QMTest: User’s Guide and Reference

CodeSourcery, LLC

QMTest: User's Guide and Reference
by CodeSourcery, LLC
Copyright © 2002, 2003 CodeSourcery LLC

Table of Contents

Chapter 1. Introduction

QMTest is a testing tool. You can use QMTest to test a software application, such as a database, compiler, or web browser. You can even QMTest to test a physical system (like a valve or thermometer) if you have a way of connecting it to your computer.

Code that has not been tested adequately generally does not work. Yet, many applications are deployed without adequate testing, often with catastrophic results. It is much more costly to find defects at the end of the release cycle than at the beginning. By making it easy to develop tests, and execute those tests to validate the application, QMTest makes it easy to find problems easier, rather than later.

QMTest can be extended to handle any application domain and any test format. QMTest works with existing testsuites, no matter how they work or how they are stored. QMTest's open and pluggable architecture supports a wide variety of applications.

QMTest features both an intuitive graphical user interface and a conventional command-line interface. QMTest can run tests in serial, in parallel on a single machine, or across a farm of possibly heterogeneous machines.

CodeSourcery provides support for QMTest. CodeSourcery can help you install, integrate, and customize QMTest. For more information, visit the QMTest web site (http://www.qmtest.com).

Chapter 2. Getting Started with QMTest

QMTest is an general-purpose, cross-platform software testing tool. QMTest can be used to test compilers, databases, graphical user interfaces, or embedded systems. QMTest provides a convenient graphical user interface for creating, managing, and executing tests, provides support for parallel test execution, and can be extended in a variety of ways.

This chapter will show you how to use QMTest by example. You will learn how to use QMTest to create tests, run tests, and examine the results.

2.1. Setting Up

A test database is a directory that QMTest uses to store tests. If you want to create a new, empty test database from scratch, you use the **qmtest create-tdb** command, but for this tutorial, you should use the sample test database provided with QMTest. Since you'll modify the test database later in the tutorial, start by making a copy of it. Copy the entire test database directory tree to another location. If you've installed QMTest in the default location you can make a copy of the sample database by running this command on a UNIX system:

```
> cp -r /usr/share/qm/tutorial/test/tdb tdb
```

On a Windows system, use this command at a DOS [1] prompt:

```
> xcopy c:\Python23\qm\tutorial\test\tdb tdb\ /s
```

The exact paths to use depend on exactly how you have built and installed QMTest. The paths above are correct for the binary RPM and Windows packages distributed by CodeSourcery. If you build from the QMTest source distribution, the tutorial may be in another location, such as `/usr/share/qm`.

Then, enter the new directory you have created. On both UNIX and Windows systems, you can do this with this command:

```
> cd tdb
```

Make sure that QMTest is in your PATH so that the operating system can find it. On UNIX, you can use this command:

```
> PATH=/usr/bin:${PATH}; export PATH
```

in the Bourne shell. In the C shell, use:

```
> setenv PATH /usr/bin:${PATH}
```

On Windows, use:

```
> PATH C:\Python23\Scripts;%PATH%
```

If you are not using Python 2.3, replace `C:\Python23` with the directory containing your Python installation.

In order to avoid having to retype these commands every time you want to use QMTest, you can set up your system so that these commands are executed automatically when you log in. Consult your system's manuals to find out how to do this.

On Windows, every command in this manual that begins with **qmtest** should be read as if it starts with **qmtest.py**. For example, if this tutorial instructs you to type:

```
> qmtest run
```

you should instead type:

```
> qmtest.py run
```

on a Windows system.

2.2. Starting the Graphical Interface

To examine the tests in the test database, you can use QMTest's graphical user interface. To start the graphical user interface, use the **qmtest gui** command, like this:

```
> qmtest gui
```

You will see output similar to:

```
QMTest running at http://127.0.0.1:1158/test/dir
```

After a moment, a new web browser window will open, and you will see the QMTest graphical user interface (GUI). If a web browser window does not open, you will have to manually enter the URL that QMTest printed out (`http://127.0.0.1:1158/test/dir` in the example above) into your browser. Alterantively, you can edit your QM configuration file to tell QM how to invoke your browser and then start the GUI again.

As you can see, QMTest creates a graphical user interface using your web browser.

The page you see in your browser shows the contents of the test database. You can see that there are three tests in the database, named `exec0`, `exec1`, and `exec2`. You can always click on Directory under QMTest's View menu to see this display.

2.3. Running Tests

To run all the tests, choose All Tests from the Run menu. QMTest will display the test results page. As the tests run, this page will be automatically updated. (If you do not want to wait for QMTest to update the page, you can manually reload the page in your browser.) After a few moments, QMTest will display the test results for the three tests in the database. The upper part of the screen gives a statistical overview of the test results. Of the three tests, two passed. However, one test failed. You can use this statistical information to get a quick overview of your application's correctness.

In addition to showing you how many tests passed and how many failed, QMTest shows you whether how many tests had *unexpected* pass or fail outcomes. If you know that certain tests will fail, you can tell QMTest that they are expected to fail. Then, if you are testing a change to your application, you can easily see whether your change made things better or worse. As long as there are no unexpected failures, your change did not cause any problems.

If you have not explicitly set up an expectation for a test, QMTest assumes that the test is expected to pass. That is why QMTest indicated that there was one unexpected failure when you ran the tests.

Below the statistics section, QMTest displays detailed information about each test. In this case, you can see that the `exec1` test is the one that is failing. You can click on the Details link to get additional information about why a particular test is failing.

2.4. Setting Expectations

The easiest way to create expectations is to tell QMTest that you expect future results to be the same as the results you just obtained. Save the results of your test run by choosing Save Results from the File menu. QMTest will prompt you for a file to use to store the results of your test run. If you exit QMTest, you can reload this file to recover your test results without rerunning the tests.

You can also use this file to set QMTest's expectations. Choose Load Expectations from the File menu and provide the same file name that you used when you told QMTest when you asked it to save your results earlier.

Then, QMTest will redisplay the test results, but now you will see that there are no unexpected failures; the current results match your expectations.

You can also manually edit expectations. Click on the Expectation link next to a particular test to set the expectation. To set the expectation for `exec1` back to PASS, click on the Expectation link next to `exec1`, choose PASS, and click the OK button. You will see that now the `exec1` failure is unexpected again. You can save your expectations by choosing Save Expectations from the File menu.

2.5. Examining Tests

Click on the `exec1` label to examine the failing test. QMTest will display information about the test. The most important information about the test is its *test class*. This test is an instance of the `python.ExecTest` class. The test class indicates what kind of test `exec1` is. QMTest gives a brief description of the test class in the GUI; a `python.ExecTest` checks that a Python expression evaluates to true.

For more details about the test class, you can click on the Help link to the right of the description. QMTest will pop up a window that describes the test in more detail. In summary, a `python.ExecTest` executes some setup code. Then, a Python expression is evaluated. If the expression evaluates to true, the test passes; otherwise, it fails.

The setup code and the expression are the *arguments* to the test class. Every test class takes arguments; the arguments are what differentiate one instance of a test class from another. QMTest displays the arguments for the `exec1` test in the GUI. In this case, the sequence of statements is simply the single statement `x = 2`, which assigns `2` to the variable `x`. The expression is `x + x == 5`, which compares `x + x` with `5`. Since `x` is `2` in this case, this expression evaluates to false. That is why the test fails.

You can click on the Help link next to each argument to get more details about exactly what the argument is for.

2.6. Modifying and Creating Tests

To fix the test, you need to change the arguments to the test. Select Edit Test from the Edit menu. QMTest will display a form that allows you to change the arguments to the test.

Change the second argument, labeled "Python Expression," to `x + x == 4`. Then click on the OK button at the bottom of the page to save your changes. Choose This Test from the Run menu and observe that the test now passes.

Creating a new test works in a similar way. Choose Directory under the View menu to return to the main QMTest page. Then, select New Test from the File menu to create a new test. QMTest displays a form that contains two fields: the test name, and the test class. The test name identifies the test; the test class indicates what kind of test will be created.

