case. If a Dependency reference node is executed interactively, the referenced Dependency is determined and the dependency stack resolved accordingly as described in [chapter 12^{\(90\)}](#).

Attributes:

The screenshot shows a 'Dependency reference' dialog box with the following sections:

- Referenced dependency:** A text field with a 'Start' button.
- Dependency namespace:** A text field.
- Always execute, even in test-suite and test-set nodes:** A checkbox.
- Variable definitions:** A section with icons for adding, deleting, and copying variables, and a table with columns 'Name' and 'Value'.
- QF-Test ID:** A text field.
- Delay before (ms) / Delay after (ms):** Two text fields.
- Comment:** A text area.

Figure 30.11: Dependency reference attributes

Referenced dependency

The full name of the Dependency⁽³⁴⁴⁾, created from the Names⁽³⁸⁶⁾ of its Package⁽³⁸⁵⁾ parents and its own Name⁽³⁴⁵⁾, joined by a dot. The button above the attribute brings up a dialog in which you can select the Dependency interactively. By selecting the "Copy parameters" checkbox you can adopt the Dependency's Parameter default values⁽³⁴⁶⁾ as parameters for the Dependency reference node to save typing.

Variable: Yes

Restrictions: Must not be empty.

Dependency namespace

Normally there is only a single stack of dependencies, but in some cases, e.g.

when mixing tests for independent SUT clients, it can be useful to have independent sets of dependencies for different parts of the test, so that resolving the dependencies for one part doesn't necessarily tear down everything required for a different part.

By setting the Dependency namespace attribute of a Dependency reference node you can tell QF-Test to resolve the target dependency in that namespace. The default dependency stack will be completely ignored in that case. If there is a dependency stack remaining from a previous dependency resolution in the same namespace it will be used for comparison instead, otherwise a new stack will be created.

Variable: Yes

Restrictions: None

Always execute, even in test-suite and test-set nodes

Normally a Dependency is only executed if it belongs to a Test-case node. Dependencies in Test-suite or Test-set nodes are simply inherited by the Test-case descendants of these nodes. However, in some cases it is useful to resolve a Dependency early, for example when the Dependency provides parameters for a test-run that are required to evaluate a Condition⁽³¹⁸⁾ of a subsequent Test-case. This can be achieved by activating this option.

Variable: No

Restrictions: None

Variable definitions

These variables override the Parameter default values⁽³⁴⁶⁾ of the Dependency referenced by this Dependency reference. See section 2.2.5⁽¹³⁾ about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.3.3 Setup

This node is just like a Sequence⁽³³⁴⁾ except for its special place in a Dependency⁽³⁴⁴⁾, Test-set⁽³²¹⁾, Test-case⁽³¹⁴⁾ or Test⁽³³⁰⁾ node.

Contained in: Dependency⁽³⁴⁴⁾, Test-set⁽³²¹⁾, Test-case⁽³¹⁴⁾ or Test⁽³³⁰⁾

Children: Any

Execution: The Variable definitions⁽³⁵¹⁾ of the Setup are bound and its child nodes executed one by one. After the execution of the last child is complete, the variables are unbound again.

Attributes:

Setup

Name
Open table dialog

Variable definitions

Name	Value

Maximum error level
Exception

QF-Test ID

Delay before (ms) Delay after (ms)

Comment
Open the dialog with the table

Figure 30.12: Setup attributes

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and tell something about the function of the sequence.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see [chapter 7^{\(57\)}](#)). See [section 2.2.5^{\(13\)}](#) about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run-log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run-log node represents the worst of its child nodes' states. Using this attribute,

the maximum error state that the run-log node for a sequence will propagate, can be limited.

Note

This value has no effect on the way exceptions are handled. It only affects the error states of the run-log nodes and by that the exit code of QF-Test when run in batch mode (see. [section 1.6^{\(7\)}](#)). It also has no effect on the creation of compact run-logs (see command line argument `-compact(574)`). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the [Default delay^{\(288\)}](#) from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option [External editor command^{\(247\)}](#) lets you define an external editor in which comments can be edited conveniently by

pressing `[Alt-Return]` or by clicking the  button.

Variable: Yes

Restrictions: None

30.3.4 Cleanup



This node is just like a Sequence⁽³³⁴⁾ except for its special place in a Dependency⁽³⁴⁴⁾, Test-set⁽³²¹⁾, Test-case⁽³¹⁴⁾ or Test⁽³³⁰⁾ node.

Contained in: Dependency⁽³⁴⁴⁾, Test-set⁽³²¹⁾, Test-case⁽³¹⁴⁾ or Test⁽³³⁰⁾

Children: Any

Execution: The Variable definitions⁽³⁵³⁾ of the Cleanup are bound and its child nodes executed one by one. After the execution of the last child is complete, the variables are unbound again.

Attributes:

The screenshot shows the 'Cleanup' dialog box with the following fields and sections:

- Name:** A text field containing 'Close table dialog'.
- Variable definitions:** A section with icons for adding, deleting, and editing variables, and a table with columns 'Name' and 'Value'.
- Maximum error level:** A dropdown menu set to 'Exception'.
- QF-Test ID:** A text field.
- Delay before (ms):** A text field.
- Delay after (ms):** A text field.
- Comment:** A text area containing 'Reset the table to its initial state and close the dialog.'

Figure 30.13: Cleanup attributes

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and tell something about the function of the sequence.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the

execution of the sequence's child nodes (see [chapter 7^{\(57\)}](#)). See [section 2.2.5^{\(13\)}](#) about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run-log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run-log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run-log node for a sequence will propagate, can be limited.

Note

This value has no effect on the way exceptions are handled. It only affects the error states of the run-log nodes and by that the exit code of QF-Test when run in batch mode (see. [section 1.6^{\(7\)}](#)). It also has no effect on the creation of compact run-logs (see command line argument `-compact(574)`). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the [Default delay^{\(288\)}](#) from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option [External editor command^{\(247\)}](#) lets

you define an external editor in which comments can be edited conveniently by pressing **[Alt-Return]** or by clicking the  button.

Variable: Yes

Restrictions: None

30.3.5 Error handler



This node is just like a Sequence⁽³³⁴⁾ except for its special place in a Dependency⁽³⁴⁴⁾ (see chapter 12⁽⁹⁰⁾).

Contained in: Dependency⁽³⁴⁴⁾

Children: Any

Execution: The Variable definitions⁽³⁵⁶⁾ of the Error handler are bound and its child nodes executed one by one. After the execution of the last child is complete, the variables are unbound again.

Attributes:

The screenshot shows the 'Error handler' dialog box with the following fields and controls:

- Name:** A text field containing 'NotStarted'.
- Variable definitions:** A section with icons for adding, deleting, and editing variables, and a table with columns 'Name' and 'Value'.
- Maximum error level:** A dropdown menu currently set to 'Exception'.
- QF-Test ID:** A text field.
- Delay before (ms):** A text field.
- Delay after (ms):** A text field.
- Comment:** A text area with a pencil icon for editing.

Figure 30.14: Error handler attributes

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and tell something about the function of the sequence.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see [chapter 7^{\(57\)}](#)). See [section 2.2.5^{\(13\)}](#) about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run-log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run-log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run-log node for a sequence will propagate, can be limited.

Note

This value has no effect on the way exceptions are handled. It only affects the error states of the run-log nodes and by that the exit code of QF-Test when run in batch mode (see. [section 1.6^{\(7\)}](#)). It also has no effect on the creation of compact run-logs (see command line argument `-compact(574)`). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the [Default delay^{\(288\)}](#) from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.4 Data driver

The term *Data-driven Testing* refers to a common method in automated testing where test-cases are executed several times with different sets of data defined. With QF-Test's highly flexible variables there is no limit on how this data can be used but the most common case is for event input values and expected check values.

QF-Test's data-driving mechanism consists of the Data driver⁽³⁵⁷⁾ node used to provide a data-driving context and several kinds of Data binders. Currently available are the Data table⁽³⁶⁰⁾ node that stores data internally within QF-Test the CSV data file⁽³⁷²⁾ node that reads data from a CSV file, the Database⁽³⁶³⁾ node that reads data from a database and the Excel data file⁽³⁶⁸⁾ that reads data from an Excel file. An extension API for plugging in arbitrary external data is also available.

Further information about how the various parts of the data driver mechanism are working together is provided in chapter 14⁽¹²¹⁾.

30.4.1 Data driver



Except for its special place in a Test-set⁽³²¹⁾, a Data driver is just like a normal Sequence⁽³³⁴⁾. It provides a context for one or more Data binders to register themselves during the execution of the Data driver. The Test-set then iterates over the sets of data provided by the registered Data binders and executes its child nodes

as described in [chapter 14](#)⁽¹²¹⁾. For this purpose a Data driver node needs to be placed in a [Test-set](#)⁽³²¹⁾ node, between the optional [Dependency](#)⁽³⁴⁴⁾ and [Setup](#)⁽³⁵⁰⁾ nodes.

Contained in: [Test-set](#)⁽³²¹⁾, [Test](#)⁽³³⁰⁾.

Children: Any

Execution: When a Test-set or Test is executed it checks for a Data driver and runs it. The contents of the Data driver node are not limited to Data binders, but can hold any executable node so that they can perform any setup that may be required to retrieve the data. Thus it is also possible to share Data binders by placing them inside a [Procedure](#)⁽³⁷⁸⁾ and calling the Procedure from inside the Data driver. Any Data binders registered within this Data driver's context will then be queried for data by the Test-set.

Attributes:

The screenshot shows a 'Data driver' dialog box with the following fields and sections:

- Name:** A text field containing the word 'Data'.
- Name for loop pass in the run-log:** An empty text field.
- Name for separate run-log:** An empty text field.
- Variable definitions:** A section with a table.

Name	Value
- QF-Test ID:** An empty text field.
- Delay before (ms):** An empty text field.
- Delay after (ms):** An empty text field.
- Comment:** A text area with a small icon to its left.

Figure 30.15: Data driver attributes

Name

The name of a Data driver is a kind of short description. It is displayed in the tree view, so it should be concise and tell something about the kind of data provided.

Variable: No

Restrictions: None

Name for loop pass in the run-log

A separate name for each iteration to be used in the run-log. It can make use of the variables bound as a result of the data-driving which makes it easier to locate a specific step of the iteration.

Variable: Yes

Restrictions: None

Name for separate run-log

If this attribute is set it marks the node as a breaking point for split run-logs and defines the filename for the partial log. Every time an iteration of the Data driver finishes, the respective log entry is removed from the main run-log and saved as a separate, partial run-log. This operation is completely transparent, the main run-log retains references to the partial logs and is fully controllable. Please see [section 8.1.4^{\(72\)}](#) for further information about split run-logs.

This attribute has no effect if the option [Create split run-logs^{\(303\)}](#) is disabled or split run-logs are explicitly turned off for batch mode via the [-splitlog^{\(583\)}](#) command line argument.

There is no need to keep the filename unique. If necessary, QF-Test appends a number to the filename to avoid collisions. The filename may contain directories and, similar to specifying the name of a run-log in batch mode on the command line, the following placeholders can be used after a '%' or a '+' character:

Character	Replacement
%	Literal '%' character.
+	Literal '+' character.
i	The current runid as specified with -runid <ID>⁽⁵⁸²⁾ .
r	The error level of the partial log.
w	The number of warnings in the partial log.
e	The number of errors in the partial log.
x	The number of exceptions in the partial log.
t	The thread index to which the partial log belongs (for tests run with parallel threads).
y	The current year (2 digits).
Y	The current year (4 digits).
M	The current month (2 digits).
d	The current day (2 digits).
h	The current hour (2 digits).
m	The current minute (2 digits).
s	The current second (2 digits).

Table 30.5: Placeholders for the Name for separate run-log attribute

Variable: Yes

Restrictions: None, characters that are illegal for a filename will be replaced with ' '.

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see [chapter 7^{\(57\)}](#)). See [section 2.2.5^{\(13\)}](#) about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the [Default delay^{\(288\)}](#) from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option [External editor command^{\(247\)}](#) lets you define an external editor in which comments can be edited conveniently by

pressing **Alt-Return** or by clicking the  button.

Variable: Yes

Restrictions: None

30.4.2 Data table

Note



A Data table provides a convenient interface for storing test-data in tabular form directly inside QF-Test. For details about the data-driving mechanism please see [chapter 14](#)⁽¹²¹⁾.

Contained in: Any

Children: None

Execution: The Data table expands variable values in the table according to the option [When binding variables, expand values immediately](#)⁽³⁰⁹⁾. Each row is expanded individually left to right, meaning that - within the same row - a cell may refer to a variable bound in a column further to the left. Then the Data table registers itself with the Data driver context. If no such context is available a warning message will be issued. When all Data binders have been registered the [Test-set](#)⁽³²¹⁾ will query the Data table in order to iterate over the available sets of data.

Attributes:

Figure 30.16: Data table attributes

Name

The Name of a Data binder is mandatory. It is used to distinguish Data binders in the same Data driver context. A [Break](#)⁽³⁹⁴⁾ node executed during data-driven testing can be used to break out of a specific loop by referring to the Data binder's Name.

Variable: Yes

Restrictions: None

Iteration counter

The name of the variable that the iteration counter will be bound to.

Variable: Yes

Restrictions: None

Iteration ranges

An optional set of indexes or ranges to use from the bound data. This is especially useful during test development in order to run sample tests with just a single index or a subset of the given data.

Ranges are separated by ','. Each range is either a single index or an inclusive range of the form 'from-to' or 'from:to' where 'to' is optional. Indexes or ranges may be specified multiple times, overlap or be given in descending order. Indexes are 0-based, negative indexes are counted from the end, -1 being the last item. An invalid syntax or an index outside the valid data range will cause a `BadRangeException`⁽⁵⁶¹⁾.

The following table shows some example range specifications and the resulting indexes, based on a set of 20 entries.

Iteration ranges	Resulting indexes
0	[0]
-2, -1	[18, 19]
1-2,4:5	[1, 2, 4, 5]
18:,-3-	[18, 19, 17, 18, 19]
3-2,16:15	[3, 2, 16, 15]

Table 30.6: Iteration range examples

Note

The value bound for the Iteration counter reflects the index in the current interval, not the counter of actual iterations, e.g. if you specify '2' there will be a single iteration with the Iteration counter bound to '2', not '0'.

Variable: Yes

Restrictions: Valid syntax and index values

Data bindings

This is where the actual test data is defined. Each column of the table represents one variable with its name specified in the column header. Each row is a set of data, one value per variable. Thus the number of rows determines the number of iterations of the data-driven loop. To start entering the data you first need to add

columns to the table to define the variables, then add rows to fill in the values. See also [section 2.2.5^{\(13\)}](#) about how to work with the table.

Variable: Yes, even the column headers

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the [Default delay^{\(288\)}](#) from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option [External editor command^{\(247\)}](#) lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.4.3 Database



A Database node is used to load external data from a database. To access the database it is required to have the jar file with the database driver in QF-Test's plugin-directory ([section 37.2^{\(620\)}](#)) or in the CLASSPATH environment variable at QF-Test's start-up time. For more information about the connection mechanism to your database ask the developers or see www.connectionstrings.com.

For further details about the data-driving mechanism please see [chapter 14^{\(121\)}](#).

Note

Contained in: Any

Children: None

Execution: The Database loads the data from the database and expands variable values according to the option When binding variables, expand values immediately⁽³⁰⁹⁾. Each row is expanded individually left to right, meaning that - within the same row - a cell may refer to a variable bound in a column further to the left. Then the Database node registers itself with the Data driver context. If no such context is available a warning message will be issued. The result columns of the used SQL statement will be used for the variable names. The capitalization depends on the database driver, e.g. completely capitalized letters for Oracle. When the SQL statement has been executed all Data binders are registered for the Test-set⁽³²¹⁾. The Test-set⁽³²¹⁾ will query the Database node in order to iterate over the available sets of data.

Attributes:

Database	
Name	Iteration counter
Values	
Iteration ranges	
SQL-statement	
select * from users	
Driver class	
org.postgresql.Driver	
Connection string	
jdbc:postgresql://localhost/test	
User	Password
qfs	
QF-Test ID	
Delay before (ms)	Delay after (ms)
 Comment	

Figure 30.17: Database attributes

Name

The Name of a Data binder is mandatory. It is used to distinguish Data binders in the same Data driver context. A `Break`⁽³⁹⁴⁾ node executed during data-driven testing can be used to break out of a specific loop by referring to the Data binder's Name.

Variable: Yes

Restrictions: None

Iteration counter

The name of the variable that the iteration counter will be bound to.

Variable: Yes

Restrictions: None

Iteration ranges

An optional set of indexes or ranges to use from the bound data. This is especially useful during test development in order to run sample tests with just a single index or a subset of the given data.

Ranges are separated by ','. Each range is either a single index or an inclusive range of the form 'from-to' or 'from:to' where 'to' is optional. Indexes or ranges may be specified multiple times, overlap or be given in descending order. Indexes are 0-based, negative indexes are counted from the end, -1 being the last item. An invalid syntax or an index outside the valid data range will cause a `BadRangeException`⁽⁵⁶¹⁾.

The following table shows some example range specifications and the resulting indexes, based on a set of 20 entries.

Iteration ranges	Resulting indexes
0	[0]
-2, -1	[18, 19]
1-2,4:5	[1, 2, 4, 5]
18:,-3-	[18, 19, 17, 18, 19]
3-2,16:15	[3, 2, 16, 15]

Table 30.7: Iteration range examples

Note

The value bound for the Iteration counter reflects the index in the current interval, not the counter of actual iterations, e.g. if you specify '2' there will be a single iteration with the Iteration counter bound to '2', not '0'.

Variable: Yes

Restrictions: Valid syntax and index values

SQL statement

The SQL query that should be executed to get the desired test-data. This statement is supposed to be a *select* statement. Each column will stand for a variable. The capitalization of the columns depends on the kind of the used database driver, e.g. most of the Oracle drivers return completely capitalized variables.

Variable: Yes

Restrictions: Must not be empty

Driver class

The class name of the database driver.

Note

The jar file with the database driver has to be placed in the CLASSPATH or in QF-Test's plugin directory before launching QF-Test.

Here is a list of the most common database drivers:

Database	Classname of JDBC-driver
Borland Interbase	<code>interbase.interclient.Driver</code>
DB2	<code>com.ibm.db2.jcc.DB2Driver</code>
Informix	<code>com.informix.jdbc.IfxDriver</code>
IDS Server	<code>ids.sql.IDSDriver</code>
MS SQL Server 2000	<code>com.microsoft.jdbc.sqlserver.SQLServerDriver</code>
MS SQL Server 2005	<code>com.microsoft.sqlserver.jdbc.SQLServerDriver</code>
mSQL	<code>COM.imaginary.sql.msql.MsqlDriver</code>
MySQL	<code>com.mysql.jdbc.Driver</code>
Oracle	<code>oracle.jdbc.driver.OracleDriver</code>
Pointbase	<code>com.pointbase.jdbc.jdbcUniversalDriver</code>
PostgreSQL	<code>org.postgresql.Driver</code>
Standard Driver	<code>sun.jdbc.odbc.JdbcOdbcDriver</code>
Sybase	<code>com.sybase.jdbc2.jdbc.SybDriver</code>

Table 30.8: Database drivers

Variable: Yes

Restrictions: Must not be empty

Connection string

The connection string for database connection, typically something like:
`jdbc:databasetype://databasehost/databasename.`

Here is a list of the most common database connection strings:

Database	Example
Derby	<code>jdbc:derby:net://databaseserver:port/</code>
IBM DB2	<code>jdbc:db2://database</code>
HSQLB	<code>jdbc:hsqldb:file:database</code>
Interbase	<code>jdbc:interbase://databaseserver/database.gdb</code>
MS SQL Server 2000	<code>jdbc:microsoft:sqlserver://databaseserver: port;DatabaseName=database;</code>
MS SQL Server 2005	<code>jdbc:sqlserver://databaseserver: port;DatabaseName=database;</code>
MySQL	<code>jdbc:mysql://databaseserver/database</code>
PostgreSQL	<code>jdbc:postgresql://databaseserver/database</code>
ODBC Data Sources	<code>jdbc:odbc:database</code>
Oracle Thin	<code>jdbc:oracle:thin:@databaseserver:port: database</code>
Sybase	<code>jdbc:sybase:tds:databaseserver:port/database</code>

Table 30.9: Database connection strings

Variable: Yes**Restrictions:** Must not be empty**User**

The name of the user to use when connecting to the database. If your database connection doesn't require a user you can leave this field empty.

Variable: Yes**Restrictions:** None**Password**

The password to use when connecting to the database. If your database connection doesn't require a password you can leave this field empty.

Variable: Yes**Restrictions:** None**QF-Test ID**

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.4.4 Excel data file

An Excel data file is used to load external data from an Excel-file to provide test-data for data-driven testing. The first row must contain the names of the variables to bind. The rest of the rows should contain the values to be used for the iteration steps.

Note

It is possible that the contents of some cells are not read correctly. This can happen especially for cells of the type "date", "time" or "currency". The reason is that Excel stores the value and the format of a cell separately and the Java package used to parse Excel files doesn't support all possibilities. Still, the Excel data file should read most cells correctly, but in case of problems please change the type of the problematic cells in the Excel file to "text". Once read by QF-Test, all values are treated as strings anyway.

For further details about the data-driving mechanism please see chapter 14⁽¹²¹⁾.

Contained in: Any

Children: None

Execution: The Excel data file node loads the data from the Excel file and expands variable values in the table according to the option When binding variables, expand values immediately⁽³⁰⁹⁾. Each row is expanded individually left to right, meaning that - within the same row - a cell may refer to a variable bound in a column further to the left.

Then the Excel data file registers itself with the Data driver context. If no such context is available a warning message will be issued. When all Data binders have been registered the Test-set⁽³²¹⁾ will query the Excel data file node in order to iterate over the available sets of data.

Attributes:

Excel file	
Name	Iteration counter
Values	
Iteration ranges	
Excel file name	
testData.xls	
Worksheet name	
Override date format (e.g. MM/dd/yyyy)	
☒ Variables in rows	
QF-Test ID	
Delay before (ms)	Delay after (ms)
Comment	

Figure 30.18: Excel data file attributes

Name

The Name of a Data binder is mandatory. It is used to distinguish Data binders in the same Data driver context. A Break⁽³⁹⁴⁾ node executed during data-driven testing can be used to break out of a specific loop by referring to the Data binder's Name.

Variable: Yes

Restrictions: None

Iteration counter

The name of the variable that the iteration counter will be bound to.

Variable: Yes

Restrictions: None

Iteration ranges

An optional set of indexes or ranges to use from the bound data. This is especially useful during test development in order to run sample tests with just a single index or a subset of the given data.

Ranges are separated by ','. Each range is either a single index or an inclusive range of the form 'from-to' or 'from:to' where 'to' is optional. Indexes or ranges may be specified multiple times, overlap or be given in descending order. Indexes are 0-based, negative indexes are counted from the end, -1 being the last item. An invalid syntax or an index outside the valid data range will cause a `BadRangeException`⁽⁵⁶¹⁾.

The following table shows some example range specifications and the resulting indexes, based on a set of 20 entries.

Iteration ranges	Resulting indexes
0	[0]
-2, -1	[18, 19]
1-2,4:5	[1, 2, 4, 5]
18:,-3-	[18, 19, 17, 18, 19]
3-2,16:15	[3, 2, 16, 15]

Table 30.10: Iteration range examples

Note

The value bound for the Iteration counter reflects the index in the current interval, not the counter of actual iterations, e.g. if you specify '2' there will be a single iteration with the Iteration counter bound to '2', not '0'.

Variable: Yes

Restrictions: Valid syntax and index values

Excel file name

The name of the Excel file to read the test data. Relative path names are resolved relative to the directory of the test-suite.

The button above the attribute brings up a dialog in which you can select the Excel file interactively. You can also get to this dialog by pressing `Shift-Return` or `Alt-Return` when the focus is in the text field.

Variable: Yes

Restrictions: Must not be empty

Worksheet name

The name of the worksheet that contains the test-data. If empty, the first sheet will be used.

Variable: Yes

Restrictions: None

Override date format

This value specifies a date format which will be used for all date values specified in the Excel file. The format must be specified as for the Java class `SimpleDateFormat`, i.e. 'd' for day, 'M' for month, 'y' for year. Thus 'dd' stands for two digits of a day and similar for 'MM', 'yy' or 'yyyy'.

Variable: Yes

Restrictions: None

Variables in rows

If this attribute is checked, the names of the variables will be taken from the first row. If it's not checked the names will come from the first column.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets

you define an external editor in which comments can be edited conveniently by

pressing `[Alt-Return]` or by clicking the  button.

Variable: Yes

Restrictions: None

30.4.5 CSV data file



A CSV data file node is used to load external data from a file and make it available for data-driven testing. *CSV* stands for *Comma-separated Values*, a more or less standard plain text file format. Each line in the file contains one set of data with the values separated by a separator character, often but not always a comma (`,`). For use with a CSV data file node the first line of the CSV file must contain the names of the variables to bind. The rest of the lines should contain the values to be used for the iteration steps.

Unfortunately, with CSV files there are non-uniform definitions for things like quoting, white-space, multi-line values or embedded separator characters. Two de-facto standards exist, one used by Microsoft Excel and one by the rest of the world. QF-Test supports both of these. For further details about the data-driving mechanism please see [chapter 14^{\(121\)}](#).

Contained in: Any

Children: None

Execution: The CSV data file node loads the data from the CSV file and expands variable values in the table according to the option When binding variables, expand values immediately⁽³⁰⁹⁾. Each row is expanded individually left to right, meaning that - within the same row - a cell may refer to a variable bound in a column further to the left. Then the CSV data file registers itself with the Data driver context. If no such context is available a warning message will be issued. When all Data binders have been registered the Test-set⁽³²¹⁾ will query the CSV data file node in order to iterate over the available sets of data.

Attributes:

Figure 30.19: CSV data file attributes

Name

The Name of a Data binder is mandatory. It is used to distinguish Data binders in the same Data driver context. A Break⁽³⁹⁴⁾ node executed during data-driven testing can be used to break out of a specific loop by referring to the Data binder's Name.

Variable: Yes

Restrictions: None

Iteration counter

The name of the variable that the iteration counter will be bound to.

Variable: Yes

Restrictions: None

Iteration ranges

An optional set of indexes or ranges to use from the bound data. This is

especially useful during test development in order to run sample tests with just a single index or a subset of the given data.

Ranges are separated by ','. Each range is either a single index or an inclusive range of the form 'from-to' or 'from:to' where 'to' is optional. Indexes or ranges may be specified multiple times, overlap or be given in descending order. Indexes are 0-based, negative indexes are counted from the end, -1 being the last item. An invalid syntax or an index outside the valid data range will cause a `BadRangeException`⁽⁵⁶¹⁾.

The following table shows some example range specifications and the resulting indexes, based on a set of 20 entries.

Iteration ranges	Resulting indexes
0	[0]
-2, -1	[18, 19]
1-2,4:5	[1, 2, 4, 5]
18:,-3-	[18, 19, 17, 18, 19]
3-2,16:15	[3, 2, 16, 15]

Table 30.11: Iteration range examples

Note

The value bound for the Iteration counter reflects the index in the current interval, not the counter of actual iterations, e.g. if you specify '2' there will be a single iteration with the Iteration counter bound to '2', not '0'.

Variable: Yes

Restrictions: Valid syntax and index values

CSV file name

The name of the CSV file to get the test data from. Relative path names are resolved relative to the directory of the test-suite.

The button above the attribute brings up a dialog in which you can select the CSV file interactively. You can also get to this dialog by pressing `Shift-Return` or `Alt-Return` when the focus is in the text field.

Variable: Yes

Restrictions: Must not be empty

File encoding

An optional encoding for the CSV file, "UTF-8" for example. If no encoding is specified, the file will be read with the default encoding of the Java VM.

Variable: Yes

Restrictions: The encoding must be supported by the Java VM.

Read Microsoft Excel CSV format

If this option is active QF-Test will try to parse the CSV file using the format used by Microsoft Excel.

Variable: Yes

Restrictions: None

Separator character

In this attribute you can specify the character to be used as separator for data values within the CSV file. If no separator is defined a comma (',') is used as the default value. If Read Microsoft Excel CSV format⁽³⁷⁵⁾ is activated this attribute is ignored.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing **[Alt-Return]** or by clicking the  button.

Variable: Yes

Restrictions: None

Note

30.4.6 Data loop



A Data loop node is a simple loop with a single variable bound as the iteration counter. It is useful for executing Test-cases⁽³¹⁴⁾ multiple times by placing them inside a Test-set⁽³²¹⁾ with a Data driver⁽³⁵⁷⁾ holding a Data loop.

Contained in: Any

Children: None

Execution: During execution all the Data loop node does is register itself with the Data driver context. If no such context is available a warning message will be issued. When all Data binders have been registered the Test-set⁽³²¹⁾ will query the Data loop node in order to iterate through the loop and bind the iteration counter.

Attributes:

Data loop	
Name	Iteration counter
Loop	i
Iteration ranges	
Number of iterations	
10	
QF-Test ID	
Delay before (ms)	Delay after (ms)
 Comment	

Figure 30.20: Data loop attributes

Name

The Name of a Data binder is mandatory. It is used to distinguish Data binders in the same Data driver context. A Break⁽³⁹⁴⁾ node executed during data-driven testing can be used to break out of a specific loop by referring to the Data binder's Name.

Variable: Yes

Restrictions: None

Iteration counter

The name of the variable that the iteration counter will be bound to.

Variable: Yes

Restrictions: None

Iteration ranges

An optional set of indexes or ranges to use from the bound data. This is especially useful during test development in order to run sample tests with just a single index or a subset of the given data.

Ranges are separated by ','. Each range is either a single index or an inclusive range of the form 'from-to' or 'from:to' where 'to' is optional. Indexes or ranges may be specified multiple times, overlap or be given in descending order. Indexes are 0-based, negative indexes are counted from the end, -1 being the last item. An invalid syntax or an index outside the valid data range will cause a `BadRangeException`⁽⁵⁶¹⁾.

The following table shows some example range specifications and the resulting indexes, based on a set of 20 entries.

Iteration ranges	Resulting indexes
0	[0]
-2, -1	[18, 19]
1-2,4:5	[1, 2, 4, 5]
18:,-3-	[18, 19, 17, 18, 19]
3-2,16:15	[3, 2, 16, 15]

Table 30.12: Iteration range examples

Note

The value bound for the Iteration counter reflects the index in the current interval, not the counter of actual iterations, e.g. if you specify '2' there will be a single iteration with the Iteration counter bound to '2', not '0'.

Variable: Yes

Restrictions: Valid syntax and index values

Number of iterations

The number of iterations of the loop.

Variable: Yes

Restrictions: > 0

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.5 Procedures

Procedures are a means to collect some of the basic building blocks of a test-suite like events⁽⁴⁴⁶⁾ and checks⁽⁴⁶⁹⁾ into a larger, reusable structure.

Procedures can be called from any other part of the test-suite and even from different suites (see section 17.1⁽¹⁵⁰⁾). You can pass parameters to a procedure in the form of variable definitions (see chapter 7⁽⁵⁷⁾).

A typical example would be a procedure that selects a menu item in a menu. Its parameters could be the client name of the SUT, the name of the menu and the name of the menu item.

30.5.1 Procedure

Note



A Procedure is a Sequence⁽³³⁴⁾ that is executed from some other place by a Procedure call⁽³⁸¹⁾.

The parameters of the procedure are not defined explicitly. Instead they are a consequence of the variable references in the children of the Procedure. You may want to define fallback values for some or all of the parameters in the Variable definitions⁽³⁸⁰⁾. In any case it is a good idea to document the required parameters in the Comment⁽³⁸¹⁾ attribute.

A Procedure can return a value to the calling node with the help of a Return⁽³⁸⁴⁾ node. Without such a node a Procedure implicitly returns the empty string.

Contained in: Package⁽³⁸⁵⁾, Procedures⁽³⁸⁷⁾

Children: Any

Execution: The Procedure's variables are bound as fallback values. The child nodes are executed one by one, then the fallback values are unbound again.

Attributes:

Procedure	
Name	
expandNode	
Name	Value
client	SUT
Maximum error level	
Exception	
QF-Test ID	
Delay before (ms)	Delay after (ms)
Comment	
Expand a tree node.	
@param	node Name of the node.

Figure 30.21: Procedure Attributes

Name

A Procedure is identified by its name and the names of its Package⁽³⁸⁵⁾ ancestors, so you should assign a name with a meaning that is easy to recognize and remember.

Variable: No

Restrictions: Must not be empty or contain the characters '.' or '#'.

Variable definitions

Here you can define default or fallback values for the Procedure's parameters (see chapter 7⁽⁵⁷⁾). Defining these values also serves as documentation and is a valuable time-saver when using the dialog to select the Procedure for the Procedure name⁽³⁸²⁾ attribute of a Procedure call⁽³⁸¹⁾. See section 2.2.5⁽¹³⁾ about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run-log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run-log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run-log node for a sequence will propagate, can be limited.

Note

This value has no effect on the way exceptions are handled. It only affects the error states of the run-log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.6⁽⁷⁾). It also has no effect on the creation of compact run-logs (see command line argument -compact⁽⁵⁷⁴⁾). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.5.2 Procedure call



This node calls a Procedure⁽³⁷⁸⁾ in the same or a different test-suite, meaning that execution continues in the Procedure. When the Procedure is finished, the value returned by the Procedure is bound to the variable defined in the Variable for return value⁽³⁸²⁾ attribute and execution returns to the Procedure call and thus to its parent node.

The name of the Procedure⁽³⁷⁸⁾ to call is determined by its Name⁽³⁷⁹⁾ and the Names⁽³⁸⁶⁾ of its Package⁽³⁸⁵⁾ parents. These are concatenated with a dot ('.') as separator, starting with the outermost Package and ending in the Procedure's name. Thus to call a Procedure named `expandNode` in a Package named `tree` that is itself a child of a Package named `main`, set the Procedure name⁽³⁸²⁾ attribute to `main.tree.expandNode`.

See section 17.1⁽¹⁵⁰⁾ about how to call a Procedure in a different test-suite.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The Variable definitions⁽³⁸³⁾ of the Procedure call are bound, the target Procedure⁽³⁷⁸⁾ is determined and execution passed to it. After the Procedure returns, the Procedure call's variables are unbound again.

Attributes:

Procedure call

Procedure name
tree.expandNode

Variable for return value

☐ Local variable

Variable definitions

Name	Value
node	test

QF-Test ID

Delay before (ms) Delay after (ms)

Comment
Expand tree node "test"

Figure 30.22: Procedure call Attributes

Name

The full name of the Procedure⁽³⁷⁸⁾, created from the Names⁽³⁸⁶⁾ of its Package⁽³⁸⁵⁾

parents and its own Name⁽³⁷⁹⁾, joined by a dot. The button brings up a dialog in which you can select the Procedure interactively. By selecting the "Copy parameters" checkbox you can adopt the Procedure's default values as parameters for the Procedure call node to save typing.

Variable: Yes

Restrictions: Must not be empty.

Variable for return value

The value returned by the Procedure, either through a Return⁽³⁸⁴⁾ node or the empty string, is bound to the variable defined in this optional attribute. Additionally, the most recent return value is always available as the special variable `${qftest:return}`.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure⁽³⁷⁸⁾, Dependency⁽³⁴⁴⁾ or Test-case⁽³¹⁴⁾ node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test-case node or, if there is no such node, in the global bindings. See chapter 7⁽⁵⁷⁾ for a detailed explanation of variable binding and lookup.

Variable: No

Restrictions: None

Variable definitions

This is where you define the parameter values for the Procedure⁽³⁷⁸⁾ (see chapter 7⁽⁵⁷⁾). See section 2.2.5⁽¹³⁾ about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.5.3 Return



This node can be used to return from a Procedure⁽³⁷⁸⁾ prematurely and also to pass a return value to the calling node.

From a script, the same effect can be achieved by raising a `ReturnException`.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: A `ReturnException` is thrown. If it is caught by a matching Procedure, the Procedure is terminated and the return value passed to the caller. If the node is executed outside a Procedure it will lead to an error.

Attributes:

Return	
Return value	
result	
QF-Test ID	
Delay before (ms)	Delay after (ms)
 Comment	
Return from the procedure	

Figure 30.23: Return Attributes

Return value

The value to return from the Procedure. May be empty in which case the empty string is returned.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.5.4 Package

The only use of Packages is to structure the Procedures⁽³⁷⁸⁾ of a test-suite. The Name⁽³⁸⁶⁾ of a Package is part of the Procedure's fully qualified name, as required by a Procedure call⁽³⁸¹⁾.

Contained in: Package⁽³⁸⁵⁾, Procedures⁽³⁸⁷⁾

Children: Package⁽³⁸⁵⁾, Procedure⁽³⁷⁸⁾

Execution: Cannot be executed.

Attributes:

Note

Figure 30.24: Package Attributes

Name

The name of a Package is part of the identification of the Procedures⁽³⁷⁸⁾ it contains, so you should assign a name with a meaning that is easy to recognize and remember.

Variable: No

Restrictions: Must not be empty or contain the characters '.' or '#'.

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Border for relative calls

This flag determines whether relative procedure calls, test calls or dependency references are allowed within that certain node. Relative calls passing that border are not allowed. If that attribute is not specified in the hierarchy, no relative calls are allowed.

Variable: No

Restrictions: None

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes,

this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing **[Alt-Return]** or by clicking the  button.

Variable: Yes

Restrictions: None

30.5.5 Procedures



This is the root of all Packages⁽³⁸⁵⁾ and Procedures⁽³⁷⁸⁾.

Contained in: Root node

Children: Package⁽³⁸⁵⁾, Procedure⁽³⁷⁸⁾

Execution: Cannot be executed.

Attributes:

Figure 30.25: Procedures Attributes

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Border for relative calls

This flag determines whether relative procedure calls, test calls or dependency references are allowed within that certain node. Relative calls passing that border

are not allowed. If that attribute is not specified in the hierarchy, no relative calls are allowed.

Variable: No

Restrictions: None

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.6 Control structures

Besides the standard sequence⁽³¹⁴⁾ types QF-Test has a set of special control structures. Conditional processing is handled via If⁽³⁹⁶⁾, Elseif⁽³⁹⁹⁾ and Else⁽⁴⁰³⁾ nodes. Loops⁽³⁸⁸⁾ and While⁽³⁹¹⁾ nodes can be aborted with a Break⁽³⁹⁴⁾ node. Exceptions are handled by Try⁽⁴⁰⁵⁾, Catch⁽⁴⁰⁸⁾ and Finally⁽⁴¹¹⁾ nodes.

Beyond that, full scripting is available for the Jython language (formerly called *JPython*) and Groovy as documented in chapter 13⁽⁹⁹⁾.

30.6.1 Loop



This node is basically the same as a Sequence⁽³³⁴⁾ except that its children can be executed more than once. This is useful in two ways. For one thing, a test sequence that executes OK a hundred times is more trustworthy than a sequence that only runs once. The other use is to run a number of similar jobs with slight variations. To that end, the count of the current iteration is bound as a variable during execution.

Special Loops with varying increments can be achieved by changing the value of the Iteration counter⁽³⁹⁰⁾ during execution.

Execution of a Loop can be terminated prematurely with the help of a Break⁽³⁹⁴⁾ node. An optional Else⁽⁴⁰³⁾ node may be placed at the end of the Loop. It is executed if all iterations of the Loop are run through completely without hitting a Break.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: Any

Execution: The Variable definitions⁽³⁹⁰⁾ of the Loop are bound. The Iteration counter⁽³⁹⁰⁾ is initialized to 0 and the child nodes are executed one by one. For each iteration the Iteration counter is increased by one and the children are executed again. After the final execution of the last child is complete, the Iteration counter and the Variable definitions are unbound again.

Attributes:

Loop					
Name Open and close dialog					
Number of iterations \$(count)	Iteration counter i				
Variable definitions <table border="1"> <thead> <tr> <th>Name</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td> </td> <td> </td> </tr> </tbody> </table>		Name	Value		
Name	Value				
Maximum error level Exception					
QF-Test ID 					
Delay before (ms) 	Delay after (ms) 				
Comment Open and close the dialog \$(count) times					

Figure 30.26: Loop attributes

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and tell something about the function of the sequence.

Variable: No

Restrictions: None

Number of iterations

The number of iterations of the loop.

Variable: Yes

Restrictions: > 0

Iteration counter

The name of the variable that will hold the iteration count during the execution. Make sure to use different Iteration counter names for nested loops.

Variable: Yes

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see [chapter 7^{\(57\)}](#)). See [section 2.2.5^{\(13\)}](#) about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run-log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run-log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run-log node for a sequence will propagate, can be limited.

Note

This value has no effect on the way exceptions are handled. It only affects the error states of the run-log nodes and by that the exit code of QF-Test when run in batch mode (see. [section 1.6^{\(7\)}](#)). It also has no effect on the creation of compact run-logs (see command line argument `-compact(574)`). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

The QF-Test ID of the Loop node can be used in a [Break^{\(394\)}](#) node to terminate an outer loop explicitly when loops are nested.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.6.2 While



This is a sequence that is executed repeatedly as long as a condition is fulfilled.

The loop can be terminated prematurely with the help of a Break⁽³⁹⁴⁾ node.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: Any

Execution: The Variable definitions⁽³⁹³⁾ of the While node are bound. If the condition evaluates to true, the child nodes are executed one by one. This is repeated until the condition evaluates to false or the loop is terminated through a Break⁽³⁹⁴⁾ node or an Exception. Finally the Variable definitions are unbound again.

Attributes:

While

Condition
\$(width) > 100

Name

Variable definitions

Name	Value

Maximum error level
Exception

QF-Test ID

Delay before (ms) Delay after (ms)

Comment
Shrink window until width reaches 100

Figure 30.27: While attributes

Condition

A condition is an expression that evaluates to either true or false. QF-Test discriminates between simple expression that it evaluates itself and complex expressions that are passed to the Jython script language to evaluate.

An empty string or the string `false` (regardless of case) is interpreted as false, the string `true` as true. Whole numbers are true if and only if they are non-zero.

Evaluating expressions in Jython opens the way for powerful expression handling. Jython supports the standard operators `==`, `!=`, `>`, `>=`, `<` and `<=`. You can combine expressions with `and` and `or` and define their priority with braces.

Note

Accessing QF-Test variables in a condition follows the same rules as in Jython scripts (see [section 13.3.3^{\(103\)}](#)). You can use the standard QF-Test syntax `$ ( . . . )` and `${ . . . : . . . }` for numeric or boolean values. String values should be accessed with `rc.lookup ( . . . )`.

Some examples:

Expression	Value
Empty String	False
0	False
21	True
False	False
True	True
bla bla	Syntax error
25 > 0	True
\${qftest:batch}	True if QF-Test is run in batch mode
not \${qftest:batch}	True if QF-Test is run in interactive mode
rc.lookup("system", "java.version") == "1.3.1"	True if JDK Version is 1.3.1
rc.lookup("system", "java.version")[0] == "1"	True is JDK Version starts with 1
(1 > 0 and 0 == 0) or 2 < 1	True

Table 30.13: Condition examples

Variable: Yes

Restrictions: Valid syntax

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and tell something about the function of the sequence.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see [chapter 7^{\(57\)}](#)). See [section 2.2.5^{\(13\)}](#) about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run-log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run-log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run-log node for a sequence will propagate, can be limited.

Note

This value has no effect on the way exceptions are handled. It only affects the error states of the run-log nodes and by that the exit code of QF-Test when run in batch mode (see. [section 1.6^{\(7\)}](#)). It also has no effect on the creation of compact run-logs (see command line argument `-compact(574)`). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

The QF-Test ID of the While node can be used in a [Break^{\(394\)}](#) node to terminate an outer loop explicitly when loops are nested.

Variable: No

Restrictions: Must not be empty, contain any of the characters '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the [Default delay^{\(288\)}](#) from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option [External editor command^{\(247\)}](#) lets you define an external editor in which comments can be edited conveniently by

pressing `[Alt-Return]` or by clicking the  button.

Variable: Yes

Restrictions: None

30.6.3 Break



This node is used to terminate a Loops⁽³⁸⁸⁾ or a While⁽³⁹¹⁾ node prematurely.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: A `BreakException` is thrown. If it is caught by a matching loop, the loop is terminated, otherwise it will lead to an error.

Attributes:

The screenshot shows a dialog box titled "Break". It contains several input fields: "QF-Test loop ID", "QF-Test ID", "Delay before (ms)", "Delay after (ms)", and a "Comment" field with a text area containing "Break innermost loop".

Figure 30.28: Break attributes

QF-Test loop ID

For nested loops you can specify the loop to terminate by specifying the QF-Test ID⁽³⁹⁰⁾ of a Loop⁽³⁸⁸⁾ node and refer to it here. It works with QF-Test ID⁽³⁹⁴⁾ for a While⁽³⁹¹⁾ node respectively. In case this field is empty the innermost loop is terminated.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.6.4 If



Like in Java the child nodes of this node are executed only if a condition evaluates to true. However QF-Test differs from common programming languages in the way alternative branches are arranged.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: Any executable node, followed by an arbitrary number of Elseif⁽³⁹⁹⁾ nodes with an optional Else⁽⁴⁰³⁾ at the end.

Execution: The Variable definitions⁽³⁹⁸⁾ of the If node are bound. If the condition evaluates to true, the normal child nodes are executed one by one. Otherwise the conditions of the Elseif⁽³⁹⁹⁾ nodes are evaluated and the first Elseif node whose condition evaluates to true is executed. If none of the conditions are true or no Elseif nodes exist, the Else⁽⁴⁰³⁾ node is executed, if one exists. Finally the Variable definitions are unbound again.

Attributes:

If

Condition

Name

Variable definitions

Name	Value

Maximum error level
 Exception

QF-Test ID

Delay before (ms) Delay after (ms)

Comment

Figure 30.29: If attributes

Condition

A condition is an expression that evaluates to either true or false. QF-Test discriminates between simple expression that it evaluates itself and complex expressions that are passed to the Jython script language to evaluate.

An empty string or the string `false` (regardless of case) is interpreted as false, the string `true` as true. Whole numbers are true if and only if they are non-zero.

Evaluating expressions in Jython opens the way for powerful expression handling. Jython supports the standard operators `==`, `!=`, `>`, `>=`, `<` and `<=`. You can combine expressions with `and` and `or` and define their priority with braces.

Note

Accessing QF-Test variables in a condition follows the same rules as in Jython scripts (see [section 13.3.3^{\(103\)}](#)). You can use the standard QF-Test syntax `$ ( . . . )` and `${ . . . : . . . }` for numeric or boolean values. String values should be accessed with `rc.lookup ( . . . )`.

Some examples:

Expression	Value
Empty String	False
0	False
21	True
False	False
True	True
bla bla	Syntax error
25 > 0	True
\${qftest:batch}	True if QF-Test is run in batch mode
not \${qftest:batch}	True if QF-Test is run in interactive mode
rc.lookup("system", "java.version") == "1.3.1"	True if JDK Version is 1.3.1
rc.lookup("system", "java.version")[0] == "1"	True is JDK Version starts with 1
(1 > 0 and 0 == 0) or 2 < 1	True

Table 30.14: Condition examples

Variable: Yes

Restrictions: Valid syntax

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and tell something about the function of the sequence.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see [chapter 7^{\(57\)}](#)). See [section 2.2.5^{\(13\)}](#) about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run-log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run-log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run-log node for a sequence will propagate, can be limited.

Note

This value has no effect on the way exceptions are handled. It only affects the error states of the run-log nodes and by that the exit code of QF-Test when run in batch mode (see. [section 1.6^{\(7\)}](#)). It also has no effect on the creation of compact run-logs (see command line argument `-compact(574)`). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the [Default delay^{\(288\)}](#) from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option [External editor command^{\(247\)}](#) lets you define an external editor in which comments can be edited conveniently by

pressing `[Alt-Return]` or by clicking the  button.

Variable: Yes

Restrictions: None

30.6.5 Elseif



This node is an alternative branch in an If⁽³⁹⁶⁾ node. If the condition of the If node evaluates to false, the first Elseif node whose condition is true is executed.

Contained in: If⁽³⁹⁶⁾

Children: Any

Execution: The Variable definitions⁽⁴⁰¹⁾ of the Elseif are bound and its child nodes executed one by one. After the execution of the last child is complete, the variables are unbound again.

Attributes:

Elseif	
Condition	
'\${system:os.name}'.find('Linux') >= 0	
Name	
On Linux	
Variable definitions	
Name	Value
Maximum error level	
Exception	
QF-Test ID	
Delay before (ms)	Delay after (ms)
Comment	
Is the test run on a Linux system?	

Figure 30.30: Elseif attributes

Condition

A condition is an expression that evaluates to either true or false. QF-Test discriminates between simple expression that it evaluates itself and complex expressions that are passed to the Jython script language to evaluate.

An empty string or the string `false` (regardless of case) is interpreted as false, the string `true` as true. Whole numbers are true if and only if they are non-zero.

Evaluating expressions in Jython opens the way for powerful expression handling. Jython supports the standard operators `==`, `!=`, `>`, `>=`, `<` and `<=`. You can combine expressions with `and` and `or` and define their priority with braces.

Note

Accessing QF-Test variables in a condition follows the same rules as in Jython scripts (see [section 13.3.3^{\(103\)}](#)). You can use the standard QF-Test syntax `$(...)` and `#{...:...}` for numeric or boolean values. String values should be accessed with `rc.lookup(...)`.

Some examples:

Expression	Value
Empty String	False
0	False
21	True
False	False
True	True
bla bla	Syntax error
<code>25 > 0</code>	True
<code>#{qftest:batch}</code>	True if QF-Test is run in batch mode
<code>not #{qftest:batch}</code>	True if QF-Test is run in interactive mode
<code>rc.lookup("system", "java.version") == "1.3.1"</code>	True if JDK Version is 1.3.1
<code>rc.lookup("system", "java.version")[0] == "1"</code>	True is JDK Version starts with 1
<code>(1 > 0 and 0 == 0) or 2 < 1</code>	True

Table 30.15: Condition examples

Variable: Yes

Restrictions: Valid syntax

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and tell something about the function of the sequence.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see [chapter 7^{\(57\)}](#)). See [section 2.2.5^{\(13\)}](#) about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run-log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run-log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run-log node for a sequence will propagate, can be limited.

Note

This value has no effect on the way exceptions are handled. It only affects the error states of the run-log nodes and by that the exit code of QF-Test when run in batch mode (see. section 1.6⁽⁷⁾). It also has no effect on the creation of compact run-logs (see command line argument `-compact(574)`). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing `[Alt-Return]` or by clicking the  button.

Variable: Yes

Restrictions: None

30.6.6 Else



An Else node is executed if neither the condition of its If⁽³⁹⁶⁾ parent, nor the condition of its Elseif⁽³⁹⁹⁾ siblings evaluate to true.

Contained in: If⁽³⁹⁶⁾, Loop⁽³⁸⁸⁾, Try⁽⁴⁰⁵⁾

Children: Any

Execution: The Variable definitions⁽⁴⁰⁴⁾ of the Else are bound and its child nodes executed one by one. After the execution of the last child is complete, the variables are unbound again.

Attributes:

Else	
Name	
Other systems	
Variable definitions	
Name	Value
Maximum error level	
Exception	
QF-Test ID	
Delay before (ms)	Delay after (ms)
Comment	
Neither Windows nor Linux.	

Figure 30.31: Else attributes

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and tell something about the function of the sequence.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see [chapter 7^{\(57\)}](#)). See [section 2.2.5^{\(13\)}](#) about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run-log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run-log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run-log node for a sequence will propagate, can be limited.

Note

This value has no effect on the way exceptions are handled. It only affects the error states of the run-log nodes and by that the exit code of QF-Test when run in batch mode (see. [section 1.6^{\(7\)}](#)). It also has no effect on the creation of compact run-logs (see command line argument `-compact(574)`). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the [Default delay^{\(288\)}](#) from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.6.7 Try

A Try is a special sequence whose semantic equates the Java `try-catch-finally` composition. As in Python, this composition was extended to support optional `else` blocks. A Try behaves the same way as a special Sequence⁽³³⁴⁾ with the extension, that exception handling is possible. Like a Sequence it has a set of normal child nodes that it executes one by one. After these may come an arbitrary number of Catch⁽⁴⁰⁸⁾ nodes with an optional Else⁽⁴⁰³⁾ node followed by an optional Finally⁽⁴¹¹⁾ node at the end.

If an exception is thrown during the execution of one of the normal child nodes, the Catch nodes are tested for whether they are able to catch that exception. The first one found is executed and the Try will be exited normally afterwards without continuing with the execution of the normal child nodes and without passing the exception on. If no matching Catch is found, the exception will terminate the Try immediately (almost, see below) and be passed onto the Try's parent.

A possible Else⁽⁴⁰³⁾ child node at the end of a try node will be executed, if and only if no Catch⁽⁴⁰⁸⁾ nodes had been executed. This means, it is executed, when no exception in the try block was thrown.

If the Try has a Finally⁽⁴¹¹⁾ child node, this node will be executed just before the Try finishes, no matter whether an exception is thrown and whether it is handled or not.

Note

Exceptions can also be caught implicitly by a Test⁽³³⁰⁾, which is less flexible but often more convenient.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: Any executable node, followed by an arbitrary number of Catch⁽⁴⁰⁸⁾ nodes with an optional Else⁽⁴⁰³⁾ node and/or an optional Finally⁽⁴¹¹⁾ node at the end.

Execution: The Variable definitions⁽⁴⁰⁶⁾ of the Try are bound and its normal child nodes executed one by one. If an exception is thrown, execution of the normal children is terminated. If a Catch node with a matching Exception class⁽⁴⁰⁹⁾ is found it is executed. Before exiting the Try its Finally node is executed unconditionally. After unbinding the Variable definitions, the Try is either exited cleanly if no exception was thrown or the exception was caught, or it passes on the uncaught exception.

Attributes:

The screenshot shows a dialog box titled "Try". It contains the following fields and sections:

- Name:** A text field containing "Exit SUT".
- Variable definitions:** A section with icons for adding, deleting, and editing variables, followed by a table with two columns: "Name" and "Value". The table is currently empty.
- Maximum error level:** A dropdown menu currently set to "Exception".
- QF-Test ID:** An empty text field.
- Delay before (ms):** An empty text field.
- Delay after (ms):** An empty text field.
- Comment:** A text area containing the text: "Try to exit the SUT cleanly. If that doesn't work, kill its process."

Figure 30.32: Try attributes

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and tell something about the function of the sequence.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see chapter 7⁽⁵⁷⁾). See section 2.2.5⁽¹³⁾ about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run-log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run-log node represents the worst of its child nodes' states. Using this attribute, the maximum error state that the run-log node for a sequence will propagate, can be limited.

Note

This value has no effect on the way exceptions are handled. It only affects the error states of the run-log nodes and by that the exit code of QF-Test when run in batch mode (see. [section 1.6^{\(7\)}](#)). It also has no effect on the creation of compact run-logs (see command line argument `-compact(574)`). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

For a Try node, the error state of the run-log is additionally affected by the Maximum error level⁽⁴¹⁰⁾ of the Catch⁽⁴⁰⁸⁾ node that handles an exception.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets

you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.6.8 Catch



A Catch is a special Sequence⁽³³⁴⁾ that can only be placed inside a Try⁽⁴⁰⁵⁾ node or a Dependency⁽³⁴⁴⁾. Its job is to catch exceptions that may arise during the execution of a Try's or a Test-case⁽³¹⁴⁾ with a Dependency.

A Catch can handle an exception if the class of the exception is the same as the Catch node's Exception class⁽⁴⁰⁹⁾ attribute or a derived class thereof, just as in Java.

Contained in: Try⁽⁴⁰⁵⁾, Dependency⁽³⁴⁴⁾

Children: Any

Execution: The Variable definitions⁽⁴¹⁰⁾ of the Catch are bound and its child nodes executed one by one. After the execution of the last child is complete, the variables are unbound again.

Attributes:

Catch

Exception class
ClientNotTerminatedException

Expected message
[Empty text box]

☐ As regexp

Name
SUT didn't terminate

Variable definitions

Name	Value

Maximum error level
No Error

QF-Test ID
[Empty text box]

Delay before (ms) Delay after (ms)
[Empty text box] [Empty text box]

Comment
SUT didn't terminate

Figure 30.33: Catch attributes

Exception class

This *ComboBox* lets you select the class of the exception that is to be caught. All QF-Test exceptions are derived from the class `TestException`⁽⁵⁵⁸⁾. For details about the possible exceptions see chapter 31⁽⁵⁵⁸⁾.

Variable: No

Restrictions: None

Expected message

You can further qualify the exception to catch by specifying a message to look for. If this attribute is empty, all exceptions of the specified class are caught. Otherwise it is compared to `exception.getMessage()` and the exception is caught only in case of a match.

Variable: Yes

Restrictions: Valid regexp if required.

As regexp

If this attribute is set, the exception message is matched against a regexp (see [section 36.4^{\(614\)}](#)) instead of comparing plain strings.

Variable: Yes

Restrictions: None

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and tell something about the function of the sequence.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see [chapter 7^{\(57\)}](#)). See [section 2.2.5^{\(13\)}](#) about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

In contrast to the [Maximum error level^{\(335\)}](#) of other [sequences^{\(314\)}](#), this attribute does not determine the error state propagated by the run-log node for the Catch node itself, but for the log of its parent [Try^{\(405\)}](#) node, provided that the Catch is executed in order to handle an exception.

The error state for any warnings, errors or exceptions that happen during the execution of the Catch node are **not** limited by the setting of this attribute. Otherwise problems occurring during exception handling might accidentally go unnoticed.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.6.9 Finally

A Finally node, which can only be placed at the end of a Try node, will always be executed as the last thing just before exiting the Try, no matter what happened there. This is used primarily to ensure that cleanup code like removing a temporary file or terminating a process is executed under any conditions.

Contained in: Try⁽⁴⁰⁵⁾

Children: Any

Execution: The Variable definitions⁽⁴¹²⁾ of the Finally are bound and its child nodes executed one by one. After the execution of the last child is complete, the variables are unbound again.

Attributes:

Finally

Name
Make sure the SUT is terminated

Variable definitions

Name	Value

Maximum error level
Exception

QF-Test ID

Delay before (ms) Delay after (ms)

Comment
Kill the SUT's process if it is still alive.

Figure 30.34: Finally attributes

Name

The name of a sequence is a kind of short description. It is displayed in the tree view, so it should be concise and tell something about the function of the sequence.

Variable: No

Restrictions: None

Variable definitions

This is where you define the values of the variables that remain bound during the execution of the sequence's child nodes (see [chapter 7^{\(57\)}](#)). See [section 2.2.5^{\(13\)}](#) about how to work with the table.

Variable: Variable names no, values yes

Restrictions: None

Maximum error level

When a warning, error or exception occurs during a test run, the state of the corresponding node of the run-log is set accordingly. This state is normally propagated to the parent node in a way that ensures that the error state of a run-log node represents the worst of its child nodes' states. Using this attribute,

the maximum error state that the run-log node for a sequence will propagate, can be limited.

Note

This value has no effect on the way exceptions are handled. It only affects the error states of the run-log nodes and by that the exit code of QF-Test when run in batch mode (see. [section 1.6^{\(7\)}](#)). It also has no effect on the creation of compact run-logs (see command line argument `-compact(574)`). The node for a sequence in which a warning, error or exception occurs is never removed from a compact log, even if the error is not propagated due to the setting of this attribute.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the [Default delay^{\(288\)}](#) from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option [External editor command^{\(247\)}](#) lets you define an external editor in which comments can be edited conveniently by

pressing `[Alt-Return]` or by clicking the  button.

Variable: Yes

Restrictions: None

30.6.10 Throw



If you need to handle an exceptional situation, you can use this node to throw an explicit Exception.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: A UserException⁽⁵⁶⁴⁾ is thrown, its message taken from the Exception message⁽⁴¹⁴⁾ attribute.

Attributes:

Figure 30.35: Throw attributes

Exception message

An arbitrary message for the UserException⁽⁵⁶⁴⁾ to throw.

Variable: Yes

Restrictions: Must not be empty.

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.6.11 Rethrow

An exception that was caught by a Catch⁽⁴⁰⁸⁾ node can be thrown again with the help of a Rethrow node. This is especially useful if you need to catch all kinds of exceptions except one. To handle that case, create a Try⁽⁴⁰⁵⁾ node with a Catch for the special exception followed by a Catch for a TestException⁽⁵⁵⁸⁾. Then place a Rethrow in the first Catch node.

Contained in: All kinds of sequences⁽³¹⁴⁾. The Rethrow node doesn't have to be placed directly below a Catch node.

Children: None

Execution: The last exception caught by a Catch⁽⁴⁰⁸⁾ node is thrown again. If no such exception exists, a CannotRethrowException⁽⁵⁶⁴⁾ is thrown.

Attributes:

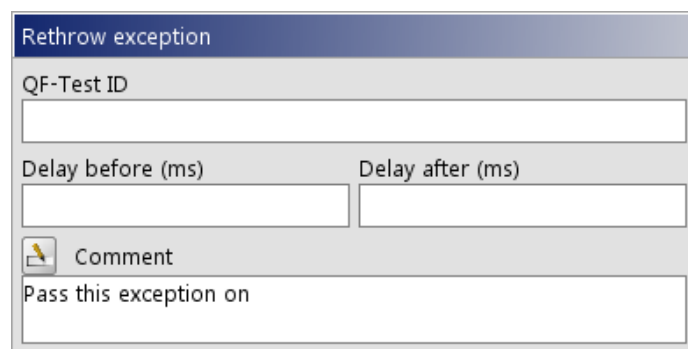


Figure 30.36: Rethrow attributes

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing **[Alt-Return]** or by clicking the  button.

Variable: Yes

Restrictions: None

30.6.12 Server script

Server side scripts are executed by an interpreter (Jython or Groovy) embedded into QF-Test. Scripting is explained in chapter 13⁽⁹⁹⁾ and chapter 37⁽⁶¹⁹⁾.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The script is executed by an embedded interpreter.

Attributes:

Note

Server script

Script

```

1 # read values from file
2 values = readValues("someFile")
3 # create new rows
4 for val in values:
5     rc.callProcedure("table.createRow", val
6

```

Script language

Jython

Name

fill table

QF-Test ID

Delay before (ms)

Delay after (ms)

Comment

Figure 30.37: Server script attributes

Script

The script to execute.

Note

QF-Test variables are expanded before the script is passed to the Jython interpreter. This is convenient for numeric values. Strings values on the other hand require quoting for the Jython interpreter and if the value of the variable contains quotes itself this can cause problems. `rc.lookup(...)` is the preferred method in this case (see [chapter 37^{\(619\)}](#) for details).

Note

In spite of syntax highlighting and automatical indentation this attribute might not be the right place to write complex scripts. There are many excellent editors that are much better suited to this task. The option [External editor command^{\(247\)}](#) lets you define an external editor in which scripts can be edited conveniently by pressing



[Alt-Return] or by clicking the  button. Complex scripts can also be written as separate modules which can then be imported for use in this attribute. See [chapter 37^{\(619\)}](#) for details.

Variable: Yes

Restrictions: Valid syntax

Script language

This attribute determines the interpreter in which to run the script, or in other words, the scripting language to use. Possible values are "Jython" and "Groovy".

Variable: No

Restrictions: None

Name

The name of a script is a kind of short description. It is displayed in the tree view, so it should be concise and tell something about the function of the script.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

Note

30.6.13 SUT script



Client side scripts are executed by an interpreter (Jython or Groovy) that QF-Test embeds into the SUT. Scripting is explained in [chapter 13^{\(99\)}](#) and [chapter 37^{\(619\)}](#).

Contained in: All kinds of [sequences^{\(314\)}](#).

Children: None

Execution: The script is executed inside the SUT by an embedded interpreter.

Attributes:

SUT script

Client
SUT

Script

```

1 # get the text field component
2 field = rc.getComponent("tfName")
3 # read and log its value
4 rc.logMessage("Name: " + field.getText())
5

```

Script language
Jython

GUI engine

Name
log name

QF-Test ID

Delay before (ms) Delay after (ms)

Comment

Figure 30.38: SUT script attributes

Client

The name of the SUT client process in which to execute the script.

Variable: Yes

Restrictions: Must not be empty.

Script

The script to execute.

Note

QF-Test variables are expanded before the script is passed to the Jython interpreter. This is convenient for numeric values. Strings values on the other hand require quoting for the Jython interpreter and if the value of the variable contains quotes itself this can cause problems. `rc.lookup(...)` is the preferred method in this case (see [chapter 37^{\(619\)}](#) for details).

Note

In spite of syntax highlighting and automatical indentation this attribute might not be the right place to write complex scripts. There are many excellent editors that are much better suited to this task. The option [External editor command^{\(247\)}](#) lets you define an external editor in which scripts can be edited conveniently by pressing

[Alt-Return](#) or by clicking the  button. Complex scripts can also be written as separate modules which can then be imported for use in this attribute. See [chapter 37^{\(619\)}](#) for details.

Variable: Yes

Restrictions: Valid syntax

Script language

This attribute determines the interpreter in which to run the script, or in other words, the scripting language to use. Possible values are "Jython" and "Groovy".

Variable: No

Restrictions: None

GUI engine

The GUI engine in which to execute the script. Only relevant for SUTs with more than one GUI engine as described in [chapter 33^{\(590\)}](#).

Variable: Yes

Restrictions: See [chapter 33^{\(590\)}](#)

Name

The name of a script is a kind of short description. It is displayed in the tree view, so it should be concise and tell something about the function of the script.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing **Alt-Return** or by clicking the  button.

Variable: Yes

Restrictions: None

30.7 Processes

In order to enable QF-Test to communicate with the SUT, the SUT's process must be started by QF-Test. For details about how the application is started by QF-Test and which kind of node is most applicable in your case, see chapter 34⁽⁵⁹²⁾.

The following nodes are used to start or stop processes, to wait for a connection with the SUT or to wait for the termination of a process and check its exit value. The number of concurrent processes managed by QF-Test is limited only by the underlying system, but you have to assign a unique name to each process by which other nodes will identify it.

We will refer to processes run by QF-Test as clients. QF-Test distinguishes between two types of clients: arbitrary processes that are simply started and stopped and SUT clients, the actual Java applications that QF-Test interacts with.

Note

Standard input and output of a client is redirected to a terminal that you can open from the **Clients** menu. The client's output is also stored in the log of a test run.

30.7.1 Start Java SUT client



This node provides the standard and most flexible way of starting an SUT client. To use it, you must know the Java command that starts your application. If the SUT is run through a script, use a Start SUT client⁽⁴²⁶⁾ node instead (see chapter 34⁽⁵⁹²⁾).

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The command line for the program is built from the attributes and the process is started. Its input and output are redirected to QF-Test.

Attributes:

Start Java SUT client

Client
SUT

Executable
\${qftest:java}

Directory

Class name
de.qfs.apps.qftest.demo.Increment

Parameters
Executable (2) \ Class (0) \ Extra (0) \

Executable parameters

Parameter
-classpath
\${classpath}

QF-Test ID

Delay before (ms) Delay after (ms)
 Delay after (ms)

Comment
Start SUT: Increment demo

Figure 30.39: Start Java SUT client attributes

Client

This is the identifier for the SUT client. It must remain unique as long as the process is alive. Other nodes will refer to the client by this name.

Variable: Yes

Restrictions: Must not be empty

Executable

The Java program to run, typically `java` or `javaw` on Windows. If you want to run a specific Java virtual machine you should give the full path name of the executable.

The file button  brings up a dialog in which you can select the program file interactively. You can also get to this dialog by pressing `[Shift-Return]` or `[Alt-Return]`, when the focus is in the text field.

Variable: Yes

Restrictions: Must not be empty

Directory

Here you can set the working directory for the program. If you leave this value empty, the program will inherit QF-Test's working directory.

The directory button  brings up a dialog in which you can select the directory interactively. You can also get to this dialog by pressing `[Shift-Return]` or `[Alt-Return]`, when the focus is in the text field.

Variable: Yes

Restrictions: Must be empty or an existing directory

Class name

The fully qualified name of the Java class of the SUT whose `main(String[])` method will be called. If your application is run from an executable jar archive with the `-jar` parameter, you must leave this attribute empty.

Note

QF-Test can only run the `main` method of the class if both the class itself and the `main` method are declared `public`. Please verify this if you get an `IllegalAccessException` when you try to start your application.

Variable: Yes

Restrictions: Valid class name

Parameters

There is one tab for each of the following parameter lists:

Executable

The command line arguments for the Java executable. Put each parameter on a line of its own. Special quoting of whitespace or symbols is not required.

For example to set the classpath, set one line to `-classpath` and the following line to the desired value of the classpath. You don't need to concern yourself with QF-Test's jar files.

Class

These are the arguments that are passed to the `main(String[])` method

of the class to be started. Again, each argument requires a row of its own and special quoting is not required.

Extra

Extra parameters for the class `de.qfs.apps.qftest.start.StartClient` which is the wrapper class started by QF-Test in order to gain access to the `EventQueue`, establish an RMI connection between the SUT and QF-Test and then start the main class of the SUT. These parameters are intended for future extensions, scripting and internal logging.

See [section 2.2.5^{\(13\)}](#) about how to work with the tables.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the [Default delay^{\(288\)}](#) from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option [External editor command^{\(247\)}](#) lets you define an external editor in which comments can be edited conveniently by

pressing `[Alt-Return]` or by clicking the  button.

Variable: Yes

Restrictions: None

30.7.2 Start SUT client



If your application is normally started through a shell script or a special executable starter, the most convenient way to run it from QF-Test is through this kind of node. Depending on how the application is actually started by this script, some modifications may be required. A typical symptom is that the SUT starts up fine, but the connection to QF-Test is not established. In that case, read [chapter 34](#)⁽⁵⁹²⁾.

Contained in: All kinds of [sequences](#)⁽³¹⁴⁾.

Children: None

Execution: The command line for the program is built from the attributes and the process is started. Its input and output are redirected to QF-Test.

Attributes:

Figure 30.40: Start SUT client attributes

Client

This is the identifier for the SUT client. It must remain unique as long as the process is alive. Other nodes will refer to the client by this name.

Variable: Yes

Restrictions: Must not be empty

Executable

The program to run. If the executable file is not located in a directory on the `PATH` you must give the full path name.

The file button  brings up a dialog in which you can select the program file interactively. You can also get to this dialog by pressing `[Shift-Return]` or `[Alt-Return]`, when the focus is in the text field.

Variable: Yes

Restrictions: Must not be empty

Directory

Here you can set the working directory for the program. If you leave this value empty, the program will inherit QF-Test's working directory.

The directory button  brings up a dialog in which you can select the directory interactively. You can also get to this dialog by pressing `[Shift-Return]` or `[Alt-Return]`, when the focus is in the text field.

This directory will become the current working directory **after** executing the program. As a result, a script named, say, `./copy_data` will be looked up relative to QF-Test's working directory and not the directory given. Only the path names referred to in the script itself will be resolved relative to the new directory.

Variable: Yes

Restrictions: Must be empty or an existing directory

Parameters

There is one tab for each of the following parameter lists:

Executable

The command line arguments for the executable. Put each parameter on a line of its own. Special quoting of whitespace or symbols is not required.

Extra

Extra parameters for the class

Note

`de.qfs.apps.qftest.start.StartClient` which is used to gain access to the `EventQueue`, establish an RMI connection between the SUT and QF-Test and then start the main class of the SUT. These parameters are intended for future extensions, scripting and internal logging.

See [section 2.2.5^{\(13\)}](#) about how to work with the tables.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters `'\'`, `'#'`, `'$'`, `'@'`, `'&'`, or `'%'` or start with an underscore (`'_'`).

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the [Default delay^{\(288\)}](#) from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option [External editor command^{\(247\)}](#) lets you define an external editor in which comments can be edited conveniently by

pressing `[Alt-Return]` or by clicking the  button.

Variable: Yes

Restrictions: None

30.7.3 Start browser



This node is used to start a web browser specifically for web testing. To test a Java applet in a browser, use a [Start SUT client^{\(426\)}](#) node instead. If the

process for the SUT is already running, this node can also be used to open an additional browser window in the same process.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The command line for the browser program is built from the attributes and the process is started. Its input and output are redirected to QF-Test.

Attributes:

Start/open browser

Client
SUT

URL
www.qfs.de

 ☐ Open new browser window in existing client

Browser type
ie

 Directory of browser installation

Name of the browser window
mainwin

Geometry of browser window

X	Y
Width	Height

 Executable
\${qftest:java}

Parameters

Java VM (0) Extra (0)

     Executable parameters

Parameter

QF-Test ID

Delay before (ms) Delay after (ms)

 Comment
Start browser with a given URL.

Figure 30.41: Start browser attributes

Client

This is the identifier for the SUT client. It must remain unique as long as the process is alive. Other nodes will refer to the client by this name.

Variable: Yes

Restrictions: Must not be empty

URL

The URL of the web page to be shown in the browser. As a special case the URL "about:nowindow" can be used to start the SUT process without opening an initial browser window. In that state, cache or preference settings for the browser can be modified which may be a prerequisite for opening a given URL. The browser window with the actual target URL can then be opened via another Start browser node.

Variable: Yes

Restrictions: Must not be empty

Open new browser window in existing client

If this attribute is set, QF-Test will try to open another browser window in an existing SUT client instead of starting a new process. Of course in that case the SUT client must already be running.

Variable: Yes

Restrictions: None

Browser type

The kind of browser to open a window for. Currently supported values on Windows are "ie" for Internet Explorer, "mozilla" for Mozilla Firefox and "chrome" for Google Chrome. On Linux, only "mozilla" is supported.

Variable: Yes

Restrictions: Legal values are "ie", "mozilla" or "chrome".

Directory of Mozilla installation

For Firefox the installation directory of the browser must be set here.

Variable: Yes

Restrictions: Must be empty (for ie and chrome) or an existing directory.

Name of the browser window

This attribute can be ignored unless you need to test a web application with multiple open browser windows holding similar documents. In that case the Name of the browser window attribute can be used to identify the browser window. Here you can specify the name for the browser window to be opened.

Variable: Yes

Restrictions: None

Geometry of browser window

These optional attributes for the X/Y coordinate, width and height can be used to specify the geometry of the browser window to open.

Variable: Yes

Restrictions: Width and height must not be negative.

Executable

The Java program to start the browser with, typically `java` or `javaw` on Windows. If you want to run a specific Java virtual machine you should give the full path name of the executable.

Note

If the web application does include a Java Applet, make sure that you specify the System JRE here that is used for the Applet plugin, otherwise the browser will crash.

The default value is `${qftest:java}` which is the Java program that QF-Test itself runs on. Currently only 32bit browsers are supported on Windows, so there you must either run QF-Test itself with a 32bit JRE or specify a 32bit Java here. On Linux you need to ensure that the Java program used here is of the same 32bit or 64bit variant as the Firefox browser specified in the Directory of Mozilla installation attribute.

The file button  brings up a dialog in which you can select the program file interactively. You can also get to this dialog by pressing `[Shift-Return]` or `[Alt-Return]`, when the focus is in the text field.

Variable: Yes

Restrictions: Must not be empty

Parameters

There is one tab for each of the following parameter lists:

Java VM

Special command line arguments for the `java` executable in which the browser is embedded. Can be used to specify the available memory (default is 200MB), define system properties, enable debugging, etc. Put each parameter on a line of its own. Special quoting of whitespace or symbols is not required.

Extra

Extra parameters for the QF-Test wrapper that embeds the browser and establishes an RMI connection between the SUT and QF-Test. These parameters are intended for future extensions, scripting and internal logging.

See [section 2.2.5^{\(13\)}](#) about how to work with the tables.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the [Default delay^{\(288\)}](#) from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option [External editor command^{\(247\)}](#) lets you define an external editor in which comments can be edited conveniently by

pressing **[Alt-Return]** or by clicking the  button.

Variable: Yes

Restrictions: None

30.7.4 Start process

Note



To run an arbitrary program during a test, you can either use this node or an Execute shell command⁽⁴³⁶⁾ node. This node is preferable if you need to pass possibly complex arguments to the executable.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The command line for the program is built from the attributes and the process is started. Its input and output are redirected to QF-Test.

Attributes:

The screenshot shows a 'Start process' dialog box with the following fields and values:

- Client:** server
- Executable:** server
- Directory:** (empty)
- Executable parameters:**
 - Parameter:** (empty)
- QF-Test ID:** (empty)
- Delay before (ms):** (empty)
- Delay after (ms):** (empty)
- Comment:** Start server process

Figure 30.42: Start process attributes

Client

This is the identifier for the SUT client. It must remain unique as long as the process is alive. Other nodes will refer to the client by this name.

Variable: Yes

Restrictions: Must not be empty

Executable

The program to run. If the executable file is not located in a directory on the `PATH` you must give the full path name.

The file button  brings up a dialog in which you can select the program file interactively. You can also get to this dialog by pressing `[Shift-Return]` or `[Alt-Return]`, when the focus is in the text field.

Variable: Yes

Restrictions: Must not be empty

Directory

Here you can set the working directory for the program here. If you leave this value empty, the program will inherit QF-Test's working directory.

The directory button  brings up a dialog in which you can select the directory interactively. You can also get to this dialog by pressing `[Shift-Return]` or `[Alt-Return]`, when the focus is in the text field.

Note

This directory will become the working directory of the newly started process. It does not affect the working directory of QF-Test. As a result, a program named, say, `./startserver` will be looked up relative to QF-Test's working directory and not the directory given. Only the path names referred to in the program itself will be resolved relative to the given directory.

Variable: Yes

Restrictions: Must be empty or an existing directory

Parameters

The command line arguments for the executable. Put each parameter on a line of its own. Special quoting of whitespace or symbols is not required.

See [section 2.2.5^{\(13\)}](#) about how to work with the tables.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters `'\'`, `'#'`, `'$'`, `'@'`, `'&'`, or `'%'` or start with an underscore (`'_'`).

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.7.5 Execute shell command



This node is a convenient way to execute shell commands during a test. The shell that will execute the command can be specified with the command line arguments -shell <executable>⁽⁵⁸³⁾ and -shellarg <argument>⁽⁵⁸³⁾ for QF-Test. The default for Unix is `/bin/sh`, on Windows either `command.com` or `cmd.exe` is used, the value of the COMSPEC environment variable.

When the shell is started it is treated like every other process started by QF-Test, so you can kill it or wait for it to terminate and check its exit code.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: A shell is started to execute the command. Its input and output are redirected to QF-Test.

Attributes:

Figure 30.43: Execute shell command attributes

Client

This is the identifier for the SUT client. It must remain unique as long as the process is alive. Other nodes will refer to the client by this name.

Variable: Yes

Restrictions: Must not be empty

Shell command

The command to execute in the shell. Enter this just as you would at the command prompt.

Windows

On Windows systems, quoting of arguments with blanks can be a little tricky. If you're using the standard Windows shell, simply use double quotes as always, for example `dir "C:\Program Files"`. If you're using a Unix shell on Windows by specifying the `-shell <executable>`⁽⁵⁸³⁾ command line argument, use single quotes instead, i.e. `ls 'C:/Program Files'`.

Variable: Yes

Restrictions: Must not be empty

Directory

Here you can set the working directory for the shell. If you leave this value empty, the shell will inherit QF-Test's working directory.

The directory button  brings up a dialog in which you can select the directory interactively. You can also get to this dialog by pressing `[Shift-Return]` or `[Alt-Return]`, when the focus is in the text field.

Variable: Yes

Restrictions: Must be empty or an existing directory

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing `[Alt-Return]` or by clicking the  button.

Variable: Yes

Restrictions: None

30.7.6 Wait for client to connect



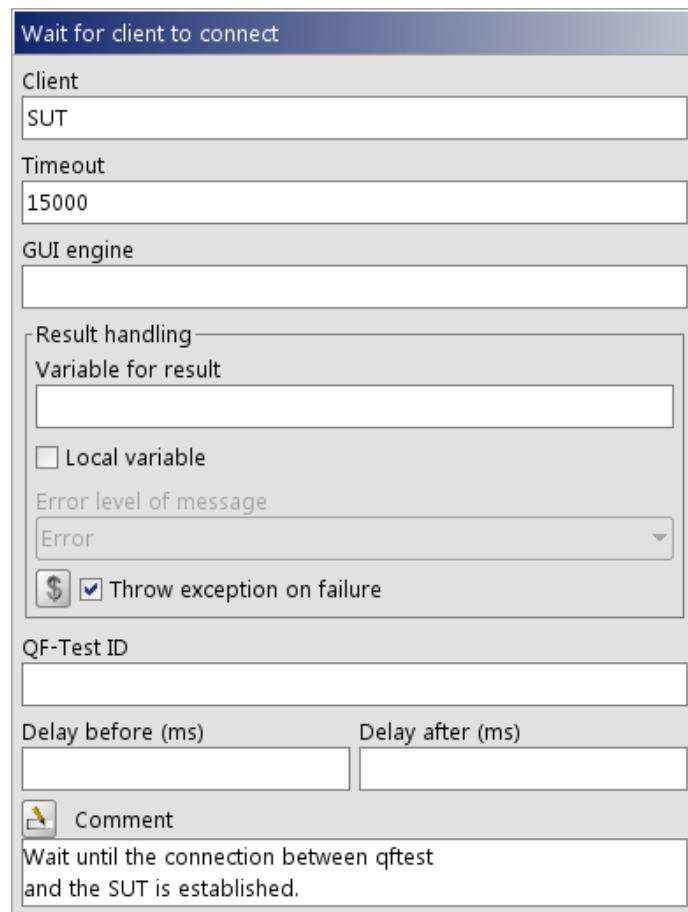
This node is used to make sure that the connection to a client is established. If after a configurable timeout the client is still not connected, a ClientNotConnectedException⁽⁵⁶²⁾ is thrown. You can also use the Variable for result attribute to store the result into a variable and the Throw exception on failure attribute to suppress the exception.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: QF-Test waits until either the connection to the Java client is established or the timeout is up.

Attributes:



Wait for client to connect

Client
SUT

Timeout
15000

GUI engine

Result handling
Variable for result

☐ Local variable

Error level of message
Error

☒ Throw exception on failure

QF-Test ID

Delay before (ms) Delay after (ms)

 Comment
Wait until the connection between qfTest
and the SUT is established.

Figure 30.44: Wait for client to connect attributes

Client

The name of the Java client to wait for.

Variable: Yes

Restrictions: Must not be empty

Timeout

Time limit in milliseconds.

Variable: Yes

Restrictions: ≥ 0

GUI engine

The GUI engine to wait for. Only relevant for SUTs with more than one GUI engine as described in [chapter 33^{\(590\)}](#).

Variable: Yes

Restrictions: See [chapter 33^{\(590\)}](#)

Variable for result

This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

Note

If this attribute is set, the attribute Error level of message is ignored and no error is reported. The attribute Throw exception on failure always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing [Procedure^{\(378\)}](#), [Dependency^{\(344\)}](#) or [Test-case^{\(314\)}](#) node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test-case node or, if there is no such node, in the global bindings. See [chapter 7^{\(57\)}](#) for a detailed explanation of variable binding and lookup.

Variable: No

Restrictions: None

Error level of message

This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

Note

If the attribute Throw exception on failure is set, this attribute is irrelevant and if Variable for result is set this attribute is ignored.

Variable: No

Restrictions: None

Throw exception on failure

Throw an exception in case of failure. For 'Check...' nodes a `CheckFailedException`⁽⁵⁶¹⁾ is thrown, for 'Wait for...' nodes the respective specific exception.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.7.7 Stop client

This node forcibly terminates a client process started by QF-Test. If the process has already terminated, nothing is changed. Otherwise, if the client is an SUT client, QF-Test first tries to call the Java method `System.exit(-1)` in the SUT to achieve a clean shutdown. If the client is not an SUT client or the process is still alive after the `exit` call, the process is killed and its exit code is set to -1.

Windows

Note: On Windows child processes started by a process are not terminated when the parent process is killed. As explained in [chapter 34^{\(592\)}](#), the `java` program for the SUT is not started directly by QF-Test, but through an intermediate program. This means that when the `System.exit(-1)` call fails to terminate the SUT for some reason, the process for the SUT's Java VM will be left hanging around.

Contained in: All kinds of [sequences^{\(314\)}](#).

Children: None

Execution: Terminates the last process that was started under the given name.

Attributes:

Stop client	
Client	
SUT	
QF-Test ID	
Delay before (ms)	Delay after (ms)
Comment	
Kill the SUT's process	

Figure 30.45: Stop client attributes

Client

The client name of the process that is to be killed.

Variable: Yes

Restrictions: Must not be empty

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters `'\'`, `'#'`, `'$'`, `'@'`, `'&'`, or `'%'` or start with an underscore (`'_'`).

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the [Default delay^{\(288\)}](#) from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing `[Alt-Return]` or by clicking the  button.

Variable: Yes

Restrictions: None

30.7.8 Wait for process to terminate



This node waits until a process that was started by QF-Test terminates. If a given time limit is exceeded and the process is still alive. Per default a `ClientNotTerminatedException`⁽⁵⁶³⁾ is thrown. Otherwise the client's exit code is read and tested against a given value. You can also use the Variable for result attribute to store the result into a variable and the Throw exception on failure attribute to suppress the exception.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: Waits for the end of a process and checks its exit value unless the timeout is exceeded.

Attributes:

Wait for process to terminate	
Client	SUT
Timeout	3000
Expected exit code	0
Result handling	
Variable for result	
☐ Local variable	
Error level of message	Error
☒ Throw exception on failure	
QF-Test ID	
Delay before (ms)	
Delay after (ms)	
 Comment	
Wait for the SUT to exit and check its exit value	

Figure 30.46: Wait for process to terminate attributes

Client

The client name of the process to wait for.

Variable: Yes

Restrictions: Must not be empty

Timeout

Time limit in milliseconds.

Variable: Yes

Restrictions: ≥ 0

Expected exit code

If this attribute is set it is used to validate the exit code of the process. The type of the comparison is defined by prepending one of the four operators `==`, `!=`, `<` and

>. Just a number without preceding operation test for equality. If the test fails, an `UnexpectedExitCodeException`⁽⁵⁶³⁾ is thrown. The exception won't be thrown, if the attribute `Throw exception on failure` is not set.

Examples: If this attribute is set to **0**, every exit code that is not 0 causes an exception. This is the same as for **==0**. A value of **>0** causes an exception for every exit code equal to or less than 0. This exception can be suppressed by un-checking the attribute `Throw exception on failure`.

Variable: Yes

Restrictions: See above

Variable for result

This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

Note

If this attribute is set, the attribute `Error level of message` is ignored and no error is reported. The attribute `Throw exception on failure` always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing `Procedure`⁽³⁷⁸⁾, `Dependency`⁽³⁴⁴⁾ or `Test-case`⁽³¹⁴⁾ node. If no such binding exists, a new binding is created in the currently executing `Procedure`, `Dependency` or `Test-case` node or, if there is no such node, in the global bindings. See [chapter 7](#)⁽⁵⁷⁾ for a detailed explanation of variable binding and lookup.

Variable: No

Restrictions: None

Error level of message

This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

Note

If the attribute `Throw exception on failure` is set, this attribute is irrelevant and if `Variable for result` is set this attribute is ignored.

Variable: No

Restrictions: None

Throw exception on failure

Throw an exception in case of failure. For 'Check...' nodes a `CheckFailedException`⁽⁵⁶¹⁾ is thrown, for 'Wait for...' nodes the respective specific exception.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.8 Events

This section lists all kinds of nodes that trigger actions in the SUT. Besides true Java events these include pseudo events with special behavior.

