QSpin
Quality & SPI Network

Keywords:	QTaste, test, guidelines, user manual

REVISION RECORD

Rev.	Date	Description	Pages
1.0	02/10/2009	First version for Open Source release	All
1.1	29/03/2012	Minor updates (QTASTE_JYTHON_LIB), update some url for source forge	All

© 2009   QSpin SA

TABLE OF CONTENTS

TABLE OF FIGURES

1. INTRODUCTION

1.1. Why QTaste?

QTaste framework (QSpin Tailored Automated System Test Environment) is a generic test environment customizable to test different kind of systems. It can be used to test simple and complex hardware or software systems including a lot of different technologies. For that reason, the test api has to be "tailored" in order to enable the kernel to communicate with your system.

1.2. Purpose

This document describes the installation and configuration steps of the "QSpin Tailored Automated System Test Environment" (QTaste). It provides useful information to the test designer and to the developer in order to define a new test script to be used by the QTaste framework.

The test designer is responsible for writing test scripts based on the requirement documents. Associated to this task, the developer needs to provide a set of verbs that can be used by the script. The developer is responsible for the development of the newly defined QTaste verbs. Those activities must be performed with respect to the QTaste architecture.

1.3. Definitions, Acronyms and Abbreviations

Term	Description
Acceptance criterion	Set of expected results expected at the output of the system when specific test data are provided. These can be exact values, ranges, probabilities, ...
Actual outcome	Contains a brief description of what the tester saw after the test steps have been completed. This is compared to the expected results in order to decide if the test is Success/Fail.
Agent	Agents are active objects.They are capable of performing operations. Such agents may be software agents, hardware devices, or humans. They read inputs and generate outputs supposedly according to their requirements.
Automated testing	Execution of tests without the intervention of the tester. Limited intervention of the tester for the introduction of results that are not available through interfaces, protocols and middlewares are possible during automated testing.
Control script	A control script is a shell script able to start or stop a System Under Test (SUT).
Developer	Person implementing some component, hardware or software and their stubs.

Environment	(Sub-)set of systems, modules and components needed to perform a specific execution of a (sub-) set of operation of the original and complete system.
Expected results	is a description of the results of the successful execution of a test case
Failure	in case of a failure, the software does not do what the user expects, according to the validation rule
GPL	The GNU General Public License (www.gnu.org)
Input data	is a set of values for input variables of a test case
Intrusive, non-intrusive and low-intrusive test methods	Intrusion means that the test is not using the public API of the tested component. Low-intrusion means that the behaviour of the component has been changed using the public API of a sub-component.
LGPL	The GNU Lesser General Public License (www.gnu.org)
Manual testing	Execution of tests that are requiring the interaction of the tester or the test designer, to request some operation, to introduce test data, to introduce test results or to introduce validation rules.
Nominal cases	Test case that is likely to happen during the nominal exploitation of the system.
Non-nominal cases	Test case that may happen during the exploitation of the system during a fault or a degradation
Open Source	source code is provided
Proxy	A class functioning as an interface to another thing. The other thing could be anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate.
Requirement	Is prescriptive statement of intent about the system to be, to be enforced by a single agent of the system-to-be through actuation of the variable under its control.
Simulated environment	Environment that is using simulated and dummy (sub-)systems, modules and components.
Stub	A piece of software or hardware that mimics the activity of a missing component
SUT	System Under Test
Test API	is the API providing methods to test a system or component and check the results
Test bed	the aggregate of system under test, the test system, and the simulated environment.
Test campaign	is a predefined set of test suites or test cases that will be executed in order to get feedback about the quality of a system.
Test case	is a scenario composed of a sequence of actions and validations under which a tester will determine if a requirement or use case upon an application is partially or fully satisfied. It may take many test cases to determine that a requirement is fully satisfied.
Test data	is a set of values for test cases, used as input data and as expected results
Test Designer	Person designing the tests to be executed on the QTaste. This person has a good understanding of the system under test.

Test environment	Synonym of test bed
Test Extension	is an extension to the Test API mainly used for Stress Test in order to « break » some components.
Test log	is a log of all the test verbs that have been executed and of their actual results
Test management tool	is a tool used to define and organize test cases, test data, test suites to perform a test campaign. It helps the tester to do the follow up of the test results.
Test report	is a report containing the result of the execution of a test suite
Test script	is a short program written in a test-dedicated programming language used to test part of the functionality of a software system. It mentions a set of steps that should be performed in order to execute the test case.
Test suite	is a collection of test cases
Tester	Person executing the tests defined by the Test Designer on the QTaste. Less skills are required than for the Test Designer, notably, he should not need to be an expert of the system under test in order to perform the tests.
Use case	is a technique for documenting the requirements of a new system or software change. Each use case provides one or more scenarios that convey how the system should interact with the end user or another system to achieve a specific business goal. Use cases typically avoid technical jargon, preferring instead the language of the end user or domain expert. Use cases are often co-authored by requirements engineers and stakeholders.
Validation	Checking that the behaviour of the system under test matches the expected behaviour, as described by its requirements.
Validation rule	Rule checking if the expected behaviour and the actual outcome are identical or are matching acceptance criterion (parametric check)
Verb	is an abstract definition of a functionality of the test API

2. OVERVIEW OF THE QTASTE FRAMEWORK
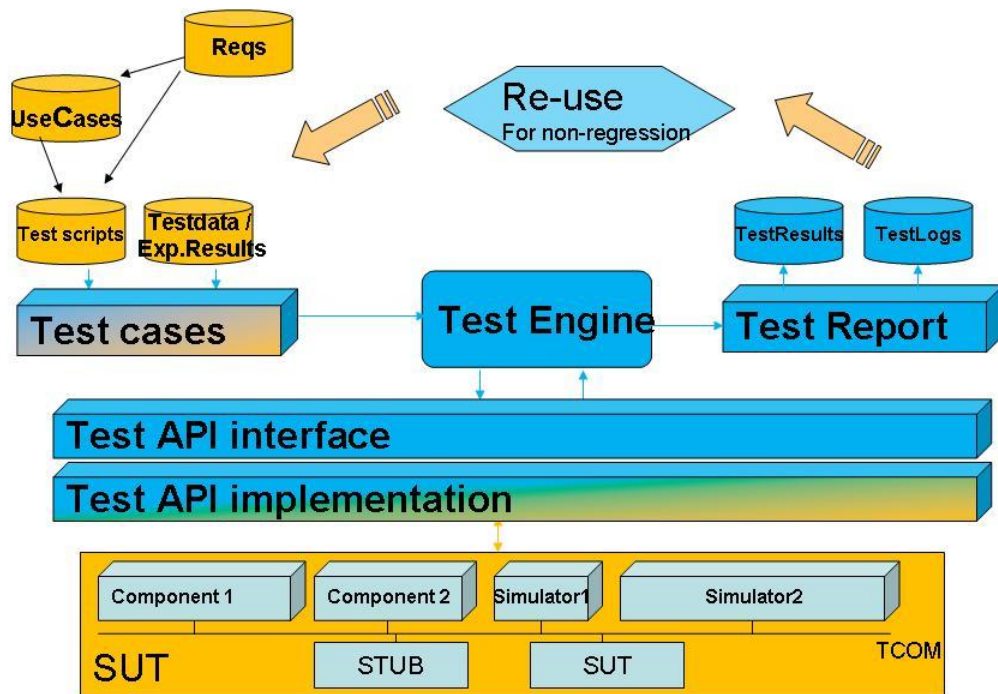
Figure 1: QTaste "Overall system architecture overview"

Requirements, taken from a requirement management tool or from a document are derived into Use cases and Test scripts specific to the SUT business.

Test scripts correspond to a sequence of Test API calls expressed in python scripting language. Each call corresponds to an operation or to a validation. Test cases are the combination of test scripts and the associated data.

An operation consists of a set of actions on the system under test. Validation consists of a check enabling the validation of a previous set of operations.

In order to support a data driven approach, Test Data are stored independently of test scripts in a CSV (Comma Separated Values) file.

```
##
# TestCalculator - Test the windows calculator.
# <p>
# Start the Windows calculator, perform simple computations and check results.
##

from qtaste import *

calculator = testAPI.getWindows()
expectedResult = testData.getValue("RESULT")

def testCalculator():
    """
    @step     Perform a simple computation using the Windows calculator
    @expected Compare the result provided by the calculator with the expected results
    """
    sessionID = calculator.startApplication("calc")
    calculator.pressButton("Calculator", "CE")
    calculator.pressButton("Calculator", "_9")
    calculator.pressButton("Calculator", "_+")
    calculator.pressButton("Calculator", "_7")
    calculator.pressButton("Calculator", "_=")
    text = calculator.getText("Calculator", "Edit").strip()
    expectedResult = testData.getValue("RESULT")
    if text != expectedResult:
        testAPI.stopTest(Status.FAIL, "Expected to get " + expectedResult + " but got " + text)
    calculator.stopApplication()

doStep(testCalculator)
```

Figure 2: QTaste "Example of python test script"

#	COMMENT	NUMBER1	OPERATOR	NUMBER2	RESULT
1	Verify the result of a addition	9	+	9	18,
2	Verify the result of a substraction	34	-	4	30,
3	Verify the result of a division	6	/	3	2,
4	Verify the result of a mulitplication	6	*	4	24,

Figure 3: QTaste "Example of Test Data"

The table contains a list of columns names and associated values used by the Test scripts as inputs or expected outputs. These values are defined by the test designer. The values can correspond to a string input, numeric values, date, file path or any other kind of data format such as ranges or fuzzy information.

Each line of the sheet corresponds to a specific test with predefined data. It may correspond to the "nominal case" scenario or it could be a set of data that will lead to an expected failure (e.g. value out of range) for "non nominal case" scenarios.

The combination of the test script and the test data allows the test designer to generate tests cases that will be run by the test engineer using QTaste. The creation of the test cases is a shared responsibility between the test designer and the developer, as specific formalism and functions might be required in order to allow the Test Engine to correctly parse and execute the complete sequences.

Once the test cases are executed by QTaste, test report is generated giving the result of the test case execution.