Test names must be composed entirely of lowercase letters, numbers, the “_” character, and the “.” character. You can think of test names like file names. The “.” character takes the place of “/” on UNIX or “\” on Windows; it allows you to place a test in a particular *directory*. For example, the test name `a.b.c` names a test named `c` in the directory `a.b`. The directory `a.b` is a subdirectory of the directory `a`. By organizing your tests in directories, you will make it easier to keep track of your tests. In addition, QMTest can automatically run all the tests in a particular directory, so by using directories you will make it easy to run a group of related tests at once.

Enter `command.test1` for the test name. This will create a new test named `test1` in the `command` directory. Choose `command.ShellCommandTest` as the test class. This kind of test runs a command and compares its actual output against the expected output. If they match, the test passes. This test class is useful for testing many programs. Click on the Next button to continue.

Now, QMTest will present you with a form that looks just like the form you used to edit `exec1`, except that the arguments are different. The arguments are different because you’re creating a different kind of test. Enter `echo test` in the Command field. This command will produce an output (the word `test`), so find the Standard Output box and enter `test` in this box. When you are done, click the OK button at the bottom of the form.

Now you can select This Test from the Run menu to run the test.

When you’re done experimenting with QMTest, choose Exit from the File menu.

2.7. Using the Command-Line Interface

All of QMTest’s functionality is available from the command-line, as well as in the graphical user interface. When you invoke **qmtest** on the command line, you specify a command argument, which tells the program which action to perform. Some commands require additional options and arguments, which you place after the command. There are a few options that apply to all commands; to use these options place them before the command name. For example, in the command:

```
    > qmtest -D . run -f full exec1
```

the `-D .` option is a general **qmtest** option, **run** is the QMTest command, the `-f full` applies to the **run** command, and `exec1` is an argument to the run command. This command tests QMTest to run the `exec1` test from the test database in `tdb`, and to use the `full` format when displaying the results.

To see a list of available commands, and general options to **qmtest**, invoke it with the `--help` (or `-h`) option. To see a description of each command, and additional options specific to that command, invoke **qmtest *command* --help**.

By this point, you have modified the test database using the GUI and have fixed the failing test. Recreate the original database now by removing and recreating the `tdb` directory. On a UNIX system use these commands:

```
     > cd ..
     > rm -rf tdb
     > cp -r /usr/share/qm/tutorial/test/tdb tdb
     > cd tdb
```

On a Windows system, use these commands instead:

```
> cd ..
> rmdir /s tdb
> xcopy c:\Python23\qm\tutorial\test\tdb tdb\ /s
> cd tdb
```

The command for running tests is **qmtest run**. Assuming you made a copy of the example test database as described in the previous section, execute the following command to run all the tests in the database:

```
> qmtest run
```

QMTest runs the tests, and prints a summary of the test run:

```
--- TEST RESULTS -------------------------------------------------------------

  exec0                                         : PASS

  exec1                                         : FAIL
    Expression evaluates to false.

    ExecTest.expr:
      x + x == 5

    ExecTest.value:
      0

    qmtest.target:
      local

  exec2                                         : PASS

--- TESTS THAT DID NOT PASS --------------------------------------------------

  exec1                                         : FAIL
    Expression evaluates to false.


--- STATISTICS ---------------------------------------------------------------

       3        tests total
       1 ( 33%) tests FAIL
       2 ( 67%) tests PASS
```

QMTest shows you the result of the tests as they execute. Then, there is a summary description containing statistics similar to those shown in the graphical user interface. Finally, QMTest lists the tests that did not pass, along with the cause of the failure.

2.8. Expectations on the Command Line

When you run QMTest on the command line, it automatically creates a results file called `results.qmr`. You can specify a different filename with the `-o` option. Run this command:

```
> qmtest run -o expected.qmr
```

to save the results to a file named `expected.qmr` instead of the default `results.qmr`.

Now, when you rerun the tests you can tell QMTest to use `expected.qmr` as the *expected results file*, like this:

```
> qmtest run -O expected.qmr
```

QMTest will rerun the tests, but this time it will not mention the failure of `exec1`. The output will look like:

```
--- TEST RESULTS -------------------------------------------------------------

  exec0                                       : PASS

  exec1                                       : XFAIL
    Expression evaluates to false.

    ExecTest.expr:
      x + x == 5

    ExecTest.value:
      0

    qmtest.target:
      local

  exec2                                       : PASS

--- TESTS WITH UNEXPECTED OUTCOMES -------------------------------------------

  None.


--- STATISTICS ---------------------------------------------------------------

       3        tests total
       3 (100%) tests as expected
```

Note that QMTest indicates that there were no tests with unexpected outcomes, even though `exec1` still fails. The `XFAIL` notation indicates that the test failed, but that failure was expected. In contrast, `XPASS` means that a test passed unexpectedly.

2.9. Reviewing Results

You can use the results file generated by QMTest to get additional information about the tests that failed. The default results file name is `results.qmr` and is placed in the directory where you ran QMTest.

To examine the results file, use the **summarize** command, like this:

```
> qmtest summarize -f full
```

The `-f full` option indicates that the output should be displayed in more detail. The output will look like:

```
--- TEST RESULTS --------------------------------------------------------------

  exec0                                         : PASS

    qmtest.target:
      local

  exec1                                         : FAIL
    Expression evaluates to false.

    ExecTest.expr:
      x + x == 5

    ExecTest.value:
      0

    qmtest.target:
      local

  exec2                                         : PASS

    qmtest.target:
      local

--- TESTS THAT DID NOT PASS ---------------------------------------------------

  exec1                                         : FAIL
    Expression evaluates to false.


--- STATISTICS ----------------------------------------------------------------

       3        tests total
       1 ( 33%) tests FAIL
       2 ( 67%) tests PASS
```

The detailed information indicates what went wrong. The test value was `0` which is considered false by Python. The information displayed by the "full" format is domain-dependent; it depends on the kind of application you are testing. The tests in the sample database test basic functionality of the Python interpreter, so the full report contains information about Python concepts called exceptions and tracebacks. If you were testing a different application, the full report would contain different information. For example, if you were testing a database, the detailed results might refer to queries and records.

Notes

1. Under Windows, you must use the standard Windows command shell (DOS) to run QMTest; alternative shells (such as Cygwin) will not work with QMTest.

Chapter 3. Using QMTest

This chapter describes QMTest in more detail. It explains the fundamental concepts that QMTest uses, the test classes that come with QMTest, and how to extend QMTest to support new application domains.

The central principle underlying the design of QMTest is that the problem of testing can be divided into a domain-dependent problem and a domain-independent problem. The domain-dependent problem is deciding what to test and how to test it. For example, should a database be tested by performing unit tests on the C code that makes up the database, or by performing integration tests using SQL queries? How should the output of a query asking for a set of records be compared to expected output? Does the order in which records are presented matter? These are questions that only someone who understands the application domain can answer.

The domain-independent part of the problem is managing the creation of tests, executing the tests, and displaying the results for users. For example, how does a user create a new test? How are tests stored? Should failing tests be reported to the user, even if the failure was expected? These questions are independent of the application domain; they are just as relevant for compiler tests as they are for database tests.

QMTest is intended to solve the domain-independent part of the problem and to offer a convenient, powerful, and flexible interface for solving the domain-dependent problem. QMTest is both a complete application, in that it can be used “out of the box” to handle many testing domains, and infrastructure, in that it can be extended to handle other domains.

3.1. QMTest Concepts

This section presents the concepts that underlie QMTest’s design. By understanding these concepts, you will be able to better understand how QMTest works. In addition, you will find it easier to extend QMTest to new application domains.

3.1.1. Tests

A *test* checks for the correct behavior of the target application. What constitutes correct behavior will vary depending on the application domain. For example, correct behavior for a database might mean that it is able to retrieve records correctly while correct behavior for a compiler might mean that it generates correct object code from input source code.

Every test has a name that uniquely identifies the test, within a given test database. Test names must be composed entirely of lowercase letters, numbers, the “_” character, and the “.” character. You can think of test names like file names. The “.” character takes the place of “/”; it allows you to place a test in a particular *directory*. For example, the test name `a.b.c` names a test named `c` in the directory `a.b`. The directory `a.b` is a subdirectory of the directory `a`.

Every test is an instance of some test class. The test class dictates how the test is run, what constitutes success, and what constitutes failure. For example, the `command.ExecTest` class that comes with QMTest executes the target application and looks at its output. The test passes if the actual output exactly matches the expected output.