30.8.1 Mouse event



Mouse events simulate mouse movement and clicks as well as drag and drop operations.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The coordinates and other attributes of the event are sent to the SUT together with the data about the target component. The `TestEventQueue` determines the corresponding component in the SUT, adjusts the coordinates and triggers the resulting event.

Attributes:

Mouse event	
Client	
SUT	
QF-Test component ID	
bExit	
Event details	
Mouse click	
X	Y
1	1
Modifiers	Click count
16	1
☐ Popup trigger	
☐ Replay as "hard" event	
QF-Test ID	
Delay before (ms)	Delay after (ms)
Comment	
Move the mouse pointer to the Exit button.	

Figure 30.47: Mouse event attributes

Client

The name of the SUT client process to which the event is sent.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID⁽⁵³⁶⁾ of the Window⁽⁵³⁵⁾, Component⁽⁵⁴⁵⁾ or Item⁽⁵⁵¹⁾ node that is the target of the event.



The  button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing **[Shift-Return]** or **[Alt-Return]**, when the focus is in the text field. As an alternative you can copy the target node with **[Ctrl-C]** or **[Edit→Copy]** and insert its QF-Test component ID into the text field by pressing **[Ctrl-V]**.

By using a special syntax you can target Items⁽⁵⁵¹⁾ directly or access components from another suite (see section 6.3⁽⁵⁴⁾ and section 17.1⁽¹⁵⁰⁾ for details).

Variable: Yes

Restrictions: Must not be empty.

Event

This *ComboBox* lets you choose the type of the event. `MOUSE_MOVED`, `MOUSE_PRESSED`, `MOUSE_RELEASED`, `MOUSE_CLICKED` and `MOUSE_DRAGGED` are the standard event IDs of the Java class `MouseEvent`.

The abstract 'Mouse click' event is a compound of the events `MOUSE_MOVED`, `MOUSE_PRESSED`, `MOUSE_RELEASED` and `MOUSE_CLICKED`. During replay the pseudo event is simulated as four separate events. This adds to the clarity of a test-suite and simplifies editing.

The events `MOUSE_DRAG_FROM`, `MOUSE_DRAG_OVER` and `MOUSE_DROP_TO` are used to simulate Drag&Drop in the SUT. See section 36.1⁽⁶¹¹⁾ for details.

Variable: No

Restrictions: None

X/Y

These are the coordinates of the *MouseEvent*. They are relative to the upper left corner of the Window⁽⁵³⁵⁾, Component⁽⁵⁴⁵⁾ or Item⁽⁵⁵¹⁾ that is the target of the event. They can be negative, e.g. to simulate a click on the expansion toggle of a node in a `JTree`.

Most of the time the exact coordinates for a mouse don't really matter, any place within the target will do. In this case you should leave the X and Y values empty to

tell QF-Test to aim at the center of the target. Where possible QF-Test will leave the values empty when recording, provided the option Record MouseEvents without coordinates where possible ⁽²⁵⁶⁾ is active.

Variable: Yes

Restrictions: Valid number or empty

Modifiers

This value reflects the state of the mouse buttons and the modifier keys **Shift**, **Control**, **Alt** and **Meta** during a mouse or key event. States are combined by adding up their values.

Value	Key/Button
1	Shift
2	Control
4	Meta or right mouse button
8	Alt or middle mouse button
16	Left mouse button

Table 30.16: Modifier values

Variable: Yes

Restrictions: Valid number

Click count

This value lets a Java program distinguish between a single and a double (or even multiple) click.

Variable: Yes

Restrictions: Valid number

Popup trigger

If this attribute is set, the event can trigger a *PopupMenu*. This is Java's somewhat peculiar way of supporting different conventions for triggering *Popups* on different platforms.

Variable: Yes

Restrictions: None

Replay as "hard" event

If this attribute is set the event is replayed as a hard event, meaning it is triggered as a real system event that moves the mouse around and not just inserted as soft

event into the event queue. Soft events are typically better because they avoid impact of concurrent user mouse actions and are less likely to break due to interference from an overlapping window. Nevertheless there are certain special situations where hard events are helpful.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.8.2 Key event



Key events simulate keyboard input in the SUT. These are used mainly for control and function keys. Input of text is better represented as a Text input⁽⁴⁵⁴⁾.

The special keyboard event `InputMethodEvent` supports international character codes. QF-Test doesn't support `InputMethodEvents` directly. Instead, it

converts events of type `INPUT_METHOD_TEXT_CHANGED` into Key events of type `KEY_TYPED`.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The key codes of the event are sent to the SUT together with the data about the target component. The `TestEventQueue` determines the corresponding component in the SUT and triggers the resulting event.

Attributes:

The screenshot shows a 'Key event' dialog box with the following fields and sections:

- Client:** A text field containing 'SUT'.
- QF-Test component ID:** A text field containing 'winMain'.
- Event details:** A section containing:
 - A dropdown menu set to 'Keystroke'.
 - A text field containing 'Key: Enter'.
 - A table with three columns: 'Key code', 'Key char', and 'Modifiers'. The values are '10', '10', and '0' respectively.
- QF-Test ID:** An empty text field.
- Delay before (ms):** An empty text field.
- Delay after (ms):** An empty text field.
- Comment:** A text area containing 'Activate default button with RETURN.'

Figure 30.48: Key event attributes

Client

The name of the SUT client process to which the event is sent.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID⁽⁵³⁶⁾ of the Window⁽⁵³⁵⁾, Component⁽⁵⁴⁵⁾ or Item⁽⁵⁵¹⁾ node that is the target of the event.



The  button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing **[Shift-Return]** or **[Alt-Return]**, when the focus is in the text field. As an alternative you can copy the target node with **[Ctrl-C]** or **[Edit→Copy]** and insert its QF-Test component ID into the text field by pressing **[Ctrl-V]**.

By using a special syntax you can target Items⁽⁵⁵¹⁾ directly or access components from another suite (see section 6.3⁽⁵⁴⁾ and section 17.1⁽¹⁵⁰⁾ for details).

Variable: Yes

Restrictions: Must not be empty.

Event

This *ComboBox* lets you choose the type of the event. `KEY_PRESSED`, `KEY_TYPED` and `KEY_RELEASED` are the standard event IDs of the Java class `KeyEvent`.

The 'Keystroke' pseudo event is a compound of the events `KEY_PRESSED`, `KEY_TYPED` and `KEY_RELEASED`. During replay the pseudo event is simulated as two or three separate events, depending on whether it is a printable character key, or a control or function key. In the latter case, no `KEY_TYPED` event is generated.

Variable: No

Restrictions: None

Key

This is a convenience method to set Key code⁽⁴⁵²⁾, Key char⁽⁴⁵²⁾ and Modifiers⁽⁴⁵³⁾ directly by pressing the corresponding key while this component has the keyboard focus. For `KEY_TYPED` events Key char⁽⁴⁵²⁾ is set to 0.

Unfortunately you can't select the **[Tab]** key this way since it is used for keyboard traversal. Key code⁽⁴⁵²⁾ and Key char⁽⁴⁵²⁾ for the **[Tab]** key are both 9.

Variable: No

Restrictions: None

Key code

This is a Java specific code for the key, the `keyCode` member of the Java class `KeyEvent`.

Variable: Yes

Restrictions: Valid number

Key char

This is the `keyChar` member of the Java class `KeyEvent`. Its value is the character generated created by the last key press, taking the state of the `Shift` key into account. Control and function keys always have a Key char value of 65535.

Variable: Yes

Restrictions: Valid number

Modifiers

This value reflects the state of the mouse buttons and the modifier keys `Shift`, `Control`, `Alt` and `Meta` during a mouse or key event. States are combined by adding up their values.

Value	Key/Button
1	Shift
2	Control
4	Meta or right mouse button
8	Alt or middle mouse button
16	Left mouse button

Table 30.17: Modifier values

Variable: Yes

Restrictions: Valid number

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters `'\'`, `'#'`, `'$'`, `'@'`, `'&'`, or `'%'` or start with an underscore (`'_'`).

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.8.3 Text input

 This is a pseudo event for simulating text input from the keyboard. A single Text input node replaces a whole sequence of Key event⁽⁴⁵⁰⁾ nodes. To achieve this, QF-Test takes advantage of the fact that AWT and Swing text fields listen only to `KEY_TYPED` events, so input of text can be simulated by triggering one `KEY_TYPED` event per text character. If your SUT uses custom text components that actually require the `KEY_PRESSED` and `KEY_RELEASED` events, you cannot use this node type but have to resort to plain Key events⁽⁴⁵⁰⁾.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The text is sent to the SUT together with the data about the target component. The `TestEventQueue` determines the corresponding component in the SUT and triggers the resulting events.

Attributes:

Figure 30.49: Text input attributes

Client

The name of the SUT client process to which the event is sent.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID⁽⁵³⁶⁾ of the Window⁽⁵³⁵⁾, Component⁽⁵⁴⁵⁾ or Item⁽⁵⁵¹⁾ node that is the target of the event.



The  button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing **[Shift-Return]** or **[Alt-Return]**, when the focus is in the text field. As an alternative you can copy the target node with **[Ctrl-C]** or **[Edit→Copy]** and insert its QF-Test component ID into the text field by pressing **[Ctrl-V]**.

By using a special syntax you can target Items⁽⁵⁵¹⁾ directly or access components from another suite (see section 6.3⁽⁵⁴⁾ and section 17.1⁽¹⁵⁰⁾ for details).

Variable: Yes

Restrictions: Must not be empty.

Text

The text that is to be sent to the SUT.

Note

When inserting text into a password field it may be desirable to avoid having the password show up as plain text in the test-suite or a run-log. To that end the password can be crypted by inserting the plain-text password, right-clicking and selecting Crypt password from the popup menu. Please be sure to specify a password salt before crypting via the option Salt for crypting passwords⁽²⁷²⁾.

Variable: Yes

Restrictions: No line breaks are possible.

Clear target component first

If this attribute is set, and the target component is a text field or text area, the contents of the target component are removed before the new text is inserted.

Variable: Yes

Restrictions: None

Replay single events

If the target component is a text field or a text area, the text can optionally be inserted by manipulating the component directly through its API. This is much faster, but if KeyListeners are registered on the component they will not be notified of the change. If this attribute is set, key events are simulated for every single character.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing **Alt-Return** or by clicking the  button.

Variable: Yes

Restrictions: None

30.8.4 Window event

WindowEvents are of limited use for a test-suite since most of them are not generated as a direct consequence of some user interaction. The only exception is the `WINDOW_CLOSING` event that is triggered when the user closes a window. It is also possible to simulate `WINDOW_ICONIFIED` and `WINDOW_DEICONIFIED` events.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The event is sent to the SUT together with the data about the target window. The `TestEventQueue` determines the corresponding window in the SUT and triggers the resulting event.

Attributes:

The screenshot shows a 'Window event' dialog box with the following fields and options:

- Client:** A text field containing 'SUT'.
- QF-Test component ID:** A text field containing 'winMain'.
- Event details:** A dropdown menu showing 'WINDOW_CLOSING'.
- QF-Test ID:** An empty text field.
- Delay before (ms):** An empty text field.
- Delay after (ms):** An empty text field.
- Comment:** A text field containing 'Close main window.'

Figure 30.50: Window event attributes

Client

The name of the SUT client process to which the event is sent.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID⁽⁵³⁶⁾ of the Window⁽⁵³⁵⁾, Component⁽⁵⁴⁵⁾ or Item⁽⁵⁵¹⁾ node that is the target of the event.



The  button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing **[Shift-Return]** or **[Alt-Return]**, when the focus is in the text field. As an alternative you can copy the target node with **[Ctrl-C]** or **[Edit→Copy]** and insert its QF-Test component ID into the text field by pressing **[Ctrl-V]**.

By using a special syntax you can target Items⁽⁵⁵¹⁾ directly or access components from another suite (see section 6.3⁽⁵⁴⁾ and section 17.1⁽¹⁵⁰⁾ for details).

Variable: Yes

Restrictions: Must not be empty.

Event

This *ComboBox* lets you choose the type of the event. Possible values are `WINDOW_OPENED`, `WINDOW_CLOSING`, `WINDOW_CLOSED`, `WINDOW_ACTIVATED`, `WINDOW_DEACTIVATED`, `WINDOW_ICONIFIED` and `WINDOW_DEICONIFIED`, the standard event IDs of the Java class `WindowEvent`.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters `'\'`, `'#'`, `'$'`, `'@'`, `'&'`, or `'%'` or start with an underscore (`'_'`).

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing `[Alt-Return]` or by clicking the  button.

Variable: Yes

Restrictions: None

30.8.5 Component event

The name Component event for this node may be misleading, since QF-Test filters all of these events except for windows, but since they represent the Java class `ComponentEvent`, it is better to stick to that name. Except for

`COMPONENT_MOVED` and `COMPONENT_SIZED` events on a window, which are the result of the user moving or resizing the window interactively, all *ComponentEvents* are artificial and thus ignored.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The event is sent to the SUT together with the data about the target window. The `TestEventQueue` determines the corresponding window in the SUT and triggers the resulting event.

Attributes:

The screenshot shows a 'Component event' dialog box. It has a title bar 'Component event'. Inside, there are several sections: 'Client' with a text field containing 'SUT'; 'QF-Test component ID' with a text field containing 'winMain'; 'Event details' with a dropdown menu set to 'COMPONENT_RESIZED'; 'X/Width' with a text field containing '600'; 'Y/Height' with a text field containing '400'; 'QF-Test ID' with an empty text field; 'Delay before (ms)' and 'Delay after (ms)' with empty text fields; and 'Comment' with a text field containing 'Resize main window to 600x400.'.

Figure 30.51: Component event attributes

Client

The name of the SUT client process to which the event is sent.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID⁽⁵³⁶⁾ of the Window⁽⁵³⁵⁾, Component⁽⁵⁴⁵⁾ or Item⁽⁵⁵¹⁾ node that is the target of the event.



The  button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing **[Shift-Return]** or **[Alt-Return]**, when the focus is in the text field. As an alternative you can copy the target node with **[Ctrl-C]** or **[Edit→Copy]** and insert its QF-Test component ID into the text field by pressing **[Ctrl-V]**.

By using a special syntax you can target Items⁽⁵⁵¹⁾ directly or access components from another suite (see section 6.3⁽⁵⁴⁾ and section 17.1⁽¹⁵⁰⁾ for details).

Variable: Yes

Restrictions: Must not be empty.

Event

This *ComboBox* lets you choose the type of the event. Possible values are `COMPONENT_SIZED` and `COMPONENT_MOVED`, the standard event IDs of the Java class `ComponentEvent`.

Variable: No

Restrictions: None

Y/Height

For a `COMPONENT_MOVED` event set this to the new Y-coordinate of the window, for a `COMPONENT_SIZED` event to its new height.

Variable: Yes

Restrictions: Valid number, height > 0

X/Width

For a `COMPONENT_MOVED` event set this to the new X-coordinate of the window, for a `COMPONENT_SIZED` event to its new width.

Variable: Yes

Restrictions: Valid number, width > 0

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters `'\'`, `'#'`, `'$'`, `'@'`, `'&'`, or `'%'` or start with an underscore (`'_'`).

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.8.6 Selection

A Selection node represents an abstract event like selecting a menu item, choosing an entry in a combo box or selecting something from or closing a system dialog. Currently this event node is used only for SWT and web SUT clients, where some events cannot be triggered by "soft" mouse events, . The alternative of using "hard" events has some disadvantages as described for the Replay as "hard" event⁽⁴⁴⁹⁾ attribute of a Mouse event⁽⁴⁴⁷⁾.

The Detail⁽⁴⁶⁵⁾ attribute determines the kind of operation to perform, or the value to select, depending on the target component.

SWT

The following combinations of widgets and Detail values are currently supported for SWT:

Class	Detail attribute	Action
CCombo sub-item	Empty	Choose item as current value
ColorDialog	Color value in hexadecimal #rrggbb format	Select given color
ColorDialog	CANCEL	Abort color selection
Combo sub-item	Empty	Choose item as current value
CTabFolder sub-item	Empty	Select tab
CTabFolder sub-item	close	Close tab
DirectoryDialog	Directory name	Select given directory
DirectoryDialog	CANCEL	Abort directory selection
FileDialog	File name, including directory	Select given file
FileDialog	CANCEL	Abort file selection
FontDialog	Font description, system specific	Select given font
FontDialog	CANCEL	Abort file selection
Menu	close	Cancel the menu (close without selection)
MenuItem	Empty	Select item
MessageBox	Any of OK YES NO CANCEL ABORT RETRY IGNORE OPEN SAVE	Close with the given value as the user's choice
ProgressBar	ProgressBar value	Set the given value
Scale	Scale value	Set the given value
Slider	Slider value	Set the given value
Spinner	Spinner value	Set the given value
TabFolder sub-item	Empty	Select tab

Table 30.18: Supported SWT widgets for a Selection event

Note

Eclipse/RCP makes heavy use of dynamic `CTabFolders` where the items can be moved between folders. The items represent the actual business objects whereas the folders are just scaffolding to hold them. Besides, the layout of the folders and items can change drastically when switching perspectives. Thus it is often desirable to be able to select or close an item independent of the `CTabFolder` it currently resides in. This can be done by using the Procedures⁽³⁷⁸⁾ `qfs.qft#qfs.swt.ctabfolder.selectTab` and `qfs.qft#qfs.swt.ctabfolder.closeTab`, provided in the `qfs.qft` standard library. Besides the ubiquitous `client` parameter, the only other parameter `tabname` must be set to the name of the item to select or close.

Web

The following combinations of DOM nodes and Detail values are currently supported for web SUT clients:

Node type	Detail attribute	Action
Confirmation dialog	Any of OK YES NO CANCEL RETRY	Close the dialog with the given value as the user's choice
File dialog for download	The file to save to or CANCEL	Close the dialog and save to the specified file or abort the download
Login dialog	Username Password or CANCEL	Close the dialog and login with the specified data or abort. The password can be crypted by right-clicking the Detail field and selecting Crypt password . Please be sure to specify a password salt before crypting via the option Salt for crypting passwords⁽²⁷²⁾ .
Prompt dialog	The text to enter or CANCEL	Close the dialog and return the specified text or abort
Top-level DOCUMENT	back	Navigate back to the previous page
Top-level DOCUMENT	forward	Navigate forward to the next page
Top-level DOCUMENT	goto:URL	Navigate to the specified URL
Top-level DOCUMENT	refresh	Reload the current page
Top-level DOCUMENT	stop	Stop loading the current page
OPTION or SELECT sub-item	0	Choose the OPTION as current value
OPTION or SELECT sub-item	1	Add the OPTION to the selection
OPTION or SELECT sub-item	-1	Remove the OPTION from the selection

Table 30.19: Supported DOM nodes for a Selection event

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The event is sent to the SUT together with the data about the target component. The component is resolved and an action performed which depends on the type of component as listed in the table above.

Attributes:

Figure 30.52: Selection attributes

Client

The name of the SUT client process to which the event is sent.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID⁽⁵³⁶⁾ of the Window⁽⁵³⁵⁾, Component⁽⁵⁴⁵⁾ or Item⁽⁵⁵¹⁾ node that is the target of the event.



The  button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing **Shift-Return** or **Alt-Return**, when the focus is in the text field. As an alternative you can copy the target node with **Ctrl-C** or **Edit→Copy** and insert its QF-Test component ID into the text field by pressing **Ctrl-V**.

By using a special syntax you can target Items⁽⁵⁵¹⁾ directly or access components from another suite (see section 6.3⁽⁵⁴⁾ and section 17.1⁽¹⁵⁰⁾ for details).

Variable: Yes

Restrictions: Must not be empty.

Detail

The Detail attribute specifies the kind of operation to perform or the value to select, depending on the target component. The possible combinations are listed in detail above.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.8.7 File selection

A File selection is a pseudo event that you only need in a special case.

If your SUT uses the standard AWT file selection dialog, implemented by the class `java.awt.FileDialog`, QF-Test has no chance to record the events the user generates while selecting a file, since they are all handled by the underlying

system and never passed on to Java. For the same reason, the selection cannot be simulated as a sequence of mouse and key events. This is not the case with Swing's `javax.swing.JFileChooser` which is implemented as a normal dialog.

Therefore QF-Test just records the result of the file selection in the form of this node and stores that data in the FileDialog upon replay before closing it. For the SUT there is no difference to an actual selection by the user.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: File and directory are stored in an open `java.awt.FileDialog` and the dialog is closed. If no FileDialog is open, a `ComponentNotFoundException`⁽⁵⁵⁸⁾ is thrown.

Attributes:

File selection	
Client	SUT
File	test.dat
Directory	/tmp
GUI engine	
QF-Test ID	
Delay before (ms)	Delay after (ms)
Comment	Select file /tmp/test.dat.

Figure 30.53: File selection attributes

Client

The name of the SUT client process to which the event is sent.

Variable: Yes

Restrictions: Must not be empty.

File

The name of the file (without the directory part) that is to be selected.

Variable: Yes

Restrictions: Must not be empty

Directory

The directory of the file that is to be selected.

Variable: Yes

Restrictions: Must not be empty

GUI engine

The GUI engine in which to look for a file selection dialog. Only relevant for SUTs with more than one GUI engine as described in [chapter 33^{\(590\)}](#).

Variable: Yes

Restrictions: See [chapter 33^{\(590\)}](#)

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the [Default delay^{\(288\)}](#) from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option [External editor command^{\(247\)}](#) lets you define an external editor in which comments can be edited conveniently by

pressing **Alt-Return** or by clicking the  button.

Variable: Yes

Restrictions: None

Note

30.9 Checks

Checks are the means by which QF-Test validates correctness of the SUT. If the current state of a component doesn't match the expected value of the check node a message with a configurable error level is written to the run-log. Additionally or alternatively a `CheckFailedException`⁽⁵⁶¹⁾ can be thrown and the result of the check can be assigned to a variable.

By setting a `Timeout`⁽⁴⁷⁴⁾ checks can also be used to wait for a component to attain a certain state, e.g. to wait for a `MenuItem` to become enable or for a `CheckBox` to become selected.

Like `events`⁽⁴⁴⁶⁾ each check must have a target window, component or sub-item. Depending on that target, different kinds of checks are supported which can be recorded in check-mode as described in section 4.3⁽³³⁾ via reight-clicking the target component and selecting the check from the resulting menu. In case you need a check for a special component that is not offered by default you can implement custom checks via the `Checker` extension API described in section 39.3⁽⁷¹⁵⁾.

There are six different data types available for checks and each of these corresponds to a specific check node. Because it is possible to have different checks of the same data type for the same target component, e.g. for the enabled state and the editable state of a text field, both boolean, checks are further qualified by their `Check` type identifier attribute. In most cases, when data type and target component are sufficient to uniquely identify the kind of check, the value of this attribute is *default*. In case the specified check is not supported for the given target component a `CheckNotSupportedException`⁽⁵⁶¹⁾ is thrown.

By default a Boolean check provides the following Check types (if the respective component supports it):

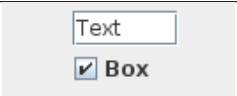
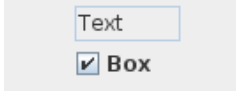
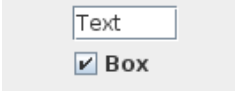
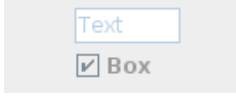
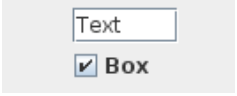
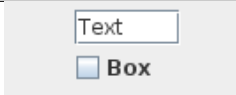
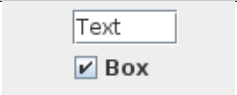
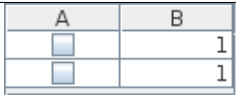
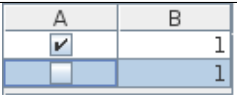
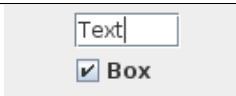
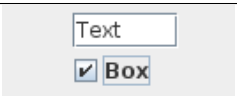
Check	Example 1	Example 2	Details
visible			Example 1 shows a invisible Textfield and CheckBox.. Both are visible at Example 2.
editable			Example 1 shows a Textfield which is not editable. The Textfield at Example 2 is editable. The CheckBox does not support this kind of check.
enabled			Neither the Textfield nor the Check-Box is enabled at Example 1, which means you cannot interact with them. Example 2 shows enabled components.
selected			Example 1 shows a CheckBox which is not selected. It is selected at Example 2. The Textfield does not support this kind of check.
selected (Table)			Example 1 shows a Table where no cell is selected. The lower left cell is selected at Example 2, indicated by a frame. The check concerns the selection, not the CheckBox.
focused			Example 1 shows a focused Textfield (indicated by the cursor). The CheckBox is focused at Example 2 (indicated by a frame).

Table 30.20: Provided Check types of Boolean check

30.9.1 Check text



Validates the text displayed in a component or sub-item.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The text is sent to the SUT together with the data about the target component. The `TestEventQueue` determines the corresponding component in the SUT, reads its displayed text and compares it to the required value.

Attributes:

Check text

Client
SUT

QF-Test component ID
tFirstName

Text
Greg

☐ As regexp

☐ Negate

Check type identifier
default

Timeout

Result handling

Variable for result

☐ Local variable

Error level of message
Error

☐ Throw exception on failure

Name

QF-Test ID

Delay before (ms)

Delay after (ms)

Comment
Check text in "First name" field

Figure 30.54: Check text attributes

Client

The name of the SUT client process to which the check is sent.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID⁽⁵³⁶⁾ of the Window⁽⁵³⁵⁾, Component⁽⁵⁴⁵⁾ or Item⁽⁵⁵¹⁾ node that is the target of the check.



The  button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing **Shift-Return** or **Alt-Return**, when the focus is in the text field. As an alternative you can copy the target node with **Ctrl-C** or **Edit→Copy** and insert its QF-Test component ID into the text field by pressing **Ctrl-V**.

By using a special syntax you can target Items⁽⁵⁵¹⁾ directly or access components from another suite (see section 6.3⁽⁵⁴⁾ and section 17.1⁽¹⁵⁰⁾ for details).

Variable: Yes

Restrictions: Must not be empty.

Text

The value to which the text displayed by the component or sub-item is compared.

You can select **Escape text for regular expressions** from the context menu for escaping special characters of regular expressions of that text.

Variable: Yes

Restrictions: Valid regexp if required.

As regexp

If this attribute is set, the component's text is matched against a regexp (see section 36.4⁽⁶¹⁴⁾) instead of comparing plain strings.

Variable: Yes

Restrictions: None

Check type identifier

This attribute specifies the kind of check to perform. This makes it possible to support different kinds of checks of the same data type for a given target component without any ambiguity. With the help of a `Checker` additional check types can be implemented as shown in section 39.3⁽⁷¹⁵⁾.

Variable: Yes

Restrictions: Must not be empty

Negate

This flag determines whether to execute a positive or a negative check. If it is set,

a negative check will be performed, i.e. the checked property must not match the expected value.

Variable: Yes

Restrictions: None

Timeout

Time limit in milliseconds until the check must succeed. To disable waiting, leave this value empty or set it to 0.

Variable: Yes

Restrictions: Must not be negative.

Variable for result

This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

If this attribute is set, the attribute Error level of message is ignored and no error is reported. The attribute Throw exception on failure always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure⁽³⁷⁸⁾, Dependency⁽³⁴⁴⁾ or Test-case⁽³¹⁴⁾ node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test-case node or, if there is no such node, in the global bindings. See chapter 7⁽⁵⁷⁾ for a detailed explanation of variable binding and lookup.

Variable: No

Restrictions: None

Error level of message

This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

If the attribute Throw exception on failure is set, this attribute is irrelevant and if Variable for result is set this attribute is ignored.

Variable: No

Note

Note

Restrictions: None

Throw exception on failure

Throw an exception in case of failure. For 'Check...' nodes a CheckFailedException⁽⁵⁶¹⁾ is thrown, for 'Wait for...' nodes the respective specific exception.

Variable: No

Restrictions: None

Name

An optional name for the Check, mostly useful for the report.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

Note

30.9.2 Boolean check



Compares the current state of a component or sub-item with an expected value.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The expected state is sent to the SUT together with the data about the target component. The `TestEventQueue` determines the corresponding component in the SUT, determines its state and compares it to the required value.

Attributes:

Boolean state check

Client
SUT

QF-Test component ID
bExit

☒ Expected state

Check type identifier
enabled

Timeout

Result handling
Variable for result

☐ Local variable

Error level of message
Error

☐ Throw exception on failure

Name

QF-Test ID

Delay before (ms) Delay after (ms)

Comment
Exit Button must be enabled

Figure 30.55: Boolean check attributes

Client

The name of the SUT client process to which the check is sent.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID⁽⁵³⁶⁾ of the Window⁽⁵³⁵⁾, Component⁽⁵⁴⁵⁾ or Item⁽⁵⁵¹⁾ node that is the target of the check.



The  button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing `[Shift-Return]` or `[Alt-Return]`, when the focus is in the text field. As an alternative you can copy the target node with `[Ctrl-C]` or `[Edit→Copy]` and insert its QF-Test component ID into the text field by pressing `[Ctrl-V]`.

By using a special syntax you can target `Items(551)` directly or access components from another suite (see [section 6.3^{\(54\)}](#) and [section 17.1^{\(150\)}](#) for details).

Variable: Yes

Restrictions: Must not be empty.

Expected state

The value to which the current state of the component or sub-item is compared.

Variable: Yes

Restrictions: None

Check type identifier

This attribute specifies the kind of check to perform. This makes it possible to support different kinds of checks of the same data type for a given target component without any ambiguity. Boolean check nodes in particular are often used to check different kinds of states like 'enabled', 'editable' or 'selected' (like described in [section 30.9^{\(469\)}](#)). With the help of a `Checker` additional check types can be implemented as shown in [section 39.3^{\(715\)}](#).

Variable: Yes

Restrictions: Must not be empty

Timeout

Time limit in milliseconds until the check must succeed. To disable waiting, leave this value empty or set it to 0.

Variable: Yes

Restrictions: Must not be negative.

Variable for result

This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

Note

If this attribute is set, the attribute `Error level` of message is ignored and no error is reported. The attribute `Throw exception on failure` always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure⁽³⁷⁸⁾, Dependency⁽³⁴⁴⁾ or Test-case⁽³¹⁴⁾ node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test-case node or, if there is no such node, in the global bindings. See chapter 7⁽⁵⁷⁾ for a detailed explanation of variable binding and lookup.

Variable: No

Restrictions: None

Error level of message

This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

Note

If the attribute Throw exception on failure is set, this attribute is irrelevant and if Variable for result is set this attribute is ignored.

Variable: No

Restrictions: None

Throw exception on failure

Throw an exception in case of failure. For 'Check...' nodes a CheckFailedException⁽⁵⁶¹⁾ is thrown, for 'Wait for...' nodes the respective specific exception.

Variable: No

Restrictions: None

Name

An optional name for the Check, mostly useful for the report.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.9.3 Check items



Validates multiple displayed text strings in a component or sub-item.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The text strings are sent to the SUT together with the data about the target component. The `TestEventQueue` determines the corresponding component in the SUT, reads its displayed values and compares them to the required values.

Attributes:

Check items

Client

SUT

QF-Test component ID

tabNames.FirstName

Items

	Text	Regexp
0	Greg	☐
1	James	☐
2	Tina	☐

Check type identifier

default

Timeout

Result handling

Variable for result

☐ Local variable

Error level of message

Error

☒ Throw exception on failure

Name

QF-Test ID

Delay before (ms)

Delay after (ms)

Comment

Check column "First name" including sort order

Figure 30.56: Check items attributes

Client

The name of the SUT client process to which the check is sent.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID⁽⁵³⁶⁾ of the Window⁽⁵³⁵⁾, Component⁽⁵⁴⁵⁾ or Item⁽⁵⁵¹⁾ node that is the target of the check.



The button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing **Shift-Return** or **Alt-Return**, when the focus is in the text field. As an alternative you can copy the target node with **Ctrl-C** or **Edit→Copy** and insert its QF-Test component ID into the text field by pressing **Ctrl-V**.

By using a special syntax you can target Items⁽⁵⁵¹⁾ directly or access components from another suite (see section 6.3⁽⁵⁴⁾ and section 17.1⁽¹⁵⁰⁾ for details).

Variable: Yes

Restrictions: Must not be empty.

Items

This table holds the values to which the text strings displayed by the component or sub-item are compared. Each row represents one sub-item of the target component. The "Text" column holds the actual value while the "Regexp" column determines if this value is treated as a plain string or as a regexp (see section 36.4⁽⁶¹⁴⁾).

See section 2.2.5⁽¹³⁾ about how to work with tables in QF-Test.

You can select **Escape text for regular expressions** from the context menu for escaping special characters of regular expressions of that cell text.

Variable: Yes for the "Text" column, otherwise no.

Restrictions: Valid regexp if required.

Check type identifier

This attribute specifies the kind of check to perform. This makes it possible to support different kinds of checks of the same data type for a given target component without any ambiguity. With the help of a `Checker` additional check types can be implemented as shown in section 39.3⁽⁷¹⁵⁾.

Variable: Yes

Restrictions: Must not be empty

Timeout

Time limit in milliseconds until the check must succeed. To disable waiting, leave this value empty or set it to 0.

Variable: Yes

Restrictions: Must not be negative.

Variable for result

This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

Note

If this attribute is set, the attribute Error level of message is ignored and no error is reported. The attribute Throw exception on failure always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure⁽³⁷⁸⁾, Dependency⁽³⁴⁴⁾ or Test-case⁽³¹⁴⁾ node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test-case node or, if there is no such node, in the global bindings. See chapter 7⁽⁵⁷⁾ for a detailed explanation of variable binding and lookup.

Variable: No

Restrictions: None

Error level of message

This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

Note

If the attribute Throw exception on failure is set, this attribute is irrelevant and if Variable for result is set this attribute is ignored.

Variable: No

Restrictions: None

Throw exception on failure

Throw an exception in case of failure. For 'Check...' nodes a CheckFailedException⁽⁵⁶¹⁾ is thrown, for 'Wait for...' nodes the respective specific exception.

Variable: No

Restrictions: None

Name

An optional name for the Check, mostly useful for the report.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.9.4 Check selectable items

Validates multiple displayed text strings in a component or sub-item and additionally checks their selection state.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The text strings and selection state data are sent to the SUT together with the data about the target component. The `TestEventQueue` determines the corresponding component in the SUT, reads its displayed values, compares them to the required values and checks the required selection state.

Attributes:

Check selectable items		
Client		
SUT		
QF-Test component ID		
tabNames.FirstName		
Items		
Text	Regexp	Selected
Greg	☐	☐
James	☐	☒
Tina	☐	☐
Check type identifier		
default		
Timeout		
Result handling		
Variable for result		
☐ Local variable		
Error level of message		
Error		
☐ Throw exception on failure		
Name		
QF-Test ID		
Delay before (ms)	Delay after (ms)	
Comment		
Check column "First name" including sort order, second row must be selected		

Figure 30.57: Check selectable items attributes

Client

The name of the SUT client process to which the check is sent.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID⁽⁵³⁶⁾ of the Window⁽⁵³⁵⁾, Component⁽⁵⁴⁵⁾ or Item⁽⁵⁵¹⁾ node that is the target of the check.



The  button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing **[Shift-Return]** or **[Alt-Return]**, when the focus is in the text field. As an alternative you can copy the target node with **[Ctrl-C]** or **[Edit→Copy]** and insert its QF-Test component ID into the text field by pressing **[Ctrl-V]**.

By using a special syntax you can target Items⁽⁵⁵¹⁾ directly or access components from another suite (see section 6.3⁽⁵⁴⁾ and section 17.1⁽¹⁵⁰⁾ for details).

Variable: Yes

Restrictions: Must not be empty.

Items

This table holds the values to which the text strings displayed by the component or sub-item are compared. Each row represents one sub-item of the target component. The "Text" column holds the actual value while the "Regexp" column determines if this value is treated as a plain string or as a regexp (see section 36.4⁽⁶¹⁴⁾). The "Selected" column holds the required selection state.

See section 2.2.5⁽¹³⁾ about how to work with tables in QF-Test.

You can select **[Escape text for regular expressions]** from the context menu for escaping special characters of regular expressions of that cell text.

Variable: Yes for the "Text" column, otherwise no.

Restrictions: Valid regexp if required.

Check type identifier

This attribute specifies the kind of check to perform. This makes it possible to support different kinds of checks of the same data type for a given target component without any ambiguity. With the help of a `Checker` additional check types can be implemented as shown in section 39.3⁽⁷¹⁵⁾.

Variable: Yes

Restrictions: Must not be empty

Timeout

Time limit in milliseconds until the check must succeed. To disable waiting, leave this value empty or set it to 0.

Variable: Yes

Restrictions: Must not be negative.

Variable for result

This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

Note

If this attribute is set, the attribute Error level of message is ignored and no error is reported. The attribute Throw exception on failure always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure⁽³⁷⁸⁾, Dependency⁽³⁴⁴⁾ or Test-case⁽³¹⁴⁾ node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test-case node or, if there is no such node, in the global bindings. See chapter 7⁽⁵⁷⁾ for a detailed explanation of variable binding and lookup.

Variable: No

Restrictions: None

Error level of message

This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

Note

If the attribute Throw exception on failure is set, this attribute is irrelevant and if Variable for result is set this attribute is ignored.

Variable: No

Restrictions: None

Throw exception on failure

Throw an exception in case of failure. For 'Check...' nodes a

`CheckFailedException`⁽⁵⁶¹⁾ is thrown, for 'Wait for...' nodes the respective specific exception.

Variable: No

Restrictions: None

Name

An optional name for the Check, mostly useful for the report.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.9.5 Check image



Checks the displayed image of a component. This check is supported for all kinds of components and for sub-items.

It is possible to check only parts of an image. To that end, a region within the image can be defined, either by dragging a rectangle or by editing the region attributes. Also, if the check image is smaller than the component, an offset into the component can be given. When recording only the visible part of a component or when cropping the check image to size after defining a region, the offset is determined automatically.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The image data is sent to the SUT together with the data about the target component. The `TestEventQueue` determines the corresponding component in the SUT and compares its current image to the required value.

Attributes:

Check image	
Client	
SUT	
 QF-Test component ID	
tb_debugger.singlestep	
Position of image relative to component	
X offset	Y offset
 Image (34x34) [1:1]	
	
Actual check region inside image	
X	Y
Width	Height
Algorithm for image comparison	
☒  ☐ Negate	
Check type identifier	
default	
Timeout	
Result handling	
Variable for result	
☐ Local variable	
Error level of message	
Error	
☒  ☐ Throw exception on failure	
Name	
QF-Test ID	
Delay before (ms)	Delay after (ms)
 Comment	

Figure 30.58: Check image attributes

Client

The name of the SUT client process to which the check is sent.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID⁽⁵³⁶⁾ of the Window⁽⁵³⁵⁾, Component⁽⁵⁴⁵⁾ or Item⁽⁵⁵¹⁾ node that is the target of the check.



The  button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing **Shift-Return** or **Alt-Return**, when the focus is in the text field. As an alternative you can copy the target node with **Ctrl-C** or **Edit→Copy** and insert its QF-Test component ID into the text field by pressing **Ctrl-V**.

By using a special syntax you can target Items⁽⁵⁵¹⁾ directly or access components from another suite (see section 6.3⁽⁵⁴⁾ and section 17.1⁽¹⁵⁰⁾ for details).

Variable: Yes

Restrictions: Must not be empty.

'Position of image relative to component'

If the image is smaller than the component, these offset coordinates together with the size of the image determine the sub-region of the component that is checked.

Variable: Yes

Restrictions: Must not be negative.

Image

The recorded image of the component. It can be displayed at different zoom levels, saved to or loaded from disk in the PNG format, or edited in an external imaging tool. The tool to use must be defined with the option External imaging program⁽²⁴⁸⁾.

Variable: Yes

Restrictions: Must not be negative.

'Actual check region inside image'

If only some part of the displayed image of a component needs to be checked, a region within the image can be defined that limits the comparison to that area.

Variable: Yes

Restrictions: Must not be negative.

'Algorithm for image comparison'

This attribute defines a special algorithm for image comparison. See [Details about the algorithm for image comparison^{\(769\)}](#) for further information.

Variable: Yes

Restrictions: Must match a special syntax.

Check type identifier

This attribute specifies the kind of check to perform. This makes it possible to support different kinds of checks of the same data type for a given target component without any ambiguity. With the help of a `Checker` additional check types can be implemented as shown in [section 39.3^{\(715\)}](#).

Variable: Yes

Restrictions: Must not be empty

Negate

This flag determines whether to execute a positive or a negative check. If it is set, a negative check will be performed, i.e. the checked property must not match the expected value.

Variable: Yes

Restrictions: None

Timeout

Time limit in milliseconds until the check must succeed. To disable waiting, leave this value empty or set it to 0.

Variable: Yes

Restrictions: Must not be negative.

Variable for result

This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

If this attribute is set, the attribute `Error level of message` is ignored and no error is reported. The attribute `Throw exception on failure` always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

Note

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure⁽³⁷⁸⁾, Dependency⁽³⁴⁴⁾ or Test-case⁽³¹⁴⁾ node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test-case node or, if there is no such node, in the global bindings. See chapter 7⁽⁵⁷⁾ for a detailed explanation of variable binding and lookup.

Variable: No

Restrictions: None

Error level of message

This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

Note

If the attribute Throw exception on failure is set, this attribute is irrelevant and if Variable for result is set this attribute is ignored.

Variable: No

Restrictions: None

Throw exception on failure

Throw an exception in case of failure. For 'Check...' nodes a CheckFailedException⁽⁵⁶¹⁾ is thrown, for 'Wait for...' nodes the respective specific exception.

Variable: No

Restrictions: None

Name

An optional name for the Check, mostly useful for the report.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.9.6 Check geometry



Checks the location and size of a component. This check is supported for all kinds of components but not for sub-items.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The geometry data is sent to the SUT together with the data about the target component. The `TestEventQueue` determines the corresponding component in the SUT and compares location and size to the required values.

Attributes:

Check geometry	
Client	
SUT	
QF-Test component ID	
winMain	
Geometry	
X	Y
200	300
Width	Height
600	400
☒ Negate	
Check type identifier	
default	
Timeout	
Result handling	
Variable for result	
☐ Local variable	
Error level of message	
Error	
☒ Throw exception on failure	
Name	
QF-Test ID	
Delay before (ms)	Delay after (ms)
☒ Comment	
Check the geometry of the main window	

Figure 30.59: Check geometry attributes

Client

The name of the SUT client process to which the check is sent.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID⁽⁵³⁶⁾ of the Window⁽⁵³⁵⁾, Component⁽⁵⁴⁵⁾ or Item⁽⁵⁵¹⁾ node that is the target of the check.



The  button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing **[Shift-Return]** or **[Alt-Return]**, when the focus is in the text field. As an alternative you can copy the target node with **[Ctrl-C]** or **[Edit→Copy]** and insert its QF-Test component ID into the text field by pressing **[Ctrl-V]**.

By using a special syntax you can target Items⁽⁵⁵¹⁾ directly or access components from another suite (see section 6.3⁽⁵⁴⁾ and section 17.1⁽¹⁵⁰⁾ for details).

Variable: Yes

Restrictions: Must not be empty.

Geometry

The X/Y coordinates, width and height to which the location and size of the component are compared. To check only some of these, e.g. just the location or just the size, leave the others empty.

Variable: Yes

Restrictions: Valid number, width and height > 0

Check type identifier

This attribute specifies the kind of check to perform. This makes it possible to support different kinds of checks of the same data type for a given target component without any ambiguity. With the help of a `Checker` additional check types can be implemented as shown in section 39.3⁽⁷¹⁵⁾.

Variable: Yes

Restrictions: Must not be empty

Negate

This flag determines whether to execute a positive or a negative check. If it is set, a negative check will be performed, i.e. the checked property must not match the expected value.

Variable: Yes

Restrictions: None

Timeout

Time limit in milliseconds until the check must succeed. To disable waiting, leave this value empty or set it to 0.

Variable: Yes

Restrictions: Must not be negative.

Variable for result

This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

Note

If this attribute is set, the attribute Error level of message is ignored and no error is reported. The attribute Throw exception on failure always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure⁽³⁷⁸⁾, Dependency⁽³⁴⁴⁾ or Test-case⁽³¹⁴⁾ node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test-case node or, if there is no such node, in the global bindings. See chapter 7⁽⁵⁷⁾ for a detailed explanation of variable binding and lookup.

Variable: No

Restrictions: None

Error level of message

This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

Note

If the attribute Throw exception on failure is set, this attribute is irrelevant and if Variable for result is set this attribute is ignored.

Variable: No

Restrictions: None

Throw exception on failure

Throw an exception in case of failure. For 'Check...' nodes a

`CheckFailedException`⁽⁵⁶¹⁾ is thrown, for 'Wait for...' nodes the respective specific exception.

Variable: No

Restrictions: None

Name

An optional name for the Check, mostly useful for the report.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.10 Queries

Automating a GUI test gets difficult whenever the SUT shows dynamic or unpredictable behavior, i.e. when it displays values that change with every run of the program. This is

Note

typically the case for fields that are filled automatically with things like the current time, an automatic ID from a database, etc.

QF-Test addresses this issue with means to read values from the SUT's components or to determine the numerical index of a sub-item when given its name. These values are stored in variables to be used again later as the test proceeds.

30.10.1 Fetch text



This node lets you read a value from the SUT during the execution of a test run. The text is assigned to the local or global variable (see [chapter 7^{\(57\)}](#)) named by the Variable name⁽⁵⁰¹⁾ attribute.

Not all components display text and some complex components contain multiple textual items, so this operation is only supported for certain components or sub-items. If you try to fetch the text from the wrong component, an OperationNotSupportedException⁽⁵⁶¹⁾ is thrown, while an unsupported sub-item leads to an UnexpectedIndexException⁽⁵⁶²⁾. The following table lists the supported component and sub-item targets for this operation. (P/S) means primary/secondary index.

Web In web applications every node could contain some text, so QF-Test returns either the text or an empty value, but never throws an OperationNotSupportedException.

Class	Index (P/S)	Result
AbstractButton	-/-	getText ()
Dialog	-/-	getTitle ()
Frame	-/-	getTitle ()
ComboBox	-/-	Current value (use renderer)
ComboBox	List item/-	List item (use renderer)
JEditorPane	Character index/-	Structural element at index (experimental)
Label	-/-	getText ()
List	Item/-	Item (use renderer)
TabbedPane	Tab/-	Title of tab
Table	Column/Row	Cell contents (use renderer)
JTableHeader	Column/-	Column title (use renderer)
TextArea	Line number/-	Line of text
JTextComponent	-/-	getText ()
Tree	Node/-	Node (use renderer)
Label	-/-	getText ()
TextField	-/-	getText ()

Table 30.21: Components supported by Fetch text

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The data of the target component is sent to the SUT. The `TestEventQueue` determines the corresponding component, retrieves the requested value and sends it back to QF-Test, where it is stored in a global variable.

Attributes:

The screenshot shows a 'Fetch text' dialog box. It has a title bar 'Fetch text'. Below the title bar, there are several sections: 'Client' with a text field containing 'SUT'; 'QF-Test component ID' with a text field containing 'tFirstName'; 'Variable name' with a text field containing 'firstname'; a checkbox labeled 'Local variable' which is unchecked; 'QF-Test ID' with an empty text field; 'Delay before (ms)' and 'Delay after (ms)' with empty text fields; and a 'Comment' section with a text area containing the text 'Set the value of the variable "firstname" to the contents of the "First name" text field'.

Figure 30.60: Fetch text attributes

Client

The name of the SUT client process from which to query the data.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID⁽⁵³⁶⁾ of the Window⁽⁵³⁵⁾, Component⁽⁵⁴⁵⁾ or Item⁽⁵⁵¹⁾ node that is to be queried.



The  button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing **Shift-Return** or **Alt-Return**,

when the focus is in the text field. As an alternative you can copy the target node with **Ctrl-C** or **Edit→Copy** and insert its QF-Test component ID into the text field by pressing **Ctrl-V**.

By using a special syntax you can target Items⁽⁵⁵¹⁾ directly or access components from another suite (see section 6.3⁽⁵⁴⁾ and section 17.1⁽¹⁵⁰⁾ for details).

Variable: Yes

Restrictions: Must not be empty.

Variable name

The name of the global variable to which the result of the query is assigned (see chapter 7⁽⁵⁷⁾).

Variable: Yes

Restrictions: Must not be empty.

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure⁽³⁷⁸⁾, Dependency⁽³⁴⁴⁾ or Test-case⁽³¹⁴⁾ node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test-case node or, if there is no such node, in the global bindings. See chapter 7⁽⁵⁷⁾ for a detailed explanation of variable binding and lookup.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.10.2 Fetch index

With the help of this node you can determine the index of a sub-item during the execution of a test, provided its displayed text is known. Obviously only an Item⁽⁵⁵¹⁾ is supported as the target component. The result is assigned to the local or global variable (see chapter 7⁽⁵⁷⁾) named by the Variable name⁽⁵⁰³⁾ attribute.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The data of the target component is sent to the SUT. The `TestEventQueue` determines the corresponding component, searches for the requested sub-item and sends its index back to QF-Test, where it is stored in a global variable.

Attributes:

Figure 30.61: Fetch index attributes

Client

The name of the SUT client process from which to query the data.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID⁽⁵³⁶⁾ of the Window⁽⁵³⁵⁾, Component⁽⁵⁴⁵⁾ or Item⁽⁵⁵¹⁾ node that is to be queried.



The  button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing **Shift-Return** or **Alt-Return**, when the focus is in the text field. As an alternative you can copy the target node with **Ctrl-C** or **Edit→Copy** and insert its QF-Test component ID into the text field by pressing **Ctrl-V**.

By using a special syntax you can target Items⁽⁵⁵¹⁾ directly or access components from another suite (see section 6.3⁽⁵⁴⁾ and section 17.1⁽¹⁵⁰⁾ for details).

Variable: Yes

Restrictions: Must not be empty.

Variable name

The name of the global variable to which the result of the query is assigned (see [chapter 7^{\(57\)}](#)).

Variable: Yes

Restrictions: Must not be empty.

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing [Procedure^{\(378\)}](#), [Dependency^{\(344\)}](#) or [Test-case^{\(314\)}](#) node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test-case node or, if there is no such node, in the global bindings. See [chapter 7^{\(57\)}](#) for a detailed explanation of variable binding and lookup.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the [Default delay^{\(288\)}](#) from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option [External editor command^{\(247\)}](#) lets you define an external editor in which comments can be edited conveniently by

pressing **Alt-Return** or by clicking the  button.

Note

Variable: Yes

Restrictions: None

30.10.3 Fetch geometry



Use this node to find out the geometry of a window, component or sub-item in the SUT. The result is stored in up to four local or global variables, one each for the X and Y coordinates, width and height.

This node is useful if you want to set the $X^{(448)}$ and $Y^{(448)}$ coordinates of a Mouse event⁽⁴⁴⁷⁾ relative to the right or bottom border of the target component. Simply fetch the component's width and height and define the coordinates using the extended variable syntax for expressions (see section 7.6⁽⁶⁶⁾).

The following table lists the supported sub-item targets for this operation. (P/S) means primary/secondary index.

Class	Index (P/S)	Result
JList	Item/-	Item
JTabbedPane	Tab/-	Tab
JTable	Column/-	Column
JTable	Column/Row	Cell
JTableHeader	Column/-	Column title
JTree	Node/-	Node

Table 30.22: Components supported by Fetch geometry

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The data of the target component is sent to the SUT. The `TestEventQueue` determines the corresponding component, retrieves its geometry and sends it back to QF-Test, where the values are stored in global variables.

Attributes:

Figure 30.62: Fetch geometry attributes

Client

The name of the SUT client process from which to query the data.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID⁽⁵³⁶⁾ of the Window⁽⁵³⁵⁾, Component⁽⁵⁴⁵⁾ or Item⁽⁵⁵¹⁾ node that is to be queried.



The  button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing **[Shift-Return]** or **[Alt-Return]**, when the focus is in the text field. As an alternative you can copy the target node with **[Ctrl-C]** or **[Edit→Copy]** and insert its QF-Test component ID into the text field by pressing **[Ctrl-V]**.

By using a special syntax you can target Items⁽⁵⁵¹⁾ directly or access components from another suite (see section 6.3⁽⁵⁴⁾ and section 17.1⁽¹⁵⁰⁾ for details).

Variable: Yes

Restrictions: Must not be empty.

Location relative to window

This attribute determines whether the X and Y coordinate of a component or sub-item is calculated relative to its parent component or relative to its parent window. For windows there is no difference.

Variable: No

Restrictions: None

Variable for x

The name of the global variable to which the X coordinate of the window, component or sub-item is assigned (see [chapter 7^{\(57\)}](#)). If you are not interested in the X coordinate, leave this value empty.

Variable: Yes

Restrictions: None

Variable for y

The name of the global variable to which the Y coordinate of the window, component or sub-item is assigned (see [chapter 7^{\(57\)}](#)). If you are not interested in the Y coordinate, leave this value empty.

Variable: Yes

Restrictions: None

Variable for width

The name of the global variable to which the width of the window, component or sub-item is assigned (see [chapter 7^{\(57\)}](#)). If you are not interested in the width, leave this value empty.

Variable: Yes

Restrictions: None

Variable for height

The name of the global variable to which the height of the window, component or sub-item is assigned (see [chapter 7^{\(57\)}](#)). If you are not interested in the height, leave this value empty.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create local or global variable bindings. If unset, the variables are bound in the global variables. If set, the topmost current binding for a variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure⁽³⁷⁸⁾, Dependency⁽³⁴⁴⁾ or Test-case⁽³¹⁴⁾ node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test-case node or, if there is no such node, in the global bindings. See chapter 7⁽⁵⁷⁾ for a detailed explanation of variable binding and lookup.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.11 Miscellaneous

This section lists the remaining nodes that don't fit in well with any of the other sections.

30.11.1 Set variable



This node lets you set the value of a global variable. If the test is run interactively from QF-Test and not in batch mode (see [section 1.6^{\(7\)}](#)) you can optionally set the value interactively.

Contained in: All kinds of [sequences^{\(314\)}](#).

Children: None

Execution: If the test is run interactively and the [Interactive^{\(511\)}](#) attribute is set, a dialog is shown in which the value for the variable can be entered. If the [Timeout^{\(511\)}](#) is exceeded or the value is confirmed with the OK button, the variable is bound accordingly in the global variables. If the dialog is canceled, the test run is stopped. In the non-interactive case the variable is bound directly to the [Default value^{\(511\)}](#).

Attributes:

Figure 30.63: Set variable attributes

Variable name

The name of the global variable to which the value is assigned (see [chapter 7^{\(57\)}](#)).

Variable: Yes

Restrictions: Must not be empty.

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing [Procedure^{\(378\)}](#), [Dependency^{\(344\)}](#) or [Test-case^{\(314\)}](#) node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test-case node or, if there is no such node, in the global bindings. See [chapter 7^{\(57\)}](#) for a detailed explanation of variable binding and lookup.

Variable: No

Restrictions: None

Default value

The default value for the variable if the test is run non-interactively, the Interactive⁽⁵¹¹⁾ attribute is not set or the Timeout⁽⁵¹¹⁾ is exceeded.

Variable: Yes

Restrictions: None

Interactive

Whether a dialog should be shown in which the value for the global variable can be entered. Ignored if the test is run non-interactively.

Variable: Yes

Restrictions: None

Description

A short description to display in the dialog. If you leave this value empty, the description `Value for <Variable name>` will be used.

Variable: Yes

Restrictions: None

Timeout

An optional timeout value for the dialog. If the dialog is shown and the value is left unchanged for the specified period of time, the dialog is closed automatically and the default value is used. This avoids blocking a test that is started interactively from QF-Test and then left to run unattended.

Variable: Yes

Restrictions: Empty or > 0.

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number >= 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.11.2 Wait for component to appear

This node is very important for the timing of a test run. The reaction time of the SUT varies depending on system and memory load, so it may take a while until, say, a complex dialog is opened. This node will delay further execution of the test until the desired component or sub-item is available. If the time limit is exceeded without success, a ComponentNotFoundException⁽⁵⁵⁸⁾ is thrown. You can also use the Variable for result attribute to store the result into a variable and the Throw exception on failure attribute to suppress the exception.

This node is intended only for relatively long delays. Short delays are handled automatically by the Timeout options⁽²⁹⁰⁾.

By setting the Wait for absence⁽⁵¹⁴⁾ attribute this node can also be used to ensure the absence of a component.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The data of the target component are sent to the SUT. The `TestEventQueue` waits until either the corresponding component becomes available or the time limit is exceeded.

Attributes:

Figure 30.64: Wait for component to appear attributes

Client

The name of the Java process in which to wait.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID⁽⁵³⁶⁾ of the Window⁽⁵³⁵⁾, Component⁽⁵⁴⁵⁾ or Item⁽⁵⁵¹⁾ to wait for.



The button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing **[Shift-Return]** or **[Alt-Return]**, when the focus is in the text field. As an alternative you can copy the target node

with **Ctrl-C** or **Edit→Copy** and insert its QF-Test component ID into the text field by pressing **Ctrl-V**.

By using a special syntax you can target `Items(551)` directly or access components from another suite (see [section 6.3^{\(54\)}](#) and [section 17.1^{\(150\)}](#) for details).

Variable: Yes

Restrictions: Must not be empty.

Timeout

Time limit in milliseconds.

Variable: Yes

Restrictions: ≥ 0

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Wait for absence

If this attribute is set, QF-Test waits for the absence of a component. This is useful e.g. to ensure that a dialog has closed or was never opened in the first place. If the component does exist, a `ComponentFoundException(558)` is thrown.

It is possible to check for the absence of a sub-item. In this case, the complex component that holds the item must be present, otherwise a `ComponentNotFoundException(558)` is thrown.

Variable: Yes

Restrictions: None

Variable for result

This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

Note

If this attribute is set, the attribute Error level of message is ignored and no error is reported. The attribute Throw exception on failure always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure⁽³⁷⁸⁾, Dependency⁽³⁴⁴⁾ or Test-case⁽³¹⁴⁾ node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test-case node or, if there is no such node, in the global bindings. See chapter 7⁽⁵⁷⁾ for a detailed explanation of variable binding and lookup.

Variable: No

Restrictions: None

Error level of message

This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

If the attribute Throw exception on failure is set, this attribute is irrelevant and if Variable for result is set this attribute is ignored.

Variable: No

Restrictions: None

Throw exception on failure

Throw an exception in case of failure. For 'Check...' nodes a CheckFailedException⁽⁵⁶¹⁾ is thrown, for 'Wait for...' nodes the respective specific exception.

Variable: No

Restrictions: None

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets

Note

you define an external editor in which comments can be edited conveniently by

pressing `[Alt-Return]` or by clicking the  button.

Variable: Yes

Restrictions: None

30.11.3 Wait for document to load



This node is a variant of the Wait for component to appear⁽⁵¹²⁾ node specifically for web pages. It not only checks the existence of a given document. If the target document was already known to exist the last time an event was replayed, this node waits for the document to get reloaded. When working with web pages it is often the case that the same or very similar documents are loaded many times. Without this node's functionality QF-Test could not discern the case where the old document is still around from the one where the document is already reloaded. In the former case, replaying an event could cause it to have no effect at all because at the same time reloading of the document begins.

The Name of the browser window attribute can be used to limit the search to a given browser window or to define a name for a new window. If Stop loading if timeout exceeded is set, QF-Test will abort loading the document if it doesn't finish in time. You can also use the Variable for result attribute to store the result in a variable and the Throw exception on failure attribute to suppress the DocumentNotLoadedException⁽⁵⁵⁸⁾.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The data of the target document are sent to the SUT. The `TestEventQueue` waits until corresponding document gets loaded or the time limit is exceeded in which case a `DocumentNotLoadedException` is thrown.

Attributes:

Web

Wait for document to load	
Client	SUT
 QF-Test component ID	www.qfs.de
Name of the browser window	
Timeout	15000
 ☐ Stop loading if timeout exceeded	
Result handling	
Variable for result	
☐ Local variable	
Error level of message	Error
 ☒ Throw exception on failure	
QF-Test ID	
Delay before (ms)	Delay after (ms)
 Comment	
Wait at most fifteen seconds for loading of the web page www.qfs.de to complete.	

Figure 30.65: Wait for document to load attributes

Client

The name of the SUT client process from which to query the data.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The QF-Test ID⁽⁵³⁶⁾ of the Window⁽⁵³⁵⁾, Component⁽⁵⁴⁵⁾ or Item⁽⁵⁵¹⁾ node that is to be queried.



The  button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing **Shift-Return** or **Alt-Return**, when the focus is in the text field. As an alternative you can copy the target node with **Ctrl-C** or **Edit→Copy** and insert its QF-Test component ID into the text field by pressing **Ctrl-V**.

By using a special syntax you can target Items⁽⁵⁵¹⁾ directly or access components from another suite (see section 6.3⁽⁵⁴⁾ and section 17.1⁽¹⁵⁰⁾ for details).

Variable: Yes

Restrictions: Must not be empty.

Name of the browser window

This attribute has a dual use. If set to an existing name for a browser window, QF-Test waits for the document to load in that window. If the name is set but no browser window by that name exists, the search is limited to documents in new or not-yet-named windows. If the wait succeeds and a new document is loaded, the window name is assigned to the document's browser window. This is the only way to define a name for a popup window. Explicitly launched browsers can have their name set via the Name of the browser window⁽⁴³¹⁾ attribute of a Start browser⁽⁴²⁸⁾ node.

Variable: Yes

Restrictions: None

Timeout

Time limit in milliseconds.

Variable: Yes

Restrictions: ≥ 0

Stop loading if timeout exceeded

If this attribute is set and the timeout is exceeded without a matching document finishing to load, QF-Test will cause loading to stop, either in all browsers or, if Name of the browser window is set, the browser window specified therein. Result and exception handling are not affected by this attribute. If the timeout is exceeded the operation is considered a failure regardless of whether loading is stopped or not.

Variable: Yes

Restrictions: None

Variable for result

This optional attribute determines the name for the result variable of the action. If

set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

Note

If this attribute is set, the attribute Error level of message is ignored and no error is reported. The attribute Throw exception on failure always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure⁽³⁷⁸⁾, Dependency⁽³⁴⁴⁾ or Test-case⁽³¹⁴⁾ node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test-case node or, if there is no such node, in the global bindings. See chapter 7⁽⁵⁷⁾ for a detailed explanation of variable binding and lookup.

Variable: No

Restrictions: None

Error level of message

This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

Note

If the attribute Throw exception on failure is set, this attribute is irrelevant and if Variable for result is set this attribute is ignored.

Variable: No

Restrictions: None

Throw exception on failure

Throw an exception in case of failure. For 'Check...' nodes a CheckFailedException⁽⁵⁶¹⁾ is thrown, for 'Wait for...' nodes the respective specific exception.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.11.4 Wait for download to finish

Web



This specialized node is applicable only for web Clients. It can be used to wait for the completion of a download that was earlier started via QF-Test. This is important in case you need to verify the contents of the file or to measure the time it took to download it.

If the timeout is exceeded without the download finishing, a `DownloadNotCompleteException` is thrown unless suppressed via the `Throw exception on failure` attribute. Either way the download can be canceled by activating the `Cancel download if timeout exceeded` attribute, which may be necessary to enable another download to the same file. The result can also be stored in a variable by defining its name in the `Variable for result` attribute.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The target file identifying the download is sent to the SUT where QF-Test waits for the download to finish or the time limit is exceeded in which case a `DownloadNotCompleteException` is thrown.

Attributes:

Wait for download to finish

Client
SUT

File
/tmp/test.dat

Timeout
60000

☐ Cancel download if timeout exceeded

Result handling

Variable for result

☐ Local variable

Error level of message
Error

☒ Throw exception on failure

QF-Test ID

Delay before (ms) Delay after (ms)

Comment
Wait at most sixty seconds for the download of the file /tmp/test.dat to finish.

Figure 30.66: Wait for download to finish attributes

Client

The name of the SUT client process from which to query the data.

Variable: Yes

Restrictions: Must not be empty.

File

The target file for the downloaded.

Variable: Yes

Restrictions: Valid file name

Timeout

Time limit in milliseconds.

Variable: Yes

Restrictions: ≥ 0

Cancel download if timeout exceeded

If this attribute is set and the timeout is exceeded without the download finishing the download is canceled.

Variable: Yes

Restrictions: None

Variable for result

This optional attribute determines the name for the result variable of the action. If set, the respective variable will be set to 'true' for a successful check or wait and to 'false' in case of failure.

Note

If this attribute is set, the attribute Error level of message is ignored and no error is reported. The attribute Throw exception on failure always remains effective, so it is possible to set a result variable and still throw an exception.

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create a local or global variable binding. If unset, the variable is bound in the global variables. If set, the topmost current binding for the variable is replaced with the new value, provided this binding is within the context of the currently executing Procedure⁽³⁷⁸⁾, Dependency⁽³⁴⁴⁾ or Test-case⁽³¹⁴⁾ node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test-case node or, if there is no such node, in the global bindings. See chapter 7⁽⁵⁷⁾ for a detailed explanation of variable binding and lookup.

Variable: No

Restrictions: None

Error level of message

This attribute determines the error level of the message that is logged in case of failure. Possible choices are message, warning and error.

Note

If the attribute Throw exception on failure is set, this attribute is irrelevant and if Variable for result is set this attribute is ignored.

Variable: No

Restrictions: None

Throw exception on failure

Throw an exception in case of failure. For 'Check...' nodes a `CheckFailedException`⁽⁵⁶¹⁾ is thrown, for 'Wait for...' nodes the respective specific exception.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.11.5 Load resources



This node is used to load a `ResourceBundle` and make its values available for the extended variable syntax `${group:name}` (see [section 7.5^{\(62\)}](#)). To learn more about `ResourceBundles` see the description of the `ResourceBundle(524)` attribute.

Contained in: All kinds of `sequences(314)`.

Children: None

Execution: The `ResourceBundle` is loaded and its values are made available under the `Group(524)` name.

Attributes:

Load resources	
Group	
msg	
ResourceBundle	
rsc.messages	
Locale	
en_US	
QF-Test ID	
Delay before (ms)	Delay after (ms)
Comment	
Load American message strings for \${msg:...}	

Figure 30.67: Load resources attributes

Group

The name of the group by which values of the `ResourceBundle` are referred to. The value of a definition of the form `name=value` in the `ResourceBundle` can be retrieved with `${group:name}` (see [section 7.5^{\(62\)}](#)).

Variable: Yes

Restrictions: Must not be empty and should not contain special characters like `'` or `$`.

ResourceBundle

The name of the `ResourceBundle` to load. A little Java background is needed to understand this attribute.

The resources are read with the help of the Java method `ResourceBundle.getBundle()`. For this to work, a matching file with the extension `.class` or `.properties` must be located somewhere on the `CLASSPATH`. Use the fully qualified name for the file, including packages, with a dot (`.`) as separator, but without extension or locale identifier.

Example: QF-Test comes with a German `ResourceBundle` in the file `de/qfs/apps/qftest/resources/properties/qftest_de.properties`, which is contained in the archive `qfshared.jar`. To load that `ResourceBundle`, set this attribute to `de.qfs.apps.qftest.resources.properties.qftest` and the Locale⁽⁵²⁵⁾ to `de`.

Variable: Yes

Restrictions: Must name a `ResourceBundle` on the `CLASSPATH`.

Locale

The main use of `ResourceBundles` is to provide data in different languages. This attribute determines, which version of a `ResourceBundle` is retrieved. The value must follow the ISO standard *language_country_variant*. *Language* is a two letter lowercase code like `en` for English, *country* a two letter uppercase code like `US` for American or `UK` for British English. The *variant* discriminates further but is rarely used.

As mentioned, QF-Test relies on the Java method `ResourceBundle.getBundle()` to load the `ResourceBundle`, which is described in detail in the Java documentation and works as follows:

To load a `ResourceBundle` named `res` for the locale `en_US`, Java first searches the `CLASSPATH` for a file named `res_en_US.class` or `res_en_US.properties`, then for `res_en.class` or `res_en.properties` and finally for `res.class` and `res.properties`. The less specific files are loaded even if more specific files are found, but only values not defined in the more specific files are used. That way you can define all English resources in `res_en.properties` and place only those that differ in `res_en_UK.properties` and `res_en_US.properties`.

Unfortunately Java has a "feature" that can lead to surprising results. If no specific file but only the base file `res.properties` is found, Java tries the whole process a second time, this time for the current default locale of the VM. As a result, if the current locale for QF-Test is German and you want to load English resources that are defined in `res.properties` and no `res_en.properties` exists, Java will load the German version from `res_de.properties`, even if you request the

locale `en`. You can work around this by setting this attribute to the underscore `'_'`. In that case, only the base file `res.properties` is loaded.

To use the current locale of the VM, leave this value empty.

Variable: Yes

Restrictions: Empty or valid locale identifier.

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters `'\'`, `'#'`, `'$'`, `'@'`, `'&'`, or `'%'` or start with an underscore (`'_'`).

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing `[Alt-Return]` or by clicking the  button.

Variable: Yes

Restrictions: None

30.11.6 Load properties



This node is used to load data from a `Properties` file and make its values available for the extended variable syntax `${group:name}` (see section 7.5⁽⁶²⁾).

`Properties` files are easier to handle than `ResourceBundles` since you request the file directly, but they are less powerful. The format of a `Properties` file is simple: lines

Note

of the form `name=value` with arbitrary whitespace around the '=' character. Complex definitions spanning multiple lines are possible. Please see the Java documentation for details or ask a developer.

Contained in: All kinds of sequences⁽³¹⁴⁾.

Children: None

Execution: The `Properties` file is loaded and its values are made available under the Group⁽⁵²⁷⁾ name.

Attributes:

The screenshot shows a dialog box titled "Load properties". It has several input fields and a comment section. The "Group" field contains the text "label". The "Properties file" field contains the text "../props/labels.properties". The "QF-Test ID" field is empty. The "Delay before (ms)" and "Delay after (ms)" fields are empty. The "Comment" field contains the text "Load properties for \${label:...}" and has a small icon to its left.

Figure 30.68: Load properties attributes

Group

The name of the group by which values of the `Properties` file are referred to. The value of a definition of the form `name=value` in the `Properties` file can be retrieved with `${group:name}` (see section 7.5⁽⁶²⁾).

Variable: Yes

Restrictions: Must not be empty and should not contain special characters like ':' or '\$'.

Properties file

The file to load the `Properties` from. This can either be an absolute path name or a path relative to the directory of the current suite. In either case you should always use '/' as the separator for directories, even under Windows. QF-Test will translate this to the correct value for the current operating system.

The "...” button brings up a dialog in which you can select the file interactively. You can also get to this dialog by pressing `[Shift-Return]` or `[Alt-Return]`, when the focus is in the text field.

Variable: Yes

Restrictions: Must be an existing `Properties` file.

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters `'\'`, `'#'`, `'$'`, `'@'`, `'&'`, or `'%'` or start with an underscore (`'_'`).

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing `[Alt-Return]` or by clicking the  button.

Variable: Yes

Restrictions: None

30.12 HTTP Requests

This section describes how to send HTTP Request using QFTest.

30.12.1 Server HTTP request

Note

Web



This highly specialized node sends a GET or POST request via HTTP/HTTPS directly to a web server. Such a request can be very helpful for load tests or mass data computing scenarios (e.g. filling out a form) since the simulation of user interactions and the respective loading time of the SUT are omitted during replay. The use of requests is an enhancement of the functionalities for load tests and data-driven testing described in [chapter 21^{\(197\)}](#) and [section 30.4^{\(357\)}](#).

If the status code returned from the server is 400 or higher, a corresponding error message is created in the run-log. A detailed description of the different status codes can be found at <http://www.w3.org/Protocols/HTTP/HTRESP.html>. Additionally you can store the response from the server in a variable and if the attribute `Add server response to run-log` is active the response is also written to the run-log.

Contained in: All kinds of [sequences^{\(314\)}](#).

Children: None

Execution: The GET/POST request is sent directly by QF-Test via HTTP/HTTPS to the specified URL. If the status code returned from the server is not equal to 200 (status=OK) a corresponding error message is created in the run-log.

Attributes:

Server HTTP request

URL
https://www.google.de/search

Method
GET

Parameters

Name	Value
q	qf-test

Variable for server response

Variable for HTTP status code

☐ Local variable

☐ Add server response to run-log

Timeout

Error level if time limit exceeded
Error

QF-Test ID

Delay before (ms)

Delay after (ms)

Comment
Execute HTTP-Request on https://www.google.de/search to search for "qf-test".

Figure 30.69: Server HTTP request Attribute

URL

The URL to which to send the request, not including parameters. HTTP and HTTPS are acceptable values for the protocol.

Variable: Yes

Restrictions: Must not be empty.

Method

This attribute defines the method of the request, GET or POST.

Variable: No

Restrictions: None

Parameters

Here you can specify the parameters for the request. The parameters will be URL encoded by QFTest at execution. See [section 2.2.5^{\(13\)}](#) for further information how to work with the table.

Variable: Yes

Restrictions: None

Variable for server response

The name of the variable to which the server response is assigned (see [chapter 7^{\(57\)}](#)).

Variable: Yes

Restrictions: None

Variable for HTTP status code

The name of the variable to which the HTTP status code is assigned (see [chapter 7^{\(57\)}](#)).

Variable: Yes

Restrictions: None

Local variable

This flag determines whether to create local or global variable bindings. If unset, the variables are bound in the global variables. If set, the topmost current binding for a variable is replaced with the new value, provided this binding is within the context of the currently executing [Procedure^{\(378\)}](#), [Dependency^{\(344\)}](#) or [Test-case^{\(314\)}](#) node. If no such binding exists, a new binding is created in the currently executing Procedure, Dependency or Test-case node or, if there is no such node, in the global bindings. See [chapter 7^{\(57\)}](#) for a detailed explanation of variable binding and lookup.

Variable: No

Restrictions: None

Add server response to run-log

If activated the server response is written to the run-log in addition to the status code.

Variable: Yes

Restrictions: None

Timeout

Time limit in milliseconds until the HTTP Request must succeed. To disable the limit, leave this value empty.

Variable: Yes

Restrictions: Must not be negative.

Error level if time limit exceeded

This attribute determines what happens in case the time limit is exceeded. If set to "exception", a `CheckFailedException`⁽⁵⁶¹⁾ will be thrown. Otherwise a message with the respective error-level will be logged in the run-log.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing `[Alt-Return]` or by clicking the  button.

Variable: Yes

Restrictions: None

30.12.2 Browser HTTP request

Note

Web



This highly specialized node sends a GET or POST request via HTTP/HTTPS directly to a web server. Such a request can be very helpful for load tests or mass data computing scenarios (e.g. filling out a form) since the simulation of user interactions and the respective loading time of the SUT are omitted during replay. The use of requests is an enhancement of the functionalities for load tests and data-driven testing described in [chapter 21^{\(197\)}](#) and [section 30.4^{\(357\)}](#).

Contained in: All kinds of [sequences^{\(314\)}](#).

Children: None

Execution: The GET/POST request is sent within the browser via HTTP/HTTPS to the specified URL. The response is shown directly in the Browser.

Attributes:

Browser HTTP request	
Client	
SUT	
QF-Test component ID	
www.qfs.de	
URL	
http://www.qfs.de/cgi-bin/xapian-omega	
Method	
POST	
Parameters	
Name	Value
P	qftest
DEFAULTOP	and
DB	weben/webde/manualen/manualde/tutorialde
FMT	qfsde
QF-Test ID	
Delay before (ms)	Delay after (ms)
Comment	
Execute HTTP-Request on http://www.qfs.de to search for "qftest".	

Figure 30.70: Browser HTTP request Attribute

Client

The name of the SUT client process in which to execute the request.

Variable: Yes

Restrictions: Must not be empty.

QF-Test component ID

The Web page⁽⁵⁴⁰⁾ in which the request should be submitted.



The  button brings up a dialog in which you can select the component interactively. You can also get to this dialog by pressing **[Shift-Return]** or **[Alt-Return]**, when the focus is in the text field. As an alternative you can copy the target node with **[Ctrl-C]** or **[Edit→Copy]** and insert its QF-Test component ID into the text field by pressing **[Ctrl-V]**.

By using a special syntax you can target Items⁽⁵⁵¹⁾ directly or access components from another suite (see section 6.3⁽⁵⁴⁾ and section 17.1⁽¹⁵⁰⁾ for details).

Variable: Yes

Restrictions: Must not be empty.

URL

The URL to which to send the request, not including parameters. HTTP and HTTPS are acceptable values for the protocol.

Internationalized domain names (IDN) are not supported in the URL attribute as well as links to local file system starting with 'file:///'.

Variable: Yes

Restrictions: Must not be empty.

Method

This attribute defines the method of the request, GET or POST.

Variable: No

Restrictions: None

Parameters

Here you can specify the parameters for the request. The parameters will be URL encoded by QFTest at execution. See section 2.2.5⁽¹³⁾ for further information how to work with the table.

Variable: Yes

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Delay before/after

These attributes cause a delay before or after the execution of the node. If a value is empty, the Default delay⁽²⁸⁸⁾ from the global options is used.

Variable: Yes

Restrictions: Valid number ≥ 0

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.13 Windows, Components and Items

Windows⁽⁵³⁵⁾, Components⁽⁵⁴⁵⁾ and Items⁽⁵⁵¹⁾ are the foundation on which a test-suite is built. They represent the windows and components of the SUT as well as the sub-items of complex components.

Many of the other node types need a window or component as a target, e.g. events⁽⁴⁴⁶⁾ or checks⁽⁴⁶⁹⁾. To that end their QF-Test component ID attribute must be set to the QF-Test ID⁽⁵³⁶⁾ of an existing Window⁽⁵³⁵⁾, Component⁽⁵⁴⁵⁾ or Item⁽⁵⁵¹⁾ node.

All of the windows and components of a test-suite are collected under the Windows and components⁽⁵⁵⁶⁾ node, which is always located at the bottom of the suite.

30.13.1 Window

Note



This node is a surrogate for a window in the SUT. Events⁽⁴⁴⁶⁾, checks⁽⁴⁶⁹⁾ and other nodes refer to it by its QF-Test ID⁽⁵³⁶⁾.

Contained in: Window group⁽⁵⁵⁴⁾, Windows and components⁽⁵⁵⁶⁾.

Children: Component group⁽⁵⁵⁵⁾, Component⁽⁵⁴⁵⁾.

Execution: Cannot be executed.

Attributes:

Window

QF-Test ID
winMain

Class name
javax.swing.JFrame

Name
Main window

Feature
Titel

☐ As regexp

Extra features
State Regexp Negate Name

☐ Modal

Geometry
X Y
300 100
Width Height
300 200

GUI engine
awt

Comment
The main window of the application.

Figure 30.71: Window attributes

QF-Test ID

This ID is the means by which other nodes refer to this window. Therefore it may appear in many places and you should take care to assign an ID with a meaning, i.e. one that is easy to remember and recognize. A QF-Test ID must be unique within the test-suite.

Variable: No

Restrictions: Must not be empty, contain any of the characters '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Class name

The fully qualified name of the Java class of the window or one of its superclasses.

Variable: Yes

Restrictions: Must not be empty.

Name

The name of the window that was set by the developers of the SUT with the `setName(String)` method. See [section 36.2^{\(612\)}](#) for why and how names should be set on all "interesting" components.

Variable: Yes

Restrictions: None

Feature

If there is no [Name^{\(537\)}](#) available for the window, QF-Test tries to recognize it by a characteristic feature. In the case of a *Frame* or a *Dialog* this is the window's title. See [section 36.2^{\(612\)}](#) for more information about component recognition.

You can select Escape text for regular expressions from the context menu for escaping special characters of regular expressions of that text.

Variable: Yes

Restrictions: None

As regexp

If this attribute is set, the [Feature^{\(537\)}](#) is a regular expression (see [section 36.4^{\(614\)}](#)).

Variable: Yes

Restrictions: None

Extra features

Besides the main Feature a Component can have additional features represented as name/value pairs. The kind of component determines which extra features are recorded. Each extra feature can have one of three states:

Ignore (Hint for searching: currently represented as 0)

This extra feature is just for information. It has no influence on component recognition.

Should match (Hint for searching: currently represented as 1)

Target components matching this extra feature have a higher probability for component recognition than those that don't match it.

Must match (Hint for searching: currently represented as 2)

The target component must match this extra feature. Any component not matching it is no candidate for component recognition.

Additional columns allow for matching against a regular expression or to negate the expression, e.g. to define that the "class" attribute of a DOM node in a web page should not be "dummy". The absence of an extra feature can be enforced by adding one with an empty value.

You can select Escape text for regular expressions from the context menu for escaping special characters of regular expressions of the value.

QF-Test automatically assigns some extra features to recorded components.

Name	Engine	Description
columns	Web	Column count in TABLE components.
imagehash	Swing, SWT	Shows the hash value of icons of a button or menuitem.
qfs:class	All	Dedicated component class, e.g. de.qfs.QfsTextField.
qfs:genericclass	All	Generic class of the component, e.g. TextField.
qfs:item	Web	Shows the item index of a DomNode, if it's an item of a complex GUI component. This could get recorded, if some child nodes are additionally recorded as those component might be interesting as well. This could affect content of lists, tables, tabfolders or trees.
qfs:label	All	Shows a matching label for the component, e.g. the text of a button or a label close to the respective component. If no own text or label could be found, it also tries to use tooltips or icon descriptions.
qfs:matchindex	All	Index of components with the same name. Possibly assigned automatically when the <u>Validate component recognition during recording</u> ⁽²⁶³⁾ option is active.
qfs:modal	Web	Shows if a component of the class "Window" is modal.
qfs:originalid	Web	Shows the real 'ID' attribute specified in the DOM for that node.
qfs:systemclass	All	The toolkit-specific system class, e.g. javax.swing.JTextField.
qfs:type	All	The generic type of the component, e.g. TextField:PasswordField.

Table 30.23: Extra features assigned by QF-Test

Variable: Yes**Restrictions:** Names must not be empty

Modal

In case of a *Dialog* this attribute determines whether the dialog is modal.

Variable: Yes

Restrictions: None

Geometry

The X/Y coordinate, width and height of the window form the basis for the recognition⁽⁶¹²⁾ of the window. However, they play a minor roll as long as either a Name⁽⁵³⁷⁾ was provided or a Feature⁽⁵³⁷⁾ is available.

For windows whose location and size vary widely you should clear these attributes.

If no values are specified, the recognition algorithm starts with a perfect geometry match for all candidates. To prevent false positive hits you can disable geometry matching by setting the values to a single '-' character.

Variable: Yes

Restrictions: Width and height must not be negative.

GUI engine

The GUI engine to which the Window and all its Component children belong. QF-Test records `awt` for AWT/Swing and `swt` for SWT. Only really relevant for SUTs with more than one GUI engine as described in chapter 33⁽⁵⁹⁰⁾.

Variable: Yes

Restrictions: See chapter 33⁽⁵⁹⁰⁾

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.13.2 Web page

Web



A Web page is a variant of a Window⁽⁵³⁵⁾ node specifically used for testing web applications. It represents the top-level document in a Browser. Nested documents in FRAME nodes are represented as Components⁽⁵⁴⁵⁾.

In contrast to a Window a Web page has no Class name, Modal, geometry or GUI engine attributes as these are either implicitly defined or redundant.

Contained in: Window group⁽⁵⁵⁴⁾, Windows and components⁽⁵⁵⁶⁾.

Children: Component group⁽⁵⁵⁵⁾, Component⁽⁵⁴⁵⁾.

Execution: Cannot be executed.

Attributes:

Figure 30.72: Web page attributes

QF-Test ID

This ID is the means by which other nodes refer to this page. Therefore it may appear in many places and you should take care to assign an ID with a meaning, i.e. one that is easy to remember and recognize. A QF-Test ID must be unique within the test-suite.

Variable: No

Restrictions: Must not be empty, contain any of the characters '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Name of the browser window

This attribute can be ignored unless you need to test a web application with multiple open browser windows holding similar documents. In that case the Name of the browser window attribute can be used to identify the browser window. The name of a browser window can be defined via the Name of the browser window⁽⁴³¹⁾ attribute of a Start browser⁽⁴²⁸⁾ node.

Variable: Yes

Restrictions: None

Name

A web page has no name unless one is implemented via a NameResolver. See section 39.1⁽⁶⁷⁸⁾ about the extension API for NameResolvers.

Variable: Yes

Restrictions: None

Feature

The main Feature of a web page is its URL with the parameters removed. If the option Limit URL feature of 'Web page' node to host or file⁽²⁹⁵⁾ is set, the URL is further reduced to the host or file name.

See section 36.2⁽⁶¹²⁾ for more information about component recognition.

You can select Escape text for regular expressions from the context menu for escaping special characters of regular expressions of that text.

Variable: Yes

Restrictions: None

As regexp

If this attribute is set, the Feature⁽⁵⁴²⁾ is a regular expression (see section 36.4⁽⁶¹⁴⁾).

Variable: Yes

Restrictions: None

Extra features

Besides the main Feature a Component can have additional features represented as name/value pairs. The kind of component determines which extra features are recorded. Each extra feature can have one of three states:

4.0+

Ignore (Hint for searching: currently represented as 0)

This extra feature is just for information. It has no influence on component recognition.

Should match (Hint for searching: currently represented as 1)

Target components matching this extra feature have a higher probability for component recognition than those that don't match it.

Must match (Hint for searching: currently represented as 2)

The target component must match this extra feature. Any component not matching it is no candidate for component recognition.

Additional columns allow for matching against a regular expression or to negate the expression, e.g. to define that the "class" attribute of a DOM node in a web page should not be "dummy". The absence of an extra feature can be enforced by adding one with an empty value.

You can select Escape text for regular expressions from the context menu for escaping special characters of regular expressions of the value.

QF-Test automatically assigns some extra features to recorded components.

Name	Engine	Description
columns	Web	Column count in TABLE components.
imagehash	Swing, SWT	Shows the hash value of icons of a button or menuitem.
qfs:class	All	Dedicated component class, e.g. de.qfs.QfsTextField.
qfs:genericclass	All	Generic class of the component, e.g. TextField.
qfs:item	Web	Shows the item index of a DomNode, if it's an item of a complex GUI component. This could get recorded, if some child nodes are additionally recorded as those component might be interesting as well. This could affect content of lists, tables, tabfolders or trees.
qfs:label	All	Shows a matching label for the component, e.g. the text of a button or a label close to the respective component. If no own text or label could be found, it also tries to use tooltips or icon descriptions.
qfs:matchindex	All	Index of components with the same name. Possibly assigned automatically when the <u>Validate component recognition during recording</u> ⁽²⁶³⁾ option is active.
qfs:modal	Web	Shows if a component of the class "Window" is modal.
qfs:originalid	Web	Shows the real 'ID' attribute specified in the DOM for that node.
qfs:systemclass	All	The toolkit-specific system class, e.g. javax.swing.JTextField.
qfs:type	All	The generic type of the component, e.g. TextField:PasswordField.

Table 30.24: Extra features assigned by QF-Test

Variable: Yes**Restrictions:** Names must not be empty

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

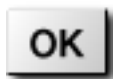
Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.13.3 Component

Component nodes represent the components of the SUT. Other nodes refer to them by their QF-Test ID⁽⁵⁴⁶⁾, similar to the way Windows⁽⁵³⁵⁾ are referenced.

