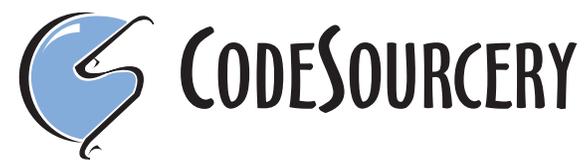

QMTTest: User's Guide



QMTTest: User's Guide

CodeSourcery, Inc.

Copyright © 2002-2006 CodeSourcery Inc

QMTTest is a testing tool. You can use QMTTest to test a software application, such as a database, compiler, or web browser. You can even QMTTest 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, QMTTest makes it easy to find problems easier, rather than later.

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

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

CodeSourcery provides support for QMTTest. CodeSourcery can help you install, integrate, and customize QMTTest. For more information, visit the QMTTest web site¹.

I. REQUIREMENTS ON BOTH UNMODIFIED AND MODIFIED VERSIONS

Open Publication work may be reproduced and distributed in whole or in part, in any medium physical or electronic, provided that the terms of this license are adhered to, and that this license or an incorporation of it by reference is displayed in the reproduction.

Proper form for incorporation of this license by reference is as follows:

Copyright © 2000, 2001 by CodeSourcery LLC. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License.

Commercial redistribution of material covered by this license is permitted.

Any publication in standard (paper) book form shall require the citation of the original author and (where applicable) publisher.

II. COPYRIGHT

The copyright to each Open Publication is owned by its author(s) or designee(s).

III. SCOPE OF LICENSE

The license terms below apply to all Open Publication works.

AGGREGATION. Mere aggregation of Open Publication works or a portion of an Open Publication work with other works or programs on the same media shall not cause this license to apply to those other works. The aggregate work shall contain a notice specifying the inclusion of the Open Publication material and appropriate copyright notice.

SEVERABILITY. If any part of this license is found to be unenforceable in any jurisdiction, the remaining portions of the license remain in force.

NO WARRANTY. Open Publication works are licensed and provided 'as is' without warranty of any kind, express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose or a warranty of non-infringement.

IV. REQUIREMENTS ON MODIFIED WORKS

¹ <http://www.qmtest.com>

All modified versions of documents covered by this license, including translations, anthologies, compilations and partial documents, must meet the following requirements:

1. The modified version must be labeled as such.
2. The person making the modifications must be identified and the modifications dated.
3. Acknowledgement of the original author and publisher if applicable must be retained according to normal academic citation practices.
4. The location of the original unmodified document must be identified.
5. The original author's (or authors') name(s) may not be used to assert or imply endorsement of the resulting document without the original author's (or authors') permission.

V. GOOD-PRACTICE RECOMMENDATIONS

In addition to the requirements of this license, it is requested from and strongly recommended of redistributors that:

1. If you are distributing Open Publication works on hardcopy or CD-ROM, you provide email notification to the authors of your intent to redistribute at least thirty days before your manuscript or media freeze, to give the authors time to provide updated documents. This notification should describe modifications, if any, made to the document.
2. All substantive modifications (including deletions) be either clearly marked up in the document or else described in an attachment to the document.

Finally, while it is not mandatory under this license, it is considered good form to offer a free copy of any hardcopy and CD-ROM expression of an Open Publication-licensed work to its author(s).

Table of Contents

1. QMTest Concepts	1
1.1. Tests	2
1.1.1. Running Tests	3
1.1.2. Prerequisite Tests	3
1.1.3. Ordering and Dependencies	4
1.2. Test Results	5
1.2.1. Outcomes	5
1.2.2. Annotations	5
1.2.3. Expected Outcomes	6
1.3. Test Suites	6
1.3.1. Implicit Test Suites	6
1.3.2. Explicit Test Suites	7
1.4. Test Database	8
1.5. Expectation Database	8
1.6. Context	8
1.7. Resources	9
1.8. Targets	10
1.9. Hosts	11
2. Invoking QMTest	12
2.1. qmtest	13
2.1.1. Synopsis	13
2.1.2. Options	13
2.2. qmtest create	13
2.2.1. Summary	13
2.2.2. Synopsis	13
2.2.3. Description	13
2.2.4. Example	14
2.3. qmtest create-target	15
2.3.1. Summary	15
2.3.2. Synopsis	15
2.3.3. Description	15
2.4. qmtest create-tdb	15
2.4.1. Summary	15
2.4.2. Synopsis	15
2.4.3. Description	15
2.5. qmtest gui	16
2.5.1. Summary	16
2.5.2. Synopsis	16
2.5.3. Description	16
2.6. qmtest extensions	17
2.6.1. Summary	17
2.6.2. Synopsis	17
2.6.3. Description	17
2.7. qmtest describe	17
2.7.1. Summary	17
2.7.2. Synopsis	18
2.7.3. Description	18
2.8. qmtest ls	18
2.8.1. Summary	18
2.8.2. Synopsis	18
2.8.3. Description	18