The arguments to the test parameterize the test; they are what make two instances of the same test class different from each other. For example, the arguments to `command.ExecTest` indicate which application to run, what command-line arguments to provide, and what output is expected.

Sometimes, it may be pointless to run one test unless another test has passed. Therefore, each test can have a set of associated *prerequisite tests*. If the prerequisite tests did not pass, QMTest will not run the test that depends upon them.

3.1.2. Resources

Some tests take a lot of work to set up. For example, a database test that checks the result of SQL queries may require that the database first be populated with a substantial number of records. If there are many tests that all use the same set of records, it would be wasteful to set up the database for each test. It would be more efficient to set up the database once, run all of the tests, and then remove the databases upon completion.

You can use a *resource* to gain this efficiency. If a test depends on a resource, QMTest will ensure that the resource is available before the test runs. Once all tests that depend on the resource have been run QMTest will destroy the resource.

Just as every test is an instance of a *test class*, every resource is an instance of a *resource class*. The resource class explains how to set up the resource and how to clean up when it is no longer needed. The arguments to the resource class are what make two instances of the same resource class different from each other. For example, in the case of a resource that sets up a database, the records to place in the database might be given as arguments. Every resource has a name, using the same format that is used for tests.

Under some circumstances (such as running tests on multiple machines at once), QMTest may create more than one instance of the same resource. Therefore, you should never depend on there being only one instance of a resource. In addition, if you have asked QMTest to run tests concurrently, two tests may access the same resource at the same time. You can, however, be assured that there will be only one instance of a particular resource on a particular target at any one time.

Tests have limited access to the resources on which they depend. A resource may place additional information into the context (Section 3.1.3) that is visible to the test. However, the actual resource object itself is not available to tests. (The reason for this limitiation is that for a target consisting of multiple processes, the resource object may not be located in the process as the test that depends upon it.)

Setting up or cleaning up a resource produces a result, just like those produced for tests. QMTest will display these results in its summary output and record them in the results file.

3.1.3. Context

When you create a test, you choose arguments for the test. The test class uses this information to run the test. However, the test class may sometimes need information that is not available when the test is created. For example, if you are writing compiler tests to verify conformance with the C programming language specification, you do not know the location of the C compiler itself. The C compiler may be installed in different locations on different machines.

A *context* gives users a way of conveying this kind of information to tests. The context is a set of key/value pairs. The keys are always strings. The values of all context properties provided by the user are strings. In general, all tests in a given use of QMTest will have the same context. However, when a resource is set up, it may place additional information in the context of those tests that depend upon it. The values inserted by the resource may have any type, so long as they can be "pickled" by Python.

All context properties whose names begin with "`qmtest.`" are reserved for use by QMTest. The values inserted by QMTest may have any type. Test and resource classes should not depend on the presence or absence of these properites.

3.1.4. Test Results

A *result* is an *outcome* together with some *annotations*. The outcome indicates whether the test passed or failed. The annotations give additional information about the result, such as the manner in which the test failed, the output the test produced, or the amount of time it took to run the test.

3.1.4.1. Outcomes

The outcome of a test indicates whether it passed or failed, or whether some exceptional event occurred. There are four test outcomes:

- PASS: The test succeeded.
- FAIL: The test failed.
- ERROR: A problem occurred in the test execution environment, rather than in the tested system. For example, this outcome is used when the test class attempted to run an executable in order to test it, but could not because the system call to create a new process failed.

  This outcome may also indicate a defect in QMTest or in the test class.
- UNTESTED: QMTest did not attempt to execute the test. For example, this outcome is used when QMTest determines that a prerequisite test failed.

3.1.4.2. Annotations

An annotation is a key/value pair. Both the keys and values are strings. The value is HTML. When a test (or resource) runs it may add annotations to the result. These annotations are displayed by QMTest and preserved in the results file. If you write your own test class, you can use annotations to store information that will make your test class more informative.

3.1.5. Test Suite

A *test suite* is a collection of tests. QMTest can run an entire test suite at once, so by grouping tests together in a test suite, you make it easier to run a number of tests at once. A single test can be a member of more than one test suite. A test suite can contain other test suites; the total set of tests in a test suite includes both those tests included directly and those tests included as part of another test suite. Every test suite has a name, following the same conventions given above for tests and resources.

One use of test suites is to provide groups of tests that are run in different situations. For example, the `nightly` test suite might consist of those tests that should be run automatically every night, while the `checkin` test suite might consist of those tests that have to pass before any changes are made to the target application.

3.1.5.1. Implicit Test Suites

Section 3.1.1 explains how you may arrange tests in a tree hierarchy, using a period (“`.`”) as the path separator in test names. QMTest defines an *implicit test suite* for each directory. The name of these implicit test suites is the same as the name of the directory. The implicit test suite corresponding to a directory contains all tests in that directory or its subdirectories.

Consider, for example, a test database which contains tests with these names:

```
back_end.db_1
back_end.db2
front_end.cmdline
front_end.gui.widget_1
front_end.gui.widget_2
```

For this test database, QMTest defines implicit test suites with IDs `back_end`, `front_end`, and `front_end.gui`. The test suite `front_end` contains the tests `front_end.cmdline`, `front_end.gui.widget_1`, and `front_end.gui.widget_2`.

The suite named "`.`" (a single period) is the implicit test suite corresponding to the root directory in the test database. This suite therefore contains all tests in the database. For example, the command

```
> qmtest run .
```

is equivalent to:

```
> qmtest run
```

Both commands run all tests in the database.

3.1.6. Test Database

A *test database* stores tests, test suites, and resources. When you ask QMTest for a particular test by name, it queries the test database to obtain the test itself. QMTest stores a test database in a single directory, which may include many files and subdirectories.

In general, QMTest can only use one test database at a time. However, it is possible to create a test database which contains other test databases. This mechanism allows you to store the tests associated with different parts of a large application in different test databases, and still combine them into a single large test database when required.

A single test database can store many different kinds of tests. By default, QMTest stores tests, resources, and test suites in the test database using subdirectories containing XML files. Generally, there should be no need to examine or modify these files directly. However, the use of an XML format makes it easy for you to automatically generate tests from another program, if required.

3.1.7. Targets

A *target* is QMTest's abstraction of a machine. By using multiple targets, you can run your tests on multiple machines at one. If you have many tests, and many machines, you can greatly reduce the amount of time it takes to run all of your tests by distributing the tests across multiple targets.

By default, QMTest uses only one target: the machine on which you are running QMTest. You may specify other targets by creating a target file, which lists the available targets and their attributes, and specifying the target file when you invoke **qmtest**.

Each target is a member of a single *target group*. All machines in the same target group are considered equivalent. A target group is specified by a string. If you are testing software on multiple platforms at once, the target group might correspond to machines running the same operating system. For example, all Intel 80386 compatible machines running GNU/Linux might be in the "`i386-pc-linux-gnu`" target group.

Section 3.6 describes how you specify and use targets with QMTest.

3.2. Running Tests

To run one or more tests, use the **qmtest run** command. Each invocation of the **qmtest run** command is a single test run, and produces a single set of test results and statistics. Specify as arguments the names of tests and test suites to run. Even if you specify a test more than once, either directly or by incorporation in a test suite, QMTest runs it only once.

If you wish to run all tests in the test database, use the implicit test suite `.` (a single period; see Section 3.1.5.1), or omit all IDs from the command line.

QMTest can run tests in multiple concurrent threads of execution or on multiple remote hosts. See the documentation for the **run** command for details.

3.2.1. Ordering and Dependencies

Given one or more input test names and test suite names, QMTest employs the following procedure to determine which tests and resources to run and the order in which they are run.

1. QMTest resolves test names and test suite names. Test suites are expanded into the tests they contain. Since test suites may contain other test suites, this process is repeated until all test suites have been expanded. The result is a set of tests that are to be run.
2. QMTest computes a schedule for running the tests to be run such that a test's prerequisites are run before the test itself is run. Prerequisites not included in the test run are ignored. Outside of this condition, the order in which tests are run is undefined.

   If QMTest is invoked to run tests in parallel or distributed across several targets, the tests are distributed among them as well. QMTest does not guarantee that a test's prerequisites are run on the same target, though. On each target, tests are assigned to the next available concurrent process or thread.