Test suite TestSuites\TestTranslate report

Testbed: demo_web

Report generation date: 2009-10-29 15:47:07

QTaste kernel version: qtaste-kernel-1.0-RC1
QTaste testAPI version: qtaste-demo-testapi-1.0-RC1
SUT version: undefined

Start execution	End execution	Tests executed	Tests passed	Tests failed	Tests in errors
2009-10-29 15:45:14	2009-10-29 15:47:07	3/3	2/3	1/3	0/3

Number of SUT restart to ensure clean SUT state: 1/3

Executive summary

Test script	Row	Result
TestTranslate	1	✓ Passed
TestTranslate	2	✗ Failed
TestTranslate	2	✗ Failed
TestTranslate	3	✓ Passed

Test script Start SUT

Version: -

Row	Status	Description
Start SUT	✓ Passed	Passed

Test script TestTranslate

Version: unversioned

Row	Status	Description
1 - translate good morning	✓ Passed	Passed
2 - translate people	✗ Failed	Expected to get blabla but got les gens Script stack trace: function stopTest function checkTranslation at file D:\QTaste_Integration_trunk\demo\TestSuites\TestTranslate\TestScript.py line 43 at file D:\QTaste_Integration_trunk\demo\TestSuites\TestTranslate\TestScript.py line 46 Date: 2009-10-29 15:46:20
Restart SUT	✓ Passed	Passed
2 - translate people	✗ Failed	Expected to get blabla but got les gens Script stack trace: function stopTest function checkTranslation at file D:\QTaste_Integration_trunk\demo\TestSuites\TestTranslate\TestScript.py line 43 at file D:\QTaste_Integration_trunk\demo\TestSuites\TestTranslate\TestScript.py line 46 Date: 2009-10-29 15:46:41
Restart SUT	✓ Passed	Passed
3 - translate wallet	✓ Passed	Passed

Test script Stop SUT

Version: -

Row	Status	Description
Stop SUT	✓ Passed	Passed

Legend: ✓ Passed ✗ Failed ⚠ Test in error 🔵 Running

Test summary

Testbed configuration

File: Testbeds/demo_web.xml

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<testbed_configuration>
    <testapi_implementation>
        <import>com.qspin.qtaste.testapi.impl.demo</import>
    </testapi_implementation>
    <control_script>control_selenium.py</control_script>
    <multiple_instances_components>
        <Selenium id="TranslateApp">
            <host>localhost</host>
            <port>4444</port>
            <url>http://babelfish.yahoo.com</url>
        </Selenium>
    </multiple_instances_components>
</testbed_configuration>
```

Testbed control script

File: Testbeds/ControlScripts/control_selenium.py

```
from controlscript import *

ControlScript([
    JavaProcess("Selenium Server",
            mainClassOrJar="demo/selenium-server.jar",
            checkAfter=5)
])
```

Figure 4: QTaste "Example of Test report"

3.　THE QTASTE TESTAPI PARENT

Each testapi have to inherit of testapi-parent component. This is used to make the internal qtaste dependencies available to the custom testapi. It is also used to share the configuration of maven plugins. (like plugins for the testapi documentation, etc)

4.　OVERVIEW OF THE QTASTE DIRECTORY STRUCTURES

4.1.　QTaste kernel directories

The following directories are located in the QTaste root directory.

Directory	Description
bin	This directory contains scripts to start the test engine, generate documentations and start simulators
conf	This directory contains the configuration files
kernel/target	This directory contains the QTaste kernel jar
lib	This directory contains all the shared libraries (dll and so) required by the QTaste framework
tools	This directory contains utilities and tools

4.2. Test specific directories

The following directories are located in a specific test directory.

Directory	Description
Testbeds	This directory contains the testbeds configuration files
Testbeds/ControlScripts	This directory contains the control scripts associated to the testbeds
TestCampaigns	This directory contains the test campaigns description files
TestSuites	This directory contains different test suites containing test scripts and test data
log	This directory contains execution logs of the QTaste framework. It will be automatically created after the first execution of the Test Engine
reports	This directory contains the test reports generated by the Test Engine
testapi/target	This directory contains the QTaste TestAPI jar and the generated HTML TestAPI documentation (in TestAPI-doc subdirectory)

5. USING QTASTE FRAMEWORK

5.1. Test Automation workflow

In order to develop new test cases using the QTaste framework, it is really important to understand the following points:

- Identification of the architectural components of the SUT and their interfaces that will be used to perform the tests. Which technologies are required to communicate with the components interfaces?
- Are the components testapi already implemented in QTaste? Are the technologies required to communicate with components already supported by the QTaste?
- Do the components have the required verbs to perform the operations and checks?
- What are the parameters required to perform operations on the components? Identify variables that can be used as "Test Data"
- What configuration parameters of the testbed are required in order to use the components in any testbed configurations?
- Design the QTaste components interfaces and develop component implementations
- Develop the python test script(s)
- Add rows in the "Test Data" to test specific cases
- Execute the test suite(s)
- Analyse the test reports and report test failures

5.2. Starting the Test Engine

The command to execute a test suite or test script is:

- ${QTASTE_HOME}/bin/qtaste_start.bat (or qtaste_start.sh for Unix platform)

which must be run from the test specific directory containing the directories specified in section 4.2.

To simplify usage, it is advised to put the ${QTASTE_HOME}/bin directory in your path.

Usage: <command> -testsuite <testsuiteDirectory> -testbed <configFileName.xml>
                 [-sutversion <SUT_version>] [-engine <engineFileName.xml>]
                 [-loop [<count> | <hours>h]]

Option	Parameters	Presence	Meaning
-testsuite	testsuiteDirectory	MANDATORY	Specify the directory containing the testsuite(s)
-testbed	configFileName.xml	MANDATORY	Specify the testbed configuration file to be used for the test.

-sutversion	SUT_version	OPTIONAL	Specify the SUT version that will be reported.
-engine	engineFileName.xml	OPTIONAL	Specify the engine configuration to be used for the test. By default, the file conf/engine.xml is used.
-loop	None | <count> | <hours>h	OPTIONAL	Specify to execute the test suite in loop, respectively infinitely, <count> times or during <hours> hours.

5.3. Starting the Meta-Campaign launcher

A meta-campaign specifies a set of testsuites or testscripts to be executed on specified testbeds. Optionally, the number of times they must be executed can be specified as well as the test data rows for which a test script must be executed.

The command to execute a meta-campaign is:
- ${QTASTE_HOME}/bin/qtaste_campaign_start.bat (or qtaste_campaign_start.sh for Unix platform)

which must be run from the test specific directory containing the directories specified in section 4.2.
To simplify usage, it is advised to put the ${QTASTE_HOME}/bin directory in your path.

Usage: <command> <campaignFileName> [-sutversion <SUT_version>]

Option	Parameters	Presence	Meaning
	campaignFileName	MANDATORY	Specify the name of the XML campaign file
-sutversion	SUT_version	OPTIONAL	Specify the SUT version that will be reported.

The format of the campaign file is the following:

```
<campaign name="Campaign_Name">
   <run testbed="testbedName.xml">
      <testsuite directory="testSuiteDirName">
         [<testdata selector="commaSeparatedListOfRowId"/>] (optional, to
            execute scripts only for specified test data rows; row id starts at 1)
         [<count>numberOfTimesOrHoursToExecute</count>] (optional, to execute in
         loop)
         [<loopInHours/>] (optional, if numberOfTimesOrHoursToExecute is in hours)
      </testsuite>
      <testsuite …> … </testsuite>
      …
   </run>
   <run …> … </run>
   …
</campaign>
```

Here below an example of campaign file:

```
<campaign name="Campaign example">
    <run testbed="enginetest.xml">
        <testsuite directory="TestSuites/TestSuite_QTaste/EngineSuite/QTaste_SCRIPT"/>
        <testsuite
directory="TestSuites/TestSuite_QTaste/EngineSuite/QTaste_KERNEL/QTaste_RES_06">
            <testdata selector="1,4,5"/>
            <count>2</count>
        </testsuite>
        <testsuite
directory="TestSuites/TestSuite_QTaste/EngineSuite/QTaste_DATA/QTaste_DATA_01"/>
        <testsuite
directory="TestSuites/TestSuite QTaste/EngineSuite/QTaste DATA/QTaste DATA 04"/>
        <testsuite
directory="TestSuites/TestSuite QTaste/EngineSuite/QTaste DATA/QTaste DATA 05"/>
    </run>
</campaign>
```

5.4. Starting QTaste Graphical User Interface

The QTaste Graphical User Interface can be used in order to interact with the QTaste test engine and QTaste test campaign.
The command to execute QTaste with its GUI is:

- ${QTASTE_HOME}/bin/ qtasteUI_start.bat (or qtasteUI_start.sh for Unix platform)

which must be run from the test specific directory containing the directories specified in section 4.2.
To simplify usage, it is advised to put the ${QTASTE_HOME}/bin directory in your path.

Usage: <command> -testsuite <testsuiteDirectory> [-testbed <configFileName.xml>]
[-engine <engineFileName.xml>] [-loop [<count> | <hours>h]]

Option	Parameters	Presence	Meaning
-testsuite	testsuiteDirectory	OPTIONAL	Specify the directory containing the testsuite(s) to use when the GUI is started. So using this parameter, a test suite is launched.
-testbed	configFileName.xml	OPTIONAL	Specify the testbed configuration file to select, if not specified the last used testbed is selected.
-engine	engineFileName.xml	OPTIONAL	Specify the engine configuration to be used for the test. By default, the file conf/engine.xml is used.
-loop	None	<count>	<hours>h	OPTIONAL	Specify to execute the test suite in loop, respectively infinitely, <count> times or during <hours> hours. This parameter is only taken into account when the parameter testsuite is also specified.

5.5. HTML Test report document

Test suite TestSuites\TestCalculator report

Testbed: demo_gui

Report generation date: 2009-10-02 14:20:08

QTASTE kernel version: continuous-SNAPSHOT (build r3512)
QTASTE testAPI version: qtaste-demo-testapi-continuous-SNAPSHOT (build rnull)
SUT version: undefined

Start execution	End execution	Tests executed	Tests passed	Tests failed	Tests in errors
2009-10-02 14:19:29	2009-10-02 14:20:08	1/1	1/1	0/1	0/1

Number of SUT restart to ensure clean SUT state: 0/1
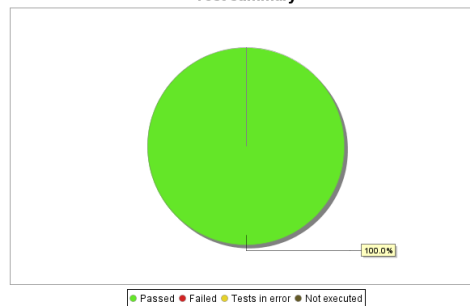
Executive summary

Test script	Row	Result
TestCalculator	1	✔ Passed

Test script TestCalculator

Version: undefined

Row	Status	Description
1 - Verify the result of a addition	✔ Passed	Passed

Legend: ✔ Passed ✘ Failed ⚠ Test in error ◯ Running

Test summary

● Passed ● Failed ● Tests in error ● Not executed

Testbed configuration

File: Testbeds/demo_gui.xml

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<testbed_configuration>
    <testapi_implementation>
        <import>com.qspin.qtaste.testapi.impl.demo</import>
    </testapi_implementation>
    <control_script>control_windows.py</control_script>
    <singleton_components>
        <Windows>
            <host>localhost</host>
            <port>8080</port>
        </Windows>
    </singleton_components>
</testbed_configuration>
```

Testbed control script

File: Testbeds/ControlScripts/control_windows.py

```
from controlscript import *

ControlScript([
    NativeProcess("Windows control native Agent",
            executable="python.exe",
            args="demo/pywinauto-0.3.8/XMLRPCServer.py",
            checkAfter=5)
])
```

Figure 5: QTaste "HTML Test reports"

Some additional remarks about HTML test reports document:
- The versions of the test scripts may be extracted from a version control tool (like subversion repository as example) during the execution of the tests. The version of will be reported only if:
  - ○ the subversion command-client ("svn") is available in the PATH
  - ○ testsuites are used in a svn repository and are a "tagged" as version. (rely on svn URL) .

- o The engine configuration file (conf/engine.xml) tag <version_control> is configured properly. (com.qspin.qtaste.util.versioncontrol.impl.SubversionVersionControl)

The version will be set to "undefined" in all the other cases.

- SUT version field is coming from either the SUT version field if the test is started from the QTaste GUI or the "-sutversion" parameter if started from the command-line.