The Window⁽⁵³⁵⁾ node that is the direct or indirect parent of the component must be the equivalent of the component's window parent in the SUT.

Contained in: Component group⁽⁵⁵⁵⁾, Window⁽⁵³⁵⁾.

Children: Item⁽⁵⁵¹⁾.

Execution: Cannot be executed.

Attributes:

The screenshot shows a 'Component' dialog box with the following fields and sections:

- QF-Test ID:** bExit
- Class name:** javax.swing.JButton
- Name:** Exit
- Feature:** Exit
 - ☐ As regexp
 - Extra features:** A table with columns: State, Regexp, Negate, Name. The table is currently empty.
- Structure:**
 - Class index:** (empty field)
 - Class count:** (empty field)
- Geometry:**
 - X:** 160
 - Y:** 166
 - Width:** 59
 - Height:** 25
- Comment:** Closes the main window.

Figure 30.73: Component attributes

QF-Test ID

This ID is the means by which other nodes refer to this component. Therefore it may appear in many places and you should take care to assign an ID with a meaning, i.e. one that is easy to remember and recognize. A QF-Test ID must be unique within the test-suite.

Variable: No

Restrictions: Must not be empty, contain any of the characters '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Class name

For SWT and Swing application this is the fully qualified name of the Java class of the component or one of its super-classes whereas for web applications there is a pseudo class hierarchy described in [section 5.4^{\(41\)}](#).

The actual class recorded by QF-Test depends on the setting of the option `Record system class only(263)` and on potentially registered `ClassNameResolvers` (see [section 39.1.2^{\(680\)}](#)). Upon replay, class matching is based on the actual or pseudo class hierarchy, so you can manually change this attribute to any of the element's base classes.

Variable: Yes

Restrictions: Must not be empty.

Name

The name of the component that was set by the developers of the SUT with the `setName(String)` method. See [section 36.2^{\(612\)}](#) for why and how names should be set on all "interesting" components.

Variable: Yes

Restrictions: None

Feature

If there is no `Name(547)` available for the component, QF-Test tries to recognize it by a characteristic feature.

See [section 36.2^{\(612\)}](#) for more information about component recognition.

You can select Escape text for regular expressions from the context menu for escaping special characters of regular expressions of that text.

Variable: Yes

Restrictions: None

As regexp

If this attribute is set, the `Feature(547)` is a regular expression (see [section 36.4^{\(614\)}](#)).

Variable: Yes

Restrictions: None

Extra features

Besides the main Feature a Component can have additional features represented as name/value pairs. The kind of component determines which extra features are recorded. Each extra feature can have one of three states:

Ignore (Hint for searching: currently represented as 0)

This extra feature is just for information. It has no influence on component recognition.

Should match (Hint for searching: currently represented as 1)

Target components matching this extra feature have a higher probability for component recognition than those that don't match it.

Must match (Hint for searching: currently represented as 2)

The target component must match this extra feature. Any component not matching it is no candidate for component recognition.

Additional columns allow for matching against a regular expression or to negate the expression, e.g. to define that the "class" attribute of a DOM node in a web page should not be "dummy". The absence of an extra feature can be enforced by adding one with an empty value.

You can select Escape text for regular expressions from the context menu for escaping special characters of regular expressions of the value.

QF-Test automatically assigns some extra features to recorded components.

Name	Engine	Description
columns	Web	Column count in TABLE components.
imagehash	Swing, SWT	Shows the hash value of icons of a button or menuitem.
qfs:class	All	Dedicated component class, e.g. de.qfs.QfsTextField.
qfs:genericclass	All	Generic class of the component, e.g. TextField.
qfs:item	Web	Shows the item index of a DomNode, if it's an item of a complex GUI component. This could get recorded, if some child nodes are additionally recorded as those component might be interesting as well. This could affect content of lists, tables, tabfolders or trees.
qfs:label	All	Shows a matching label for the component, e.g. the text of a button or a label close to the respective component. If no own text or label could be found, it also tries to use tooltips or icon descriptions.
qfs:matchindex	All	Index of components with the same name. Possibly assigned automatically when the <u>Validate component recognition during recording</u> ⁽²⁶³⁾ option is active.
qfs:modal	Web	Shows if a component of the class "Window" is modal.
qfs:originalid	Web	Shows the real 'ID' attribute specified in the DOM for that node.
qfs:systemclass	All	The toolkit-specific system class, e.g. javax.swing.JTextField.
qfs:type	All	The generic type of the component, e.g. TextField:PasswordField.

Table 30.25: Extra features assigned by QF-Test

Variable: Yes**Restrictions:** Names must not be empty

Structure

These two fields hold additional structural information needed for component recognition⁽⁶¹²⁾ during a test run. The Class count is the number of components within the same parent container and with the same class (or a class derived thereof). The Class index is the index that this component has in the list of these components with matching class. As usual the first component has index 0.

When counting the components of matching class, invisible components are considered as well. This is more robust but means that the values may be higher than expected.

Note

It is possible to specify just the Class index or the Class count attribute.

Variable: Yes

Restrictions: None

Geometry

The X/Y coordinate, width and height of the component form the basis for the recognition⁽⁶¹²⁾ of the component. However, they play a minor role as long as either a Name⁽⁵⁴⁷⁾ was provided or a Feature⁽⁵⁴⁷⁾ or structural information⁽⁵⁴⁹⁾ is available.

For components whose location or size typically vary widely at runtime, these values are not recorded.

Note

If no values are specified, the recognition algorithm starts with a perfect geometry match for all candidates. To prevent false positive hits you can disable geometry matching by setting the values to a single '-' character.

Variable: Yes

Restrictions: Width and height must not be negative.

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.13.4 Item



For complex Swing components like `JTable` or `JTree` it is possible to define Mouse events⁽⁴⁴⁷⁾, checks⁽⁴⁶⁹⁾ or queries⁽⁴⁹⁸⁾ relative to a sub-item of the component instead of the component itself. Such a sub-item is identified with the help of an index, the Primary index⁽⁵⁵²⁾. This index can be given in one of three ways: as a string, a number or a regexp (see section 36.4⁽⁶¹⁴⁾). A string or regexp designate a sub-item with a corresponding representation while a number refers to a sub-item by its index. Like in Java the first sub-item's index is 0.

The `JTable` supports an additional index to refer directly to a table cell. The Primary index⁽⁵⁵²⁾ determines the column and the Secondary index⁽⁵⁵²⁾ the row of the cell.

There are two representations for the nodes of a `JTree` component, flat like a list or as a hierarchy using paths. A node named `tmp` under a node named `usr` is represented as just `tmp` in the first case, as `/usr/tmp` in the latter. The option Represent tree node as path⁽²⁶⁹⁾ determines the representation used when recording sub-items.

Currently sub-items are supported for the following Swing components:

Class	Primary	Secondary
<code>JComboBox</code>	List element	-
<code>JEditorPane</code>	Structural element (experimental)	-
<code>JList</code>	List element	-
<code>JTabbedPane</code>	Tab	-
<code>JTable</code>	Table column	Row
<code>JTableHeader</code>	Table column	-
<code>JTextArea</code>	Line	-
<code>JTree</code>	Node/Row	-

Table 30.26: Sub-items of complex Swing components

Contained in: Component⁽⁵⁴⁵⁾.

Children: None.

Execution: Cannot be executed.

Attributes:

The screenshot shows a dialog box titled 'Item'. It contains the following fields and options:

- QF-Test ID:** A text field containing 'cellGreg'.
- Primary index:** A text field containing 'First name'.
- As string / As number / As regexp:** Three radio buttons. 'As string' is selected.
- Secondary index:** A checked checkbox.
- Secondary index text:** A text field containing 'Greg'.
- As string / As number / As regexp:** Three radio buttons. 'As string' is selected.
- Comment:** A text area with a yellow notepad icon. It contains the text 'Cell "Greg" in the "First name" column of the table.'

Figure 30.74: Item attributes

QF-Test ID

This ID is the means by which other nodes refer to this sub-item. Therefore it may appear in many places and you should take care to assign an ID with a meaning, i.e. one that is easy to remember and recognize. A QF-Test ID must be unique within the test-suite.

Variable: No

Restrictions: Must not be empty, contain any of the characters '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Primary index

Designates the sub-item. Depending on whether As string⁽⁵⁵³⁾, As number⁽⁵⁵³⁾ or As regexp⁽⁵⁵³⁾ is selected, the sub-item is determined by its index or by a string or regexp match.

It is OK to have an empty index, e.g. to designate a table column with an empty heading.

Variable: Yes

Restrictions: Must be a valid number or regexp if required.

Secondary index

For the `JTable` class two kinds of sub-items are supported. If only the Primary index⁽⁵⁵²⁾ is given, a whole column is referenced. An additional Secondary index designates a cell in this column.

To define a secondary index its checkbox must be selected first. This is necessary to tell an empty index from a non-existent one. An empty secondary index designates a table cell with empty content.

Variable: Yes

Restrictions: Must be a valid number or regexp if required.

As string

The Primary index⁽⁵⁵²⁾ or Secondary index⁽⁵⁵²⁾ is interpreted as a plain string. The sub-item is determined by matching its representation against that string.

Variable: No

Restrictions: None

As number

The Primary index⁽⁵⁵²⁾ or Secondary index⁽⁵⁵²⁾ is interpreted as a number. The sub-item is determined by its index.

Variable: No

Restrictions: None

As regexp

The Primary index⁽⁵⁵²⁾ or Secondary index⁽⁵⁵²⁾ is interpreted as a regexp (see section 36.4⁽⁶¹⁴⁾). The sub-item is determined by matching its representation against that regexp.

Variable: No

Restrictions: None

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.13.5 Window group



A Window group's only purpose is to provide structure to the Windows⁽⁵³⁵⁾ of a test-suite. If you have to manage a large number of Windows⁽⁵³⁵⁾ you can even nest Window groups.

Contained in: Windows and components⁽⁵⁵⁶⁾, Window group⁽⁵⁵⁴⁾.

Children: Window group⁽⁵⁵⁴⁾, Window⁽⁵³⁵⁾.

Execution: Cannot be executed.

Attributes:

Figure 30.75: Window group attributes

Name

You can choose an arbitrary name. It is displayed in the tree view of the suite.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes,

this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

30.13.6 Component group



A Component group's only purpose is to provide structure to the Components⁽⁵⁴⁵⁾ of a Window⁽⁵³⁵⁾. If you have to manage a large number of Components⁽⁵⁴⁵⁾ inside a Window⁽⁵³⁵⁾ you can even nest Component groups.

Contained in: Window⁽⁵³⁵⁾, Component group⁽⁵⁵⁵⁾.

Children: Component group⁽⁵⁵⁵⁾, Component⁽⁵⁴⁵⁾.

Execution: Cannot be executed.

Attributes:

Figure 30.76: Component group attributes

Name

You can choose an arbitrary name. It is displayed in the tree view of the suite.

Variable: No

Restrictions: None

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing **[Alt-Return]** or by clicking the  button.

Variable: Yes

Restrictions: None

30.13.7 Windows and components



The Windows and components node is always located at the end of the suite. It is the place where all of the Windows⁽⁵³⁵⁾, Components⁽⁵⁴⁵⁾ and Items⁽⁵⁵¹⁾ of the test-suite are collected.

Contained in: Root node

Children: Window group⁽⁵⁵⁴⁾, Window⁽⁵³⁵⁾.

Execution: Cannot be executed.

Attributes:

Figure 30.77: Windows and components attributes

QF-Test ID

At the moment the QF-Test ID attribute has no meaning for this type of node.

Variable: No

Restrictions: Must not contain any of the characters '\', '#', '\$', '@', '&', or '%' or start with an underscore ('_').

Comment

Here you can enter a comment that explains the purpose of this node. This is the preferred way of documenting the test-suite.

Note

For detailed documentation, especially for Test-set, Test-case or Procedure nodes, this text area might not be the right place. There are many excellent editors that are much better suited to this task. The option External editor command⁽²⁴⁷⁾ lets you define an external editor in which comments can be edited conveniently by

pressing Alt-Return or by clicking the  button.

Variable: Yes

Restrictions: None

Chapter 31

Exceptions

There are quite a lot of exceptions that can be thrown during the execution of a test. This chapter lists the exceptions in hierarchical order, shows the typical error messages and gives a short explanation.

If you want to work with those exceptions in scripts, please take a look into section 37.9⁽⁶⁴⁵⁾.

TestException

This is the base class of all exceptions that can be thrown during a test run. The actual exception thrown should almost always be of a derived class. A Catch⁽⁴⁰⁸⁾ with the Exception class⁽⁴⁰⁹⁾ set to `TestException` will catch all possible exceptions. Just like in Java you should normally not use such a Catch⁽⁴⁰⁸⁾ since it may hide unexpected Exceptions.

ComponentNotFoundException

This exception is thrown whenever the target component for an event⁽⁴⁴⁶⁾ or a check⁽⁴⁶⁹⁾ cannot be determined. Failure of a Wait for component to appear⁽⁵¹²⁾ will also cause a `ComponentNotFoundException` unless the node's Wait for absence⁽⁵¹⁴⁾ attribute is set.

DocumentNotLoadedException

This exception is a variant of `ComponentNotFoundException` and thrown specifically if a Wait for document to load⁽⁵¹⁶⁾ node fails.

ComponentFoundException

This is the opposite of a `ComponentNotFoundException`, thrown by a Wait for component to appear⁽⁵¹²⁾ with the Wait for absence⁽⁵¹⁴⁾ attribute set.

ModalDialogException

This exception is thrown when an event⁽⁴⁴⁶⁾ is blocked by a modal dialog. See the option Check for modal dialogs⁽²⁸¹⁾ for details.

ComponentCannotGetFocusException

Note

This exception is obsolete and should not occur anymore.

This exception is thrown when the target component for a Key event⁽⁴⁵⁰⁾ or Text input⁽⁴⁵⁴⁾ is a text component that cannot get the keyboard focus for some reason. With JDK 1.4 the event cannot be delivered to the component in that case.

DisabledComponentException

This exception is thrown when the target component for a Mouse event⁽⁴⁴⁷⁾, Key event⁽⁴⁵⁰⁾ or Text input⁽⁴⁵⁴⁾ is not enabled. In that case the event would be silently ignored, very likely leading to unexpected results during further execution of the test.

For backwards compatibility this kind of exception can be suppressed by deactivating the option Throw DisabledComponentException⁽²⁸²⁾.

DisabledComponentStepException

This exception is thrown when the target component for a Mouse event⁽⁴⁴⁷⁾, Key event⁽⁴⁵⁰⁾ or Text input⁽⁴⁵⁴⁾ is disabled under Windows and components.

BusyPaneException

This exception is thrown when the target component for a Mouse event⁽⁴⁴⁷⁾, Key event⁽⁴⁵⁰⁾ or Text input⁽⁴⁵⁴⁾ is covered by a `GlassPane` with a "busy" mouse cursor. The option Wait for 'busy' GlassPane (ms)⁽²⁹²⁾ determines how long QF-Test will wait for the `GlassPane` to disappear before the exception is actually triggered.

InvisibleDnDTargetException

This exception is thrown when the location in the target component for a Mouse event⁽⁴⁴⁷⁾ of type `DRAG_FROM`, `DRAG_OVER` or `DROP_TO` for a `Drag&Drop`

operation is invisible and cannot be made visible by scrolling the target component.

DeadlockTimeoutException

This exception is thrown when the SUT does not react for a given amount of time which is defined in the option Deadlock detection (s)⁽²⁹¹⁾.

VariableException

This exception is never thrown itself but is the base class for exceptions thrown in the context of variable expansion.

BadVariableSyntaxException

This exception is thrown when a value that is to be expanded doesn't follow a proper variable syntax (e.g. no closing brace).

MissingPropertiesException

This exception is thrown when no properties or ResourceBundle are available for the group name of a property or resource looked up with `${group:prop}` (see Load resources⁽⁵²³⁾ and Load properties⁽⁵²⁶⁾).

MissingPropertyException

This exception is thrown when a property looked up with `${id:property}` is not available (see Load resources⁽⁵²³⁾ and Load properties⁽⁵²⁶⁾).

RecursiveVariableException

This exception is thrown when the expansion of a variable expression leads to recursive variable lookup, e.g. if you set the variable named `x` to `$(y)` and the variable named `y` to `$(x)` and then try to expand the value `$(x)`.

UnboundVariableException

This exception is thrown when a variable for a value that is to be expanded doesn't exist.

VariableNumberException

This exception is thrown when a variable expansion for a numeric attributes results in something other than a number.

BadExpressionException

This exception is thrown when evaluating a `$[...]` expression fails (see [section 7.6^{\(66\)}](#)).

BadTestException

This exception is thrown when evaluating the [Condition^{\(397\)}](#) of an [If^{\(396\)}](#) or [Elseif^{\(399\)}](#) node fails.

BadRegexpException

This exception is thrown whenever converting a String to a regular expression (see [section 36.4^{\(614\)}](#)) fails, e.g. for [Item^{\(551\)}](#) or [Check text^{\(470\)}](#) nodes.

BadRangeException

This exception is thrown when the syntax of the Iteration ranges for a Data binder node has invalid syntax or specifies an index outside the valid data range.

CannotExecuteException

This exception is thrown when execution of a process from a [Start SUT client^{\(426\)}](#), [Start Java SUT client^{\(422\)}](#) or [Start process^{\(433\)}](#) node fails.

InvalidDirectoryException

This exception is thrown when the [Directory^{\(427\)}](#) attribute of a [Start SUT client^{\(426\)}](#) node refers to a non-existent directory.

CheckFailedException

This exception is thrown when a [check^{\(469\)}](#) node with activated [Throw exception on failure^{\(475\)}](#) attribute fails.

CheckNotSupportedException

As explained in the section about checks⁽⁴⁶⁹⁾, each check can handle only a limited set of target components. This exception is thrown when the target component is not suitable for a check.

OperationNotSupportedException

This exception is thrown when an operation like Fetch text⁽⁴⁹⁹⁾ is not supported for the designated target component.

BadComponentException

This exception is thrown, when a component for an event⁽⁴⁴⁶⁾ is not suitable, i.e. a non-window for a Window event⁽⁴⁵⁷⁾.

IndexFormatException

This exception is thrown when an invalid index format for a sub-item is encountered (see section 6.3⁽⁵⁴⁾).

IndexFoundException

This exception is thrown when a sub-item is found during execution of a Wait for component to appear⁽⁵¹²⁾ node that looks for the absence of the item.

IndexNotFoundException

This exception is thrown when no sub-item can be located for a given index.

IndexRequiredException

This exception is thrown when no sub-item index is provided for an operation that requires one, e.g. a Check text⁽⁴⁷⁰⁾ on a `JTree`.

UnexpectedIndexException

This exception is thrown when a sub-item index is provided for an operation that does not require one, e.g. a Check items⁽⁴⁸⁰⁾ on a `JTree`.

ClientNotConnectedException

This exception is thrown when the target client for an operation is not connected. It differs from a NoSuchClientException⁽⁵⁶²⁾ in that there is an active process for that name but no RMI connection.

NoSuchClientException

This exception is thrown when the target client for an operation does not exist.

DuplicateClientException

This exception is thrown if an attempt is made to run more than one client simultaneously under the same name.

UnexpectedClientException

This exception is thrown when an unexpected exception is thrown in the SUT during the replay of an event⁽⁴⁴⁶⁾. Unless it is due to a bug in QF-Test, it indicates a problem in the SUT.

ClientNotTerminatedException

This exception is thrown when a Wait for process to terminate⁽⁴⁴³⁾ node is executed and the process doesn't terminate.

UnexpectedExitCodeException

This exception is thrown when the exit code of a terminated client doesn't match the expected value in a Wait for process to terminate⁽⁴⁴³⁾ node's Expected exit code⁽⁴⁴⁴⁾ attribute.

BadExitCodeException

This exception is thrown when the Expected exit code⁽⁴⁴⁴⁾ attribute of a Wait for process to terminate⁽⁴⁴³⁾ node doesn't match the specification and cannot be parsed.

ComponentIdMismatchException

This exception is thrown when the QF-Test component ID⁽⁴⁴⁸⁾ attribute of a node points to a node that is not a Window⁽⁵³⁵⁾, Component⁽⁵⁴⁵⁾ or Item⁽⁵⁵¹⁾.

UnresolvedComponentIdException

This exception is thrown when the target of the QF-Test component ID⁽⁴⁴⁸⁾ attribute of a node cannot be determined.

TestNotFoundException

This exception is thrown when the Test⁽³³⁰⁾ for a Test call⁽³²⁷⁾ cannot be determined.

DependencyNotFoundException

This exception is thrown when the Dependency⁽³⁴⁴⁾ for a Dependency reference⁽³⁴⁷⁾ cannot be determined.

ProcedureNotFoundException

This exception is thrown when the Procedure⁽³⁷⁸⁾ for a Procedure call⁽³⁸¹⁾ cannot be determined.

StackOverflowException

This exception is thrown when the nesting of Procedure calls⁽³⁸¹⁾ gets too deep, hinting to a problem with endless recursion. See also the option Call stack size⁽²⁷¹⁾.

UserException

This exception is thrown explicitly by a Throw⁽⁴¹³⁾ node.

CannotRethrowException

This exception is thrown when an attempt is made to rethrow an exception with a Rethrow⁽⁴¹⁵⁾ node but no exception was caught by a Catch⁽⁴⁰⁸⁾ node.

ScriptException

This exception is thrown when the execution of a script from a Server script⁽⁴¹⁶⁾ or SUT script⁽⁴¹⁹⁾ fails.

BreakException

This is not a standard `TestException` and cannot be caught by a Catch⁽⁴⁰⁸⁾ node. It is thrown by a Break⁽³⁹⁴⁾ node in order to break out of a loop. From a script, raising a `BreakException` will have the same effect. If thrown outside of a loop, a `BreakException` will cause the error below.

ReturnException

This is not a standard `TestException` and cannot be caught by a Catch⁽⁴⁰⁸⁾ node. It is thrown by a Return⁽³⁸⁴⁾ node in order to return from a Procedure⁽³⁷⁸⁾. From a script, raising a `ReturnException` will have the same effect. If thrown outside of a `Procedure`, a `ReturnException` will cause the error below.

TestOutOfMemoryException

This is a special exception that is thrown when QF-Test determines that it is running out of memory during test execution. The exception causes the test to abort immediately and cannot be caught because once QF-Test has run out of memory there is little it can do to handle it. QF-Test tries to keep a little reserve memory so it will at least try to save the run-log.

Part IV

Technical reference

Chapter 32

Command line arguments and exit codes

32.1 Call syntax

The call syntax for interactive and batch mode varies widely since some command line arguments are specific to interactive mode or batch mode or even sub-modes of batch mode. Note that all of the arguments have sensible default values which you only need to override for special cases. In most cases you'll only need to execute either `qftest [<suite> | <run-log>]*` to run QF-Test in interactive mode, or `qftest -batch [-runlog [<file>]] [-report <directory>] <suite>` to execute a test in batch mode.

Windows

The program `qftest.exe` is a Windows GUI application. When started from a command shell, it will not wait for QF-Test to terminate but return immediately. Thus, when executing a test in batch mode, you cannot see whether QF-Test has finished or not (you may put the command into a `.bat` file to deal with this behaviour). Furthermore you won't see any output from QF-Test in the console window when using `qftest.exe`. For both reasons you may prefer to utilize the `qftestc.exe` Console application when launching QF-Test from a command shell: It waits for QF-Test to terminate and print output from Server scripts⁽⁴¹⁶⁾ will be displayed in the console window. Apart from that, everything said about `qftest.exe` in this chapter holds true for `qftestc.exe` too.

Interactive mode

The full call syntax for interactive mode is:

```
qftest [-dbg(571)] [-java <executable>(572)]
[-noconsole(572)] [-plugindir <directory>(572)]
[-J<java-argument>]* [-daemon(574)] [-daemonhost <host>(575)]
[-daemonport <port>(575)] [-daemonrmiport <port>(575)]
```

```

[-dontkillprocesses(575)] [-engine <engine>(575)] [-help(576)]
[-jythonport <number>(577)] [-keystore <keystore file>(577)]
[-keypass <keystore password>(577)] [-libpath <path>(577)]
[-license <file>(577)] [-options <file>(578)] [-port <number>(579)]
[-reuse(581)] [-runlogdir <directory>(582)] [-runtime(582)]
[-serverhost <host>(583)] [-shell <executable>(583)]
[-shellarg <argument>(583)] [-systemcfg <file>(584)]
[-tempdir <directory>(584)] [-usercfg <file>(586)]
[-variable <name>=<value>(586)] * [-version(586)] [<suite> |
<run-log>] *

```

There are several sub-modes for running QF-Test in batch mode. The default is to execute one or more test-suites. Alternatively QF-Test can be invoked to create test documentation from test-suites or reports from run-logs or to instrument one or more JDKs or JREs. QF-Test can also be run in daemon mode where it sits in the background waiting for calls from the outside telling it what to do (see [chapter 40^{\(741\)}](#) for further information about the daemon mode). Finally, showing help or version information can also be seen as separate sub-modes.

Test execution

To execute one or more test-suites and create a run-log and/or report as a result, use:

```

qftest -batch [-run(581)] [-dbg(571)] [-java <executable>(572)]
[-noconsole(572)] [-plugindir <directory>(572)]
[-J<java-argument>] * [-clearglobals(574)] [-compact(574)]
[-engine <engine>(575)] [-exitcodeignoreexception(575)]
[-exitcodeignoreerror(576)] [-exitcodeignorewarning(576)]
[-jythonport <number>(577)] [-keystore <keystore file>(577)]
[-keypass <keystore password>(577)] [-libpath <path>(577)]
[-license <file>(577)] [-nolog(577)] [-nomessagewindow(578)]
[-options <file>(578)] [-port <number>(579)]
[-report <directory>(579)] [-report.checks(579)]
[-report.doctags(579)] [-report.errors(580)]
[-report.exceptions(580)] [-report.html <directory>(580)]
[-report.ignorenotimplemented(580)]
[-report.ignoreskipped(580)] [-report.junit <directory>(580)]
[-report.name <name>(580)] [-report.nodeicons(580)]
[-report.pashtml(581)] [-report.piechart(581)]
[-report.scale-thumbnails <percent>(581)]
[-report.teststeps(581)] [-report.thumbnails(581)]
[-report.warnings(581)] [-report.xml <directory>(581)]
[-runid <ID>(582)] [-runlogdir <directory>(582)]

```

```
[-runlog [<file>](582)] [-runtime](582) [-serverhost <host>](583)]
[-shell <executable>](583)] [-shellarg <argument>](583)]
[-sourcedir <directory>](583)] [-splitlog](583)]
[-systemcfg <file>](584)] [-test <n>|<ID>](584)] *
[-threads <number>](586)] * [-variable <name>=<value>](586)] *
[-verbose [<level>](586)] <suite>+
```

Test execution via QF-Test daemon

The following parameters can be specified when executing a test-case by calling a daemon:

```
qftest -batch -calldaemon(574) [-cleanup](574) [-dbg](571)]
[-java <executable>](572)] [-noconsole](572)] [-J<java-argument>](572)] *
[-daemonhost <host>](575)] [-daemonport <port>](575)]
[-exitcodeignoreexception](575)] [-exitcodeignoreerror](576)]
[-exitcodeignorewarning](576)] [-keystore <keystore file>](577)]
[-keypass <keystore password>](577)] [-nomessagewindow](578)]
[-ping](578)] [-options <file>](578)] [-runid <ID>](582)]
[-runlogdir <directory>](582)] [-runlog [<file>](582)]
[-startclean](583)] [-startsut](583)] [-stopclean](584)]
[-stoprun](584)] [-suitedir <dir>](584)] [-terminate](584)]
[-timeout <milliseconds>](586)] [-variable <name>=<value>](586)] *
[-verbose [<level>](586)] <suite#test-case>
```

Create test documentation

Package or test-case documentation can be create for one or more test-suites or whole directories. This is described further in [chapter 15](#)⁽¹²⁹⁾. The command line syntax is:

```
qftest -batch -gendoc](576) [-dbg](571)] [-java <executable>](572)]
[-noconsole](572)] [-J<java-argument>](572)] * [-license <file>](577)]
[-nomessagewindow](578)] [-options <file>](578)]
[-pkgdoc <directory>](578)] [-pkgdoc.dependencies](578)]
[-pkgdoc.doctags](578)] [-pkgdoc.html <directory>](578)]
[-pkgdoc.includelocal](578)] [-pkgdoc.nodeicons](579)]
[-pkgdoc.passhtml](579)] [-pkgdoc.sortpackages](579)]
[-pkgdoc.sortprocedures](579)] [-pkgdoc.xml <directory>](579)]
[-sourcedir <directory>](583)] [-testdoc <directory>](584)]
[-testdoc.doctags](585)] [-testdoc.followcalls](585)]
[-testdoc.html <directory>](585)] [-testdoc.nodeicons](585)]
[-testdoc.passhtml](585)] [-testdoc.sorttestcases](585)]
[-testdoc.sorttestsets](585)] [-testdoc.teststeps](585)]
[-testdoc.xml <directory>](585)] (<suite> | <directory>) +
```

Create a report from run-logs

To create a report from one or more run-logs or whole directories use:

```
qftest -batch -genreport(576) [-dbg(571)] [-java <executable>(572)]
[-noconsole(572)] [-J<java-argument>]* [-license <file>(577)]
[-nomessagewindow(578)] [-options <file>(578)]
[-report <directory>(579)] [-report.checks(579)]
[-report.doctags(579)] [-report.errors(580)]
[-report.exceptions(580)] [-report.html <directory>(580)]
[-report.ignorenotimplemented(580)]
[-report.ignorestipped(580)] [-report.junit <directory>(580)]
[-report.name <name>(580)] [-report.nodeicons(580)]
[-report.passhtml(581)] [-report.piechart(581)]
[-report.scale-thumbnails <percent>(581)]
[-report.teststeps(581)] [-report.thumbnails(581)]
[-report.warnings(581)] [-report.xml <directory>(581)]
[-runlogdir <directory>(582)] (<run-log> | <directory>)+
```

Instrument one or more JDKs or JREs

This is useful for automating tests that include clean installation of an SUT that comes with its own JRE, so it needs to be instrumented before the test on the SUT can be run. This can be done as follows:

```
qftest -batch -instrument(577) [-dbg(571)]
[-java <executable>(572)] [-noconsole(572)] [-J<java-argument>]*
[-license <file>(577)] [-nomessagewindow(578)]
[-options <file>(578)] <path to JDK or JRE>+
```

Daemon mode

To run QF-Test in daemon mode as described [chapter 40](#)⁽⁷⁴¹⁾ use:

```
qftest -batch -daemon(574) [-dbg(571)]
[-java <executable>(572)] [-noconsole(572)]
[-plugindir <directory>(572)] [-J<java-argument>]*
[-daemonhost <host>(575)] [-daemonport <port>(575)]
[-daemonrmipport <port>(575)] [-engine <engine>(575)]
[-jythonport <number>(577)] [-keystore <keystore file>(577)]
[-keypass <keystore password>(577)] [-libpath <path>(577)]
[-license <file>(577)] [-nolog(577)] [-nomessagewindow(578)]
[-options <file>(578)] [-port <number>(579)] [-runtime(582)]
[-serverhost <host>(583)] [-shell <executable>(583)]
[-shellarg <argument>(583)] [-splitlog(583)]
[-systemcfg <file>(584)] [-variable <name>=<value>(586)]*
```

Import one test-suite into another

```
qftest -batch -import(576) [-import.from <test-suite>(576)]
[-import.into <test-suite>(576)] [-import.components(576)]
[-import.procedures(577)] [-import.tests(577)]
```

Analyze references of a test-suite

```
qftest -batch -analyze(573) [-analyze.target <directory>(573)]
[-suitedir <dir>(584)] [-analyze.references(573)]
[-analyze.duplicates(573)] [-analyze.invalidchar(573)]
[-analyze.emptynodes(573)] [-analyze.components(573)]
[-analyze.procedures(573)] [-analyze.dependencies(573)]
[-analyze.tests(573)] [-analyze.packages(574)]
[-remove.unusedcallables(574)] [-remove.unusedcomponents(574)]
[-analyze.transitive(574)] [-analyze.followincludes(574)]
(<suite> | <directory>)+
```

Get version information

```
qftest -batch -version(586)
```

Get help

```
qftest -batch -help(576)
```

32.2 Command line arguments

Command line arguments for QF-Test fall in three categories. They can be mixed freely.

32.2.1 Arguments for the starter script

These arguments are evaluated directly by the `qftest` shell script or executable and override settings determined during installation. Under Unix these settings are stored in the file `launcher.cfg` in QF-Test's root directory, under Windows they are stored in the registry.

-batch

Run QF-Test in batch mode. This causes QF-Test to load and execute a test-suite directly and finish with an exit code that represents the result of the test run.

-dbg

Turn on debugging output for the starter script. The same effect is achieved by setting the environment variable `QFTEST_DEBUG` to a non-empty value. On Windows this causes QF-Test to open a console window to display the output that would otherwise be invisible unless the argument `-noconsole`⁽⁵⁷²⁾ is also given. This also turns on debugging output for the `qfclient` and `java` helper programs when using the old connection mechanism (see [chapter 34](#)⁽⁵⁹²⁾).

-java <executable>

The Java executable used to run QF-Test. The default is `java` on Unix and `javaw.exe` on Windows, unless a different value was set during installation.

-noconsole (Windows only)

On Windows this argument suppresses the console window that would otherwise be opened in case `-dbg`⁽⁵⁷¹⁾ is specified.

-plugindir <directory>

This argument overrides the default location of the directory for plugins, jar files that should be made accessible to scripts. The default directory called `plugin` is located under QF-Test's root directory `qftest`. See [section 37.2](#)⁽⁶²⁰⁾ for more information about plugins.

32.2.2 Arguments for the Java VM

You can pass arguments to the Java VM through the starter script by prepending them with `-J`, e.g. `-J-Duser.language=en` to set a system property. To set the classpath, prepend `-J` only to the `-cp` or `-classpath` argument, not to the actual value, e.g. `-J-classpath myclasses.jar`. When setting the classpath this way, QF-Test's own jar archives need not to be taken into account.

32.2.3 Arguments for QF-Test

The rest of the arguments are handled by QF-Test itself when it is executed by the Java virtual machine. These arguments can also be placed in a file using the syntax `<name>=<value>` for arguments with parameters or `<name>=true` or `<name>=false` to turn a simple argument on or off. By default this file is called `qftest.options`, located in the `bin` directory of QF-Test and used only for internal logging purposes. If you change anything in this file, you can move it to QF-Test's root directory so your changes

will still apply after upgrading. Alternatively you can use the `-options <file>`⁽⁵⁷⁸⁾ argument to specify a different file. Arguments given on the command line override arguments from an option file except for those which can be given multiple times. In the latter case the arguments are merged.

-analyze (batch mode only)

Run the static validation of test-suites. Analyzing results are stored to a given file.

-analyze.target <directory> (batch mode only)

The target folder, where the result file should be created.

-analyze.references (batch mode only)

Switch for analyzing references the given test-suite.

-analyze.duplicates (batch mode only)

Switch for analyzing duplicates of the given test-suite.

-analyze.invalidchar (batch mode only)

Checks the given test-suite for invalid characters in node names.

-analyze.emptynodes (batch mode only)

Checks the given test-suite for empty nodes.

-analyze.components (batch mode only)

Switch for analyzing components of the given test-suite.

-analyze.dependencies (batch mode only)

Switch for analyzing dependency references of the given test-suite.

-analyze.procedures (batch mode only)

Switch for analyzing procedure calls of the given test-suite.

-analyze.tests (batch mode only)

Switch for analyzing test calls of the given test-suite.

-analyze.packages (batch mode only)

Switch for analyzing packages of the given test-suite.

-remove.unusedcallables (batch mode only)

Switch to remove unused callable nodes.

-remove.unusedcomponents (batch mode only)

Switch to remove unused components.

-analyze.transitive (batch mode only)

Switch for analyzing references and calls transitively, i.e. follow the calls and analyze its content also.

-analyze.followincludes (batch mode only)

Switch for analyzing all included test-suites of the given test-suite.

-calldaemon (batch mode only)

Connect to a running QF-Test daemon to execute a test-case.

-cleanup (calldaemon mode only)

With this argument all TestRunDaemons belonging to a daemon are cleaned up and all clients killed before running the test.

-clearglobals (batch mode only)

If more than one test-suite is specified for batch execution, clear global variables and resources before the execution of each test-suite.

-compact (batch mode only)

Create a compact run-log which retains only those branches in which a warning, error or exception occurred. Ignored if log file creation is suppressed with -nolog⁽⁵⁷⁷⁾.

-daemon

Run QF-Test in daemon mode. Further information is provided in chapter 40⁽⁷⁴¹⁾.

-daemonhost <host> (daemon or calldaemon mode only)

In calldaemon mode, specify the host where to locate the QF-Test daemon. The default is localhost. When starting a daemon, either in interactive or in batch mode, this parameter defines the hostname or IP address that daemon objects use at RMI level. The default in this case is chosen by Java, typically the IP address of the primary local network interface.

-daemonport <port>

Specify the registry port for the QF-Test daemon to listen on and to connect to respectively. The default is 3543 or the port defined with `-port <number>`⁽⁵⁷⁹⁾.

-daemonrmipport <port>

Specify the port that the QF-Test daemon should use for RMI communication. Useful only when running the daemon behind a firewall. When running unprotected without SSL (see section 40.3⁽⁷⁵⁵⁾) this can be identical to the daemon port specified with `-daemonport <port>`⁽⁵⁷⁵⁾. If using SSL, two different ports are required.

-dontkillprocesses (batch mode only)

When finishing batch execution, don't explicitly kill processes started by QF-Test as part of the tests. However, whether or not a sub-process of QF-Test survives QF-Test's exit is system-dependent.

-engine <engine>

Specify which engine license(s) to use. This option is only useful in case the QF-Test license contains a mix of GUI engine(s) with different numbers of engine licenses. In that case it may be necessary to specify the engine license(s) to use in order to prevent license conflicts with colleagues using the same license. Possible values are "all" to use all supported licenses, "ask" to bring up a dialog for engine selection or any combination of "awt" for AWT/Swing, "fx" for JavaFX, "swt" for SWT or "web" for Web, e.g. "awt,web". This is explained in more detail in section 29.1.7⁽²⁵³⁾.

-exitcodeignoreexception (batch and calldaemon mode)

For exit code calculation exceptions, errors and warnings are ignored. This means if exceptions, errors or warnings occurred during a test-run the exit code will be 0. This option is helpful when integrating QF-Test with build tools that rate a build as failed dependent on the the exit code.

-exitcodeignoreerror (batch and calldaemon mode)

For exit code calculation errors and warnings are ignored. This means if just errors and warnings occurred during a test-run the exit code will be 0. This option is helpful when integrating QF-Test with build tools that rate a build as failed dependent on the the exit code.

-exitcodeignorewarning (batch and calldaemon mode)

For exit code calculation warnings are ignored. This means if just warnings occurred during a test-run the exit code will be 0. This option is helpful when integrating QF-Test with build tools that rate a build as failed dependent on the the exit code.

-gendoc (batch mode only)

Tell QF-Test that this batch run serves to create test documentation from test-suites.

-genreport (batch mode only)

Tell QF-Test that this batch run serves to create a report from run-logs.

-help

Show help about available command line arguments.

-import (batch mode only)

Import a given test-suite into another one. This mode can be used to merge two test-suites.

-import.from <test-suite> (batch mode only)

The source test-suite, which should be imported into another one.

-import.into <test-suite> (batch mode only)

The target tests-suite for importing.

-import.components (batch mode only)

Switch for merging components of two given test-suites.

-import.procedures (batch mode only)

Switch for merging packages and procedures of two given test-suites.

-import.tests (batch mode only)

Switch for merging test-cases and test-sets of two given test-suites.

-instrument (batch mode only)

Instrument one or more JDKs or JREs automatically. Instrumentation and running of a subsequent test cannot be combined in a single call. They have to be executed as separate commands.

-jythonport <number>

Tell the embedded Jython interpreter to listen for TCP connections at the specified port. You can then use `telnet` to connect to that port and get an interactive Jython command line.

-keystore <keystore file>

An alternative keystore file to use for securing daemon communication with SSL. See [section 40.3^{\(755\)}](#) for details. To disable SSL by specifying no keystore, use this argument in the form `-keystore=`.

-keypass <keystore password>

The password for the keystore file used for securing daemon communication with SSL. See [section 40.3^{\(755\)}](#) for details.

-libpath <path>

Override the library path option ([Directories holding test-suite libraries^{\(252\)}](#)). The directories of the library path should be separated by the standard path separator character for the system, i.e. ';' for Windows and ':' for Unix. QF-Test's `include` directory will automatically be appended to the path.

-license <file>

Set the location of the license file (see [section 1.4^{\(6\)}](#)).

-nolog (batch mode only)

Suppress the automatic creation of a run-log. If any of [-runlog \[<file>\]^{\(582\)}](#),

`-report <directory>(579)`, `-report.html <directory>(580)`,
`-report.xml <directory>(581)` or `-report.junit <directory>(580)` is
given, this argument is ignored. This option is retained for backwards
compatibility only. To keep memory use manageable, split run-logs should be
used instead (see `-splitlog(583)`).

`-nomessagewindow` (batch mode only)

In case of a fatal error in batch mode QF-Test prints an error message to the console and for improved visibility also brings up an error dialog for about 30 seconds. That dialog can be suppressed with the help of this argument. Batch commands that don't require an actual display, i.e. all batch commands that do not execute tests, will run in AWT headless mode if this argument is specified.

`-options <file>`

Override the location of the file used to specify additional command line arguments. This argument can be given more than once to use several sources of command line arguments.

`-ping` (calldaemon mode only)

Use this option, if you want to check whether a daemon is up and running.

`-pkgdoc <directory>` (batch mode only)

With this argument QF-Test creates both HTML and XML pkgdoc documentation. If no directory is given, it is created from the basename of the suite.

`-pkgdoc.dependencies` (batch mode only)

Whether to list dependencies when creating the pkgdoc documentation. Default is true, use `-pkgdoc.dependencies=false` to disable.

`-pkgdoc.doctags` (batch mode only)

Whether to use the QFS doctag extensions when creating the pkgdoc documentation. Default is true, use `-pkgdoc.doctags=false` to disable.

`-pkgdoc.html <directory>` (batch mode only)

With this argument QF-Test creates HTML pkgdoc documentation. If no directory is given, it is created from the basename of the suite.

`-pkgdoc.includelocal` (batch mode only)

Whether to include local packages and procedures (those whose names begin with an '_'). Default is false.

-pkgdoc.nodeicons (batch mode only)

Whether to show icons for nodes in the pkgdoc documentation. Default is true, use `-pkgdoc.nodeicons=false` to disable.

-pkgdoc.passhtml (batch mode only)

Whether to pass HTML tags in comments through to the HTML pkgdoc. Default is true, use `-pkgdoc.passhtml=false` to disable.

-pkgdoc.sortpackages (batch mode only)

Whether to sort packages alphabetically. Default is true, use `-pkgdoc.sortpackages=false` to disable.

-pkgdoc.sortprocedures (batch mode only)

Whether to sort procedures alphabetically. Default is true, use `-pkgdoc.sortprocedures=false` to disable.

-pkgdoc.xml <directory> (batch mode only)

With this argument QF-Test creates XML pkgdoc documentation. If no directory is given, it is created from the basename of the suite.

-port <number>

The TCP port on which QF-Test communicates with the SUT. By default QF-Test uses an arbitrary dynamic port where it creates its own RMI registry. A specific port should only be requested if it must be hard-coded when starting the SUT.

-report <directory> (batch mode only)

Create a combined XML/HTML report. The directory name may contain placeholders as explained in [section 32.2.4^{\(587\)}](#).

-report.checks (batch mode only)

Whether to list checks in the report. Default is false.

-report.doctags (batch mode only)

Whether to use the QFS doctag extensions when creating the report. Default is true, use `-report.doctags=false` to disable.

`-report.errors` (batch mode only)

Whether to list errors in the report. Default is true, use `-report.errors=false` to disable.

`-report.exceptions` (batch mode only)

Whether to list exceptions in the report. Default is true, use `-report.exceptions=false` to disable.

`-report.html <directory>` (batch mode only)

Create an HTML report. The directory name may contain placeholders as explained in [section 32.2.4^{\(587\)}](#).

`-report.ignorenotimplemented` (batch mode only)

Whether to ignore nodes that are not implemented in the report in which case the legend and respective columns for not implemented tests are also not shown. Default is false.

`-report.ignorestipped` (batch mode only)

Whether to ignore skipped nodes in the report in which case the legend and respective columns for skipped tests are also not shown. Default is false.

`-report.junit <directory>` (batch mode only)

Create a report in JUnit XML format as understood by many Continuous Integration Tools. The directory name may contain placeholders as explained in [section 32.2.4^{\(587\)}](#).

`-report.name <name>` (batch mode only)

Specify the name for the report, meaning its identifier, not a file name. Default is the runid. The name may contain placeholders as explained in [section 32.2.4^{\(587\)}](#).

`-report.nodeicons` (batch mode only)

Whether to show icons for nodes in the report. Default is true, use `-report.nodeicons=false` to disable.

-report.passhtml (batch mode only)

Whether to pass HTML tags in comments through to the HTML report. Default is true, `-report.passhtml=false` to disable.

-report.piechart (batch mode only)

Whether to create a pie chart in the top part of the HTML report. Default is true, `-report.piechart=false` to disable.

-report.scale-thumbnails <percent> (batch mode only)

How to scale thumbnail images for screenshots in the error listings of the report, an integer value interpreted as percent of the original image size. Default is 20.

-report.teststeps (batch mode only)

Whether to list test-steps in the report. Default is true, use `-report.teststeps=false` to disable.

-report.thumbnails (batch mode only)

Whether to display thumbnail images for screenshots in the error listings of the report. Default is false.

-report.warnings (batch mode only)

Whether to show warning information in the report. Default is true.

-report.xml <directory> (batch mode only)

Create an XML report. The directory name may contain placeholders as explained in [section 32.2.4^{\(587\)}](#).

-reuse (interactive mode only)

This argument is used mainly when launching QF-Test from a desktop icon or the Windows explorer through a file association. It tells the newly started QF-Test instance to search for an already running version of QF-Test and ask that to open the given file(s). If another instance can be reused in that way, the newly started program will terminate immediately and new windows for the file(s) will be opened by the old instance.

-run (batch mode only)

Explicitely tell QF-Test that this batch run is for actual test execution as opposed to generating documentation or a report. As this is the default operation for batch mode this argument can be omitted.

-runid <ID> (batch and calldaemon mode)

Specify an ID for the test-run. The ID may contain placeholders as explained in [section 32.2.4^{\(587\)}](#) and will itself serve as a replacement for the placeholder %i/+i.

-runlog [<file>] (batch and calldaemon mode)

Save the run-log in the given file. The optional filename may contain placeholders as explained in [section 32.2.4^{\(587\)}](#). If no filename is given, it is composed of the basename of the suite and a timestamp. If missing, the extension .qrl is added automatically and the run-log is saved compressed. Otherwise the extension .qrl or .qrz determines compression. Even without this argument a run-log is created unless suppressed with [-nolog^{\(577\)}](#) or when a report is generated. The default value is %p%b.qrz. In calldaemon mode, a run-log will be stored only if a (local) filename is specified.

-runlogdir <directory>

In interactive mode this argument overrides the option [Directory for run-logs^{\(301\)}](#) at a special layer for command line arguments. If specified, interactively changing the option has no effect whereas changing it at script level is still possible. In batch mode this directory serves as the target base directory for saving run-logs unless the filename of the run-logs specified with [-runlog \[<file>\]^{\(582\)}](#) is an absolute path. If this argument is given and a report is generated, the files in the report will be laid out according to the structure of the run-logs relative to this directory. The directory name may contain placeholders as explained in [section 32.2.4^{\(587\)}](#).

-runtime

Use a runtime license only. In batch mode, QF-Test will normally use a runtime license (or multiple runtime licenses if [-threads <number>^{\(586\)}](#) is given). If not enough free runtime licenses are available, full development licenses will be used instead unless [-runtime](#) is given in which case no development license is used and QF-Test will fail with an error message. In interactive mode, if [-runtime](#) is given, QF-Test will use a runtime license instead of a full development license. In that mode, any test-suite can be loaded and tests can be run interactively as usual, including debugging support. Saving of test-suites will be disabled, however, though test-suites can be modified for temporary experiments.

-shell <executable>

The shell to use when executing a [Execute shell command](#)⁽⁴³⁶⁾ node. Default for Unix is `/bin/sh`, for Windows `COMMAND.COM` or `cmd.exe`.

-shellarg <argument>

The argument that causes the shell specified with [-shell <executable>](#)⁽⁵⁸³⁾ to execute the following argument and then exit. For Unix shells this is typically `-c`, `COMMAND.COM` and `cmd.exe` expect `/c`. If you have Unix tools installed on Windows and specify `sh` or `bash` as the shell to use, don't forget to change [-shellarg <argument>](#) to `-c`.

-serverhost <host>

Set the hostname for the machine QF-Test is running on. This is passed to the SUT and usually determined automatically. You may need to set it when running QF-Test and the SUT on different machines and you experience troubles with reverse name lookup.

-sourcedir <directory> (batch mode only)

If this argument is given and a report is generated, the files in the report will be laid out according to the structure of the test-suites relative to this directory unless [-runlogdir <directory>](#)⁽⁵⁸²⁾ is also specified. In any case, the directory of a test-suite listed in the report will only be listed if this argument is specified and the test-suite is located below this directory.

-splitlog (batch mode only)

In batch mode split run-logs (see [section 8.1.4](#)⁽⁷²⁾) are enabled by default and can be turned off via `-splitlog=false`. If `-splitlog` is explicitly specified without parameter, the default extension for run-logs is changed from `.qrz` to `.qzp` so as to create split run-logs in ZIP format. The same can be achieved by specifying the name of the run-log explicitly with the desired extension.

-startclean (calldaemon mode only)

With this argument all contexts of the shared TestRunDaemon are cleaned up and released before running the test.

-startsut (only for internal use)

This argument is used to start a client application on a remote host. You should

not work with this argument directly. The standard library `qfs.qft` contains a procedure `qfs.daemon.startRemoteSUT` which can be use for this purpose.

-stopclean (calldaemon mode only)

With this argument all contexts of the shared `TestRunDaemon` are cleaned up and released after running the test.

-stoprun (calldaemon mode only)

Stop a running test executed by the Daemon on the given host and port. This argument can be combined with `-cleanup(574)` or `-stopclean(584)`.

-suitedir <dir> (calldaemon mode only)

Specify the (remote) directory where the QF-Test Daemon looks for test-suites. Use an absolute path when specifying the test to execute, if this argument is not given.

-systemcfg <file>

Set the location of the system configuration file (see [section 1.5^{\(7\)}](#)).

-tempdir <directory> (interactive mode only)

Can be used to specify a temporary directory which is needed on Windows only for the context sensitive help system. By default the values of the environment variables `TEMP` and `TMP` are tried.

-terminate (calldaemon mode only)

Use this option, if you want to terminate a running QF-Test daemon.

-test <n> | <ID> (batch mode only)

Without this argument all the top-level tests of the suite are executed one after the other. Using `-test <n> | <ID>`, you can select specific tests. An arbitrary node located anywhere in a test-suite can be accessed by its QF-Test ID⁽³³³⁾. Test-case⁽³¹⁴⁾ or Test-set⁽³²¹⁾ nodes can be referenced also by its qualified name. Tests on the top level of the suite can also be selected by their index, the index for the first Test being 0. You can use this argument multiple times, even for the same test.

-testdoc <directory> (batch mode only)

With this argument QF-Test creates both HTML and XML testdoc documentation. If no directory is given, it is created from the basename of the suite.

-testdoc.doctags (batch mode only)

Whether to use the QFS doctag extensions when creating the testdoc documentation. Default is true, use `-testdoc.doctags=false` to disable.

-testdoc.followcalls (batch mode only)

By default QF-Test ignores Test call⁽³²⁷⁾ nodes during testdoc creation. With this argument the target Test-case, Test-set or whole test-suite are processed as if they were part of the original test-suite. Thus it is possible to create a subset testdoc documentation by creating a dedicated test-suite with Test calls to the required parts.

-testdoc.html <directory> (batch mode only)

With this argument QF-Test creates HTML testdoc documentation. If no directory is given, it is created from the basename of the suite.

-testdoc.nodeicons (batch mode only)

Whether to show icons for nodes in the testdoc documentation. Default is true, use `-testdoc.nodeicons=false` to disable.

-testdoc.passhtml (batch mode only)

Whether to pass HTML tags in comments through to the HTML testdoc. Default is true, use `-testdoc.passhtml=false` to disable.

-testdoc.sorttestsets (batch mode only)

Whether to sort test-sets alphabetically. Default is true, use `-testdoc.sorttestsets=false` to disable.

-testdoc.sorttestcases (batch mode only)

Whether to sort test-cases alphabetically. Default is true, use `-testdoc.sorttestcases=false` to disable.

-testdoc.teststeps (batch mode only)

Whether to list test-steps in the testdoc documentation. Default is true, use `-testdoc.teststeps=false` to disable.

-testdoc.xml <directory> (batch mode only)

With this argument QF-Test creates XML testdoc documentation. If no directory is given, it is created from the basename of the suite.

-threads <number> (batch mode only)

Run the same test-suite in a number of parallel threads. Typically used for the purpose of load testing. One license is required per thread, so normally `-runtime(582)` should be specified as well. See [chapter 21^{\(197\)}](#) for more information about load tests.

-timeout <milliseconds> (calldaemon mode only)

Give a timeout value in milliseconds for the test run when executing a test through the QF-Test daemon. Default is infinite.

-usercfg <file> (interactive mode only)

Set the location of the user configuration file (see [section 1.5^{\(7\)}](#)).

-variable <name>=<value>

To override a system or suite variable definition (see [chapter 7^{\(57\)}](#)) use this argument to set the variable named <name> to the value <value>. Using this argument multiple times you can define more than one variable.

-verbose [<level>]

Print progress and status information during a test-run to the console. This is in particular useful when driving a test via `-calldaemon`, because the actual test usually runs on a different host where you may not be able to observe it. Note that on Windows you need to use `qftestc.exe` (instead of `qftest.exe`) to see the output. Specifying a level of verbosity is optional, possible values are `all` (print all nodes) and `tests` (default, only Test-set, Test-case and Test nodes are printed out). Each level can be combined with `errors` (print error and exception messages) like `tests,errors`.

-version

Print version information and exit.

32.2.4 Placeholders in the filename parameter for run-log and report

The filename given in any of the command line arguments `-runid <ID>`⁽⁵⁸²⁾, `-runlog [<file>]`⁽⁵⁸²⁾, `-runlogdir <directory>`⁽⁵⁸²⁾, `-report <directory>`⁽⁵⁷⁹⁾, `-report.html <directory>`⁽⁵⁸⁰⁾, `-report.name <name>`⁽⁵⁸⁰⁾, `-report.xml <directory>`⁽⁵⁸¹⁾ or `-report.junit <directory>`⁽⁵⁸⁰⁾, may contain placeholders of the form %X or +X (the latter must be used on Windows where '%' is a special character) where X may be any of the characters listed in the table below. QF-Test will fill in the respective value when creating the run-log or report. All time values refer to the time the test was started.

Note

When executing multiple test-suites, be sure to include the base name of the suite in the filename by specifying %b. Otherwise only a single run-log or report may be written that represents only the test-run of the last test-suite.

Character	Replacement
%	Literal '%' character.
+	Literal '+' character.
i	The current runid as specified with <code>-runid <ID></code> ⁽⁵⁸²⁾ .
p	The directory of the test-suite relative to <code>-sourcedir <directory></code> ⁽⁵⁸³⁾ . Expands to the absolute directory in case <code>-sourcedir <directory></code> is unspecified and is empty if <code>-sourcedir <directory></code> is specified but the test-suite is not located below it.
P	The absolute directory of the test-suite. May only be given at the beginning.
b	The basename of the test-suite, exclusive directory and .qft extension.
r	The return value (or "exit code") of the test-run (<code>-runlog</code> only).
w	The number of warnings in the test-run (<code>-runlog</code> only).
e	The number of errors in the test-run (<code>-runlog</code> only).
x	The number of exceptions in the test-run (<code>-runlog</code> only).
y	The current year (2 digits).
Y	The current year (4 digits).
M	The current month (2 digits).
d	The current day (2 digits).
h	The current hour (2 digits).
m	The current minute (2 digits).
s	The current second (2 digits).

Table 32.1: Placeholders in filename parameters

So if, for example, you want to save the run-log in a subdirectory of your test-suite directory called `logs` and want to include a timestamp and the exit code, use

```
-runlog %p/logs/%b-%y%M%d-%h%m%s-%r.qrl
```

Note

It is possible to use %b, %p and %P for collective parameters like runid or report. This makes sense only when processing a single test-suite. When processing multiple test-suites, the name of the first test-suite is used.

32.3 Exit codes for QF-Test

When run in interactive mode, the exit code of QF-Test is not very useful. It is either negative if QF-Test fails to start or 0.

In batch mode however the exit code expresses the result of the test run. Negative values represent fatal errors that prevent the test from being executed, while positive values stand for errors during the test run. Note that many systems only support exit codes between 0 and 255, so every exit code may have to be calculated modulo 256, i.e. -1=255, -2=254 and so on.

The following exit codes are currently defined:

Value	Meaning
0	Everything OK
1	Warnings occurred during the test run
2	Errors occurred during the test run
3	Exceptions occurred during the test run
-1	Unexpected Exception
-2	Bad command line arguments
-3	Missing or invalid license
-4	Errors while setting up the RMI connection
-5	Errors while loading the test-suite
-6	The test-suite doesn't contain any tests

Table 32.2: Exit codes for QF-Test

Besides, there are special exit codes when running QF-Test with the `-calldaemon` argument:

Value	Meaning
-7	Daemon could not be found
-8	Failed to get or create a TestRunDaemon
-9	Failed to get or create a run context
-10	The test could not be started
-11	The test has not ended within the given timeout

Table 32.3: `calldaemon` exit codes for QF-Test

Chapter 33

GUI engines

Swing, JavaFX, SWT can be combined together in a single application not only by using top-level windows of different technologies but also by embedding components of one technology into windows of another. QF-Test supports testing such kinds of applications.

4+

Also web pages can be integrated into Java applications by use of embedded browsers, e.g. JavaFX's `WebView` component of `JxBrowser`. QF-Test provides support for a number of such hybrid combinations.

To that end, the concept of a *GUI engine* was introduced. One GUI engine is responsible for handling recording and replay for one GUI toolkit thread. Normal applications have only one such thread. As explained above, combinations of Swing, JavaFX and SWT are possible that have one thread each and will thus require two GUI engines to operate in parallel. In theory it is also possible to have multiple GUI engines of the same kind, e.g. by creating multiple instances of the `SWT Display` class.

Each QF-Test GUI engine is identified by a token for the GUI toolkit and a number. `awt0`, `fx0` and `swt0` are the primary GUI engine for AWT/Swing, JavaFX and SWT. Unless you have a *very* special application you will never need to concern yourself with the number of the engine, as there will never be an engine called `awt1`, `fx1` or `swt1` and the alias `awt,fx` or `swt` is sufficient. When recording, QF-Test always uses the latter.

Note

If your application is either AWT/Swing only , JavaFX only or SWT only, engine names can be left empty.

Typically engine identifiers are automatically set correctly during replay. Only when inserted by hand they need to be considered. In a test-suite, engine identifiers are now stored in the following places.

- Wait for client to connect⁽⁴³⁸⁾ nodes. Only required if your application combines AWT/Swing, JavaFX and/or SWT. By specifying the engine attribute you can wait for the respective GUI engine to become initialized.

- Window⁽⁵³⁵⁾ nodes. The engine of a Window node marks the window and all its child nodes as being either AWT/Swing, JavaFX or SWT. Embedded components of the other kind will be moved to a node for a pseudo window.
- SUT script⁽⁴¹⁹⁾ nodes. An SUT script is executed on the event dispatch thread of the SUT, so for combined AWT/Swing, JavaFX and/or SWT applications the engine is required to specify whether the script should be run on the AWT/Swing, JavaFX or the SWT thread. Thus, an SUT script node can only retrieve and interact with components of one kind.
- File selection⁽⁴⁶⁶⁾ nodes. For Swing applications, the File selection node is rarely used because the Swing `JFileChooser` is implemented in Java and can be fully controlled by QF-Test. The SWT `FileDialog` on the other hand is similar to the AWT `FileChooser`. Both are implemented natively and QF-Test has no control over the individual controls. Also the JavaFX `FileChooser` needs special handling. Thus file selection must be replayed using the File selection node. Because this node is not explicitly associated with a Component or Window node, the engine has to be specified within the node.

Chapter 34

Running an application from QF-Test

Note

The Quickstart Wizard⁽²⁵⁾ is the recommended tool to set up your SUT for testing. It results in an advanced setup sequence already prepared for later requirements.

This chapter contains some details in case you want to create a setup sequence by hand.

34.1 Various methods to start the SUT

With the Quickstart Wizard QF-Test offers a utility to guide you through the steps of creating a start sequence for your SUT. Please refer to chapter 3⁽²⁴⁾ for more information about the Quickstart Wizard.

Nevertheless we also want to describe how to create a start sequence for your application manually. There are basically two ways to start a Java application as an SUT from QF-Test. The first one is based on the standard `java . . .` command line with its two variants for either starting a class or a jar file. The alternative is running a script or executable file which will then start the Java program. Indirect methods like launching the SUT through `ant` also fall into this category, as do Java WebStart and applets run in the Java plugin of a web browser.

The following examples show some typical setups. To get more detailed information about the required attributes, please follow the respective links to the reference manual. The tutorial also includes a number of examples.

Independent of how the SUT is started, the respective node should typically be followed immediately by a Wait for client to connect⁽⁴³⁸⁾ node with an identical Client attribute. Again, see the reference manual for further details.

34.1.1 A standalone script or executable file

If your application is started through a script or a binary executable, create a Start SUT client⁽⁴²⁶⁾ as follows:

Figure 34.1: Starting the SUT from a script or executable

- Create a Start SUT client⁽⁴²⁶⁾ node.
- Assign a name to the client in the Client⁽⁴²⁶⁾ attribute.
- Set the Executable⁽⁴²⁷⁾ to the script or executable that starts your application. If the program is not located in a directory on the `PATH`, the full path is required.
- Set the Directory⁽⁴²⁷⁾ attribute to the working directory for your application.
- One thing to watch out for in scripts is redirection of the standard output and error streams (e.g. `>myapp.log`) which you may want to remove so that the output

of the SUT reaches QF-Test and is captured in the run-log. Similarly, the `start` command in Windows batch files causes the SUT to detach and keeps the output away from QF-Test.

34.1.2 An application launched through Java WebStart

Using the new connection mechanism, an application launched through Java WebStart can be started directly from QF-Test without the need to modify any JNLP files (**so do not use** `Extras→Create WebStart SUT client starter...`). Instead create a Start SUT client⁽⁴²⁶⁾ node as follows:

The screenshot shows the 'Start SUT client' dialog box. It contains the following fields and sections:

- Client:** A text field with the value 'SUT'.
- Executable:** A text field with the value 'javaws'.
- Directory:** An empty text field.
- Parameters:** A section with two tabs: 'Executable (1)' and 'Extra (0)'. The 'Executable (1)' tab is active, showing a list of parameters. The first parameter is 'http://java.sun.com/products/javawebstart/apps/draw.jnlp'.
- QF-Test ID:** An empty text field.
- Delay before (ms):** An empty text field.
- Delay after (ms):** An empty text field.
- Comment:** A text area containing the text 'Start SUT through Java WebStart'.

Figure 34.2: Starting the SUT through Java WebStart

- Create a Start SUT client⁽⁴²⁶⁾ node.
- Assign a name to the client in the Client⁽⁴²⁶⁾ attribute.

- Set the Executable⁽⁴²⁷⁾ attribute to the Java WebStart executable which is typically called `javaws` and located somewhere inside the JDK or JRE. You'll probably have to specify the full path.
- For Java WebStart the Directory⁽⁴²⁷⁾ attribute typically is of no consequence except that Java WebStart is looking in that directory for a file named `.javaws` which can contain settings like debug levels.
- Create an entry in the Parameters⁽⁴²⁷⁾ for the executable to specify the URL for the application's JNLP descriptor.

34.1.3 An applet in a web browser

Applet testing can be done directly in a browser.

Note

In case you own a license that allows web testing with QF-Test and the supported browsers suffice you can also use the "normal" web testing feature (Quickstart your application⁽²⁴⁾).

Start SUT client

Client
SUT

Executable
mozilla

Directory

Parameters
Executable (1) Extra (0)

Executable parameters
Parameter
http://java.sun.com/docs/books/tutorial/uiswing/misc/IconI

QF-Test ID

Delay before (ms) Delay after (ms)
 Delay after (ms)

Comment
Start web browser with an applet page

Figure 34.3: Starting the SUT as an applet in a web browser

- Create a Start SUT client⁽⁴²⁶⁾ node.
- Assign a name to the client in the Client⁽⁴²⁶⁾ attribute.
- Set the Executable⁽⁴²⁷⁾ attribute to the browser that you want to start, i.e. `firefox`, `iexplore`, `chrome`, `safari` to name just a few. You may have to specify the full path.
- For starting of an applet, the Directory⁽⁴²⁷⁾ attribute is of no consequence.
- Create an entry in the Parameters⁽⁴²⁷⁾ for the executable to specify the URL for the applet's web page.

Note**Important notes on applet testing**

Applet testing will only work in a reliable way when

- a new browser process is started,
- no dialogs (like questions on session restore or default browser setting) will block the browser start,
- on Windows 8 the browser is started on the desktop, not as a Modern UI (former Metro) application.

The procedure `qfs.web.browser.general.startAsNewProcess` that is also used when creating the start sequence with the Quickstart Wizard⁽²⁵⁾ should solve the first two points. It uses browser specific parameters to ensure a new process when possible and avoids blocking dialogs e.g. by using new empty profiles. Please see the procedure documentation for further details.

Note Browser or the Java plugin itself might warn when detecting a new or an outdated JRE. In case you cannot or don't want to always test with the latest plugin, you might want to avoid those blocking dialogs. The JRE should provide respective option in its settings. For Google chrome you can use the additional command line parameters `-always-authorize-plugins` and `-allow-outdated-plugins`. For Firefox some hints are given at <http://mzl.la/MN0qiT>.

Windows Concerning Windows 8 there are following means to ensure Internet Explorer is started as desktop application:

- Either set another browser as default browser. This should force IE to desktop mode.
- or within the IE settings set the option Internet options→Programs→Choose how to open links to "Always in Internet Explorer on the Desktop"
- or directly change set the registry key

```
HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\
Main\AssociationActivationMode
```

to value 2.

0 = Let Internet Explorer decide

1 = Always in Internet Explorer (this is Metro IE)

2 = Always in Internet Explorer on the desktop

Further background information

As described above, the `qfs.web.browser.general.startAsNewProcess` helps to ensure a new process. Hence some hints should be given on some of the techniques used. You can have a look the procedures code as well, of course.

Windows

There are different command line parameters for different Internet Explorer versions to force a new process is being started:

- IE6: -new
- IE7: -new (default)
- IE8 and later: -noframemerging

(See also <http://msdn.microsoft.com/en-us/library/hh826025%28v=vs.85%29.aspx>). There is also the option to ensure this behavior be respective settings in the registry but using the commandline parameters is the less invasive way.

Note

Firefox has a built-in session restore feature that causes conflicts with QF-Test. When QF-Test terminates a Java applet and thus the browser, Firefox treats this as a crash and offers to restore the session on the next start through a dialog that cannot be controlled by QF-Test so that the test cannot continue unobserved.

To work around this problem, either always start with an empty new profile or create a separate user profile as described at http://kb.mozillazine.org/Profile_Manager and always start Firefox from QF-Test with this profile using the command `firefox -P <profile>`. Then disable session restore for this user by editing the preferences as described at http://kb.mozillazine.org/Editing_configuration. You need to add an option called `browser.sessionstore.resume_from_crash` and set it to `false`.

34.1.4 An application started with `java -jar <archive>`

If your application is normally launched through a command of the form `java -jar <archive>`, create a Start Java SUT client⁽⁴²²⁾ node as follows:

Start Java SUT client

Client
SUT

Executable
\${qftest:java}

Directory
.../application/directory

Class name

Parameters

Executable (2) \ Class (0) \ Extra (0)

Executable parameters

Parameter
-jar
theapp.jar

QF-Test ID

Delay before (ms) Delay after (ms)

Comment
Start SUT as java -jar theapp.jar

Figure 34.4: Starting the SUT from a jar archive

- Create a Start Java SUT client⁽⁴²²⁾ node.
- Assign a name to the client in the Client⁽⁴²³⁾ attribute.
- If necessary, change the Executable⁽⁴²³⁾ attribute. Its default value `${qftest:java}` is the `java` executable that QF-Test was started with.
- Set the Directory⁽⁴²⁴⁾ attribute to the working directory for your application.
- Create two entries in the Parameters⁽⁴²⁴⁾ table for the executable. Set the first to `-jar` and the second to the name of the archive. Unless the archive is located in the Directory⁽⁴²⁴⁾ selected above, its full path is required.

34.1.5 An application started with `java -classpath <classpath> <class>`

If your application is normally launched through a command of the form `java -classpath <classpath> <class>`, create a Start Java SUT client⁽⁴²²⁾ node as follows:

Figure 34.5: Starting the SUT via the main class

- Create a Start Java SUT client⁽⁴²²⁾ node.
- Assign a name to the client in the Client⁽⁴²³⁾ attribute.
- If necessary, change the Executable⁽⁴²³⁾ attribute. Its default value `${qftest:java}` is the java executable that QF-Test was started with.

- Set the Directory⁽⁴²⁴⁾ attribute to the working directory for your application.
- Set the Class name⁽⁴²⁴⁾ attribute to the fully qualified name of the application's starter class (the class with the `main()` method), just like for `java`.
- Create two entries in the Parameters⁽⁴²⁴⁾ table for the executable. Set the first to `-classpath` and the second to the list of jar files and directories that constitute the classpath. The full path is required for jar archives not located in the Directory⁽⁴²⁴⁾ selected above. This argument can get very long and hard to edit directly in the table. See section 2.2.5⁽¹³⁾ about how to pop up a dialog for more convenient editing.

34.1.6 A web application in a browser

Web Note

Currently web testing with QF-Test requires a 32bit Browser. Since there is little need for more than 4GB of memory in a Browser, 64bit Browsers are still the exception but on 64bit Linux systems the default browser may be 64bit in which case you need to install an additional 32bit Firefox. Also, the default JDK/JRE on 64bit Systems is often a 64bit one. In that case you must install an additional 32bit JDK or JRE and run QF-Test with that. To do so, either make the 32bit JDK the default when installing QF-Test or via the Java configuration dialog, or run QF-Test from the command line with `-java <executable>`⁽⁵⁷²⁾. If you have a 32bit browser and a 32bit JDK and the browser still won't start from QF-Test, please check if the JDK is an original from Sun or an openJDK.

Like Swing, JavaFX or SWT clients, a web-based SUT - i.e. a browser - is started as a separate process from within QF-Test. In order to gain access to the internals of the browser and the web page shown with its *Document Object Model* (DOM), QF-Test embeds a standard browser like Internet Explorer or Mozilla in a special wrapper application. The technology for embedding and accessing those standard browsers enables efficient access to the DOM beyond the browsers' standard interfaces and a unified interface that hides browser differences and enables QF-Test - and thus you - to focus on test automation with a single set of tests for all supported browsers on multiple platforms.

Start/open browser

Client
SUT

URL
www.qfs.de

☒ Open new browser window in existing client

Browser type
ie

Directory of browser installation

Name of the browser window
mainwin

Geometry of browser window

X	Y
Width	Height

☒ Executable

Executable
\${qftest:java}

Parameters

Java VM (0) Extra (0)

Executable parameters

Parameter

QF-Test ID

Delay before (ms) Delay after (ms)

Comment
Start browser with a given URL.

Figure 34.6: Starting a web-based SUT in a browser

A Start browser⁽⁴²⁸⁾ node can be used to launch a browser. One SUT web client process can display multiple browser windows, which can even be a mix of Internet

Explorer and Mozilla windows. Additional browser windows in an already running process can be opened via a Start browser node with the attribute Open new browser window in existing client⁽⁴³¹⁾ set or as the result of clicking a link in a web page that opens a popup window.

Note

QF-Test works with 32bit Firefox from version 3.6. On a 64bit machine you must install a 32bit variant of the browser you want to test with.

Note

When setting up the startup sequence with the Quickstart Wizard⁽²⁵⁾ or defining your own Directory of Mozilla installation⁽⁴³¹⁾ attribute, try pointing QF-Test to a current Firefox installation. If you get a message saying that the *Gecko Runtime Environment* (GRE) cannot be determined, please make sure that you have got a 32bit browser installed. On Linux, the standard browser for your distribution may be installed in various places.

Chapter 35

JDK and SWT instrumentation

35.1 JDK instrumentation

Swing

Note

QF-Test connects to AWT/Swing applications automatically unless you explicitly turn off the option Connect without JDK instrumentation (Swing)⁽²⁷⁵⁾ or use a non-standard JDK. In such a special case you must instrument the JDK first.

To instrument a JDK you need write permission for some of its directories, so you may have to run QF-Test with administrator privileges to perform this step. The accessibility interface and all the technical details of the modifications to the JDK are described in detail in chapter 35⁽⁶⁰⁴⁾. This section concentrates on how to instrument or de-instrument JDKs and how QF-Test maintains information about which JDKs have been instrumented.

Having started QF-Test, please select the menu item Extras→Manage JDK instrumentation... to bring up the following dialog:

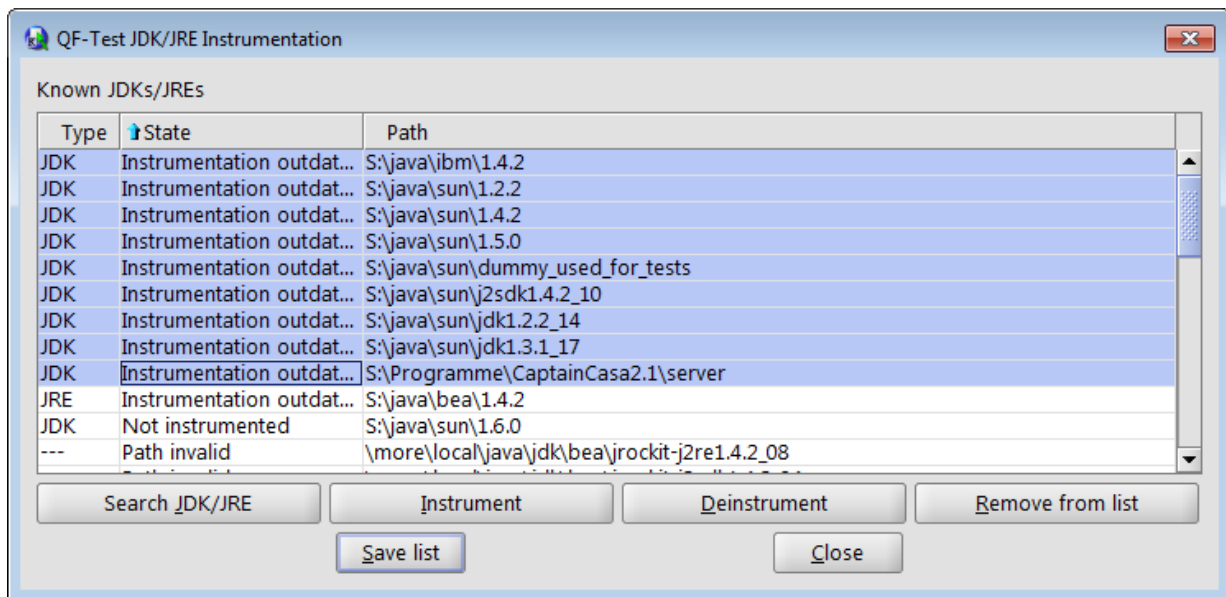


Figure 35.1: Dialog for JDK instrumentation

Note

If you get a warning that the file used for storing information about JDK instrumentation is currently locked, this typically means that some other user working with the same QF-Test installation is currently instrumenting JDKs. In that case, try again a little bit later. If the lock doesn't go away even though nobody seems to be working on the instrumentation, you can ignore it.

The instrumentation dialog shows a list of JDKs and JREs which QF-Test "knows" about. Initially the list will only contain the JDK with which QF-Test itself is running. If your SUT will be started with a different JDK or JRE you need to tell QF-Test about it. To do so, use the "Search JDK/JRE" button to bring up the standard file selection dialog and choose a directory. QF-Test will search that directory and its direct or indirect sub-directories and add all JDKs it finds to the list.

If you don't know which JDK your SUT is going to run with, first instrument the one JDK that's always shown, then try if it works. If it doesn't work, the record button  will remain deactivated when you start the SUT from QF-Test. In that case you can either ask your developers or search for the JDK or JRE yourself. On Windows, C:\Program Files\Java is the most likely candidate. On Unix systems locations vary. Possible candidates are /usr/lib/java, /usr/share/java, /usr/local/java, etc. It is also possible that your application comes with its own JRE so letting QF-Test search the directory where the SUT is installed may also turn up something.

The 'Type' displayed in the first column of the list is usually JDK or JRE, '—' denotes an invalid path. This type information is only given for completeness and you don't really need to concern yourself with it.

The value in the 'State' column is important. The values in this column can be any of the following:

Not instrumented

The JDK is in its initial, unmodified state.

Instrumentation current

The JDK has been instrumented and the installed jar file corresponds the current QF-Test version.

Instrumentation outdated

The JDK has been instrumented but the current QF-Test version has a newer version of the installed jar file, so the JDK should be instrumented again.

Instrumentation inconsistent

The JDK has been partly instrumented, either because something has gone wrong during (de)instrumentation or because somebody (or some other tool) has modified the JDK. You should instrument or de-instrument such a JDK as required.

Path invalid

Either the JDK no longer exists, in which case you can remove it safely from the list, or it belongs to some other machine as explained below, in which case you should ignore it.

To instrument, de-instrument or remove one or more JDKs from the list, select the entry or entries in the list and press the respective button. When done, save the list and close the dialog.

The list of JDKs QF-Test knows about is stored in the file `qfconnect.properties` in QF-Test's root directory. If QF-Test is installed on a network file system which is shared by multiple computers, possibly with different operating systems, this single file will be shared by all systems. As you can see in the image above, this is not a problem. When you bring up the dialog, QF-Test will check all the saved JDKs to see whether they exist on the current system. If not, they will be marked with the state "Path invalid" and can be safely ignored. In short, you can run the instrumentation first from a Windows system and instrument your Windows JDKs, then from a Unix system to instrument those JDKs and that information can safely coexist.

In case you don't have write permissions for this file, QF-Test will not be able to save the list of instrumented JDKs, but that shouldn't cause any trouble with the instrumentation itself.

35.2 JDK instrumentation background

Introduced in QF-Test version 1.07, instrumentation of the JDK or JRE on which the SUT will run is now the primary method of providing a way for QF-Test to interact with the SUT. This chapter explains the technical details of what's going on behind the scenes when the JDK is instrumented and when the SUT is started in an instrumented JDK.

3.1.4+

Normally you don't need to instrument standard JDKs because QF-Test passes the same information to the SUT's JDK via environment variables. However, for special cases instrumentation may be required anyway and the information in this chapter is still valid.

35.2.1 Modifications to the JDK

When Instrumenting the JDK, QF-Test makes use of the official accessibility interface which is provided by Java for just this purpose. It can be used by accessibility and capture replay tools to interact with Java applications without those applications knowing about it and without requiring any changes to those applications.

To activate this interface, QF-Test creates or modifies the file `.../lib/accessibility.properties` in the JDK installation and adds the class `de.qfs.apps.qftest.start.Connector` to the property "assistive_technologies". This has the effect that this class will be instantiated whenever the AWT toolkit is initialized in any Java application or applet that is run in this Java VM.

To make sure that this class can always be found without modifications to the `CLASSPATH`, the file `qfconnect.jar`, which contains the `Connector` class, is placed in the Java extensions directory `.../lib/ext`.

When de-instrumenting the JDK, QF-Test first removes its entry from the `assistive_technologies` property in `accessibility.properties` so the `Connector` class will no longer be used. Next QF-Test tries to remove `qfconnect.jar` from the `.../lib/ext` directory. On Windows systems this will not work if QF-Test is running inside the same Java VM, so the file should be removed by hand. However, this is not essential because it won't affect the JDK in any way after `accessibility.properties` has been updated.

35.2.2 Effects of the instrumentation

When the class `de.qfs.apps.qftest.start.Connector` is instantiated during AWT toolkit initialization of a Java application, the first thing it does is check whether the application was launched by QF-Test, i.e. whether it is a *System Under Test* or a plain application. In the latter case, the connector terminates immediately and doesn't affect the application in any way.

In case of an SUT, the connector sets up a special class loader that will load the classes required for QF-Test's SUT wrapper, then set up and initialize that wrapper which will finally connect to QF-Test over RMI, just as the old wrapper used to.

This technique differs from the old technique of modifying the SUT's command line in a number of subtle ways:

- The SUT process is launched directly by QF-Test without interim `qfclient` and `java` envelopes.
- The classes of the QF-Test wrapper around the SUT are all loaded through the special class loader, so they are separated completely from the SUT's classes. So even if the SUT uses some software that QF-Test also uses and the required versions differ, this shouldn't cause any problems.
- The class loader automatically ensures that the QF-Test wrapper has all the necessary permissions. No explicit permission grants or changes to the policy file are required.
- This technique makes it possible to test applications launched directly from an executable file. Java WebStart based applications can be tested without the need to modify `jnlp` files and even applets that run in the Java plugin inside a web browser.

35.3 SWT instrumentation

Some special setup is required for testing SWT based applications with QF-Test/swt. Because SWT was not written with testability in mind, applications need to be run with a slightly modified SWT version to enable testing. In this version we have added the necessary hooks for event filtering and component tracking. The changes are transparent so that the behavior of an application is not changed, regardless of whether it is run inside or outside of QF-Test.

If you use QF-Test's Quickstart Wizard to create the start sequence for your SUT (see chapter 3⁽²⁴⁾), it will take care of SWT instrumentation as well. For those with an aversion to wizard dialogs, the manual way is described next.

The standard library `qfs.qft`, which is part of the QF-Test distribution and described in detail in the tutorial, contains a Procedure⁽³⁷⁸⁾ with which to perform the SWT instrumentation. It is named `setup` and located in the Package⁽³⁸⁵⁾ `qfs.swt.instrument`. Insert a Procedure call⁽³⁸¹⁾ node before the start node for your SUT in your setup sequence. Set its Procedure name⁽³⁸²⁾ attribute to `qfs.qft#qfs.swt.instrument.setup` and in the Variable definitions⁽³⁸³⁾ set the parameter `sutdir` to the installation directory of your application. The plugin parameter can be left empty except when you are testing an Eclipse/RCP application that does not follow the standard plugin directory layout. In that case you can specify the plugin file to instrument directly via the `plugin` parameter. That's all. In case you want to know what goes on behind the scenes, all manual steps are described further on in this section.

35.3.1 Preparation for manual SWT instrumentation

Supported architectures for SWT testing are 32 and 64 bit Windows and 32 and 64 bit Linux with Gtk. The instrumentation files are provided in directories called `.../qftest-4.0.8/swt/$ARCH/$VERSION` where `$ARCH` is either `win32`, `win32-64`, `linux-gtk` or `linux-gtk-64` and `$VERSION` is one of the supported SWT versions.

First you need to determine whether your application is a standalone SWT application or is based on eclipse. To do so, simply take a look at the directory structure of your application. If you find a directory called `plugins` containing a file called `org.eclipse.swt.win32.win32.x86_X.Y.Z.jar` (on Windows) or `org.eclipse.swt.gtk.linux.x86_X.Y.Z.jar` (on Linux), with `X.Y.Z` representing a version number like `3.2.0`, your application is based on eclipse. For a standalone SWT application you should find a file called `swt.jar`, typically inside a directory called `lib`.

35.3.2 Manual SWT instrumentation for eclipse based applications

Simply replace the SWT plugin jar with one instrumented by QF-Test. To create the instrumented plugin you must run the Procedure `qfs.qft#qfs.swt.instrument.setup` described above once with your original plugin (or a copy thereof) specified in the `plugin` parameter. QF-Test will create a backup copy of the original jar named `_org.eclipse.swt....jar.orig`. Next copy the instrumented plugin to the `plugin` directory of your application.

Finally, start your application once from the command line with the `-clean` command line argument to have it rebuild its plugin cache, e.g.

```
eclipse -clean
```

Your application's binary name may be different from `eclipse`, but all eclipse based applications should support the `-clean` argument.

35.3.3 Manual instrumentation for standalone SWT applications

For standalone SWT applications, replace the `swt.jar` file with the one provided with QF-Test. You may want to create a backup of the original first.

Note

If you are launching the client application by means of a Start Java SUT client⁽⁴²²⁾ node, you can set the classpath to point to the corresponding `.../qftest-4.0.8/swt/$ARCH/$VERSION/swt.jar` archive and leave your original file untouched.

Chapter 36

Technical details about miscellaneous issues

36.1 Drag&Drop

From the beginning Drag&Drop has been hard to implement in Java. JDK 1.1 had no Drag&Drop support at all and the first steps taken with JDK 1.2 were far from satisfying. In JDK 1.3 Drag&Drop has matured, but there are still problems with stability. That Drag&Drop is so hard to get right is partly due to the fundamental difference in the way Drag&Drop is implemented by the underlying systems. Another reason is the goal to support Drag&Drop between Java and the underlying system and not just between Java applications.

As a result Drag&Drop for Java is implemented "native", meaning at the level of the operating system, so the events involved can be neither interpreted, nor generated by QF-Test. Hence Drag&Drop cannot be replayed directly.

Even so QF-Test has support for Drag&Drop, at least for JDK 1.3 and above. Direct recording of Drag&Drop is supported with JDK 1.4 and above. Replay of Drag&Drop is achieved through the special Mouse events⁽⁴⁴⁷⁾ `DRAG_FROM` and `DROP_TO` as well as the optional `DRAG_OVER`. To replay these events, QF-Test makes use of the AWT Robot that was added to Java with version 1.3. The AWT Robot makes it possible to generate "hard" events at system level. These "hard" events actually move the mouse cursor across the screen and can trigger a Drag&Drop operation.

Using Drag&Drop together with the `[Shift]` and or `[Control]` key is also supported. To simulate a Drag&Drop with the `[Control]` key pressed, change the Modifiers⁽⁴⁴⁹⁾ attributes of the `DRAG_FROM`, `DRAG_OVER` and `DROP_TO` events to include the Control modifier by adding 2 to the current value. It is even possible to simulate the `[Shift]` or `[Control]` key being pressed or released during the drag operation by changing the Modifiers of only some of the events.

As mentioned above, Drag&Drop used to be a bit unstable on some systems. In some cases with older JDKs the use of the AWT Robot to simulate Drag&Drop could crash the SUT or even the whole system. Nowadays the situation is much better. The introduction of mouse movement interpolation helped improve the reliability of Drag&Drop replay a lot. Please see [section 29.3.5^{\(288\)}](#) for details.