3. QMTest determines the required resources for the tests to be run. If several tests require the same resource, QMTest attempts to run all of the tests on the same target. In this case, the resource is set up and cleaned up only once. In some cases, QMTest may schedule the tests on multiple targets; in that case, the resource is set up and cleaned up once on each target.

In some cases, a test, resource setup function, or resource cleanup function is not executed:

- A test specifies for each of its prerequisite tests an expected outcome. If the prerequisite is included in the test run and the actual outcome of the prerequisite test is different from the expected outcome, the test is not run. Instead, it is given an UNTESTED outcome.

  If a test's prerequisite is not included in the test run, that prerequisite is ignored.

- If a setup function for one of the resources required by a test fails, the test is given an UNTESTED outcome.
- The cleanup function of a resource is run after the last test that requires that resource, whether or not that test was run. The cleanup function is run even if the setup function failed.

3.2.2. The Context

QMTest passes a context object to the `Run` method of a test that is run and to the `SetUp` method of a resource.

Most of the properties of the context are the same for all tests and resource functions run during a single test run. These properties are configured as part of the test run. For example, when you run tests using the **qmtest run** command, you may specify individual context properties with the `--context` (`-c`) or `--load-context` (`-C`) options.

In addition, a resource setup function may add additional properties to a context. These added properties do not become part of the common context; they are hidden from other tests and resources except that the properties added by a resource are visible to tests that require that resource.

For instance, a resource `SetUp` function might allocate the resource and place a handle to it (for instance, a temporary directory name or a database session key) in the context as a context property. Tests that require that resource have access to the temporary resource via the handle stored in the context. The resource's cleanup function also uses the handle to deallocate the resource. That information should be stored in the resource object itself since no context is made available to the `CleanUp`.

3.3. Test Database Contents

The default QMTest test database implementation stores the database as a directory hierarchy containing XML files. Each QMTest subdirectory is represented by a subdirectory in the filesystem. A test, suite, or resource is represented by an XML file. These files have file extensions `.qmt`, `.qms`, and `.qma`, respectively.

Expert QMTest users may modify the contents of the test database directly by editing these files. However, it is the user's responsibility to ensure the integrity and validity of the XML contents of each file. For example, file and directory names should contain only characters allowed in identifiers (lower-case letters, digits, hyphens, and underscores); a period should only be used before a file extension, such as `.qmt`. Also, the files and directories in a test database should not be modified directly while QMTest is running with that test database.

3.4. Invoking QMTest

All QMTest functionality is available using the **qmtest** command.

3.4.1. qmtest

3.4.1.1. Synopsis

qmtest [*`option`* ...] *`command`* [*`command-option`* ...] [*`argument`* ...]

3.4.1.2. Options

These options can be used with any QMTest command, and must precede the command name on the command line.

All options are available in a "long form" prefixed with "--" (two hyphens). Some options also may be specified in a "short form" consisting of a single hyphen and a one-letter abbreviation. Short-form options may be combined; for example, **-abc** is equivalent to **-a -b -c**.

`-D` *`path`*
`--tdb` *`path`*

> Use the test database located in the directory given by *`path`*. This flag overrides the value of the environment variable QMTEST_DB_PATH. If neither this flag nor the environment variable is specified, QMTest assumes that the current directory should be used as the database. See Section 3.1.6.

`-h`
`--help`

> Display help information, listing commands and general options for the **qmtest** command.

`--version`

> Describe the version of QMTest in use.

Additional options are available for specific commands; these are presented with each command. Options specific to a command must follow the command on the command line. Specify the `--help` (`-h`) option after the command for a description of the command and a list of of available options for that command.

3.4.2. qmtest create

3.4.2.1. Summary

Create a new extension instance.

3.4.2.2. Synopsis

qmtest create [*`option`* ...] *`kind descriptor`*

3.4.2.3. Description

The **qmtest create** creates a new extension instance. For example, this command can be used to create a new test or resource. For a list of the kinds of extensions supported by QMTest, run **qmtest extensions**. The *`kind`* must be one of these extension kinds.

The descriptor specifies an extension class and (optionally) attributes for that extension class. The form of the descriptor is ***`class`*(*`attributes`*)**, where the attributes are of the form ***`attr`*** `= "`***`val`***`"`. If there are no attributes, the parentheses may be omitted.

The *`class`* may be either the path to an extension file or a QMTest class name in the form *`module.class`*. If the *`class`* is the path to an extension file (such as an existing test or resource file), the class name used is the one provided in the file; otherwise the class named used is the name provided on the command line.

The attributes used to construct the extension instance come from three sources: the attributes in the extension file (if the *`class`* is the path to an extension file), the `--attribute` options provided on the command line, and the explicit attributes provided in the descriptor. If multiple values for the the same attribute name are provided, the value used is taken from the first source in the following list for which there is a value: the rightmost attribute provided in the descriptor, the extension file, or the rightmost `--attribute` present on the command line.

The new extension file is written to the file specified with the `--output` option, or to the standard output if no `--output` is specified.

The **create** command accepts these options:

`-a` *`name=value`*
`--attribute` *`name=value`*

Set the target class argument *`name`* to *`value`*. The set of valid argument names and valid values is dependent on the extension class in use.

`-o` *`file`*
`--output` *`file`*

Write a description of the extension instance to *`file`*.

3.4.2.4. Example

This command:

```
qmtest create -a format=stats -o rs
       result_stream text_result_stream.TextResultStream(filename="rs")
```

creates a file called `rs` containing an instance of `TextResultStream`.

3.4.3. qmtest create-target

3.4.3.1. Summary

Create a new target.

3.4.3.2. Synopsis

qmtest create-target [*`option`* ...] *`name class`* [*`group`*]

3.4.3.3. Description

The **qmtest create-target** command creates a new target. A target is an entity that runs tests; normally, a target corresponds to a particular machine.

The target's name and class must be specified. An optional group may also be specified. When QMTest decides which target to use to run a particular tests, it will select a target that matches the test's requested target group.

The **create-target** command accepts these options:

`-a` *`name=value`*
`--attribute` *`name=value`*

Set the target class argument *`name`* to *`value`*. The set of valid argument names and valid values is dependent on the target class in use.

`-T` *`file`*
`--targets` *`file`*

Write the target description to the indicated *`file`*. If there are already targets listed in *`file`*, they will be preserved, except that any target with the same name as the new target will be removed. If this option is not present, the file used will be the `QMTest/targets` file in the test database directory.

3.4.4. qmtest create-tdb

3.4.4.1. Summary

Create a new test database.

3.4.4.2. Synopsis

qmtest create-tdb [*`option`* ...]

3.4.4.3. Description

The **qmtest create-tdb** command creates a new, empty test database. A test database is a directory in which QMTest stores configuration files, tests, and other data. Certain test database classes may also store data elsewhere, such as in an external relational database.

The test database is created in the directory specified by `--tdb` (`-D`) option or by setting the QMTEST_DB_PATH environment variable. If no database path is specified, QMTest assumes that the current directory is the test database.

By default, QMTest creates a new test database that uses the standard XML-based implementation. (See Section 4.5 for information about writing a test database class.)

The **create-tdb** command accepts these options:

`-a` *`name=value`*
`--attribute` *`name=value`*

Set the database attribute *`name`* to *`value`*. The set of attribute names and valid values is dependent on the database class in use. The default database class accepts no attributes.

`-c` *`class`*
`--class` *`class`*

Use the test database class given by *`class`*. The *`class`* may have the general form described in Section 3.4.2. Once you create a test database, you cannot change the test database implementation it uses. If you do not use this option, QMTest will use the default test database implementation, which uses an XML file format to store tests.

3.4.5. qmtest gui

3.4.5.1. Summary

Start the graphical user interface.

3.4.5.2. Synopsis

qmtest gui [`option` ...]

3.4.5.3. Description

The **qmtest gui** starts the graphical user interface. The graphical user interface is accessed through a web browser. You must have a web browser that supports JavaScript to use the graphical interface. QMTest has been tested with recent versions of Internet Explorer and Netscape Navigator. Other web browsers may or may nor work with QMTest.

The **gui** command accepts these options:

`-A` *`address`*
`--address` *`address`*

Bind the server to the indicated internet *`address`*, which should be a dotted quad. By default, the server binds itself to the address `127.0.0.1`, which is the address of the local machine. If you specify another address, the server will be accessible to users on other machines. QMTest does not perform any authentication of remote users, so you should not use this option unless you have a firewall in place that blocks all untrusted users.