5.6. Test API Documentation

The HTML Test API documentation is automatically generated by the maven Test API projects, in the *target/TestAPI-doc* subdirectory and is accessible from the GUI (see Figure 28: Navigation buttons).

Figure 6: QTaste "Test API documentation"

5.7. Additional Java librairies

If for some reason, QTaste needs additional libraries, they can be specified through the environment variable QTASTE_CLASSPATH. These libraries will be automatically added to the class path and available during the QTASTE execution.

6. QTASTE GRAPHICAL USER INTERFACE

6.1. Starting tests

6.1.1. QTaste User Interface startup

The command to start the QTaste User Interface is the following:
${QTASTE_HOME}/bin/qtasteUI_start.bat (or qtasteUI_start.sh for Unix platform)
which must be run from the test specific directory containing the directories specified in section 4.2.

6.1.2. Testbed management

For first startup a default testbed is selected, otherwise the last selected testbed during previous sessions will be selected by default.
The user can change the testbed for which test must be executed by using the dedicated combo box displayed in the Information panel.

Figure 7: Current selected testbed

When a testbed is selected, the current selection is stored in the preferences file (${QTASTE_HOME}/conf/gui.xml) in order to select it by default at the next launch.

A contextual menu is available on this list to view or edit the testbed configuration file.
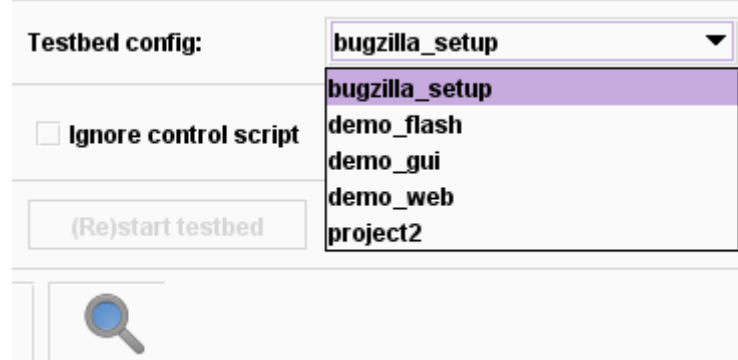
Figure 8: Testbed selection

6.1.3. Test suite execution

From the selection panel where the test cases are displayed, select the test case (Figure 9: test case execution) or test suite to execute (Figure 10: Test suite execution), then press on the "Execute test"' button (Figure 11: Execute test button).

Figure 9: test case execution

Selection of a test suite containing several test cases
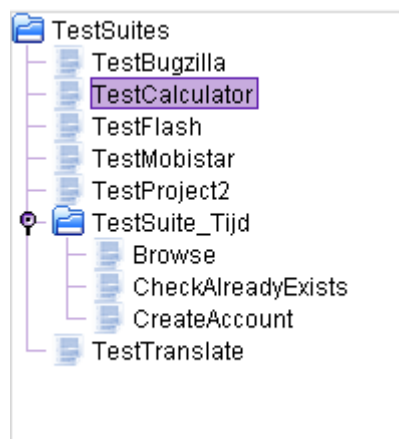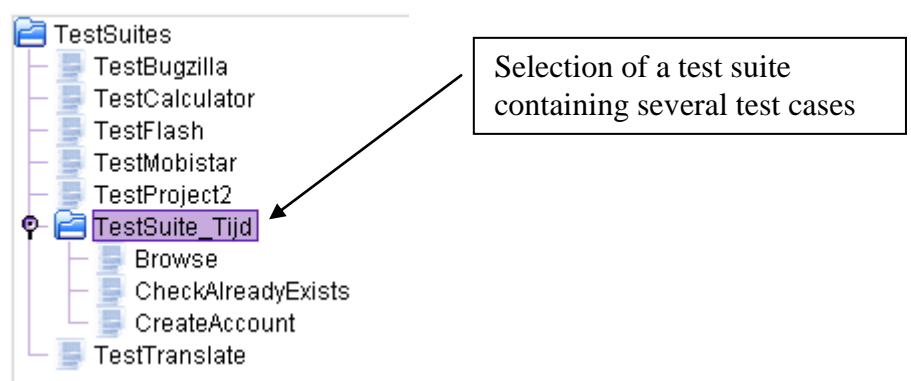
Figure 10: Test suite execution

Figure 11: Execute test button

6.1.4. Test case result analysis

The tab "Test Case Results" is automatically displayed when the executed test is started.

Figure 12: Test Case Results panel

The test case results panel displays the following information:
- Summary of the test case execution status;
- Error details when test case is failed
- Stack trace when test case is failed
- Log4j panel displayed QTaste logs

When a test case result is failed (or Not available status), the corresponding icon is displayed. By selecting the row, the error details and the available stack traces will be displayed.

From the Error details, the user can double click on the row and the Python editor will be opened at the line of the detected error. This is available for any python script except for the one having the file name "embedded_jython" which is built dynamically by QTaste during the test execution.

By selecting test case result row, the Log4j panel will point to the first log associated to the test case.

For more details about the use of the Log4j panel filters, please refer to **Error! Reference source not found.**.

It is possible to re-run a specific test case row (or several rows) using the popup menu on the test case result row, and select "Re-execute test(s)". Copy the details of the error in the clipboard or generate a test campaign from the test in errors.

Figure 13: Re-execute test(s) pop-up

© 2009 QSpin SA

6.2. QTaste Main window

When the QTaste GUI application is started, the main window is displayed (Figure 14: Main QTaste User Interface window).
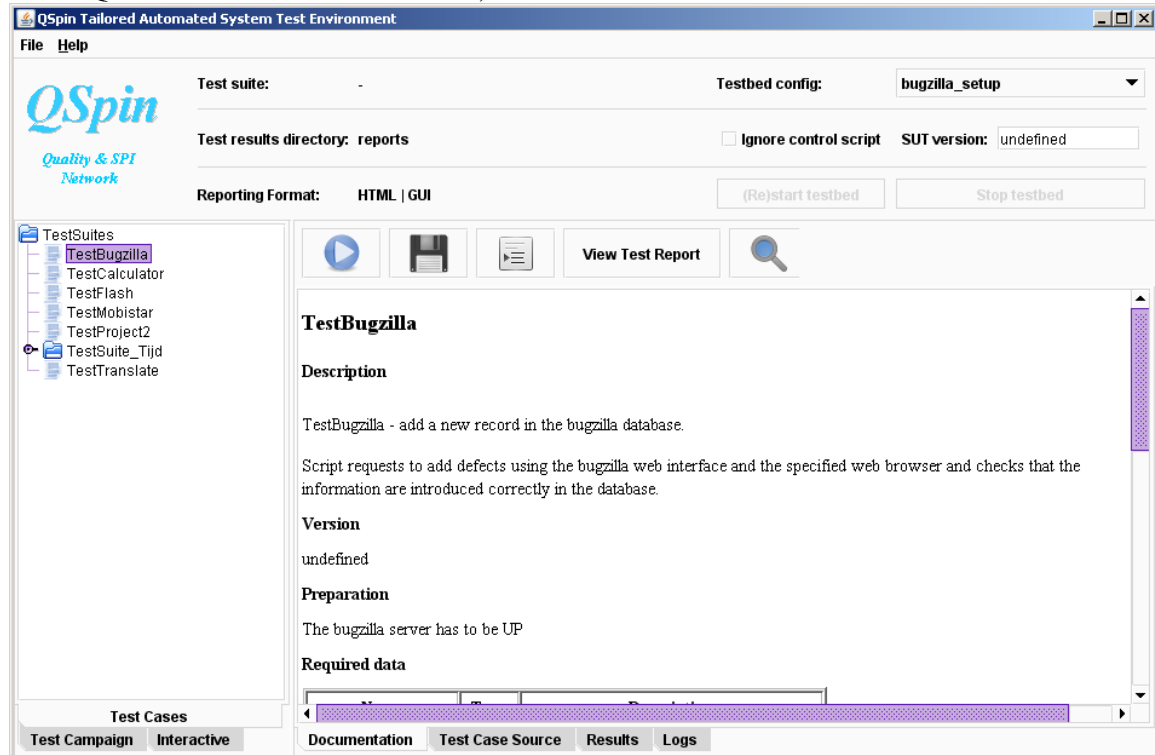
Figure 14: Main QTaste User Interface window

This main window is split into the following panels:

- Menu items
- Information panel: this panel located at the upper side is designed to display useful information about the current configuration of the QTaste software
- Selection panel: this panel is designated to select an action to be performed by QTaste. This panel is subdivided into three different tabs:
  - Test case tree view: this view displays the list of available test suites that can be executed by QTaste
  - Test Campaign view: from this window, test cases can be added or removed to a test campaign
  - Interactive view: this view allows the tester to execute QTaste verbs independently of a test script in order to validate its implementation or to observe the SUT behavior once the selected verb is called
- Main panel: this panel is the main view of the QTaste GUI. This view differs depending on the selected view from the selection panel.
  - Test case main panel: from this view the tester can view or edit test scripts, run a test suite, check the status of the test execution and also the logging generated by QTaste and its component supporting this feature.
  - Test Campaign editor panel: from this view the tester can define or modify test campaign. The result of this edition will result of a xml file that can be

used by the test campaign (see Starting the Meta-Campaign launcher); From this panel it is also possible to start a selected test campaign.
- o Interactive panel: from this panel status and results of the calls done using QTaste interactive mode is displayed

6.2.1.1. Menu items

Figure 15: Help menu content

6.2.1.2. Help menu

Two menu items are available:
- - About: display the QTaste information
- - User Manual: open Internet browser to this document in HTML version.

6.2.2. Information panel

Figure 16: Information panel window

This panel displays the following information:
- Test suite currently set
- Test result directory location (relative to QTaste main path);
- Reporting format used (separated by "|" if different reports format generated by the test engine);
- Testbed config: indicates the current testbed selection. When no testsuite is being executed, it is possible to change the testbed using the combo box.
- Ignore control script: by selecting this check box, when test engine is started, the control script defined in the testbed configuration is ignored.
- Restart testbed button: from this button, it is possible to start the testbed using the control script as it is defined in the testbed configuration file. This button is disabled when "Ignore control script" is not selected or when a test is being executed.
- Stop testbed button: from this button, it is possible to stop the testbed using the control script as it is defined in the testbed configuration file. This button is disabled when "Ignore control script" is not selected or when a test is being executed.

- SUT version is used to set the version of the "System Under Test". This will be present in the generated reports.

6.2.3. Selection panel

This selection panel displays 3 different tabs depending on the desired action:
- "Test case selection view" to perform actions linked to test case edition and execution
- "Test Campaign view" to define and edit test campaigns
- "Interactive view" in order to execute tests using the interactive mode of QTaste

Figure 17: selection panel

6.2.3.1. Test case selection view

From this window, the tester can navigate through the different test suites. The tree view displays directories found in the main directory TestSuites. All directories containing a TestScript.py file will be displayed as a test script with the name of the directory where the script is found.
In case no associated TestData.csv file is defined or in case the csv file is empty, an error icon is displayed on the test script. See **Error! Reference source not found.**
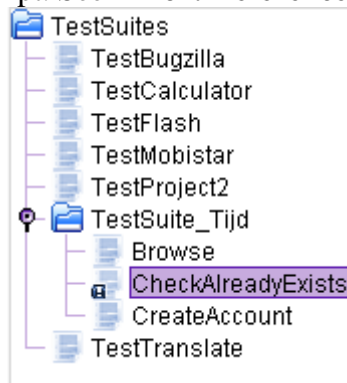
Figure 18: Test script error indicator

From this navigation tree, following mouse actions are possible:
- Left mouse click: selection of a test script or a test suite
- Popup mouse click (triggered by right click on Windows and Unix): this action invokes a popup menu (Figure 19: Test script popup window) from which it is possible to:
  - Run the selected script
  - Run the selected script in loop mode (can run the same script in loop)
  - Debug the script (run in debug mode)
  - Generate documentation: selecting this option will force the generation of the test documentation.
  - Edit the testscript in an external editor.
  - Open the testscript directory using a file browser.