Note

On Windows simulating Drag&Drop may conflict with some mouse drivers. In case of problems reduce the speed of the mouse cursor to 50% in the mouse settings of the control panel.

36.2 Component recognition

The process of identifying components in the SUT during a test run is rather complicated. The problem is that QF-Test has to be extremely flexible in order to accommodate a wide range of applications and to allow for a certain amount of change in the GUI of the SUT, due to dynamic behavior or changes between different versions of the SUT, while still being able to find the component intended by the developer of the test-suite.

This process can be simplified - and its reliability improved considerably - by defining unique names for most of the "interesting" components of the SUT. This has to be done by the developer of the application with the Java method `setName(String)`. If you can get your developers to set unique names for your components or windows, you won't have any problems with component recognition. Even drastic changes in the GUI can be handled if names are being used. Another advantage is that the name is language-independent, so you can use one and the same test-suite to test applications that support different languages. To ease the process of assigning names, QF-Test has a function to suggest names for those components where a name will improve testability. See the option [Hotkey for components^{\(261\)}](#) for further information.

To determine the target component for an event, QF-Test calculates match probabilities for the components of the SUT. The component with the highest probability is used, provided that probability is higher than a configurable bound. First the probabilities for the known windows are determined. Then the search continues among the components of the windows with sufficient probability.

This process is repeated for every hierarchy level of the component, meaning each direct or indirect parent node of the `Component(545)` node, but working from top to bottom. At every level all components matching the `Class name(546)` attribute are ordered by decreasing probability and then searched for the component on the next hierarchy level. Components that aren't visible are not considered.

The probability of a component is calculated as follows:

- Each component starts with a probability of 99%. If geometry information is avail-

able, deviations in position or size lead to a probability reduction.

- Then follow three tests that may either be skipped if no info is given, succeed or fail. A skipped test doesn't change the probability, a failed test reduces it below a given value, while a successful test raises it above another given value.
- The first test is structural and applies only to Component⁽⁵⁴⁵⁾, not to Windows⁽⁵³⁵⁾. All components within the currently searched parent container, whose class is the same as or derived of the given Class name⁽⁵⁴⁶⁾, are collected in a list (including invisible components). Then the Class count⁽⁵⁴⁹⁾ is compared to the size of the list and the Class index⁽⁵⁴⁹⁾ must match the position of the component in the list.
- The next test is for a Feature⁽⁵⁴⁷⁾ match, possibly including tests for Extra features⁽⁵⁴⁷⁾. If an Extra feature check of type 'must match' fails, the component is discarded.
- Finally the Name⁽⁵⁴⁷⁾ is tested. The facts that this is the final test and that the bonus and penalty values for this test default to the highest and lowest values, makes this the deciding test, but only if a name is given.

For dialogs there's an additional step to test its modality. Typically a dialog is either modal or non-modal, so by default a modal mismatch prevents recognition. However, sometimes the same dialog may be modal or non-modal depending on the context. If your SUT contains such a dialog, adjust the modal penalty to some value above the minimum value for component recognition.

If the final probability doesn't reach the minimum value, the component is discarded. Finally the component with the highest probability is used. If the component used had a structure, feature or name mismatch, a warning is issued, since this indicates that it may not be the right component after all. Most of the time this is just a sign that the SUT has changed slightly and the components should be updated before too many changes accumulate and start confusing the identification process.

Though this process is already dominated by the Name⁽⁵⁴⁷⁾ attribute, you can attach even more importance to it by setting the options Name override mode (replay)⁽²⁸⁵⁾ and Name override mode (record)⁽²⁶⁴⁾ to "Override everything". In that case QF-Test simplifies the search for a component by skipping the intermediate containers and going straight from the window to the component with the given name and class. The big advantage of this approach is, that the component will still be recognized, even if you introduce a new container in between the window and the component. The prerequisite for using this method is that you can guarantee that if a name is set on a component, it is going to be unique among the simultaneously visible components of the same class in one window.

If this kind of uniqueness is not possible, the next best setting for these options is "Hierarchical resolution". It requires that components with identical names at least have differently named ancestors. This setting still retains most of the advantages and flexibility of names but in this case recognition will break if a named component is moved from its parent to a differently named parent.

36.3 Timing

Besides component recognition, timing is an inherently difficult problem in automatic testing. No two runs of a Java program are identical when it comes to timing. Too much depends on things like system load or memory usage. This can lead to a situation where a target component for an event is not available, because the VM is still busy popping up the dialog window that contains it.

To avoid needless failures of tests, QF-Test combines several tactics:

- Events are synchronized with the AWT event queue, meaning that after each event sent to the SUT, QF-Test waits until the events generated as side effects have been processed, before it sends another event.
- In some cases, especially with asynchronous updates, this doesn't suffice, so whenever a component is not available, QF-Test waits for a certain amount of time to give it a chance to appear. This delay can be customized through the option Wait for non-existent component (ms)⁽²⁹¹⁾.
- An additional timeout defined by the option Wait for non-existent item (ms)⁽²⁹¹⁾ applies when looking for sub-items of a component, e.g. a tree node.
- These default timeouts should be kept rather short, a few seconds at most, so in addition you can set individual timeouts and delays wherever applicable.

36.4 Regular expressions

The regular expressions that you can use in the search and replace dialogs and in places like the Feature⁽⁵⁴⁷⁾ attributes, the Primary index⁽⁵⁵²⁾ of an Item⁽⁵⁵¹⁾ node or in checks⁽⁴⁶⁹⁾ all use standard Java regexp syntax. Before QF-Test version 3.1 the default was the GNU regexp package (see also appendix E⁽⁹¹⁰⁾). It can still be activated via the option Use old-style GNU regexps (from before QF-Test version 3.1)⁽²⁴¹⁾. One major difference between the two is that '{' and '}' are special characters that need to be escaped with '\\' for Java regexps, but normal characters for GNU:

Detailed regexp documentation with links to further information and even a whole book about regular expressions are provided in the Java documentation for the class `java.util.regex.Pattern` at <http://download.oracle.com/javase/1.5.0/docs/api/java/util/regex/Pattern.html>. It's also worth to have a look at the Wikipedia article about regular expressions.

Following is a short summary of the basics:

- A '.' stands for any one character except line breaks. With the new Java regexps you can start your regexp with the embedded flag '(?s)' to treat multi-line text like

a single line so `'.'` will match everything. The old GNU regexps have no such flag, so you need to use `'(.|\n)'` to match everything. Unfortunately this expression causes a `StackOverflowException` with Java regexps, so if QF-Test finds this expression in any regular expression it will treat it as a GNU regexp regardless of the option setting.

- Character in between '[' and ']' match any one of these characters.
- A '?' says the preceding element is optional, i.e. it may appear 0 or 1 times.
- '+' means at least one of the preceding element.
- '*' means 0 or more of the preceding element.
- A group is created with '(' and ')'. A '?', '+' or '*' after the closing brace refers to the whole group. All groups in a regexp are numbered in the order of their opening brace. The first group has the number 1, 0 stands for the whole regexp. For search and replace, `$n` in the replacement string expands to the part of the original value matched by the `n`th group. Example: To change the extension of file names starting with `/tmp/` from `.foo` to `.bar`, search for `(/tmp/.*)\.foo` and replace with `$1.bar`.
- A '|' separates alternatives in a group.
- '\' quotes to suppress the special meaning of the following character or introduces special characters, e.g. '\n' for LineFeed (a line break), '\r' for CarriageReturn (not needed for QF-Test, see [section 36.5^{\(616\)}](#)) or '\t' for Tab.

Examples:

- To match any text that contains the word 'tree' use `'.*tree.*'`.
- To match arbitrary text possibly including line breaks: `'(?:s).*'` for Java regexps and `'(.|\n)*'` for GNU.
- To replace 'tree' in arbitrary text with 'node' use `'(.*?)tree(.*?)'` to search and `$1node$2` to replace. In the replace dialog simply replace `tree` with `node` and disable the "Match whole string" check box to achieve the same effect.
- To search for 'name' or 'names': `'names?'`
- To search for 'tree' or 'node': `'(tree|node)'`
- An arbitrary word consisting of letters and numbers: `[0-9a-zA-Z]+`
- ...

4.0+

QF-Test allows you to use the context menu item Escape text for regular expressions on all attributes which allow regular expressions in order to escape special characters of regular expressions correctly with '\'.

36.5 Line breaks under Unix and Windows

The difference in the treatment of line breaks between Unix and Windows is a well-known problem. While Unix uses a single LineFeed character ('\n', hex 0x0A) as line separator, Windows uses the combination CarriageReturn/LineFeed ('\r\n', hex 0x0D0A). Java automatically converts text as needed which generally works well.

However, the XML standard specifies that an XML parser has to convert line breaks of any type into LineFeed only, regardless of the system under which it is running. This could lead to trouble, for example when checking a multi line text field. QF-Test works around the problem by converting all text strings read from the SUT to the Unix version with LineFeed only. This has the added benefit that tests created on one system will run unchanged on the other.

36.6 Quoting and escaping special characters

A common problem for most complex systems is the treatment of characters with a special meaning. A typical example are blanks in filenames. To specify such filenames on the command line, they need to be protected either by using double quotes or by escaping the blanks with a backslash character ('\').

Since QF-Test makes use of special characters in various contexts while reading arbitrary strings from the SUT that may contain any character, some kind of quoting mechanism is unavoidable. That QF-Test runs on various operating systems and makes use of regular expressions⁽⁶¹⁴⁾ which have their own set of special characters doesn't make things any easier. However, QF-Test attempts to keep things as simple as possible by restricting quoting to the places where it can't be avoided and by quoting all strings read from the SUT during recording correctly.

The most prominent special character for QF-Test is the '\$' sign used for variable expansion. Variable syntax is applicable in almost all attributes. If you need a literal '\$' character in an attribute value you have to double it.

Example: To verify with a Check text⁽⁴⁷⁰⁾ node that a text field contains the string "4 US\$", set the Text⁽⁴⁷³⁾ attribute to "4 US\$".

Other special characters are used only in a few places and must be quoted only there. These are the '#' character used for Procedure⁽³⁷⁸⁾ and Component⁽⁵⁴⁵⁾ access across suites

and the characters '@', '&' and '%' for the special syntax for sub-item access⁽⁵⁴⁾. Since these are used as separators they cannot be escaped by doubling them, so QF-Test follows the convention to use the backslash '\' as escape character which turns the backslash itself into another special character. To avoid quoting-hell with Windows file-names, QF-Test only uses quoting where the above characters are used and even there a single backslash that is not followed by a special character is interpreted literally.

To be precise, you have to escape the characters '#' and '\' in the Procedure name⁽³⁸²⁾ attribute of a Procedure call⁽³⁸¹⁾ and the characters '#', '\', '@', '&' and '%' in the attributes QF-Test component ID⁽⁴⁴⁸⁾ of events and checks as well as the Primary index⁽⁵⁵²⁾ and Secondary index⁽⁵⁵²⁾ of an Item⁽⁵⁵¹⁾. Remember that the backslash is also used as the escape character for regular expressions⁽⁶¹⁴⁾, so to get a literal '\' into a regexp for a sub-item, you first need to escape it for the regexp itself, i.e. "\\\\", then escape these for QF-Test leading to "\\\\\\\\".

There is one more special case that requires a special character and corresponding quoting. This is the '/' character used as separator for tree path sub-items of a `JTree` component. Thus the '/' must be quoted if and only if you need a literal / in a sub-item of a `JTree` component. Sub-items of other components don't require special handling.

36.7 Include file resolution

This is a section you should hopefully never need to read. It explains in detail how implicit Procedure and Component references are resolved during test replay. If you need to read it, your test-suite include hierarchy probably is too complicated and you should consider simplifying your includes.

There are basically two scenarios in which QF-Test must implicitly resolve a Procedure or Component reference to another suite when the requested Procedure or Component cannot be found in the current (or explicitly referenced) suite:

- The current suite includes other suites (by defining them in the Include files⁽³¹²⁾ attribute of the Test-suite⁽³¹¹⁾ root node). In this case, QF-Test searches all included suites in the given order.
- The current suite (or rather one of its Procedures) was called by another suite. Here QF-Test searches the calling suite for the requested node.

The whole thing gets complicated, when (possibly indirect) Procedure calls across test-suite boundaries and (possibly indirect, maybe even recursive) includes are combined. Following are detailed explanations of the search algorithm that will hopefully enable you to debug and resolve any include-file related problems.

- Whenever execution leaves the current suite to continue with some Procedure or to retrieve a Component, the other suite becomes the current suite. This process is complemented by two things: the old suite is pushed onto the so-called *call-stack* and the variable bindings of the new current suite are pushed on top of the *fallback bindings stack* (see [chapter 7^{\(57\)}](#)), so they override the bindings of the old suite. In the run-log this process is documented by adding a *Suite-change* node which holds all of the run-log nodes for the execution that takes place outside the old suite.
- Any search through test-suites starts with the current suite, then continues top-down through the call-stack. So if, for example, A calls B which calls C, then C is searched first, followed by B and finally A.
- Includes are considered stronger bindings than the call-stack. This means that during the search through the current suite and the suites on the call-stack, at each step the included test-suites are searched before moving to the next suite on the call-stack. For example, if A calls B which includes C, A is on the call-stack and B is the current Suite, then B will be searched first, then C, and lastly A.
- In case of multiple, possibly indirect includes, the search is always conducted depth-first in the order in which the include files are listed. This means that if A includes B and C, and B includes D, first A is searched, followed by B, then D and then C.
- If a Procedure is found (possibly indirectly) in an included test-suite (as opposed to the current suite, an explicitly referenced suite or a suite on the call-stack), the change from the old current suite to the new current suite doesn't take place in one step. This has to be illustrated with an example right from the start or we'll get totally lost: Let's say A calls B and that A includes C. B calls a Procedure which is found in C by way of A. Instead of changing suites directly from B to C, A will first become the current suite and then C. As a consequence, A gets pushed onto the call-stack again on top of B and its variable bindings are also pushed again on top of B's bindings on the fallback bindings stack. The reasoning behind this is that C, which is now the current suite, is "closer to" A, which includes C, than it is to B, which only happened to be called by A. One could also say that inclusion creates a kind of union, so that to B, A and C will always appear as a single test-suite as long as B doesn't call C explicitly.
- That's it, except for one thing: During a search QF-Test never searches a suite twice. This would be useless in any case, but it is more than an optimization, since it prevents trouble with recursive includes if A includes B and B includes A.

If you really have a problem determining how, why or why not a certain Procedure or Component was retrieved, first take a look at the run-log. It shows exactly which suites were used and which variable expansions took place.

Chapter 37

Jython and Groovy scripting

This section explains technical details about the Jython integration in QF-Test and serves as a reference for the whole API exposed by QF-Test for use in Jython and Groovy scripts. For a more gentle introduction including examples please take a look at [chapter 13^{\(99\)}](#).

37.1 Module load-path (Jython)

The load-path for Python/Jython modules, `sys.path`, is assembled from various sources in the following order:

- the directory `qftest/jython`
- the directory `qftest/qftest-4.0.8/jython/Lib`
- directories defined in the system property `python.path`

In addition, during Server script or SUT script node execution, the directory of the containing test-suite is prepended to the path.

The directory `qftest/qftest-4.0.8/jython/Lib` contains Jython-internal stuff, the complete Python library and the following modules provided by Quality First Software GmbH: `qfcommon.py`, `qftest.py` and `qfclient.py`. You should not modify these files, since they may change in later versions of QF-Test.

The directory `qftest/jython` is the place to put your own shared modules. These will be left untouched during an update of QF-Test. Modules that are specific to a test-suite can also be placed in the same directory as the test-suite. The file extension for all modules must be `.py`.

To add additional directories to the load-path, define the `python.path` system property.

37.2 The plugin directory

Jython and Groovy can also be used to access Java classes and methods beyond the scope of QF-Test by simply importing such classes, e.g.

```
from java.util import Date
from java.text import SimpleDateFormat
print SimpleDateFormat("yyyy-MM-dd").format(Date())
```

Example 37.1: Accessing Java classes from Jython

The classes available for import are those on the `CLASSPATH` during startup of QF-Test or the SUT respectively, all classes of the standard Java API and QF-Test's own classes. For the SUT things also depend on the `ClassLoader` concept in use. WebStart and Eclipse/RCP in particular make it difficult to import classes directly from the SUT.

Additionally, there are plugin directories into which you can simply drop a jar file to make it available to scripts. By default, the main plugin directory is called `plugin` (big surprise here) and is located under QF-Test's root directory `qftest`. This location can be overridden with the command line argument `-pluginindir <directory>`⁽⁵⁷²⁾.

Jar files in the main plugin directory are available to both Server script and SUT script nodes. To make a jar available solely to Server scripts or solely to SUT scripts, drop it in the respective sub-directory called `qftest` or `sut` instead.

37.3 The package cache (Jython)

In order to be able to import Java packages, Jython maintains a cache of package information. By default this cache is located under `qftest/jython/cachedir`. If this directory is not writable, `.qftest/jython-cachedir` under the user's home directory is used instead. The location of the directory can be overridden with the `python.cachedir` system property.

When running QF-Test for the first time after installing it, you may see a number of messages of the form

```
*sys-package-mgr*: processing...
```

Later you should see these messages only when jar-files on the `CLASSPATH` have been modified or new ones added. Sometimes Jython goes into "hiccup mode" and regenerates the cache for some jar-files every time QF-Test or the SUT is started. In that case, simply remove the whole `qftest/jython/cachedir` or `.qftest/jython-cachedir` and the problem should go away.

37.4 Initialization (Jython)

During QF-Test and SUT startup an embedded Jython interpreter is created. For QF-Test, the module named `qftest` is imported, for the SUT the module named `qfclient`. Both are based on `qfcommon` which contains shared code. These modules are required to provide the run-context interface and to set up the global namespace.

Next the load-path `sys.path` is searched for your personal initialization files. For QF-Test initialization, the file called `qfserver.py` is loaded, the file called `qfsut.py` is used for the SUT. In both cases `execfile` is used to execute the contents of these files directly in the global namespace instead of loading them as modules. This is much more convenient for an initialization file because everything defined and all modules imported will be directly available to Server scripts and SUT scripts. Note that at initialization time no run-context is available and no test-suite-specific directory is added to `sys.path`.

37.5 Namespace environment for script execution (Jython)

The environments in which Server scripts or SUT scripts are executed are defined by the global and local namespaces in effect during execution. Namespaces in Jython are dictionaries which serve as containers for global and local variable bindings.

The global namespace is shared between all scripts run in the same Jython interpreter. Initially it will contain the classes `TestException` and `UserException`, the module `qftest` or `qfclient` for QF-Test or the SUT respectively, and everything defined in or imported by `qfserver.py` or `qfsut.py`. When assigning a value to a variable declared to be global with the `global` statement, that variable is added to the global namespace and available to scripts run consecutively. Additionally, QF-Test ensures that all modules imported during script execution are globally available.

The local namespace is unique for each script and its lifetime is limited to the script's execution. Upon invocation the local namespace contains `rc`, the interface to QF-Test's run-context, and `true` and `false` bound to 1 and 0 respectively for better integration with QF-Test.

Accessing or setting global variables in a different Jython interpreter is enabled through the methods `fromServer`, `fromSUT`, `toServer` and `toSUT`.

37.6 Run-context API

The run-context object `rc` is an interface to the execution state of the currently running test in QF-Test. Providing this wrapper instead of directly exposing QF-Test's Java API leaves us free to change the implementation of QF-Test without affecting the interface for scripts.

Following is a list of the methods of the run-context object `rc` in alphabetical order. The syntax used is a bit of a mixture of Java and Python. Python doesn't support static typing, but the parameters are passed on to Java, so they must be of the correct type to avoid triggering exceptions. If a parameter is followed by an '=' character and a value, that value is the default and the parameter is optional.

Note

Please note that the Groovy syntax for keyword parameters is different from Jython and requires a ':' instead of '='. The tricky bit is that, for example, `rc.logMessage("bla", report=true)` is perfectly legal Groovy code yet doesn't have the desired effect. The '=' here is an assignment resulting in the value `true`, which is simply passed as the second parameter, thus the above is equal to `rc.logMessage("bla", true)` and the `true` is passed to `dontcompactify` instead of `report`. The correct Groovy version is `rc.logMessage("bla", report:true)`.

```
void addDaemonLog(byte[] data, String name=None, String
comment=None, String externalizename=None)
```

Add a run-log retrieved from a `DaemonRunContext` to the current run-log.

Parameters

data	The byte array retrieved via <code>DaemonRunContext.getRunLog()</code> .
name	An optional name for the daemon log node. If unspecified the ID of the Daemon is used.
comment	An optional comment for the daemon log node.
externalizename	An optional name to externalize the daemon log and save it as a partial log of a split run-log.

```
void addResetListener(ResetListener listener)
```

Server only. Register a `ResetListener` within the current run-context.

Parameters

listener	The Listener that should be added. The listener should implement the interface <code>de.qfs.apps.qftest.extensions.qftest.ResetListener</code> .

```
void addTestRunListener(TestRunListener listener)
```

Register a `TestRunListener` with the current run-context. In interactive mode and batch mode there is a single, shared run-context, so the listener will remain in effect until it gets removed via `removeTestRunListener` or `clearTestRunListeners`. In daemon mode, each `DaemonRunContext` has its own set of listeners. See [section 39.5^{\(728\)}](#) for details about the `TestRunListener` API.

Parameters

listener	The listener to register.

```
String callProcedure(String name, dictionary parameters=None)
```

Call a [Procedure^{\(378\)}](#) in a test-suite.

As a convenience, this method can also be called from an SUT script. Care should be taken however, because the script is executed inside the AWT event dispatch thread, so weird side-effects are possible, though QF-Test does its best to avoid these. If possible, call Procedures from a Server script instead.

Parameters

name	The fully qualified name of the Procedure.
parameters	The parameters for the Procedure. This should be a dictionary. Its keys and values can be arbitrary values. They are converted to strings for the call.

Returns	The value returned from the Procedure through an optional Return⁽³⁸⁴⁾ node.

```
int callTest(String name, dictionary parameters=None)
```

Server only. Call a [Test-case^{\(314\)}](#) or [Test-set^{\(321\)}](#) in a test-suite.

Parameters

name	The fully qualified name of the Test-case or Test-set.
parameters	The parameters for the Test-case or Test-set. This should be a dictionary. Its keys and values can be arbitrary values. They are converted to strings for the call.

Returns	The final state of the execution. Either <code>rc.OK</code> , <code>rc.WARNING</code> , <code>rc.ERROR</code> , <code>rc.EXCEPTION</code> , <code>rc.SKIPPED</code> or <code>rc.NOT_IMPLEMENTED</code> .

```
Boolean check(boolean condition, String message, int  
level=rc.ERROR, boolean report=true, boolean nowrap=false)
```

Check or "assert" that a condition is true and log a message according to the result.

Parameters

condition The condition to evaluate.

message The message to log. It will be preceded by "Check OK: " or "Check failed: " depending on the result. For the old-style XML or HTML report the message will be treated like a Check node if it starts with an '!' character.

level The error level in case of failure. The following constants are defined in the run-context:

- `rc.OK`
- `rc.WARNING`
- `rc.ERROR`
- `rc.EXCEPTION`

If the level is `rc.EXCEPTION`, a `UserException`⁽⁵⁶⁴⁾ will be thrown if the check fails.

report If true, the check will appear in the report. Only applicable if `level <= rc.WARNING`.

nowrap If true, lines of the message will not be wrapped in the report. Use for potentially long messages.

Returns The result of the check.

```
Boolean checkEqual(Object actual, Object expected, String  
message, int level=rc.ERROR, boolean report=true, boolean  
nowrap=false)
```

Check or "assert" that an object matches a given value and log a message according to the result. Comparison is done using the == operator.

Parameters

actual The actual value.

expected The expected value.

message The message to log. It will be preceded by "Check OK: " or "Check failed: " depending on the result. In case of failure, the expected and actual values will also be logged.

level The error level in case of failure. The following constants are defined in the run-context:

- `rc.OK`
- `rc.WARNING`
- `rc.ERROR`
- `rc.EXCEPTION`

If the level is `rc.EXCEPTION`, a `UserException`⁽⁵⁶⁴⁾ will be thrown if the check fails.

report If true, the check will appear in the report. Only applicable if `level <= rc.WARNING`.

nowrap If true, lines of the message will not be wrapped in the report. Use for potentially long messages.

Returns The result of the check.

```
Boolean checkImage(ImageRep actual, ImageRep expected, String
message, int level=rc.ERROR, boolean report=true, boolean
nowrap=false)
```

Check or "assert" two given `ImageRep` (see [section 39.8.1^{\(737\)}](#)) objects for equality and log a message according to the result. Comparison is done using the `equals` method of the `ImageComparator` (see [section 39.8.2^{\(740\)}](#)) of the expected object.

Parameters

actual	The actual value <code>ImageRep</code> object.
expected	The expected <code>ImageRep</code> object.
message	The message to log. It will be preceded by "Check OK: " or "Check failed: " depending on the result. In case of failure, the expected and actual values will also be logged. For the old-style XML or HTML report the message will be treated like a Check node if it starts with an '!' character.
level	The error level in case of failure. The following constants are defined in the run-context: • <code>rc.OK</code> • <code>rc.WARNING</code> • <code>rc.ERROR</code> • <code>rc.EXCEPTION</code> If the level is <code>rc.EXCEPTION</code> , a <code>UserException⁽⁵⁶⁴⁾</code> will be thrown if the check fails.
report	If true, the check will appear in the report. Only applicable if <code>level <= rc.WARNING</code> .
nowrap	If true, lines of the message will not be wrapped in the report. Use for potentially long messages.
Returns	The result of the check.

```
Object[] checkImageAdvanced(ImageRep actual, ImageRep expected,
String message, String algorithm, int level=rc.ERROR, boolean
report=true, boolean nowrap=false)
```

Check or "assert" two given `ImageRep` (see [section 39.8.1^{\(737\)}](#)) objects for equality and log a message according to the result. Comparison is done using the specified algorithm.

Parameters

actual	The actual value <code>ImageRep</code> object.
expected	The expected <code>ImageRep</code> object.
message	The message to log. It will be preceded by "Check OK: " or "Check failed: " depending on the result. In case of failure, the expected and actual values will also be logged. For the old-style XML or HTML report the message will be treated like a Check node if it starts with an '!' character.
algorithm	The algorithm to use for the comparison as described in chapter 44⁽⁷⁶⁹⁾ .
level	The error level in case of failure. The following constants are defined in the run-context:

- `rc.OK`
- `rc.WARNING`
- `rc.ERROR`
- `rc.EXCEPTION`

If the level is `rc.EXCEPTION`, a [`UserException\(564\)`](#) will be thrown if the check fails.

report If true, the check will appear in the report. Only applicable if `level <= rc.WARNING`.

nowrap If true, lines of the message will not be wrapped in the report. Use for potentially long messages.

Returns An array with following content:
 The result of the check as Boolean.
 The result of the check as probability.
 The transformed image of the expected image as `ImageRep`, depending on the algorithm.
 The transformed image of the actual image as `ImageRep`, depending on the algorithm.
 Further information where appropriate.

```
void clearGlobals()
```

Server only. Undefine all global variables.

```
void clearProperties(String group)
```

Server only. Delete a given set of loaded properties or resources.

Parameters

group	The group name of the properties or resources.

```
void clearTestRunListeners()
```

Remove all `TestRunListeners` from the current run-context.

```
String expand(String text)
```

Expand a string using standard QF-Test variable expansion for `$ ( ... )` or `${ ... : ... }` syntax.

Remember to double the '\$' signs to avoid expansion before the script is executed (see section 36.6⁽⁶¹⁶⁾).

Parameters

text	The string to expand.

Returns	The expanded string.

```
Object fromServer(String name)
```

SUT only. Retrieve the value of a global variable in the Jython or Groovy interpreter of QF-Test. If the variable is undefined, a `KeyError` is raised.

Parameters

name	The name of the variable.

Returns	The value of the variable.

```
Object fromSUT(String client, String name)
```

Server only. Retrieve the value of a global variable in the Jython or Groovy interpreter of the SUT. If the variable is undefined, a `KeyError` is raised.

Parameters

client	The name of the SUT client.

name	The name of the variable.

Returns	The value of the variable.

```
Boolean getBool(String varname)
```

Look up the value of a QF-Test variable, similar to `lookup()`, but treat it as a boolean.

Parameters

varname	The name of the variable.

Returns	The value of the variable.

Boolean `getBool(String group, String name)`

Look up the value of a QF-Test resource or property, similar to `lookup()`, but treat it as a boolean.

Parameters

group The name of the group.
name The name of the resource or property.

Returns The value of the resource or property.

Exception `getCaughtException()`

Server only. If the script is run inside a `Catch`⁽⁴⁰⁸⁾ node, the exception that was caught is returned. Similarly, if a `Test`⁽³³⁰⁾ node has the `Implicitly catch exceptions`⁽³³³⁾ attribute set, the caught exception is available during execution of the Test node's `Cleanup`⁽³⁵²⁾ node. In all other cases, `None` is returned.

Returns The caught exception.

Component `getComponent(String id, int timeout=0, boolean hidden=false)`

SUT only. Find a component or a component's sub-item using QF-Test's component recognition mechanism.

Parameters

id The `QF-Test ID`⁽⁵⁴⁶⁾ of the `Component`⁽⁵⁴⁵⁾ node that represents the component in the test-suite.

timeout This parameter is ignored and always 0 for SUT scripts that are running on the event dispatch thread of the respective GUI engine because it is not possible to free this thread in a safe way in order to wait for the respective component.

hidden If true, find invisible components as well. Useful for menu-items.

Returns The actual Java component. For sub-items, a pair of the form `(component, index)` is returned, where the type of `index` depends on the type of the item. For tree nodes it is a `javax.swing.tree.TreePath` object, for table-cells a pair of the form `(row, column)` and an integer for all other kinds of items.
 Column indexes returned are always given in table coordinates, not in model coordinates.

List `getConnectedClients()`

Get the names of the currently connected SUT clients.

Returns A list with the names of the currently connected SUT clients, an empty list in case there are none.

Properties `getGlobals()`

Get the global variables bound in the current context.

Returns The global variables of the current context.

Integer `getInt(String varname)`

Look up the value of a QF-Test variable, similar to `lookup()`, but treat it as an integer.

Parameters

varname The name of the variable.

Returns The value of the variable.

Integer `getInt(String group, String name)`

Look up the value of a QF-Test resource or property, similar to `lookup()`, but treat it as an integer.

Parameters

group The name of the group.

name The name of the resource or property.

Returns The value of the resource or property.

Object `getLastComponent()`

SUT only. Get the last component that was addressed by QF-Test for replaying some event, check or miscellaneous operation. Calls to `rc.getComponent()` have no impact.

Returns The last component addressed by QF-Test.

Exception `getLastException()`

Server only. Get the last exception (caught or uncaught) that was thrown during the test-run. In most cases `getCaughtException` is probably more useful.

Returns The most recent exception that was thrown.

Object `getLastItem()`

SUT only. Get the last item that was addressed by QF-Test for replaying some event, check or miscellaneous operation. Calls to `rc.getComponent()` have no impact.

Returns The last item addressed by QF-Test.

Properties `getLocals(nonEmpty=false)`

Get the innermost local bindings of the context. Mostly useful within a procedure to get the parameters of the procedure call and implement something similar to keyword arguments in Jython or Groovy.

Parameters

nonEmpty True to get the first non-empty set of bindings, false to get the innermost bindings even when empty.

Returns The innermost local variable bindings of the current context.

Number getNum(String varname)

Look up the value of a QF-Test variable, similar to `lookup()`, but treat it as a number, i.e. as int or float for Jython and as Integer or BigDecimal for Groovy.

Parameters

varname The name of the variable.

Returns The value of the variable.

Number getNum(String group, String name)

Look up the value of a QF-Test resource or property, similar to `lookup()`, but treat it as a number, i.e. as int or float for Jython and as Integer or BigDecimal for Groovy.

Parameters

group The name of the group.

name The name of the resource or property.

Returns The value of the resource or property.

Object getOption(String name)

Get an option value at run time. This method is provided more for the sake of completeness, you will probably not need it. For the obvious use case of restoring the value of an option to its previous value after a change with `setOption` you should use `unsetOption` instead because values set at script level hide values set interactively in the options dialog.

Parameters

name The name of the option, a constant from the `Options` class which is automatically imported in Jython and Groovy scripts. The names of the options that can be read in this way are documented in [chapter 29^{\(237\)}](#).

Returns The current value of the option.

Properties getProperties(String group)

Get a set of loaded properties or resources.

Parameters

group The group name of the properties or resources.

Returns The variables bound for the given group or null if no such group exists.

String getStr(String varname, boolean expand=true)

Look up the value of a QF-Test variable, similar to `lookup()`.

Parameters

varname	The name of the variable.
expand	Whether to expand the value of the variable recursively. If, for example, the value of <code>\$(varname)</code> is the literal string <code>\$(othervar)</code> , this method will return the expanded value of <code>\$(othervar)</code> if <code>expand</code> is true and <code>\$(othervar)</code> itself if <code>expand</code> is false. Note that if you want to set this parameter, you must use Python keyword syntax to avoid conflicts with <code>getStr(String group, String name)</code> , i.e. <code>rc.getStr("var", expand=0)</code> instead of <code>rc.getStr("var", 0)</code> .

Returns The value of the variable.

String getStr(String group, String name, boolean expand=true)

Look up the value of a QF-Test resource or property, similar to `lookup()`.

Parameters

group	The name of the group.
name	The name of the resource or property.
expand	Whether to expand the value of recursively. If, for example, the value of <code>#{group:name}</code> is the literal string <code>\$(othervar)</code> , this method will return the expanded value of <code>\$(othervar)</code> if <code>expand</code> is true and <code>\$(othervar)</code> itself if <code>expand</code> is false.

Returns The value of the resource or property.

String id(String id)

Return the QF-Test ID of a specified component. This method should be used to take care that this QF-Test component ID becomes updated when moving or changing the QF-Test ID of the referenced component.

Parameters

id	The QF-Test component ID.

Returns The QF-Test component ID.

```
boolean isOptionSet(String name)
```

Test whether an option has been set at script level.

Parameters

name	The name of the option, a constant from the <code>Options</code> class which is automatically imported in Jython and Groovy scripts. The names of the options that can be read in this way are documented in chapter 29⁽²³⁷⁾ .

Returns	True if the option has been set, false otherwise.

```
boolean isResetListenerRegistered(ResetListener listener)
```

Server only. Checks if a `ResetListener` is registered.

Parameters

listener	The <code>ResetListener</code> to check, if it is registered.

Returns	True if the <code>ResetListener</code> has been registered, otherwise False.

```
void logDiagnostics(String client)
```

Server only. Adds event information stored in the SUT for possible error diagnosis to the run-log.

Parameters

client	The name of the SUT client from which to get the information.

```
void logError(String msg, boolean nowrap=false)
```

Add an error message to the run log.

Parameters

msg	The message to log.
nowrap	If true, lines of the message will not be wrapped in the report. Use for potentially long messages.

```
void logMessage(String msg, boolean dontcompactify=false,  
boolean report=false, boolean nowrap=false)
```

Add a plain message to the run log.

Parameters

msg	The message to log.
dontcompactify	If true, the message will never be removed from a compact run-log.
report	If true, the message will appear in the report.
nowrap	If true, lines of the message will not be wrapped in the report. Use for potentially long messages.

```
void logWarning(String msg, boolean report=true, boolean nowrap=false)
```

Add a warning message to the run log.

Parameters

msg	The message to log.
report	If true (the default), the warning will be listed in the report. Set this to false to exclude this specific warning from the report.
nowrap	If true, lines of the message will not be wrapped in the report. Use for potentially long messages.

```
void logImage(ImageRep image, String title=None, boolean dontcompactify=false, boolean report=false)
```

Add an `ImageRep` (see [section 39.8.1^{\(737\)}](#)) object to the run log.

Parameters

image	The <code>ImageRep</code> object to log.
title	An optional title for the image.
dontcompactify	If true, the message will never be removed from a compact run-log.
report	True to log the image in the report (implies <code>dontcompactify</code>).

```
String lookup(String varname, boolean expand=true)
```

Look up the value of a QF-Test variable, similar to `$(varname)`.

Parameters

varname	The name of the variable.
expand	Whether to expand the value of the variable recursively. If, for example, the value of <code>\$(varname)</code> is the literal string <code>"\$(othervar)"</code> , this method will return the expanded value of <code>\$(othervar)</code> if <code>expand</code> is true and <code>"\$(othervar)"</code> itself if <code>expand</code> is false. Note that if you want to set this parameter, you must use Python keyword syntax to avoid conflicts with <code>lookup(String group, String name)</code> , i.e. <code>rc.lookup("var", expand=0)</code> instead of <code>rc.lookup("var", 0)</code> .

Returns The value of the variable.

```
String lookup(String group, String name, boolean expand=true)
```

Look up the value of a QF-Test resource or property, similar to `${group:name}`.

Parameters

group	The name of the group.
name	The name of the resource or property.
expand	Whether to expand the value of recursively. If, for example, the value of <code>\${group:name}</code> is the literal string <code>"\$(othervar)"</code> , this method will return the expanded value of <code>\$(othervar)</code> if <code>expand</code> is true and <code>"\$(othervar)"</code> itself if <code>expand</code> is false.

Returns The value of the resource or property.

```
void overrideElement(String id, Component com)
```

SUT only. Override the target GUI element for component recognition for an element with the given QF-Test ID. When that QF-Test ID is referenced QF-Test ignores all associated information and directly returns the given element.

Parameters

id	The QF-Test ID of the GUI element to override.
com	The GUI element to return as the resolved target. None/null to revert to the default mechanism.

```
void removeResetListener(ResetListener listener)
```

Server only. Remove a `ResetListener`.

Parameters

listener	The <code>ResetListener</code> to remove.

```
void removeTestRunListener(TestRunListener listener)
```

Remove a `TestRunListener` from the current run-context.

Parameters

listener	The listener to remove.

```
void resetDependencies(String namespace=None)
```

Completely reset the dependency stack without executing any cleanup.

Parameters

namespace	An optional namespace to reset the dependencies for.

```
void resolveDependency(String dependency, String  
namespace=None, dictionary parameters=None)
```

Resolve a Dependency⁽³⁴⁴⁾.

Parameters

dependency	The fully qualified name of the Dependency to resolve.
namespace	An optional namespace to resolve the Dependency in.
parameters	The parameters for the Dependency. This should be a dictionary. Its keys and values can be arbitrary values. They are converted to strings for the call.

```
void rollbackDependencies(String namespace=None)
```

Unroll the dependency stack.

Parameters

namespace	An optional namespace to unroll the dependencies in.

```
void setGlobal(String name, object value)
```

Define a global QF-Test variable.

Parameters

name	The name of the variable.
value	An arbitrary value for the variable. It is automatically converted to a string. A value of <code>None</code> unsets the variable.

```
void setLocal(String name, object value)
```

Define a local QF-Test variable.

Parameters

name	The name of the variable.
value	An arbitrary value for the variable. It is automatically converted to a string. A value of <code>None</code> unsets the variable.

void setOption(String name, object value)

Set an option value at run time. Any value thus set overrides the value read from the system configuration file or set via the option dialog, but is never shown in the option dialog or saved to a configuration file. The default value can be restored via `unsetOption`.

Parameters

name	The name of the option, a constant from the <code>Options</code> class which is automatically imported in Jython and Groovy scripts. The names of the options that can be set in this way are documented in chapter 29⁽²³⁷⁾ .
value	The value to set, typically a boolean, a number or a constant from the <code>Options</code> class for options edited via a drop-down list. For hotkey options like the "Don't panic" hotkey this value should be a string like "F12" or "Shift-F6". Supported modifiers are "Shift", "Control" or "Ctrl", "Alt" and "Meta" and combinations thereof. Key specifiers are prepended with "VK_" and then looked up in the class <code>java.awt.event.KeyEvent</code> . Case is irrelevant for both, so "shift-alt-enter" will work as well.

void setProperty(String group, String name, object value)

Set the value of a resource or property in a group.

Parameters

group	The name of the group. A new group is created automatically if necessary.
name	The name of the resource or property.
value	An arbitrary value for the property. It is automatically converted to a string. A value of <code>None</code> unsets the property. This method also works for the special groups 'system' and 'env' and can be used as a means to set environment variables and system properties. Values in the special group 'qftest' cannot be set or changed that way.

void skipTestCase()

Stop the execution of the current test-case and mark it as skipped.

void skipTestSet()

Stop the execution of the current test-set and mark it as skipped.

void stopTest()

Terminate the current test-run.

```
void stopTestCase(boolean expectedFail=false)
```

Stop the execution of the current test-case.

Parameters

expectedFail	If true, mark possible errors in this test-case as expected failures.

```
void stopTestSet()
```

Stop the execution of the current test-set.

```
void syncThreads(String name, int timeout, int count=-1,
boolean throw=true, int remote=0)
```

Server only. Synchronize a number of parallel threads for load testing. The current thread is blocked until all threads have reached this synchronization point or the timeout is exceeded. In the latter case, a `TestException`⁽⁵⁵⁸⁾ is thrown or an error logged.

Parameters

name	An identifier for the synchronization point.
timeout	The maximum time to wait in milliseconds.
count	The number of threads to wait for. Default value -1 means all threads in the current QF-Test instance.
throw	Whether to throw an exception (default) or just log an error if the timeout is exceeded without all threads reaching the synchronization point.
remote	The number of QF-Test instances - potentially running on different machines - to synchronize. Default 0 means don't do remote synchronization.

```
void toServer(...)
```

SUT only. Set some global variables in the Jython or Groovy interpreter of QF-Test. Each argument can be any of:

A string

This is treated as the name of a global variable in the local interpreter. The variable by the same name in QF-Test's interpreter is set to its value.

A dictionary with string keys

For each key in the dictionary, a global variable by that name is set to the corresponding value from the dictionary.

A keyword argument in the form `name=value`

The global variable named `name` is set to `value`.

```
void toSUT(String client, ...)
```

Server only. Set some global variables in the Jython or Groovy interpreter of the SUT. Except for `client`, each argument can be any of:

A string

This is treated as the name of a global variable in the local interpreter. The variable by the same name in SUT's interpreter is set to its value.

A dictionary with string keys

For each key in the dictionary, a global variable by that name is set to the corresponding value from the dictionary.

A keyword argument in the form `name=value`

The global variable named `name` is set to `value`.

Parameters

client	The name of the SUT client.

```
void unsetOption(String name)
```

Restore an option value by removing a possible override from a previous call to `setOption`.

Parameters

name	The name of the option to unset, a constant from the <code>Options</code> class which is automatically imported in Jython and Groovy scripts. The constants for options that can be set in this way are documented in chapter 29⁽²³⁷⁾ .

37.7 The `qf` module

In some cases there is no run-context available, especially when implementing some of the extension interfaces described in the following sections. The module `qf` enables logging in those cases and also provides some generally useful methods that can be used without depending on a run-context. Following is a list of the methods of the `qf` module in alphabetical order. Unless mentioned otherwise, methods are available in Groovy and Jython and for both Server script and SUT script nodes.

Note

Please note that the Groovy syntax for keyword parameters is different from Jython and requires a `:` instead of `=`. The tricky bit is that, for example, `qf.logMessage("bla", report=true)` is perfectly legal Groovy code yet doesn't have the desired effect. The `=` here is an assignment resulting in the value `true`, which is simply passed as the

second parameter, thus the above is equal to `qf.logMessage("bla", true)` and the `true` is passed to `dontcompactify` instead of `report`. The correct Groovy version is `qf.logMessage("bla", report:true)`.

String boolean getClassname(Object objectOrClass)

Get the fully qualified name of the Class of a Java object, or of a Java class itself. Mostly useful for Jython where getting the name of a class can become a real hassle.

Parameters

objectOrClass The Java object or class to get the class name for.

Returns The class name or None in case something non-Java is passed in.

boolean isInstance(Object object, String className)

This is a simple alternative to `instanceof` in Groovy and `isinstance()` in Jython that deliberately compares class and instance names only so conflicts with differing class loaders are avoided.

Parameters

object The object to check.

className The name of the class or interface to test for.

Returns True if the object is an instance of the given class or implements the given interface.

String toString(Object object, String nullValue)

Get the string representation of an object. Mostly useful for Jython but sometimes also useful for Groovy thanks to the default conversion of null to the empty string.

Parameters

object The object to get the string representation for.

nullValue The value to return if object is None, the empty string by default.

Returns Jython 8-bit or unicode strings are returned unchanged, Java objects are turned into a string via `toString`. In Jython, everything else is converted into an 8-bit Jython string.

void setProperty(Object object, String name, Object value)

Set an arbitrary property for an object. For Swing, SWT or web components the value is stored in the respective user data via `putClientProperty`, `setData` or `setProperty` respectively. For everything else a `WeakHashMap` is used. Either way the property will not prevent garbage collection of the object.

Parameters

object The object to set the property for.

name The name of the property.

value The value to set. Null to remove the property.

Object `getProperty(Object object, String name)`

Get a property for an object that was previously set via `setProperty`.

Parameters

object The object to get the property for.

name The name of the property.

Returns The property value.

void `logError(String msg, boolean nowrap=false)`

Add an error message to the run-log. If a run-context is available it is used and logging takes effect immediately. Otherwise the message is buffered and logged at the next opportunity.

Parameters

msg The message to log.

nowrap If true, lines of the message will not be wrapped in the report. Use for potentially long messages. This parameter has no effect if the message needs to be buffered.

void `logMessage(String msg, boolean dontcompactify=false, boolean report=false, boolean nowrap=false)`

Add a plain message to the run log. If a run-context is available it is used and logging takes effect immediately. Otherwise the message is buffered and logged at the next opportunity.

Parameters

msg The message to log.

dontcompactify If true, the message will never be removed from a compact run-log.

report If true, the message will appear in the report.

nowrap If true, lines of the message will not be wrapped in the report. Use for potentially long messages. This parameter has no effect if the message needs to be buffered.

```
void logWarning(String msg, boolean report=true, boolean
nowrap=false)
```

Add a warning message to the run log. If a run-context is available it is used and logging takes effect immediately. Otherwise the message is buffered and logged at the next opportunity.

Parameters

msg	The message to log.
report	If true (the default), the warning will be listed in the report. Set this to false to exclude this specific warning from the report.
nowrap	If true, lines of the message will not be wrapped in the report. Use for potentially long messages. This parameter has no effect if the message needs to be buffered.

37.8 Image API

3.0+

The Image API provides classes and interfaces to take screenshots, to save or load images or for own image comparisons.

37.8.1 The ImageWrapper class

For taking screenshots you can use the Jython class `ImageWrapper`, located in the module `imagewrapper.py`, which comes with the QF-Test installation.

Here is a short sample Jython script demonstrating the usage of the Image API:

```
from imagewrapper import ImageWrapper
#create ImageWrapper instance
iw = ImageWrapper(rc)
#take screenshot of the whole screen
currentScreenshot = iw.grabScreenshot()
#save screenshot to a file
iw.savePng("/tmp/screenshot.png", currentScreenshot)
```

Example 37.2: Image API in Jython

And the same in Groovy:

```
import de.qfs.ImageWrapper
def iw = new ImageWrapper()
def currentScreenshot = iw.grabScreenshot()
iw.savePng("/tmp/screenshot.png", currentScreenshot)
```

Example 37.3: Image API in Groovy

Following is a list of the methods of the `ImageWrapper` class in alphabetical order. The syntax used is a bit of a mixture of Java and Python. Python doesn't support static typing, but the parameters are passed on to Java, so they must be of the correct type to avoid triggering exceptions. If a parameter is followed by an '=' character and a value, that value is the default and the parameter is optional.

ImageWrapper ImageWrapper(RunContext rc)

Constructor method of the `ImageWrapper` class.

Parameters

rc	The current run context of QF-Test.

ImageRep grabScreenshot(int x=None, int y=None, int width=None, int height=None)

Take screenshot of the whole screen. If you use the parameters x, y, width and height, you can take a screenshot of a specific region of the screen.

Parameters

x	The X coordinate of the left upper corner of the region to take the screenshot.
y	The Y coordinate of the left upper corner of the region to take the screenshot.
width	The width of the region to take the screenshot.
height	The height of the region to take the screenshot.

Returns	An <code>ImageRep</code> object containing the actual screenshot.

```
ImageRep grabImage(Object com, int x=None, int y=None, int width=None, int height=None)
```

Take screenshot of a given component. If you use the parameters x, y, width and height, you can take a screenshot of a specific region of the component.

Parameters

com	The QF-Test ID of the component to take a screenshot from.
x	The X coordinate of the left upper corner of the region to take the screenshot.
y	The Y coordinate of the left upper corner of the region to take the screenshot.
width	The width of the region to take the screenshot.
height	The height of the region to take the screenshot.

Returns An `ImageRep` object containing the actual screenshot.

```
ImageRep[] grabScreenshots(int monitor=None)
```

Take screenshots of all available screens. This procedure might be useful, if you work with more than one screen.

If you want to take a screenshot of one specific screen, you can also use this procedure.

Parameters

monitor	Index of the monitor to take the screenshot from. The first monitor has 0, the second 1 and so on.

Returns An array of `ImageRep` objects of all screenshots or the specific `ImageRep` object, if the `monitor` parameter has been used.

```
int getMonitorCount()
```

Return the number of monitors.

Returns The total number of monitors.

```
void savePng(String filename, ImageRep image)
```

Save the given `ImageRep` object to a file. The file will be in PNG format.

Parameters

filename	The path to the file, where the image should be stored to.
image	The <code>ImageRep</code> object to store.

```
ImageRep loadPng(String filename)
```

Load an image from a given file return an `ImageRep` object containing this image. The file has to contain the image in PNG format.

Parameters

filename	The path to the file, where the image is stored.

Returns An `ImageRep` object containing loaded image.

37.9 Exception handling

All QF-Test exceptions listed in [chapter 31](#)⁽⁵⁵⁸⁾ are automatically imported in Jython scripts and can be used for `try/except` clauses like

```
try:
    com = rc.getComponent("someId")
except ComponentNotFoundException:
    ...
```

When working with Groovy you must first import the exception:

```
import de.qfs.apps.qftest.shared.exceptions.
    ComponentNotFoundException
try {
    com = rc.getComponent("someId")
} catch (ComponentNotFoundException) {
    ...
}
```

Only the following exceptions should be raised explicitly from script code (with `raise` or `throw new` respectively):

- `UserException("Some message here...")` should be used to signal exceptional error conditions.
- `BreakException()` or `raise BreakException("loopId")` can be used to break out of a [Loop](#)⁽³⁸⁸⁾ or [While](#)⁽³⁹¹⁾ node, either without parameters to break out of the innermost loop or with the QF-Test loop ID parameter to break out of a specific loop with the respective QF-Test ID.
- `ReturnException()` or `raise ReturnException("value")` can be used to return - with or without a value - from a [Procedure](#)⁽³⁷⁸⁾ node, similar to executing a [Return](#)⁽³⁸⁴⁾ node.

37.10 Debugging scripts (Jython)

When working with Jython modules you don't have to restart QF-Test or the SUT after you made changes. You can simply use `reload(<modulename>)` to load the module anew.

Debugging scripts in an embedded Jython interpreter can be tedious. To simplify this task, QF-Test offers an active terminal window for communicating with each interpreter. These terminals are accessible through the **Clients** menu or through **Extras→Jython Terminal...**.

Alternatively, a network connection can be established to talk remotely to the Jython interpreter - in QF-Test as well as within the SUT - and get an interactive command line. To enable this feature you must use the command line argument `-jythonport <number>`⁽⁵⁷⁷⁾ to set the port number that the Jython interpreter should listen on. For the SUT `-jythonport=<port>` can be defined in the "Extra" Parameters⁽⁴²⁴⁾ of the Start Java SUT client⁽⁴²²⁾ or Start SUT client⁽⁴²⁶⁾ node. You can then connect to the Jython interpreter, for example with

```
telnet localhost <port>
```

Combined with Jython's ability to access the full Java API, this is not only useful for debugging scripts but can also be used to debug the SUT itself.

Chapter 38

Special features for web applications

Web

This chapter addresses topics that are only relevant when testing web applications in a browser.

38.1 Pseudo DOM API

To a certain extent QF-Test exposes the DOM of a web based SUT to SUT script⁽⁴¹⁹⁾ nodes. This API is not equivalent to working directly at the Javascript level which can be done via the methods `toJS` and `evalJS` described in this chapter. With this API is possible to traverse the DOM and retrieve and set attributes of the respective nodes, but not to manipulate the structure of the DOM. Thus this API is useful for implementing `Name- or FeatureResolvers` as described in section 39.1⁽⁶⁷⁸⁾.

Note

The DOM can differ depending on the browser, so you should try not to rely too much on child indexes in your resolvers or scripts in case of cross-browser testing if viable.

QF-Test's DOM API comprises a hierarchy of the following five classes:

38.1.1 The abstract `Node` class

All classes of QF-Test's pseudo DOM are derived from this class and thus implement the following interface. Its is located in the package `de.qfs.apps.qftest.client.web.dom`.

String getNodeTypes()

Get an identifier for the type of the node. Though this method is not fully in line with pure OO doctrine, while traversing the DOM it is often necessary to find out the type of a given node and this is quite convenient.

Returns A string specifying the type of node. The respective constants are defined in the concrete sub-classes: `DocumentNode.DOCUMENT_NODE`, `FrameNode.FRAME_NODE`, `DomNode.DOM_NODE` and `DialogNode.DIALOG_NODE`.

String getName()

Get the tag name of the node. The tag name is the type of node in upper case like "HTML" for an <HTML> node. Plain text nodes are represented as a `DomNode` with tag name "#text". Pseudo tag names are defined for `DocumentNodes` ("DOCUMENT") and `DialogNodes` ("DIALOG").

Returns The tag name of the node.

String getAttribute(String name)

Get the value of an attribute of the node. For convenience this method is defined at Node level. If this node is not a `DomNode` the result will always be null. In some cases this method also returns attributes that have not been explicitly specified, like the width and height of an IMG node. Which attributes can be retrieved in that way is browser dependent.

Parameters

name The name of the attribute

Returns The value of the attribute or null if no such attribute is available for the node.

String getAttributeIfSpecified(String name)

Get the value of an attribute of the node if it is explicitly specified in the HTML code. For convenience this method is defined at Node level. If this node is not a `DomNode` the result will always be null.

Parameters

name The name of the attribute

Returns The value of the attribute or null if no such attribute is explicitly specified for the node.

boolean isAttributeSpecified(String name)

Test whether an attribute of the node is explicitly specified in the HTML code. For convenience this method is defined at Node level. If this node is not a `DomNode` the result will always be false.

Parameters

name The name of the attribute

Returns True if an attribute is explicitly specified for the node.

String getText()

Get the text content of a Node. In case of a plain text node, return its value. Otherwise traverse the DOM and collect all nested text node's values. Care is taken to collapse whitespace between nodes to get as close as possible to what is displayed in the browser.

Returns The whole direct and indirect text content of a node.

String getVisibleText()

Get the text content of a Node. In case of a plain text node, return its value. Otherwise traverse the DOM and collect all nested text node's values, ignoring invisible nodes. Care is taken to collapse whitespace between nodes to get as close as possible to what is displayed in the browser.

Returns The whole direct and indirect visible text content of a node.

String getFlatText()

Get the flat text content of a Node. In case of a plain text node, return its value. Otherwise collect the values of all text nodes directly contained within this node, excluding those nested at a deeper level.

Returns The whole direct text content of a node.

String getVisibleFlatText()

Get the flat text content of a Node. In case of a plain text node, return its value. Otherwise collect the values of all text nodes directly contained within this node, excluding invisible ones and those nested at a deeper level.

Returns The whole direct visible text content of a node.

String getSimpleText()

Get the simple text content of a Node. In case of a plain text node, return its value. Otherwise traverse the DOM and collect all text nodes contained directly within this node or in "simple" nodes like , but do not descend into structurally more complex nodes like <TABLE>.

Returns The simple text content of a node.

String getVisibleSimpleText()

Get the simple text content of a Node. In case of a plain text node, return its value. Otherwise traverse the DOM and collect all visible text nodes contained directly within this node or in "simple" nodes like , but do not descend into structurally more complex nodes like <TABLE>.

Returns The visible simple text content of a node.

DocumentNode getDocument()

Get the document to which this node belongs.

Returns The document to which this node belongs. A DocumentNode will return itself, a DialogNode will return null.

Browser isBrowserIE()

Test whether the browser to which the node belongs is an Internet Explorer.

Returns True for Internet Explorer, false for Mozilla derivatives.

Browser isBrowserMozilla()

Test whether the browser to which the node belongs is a Mozilla derivative.

Returns True for Mozilla derivatives, false for Internet Explorer.

Node getParent()

Get the parent of a node. A top-level `DocumentNode` or a `DialogNode` will return null. Everything else should have a parent unless removed from the DOM via Javascript.

Returns The parent of the node.

int getChildCount()

Get the number of child nodes.

Returns The number of child nodes.

Node getChild(int index)

Get the child node at the given index.

Parameters

index The index of the child node, starting with 0.

Returns The child node at the index.

Throws

IllegalArgumentException if index is negative or exceeds the number of child nodes.

Node[] getChildren()

Get the children of this node as an array.

Returns The child nodes.

int getIndexOfChild(Node child)

Get the index of a given child node.

Parameters

child The child node to get the index for.

Returns The index of the child node, starting with 0, or -1 in case it is not a child of the this node.

Node[] getElementsByTagName(String tagName)

Get all direct or indirect child nodes of this node with a given tag name.

Parameters

tagName The tag name to look for.

Returns An array of child nodes with the given tag name. If none are found, an empty array is returned. A `DialogNode` will always return an empty array, a `FrameNode` forwards the call to its `DocumentNode` child node and a `DocumentNode` to its `<HTML>` root element.

Node getElementById(String id)

Get a direct or indirect child node of this node with a given ID. The given ID is not compared to the node's original 'ID' attribute, but to its ID, which may have been modified by an `IdResolver` (see [section 39.1.5^{\(682\)}](#)).

Parameters

id The ID to look for.

Returns The child node with the given ID, an arbitrary one in case of multiple matches or null if none is found. A `DialogNode` will always return null, a `FrameNode` forwards the call to its `DocumentNode` child node and a `DocumentNode` to its `<HTML>` root element.

Node[] getElementsById(String id)

Get all direct or indirect child nodes of this node with a given ID. The given ID is not compared to the node's original 'ID' attribute, but to its ID, which may have been modified by an `IdResolver` (see [section 39.1.5^{\(682\)}](#)).

Parameters

id The ID to look for.

Returns An array of child nodes with the given ID. If none are found, an empty array is returned. A `DialogNode` will always return an empty array, a `FrameNode` forwards the call to its `DocumentNode` child node and a `DocumentNode` to its `<HTML>` root element.

Node[] getElementByIdAndTagName(String id, String tagName)

Get all direct or indirect child nodes of this node with a given ID and a given tag name. The given ID is not compared to the node's original 'ID' attribute, but to its ID, which may have been modified by an `IdResolver` (see [section 39.1.5^{\(682\)}](#)).

Parameters

id The ID to look for.

tagName The tag name to look for.

Returns An array of child nodes with the given ID and the given tag name. If none are found, an empty array is returned. A `DialogNode` will always return an empty array, a `FrameNode` forwards the call to its `DocumentNode` child node and a `DocumentNode` to its <HTML> root element.

Object getProperty(String name)

Retrieve a user-defined property.

Parameters

name The name of the property to retrieve.

Returns The value of the specified property or null.

setProperty(String name, Object value)

Set a user-defined property.

Parameters

name The name of the property.

value The property value or null to remove an existing property.

38.1.2 The DocumentNode class

The `DocumentNode` class is derived from `Node` and also resides in the package `de.qfs.apps.qftest.client.web.dom`. In addition to the methods defined in the `Node` class and explained above, `DocumentNode` provides the following:

boolean hasParent()

To test whether a document is a top-level document this method should be used instead of testing whether the result of `getParent()` is null. The reason is that loading of nested child documents may be completed before loading of the main document. During this time it is known that the nested document will have a parent, but the parent is not available yet.

Returns True if the document has a parent, false if it is a top-level document.

DomNode **getRootElement()**

Get the <HTML> root element of the document.

Returns The root element of the document.

FrameNode[] **getFrames()**

Get the child frames of the document.

Returns The child frames of the document, an empty array in case there are none.

String **getUrl()**

Get the URL of the document.

Returns The URL of the document.

String **getTitle()**

Get the title of the document as defined in the <TITLE> of the <HEAD> of its root element.

Returns The title of the document.

String **getSourcecode()**

Get the HTML source code of the document in its current state, which is not necessarily the same as what was loaded when the document was opened because attributes or the DOM's structure may have been changes since, e.g. via Javascript.

Returns The current HTML code of the document.

Object **evalJS(String script)**

Evaluate some Javascript code in the context of this document.

Parameters**script** The script to execute.**Returns** Whatever the script returns, converted to the proper object type. Even returning a DOM node, frame or document works.

In addition to the instance methods describe above, the `DocumentNode` class provides some static convenience methods for manipulating URLs.

static String **getUrlBase(String url)**

Get the base part of an URL, including host and directory but excluding possible parameters separated by '?' and excluding a possible Tomcat session ID.

Parameters**url** The URL to get the base part of.**Returns** The base part of the URL.

int[] getGeometry()

Get the location and size of the frame relative to its parent frame or the browser window's display area.

Returns The frame geometry in the form [x, y, width, height].

38.1.4 The DomNode class

The `DomNode` class is derived from `Node` and also resides in the package `de.qfs.apps.qftest.client.web.dom`. In addition to the methods defined in the `Node` class and explained above, `DomNode` provides the following:

getId()

Get the cached ID of the node which may have been processed by an `IdResolver` as described in [section 39.1.5^{\(682\)}](#). The original, unmodified 'ID' attribute is available via `getAttribute("id")`.

Returns The cached ID of the node.

setAttribute(String name, String value)

Set an attribute value on the node.

Parameters

name The name of the attribute.

value The value to set.

Node[] getChildrenByTagName(String tagName)

Get all direct child nodes of this node with a given tag name.

Parameters

tagName The tag name to look for.

Returns An array of child nodes with the given tag name. If none are found, an empty array is returned.

boolean hasFocus()

Test whether the node has the keyboard focus.

Returns True if the node has the keyboard focus.

requestFocus()

Tell the node to request the keyboard focus. This is a best-effort implementation. Whether the node actually receives the keyboard focus depends on the browser and operating system settings.

boolean isShowing()

Test whether the node is visible or can be made visible by scrolling.

Returns True if the node is visible.

void scrollVisible()

Try to scroll the page or a frame so as to make the node fully visible. This is a best effort implementation. Whether the node can actually be made visible depends on the browser and operating system settings.

int[] getLocationOnScreen()

Get the location and size of the node relative to desktop.

Returns The node's geometry in the form [x, y, width, height].

void toJS(String name)

Set a Javascript variable in the context of this node's document to this node.

Parameters

name The name of the variable to set.

Object evalJS(String script)

Evaluate some Javascript code in the context of this node's document.

Parameters

script The script to execute.

Returns Whatever the script returns, converted to the proper object type. Even returning a DOM node, frame or document works.

38.1.5 The DialogNode class

The `DialogNode` class, also derived from `Node` is not a standard DOM class, but created solely for convenient access to dialogs within QF-Test. It also resides in the package `de.qfs.apps.qftest.client.web.dom` but has very little to do with the other `Node` classes. A `DialogNode` represents a message or error dialog which can be triggered via Javascript. It provides the following methods:

long getStyle()

Get the style of the dialog.

Returns The style of the dialog. Any of the constants `STYLE_ALERT`, `STYLE_CONFIRM` or `STYLE_AUTHENTICATE` defined in the `DialogNode` class.

String getTitle()

Get the title of the dialog.

Returns The title of the dialog.

String `getText()`

Get the message text of the dialog.

Returns The message text of the dialog.

38.2 Special support for various AJAX toolkits

The acronym *AJAX* for "Asynchronous Javascript and XML" is generally associated with modern web applications that provide a more interactive look and feel than classical FORM based applications, almost comparable to desktop applications. Behind this common acronym there is a whole zoo of toolkits that drive those applications, each with different key aspects and unique widget sets. Such toolkits pose a problem for QF-Test and in fact any automated testing tool for several reasons:

- The actual component hierarchy is created automatically from abstract widgets like Buttons or Lists. Often each widget is implemented as a number of DIV nodes. This leads to very deeply nested hierarchies with very little structure.
- IDs are either not assigned at all or automatically created and thus worse than useless.
- The asynchronous communication with the web server and dynamic creation of DOM nodes may cause timing-related problems.

There is no panacea to address these problems in a generic way. In most cases QF-Test can interact with AJAX applications out-of-the-box, but component recognition and performance are not ideal. Optimal testability can only be achieved with special case handling that exactly fits a given toolkit and takes advantage of its peculiarities.

For the following list of toolkits QF-Test provides resolvers for optimized handling:

Toolkit name	Toolkit homepage
Ext GWT	http://www.extjs.com
Ext JS	http://www.extjs.com
Google Web Toolkit (GWT)	http://www.gwtproject.org
ICEfaces	http://icesoft.org
jQuery UI	http://jqueryui.com/
jQuery EasyUI	http://jeasyui.com/
Prime Faces	http://primefaces.org
Qooxdoo	http://qooxdoo.org
Rich Ajax Platform (RAP)	http://www.eclipse.org/rap/
RichFaces	http://www.jboss.org/richfaces
Vaadin	http://vaadin.com
ZK	http://zkoss.org

Table 38.1: Supported AJAX toolkits

QF-Test is even able to automatically detect whether one of those toolkits is used in your web application and to install the respective resolver as described below.

38.2.1 AJAX resolver concepts

An AJAX resolver is a set of resolvers and other methods implemented specifically for a given AJAX toolkit. Most notably QF-Test tries to assign individual classes matching the high-level widgets to DOM nodes and remove intermediate nodes that are just an implementation detail. Name, Feature and Extra feature attributes are determined in a way suitable for the toolkit and events are simulated on the correct DOM node in a way that most closely matches user interaction. These measures drastically reduce the component hierarchy and increase the reliability and performance of component recognition and replay. Timing and synchronization are also addressed.

As a necessary consequence the components and events recorded for a given web application vary drastically with and without an active AJAX resolver and are not compatible with each other. Thus the decision whether or not to use an AJAX resolver should be made as early as possible, otherwise tests will either need to be reimplemented after activating the resolver or tests with and without resolver must be cleanly separated. If a resolver is available for your application you should practically always use it. The only exception is if the existing test-base is already too large, mostly complete and stable.

Implementing AJAX resolvers is an ongoing process. As changes to a resolver can have consequences in terms of how components are recorded and recognized we need to make sure that you can update your QF-Test version without losing backwards compatibility with existing tests. To that end, AJAX resolvers have their own three-digit

versioning scheme, starting with version 1.0.0, with the following meaning:

- A change in the minor version, e.g. 1.0.1, means that compatibility is not affected.
- Medium version changes, e.g. 1.1.0, imply that tests recorded with a lower medium version should still replay, but recording additional sequences may record components in a different way. This is not critical, but you may want to avoid it in case the benefits of the newer version don't outweigh it.
- A major version change, e.g. 2.0.0, can be incompatible even for replay so that manual updates or new recording of the affected components may be required.

AJAX resolvers are activated via the Procedure `qfs.web.ajax.installToolkitResolver` in the standard library `qfs.qft` where you have to provide the version to use. You can choose to specify only the major version, in which case QF-Test uses the latest medium.minor version available for this major version. This is normally the best option and used in the SUT startup sequences created with QF-Test's quickstart wizard (see [chapter 3^{\(24\)}](#)). Alternatively you can specify major.medium version or even major.medium.minor to use an exact version and thus run your tests with the resolver version with which they were created.

38.2.2 Supported AJAX toolkits

Unfortunately the number of available AJAX toolkits keeps growing rapidly and popularity changes frequently so we have to weigh carefully which toolkit to support and to which degree. Cooperation with QF-Test users also plays a major role. Resolver development may also be frozen - except for critical bug fixes - if a toolkit loses relevance.

This section lists the supported AJAX toolkits and versions with the available resolver versions. Newer toolkit versions don't necessarily require an update of the resolver. If your application uses a newer version of the toolkit than the supported versions listed below, please use the latest AJAX resolver available for this toolkit and let us know about your results.

Ext JS

The resolver was initially created for Ext JS version 2 but does support Ext JS version 6 in the meantime.

Resolver version	QF-Test version	Remarks
3.1.1	4.0.7	Support for Ext JS version 6
3.1.0	4.0.3	Further generic classes
3.0.0	4.0.2	Support for Ext JS version 5
2.0.0	3.5M1	Support for Ext JS version 4.1, generic classes and sub-items
1.2.0	3.4.1	Support for Ext JS version 4
1.1.0	3.3.0	Improved item support for trees, grids, tab panels and combo lists
1.0.0	3.1.1	Initial version

Table 38.2: Ext JS resolver versions

Ext GWT

There is no separate Ext GWT resolver. Ext GWT is in fact a combination of Ext JS and GWT and QF-Test supports it by installing both the Ext JS and the GWT resolver.

Google Widget Toolkit - GWT

The Google Widget Toolkit has been supported starting with GWT version 1.5. A GWT demo test-suite, tested against GWT version 2.3, is provided in the file `.../demo/web/gwt.qft`. This demo makes use of generic components and provides several useful procedures for working with some common GWT widgets.

Resolver version	QF-Test version	Remarks
2.0.0	4.0.3	New implementation written from scratch with support for a wider range of GWT components based on GWT version 2.7.
1.4.0	3.4.2	Fixed filtering of DIV nodes
1.3.0	3.4.1	Added support for GWT-TABLAYOUTPANEL and GWT-TABLAYOUTPANELTAB
1.2.0	3.4.0	Filtering of DIV nodes, improved extra features, limited handling for obfuscated GWT class names
1.1.0	3.3.0	Improved support for GWT-TREEITEM, GWT-STACKPANELITEM and GWT-TABBARITEM
1.0.1	3.1.4	Replay improvements for GWT 2.0
1.0.0	3.1.1	Initial version

Table 38.3: GWT resolver versions

ICEfaces

The resolver was developed against ICEfaces version 3.3.

Resolver version	QF-Test version	Remarks
1.0.0	3.5.3	Initial version with support for generic classes and sub-items based on the showcase demo, including support for ACE components

Table 38.4: ICEfaces resolver versions

jQuery UI

The resolver was developed against jQuery UI version 1.10.3.

Resolver version	QF-Test version	Remarks
1.1.0	4.0.0	Initial version with support for generic classes and sub-items based on the online demo widgets

Table 38.5: jQuery UI resolver versions

jQuery EasyUI

The resolver was developed against jQuery EasyUI version 1.4.1.

Resolver version	QF-Test version	Remarks
1.0.0	4.0.4	Initial version with support for generic classes and sub-items based on the online demo widgets

Table 38.6: jQuery EasyUI resolver versions

PrimeFaces

The resolver was developed against PrimeFaces version 3.5 but does support PrimeFaces version 5 in the meantime.

Resolver version	QF-Test version	Remarks
2.0.0	4.0.2	Support for PrimeFaces version 5
1.0.0	3.5.1	Initial version with generic classes and sub-items based on the version 3.5 showcase demo

Table 38.7: PrimeFaces resolver versions

Qooxdoo

The resolver was developed against qooxdoo version 0.7.2. Currently not under active development.

Resolver version	QF-Test version	Remarks
1.1.0	3.3.0	Improved support for list and menu items
1.0.0	3.1.1	Initial version

Table 38.8: Qooxdoo resolver versions

Remote Application Platform - RAP

The resolver was developed against RAP version 1. Support for RAP version 2 and higher is complete.

Resolver version	QF-Test version	Remarks
3.0.3	4.0.6	Updated for RAP version 3.0; retrieval of names set via <code>widget.setData("name", ...)</code>
3.0.2	-	Experimental internal version
3.0.1	-	Experimental internal version
3.0.0	3.5.1	Updated for RAP version 2.0 (Remote Application Platform)
2.0.0	3.5M1	Support for generic classes and sub-items. RAP Version 1.5 (Rich Ajax Platform)
1.1.0	3.3.0	Improved item support for lists, tab folders, tables and trees
1.0.1	3.3.0	Improved widget recognition
1.0.0	3.1.1	Initial version

Table 38.9: RAP resolver versions

RichFaces

The resolver was developed against RichFaces versions 3, 4 and 5.

Resolver version	QF-Test version	Remarks
3.0.1	4.0.4	Improved handling of enabled state of elements.
3.0.0	4.0.0	Make use of generic classes and updated to work with experimental version of RichFaces 5
2.0.2	3.5.2	Fixes for replaying clicks on Windows 7 systems. Recording and replaying of file:upload elements works correctly now.
2.0.1	3.5M3	Minor fixes
2.0.0	3.5M1	Support for generic classes and sub-items
1.4.0	3.5.0	Supports items for trees, lists, tabs and tables for RichFaces 3 and RichFaces 4.
1.3.0	3.4.8	Replaying menu selections on ddm-items as hard events. Fixed wrong event filtering for rich-tree components. Several minor fixes for abstract coordinates and replay. Tested with RichFaces v3.3.3 and v4.2.2.
1.2.0	3.4.1	Supports RichFaces 4
1.1.0	3.3.0	Major rewrite
1.0.0	3.1.4	Initial version

Table 38.10: RichFaces resolver versions

Vaadin

Vaadin is another toolkit that is based on GWT but comes with additional features and its own widget set.

Resolver version	QF-Test version	Remarks
1.1.0	3.5.0	Updated for Vaadin 7 based on the dashboard Vaadin demo
1.0.0	3.5M1	Initial version with support for generic classes and sub-items

Table 38.11: Vaadin resolver versions

ZK

The resolver was developed for versions 6.5 and 7.

Resolver version	QF-Test version	Remarks
1.2.0	4.0.5	Fixed some rare issues when reading text from table cells or tree nodes containing multiple components.
1.1.1	4.0.3	Improvements in determining text of MenuItem's.
1.1.0	3.5.90	Support for common generic classes and use of IDs from zul files as component names.
1.0.1	3.5.2	Minor improvements on component recognition and better filtering of auto generated ZK IDs.
1.0.0	3.5.1	Initial version with support for generic classes, sub-items and checks.

Table 38.12: ZK resolver versions

38.2.3 Setting unique IDs

Any AJAX toolkit has its custom way of setting unique IDs. Please find details about the supported ones in the following chapter:

Ext JS

You can set IDs like

```
var container = Ext.create('Ext.container.Container', {
id: 'MyContainerId',
... });
```

As alternative you can also call `container.getEl().set({ 'qfs-id': 'myId' })`; In this case you will need to implement a NameResolver for reading 'qfs-id' as name for QF-Test.

GWT

The simplest way is calling the method `widget.getElement().setId("myId");` for the required widgets.

As an alternative you can also call `widget.ensureDebugId("myId")`. But if you want to use that method you need to modify your `xxx.gwt.xml` file to enable debug IDs. Add `<inherits name="com.google.gwt.user.Debug"/>` to the file.

It's also possible to set a custom identifier which can then be used via a NameResolver. For example call `setAttribute("qfs-id", "myId")` to set an 'qfs-id' attribute.

ICEfaces

The simplest solution is to set the 'ID' attribute `<p:inputText id="myId"/>` for any required component in the xhtml definition.

jQuery UI

The simplest solution is to set the 'ID' attribute `<p:inputText id="myId"/>` for any required component. Additionally, you can give an existing element an id with: `$(element).attr("id", "myId");`

jQuery EasyUI

The simplest solution is to set the 'ID' attribute `<p:inputText id="myId"/>` for any required component.

PrimeFaces

The simplest solution is to set the 'ID' attribute `<p:inputText id="myId"/>` for any required component in the xhtml definition.

RAP

Starting with RAP version 2.2 a name set via `widget.setData("name", ...)` is retrieved automatically by QF-Test, just as for SWT.

For RAP versions older than 3.0.0 the following technique is also available. It is discouraged because it deviates from the SWT standard and requires additional settings for the webserver:

Call the method `widget.setData(WidgetUtil.CUSTOM_WIDGET_ID, "myId");` for the required widgets.

After applying IDs to components you need to modify your webserver environment and specify the following parameter `-Dorg.eclipse.rap.rwt.enableUITests=true` before launching the webserver.

Note: The VM argument was renamed in RAP 2.0. For RAP versions older than 2.0 its name is `-Dorg.eclipse.rap.rwt.enableUITests=true`.

RichFaces

You need to set the 'ID' attribute in your source code or graphical editor.

Vaadin

The simplest solution is to call `widget.setId("id")` (`widget.setDebugId("id")` for Vaadin version < 7) for any required component.

You can also set a custom stylesheet class, which you could read with a `NameResolver`. Therefore call `widget.setStyleName("qfs-id=myId")`.

ZK

QF-Test uses the widget ID, which is also used in the `zul` files, so you should get meaningful IDs for most of the objects.

The ZK framework also offers a custom `IDGenerator` to set such IDs. But implementing this could be quite exhaustive. In this case it might be a better choice to rely on the default mechanism of QF-Test.

38.3 Customizing QF-Test for your AJAX toolkit

4.0+

Besides the specifically supported AJAX toolkits (see [section 38.2^{\(657\)}](#)) there are lots of other standard toolkits on the market. In addition many companies implement their own customized toolkit or extend standard toolkits to their needs. This huge variety of different ways of creating web pages makes it impossible for a generic tool like QF-Test to provide support for each toolkit out-of-the-box.

To get out of this dilemma QF-Test provides a generic configurable `CustomWebResolver` to influence QF-Test's component recognition and adapt it specifically to your web page implementation.

38.3.1 Determining specifics of your web page

Note

This section provides a first theoretical overview, the next section then shows a real example.

HTML is a very flexible language to describe the content of web pages. But unfortunately there are no real standards with regards to components which should be used to draw

a button, a textfield or a table showing data. That's why nearly any toolkit invented a custom way of drawing such components.

Due to that flexibility your first step must be understanding how your utilized toolkit creates those components and what are the specific attributes or properties of those generated HTML components.

Once you determined how your toolkit works, you need to teach QF-Test how to deal with those findings. This could be achieved by using the `CustomWebResolver` interface and calling specific procedures. In the end of the day you teach QF-Test which HTML tag represents a button, which HTML property represents a table showing data and so on. From QF-Test's perspective you need to assign generic classes to those HTML tags. Those generic classes will then trigger the default mechanism of component recognition and item resolution of QF-Test as you are used from any other engine. You can find more detail about generic classes at [chapter 46^{\(790\)}](#).

Investigating the HTML code of a web page could be achieved by using your browser's built-in development tools or browser extensions like FireBug.

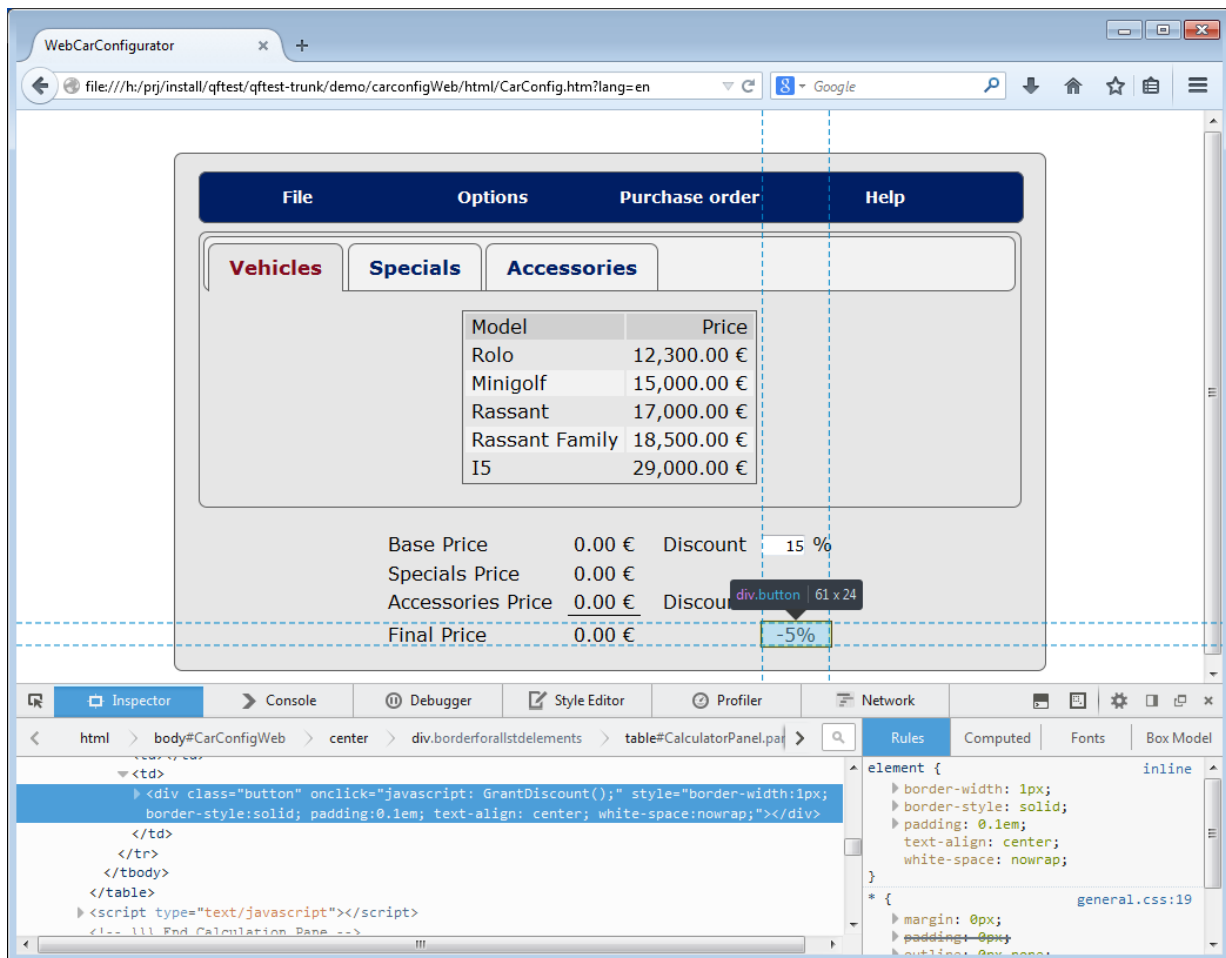


Figure 38.1: HTML code inspection in Firefox

In most cases the HTML attribute `class` is very significant for component recognition. In such an attribute you can find information about the real type of that component, e.g. whether the component acts as button or table. Other toolkits might work with `id` or any other attribute. Of course there are also toolkits around which highly use JavaScript methods. For adapting QF-Test to those toolkits you need to implement much more resolvers, so we set the focus on the simple use-case here.

First you should call the procedure `qfs.web.ajax.installCustomWebResolver` from the standard library `qfs.qft` and fill in the required parameters.

As already mentioned, the `class` attribute provided the class information in most cases, so you could set the parameter `goodCssClasses` to `%. *` which tells QF-Test to record those classes as component classes.

The second step is to map those dedicated values to QF-Test's generic classes via the parameter `genericClasses`, which is described in the next section.

38.3.2 Example for WebCarConfigurator

As this approach is quite difficult to understand just via reading, we describe a sample implementation using the WebCarConfigurator in this section. You can find the WebCarConfigurator files in the folder `qftest-4.0.8/demo/carconfigWeb/carconfigWeb_en.qft`.



Figure 38.2: WebCarConfigurator

As stated in the previous section we need to figure out, which attribute shows the required information, which points us to a generic class.

Simple class mapping

For the beginning we want to resolve the recognition of the '-5%' button in the right bottom corner. So let's analyze it.

First record a simple text check or a mouse-click on that button. Then jump to the recorded components via **Locate component**. There you can see that you received a component of the class `DIV` and an empty name. In the other attributes you can't find anything useful. It is especially important that QF-Test didn't record the actual text of '-5%' in any attributes. So QF-Test has no good information for recognizing that component. There is just the geometry and the structure information. Now let's make this component more readable for us and the QF-Test's component recognition more robust.

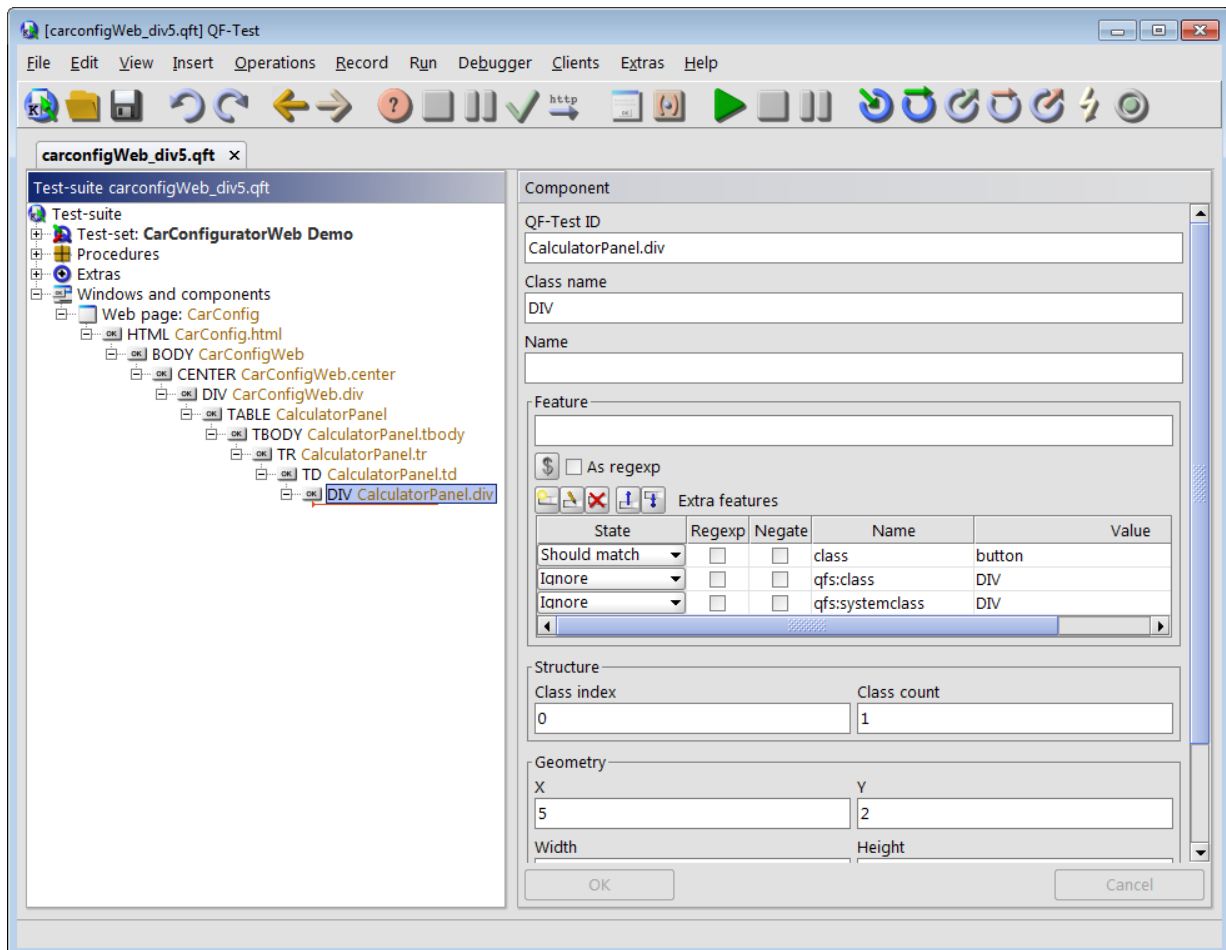


Figure 38.3: Recording of '-5%' button in WebCarConfigurator

Once we analyze the recorded component more in detail we discover that there is an

extra feature `class` with the value `button`. Now we can suppose that a button will have that particular attribute.