2.9. qmltest register	18
2.9.1. Summary	18
2.9.2. Synopsis	18
2.9.3. Description	19
2.10. qmltest run	19
2.10.1. Summary	19
2.10.2. Synopsis	19
2.10.3. Description	19
2.11. qmltest summarize	21
2.11.1. Summary	21
2.11.2. Synopsis	21
2.11.3. Description	22
2.12. qmltest report	22
2.12.1. Summary	22
2.12.2. Synopsis	22
2.12.3. Description	22
2.13. Environment Variables	22
2.14. Configuration Variables	23
2.15. Return Value	23
3. Customizing QMLTest	24
3.1. Extensions	25
3.2. Tests	26
3.2.1. <code>command.ExecTest</code>	26
3.2.2. <code>command.ShellCommandTest</code>	27
3.2.3. <code>command.ShellScriptTest</code>	27
3.2.4. <code>CompilationTest</code>	28
3.3. Test Suites	28
3.4. Test Resources	28
3.4.1. <code>TempDirectoryResource</code>	28
3.4.2. <code>CompilerTable</code>	28
3.5. Test Databases	29
3.5.1. <code>XMLDatabase</code>	29
3.5.2. <code>CompilationTestDatabase</code>	29
3.6. Expectation Databases	30
3.6.1. <code>PreviousTestRun</code>	30
3.6.2. <code>XMLExpectationDatabase</code>	31
3.7. Test Targets	31
3.7.1. Target Specification	31
3.7.2. Target Classes	32
3.8. Hosts	33
3.8.1. <code>local_host.LocalHost</code>	33
3.8.2. <code>ssh_host.SSHHost</code>	33
3.8.3. <code>ssh_host.RSSHHost</code>	33
3.8.4. <code>simulator.Simulator</code>	34
3.9. Result Streams and Result Readers	34
3.9.1. <code>text_result_stream.TextResultStream</code>	34
3.9.2. <code>xml_result_stream.XMLResultStream</code>	34
3.9.3. <code>pickle_result_stream.PickleResultStream</code>	34
3.9.4. <code>sql_result_stream.SQLResultStream</code>	34
3.10. The QMLTest Configuration File	34
3.10.1. Configuration Variables	35
4. Extending QMLTest	36
4.1. Extension Classes	37
4.2. Field Classes	38

4.2.1. Built-In Field Classes	39
4.2.2. Writing Field Classes	39
4.3. Writing Test Classes	40
4.4. Writing Resource Classes	41
4.5. Writing Database Classes	41
4.6. Registering and Distributing Extension Classes	42

Chapter 1

QMTest Concepts

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

The central principle underlying the design of QMTTest 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.

QMTTest 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. QMTTest 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.

Throughout this chapter we will use the `qmtest` application with a variety of parameters and options. For a full description of the `qmtest` please refer to the command-line reference.

The following commands create a simple test database in the current working directory. This test database will be used throughout the following sections of this tutorial.

```
> mkdir tdb
> cd tdb
> qmtest create-tdb
```

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 QMTTest 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.

The `python.ExecTest` class is similar to `command.ExecTest`, but, instead of executing a command using the system shell, it evaluates an expression in the Python programming language. The test passes if (and only if) the expression is true. In Python, the expressions `True` and `False` are literals.

The following creates two trivial tests, `python_pass` and `python_fail`:

```
> qmtest create --id=python_pass -a expression='True' test python.ExecTest
> qmtest create --id=python_fail -a expression='False' test python.ExecTest
```

The first test will always pass while the second will always fail.

The **qmtest ls** command will show the content of the test database:

```
> qmtest ls
python_fail
python_pass
```

Similar to the Unix **ls** command, the `-l` option can be used to provide a detailed listing, with kind ("test", "resource", or "suite"), extension class, and id:

```
> qmtest ls -l
test python.ExecTest python_fail
test python.ExecTest python_pass
```

1.1.1 Running Tests

To run one or more tests, use the **qmtest run** command:

```
> qmtest run
```

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 1.3.1, "Implicit Test Suites"), 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.