- o Copy test script (only available when a test script is selected): the tester can then enter the name of the name of the destination test script. This will copy the selected script to the same parent directory of the source; This action copies also the testdata.csv file.
- o Create new test script (only available when other than test script is selected): this will define a new script into the selected directory. This action generates also an empty testdata.csv file.
- o Rename the testscript
- o Remove the testscript

Figure 19: Test script popup window

6.2.3.2.   Test Campaign view

This view displays the list of test cases (as done in the Test case view) from which the user can drag/drop test cases or test suites into a test campaign.
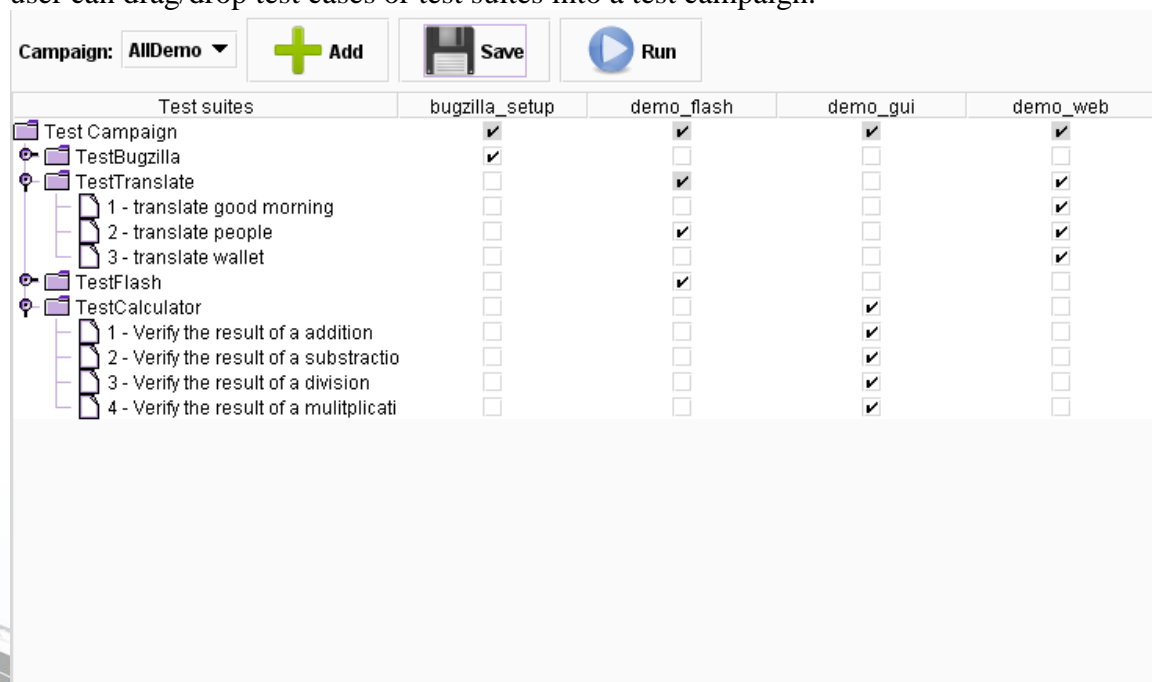
Figure 20: Test Campaign main view panel

6.2.3.3.   Interactive view

In this view, the list of QTaste components are displayed with all accessible verbs associated to it. At any time, the user can invoke a method using this mode. This mode supposes that the SUT is already started.

Figure 21: Interactive main panel

From the selection view, the user can select the component from which the verb can be called (only components defined in the selected testbed are displayed).
To start the interactive mode, click on the button "Start Interactive mode": this action will re-direct the log4j logs to the interactive panel.
To invoke a verb, just double-click on it if there is no parameter. Otherwise, a single-click will display the python call associated to this verb with the parameter types as arguments, change them to the actual argument and click on the "Send" button to invoke the verb.
The result of the call is displayed in the column "Result".
The test data can be passed through the Test Data editor displayed in this panel.
Please notice that some variable may be required in order to execute the command interactively (i.e.: INSTANCE_ID, etc.).

6.2.4. Main panel

This panel has the goal to display the following information:
- Status of a test case run;
- Display of log4j logs;
- Display and edit test scripts and test data files;
- Display the test script documentation

© 2009   QSpin SA

6.2.4.1. Test Case documentation

The documentation of a test case is automatically displayed when a test case is selected from the selection panel (see also [1]).
This documentation is automatically generated when modification on the TestScript.py has been identified. At any time, the use can force the generation of the documentation through the pop-up window accessible from the selection panel.

6.2.4.2. Test Case source editor

This editor is displayed when the user selected a test case from the selection panel and the tab "Test Case source" is selected [2].
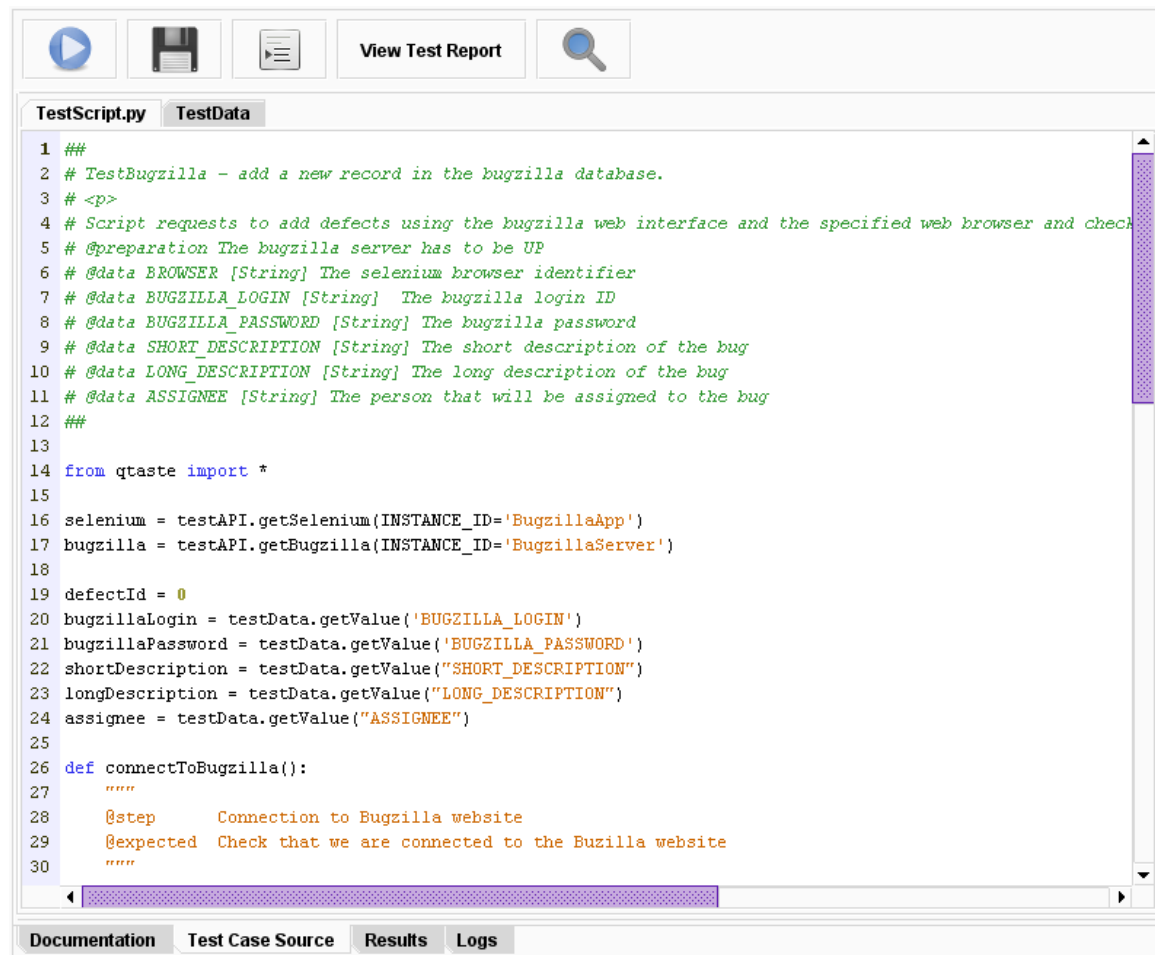
Figure 22: Test case source window

---

[1] From "gui.xml" file located in "/conf " directory you can select which tab is displayed when a test case is selected by adding the following line:
< test_case_tab_on_select >*tabName*</ test_case_tab_on_select >, where *tabName* is "doc", "source", "results", "logs" or "none" (to avoid switching tab)

[2] From "gui.xml" file located in "/conf " directory you can select which tab is displayed when a test case is selected by adding the following line:
< test_case_tab_on_select >*tabName*</ test_case_tab_on_select >, where *tabName* is "doc", "source", "results", "logs" or "none" (to avoid switching tab)

When the content of the file is modified the "*" is displayed to inform the user that modifications occurred in the file.
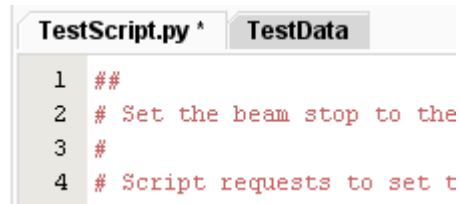
Figure 23: Test script not saved

To save the modifications (all opened files), "CTRL-S" can be used or just click on the save button.
Following popup menu is opened on the testcase source editor:

Figure 24: Test editor pop-up

If the user decides to run the test case being modified, the opened files are automatically saved.
If the user selects another test case from the selection panel while files are modified without save, a confirmation dialog window is displayed asking the user if he want to save first the files.

Test data (csv file) can also be edited by selecting "test data".

Figure 25: Test Data editor

A contextual menu is available in order to:
- Add a variable: this will add a column into the csv.
- Rename variable: this will rename the currently selected variable.
- Remove a variable: this will remove a column from the csv.
- Add row: this will define a new test case.
- Insert row: this will insert a row in the currently selected position.
- Duplicate row: this will copy the currently selected row.
- Remove row: this will remove a test case.
- Save the changes into the csv file.

Figure 26: Contextual menu of the Test data editor

6.2.4.3. Test Case Result

This panel is automatically displayed when a run has been launched. This panel is divided into 4 parts:
- Test run summary
- Test case error identification
- Test case error stack trace
- Test case logs

© 2009 QSpin SA

6.2.4.4. Test Case logs

This panel displays the Log4J data as it also stored in the log file (*<test_specific_dir>/log/QTaste.log*, where *<test_specific_dir>* is the test specific directory specified in section 4.2).