So, insert a procedure call to `qfs.web.ajax.installCustomWebResolver` from the standard library `qfs.qft`. As we have detected the `class` attribute holding that class information for QF-Test, we can set the parameter `goodCssClasses` to `%.*`. This parameter will change the classes of all components to the first value of their `class` attribute. In detail `%.*` means a regular expression (`%`) will be used for assigning classes. In that regular expression any match is allowed (`.*`). That's actually a good choice once you start with a new web page. Once you call the procedure and re-record the components, you will get a recording like this:

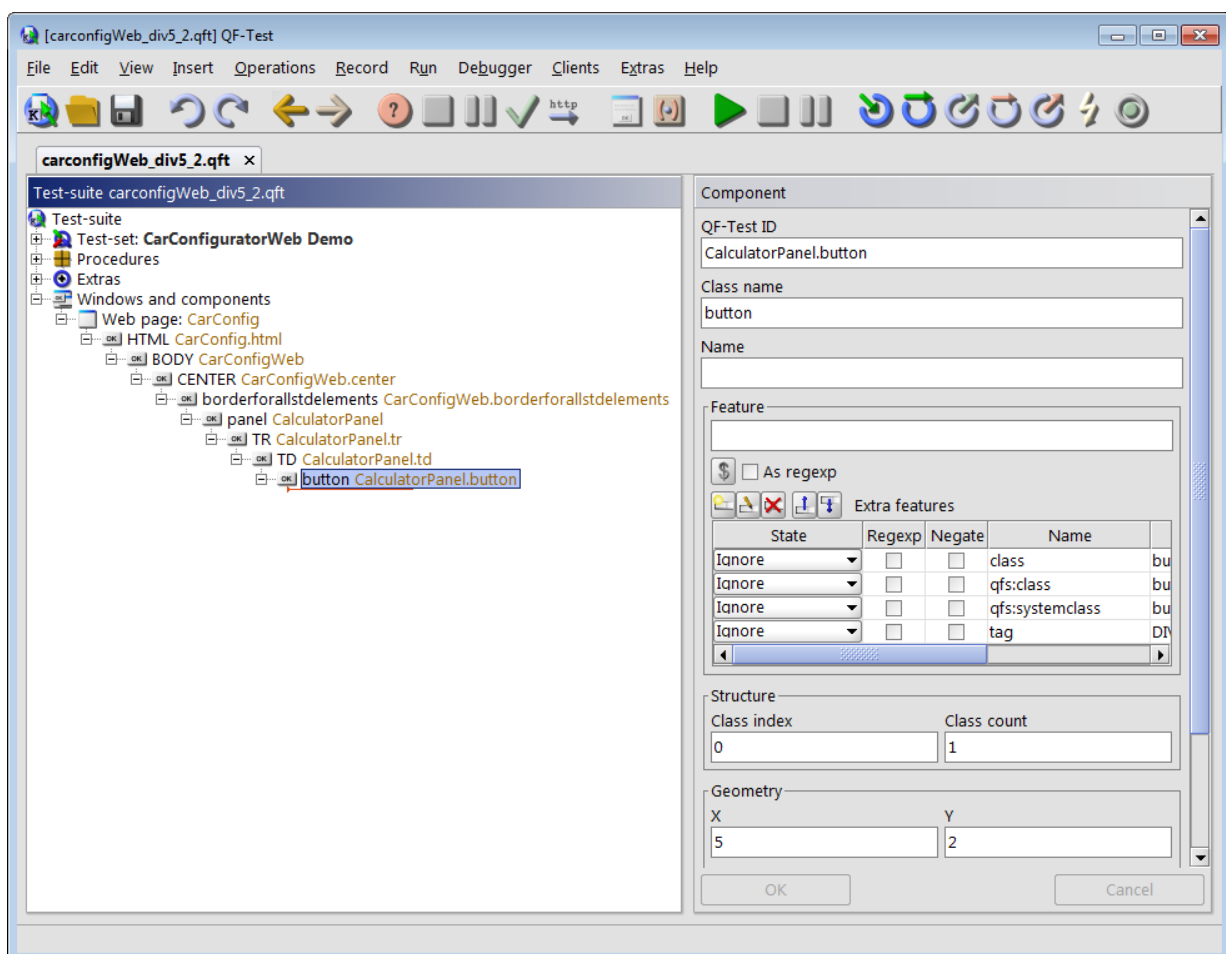


Figure 38.4: Recording with `goodCssClasses` in WebCarConfigurator

Now we can map that `button` to QF-Test's generic class `Button`. Therefore we set the parameter `genericClasses` to `button=Button`. The expression `button=Button` means that any component of the class `button` will get a generic

class called `Button`. Then you can delete the previously recorded component, run the procedure `qfs.web.ajax.installCustomWebResolver`, reload the web page and re-record that component. You will get the following recording:

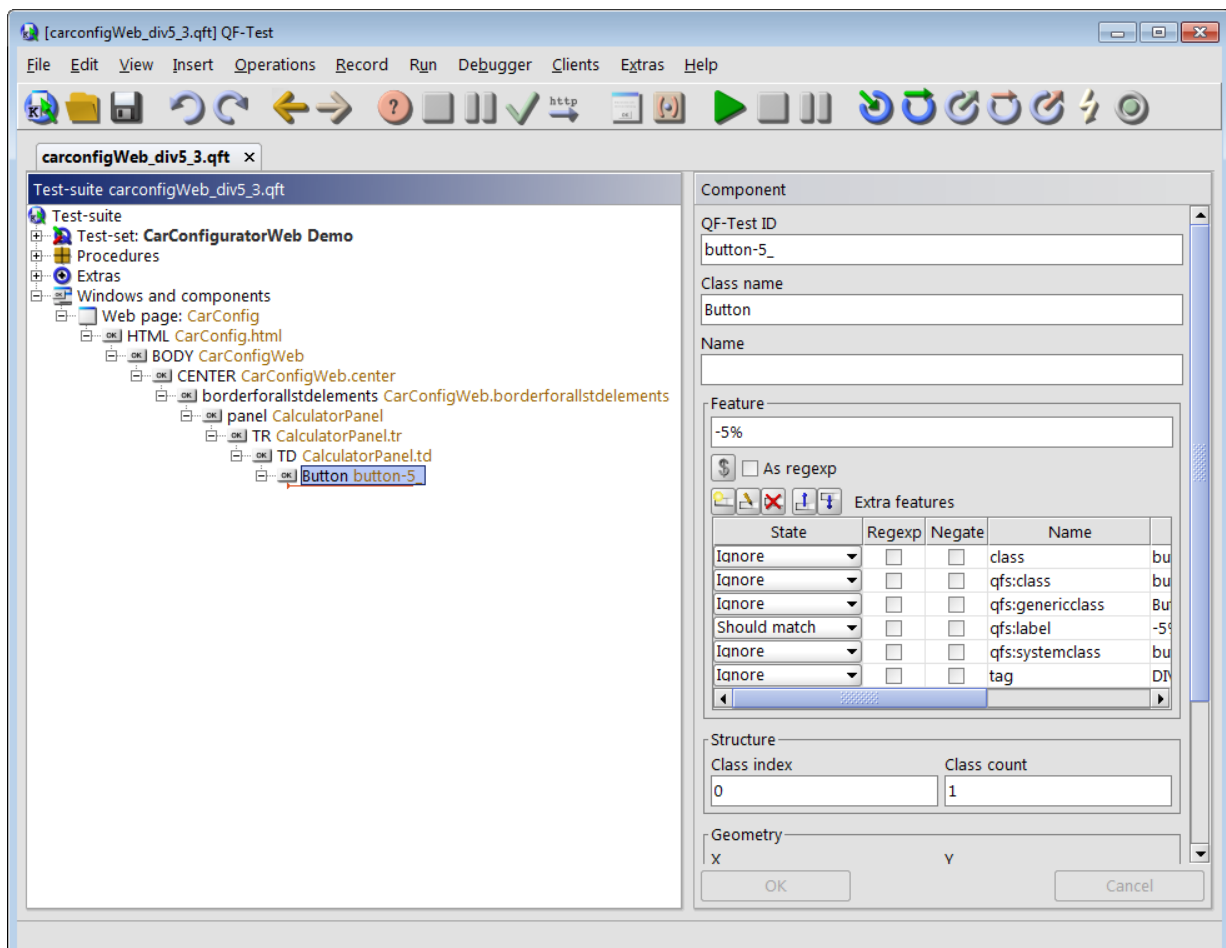


Figure 38.5: Recording with genericClasses in WebCarConfigurator

So you can see that you got a click on component `button-5_`. Once you jump to the recorded component you can see the class `Button` has been assigned as well as the '-5%' as value for the feature and we even got the extra feature `qfs:label` with that text. This component will be treated as button by QF-Test now. Of course you should advise the development team to assign a dedicated ID to that button as well.

Advanced class mapping

After that simple case, let's go for a more complex scenario. Let's analyze how the textfields showing the selected price information are treated, e.g. the final price textfield.

So we need to record some mouse clicks or text checks on those textfields again. After recording we need to jump to the recorded components and analyze them.

We got some SPAN nodes recorded. Here we have no `class` attribute, but they have an `id` attribute assigned at least. So, we can conclude that the `id` is very specific to the particular field. If you select its parent component, which is a `textfield` node, you can find a `class` attribute again which corresponds to the actual component class. QF-Test also highlights the entire textfield in the application once you observe the component highlighting. So we could suppose that this `textfield` describes real textfield components here. The component class `textfield` appears due to the call of the `CustomWebResolver` procedure with the parameter `goodCssClasses` and its value `%.*`. Without calling that procedure before we would get the class `TD`, but this tag information can also be found in the recorded component now.

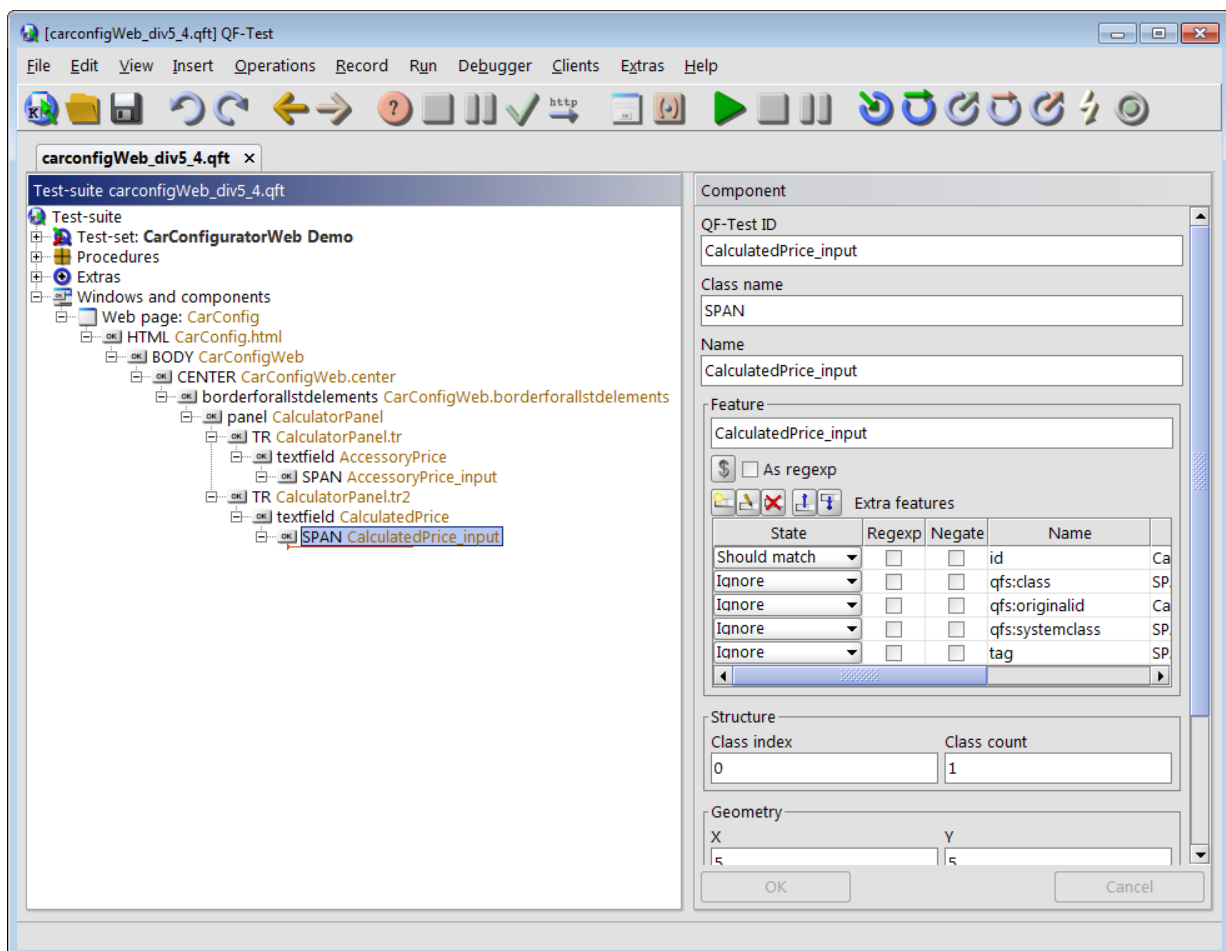


Figure 38.6: Recording of SPAN textfields

So, now let's extend the parameters of our procedure call of

`qfs.web.ajax.installCustomWebResolver`. We need to map that `textfield` value to QF-Test's generic class `TextField`. Therefore we extend the parameter `genericClasses` to `textfield=TextField,button=Button`. As you can see the `genericClasses` parameter holds a comma-separated list.

Once you delete the previously recorded component, re-run the procedure `qfs.web.ajax.installCustomWebResolver`, reload the web page and re-record that component again. Then you will get the following recording:

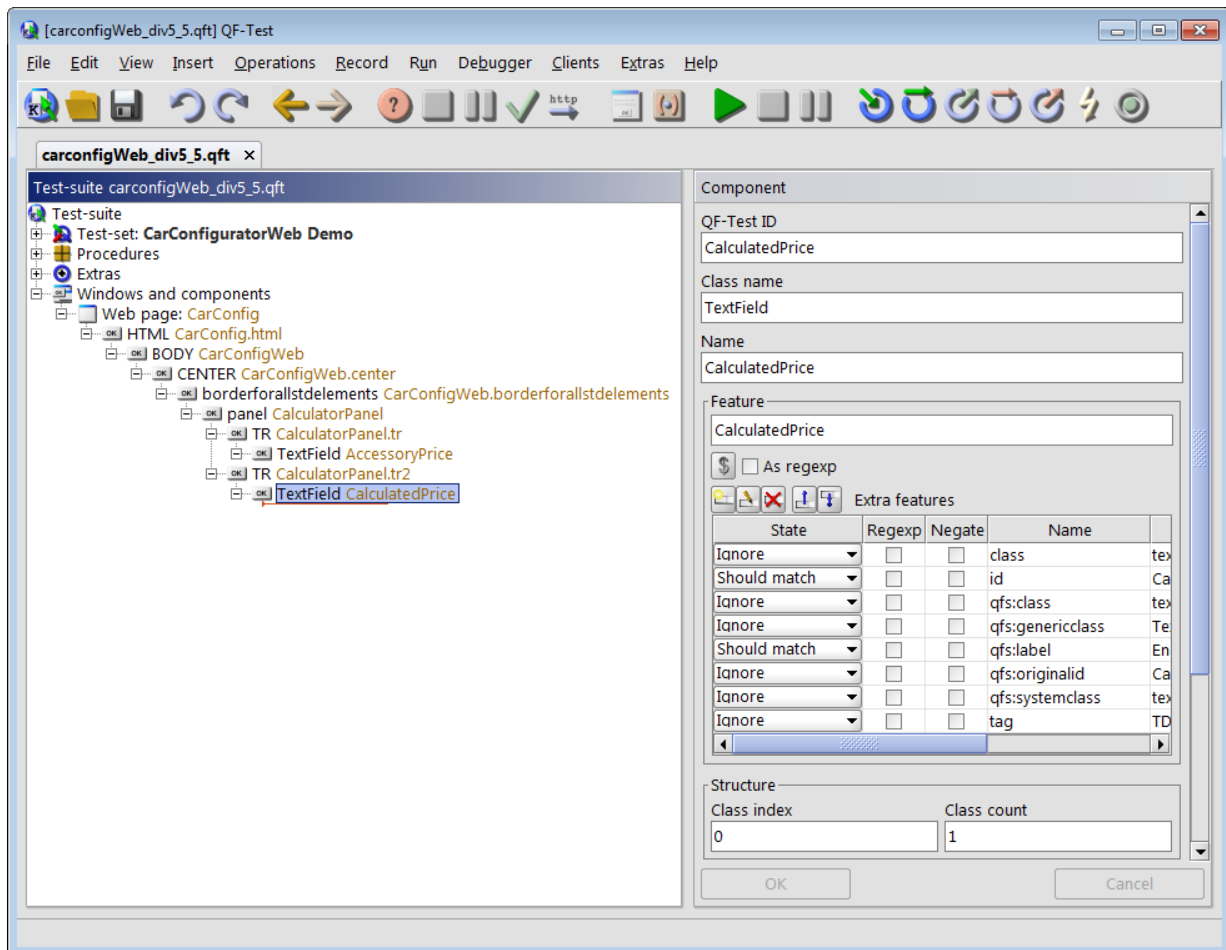


Figure 38.7: Recording TextFields in WebCarConfigurator

You can see that we got rid of at least one level of components and the QF-Test specific attributes as the extra feature `qfs:label` as has been automatically assigned to that new component.

Mapping of container elements like tables

The previously described approach will work for most standard components as buttons or checkboxes. But besides those components there are also complex components in our GUI. Those components represent data and we would like to address their content by QF-Test's sub-item syntax. Those components could be tables, trees or lists. For those components we need to map the dedicated class as well as the item class.

Our example is the table showing the cars of the WebCarConfigurator. Therefore we need to record some clicks on the shown cars and analyze the recording. The standard recording looks like this:

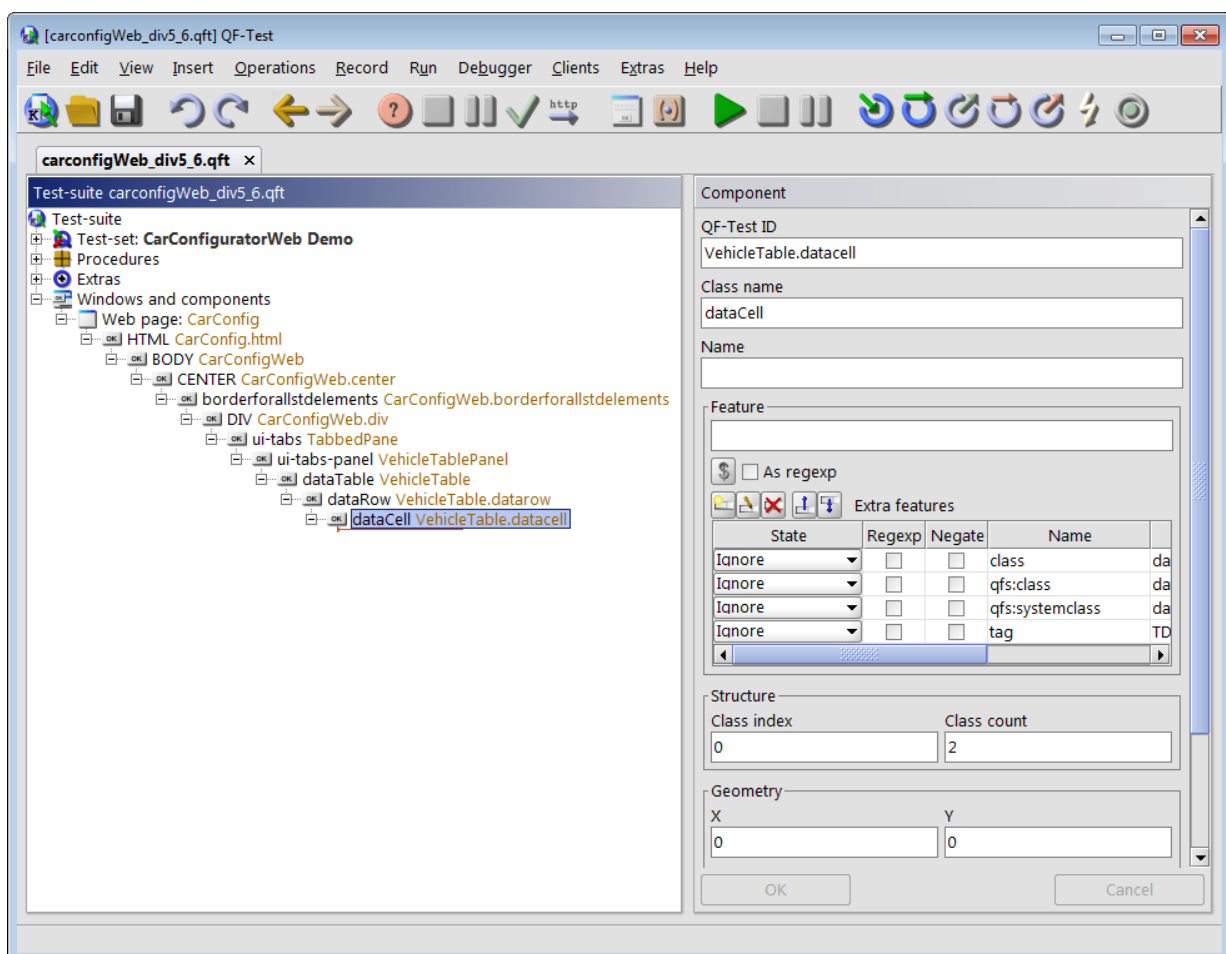


Figure 38.8: Recording of table in WebCarConfigurator

The click was recorded on a `dataCell` component, which is child of a `dataRow` component, which is part of a `dataTable` component. The recorded `dataCell` component contains an extra feature `tag` of the value `TD`. The `dataRow` component has the

value `TR`, and the `dataTable` has the value `TABLE`. Those classes are a consequence of the parameter `goodCssClasses` and its value `%.*` from the previous call of the `CustomWebResolver` procedure. Without calling that procedure upfront, we would get classes like `TD`, `TR` or `TABLE`.

Once we simply select those nodes and observe the component highlighting in the SUT we could notice, that a `TD` node represents a cell, a `TR` node represents a row of a table and a `TABLE` node represents an entire table. Exactly those nodes need to be investigated now in order to create a good mapping to generic classes.

Let's start with the `TABLE` node. This node has a `class` attribute with the value `dataTable`. This seems to be a clear sign, that any `dataTable` represents tables. So we select the call of the `qfs.web.ajax.installCustomWebResolver` again and we need extend the parameter `genericClasses` to `dataTable=Table,textField=TextField,button=Button`.

The next step is the row of the table. After selecting the `TR` node we can see that there is another `class` attribute with the value `dataRow`. This seems to be a clear case again. Now we need to add that value the procedure call of `qfs.web.ajax.installCustomWebResolver` again and we extend the parameter `genericClasses` to `dataRow=TableRow,dataTable=Table,textField=TextField,button=Button`.

Last but not least we need to analyze the `TD` node. Suprisingly we can find another `class` attribute with the value `dataCell`. Then let's add this to our procedure call as usual. Now `genericClasses` looks like `dataCell=TableCell,dataRow=TableRow,dataTable=Table,textField=TextField,button=Button`. Note, that you are allowed to enter line breaks after each comma to ensure a better readability.

The next step is again, to delete the previously recorded component, run the `qfs.web.ajax.installCustomWebResolver`, re-load the web page and re-record a click again.

As result we get a click with the typical QF-Test item syntax on a component like `VehicleTable@Model&0` (or any other row, depending what you have clicked on). In the recorded components you will just see the `Table` object and no child component anymore as those will be treated as items of a table by QF-Test from now on.

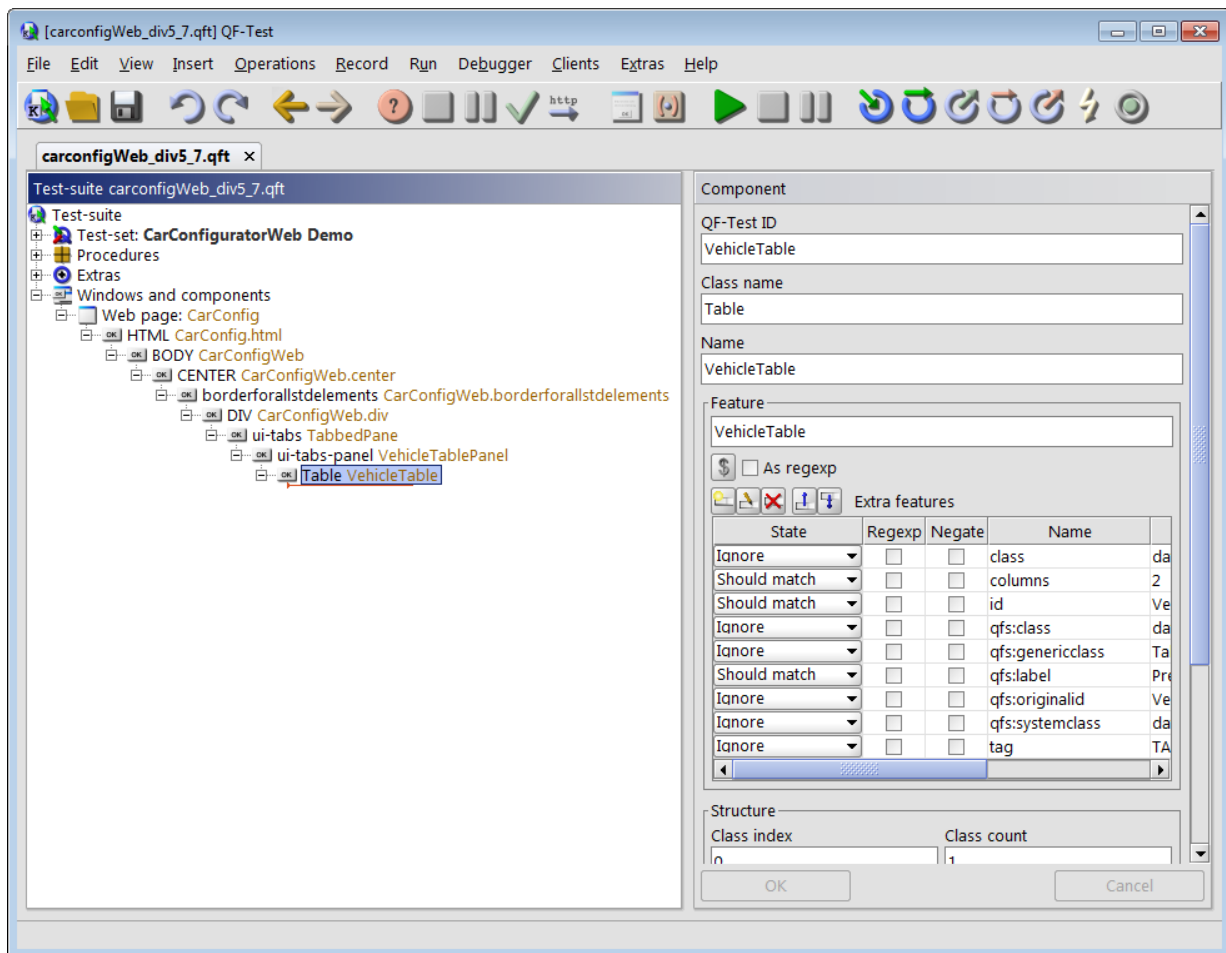


Figure 38.9: Recording of resolved table item in WebCarConfigurator

In [chapter 46](#)⁽⁷⁹⁰⁾ you can find a full list of any possible container and item classes for lists, trees and any other component.

The next steps will be to analyze the menu items, the tabfolder etc.. As this section could become very very long we have provided a fully resolved example in the demo test-suite `qftest-4.0.8/demo/carconfigWeb/carconfigWeb_en.qft` in the test-case `CarConfiguratorWeb Demo.Usage of CustomWebResolver.Create Vehicle with CustomWebResolver.`

Chapter 39

Extension APIs

QF-Test provides some extension APIs that let you extend its functionality. The interfaces can conveniently be implemented in Jython, Groovy or Java. In the latter case, the compiled classes should be put in a jar file and placed in the plugin directory (see [section 37.2^{\(620\)}](#)) so Jython or Groovy can be used to glue things together.

39.1 Resolver hooks

Note

Resolvers have quite a history in QF-Test. In version 2.0 SWT support was added to QF-Test. The `ComponentNameResolver` interface of previous QF-Test versions had been built for AWT/Swing `Components` and was not portable. It was replaced with the more generic `NameResolver` interface and associated `ResolverRegistry` methods.

With QF-Test version 3.1, both `NameResolver` and `FeatureResolver` were replaced with more elegant versions named - for lack of alternatives - `NameResolver2` and `FeatureResolver2`. These modern versions pass the default name or feature determined by QF-Test directly to the resolver's method which greatly simplifies implementation of the interface.

Similarly, the old `ItemNameResolver` that was only available for Swing was replaced with the new `ItemNameResolver2` with a matching `ItemValueResolver2`, which work for all engines. For backwards compatibility all the old interfaces and methods are still available and will continue to work, but they should no longer be used in new tests and are no longer documented here.

39.1.1 Overview over the supported kinds of resolvers

This extension API lets you install hooks that can modify the way QF-Test recognizes and records components and items. This is a very powerful feature that gives you fine-grained control over QF-Test's component management. It is far less complicated than it looks. Using the `resolvers` module described in [section 39.1.12^{\(689\)}](#), a full featured resolver can be implemented and installed with a minimum of code.

The following kinds of hooks are currently provided:

- A `ClassNameResolver` can change the class that QF-Test records for a component. For Swing, JavaFX and SWT its use is rather limited, because it is only allowed to return an actual base class of the component's class. However, for web applications this is perhaps the most important of all resolvers, because it can assign arbitrary custom class names to DOM nodes, thus turning deeply nested hierarchies of DIV nodes into something useful. See also [section 5.4^{\(41\)}](#) about the pseudo class hierarchy for web applications.
- A `NameResolver2` can change (or provide) the name of a GUI element as set with `setName()` for AWT/Swing, `setId()` or `fx:id` attribute for JavaFX, `setData(name, ...)` for SWT or via the 'ID' attribute of a DOM node for web applications. This can be very useful when setting names in the source code is not an option, like for third-party code or when child components of complex components are not readily accessible. For example, QF-Test provides a name resolver for Swing's `JFileChooser` dialog which you can read more about in the Tutorial's chapter about the standard library.
- An `IdResolver` can change, suppress or provide the 'ID' attribute of a DOM node at an early stage which makes it very efficient.
- A `FeatureResolver2` can provide a feature for a GUI element.
- An `ExtraFeatureResolver` can provide extra features for a GUI element.
- An `ItemNameResolver2` can change (or provide) the textual representation of the index for addressing a sub-item of a complex component.
- An `ItemValueResolver2` can change (or provide) the textual representation of the value of a sub-item of a complex component as used for a Check text or retrieved via a Fetch text node.
- A `TreeTableResolver` helps QF-Test recognize `TreeTable` components. A `TreeTable` is a mixture between a table and a tree. It is not a standard Swing component, but most `TreeTables` are implemented similarly by using a tree as the renderer component for one column of the table. Once QF-Test recognizes a `TreeTable` as such, it treats the row indexes of all table cells as tree indexes which

Web

Swing

is a lot more useful in that context than standard table row indexes. In addition, geometry information for cells in the tree column is based on tree nodes instead of table cells.

- An `InterestingParentResolver` influences which components will be treated as interesting or ignorable by QF-Test recording.

Multiple resolvers with different tasks can be created and registered at run-time. The respective interfaces can be implemented directly by Java classes. However, this would make the code of your application depend on QF-Test classes. The preferred alternative is to implement the resolver interfaces in Jython or Groovy, ideally via the Jython `resolvers` module or the Groovy `Resolvers` class. That way the whole mechanism can be strictly separated from the SUT and will not interfere with the build process.

Before we explain how to (un-)register resolvers, we'll take a look at their interfaces.

39.1.2 The `ClassNameResolver` interface

The `de.qfs.apps.qftest.extensions.ClassNameResolver` interface consists of a single method:

`String getClassName(Object element, String name)`

Determine the name of the class of a GUI element.

Parameters

<code>element</code>	The element to determine the class name for.
<code>name</code>	The original class name that QF-Test would use without a resolver.

Returns	The class name to use or null if the element is not handled by this resolver.

After QF-Test determines the class name of a GUI element the registered `ClassNameResolvers` get a chance to override this class name. The first resolver that returns a non-null value determines the outcome. If no resolvers are registered or all of them return null the original class name is used. The resolver is free to return any arbitrary class name. Those names will be treated as normal classes in QF-Test internal methods.

For performance reasons classes are cached so the resolver will only be called at most once for each element. If you change your resolver you need to re-load or to close and re-open the area which shows the component.

For web applications QF-Test uses a pseudo class hierarchy which is described in [section 5.4^{\(41\)}](#).

3.1+

Web

39.1.3 The `GenericClassNameResolver` interface

The `de.qfs.apps.qftest.extensions.GenericClassNameResolver` interface consists of a single method:

String getGenericClassName(Object element, String name)

Determine the name of the generic class of a GUI element.

Parameters

element	The element to determine the generic class name for.
name	The original generic class name that QF-Test would use without a resolver. May be null.

Returns	The generic class name to use or null if the element is not handled by this resolver. An empty string to suppress the generic class determined by QF-Test.

After QF-Test determines the generic class name of a GUI element the registered `GenericClassNameResolvers` get a chance to override this generic class name. The first resolver that returns a non-null value determines the outcome. If no resolvers are registered or all of them return null the original generic class name is used.

For performance reasons classes are cached so the resolver will only be called at most once for each element. If you change your resolver you need to re-load or to close and re-open the area which shows the component.

39.1.4 The `NameResolver2` interface

The `de.qfs.apps.qftest.extensions.NameResolver2` interface consists of a single method:

String getName(Object element, String name)

Determine the name of a GUI element.

Parameters

element	The element to determine the name for.
name	The original name that QF-Test would use without a resolver.

Returns	The name to use or null if the element is not handled by this resolver. Returning an empty string suppresses the original name.

After QF-Test determines the name of a GUI element the registered `NameResolvers` get a chance to override or suppress this name. The first resolver that returns a non-null

4.0+

3.1+

value determines the outcome. If no resolvers are registered or all of them return null the original name is used.

In some cases it may be desirable to suppress an element's name, for example for names which are not unique or which - even worse - vary depending on the situation. To do so, `getName` should return the empty string.

39.1.5 The `IdResolver` interface

The `de.qfs.apps.qftest.extensions.IdResolver` interface consists of a single method:

`String getId(DomNode node, String id)`

Determine the ID of a `DomNode`. The resolved ID will be cached and can later be retrieved via `node.getId()`, whereas `node.getAttribute("id")` always returns the original, unmodified 'ID' attribute.

Parameters

`node`

The `DomNode` to determine the ID for.

`id`

The ID that QF-Test has determined, possibly with suppressed numerals, depending on the setting of the option Eliminate all numerals from 'ID' attributes⁽²⁹⁴⁾. To implement the resolver based on the original 'ID' attribute, simply fetch this with `node.getAttribute("id")`.

Returns

The ID or null if the element is not handled by this resolver. Returning an empty string will suppress or hide the node's actual ID.

When QF-Test registers the nodes of a web page's DOM it also caches the "id" attribute of those nodes. Since many web applications are based on automatically generated IDs, some processing is often required, similar to the suppression of numerals that QF-Test does automatically, depending on the option Eliminate all numerals from 'ID' attributes⁽²⁹⁴⁾.

The 'ID' attribute of a node can show up in multiple places, most notably as the node's Name⁽⁵⁴⁷⁾ (depending on the option Turn 'ID' attribute into name where "unique enough"⁽²⁹⁴⁾), its Feature⁽⁵⁴⁷⁾ and an Extra feature⁽⁵⁴⁷⁾. Thus it is much more efficient to implement an `IdResolver` to modify the ID once, than to implement separate `Name`-, `Feature`- and `ExtraFeatureResolvers`. More importantly, changing a node's 'ID' attribute can have a major impact on whether the attribute is unique and QF-Test's mechanism for using an ID as a Name takes uniqueness into account, so an `IdResolver` is allowed to return non-unique IDs whereas a `NameResolver2` is not.

39.1.6 The FeatureResolver2 interface

The `de.qfs.apps.qftest.extensions.FeatureResolver2` interface consists of a single method:

String getFeature(Object element, String feature)

Determine the feature of a GUI element.

Parameters

element	The element to determine the feature for.
feature	The original feature that QF-Test would use without a resolver.

Returns	The feature to use or null if the element is not handled by this resolver. Returning an empty string suppresses the original feature.

After QF-Test determines the feature of a GUI element the registered `FeatureResolvers` get a chance to override or suppress this feature. The first resolver that returns a non-null value determines the outcome. If no resolvers are registered or all of them return null the original feature is used.

To suppress an element's feature, `getFeature` should return the empty string.

39.1.7 The ExtraFeatureResolver interface

The `de.qfs.apps.qftest.extensions.ExtraFeatureResolver` interface consists of a single method:

ExtraFeatureSet getExtraFeatures(Object element, ExtraFeatureSet features)

Determine extra features for a GUI element.

Parameters

element	The element to determine the extra features for.
features	The extra features determined by QF-Test itself, an empty set in case there are none. These can be modified in place or ignored and a different <code>ExtraFeatureSet</code> returned.

Returns	The original features modified in place or a different set, null if the element is not handled by this resolver. To suppress all the element's original extra features return an empty <code>ExtraFeatureSet</code> .

After QF-Test has determined the extra features for a GUI element, the registered `ExtraFeatureResolvers` will get a chance to override these features. In contrast to other resolvers, QF-Test does not stop when the first resolver returns a non-null value. Instead it passes its result as input to the next resolver which makes it possible to register several `ExtraFeatureResolvers` that handle different extra features. If no resolvers are registered or all of them return null, QF-Test will proceed to use the original set.

Of course, in order to be able to implement the `getExtraFeatures` method properly, you need to know the details for the API of the classes involved, namely `ExtraFeature` and `ExtraFeatureSet`. These are explained below. For an example implementation please see [example 39.6^{\(691\)}](#).

Instances of the class `de.qfs.apps.qftest.shared.data.ExtraFeature` represent one extra feature for a GUI element, comprising its name and value along with information about whether the feature is expected to match, whether it is a regular expression and whether the match should be negated. For possible states the class defines the constants `STATE_IGNORE`, `STATE_SHOULD_MATCH` and `STATE_MUST_MATCH`.

ExtraFeature (String name, String value)

Create a new `ExtraFeature` with the default state `STATE_IGNORE`.

Parameters

name	The name of the <code>ExtraFeature</code> .
value	The value of the <code>ExtraFeature</code> .

ExtraFeature (int state, String name, String value)

Create a new `ExtraFeature` with a given state.

Parameters

state	The state of the <code>ExtraFeature</code> . Possible values are <code>STATE_IGNORE</code> , <code>STATE_SHOULD_MATCH</code> , <code>STATE_MUST_MATCH</code> .
name	The name of the <code>ExtraFeature</code> .
value	The value of the <code>ExtraFeature</code> .

int getState()

Get the state of the `ExtraFeature`.

Returns The state of the `ExtraFeature`.

void setState(int state)

Set the state of the `ExtraFeature`.

Parameters

state	The state to set.

String getName()

Get the name of the ExtraFeature.

Returns The name of the ExtraFeature.

void setName(String name)

Set the name of the ExtraFeature.

Parameters**name** The name to set.

String getValue()

Get the value of the ExtraFeature.

Returns The value of the ExtraFeature.

void setValue(String value)

Set the value of the ExtraFeature.

Parameters**value** The value to set.

boolean getRegexp()

Get the regexp state of the ExtraFeature.

Returns Whether the ExtraFeature's value is a regular expression.

void setRegexp(boolean regexp)

Set the regexp state of the ExtraFeature.

Parameters**regexp** The regexp state to set.

boolean getNegate()

Get the negate state of the ExtraFeature.

Returns The negate state of the ExtraFeature.

void setNegate(boolean negate)

Set the negate state of the ExtraFeature.

Parameters**negate** The negate state to set.

The class `de.qfs.apps.qftest.shared.data.ExtraFeatureSet` collects ExtraFeatures into set:

ExtraFeatureSet ()

Create a new, empty ExtraFeatureSet.

```
void add(ExtraFeature extraFeature)
```

Add an extra feature to the set, potentially replacing a feature with the same name.

Parameters

extraFeature	The extra feature to add.

```
ExtraFeature get(String name)
```

Get an extra feature from the set.

Parameters

name	The name of the extra feature to get.

Returns	The extra feature or null if no feature by that name is stored in the set.

```
ExtraFeature remove(String name)
```

Remove an extra feature from the set.

Parameters

name	The name of the extra feature to remove.

Returns	The extra feature that was removed or null if no feature by that name was stored in the set.

```
ExtraFeature[] toArray()
```

Get all extra features in the set.

Returns	An array of the contained extra features, sorted by name.

39.1.8 The `ItemNameResolver2` interface

Unless you use the `resolvers` module described in [section 39.1.12^{\(689\)}](#), an `ItemNameResolver2` must be registered with the `ItemRegistry` described in [section 39.2.4^{\(710\)}](#) and not the `ResolverRegistry`.

The `de.qfs.apps.qftest.extensions.items.ItemNameResolver2` interface consists of a single method:

3.1+

Note

String getItemName(Object element, Object item, String name)

Determine the name of an item of a complex GUI element that will be used for the textual representation of the item's index.

Parameters

element	The GUI element to which the item belongs.
item	The item to get the name for. Its type depends on the GUI element and the registered <code>ItemResolvers</code> as described in section 39.2.5⁽⁷¹³⁾ .
name	The original name that QF-Test would use without a resolver.

Returns The name to use or null if the resolver does not handle this element or item.

After QF-Test determines the name for an item's index the registered `ItemNameResolvers` get a chance to override. The first resolver that returns a non-null value determines the outcome. If no resolvers are registered or all of them return null the original name is used.

39.1.9 The `ItemValueResolver2` interface

Unless you use the `resolvers` module described in [section 39.1.12^{\(689\)}](#), an `ItemValueResolver2` must be registered with the `ItemRegistry` described in [section 39.2.4^{\(710\)}](#) and not the `ResolverRegistry`.

The `de.qfs.apps.qftest.extensions.items.ItemValueResolver2` interface consists of a single method:

String getItemValue(Object element, Object item, String value)

Determine the value of an item of a complex GUI element that will be used for its textual representation in a [Check text^{\(470\)}](#) or a [Fetch text^{\(499\)}](#) node.

Parameters

element	The GUI element to which the item belongs.
item	The item to get the value for. Its type depends on the GUI element and the registered <code>ItemResolvers</code> as described in section 39.2.5⁽⁷¹³⁾ .
value	The original value that QF-Test would use without a resolver.

Returns The value to use or null if the resolver does not handle this element or item.

3.1+

Note

After QF-Test determines the value for an item's index the registered `ItemValueResolvers` get a chance to override. The first resolver that returns a non-null value determines the outcome. If no resolvers are registered or all of them return null the original value is used.

39.1.10 The `TreeTableResolver` interface

Note

This interface is irrelevant for SWT. Multi-column SWT trees are automatically supported by QF-Test.

A `de.qfs.apps.qftest.extensions.TreeTableResolver` must implement the following two methods:

`JTree getTree(JTable table)`

Determine the tree component used to implement a `TreeTable`.

Parameters

table The `JTable` component to determine the tree for.

Returns The tree or null if the `JTable` is a plain table and not a `TreeTable`.

`int getTreeColumn(JTable table)`

Determine the column index of the tree component in a `TreeTable`. Most implementations place the tree in the first column, in which case the index is 0.

Parameters

table The `JTable` component to determine the tree's column index for.

Returns The column index or -1 if the `JTable` is a plain table and not a `TreeTable`. The column index must always be given in the table's model coordinates, not in view coordinates.

Most `TreeTableResolvers` are trivial to implement. The following Jython example works well for the `org.openide.explorer.view.TreeTable` component used in the popular netBeans IDE, provided that the resolver is registered for the `TreeTable` class:

```
def getTreeMethod(table):
    return table.getCellRenderer(0,0)
def getColumn(table):
    return 0
resolvers.addTreeTableResolver("treetableResolver", getTreeMethod, \
    getColumn, "org.openide.explorer.view.TreeTable")
```

Example 39.1: `TreeTableResolver` for the netBeans IDE.

39.1.11 The InterestingParentResolver interface

A `de.qfs.apps.qftest.extensions.InterestingParentResolver` must implement the following method:

Boolean isInterestingParent(Object parent, boolean interesting)

Determine whether a parent element is interesting.

Parameters

parent The (direct or indirect) parent element.

interesting Whether QF-Test considers the parent interesting without resolver.

Returns Boolean.TRUE if interesting, Boolean.FALSE if not, null if this resolver cannot tell either way.

39.1.12 Simplified resolver creation with the `resolvers` module

Resolvers of all kinds can be implemented in Java, Jython or Groovy and then registered with the `ResolverRegistry`. Before describing this relatively complex process in detail in the following section, we'll show some resolver examples implemented with the help of the `resolvers` module provided with QF-Test. Unless you want to understand how the `resolvers` module itself works or want to implement a resolver in Java, these examples should be sufficient to get you going and skip the following sections.

Except for the more complex `ItemResolver` and `Checker` interfaces described in the following sections, which are not supported by the `resolvers` module, a resolver consists of a single method that needs to be implemented (one and a half in case of the `TreeTableResolver`). What makes resolvers difficult at Java level is the need for describing such a method as an interface that has to be implemented by a class. Then an instance of that class has to be created and registered for use by QF-Test. If you don't get it right on first try, that instance has to be deregistered before a new instance from a new class can be created and instance thereof registered, otherwise there may be interference between the two versions of the resolver. Add some code for error handling and you've got many times more glue code than actual "flesh".

Fortunately, methods are first level objects in Jython, as are Groovy closures. This and the fact that the glue code is practically the same for all resolvers make it possible to move the tedious parts out of the way into a module and concentrate on the actual task at hand. Following is an example of a `NameResolver2` for `JMenuItems` that promotes the label of a menu item to a component name for menus that have no name of their own:

```
def getName(menuItem, name):
    if not name:
        return menuItem.getLabel()
resolvers.addNameResolver2("menuItems", getName,
                           "javax.swing.JMenuItem")
```

Example 39.2: Simple Jython NameResolver2 for JMenuItem

As you see it took only four lines of code to do that. The first three lines define a `getName` method for the resolver. The fourth line calls the `addNameResolver2` function of the `resolvers` module which is always automatically available in Jython SUT script nodes. The three parameters are an identifier for the resolver, the actual method and one or more optional targets, in this case the class name of the components for which we want to register the resolver.

Give it a try. Copy the example above into an SUT script⁽⁴¹⁹⁾ node and execute it. If your application is based on SWT instead of Swing, replace `getLabel()` with `getText()` and `javax.swing.JMenuItem` with `org.eclipse.swt.MenuItem`. Then record some menu actions into a new, empty test-suite. You'll find that all recorded menu item components without their own name will now have names set according to their labels. If `setName` is not used in your application and the labels of menu items are more or less static while the structure of the items changes often, this could even be a very useful thing. To install the resolver permanently, just move the SUT script node right after the Wait for client to connect⁽⁴³⁸⁾ node of your SUT startup sequence.

The same can be done in Groovy via the `resolvers` object that is also always available and has the same API. It expects a closure to create a resolver from as follows:

```
def getName = {menuItem, name ->
    if (name == null) {
        return menuItem.getLabel()
    }
}
resolvers.addNameResolver2("menuItems", getName,
                           "javax.swing.JMenuItem")
```

Example 39.3: Simple Groovy NameResolver2 for JMenuItem

It is possible to register a resolver for multiple component classes at once:

```
def getName(com, name):
    return com.getText()
resolvers.addNameResolver2("labels", getName, "javax.swing.JLabel",
                           "javax.swing.AbstractButton")
```

Example 39.4: Registering a NameResolver2 for multiple target classes

The other resolvers work similarly. Following is an example for an `ItemNameResolver2` that makes the ID of a `JTable` column available as an item index:

```
def getColumnId(table, item, name):
    if item[1] < 0: # whole column addressed
        id = table.getColumnModel().getColumn(item[1]).getIdentifier()
    if id:
        return str(id)
resolvers.addItemNameResolver2("tableColumnId", getColumnId,
                                "javax.swing.table.TableColumn")
```

Example 39.5: `ItemNameResolver2` for `JTable` columns

`FeatureResolvers` and `ExtraFeatureResolvers` are handled just like `NameResolvers`, but `ExtraFeatureResolvers` are a bit more complex, though not overly so, as the following example shows:

```
from de.qfs.apps.qftest.shared.data import ExtraFeature, ExtraFeatureSet
def getExtraFeatures(node, features):
    features.add(ExtraFeature(ExtraFeature.STATE_SHOULD_MATCH,
                              "myname", "myvalue"))
    return features
resolvers.addExtraFeatureResolver("silly example", getExtraFeatures,
                                  "INPUT")
```

Example 39.6: `ExtraFeatureResolver` that adds an extra feature to all `INPUT` nodes

`TreeTableResolvers` for Swing `TreeTables` are created and registered in a slightly different way:

```
def getTree(table):
    return table.getTree()
def getColumn(table):
    return 0
resolvers.addTreeTableResolver("treeTable", getTree, getColumn,
                                "my.package.TreeTable")
```

Example 39.7: `TreeTableResolver` for Swing `TreeTable` with optional `getColumn` method

The above example shows a typical `TreeTableResolver`. Since almost all `TreeTables` display the tree in the first column of the table, the `getColumn` method is optional. If none is provided a default implementation for the first column is used:

```
def getTree(table):
    return table.getTree()
resolvers.addTreeTableResolver("treeTable", getTree, None,
                               "my.package.TreeTable")
```

Example 39.8: Simplified `TreeTableResolver`

If no dedicated `getTree` method is available, the cell renderer of the column containing the tree (typically 0) might work, as it is typically derived from `JTree`.

```
def getTree(table):
    return table.getCellRenderer(0,0)
resolvers.addTreeTableResolver("treeTable", getTree,
                               "my.package.TreeTable")
```

Example 39.9: Simple `TreeTableResolver` using `getCellRenderer` method

Following is the complete API of the `resolvers` module:

```
addNameResolver2(String name, Method method, Object
target=None, ...)
```

Register a `NameResolver2` for the given target(s), based on the given method. If another resolver was previously registered under the given name, unregister that first.

Parameters

name	The name under which to register the resolver.
method	The method that implements the resolver's <code>getName()</code> method.
target	One or more optional targets to register the resolver for. Each can be any of the following:

- An individual component
- The fully qualified name of a class

If no target is given a global resolver for all components is registered.

```
addIdResolver(String name, Method method, Object target=None, ...)
```

Register an `IdResolver` for the given target(s), based on the given method. If another resolver was previously registered under the given name, unregister that first.

Parameters

name	The name under which to register the resolver.
method	The method that implements the resolver's <code>getId()</code> method.
target	One or more optional targets to register the resolver for. Each can be any of the following:

- An individual component
- The fully qualified name of a class

If no target is given a global resolver for all components is registered.

```
addFeatureResolver2(String name, Method method, Object target=None, ...)
```

Register a `FeatureResolver2` for the given target(s), based on the given method. If another resolver was previously registered under the given name, unregister that first.

Parameters

name	The name under which to register the resolver.
method	The method that implements the resolver's <code>getFeature()</code> method.
target	One or more optional targets to register the resolver for. Each can be any of the following:

- An individual component
- The fully qualified name of a class

If no target is given a global resolver for all components is registered.

```
addExtraFeatureResolver(String name, Method method, Object  
target=None, ...)
```

Register an `ExtraFeatureResolver` for the given target(s), based on the given method. If another resolver was previously registered under the given name, unregister that first.

Parameters

name	The name under which to register the resolver.
method	The method that implements the resolver's <code>getExtraFeature()</code> method.
target	One or more optional targets to register the resolver for. Each can be any of the following: • An individual component • The fully qualified name of a class

If no target is given a global resolver for all components is registered.

```
addItemNameResolver2(String name, Method method, Object  
target)
```

Register an `ItemNameResolver2` for the given target(s), based on the given method. If another resolver was previously registered under the given name, unregister that first.

Parameters

name	The name under which to register the resolver.
method	The method that implements the resolver's <code>getItemName()</code> method.
target	The target to register the resolver for. It can be any of the following: • An individual component • The fully qualified name of a class

```
addItemValueResolver2(String name, Method method, Object target=None, ...)
```

Register an `ItemValueResolver2` for the given target(s), based on the given method. If another resolver was previously registered under the given name, unregister that first.

Parameters

name	The name under which to register the resolver.
method	The method that implements the resolver's <code>getItemValue()</code> method.
target	The target to register the resolver for. It can be any of the following: • An individual component • The fully qualified name of a class

```
addTreeTableResolver(String name, Method getTable, Method getColumn=None, Object target=None)
```

Register a `TreeTableResolver` for the given target(s), based on the given methods. If another resolver was previously registered under the given name, unregister that first.

Parameters

name	The name under which to register the resolver.
getTable	The method that implements the resolver's <code>getTable()</code> method.
getColumn	The method that implements the resolver's <code>getColumn()</code> method.
target	One optional target to register the resolver for. It can be any of the following: • An individual component • The fully qualified name of a class

If no target is given a global resolver for all components is registered.

```
addClassNameResolver(String name, Method method, Object  
target=None, ...)
```

Register a `ClassNameResolver` for the given target(s), based on the given method. If another resolver was previously registered under the given name, unregister that first.

Parameters

name	The name under which to register the resolver.
method	The method that implements the resolver's <code>getClassName()</code> method.
target	One or more optional targets to register the resolver for. Each can be any of the following:

- An individual component
- The fully qualified name of a class

If no target is given a global resolver for all components is registered.

```
addGenericClassNameResolver(String name, Method method, Object  
target=None, ...)
```

Register a `GenericClassNameResolver` for the given target(s), based on the given method. If another resolver was previously registered under the given name, unregister that first.

Parameters

name	The name under which to register the resolver.
method	The method that implements the resolver's <code>getClassName()</code> method.
target	One or more optional targets to register the resolver for. Each can be any of the following:

- An individual component
- The fully qualified name of a class

If no target is given a global resolver for all components is registered.

```
removeResolver(String name)
```

Unregister a resolver from all the targets it was registered to.

Parameters

name	The name under which the resolver was registered.

39.1.13 The ResolverRegistry

Now it is time to take a look at the Java API for registering resolvers on which the `resolvers` module is built. The singleton class `de.qfs.apps.qftest.extensions.ResolverRegistry` is the central agent for registering and removing name resolvers.

All kinds of resolvers can be registered on individual components or on classes. Resolvers registered on a component will only be called if their service is required for that specific component. Resolvers registered on a class will be called for every object of that class or a class derived from it.

`NameResolvers`, `FeatureResolvers` and `TreeTableResolvers` can also be registered globally so they will be called for each and every name or feature. This is similar to but more efficient than registering them on the `java.lang.Object` class.

If multiple resolvers are registered globally or registered on the same object or class, the resolver added last will be called first.

The `ResolverRegistry` API is pretty straightforward:

static ResolverRegistry instance()

There can only ever be one `ResolverRegistry` object and this is the method to get hold of this singleton instance.

Returns The singleton `ResolverRegistry` instance.

static String getElementName(Object element)

This static method should be used instead of `com.getName()` or `widget.getData()` by resolvers that need to operate based on existing names of elements, except for a `NameResolver2` that gets just the original name passed to its `getName` method. This method suppresses trivial or AWT/Swing specific component names that QF-Test handles specially.

Parameters

element The element to get the name for.

Returns The name of the element or `null` if a trivial or special name is suppressed.

static boolean isInstance(Object object, String className)

This static method should be used instead of `instanceof` (or `isinstance()` in Jython). It checks whether an object is an instance of a given class. The check is performed by name instead of through reflection, so conflicts with differing class loaders are prevented and there is no need to import the class to check against.

Parameters

object The object to check.

className The name of the class to test for.

Returns True if the object is an instance of the given class.

```
void registerNameResolver2(Object element, NameResolver2  
resolver)
```

Register a name resolver for an individual GUI element. The resolver does not prevent garbage collection and will be removed automatically when the element becomes unreachable.

Parameters

element	The GUI element to register for.
resolver	The resolver to register.

```
void unregisterNameResolver2(Object element, NameResolver2  
resolver)
```

Unregister a name resolver for an individual GUI element.

Parameters

element	The GUI element to unregister for.
resolver	The resolver to unregister.

```
void registerNameResolver2(String clazz, NameResolver2  
resolver)
```

Register a name resolver for a GUI element class.

Parameters

clazz	The name of the class to register for.
resolver	The resolver to register.

```
void unregisterNameResolver2(String clazz, NameResolver2  
resolver)
```

Unregister a name resolver for a GUI element class.

Parameters

clazz	The name of the class to unregister for.
resolver	The resolver to unregister.

```
void registerNameResolver2(NameResolver2 resolver)
```

Register a generic or global name resolver.

Parameters

resolver	The resolver to register.

```
void unregisterNameResolver2(NameResolver2 resolver)
```

Unregister a generic or global name resolver.

Parameters

resolver	The resolver to unregister.

void registerIdResolver(Object element, IdResolver resolver)

Register an ID resolver for an individual GUI element. The resolver does not prevent garbage collection and will be removed automatically when the element becomes unreachable.

Parameters

element	The GUI element to register for.
resolver	The resolver to register.

void unregisterIdResolver(Object element, IdResolver resolver)

Unregister an ID resolver for an individual GUI element.

Parameters

element	The GUI element to unregister for.
resolver	The resolver to unregister.

void registerIdResolver(String clazz, IdResolver resolver)

Register an ID resolver for a GUI element class.

Parameters

clazz	The name of the class to register for.
resolver	The resolver to register.

void unregisterIdResolver(String clazz, IdResolver resolver)

Unregister an ID resolver for a GUI element class.

Parameters

clazz	The name of the class to unregister for.
resolver	The resolver to unregister.

void registerIdResolver(IdResolver resolver)

Register a generic or global ID resolver.

Parameters

resolver	The resolver to register.

void unregisterIdResolver(IdResolver resolver)

Unregister a generic or global ID resolver.

Parameters

resolver	The resolver to unregister.

```
void registerFeatureResolver2(Object element, FeatureResolver2  
resolver)
```

Register a feature resolver for an individual GUI element. The resolver does not prevent garbage collection and will be removed automatically when the element becomes unreachable.

Parameters

element	The GUI element to register for.
resolver	The resolver to register.

```
void unregisterFeatureResolver2(Object element,  
FeatureResolver2 resolver)
```

Unregister a feature resolver for an individual GUI element.

Parameters

element	The GUI element to unregister for.
resolver	The resolver to unregister.

```
void registerFeatureResolver2(String clazz, FeatureResolver2  
resolver)
```

Register a feature resolver for a GUI element class.

Parameters

clazz	The name of the class to register for.
resolver	The resolver to register.

```
void unregisterFeatureResolver2(String clazz, FeatureResolver2  
resolver)
```

Unregister a feature resolver for a GUI element class.

Parameters

clazz	The name of the class to unregister for.
resolver	The resolver to unregister.

```
void registerFeatureResolver2(FeatureResolver2 resolver)
```

Register a generic or global feature resolver.

Parameters

resolver	The resolver to register.

```
void unregisterFeatureResolver2(FeatureResolver2 resolver)
```

Unregister a generic or global feature resolver.

Parameters

resolver	The resolver to unregister.

```
void registerExtraFeatureResolver(Object element,  
ExtraFeatureResolver resolver)
```

Register an extra feature resolver for an individual GUI element. The resolver does not prevent garbage collection and will be removed automatically when the element becomes unreachable.

Parameters

element	The GUI element to register for.
resolver	The resolver to register.

```
void unregisterExtraFeatureResolver(Object element,  
ExtraFeatureResolver resolver)
```

Unregister an extra feature resolver for an individual GUI element.

Parameters

element	The GUI element to unregister for.
resolver	The resolver to unregister.

```
void registerExtraFeatureResolver(String clazz,  
ExtraFeatureResolver resolver)
```

Register an extra feature resolver for a GUI element class.

Parameters

clazz	The name of the class to register for.
resolver	The resolver to register.

```
void unregisterExtraFeatureResolver(String clazz,  
ExtraFeatureResolver resolver)
```

Unregister an extra feature resolver for a GUI element class.

Parameters

clazz	The name of the class to unregister for.
resolver	The resolver to unregister.

```
void registerExtraFeatureResolver(ExtraFeatureResolver  
resolver)
```

Register a generic or global extra feature resolver.

Parameters

resolver	The resolver to register.

```
void unregisterExtraFeatureResolver(ExtraFeatureResolver  
resolver)
```

Unregister a generic or global extra feature resolver.

Parameters

resolver	The resolver to unregister.

```
void registerTreeTableResolver(Component com, TreeTableResolver resolver)
```

Register a TreeTable resolver for a specific component. The resolver will be removed automatically if the component becomes invisible.

Parameters

com	The component to register for.
resolver	The resolver to register.

```
void unregisterTreeTableResolver(Component com, TreeTableResolver resolver)
```

Unregister a TreeTable resolver for a specific component.

Parameters

com	The component to unregister for.
resolver	The resolver to unregister.

```
void registerTreeTableResolver(String clazz, TreeTableResolver resolver)
```

Register a TreeTable resolver for a component class.

Parameters

clazz	The name of the class to register for.
resolver	The resolver to register.

```
void unregisterTreeTableResolver(String clazz, TreeTableResolver resolver)
```

Unregister a TreeTable resolver for a component class.

Parameters

clazz	The name of the class to unregister for.
resolver	The resolver to unregister.

```
void registerTreeTableResolver(TreeTableResolver resolver)
```

Register a generic or global TreeTable resolver.

Parameters

resolver	The resolver to register.

```
void unregisterTreeTableResolver(TreeTableResolver resolver)
```

Unregister a generic or global TreeTable resolver.

Parameters

resolver	The resolver to unregister.

```
void unregisterResolvers(Object element)
```

Unregister all resolvers for a specific GUI element.

Parameters

element	The element to unregister for.

```
void unregisterResolvers(String clazz)
```

Unregister all resolvers for the class of a GUI element, value or renderer.

Parameters

clazz	The name of the class to unregister for.

39.1.14 Error handling

All exceptions thrown inside a name resolver will be caught and handled by the `ResolverRegistry`. However, instead of dumping a stack trace, the registry will only print a short message like "Exception inside NameResolver2" because some resolvers may be called very often, so for a buggy resolver printing a stack trace for every error could flood the net and the client terminal. Therefore name resolvers should include their own error handling. This can still generate a lot of output in some cases, but especially for scripts the output will be more useful than a Java stack trace.

39.2 Implementing custom item types with the `ItemResolver` interface

As described in chapter 6⁽⁵²⁾, QF-Test is able to dig below the given structure of GUI elements and work with sub-items that are not GUI components themselves, like the cells of a table, nodes in a tree or drawings on a canvas. Such items are implemented via the `ItemResolver` mechanism which can be used to implement your own custom items.

The `ItemResolver` interface is more complex than the simple `NameResolver2` or `FeatureResolver2` described in the previous section and cannot be implemented without solid programming skills and a thorough understanding of the underlying concepts. Also, `ItemResolvers` and `Checkers`, described in the next section, are closely related and need to be implemented together if you want to be able to perform checks for your items.

On the upside the whole mechanism is very powerful and once implemented and registered, your `ItemResolvers` will integrate smoothly with QF-Test so that there is no distinction between standard and custom items, so don't let yourself be deterred.

You can find demo implementations in the directory `qftest-4.0.8/Jython/Lib un-`

der QF-Test's root directory. The respective demo files are `ktable.py` and `gef.py`. Both resolvers are for SWT specific tables, but the concept is the same for all engines.

39.2.1 `ItemResolver` concepts

Before you start implementing an `ItemResolver` you need to determine the kinds of items that your GUI element might hold. There might be more than one kind, though the decision is arbitrary. For example, we implemented items for all standard tables of Swing, SWT and JavaFX so that an item can either be a table cell or a complete column. The latter is useful because it makes it possible to implement a check that checks a whole table column at once.

First you need to decide how to represent your items internally. You can use any kind of `Object` because QF-Test doesn't ever examine your internal representation itself, it just passes it to the methods of your `ItemResolver`. What works best depends on the API of the GUI element. If it already comes with its own concept for sub-items it may be best to reuse those classes.

The most important decision to make is how to represent the item to the user of QF-Test. As described in [chapter 6^{\(52\)}](#), the user can address an item via a numerical index or a textual one which can also be matched by a regular expression. You need to be able to provide a two-way mapping between an item and its index(es), i.e. you need to be able to answer the following two questions:

- Given an item, what is its numerical and its textual index?
- Given a numerical or textual index, which item matches that index best?

The issues involved in naming sub-items are the same as those for setting component names. Please take a thorough look at [section 5.6^{\(44\)}](#) and [section 5.7^{\(46\)}](#) before continuing.

An single numerical or textual index is represented by a `SubItemIndex` object. The current item concept supports addressing an item like a table cell via a primary and a secondary index, but in the future we hope to support indexes to any depth, so instead of using a path for a tree node it could be addressed with a mixed-type index in a form like `"tree@Root&1%Name:.*"`. Therefore the complete index is represented as an array of `SubItemIndex` objects, though currently limited to single or two-element arrays.

Most items have a geometry, i.e. a location and a size. The coordinates for an item are always calculated relative to the true upper left origin of the element, regardless of whether it is scrolled, so they are independent of the current scroll position of the element. For items where geometry is not applicable or cannot be determined, coordinates can be ignored and the methods `getItemLocation` and `getItemSize` should simply return `[0,0]`.

39.2.2 The `ItemResolver` interface

The methods of the interface `de.qfs.apps.qftest.extensions.items.ItemResolver` fall into three categories: Retrieving an item, mapping between an item and its index and retrieving miscellaneous information for - or performing actions on - an item.

Object `getItem(Object element, int x, int y)`

Get a sub-item for a GUI element at a given location. For items where geometry is not applicable or cannot be determined, the coordinates can be ignored and the item determined based on some other means like selection.

Parameters

element The GUI element to get the item for.

x The X coordinate relative to element.

y The Y coordinate relative to element.

Returns An arbitrary object representing an item or null if there is no item at the given location.

SubItemIndex[] `getItemIndex(Object element, Object item, int type)`

Get the `SubItemIndex(es)` for a sub-item of a GUI element.

Parameters

element The GUI element to which the item belongs.

item The item to get the index for.

type The type of index to get. Possible values are `INTELLIGENT`, `AS_STRING` and `AS_NUMBER`, all defined in the `SubItemIndex` class. Unless only one kind of index is supported, a textual index should be returned for `AS_STRING` and a numerical index for `AS_NUMBER`. If the type is `INTELLIGENT` you are free to return whatever best represents the given item, even a mixed index like a column title and a row index for a table cell.

Returns An array of `SubItemIndex` objects. Currently only single or two-element arrays are allowed.

Object getItemForIndex(Object element, SubItemIndex[] idx)

Get an item for a given sub-item index.

At the end of this procedure a call of `setIndexesResolved` might be required in case more than one index is resolved.**Parameters****element** The GUI element to get the item for.**idx** The sub-item index(es) for the item.**Returns** The item that best matches the given index.**Throws****IndexNotFoundException** If no item matches the given index. Use the constructor `de.qfs.apps.qftest.shared.exceptions.IndexNotFoundException` for this case.

void setIndexesResolved(int num)Notify the registry about how many item indexes were resolved during item resolution in `getItemForIndex`. If this method is not called at the end of `getItemForIndex`, the registry assumes one index.**Parameters****num** The number of indexes resolved.

String getItemValue(Object element, Object item)

Get the value of a sub-item to be used for a text check.

Parameters**element** The GUI element to which the item belongs.**item** The item whose value to get.**Returns** A string representing the value of the item, i.e. its contents, label, whatever.

int[] getItemLocation(Object element, Object item)

Get the location of a sub-item relative to its parent element.

Parameters**element** The GUI element to which the item belongs.**item** The item whose location to get.**Returns** The item's location as a two-element int array [x,y]. The location returned must always be relative to the upper left corner of the whole element, even if that corner is not currently visible, for example because it has been scrolled outside the visible area. For items without geometry simply return [0,0].

```
int[] getItemSize(Object element, Object item)
```

Get the size of a sub-item relative to its parent element.

Parameters

element The GUI element to which the item belongs.

item The item whose size to get.

Returns The item's size as a two-element int array [width,height].
For items without geometry simply return [0,0].

```
Boolean repositionMouseEvent(Object element, Object item, int[] pos)
```

Change the coordinates for a mouse event on a sub-item of a GUI element to a default location - typically the center - if the coordinates may be safely ignored during replay. This method is called only if the option Record MouseEvents without coordinates where possible ⁽²⁵⁶⁾ is active. Whether or not it is safe to override the event coordinates may depend on the original coordinates. For example, QF-Test repositions events on a tree node with positive coordinates pointing inside the node to its center whereas negative coordinates indicate a click on the expansion toggle and are left unchanged.

Parameters

element The GUI element to which the item belongs.

item The target item for the event.

pos A two-element int array of the form [x,y] with the coordinates of the event relative to the item. Its values can be modified in place. You can either set them to a specific coordinate or to [Integer.MAX_VALUE,Integer.MAX_VALUE] to ignore coordinates for this event so that it will later be replayed on the center of the target.

Returns Boolean.TRUE if the position was modified,
Boolean.FALSE if it was left unchanged. Null to signal that this resolver does not handle the element.

Throws

BadItemException If element and item types don't match up (should never happen).

```
Boolean scrollItemVisible(Object element, Object item, int x,
int y)
```

Scroll a GUI element so that an item becomes visible. If possible, the item should be made fully visible. In case the item doesn't fit into the visible region of the element, at least the given position should be shown. In most cases you can simply return null to let QF-Test handle scrolling. Sometimes however, scrolling cannot be implemented generically based on the item's geometry, because some items need special compensation, for example an SWT Table with a visible header. For GUI elements that cannot be scrolled, this method should simply return `Boolean.FALSE`. If `Boolean.TRUE` is returned, QF-Test will call the method again after a short interval because SWT sometimes interferes with scrolling. In the ideal case the second call should return `FALSE` because the position is already OK. After three attempts that return `TRUE` QF-Test gives up and signals an error.

Parameters

element	The GUI element to which the item belongs.
item	The item that must be made visible.
x	The X-coordinate of the position relative to the item that must always become visible.
y	The Y-coordinate of the position relative to the item that must always become visible.

Returns `Boolean.TRUE` if the scroll position of the element change, `Boolean.FALSE` if the position is unchanged scrolled or if the element cannot be scrolled. Null to signal that QF-Test should use its default mechanism and try to scroll the item itself.

```
int getItemCount(Object element, Object item)
```

Get the number of items in a GUI element or at the next item level. Though this method is currently unused, partly because the multi-level item concept for trees is not implemented yet, it should be implemented if possible. In the future it may be used for a 'Fetch count' node similar to `Fetch index`⁽⁵⁰²⁾ or `Fetch text`⁽⁴⁹⁹⁾.

Parameters

element	The GUI element for which to get the item count.
item	Null to get the number items at the top level of the element. An item to get the number of its sub-items.

Returns The number of items or -1 if there is no further sub-item level.

39.2.3 The class `SubItemIndex`

As explained in the previous section, a `de.qfs.apps.qftest.shared.data.SubItemIndex` represents a (partial) index for a sub-item of a complex GUI element. This class defines some constants with the following meanings:

STRING

This is a textual index

NUMBER

This is a numerical index

REGEXP

This is a regular expression to match a textual index

INTELLIGENT

When retrieving an index, use whichever type best suits the item

AS_STRING

Retrieve a textual index

AS_NUMBER

Retrieve a numerical index

It also provides the following methods:

`SubItemIndex(String index)`

Create a new `SubItemIndex` of type `STRING`.

Parameters

<code>index</code>	The textual index.

`SubItemIndex(int index)`

Create a new `SubItemIndex` of type `NUMBER`.

Parameters

<code>index</code>	The numerical index.

`String getIndex()`

Get the index as a String.

Returns The index converted to a String.

`String getType()`

Get the type of the index.

Returns The type of the index, one of `STRING`, `NUMBER` or `REGEXP`.

`int asNumber()`

Get the index as a number.

Returns The numerical index.**Throws****`IndexFormatException`** If the index is not of type `NUMBER` or cannot be parsed as an integer.

`boolean matches(String name)`

Test whether the index matches a given item name.

Parameters**`name`** The name to match.**Returns** True if the index is not numerical and matches the given name.**Throws****`IndexFormatException`** If the index contains a malformed regular expression.

39.2.4 The `ItemRegistry`

Once implemented and instantiated, your `ItemResolver` must be registered with the `ItemRegistry`. The class `de.qfs.apps.qftest.extensions.items.ItemRegistry` has the following interface:

`static ItemRegistry instance()`There can only ever be one `ItemRegistry` object and this is the method to get hold of this singleton instance.**Returns** The singleton `ItemRegistry` instance.

```
void registerItemResolver(Object element, ItemResolver  
resolver)
```

Register an `ItemResolver` for an individual GUI element.

Parameters

element The GUI element to register for. The resolver does not prevent garbage collection and will be removed automatically when the element becomes unreachable.

resolver The `ItemResolver` to register.

```
void unregisterItemResolver(Object element, ItemResolver  
resolver)
```

Unregister an `ItemResolver` for an individual GUI element.

Parameters

element The GUI element to unregister for.

resolver The `ItemResolver` to unregister.

```
void registerItemResolver(String clazz, ItemResolver resolver)
```

Register an `ItemResolver` for a class of GUI elements.

Parameters

clazz The class of GUI element to register for.

resolver The `ItemResolver` to register.

```
void unregisterItemResolver(String clazz, ItemResolver  
resolver)
```

Unregister an `ItemResolver` for a class of GUI elements.

Parameters

clazz The class of GUI element to unregister for.

resolver The `ItemResolver` to unregister.

```
void registerItemNameResolver2(Object element,  
ItemNameResolver2 resolver)
```

Register an `ItemNameResolver2` for an individual GUI element.

Parameters

element The GUI element to register for. The resolver does not prevent garbage collection and will be removed automatically when the element becomes unreachable.

resolver The `ItemNameResolver2` to register.

```
void unregisterItemNameResolver2(Object element,
ItemNameResolver2 resolver)
```

Unregister an `ItemNameResolver2` for an individual GUI element.

Parameters

element	The GUI element to unregister for.
resolver	The <code>ItemNameResolver2</code> to unregister.

```
void registerItemNameResolver2(String clazz, ItemNameResolver2
resolver)
```

Register an `ItemNameResolver2` for a class of GUI elements.

Parameters

clazz	The class of GUI element to register for.
resolver	The <code>ItemNameResolver2</code> to register.

```
void unregisterItemNameResolver2(String clazz,
ItemNameResolver2 resolver)
```

Unregister an `ItemNameResolver2` for a class of GUI elements.

Parameters

clazz	The class of GUI element to unregister for.
resolver	The <code>ItemNameResolver2</code> to unregister.

```
void registerItemValueResolver2(Object element,
ItemValueResolver2 resolver)
```

Register an `ItemValueResolver2` for an individual GUI element.

Parameters

element	The GUI element to register for. The resolver does not prevent garbage collection and will be removed automatically when the element becomes unreachable.
resolver	The <code>ItemValueResolver2</code> to register.

```
void unregisterItemValueResolver2(Object element,
ItemValueResolver2 resolver)
```

Unregister an `ItemValueResolver2` for an individual GUI element.

Parameters

element	The GUI element to unregister for.
resolver	The <code>ItemValueResolver2</code> to unregister.

```
void registerItemValueResolver2(String clazz,  
ItemValueResolver2 resolver)
```

Register an `ItemValueResolver2` for a class of GUI elements.

Parameters

clazz	The class of GUI element to register for.
resolver	The <code>ItemValueResolver2</code> to register.

```
void unregisterItemValueResolver2(String clazz,  
ItemValueResolver2 resolver)
```

Unregister an `ItemValueResolver2` for a class of GUI elements.

Parameters

clazz	The class of GUI element to unregister for.
resolver	The <code>ItemValueResolver2</code> to unregister.

39.2.5 Default item representations

For the implementation of the `ItemNameResolver2`, `ItemValueResolver2` and `Checker` interfaces it is important to know which kind of Object is used for the internal representation of an item. This internal representation will be passed to the methods `getItemName`, `getItemValue`, `getCheckData` and `getCheckDataAndItem`.

The following table lists the complex GUI elements and the default internal item representation used by QF-Test standard `ItemResolvers` for JavaFX.

GUI element class	Item type
Accordion	Integer index
ChoiceBox	Integer index
ComboBox	Integer index
ListView	Integer index
TabPane	Integer index
TableView	int array [column,row] with row < 0 to represent a whole column
TableHeaderRow	Integer column index
TextArea	Integer line
TreeView	TreeItem object

Table 39.1: Internal item representations for JavaFX GUI elements

The following table lists the complex GUI elements and the default internal item representation used by QF-Test standard `ItemResolvers` for Swing.

39.2. Implementing custom item types with the `ItemResolver` interface 714

GUI element class	Item type
<code>JComboBox</code>	<code>Integer</code> index
<code>JList</code>	<code>Integer</code> index
<code>JTabbedPane</code>	<code>Integer</code> index
<code>JTable</code>	<code>int</code> array [column,row] with row < 0 to represent a whole column
<code>JTableHeader</code>	<code>Integer</code> column index
<code>JTextArea</code>	<code>Integer</code> line
<code>JTree</code>	<code>TreePath</code> path

Table 39.2: Internal item representations for Swing GUI elements

SWT

The following table lists the complex GUI elements and the default internal item representation used by QF-Test standard `ItemResolvers` for SWT.

GUI element class	Item type
<code>CCombo</code>	<code>Integer</code> index
<code>Combo</code>	<code>Integer</code> index
<code>CTabFolder</code>	<code>Integer</code> index
<code>List</code>	<code>Integer</code> index
<code>StyledText</code>	<code>Integer</code> line
<code>TabFolder</code>	<code>Integer</code> index
<code>Table</code>	<code>int</code> array [column,row] or just <code>Integer</code> column to represent a whole column
<code>Text</code>	<code>Integer</code> line
<code>Tree</code>	<code>Object</code> array [<code>Integer</code> column, <code>TreeItem</code> row] or just <code>Integer</code> column to represent a whole column

Table 39.3: Internal item representations for SWT GUI elements

Web

The following table lists the complex GUI elements and the default internal item representation used by QF-Test standard `ItemResolvers` for Web.

GUI element class	Item type
<code>SELECT</code> node	<code>OPTION</code> node
<code>TEXTAREA</code> node	<code>Integer</code> line

Table 39.4: Internal item representations for DOM nodes

39.3 Implementing custom checks with the `Checker` interface

3.1+

Checks are one of QF-Test's most useful features. Test automation would be mostly useless without the ability to verify the results of simulated actions. However, the default set of Checks available in QF-Test is naturally limited to checking the most common attributes of standard components. For special attributes or custom components you can resort to read the value in an SUT script⁽⁴¹⁹⁾ and use the method `rc.checkEqual()` to compare it against the expected value. Such an SUT script is perfectly fine, it performs and integrates well, is flexible and can be modularized by placing it inside a Procedure⁽³⁷⁸⁾. It has two major disadvantages however: It cannot be recorded and it is daunting for non-programmers.

With the help of the API described in this section the default set of Checks in QF-Test can be extended. In fact, QF-Test's own new-style checks are implemented exactly this way. By implementing and registering a `Checker` for a given type of GUI element and possibly item you can create your own checks that can be recorded and replayed just like the standard ones.

To make this as simple as possible, QF-Test handles everything from showing the check in the check popup menu, fetching the check data, recording the respective Check node to store that data, sending the data back to the SUT upon replay, fetching the then current check data, comparing it to the expected value and reporting success or mismatch. All that is left for you to do is tell QF-Test which checks your `Checker` implements and for each of these provide the check data on request.

Illustrative examples are provided at the end of the chapter and in the test-suite `carconfig_en.qft`, located in the directory `demo/carconfig` in your QF-Test installation.

39.3.1 The `Checker` interface

The interface `de.qfs.apps.qftest.extensions.checks.Checker` must be implemented in order to add custom checks for your application. The associated helper classes and interfaces are documented in the subsequent sections.

`CheckType[] getSupportedCheckTypes(Object element, Object item)`

Get the types of checks supported for the given GUI element and optional item.

Parameters

`element` The GUI element for which to get the available checks.

`item` An optional item within the element to check. Its type depends on the GUI element and the registered `ItemResolvers` as described in [section 39.2.5^{\(713\)}](#).

Returns An array of `CheckTypes` supported by your checker. The first element is the default for recording a check with a left-click. If the item is null, return only those kinds of checks that can be applied to the whole GUI element. Otherwise it is best to provide all available checks because even though the user may have right-clicked on an item, he may still want to record a check on the whole GUI element.

`CheckData getCheckData(Object element, Object item, CheckType type)`

Get the check data for the current state of the GUI element or item. This method is used during replay.

Parameters

`element` The GUI element for which to get the check data.

`item` An optional item within the element to check. Its type depends on the GUI element and the registered `ItemResolvers` as described in [section 39.2.5^{\(713\)}](#).

`type` The type of check to perform.

Returns The check data for the current state of the GUI element itself, in case item is null, or the given item within the element. The kind of check to perform is specified via the type parameter, which should normally be one of those formerly returned by `getSupportedCheckTypes`. If you cannot perform the requested check for the given type and target, return null.

Pair `getCheckDataAndItem(Object element, Object item, CheckType type)`

Get the check data for the current state of the GUI element or item and also the item that the check actually applies to. This method is used during recording, where QF-Test not only needs to know, which data to record, but also whether to record the check for the GUI element as a whole or for an item. To implement this method without duplicating any code, call your own `getCheckData` method to retrieve the check data.

Parameters

element	The GUI element for which to get the check data.
item	An optional item within the element to check. Its type depends on the GUI element and the registered <code>ItemResolvers</code> as described in section 39.2.5⁽⁷¹³⁾ .
type	The type of check to perform.

Returns	A <code>Pair</code> with the check data for the current state of the GUI element, and the item this check should be performed on, which may be null.

39.3.2 The class `Pair`

The class `de.qfs.lib.util.Pair` for the return value of `getCheckDataAndItem` is a simple utility class that often comes in handy for grouping two values. You'll only need its constructor, but of course you can also read its values:

Pair(`Object first, Object second`)

Create a new `Pair`.

Parameters

first	The first object. May be null.
second	The second object. May be null.

Object `getFirst()`

Get the first object of the `Pair`.

Returns	The first object.

Object `getSecond()`

Get the second object of the `Pair`.

Returns	The second object.

39.3.3 The `CheckType` interface and its implementation `DefaultCheckType`

A `de.qfs.apps.qftest.extensions.checks.CheckType` encapsulates information for a specific kind of check. It combines a `CheckDataType` with an identifier and provides a user-friendly representation of the check for the check popup menu. Unless you need to provide multi-lingual representations of the check you should never implement this interface yourself, but simply instantiate a `de.qfs.apps.qftest.extensions.checks.DefaultCheckType` instead:

`DefaultCheckType(String identifier, CheckDataType dataType, String description)`

Create a new `DefaultCheckType`.

Parameters

identifier	The identifier for the check. The QF-Test standard checks all use lower case identifiers. To prevent conflicts, simply start your custom check identifiers with a capital letter.
dataType	The <code>CheckDataType</code> required for your check data.
description	The description to show in the check popup menu for this check.

For completeness sake, following are the methods of the `CheckType` interface:

`String getIdentifier()`

Get the identifier for the check type.

Returns The identifier for the check type.

`CheckDataType getDataType()`

Get the `CheckDataType` for the check type.

Returns The data type for the check type.

`String getDescription()`

Get the localized description to show for this check type in the check popup menu.

Returns The description for the check type.

39.3.4 The class `CheckDataType`

The class `de.qfs.apps.qftest.extensions.checks.CheckDataType` is similar to an `Enum`. It defines a number of constant `CheckDataType` instances that simply serve to identify the kind of data that a check operates on. Each constant corresponds to one or more of the available Check nodes of QF-Test.

Besides serving as a constant identifier, a `CheckDataType` has no public attributes or methods and you cannot add any new `CheckDataTypes`. If you want to implement a check of a kind that does not fit the existing data types you'll need to convert your data so that it does, for example by using a string representation. The following `CheckDataType` constants are defined:

STRING

A single string. Used by the Check text⁽⁴⁷⁰⁾ node.

STRING_LIST

A list of string items, like the cells in a table column. Used by the Check items⁽⁴⁸⁰⁾ node.

SELECTABLE_STRING_LIST

A list of selectable string items, like the elements of a list. Used by the Check selectable items⁽⁴⁸⁴⁾ node.

BOOLEAN

A boolean state, either true or false. Used by the Boolean check⁽⁴⁷⁶⁾ node.

GEOMETRY

A set of four integer values for X and Y coordinates, width and height. Not all have to be defined. Used by the Check geometry⁽⁴⁹⁴⁾ node.

IMAGE

An image of a whole component or item or a sub-region thereof. Used by the Check image⁽⁴⁸⁸⁾ node.

39.3.5 The class `CheckData` and its subclasses

The class `de.qfs.apps.qftest.shared.data.check.CheckData` and its subclasses, all from the same package, complete the `Checker` API. A `CheckData` encapsulates the actual data for a check, must be returned from `Checker.getCheckData()` and is used to exchange this check data between the SUT and QF-Test. There is one concrete `CheckData` subclass corresponding to each `CheckDataType`. You'll only ever need to use their constructors, so that's what we'll list here. Only two of these classes are publicly available so far:

StringCheckData(String identifier, String value)Create a new `StringCheckData`.**Parameters**

identifier	The identifier for the check type. Should normally match the identifier of the <code>type</code> argument passed to <code>Checker.getCheckData</code> .
value	The actual value for the check, a <code>String</code> .

BooleanCheckData(String identifier, boolean value)Create a new `BooleanCheckData`.**Parameters**

identifier	The identifier for the check type. Should normally match the identifier of the <code>type</code> argument passed to <code>Checker.getCheckData</code> .
value	The actual value for the check, a boolean state.

StringItemsCheckData(String identifier, String[] values)Create a new `StringItemsCheckData`.**Parameters**

identifier	The identifier for the check type. Should normally match the identifier of the <code>type</code> argument passed to <code>Checker.getCheckData</code> .
values	The actual value for the check, an array of <code>Strings</code> .

SelectableItemsCheckData(String identifier, Object[][] values)Create a new `SelectableItemsCheckData`.**Parameters**

identifier	The identifier for the check type. Should normally match the identifier of the <code>type</code> argument passed to <code>Checker.getCheckData</code> .
values	The actual value for the check, an array of arrays with a <code>String</code> , a <code>Boolean</code> for the regexp flag and a <code>Boolean</code> for the selected flag.

```
GeometryCheckData(String identifier, int x, int y, int width,
int height)
```

Create a new `GeometryCheckData`.