`-c` *`name=value`*
`--context` *`name=value`*

For details about this option, see the description of the **qmtest run** command.

`-C` *`file`*
`--load-context` *`file`*

For details about this option, see the description of the **qmtest run** command.

`--daemon`

Run the QMTest GUI as a daemon. In this mode, QMTest will detach from the controlling terminal and run in the background until explicitly shutdown.

`-j` *`count`*
`--concurrency` *`count`*

For details about this option, see the description of the **qmtest run** command.

`--no-browser`

Do not attempt to start a web browser when starting the GUI. QMTest will still print out the URL at which the server can be accessed. You can then connect to this URL manually using the browser of your choice.

`-O` *`file`*
`--outcomes` *`file`*

For details about this option, see the description of the **qmtest run** command.

`--pid-file` *`path`*

Specify the *`path`* to which the QMTest GUI will write its process ID. This option is useful if you want to run QMTest as a daemon. If this option is not provided, no PID file is written. If you specify this option, but *`path`* is the empty string, QMTest will check the `.qmrc` configuration file for a `pid-file` entry. If there is no such entry, QMTest will use an appropriate platform-specific default value.

`--port` *`port`*

Specify the *`port`* on which the QMTest GUI will listen for connections. If this option is not provided, QMTest will select an available port automatically.

`-T` *`file`*
`--targets` *`file`*

For details about this option, see the description of the **qmtest run** command.

3.4.6. qmtest extensions

3.4.6.1. Summary

List available extension classes.

3.4.6.2. Synopsis

qmtest extensions [*`option`* ...]

3.4.6.3. Description

The **qmtest extensions** lists available extension classes and provides a brief description of each class. You can use this command to list all of the available extension classes, or to list all of the available extension classes of a particular type. For example, you can use this command to list all of the available test classes.

The **extensions** command accepts these options:

`-k` *`kind`*
`--kind` *`kind`*

List the available extension classes of the indicated *`kind`*. The *`kind`* must be one of `test`, `resource`, `target`, or `database`.

3.4.7. qmtest register

3.4.7.1. Summary

Register an extension class.

3.4.7.2. Synopsis

qmtest register `*kind*` `*class-name*`

3.4.7.3. Description

The **qmtest register** registers an extension class with QMTest. As part of this process, QMTest will load your extension class. If the extension class cannot be loaded, QMTest will tell you what went wrong.

QMTest will search for your extension class in the directories it would search when running tests, including those given by the environment variable QMTEST_CLASS_PATH.

The *`kind`* argument tells QMTest what kind of extension class you are registering. If you invoke **qmtest register** with no arguments it will provide you with a list of the available extension kinds.

The *`class-name`* argument gives the name of the class in the form `module.Class`. QMTest will look for a file whose basename is the module name and whose extension is either `py`, `pyc`, or `pyo`.

3.4.8. qmtest run

3.4.8.1. Summary

Run tests or test suites.

3.4.8.2. Synopsis

qmtest run [*`option`* ...] [*`test-name`* | *`suite-name`*]...

3.4.8.3. Description

The **qmtest run** command runs tests and displays the results. If no test or suite names are specified, QMTest runs all of the tests in the test database. If test or suite names are specified, only those tests or suites are run. Tests listed more than once (directly or by inclusion in a test suite) are run only once.

The **run** command accepts these options:

`-c` *`name`*`=`*`value`*
`--context` *`name`*`=`*`value`*

Add a property to the test execution context. The name of the property is *`name`*, and its value is set to the string *`value`*.

This option may be specified multiple times.

`-C` *`file`*
`--load-context` *`file`*

Read properties for the test execution context from the file *`file`*.

The file should be a text file with one context property on each line, in the format *`name`*`=`*`value`*. Leading and trailing whitespace on each line are ignored. Also, blank lines and lines that begin with "#" (a hash mark) are ignored as comments.

This option may be specified more than once, and used in conjunction with the `--context` option. All of the context properties specified are added to the eventual context. If a property is set more than once, the last value provided is the one used.

If this option is not specified, but a file named `context` exists in the current directory, that file is read. The properties specified in this file are processed first; the values in this file can be overridden by subsequent uses of the `--context` option on the command line.

`-f` *`format`*
`--format` *`format`*

Control the format used when displaying results. The format specified must be one of `full`, `brief`, `stats`, `batch`, or `none`. The `brief` format is the default if QMTest was invoked interactively; the `batch` format is the default otherwise. In the `full` format, QMTest displays any annotations provided in test results. In the `brief` mode only the causes of failures are shown; detailed annotations are not shown. In the `stats` format, no details about failing tests are displayed; only statistics showing the number of passing and failing tests are displayed. In the `batch` mode, the summary is displayed first, followed by detailed results for tests with unexpected outcomes. In the `none` mode, no results are displayed, but a results file is still created, unless the `--no-output` option is also provided.

`-j` *`count`*
`--concurrency` *`count`*

Run tests in multiple *`count`* concurrent processes on the local computer. On multiprocessor machines, the processes may be scheduled to run in parallel on different processors. QMTest automatically collects results from the processes and presents combines test results and summary. By default, one process is used.

This option may not be combined with the `--targets` (`-T`) option.

`--no-output`

Do not produce a test results file.

`-o` *`file`*
`--output` *`file`*

Write full test results to *`file`*. Specify "`-`" (a hyphen) to write results to standard output. If neither this option nor `--no-output` is specified, the results are written to the file named `results.qmr` in the current directory.

`-O` *`file`*
`--outcomes` *`file`*

Treat *`file`* as a set of expected outcomes. The *`file`* must have be a results file created either by **qmtest run**, or by saving results in the graphical user interface. QMTest will expect the results of the current test run to match those specified in the *`file`* and will highlight differences from those results.

`--random`

Run the tests in a random order.

This option can be used to find hidden dependencies between tests in the testsuite. (You may not notice the dependencies if you always run the tests in the same order.)

`--rerun` *`file`*

Rerun only those tests that had unexpected outcomes.

The tests run are determined as follows. QMTest starts with all of the tests specified on the command line, or, if no tests are explicitly specified, all of the tests in the database. If no expectations file is specified (see the description of the `--outcomes` option), then all tests that passed in the results file indicated by the `--rerun` option are removed form the set of eligible tests. If an expectations file is specified, then the tests removed are tests whose outcome in the results file indicated by the `--rerun` option is the same as in the expectations file.

The `--rerun` provides a simple way of rerunning failing tests. If you run your tests and notice failures, you might try to fix those failing tests. Then, you can rerun the failing tests to see if you succeeded by using the `--rerun` option.

`--result-stream` *`descriptor`*

Specify an additional output result stream. The descriptor is in the format described in Section 3.4.2.

`--seed` *`integer`*

If the `--random` is used, QMTest randomizes the order in which tests are run, subject to the constraints described in Section 3.2.1. By default, the random number generator is seeded using the system time.

For debugging purposes, it is sometimes necessary to obtain a reproducible sequence of tests. Use the `--seed` option to specify the seed for the random number generator.

Note that even with the same random number seed, if tests are run in parallel, scheduling uncertainty may still produce variation in the order in which tests are run.

`-T` *`file`*
`--targets` *`file`*

Use targets specified in target specification file *`file`*. If this option is not present, the `QMTest/targets` in the test database directory will be used. If that file is not present, the tests will be run in serial on the local machine.

3.4.9. qmtest summarize

3.4.9.1. Summary

The **qmtest summarize** displays information stored in a results file.

3.4.9.2. Synopsis

qmtest summarize [`*option*` ...] [`*test-name*` | `*suite-name*`]...

3.4.9.3. Description

The **qmtest summarize** extracts information stored in a results file and displays this information on the console. The information is formatted just as if the tests had just been run, but QMTest does not actually run the tests.

The **summarize** command accepts the following options:

`-f` *`format`*
`--format` *`format`*

For details about this option, see the description of the **qmtest run** command.

`-O` *`file`*
`--outcomes` *`file`*

For details about this option, see the description of the **qmtest run** command.

`--result-stream` *`descriptor`*

Specify an additional output result stream. The descriptor is in the format described in Section 3.4.2.