From this panel, the logs from LOG4J clients are also displayed: this feature is only available if the client application has been configured to be connected to QTaste log4J server.

Time	Level	Source	@	Step	Message
10:03:59	INFO	QTASTE	com.qspin.qtaste.testsuite.impl.DirectoryTestSuite		Adding test case TestSuites\TestCalculator\TestScript.py TestData.csv
10:03:59	INFO	QTASTE	com.qspin.qtaste.kernel.engine.TestEngine		Stopping SUT using command 'Testbeds/ControlScripts/control_windows.py stop '
10:04:00	INFO	QTASTE	com.qspin.qtaste.kernel.engine.TestEngine		SUT stopped
10:04:00	INFO	QTASTE	com.qspin.qtaste.kernel.engine.TestEngine		Starting SUT using command 'Testbeds/ControlScripts/control_windows.py start '
10:04:07	INFO	QTASTE	com.qspin.qtaste.kernel.engine.TestEngine		SUT started
10:04:07	INFO	QTASTE	com.qspin.qtaste.kernel.engine.TestEngine		Starting TestEngine
10:04:07	INFO	QTASTE	com.qspin.qtaste.datacollection.collection.ProbeManager		Starting ProbeManager:
10:04:07	INFO	QTASTE	com.qspin.qtaste.testsuite.TestScript		Executing test script: TestCalculator (row 1)
10:04:07	INFO	QTASTE	com.qspin.qtaste.testsuite.TestScript		Not using test timeout because running in debug mode

Level: ☐ Trace ☐ Debug ☑ Info ☑ Warn ☑ Error ☑ Fatal ☑ Test case or verb
Source: ☑ QTASTE

Figure 27: Log4J panel

Following information is displayed:
- Time: time of the log
- Level: log4J level (Trace, Debug, Info, Warn, Error or Fatal)
- Source: QTaste for all logs generated by the QTaste application, otherwise the name of the application as specified in the QTaste appender of the client application (log4j.appender.QTaste.application)
- @: The name of the logger
- Step: step of the current test (when applicable)
- Message: log4j message

It is possible to filter messages using the check boxes. It is possible to filter log4j messages based on its level and the application generating the message.

6.2.4.5. Test case execution navigation buttons

At upper side of the main panel buttons are displayed to give "quick" access to some actions.
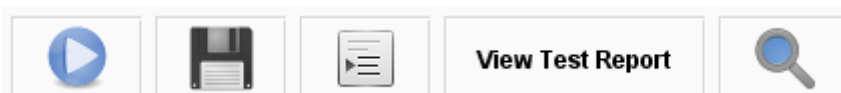
Figure 28: Navigation buttons

- Execute button: by clicking on this button, the current selected test script or test suite is executed
- Save button: by clicking on this button the current test script selected or test data is saved (and compiled through Python compiler)
- Debug button: by clicking on this button, the execution of the test script is launched in debug mode (see QTaste debugging mode)
- "View Test Report": use this button top open the last generated (or generating) test report in your internet browser.
- Show test API button: use this button to open the main page in your internet browser of the QTaste test API documentation
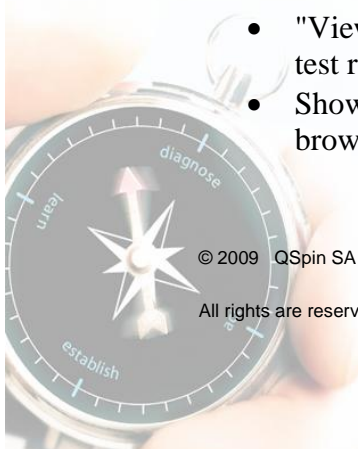
- Stop button (displayed only when a test case is being executed): abort the test case execution. In that case the status of the test is set to "Not Available" with the reason "Test aborted by the user"

Figure 29: Stop button

6.3. QTaste debugging mode

6.3.1. Setting breakpoint

To set a breakpoint, just open the test script file and click on the line number panel where the script must be stopped.
Remark: breakpoint can only be set at the TestScript.py file.
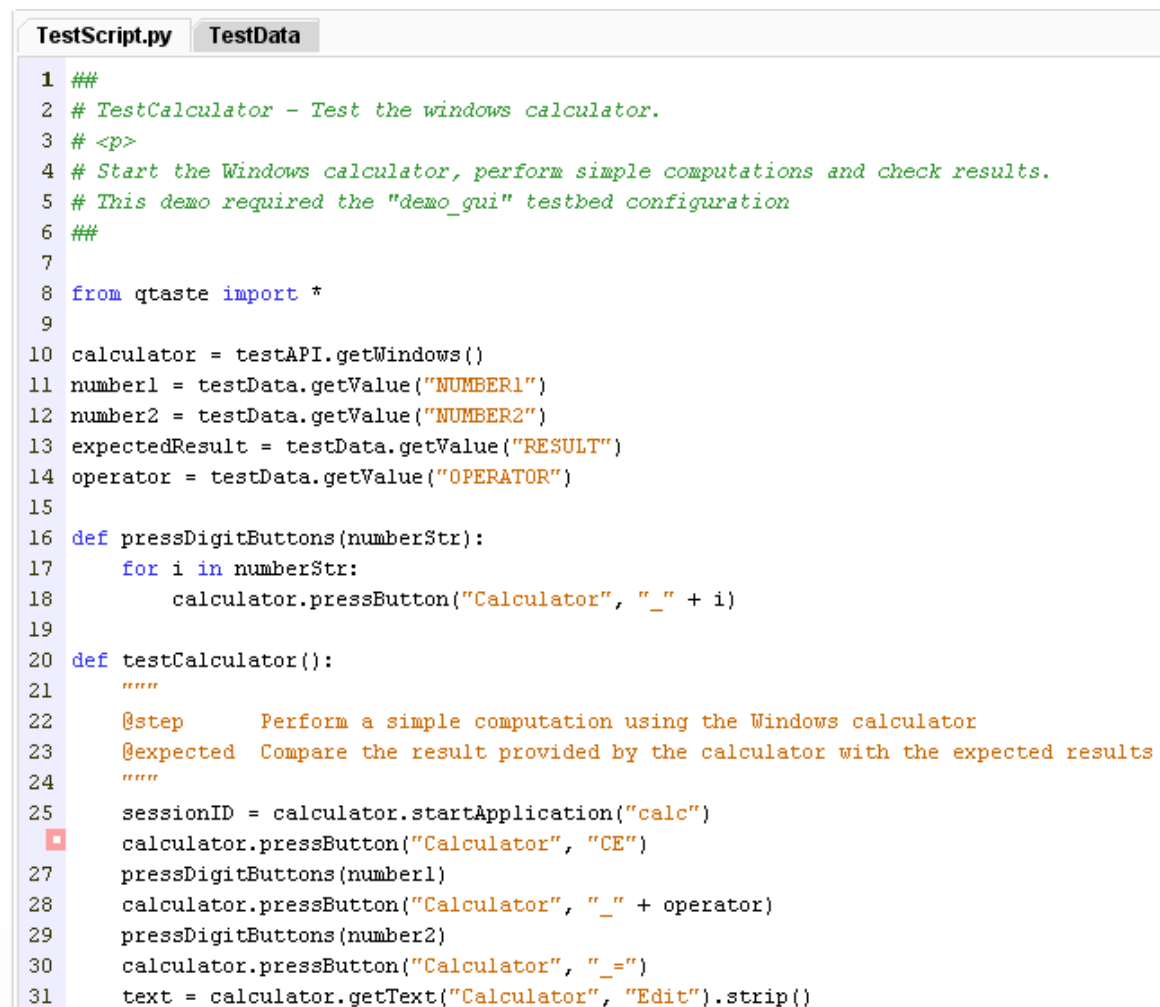
Figure 30: Setting a breakpoint

6.3.2. Script breakpoint

When a test is executed in debug mode (click on"Debug test" button) the script is stopped at the selected breakpoint.

Automatically, the QTaste user Interface displays the Test Case source window by highlighting the current execution line with the current python variable values.
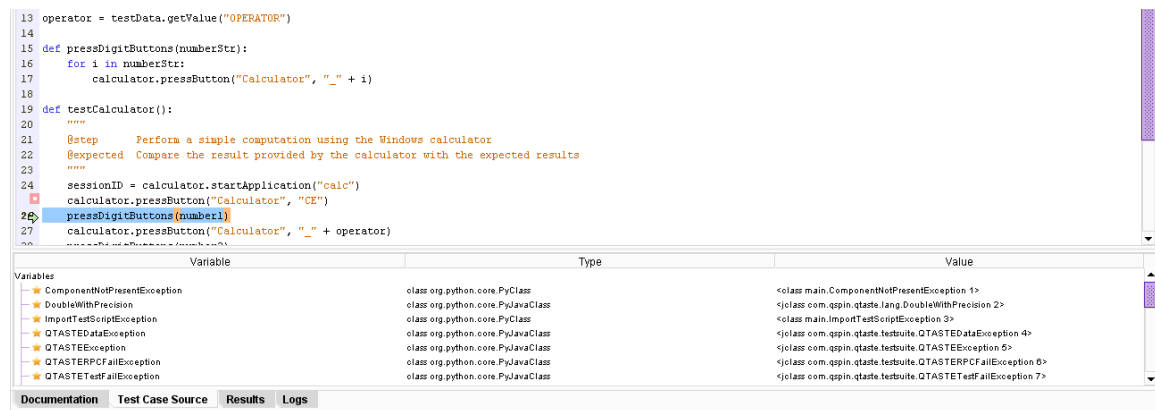
Figure 31: Debugging variables

6.3.3. Script break actions

The navigation buttons are updated to add buttons enable to continue the test script execution. Three actions are possible when a test script is break:

- Continue: continue the script until next breakpoint is reached
- Step over: Step over the test script execution
- Step into: Step into the test script execution
- Stop: abort the test script execution

Figure 32: Script break actions

7. QTASTE BUILD PROCEDURES

7.1. QTaste and testapi demo compilation pre-requisites

- Maven 2 has to be installed and available in the PATH environment variable (recommended version 2.2.1 or higher) can be downloaded from http://maven.apache.org/download.html

- Some libraries required by QTaste are not available on a public maven repository, so the qtaste_mvn_missing_dependencies.zip has to be uncompressed in the maven2 repository directory (i.e: ~/.m2/repository on linux). (Available on SourceForge http://sourceforge.net/projects/qtaste/files/)

7.2. QTaste compilation procedure

The QTaste framework is composed of the **kernel** and the **test API**.

The kernel is generic while the test API depends on the interfaces of the SUT.

For that reason, it is quite important to be able to recompile the test API especially if these interfaces are not stabilized.

The QTaste framework is using maven "pom.xml" file to be built. This procedure does not take into account the installation of the SUT as this step are specific to the system to be tested.

Retrieve the QTaste source code from source forge available on http://sourceforge.net/projects/qtaste/

7.2.1. Kernel compilation

To compile and package the QTaste kernel, use the jython packager script named packager_qtaste_Kernel.py, in <QTASTE_SOURCE_ROOT>. When executed it creates a zip archive called qtaste-kernel_X.Y.zip" where X.Y is the qtaste/kernel maven project version.