Parameters

identifier	The identifier for the check type. Should normally match the identifier of the <code>type</code> argument passed to <code>Checker.getCheckData</code> .
x	The x-coordinate for the check.
y	The y-coordinate for the check.
width	The width for the check.
height	The height for the check.

```
ImageCheckData(String identifier, ImageRep image, int xOffset,
int yOffset, int subX, int subY, int subWidth, int subHeight)
```

Create a new `ImageCheckData`.

Parameters

identifier	The identifier for the check type. Should normally match the identifier of the <code>type</code> argument passed to <code>Checker.getCheckData</code> .
image	The image for the check. See section 39.8 ⁽⁷³⁷⁾ .
xOffset	An optional x-offset.
yOffset	An optional y-offset.
subX	The X-coordinate of an optional check region.
subY	The Y-coordinate of an optional check region.
subWidth	The Width of an optional check region.
subHeight	The Height of an optional check region.

Furthermore you can define an optional algorithm for an `ImageCheckData`.

```
void setAlgorithm(String algorithm)
```

Sets an algorithm. A detailed description can be found in [Details about the algorithm for image comparison](#)⁽⁷⁶⁹⁾.

39.3.6 The CheckerRegistry

Once implemented and instantiated, your `Checker` must be registered with the `CheckerRegistry`. The class `de.qfs.apps.qftest.extensions.checks.CheckerRegistry` has the following interface:

static `CheckerRegistry instance()`

There can only ever be one `CheckerRegistry` object and this is the method to get hold of this singleton instance.

Returns The singleton `CheckerRegistry` instance.

void `registerChecker(Object element, Checker resolver)`

Register a `Checker` for an individual GUI element.

Parameters

element The GUI element to register for. The checker does not prevent garbage collection and will be removed automatically when the element becomes unreachable.

resolver The `Checker` to register.

void `unregisterChecker(Object element, Checker resolver)`

Unregister a `Checker` for an individual GUI element.

Parameters

element The GUI element to unregister for.

resolver The `Checker` to unregister.

void `registerChecker(String clazz, Checker resolver)`

Register a `Checker` for a class of GUI elements.

Parameters

clazz The class of GUI element to register for.

resolver The `Checker` to register.

void `unregisterChecker(String clazz, Checker resolver)`

Unregister a `Checker` for a class of GUI elements.

Parameters

clazz The class of GUI element to unregister for.

resolver The `Checker` to unregister.

39.3.7 Custom checker example

The following Jython SUT script illustrates how to put everything together. Let's say you have a Java Swing application and want to check all labels which reside in a panel at once. To this end, your custom checker needs to iterate over all components contained in the panel and its children respectively, identify the labels and generate a list of all their text contents. In QF-Test notation, this means you need to create a `CheckDataType.STRING_LIST` check type and return the data in an `StringItemsCheckData` object:

```

from de.qfs.apps.qftest.extensions import ResolverRegistry
from de.qfs.apps.qftest.extensions.checks import CheckerRegistry, \
    Checker, DefaultCheckType, CheckDataType
from de.qfs.apps.qftest.extensions.items import ItemRegistry
from de.qfs.apps.qftest.shared.data.check import StringItemsCheckData
from de.qfs.lib.util import Pair
from java.lang import String
import jarray
componentClass = "javax.swing.JPanel"
allLabelsCheckType = DefaultCheckType("AllLabels",
    CheckDataType.STRING_LIST,
    "All labels in the panel")
class AllLabelsChecker(Checker):
    def __init__(self):
        pass
    def getSupportedCheckTypes(self, com, item):
        return jarray.array([allLabelsCheckType], DefaultCheckType)
    def getCheckData(self, com, item, checkType):
        if allLabelsCheckType.getIdentifier() == checkType.getIdentifier():
            labels = self._findLabels(com)
            labels = map(lambda l: l.getText(), labels)
            values = jarray.array(labels, String)
            return StringItemsCheckData(checkType.getIdentifier(), values)
        return None
    def getCheckDataAndItem(self, com, item, checkType):
        data = self.getCheckData(com, item, checkType)
        if data is None:
            return None
        return Pair(data, None)
    def _findLabels(self, com, labels=None):
        if labels is None:
            labels = []
        if ResolverRegistry.instance().isInstance(com, "javax.swing.JLabel"):
            labels.append(com)
        for c in com.getComponents():
            self._findLabels(c, labels)
        return labels
    def unregister():
        try:
            CheckerRegistry.instance().unregisterChecker(
                componentClass, allLabelsChecker)
        except:
            pass
    def register():
        unregister()
        global allLabelsChecker
        allLabelsChecker = AllLabelsChecker()
        CheckerRegistry.instance().registerChecker(
            componentClass, allLabelsChecker)
register()

```

Example 39.10: Check all labels in a panel

After running that script once, you'll find a new entry "All labels in the panel" among the entries in the check type menu as soon as you right click on a `JPanel` component while being in recording mode (cf. [section 4.3^{\(33\)}](#)). If you want to use the `allLabelsChecker` all over your client application, you can put the above SUT script behind your Wait for client to connect node in the Setup sequence. Otherwise, you may register the checker only when it is actually needed as shown above and remove it afterwards by means of another SUT script:

```
from de.qfs.apps.qftest.extensions.checks import CheckerRegistry
global allLabelsChecker
def unregister():
    try:
        CheckerRegistry.instance().unregisterChecker(
            "javax.swing.JPanel", allLabelsChecker)
    except:
        pass
unregister()
```

Example 39.11: Remove the label checker

39.4 Working with the Eclipse Graphical Editing Framework (GEF)

3.2+

The Graphical Editing Framework (GEF) is a set of Eclipse plugins for creating editors that support visual editing of arbitrary models. This framework is very popular and QF-Test has supported recording and playback of GEF items for a long time (since about version 2.2). It is also a good example for the power of the `ItemResolver` concept (see [section 39.2^{\(703\)}](#)), because the `gef` Jython module contains an implementation of just that interface.

The `gef` module can deal with GEF editors at a generic level and even support several editors at once. Though reasonable item names are provided out of the box also for GMF applications, there are limits to what can be determined automatically. Depending on the underlying model classes, there might still remain some work for you: Implementing custom resolvers to provide useful names and values for your items.

39.4.1 Recording GEF items

The actual GEF component is the `FigureCanvas`. This control displays `Figures` which represent `EditParts`. When recording a mouse click on such an element, QF-Test does not register a pure Mouse event node for the canvas component with the cor-

responding (x,y) position but tries to recognize the object under the mouse cursor. For example, the recorded QF-Test component ID may look like

```
canvas@/Diagram/My ship/ShipLargeCargo (wine)
canvas@Connection-2
canvas@/Diagram/Rectangle 16329001
```

where "canvas" is the QF-Test ID of the `FigureCanvas` component, followed by the item index of the recognized `EditPart` (see [chapter 6^{\(52\)}](#)). `EditParts` reside in a tree like hierarchy which is reflected in the index by a path separator '/'. The names of the individual items are generated as follows:

- The item name is `getModel().toString()` unless it contains a hash value (e.g. `NodeImpl@2d862`).
- QF-Test tries to extract a name for the item from the model ("My ship" in the above examples).
- The class name along with a description gets recorded, e.g. "ShipLargeCargo (wine)".
- If there's no description, an index is appended to the class name when there's more than one item of that class, e.g. "Connection-2" for the third connection.
- The root `EditPart` always reads "Diagram".

As one can imagine, those generated item names may not always be useful. For example, items might be deleted so that the recorded index is not longer valid. Or the generated item name is unstable as "Rectangle 16329001" in the GEF Shapes example: The number is random and when restarting the application a different one will be created. Three options exist to overcome the problem:

- Instead of working with a textual index, you can try to go with a numerical one. To this end, open the recording options and set the 'Sub-Item format' to 'Number' (see [section 29.2.4^{\(267\)}](#)). This is probably not satisfying because a numerical index like `/0/1` tells nothing about an item.
- Get in touch with your developers and convince them to provide a useful implementation of the `toString()` method of the item's model. It would make live easy for you, but only if the developers are cooperative.
- Write an `ItemNameResolver2`. This is the tough course but unfortunately the most likely scenario. It is covered in the next section.

39.4.2 Implementing a GEF ItemNameResolver2

As stated in [section 39.1](#)⁽⁶⁷⁸⁾, an `ItemNameResolver2` is the hook to change or provide names for items. To get started, insert a new Jython [SUT script](#)⁽⁴¹⁹⁾ in the [Extras](#)⁽³⁴²⁾ node with the following code:

```
def getItemName(canvas, item, name):
    print "name: %s" %name
    print "item: %s" %(item.__class__)
    model = item.getModel()
    print "model: %s" %(model.__class__)
    resolvers.addItemNameResolver2("myGefItemNames", getItemName,
        "org.eclipse.draw2d.FigureCanvas")
```

Example 39.12: Get started with a GEF ItemNameResolver2

To ease the installation of the resolver we use the `resolvers` module described in [section 39.1.12](#)⁽⁶⁸⁹⁾. The resolver gets registered for the `FigureCanvas` class where the items reside. The default item name provided by QF-Test is supplied as the last argument to our function `getItemName()`. Now run the script, press the record button and then simply move the mouse over your figures on the canvas - supposing you have created some of them previously. Note that this first resolver implementation does nothing but print out some information into the terminal, something like

```
name: Rectangle 16329001
item: org.eclipse.gef.examples.shapes.parts.ShapeEditPart
model: org.eclipse.gef.examples.shapes.model.RectangularShape
```

The question is now: Does the model of the GEF `EditPart` provide any property that might be used as name for the item? The answer in the case of the GEF Shapes example is "No", and hopefully you are in a better situation with your application. To find out insert a line

```
print dir(model)
```

in the `getItemName()` function and run the script again. Now you will also see the methods of the model when moving the mouse over the items in record mode. With a bit of luck you will find methods like `getId()` or `getLabel()` and can create a resolver like this:

```
def getItemName(canvas, item, name):
    model = item.getModel()
    return model.getId()
    resolvers.addItemNameResolver2("myGefItemNames", getItemName,
        "org.eclipse.draw2d.FigureCanvas")
```

Example 39.13: A simple ItemNameResolver2

Let's go back to the GEF Shapes example where we don't have such useful methods. Only geometry information is available for the shapes and that is not really helpful. At least we can distinguish between rectangles and ellipses. To make the item names unique we simply add a child index as shown in the following resolver:

```
def getItemName(canvas, item, name):
    name = None
    shapes = "org.eclipse.gef.examples.shapes"
    diagrammEditPart = shapes + ".parts.DiagramEditPart"
    shapeEditPart = shapes + ".parts.ShapeEditPart"
    connectionEditPart = shapes + ".parts.ConnectionEditPart"
    ellipticalShape = shapes + ".model.EllipticalShape"
    rectangularShape = shapes + ".model.RectangularShape"
    if qf.isInstance(item, shapeEditPart):
        siblings = item.getParent().getChildren()
        for i in range(len(siblings)):
            if (item == siblings[i]):
                if qf.isInstance(item.getModel(), ellipticalShape):
                    name = "Ellipse " + str(i)
                elif qf.isInstance(item.getModel(),
                                    rectangularShape):
                    name = "Rectangle " + str(i)
    elif qf.isInstance(item, connectionEditPart):
        source = item.getSource()
        target = item.getTarget()
        sourceName = getItemName(canvas, source, str(source.getModel()))
        targetName = getItemName(canvas, target, str(target.getModel()))
        name = "Connection " + sourceName + " " + targetName
    elif qf.isInstance(item, diagrammEditPart):
        name = "Diagram"
    return name
resolvers.addItemNameResolver2("shapesItemNames", getItemName,
                                "org.eclipse.draw2d.FigureCanvas")
```

Example 39.14: An ItemNameResolver2 for GEF Shapes

With this resolver in place, the item index for a rectangle becomes

```
/Diagram/Rectangle 1
```

where the trailing number is the child index of the item. The above implementation also provides names for the connections by calling `getItemName()` recursively for the source and the target item of the connection. Checking the types with `qf.isInstance()` (see [section 37.7^{\(639\)}](#)) will save you the need to import the GEF classes, something that is not trivial.

Once your resolver is working fine you should move the script into your `Setup(350)` sequence right behind the `Wait for client to connect(438)` node. This way the resolver will be registered automatically when the SUT starts.

39.4.3 Implementing a GEF ItemValueResolver2

Usually a GEF editor consists of two parts. Having focused so far on the canvas where you draw the figures, we now take a look at the palette where you select the kind of figure to draw (e.g. 'Rectangle', 'Ellipse' or 'Connection'). Its entries look like tool buttons but actually the palette is a `FigureCanvas` too. You will be glad to know that this one works out of the box, that is without implementing an `ItemNameResolver2`. When you click for example on the 'Rectangle' button, QF-Test recognizes a

```
/Palette Root/Palette Container (Shapes)/Palette Entry
(Rectangle)
```

item. What will happen when you record a check (cf. [section 4.3^{\(33\)}](#)) for the 'Object value' for this button? You may expect to get the button text 'Rectangle' but in fact the value of this item is

```
Palette Entry (Rectangle)
```

The reason is that by default name and value of an item are the same. To alter this behavior and provide customized values you need to implement an `ItemValueResolver2`. This interface is very similar to the `ItemNameResolver2` above. For the palette we can code the following one:

```
def getItemValue(canvas, item, value):
    value = None
    paletteEditPart = \
        "org.eclipse.gef.ui.palette.editparts.PaletteEditPart"
    if qf.isInstance(item, paletteEditPart):
        value = item.getModel().getLabel()
    return value
resolvers.addItemValueResolver2("shapesItemValues", getItemValue,
    "org.eclipse.draw2d.FigureCanvas")
```

Example 39.15: An `ItemValueResolver2` for the GEF Shapes palette

The method `getLabel()` returns the text as displayed in the palette.

39.5 Test-run listeners

3.1+

Once registered with the current run-context via `rc.addTestRunListener`, an implementation of the `TestRunListener` interface will get notified whenever test execution enters or exits a node and when a problem occurs. An illustrative example is provided in the test-suite `TestRunListener.qft`, located in the directory `demo/runlistener` in your QF-Test installation. Best deactivate the debugger before running the whole test-suite.

Note

A variant of the `TestRunListener` interface called `DaemonTestRunListener` can be used to monitor a test-run remotely via the daemon API. It is described in [section 40.2.5^{\(755\)}](#).

The run listener API consists of the following classes:

39.5.1 The `TestRunListener` interface

The interface `de.qfs.apps.qftest.extensions.qftest.TestRunListener` has to be implemented and registered with a run-context via `rc.addTestRunListener()`.

Note

To implement the interface you can also derive from the class `de.qfs.apps.qftest.extensions.qftest.AbstractTestRunListener` which provides empty implementations of all methods so you only need to implement those methods you are interested in.

`void runStarted(TestRunEvent event)`

Notify the listener that a test-run was started.

Parameters

event	The event containing the details, irrelevant in this case.

`void runStopped(TestRunEvent event)`

Notify the listener that a test-run was stopped.

Parameters

event	The event containing the details, irrelevant in this case.

`void nodeEntered(TestRunEvent event)`

Notify the listener that a node is being entered.

Parameters

event	The event containing the details.

`void nodeExited(TestRunEvent event)`

Notify the listener that a node was just exited.

Parameters

event	The event containing the details.

`void problemOccurred(TestRunEvent event)`

Notify the listener that a problem occurred.

Parameters

event	The event containing the details.

39.5.2 The class `TestRunEvent`

The class `de.qfs.apps.qftest.extensions.qftest.TestRunEvent` holds information about the currently executed nodes and the current error state. It defines the following constants for execution states and error levels: `STATE_NOT_IMPLEMENTED`, `STATE_SKIPPED`, `STATE_OK`, `STATE_WARNING`, `STATE_ERROR` and `STATE_EXCEPTION`. The first two states apply only to Test-set⁽³²¹⁾ and Test-case⁽³¹⁴⁾ nodes.

In the `runStopped` method you can also check whether the test run has been interrupted or completed normally. Therefore the constants `STATE_RUN_INTERRUPTED` and `STATE_RUN_TERMINATED` are defined.

`TestSuiteNode getNode()`

Get the current node

Returns The current node or null for `runStarted` and `runStopped`.

`TestSuiteNode[] getPath()`

Get the whole tree path for the current node.

Returns The path for the current node as seen from the run-log, equivalent to the execution stack, or null for `runStarted` and `runStopped`. The last node in the array is the current node.

`int getState()`

Get the overall execution state for the current node.

Returns The overall propagating state for the current node, the highest error level of this node and its children, possibly limited by Maximum error level⁽³³⁵⁾ attributes. Available only for `nodeExited` and `problemOccurred`.

`int getLocalState()`

Get the execution state for the current node.

Returns The local execution state for the current node, the highest error level of this node and its children regardless of whether this state propagates to the top. Available only for `nodeExited` and `problemOccurred`. For `Test-set` and `Test-case` nodes the local state can also be one of `STATE_SKIPPED` or `STATE_NOT_IMPLEMENTED`.

String getMessage()

Get the current error message.

Returns The message for the current warning, error or exception. Only available for `problemOccurred`.

String getExceptions()

Get the exception count for the exited node.

Returns The total exception count for the node just exited. Only available for `nodeExited`.

String getErrors()

Get the error count for the exited node.

Returns The total error count for the node just exited. Only available for `nodeExited`.

String getWarnings()

Get the warning count for the exited node.

Returns The total warning count for the node just exited. Only available for `nodeExited`.

39.5.3 The class `TestSuiteNode`

A `de.qfs.apps.qftest.extensions.qftest.TestSuiteNode` is a representation of a QF-Test node that is currently being executed, including information about the type of node, its name, comment, etc.

String getSuite()

Get the test-suite to which the node belongs.

Returns The full path of the test-suite to which the node belongs.

String getType()

Get the type of the node.

Returns The type of node, the last part of the name of the class implementing the node's behavior.

String getName()

Get the name of the node.

Returns The name of the node or null if that type of node does not have a 'Name' attribute.

String getReportName()

Get the report name of the node.

Returns The expanded report name for Test-set⁽³²¹⁾ and Test-case⁽³¹⁴⁾ nodes. The normal name for other nodes or null if undefined.

String getTreeName()

Get the tree name of the node.

Returns The name of the node as displayed in the test-suite tree.

String getId()

Get the ID of the node.

Returns The ID of the node.

String getComment()

Get the comment of the node.

Returns Get the expanded comment of the node.

39.6 ResetListener

4.0.3+

During test development you sometimes want to reset all QF-Test variables, as well as stopping all connected clients. By this step it is guaranteed, that a test execution does not fail due to side impacts coming from a previous one. This could happen if global variables get an incorrect value assigned or because the connected client may be in a wrong state.

For resetting all QF-Test variables and clients, you could utilize the menu action Run→Reset everything. This action could be utilized for further use-cases. It could happen, that too many variables were deleted or another typical situation could to reset global Jython or Groovy variables. Those variables cannot be reset by this menu action per default. In addition you might want to de-register a TestRunListener. In order to meet those requirements and to offer the capability to implement custom post conditions, the ResetListener interface has been introduced.

Each class object that implements the ResetListener Interface `de.qfs.apps.qftest.extensions.qftest.ResetListener` and was registered in QF-Test represents a ResetListener. To manage those ResetListener, the QF-Test runcontext provides the methods⁽⁶²²⁾ `addResetListener`, `isResetListenerRegistered` and `removeResetListener`.

The ResetListener Interface is composed of two methods:

beforeReset ()

This method will be called, before all QF-Test variables and clients will get resetted. Implementing this method may also prevent certain clients from closing, when the `Run→Reset everything` function gets invoked.

Returns A `java.util.Set<String>` Object, that contains the name of all clients, that should not get closed/resetted.

afterReset ()

This method will be called, once all QF-Test variables and clients got resetted.

The following example describes the implementation and registration of a `ResetListener`. The `ResetListener` prevent, that the QF-Test variable "client" gets deleted by the `Reset` function. It prevents the closing of the "client" SUT too.

```
from java.util import HashSet
from de.qfs.apps.qftest.extensions.qftest import ResetListener
from de.qfs.apps.qftest.shared.exceptions import UnboundVariableException
class L(ResetListener):
    def beforeReset(self):
        try:
            self.client = rc.lookup("client")
            h = HashSet()
            h.add(self.client)
            return h
        except UnboundVariableException:
            self.client = None
    def afterReset(self):
        if self.client != None:
            rc.setGlobal("client", self.client)
global resetListener
try:
    resetListener
except:
    resetListener = L()
    rc.addResetListener(resetListener)
```

Example 39.16: An example of the implementation of a `ResetListener`

39.7 DOM processors

When creating reports from a run-log, or package documentation from a test-suite, QF-Test operates in a two-step process. The first step creates an XML document which is transformed to an HTML document in the second step. Both transformations are done using XSLT stylesheets.

Note

The term DOM (for *Document Object Model*) also applies to XML documents, not only to HTML web pages. This section is all about XML and XSLT and not about the DOM of a web SUT.

However, XSLT stylesheets are not very useful when it comes to parsing plain text. The Comment fields of Test, Sequence, Package or Procedure nodes often contain some internal structure that XSLT cannot make use of. Additionally, the internal structures employed by users may vary, depending on the conventions used. A typical example is the use of JavaDoc tags to describe parameters of Procedure nodes. Here's an example Comment for the Procedure `qfs.swing.menu.select` from our standard library after the first step of the transformation:

```
<comment>Select an item from a menu.  
For example: for the File -> Open action, the QF-Test component ID  
"File" is the menu, and the QF-Test component ID "Open" is the item.  
@param client The name of the SUT client.  
@param menu The QF-Test ID of the menu.  
@param item The QF-Test ID of the menu item.</comment>
```

Example 39.17: Example Comment after first step transformation

It is very difficult to make use of the `@param` tags with XSLT alone. This is where DOM processors enter the scene. Between the first and second transformation, QF-Test can optionally run an additional transformation directly on the DOM tree of the XML document generated by the first step. During that extra transformation, QF-Test traverses the DOM tree, calling the registered DOM processors for each node to give them a chance to manipulate the DOM.

Note

For JDK 1.4 the XML Document Object Model (DOM) is part of the standard API. For earlier JDK versions it is provided by XML parser xerces (from the Apache project) which QF-Test includes. The API documentation for the DOM is available at <http://download.oracle.com/javase/1.5.0/docs/api/org/w3c/dom/package-summary.html>.

39.7.1 The DOMProcessor interface

The interface that must be implemented is `de.qfs.apps.qftest.extensions.DOMProcessor`. It is quite trivial:

Element process (Element node)

Process one element node.

Returns An element node or null. If null is returned, the child nodes of the node are processed normally. Otherwise, the child nodes are not processed. If a node other than the original node is returned, the original node is replaced with the return value.

In the `process` method, the processor is free to do whatever it likes, as long as it constrains itself to the node passed in and its sub-nodes. The node can be replaced simply by returning some different element node.

Note

To remove an element node from the DOM, the `DOMProcessor` must be registered on an ancestor of the node, its parent node, for example. The current node may not be removed from the DOM in the `process` method.

QF-Test provides two example implementations of DOM processors. The `ParagraphProcessor` is available in the `misc` directory for illustration. It is used internally to break comments which contain empty lines into paragraphs.

Also to be found in the `misc` directory is the `DocTagProcessor` which is used to transform JavaDoc tags like `@param` or `@author` to an XML DOM sub-tree. After processing, the above example would look as follows:

```
<comment>Select an item from a menu.
For example: for the File -> Open action, the QF-Test component ID
  "File" is the menu, and the QF-Test component ID "Open" is the item.</comment>
<param name="client">The name of the SUT client.</param>
<param name="menu">The QF-Test component ID of the menu.</param>
<param name="item">The QF-Test component ID of the menu item.</param>
```

Example 39.18: Example comment after DOM processing

Transforming the above into useful HTML during the second stage transformation is now straightforward.

39.7.2 The `DOMProcessorRegistry`

Before a DOM processor can be used, it must be registered for the kind of node(s) it applies to. This is done through the `DOMProcessorRegistry`.

There is one `DOMProcessorRegistry` instance object per kind of transformation, each identified by a string. Currently these identifiers are `"report"` for report generation and `"testdoc"` and `"pkgdoc"` for test-set and package documentation. To get hold of a registry instance, use the static `instance` method:

```
DOMProcessorRegistry instance(String identifier)
```

Get hold of a registry instance.

Parameters

identifier	The identifier for the kind of transformation.

The rest of the methods consist of the typical set of register/unregister variants:

```
void registerDOMProcessor(DOMProcessor processor)
```

Register a generic DOM processor that will be called for all kinds of nodes.

Parameters

processor	The processor to register.

```
void registerDOMProcessor(String node, DOMProcessor processor)
```

Register a DOM processor for a specific kind of node.

Parameters

name	The type of the node.
processor	The processor to register.

```
void unregisterDOMProcessor(DOMProcessor processor)
```

Unregister a generic DOM processor.

Parameters

processor	The processor to unregister.

```
void unregisterDOMProcessor(String node, DOMProcessor processor)
```

Unregister a DOM processor for a specific kind of node.

Parameters

name	The type of the node.
processor	The processor to unregister.

```
void unregisterDOMProcessors()
```

Unregister all DOM processors.

39.7.3 Error handling

Exceptions raised during DOM processing by the `process` method of a `DOMProcessor` are caught and duly reported, the transformation is stopped in that case.

39.8 Image API extensions

3.0+

The Image API of QF-Test makes use of a technology independent class for image representation `ImageRep` and it offers an interface to implement an own image compare algorithm.

39.8.1 The `ImageRep` class

The class `de.qfs.apps.qftest.shared.extensions.image.ImageRep` is the wrapper class for technology independent images.

The class holds the image representation either as ARGB data, which is an int array or as RGB data, which is a byte array. Besides the image data, you can also specify a name, the width and the height for the `ImageRep` object.

The `ImageRep` class also contains an `equals` method for comparisons. If you want to implement your own image comparison algorithm, you have to implement your own `ImageComparator`. This implementation has to be registered at the `ImageRep` object. See [section 39.8.2^{\(740\)}](#) for further information.

`ImageRep ImageRep()`

Constructor for the `ImageRep` class.

`ImageRep ImageRep(String name, byte[] rgb, boolean png, int width, int height)`

Constructor method for the `ImageRep` class.

Parameters

name	The name of the <code>ImageRep</code> object.
rgb	A byte array containing RGB data of the image.
png	Whether the image is already png encoded.
width	The width of the image.
height	The height of the image.

```
ImageRep ImageRep(String name, int[] argb, boolean png, int width, int height)
```

Constructor method for the `ImageRep` class.

Parameters

name	The name of the <code>ImageRep</code> object.
argb	An int array containing the ARGB data of the image.
png	Whether the image is already png encoded.
width	The width of the image.
height	The height of the image.

```
void setName(String name)
```

Set the name.

Parameters

name	The new name.

```
String getName()
```

Get the name.

Returns	The current name.

```
void setWidth(int width)
```

Set the width.

Parameters

width	The new width.

```
int getWidth()
```

Get the width.

Returns	The current width.

```
void setHeight(int height)
```

Set the height.

Parameters

height	The new height.

```
int getHeight()
```

Get the height.

Returns	The current height.

```
void setPng(byte[] png)
```

Set the RGB data.

Parameters

png	The new RGB data.

byte[] getPng()

Get the RGB data. If no RGB data is set, the ARGB data will be translated into the RGB data.

Returns The current RGB data.

void setARGB(int[] argb)

Set the ARGB data.

Parameters

argb The new ARGB data.

int[] getARGB()

Get the ARGB data. If no ARGB data is set, the RGB data will be translated into the ARGB data.

Returns The current ARGB data.

void setComparator(ImageComparator comparator)

Set `ImageComparator` implementation, which will be used for comparing images.

Parameters

comparator The new `ImageComparator` implementation.

ImageComparator getComparator()

Get the current `ImageComparator` implementation.

Returns The current `ImageComparator` implementation.

boolean equals(ImageRep compare)

Return, whether the current object is equal to a given object. It uses the `equals` method of the current `ImageComparator` implementation.

Parameters

compare The `ImageRep` object to compare with.

Returns True, if images are equal, otherwise false.

int getPixel(int x, int y)

Get an ARGB pixel value.

Parameters

x The X coordinate of the pixel.

y The Y coordinate of the pixel.

Returns The pixel value.

```
void crop(int x, int y, int width, int height)
```

Crop the image to a specified sub-region.

Parameters

x	The X coordinate of the left upper corner of the sub-region.
y	The Y coordinate of the left upper corner of the sub-region.
width	The width of the sub-region.
height	The height of the sub-region.

39.8.2 The ImageComparator interface

The `de.qfs.apps.qftest.shared.extensions.image.ImageComparator` interface has to be implemented, if you want to implement your own image comparison algorithm.

The implementation has then to be registered at the used `ImageRep` objects using the `setComparator` method.

```
boolean equals(ImageRep actual, ImageRep expected)
```

Return, whether the current object is equal to a given object. It uses the `equals` method of the current `ImageComparator` implementation.

Parameters

actual	The actual <code>ImageRep</code> object.
expected	The expected <code>ImageRep</code> object.
Returns	True, if images are equal, otherwise false.

Chapter 40

Daemon mode

!!! Warning !!!

Anybody with access to the QF-Test daemon can start any program on the machine running the daemon with the rights of the user account that the daemon is running under, so access should be granted only to trusted users.

If you are not running the daemon in a secure environment where every user is trusted or if you are creating your own library to connect to the QF-Test daemon, you definitely should **read section 40.3**⁽⁷⁵⁵⁾ below.

40.1 Daemon concepts

In daemon mode (comparable, but not equivalent to a "service" on Windows) QF-Test listens to RMI connections and providing an interface for remote test execution. This is useful for simplifying test execution in a distributed load-testing scenario, but also for integration with existing test-management or test-execution tools.

There are three command line arguments, all of which are available in batch and interactive mode:

- -daemon⁽⁵⁷⁴⁾ - Run QF-Test in daemon mode.
- -daemonport <port>⁽⁵⁷⁵⁾ - Specify the port the daemon should listen at, the default is 3543.
- -daemonrmiport <port>⁽⁵⁷⁵⁾ - Specify the RMI port the daemon should use. Useful only when running the daemon behind a firewall. Default is a random free port.

When run in batch and daemon mode, QF-Test will not use a license. Licenses will be acquired during use as described below. In interactive daemon mode, QF-Test will

be fully functional and thus use a license. In addition it will accept connections from the outside and use additional licenses during use, similar to batch mode. The latter scenario is useful during development of distributed tests.

Tests can be executed on a running daemon in one of two ways:

- Use of QF-Test's batch commandline option `-calldaemon`⁽⁵⁷⁴⁾.
- Direct implementation against the daemon API, documented in [section 40.2](#)⁽⁷⁴²⁾.

Both options are explained in more detail and accompanied by examples in [section 16.2](#)⁽¹⁴⁴⁾.

40.2 Daemon API

The daemon API provides all methods necessary to directly control test execution via a QF-Test daemon. Typically the following relatively simple steps need to be implemented:

Note

When writing an application based on the daemon API you need to take security considerations into account and either disable security or set some RMI-specific properties as described in [section 40.3](#)⁽⁷⁵⁵⁾.

- Get hold of a `Daemon` using the `DaemonLocator` service.
- Either get hold of the shared `TestRunDaemon` or have the `Daemon` create a `TestRunDaemon`. You can use the `TestRunDaemon` to define global variables and the root directory of your test-suites for the upcoming test-runs.
- Either get hold of the `TestRunDaemon`'s shared `DaemonRunContext` or have it create one or more `DaemonRunContext` instances. A `DaemonRunContext` represents one test execution context for functional or load testing. Each `DaemonRunContext` is independent of the others, except for a group of contexts created via `TestRunDaemon.createContexts(int threads)` which creates a group of related contexts for load-testing similar to running QF-Test with the `-threads <number>`⁽⁵⁸⁶⁾ command line argument. Each `DaemonRunContext` requires one QF-Test development or runtime license during its lifetime.
- Now you can tell the `DaemonRunContext` to execute tests on your behalf, either whole test-suites or `Test-set`⁽³²¹⁾ or `Test-case`⁽³¹⁴⁾ nodes. With properly implemented `Dependencies`⁽³⁴⁴⁾ you can have a `Daemon` execute Test-cases on your behalf in any arbitrary order without explicitly having to take care of any setup or cleanup tasks.
- The `DaemonRunContext` also has methods to check the current state of a test-run or wait for it to finish.

- Finally you can get the run-log of the test-run from the Daemon. Right now, all you can do is save it, but we will open the run-log API so that you will be able to integrate run-logs from various daemons and test-runs into a single run-log.

For proper dependency management, including rollback of no longer required dependencies between Test-case invocations, it is important to use the same `DaemonRunContext` for execution of each related Test-case. The easiest way to achieve this is to use the shared `TestRunDaemon` and its shared `DaemonRunContext` each time you talk to a given Daemon.

The following sections provide a complete reference for the daemon API. Further explanations and examples are provided in [section 16.2^{\(144\)}](#).

40.2.1 The DaemonLocator

The singleton class `de.qfs.apps.qftest.daemon.DaemonLocator` can be used to get hold of `Daemon` instances.

public static DaemonLocator instance()

Get the singleton instance.

Returns The singleton instance.

public Daemon locateDaemon(String host, int port)

Get a Daemon from a specific host and port.

Parameters

host The target host, name or IP string.

port The target port.

Returns The daemon or null if none can be found.

public Daemon[] locateDaemons(long timeout)

Get all known daemons.

Parameters

timeout The time in milliseconds to wait for daemons to react.

Returns The known daemons.

40.2.2 The Daemon

The `de.qfs.apps.qftest.daemon.Daemon` interface is an envelope for various kinds of QF-Test daemons. Currently only the `TestRunDaemon` is available, but something like an `SUTClientStarterDaemon` is already planned and others may follow.

public void ping()

Test whether the Daemon is still alive.

Throws**RemoteException** If something RMI specific goes wrong.

public String getHost()

Get the host of the Daemon.

Returns The host of the Daemon.**Throws****RemoteException** If something RMI specific goes wrong.

public String getIp()

Get the ip of the Daemon.

Returns The ip of the Daemon.**Throws****RemoteException** If something RMI specific goes wrong.

public int getPort()

Get the port of the Daemon.

Returns The port of the Daemon.**Throws****RemoteException** If something RMI specific goes wrong.

public TestRunDaemon getSharedTestRunDaemon()

Get the shared TestRunDaemon.

Returns The shared TestRunDaemon.**Throws****RemoteException** If something RMI specific goes wrong.

public TestRunDaemon createTestRunDaemon()

Get access to a TestRunDaemon.

Returns A TestRunDaemon.**Throws****RemoteException** If something RMI specific goes wrong.

public TestRunDaemon[] getTestRunDaemons()

Get all TestRunDaemons created by the Daemon that are still live.

Returns The set of live TestRunDaemons created by the Daemon.
Does not include the shared TestRunDaemon.**Throws****RemoteException** If something RMI specific goes wrong.

```
public void terminate(int exitCode)
```

Terminate the daemon process by calling `System.exit`.

Parameters

exitCode The exit code for the daemon.

Throws

RemoteException If something RMI specific goes wrong.

```
public void killClients()
```

Kill all clients that belong to the VM of the daemon.

Throws

RemoteException If something RMI specific goes wrong.

```
void cleanup()
```

Clean up all `TestRunDaemons` belonging to this `Daemon` and then kill all clients. The default timeout of 30 seconds is used to wait for possible dependency rollback.

Throws

RemoteException If something RMI specific goes wrong.

```
void cleanup(long timeout)
```

Clean up all `TestRunDaemons` belonging to this `Daemon` and then kill all clients.

Parameters

timeout The maximum time in milliseconds to wait for possible dependency rollback.

Throws

RemoteException If something RMI specific goes wrong.

40.2.3 The TestRunDaemon

The `de.qfs.apps.qftest.daemon.TestRunDaemon` is the outer interface for test execution. It is used to define the environment for associated test-runs and create the `DaemonRunContext` instances that handle the actual test execution.

Miscellaneous

```
public Daemon getDaemon()
```

Get the `Daemon` to which the `TestRunDaemon` belongs.

Returns The `Daemon` of the `TestRunDaemon`.

Throws

RemoteException If something RMI specific goes wrong.

Global variable handling

```
public void setGlobal(String name, String value)
```

Set a global variable value.

Parameters

name The name of the global variable.

value The value of the global variable.

Throws

RemoteException If something RMI specific goes wrong.

```
public String getGlobal(String name)
```

Get a global variable value.

Parameters

name The name of the global variable.

Returns The value of the global variable or null if undefined.

Throws

RemoteException If something RMI specific goes wrong.

```
public Properties getGlobals()
```

Get all global variable values.

Returns All global variable values.

Throws

RemoteException If something RMI specific goes wrong.

```
public void clearGlobals()
```

Clear all global variables.

Throws

RemoteException If something RMI specific goes wrong.

Test execution

```
public void setRootDirectory(String directory)
```

Set the test-suite root directory for new created daemon run-contexts.

Parameters

directory The new root directory.

Throws

RemoteException If something RMI specific goes wrong.

public DaemonRunContext getSharedContext ()

Get the shared daemon run-context. Needs to acquire a license if the shared context must be created first.

Returns The shared context or null if no license can be acquired.

Throws

RemoteException If something RMI specific goes wrong.

public DaemonRunContext createContext ()

Create a single daemon run-context. Needs to acquire a license.

Returns The context or null if no license can be acquired.

Throws

RemoteException If something RMI specific goes wrong.

public DaemonRunContext[] createContexts(int threads)

Create daemon run-contexts for multiple threads. Needs to acquire one license per thread.

Parameters

threads The number of threads for the contexts.

Returns The contexts or null if not enough licenses can be acquired.

Throws

RemoteException If something RMI specific goes wrong.

public DaemonRunContext[] getContexts ()

Get all DaemonRunContexts created by the TestRunDaemon that are still live and have not been released.

Returns The set of live DaemonRunContexts created by the TestRunDaemon. Does not include the shared DaemonRunContext.

Throws

RemoteException If something RMI specific goes wrong.

void cleanup ()

Clean up and release all contexts belonging to this TestRunDaemon. The default timeout of 30 seconds is used to wait for possible dependency rollback.

Throws

RemoteException If something RMI specific goes wrong.

void cleanup(long timeout)

Clean up and release all contexts belonging to this TestRunDaemon.

Parameters

timeout	The maximum time in milliseconds to wait for possible dependency rollback.

Throws

RemoteException	If something RMI specific goes wrong.

Identification

public void setIdentifier(String identifier)

Set an identifier for the TestRunDaemon. This can be useful in identifying a TestRunDaemon retrieved via `Daemon.getTestRunDaemons()`.

Parameters

identifier	The identifier to set.

Throws

RemoteException	If something RMI specific goes wrong.

public String getIdentifier()

Get the identifier for the TestRunDaemon. If no identifier was previously set with `setIdentifier()`, a default identifier is created from the name of the Daemon, to which the TestRunDaemon belongs, and a counter.

Returns	The identifier for the TestRunDaemon.

Throws

RemoteException	If something RMI specific goes wrong.

40.2.4 The DaemonRunContext

The `de.qfs.apps.qftest.daemon.DaemonRunContext` interface is in charge of the actual test execution.

Following run states are defined:

State	Value	Description
STATE_INVALID	-1	Invalid after release - cannot be reactivated.
STATE_IDLE	0	No run scheduled.
STATE_SCHEDULED	1	Run scheduled but not started.
STATE_RUNNING	2	Running.
STATE_PAUSED	3	Running but paused.
STATE_FINISHED	4	Run finished, result and run-log available.

Table 40.1: The run state

```
public TestRunDaemon getTestRunDaemon()
```

Get the TestRunDaemon to which the DaemonRunContext belongs.

Returns The TestRunDaemon of the DaemonRunContext.

Throws

RemoteException If something RMI specific goes wrong.

```
public int getNumThreads()
```

Get the number of threads in the group to which the DaemonRunContext belongs.

Returns The number of threads of the DaemonRunContext's group.

Throws

RemoteException If something RMI specific goes wrong.

```
public int getThreadNum()
```

Get the thread index of the DaemonRunContext.

Parameters

Returns The thread index of the DaemonRunContext.

Throws

RemoteException If something RMI specific goes wrong.

```
public int getRunState()
```

Get the current run state of the context. One of STATE_IDLE, STATE_SCHEDULED, STATE_RUNNING, STATE_PAUSED, STATE_FINISHED.

Returns The current run state.

Throws

RemoteException If something RMI specific goes wrong.

```
public boolean waitForRunState(int state, long timeout)
```

Wait for the context to reach a given state.

Parameters

state The state to wait for.

timeout Maximum time in milliseconds to wait.

Returns True if the state was reached, false if timed out.

Throws

RemoteException If something RMI specific goes wrong.

```
public void setRootDirectory(String directory)
```

Set the test-suite root directory for the next test-run.

Parameters

directory The new root directory.

Throws

RemoteException If something RMI specific goes wrong.

```
public boolean runTest(String test, Properties bindings=None)
```

Run a test in the run-context.

Parameters

test

The test to run, of the form `Suite#Test` where `#Test` is optional and `Test` is the fully qualified name of a `Test`-set or `Test`-case or just `"`. The latter is equivalent to specifying just `Suite` and causes the whole test-suite to be executed. Examples:

<code>MySuite</code>	Runs whole test-suite <code>MySuite</code> .
<code>MySuite#.</code>	Runs whole test-suite <code>MySuite</code> .
<code>MySuite#MyTestSet</code>	Runs test-set <code>MyTestSet</code> in test-suite <code>MySuite</code> .
<code>MySuite#MyTestSet.MyTestCase</code>	Runs test-case <code>MyTestCase</code> located in test-set <code>MyTestSet</code> in test-suite <code>MySuite</code> .

bindings

An optional set of variable bindings. These variables have higher precedence than the globals or any bindings on the fallback stack.

Returns

True if the test was started, false if suite or test could not be found.

Throws

RemoteException

If something RMI specific goes wrong.

IllegalStateException

If no run can be started in the current state, i.e. if the state is neither `STATE_IDLE` nor `STATE_FINISHED`.

```
public boolean callProcedure(String procedure, Properties
bindings=None)
```

Call a procedure in the run-context.

Parameters

procedure The procedure to run, of the form `Suite#Procedure` where `Procedure` is the fully qualified name of the Procedure.

bindings An optional set of variable bindings. These variables have higher precedence than the globals or any bindings on the fallback stack.

Returns True if the procedure call was started, false if suite or procedure could not be found.

Throws

RemoteException If something RMI specific goes wrong.

IllegalStateException If no run can be started in the current state, i.e. if the state is neither `STATE_IDLE` nor `STATE_FINISHED`.

```
public String getLastTest()
```

Get the name of the test that is currently running or was last run on this `DaemonRunContext`.

Returns The name of the currently running or most recently executed test.

Throws

RemoteException If something RMI specific goes wrong.

```
public void stopRun()
```

Stop the test-run.

Throws

RemoteException If something RMI specific goes wrong.

IllegalStateException If no run was scheduled.

```
public int getResult()
```

Get the result of the test run.

Returns 0, 1, 2, 3 for OK, warning, error, exception.

Throws

RemoteException If something RMI specific goes wrong.

IllegalStateException If the state isn't `STATE_FINISHED`.

public byte[] getRunLog()

Get the run-log of the test run.

Returns The run-log dumped into a byte array.

Throws

RemoteException If something RMI specific goes wrong.

IllegalStateException If the state isn't STATE_FINISHED.

void rollbackDependencies()

Roll back the dependencies for this DaemonRunContext.

Throws

RemoteException If something RMI specific goes wrong.

IllegalStateException If no run can be started in the current state, i.e. if the state is neither STATE_IDLE nor STATE_FINISHED.

public void release()

Release the DaemonRunContext and return its license. If a test is running, stop it.

Throws

RemoteException If something RMI specific goes wrong.

IllegalStateException If no DaemonRunContext was allocated.

public void setIdentifier(String identifier)

Set an identifier for the DaemonRunContext. This can be useful in identifying a DaemonRunContext retrieved via TestRunDaemon.getContexts().

Parameters

identifier The identifier to set.

Throws

RemoteException If something RMI specific goes wrong.

public String getIdentifier()

Get the identifier for the DaemonRunContext. If no identifier was previously set with setIdentifier(), a default identifier is created from the name of the TestRunDaemon, to which the DaemonRunContext belongs, and a counter.

Returns The identifier for the DaemonRunContext.

Throws

RemoteException If something RMI specific goes wrong.

public String getGlobal(String name)

Retrieve the value of a global variable from the DaemonRunContext.

Parameters

name The name of the variable.

Returns The value of the variable or null if undefined.

Throws

RemoteException If something RMI specific goes wrong.

public Properties getGlobals()

Retrieve all global variables from the DaemonRunContext.

Returns The global variables.

Throws

RemoteException If something RMI specific goes wrong.

public String getProperty(String group, String name)

Retrieve the value of a property or resource from the DaemonRunContext.

Parameters

name The name of the property or resource group.

name The name of the property.

Returns The value of the property or null if undefined.

Throws

RemoteException If something RMI specific goes wrong.

public Properties getProperties(String group)

Retrieve all properties from a property or resource group from the DaemonRunContext.

Parameters

name The name of the property or resource group.

Returns The properties or null if no such group exists.

Throws

RemoteException If something RMI specific goes wrong.

```
void addTestRunListener(DaemonTestRunListener listener, boolean  
synchronous, long timeout)
```

Add a `DaemonTestRunListener` to the `DaemonRunContext`.

Parameters

listener	The listener to add.
synchronous	Whether the listener should get notified synchronously, in which case the test-run will be blocked while the listener is processing the event.
timeout	Timeout in milliseconds for callbacks to the listener. If the listener does not reply within that time the listener is automatically unregistered to prevent further problems. In case of a synchronous listener the test-run will then continue. A value of 0 means no timeout which is dangerous but may be useful.

```
void removeTestRunListener(DaemonTestRunListener listener)
```

Remove a `DaemonTestRunListener` from the `DaemonRunContext`.

Parameters

listener	The listener to remove.

```
void clearTestRunListeners()
```

Remove all `DaemonTestRunListeners` from the `DaemonRunContext`.

40.2.5 The `DaemonTestRunListener`

The `de.qfs.apps.qftest.daemon.DaemonTestRunListener` interface is identical to the `de.qfs.apps.qftest.extensions.qftest.TestRunListener` interface described in [section 39.5^{\(728\)}](#), except that its methods can throw a `RemoteException` on RMI failure. When implementing this interface you must derive your class from `java.rmi.server.UnicastRemoteObject`.

You can register the listener with a `DaemonRunContext` via its `addTestRunListener` method described in the previous section.

40.3 Daemon security considerations

Anybody with access to the QF-Test daemon can start any program on the daemon machine with the rights of the user account that the daemon is running under, so care should be taken to only allow trusted users to connect to the daemon.

Of course the QF-Test daemon should always be run on a machine that is protected from outside access by a firewall. If all users that can access the machine behind the firewall are trusted, that is sufficient. If the set of trusted users needs to be limited further, please read on.

By default the QF-Test daemon uses SSL to secure the RMI connection. However, unless you take additional precautions, this only means that the network traffic between the daemon and its client is encrypted. To restrict access to certain users, one further step is required.

Setting up SSL communication can be very complex. One usually has to learn about keys, certificates, certificate authorities, chains of trust etc. Fortunately, this is a very special case, and the fact that once a user has access to the QF-Test daemon he also has control over the daemon's machine means that there is no distinction between the daemon administrator and a daemon user. Without going into details, QF-Test normally uses a single keystore file with a single self-signed certificate on both daemon and client side. More complex scenarios are possible but beyond the scope of this manual. The default keystore file is named `daemon.keystore` and provided in QF-Test's version-specific directory. By creating your own keystore as described below you can ensure that only users that have read access to that keystore file can interact with the daemon.

40.3.1 Creating your own keystore

To create the keystore file you need a current JDK version 1.5 or higher, a JRE is not sufficient. In a shell or console window, execute the following command (you may need to prepend the path to the `keytool` program which resides in the JDK's `bin` directory):

```
keytool -keystore daemon.keystore -genkey -alias "qftest daemon"  
-keyalg DSA -validity 999999
```

Example 40.1: Creating a keystore for secure daemon communication

For further information about the `keytool` command please see <http://download.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.htm>.

When asked for the password for the keystore enter `123456`. When asked for your name or organization, feel free to provide answers or not. For QF-Test these don't make any difference. Of course you can provide a secure password instead of `123456`, but that will only complicate starting the daemon and its client and not contribute much to security. You could also use a shorter validity, but in case the keystore file gets into the wrong hands, all you need to do is set up your daemon and users with a new one, so the old one becomes useless.

40.3.2 Specifying the keystore

You have several options to tell QF-Test to use your keystore instead of the default one:

- Save the file as `daemon.keystore` in QF-Test's root directory
- Save the file as `daemon.keystore` in your QF-Test user directory, i.e. `~/.qftest` on Unix, `C:\Users\<username>\.qftest` on Windows 7 and Vista or `C:\Documents and settings\<username>\.qftest` on Windows XP.
- Save the file wherever you like and provide its location to QF-Test via the command line argument `-keystore <keystore file>`⁽⁵⁷⁷⁾.

In case you specified your own password for the keystore you also need to specify that via the command line argument `-keypass <keystore password>`⁽⁵⁷⁷⁾.

In case you would like to start the daemon without any SSL support, for example to interact with a QF-Test version older than 3.5, either remove the file called `daemon.keystore` from QF-Test's version specific directory or use the command line argument in the form `-keystore=` to specify no keystore.

40.3.3 Specifying the keystore on the client side

If you use `qftest -batch -calldaemon` to access the daemon or script nodes from within QF-Test, the options for the client are the same as for the daemon.

To access the daemon over SSL from your own code via the daemon API you must set the System properties `javax.net.ssl.keyStore` and `javax.net.ssl.trustStore` to the keystore file and `javax.net.ssl.keyStorePassword` to the password for the keystore file. See [section 40.2](#)⁽⁷⁴²⁾ for details about the daemon API and [section 16.2](#)⁽¹⁴⁴⁾ for examples.

Chapter 41

The Procedure Builder definition file

For general information about the Procedure Builder, please see chapter [chapter 18^{\(159\)}](#).

41.1 Placeholders

In the definition of procedures, packages and even in the nodes of the procedures, you can use placeholders. The following table shows all placeholders and their meaning:

Placeholder	Description
<COMPID>	The QF-Test ID of the component
<COMPNAME>	The name of the component
<COMPFEATURE>	The feature of the component
<COMPCLASS>	The recorded class of the component
<COMPTKCLASS>	The toolkit specific class of the component
<COMPSYSCCLASS>	The system class of the component
<COMPGENCLASS>	The generic class of the component
<COMPEF-name-of-extra-feature>	The value of the given extra feature
<CURRENTVALUE>	The current value of the component, e.g. the text of a text-field or the current selection of a combo-box etc.
<CURRENTENABLEDSTATE>	The current enabled state of the component
<CURRENTSELECTEDSTATE>	The current selectable state of the component
<CURRENTEDITABLESTATE>	The current editable state of the component
<PCOMPID>	The QF-Test ID of the parent component
<PCOMPNAME>	The name of the parent component
<PCOMPFEATURE>	The feature of the parent component
<PCOMPCLASS>	The class of the parent component
<PCOMPEF-name-of-extrafeature>	The value of the given extra feature of the parent component
<GPCOMPID>	The QF-Test ID of the grandparent component
<GPCOMPNAME>	The name of the grandparent component
<GPCOMPFEATURE>	The feature of the grandparent component
<GPCOMPCLASS>	The class of the grandparent component
<GPCOMPEF-name-of-extrafeature>	The value of the given extra feature of the grandparent component
<ENGINE>	The engine's name, either 'awt', 'swt', 'web' or 'fx'.
<ENGINE2>	The alternative engine's name, either 'swing', 'swt', 'web' or 'fx'.

Table 41.1: Placeholders for component procedures

For procedures of container or composite components you can also use placeholders for the child components. Please see following table for those additional variables:

Variable	Description
<CCOMPID>	The QF-Test ID of the child component
<CCOMPNAME>	The name of the child component
<CCOMPFEATURE>	The feature of the child component
<CCOMPCLASS>	The recorded class of the child component
<CCOMPTKCLASS>	The toolkit class of the child component
<CCOMPSYSCCLASS>	The system class of the child component
<CCOMPGENCLASS>	The generic class of the child component
<CCOMPEF-name-of-extrafeature>	The value of the given extra feature
<CCURRENTVALUE>	The current value of the child component, e.g. the text of a text-field or the current selection of a combo-box etc.
<CCURRENTENABLEDSTATE>	The current enabled state of the child component
<CCURRENTSELECTEDSTATE>	The current selectable state of the child component
<CCURRENTEDITABLESTATE>	The current editable state of the child component

Table 41.2: Additional placeholders for container procedures

41.2 Conditions for Package and Procedure Definition

You can influence the creation of packages and procedures via using the 'comment' attribute of the 'Package' or 'Procedure' node.

Comment-Attribute	Description
@NOTINHERIT	If this value is set, then this package or procedure is only defined for the exact class and not for its descendants.
@FORCHILDREN	If you define procedures for container procedures, you can set this value to tell QF-Test to create those procedures for calling the child components' procedures.
@EXCEPT	Specifies, whether the package or the procedure should be defined for a certain class or not. This construct might be useful, if you define a package or procedure for an abstract class and the procedures should not be created for all of its descendants.
@CONDITION	Specifies a dedicated condition for creating packages or procedures. You can define conditions as Jython, Groovy or regular expression. Please see below for further details.
@SWTSTYLE	Evaluate the style attribute of a given SWT-button. This SWT specific attribute got required, because SWT distinguishes between check-boxes, buttons or radio-buttons only through the style attribute. For SWT-buttons you can define something like @SWTSTYLE=PUSH or @SWTSTYLE=RADIO for radio buttons.
@SUBITEM	This value is only valid for menu actions up-to-now. If this flag is set, then the according package or procedure will be created for sub-items on the second level of a menu only.

Table 41.3: Comment attributes for procedure creation

41.3 Interpretation of the Component Hierarchy

It might be interesting to make use of the component-hierarchy in the package structure. This approach allows the tester to locate the component-specific procedures quite easy. If you want to create component-hierarchy packages, you can use two placeholders in the package-names:

Hierarchy-Placeholder	Description
<HIERARCHY>	Create packages for the full component-hierarchy by using the QF-Test component ID.
<HIERARCHY_NAME>	Create packages for the full component-hierarchy by using the name of the component. If a component in the hierarchy has no name, the component won't get taken into account.
<HIERARCHY_FEATURE>	Create packages for the full component-hierarchy by using the recorded feature of the component. If a component in the hierarchy has no feature, the component won't get taken into account.
<IHIERARCHY>	Create packages only for interesting components in the component-hierarchy by using the QF-Test component ID. An interesting component is a component with a feature.
<IHIERARCHY_NAME>	Create packages only for interesting components in the component-hierarchy by using the name of the component. An interesting component is a component with a feature.
<IHIERARCHY_FEATURE>	Create packages only for interesting components in the component-hierarchy by using the feature of the component. An interesting component is a component with a feature.
<MHIERARCHY>	Create packages only for menu components in the component-hierarchy by using the QF-Test component ID. It only takes components into account, which act as menu or menu-item.
<MHIERARCHY_NAME>	Create packages only for menu components in the component-hierarchy by using the component-name. It only takes components into account, which act as menu or menu-item. If no name is set, the component will be ignored.
<MHIERARCHY_FEATURE>	Create packages only for menu components in the component-hierarchy by using the component-feature. It only takes components into account, which act as menu or menu-item. If no feature is set, the component will be ignored.

Table 41.4: Hierarchy placeholders

41.4 Details about the @CONDITION tag

Using the @CONDITION tag allows you to configure, whether a dedicated node should be created or not during creation time.

Such conditions might be used to check a certain name or for appearance of a dedicated letter in the feature. If this condition is not fulfilled, the node won't be created. You can use all known placeholders, e.g. <COMPID> or <CCOMPNAME>.

Value of Condition	Meaning
@CONDITION jython "<COMPFEATURE>".startswith("abc")	Here we define a Jython condition, which will create the according node, if the feature of the current component starts with 'abc'. It is possible to use any string or comparing method of Jython.
@CONDITION groovy "<COMPFEATURE>".startsWith("abc")	Here we define a Groovy condition, which will create the according node, if the feature of the current component starts with 'abc'. It is possible to use any string or comparing method of Groovy.
@CONDITION regexp "<COMPFEATURE>" =~ "abc.*"	Here we define a regular expression, which will create the according node, if the feature of the current component starts with 'abc'. It is possible to use all capabilities of Java regular expressions.
@CONDITION regexp "<COMPFEATURE>" !~ "abc.*"	Here we define a regular expression condition, which will create the according node, if the feature of the current component does not start with 'abc'. It is possible to use all capabilities of Java regular expressions.

Table 41.5: Samples for the @CONDITION tag

If you need more than one row, you have to use a '\ ' at the end of the first row.

Chapter 42

The ManualStepDialog

The `ManualStepDialog` is a Java class delivered by QF-Test. If you want to make use of this dialog for your own tests, please see following sample script and API specification.

```
from de.qfs.apps.qftest import ManualStepDialog
    #create the dialog and show it immediately
    manualDialog = ManualStepDialog(None, "New Test Case Title", \
    "Step Description", "Expected Test Result")
    #did the test fail or succeed?
    failOrSuccess = manualDialog.getResult()
    #get the content of the received result
    receivedResult = manualDialog.getReceivedResult()
    #get the execution information, whether skipped or canceled
    execInfo = manualDialog.getExecInfo()
```

Example 42.1: Example use of `ManualStepDialog`

42.1 The ManualStepDialog API

**`ManualStepDialog(Component parent, String title, String
stepText, String expResult)`**

Create a new `ManualStepDialog`.

Parameters

parent	The parent component for the dialog.
title	The title of the dialog.
stepText	The text for the step description text-field.
expResult	The text for the expected result text-field.

String getResult()

Get the result of the test-step. Please, see chapter [section 22.5^{\(205\)}](#) for all possible results.

Returns A string containing the result.

void setResult(String newResult)

Set the result of the test-step. Please, see chapter [section 22.5^{\(205\)}](#) for all possible results.

Parameters

newResult The result of the test-step.

String getExecInfo()

Get the execution information of the test-step.

Returns A string containing the execution information.

void setExecInfo(String newExecInfo)

Set the execution information of the test-step.

Parameters

newExecInfo The execution information of the test-step.

String getReceivedResult()

Get the received result of the test-step.

Returns A string containing the received result.

void setReceivedResult(String newRecResult)

Set the received result of the test-step.

Parameters

newReceived Result The received result of the test-step.

boolean isStatusFailed()

Test whether the status is FAILED.

Returns True if the status is failed, false otherwise.

boolean isStatusPassed()

Test whether the status is PASSED.

Returns True if the status is passed, false otherwise.

boolean isStatusSkipped()

Test whether the status is SKIPPED.

Returns True if the status is skipped, false otherwise.

boolean isStatusCanceled()

Test whether the status is CANCELED.

Returns True if the status is canceled, false otherwise.

Chapter 43

Details about transforming nodes

3.1+

43.1 Introduction

The transformation mechanism allows you to turn a node into another type, e.g. a Sequence into a Procedure or a Test into a Test-case. Such actions could be required for re-factoring purposes or simply to make test development for efficient.

You can transform nodes via a right mouse click and then selecting **Transform node into** and the desired type.

Note

QF-Test only shows transformations that are possible in the current context of the test-suite, so sometimes you may not see all possible transformations.

43.2 Transformation with type changes

The following transformations also change the type of child nodes of the converted nodes:

1. Test-set into Test-case
 - (a) Data driver into disabled Sequence
 - (b) Test-case into Test
2. Test-set into Test
 - (a) Dependency into disabled Sequence
 - (b) Dependency reference into disabled Procedure call
 - (c) Test-case into Test

3. Test-case into Test-set
 - (a) All children are packed into a new sub Test-case node.
4. Test-case into Test
 - (a) Dependency into disabled Sequence
 - (b) Dependency reference into disabled Procedure call
5. Test into Test-set
 - (a) If there are only Data driver and Test children, the Data driver is turned into a disabled Test.
 - (b) Otherwise all children are packed into a new sub Test node.
6. Test into Test-set recursive
 - (a) If there are only Data driver and Test children, the Data driver is turned into a disabled Test.
 - (b) Otherwise all children are packed into a new sub Test-case node.
7. Test into Test-case
 - (a) Data driver into disabled Sequence

43.3 Additional transformations below the Extras node

The following transformations are allowed below the Extras node only:

43.3.1 Transformations without side-effects

1. Sequence into Cleanup
2. Sequence into Setup
3. Sequence into Test-case
4. Cleanup into Procedure
5. Setup into Procedure
6. Procedure into Sequence
7. Procedure into Test-case
8. Procedure into Test

43.3.2 Transformations with side-effects

The following transformations also change the type of some child nodes:

1. Test-set into Package
 - (a) Cleanup into disabled Procedure
 - (b) Data driver into disabled Procedure
 - (c) Dependency reference into disabled Dependency containing the previous Dependency reference
 - (d) Setup into disabled Procedure
 - (e) Test into Procedure
 - (f) Test-case into Procedure
 - (g) Test call into disabled Procedure containing the Test call
2. Test-case into Procedure
 - (a) Cleanup into disabled Sequence
 - (b) Dependency into disabled Sequence
 - (c) Dependency reference into disabled Sequence
 - (d) Setup into disabled Sequence
3. Test into Procedure
 - (a) Data driver into disabled Sequence
4. Test into Package
 - (a) All child nodes are packed into a Procedure node.
5. Test into Sequence
 - (a) Data driver into disabled Sequence
6. Package into Test-set
 - (a) Package into Test-set
 - (b) Procedure into Test-case
7. Package into Test
 - (a) Dependency into disabled Test
 - (b) Package into Test
 - (c) Procedure into Test

Chapter 44

Details about the algorithm for image comparison

3.3+

44.1 Introduction

The classic Check image⁽⁴⁸⁸⁾ node is only minimally tolerant towards deviations. Using the default algorithm of comparing pixel by pixel it is not possible to check images that are generated in a not-quite-deterministic way or differ in size.

By using the attribute Algorithm for image comparison⁽⁴⁹¹⁾ it is possible to define a special algorithm which is tolerant towards certain image deviations. The attribute must start with the algorithm definition in the form `algorithm=<algorithm>` followed by all required parameters, separated by semicolons. Parameters may be defined in any order, and use of variables is possible, e.g.:

```
algorithm=<algo>;parameter1=value1;parameter2=value2;  
expected=$(expected)
```

3.5.1+

Since QF-Test 3.5.1 the definition does not need so start with `algorithm=<algorithm>` but can simply begin with `<algorithm>`.

It is also no longer necessary to define the parameter 'expected'. QF-Test uses a default value if it is not set. Please see below for more information.

A detailed description of the available algorithms and their parameters is provided in the following section. For illustration, explanations are based on their effects on the following image:



Figure 44.1: Original image

In the related run-log (see [section 8.1^{\(68\)}](#)) of a failed check you have the opportunity to analyze the results of the algorithm as well as the result of probability calculation. If you activate the option [Log successful advanced image checks^{\(307\)}](#) all tolerant image checks will be logged for further analysis.

44.2 Description of algorithms

44.2.1 Classic image check

Description

The classic - or default - image check compares the color value of every single expected and actual pixel. If at least one expected pixel differs from the actual pixel the check fails. The option [Tolerance for checking images^{\(282\)}](#) defines the tolerance for comparing pixel values.

Purpose

This pixel based check is suitable if you expect an exact image match with minimal tolerances or any deviations. Whenever your application renders the component not fully deterministically, this algorithm is not suitable.

Example

The classic image check doesn't transform the image, thus the result looks identical to the original image.



Figure 44.2: Classic image check

The classic image check is used when the Algorithm for image comparison⁽⁴⁹¹⁾ attribute is empty.

44.2.2 Pixel-based identity check

Description

This algorithm is similar to the classic algorithm, but accepts an amount of unexpected pixels. It splits every pixel in its three sub-pixels red, green and blue. Afterwards it checks every actual color value against the expected color value. The final result is the amount of identical pixels divided by the total amount of pixels. The calculated result is checked against an expected value.

Purpose

If your images are not rendered fully deterministic but you accept a certain percentage of unexpected pixels, this algorithm may be useful. But it is not suitable, if the actual images are used to have shifts or distortions.

Example

The result image of the exemplary algorithm

```
algorithm=identity;expected=0.95
```

looks identical to the original image because this algorithm does not manipulate the image.



Figure 44.3: Pixel-based identity check

Parameters

algorithm=identity

The 'Pixel based identity check' should be used.

expected (optional, but recommended)

Defines which probability you expect.

Valid values are between 0.0 and 1.0. If not defined, use 0.98.

resize (optional)

Defines, if the actual image should be resized before calculation to match the size of the expected image.

Valid values are "true" and "false".

find (optional)

Defines an image-in-image search.

A detailed description of this parameter can be found in [section 44.3.1^{\(783\)}](#).

44.2.3 Pixel-based similarity check

Description

This algorithm splits every pixel in its three sub-pixels red, green and blue. Afterwards it checks every actual color value against the expected color value to calculate a percental similarity. All percental deviations are added up and used for calculation. The final result is the average deviation over all color values and all pixels. The calculated result is checked against an expected value.

Purpose

If your images are not rendered fully deterministic but you accept a certain deviation, this algorithm is a possible candidate for your purpose.

If you accept deviations for some pixels, but the average deviation all over the image is small, this algorithm is also suitable.

But it is not suitable, if the actual images are used to have shifts or distortions.

Example

The result image of the exemplary algorithm

```
algorithm=similarity;expected=0.95
```

looks identical to the original image because this algorithm does not manipulate the image.



Figure 44.4: Pixel-based similarity check

Parameters

algorithm=similarity

The 'Pixel based similarity check' should be used.

expected (optional, but recommended)

Defines which probability you expect.

Valid values are between 0.0 and 1.0. If not defined, use 0.98.

resize (optional)

Defines, if the actual image should be resized before calculation to match the size of the expected image.

Valid values are "true" and "false".

find (optional)

Defines an image-in-image search.

A detailed description of this parameter can be found in [section 44.3.1^{\(783\)}](#).

44.2.4 Block-based identity check**Description**

This algorithm partitions the image into quadratic blocks with a selectable size. The color value of each of these blocks is calculated as the average of the color values of the pixels the block contains. If the width or height of the image is not a multiple of the block size, the blocks at the right and bottom edge are cropped and weighted accordingly.

The actual blocks are checked against the expected blocks. The final result is the amount of identical blocks divided by the total amount of blocks.

Purpose

This algorithm's purpose is to check an image which only differs at some parts but is identical at the remaining parts.

Example

The exemplary algorithm

```
algorithm=block;size=10;expected=0.95
```

results in the following image:



Figure 44.5: Block-based identity check

Parameters

algorithm=block

The algorithm 'Block-based identity check' should be used.

size

Defines the size of each block.

Valid values are between 1 and the image size.

expected (optional, but recommended)

Defines the minimal match probability for the check to succeed.

Valid values are between 0.0 and 1.0. If not defined, use 0.98.

resize (optional)

Defines, if the actual image should be resized before calculation to match the size of the expected image.

Valid values are "true" and "false".

find (optional)

Defines an image-in-image search.

A detailed description of this parameter can be found in [section 44.3.1^{\(783\)}](#).

44.2.5 Block-based similarity check

Description

This algorithm also partitions the image in quadratic blocks with a selectable size. The color value of each of these blocks is calculated as the average of the color values of the pixels the block contains. If the width or height of the image is not a multiple of the block size, the blocks at the right and bottom edge are cropped and weighted accordingly.

The color value of each expected block is checked against the actual block. Their color values are analyzed for percental similarity. The final result is the average similarity of all blocks with their weight taken into account.

Purpose

This algorithm is suitable for checking images with similar color variances.

Example

The exemplary algorithm

`algorithm=blocksimilarity;size=5;expected=0.95`
results in the following image:



Figure 44.6: Block-based similarity check

Parameters

algorithm=blocksimilarity

The algorithm 'Block-based similarity check' should be used.

size

Defines the size of each block.

Valid values are between 1 and the image size.

expected (optional, but recommended)

Defines the minimal match probability for the check to succeed.

Valid values are between 0.0 and 1.0. If not defined, use 0.98.

resize (optional)

Defines, if the actual image should be resized before calculation to match the size of the expected image.

Valid values are "true" and "false".

find (optional)

Defines an image-in-image search.

A detailed description of this parameter can be found in [section 44.3.1^{\(783\)}](#).

44.2.6 Histogram check

Description

To create a histogram, an image is first broken into its three base colors red, green and blue. Then the color values for each pixel are analyzed to partition them into a definable amount of categories (known as buckets when talking about histograms). The actual fill level of each bucket is compared to the expected level. The result of the algorithm is a comparison of the relative frequencies of color categories.

Purpose

Histograms are used for many scenarios. For example it is possible to check for color tendencies or do brightness analyses.

However, histograms are not suitable for checking rather plain-colored images.

Example

The exemplary algorithm

```
algorithm=histogram;buckets=64;expected=0.95
```

results in the following image:

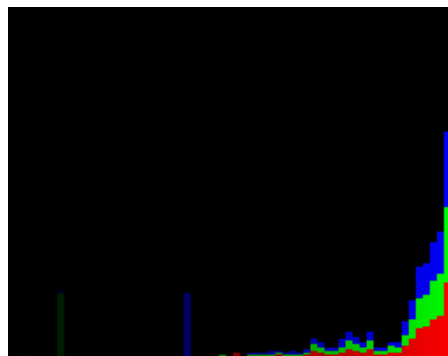


Figure 44.7: Histogram

Parameters

algorithm=histogram

A 'Histogram' should be used for this image check.

buckets

Defines how many buckets to use.

Valid values are a power of 2 between 2 and 256.

expected (optional, but recommended)

Defines the minimal match probability for the check to succeed.

Valid values are between 0.0 and 1.0. If not defined, use 0.98.

resize (optional)

Defines, if the actual image should be resized before calculation to match the size of the expected image.

Valid values are "true" and "false".

find (optional)

Defines an image-in-image search.

A detailed description of this parameter can be found in [section 44.3.1^{\(783\)}](#).

44.2.7 Analysis with Discrete Cosine Transformation

Description

The Discrete Cosine Transformation is a real-valued, discrete, linear, orthogonal transformation which transforms the discrete signal from local range to frequency range.

After transforming an image, you can eliminate low-order (fast oscillating) frequencies. The remaining high-order (slow oscillating) frequencies with the steady component (zeroth frequency = $0 \cdot \cos(x) + y$) can now be analyzed. You can define how many frequencies per basic color should be used for this image check. You can also specify a tolerance to accept cosine-oscillations as identical which actually differ. Low-order frequencies get weighted less than high-order frequencies when calculating the result.

Purpose

The Discrete Cosine Transformation is suitable for many kinds of image checks, which require certain tolerances. The more frequencies are used for analysis the sharper the image check is.

Example

The exemplary algorithm

```
algorithm=dct;frequencies=20;tolerance=0.1;expected=0.95
```

results in the following image:



Figure 44.8: Analysis with Discrete Cosine Transformation

Parameters

algorithm=dct

'Analysis with Discrete Cosine Transformation' should be used for this image check.

frequencies

Defines how many frequencies to analyze.

Valid values are between 0 (steady component only) and the area of the image.

The less frequencies are analyzed the more tolerant the check is. The tolerance is also dependent on the size of the image.

tolerance

Defines the (non-linear) tolerance for accepting different cosine-oscillations as identical.

Valid values are between 0.0 and 1.0.

The value 1.0 means every image matches every other image, because the maximum difference of each frequency is tolerated. A value of 0.0 means frequencies only match if they are exactly the same. A value of 0.1 is a good starting point because only quite similar frequencies are accepted as identical.

expected (optional, but recommended)

Defines the minimal match probability for the check to succeed.
Valid values are between 0.0 and 1.0. If not defined, use 0.98.

resize (optional)

Defines, if the actual image should be resized before calculation to match the size of the expected image.
Valid values are "true" and "false".

find (optional)

Defines an image-in-image search.
A detailed description of this parameter can be found in [section 44.3.1^{\(783\)}](#).

44.2.8 Block-based analysis with Discrete Cosine Transformation

Description

When using this algorithm the image is first partitioned into quadratic blocks with a selectable size (see [section 44.2.4^{\(774\)}](#)). Afterwards every partition is analyzed using a Discrete Cosine Transformation (see [section 44.2.7^{\(778\)}](#)). The final result is the average of all results of these Discrete Cosine Transformations with consideration of the blocks and their weight.

Purpose

The Discrete Cosine Transformation used on the whole image deviates strongly in case of significant brightness differences occurring in the middle of the image because then the steady component (zeroth frequency), which is the highest weighted part, varies strongly. The partitioning circumvents this behavior because now only the affected partitions result in intense deviations while the other partitions stay untouched.

Example

The exemplary algorithm

```
algorithm=dctblock;size=32;frequencies=4;tolerance=0.1;  
expected=0.95  
results in the following image:
```



Figure 44.9: Block-based analysis with Discrete Cosine Transformation

Parameters

algorithm=dctblock

'Blocks for analysis with Discrete Cosine Transformation' should be used for this image check.

size

Defines the size of each block.

Valid values are between 1 and the image size.

frequencies

Defines how many frequencies to analyze.

Valid values are between 0 (steady component only) and the area of a block. The less frequencies are analyzed the more tolerant the check is. The tolerance is also dependent on the size of the image.

tolerance

Defines the (non-linear) tolerance for accepting different cosine-oscillations as identical.

Valid values are between 0.0 and 1.0.

The value 1.0 means every image matches every other image, because the maximum difference of each frequency is tolerated. A value of 0.0 means frequencies only match if they are exactly the same. A value of 0.1 is a good starting point because only quite similar frequencies are accepted as identical.

expected (optional, but recommended)

Defines the minimal match probability for the check to succeed.

Valid values are between 0.0 and 1.0. If not defined, use 0.98.

resize (optional)

Defines, if the actual image should be resized before calculation to match the size of the expected image.

Valid values are "true" and "false".

find (optional)

Defines an image-in-image search.

A detailed description of this parameter can be found in [section 44.3.1^{\(783\)}](#).

44.2.9 Bilinear Filter

Description

This algorithm shrinks the image to a chooseable percental size. Afterwards the image gets resized to its original size by use of a bilinear filter. This filter effects a blurring, because every color value is calculated by use of neighbor pixels.

The final result is the average deviation over all color values and all pixels of this transformed images.

Purpose

Depending on the chosen sharpness the images loose any desired image information. Thus this algorithm is valuable for nearly any scenario.

Example

The exemplary algorithm

```
algorithm=bilinear;sharpness=0.2;expected=0.95
```

results in the following image:



Figure 44.10: Bilinear Filter

Parameters

algorithm=bilinear

A 'Bilinear Filter' should be used for this image check.

sharpness

Defines the sharpness of this bilinear filter.

Valid values are between 0.0 (complete loss of information) and 1.0 (no loss of information).

The sharpness is a linear parameter. This means a value of 0.5 eliminates exactly half (plusmins rounding to entire pixels) of information.

expected (optional, but reommended)

Defines the minimal match probability for the check to succeed.

Valid values are between 0.0 and 1.0. If not defined, use 0.98.

resize (optional)

Defines, if the actual image should be resized before calculation to match the size of the expected image.

Valid values are "true" and "false".

find (optional)

Defines an image-in-image search.

A detailed description of this parameter can be found in [section 44.3.1^{\(783\)}](#).

44.3 Description of special functions

44.3.1 Image-in-image search

Description

The image-in-image search allows to find an expected image within a (larger) image. The check is successful when the expected image can be found anywhere using the defined algorithm. Furthermore, you can determine the position of the match.

The following combination of parameters are valid: `find=best` or `find=anywhere`

```
find=best(resultX, resultY)      or      find=anywhere(resultX,  
resultY)
```

Purpose

The image-in-image search allows to compare images if you don't know the exact position and thus cannot define an offset. The search can be combined with any algorithm and is thus valuable for any purpose.

Example

The exemplary algorithm

```
algorithm=similarity;expected=0.95;find=best(resultX,resultY)
```

uses pixel-based similarity check (see [section 44.2.3^{\(772\)}](#)) to find an image of Q as part of the full image. The got image with highlighted region can be found within the run-log. Besides, the variables resultX and resultY are set to the location of the found image.



Figure 44.11: Image-in-image search: Expected image



Figure 44.12: Image-in-image search: Got image

Parameters

find=best

Defines that the best match should be used.

find=anywhere

Defines that the first match which exceeds the expected match probability should be used.

The image-in-image search uses multiple threads and thus finding anywhere is non-deterministic.

resultX

`resultX` is the name of a QF-Test variable which holds the x-position of the found image.

If a variable for the x-position is defined, a variable for the y-position has to be defined as well (see syntax above).

resultY

`resultY` is the name of a QF-Test variable which holds the y-position of the found image. If a variable for the y-position is defined, a variable for the x-position has to be defined as well (see syntax above).

Chapter 45

Result lists

3.2+

45.1 Introduction

Search operations like locating a reference or searching for a particular value can produce large result sets, as can operations like renaming components or procedures. In order to provide a better overview of all affected nodes QF-Test shows a mass result list at the end of such operations. This list holds all nodes that have been touched by the respective operation. Besides providing an overview that list allows you to apply mass operations on all nodes quite easily. Such mass operations could be toggling a mark at all found nodes or deleting all of them from the test-suites. Those operations can be triggered by the  menu or a the table's context menu.

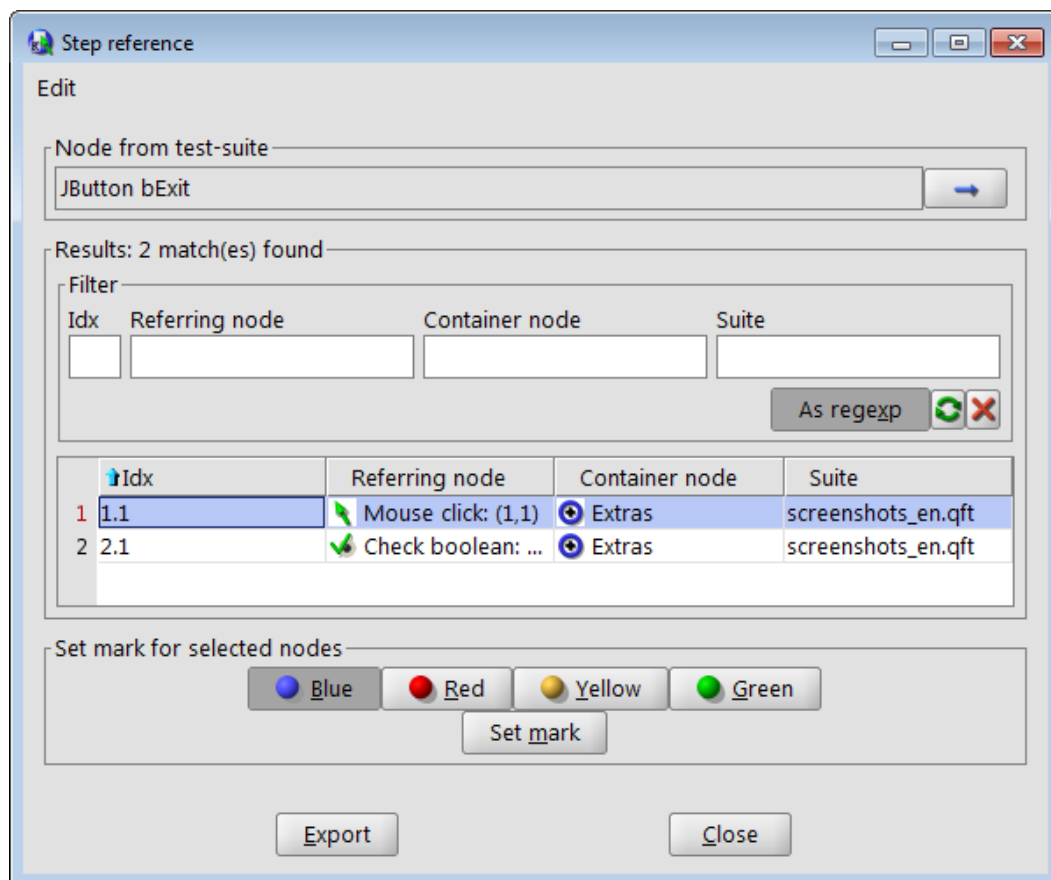


Figure 45.1: Sample result list for 'Locate references'

The table comes with a context menu as well. This context menu allows you to perform several actions on the shown nodes. If you select entries in the result list and perform a right mouse-click you can raise some very useful actions from that menu like jumping to the node or disabling it. Depending on the type of the list there might be specific actions which only make sense in that particular context. A list of possible actions is provided in [section 45.2^{\(788\)}](#).

Such result lists are available for the following actions:

- Once you press *Show result list* on the Search or Replace dialog.
- As result of the **Locate reference** action for callable nodes or components.
- As result of any command which updates several nodes.
- As result of any replacement action from the Replace dialog.
- As result of several analyzing actions under the **Additional node operations** menu.

- Once you open a list of breakpoints under `Debugger→List of all breakpoints...`.
- Once a duplicate QF-Test component ID gets inserted or a suite with duplicate components get opened.
- Once a node transformation changes the type of some child nodes.
- Once you press `Edit→Open error list` in the run-log.
- If you update components via `Update components` all errors will be shown in such a list.

If you have lots of entries in the table you can also apply a filter at the top for limiting the nodes to the given filter values. Once you reset the filter you will see all nodes again.

45.2 Specific list actions

45.2.1 All types of lists

Following actions are available on all types of lists:

- Toggle mark
- Toggle disabled state
- Jump to node in test-suite
- Set a breakpoint
- Delete a node from the list only
- Delete a node from test-suite
- Store the result in a `.qcv` file, for details see [section 45.3^{\(789\)}](#)
- If the according action affects only one node you can also jump back to this.

45.2.2 Replacing

The replacement list provides the capability of replacing only the selected values.

45.2.3 Error list

The error list of the run-log provides capabilities to update failing checks in a bunch. It doesn't allow to delete a node from the run-log of course.

45.3 Exporting and loading results

3.5+

In case you have a very large result list that you need to work on in more than one session, you can store its results in an external file using the *Export* button. The export process creates a `.qcv` file, which is more or less a CSV file holding the information in a QF-Test internal format. To proceed with your work at a later time, simply load the previously exported `.qcv` file via Operations→Load result list... into QF-Test.

Chapter 46

Generic classes

4.0+

QF-Test abstract recorded class from the framework specific classes in order to get classes of common use. Those classes are called generic classes. This concept provides a better readability and clearer understanding of components. Furthermore already created tests can be re-used once the target technology is switched or if you want to maintain tests in various technologies in parallel.

Besided those generic classes QF-Test records generic types as well. Those types give a more detailed specification of the target component. A typical use-case are password fields. Those fields have the generic class `TextField`, but they have that specific characteristic to enter passwords, so they get an additional generic type `TextField:PasswordField`. Using those types makes the recognition of generic classes for certain categories stricter and more appropriate.

46.1 Overview generic classes

Each generic class has dedicated properties, which will be described within this chapter. These classes describe components, which can be used to enter values or to show simple text or images:

Class	Description
ComboBox	Consists of a textfield and a list of selectable options.
Graphics	Shows a graphics or a diagram. Clicks could raise an action, but that's not mandatory.
Icon	Shows an image. Clicks could raise an action, but that's not mandatory.
Label	Shows some text, like a caption. Clicks could raise an action, but that's not necessary.
ProgressBar	Shows the current progress of an action.
Slider	Allows the user to select a value via a thumb.
Spinner	Allows the user to select a value via arrow-buttons combined with a textfield.
Text	Shows common text with multiple rows. A user can't enter any value.
TextArea	Allows the user to enter text, even with multiple lines. It is able to show that text as well.
TextField	Allows the user to enter single-line text. It is able to show that text as well.

Table 46.1: Generic classes for input and output

Those generic classes are normally used for direct interaction with the user like clicks. Those components could have a state:

Class	Description
Button	Raises an action, once it is clicked. It normally doesn't have a defined state.
CheckBox	Has a dedicated state. Usually you can check and un-check those components.
Closer	Closes a component, e.g. the 'X' button of a window.
Link	Allows the user to navigate to another area of the application.
Maximizer	Maximizes the size of a component, e.g. of a window.
MenuItem	Is shown in menus. A click normally raises an action or changes the application's state.
Minimizer	Minimizes the size of a component, e.g. of a window.
RadioButton	Stands for a selectable option. It is typically used for selecting a state, if various states are allowed.
Restore	Re-creates the original size of a component, e.g. of a window.
Sizer	Modifies the size of a component, e.g. of a window.
Thumb	Can be used to thumb through certain values on a slider component.
ToggleButton	Can be clicked like a button and has a certain state as well. Sometimes clicks also raise actions.
ToolBarItem	Stands for a clickable component inside a toolbar.

Table 46.2: Generic classes for interactions

Those components can be applied as containers, to organize other components:

Class	Description
Accordion	Can be used as navigator between components. Components can be expanded and collapsed.
Calendar	Can be used to select a date or time.
ColorPicker	Can be used to select a certain color.
FileChooser	Can be used to select a file. Usually it consists of a textfield, a list of files and a button to choose.
Indicator	Shows a message after an input event. Typically they appear after entering something into textfields, could also contain an icon.
Menu	Contains multiple menu items.
Popup	Shows components, which are only shown after a click on certain buttons and which belong to that button, e.g. the list after clicking on a button of a combobox.
Panel	Contains various components. Can be used to organize the UI.
SplitPanel	Shows components. Those components are located in resizable areas.
TabPanel	Can be used to navigate between components. Those components are shown in dedicated cards ("tabs"). Only one tab can be visible at once.
ToolBar	Stands for a toolbar. It typically contains several menu-items and important buttons to raise most common actions.
ToolTip	A window, which gets opened as some kind of hint. Normally shown, if the user moves the mouse over a component.
Window	Stands for a usual window.

Table 46.3: Generic classes for container components

Besides classes for container components, there are also classes which allows the user to show content in a well-organized structure:

Class	Description
List	Shows multiple values. Values could be selectable as well.
Table	Shows multiple values. It has several dimensions columns. Values could be selectable as well.
TableFooter	Footer row of a table.
TableHeader	Header row of a table.
Tree	Shows content as tree. Can be used to show values in certain categories.
TreeTable	Shows content as tree. Can be used to show values in certain categories. Values normally consist of several columns like in a table.

Table 46.4: Generic classes for data containers

Data itself is implemented with following classes:

Class	Description
Item	Sub-item of a list, can be selectable.
TableCell	A value of a table. It is identified by its column and its position in the row.
TableHeaderCell	Name of a column
TableRow	Row of a table
TreeNode	Sub-item of a tree

Table 46.5: Generic classes for data

Besides classes for input, interaction, data or structuring a couple of classes exist, which only appear in very certain contexts:

Class	Description
BusyPane	Is drawn over other components to lock them.
Divider	Splits areas of certain components. It can be moved.
Expander	Is used to expand or collapse a component, e.g. for a node of a tree.
ScrollBar	Is used for scrolling a component. Usually it contains a something like a thumb.
Separator	Splits areas of certain components. It cannot be moved.
Spacer	Acts as some kind of indentation, e.g. for nodes of a tree.