3.4.10. Environment Variables

QMTest recognizes the following environment variables:

QMTEST_CLASS_PATH

If this environment variable is set, it should contain a list of directories in the same format as used for the system's PATH environment variable. These directories are searched (before the directories that QMTest searches by default) when looking for extension classes such as test classes and database classes.

QMTEST_DB_PATH

If this environment variable is set, its value is used as the location of the test database, unless the `--tdb` (`-D`) option is used. If this environment variable is not set and the `--tdb` option is not used, the current directory is used as the test database.

3.4.11. Configuration Variables

These configuration variables are used by QMTest. You should define them in the `[qmtest]` section of your QM configuration file.

pid-file

The default path to use when creating a PID file with the `--pid-file` option. (See Section 3.4.5 for more information about this option.) If this entry is not present, an appropriate platform-specific default value is used.

3.4.12. Return Value

If QMTest successfully performed the action requested, QMTest returns 0. For the **qmtest run** or **qmtest summarize** commands, success implies not only that the tests ran, but also that all of the tests passed (if the `--outcomes` option was not used) or had their expected outcomes (if the `--outcomes` option was used).

If either the **run** command or the **summarize** command was used, and at least one test failed (if the `--outcomes` option was not used) or had an unexpected outcome (if the `--outcomes` option was used), **qmtest** returns 1.

If QMTest could not perform the action requested, **qmtest** returns 2.

3.5. Test and Resource Classes

This section describes test classes and resource classes included with QMTest. Section 4.3 provides instructions for writing your own test classes, Section 4.4 for resource classes.

3.5.1. Test Classes

3.5.1.1. `command.ExecTest`

The `command.ExecTest` test class runs a program from an ordinary executable file. Each test specifies the program executable to run, its full command line, and the data to feed to its standard input stream. `ExecTest` collects the complete text of the program’s standard output and standard error streams and the program’s exit code, and compares these to expected values specified in the test. If the standard output and error text and the exit code match the expected values, the test passes.

A `command.ExecTest` test supplies the following arguments:

Program (text field)

The name of the executable file to run. `command.ExecTest` attempts to locate the program executable in the path specified by the path property of the test context.

Argument List (set of strings)

The argument list for the program. The elements of this set are sequential items from which the program's argument list is constructed. `command.ExecTest` automatically prepends an implicit zeroth element, the full path of the program.

Standard Input (text field)

Text or data to pass to the program's standard input stream. This data is written to a temporary file, and the contents of the file are directed to the program's standard input stream.

Environment (set of strings)

The environment (i.e. the set of environment variables) available to the executing program. Each element of this argument is a string of the form "*`VARIABLE`*=*`VALUE`*".

`command.ExecTest` adds additional environment variables automatically.

In addition, every context property whose value is a string is accessible as an environment variable; the name of the environment variable is the name of the context property, prefixed with "QMV_" and with any dots (".") replaced by a double underscore ("__"). For example, the value of the context property "CompilerTable.c_path" is available as the value of the environment variable "QMV_CompilerTable__c_path".

Expected Exit Code (integer field)

The exit code value expected from the program. If the program produces an exit code value different from this one, the test fails.

Expected Standard Output (text field)

The text or data which the program is expected to produce on its standard output stream. The actual text or data written to standard output is captured, and `command.ExecTest` performs a bytewise comparison to the expected text or data. If they do not match, the test fails.

Expected Standard Error (text field)

The text or data which the program is expected to produce on its standard error stream. The actual text or data written to standard error is captured, and `command.ExecTest` performs a bytewise comparison to the expected text or data. If they do not match, the test fails.

3.5.1.2. `command.ShellCommandTest`

`command.ShellCommandTest` is very similar to `command.ExecTest`, except that it runs a program via the shell rather than directly. Instead of specifying an executable to run and the elements of its argument list, a test provides a single command line. The shell is responsible for finding the executable and constructing its argument list.

Standard input and the environment are specified in the test. The test passes if the command produces the expected standard output, standard error, and exit code.

Note that most shells create local shell variables to mirror the contents of the environment when the shell starts up. Therefore, the environment set up by a `command.ShellCommandTest`, including the contents of the test context, are directly accessible via shell variables. The syntax to use depends on the particular shell.

`command.ShellCommandTest` has the same fields as `command.ExecTest`, except that the Program and Argument List properties are replaced with these:

Command (text field)

The command to run. The command is delivered verbatim to the shell. The shell interprets the command according to its own quoting rules and syntax.

3.5.1.3. `command.ShellScriptTest`

`command.ShellScriptTest` is an extension of `command.CommandTest` that lets a test specify an entire shell script instead of a single command. The script specified in the test is written to a temporary file, and this file is interpreted by the specified shell or command interpreter program.

Standard input, the environment, and the argument list to pass to the script are specified in the test. The test passes if the script produces the expected standard output, standard error, and exit code.

Note that most shells create local shell variables to mirror the contents of the environment when the shell starts up. Therefore, the environment set up by a `command.ShellScriptTest`, including the contents of the test context, are directly accessible via shell variables. The syntax to use depends on the particular shell.

`command.ShellScriptTest` has the same fields as `command.ExecTest`, except that the Program property is replaced with:

Script (text field)

The text of the script to run.

3.6. Test Targets

Test targets represent entities that QMTest uses to run tests. See Section 3.1.7 for an overview of how QMTest uses targets.

3.6.1. Target Specification

Each target specification includes the following:

1. The name of the target. This is a name identifying the target, such as the host name of the computer which will run the tests. Target names should be unique in a single target file.
2. The *target class*. Similar to a test class, a target class is a Python class which implements a type of target. As with test classes, a target class is identified by its name, which includes the module name and the class name.

   For example, `thread_target.ThreadTarget` is the name of a target class, provided by QMTest, which runs tests in multiple threads on the local computer.

   QMTest includes several target class implementations. See Section 3.6.2 for details.
3. A target group name. The test implementor may choose the syntax of target group names in a test implementation. Target groups may be used to encode information about target attributes, such as architecture and operating system, and capabilities.
4. Optionally, a target specification may include additional properties. Properties are named and have string values. Some target classes may use property information to control their configuration. For instance, a target class which executes tests on a remote computer would extract the network address of the remote computer from a target property.

3.6.2. Target Classes

QMTest includes these target class implementations.

3.6.2.1. `SerialTarget`

The `serial_target.SerialTarget` target class runs tests one after the other on the machine running QMTest. If you use a `SerialTarget`, you should not also use any other targets, including another `SerialTarget` at the same time.

3.6.2.2. `ThreadTarget`

The `thread_target.ThreadTarget` target class runs tests in one or more threads on the machine running QMTest. The `ThreadTarget` can be used to run multiple tests at once.

`ThreadTarget` uses the following properties:

- The concurrency specifies the number of threads to use. Larger numbers of threads will allow QMTest to run more tests in parallel. You can experiment with this value to find the setting that allows QMTest to run tests most quickly.

3.6.2.3. `ProcessTarget`

The `process_target.ProcessTarget` target class run tests in one more processes on the machine running QMTest. This target class is not available on Windows. Like `ThreadTarget`, `ProcessTarget` can be used to run multiple tests simultaneously.

In general, you should use `ThreadTarget` instead of `ProcessTarget` to maximize QMTest performance. However, on machines that do not have threads, `ProcessTarget` provides an alternative way of running tests in parallel.

`ProcessTarget` uses the following properties:

- The concurrency specifies the number of processes to use. Larger numbers of processes will allow QMTest to run more tests in parallel. You can experiment with this value to find the setting that allows QMTest to run tests most quickly.
- QMTest uses the path given by the **qmtest** property to create additional QMTest instances. By default, the path `/usr/local/bin/qmtest` is used.

3.6.2.4. `RemoteShellTarget`

The `rsh_target.RSHTarget` target class runs tests on a remote computer via a remote shell invocation (**rsh**, **ssh**, or similar). This target uses a remote shell to invoke a program similar to the **qmtest** command on the remote computer. This remote program accepts test commands and responds with results from running these tests.

To use `RSHTarget`, the remote computer must have QMTest installed and must contain an identical copy of the test database. QMTest does not transfer entire tests over the remote shell connection; instead, it relies on the remote test database for loading tests.