To compile the QTaste kernel, without packaging it, in order to use it in-place, use the following procedure:

- Go into the <QTASTE_SOURCE_ROOT>/kernel
- Launch the build.cmd/sh script
- The file "kernel/target/qtaste-kernel.jar" will be created and installed in the local maven repository
- The file "kernel/target/qtaste-kernel-deploy.jar" will be created

If dependencies problems appear while compiling, please check that you uncompress the "qtaste_mvn_missing_dependencies.zip" in your maven repository directory.

© 2009   QSpin SA
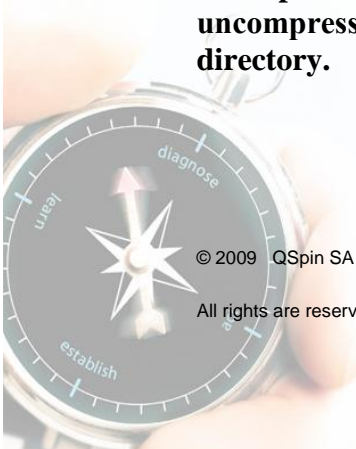
7.2.2. Test API compilation

To compile a Demo test API project, use the following procedure:

- Go into the <QTASTE_SOURCE_ROOT>

- Type the command "mvn install" in order to compile the testapi parent and the toolbox module

- Go into the <TEST_SOURCE_ROOT> (<QTASTE_SOURCE_ROOT>/demo)

- Launch the build.cmd/sh script

- The file "demo/testapi/target/qtaste-testapi.jar" will be created and installed in the local maven repository

- The file "demo/testapi/target/qtaste-testapi-deploy.jar" will be created and the test api documentation will be generated in the testapi/target/TestAPI-doc directory.

Please notice that, as explained above, the Test API depends on the SUT. If the Test API doesn't compile, it is probably due to a change in the interfaces of the SUT. If it happens, you have to adapt the code of the test API to make it compile with the new SUT interfaces.

7.3. CSV file generation from Excel

The initial test data is an Excel file having the name "TestData.xls".
To generate the CSV file really used by QTaste, the QTaste Excel macro can be used (see (Optional) Installation of the Excel Macro).
Using the "CTRL-S" key, the macro generates the CSV file according to the content of the Excel sheet. The macro is mainly responsible to use the correct data separator and convert numerical decimal value using a correct decimal separator into the CSV file.

8. GUIDELINES APPLICABLE FOR THE TEST DESIGNER

8.1. Test Scripts

8.1.1. Guidelines and requirements to write test cases

Test designer and tester shall never assume that a verb will be executed and return the expected result.
Test designer shall use checks to compare your expectation and the actual outcome.
Ideally, each verb should be validated by at least one check operation; this check operation is either another verb or a check done at the script level.
First step is to validate your script before trusting its results.

8.2. Test Data

Test data are stored in a CSV file and can be edited either from the QTaste GUI (see Figure 25: Test Data editor) or in a Microsoft Excel sheet and saved back to a CSV file. An Excel macro can be used to ease the conversion of the Test Data into a CSV file (see CSV file generation from Excel).

The first line of the file corresponds to the names of the test data and other lines correspond to test data values for a run of the test script.
Test designer should try to find a meaningful name for test data and shall assure that all the test data have to be defined before the test is executed.

There are some special test data associated with the QTaste test engine:

Test data name	Comment
COMMENT	This free text variable is used by the Test Designer to comment the values inserted in the line and their impact on the test at run time. This information will be used during the test report generation.
TIMEOUT	This variable is used as a maximum time expressed in seconds for the test execution of one test row. By the default the value is 60 seconds. This value can also be set at the beginning of the script (in order to set the value for all test rows) or in the testdata file is value can differ from one test row to another.
FILE_*	Column names starting with "FILE_" are automatically loaded by the Test Engine and their content can be retrieved by the scripts using the TestData (can be useful for data that need to be used on remote hosts).

8.2.1. Test data usage.

The test data file is created in a CSV file named *TestData.csv* in the same directory as the test script itself. Following table provides an example of the content of this file.

#	COMMENT	WORD	TRANSLATION	BROWSER
1	translate good morning	good morning	bonjour	*iexplore
2	translate people	people	blabla	*iexplore
3	translate wallet	wallet	pochette	*firefox

Figure 33: QTaste "Test data usage"

As already explained, the COMMENT test data is dedicated to define a free text comment associated to the row.

The next columns contain test data values to be used by the test script.

The test data can be changed inside the script itself using the python syntax (see Test data).

© 2009 QSpin SA

9. GUIDELINES APPLICABLE FOR THE DEVELOPER

9.1. Introduction

This section describes the guidelines to be followed by the developer to develop or modify a component of the test API. The development of the test API consists of the following activities:

- Define new test API components if necessary.
- Define new QTaste verbs applicable for a defined test API component.

9.2. Test API

9.2.1. Test API Components

Each component will be composed of:

- a java interface defining the verbs available for the component, extending *SingletonComponent* or *MultipleInstancesComponent* interface from the *com.qspin.qtaste.kernel.testapi* package;
- java classes implementing the previous interface, we recommend to use the following convention for package names. *com.<company>.qtaste.testapi.impl*.

Verbs are simply java methods with no specific signature.

For a component that must have only one instance, the java interface must extend the *SingletonComponent* interface and the matching implementing classes must have a constructor with no parameter.

For a multiple instances component that must be identified by an id, the java interface must extend the *MultipleInstancesComponent* and the matching implementing classes must have a constructor with one id *String* parameter, the instance id and implement the *getInstanceId* method:

> *public <Component>(String instanceId) { ... }*
> *public String getInstanceId() { ...}*

All component implementation classes must implement the *initialize()* and *terminate()* methods of the *Component* interface, which will be called on all components respectively before and after a test is executed. It can be e.g. used to connect to and disconnect from the SUT.

9.2.2. Exception handling

Test API should be compliant with the QTaste Exception handling policy in order to publish the appropriate test results.
A verb may stop a test execution and set the test result to "failed" by throwing an *QTasteTestFailException*, or to not available by throwing any other exception.

9.2.3. Usage of TCOM

The test API implementation has the goal to communicate to the SUT. The way to communicate to the SUT depends on the kind of communication available.

For this purpose, TCOM classes are provided and can be used by the developer.

For example if you need to define a new JMX client to initiate the connection with the SUT using JMX, you can use the class *"com.qspin.qtaste.tcom.jmx.impl.JMXClient"*.

9.2.4. Test API documentation

The test API can be documented using Javadoc. The documentation is generated through the provided doclet *TestAPIDoclet*, which is automatically run when compiling the testapi project (using maven). In order to get a complete documentation following doclets have been defined:

- Only interfaces in the *"com.qspin.qtaste.testapi.api"* package, extending (indirectly) the *com.qspin.qtaste.kernel.testapi.TestAPIComponent* interface are documented
- The specific configuration of the component is specified in the interface documentation, using @config tags; the syntax is "*@config CONFIG_NAME [Type] Description*", where *Type* is *Integer*, *Double*, *Boolean* or *String*
- Verbs are documented using standard Javadoc tags *@param*, *@return* and *@throws*

Here below an example of the documentation:

```
/**
 * Compare the content of the specified defectID record with the expected value provided
as parameter
 * @param defectID the identified of the record
 * @param shortDescription the title of the defect
 * @param longDescription the long description of the defect
 * @param assignee the identifier of the person assigned to this defect
 * @throws QTasteTestFailException If the content of the DB doesn't correspond to the
specified values
 * @throws Exception Throw an exception in case of database connection errors
 */ String assignee) throws QTasteTestFailException, Exception;
```

From this example, following documentation will be generated:

checkDatabase

```
void checkDatabase(int defectId,
                   String shortDescription,
                   String longDescription,
                   String assignee)
       throws QTASTETestFailException,
                   Exception
```

Compare the content of the specified defectID record with the expected value provided as parameter

Parameters:

    defectId - the identified of the record

    shortDescription - the title of the defect

    longDescription - the long description of the defect

    assignee - the identifier of the person assigned to this defect

Throws:

    QTASTETestFailException - If the content of the DB doesn't correspond to the specified values

    Exception - Throw an exception in case of database connection errors

Figure 34: QTaste "Example of generated Test API documentation"

9.3. Logging levels

We defined some "recommendations" for the usage of logs severity levels:

Level of logs	Used to
FATAL	Used to report unexpected and unrecoverable errors. (no guarantee that the Test Engine execution will behave correctly)
ERROR	Unexpected error but the Test Engine can continue to run normally
WARN	For example, to indicate that the test scenario is non-nominal.
INFO	Used to log what the Test Engine has done
DEBUG	Used for debugging purpose while using the test engine.
TRACE	Used for debugging purpose during development phases of components.

10. CONFIGURATION OF THE QTASTE FRAMEWORK

10.1. General remarks about XML configuration files

- The configuration files of the QTaste framework are stored in XML files located under the "conf" directory. The tag names used in the XML files are **case sensitive.**

```
<smartsockets></smartsockets>     is valid
<SmartSockets></SmartSockets>    is not valid
```

- Properties can be used as "variable" to avoid repetition of configuration items.

```
<sut_host>myhost</sut_host>          to define a variable sut_host
<host1>${sut_host}</host1>          use the value of the variable sut_host
<host2>$(sut_host)</host2>
<host3>AnotherHostName</host3>
```

- Entity can be added to avoid repetition of settings or block of configuration.

The following configuration file shows a block of configuration stored in the file called "component_web.inc".

```
<!-- defines the Selenium component -->
<!-- >${selenium_host} must have been previously defined -->
<Selenium id="TranslateApp">
     <host>${selenium_host}</host>
     <port>4444</port>
     <url>http://babelfish.yahoo.com</url>
</Selenium>
```

The following configuration file will reused the configuration block defined in the file "component_web.inc".

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE configuration [<!ENTITY components_web SYSTEM
"component_web.inc">]>
<configuration>
     <multiple_instances_components>
       The component of the irradiation controller will be copied here!!!
       &component_web;
     </multiple_instances_components>
</testbed_configuration>
```

- XML elements are accessed by the QTaste kernel and Test API components. They can access nested element using a dot notation. This notation will be used in order to improve the readability of the configuration elements.