Table 46.6: Generic classes for specific components

46.2 Mapping of complex components

46.2.1 Mapping of Lists

In order to resolve List components correctly, it's necessary to map the component containing all list entries i.e the list itself as well as the components which represent the individual list entries.

Class	Description
List	Represents the List component, contains all list entries.
Item:ListItem	Represents the individual list entry.
CheckBox:ListItemCheckBox	Optional: Represents a checkbox inside the list entry.
Icon:ListItemIcon	Optional: Represents an icon inside the list entry.

Table 46.7: Mapping of Lists

In case of comboboxes you could use specific combobox classes, like List:ComboBoxList and Item:ComboBoxListItem.

46.2.2 Mapping of TabPanels and accordions

In order to resolve TabPanels or Accordions correctly, it's necessary to map the component containing all entries i.e the TabPanel itself as well as the components which

represent the individual TabPanel entries.

Class	Description
TabPanel	Represents the TabPanel component, contains all tabs.
Item:TabPanelItem	Represents an individual tab.
Panel:TabPanelContent	Optional: Represents a panel containing the actual content of a selecting tab.
CheckBox:TabPanelCloser	Optional: Represents the a Closer button of one tab.
CheckBox:TabPanelCheckBox	Optional: Represents the a CheckBox of one tab.
Icon:TabPanelIcon	Optional: Represents the an Icon of one tab.

Table 46.8: Mapping of TabPanels

In case of Accordion components you can map the classes to Accordion, Item:AccordionItem etc.

46.2.3 Mapping of Tables

In order to resolve Table components correctly, it's necessary to map the component containing all entries i.e the table itself as well as the components which represent the individual rows of a table and the individual table cell entries. Furthermore it could be quite useful to map the row containing all headings as well as the specific heading entries.

Class	Description
Table	Represents the Table component, contains all rows and cells.
TableRow	Represents one table row, could contain multiple cells.
TableCell	Represents one table cell.
TableHeader	Represents one row for headings, could contain multiple headings.
TableHeaderCell	Represents one heading cell.
CheckBox:TableCellCheckBox	Optional: Represents the a CheckBox of one table cell.
Icon:TableCellIcon	Optional: Represents the an Icon of one table cell.
CheckBox:TableCellHeaderCheckBox	Optional: Represents the a CheckBox of one heading.
Icon:TableCellHeaderIcon	Optional: Represents the an Icon of one heading.

Table 46.9: Mapping of Tables

In case of TreeTables, simply replace Table with TreeTable.

46.2.4 Mapping of Trees

In order to resolve Tree components correctly, it's necessary to map the component containing all entries i.e the tree itself as well as the components which represent the individual tree entires. Furthermore it could be quite useful to map toggler button, which opens and closes a tree node.

Class	Description
Tree	Represents the Tree component, contains all tree nodes.
TreeNode	Represents one tree node.
Expander:TreeNodeExpander	Represents the toggling component which is used to open and close the tree node.
Spacer:TreeNodeSpacer	Optional: Represents the a spacing objects used to create a proper level of the tree node.
CheckBox:TreeNodeCheckBox	Optional: Represents the a CheckBox of one tree node.
Icon:TreeNodeIcon	Optional: Represents the a CheckBox of one tree node.

Table 46.10: Mapping of trees

In case of TreeTables, simply replace Tree with TreeTable.

46.3 Properties of generic classes

QF-Test records each component in a certain way. This depends on its generic class. The default behavior is described below:

46.3.1 Accordion

Recorded as: Component

Coordinates for mouse click: Sub-items or exact co-ordinates

Feature: None

qfs:label: Label close to it, tooltip

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
All items	All items of the accordion	items	All
Selected item	Currently selected item	current_item	All
All items with selection	All items of the accordion incl. their selection state	items_with_selection	At the moment not in web

Table 46.11: Checktypes for Accordion

46.3.2 BusyPane

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: None

qfs:label: None

qfs:type: None

Additional checks: None

46.3.3 Button

Recorded as: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text, tooltip

qfs:label: Own text, label close to it, tooltip, description of icon

qfs:type:

qfs:type	Description
Button:CalendarButton	Button inside a Calendar
Button:ComboBoxButton	Button inside a ComboBox
Button:PaginatorButton	Button to switch pages like in aPaginator
Button:ScrollBarButton	Button to modify the current scrolling value

Table 46.12: Special qfs:type values for Buttons

Additional checks: None

46.3.4 Calendar

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: None

qfs:label: Label close to it, tooltip

qfs:type: None

Additional checks: None

46.3.5 ColorPicker

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: None

qfs:label: None

qfs:type: None

Additional checks: None

46.3.6 CheckBox

Recorded as: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip

qfs:label: Own text, label close to it, tooltip, description of icon

qfs:type:

qfs:type	Description
CheckBox:ComboListItemCheckBox	CheckBox inside a ComboListItem
CheckBox:ListItemCheckBox	CheckBox inside a ListItem
CheckBox:MenuItemCheckBox	CheckBox inside a MenuItem
CheckBox:TableCellCheckBox	CheckBox inside a TableCell
CheckBox:TreeNodeCheckBox	CheckBox inside a TreeNode

Table 46.13: Special qfs:type values for CheckBoxes

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Checked	Check the current selection of that checkbox.	checked	All

Table 46.14: Checktypes for Checkbox

46.3.7 Closer

Recorded as: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip

qfs:label: Own text, label close to it, tooltip, description of icon

qfs:type:

qfs:type	Description
Closer:AccordionCloser	Closer for accordion items
Closer:TabPanelCloser	Closer for tabs of a TabPanel
Closer:WindowCloser	Closer for window components

Table 46.15: Special qfs:type values for Closer

Additional checks: None

46.3.8 Divider

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: None

qfs:label: None

qfs:type: None

Additional checks: None

46.3.9 ComboBox

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: Associated label, tooltip

qfs:label: Associated label, label close to it, tooltip

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Current value	The current value of that combobox	value	All
Available values	All available vales of that combobox	value	All

Table 46.16: Checktypes for ComboBox

46.3.10 Expander

Recorded as: Not recorded

Coordinates for mouse click: No recorded

Feature: Not recorded

qfs:label: Not recorded

qfs:type:

qfs:type	Description
Expander:TreeNodeExpander	Toggle of expanded/collapsed state of a tree node.

Table 46.17: Special qfs:type values for Expander

Additional checks: None

46.3.11 FileChooser

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: None

qfs:label: None

qfs:type: None

Additional checks: None

46.3.12 Graphics

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: None

qfs:label: Label close to it, tooltip, description of icon

qfs:type: None

Additional checks: None

46.3.13 Icon

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: Own text or tooltip

qfs:label: Own text, label close to it, tooltip, description of icon

qfs:type:

qfs:type	Description
Icon:ComboListItemIcon	Icon of a ComboListItem
Icon:IndicatorIcon	Icon of an Indicator
Icon:ListItemIcon	Icon of a ListItem
Icon:MenuItemIcon	Icon of a MenuItem
Icon:TableCellIcon	Icon of a TableCell
Icon:TreeNodeIcon	Icon of a TreeNode

Table 46.18: Special qfs:type values for Icon

Additional checks: None

46.3.14 Indicator

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: None

qfs:label: None

qfs:type:

qfs:type	Description
Indicator:ErrorIndicator	Shows an error
Indicator:InfoIndicator	Shows an information
Indicator:WarningIndicator	Shows a warning

Table 46.19: Special qfs:type values for Indicator

Additional checks: None

46.3.15 Item

Recorded as: Item or syntax

Coordinates for mouse click: Most appropriate position or center

Feature: Not recorded

qfs:label: Not recorded

qfs:type:

qfs:type	Description
Item:AccordionItem	Selectable tab of an accordion
Item:ComboBoxListItem	Item of a list of a combobox
Item:ListItem	Item of a list
Item:TabPanelItem	Selectable tab of a TabPanel.

Table 46.20: Special qfs:type values for Item

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Item Text	Check for the shown text	item	All
Item visible	Check, whether item is visible	item_visible	All
The item's selected state	Check, whether item is selected	item_selected	All
The item's checked state	Check, whether item is checked	item_checked	All
Item image	Check for the image of that item	item_image	Not for web

Table 46.21: Checktypes for Item

46.3.16 Label

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: Own text or tooltip

qfs:label: Own text, label close to it, tooltip, description of icon

qfs:type:

qfs:type	Description
Label:CalendarLabel	Inside a Calendar
Label:Caption	A caption of any figure or component
Label:PaginatorLabel	Shows the current page inside a paginator
Label:PanelTitle	A title of a panel
Label:WindowTitle	A title of a window

Table 46.22: Special qfs:type values for Labels

Additional checks: None**46.3.17 Link****Recorded as:** Component**Coordinates for mouse click:** Most appropriate position or center**Feature:** Own text or tooltip**qfs:label:** Own text, label close to it, tooltip, description of icon**qfs:type:**

qfs:type	Description
Link:BreadcrumbLink	Special links for navigating inside a Breadcrumb navigation area.

Table 46.23: Special qfs:type values for Links

Additional checks: None**46.3.18 List****Recorded as:** Component**Coordinates for mouse click:** Sub-items or exact co-ordinates**Feature:** None**qfs:label:** Label close to it, tooltip**qfs:type:**

qfs:type	Description
List:ComboBoxList	List of a ComboBox

Table 46.24: Special qfs:type values for List

Additional checks:

Name in Popup	Description	Name of checktype	Engine
All items	All items of the list	items	All
All items with selection	All items of the list incl. their selection state	items_with_selection	At the moment not in web
Selected item	Currently selected item	current_item	All

Table 46.25: Checktypes for List

46.3.19 Maximizer**Recorded as:** Component**Coordinates for mouse click:** Most appropriate position or center**Feature:** Own text or tooltip**qfs:label:** Own text, label close to it, tooltip, description of icon**qfs:type:**

qfs:type	Description
Maximizer:WindowMaximizer	Maximizer button for a window

Table 46.26: Special qfs:type values for Maximizer

Additional checks: None**46.3.20 Menu****Recorded as:** Component**Coordinates for mouse click:** Exact co-ordinates**Feature:** None**qfs:label:** None

qfs:type:

qfs:type	Description
Menu:MenuBar	A menu bar
Menu:PopupMenu	A menu is popping up and disappearing again

Table 46.27: Special qfs:type values for Menu

Additional checks: None

46.3.21 MenuItem

Recorded as: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip

qfs:label: Own text, label close to it, tooltip, description of icon

qfs:type: None

Additional checks: None

46.3.22 Minimizer

Recorded as: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip

qfs:label: Own text, label close to it, tooltip, description of icon

qfs:type:

qfs:type	Description
Minimizer:WindowMinimizer	Minimizer button for window

Table 46.28: Special qfs:type values for Minimizer

Additional checks: None

46.3.23 Popup

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: None

qfs:label: None

qfs:type:

qfs:type	Description
Popup:CalendarPopup	Contains a Calendar
Popup:ColorPickerPopup	Contains a ColorPicker
Popup:ComboBoxPopup	Contains a list of a combobox

Table 46.29: Special qfs:type values for Popup

Additional checks: None

46.3.24 RadioButton

Recorded as: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip

qfs:label: Own text, label close to it, tooltip, description of icon

qfs:type:

qfs:type	Description
RadioButton:ComboListItemRadioButton	RadioButton inside a ComboListItem
RadioButton:ListItemRadioButton	RadioButton inside a ListItem
RadioButton:MenuItemRadioButton	RadioButton inside a MenuItem
RadioButton:TableCellRadioButton	RadioButton inside a TableCell
RadioButton:TreeNodeRadioButton	RadioButton inside a TreeNode

Table 46.30: Special qfs:type values for RadioButtons

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Checked	Check the current selection of that radiobutton.	checked	All

Table 46.31: Checktypes for RadioButton

46.3.25 Restore

Recorded as: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip

qfs:label: Own text, label close to it, tooltip, description of icon

qfs:type:

qfs:type	Description
Restore:WindowRestore	Restore button for a window

Table 46.32: Special qfs:type values for Restore

Additional checks: None

46.3.26 Panel

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: Title, if existing

qfs:label: None

qfs:type:

qfs:type	Description
Panel:AccordionContent	Contains components of an Accordion.
Panel:Breadcrumb	Contains Breadcrumb-Links for quick navigation.
Panel:CollapsiblePanel	Can be expanded and collapsed.
Panel:Footer	Used to show a dedicated footer area.
Panel:Form	Used for entering values as a form.
Panel:Header	Used to show a dedicated header area.
Panel:Legend	Contains components which act as a legend of graphics.
Panel:MainPanel	Unique main panel of an application.
Panel:Paginator	Contains buttons to switch pages.
Panel:OptionGroup	Contains several RadioButtons.
Panel:ScrollPanel	Is scrollable and contains a ScrollBar.
Panel:TabPanelContent	Contains components of the selected TabItem of a TabPanel.
Panel:TitledPanel	Has a dedicated title.

Table 46.33: Special qfs:type values for Panel

Additional checks: None

46.3.27 ProgressBar

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: Tooltip

qfs:label: Label close to it, tooltip

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Value	The current value	value	All

Table 46.34: Checktypes for ProgressBar

46.3.28 ScrollBar

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: None

qfs:label: None

qfs:type: None

Additional checks: None

46.3.29 Separator

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: None

qfs:label: None

qfs:type: None

Additional checks: None

46.3.30 Sizer

Recorded as: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip

qfs:label: Own text, label close to it, tooltip, description of icon

qfs:type:

qfs:type	Description
Sizer:WindowSizer	Sizer button of a window

Table 46.35: Special qfs:type values for Sizer

Additional checks: None

46.3.31 Slider

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: Tooltip

qfs:label: Label close to it, tooltip

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Value	The current value	value	All

Table 46.36: Checktypes for Slider

46.3.32 Spacer

Recorded as: Not recorded

Coordinates for mouse click: Not recorded

Feature: Not recorded

qfs:label: Not recorded

qfs:type:

qfs:type	Description
Spacer:TreeNodeSpacer	Spacer component of a TreeNode

Table 46.37: Special qfs:type values for Spacer

Additional checks: None

46.3.33 Spinner

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: Tooltip

qfs:label: Label close to it, tooltip

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Value	The current value	value	All

Table 46.38: Checktypes for Spinner

46.3.34 SplitPanel

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: None

qfs:label: None

qfs:type: None

Additional checks: None

46.3.35 Table

Recorded as: Component

Coordinates for mouse click: Sub-items or exact co-ordinates

Feature: None

qfs:label: Label close to it, tooltip

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Column	All values of a certain column	column	All
Column visible	Check, whether column is visible	column_visible	All
Column with selection	All values of a certain column including the cell's selected state	column_with_selection	All
Column title	Check for the column's title	header	All
Row	All values of a certain row	row	All

Table 46.39: Checktypes for Table

46.3.36 TableCell

Recorded as: Item or syntax

Coordinates for mouse click: Most appropriate position or center

Feature: Not recorded

qfs:label: Not recorded

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Cell	Text of that cell	item	All
The cell's visible state	Check whether table cell is visible.	item_visible	All
The cell's selected state	Check whether table cell is selected.	item_selected	All
The cell's editable state	Check whether table cell is editable.	item_editable	All
The cell's checked state	Check whether content inside the tablecell is checked, e.g a Check-Box.	item_checked	All
Cell image	Check for the image of that cell.	item_image	All

Table 46.40: Checktypes for TableCell

46.3.37 TableFooter

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: None

qfs:label: None

qfs:type: None

Additional checks: None

46.3.38 TableHeader

Recorded as: Component

Coordinates for mouse click: Sub-items or exact co-ordinates

Feature: None

qfs:label: None

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
All items	All items of the table header	items	All

Table 46.41: Checktypes for TableHeader

46.3.39 TableHeaderCell

Recorded as: Item or syntax

Coordinates for mouse click: Most appropriate position or center

Feature: Not recorded

qfs:label: Not recorded

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Title	Text of that header cell	item	All
Title visible	Check whether header cell is visible.	item_visible	All
Title image	Check for the image of that header cell.	item_image	All

Table 46.42: Checktypes for TableHeaderCell

46.3.40 TabPanel

Recorded as: Component

Coordinates for mouse click: Sub-items or exact co-ordinates

Feature: None

qfs:label: Label close to it, tooltip

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
All tabs	All tabs of that panel	items	All
Selected tab	Currently selected tab	current_item	All

Table 46.43: Checktypes for TabPanel

46.3.41 TableRow

Recorded as: Not recorded

Coordinates for mouse click: Sub-items or exact co-ordinates

Feature: None

qfs:label: None

qfs:type: None

Additional checks: None

46.3.42 Text

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: None

qfs:label: None

qfs:type:

qfs:type	Description
Text:IndicatorText	Text of an Indicator
Text:ToolTipText	Text of a tooltip

Table 46.44: Special qfs:type values for Text

Additional checks: None

46.3.43 TextArea

Recorded as: Component

Coordinates for mouse click: Sub-items or exact co-ordinates

Feature: Associated label, tooltip

qfs:label: Associated label, label close to it, tooltip

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
All lines	All lines of a TextArea	items	All
Editable	Check whether area could be modified	editable	All

Table 46.45: Checktypes for TextArea

46.3.44 TextField

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: Associated label, tooltip

qfs:label: Associated label, label close to it, tooltip

qfs:type:

qfs:type	Description
TextField:CalendarTextField	Inputfield of a Calendar
TextField:ComboBoxTextField	Inputfield of a ComboBox
TextField:PasswordField	Inputfields for passwords
TextField:SpinnerTextField	Inputfield of a Spinner

Table 46.46: Special qfs:type values for TextField

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Editable	Check whether textfield could be modified	editable	All

Table 46.47: Checktypes for TextField

46.3.45 Thumb

Recorded as: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip

qfs:label: Own text, label close to it, tooltip, description of icon

qfs:type: None

Additional checks: None

46.3.46 ToggleButton

Recorded as: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip

qfs:label: Own text, label close to it, tooltip, description of icon

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Checked	Check the current selection of that button.	checked	All

Table 46.48: Checktypes for ToggleButton

46.3.47 ToolBar

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: None

qfs:label: None

qfs:type: None

Additional checks: None

46.3.48 ToolBarItem

Recorded as: Component

Coordinates for mouse click: Most appropriate position or center

Feature: Own text or tooltip

qfs:label: Own text, label close to it, tooltip, description of icon

qfs:type: None

Additional checks: None

46.3.49 ToolTip

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: Own text, tooltip

qfs:label: None

qfs:type: None

Additional checks: None

46.3.50 Tree

Recorded as: Component

Coordinates for mouse click: Sub-items or exact co-ordinates

Feature: None

qfs:label: Label close to it, tooltip

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
All nodes	All nodes in the tree	items	Not in web
All nodes with nesting	All nodes in the tree including their nesting level	nested_nodes	Not in web
All visible nodes	All nodes in the tree which are currently visible	visible_nodes	All
All visible nested nodes with nesting	All visible nodes of that tree, including their nesting level	visible_nested_nodes	All

Table 46.49: Checktypes for Tree

46.3.51 TreeNode

Recorded as: Item or syntax

Coordinates for mouse click: Sub-items or exact co-ordinates

Feature: Not recorded

qfs:label: Not recorded

qfs:type: None

Additional checks:

Name in Popup	Description	Name of checktype	Engine
Node	Text of that node	item	All
Node visible	Check whether node is loaded	item_visible	All
The node's selected state	Check whether node is selected	item_selected	All
The node's checked state	Check whether node is checked	item_checked	All
Sub-nodes with nesting	Check for all child nodes including their nesting level	nested_nodes	Not in web
All visible sub-nodes with nesting	Check for all visible child nodes including their nesting level	visible_nested_nodes	All
Node image	The image of that node	item_image	All

Table 46.50: Checktypes for TreeNode

46.3.52 TreeTable

Recorded as: Component

Coordinates for mouse click: Sub-items or exact co-ordinates

Feature: None

qfs:label: Label close to it, tooltip

qfs:type: None

Additional checks: All checks of Table and Tree objects should be possible.

46.3.53 Window

Recorded as: Component

Coordinates for mouse click: Exact co-ordinates

Feature: Title, if existing

qfs:label: None

qfs:type:

qfs:type	Description
Window:Dialog	An independent window, used for entering values or to confirm a message.
Window:EmbeddedWindow	Window of an external application, embedded to the current SUT.
Window:InternalWindow	Window showing content of another area of the SUT.
Window:Notification	Showing notifications

Table 46.51: Special qfs:type values for Window

Additional checks: None

Appendix A

FAQ - Frequently Asked Questions

Evaluation and licensing

1. Is an evaluation version available for download?

Yes. Please visit www.qfs.de/en/qftest/download.html.

2. Do I need anything else?

Normally a license file is required to run QF-Test. It will run without a license, but it will not let you save any files or load any files that were not provided by Quality First Software GmbH. This is sufficient for getting a first impression, working through the tutorial and making a first attempt at running your application under QF-Test. To go beyond that, you'll need a license file.

3. So how do I get a license?

You can obtain a free evaluation license valid for 4 weeks by filling in the request form at www.qfs.de/cgi-bin/eval.

4. How much does QF-Test cost?

License types and prices for QF-Test are listed at www.qfs.de/en/qftest/license.html.

5. Does QF-Test need an additional license server?

No. QF-Test handles multi user license management itself.

Support, training and feedback

6. Where do I get help troubleshooting?

- Before asking for help, please read through this FAQ to see if your question has already been answered.
- For beginners the learning-by-doing Tutorial proves useful, further questions are answered by the Manual.

- Webinars for beginners every Monday 16:00 CEST. Registration via qfs@qfs.de.
- There is a mailing list for QF-Test questions and discussions that you can subscribe to (see www.qfs.de/mailman/listinfo/qftest-list). At www.qfs.de/archive/qftest-list the list archive is available which contains lots of helpful postings (full text search possible).
- During evaluation period of QF-Test we grant free support via email qftest@qfs.de or phone +49 (0)8171 3864820.
- For customers we offer support with a guaranteed response time in the form of our Software Maintenance Agreement (see www.qfs.de/en/qftest/support.html).

7. What about training for QF-Test?

QF-Test trainings for beginners and advanced users in German and English language take place regularly here at QFS. There is also the option for webinar-based or on-site consulting and training. Details can be found at www.qfs.de/en/qftest/training.html.

8. How can I request an enhancement to QF-Test?

Enhancement requests should preferably be sent to the mailing list (see above), but they are also welcome anytime at qftest@qfs.de.

9. Where do I report a QF-Test bug?

Simply send an email to qftest-bug@qfs.de. and we will have a look. Please be sure to provide as much information as possible, especially test-suites and run-logs.

Test execution

10. Why do I get warnings of the Form *Missing name, I suggest...?*

This is a feature. Please read [chapter 5^{\(37\)}](#) about how names are set with `setName()` and why they are important. The warning helps to find components that don't have a name. It is not for general use, but the default setting makes people aware of the fact that the reliability of tests can be improved tremendously with little effort.

You can turn the warning off with the option `Log warning for missing name(286)`.

11. How do I run a test automatically from the command line, an ant task or some other kind of script?

You can run QF-Test in batch mode through the command line argument `-batch(571)`. Many other command line arguments can be used to configure the test-run. The exit code of QF-Test reflects the outcome of the test. See [chapter 16^{\(138\)}](#), [section 32.2^{\(571\)}](#) and [section 32.3^{\(588\)}](#) for details.

12. Is it possible to test two applications running at the same time in two different JVMs?

Yes, just start two SUT clients with different names. You can then control both of them.

13. I've got a long-running test and QF-Test runs out of memory. How can I prevent that?

To increase the available memory, start QF-Test with the argument `-J-Xmx768m` (or any other value, QF-Test uses up to 512 MB by default). On Windows you can alternatively use the QF-Test Java Configuration tool, available from the Windows system menu. On Unix re-running the QF-Test setup script (`setup.sh`, `setup.ksh`) also lets you adapt memory usage. Of course the amount of memory you can use depends on your computer. Please refer also to [chapter 1^{\(2\)}](#) for further details.

Note

If the memory chosen is too high, Java will not start and therefore also QF-Test.

There are a number of ways to reduce the memory use of QF-Test:

- Make sure that the option [Create compact run-log^{\(307\)}](#) is checked so that all irrelevant nodes are removed from run-logs.
- QF-Test keeps 4 run-logs accessible from the [Run](#) menu by default. Keep the option [Automatically save run-logs^{\(301\)}](#) active so that QF-Test can save these run-logs to files and release their memory.
- Close run-log windows that you no longer need so that the memory for these run-logs can be reclaimed.
- For long-running tests the best option is to create split run-logs (which QF-Test uses as default) so QF-Test can save partial run-logs to files instead of holding the entire run-log in memory. See [section 8.1.4^{\(72\)}](#) for details.
- If the option [Don't create run-log^{\(307\)}](#) is checked, no run-log will be generated at all. This should also be used with caution since it can be extremely difficult to interpret what happened without the help of the run-log. Use split run-logs instead.
- If your SUT prints lots of output you can reduce the number of old clients that are kept around by changing the option [Number of terminated clients in menu^{\(274\)}](#).

14. Hard mouse events und drag'n'drop operations do not work flawlessly, components cannot be found, the run-log contains black or corrupt screenshots. What do I have to take care about for test execution?

GUI testing requires an unlocked, active desktop. That is the only way to ensure that the SUT behaves the same as if a normal user interacts with it.

To make sure that your test environment complies with this requirement, you'll probably need to tweak its setup. This holds true notably for continuous integration and build tools like Jenkins/Hudson (c.f. [chapter 20^{\(187\)}](#)). Otherwise you can run into serious trouble during [Test execution^{\(138\)}](#), for example black screenshots in the run-log (c.f. [section 8.1^{\(68\)}](#)), failing drag'n'drop operations, non-working hard mouse events or even problems

during component recognition (c.f. [chapter 5^{\(37\)}](#)). Java WebStart applications fail to start up.

Before running a GUI test, check whether the following conditions are met:

- QF-Test and the SUT have to run within an active, unlocked user session.
- The Test must not be executed within the Windows service session. It also must not run without an user session.
- During the [Test execution^{\(138\)}](#) with Jenkins/Hudson or similar tools you have to ensure that the Jenkins/Hudson Windows slave does not get started as service. It has to be started either via Windows Autostart or Windows Task Scheduler within a real user session. Please ensure that you select a valid user account at the 'Security options' and that you have 'Run with highest privileges' disabled. The selected user must be logged in during the [Test execution^{\(138\)}](#), e.g. by automatically logging in, and the desktop must not be locked in any way.
- RDP connections must not be minimized or closed, that would result in a locked session. Instead of RDP you should use VNC, Teamviewer or similar tools to observe the running tests. RDP must not be used for the initial user login and start of the related session.

Technical background:

The keywords are 'session 0 isolation'. This means that every user has its own session ID, starting with 1. The session with ID 0 is reserved for services and applications without user context. It is restricted in its functionality and applications which run in this session are isolated from other sessions. Applications with GUI cannot be displayed within this session. When running a GUI application in the service session, it will not be rendered correctly and thus may not behave as expected. If you search for 'session 0 isolation' with your preferred search engine you will get extensive information, especially for Windows Vista and newer.

Due to security reasons, the well-known workaround for Windows XP and Windows 2000 via tscon.exe and redirection of session 0 is not working anymore since Windows Vista.

To get round all that problems you should consider to work with virtual machines, especially from a security point of view. If you execute the tests on a virtual machine, the above mentioned requirements apply for this virtual machine only, not for the host. You can lock the host and don't care about session management on the host.

Scripting

15. How can I access objects in my application that are not components?

You cannot get an object out of the blue, some kind of registry must exist that returns the object from a class static method. Typical examples in the

standard Java API are `java.lang.Runtime.getRuntime()` or `java.awt.Toolkit.getDefaultToolkit()`.

16. Fine, but how do I use these from Jython or Groovy respectively?

This is standard Jython stuff: Simply import the class and call its methods, e.g.

```
from java.lang import Runtime
runtime = Runtime.getRuntime()
```

In Groovy the package `java.lang` gets imported even automatically:

```
def runtime = Runtime.getRuntime()
```

You can access any class of your application the same way, provided the class is declared public. Note that you must use an SUT script node, not a Server script node.

17. How can I access additional Java classes from a script?

To make additional Java classes available to Jython and Groovy, put them in a jar file and place that in QF-Test's plugin directory (see [section 37.2^{\(620\)}](#)).

18. How can I throw an exception from a script?

There are two ways to do that:

- Jython:

```
raise UserException("Some arbitrary message")
```


Groovy:

```
import de.qfs.apps.qftest.shared.exceptions.UserException
throw new UserException("Some arbitrary message")
```
- `rc.check(condition, "Message", rc.EXCEPTION)`
will raise an exception only if the condition is false.

19. Which external editor should I use?

That's a matter of taste, to some even religion. A comprehensive list of editors for all kinds of operating systems that support Python syntax highlighting and other goodies is available at <http://www.python.org/cgi-bin/moinmoin/PythonEditors>. There are probably dozens of suitable editors with syntax highlighting for both Jython and Groovy - jEdit (<http://www.jedit.org>) is only one of them.

Web

20. Why is this small file-upload/file-download dialog showing up before the real file selection dialog gets displayed?

Before a file selection dialog is displayed the QF-Test browser is showing up a small dialog with an OK-/Cancel-button. This helper dialog is needed by QF-Test to get the data out of the file selection dialog since this dialog is created natively by the operating system. After clicking the OK-button in the helper-dialog the native file selection dialog is displayed and you can enter the filename or select the file directly.

21. Why does the replay of an already recorded file-upload-/file-download-sequence fail if I use another browser and how can I bypass that issue?

Depending on the implementation of the file-upload-/file-download on a specific page the replay might get a bit complicated and even vary between different browsers. The QF-Test standard library `qfs.qft` contains a special procedure `qfs.web.input.fileUpload` to handle this. Please use this procedure instead of your recorded sequence if you encounter problems during replay.

22. I get an error page notifying me about untrusted certificates. Unfortunately the standard dialog to add an exception is not working properly. How can I solve this problem?

This problem won't occur in QF-Test version 3.5.1 and higher, because SSL certificates are now accepted automatically. Please use the following workaround if you are using an older version of QF-Test.

There are different approaches to add a certificate using firefox.

Solution 1:

- Open the URL `chrome://pippki/content/certManager.xul` in a QF-Test browser window.
- This will show the certificate manager where you can add the certificate.
- After pressing OK the whole browser window will close.
- When loading the URL which needs a certificate again the page will load without a certificate error.

Solution 2:

- You can run a normal firefox from the command line with the following parameters
`firefox -profile "[path to your userprofile]/.qftest/mozprofile"`
(e.g.: `firefox -profile "c:/Users/user1/.qftest/mozprofile"`)
(Please close all other running firefox instances before executing this command.)
- Load the URL in the browser and add the certificate to the trusted certificates.
- When loading the URL which needs a certificate again the page will load without a certificate error.

23. I'm getting an OutOfMemoryError for the browser. How to increase memory for the QF-Test browser?

In general it is recommended to create the setup sequence by use of the quickstart wizard. In the resulting sequence within the step "Start browser without window" there is a "Start browser" node in which you can specify the maximum memory as part of the Java VM parameters via e.g. `-Xmx384m`, which means 384 MB maximum memory. The current default is 256 MB.

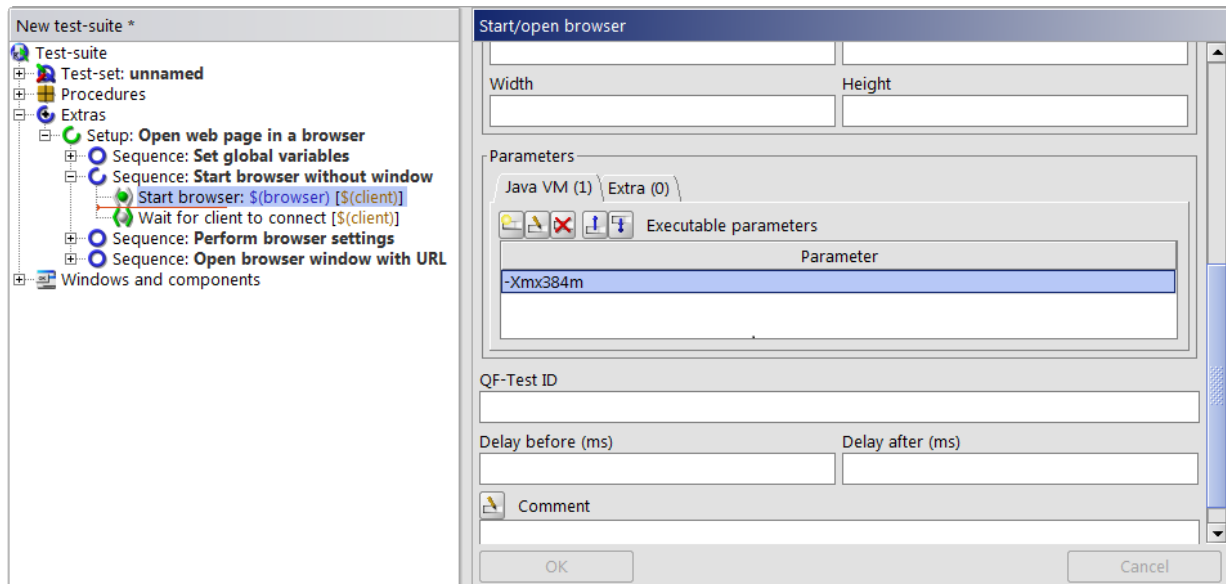


Figure A.1: Set browser maximum memory

24. The web-application opens a popup-window. If I try to close this window using a Window event "WINDOW_CLOSING" the main-window is closed instead of the popup-window at times. How can I ensure that the correct browser-window is closed ?

In order to distinguish those windows QF-Test requires additional information. This information specifies which window should be used for replaying events. It has to be set at the Wait for document to load node as well as for the recorded Web page in the attribute Name of the browser window. You can reach the recorded Web page quite fast via right mouse click at the Wait for document to load node and selecting **Locate component**.

We recommend to set that attribute to `$(default:windowname:)`.

Specifying the attribute Name of the browser window allows QF-Test to evaluate the content of that attribute in addition to the webpage's URL for recognizing the webpage itself and all underlying components. That's why QF-Test is able to distinguish between both windows during replay. Setting that value for the attribute at the Wait for document to load node assigns a new name to that window from QF-Test's perspective. You can see that name in the titlebar of the browser as well.

Before replaying the respective Wait for document to load and events for that popup window you need to set the variable `windowname` to an arbitrary value, e.g. "popup". You could also use a dedicated Set variable node in order to set that variable value. A more convenient way might be to pack all events into a sequence and define the variable `windowname` at the "Variable definitions" table of that sequence.

Now you can replay those events.

It is recommended to reset the variable `windowname` after executing those event nodes again. Otherwise QF-Test will try to replay all subsequent events on that popup window again. Therefore you can use another Set variable node for `windowname` with an empty "Default value". In case of specifying the variable at the sequence node the variable will disappear after that sequence.

If you have already recorded some tests, you should update your test-suite using a global replacement action for the attribute Name of the browser window. This action should set all values of that attribute from empty to `${default:windowname:}`. Therefore choose **Edit→Replace** from the menu and change to the advanced replacement mode via pressing the two golden arrows at the dialogs's toolbar. Now leave the "Search for" field blank and specify `${default:windowname:}` at the "Replace with" field. Then select Name of the browser window for "attribute" and check the checkbox "Whole attribute".

You need to repeat that replacement action for any test-suite which contains event and component nodes for that webpage.

Appendix B

Release notes

B.1 Version 4.0.8 - November 4, 2015

New features:

- Support was added for Firefox version 42.
- Some web components with Javascript code that handles events in special ways do not react to "soft" mouse events. Besides "hard" events that actually move the mouse cursor, QF-Test also implements "semi-hard" events by sending events at OS level to the browser window. The new procedure `qfs.web.general.clickIncreasinglyHard` in the standard library `qfs.qft` makes it easier to choose the best method and keep the tests robust.

Bugs fixed:

- Reading the output of a client process via `${qftest:client.output.<name>}` immediately after the process terminated could occasionally fail and return an empty value.
- The option settings for whether to record generic classes and system classes should have no impact on test replay. However, there was one special exception in case of Swing and SWT: If the 'Feature' attribute of ComboBox popups and lists was of the form 'Invoker class: ...', it was matched only against the class resulting from the current setting of the above options. Now the match is correctly performed against all classes of the invoker component.
- The extra feature 'qfs:class', which holds the most specific class name of a component, is now always recorded and not only if recording of generic classes is activated.

- When performing a successful search operation and starting to edit the highlighted attribute, the modification was lost when continuing the search without confirming the changes.
- When reading data from a custom formatted cell in an Excel sheet that references a non-existing worksheet or file, the custom format was not applied correctly. Also, handling of cells that contain errors has been improved.

JavaFX

- JavaFX Tables and TreeTables with nested column headers are now handled correctly. Also, automatic horizontal scrolling for hard events and image capturing was not quite accurate for these components.

JavaFX

- QF-Test now works around a Java issue where capturing an image of a JavaFX component can mess up the internal geometry of a JavaFX client.

JavaFX

- Recording events in an embedded WebView browser has been improved. Most notably the first event after a page navigation and in general navigation in frames were not handled correctly.

Web

- In special cases registration of the Ext JS resolver could delay startup of a web application up to 20 seconds.

Web

- Even when specifying a dedicated Java program in the `Executable(432)` attribute of the `Start browser(428)` node, QF-Test still took some definitions from its own Java version which might cause problems with applets running in the embedded browser.

B.2 Version 4.0.7 - August 27, 2015

New features:

Web

- Support was added for Firefox version 41.

Web

- Support for the AJAX toolkit Ext JS was updated for Ext JS version 6.

Bugs fixed:

- Memory usage for creating an XML or HTML report from a split run-log has been further reduced.
- Step-by-step replacement of parameters or variable definitions could replace all matching values of a node instead of just the ones chosen.

JavaFX

- For some JavaFX components ItemName- and ItemValueResolver were either not separated cleanly or had no effect.

- Swing**
 - Changes in the Java 9 API were causing problems with forced raising of windows and with replay of a file selection on the AWT FileDialog.
- SWT**
 - Replay of a selection event on a CCombo element used inside a Tree or Table cell could fail due to focus issues.
- Web**
 - Internet Explorer 10 and older could crash, deadlock or weirdly misbehave when running in quirks mode on pages with many nested IFRAMES.
- Web**
 - QF-Test's embedded Internet Explorer now treats Javascript code in href attributes in the same way as the normal Internet Explorer.

B.3 Version 4.0.6 - July 16, 2015

New features:

- Note**
 - Windows 10 is now officially supported. Internet Explorer in particular was causing a number of problems, ranging from crashes to broken documentMode, userAgent or language settings which should all be fixed now. Microsoft's new browser Edge is not supported yet.
- SWT**
 - Support for Eclipse/SWT 4.5 has been updated for the final release of Eclipse 4.5 "Mars".
- Web**
 - Experimental support was added for Firefox version 40.
- Web**
 - Support for the AJAX toolkit RAP was updated for RAP version 3.0. For this version - and back to RAP 2.2 - names set by development via `widget.setData("name", ...)` can now be retrieved automatically by QF-Test, similar to handling names in SWT. See [section 38.2.2^{\(662\)}](#) for further information.

Bugs fixed:

- JavaFX**
 - For JavaFX on Mac OS X QF-Test is now able to force the focus onto the SUT's windows, which is a prerequisite for stable and reliable test execution.
- JavaFX**
 - JavaFX Tables and TreeTables with invisible columns are now handled correctly.
- JavaFX**
 - Sequences recorded for editable JavaFX ComboBoxes were missing the popup Button and thus not replaying correctly. The Button and TextField inside the editable ComboBox are now automatically addressed as `combo@:Button` and `combo@:TextField`. Value checks for ComboBoxes were also improved.

- JavaFX**
 - Custom headers in JavaFX table columns are now handled correctly.
- Web**
 - On Windows the final release of Firefox 39 crashed with QF-Test 4.0.5 due to a change in the memory management of Firefox.
- Web**
 - Firefox version 38.1 ESR was not recognized correctly by QF-Test 4.0.5.
- SWT**
 - Geometry of SWT Widgets inside a titled Group was incorrect for SWT versions 4.4. and 4.5 on Linux.

B.4 Version 4.0.5 - May 12, 2015

New features:

- Web**
 - Support was added for Firefox versions 38 and 39.
 - A new standard check for tool-tips has been added.
- JavaFX**
 - For JavaFX the synchronization of QF-Test event replay with the JavaFX toolkit and its event handling and repaint mechanism has been significantly improved, which makes test replay more robust.
 - The new option Use font with support for international characters⁽²⁴⁵⁾ can be activated to enable support for display of international characters in case QF-Test doesn't detect that requirement automatically.
 - The new Procedure `qfs.utils.dumpComponents` in the standard library `qfs.qft` logs a component (sub-)hierarchy including geometry and class information which may be helpful during test development.
 - Drag&Drop operations between components of different GUI engines, independent applications or the SUT and the desktop can now be performed with the help of the Procedure `qfs.utils.dragAndDrop`.

Bugs fixed:

- Creating a thread dump of the SUT is also working for 64 bit Java VMs on Windows now.
- When recording procedures with a template file that contained an empty parameter list for container procedures, the resulting procedure calls were broken so badly that the test-suite could no longer be saved.
- Memory usage for creating an XML or HTML report from a run-log containing externalized logs has been significantly reduced.

- Swing**
 - Event replay for Swing, which was noticeably slower in QF-Test version 4.0.4, is now back to normal speed.
- JavaFX**
 - In a few special cases the replay of mouse events on JavaFX menus was broken with Java 8 update 40.
- JavaFX**
 - SUT connection with the FX engine now is being delayed until the FX toolkit has been fully initialized. This enables support for applications with embedded FX components that create the FX bridge during startup but initialize the FX toolkit later when the first FX component is shown.
- JavaFX**
 - A possible deadlock for combined Swing and FX applications with a shared event dispatch thread has been fixed.
- Web**
 - Class resolution despite compressed or obfuscated Javascript has been significantly improved so that web applications based on the AJAX toolkit GWT are running much faster now.
- Web**
 - In some cases, the check recording mode in the Chrome browser was not exited cleanly so that subsequent event recording failed.
- Web**
 - Due to a rounding issue, image checks in the Chrome browser were sometimes one pixel off.
- Web**
 - When checking text in web applications, combinations of NBSP and normal space were not always resolved correctly.
- Web**
 - The QF-Test browser was leaking memory when a nested FRAME or IFRAME was unloaded during page navigation.

B.5 Version 4.0.4 - March 5, 2015

New features:

- Web**
 - QF-Test now supports testing 64bit versions of Firefox on Linux.
- Web**
 - Dedicated support was added for the AJAX UI framework jQuery EasyUI.
- Web**
 - Support for Firefox has been updated to match the public release of Firefox version 36 and support for Firefox version 37 has been added.
- SWT**
 - Support for Eclipse/SWT 4.4 "Luna" has been updated for the Eclipse 4.4 SR2 release.

Web

- The Java version for starting QF-Test's browser wrapper is no longer limited to the one QF-Test itself is using. It can now be set in the attribute Executable⁽⁴³²⁾ of the Start browser⁽⁴²⁸⁾ node. This makes it easier to mix 32bit and 64bit Java versions or to use the System JRE for the browser in order to support embedded Java applets.
- The special variable syntax `${qftest:32}` (or its alias `${qftest:32bit}`) can be used to test whether QF-Test is running in a 32bit Java VM, with the obvious counterparts `${qftest:64}` and `${qftest:64bit}` to check for 64bit VM.
- The new command line argument `-daemonrmiport <port>`⁽⁵⁷⁵⁾ can be used to run the QF-Test daemon behind a firewall by specifying a dedicated port for the daemon's RMI communication in addition to the daemon lookup port that is set with `-daemonport <port>`⁽⁵⁷⁵⁾.
- The default value of the option Wait for non-existent component (ms)⁽²⁹¹⁾ is now temporarily set to at least 30000 ms during SUT startup. Adding a Wait for component to appear⁽⁵¹²⁾ node to the SUT startup sequence is still recommended, though less emphatically.

Bugs fixed:

- When automatically updating references after refactoring, multi-level sub-item syntax was cut off after the second item level.
- The new command 'Find/Remove unused callables' treated procedures called only from dependencies as unused.

Swing

- The workaround for the Java 8 focus issues on Windows now handles the root cause instead of trying to keep the side effects under control.

Web

- The standard library procedures in the package `qfs.web.browser.settings` now explicitly check the browser parameter for validity instead of failing silently with no effect.

Web

- Chrome is now able to record Server HTTP request⁽⁵²⁸⁾ and Browser HTTP request⁽⁵³²⁾ nodes.

Web

- Chrome could crash when navigating to a previously visited page anew.

Web

- In some cases auto-expansion of tree nodes in Chrome could cause a deadlock.

Web

- When creating a new startup sequence with the quickstart wizard for the AJAX toolkit GWT the version for the GWT resolver was incorrectly set to 1 instead of 2. The same problem affected auto-detection for GWT.

Web

- The AJAX toolkit jQuery UI was not handled by the auto-detection mechanism for AJAX resolvers.

B.6 Version 4.0.3 - January 29, 2015

New features:

Web

- Support for Firefox has been updated to match the public release of Firefox version 35 and support for Firefox version 36 has been added.

Web

- The resolver for the AJAX framework GWT has been rewritten from scratch and supports a far wider range of GWT components based on GWT version 2.7.

SWT

- Support for Eclipse/SWT 4.5 "Mars" has been added based on Eclipse version 4.5M4.

JavaFX

- Recording of Drag&Drop is now implemented for JavaFX also.

JavaFX

- In addition to JavaFX components embedded in Swing, hybrid applications that work the other way round and embed Swing components in JavaFX are now also supported. This includes support for sharing the same event dispatch thread between JavaFX and Swing.
- The new operation "Find/Remove unused callables" has been introduced in order to get rid of procedures which are never used in your test suites.
- The new procedure `qfs.check.compareDates` in the standard library `qfs.qft` can be used to compare two date strings in arbitrary format.
- The procedure `qfs.utils.xml.compareXMLFiles` has two new parameters for stripping white space and for handling namespace definitions.
- The diagnostic information logged for every error and exception now also includes a full thread dump of the SUT as well as the SUT's Java memory usage and the current system load.
- When executing QF-Test in batch mode for report generation or other commands that don't run any tests, QF-Test now runs in AWT headless mode if the command line argument `-nomessagewindow(578)` is specified. This is useful when using such commands from a continuous integration server that runs without a display.
- Server HTTP request⁽⁵²⁸⁾ nodes can now use the syntax `http://user:passwd@mydomain.com` to send requests to secured servers.
- The default Java memory setting for QF-Test is now 512 MB and the default Java memory for QF-Test's browser wrapper has been increased to 256 MB.

- The new `ResetListener` extension API can be used to influence the behavior of the command `Run→Reset everything`, e.g. to prevent a certain client process from being killed or to perform additional cleanup like deleting Jython or Groovy variables. See [section 39.6^{\(732\)}](#) for details.

Bugs fixed:

- Changing the order of `Catch(408)` nodes under a `Try(405)` node was broken by the addition of the optional `Else(403)` node in QF-Test version 4.0.2.
- Display of large amounts of output from the SUT in the terminal could slow down QF-Test and the test execution if rich text formatting was activated for the terminal.
- In very rare cases test execution in batch mode could hang on Windows when the SUT ran into a deadlock.
- Depending on timing, QF-Test sometimes did not kill all process clients when exiting.
- Importing test-suites could be very slow if tolerant class matching was activated.
- If an Excel sheet contains too deeply nested functions, QF-Test will now throw a `TestException` instead of ignoring those cells. Handling such Excel files requires an increased thread stack size which can be achieved by starting QF-Test with the command line argument `-J-Xss512k`.
- When recording procedures, some placeholders in nested sequences in the template were not replaced correctly.
- The procedure recorder is now able to create container procedures with standard event nodes and sequences without relying on component-specific procedures.
- If an HTTP exception was thrown from a `Server HTTP request(528)` node due to status code `> 399`, the variables `'responseDate'` and `'statusCode'` were not set correctly.
- The horizontal scroll bar of the internal script editor was not set correctly when hard TAB characters were contained in the code.
- Waiting for the absence of a multi-level sub-item now works correctly.
- For WebStart applications QF-Test now also automatically handles the German version of a possible HTTPS certificate warning.
- HTML reports, testdoc and pkgdoc documents could get scrambled by HTML comments split across several lines if HTML pass-through was activated.

- The declaration and encoding of XML reports, testdoc and pkgdoc documents were inconsistent if the default file encoding of QF-Test's Java VM was not ISO-8859-1.
- The tool for the "QF-Test Java Configuration" could not save values to the Windows registry if setup.exe was never run before.
- Swing**
 - When recording in Swing applications with a great number of deeply nested components, performance could suffer severely.
- Swing**
 - In Swing applications running on Java 8, bringing up the menu for check recording could subsequently block input into text fields.
- Swing**
 - The workaround for focus issues in Java 8 on Windows when changing the top-most state of a window has been improved to handle another special case.
- JavaFX**
 - For hybrid JavaFX and Swing applications replaying an event on an embedded component now correctly raises and activates the surrounding window of the other toolkit which improves the reliability of such tests.
- JavaFX**
 - Replaying a file selection in JavaFX for a save dialog now also sets the Extension-Filter to match the chosen file's extension.
- SWT**
 - Trying to record a check directly on the header of an SWT Table caused an exception.
- Web**
 - Third-party plugins and extensions were not initialized correctly for Firefox 30 and above.
- Web**
 - Resolving list items now also works for SELECT nodes and generic ComboBox elements that are located in another list.
- Web**
 - The resolver for the AJAX framework ZK has been updated to version 1.1.1 which fixes a few minor issues and improves handling of MenuItem's.
- Web**
 - Playback of semi-hard mouse events with modifiers like **Ctrl** has been fixed.
- SWT**
 - Checks for table cells in KTable components were not recorded correctly.

B.7 Version 4.0.2-p2

New features:

- Web**
 - Support for Firefox has been updated to match the public release of Firefox version 34 and support for Firefox version 35 has been added.

Bugs fixed:

Web

- In QF-Test version 4.0.2-p1 the user agent string returned by Firefox browsers was incomplete.

B.8 Version 4.0.2-p1

Bugs fixed:

Web

- Support for Firefox version 33 has been updated to cope with the new Firefox versioning scheme introduced with Firefox 33.1.

B.9 Version 4.0.2 - October 22, 2014

New features:

Web

- Support was added for Firefox versions 33 and 34.

SWT

- Support for Eclipse/SWT 4.4 "Luna" has been updated for the Eclipse 4.4 SR1 release.

JavaFX

- Support for JavaFX has been updated for some FX changes in the Java update 8u20.
- The QF-Test Agent mechanism is now compatible with the tighter security restrictions introduced with Java updates 8u25 and 7u71/7u72.

JavaFX

- JavaFX Charts are now fully supported with improved representation of the component parts and various dedicated checks.

Web

- Support for the AJAX toolkits Ext JS and PrimeFaces has been updated to include support for Ext JS version 5 and PrimeFaces version 5 (see [section 38.2.2^{\(659\)}](#) and [section 38.2.2^{\(661\)}](#)).
- The backwards compatibility for recording components with generic classes into test-suites with pre-QF-Test-4.0 components has been extended to include support for merging such components via the import feature.
- The [Try^{\(405\)}/Catch^{\(408\)}/Finally^{\(411\)}](#) construct has been extended to support an [Else^{\(403\)}](#) node which is executed in the case when the contents of the Try are executed cleanly without throwing an exception.

- Window⁽⁵³⁵⁾ nodes now also support Extra feature⁽⁵³⁷⁾ attributes, just like Component⁽⁵⁴⁵⁾ and Web page⁽⁵⁴⁰⁾ nodes.
- It is now possible to override the way how QF-Test determines generic classes by implementing a `GenericClassNameResolver` as described in section 39.1.3⁽⁶⁸¹⁾.
- There is now a dedicated section in the manual explaining how to quickly implement resolvers for AJAX web toolkits that are not yet supported out-of-the-box: section 38.3⁽⁶⁶⁶⁾.

Bugs fixed:

- In some cases the record button could remain active even though no SUT client was connected to QF-Test anymore.
- On a Windows machine with only a 64bit Java installed, QF-Test now explicitly searches for the 64bit Java installation in "C:\Program Files" in case it does not find a `javaw.exe` program on the default PATH.
- The procedures in the `qfs.utils.ssh` package did not use the correct default `known_hosts` file.
- The Excel data file⁽³⁶⁸⁾ node now handles the character encoding of old-style .xls files correctly.
- The runid was not stored correctly in a run-log in case a batch test was run with split run-logs disabled via the command line argument `-splitlog=false`⁽⁵⁸³⁾.
- Check visible tests for items could incorrectly fail with an `IndexNotFoundException` instead of reporting a passed or failed check.
- The QF-Test editor for the proc-builder configuration file could get corrupted while recording new procedures and editing the file in parallel.
- The context diff for a failed Check items⁽⁴⁸⁰⁾ was incomplete in some rare cases.
- Startup for WebStart and applets has been made more robust, including a fix for a `ClassCastException` occasionally seen in applets.
- The newly introduced support for custom tab components assigned via `JTabbedPane.setTabComponent` could break replay of existing recordings.
- Cells of a Nebula Grid component can now be recorded correctly from a scrolled location and also addressed via negative indexes.

Web

Swing

Swing

SWT

- JavaFX**
 - Recording of events on FX controls embedded in an SWT based SUT no longer produces duplicates and checks can now be recorded normally for such FX controls.
- Web**
 - The default compatibility mode for Internet Explorer 10 and up has been fixed and the user agent returned by IE for `document.documentMode` now always correctly reflects the current compatibility mode.
- Web**
 - Waiting for a dynamically inserted FRAME and its document to be loaded now works correctly in all browsers.
- Web**
 - QF-Test now uses a different method for determining the geometry of TEXT nodes in all browsers that is less invasive and more efficient. The old method could occasionally interfere with Javascript code in web pages.
- Web**
 - There was a tricky race condition mostly seen in connection with a special kind of download, triggered by opening a temporary web page which automatically closes immediately. QF-Test now handles such situations and the respective download cleanly.
- Web**
 - The Chrome browser now displays a confirmation dialog when a window gets closed and an `onbeforeunload` handler is defined.
- Web**
 - Component and procedure recording in the Chrome browser is considerably faster.
- Web**
 - The embedded Chrome browser can now show videos.
- Web**
 - Double click events are now correctly recorded in the Chrome browser.

B.10 Version 4.0.1 - August 28, 2014

New features:

- JavaFX**
 - The JavaFX `TreeTableView` component, which was introduced with Java 8, is now fully supported.
- Swing**
 - QF-Test now supports recording and checking of tabs for `JTabbedPane` components which use the `setTabComponent` method to implement some or all tabs as actual components.
 - If the File encoding⁽³⁷⁴⁾ attribute of a CSV data file⁽³⁷²⁾ node is set to "UTF-8", QF-Test now tries to determine itself whether the file starts with a Byte Order Mark (BOM) or not and handle it accordingly. QF-Test also treats an File encoding attribute with "x-UTF-8-BOM" or "x-UTF-8-No-BOM" correctly, even though Java has no direct support for these encodings.

- When moving the mouse over any image stored in a test-suite or run-log, QF-Test now displays the (X,Y) coordinates and color values of the associated pixel.

Bugs fixed:

- The timeout value of the option Wait for non-existent item (ms)⁽²⁹¹⁾ was ignored in some cases, e.g. when performing checks with a timeout.
 - An Extra feature using variables for a regular expression value could mistakenly trigger a `BadRegexpException`.
 - In QF-Test version 4, leading and trailing whitespace is removed from sub-item names. Playback compatibility was provided for most cases, but not all. This should now work consistently.
 - The context diff for a failed Check selectable items⁽⁴⁸⁴⁾ node in the run-log now includes visual representation for the expected and actual selected states and is shown correctly also in cases where there are differences only in the selected state.
 - Formulas in Excel files that return boolean values or that switch cell types more than once are now handled correctly by the Excel data file⁽³⁶⁸⁾ node and the procedure `qfs.utils.files.readExcelFile`.
- Swing**
- In a hybrid Swing and JavaFX application, handling of Swing `JTable` components was partly broken due to interference from the JavaFX `ItemResolver` for tables.
- JavaFX**
- Recording of items in a multi-rooted JavaFX tree (i.e. a tree with its primary root node hidden) is now correct.
- SWT**
- Component highlighting was broken when recording checks in an Eclipse / SWT application.
- SWT**
- There were still dependencies on a newer version of glibc left in the instrumentation files for 64bit versions of Eclipse, causing failures on some Linux systems with an old version of glibc - for example RHEL 6.5.
- Web**
- Tests with Firefox versions 27 and 28 did not work correctly.
- Web**
- Text checks for table headers were broken in some web tables.
- Web**
- A proxy auto-config (PAC) file defined in a procedure call of `qfs.web.browser.settings.setProxy` is now interpreted correctly.
- Web**
- The Chrome browser now displays a confirmation dialog when a window gets closed and an `onbeforeunload` handler is defined.

- Web**
 - Javascript errors in the Chrome browser are no longer printed to the terminal but written to the run-log.
- Web**
 - In the Chrome browser coordinates inside an IFRAME were displaced if the frame had a border defined, leading to offsets when highlighting components and to image check failures caused by insufficient automatic scrolling.
- Web**
 - Drag&Drop via the procedure `qfs.web.general.dragFromTo` now also works with Chrome.
- Web**
 - Clearing the browser cache did not work with Firefox version 32.
- Web**
 - The procedure `qfs.web.table.getRowCount` could fail in the rare case when the first column of a table is invisible.

B.11 Main new features in version 4

Note For a detailed list of new features please see the release notes for QF-Test versions 4.0.0-beta1 through version 4.0.0 below.

The following major new features have been implemented for QF-Test version 4:

Description	Further info
New GUI engine: JavaFX	Full support for JavaFX based applications
Support for Chrome browser	Currently Windows only
Improved support for Java WebStart and applets	Fast and reliable startup and connection via Java agent
Support for the AJAX framework jQuery UI	Special support for various AJAX toolkits ⁽⁶⁵⁷⁾
Generic classes for components	Uniform classes for all GUI engines (section 5.4.1 ⁽⁴²⁾)
Multi-level sub-item concept	QPath concept, similar to XPath (section 6.4 ⁽⁵⁵⁾)
Support for embedded browsers	JavaFX WebView and JxBrowser 4
Improved user interface with colored syntax highlighting	Option <u>Syntax highlighting for tree nodes</u> ⁽²⁴⁴⁾ , <u>Terminal options</u> ⁽²⁷⁶⁾
New package for client-side ssh connections	Standard library <code>qfs.qft</code> , package <code>qfs.utils.ssh</code>

Table B.1: Highlights in QF-Test version 4

Changes that can affect test execution:

- The Wait for component to appear⁽⁵¹²⁾ node implicitly waits for the respective SUT client to connect for the required GUI engine. In case the wait fails, a

ComponentNotFoundException⁽⁵⁵⁸⁾ is thrown with a message that explains whether the component was not found, the respective SUT client is not connected or whether there is no matching GUI engine available for that client. The Expected message⁽⁴⁰⁹⁾ attribute of a Catch⁽⁴⁰⁸⁾ node can be used to differentiate between those causes.

In absence of a suitable client there was an inconsistency in older QF-Test versions, causing either a ComponentNotFoundException or a ClientNotConnectedException⁽⁵⁶²⁾ to be thrown, depending on whether such a client was previously connected or not. In case your test still expects a ClientNotConnectedException you either need to change the Catch node, add an additional Catch or prepend the Wait for component to appear node with a Wait for client to connect⁽⁴³⁸⁾ node.

- The new option How to handle disabled components⁽²⁷²⁾ defines what to do when a disabled Component node is referenced during test execution. In previous versions of QF-Test the disabled state was ignored which sometimes led to confusion. The default now is to log an error. Consequently, disabled Component nodes are no longer re-used when recording.
- There is a new option called Default timeout for checks (ms)⁽²⁹¹⁾ that - as its name implies - defines a default timeout for checks that have no Timeout⁽⁴⁷⁴⁾ attribute set. It increases general stability of check execution. If your tests include a lot of Check nodes without Timeout that are expected to fail, test execution time may increase. In this case you may want to set this option back to 0, the old default or, preferably, update the respective nodes and set their Timeout attribute to 0.
- The AJAX toolkit RAP does not use the 'ID' attribute of DOM nodes to define IDs, so the option Eliminate all numerals from 'ID' attributes⁽²⁹⁴⁾ had no effect for RAP. This has been fixed, so that with the option activated, numerals are now removed from IDs generated for RAP elements. In case this causes problems with existing tests, simply deactivate the option Eliminate all numerals from 'ID' attributes⁽²⁹⁴⁾.

Software that is no longer supported:

Please see section 1.1⁽²⁾ for a detailed list of system requirements and supported technology versions.

QFS is taking pains to support even very old software. QF-Test still runs, for example, under Windows XP or with an SUT based on Java 1.5 or 1.6 but for systems that are no longer supported by their own vendors we cannot provide full support.

Libraries that are required for the Firefox and Eclipse/SWT versions listed below are no longer part of the default installation of QF-Test. The majority is still being maintained, albeit with limited testing. If you need support for such a system, please get in touch with QFS via support@qfs.de.

Web

Note

Official support has been dropped for Java 1.5, Internet Explorer 7 and older, Firefox 25 and older (except Firefox 24 ESR) as well as Eclipse/SWT 3.6 and older.

B.12 Version 4.0.0 - July 31, 2014

New features:

Web

- Support was added for Firefox versions 31 and 32.
- A more user-friendly way of updating a QF-Test license is now available via the menu Help→Update license....
- When opening a Jython or Groovy terminal for an SUT client, the command history from the most recently created Jython or Groovy terminal is copied. This makes it easier to try out the same scripts when repeatedly restarting the SUT.
- When importing QF-Test results into TestLink it is now possible to use the QF-Test ID for associating test-cases and to omit the platform.
- After starting QF-Test the record button now has a help icon that leads to the quickstart wizard for launching an SUT. This feature is intended to assist first-time users that often wonder why they cannot start recording right away.

Web

- The procedure `qfs.web.ajax.installToolkitResolver` now allows specifying default versions for automatically detected AJAX toolkit resolvers. This prevents inadvertent updates to versions with potential incompatibilities in newer QF-Test versions.

Web

- The default severity level for Javascript errors in web pages has been changed to 'Warning'. Current browsers simply ignore Javascript problems by default so that treating those as an error caused too much irritation. The setting can be changed via the option How to handle Javascript errors in a Web application⁽²⁹⁶⁾.

Bugs fixed:

SWT

- On Linux systems like RHEL 6.5 with an old version of glibc, instrumented 64bit versions of Eclipse failed to start. This dependency on a newer version of glibc has been removed.

SWT

- When the QF-Test agent was enabled, an Eclipse / SWT application with embedded Swing components connected as two SUT clients instead of one client with two GUI engines.

JavaFX

- Column header cells in JavaFX tables are now addressed correctly.

- JavaFX**
 - CheckBox elements inside JavaFX table cells or tree nodes are now recorded with the new @:CheckBox item syntax.
- Web**
 - The check for the editable state of TextField, PasswordField and TextAreas nodes was missing for web clients.
- Web**
 - A selection in a ComboBox that initiated a navigation to another page could cause Internet Explorer to crash.
- Web**
 - Check recording in the browser did not work in an area covered by an invisible IFRAME.
- Web**
 - Event synchronization for the AJAX toolkit Ext JS now handles modal dialogs correctly.
- Web**
 - Replay of text input for INPUT elements of type "number" has been fixed.
- Web**
 - When defining proxy settings for Internet Explorer via the `qfs.web.browser.settings.doStartupSettings` procedure, `proxy.pac` files were not handled correctly.
- Swing**
 - In newer Java and Browser versions the startup behavior of applets has changed, sometimes causing two JRE versions to start up simultaneously. Depending on timing, this could cause the real applet to connect as \$(client):2 instead of \$(client). QF-Test now works around this problem by connecting to an applet only after it has become visible.
- Swing**
 - The "All lines" check for Swing JTextArea components was not recorded correctly when clicking below the first line of text.
- Swing**
 - There are now two different text checks for JEditorPane components that display HTML, "Text" and "HTML source". In many cases the check against the plain, displayed text is more useful than the former default check for the HTML source code. Also, the text check for JEditorPane components that display RTF has been fixed.
 - When canceling an edit in the detail view or adding of a new node, QF-Test now checks whether modifications were made and asks for confirmation before discarding those. This confirmation dialog can be suppressed via the option Ask before discarding detail modifications⁽²⁴³⁾.
 - When toggling the disabled state of the selected node, modifications made but not confirmed in the detail view were silently discarded.

B.13 Version 4.0.0-rc1

New features:

- QF-Test now supports multi-level sub-items for addressing components within an item. A typical example is a `TextField` within a table cell that can now be addressed as `table@Header&row@:TextField`. The syntax is extremely powerful with the potential to mix XPath-like syntax with QF-Test's component representation (see [section 6.4^{\(55\)}](#)).
- The former 'Id' attribute of Window and Component nodes is now called QF-Test ID in order to disambiguate it from the various kinds of IDs used by the supported GUI engines.
- Generic classes are now also implemented for standard HTML nodes even without a dedicated AJAX resolver.
- Support for the AJAX toolkit RichFaces was updated to RichFaces version 5.
- Support for Eclipse/SWT has been updated for Eclipse version 4.4 "Luna".
- The Nebula Grid component is now supported by default and automatically recognized.
- It is now possible to define a directory for saving temporary run-logs in interactive mode via the option Directory for run-logs⁽³⁰¹⁾. During QF-Test startup this directory can be specified on the command line via `-runlogdir <directory>(582)`.
- The new option Directory map for test-suites⁽³⁰⁸⁾ can be used to map directories for test-suites. This is very useful when working with a run-log from a machine with a different file layout (e.g. Windows vs. Unix) because with the correct directory mapping QF-Test can locate a test-suite without manual help.
- There are two new special variable groups to conveniently escape special characters. `${quoteitem:...}` deals with characters like '@' or '&' in a textual sub-item index and `${quoteregexp:...}` handles '.', '(', '[' etc. in a regular expression.
- It is now possible to specify a directory as a bookmark (see [section 29.1.3^{\(246\)}](#)). When such a bookmark is selected, the file selection dialog is opened directly for this directory.

Bugs fixed:

- Multiple instances of the Chrome browser can now run simultaneously on one machine.

Web

- Performance, stability and supported feature set for browsers embedded via JavaFX WebView and JxBrowser were improved.

Web

- The AJAX toolkit RAP does not use the 'ID' attribute of DOM nodes to define IDs, so the option Eliminate all numerals from 'ID' attributes⁽²⁹⁴⁾ had no effect for RAP. This has been fixed, so that with the option activated, numerals are now removed from IDs generated for RAP elements. In case this causes problems with existing tests, simply deactivate the option Eliminate all numerals from 'ID' attributes⁽²⁹⁴⁾.
- With support for JavaFX and its Swing InterOp embedding capabilities as well as support for embedded browsers, SUT clients often have more than one GUI engine. This caused problems with SUT script nodes in the standard library `qfs.qft` which were relying on the default GUI engine. All procedures with SUT scripts now have an optional parameter named `guiengine` which is predefined correctly, so that for example a Procedure in the Package `qfs.web` automatically works correctly in an embedded browser. In the exceptional case of an SUT client with more than one GUI engine of the same kind you can define the `guiengine` parameter to address a particular engine.
- After introducing generic classes, replay was broken in case an anonymous inner class was manually assigned for a component.
- During dependency resolution, Parameter default values of Dependency nodes were not correctly bound.
- Performance of test execution in interactive mode suffered from repaint issues caused by the new syntax highlighting in the tree view.

B.14 Version 4.0.0-beta3

New features:

Web

- Support was added for Firefox version 30.

JavaFX

- Procedure recording is now fully supported for JavaFX.

Web

- For web applications based on the AJAX framework ZK QF-Test now resolves the original developer-assigned IDs and uses them as names for components.

Web

- Support for the Chrome browser is continuously improving. Up- and download are now fully supported as well as login dialogs, closing of browser windows, key and text input including modifiers and cursor position. Also, there now is a custom profile folder to separate testing from normal use with support for cookie and cache cleanup, locale and proxy settings.

Web

- Embedded browsers in JavaFX WebView or via JX Browser are also making headway and multiple embedded browsers in one window are now handled correctly.
- Simple but very useful: The dialog shown when an error or exception occurs during a test now has an additional button for jumping directly to the respective node in the run-log.

JavaFX

- A check was added for the checked state of a CheckBox in JavaFX table cells, tree nodes and list entries.
- The new procedure `qfs.web.general.dragFromTo` supports HTML5 style drag and drop operations without blocking the application, alternatively at Javascript level or via hard events.

Bugs fixed:

SWT

- The introduction of generic classes for SWT caused a serious performance issue that has been fixed.

SWT

- For SWT QF-Test refused to replay hard events on negative coordinates which, however, are necessary to reach a window on a second monitor placed to the left of the main screen on Windows.

Web

- The print dialog is now shown correctly in Firefox.
- In very special cases running a test in a QF-Test daemon could fail due to a missing engine license even though the license was available.

B.15 Version 4.0.0-beta2

New features:

- Generic classes are now also implemented for Swing and SWT.
- QF-Test records generic class, custom class and system class as extra features named `qfs:genericclass`, `qfs:class` and `qfs:systemclass` respectively. This is useful information and can be used to switch individual components to a more specific or less specific class. In case the generic class also has a more specific sub-type QF-Test also records the extra feature `qfs:type`, e.g. "PanelTitle" for a Label that is the title of a TitledPanel.

Web

- Supported standard functionality for Chrome now includes support for downloads, handling prompt and confirm dialogs as well as getting and setting the cursor position in text fields which is required for recording mouse events relative to a

character position and for Text input nodes which insert text into the middle of existing text.

- Support for embedded browsers now includes JavaFX WebView and JxBrowser 4. Though this is still work in progress, basic capture and replay for mouse events already works.
- When started from QF-Test Chrome now also automatically accepts SSL certificates for HTTPS connections.
- It is now possible to choose between different layouts of the test-suites in the selection dialog for Procedures.

Bugs fixed:

- JavaFX**
 - Thanks to improved focus management replay for JavaFX on Linux should be more reliable now, especially selection of menu items.
- JavaFX**
 - Replaying hard mouse click for the middle or right button did not work correctly with JavaFX.
- An Excel 2010 sheet with a cross-sheet reference to a pure number is now handled correctly by the Excel data file node.

B.16 Version 4.0.0-beta1

New features:

- JavaFX**
 - JavaFX is the successor of Swing as the standard UI Toolkit in Java 8. QF-Test version 4.0 introduces the new product line QF-Test/fx that fully supports testing JavaFX applications. This GUI engine is nearly complete. Missing at this point is recording of procedures and support for handling the embedded WebView as a full-featured QF-Test Web engine is incomplete.
 - Support for the Chrome browser on Windows is well advanced and mostly stable. As before with Internet Explorer and Firefox the goal is to hide the browser differences and enable a single test to run unchanged on all three browsers. Not implemented at this point are various browser dialogs as well as file up- and download. Another current restriction is that only one instance of Chrome can be run by QF-Test in parallel.
 - The concept of generic classes, initially available for the web engine only, has been extended to support JavaFX and soon all GUI engines. It is the basis for

interoperable tests that work for different GUI engines, provides a higher level of abstraction and more flexible and robust component recognition. Recording of generic classes is configured via the option Record generic class names for components⁽²⁶³⁾.

See the video Dealing with the explosion of complexity in web test automation for a vivid demonstration of the explosion of complexity in AJAX user interfaces. This applies just as well to JavaFX which shares the automation problems of a UI built from a large number of small GUI elements and for which QF-Test's approach with generic classes fits perfectly.

- Tolerant class matching when recording: Existing nodes in QF-Test's component hierarchy will be reused as long as they match one of the classes of the recorded component. This means that when switching to/from generic class recording, no components will be forcibly re-recorded. This has no impact on the 'Update components' feature which always updates the respective components according to the current recording option settings.
 - Connection with the SUT is now implemented via the Java agent mechanism by default. It is faster, more stable - especially for Java WebStart and Applets - and much more flexible and powerful. In case of problems the Java agent mechanism can be disabled via the option Connect via QF-Test agent⁽²⁷⁵⁾.
 - During SUT startup Jython and Groovy are now loaded on demand, leading to faster and more reliable startup. Jython in particular was one of the main reasons for problems with Java WebStart. This also reduces the overhead and memory footprint if either of the languages is not used at all.
- Web**
- Support was added for the AJAX framework jQuery UI.
- SWT**
- Eclipse/SWT support was added for the Eclipse 4.4 "Luna".
- SWT**
- The Nebula NatTable component is now supported by default and automatically recognized.
 - Colored syntax highlighting of tree nodes was added which significantly improves readability of test-suites.
 - The shared terminal now provides coloring of different message types. The new group of Terminal options⁽²⁷⁶⁾ encloses a bunch of new settings to adjust text display in the shared terminal. Font size and type can be defined as well as a number of regular expressions to highlight and/or filter arbitrary text in the output.
 - It is now possible to set breakpoints on Component⁽⁵⁴⁵⁾ nodes so as to enter the debugger as soon as the node gets referenced.

- For a failed Check items⁽⁴⁸⁰⁾ node both run-log and report now contain a context diff of the expected and actual values so that all deviations can quickly be identified.
- Relative Test calls⁽³²⁷⁾ with a leading '.' in the Test name⁽³²⁸⁾ attribute are now supported.
- In places where QF-Test supports regular expressions, e.g. Text⁽⁴⁷³⁾, Feature⁽⁵⁴⁷⁾ or Items⁽⁴⁸²⁾, normal text can now be turned into a properly quoted regular expression by right-clicking and selecting 'Escape text for regular expressions' from the context menu.
- Handling of items with multi-line text, e.g. tree nodes, table headers or cells, list items etc. has been improved.
- Search and replace operations for variables can now be limited further to cover only variable names or variable values.
- The manual chapter on integration with HP ALM - QualityCenter was rewritten from scratch: section 19.1⁽¹⁶³⁾.

Changes that can affect test execution:

- The Wait for component to appear⁽⁵¹²⁾ node implicitly waits for the respective SUT client to connect for the required GUI engine. In case the wait fails, a ComponentNotFoundException⁽⁵⁵⁸⁾ is thrown with a message that explains whether the component was not found, the respective SUT client is not connected or whether there is no matching GUI engine available for that client. The Expected message⁽⁴⁰⁹⁾ attribute of a Catch⁽⁴⁰⁸⁾ node can be used to differentiate between those causes.
In absence of a suitable client there was an inconsistency in older QF-Test versions, causing either a ComponentNotFoundException or a ClientNotConnectedException⁽⁵⁶²⁾ to be thrown, depending on whether such a client was previously connected or not. In case your test still expects a ClientNotConnectedException you either need to change the Catch node, add an additional Catch or prepend the Wait for component to appear node with a Wait for client to connect⁽⁴³⁸⁾ node.
- The new option How to handle disabled components⁽²⁷²⁾ defines what to do when a disabled Component node is referenced during test execution. In previous versions of QF-Test the disabled state was ignored which sometimes led to confusion. The default now is to log an error. Consequently, disabled Component nodes are no longer re-used when recording.
- There is a new option called Default timeout for checks (ms)⁽²⁹¹⁾ that - as its name implies - defines a default timeout for checks that have no Timeout⁽⁴⁷⁴⁾ attribute set.

It increases general stability of check execution. If your tests include a lot of Check nodes without Timeout that are expected to fail, test execution time may increase. In this case you may want to set this option back to 0, the old default or, preferably, update the respective nodes and set their Timeout attribute to 0.

New packages and procedures in the standard library `qfs.qft`

JavaFX

- The new `qfs.fx` package holds a full set of procedures for interacting with JavaFX components.
- The new `qfs.utils.ssh` package provides procedures for client-side ssh connections.
- There's a new check utility procedure called `qfs.check.checkValueInRange`.

Swing

- Various new procedures for interacting with Swing components:
 - `qfs.swing.checkbox.deselectWithCoordinates`
 - `qfs.swing.checkbox.selectWithCoordinates`
 - `qfs.swing.checkbox.setWithCoordinates`
 - `qfs.swing.general.clickAtComponentWithDetails`
 - `qfs.swing.general.doClickWithDetails`
 - `qfs.swing.general.waitForComponent`
 - `qfs.swing.general.waitForComponentForAbsence`
 - `qfs.swing.list.clickItemWithDetails`
 - `qfs.swing.menu.selectItemWithCoordinates`
 - `qfs.swing.menu.selectSubItemWithCoordinates`
 - `qfs.swing.menu.setCheckItemWithCoordinates`
 - `qfs.swing.menu.setSubCheckItemWithCoordinates`
 - `qfs.swing.menu.selectMenuItemInHierarchyWithCoordinates`
 - `qfs.swing.menu.selectCheckMenuItemInHierarchyWithCoordinates`
 - `qfs.swing.popupmenu.clickItemWithDetails`
 - `qfs.swing.popupmenu.selectSubItemWithCoordinates`
 - `qfs.swing.popupmenu.setCheckItemWithCoordinates`
 - `qfs.swing.popupmenu.setSubCheckItemWithCoordinates`
 - `qfs.swing.radiobutton.selectWithCoordinates`
 - `qfs.swing.table.checkbox.deselectWithCoordinates`
 - `qfs.swing.table.checkbox.selectWithCoordinates`
 - `qfs.swing.table.checkbox.setWithCoordinates`
 - `qfs.swing.table.clickCellWithDetails`
 - `qfs.swing.tree.clickNodeWithDetails`
 - `qfs.swing.tree.collapseWithCoordinates`
 - `qfs.swing.tree.collapseNodeWithCoordinates`
 - `qfs.swing.tree.expandWithCoordinates`


```
qfs.swing.tree.expandNodeWithCoordinates  
qfs.swing.tree.expandNodesWithCoordinates
```

SWT

- Various new procedures for interacting with SWT components:

```
qfs.swt.checkbox.deselectWithCoordinates  
qfs.swt.checkbox.selectWithCoordinates  
qfs.swt.checkbox.setWithCoordinates  
qfs.swt.general.clickAtComponentWithDetails  
qfs.swt.general.doClickWithDetails  
qfs.swt.general.waitForComponent  
qfs.swt.general.waitForComponentForAbsence  
qfs.swt.list.clickItemWithDetails  
qfs.swt.menu.selectItemWithCoordinates  
qfs.swt.menu.selectSubItemWithCoordinates  
qfs.swt.menu.setCheckItemWithCoordinates  
qfs.swt.menu.setSubCheckItemWithCoordinates  
qfs.swt.menu.selectMenuItemInHierarchyWithCoordinates  
qfs.swt.menu.selectCheckMenuItemInHierarchyWithCoordinates  
qfs.swt.popupmenu.clickItemWithDetails  
qfs.swt.popupmenu.selectSubItemWithCoordinates  
qfs.swt.popupmenu.setCheckItemWithCoordinates  
qfs.swt.popupmenu.setSubCheckItemWithCoordinates  
qfs.swt.radiobutton.selectWithCoordinates  
qfs.swt.table.checkbox.deselectWithCoordinates  
qfs.swt.table.checkbox.selectWithCoordinates  
qfs.swt.table.checkbox.setWithCoordinates  
qfs.swt.table.clickCellWithDetails  
qfs.swt.tree.clickNodeWithDetails  
qfs.swt.tree.collapseWithCoordinates  
qfs.swt.tree.collapseNodeWithCoordinates  
qfs.swt.tree.expandWithCoordinates  
qfs.swt.tree.expandNodeWithCoordinates  
qfs.swt.tree.expandNodesWithCoordinates
```

Web

- Various new procedures for interacting with web components:

```
qfs.web.checkbox.deselectWithCoordinates  
qfs.web.checkbox.selectWithCoordinates  
qfs.web.checkbox.setWithCoordinates  
qfs.web.general.clickAtComponentWithDetails  
qfs.web.general.doClickWithDetails  
qfs.web.general.waitForComponent  
qfs.web.general.waitForComponentForAbsence  
qfs.web.list.clickItemWithDetails
```

```
qfs.web.radiobutton.selectWithCoordinates  
qfs.web.table.clickCellWithDetails
```

Software that is no longer supported:

QFS is taking pains to support even very old software. QF-Test still runs, for example, under Windows XP or with an SUT based on Java 1.5 or 1.6 but for systems that are no longer supported by their own vendors we cannot provide full support.

Libraries that are required for the Firefox and Eclipse/SWT versions listed below are no longer part of the default installation of QF-Test. The majority is still being maintained, albeit with limited testing. If you need support for such a system, please get in touch with QFS via support@qfs.de.

Official support has been dropped for Java 1.5, Internet Explorer 7 and older, Firefox 25 and older (except Firefox 24 ESR) as well as Eclipse/SWT 3.7 and older.

B.17 Version 3.5.5

New features:

- Web** • Support for Firefox version 27 was updated and new support added for Firefox version 28.
- SWT** • Eclipse/SWT support was updated for service release 2 of Eclipse version 4.3.
- The driver test-suites and documentation for the integration with imbus TestBench were updated.
- Web** • QF-Test is now less restrictive when comparing document URLs or href attributes, so a simple change of a hostname, IP address or port no longer requires working with regular expressions in the `Feature` attribute.

Bugs fixed:

- Focus handling for replay with Java 8 has been improved and QF-Test now works around a bug in Java 8 when setting a window topmost on Windows systems.
- Web** • Proxy settings for Internet Explorer are now application-local, so even after a crash the system proxy settings remain unchanged.
- Web** • The user agent for Internet Explorer 11 is now reported correctly by QF-Test's embedded browser.
- Web** • Firefox confirmation dialogs for closing or navigating away from a web page are now handled correctly and can be interacted with.

SWT

- Input of Unicode characters from Text input⁽⁴⁵⁴⁾ nodes with Replay single events⁽⁴⁵⁶⁾ set was not replayed correctly on some Linux systems.
- Closing the QF-Test main workbench with a run-log still open, then jumping from that run-log to a test-suite caused the saved session of opened test-suites to get lost.

B.18 Version 3.5.4

New features:

Web

- Support was added for Firefox versions 26 and 27.

Web

- Support for the AJAX toolkit ZK was updated for ZK version 7.

Web

- Synchronization with AJAX HTTP requests was improved for the toolkits ExtJS, ICEfaces, PrimeFaces and ZK.

Web

- On Windows the one-time installation of the QF-Test Javascript bridge for Firefox no longer requires starting QF-Test once with administrator privileges.

Bugs fixed:

- WebStart applications are still a moving target. Various fixes for special cases were implemented to make startup of WebStart applications more reliable.
- Mouse events with variables for the X or Y coordinate that expand to empty values were not replayed to the middle of the target but to the relative coordinate 0.
- In case of deep component hierarchies QF-Test no longer logs excessive component information which, in extreme cases, could cause the SUT to run out of memory.
- A memory and thread leak in the `DaemonLocator` class has been plugged.
- The JNA library included in Jython no longer triggers stackguard warnings.
- An exception caused by a special kind of HTML table with no columns in the header has been fixed.

Web

B.19 Version 3.5.3

New features:

Web

- Support was added for the AJAX toolkit ICEfaces. See [section 38.2^{\(657\)}](#) for further information and a list of the supported AJAX toolkits and their versions.

Web

- QF-Test can now automatically detect whether a web page is based on one of the supported AJAX toolkits and installs the respective resolver.

Web

- Support was added for Firefox versions 24 and 25.
- The QF-Test demo video on the welcome dialog now links to the updated video on the QFS home page which is fully redesigned, including spoken explanations.
- It is now possible to copy and paste table cells back and forth between QF-Test tables and Excel or OpenOffice spreadsheets.
- When changing the name of a parameter in callable nodes like a Procedure QF-Test now offers to automatically update the variable name in all calling nodes.
- The standard library `qfs.qft` contains several new procedures for working with standard Swing dialogs based on the `JOptionPane` class in the package `qfs.swing.optionpane`.

Bugs fixed:

- Stability of startup of WebStart based clients and compatibility with older WebStart versions was further improved.
- Canceling a long search did not work reliably and could even lead to a deadlock in some cases.
- Replace operations for the regular expression `".*"` could lead to double replacements in some cases.
- The associated label for components was sometimes not determined correctly if the label was placed to the right of the component.

B.20 Version 3.5.2

Important changes:

- Handling of the security dialogs for Java WebStart and applets had to be changed again.
Security warnings are now closed implicitly before connecting to QF-Test and it is possible to permanently accept a certificate during this step. The details for this operation can be configured via the procedure `qfs.swing.startup.setSecurityDialogSpec`.
The default delays for WebStart and applets also changed as documented in the procedure `qfs.swing.startup.setDelays`.

New features:

SWT

- Eclipse/SWT support was updated for the final Eclipse version 4.3 "Kepler".

Web

- Support was added for Firefox version 23 and updated for version 22.
- Testlink support was updated for Testlink version 1.9.7.
- Wait for component to appear nodes now automatically handle waiting for the respective GUI engine in addition to implicitly waiting for the SUT to connect.
- Nodes for Procedures⁽³⁷⁸⁾ and Packages⁽³⁸⁵⁾ marked as "@deprecated" are now crossed out in the tree view.

Web

- The new procedure `qfs.web.general.integrateFirebug` in the standard library `qfs.qft` loads Firebug Lite into the running browser which can be very helpful when writing resolvers.
- The keyboard shortcut **Shift-Control-I** can now be used to quickly pack one or more steps into an If⁽³⁹⁶⁾ sequence.

Bugs fixed:

- The Windows executables `qftest.exe` and `qftestc.exe` failed to read the configured Java version, memory and language from the Windows registry.
- Saving test-suites and run-logs and other IO operations are now significantly faster on slow network shares.
- When changing the currently executed node via 'Continue execution from here' in the test debugger, QF-Test is now smarter in keeping as much of the current state of execution as possible, including the previously broken case when continuing from a Procedure⁽³⁷⁸⁾ node.
- On Linux systems Firefox would crash when terminated while a dialog was still showing.

Web

Web

- The recently added support for ZK and PrimeFaces was fine-tuned and fixes subtle problems with specific Trees, TreeTables and other components.

Swing

- Associated labels of Swing components, recorded as "qfs:label" extra feature, are now used for generating the QF-Test ID⁽⁵⁴⁶⁾ attribute. For web and SWT components this was already the case.
- In the QF-Test daemon `DaemonTestRunListener` instances were not unregistered correctly which could slow down daemon execution after a call like `qftest -batch -calldaemon -verbose ...`
- The SUT can now implement an `RMIClassLoaderSpi` without interference from QF-Test's RMI communication.

B.21 Version 3.5.1

Important changes:

- The Jython version provided with QF-Test was rolled back to Jython 2.5.1 because Jython 2.5.2 caused strange but reproducible problems with some SUTs. Jython is mainly used as glue code, so stability and non-interference with the SUT are far more important than compatibility with newer Python language features and modules, so the Jython version for QF-Test will remain frozen at 2.5.1 for the foreseeable future.

New features:

Web

- New support was added for the AJAX toolkits PrimeFaces and ZK and support for Eclipse RAP was updated to RAP version 2.0. See section 38.2⁽⁶⁵⁷⁾ for further information and a list of the supported AJAX toolkits and their versions.