In addition, the remote shell program must be configured to allow a remote login without additional intervention (such as typing a password). If you use **rsh**, you can use an `.rhosts` file to set this up. If you use **ssh**, you can use an SSH public key and the **ssh-agent** program for this. See the corresponding manual pages for details.

`RSHTarget` uses all of the properties given above for `ProcessTarget`. In addition, `RSHTarget` uses the following properties:

- The remote_shell property specifies the path to the remote shell program. The default value is **ssh**. The remote shell program must accept the same command-line syntax as **rsh**.
- The host property specifies the remote host name. If omitted, the target name is used.
- The database_path property specifies the path to the test database on the remote computer. The test database must be identical to the local test database. If omitted, the local test database path is used.
- The arguments property specifies additional command-line arguments for the remote shell program. The value of this property is split at space characters, and the arguments are added to the command line before the name of the remote host.

  For example, if you are using the **ssh** remote shell program and wish to log in to the remote computer using a different user account, specify the `-l username` option using the arguments property.

Chapter 4. Extending QMTest

If the built-in functionality provided with QMTest does not serve all of your needs, you can extend QMTest. All extensions to QMTest take the form of Python classes. You can write new test classes, resource classes, or database classes in this way.

The contents of the class differ depending on the kind of extension you are creating. For example, the methods that a new test class must implement are different from those that must be provided by a new database class. In each case, however, you must create the class and place it in a location where QMTest can find it. The following sections explain how to create extension classes. The last section in this chapter explains how to register your new extension classes.

4.1. Extension Classes

All extensions to QMTest are implemented by writing a new Python class. This new Python class will be derived from an appropriate existing QMTest Python class. For example, new test classes are derived from `Test` while new test database classes are derived from `Database`.

The classes from which new extensions are derived (like `Test`) are all themselves derived from `Extension` (http://www.codesourcery.com/qm/qmtest_internals_docs/qm/extension/Extension.html). The `Extension` class provides the basic framework used by all extension classes. In particular, every instance of `Extension` can be represented in XML format in persistent storage.

Every `Extension` class has an associated list of *arguments*. When an `Extension` instance is written out as XML, the value of each argument which is encoded in the output. Similarly, when an `Extension` instance is read back in, the arguments are decoded. Conceptually, two `Extension` instances are the same if they are instances of the same derived class and their arguments have the same values.

Each argument has both a name and a type. For example, every `Test` has an argument called `target_group`. The target group is a string indicating on which targets a particular test should be run.

Each argument is represented by an instance of `Field` (http://www.codesourcery.com/qm/qmtest_internals_docs/qm/fields/Field.html). A `Field` instance can read or write values in XML format. A `Field` can also produce an HTML representation of a value, or an HTML form that allows a user to update the value of the field. It is the fact that all `Extension` arguments are instances of `Field` that makes it possible to represent `Extension` instances as XML. Smilarly, it is the the use of the `Field` class that allows the user to edit tests in the QMTest GUI.

Each class derived from `Extension` may contain a variable called `arguments`. The value of `arguments` must be a list of `Field` instances. The complete set of arguments for a derived class consists of the arguments specified in the derived class together with all of those specified in base classes. In other words, a derived class should not explicitly include arguments that have already been specified in a base class.

For example, after the following class definitions:

```
class A(Extension):
     arguments = [ TextField("x") ]

 class B(A):
     arguments = [ IntegerField("y"),
                   TextField("z") ]
```

`A` has one argument (`x`) and `B` has three arguments (`x`, `y`, and `z`).

None of the arguments may have the same name as a class variable in the extension class, including class variables in base classes.

4.2. Field Classes

A `Field` (http://www.codesourcery.com/qm/qmtest_internals_docs/qm/fields/Field.html) is a named, typed component of a data structure. A `Field` can read and write XML representations of values, generate HTML representations of values, or present HTML forms that permit the user to update the value of the field. There are several classes derived from `Field` that you can use in extension classes. If none of those classes satisfy your needs, you can create a new class derived from `Field`.

Every `Field` has a name. The name is a string, and must be a valid Python identifier. (The reason for this restriction is that instances of `Extension` have an instance variables corresponding to each field.) A `Field` may also have a title, which is used when presenting the `Field` to the user. The title need not be a valid Python identifier. For example, the `RSHTarget` class has an argument whose name is `host`, but whose title is `Remote Host Name`. When accessing an instance of this class, the programmer refers to `self.host`. In the GUI, however, the user will see the value presented as `Remote Host Name`.

A `Field` may have an associated description, which is a longer explanation of the `Field` and its purpose. This information is presented to the user by the GUI.

A `Field` may have a default value. The default value is used if no explicit value is provided for the field.

This example code from `RSHTarget` shows how a `Field` is constructed:

```
qm.fields.TextField(
     name="remote_shell",
     title="Remote Shell Program",
     description="""The path to the remote shell program.

      The name of the program that can be used to create a
      remote shell.  This program must accept the same command
      line arguments as the ’rsh’ program.""",
      default_value="ssh")
```

See the internal documentation for `Field` (http://www.codesourcery.com/qm/qmtest_internals_docs/qm/fields/Field.html) for complete interface documentation.

4.2.1. Built-In Field Classes

QMTest comes with several useful field classes:

- `IntegerField` (http://www.codesourcery.com/qm/qmtest_internals_docs/qm/fields/IntegerField.html) stores integers.
- `TextField` (http://www.codesourcery.com/qm/qmtest_internals_docs/qm/fields/TextField.html) stores strings.
- `EnumerationField` (http://www.codesourcery.com/qm/qmtest_internals_docs/qm/fields/EnumerationField.html) stores one of a set of (statically determined) possible values.

- `ChoiceField` (http://www.codesourcery.com/qm/qmtest_internals_docs/qm/fields/ChoiceField.html) stores one of a set of (dynamically determined) possible values.
- `BooleanField` (http://www.codesourcery.com/qm/qmtest_internals_docs/qm/fields/BooleanField.html) stores a boolean value.
- `TimeField` (http://www.codesourcery.com/qm/qmtest_internals_docs/qm/fields/TimeField.html) stores a date and time.
- `AttachmentField` (http://www.codesourcery.com/qm/qmtest_internals_docs/qm/fields/AttachmentField.html) stores arbitrary data.
- `SetField` (http://www.codesourcery.com/qm/qmtest_internals_docs/qm/fields/SetField.html) stores multiple values of the same type.
- `TupleField` (http://www.codesourcery.com/qm/qmtest_internals_docs/qm/fields/TupleField.html) stores a fixed number of other fields.

4.2.2. Writing Field Classes

Before writing any code, you should decide what kind of data your field class will store. For example, will your field class store arbitrary strings? Or only strings that match a particular regular expression? Or will your field class store images? Once you have decided this question, you can write the `Validate` function for your field class. This function checks an input value (a Python object) for validity. `Validate` can return a modified version of the value. For example, if the field stores strings, you could choose to accept an integer as an input to `Validate` and convert the integer to a string before returning it.

The `FormatValueAsHtml` function produces an HTML representation of the value. You must define this function so that the GUI can display the value of the field. The *`style`* parameter indicates how the value should be displayed. If the style is `new` or `edit`, the HTML representation returned should be a form that the user can use to set the value. If the user does not modify the form, `ParseFormValue` should yield the value that was provided to `FormatValueAsHtml`.

The `MakeDomNodeForValue` and `GetValueFromDomNode` functions convert values to and from XML format. The `FormatValueAsText` and `ParseTextValue` functions convert to and from plain text. As with `FormatValueAsHtml` and `ParseFormValue`, these pairs of functions should be inverses of one another.

The `ParseTextValue`, `ParseFormValue`, and `GetValueFromDomNode` functions should use `Validate` to check that the values produced are permitted by the `Field`. In this way, derived classes that want to restrict the set of valid values, but are otherwise content to use the base class functionality, need only provide a new implementation of `Validate`.

All of the functions which read and write `Field` values may raise exceptions if they cannot complete their tasks. The caller of the `Field` is responsible for handling the exception if it occurs.

4.3. Writing Test Classes

If the test classes that come with QMTest do not serve your needs, you can write a new test class. A test class is a Python class derived from `Test` (http://www.codesourcery.com/qm/qmtest_internals_docs/qm/test/test/Test.html). The test class must define an `arguments` variable, whose value is a sequence of `Field`s, and a `Run` function.