How the output of the **qmtest run** command should be interpreted will be discussed in Section 1.2, "Test Results".

1.1.2 Prerequisite Tests

QMTest can avoid running one test (a "dependent test") when some other test (a "prerequisite test") has a particular outcome.

Suppose that you have a test database with a very simple test that can be run very quickly, and a very complex test that takes hours to run. You know that if the simple test fails, then there is no chance that the complex test will pass. In that case, you could make the simple test a prerequisite of the complex test. Then, when you run both tests, QMTTest will run the simple test first. If it fails, the complex test will not be run at all.

Alternatively, suppose that you have a very comprehensive test that tests ten features of your software. You also have ten separate tests, one for each feature. The comprehensive test can be run in one minute; running the separate tests takes two minutes each. So, you want to run the comprehensive test first; if it passes, there is no need to run the individual tests. However, if the comprehensive test fails, you may want to run the single tests to isolate the problem. In this case, each of the simple tests would have the comprehensive test as a prerequisite, indicating that the simple test should be run only if the comprehensive test fails.

If you explicitly run just the dependent test, QMTTest will not run the prerequisite test automatically. In other words, prerequisites are an optimization; when running both the prerequisite and the dependent test, QMTTest will run them in the order you've implied, and can omit the dependent test if it is not useful. But, QMTTest will not automatically force you to run the prerequisite tests when you only want to run the dependent test.

Because prerequisite tests are not run unless you ask for them, the dependent test should not depend in any way on the prerequisite test. Otherwise, users will see different test outcomes when they run the dependent test by itself. In other words, each test should stand alone; the order in which tests are run should not affect their outcomes.

1.1.3 Ordering and Dependencies

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

1. QMTTest 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. QMTTest 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 QMTTest is invoked to run tests in parallel or distributed across several targets, the tests are distributed among them as well. QMTTest 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. QMTTest determines the required resources for the tests to be run. If several tests require the same resource, QMTTest 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, QMTTest may schedule the tests on multiple targets; in that case, the resource is set up and cleaned up once on each target.

In the following cases, a test or resource will not be executed, even though it is included in the set of tests enumerated above:

- 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.

1.2 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.

1.2.1 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 QMLTest or in the test class.

- **UNTESTED:** QMLTest did not attempt to execute the test. For example, this outcome is used when QMLTest determines that a prerequisite test failed.

Thus, running QMLTest with the two previously defined tests will result in the following output:

```
> qmltest run
...
--- TEST RESULTS -----
python_fail                               : FAIL
  Expression evaluates to false.
python_pass                                : PASS
...
--- STATISTICS -----
      2      tests total
      1 ( 50%) tests FAIL
      1 ( 50%) tests PASS
```

In addition, the results are stored in a results file (`results.qml` by default).

1.2.2 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

QMLTest 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.

1.2.3 Expected Outcomes

The easiest way to create expectations is to tell QMLTest that you expect future results to be the same as the results you just obtained. Thus, QMLTest accepts result files obtained from prior test runs as expectations.

Thus, rerunning QMLTest, but using `results.qmr` as expectations, the test results are displayed differently:

```
> qmltest run -O results.qmr
...
--- TEST RESULTS -----
python_fail                               : XFAIL
  Expression evaluates to false.
python_pass                                : PASS
...
--- TESTS WITH UNEXPECTED OUTCOMES -----
None.
--- STATISTICS -----
      2 (100%) tests as expected
```

1.3 Test Suites

A *test suite* is a collection of tests. QMLTest 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.

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.

1.3.1 Implicit Test Suites

Section 1.1, “Tests” explains how you may arrange tests in a tree hierarchy, using a period (“.”) as the path separator in test names. QMLTest 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.

Let us create some more tests, but this time within their respective subdirectories:

```
> qmltest create --id=dir1.one -a expression='True' test python.ExecTest
> qmltest create --id=dir1.two -a expression='False' test python.ExecTest
```

```
> qmtest create --id=dir2.one -a expression='True' test python.ExecTest
> qmtest create --id=dir2.dir3.one -a expression='False' test python.ExecTest
> qmtest create --id=dir2.dir3.two -a expression='False' test python.ExecTest
```

This will create five new tests, together with three directories:

```
> qmtest ls -lR
directory          dir1
test      python.ExecTest dir1.one
test      python.ExecTest dir1.two
directory          dir2
directory          dir2.dir3
test      python.ExecTest dir2.dir3.one
test      python.ExecTest dir2.dir3.two
test      python.ExecTest dir2.one
```

These directories are implicit suites, i.e. it is possible to only address tests within them:

```
> qmtest ls -lR dir2
directory          dir2
directory          dir2.dir3
test      python.ExecTest dir2.dir3.one
test      python.ExecTest dir2.dir3.two
test      python.ExecTest dir2.one
```

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.