SWT

- Eclipse/SWT support was added for Eclipse version 4.3 "Kepler".

Web

- Support was added for Firefox versions 21 and 22.
- The new procedure `qfs.swing.filechooser.selectFiles` can be used to select several files in a multi-select enabled Swing `JFileChooser`.
- The new Browser HTTP request⁽⁵³²⁾ node can be used to replay HTTP GET or POST request directly in the running browser. In contrast to the existing Server HTTP request⁽⁵²⁸⁾ node, which interacts directly with the server, the Browser HTTP request can be combined with event and check nodes.

- HTML reports now contain pie charts to visualize the number of successful, failed and skipped tests. A click on the chart brings up a more detailed graph. The directory structure of reports has been cleaned up by moving all images and screenshots to sub-directories.
- When started from QF-Test Firefox now automatically accepts SSL certificates for HTTPS connections, limited to the current session.
- Testlink support was updated for Testlink version 1.9.4.

Bugs fixed:

- The initial connection sequence for applets and WebStart based applications had to be updated after security restraints were again tightened in recent JRE updates from Oracle. The stability of applet and WebStart startup should now be less dependent on timing and the procedure `qfs.swing.dialogs.handleSecurityWarning` is now able to handle all current security warnings for applets and WebStart.
- Exceptions caused by CSV or Excel files with inconsistent data are now handled cleanly.
- Updating components with Item⁽⁵⁵¹⁾ nodes could cause an exception.

B.22 Main new features in version 3.5

Following is an overview over the main new features that have been implemented for QF-Test since the last production version 3.4.

Description	Further info
Grouping and managing test-suites in projects	chapter 10 ⁽⁸⁶⁾
Support for AJAX toolkits ICEfaces, PrimeFaces, Vaadin and ZK	section 38.2 ⁽⁶⁵⁷⁾
Better abstraction with generic classes for web pages	section 5.4 ⁽⁴¹⁾
Full sub-item support for several AJAX toolkits	section 38.2 ⁽⁶⁵⁷⁾
Secure daemon communication via SSL	section 16.2 ⁽¹⁴⁴⁾ , chapter 40 ⁽⁷⁴¹⁾
Improved component recognition without names thanks to associated label	Attribute Extra feature ⁽⁵⁴⁷⁾
Integration with test management tool QMetry	section 19.3 ⁽¹⁷⁸⁾
Several extensions to the standard library <code>qfs.qft</code>	Standard library
Groovy updated to version 2.1.0	

Table B.2: New features in QF-Test version 3.5

B.23 Version 3.5.0

New features:

- Support was added for Firefox version 20.
- Eclipse/SWT support was updated for Eclipse versions 4.2.2 and 3.8.2.
- Groovy was updated to version 2.1.0.
- Explicit references between test-suites belonging to the same project are now resolved automatically provided that those test-suites have been saved with QF-Test version 3.5.0 or higher.
- Multi-line values can now be edited directly in QF-Test's table cells.
- Extra mouse buttons can now be used to move backwards/forwards between recently selected nodes.
- Double-clicking on nodes that reference other nodes now navigates to the target, e.g. from [Procedure call](#)⁽³⁸¹⁾ to [Procedure](#)⁽³⁷⁸⁾.
- Test-suite and run-log files can now be dragged onto QF-Test's main window to open them.

Web

SWT

- The new procedure `qfs.swing.dialogs.handleSecurityWarning` in the standard library `qfs.qft` closes the security dialogs displayed by current JREs during startup of applets or WebStart based applications.
- The procedure `qfs.web.browser.general.startAsNewProcess` was updated to include security dialog handling.
- Another new procedure named `qfs.swing.startup.startWebstartSUT` can be used to run WebStart and connect to the SUT, also handling the security dialog automatically.
- The new procedure `qfs.utils.files.readExcelFile` can be used to read data from an excel sheet into Jython or QF-Test variables.
- **Web** The `Server HTTP request`⁽⁵²⁸⁾ node now offers an optional timeout and can return the HTTP status code. Parameters are now encoded during replay instead of recording, making it easier to specify values manually. Older test-suites are converted automatically during loading to reflect this change.
- **SWT** The default names recorded for GEF items are now more meaningful and result in improved recognition.
- QF-Test now integrates with the test management tool Klaros. See [section 19.4](#)⁽¹⁸¹⁾ for further information.
- The error list for a run-log now includes a timestamp column and shows information about expected errors.
- When pausing a batch test via `[Alt-F12]`, input to the SUT is no longer blocked.
- If the new command line argument `-dontkillprocesses`⁽⁵⁷⁵⁾ is specified in batch mode, QF-Test will not explicitly terminate processes started as part of the tests.

Bugs fixed:

- A memory leak in case of large projects with many test-suites and complex includes has been plugged.
- A workaround has been implemented for a bug in recent JRE security updates that could cause batch tests to simply stop in the middle of execution.
- Variables could accidentally be removed from QF-Test component IDs when fixing references automatically.
- In special cases externalized run-logs could be missing from run-logs saved as .qzp.

- When opening an externalized log on its own, split logs contained therein could not be accessed correctly.
 - A race condition during automatic project refresh could mess up the project tree view.
 - The native file chooser for Windows is now implemented based on SWT instead of XFileDialog. This fixes issues when opening directories and when changing the file filter during "Save as..." operations.
 - In batch execution of multiple test-suites, global variables and resources are no longer cleared before the execution of each test-suite unless the new command line argument `-clearglobals(574)` is specified.
- Web**
- Firefox on Linux no longer crashes on shutdown.
- Web**
- For some MIME types downloading files did not work correctly with Firefox.
- Web**
- Checks for the enabled state could fail in Internet Explorer if the state was set via Javascript.
- Web**
- Waiting for the absence of an element could fail if the element just became invisible instead of being removed completely.
 - Variables in Data binder nodes are now expanded immediately during execution of the containing `Data driver(357)` node so that local variables bound in the Data driver are evaluated correctly.
 - A `Return(384)` in a `Finally(411)` node could hide an exception thrown in the containing `Try(405)` node.
 - Using Java 7 on Mac systems could cause an exception during startup of QF-Test.
 - In case a process simply refuses to be killed, the `Stop client(441)` node did not remove the process from the `Clients` menu, thus preventing starting a new process under the same name.
 - Image checks based on algorithms were not always shown correctly in the report.
 - In case dependency resolution fails due to an exception during setup even after unwinding the complete dependency stack for error escalation (see [section 12.3^{\(95\)}](#)), QF-Test now cleans up after the final attempt so that the next test-case starts with a clean slate.
 - Changes to the location of the search dialog and other dialogs were often ignored instead of automatically saved to the user preferences.

B.24 Version 3.5M3

New features:

Web

- Support was updated for Firefox versions 18 and 19.
- The project concept now includes support for sub-projects. See [chapter 10^{\(86\)}](#) for further information.

Web

- Javascript error filters defined in the option [Errors that should be ignored^{\(297\)}](#) are now applied to the complete message including the URL and line number of the source of the error.

SWT

- Several checks were added for Eclipse JFace components like `FormText`, `Link` or `Hyperlink`.

Bugs fixed:

- The responsiveness of QF-Test severely suffered when opening many test-suites.
- Refresh of large projects on slow filesystems like Windows network shares is now significantly faster.
- If several projects were opened, automatic refresh of the project hierarchy could mess up the project tree.

SWT

- Geometry of events relative to an SWT Shell now includes the window title and borders.

SWT

- Highlighting widgets inside `CTabFolders` in Eclipse 4.2 now uses the correct offset.
- Errors happening outside a test-case were not listed correctly in the report if the containing test-set contained several test-cases.

Web

- Downloading files with Firefox could cause a Javascript error if the file `mimeType.rdf` was missing from QF-Test's Mozilla profile. This file is now created on the fly if necessary.

Web

- Closing the browser window of Internet Explorer 10 could cause the browser to deadlock instead of terminate.

Web

- QF-Test's embedding of Internet Explorer 10 caused the browser to run in IE9 mode. Now it starts up in IE10 mode by default with the usual settings for compatibility mode enabled.

Web

- Replaying [Tab] keystrokes on Firefox did not always trigger the correct focus traversal.
- Conflicts caused by an SUT using a Groovy 2 version other than the version provided with QF-Test are now resolved.

B.25 Version 3.5M2

New features:

SWT

- Eclipse/SWT support was updated for Eclipse versions 4.2.1 and 3.8.1.
- Daemon communication is now secured via SSL. See [chapter 40^{\(741\)}](#) about how to increase security by creating an individual keystore or how to disable this feature in case you need daemon interaction with older QF-Test versions.
- QF-Test generated JUnit reports now contain `<skipped>` tags, enabling build tools like Hudson/Jenkins to correctly report skipped tests.

Bugs fixed:

- License usage for idle QF-Test daemons was not determined correctly.

Swing

- Replaying certain events in batch mode could switch QF-Test into recording mode, causing weird problems and significant loss of performance.

Swing

- Timing for startup of Java WebStart based SUTs has been further improved to significantly increase the stability of this setup.

Web

- Some focus transitions were not triggered correctly when replaying events in a browser.

Web

- The filename shown in a download dialog was not always correct.

Web

- Downloads in a Vaadin based SUT could cause an exception.

Web

- Internet Explorer 10 now reports the correct version information.

SWT

- Recording of drag and drop operations on CTabFolders in Eclipse 4.2 was broken.

SWT

- In some cases the height of SWT List items was not calculated correctly, leading to the wrong item being addressed in case of large lists with many items.
- The new Extra feature attribute `qfs:label` was sometimes not determined correctly in the presence of Swing JSplitPane or SWT SashForm components.

- Handling of an SUT with combined SWT/AWT GUI was broken. Due to an error introduced in version 3.4.11 QF-Test was treating this as two SUT clients instead of one client with two GUI engines.

B.26 Version 3.5M1

New features:

- With the introduction of projects (see chapter 10⁽⁸⁶⁾), large sets of test-suites become much more manageable and updating reverse includes is now mostly a thing of the past.
- The increasingly popular AJAX toolkit Vaadin is now fully supported.
- The integration of QF-Test with various AJAX toolkits has been brought to a new level. Generic classes provide a unified view that abstracts from the technical details, focusing on typical UI elements like Tree or Table that users are familiar with. This even includes the concept of sub-items so that addressing table cells or tree nodes is now just as easy as with Swing or SWT. This level of support is currently available for Ext JS 4, RAP, RichFaces and Vaadin.
- Support was added for Firefox version 18.
- For SUTs where only few components are named - if any - component recognition has been improved by adding a new Extra feature⁽⁵⁴⁷⁾ called `qfs:label`. It represents a best match for a label, based on various algorithms including a search for the nearest label for components that don't have one of their own. Classical forms consisting of text fields with labels next to them benefit most from it.
- Handling the opposite case, where the same component name is assigned to more than one component, is now also handled via the new option Validate component recognition during recording⁽²⁶³⁾.
- QF-Test now integrates with the test management tool QMetry. See section 19.3⁽¹⁷⁸⁾ for further information.
- Components in QF-Test's detail view are now sized dynamically depending on the contained data. Combined with improved scrolling this vastly improves creating and maintaining test-suites.
- The menu item Replay→Reset all can be used to stop the test, reset dependencies and global variables and kill all clients with a single action.

Web

Web

Web

- On Windows QF-Test now uses the native file chooser by default, configurable via the option Use native file chooser on Windows systems⁽²⁴¹⁾.
- There are new operations to search for various kinds of duplicates, empty tests or illegal characters. The result dialogs for these and similar operations as well as searches have more options and exported results can now be read back.
- Search and replace operations now cover whole projects and can be interrupted in case they take too long.
- Tables holding variable definitions or parameter lists can now be sorted.
- It is now possible to navigate from a QF-Test component ID specified in a variable table directly to the component by right-clicking the respective table cell.
- Parameters of procedures can now be added to or removed from the variable definitions of all callers via the context menu of the parameter table.
- Recording procedures for windows with many components now works significantly faster.
- Whitespace at the beginning or end of variable names is now generally removed to avoid confusion.
- There is a new hotkey (configurable via Hotkey for recording⁽²⁵⁴⁾, default **F11**) to start and stop recording directly in the SUT. The previous default binding for **F11**, component recording, has been moved to **Shift-F11**.
- There's a new kind of check for tree nodes that can handle whole (sub-)hierarchies at once, including indentation.
- Highlighting components in the SUT for event nodes now works even if the Component node for the event is not located in the same test-suite.
- Killing the SUT and other processes now works much faster.
- Leading and trailing whitespace in Feature⁽⁵⁴⁷⁾ attributes is now generally ignored. This implementation is fully backwards compatible, so it doesn't hurt if features were recorded with whitespace before.
- Line numbers are now displayed for tables where this is useful, for example in Data table⁽³⁶⁰⁾ or Check items⁽⁴⁸⁰⁾ nodes.
- In places where accidentally inserted TAB or line-break characters can do harm, most notably in variable definitions or checks, they are now represented as graphical symbols. This can be turned off via the option Show symbols for tabulator and linebreaks⁽²⁴⁵⁾.

New packages and procedures in the standard library `qfs.qft`

- The new `qfs.autoscreen` package holds procedures for locating and interacting with components on the screen based on some previously captured image.

- Several new utility procedures:

```
qfs.utils.files.compareFiles
qfs.utils.variables.deleteGlobalVariablesExcept
qfs.utils.variables.deletePropertyGroup
qfs.utils.variables.deleteOnePropertyVariable
qfs.utils.variables.listAllMembersOfPropertyGroup
qfs.utils.xml.loadPropertiesFromXML
qfs.utils.sleep
```

Swing

- Various new procedures for interacting with Swing components:

```
qfs.swing.checkbox.getValue
qfs.swing.combobox.indexOfItem
qfs.swing.combobox.getValue
qfs.swing.combobox.getValueOfItem
qfs.swing.general.doSelect
qfs.swing.menu.selectMenuItemInHierarchy
qfs.swing.menu.selectCheckMenuItemInHierarchy
qfs.swing.radiobutton
qfs.swing.radiobutton.getValue
qfs.swing.radiobutton.select
qfs.swing.tabbedpane.indexOfTab
qfs.swing.tabbedpane.getValueOfTab
qfs.swing.text.getText
```

SWT

- Various new procedures for interacting with SWT components:

```
qfs.swt.checkbox.getValue
qfs.swt.combo.indexOfItem
qfs.swt.combo.getValue
qfs.swt.combo.getValueOfItem
qfs.swt.general.doselect
qfs.swt.list.indexOfItem
qfs.swt.menu.selectMenuItemInHierarchy
qfs.swt.menu.selectCheckMenuItemInHierarchy
qfs.swt.radiobutton
qfs.swt.radiobutton.getValue
qfs.swt.radiobutton.select
qfs.swt.table.indexOfColumn
qfs.swt.table.getRowIndexofCell
qfs.swt.tabfolder.indexOfTab
```

```

qfs.swt.tabfolder.getValueOfTab
qfs.swt.text.getText
qfs.swt.tree.multicolumn
qfs.swt.tree.multicolumn.getRowIndexByCellText
qfs.swt.tree.multicolumn.getIndexOfNode

```

Web

- Various new procedures for interacting with web components:

```

qfs.web.checkbox.getValue
qfs.web.radiobutton
qfs.web.radiobutton.getValue
qfs.web.radiobutton.select
qfs.web.select
qfs.web.select.getItemCount
qfs.web.select.indexOfItem
qfs.web.select.getValueOfItem
qfs.web.select.getValue
qfs.web.select.selectValue
qfs.web.table
qfs.web.table.checks
qfs.web.table.checks.checkNumberOfRows
qfs.web.table.checks.checkNumberOfColumns
qfs.web.table.recording
qfs.web.table.recording.activateItemRecording
qfs.web.table.recording.deactivateItemRecording
qfs.web.table.getColumnCount
qfs.web.table.indexOfColumn
qfs.web.table.getNameOfColumn
qfs.web.table.getRowCount
qfs.web.table.getRowIndexOfCell
qfs.web.table.getValueOfCell
qfs.web.table.selectCell
qfs.web.text.getText

```

Web

- More web utilities, specifically `qfs.web.ajax.uninstallToolkitResolver`, the new package `qfs.web.checks.style` with several procedures for checking CSS style attributes of DOM nodes and a complete rewrite of `qfs.web.browser.general.startAsNewProcess`.

Bugs fixed:

- The modified connection mechanism still crashed Java WebStart on some occasions and has been further fine-tuned to prevent that.
- The option settings for Errors that should be ignored⁽²⁹⁷⁾ were not saved to the system configuration file.

B.27 Version 3.4.11

New features:

- QF-Test is now compatible with Java applets running on JRE version 1.6.0_35 or higher or JRE version 1.7.0_7 or higher. A security fix from Oracle broke the existing applet connection mechanism. The new mechanism should also improve stability of Java WebStart based SUTs.
- Support was added for Firefox version 17.
- Via the new option Errors that should be ignored⁽²⁹⁷⁾ it is now possible to ignore specific Javascript error messages while still treating others as an error.
- When copying multiple lines of a table in the detail view of QF-Test these are now also copied to the system clipboard as a list of TAB-separated values, thus enabling pasting into a spreadsheet.
- The new procedure `qfs.utils.logNativeMemory` in the standard library `qfs.qft` can be used to get information about the native memory use of an SUT client.
- Windows executables and installers created by QFS are now signed with an official certificate for smoother integration with Windows 7 and Windows 8.

Bugs fixed:

- Downloads of large files were sometimes truncated.
- The new INPUT types in HTML 5 are now mapped to INPUT:TEXT so that tests remain independent of whether the browser supports them.
- The command line argument `-runlogdir <directory>`⁽⁵⁸²⁾ now also works for daemon calls via `qftest -batch -calldaemon`.
- Empty formatted cells in an Excel 2010 sheet are now handled correctly by the Excel data file⁽³⁶⁸⁾ node.
- Toolbar icons for QF-Test's browser wrapper are now transparent and no longer look horrible on Windows 7.

B.28 Version 3.4.10

New features:

Web

- Support was added for Firefox versions 15 and 16.
- The performance of complex QF-Test operations like finding references has been significantly improved for large projects on slow filesystems like network shares.

Bugs fixed:

Web

- Crashes of IE and Firefox were further reduced.

B.29 Version 3.4.9

New features:

SWT

- Eclipse/SWT support was added for Eclipse "Juno" versions 4.2 and 3.8.

Web

- Support was added for Firefox version 14.
- Several new transformation options for nodes in the test-suite were introduced.

Bugs fixed:

Web

- A significant memory leak in QF-Test's browser wrapper was plugged.

Web

- Several causes for browser crashes were fixed, both for Internet Explorer and Firefox.

Web

- Component recognition now also works correctly with components recorded in a flat hierarchy, even if the web page contains FRAME nodes.

Web

- The progress bar in Internet Explorer always remained visible even when a page was completely loaded.

Web

- If the path for the Firefox installation directory contained special characters, Firefox could not be started correctly from QF-Test.

SWT

- On Windows, popup menus in an Eclipse/SWT application were sometimes mistakenly considered to be invisible.
- Files are no longer overwritten without asking when an image is saved from a Check image⁽⁴⁸⁸⁾ node.

- Component group⁽⁵⁵⁵⁾ and Window group⁽⁵⁵⁴⁾ nodes are now handled and integrated correctly when importing components from a test-suite.
- Another special case for escaping '\$' characters in component names has been fixed.

B.30 Version 3.4.8

New features:

Web

- Support was added for the current Firefox version 13.

SWT

- Eclipse/SWT support was added for Eclipse versions 3.8M7 through 3.8RC2 and experimental support for Eclipse version 4.2M7 through 4.2RC2.

Web

- Support for the Richfaces toolkit was improved.
- The Excel data file⁽³⁶⁸⁾ node now also handles Excel files with the '.xslm' extension.

Bugs fixed:

- Escaping of '\$' characters in component names was broken in the component and procedure recorder.

Web

- Enabled checks did not work correctly with Internet Explorer 9 running in IE 7 compatibility mode.

Web

- Firefox could crash when closing a browser window.

Web

- When recording procedures the calls for SELECT elements were created incorrectly.

B.31 Version 3.4.7

New features:

Web

- Support was added for the current Firefox version 12.
- The new Procedure `qfs.swt.ctabfolder.openViewMenu` can be used to generically open the ViewMenu of a CTabFolder based on the tab name, both for Eclipse 3.x and Eclipse 4.

Bugs fixed:

- Script errors were not formatted correctly when shown in the error dialog. In the run-log those errors were shown correctly.
- Excel 2010 files used in a data driver with an Excel data file⁽³⁶⁸⁾ node were not closed correctly.
- There's a new workaround for the workaround for the missing MSVC manifest in a DLL from Firefox 10 and higher. Problems with another missing MSVC DLL on Windows XP and deadlocks when loading a Flash component in Firefox on Windows have been fixed.

B.32 Version 3.4.6

New features:

- Eclipse/SWT support was added for Eclipse versions 3.7.2 and 3.8M6 and experimental support for Eclipse version 4.2M6.
- The Procedures `qfs.swt.ctabfolder.closeTab` and `qfs.swt.ctabfolder.checkTabIsDirty` now also support Eclipse 4.

Bugs fixed:

- The fix for SUTs that use RMI with a custom implementation of the Service Provider Interface (SPI) could cause a subtle class loading problem that has been fixed.
- Deleting the last row in a variable table in QF-Test could lead to an exception when subsequently trying to move a table row.
- QF-Test's handling of Javascript errors could cause Internet Explorer to crash.
- If QF-Test was accidentally started with the `bin` directory of an older QF-Test version as its working directory, starting a browser on Windows could fail due to conflicts between DLL versions.
- Use of a special API in Internet Explorer can lead to very subtle deviations in Javascript execution. QF-Test no longer uses this interface.
- On some systems the workaround for the missing MSVC manifest in a DLL from Firefox 10 and higher could cause a problem with another DLL and prevent QF-Test from starting Firefox.

B.33 Version 3.4.4

New features:

Web

- Support was added for current versions of Firefox 9 and 10.

SWT

- Eclipse/SWT support was added for Eclipse version 3.8M5 and experimental support for Eclipse version 4.2M5.

SWT

- New checks were introduced for SWT `Decorations`, small icons typically shown next to a text field. You can now check the visibility, icon and tool tip of a `Decoration`.
- Iteration ranges can now also be used when setting up data at script level via the `databinder` module or by calling one of the Procedures `qfs.databinder.bindList` or `qfs.databinder.bindSet` in the standard library `qfs.qft`. See [section 14.4^{\(126\)}](#) for further information.
- The thumbnail images shown for screenshots in the HTML report are now of higher quality.

Bugs fixed:

- A small change in QF-Test version 3.4.3 regarding the RMI communication between QF-Test and the SUT was undone because of possible side-effects when the SUT was using RMI itself.
- When using `Check image(488)` nodes with an advanced algorithm and a timeout QF-Test always used up the full timeout even when the image matched immediately.
- When a Check node with a timeout fails, QF-Test used to show the initial state of the target component in the run-log. Now it shows the state at the end of the timeout period instead.
- When using Drag&Drop to move nodes within QF-Test the tree could jump erratically in some cases.
- Trying to remove the same node twice from a search result dialog could lead to an exception.
- In the standard test-suite library `qfs.qft` the Procedure `qfs.shellutils.touch` now correctly updates the timestamp for existing files and the Procedure `qfs.autowin.getURLFromIE` now also works with Internet Explorer version 6.

- SWT**
 - When recording a mouse click on an SWT ToolItem that opens a drop down menu QF-Test now records the coordinates so the event can replay correctly.
- Web**
 - If an alert dialog was shown after closing a modal browser window, the main browser window could remain blocked after closing the alert.
- Web**
 - Javascript errors in Internet Explorer were not logged correctly in QF-Test's run-log.
- Web**
 - In some cases the initial loading of a web page did not succeed.
- Web**
 - Highlighting the selected component in a browser on Linux has been improved.
- Web**
 - Web pages with a Flash element were sometimes not displayed correctly on Linux.
- Web**
 - The name recorded for file upload on Linux contained the Windows file separator character '\'
- Web**
 - There were problems with browser startup on special Windows versions if QF-Test was installed below a directory with blanks in its name, e.g. "C:\Program Files (x86)"

B.34 Version 3.4.3

New features:

- Web**
 - Support was added for current versions of Firefox 8 and 9.
 - The Excel data file⁽³⁶⁸⁾ node now supports reading .xlsx files in Excel 2010 format.
 - The performance of some advanced image check algorithms was significantly improved.
 - It is now possible to fetch option values in a script via the new method `rc.getOption`. See section 37.6⁽⁶²²⁾ for details.

Bugs fixed:

- When switching test-suites by clicking a suite tab the keyboard focus was not correctly restored.
- The keyboard shortcut **Ctrl-D** for setting a bookmark did not work.
- Jumping from a node in the debugger window to the same node in the test-suite now works again.
- In some special cases QF-Test could kill the wrong process on Windows when trying to terminate the SUT.

B.35 Version 3.4.2

New features:

Web

- Full support was added for Firefox 6 and experimental support for Firefox 7.
- For easier integration with continuous integration tools like Hudson/Jenkins or TeamCity the exit code of a batch run can be forced to 0 even in case of warnings, errors or exceptions with the help of the command line arguments `-exitcodeignorewarning(576)`, `-exitcodeignoreerror(576)` and `-exitcodeignoreexception(575)`.

Web

- If the option `Replace no-break space with normal space` is activated, QF-Test now represents no-break space characters in a web page as normal space characters in checks and `Component(545)` node attributes to avoid confusion.

SWT

- QF-Test now supports capture and replay of "expand" and "collapse" events for SWT ExpandItems.

Bugs fixed:

SWT

- The integrated help viewer of Eclipse uses a separate SWT Display on Windows which caused problems during replay. It is now consistently represented with `GUI engine(540)` "swt1".

Web

- Direct execution of Javascript code now also works in Internet Explorer 9 despite stronger security restrictions.

Web

- The AJAX resolver for GWT has been updated to version 1.4.0 to fix filtering of DIV nodes from the component hierarchy. See `section 38.2(657)` for details about AJAX resolvers and their versioning scheme.

Web

- Opening a modal popup window could crash Firefox 6 on Windows.

Web

- Double clicks are now recorded correctly in Internet Explorer.

Web

- A `Fetch text(499)` node for a SELECT node now retrieves the text of the selected OPTION node and Fetch text nodes for OPTION elements represented as items are working correctly.
- Printing unicode strings in Jython could cause problems when running in batch mode.
- The content of `Data table(360)` nodes was missing in the run-log.

- A Wait for client to connect⁽⁴³⁸⁾ node with a very short timeout did not throw a ClientNotConnectedException⁽⁵⁶²⁾ when single-stepping in the debugger.
- The QF-Test daemon was leaking threads when starting a remote SUT process for use with the procedure `qfs.daemon.startRemoteSUT`.
- When updating failed image checks from the run-log the size of the image was not always updated correctly.
- When recording procedures, QF-Test failed to create a procedure in case the respective component had no value for a placeholder like `<COMPFEATURE>`.

B.36 Version 3.4.1

New features:

Web

- Full support was added for Firefox 5 and experimental support for Firefox 6.

SWT

- Eclipse/SWT support was updated for Eclipse version 3.7 Indigo.
- Groovy was updated to version 1.7.10.
- Error messages for errors in Groovy scripts now show proper context and line numbers.

SWT

- QF-Test can now handle custom SWT components that hide their child Widgets by overriding `getChildren()` to return an empty array.

Web

- Support was added for ExtJS version 4 and for RichFaces version 4. See section 38.2⁽⁶⁵⁷⁾ for details about the support AJAX toolkits and versions.
- The As regexp⁽⁵⁴⁷⁾ attribute of Window⁽⁵³⁵⁾ and Component⁽⁵⁴⁵⁾ nodes can now be defined as a variable value.

Web

- The startup sequence created by the quickstart wizard for web applications now has a smarter layout and is easier to adjust.
- On Windows systems QF-Test now makes use of the always-on-top attribute to force SUT windows to the front even if normally prevented by Windows. To disable this feature, deactivate the option Force window to the top when raising⁽²⁸⁰⁾.
- A new check was added to directly check the icon of a component instead of having to capture the whole component and define a check region.

Bugs fixed:

- When loading the test-suites to restore the last session upon startup, QF-Test did not check for auto-save files.
- It was possible to start parallel tests in the same context by hitting `[Return]` at the right time. This could lead to confusing results in the run-log and even to exceptions.

Swing

- Swing applications started via Java WebStart have two independent application contexts, a user context and a security context. Unless the `GUI engine(420)` attribute of a `SUT script(419)` node explicitly defines the engine, the script is now always correctly executed in the user context and not in the security context.

Swing

- Special support for JSlider components was missing.
- The special variable `${qftest:runid}` did not expand to the correct value if the runid was specified explicitly via `-runid <ID>(582)` in batch mode.

Web

- A forced download triggered via a special PHP construct could cause Internet Explorer to crash.

Web

- Clearing the cache for Firefox did not always work correctly.

Web

- Trying to record a check for an XML document displayed in Internet Explorer could crash the browser.

Web

- It is now possible to record checks for applets even when they are embedded in QF-Test's browser wrapper.
- The Groovy implementation of `qf.logError` was broken.
- In a workbench view with many open test-suites and some test-suite tabs scrolled out to the left, coordinates for dragging test-suite or node to different test-suites were off, leading to surprising results.

SWT

- QF-Test was leaking resources when taking Screenshots in an Eclipse/SWT SUT.

SWT

- Calling `Display.asyncExec` from a timer could crash Eclipse on Linux Gtk with 64bit.

SWT

- The order of `SWT.Activation` events could show very subtle differences between manual replay and automatic replay with soft events.
- A very long search string could mess up the size of the search and replace dialog.
- Markers in run-logs did not propagate beyond 'External log' nodes.
- Fallback to lazy binding did not work for not-yet-bound properties.

B.37 Main new features in version 3.4

Note

Support for Java 1.4.2 was dropped by both Jython and Groovy so QF-Test now requires at least Java version 1.5 for itself and the SUT.

Following is an overview over the main new features that have been implemented for QF-Test since the last production version 3.2.

Description	Further info
Jython updated to version 2.5.1	section 13.6 ⁽¹¹³⁾
Groovy updated to version 1.7.6	
Support for Eclipse/SWT 3.6.2 and 3.7	
Web testing support for Firefox 4 and Internet Explorer 9	
Integration with Continuous Integration tool Jenkins/Hudson	section 20.3 ⁽¹⁹⁰⁾
Fault-tolerant image checks	chapter 44 ⁽⁷⁶⁹⁾
Advanced search and replace	section 2.3 ⁽¹⁵⁾
Iteration ranges for data drivers	Iteration ranges ⁽³⁶²⁾
Direct interaction with web servers	Server HTTP request ⁽⁵²⁸⁾ node
Extended import functionality	section 17.3 ⁽¹⁵³⁾
Debugger integrated into the workbench	
Ability to create a test-suite from a run-log	section 8.1.6 ⁽⁷³⁾
Automatic splitting of run-logs and graceful handling of OutOfMemoryErrors	Option Minimum size for automatic splitting (kB) ⁽³⁰³⁾ .
Improved backwards/forwards navigation	
Improved pkgdoc and testdoc documentation	Command line arguments -testdoc.teststeps ⁽⁵⁸⁵⁾ , -testdoc.nodeicons ⁽⁵⁸⁵⁾ and -pkgdoc.nodeicons ⁽⁵⁷⁹⁾
Improved coordinate abstraction for mouse events	Option Record MouseEvents without coordinates where possible ⁽²⁵⁶⁾ , Mouse event ⁽⁴⁴⁷⁾
Variables with default values and many additional special variables	section 7.5 ⁽⁶²⁾

Table B.3: New features in QF-Test version 3.4

B.38 Version 3.4.0

New features:

- Support for testing Eclipse/SWT version 3.7 was updated.

SWT

- All Data binder nodes now have an attribute named Iteration ranges⁽³⁶²⁾ that can be used to limit the iteration to a single element or one or more intervals, which is very convenient when implementing tests, continuing a test that had to be stopped or performing a retest with a limited subset.
- In workbench mode the current session with the currently open test-suites and the positions therein is now saved when QF-Test is closed and restored upon startup, provided the option Restore last session on startup⁽²⁴⁰⁾ is active. Also added to the workbench view were the ability to save all modified test-suites via the **File** menu and switching between recently active test-suites via **Control-F6**.
- Moving backwards/forwards between selected nodes has been unified and should now work consistently even for special cases. After recording components you can jump directly to the recorded node by moving forwards, even if no node was actually recorded and the existing node is in a different test-suite.
- Filter capabilities were added to the search/replace result and other multi result dialogs.
- Marks can now also be set for nodes in a run-log, so multi-step search operations are now also available in run-logs.
- The quickstart wizard has improved support for launching applets in a browser and for setting the compatibility mode in internet explorer, backed by the procedures `qfs.web.browser.general.startAsNewProcess` and `qfs.web.browser.settings.enableCompatibilityMode` in the standard library `qfs.qft`.
- When started in batch mode with the command line argument -verbose [<level>]⁽⁵⁸⁶⁾, QF-Test now displays exception, error and warning counts for each Test-case.
- The new package `qfs.autowin.acrobat` in the standard library `qfs.qft` contains methods to save a PDF document showing in an Acrobat window or to copy its text content to a file via the clipboard (Windows only). These are complemented by the `qfs.utils.files` package with procedures to verify the content of a text file.
- The new methods `setProperty` and `getProperty` in the generic scripting module `qf` (see section 37.7⁽⁶³⁹⁾) can be used to associate arbitrary data with a GUI element, e.g. to cache some resolved feature or name values or other information that is too expensive to compute every time a resolver method is called.
- The `TestRunListener` API (see section 39.5⁽⁷²⁸⁾) has been extended to provide the number of exceptions, errors and warnings that occurred below a given node in the `TestRunEvent` of a call to `nodeExited`.

Web

- The AJAX resolver for GWT has been updated to version 1.2.0. It now filters irrelevant DIV nodes from the component hierarchy and has improved support for obfuscated GWT class names. See [section 38.2^{\(657\)}](#) for details about AJAX resolvers and their versioning scheme.

Bugs fixed:

- Early checking to handle OutOfMemoryErrors more gracefully could impact test execution speed significantly with JDK 1.5 on Windows. Tests should now run at full speed again with all JDK versions.
- Some of the advanced image check algorithms did not work correctly with JDK 1.5. All algorithms are now fully supported for JDK 1.5 and higher and return consistent results, independent of the JDK version, except for resizing, which is implemented differently in the various JDK versions and can lead to minor deviations in the resulting probability.
- The procedure `qfs.shutil.copy` in the standard library `qfs.qft` was broken. It now copies files and directories correctly, recursively if necessary and handles special cases like copying a directory into itself.

Web

- [Check image^{\(488\)}](#) nodes now work correctly in Firefox 4.0.

Web

- Scrolling an element visible was not always possible in IE9 and some special cases with a deeply nested hierarchy.

Web

- In IE9 nodes were not highlighted properly and highlighting could interfere with recording image checks.

Web

- Browser settings for IE9 did not always take effect.

Web

- A download operation triggered from a popup window could crash the browser.
- A running test in the QF-Test daemon can now be interrupted and stopped cleanly via `-stoprun(584)` even if the test is currently waiting for something.
- The structure of the JUnit XML report is now compatible with Atlassian Confluence.
- In Groovy the method `rc.getProperties()` was not implemented correctly.

B.39 Version 3.4M2

New features:

SWT

- Support was added for testing Eclipse/SWT versions 3.6.2 and 3.7 Milestone 5.

- The search and replace dialogs are now integrated so it is possible to start with a search and continue replacing, based on the same criteria. Also, the replace dialog is no longer modal so it can be kept open while editing.
- Integration of the debugger functionality into the workbench view is now complete. The current variable bindings are displayed automatically whenever the debugger is entered unless the option Show variable bindings automatically⁽²⁹⁸⁾ is turned off.
- The keyboard shortcuts `[Alt-1]` through `[Alt-9]` navigate directly to the respective test-suite tabs in the workbench view.
- Pressing `[F2]` in the test-suite tree now moves focus directly to the details field that is most relevant for the display name of the node.
- The new Procedure `qfs.client.renameClient` in the standard library `qfs.qft` can be used to rename an SUT client after it connects. This is especially useful for an SUT that launches multiple sub-processes that connect as `client:2`, `client:3` etc.
- The new special variable `$_{qftest:skipnode}` can be used to skip execution of a node. Please see section 7.5⁽⁶²⁾ for details and applicability.
- Hyperlinks in the Eclipse FormText widget are now recognized as sub-items.
- Advanced image checks are now also available to scripts via the new run-context method `rc.checkImageAdvanced`. Please see section 37.6⁽⁶²²⁾ for details.
- The various `rc.check*` methods in the run-context now return the result of the evaluated condition. Please see section 37.6⁽⁶²²⁾ for details.
- The image of a Check image node in the test-suite and the run-log as well as screenshot images can now be opened in a separate window by double-clicking the image (run-log only) or clicking the respective button above the image.
- There is a new utility module for scripting called `qf`, documented in section 37.7⁽⁶³⁹⁾. It contains variants of commonly useful methods like `isInstance` and the ability to add messages to the run-log even when no run-context is available, e.g. in `TestRunListener`, `Checker` or `Resolver` implementations.
- Templates for procedure recording have become more powerful. Procedures can now be created conditionally by placing a `@CONDITION` tag with an arbitrary Jython or Groovy script in the comment. Please see chapter 41⁽⁷⁵⁸⁾ for details.
- The Procedure `qfs.web.browser.settings.setProxy` in the standard library `qfs.qft` now also supports passing a URL for automatic proxy configuration.

Bugs fixed:

SWT

Web

- Switching execution to a different node in the debugger could unbalance the parent/child relationships of the associated nodes in the run-log.
- Setting and unsetting options at script level could mess up default option values.
- SWT • Improved double click replay for SWT on Windows.
- SWT • Some selection events for SWT now receive special treatment for focus handling.
- SWT • Empty lines at beginning of an SWT StyledText widget are now handled correctly on Linux.
- Errors in nested test-cases are now listed correctly in the report.
- Web • Proxy server settings for Internet Explorer 9 are now working correctly.
- Web • Image checks are now working correctly with Firefox 4.
- Web • Javascript and authentication dialogs are now working correctly in Firefox 4.
- QF-Test now issues a warning each time you first try to modify a write-protected test-suite.
- Doctags for Dependency nodes are now listed correctly in pkgdoc documentation.

B.40 Version 3.4M1

New features:

- Groovy was updated to version 1.7.6.
- QF-Test now actively monitors the memory available in the Java VM for running tests. When available memory runs low, the test is aborted cleanly and the run-log is saved. This also prevents extreme slow-down when memory gets tight.
- To prevent running out of memory in the first place, QF-Test now splits run-logs automatically by default, based on the option Minimum size for automatic splitting (kB)⁽³⁰³⁾.
- Managing test-suites in the workbench has been improved. The tab for the active suite is now shown with a bold font, you can drag tabs to re-order test-suites and double-clicking to the right of existing tabs creates a new suite.
- The search dialog now has additional options for limiting the search and it is also possible to search all loaded test-suites at once. To prevent visual overload for the common case, the search dialog can now be toggled between simple and advanced mode.

- The debugger has been mostly integrated into the workbench, except for the variables view which still requires the separate debugger window.
- There is a new recording mode for web applications that tells QF-Test to ignore individual events and record entire GET or POST requests instead, which are then represented by the new Server HTTP request⁽⁵²⁸⁾ node. Upon replay QF-Test interacts directly with the HTTP/HTTPS server without requiring a browser. This can be useful for load testing or mass data entry.
- For Swing and SWT components with an icon QF-Test now records a hash value derived from the icon image as an Extra feature. As long as the icon remains stable this should lead to improved recognition, especially for toolbar buttons without a label. The default state of the image hash extra feature is 'Should match'. You can change it to 'Ignore' for components where the icon is not reliable and to 'Must match' for components where the image is the best way to identify the component.
- There is a new special variable syntax that allows specifying a default value for a variable. This is extremely useful for things like generic components or really every place where a variable has a reasonable default, because that default value is then tightly coupled with the variable lookup itself and doesn't have to be specified at Sequence or test-suite level. The syntax is `${default:varname:defaultvalue}`.
- Additional special variables are now available to retrieve the name of the currently executing Test-set or Test-step in addition to the already available Test-case name. Also, counters were added for the number of test-cases executed so far, failures etc. Please see section 7.5⁽⁶²⁾ for details.
- Where available, the dialog showing the methods available for a component now also displays the results of calling those methods that represent Java Bean values.
- The option Log successful advanced image checks⁽³⁰⁷⁾ can be activated in order to store the results of a successful advanced image check in the run-log. This can be useful while tuning the parameters for an advanced image check algorithm.
- The new command line option `-verbose [<level>]`⁽⁵⁸⁶⁾ for batch execution with or without accessing a daemon causes QF-Test to print progress information during a test-run.
- The new Procedure `qfs.utils.dumpHeap` in the standard library `qfs.qft` can be used to trigger a heap dump for the Java VM of the SUT when it uses too much memory. This can be detected with the help of the Procedure `qfs.utils.logMemory` which now returns the amount of memory used.

Bugs fixed:

- When replaying mouse events without coordinates on items, QF-Test no longer targets the center of the item, but the position 5 pixels from the left, vertically centered. Apparently some events are sensitive to whether the click hits the text of an item and not just its bounding box, so short items in a broad column could cause problems.
- Calling stored procedures in a database could fail using the default Jython bindings.
- If the debugger was turned off for normal use, debugger-stepping after hitting a breakpoint could occasionally fail so the test simply continued.
- A positive search result for the type of an extra feature could cause an exception.
- Blank characters in names for saving external run-logs are now turned into an underscore because Windows doesn't handle trailing blanks in directory names gracefully.
- The input file for a CSV data file node is now closed immediately after reading the data.
- Recursive procedure calls could cause problems when importing procedures or analyzing references.

B.41 Version 3.3.0

Incompatible changes:

- Support for Java 1.4.2 had to be dropped in order to update Jython and Groovy. QF-Test now requires at least Java version 1.5.
- Interaction between Jython and Java has changed slightly between Jython 2.1 and 2.5 so occasionally scripts that worked in QF-Test may now fail or behave differently. We try to maintain compatibility where possible and provide a list of things to look out for in [section 13.6^{\(113\)}](#).

New features:

- Jython was updated to version 2.5.1. Most scripts should continue to run without problems. Incompatibilities may arise due to the changes in character encoding and because Java strings are now uniformly treated as unicode strings in Jython. Please see [section 13.6^{\(113\)}](#) for further information.
- Groovy was updated to version 1.7.2.

SWT

- Support was added for Eclipse/SWT version 3.6.1.

Web

- Experimental support was added for Firefox 4 and Internet Explorer 9.
- QF-Test now integrates well with the increasingly popular Continuous Integration system Hudson/Jenkins. This includes JUnit as an additional target format for QF-Test reports. Please see [section 20.3^{\(190\)}](#) for further information.
- Image checks are no longer restricted to testing for exact pixel-by-pixel equality. Several flexible algorithms now allow image checks with a configurable degree of deviation from the expected image. Please see [chapter 44^{\(769\)}](#) for details.
- Besides components you can now also import test-cases and procedures from one test-suite into another as described in [section 17.3^{\(153\)}](#).
- When coordinates for mouse events are irrelevant, QF-Test now removes them altogether instead of setting them to (1,1). Upon replay the event is then aimed at the center of the target component or item. See also option [Record MouseEvents without coordinates where possible^{\(256\)}](#).
- By combining search and replace operations with marks it is now possible to narrow searches in several steps, giving very fine-grained control over where to perform replacements. Instructions and examples are provided in [section 2.3.3^{\(22\)}](#).
- Moving backwards or forwards between selected nodes now works across test-suites and even run-logs. Thus a single mouse click or keystroke is now sufficient to return when following a procedure call to its target in a different test-suite or when jumping from a run-log to the respective node in the suite.
- Several new toolbar buttons were added and all the buttons now look much better. The menu structure also was cleaned up with operations on tree nodes now collected under the new [Operations](#) menu, among them many new node transformations. Also, several new keyboard shortcuts were introduced. A summary of all keyboard shortcuts is provided in [appendix C^{\(904\)}](#).
- The layout of pkgdoc and testdoc documentation now equals that of reports, including node icons and support for test-steps. These can be configured via the command line arguments [-testdoc.teststeps^{\(585\)}](#), [-testdoc.nodeicons^{\(585\)}](#) and [-pkgdoc.nodeicons^{\(579\)}](#) or the respective options in the dialog for interactive creation of those documents.
- 'Skipped' and 'Not implemented' tests can now be ignored in the report via the command line argument [-report.ignorestipped^{\(580\)}](#) or the respective option in the interactive dialog.
- Daemon tests can now be stopped at the command line via the new argument [-stoprun^{\(584\)}](#).

- The comment of a Procedure, Test-case or Dependency node can now be filled automatically with doctags for parameters, characteristic variables and author by right-clicking the node and selecting **Additional node transactions→Fill comment** from the context menu.
- The `TestRunListener` API now also provides information about whether a test-run was terminated normally (with or without exceptions) or aborted. See [section 39.5^{\(728\)}](#).
- The new method `rc.callTest` in the run context can be used to call a Test-case or Test-set. In contrast to `callProcedure`, `callTest` can only be called from Server script nodes. It returns the exit status of the called test.
- The content of QF-Test configuration files is now sorted which significantly improves handling of those files in a version control system.
- Several procedures were added to or refined in the standard library `qfs.qft`. Its documentation is now also available online and indexed by the full text search engine on the QFS website. A link to the online text search covering all technical material is provided via the menu **Help→Online search...**.
- It is now possible to create a test-suite from a run-log.
- It is now possible to pass GUI elements from an SUT script as individual targets to QF-Test's component recognition mechanism via the new run context method `overrideElement`. See also [section 13.3.11^{\(110\)}](#).

Bugs fixed:

Web

- When started from QF-Test, Internet Explorer 8 always ran in IE7 compatibility mode. This is no longer the case. Compatibility mode can now be actively selected in the quickstart wizard or via the procedure `qfs.web.browser.settings.doStartupSettings` in the standard library `qfs.qft`.

SWT

- On Linux QF-Test could sometimes crash SWT/Eclipse applications with disposed Combo popups.

SWT

- Test replay for Eclipse/SWT on Linux is now much more robust, especially when handling menus.

SWT

- Trailing empty newlines in `StyledText` widgets are now handled correctly on Linux.
- Creating reports with many screenshots now requires much less memory.

Web

- Downloads in Internet Explorer should now work consistently.

Web

- In case of overlapping nodes with transparent nodes on top QF-Test could pick the wrong node for check recording. It now lets the browser make the decision instead. Switching back to the old algorithm can be done via the option Let the browser determine the target element for check recording⁽²⁹⁶⁾.
- A memory leak in QF-Test resulting from SUT clients printing a lot of output to the shared terminal in workbench view was plugged.

B.42 Version 3.2.2

New features:

- Support was added for testing Eclipse/SWT version 3.6 "Helios".
- Multiple file selection in an SWT FileDialog can now be recorded and replayed.

SWT

Bugs fixed:

- Replay performance could suffer severely if the user's home directory was located on a network drive.
- A workaround for an AWT weakness introduced in QF-Test 3.2.1 could impact replay performance for AWT/Swing.
- Yet another performance problem resulting from very large hierarchies of Swing menus has also been fixed.
- Several special cases for uploading files with Internet Explorer are now handled correctly.
- Downloading files with Internet Explorer now works correctly in case of HTTP redirection.

Swing

Swing

Web

Web

B.43 Version 3.2.1

Bugs fixed:

- With a 1.4 JDK in interactive mode QF-Test would only start if given a file to open.
- Web-testing now works again on Windows 2000.

Web

- The QF-Test debugger used to show duplicate entries on the variable stack for Procedure call⁽³⁸¹⁾ and Test call⁽³²⁷⁾ nodes.
- There was a memory leak in connection with interactive editing of test-suites with Check image⁽⁴⁸⁸⁾ nodes for large images.
- In very special cases an import of components into their own suite could result in incorrectly updated references.
- Saving entries for zipped run-logs could occasionally fail on Windows due to a problem in the underlying library. QF-Test now works around that.
- Two problems were fixed for SUTs with more than one GUI engine, e.g. a WebStart application or combined SWT/Swing. In such a case QF-Test used to log more than one screenshot of the main screen after an error and sometimes connections from the second engine could fail.
- The upload dialog for web pages was not handled correctly for all browser versions and the procedure `qfs.web.input.fileUpload` in the standard library `qfs.qft` did not handle all cases correctly.
- In recent QF-Test versions there was an unofficial global Jython run-context `rc` available in addition to the local one. In special cases this could cause subtle namespace problems, so the (still unofficial) global version was renamed to `__rc`.
- Several minor problems and occasional exceptions in connection with the result list dialogs were fixed.

Web

B.44 New features in QF-Test version 3.2

Following is an overview over the main new features that have been implemented for QF-Test version 3.2. For detailed information about the differences between QF-Test 3.0 and 3.2, please see the following sections with the release notes for versions 3.2.0 and 3.1.x.

Description	Further info
Support for Web testing in combination with Java applets	Requires JDK 1.6.0_10 or higher
JDK instrumentation is no longer required	section 35.1 ⁽⁶⁰⁴⁾
Workbench view for a modern user interface	
Test-step ⁽³³⁶⁾ nodes for detailed reports	section 15.1 ⁽¹³⁰⁾
Refactoring support with automatic update of references	
Integration with the open source test management tool TestLink	section 19.6 ⁽¹⁸²⁾
New 'Best Practices' part in the manual	part II ⁽²⁰⁷⁾
Eclipse/SWT support updated for the latest versions including GEF	section 39.4 ⁽⁷²⁴⁾
Incremental search in all of QF-Test's text fields	section 2.3.1 ⁽¹⁹⁾
Result list dialogs for complex operations	
Component method spy	section 5.5 ⁽⁴⁴⁾
Options can be set at script level	section 13.3.9 ⁽¹⁰⁸⁾ and section 37.6 ⁽⁶²²⁾
Relative and test-suite-local procedures	section 9.3.2 ⁽⁸³⁾ and section 9.3.1 ⁽⁸³⁾
New-style Check nodes and support for custom Checkers	section 39.3 ⁽⁷¹⁵⁾
Static validation of references	
Automatic parameterization of procedures	section 9.3.4 ⁽⁸⁴⁾
Persistent bookmarks and transient marks	section 29.1.3 ⁽²⁴⁶⁾
New APIs: <code>ItemResolver</code> , <code>Checker</code> , <code>ClassNameResolver</code> and <code>TestRunListener</code>	section 39.2 ⁽⁷⁰³⁾ , section 39.3 ⁽⁷¹⁵⁾ , section 39.1.2 ⁽⁶⁸⁰⁾ and section 39.5 ⁽⁷²⁸⁾

Table B.4: New features in QF-Test version 3.2

B.45 Version 3.2.0

New features:

SWT

- Support for Eclipse/SWT has been brought up-to-date with the current versions 3.5.2 and 3.6M6.

Swing

- Swing components are no longer highlighted by exchanging their foreground and background color but by painting a rectangle, similar to SWT and Web. Target sub-items are now highlighted as well.
- The new [Boolean check](#)⁽⁴⁷⁶⁾ node with Check type identifier attribute 'visible' is similar to [Wait for component to appear](#)⁽⁵¹²⁾ node, but in contrast to the latter is listed in the report as a check and it has different default settings for the result handling.

- The most recently accessed component and possibly item in the SUT can now be retrieved in Jython and Groovy SUT script nodes via `rc.getLastComponent()` and `rc.getLastItem()`.
- Selecting a file for upload in a web form can now be recorded and replayed directly on Windows. For a cross-platform implementation it is still necessary to call the procedure `qfs.web.input.fileUpload` from the standard library `qfs.qft`.
- Additional features were added to the result lists for complex operations. For example, you can now perform actual replacements in the result list for the replace dialog or you can update multiple failed checks at once from the error list in a run-log. Besides, the result list for an operation that takes time is now filled continuously and the operation can be aborted if necessary.
- Recent search expressions are now saved in the user configuration.
- Dependency⁽³⁴⁴⁾ nodes inside Packages are now listed in the pkgdoc documentation unless disabled via the command line argument `-pkgdoc.dependencies(578)` or the respective interactive option.

Bugs fixed:

- Right-clicks for Javascript-based popup menus no longer need to be replayed as hard events in order to trigger the popup reliably.
- When recording checks the z-order of DOM nodes is now properly taken into account so that QF-Test no longer records nodes that are actually in the background, covered by other nodes.

B.46 Version 3.1.4

New features:

- It is no longer necessary to instrument standard JDKs or JREs, though it doesn't hurt to keep existing instrumentation. Please see the option Connect without JDK instrumentation (Swing)⁽²⁷⁵⁾ for further information.
- Support was added for the latest Eclipse/SWT version 3.6.
- QF-Test now supports Firefox 3.6.
- The new part II⁽²⁰⁷⁾ 'Best Practices' in the manual describes recommended procedures based on lessons learned from several customer projects and user feedback that will hopefully help you to find the best strategy for using QF-Test in your own projects.

- Reports have been restructured and further extended. Check nodes now have an optional Name attribute for their representation in the report. Besides, the performance of report creation has been drastically improved and its memory impact reduced.
- The tree display in run-logs now shows expanded values instead of variables. This can be toggled via the **View** menu.
- Compactification for run-logs is now more efficient and special placeholder log entries show where nodes were removed during compactification.
- Support for the Eclipse Graphical Editing Framework (GEF) is now included by default and works correctly with multiple concurrent GEF editors. See [section 39.4^{\(724\)}](#) for further information.
- The new procedure `qfs.web.input.fileUpload` in the standard library `qfs.qft` handles INPUT:FILE nodes for the various browser variants.
- TestLink integration has been simplified ([section 19.6^{\(182\)}](#)).
- For Swing and SWT tables and multi-column trees it is now possible to check a whole row at once.
- ExtraFeatureResolvers can now also be used for SWT and Swing. These are described in [section 39.1.7^{\(683\)}](#).
- The standard library `qfs.qft` now includes wrapper procedures for the autowin module in the package `qfs.autowin`. These can be used to check for the existence of a window and to close it on Windows systems.
- The runid of the current test-run now is now available via `${qftest:runid}`.
- Special variables or properties of type group:name can now be used for Characteristic variables.
- For nodes like Test-step or Test-set with several tabs for variable bindings the number of bound variables is now displayed in the tab which makes it much easier to locate those bindings.

Bugs fixed:

- Several causes for IE crashes were fixed.
- Wait for absence now works correctly for invisible nodes.
- QF-Test doesn't get blocked as easily when the SUT generates a lot of output and the JTextArea for the shared terminal cannot keep up.

- The internal variable stack of QF-Test could get messed up in some special cases.
- The procedure builder now correctly handles ' _ ' characters in package or class names.

B.47 Version 3.1.3

New features:

- The new Test-step⁽³³⁶⁾ node can be used to provide further structure and document tests below Test-cases⁽³¹⁴⁾. Test-steps are listed individually in the report unless deactivated with -report.teststeps⁽⁵⁸¹⁾. Similar effects can be achieved by marking arbitrary nodes with the doctag @teststep, optionally followed by a name for the step.
- The report now also can list checks (off by default) and warnings (excluding component recognition warnings, on by default) integrated into the test-step listing and also shows icons for nodes. The respective settings can be changed via -report.checks⁽⁵⁷⁹⁾, -report.warnings⁽⁵⁸¹⁾ and -report.nodeicons⁽⁵⁸⁰⁾ in batch mode or the respective options in the dialog for interactive report creation.
- In the workbench view there is now a context menu for the test-suite tabs.
- All Comment attributes can now be edited in an external editor similar to scripts.
- There are new toolbar buttons in the run-log window for navigating to the next/previous error or warning.
- The incremental search feature now has more intelligent scrolling for hits and also allows scrolling the component being searched via the Up, Down, Page up and Page down keys.
- Dependencies⁽³⁴⁴⁾ can now optionally be resolved in a namespace which allows having different sets of dependencies for different parts of a test so that the dependencies don't interfere with each other. See the Dependency namespace⁽³⁴⁸⁾ attribute for details.

Bugs fixed:

- Due to the JIDE libraries used, the new workbench view does not work with JDK 1.4. It now gets deactivated automatically when starting QF-Test with a 1.4 JDK.
- Several exceptions caused by the new workbench view and the multi result dialogs were fixed.

- In the new workbench view QF-Test could lose track about which suites are currently loaded, causing a suite to get loaded more than once, leading to confusing behavior and sometimes an exception.
- The new mechanism for on-the-fly SWT instrumentation only handled one SWT plugin, so that in case of a multi-platform Eclipse based application with several SWT plugins the correct plugin for the current platform could remain uninstrumented.
- Calling `stopRun` could cause the shared run-context in the QF-Test daemon to become unusable.
- Text checks for SWT Group widgets were missing.
- Fixed automatic scrolling of web pages for image checks and replay of hard events.
- The popup sub-menu for setting and removing marks was sometimes empty.

B.48 Version 3.1.2

New features:

- Support was added for the latest Eclipse/SWT version 3.5.1.
- We have started work on special support for several AJAX web toolkits, including Ext GWT, Ext JS, GWT, qooxdoo and RAP. Instead of deeply nested hierarchies of DIV nodes QF-Test now operates in terms of the actual rich interface of buttons, labels, trees etc. For information on how to activate this feature in its current state, please get in touch with QFS.
- QF-Test now comes with support for interacting with the open source test management tool TestLink. Please see [section 19.6^{\(182\)}](#) for details.
- By default QF-Test now starts with a "Workbench View" where all open test-suites are arranged as tabs in a single window. This feature is work in progress. In case it causes problems or if you prefer the old style, you can switch back to multiple windows via the **View** menu.
- The new function **Analyze references** in the context menu of a node can be used to statically validate the correctness of a test-suite by verifying the existence of the target nodes of [Procedure calls^{\(381\)}](#), [Test calls^{\(327\)}](#), [Dependency references^{\(347\)}](#) and references to [Components^{\(545\)}](#).

- Operations that search for or result in changes to multiple nodes now bring up a mass result dialog listing the affected nodes. You can either navigate to specific nodes directly from that dialog or use it to set marks on some or all of the resulting nodes.
- Nodes in a test-suite can now be marked with markers in four different colors for easier reference and quick navigation. Marks can be set either individually via the menu **Edit→Marks** or the context menu, or from a mass result dialog.
- All text fields, terminals etc. in QF-Test's user interface can now be searched locally by pressing **Ctrl-F** when the field owns the keyboard focus. Details are described in [section 2.3.1^{\(19\)}](#).
- The new Component Method Spy retrieves and displays fields and methods of Swing or SWT components as well as the Javascript properties of a DOM node in the web client (see [section 5.5^{\(44\)}](#)).
- After recording a new sequence and transforming it into a [Procedure^{\(378\)}](#) you can select **Parameterize node** from its context menu to tell QF-Test to automatically extract parameters for it from the contained nodes.
- The Eclipse/SWT instrumentation performed by the Procedure `qfs.qft#qfs.swt.instrument.setup` now operates on-the-fly, building a new plugin from an existing plugin and the QF-Test specific add-ons. Thus it is now possible to instrument custom SWT plugins or plugins for unofficial Eclipse versions for which QF-Test doesn't provide an exactly matching instrumented plugin.
- Event nodes can now be transformed into a [Wait for component to appear^{\(512\)}](#) node.
- The keyboard shortcut for the 'Redo' operation was changed from **Ctrl-R** to **Ctrl-Y**. Over the last years the latter has become a de facto standard.
- The warning about nested [Test-case^{\(314\)}](#) execution can now be suppressed via the option [Log warning for nested test-cases^{\(271\)}](#).
- Where scrolling a web page is required to perform a hard event or an image check, QF-Test now tries to scroll a web page itself via Javascript calls instead of relying on the browser's internal mechanism which often completely messes up the display.
- Component recognition based on geometry can now be suppressed by setting the coordinate and size attributes of a [Component^{\(545\)}](#) to '-'.
 - It is now possible to specify just the [Class count^{\(549\)}](#) or [Class index^{\(549\)}](#) of a Component node and ignore the other.

SWT

Web

Bugs fixed:

- Using the replace dialog to replace a regular expression with an empty string caused an exception.
- The function to make Procedure calls or Dependency references relative did not work reliably.
- When moving several Procedures, the automatic correction of Procedure calls could fail in some special cases.
- Procedure recording now works correctly for custom classes with a special Class-Loader.
- When creating reports, warnings can be suppressed entirely by specifying `-report.warnings=false` for batch mode or deactivating the respective option in the dialog for interactive report generation.
- Recording a click on an editable table cell could cause the cell editor to get activated in the wrong table row.
- Fetch index⁽⁵⁰²⁾ nodes were not implemented for SWT.
- Removing or marking unused components, importing and other complex component operations could fail if the test-suite contained a Data table⁽³⁶⁰⁾ node with no columns or empty table cells.

B.49 Version 3.1.1

Incompatible changes:

- The changes to the check system described below are fully backwards compatible. However, once a test-suite has been saved with QF-Test version 3.1.1 it cannot be run correctly with an older QF-Test version except for QF-Test version 3.0.3 which has explicit bidirectional compatibility support for this feature. Once you introduce QF-Test version 3.1.1 you can safely continue to work with QF-Test 3.0.3, but you should not use older versions like 3.1.0 or 3.0.2.

New features:

- The transition to the new check mechanism is now complete. Custom `Checkers` are now fully documented (section 39.3⁽⁷¹⁵⁾) and can be created for all check data types. All `Check` nodes now have a `Check type identifier` attribute and the old 'Check editable', 'Check enabled' and 'Check selected' nodes were replaced with Boolean check⁽⁴⁷⁶⁾ nodes with a matching Check type identifier⁽⁴⁷⁸⁾.

- It is now possible to bookmark test-suites, run-logs and individual nodes in a test-suite so they are directly accessible via the new sub-menu **File→Bookmarks**. Please see [section 29.1.3^{\(246\)}](#) about how to create and manage bookmarks.
- The new `ClassNameResolver` (see [section 39.1.2^{\(680\)}](#)) - though also useful for SWT and Swing applications - is an important milestone for AJAX web testing. With its help and the new pseudo class hierarchy for DOM nodes (see [section 5.4^{\(41\)}](#)) a deeply nested hierarchy of DIV nodes can be turned into a concise component hierarchy with expressive class names that vastly improve component recognition.
- [Procedures^{\(378\)}](#) can now automatically be parameterized via the sub-menu **Parameterize node** in the popup menu for the Procedure node. This feature is work in progress and will need more fine-tuning so we hope for your feedback.
- When creating testdoc documentation it is now possible to follow [Test call^{\(327\)}](#) nodes and include the target [Test-case^{\(314\)}](#), [Test-set^{\(321\)}](#) or test-suite into the generated documentation by activating the respective option in the dialog for interactive use or via the command line argument `-testdoc.followcalls(585)` in batch mode.
- Several more variants for converting and (un)packing nodes were added.

Bugs fixed:

- Several problems with split run-logs were fixed. Log entries could get lost in compact logs, when the split run-log was not saved as a zip file or when split run-logs were disabled but the attribute Name for separate run-log was defined.
- `DaemonTestRunListeners` did not work correctly in asynchronous mode.
- When creating a report, screenshots for failed [Check image^{\(488\)}](#) nodes were always saved regardless of whether the failed check actually represented an error or was just a part of a conditional construct.
- Use of the new Java regular expressions in QF-Test's replace dialog was not correct so that some replacements based on regexps were not possible.
- When determining uniqueness of 'ID' attributes for use as names, QF-Test was too strict so that in some cases potential names were not assigned.
- Instrumenting an OpenJDK directory now works.
- Waiting for a new document did not handle the window name attribute correctly.
- Waiting for a download to finish always waited for the full timeout.
- In very rare cases replaying text input could crash an SWT application on Linux.

Web

Web

Web

Web

SWT

- Option settings from `rc.setOption()` in the SUT were reset whenever options were changed in the option dialog.
- Event replay for SWT `MenuItems` with Gtk 2.4 on RedHat Enterprise Linux 4 was not reliable.
- The `ktable` and `gef` Jython modules for the respective SWT components were not correctly updated for the new `ItemResolver` and `Checker` mechanisms.
- QF-Test now properly discerns the most modal SWT `Shell` in case multiple modal `Shells` are opened simultaneously.

B.50 Version 3.1.0

Incompatible changes:

- QF-Test now uses standard Java regular expressions by default which are much more powerful than the old GNU regexp package. One major difference is that '{' and '}' are special characters for Java regexps and may need to be escaped with a '\\'. If necessary, you can switch back to the old GNU regexps via the option Use old-style GNU regexps (from before QF-Test version 3.1)⁽²⁴¹⁾. See also section 36.4⁽⁶¹⁴⁾.
- In contrast to Mozilla, Internet Explorer reports a local navigation to an anchor on the current page as a normal navigation event, so up to now a Wait for document to load⁽⁵¹⁶⁾ node was recorded which failed when replaying the same test on Mozilla. QF-Test now treats this situation uniformly by ignoring the extra navigation from IE so no Wait for document to load node is recorded with either browser. Consequently, old Wait for document to load nodes for local navigation will now fail and need to be removed from your tests.

New features:

- QF-Test's embedded browser for web testing now supports Java applets with JDK 1.6.0_10 and up, so QF-Test can now test the interaction of a web application with its embedded Java applet.
- Many of QF-Test's options, specifically those with an impact on test execution, can now be changed at run-time via the new run-context method `rc.setOption` described in section 13.3.9⁽¹⁰⁸⁾ and section 37.6⁽⁶²²⁾.
- When moving Procedure⁽³⁷⁸⁾, Dependency⁽³⁴⁴⁾ or Test-case⁽³¹⁴⁾ nodes, all references to these nodes can now be updated automatically. Similar updates are now

performed when changing the name of such nodes or QF-Test IDs of Components⁽⁵⁴⁵⁾ via the replace dialog. As an additional feature the new special Variable group `${id:...}` (see section 7.5⁽⁶²⁾) and the run-context method `rc.id` (see section 37.6⁽⁶²²⁾) can be used to mark arbitrary text as the definition of a QF-Test component ID that will be updated whenever the referenced component changes.

- Procedure calls⁽³⁸¹⁾ from one Procedure to another Procedure in the same Package hierarchy can now use a relative procedure name. Also, Procedures and Packages can now be made local to their test-suite by prefixing the name with an `'_'`. See section 9.3.2⁽⁸³⁾ and section 9.3.1⁽⁸³⁾ for details.
- The selection dialog for a Procedure, Dependency or Test-case node now also shows the comment of the selected node. Also, it is now possible to select multiple nodes in order to insert multiple calls at once.
- A Procedure call to a specific Procedure can now be created by copying the Procedure node and pasting it to a place where insertion of the Procedure is not allowed, which is basically everywhere a Procedure call is allowed except for the Extras⁽³⁴²⁾ node. Dragging and dropping a Procedure with the same effect is also supported.
- It is now possible to interact directly with the Javascript engine of the browser and execute arbitrary scripts via the methods `DomNode.toJS`, `DomNode.evalJS` and `DocumentNode.evalJS`, described in section 38.1⁽⁶⁴⁷⁾.
- When an error or exception occurs, the run-log now contains a stacktrace node, showing all nodes on the execution stack including their variable bindings, similar to the variables view in the debugger window.
- There is a new API for various kinds of resolvers and checkers, including an `ItemResolver` interface for implementing your own sub-items of custom components and a `Checker` interface for custom checks that can be captured and replayed just like QF-Test's standard checks. See section 39.2⁽⁷⁰³⁾ and section 39.3⁽⁷¹⁵⁾ for further information.
- The new `TestRunListener` API, described in section 39.5⁽⁷²⁸⁾, can be used to monitor a test-run from a script. Its variant `DaemonTestRunListener` supports the same remotely via QF-Test's daemon API. It is documented in section 40.2.5⁽⁷⁵⁵⁾.
- Support for Eclipse/SWT now includes the latest Eclipse milestone 3.5M7.
- The run-context method `rc.callProcedure` now also works from an SUT script⁽⁴¹⁹⁾ in a web application.

Web

SWT

Web

- Execution of a Test-case or Test-set can now be aborted or skipped via the run-context methods `rc.skipTestCase`, `rc.skipTestSet`, `rc.stopTestCase` and `rc.stopTestSet`, e.g. to prevent wasted time and irrelevant error messages in case a dependency cannot be resolved. See [section 37.6^{\(622\)}](#) for details.
- The name of the currently executing Test-case is now available via the special variables `${qftest:testcase.name}`, `${qftest:testcase.qname}` and `${qftest:testcase.reportname}` (see [section 7.5^{\(62\)}](#)).
- Via the new methods `setProperty` and `getProperty` in the class `Node` from QF-Test's pseudo DOM API for web applications, arbitrary data can be associated with DOM nodes without affecting the web application.
- The new command line argument `-ping(578)` for `qftest -batch -calldaemon` can be used to check whether a QF-Test daemon is up and running.

Bugs fixed:

- In QF-Test version 3.0.2 hard TAB characters in Jython or Groovy scripts were replaced with blanks when showing the script, immediately causing the respective node to become modified. This change has been taken back.
- For SUTs running on Mac OS X QF-Test disables use of the screen menu bar to allow proper capture/replay of menu actions.
- With JDK 1.6 some of QF-Test's dialogs were not displayed with the correct size.
- Activating an engine license in the option dialog now works as expected.
- When recording procedures, a new root Package is always created instead of merging the procedures into a possibly existing Package named "procbuilder".

B.51 Version 3.0.2

New features:

- Support was added for Eclipse/SWT versions 3.5M6 and 3.4.2.
- The method `callProcedure` was added to the class `DaemonRunContext` of QF-Test's daemon API. This method and the existing method `runTest` now also support direct passing of parameters to this call. See [chapter 40^{\(741\)}](#) for details.

Bugs fixed:

- Web**
 - Wait for document to load⁽⁵¹⁶⁾ nodes for web clients did not work reliably in case of multiple browser windows.
- Web**
 - Checking the enabled/disabled state of components is now also implemented for web clients.
- Web**
 - Capture and replay of MOVED and SIZED events is now also supported for browser windows of web clients.
 - QF-Test now performs regular explicit garbage collections to prevent running out of memory in the PermGen Space, a special Java memory region that causes trouble for a lot of Java applications. For the SUT these extra garbage collections can be turned off via the option Automatically perform garbage collection in the SUT⁽²⁷⁶⁾.
- SWT**
 - Disposal and recreation of an SWT `Display` is now handled correctly.
- SWT**
 - Custom SWT `Widgets` implemented as an anonymous local class were not treated correctly, possibly leading to a `ComponentNotFoundException`.
 - For checks with a very long timeout deadlock detection is now adjusted accordingly.
 - The demonstration test-suite for integration with the Imbus TestBench was updated to the latest TestBench version and now handles split run-logs correctly.

B.52 Version 3.0.1

New features:

- The Jython and Groovy run context `rc` now provides the convenience methods `getBool`, `getInt`, `getNum` and `getString` for simplifying lookup of typed variable values in scripts.
- It is now possible to specify proxy settings for a web SUT directly in the quickstart wizard or via the procedure `qfs.web.browser.settings.setProxy` in the standard library test-suite `qfs.qft`.
- An entry for Swing `JInternalFrame` components has been added to the template file for automatic procedure creation.

Bugs fixed:

- Web-testing with Mozilla on Linux no longer requires using a 1.4 JDK.

- Using the "Update component" feature for web pages with deeply nested tables could lead to duplicate component IDs.
- Left-to-right binding in Data table⁽³⁶⁰⁾ and other data nodes has been fixed.
- SQL statements in Database⁽³⁶³⁾ nodes did not work with Oracle databases, due to an automatically appended ';' character. This has been fixed. If your database engine requires that SQL statements are terminated with a ';', you have to specify it yourself.
- The procedure `qfs.swt.instrument.setup` now always picks the SWT plugin with an exact version match for instrumentation, if available. This is important for SWT 3.4, where versions 3.4.0 and 3.4.1 are mutually incompatible.
- When recording procedures, a '\$' character displayed in one of the fields of the SUT could lead to an exception.
- Groovy class loading and script parsing in the SUT should now be faster, in some cases - specifically applets - drastically.
- A caught exception in the Finally⁽⁴¹¹⁾ node of a Try⁽⁴⁰⁵⁾ node could mess up the state of an exception caught earlier in a Catch⁽⁴⁰⁸⁾ node.
- The demo test-suite `swt_addressbook.qft` was still based on SWT 3.1, support for which is no longer distributed with QF-Test. The suite is now based on SWT 3.4.

B.53 New features in QF-Test version 3

The following new features have been implemented for QF-Test version 3:

Description	Further info
Support for Web testing	Pseudo DOM API⁽⁶⁴⁷⁾ and many other places
Interoperation with HP/Mercury Quality Center and Imbus TestBench	Interaction with Test Management Tools⁽¹⁶³⁾
Groovy as a second scripting language	Scripting⁽⁹⁹⁾ , Scripting with Groovy⁽¹¹⁵⁾
New Jython version 2.2.1	Scripting⁽⁹⁹⁾
Improved Jython and Groovy script editor with syntax-highlighting	Fundamentals of the Jython integration⁽⁹⁹⁾
Automated creation of basic procedures	Automated Creation of Basic Procedures⁽¹⁵⁹⁾
Extended and unified functionality for 'Check...' and 'Wait...' nodes	Checks⁽⁴⁶⁹⁾ , Wait for component to appear⁽⁵¹²⁾ , Wait for client to connect⁽⁴³⁸⁾ etc.
Direct support for negated checks	Checks⁽⁴⁶⁹⁾
Split run-logs and IO redirection - no more Out-OfMemoryErrors	Split run-logs⁽⁷²⁾ , Log SUT output individually⁽³⁰⁸⁾ , Maximum size of client terminal (kB)⁽²⁷⁴⁾
Extensions and an external driver for the daemon API	Test execution⁽¹³⁸⁾ , Daemon mode⁽⁷⁴¹⁾
New concept: "Expected failure" for Test-case nodes	Test-case⁽³¹⁴⁾
Directing and documenting manual tests	Executing Manual Tests in QF-Test⁽²⁰¹⁾
Extended undo support	Self explanatory
Immediate vs. lazy variable bindings	Immediate and lazy binding⁽⁶⁶⁾
Image API for creating and saving screenshots	Image API⁽⁶⁴²⁾ , Image API extensions⁽⁷³⁷⁾
Crypting of text input for password fields	Salt for crypting passwords⁽²⁷²⁾ , Text⁽⁴⁵⁶⁾
Recursive sorting and transformation of nodes	Self explanatory
Revised search with highlighting of results	Self explanatory
Extensions to the standard library test-suite <code>qfs.qft</code>	pkgdoc
New tutorial chapters for advanced features	Advanced features

Table B.5: New features in QF-Test version 3

Appendix C

Keyboard shortcuts

C.1 Navigation and editing

This table gives a useful overview of QF-Test's basic and advanced navigation and editing keyboard shortcuts:

File Navigation	
Control-N	New
Control-O	Open
Control-S	Save
Basic Editing	
Control-Z	Undo last change
Control-Y	Redo last change
Search and Replace	
Control-F	Search
F3	Continue previous search
Control-G	Search again
Control-H	Replace
Workbench View	
Control-PageDown	Next test-suite
Control-PageUp	Previous test-suite
F5	Refresh project directory
Shift-F5	Rescan project directory
F6	Switch focus back and forth between suite and project view
Shift-F6	Select the current suite in the project tree, showing the project if necessary

Control-F6	Switch to previously active suite. Keep Control pressed and type F6 again to move further back. Simultaneously pressing Shift reverses the direction
Tree View	
Up / Down / Right / Left	Basic navigation
Alt-Up	Jump to previous sibling of node
Alt-Down	Jump to next sibling of node
Alt-Right	Expand node recursively
Alt-Left	Collapse node recursively
Shift-Up	Extend selection upwards
Shift-Down	Extend selection downwards
Control-Up	Move upwards without affecting selection
Control-Down	Move downwards without affecting selection
Control-Right	Scroll tree right
Control-Left	Scroll tree left
Space	Toggle selection of current node
Control-Backspace	Jump to last visited node
Shift-Control-Backspace	Jump to next selected node
Control-.	Clean tree
Alt-Return	Bring up node properties window
Shift-F10 / Windows context menu key	Bring up node popup menu
F2	Mark name or QF-Test ID of node in order to rename it
Tables	
Shift-Insert	Insert new row
Shift-Return	Edit selected row
Shift-Delete	Delete selected row
Shift-Control-Up	Move selected row up
Shift-Control-Down	Move selected row down
F2	Edit selected value
Return	Confirm changes
Escape	Discard changes
Shift/Control-Up/Down	Multi selection
Control-X / C / V	Cut / Copy / Paste
Code Editor	
Control-Space	Open available QF-Test variables for scripts or for dedicated nodes open a list of available methods.
Control-P	Find procedure definition (for lines which call a procedure)
Control-T	Find test definition (for lines which call a test)
Control-W	Find component (for lines which refer to a component)
Multi-line Text Elements	

Control-TAB	Move focus to next attribute
Shift-Control-TAB	Move focus to previous attribute
Control-Return	Confirm changes
For Procedure call nodes	
Control-P	Find procedure definition
For nodes with a QF-Test component ID attribute	
Control-W	Find component
For Test call nodes	
Control-P	Find called test definition
For Dependency reference nodes	
Control-P	Find dependency definition
Run-log	
Control-I	Open error list dialog
Control-L	Open latest run-log
Control-N	Find next error
Sift-Control-N	Find previous error
Control-T	Find node in test-suite
Control-W	Find next warning
Shift-Control-W	Find previous warning
Advanced Editing	
Control-A	Inserting a procedure call
Control-D	Add selected node to bookmarks
Shift-Control-D	Toggle disabled state of selected nodes
Control-I	Find references of node Available for components, tests, dependencies and procedures
Shift-Control-I	Pack selected nodes into if-sequence
Shift-Control-P	Transform selected nodes into procedure Available only valid for sequence and similar nodes
Shift-Control-S	Pack selected nodes into sequence
Shift-Control-T	Pack selected nodes into test-step
Shift-Control-Y	Pack selected nodes into try/catch

Table C.1: Shortcuts for navigation and editing

C.2 Record and replay functions

The following table contains important shortcuts for record and replay functions, which partly work also outside of QF-Test, i.e. directly in the SUT:

Replay	
Return	Play (execute current node)
F9	Pause
Alt-F12 (configurable)	Pause testrun (Don't Panic)
Record	
F11 (configurable)	Toggle "Record mode"
Record Checks	
F12 (configurable)	Toggle "Check mode"
Left Mouse Button	Record default check
Right Mouse Button	Show list with available checks
Record Components	
Shift-F11 (configurable)	Toggle "Record single component mode"
Control-V	Paste recorded QF-Test component ID from clipboard
Control-Backspace	Jump to last recorded component
Control-F11 (configurable)	Toggle "Record multiple components mode"
Record Procedures	
Shift-F12 (configurable)	Toggle "Record procedures mode"
Control-F12 (configurable)	Toggle "Record multiple procedures mode"
Debugger	
F7	Step in
F8	Step over
Control-F7	Step out
Shift-F9	Skip over
Control-F9	Skip out
Control-F8	Breakpoint on/off
Control-J	Jump to run-log
Control-T	Find node in test-suite
Control-,	Continue execution from currently selected node

Table C.2: Shortcuts for special record and replay functions

C.3 Keyboard helper

The following graphic may help to easily remember the assignment of functional keys used by QF-Test. It is intended to be printed, cut out and fixed above the area of functional keys F5 to F12 of your keyboard.

QF-TEST	Reload Prj Rescan Prj	Suite ↔ Prj Suite in Prj Last Suite	Step in Step out	Step over Breakpoint	Pause Skip over Skip out	Menu Popup M	Record Rec Comp Multi Re Co	Rec Check Rec Proc Don't panic Multi Re Pr	shift alt ctrl
	F5	F6	F7	F8	F9	F10	F11	F12	

Figure C.1: Keyboard helper

Appendix D

Glossary

API

Application Programming Interface, a set of package, class and method definitions that the programmer of an application can use. The Java API refers to the interface of the standardized Java class library that is shipped with each JDK.

AWT

Abstract Windowing Toolkit, the part of the Java library responsible for the display of windows and components as well as for the dispatch of events.

GUI

Graphical User Interface. An interface between a program and the user, usually consisting of windows built from components for displaying information and receiving input.

RMI

Remote Method Invocation, communication protocol/programming interface in Java for calling a method of a remote object.

SUT

System Under Test, the application being tested with QF-Test.

VM

The Java *Virtual Machine* executes Java programs that have been compiled to Java bytecode. It is responsible for the platform independence and compatibility of Java programs across various kinds of machines and operating systems.

Appendix E

Third party software

QF-Test makes use of the following software:

Apache POI

Copyright © 2009 The Apache Software Foundation
Distributed under the Apache Licence, Version 2.0 (see
`doc/licenses/apache-2.0` and `doc/licenses/poi-notice`).
URL: <http://poi.apache.org>

Bean Scripting Framework (BSF)

Copyright © 1999 International Business Machines Corporation.
See the license `doc/licenses/bsf`
URL: <http://www.alphaworks.ibm.com/tech/bsf>

Cryptix library

Copyright © 1995, 1996, 1997, 1998, 1999, 2000 The Cryptix Foundation
Limited. All rights reserved.
See the license `doc/licenses/cryptix`
URL: <http://www.cryptix.org>

dom4j

Copyright © 2001-2005 MetaStuff, Ltd. All Rights Reserved
See the license `doc/licenses/dom4j`
URL: dom4j.sourceforge.net

GNU regex

Copyright © 1998-2001 Wes Biggs.
Distributed under the GNU Lesser General Public License (see [doc/licenses/LGPL](#)).
Source code included in `misc/gnu_src.jar`
URL: <http://nlp.stanford.edu/nlp/javadoc/gnu-regexp-docs/index.html>

Groovy scripting language

Distributed under the Apache Licence, Version 2.0 (see [doc/licenses/apache-2.0](#)).
URL: <http://groovy.codehaus.org>

JCommon

Copyright © 2007-2012 Object Refinery Limited and Contributors.
Distributed under the GNU Lesser General Public License (see [doc/licenses/LGPL](#)).
URL: <http://www.jfree.org/jcommon/index.html>

jEdit Syntax Highlighting

Public domain
The jEdit 2.2.1 syntax highlighting package contains code that is Copyright © 1998-1999 Slava Pestov, Artur Biesiadowski, Clancy Malcolm, Jonathan Revusky, Juha Lindfors and Mike Dillon.
URL: <http://syntax.jedit.org>

JExcel

Copyright © 2002 Andrew Khan. All rights reserved.
Distributed under the GNU Lesser General Public License (see [doc/licenses/LGPL](#)).
Source code included in `misc/jexcelapi.tar.gz`
URL: <http://www.andykhan.com/jexcelapi/index.html>

JFreeChart

Copyright © 2005-2012 Object Refinery Limited and Contributors.
Distributed under the GNU Lesser General Public License (see [doc/licenses/LGPL](#)).
URL: <http://www.jfree.org/jfreechart/index.html>

JGoodies Looks

Copyright © 2001-2005 JGoodies Karsten Lentzsch. All rights reserved.
See the license `doc/licenses/jgoodies-looks`
URL: <http://www.jgoodies.com>

JIDE Common Layer

Copyright © 2002 - 2009 JIDE Software, Inc, all rights reserved.
Distributed under the GNU General Public License (see `doc/licenses/GPL`)
with Classpath Exception (see `doc/licenses/Classpath`).
URL: <http://www.jidesoft.com/products/oss.htm>

JNIWrapper

Copyright © 2000-2004 MIK Ltd. All rights reserved.
URL: <http://www.jniwrapper.com>

JSch

Copyright © 2002-2012 Atsuhiko Yamanaka, JCraft, Inc. All Rights Reserved.
See the license `doc/licenses/jsch`
URL: <http://www.jcraft.com/jsch/>

Jython scripting language

Copyright © 2007 Python Software Foundation; All Rights Reserved.
See the license `doc/licenses/jython`
URL: <http://www.jython.org>

JZlib

Copyright © 2000-2011 ymnk, JCraft, Inc. All rights reserved. See the license
`doc/licenses/jzlib`
URL: <http://www.jcraft.com/jzlib/>

Mozilla

Copyright © 1998-2006 Contributors to the Mozilla codebase.
Distributed under the Mozilla Public License (see
`doc/licenses/MPL-1.1.html`).
URL: <http://www.mozilla.org/projects/embedding/>

Netty

Copyright © 2009 Red Hat, Inc.
Distributed under the Apache Licence, Version 2.0 (see

doc/licenses/apache-2.0).
URL: <http://netty.io>

PngEncoder

Copyright © 1999-2000 J. David Eisenberg. All rights reserved.
Distributed under the GNU Lesser General Public License (see
doc/licenses/LGPL).
Source code included in `misc/PngEncoder.tar.gz`
URL: <http://catcode.com/pngencoder/>

Python library (distributed with Jython)

Copyright © 2001-2008 Python Software Foundation; All rights reserved.
See the license `doc/licenses/python`
URL: <http://www.python.org>

SendSignal.exe

Permission to use granted by the author Louis K. Thomas.
URL:
<http://www.latenighthacking.com/projects/2003/sendSignal/>

Sixlegs PNG library

Copyright © 1998, 1999, 2001 Chris Nokleberg
Distributed under the GNU Lesser General Public License (see
doc/licenses/LGPL).
URL: <http://www.sixlegs.com>

STAX - Streaming API for XML

Copyright © 2003-2007 The Apache Software Foundation
Distributed under the Apache Licence, Version 2.0 (see
doc/licenses/apache-2.0).
URL: <http://geronimo.apache.org>

TrueZIP Virtual File System API

Copyright © 2005-2007 Schlichtherle IT Services.
Distributed under the Apache Licence, Version 2.0 (see
doc/licenses/apache-2.0).
URL: <http://truezip.dev.java.net>

wxActiveX - Library for hosting ActiveX controls within wxwidgets programs

Copyright © 2003 Lindsay Mathieson.

Distributed under the wxActiveX Library Licence (see <doc/licenses/wxActiveX>).

URL: <http://sourceforge.net/projects/wxactivex>

wxMozilla - Embedding Mozilla in wxWindows

Copyright © 2003-2006 Jeremiah Cornelius McCarthy.

Distributed under the wxWindows Library Licence (see <doc/licenses/wxWidgets>).

URL: <http://wxmozilla.sourceforge.net/>

wxWidgets GUI API

Copyright © 1998-2005 Julian Smart, Robert Roebling et al.

Distributed under the wxWindows Library Licence (see <doc/licenses/wxWidgets>).

URL: <http://www.wxwidgets.org/>

XML Beans

Distributed under the Apache License, Version 2.0 (see <doc/licenses/apache-2.0> and <doc/licenses/xmlbeans-notice>).

URL: <http://xmlbeans.apache.org>

XML parser Xerces

Copyright © 1999-2002 The Apache Software Foundation.

Distributed under the Apache Software License (see <doc/licenses/apache-xerces>).

URL: <http://xml.apache.org/xerces-j/index.html>

XSLT engine Xalan

Copyright © 1999 The Apache Software Foundation.

Distributed under the Apache Software License (see <doc/licenses/apache-xalan>).

URL: <http://xml.apache.org/xalan-j/index.html>