The arguments to the test are the inputs to the test. The `Run` function explains how to perform the test and how to determine whether or not it passed. For example, if you want to test that a compiler correctly compiled a particular source file, the source file would be an argument to the test while the `Run` would be responsible for running the compiler and the program generated by the compiler. The path to the compiler itself would be provided via the context (Section 3.1.3); that is an input to the testing system that varies depending on the user's environment.

The `Run` function takes two arguments: the context and the result. The context object is an instance of `Context` (http://www.codesourcery.com/qm/qmtest_internals_docs/qm/test/context/Context.html). The result object is an instance of `Result` (http://www.codesourcery.com/qm/qmtest_internals_docs/qm/test/result/Result.html). The result is initialized with the PASS outcome. Therefore, if the `Run` function does not modify the result, the test will pass. If the test fails, the `Result.Fail` function should be called to indicate failure.

The `Result.Annotate` function can be used to add information to the `Result`, whether or not the test passes. For example, annotations can be used to record the time a test took to execute, or to log the output from a command run as part of the test. Every annotation is a key/value pair. Both keys and values are strings. The key created by a test class `C` should have the form `C.key_name`. The value must be valid HTML. When results are displayed in the GUI, the HTML is presented directly to the user. When results are displayed as text, the HTML is converted to plain text. That conversion uses textual devices (such as single quotes around verbatim text) to emulate the HTML markup where possible.

As a convenience, you can use Python's dictionary notation to access annotations. For example:

```
result["C.key1"] = "value"
result["C.key2"] = result["C.key1"].upper()
```

is equivalent to:

```
result.Annotate({ "C.key1" : "value"
                  "C.key2" : "VALUE" })
```

The context (like the result) is a set of key/value pairs. The keys used by a test class `C` should have the form `C.key_name`. The values are generally strings, but if a test depends on a resource, the resource can provide context values that are not strings.

If the `Run` raises an unhandled exception, QMTest creates a result for the test with the outcome ERROR. Therefore, test classes should be designed so that they do not raise unhandled exceptions when a test fails. However, QMTest handles the exception generated by the use of non-existant context variables specially. Because this situation generally indicates incorrect usage of the test suite, QMTest uses a special error message that instructs the user to supply a value for the context variable.

4.4. Writing Resource Classes

Writing resource classes is similar to writing test classes. The requirements are the same except that, instead of a `Run` function, you must provide two functions named `SetUp` and `CleanUp`. The `SetUp` function must have the same signature as a test classs `Run`. The `CleanUp` function is similar, but does not take a *`context`* parameter.

The setup function may add additional properties to the context. These properties will be visible only to tests that require this resource. To add a context property, use Python's dictionary assignment syntax.

Below is an example of setup and cleanup functions for a resource which calls `create_my_resource` and `destroy_my_resource` to do the work of creating and destroying the resource. The resource is identified by a string handle, which is inserted into the context under the name `Resource.handle`, where it may be accessed by tests. Context property names should always have the form `Class.name` so that there is no risk of collision between properties created by different resource classes.

4.5. Writing Database Classes

The test database class controls the format in which tests are stored. QMTest's default database class stores each test as an XML file, but you might want to use a format that is particularly well suited to your application domain or to your organization's arrangement of computing resources.

For example, if you were testing a compiler, you might want to represent tests as source files with special embedded comments indicating what errors are expected when compiling the test. You could write a test database class that can read and write tests in that format.

Or, if you wanted to share a single test database with many people in such a way that everyone automatically saw updates to the database, you might want to put all of the tests on a central HTTP server. You could write a test database class that retrieves tests from the server and creates new tests by uploading them to the server.

A test database class is a Python class that is derived from `Database` (http://www.codesourcery.com/qm/qmtest_internals_docs/qm/test/database/Database.html), which is itself derived from `Extension`. To create a new database class, you must define methods that read and write tests, resources, and suites.

The database is also responsible for determining how tests (and other entities stored in the database) are named. Each item stored in the database must have a unique name. For a database that stores files in the filesystem, the name of the file may be a good name. For a database of unit tests for Python module, the name of the module might be a good name for the tests. Choosing the naming convention appropriate requires understanding both the application domain and the way in which the tests will actually be stored.

The database class must have a `GetTest` function which retrieves a test from the database. The *`test_id`* parameter provide the name of the test. The `GetTest` function returns a `TestDescriptor` (http://www.codesourcery.com/qm/qmtest_internals_docs/qm/test/database/TestDescriptor.html). [1] A `TestDescriptor` indicates the test class, and the arguments to that test class. QMTest uses that information to instantiate an instance of the test class itself as appropriate.

The `Write` function is the inverse of `GetTest`. The test database is responsible for storing the `Test` provided. The name of test can be obtained by calling `GetId` on the `Test`. When the `Remove` function is called the database is responsible for removing the test named by the *`id`* parameter.

The functions that handle resources are analogous to those for tests. For exmaple, `GetResource` plays the same role for resources as `GetTest` does for tests.

4.6. Registering an Extension Class

To use your test or resource class, you must place the Python module file containing it in a directory where QMTest can find it. QMTest looks in three places when loading extension classes:

- If the environment variable QMTEST_CLASS_PATH is defined, QMTest first checks any directories listed in it. This value of this environment variable should be a list of directories to check for the module file, in the same format as the standard PATH environment variable.

- A test database may specify additional locations to check.
- QMTest checks the configuration directory (the subdirectory named `QMTest` of a test database).
- Finally, QMTest checks a standard directory. This directory, installed with QMTest, contains modules with the standard test classes described in Section 3.5.

You should generally place module files containing your test classes in the test database's `QMTest` directory, unless you plan to use the test classes in more than one test database.

You must use the **qmtest register** command to register your new extension class. You must perform this step no matter where you place the module containing your extension class.

You can refer to the new extension class using the syntax `module.Class`, where `module` is the name of the module and `Class` is the name of the class.

Notes

1. `GetTest` returns a `TestDescriptor`, rather than a `Test`, because that allows QMTest to avoid loading in the test class. If you are running many tests in parallel, on many different machines, this indirection makes QMTest more effficient; QMTest only needs to load a particular test class on a particular machine if an instance of that class is being run on that machine.

Chapter 5. The QM Configuration File

QM allows you to set up a per-user configuration file that contains your personal preferences, defaults, and settings.

The configuration file is named `$HOME/.qmrc`. On Windows, you may have to set the HOME environment variable manually.

The QM configuration file is a plain text file, with a format similar to that used in Microsoft Windows `.INI` files. It is divided into sections by headings in square brackets. Three sections are supported: `[common]` contains configuration variables common to all the QM tools, while `[test]` contains configuration variables specific to QMTest. Within each section, configuration variables are set using the syntax `variable=value`.

Here is a sample QM configuration file:

```
> cat ~/.qmrc
[common]
browser=/usr/local/bin/mozilla
```

5.1. Configuration Variables

These configuration variables are used in all QM tools. You should define them in the `[common]` section of your QM configuration file.

browser (UNIX-like platforms only)

The path to your preferred web browser. If omitted, QM attempts to run `mozilla`. The QM GUI does not correctly with Netscape 4 due to limitations in the support for JavaScript and DOM in that browser.

command_shell

The shell program to run a single shell command. The value of this property is the path to the shell executable, optionally followed by command-line options to pass to the shell, separated by spaces. The shell command to run is appended to the command.

On GNU/Linux systems, the default is `/bin/bash -norc -noprofile -c`. On other UNIX-like systems, the default is `/bin/sh -c`.

click_menus

If this option is not present, or has the value `0`, menus in the GUI are activated by moving the mouse over the menu name.

If this option has the value `1`, the menus are activated by clicking on the menu name.

remote_shell (UNIX-like platforms only)

The program used for running commands on remote computers. The program must accept the same syntax as the standard `rsh` command, and should be configured to run the command remotely without any additional interaction (such as requesting a password from the TTY). The default value is `/usr/bin/ssh`.

script_shell

The shell program to run a shell script. The value of this property is the path to the shell executable, optionally followed by command-line options to pass to the shell, separated by spaces. The filename of the shell command is appended to the command.

On GNU/Linux systems, the default is `/bin/bash -norc -noprofile`. On other UNIX-like systems, the default is `/bin/sh`.