```
<smartsockets>
      <host>myHost</host>
</smartsockets>
```

will be presented as "smartsockets.host" in the description tables.

10.2. Test engine configuration

Test engine configuration file allow the tester to:
- Enable/disable test retry when test fails
- Control the test report output formats
- Change parameters for the test reports or test campaigns doc
- Select the port for the log4j server (default is 4446), to which SUT processes can connect using a SocketAppender

```xml
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<configuration>
    <retry_test_on_fail>true</retry_test_on_fail>

    <reporting>
        <!-- Reporting format  (XML or HTML, default to HTML) -->
        <reporters>
            <format>HTML</format>
            <format>GUI</format>
        </reporters>
        <!-- Location of the HTML template -->
        <html_template>conf/reporting/html/iba</html_template>
        <html_settings>
            <generate_test_data>true</generate_test_data>
            <generate_steps_rows>true</generate_steps_rows>
            <report_stop_start_sut>true</report_stop_start_sut>
            <report_restart_sut>true</report_restart_sut>
        </html_settings>
        <!-- Location of the XML template -->
        <xml_template>conf/reporting/xml/standard</xml_template>
        <!-- Location of the generated reports -->
        <generated_report_path>reports</generated_report_path>
        <!-- Test campaign aggregated documentation parameters -->
        <test_campaign_doc>
            <remove_step_name_column>true</remove_step_name_column>
            <add_step_result_column>true</add_step_result_column>
            <duplicate_steps_per_test_data_row>true</duplicate_steps_per_test_data_row>
        </test_campaign_doc>
```

```
    </reporting>

    <log4j_server>
        <port>4446</port>
    </log4j_server>
</configuration>
```

Figure 35: Test engine configuration file

10.3. Testbed configuration

All the information related to the testbed (like connection parameters, userid, password…) must be stored in the testbed configuration file which is located in the Testbeds subdirectory of the test specific directory (see Test specific directories).

It is recommended to create a new testbed configuration file with a meaningful name for each new testbed.

The following table explains the meaning of the xml tags used by the testbed configurations files.

Tag name	Presence	Meaning
control_script	OPTIONAL	This tag point to a script that will be used to start or stop the SUT once the test engine is started/stopped. This script will be also called to restart the SUT in case of a failure. The first argument of the control script will be start or stop, to start or stop the SUT respectively. The "TESTBED" environment variable, giving the testbed configuration filename, is available in the control script.
testapi_implementation	MANDATORY	This tag is used to group the list of component implementations to load for a specific testbed.
Testapi_implementation.import	MANDATORY	This tag is used to specify the java package to be used for testapi implementations. The testbed supports multiple values for this parameter.
singleton_components	OPTIONAL	Special tag to describe the list of components of type "singleton" available in the testbed. But not the multiple instances components!!! Only singleton component should be described here.
multiple_instances_components	OPTIONAL	Special tag to define the list of components of type "multiple instances". all the treatment

		rooms available in the testbed.
probe_manager	OPTIONAL	Special tag to define all the probes required to collect asynchronous events in the testbed.
probe_manager.**probe**	OPTIONAL	This tag is used to specify the fully qualified java class name implementing the Probe interface required to collect asynchronous events.

```xml
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<testbed_configuration>
    <testapi_implementation>
        <import>com.qspin.qtaste.testapi.impl.demo</import>
        <import>com.qspin.qtaste.testapi.impl.generic</import>
    </testapi_implementation>
    <control_script>control_selenium.py</control_script>
    <multiple_instances_components>
        <Selenium id="BugzillaApp">
            <host>localhost</host>
            <port>4444</port>
            <url>http://qtaste-bugzilla</url>
        </Selenium>
        <Bugzilla id="BugzillaServer">
            <jdbcURL>jdbc:mysql://QTaste-bugzilla:3306/bugzilla3</jdbcURL>
            <jdbcDriver>com.mysql.jdbc.Driver</jdbcDriver>
            <dbuser>Taste</dbuser>
            <dbpassword>whataG00dN4m3</dbpassword>
        </Bugzilla>
    </multiple_instances_components>

    <singleton_components>
        <!—no singleton component in this testbed>
    </singleton_components>
</testbed_configuration>
```

Figure 36: Example of Testbed configuration file

10.4. Control scripts.

10.4.1. Description

As mentioned in Testbed configuration, QTaste provides a section in the testbed configuration to stop or start the SUT using a control script. This control script has a mandatory arguments "start" or "stop" and must return by an exit code different than 0 when failed.

In the QTaste framework, it is possible to write those control scripts using some classes as provided by the tools/jython/lib/lib/control_script.py python module. This python module has the purpose to give an easier way to write new control scripts but also to be OS independent.

Please refer to control_script.py file for more details.

Here below example of control script stopping/starting the Virtual box and Selenium software:

```
from controlscript_addon import *

ControlScript([
   VirtualBox("Bugzilla debian server",
         nameOfVBoxImage="QTaste-bugzilla",
         ),
   JavaProcess("Selenium Server",
         mainClassOrJar="demo/selenium-server.jar",
         checkAfter=5)
])
```

The control script needs to initialize the class "ControlScript" with a list of actions to perform what is defined in an array.

The example here upper performs 2 actions:
- Start the VirtualBox virtual machine with the image called "QTaste-bugzilla"
- Launch a Selenium server using the JavaProcess

Here below the classes defined in controlscript.py that can be used to defined new control scripts:
- JavaProcess: used to launch a java process
- NativeProcess: used to launch a simple native process
- ReplaceInFiles: used to perform a "sed" action on files.
- Rsh: to perform rsh command on remote host
- RExec: to perform rexec command on remote host
- Rlogin: to perform Rlogin
- RebootRLogin: based on Rlogin and has the goal to reboot using RLogin a remote VME host

10.4.2. Some examples

10.4.2.1. JavaProcess

```
JavaProcess("Selenium Server",
         mainClassOrJar="demo/selenium-server.jar",
         args="-specialOption option1",
         workingDir=sutDir+"/Selenium",
         checkAfter=5)
```

This example will start a java process using the jar file "demo/selenium-server.jar" with arguments specified in args variable. This java process is being launched having as working directory the variable "sutDir". The controls script will check if the java process still exits after 5 seconds.

10.4.2.2. NativeProcess

```
NativeProcess("Windows control native Agent",
```

```
    executable="python.exe",
    args="demo/pywinauto-0.3.8/XMLRPCServer.py",
    workingDir=sutDir,
    checkAfter=5)
```

This example will start a native process using the executable file "python.exe" with arguments specified in args variable. This native process is being launched having as working directory equals to the variable sutDir and the controls script will check if the process still exits after 5 seconds.

10.4.2.3. ReplaceInFiles class

```
  ReplaceInFiles(r"OLD_VALUE",
      "NEW_VALUE",
      sutDir+"/Application/config_with_simulators/config/conf_file.properties")
```

10.4.2.4. RExec class

```
RExec(startCommand="start.csh", stopCommand="stop.csh",
      host="myhost",
      use="user", password="userPassword")
```

10.4.2.5. RLogin class

```
RLogin(command="'sp cuStartup'",
          host=host, login="QTaste",
log4jconf="Testbeds/ControlScripts/x.log4j.properties")])
```

10.4.3. (Optional) Installation of the Excel Macro

In order to ease the generation of CSV files, a Microsoft Excel macro has been developed. This macro file is located in the "tools\TestData" directory of the QTaste framework. To enable it, open the file QTasteMacros.xls with Excel and save it as an add-in (select Microsoft Office Excel Add-in (*.xla) format, add-ins directory will be selected automatically), then go in Tools menu, select Add-Ins, check "QTastemacros" and click on "Ok".

This macro is generating a CSV file when you press "Ctrl-S" but the name of the xls file has to be called TestData.xls.

11. QTASTE SCRIPTING LANGUAGE GUIDE

11.1. General

QTaste scripts are written in Python language v2.2 and interpreted by Jython v2.2.1, (http://www.jython.org). Any Python feature supported by Jython v2.2.1 is thus supported by the QTaste.

The *pythonlib* subdirectories of all the directories from the test script directory up to the *TestSuites* directory are automatically added, in that order, to the Python path so that custom Python modules can easily be imported from those directories, using the Python `import` directive.

If some specific python modules have to be written, they can be added to the python class path just by adding the directory to the QTASTE_JYTHON_LIB environment variable. QTaste will automatically check this variable and use it if it's defined.

This allows for easy script reusability (see Module import).

11.1.1.　　　Test API components

Test API is accessed through components instances, which are obtained through the `testAPI` variable. This variable must be imported from the QTaste module e.g. using "`from qtaste import *`", via *get<Component>* methods. For singleton components, no argument is required. For multiple instances components, an `INSTANCE_ID` test data is required to define the instance that will be used in the CSV file, in the script via the `testData` variable (see Test data) or by passing it as a named argument to the *get<Component>* method. The returned value can be tested to check if the component is present in testbed.

Get components instances examples:
- `win = testAPI.getWindows()`
  Defines win as the singleton Windows instance
- `selenium = testAPI.getSelenium()`
  Defines selenium as the Selenium instance of the instance defined in the test data.
- `selenium = testAPI.getSelenium(INSTANCE_ID="TranslateApp")`
  Defines selenium as the Selenium instance identified by "TranslateApp"

Testing the value returned by the *get<Component>* methods makes it possible to make a script usable in different testbeds.

Example:
```
myStub = testAPI.getComponentStub()
if myStub:
    …
```

11.1.2.　　　Component verbs

Verbs are called directly on the components instances, using the required arguments.

If a verb has a return value, it is returned by the call.

Call examples:

- `selenium.click("link=New")`
  On `selenium`, calls `click()`, which has an identifier
- `text = calculator.getText("Calculator", "Edit")`
  On `calculator`, calls `getText()`, which has two parameters, and assigns its return value to the `text` variable

11.1.3. Stopping test execution

A method named `stopTest` is also available on `testAPI`, to stop the test execution and set its status to "failed" or "not available", with an associated message.

Call syntax: `testAPI.stopTest(status, message)`
  where `status` is `Status.FAIL` or `Status.NOT_AVAILABLE` (Status must be imported from qtaste module, e.g. using "`from qtaste import *`")
  and `message` is a string containing the message to report.

11.2. Test data

Test data are accessed through the `testData` variable, which must be imported from the QTaste module, e.g. using "`from qtaste import *`", on which the following methods are available:

- `String getValue(String name)`
  Returns the value of the test data with given name, as a string
- `int getIntValue(String name)`
  Returns the value of the test data with given name, as an integer
- `double getDoubleValue(String name)`
  Returns the value of the test data with given name, as a double
- `DoubleWithPrecision getDoubleWithPrecisionValue(String name)`
  Returns the value of the test data with given name, as a `DoubleWithPrecision`
- `boolean getBooleanValue(String name)`
  Returns the value of the test data with given name, as a boolean
- `byte[] getFileContentAsByteArray(String name)`
  Returns the file content of the test data file with given name (starting with "FILE_"), as a byte array
- `String getFileContentAsString(String name)`
  Returns the file content of the test data file with given name (starting with "FILE_"), as a string
- `void setValue(String name, String value)`
  Sets the value of the test data with given name to given value
- `void setIntValue(String name, int value)`
  Sets the value of the test data with given name to given int value
- `void setDoubleValue(String name, double value)`
  Sets the value of the test data with given name to given double value
- `void set DoubleWithPrecision Value(String name, DoubleWithPrecision value)`
  Sets the value of the test data with given name to given `DoubleWithPrecision` value
- `void setBooleanValue(String name, boolean value)`
  Sets the value of the test data with given name to given boolean value
- `void remove(String name)`
  Removes the test data with given name
- `boolean contains(String name)`
  Checks if test data with given name exists

Calling a get value method (`getValue`, `getIntValue`, `getDoubleValue`, `getDoubleWithPrecisionValue` or `getBooleanValue`) with an inexistent test data name terminates the script execution and the test result is set to "Not available" with a message explaining that the given data doesn't exist.

Calling a typed get value method (`getIntValue`, `getDoubleValue`, `getDoubleWithPrecisionValue` or `getBooleanValue`) terminates the script execution if the data value can't be converted to the specified type. The test result is then

set to "Not available" with a message explaining that the given data has not the correct format.

Calling `remove` with an inexistent test data name has no effect.

`DoubleWithPrecision` is a class implementing a double with a specified precision. To use the constructor, it must be imported from the QTaste module, e.g. using "`from qtaste import *`".

Note that to test equality of an object of this class with a double or integer, you must explicitly use the `equals()` method instead of the == or != operators.

11.3. Logging

Logging is done through any Apache Log4j Logger, which can be the `logger` variable, imported from the QTaste module, e.g. using "`from qtaste import *`" and is a Logger named *TestScript, or any Logger obtained using the log4j Logger class, e.g.*:

```
from org.apache.log4j import Logger
logger = Logger.getLogger("LoggerName")
```

Most useful available methods are:
- `logger.trace(Object message)`
  Logs a message object with the TRACE level
- `logger.debug(Object message)`
  Logs a message object with the DEBUG level
- `logger.info(Object message)`
  Logs a message object with the INFO level
- `logger.warn(Object message)`
  Logs a message object with the WARN level
- `logger.error(Object message)`
  Logs a message object with the ERROR level
- `logger.fatal(Object message)`
  Logs a message object with the FATAL level

The logger level is defined by the "`log4j.category.LoggerName`" property in the *log4j.properties* QTaste's configuration file. Valid values are: ALL (log all messages), TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF (doesn't log any message).
The log levels are ordered: TRACE < DEBUG < INFO < WARN < ERROR < FATAL. A log method actually logs a message only if the log level is higher or equal than the logger level.

11.4. QTaste exceptions

The following QTaste exceptions may be imported from the QTaste module, e.g. using "`from qtaste import *`:

Exception Name	Meaning
QTasteException	Base QTaste exception, from which all

	other QTaste exceptions are derived, resulting in a test "not available".
QTasteDataException	Exception thrown in case of invalid data or argument, resulting in a test "not available".
QTasteTestFailException	Exception thrown to make a test "fail".

By catching an expected exception in the script, you can write a non-nominal test. Here below is such an example.

```
try:
    myComponent.throwAnException()
except QTasteTestFailException:
    pass
```

11.5. QTaste script

Script documentation is generated through PythonDoc and an XSLT style-sheet.

11.5.1.   Test script description

The test script is described by adding a PythonDoc module comment at the beginning of the script, i.e. a comment starting and ending with a double sharp character.

The name of the test script is the name of its containing directory and, as for javadoc, the first sentence up to the first dot is the summary and the next sentences up to the first tag completes the description.

The following tags must or may be added:
- *@preparation* (optional): the description of the test preparation
- *@data* (optional): the data used by the test; the syntax is "*@data DATA_NAME [Type] Description*", where *Type* is *Integer*, *Double*, *DoubleWithPrecision*, *Boolean* or *String*. *DoubleWithPrecision* is a double with a specified precision and is represented by the following string format: "*doubleValue*(*precisionValue*)" or simply "*doubleValue*" for a precision of 0.

Here below an example of the documentation:
```
##
# TestBugzilla - add a new record in the bugzilla database.
# <p>
# Script requests to add defects using the bugzilla web interface and the specified web
browser and checks that the information are introduced correctly in the database.
# @preparation The bugzilla server has to be UP
# @data BROWSER [String] The selenium browser identifier
# @data BUGZILLA_LOGIN [String]  The bugzilla login ID
# @data BUGZILLA_PASSWORD [String] The bugzilla password
# @data SHORT DESCRIPTION [String] The short description of the bug
# @data LONG DESCRIPTION [String] The long description of the bug
# @data ASSIGNEE [String] The person that will be assigned to the bug
##
```

11.5.2.   Steps description

Each step should be defined by a Python function, and described by a docstring comment (i.e. text between triple double-quotes inside the function), containing a @*step* tag describing the step and an optional @*expected* tag describing the expected results.

Here below an example of steps description:

```
def disconnectFromBugzilla():
    """
    @step        Disconnect from bugzilla
    @expected  The session should be closed
    """
    selenium.click("link=Log out")
    selenium.waitForPageToLoad("30000")
    selenium.closeBrowser()
```

11.5.3.   Steps execution

Steps must be executed by calling the `doStep()` function or the `doSteps()` function, which must be imported from the QTaste module, e.g. using "`from qtaste import *`", rather than calling the step functions directly. This enables the QTaste to detect the beginning and end of steps, and compute the elapsed time.

The `doStep()` function, allows to execute one step, passing an optional step id and the step function as arguments. The step id is an integer or a string containing only [a-zA-Z0-9_] characters, if not given the step number is used, starting from 1 for the first step. The signature of the function is:

```
doStep([Integer_or_String stepId,] Function stepFunction)
```

Here below an example of steps execution using the `doStep()` function, without step id:

```
doStep(connectToBugzilla)
doStep(createRecord)
doStep(checkDatabaseRecord)
doStep(disconnectFromBugzilla)
```

Here below an example of steps execution using the `doStep()` function, with step id:

```
doStep(0, connectToBugzilla)
doStep(1, createRecord)
doStep('1a', checkDatabaseRecord)
doStep('2', disconnectFromBugzilla)
```

The `doSteps()` function, allows to execute a defined sequence of steps, passing a steps sequence and an optional steps selector as arguments. The steps sequence is a list or tuple of (`stepId, stepFunction`) tuples, `stepId` being an integer or a string containing only [a-zA-Z0-9_] characters. The steps selector is a string using one of the following formats:

- `"[firstStepId-lastStepId]"` to execute steps from `firstStepId` to `lastStepId`;
- `"[firstStepId-]"` to execute steps from `firstStepId` to last step;
- `"[-lastStepId]"` to execute steps from first step to `lastStepId`;
- `"[stepId]"` to execute only step `stepId`.

If no steps selector is given, all steps of the sequence are executed.

The signature of the function is:

```
doSteps(Sequence stepsSequence [, String selector])
```

Using this function allows reusability of a steps sequence if it is defined in a common module (see Module import), because it can be imported and used in several test scripts.

Here below an example of steps execution using the `doSteps()` function:

```
bugzillaSteps = [
  (0, connectToBugzilla),
  (1, createRecord),
  (2, checkDatabaseRecord),
  ('2a', disconnectFromBugzilla),
]

doSteps(bugzillaSteps)
```

11.5.4. Documentation generation

First of all, note that the documentation generation scripts must be run from the test specific directory containing the directories specified in section 4.2.

To generate the documentation of a single test script, execute the command *generate-TestScript-doc.bat(or .sh)* with the test script file name as argument. This will generate an HTML file named *TestScript-doc.html* in the test script directory.

To generate the documentation of a test suite, execute the command *generate-TestSuite-doc.bat(or .sh)* with the test suite directory name as argument. This will generate:
- HTML files named *TestScript-doc.html* for each test scripts of the test suite, aside the test scripts;
- HTML files named *TestSuite-doc.html* (main file, frameset), *TestSuite-doc-list.html* (list of test scripts) and *TestSuite-doc-summary.html* (summary of test scripts), in the test suite directory.

To generate the documentation of all test suites, execute the command *generate-TestSuites-doc.bat (or .sh)*.

To generate a Campaign test procedure document for a test campaign:

- execute the command *generate-TestCampaign-doc.bat (or .sh)* with the test campaign description XML file path as argument. This will generate an HTML file name *<campaignName>-doc.html* in the TestCampaigns directory;

11.6. Scripts reusability

11.6.1. Test script import

A test script can be reused by another script thanks to the `importTestScript()`, `isInTestScriptImport()`, `doSubStep()` and `doSubSteps()` functions, which must be imported from the QTaste module, e.g. using "`from qtaste import *`".

The `importTestScript()` function allows to import a test script in order to reuse its steps using the `doSubStep()` and `doSubSteps()` functions.
```
void importTestScript(String testScriptPath)
```

Imports the test script located in the `testScriptPath` directory and make its symbols available, including step functions and steps sequences, in a namespace named as the test case name, i.e. the name of the final directory of the test script.

Actually, the code of the test script is evaluated except for the `doStep` and `doSteps` functions which are automatically skipped to avoid executing the steps during the import.

`testScriptPath` - path to the directory of the test script to import. This path is relative to the parent directory of the current test script.
It is advised to used forward slashes "/" as directory separator instead of backslashes "\" to make path platform independent.

The `isInTestScriptImport()` function allows to check if code is executed during a test script import, i.e. during the execution of the `importTestScript()` function. It is useful to prevent execution of portion of code during a test script import.

```
Boolean isInTestScriptImport()
```
Returns `True` if code is executed during a test script import, `False` otherwise.

The `doSubStep()` function allows to execute an imported test step as a sub-step of the current step. It has the same signature as the `doStep()` function (see Steps execution).

The `doSubSteps()` function allows to execute an imported sequence of test steps as sub-steps as a sub-steps of the current step. It has the same signature as the `doSteps()` function (see Steps execution).

For example, the test script of the test case *XYZ_01* reuse the test scripts of the test cases *XYZ_01_02* and *1_PreparationPhase/2_BDS/PBSV2_BDS_PREP_01*.

Test script *XYZ_01*:

```
##
# Test Title - Description - Test case XYZ_01: Main flow.
# …
##

…

# set the MY_VARIABLE test data to False, because it is used by the XYZ_02 test script
but not declared in this test script
testData.setBooleanValue("MY VARIABLE", False)
importTestScript("../1 ABC/XYZ 01 02")
importTestScript("../2 DEF/XYZ 01")

…

def step1():
    …

def step2():
    """
    @step Based on the test case XYZ 01: Main flow perform the nominal case.
    """
    doSubSteps(XYZ_01_02.mainFlow)
```

```
def step3():
    """
    @step Based on the test case XYZ 01: Main flow perform the nominal case (steps 1 and 3
to 5).
    """
    doSubStep(XYZ_01.step1)
    doSubStep(XYZ_01.step3)
    doSubStep(XYZ_01.step4)

# main flow steps, reusable in another test scripts
# doesn't include step1 which is a initial step
# nor step5 which is a cleanup step
mainFlow=[(2, step2),
          (3, step3),
          (4, step4)]

resetSystem()
doStep(step1)
doSteps(mainFlow)
doStep(step5)
```

Some function may use the `isInTestScriptImport()` function to avoid executing the a function when a script is imported:

```
def function():
    if isInTestScriptImport():
        # test script is imported, so return immediately
        return
    …
```

11.6.2. Module import

Test scripts can import common Python modules, allowing reusing the same code from several test scripts without duplicating it.

For that reason, common modules have been created in the *TestSuites/pythonlib* directory. These common steps or common checks are implemented as classes and can therefore be derived and methods can be over-written for tests that need to modify the behaviour of some steps or to add new ones.

For example, the `ABC` module contains a `CommonSteps` class defined as follow:

```
class CommonSteps:

     def __init__(self, someValue):
         """
         @data SOME VALUE [Integer] treatment room id
         """
         …
     …

     def aFunction(self):
         """
         @step     Start the function
         @expected  Description of the expected result
         @data SOME_VARIABLE [Double] A variable
         """
         self.function5()
         self.function8()

     …
```

This class can be used to create the test steps, as in the following example:

```
import ABC

class ABCSteps(ABC.CommonSteps):
        def performThis(self):
            """
            New step, not present in ABC.CommonSteps
            @step      Request the system to …
            @expected  After 5 seconds, system should be …
            """
            self.someCommands(self.someParam)
            time.sleep(5)
            self.someOtherCommans.('prepare.state:pending', 0)
            self.checkSomething(0)

steps = ABCSteps()

def prepareTest():
    """
    @step      Prepare the test and its parameters
    @expected  …
    """
    steps.prepareTest()

…

…
doStep(prepareTest)
…
```

It is also possible to reuse a defined steps sequence, by defining the steps and steps sequence in a common module.