1.3.2 Explicit Test Suites

Explicit test suites are a means to refer to a set of tests for the purpose of executing them together, no matter how they are organized in the test database. For example, you may want to run a subset of all tests nightly. To do this, you create a suite, listing all tests (as well as contained suites) explicitly:

```
> qmtest create --id=nightly suite explicit_suite.ExplicitSuite(test_ids=["dir1.one", 'dir2.one'],
                                                                suite_ids=["dir2.dir3'])
> qmtest ls -l
directory          dir1
directory          dir2
suite      explicit_suite.ExplicitSuite nightly
> qmtest ls -l nightly
test      python.ExecTest dir1.one
directory          dir2.dir3
test      python.ExecTest dir2.one
```

1.4 Test Database

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

In general, QMTTest 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, QMTTest stores tests, test suites, and all other entities 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.

1.5 Expectation Database

While typically tests are expected to pass, it sometimes is meaningful to indicate a different expectation. QMTTest represents expectations by results. Here, the outcomes represent the expected outcome, and the annotations may explain the expectation.

An *expectation database* stores expectations associated with a test database. It provides an interface to query expectations by test id.

In the simplest case all tests are expected to pass. In a slightly less simple case, an existing set of results, obtained from a previous test run, may be used as expectations for a subsequent test run.

1.6 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 QMTTest 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 "qmttest." are reserved for use by QMTTest. The values inserted by QMTTest may have any type. Test and resource classes should not depend on the presence or absence of these properties.

To understand how a context is used during the execution of a test let us start by creating a somewhat less trivial test:

```
> qmtest create --id compile test
```

```
compilation_test.CompilationTest(executable="compile", source_files=["/path/to/compile.cc"])"
```

When run, this test will compile `/path/to/compile.cc` and run the resulting executable. The test passes if compilation succeeds, and the program exit status indicates success.

The `compilation_test.CompilationTest` test class requires that the compiler be available in the context with the key `CompilationTest.compiler_path`. You can provide a context variable to QMLTest either through a context file or on the command-line using the `-c` option. For example:

```
> qmltest run -c CompilationTest.compiler_path=g++ compile
```

will run the test using the `g++` compiler, while:

```
> qmltest run -c CompilationTest.compiler_path=/bin/CC compile
```

will run the test with the `/bin/CC` compiler.

1.7 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, QMLTest will ensure that the resource is available before the test runs. Once all tests that depend on the resource have been run QMLTest 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 targets at once), QMLTest 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 QMLTest 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 1.6, “Context”) that is visible to the test. However, the actual resource object itself is not available to tests. (The reason for this limitation is that for a target consisting of multiple processes, the resource object may not be located in the same process as the test that depends upon it.)

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

Building on the previous example of a `CompilationTest`, let us consider a situation where some test application should be run multiple times with different arguments. The test application, however, needs to be compiled first. In order to avoid recompiling the application for each test, you can create a resource that compiles the application once. Then, tests that depend on this resource can assume that the application has already been built. The following commands:

```
> qmtest create --id applet resource
    compilation_test.CompiledResource(executable="applet", source_files=["/path/to/applet.cc"])
> qmtest create --id run_applet_0 test
    compilation_test.ExecutableTest(args=["0"], resources=["applet"])
> qmtest create --id run_applet_1 test
    compilation_test.ExecutableTest(args=["1"], resources=["applet"])
```

create a resource (named "applet") for the application and two tests ("run_applet_0" and "run_applet_1"), both of which make use of "applet", but which pass it different command-line options.

```
> qmtest run -c CompilationTest.compiler_path=g++ run_applet_0 run_applet_1
--- TEST RESULTS -----
Setup applet                               : PASS
run_applet_0                               : PASS
run_applet_1                               : PASS
Cleanup applet                             : PASS
```

1.8 Targets

A *target* is QMLTest's abstraction of a machine. By using multiple targets, you can run your tests on multiple machines at once. 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, QMLTest uses only one target: the machine on which you are running QMLTest. 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 targets 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.

QMLTest by default executes all tests using a 'serial target', i.e. one after another. This behavior can be changed by specifying a different concurrency strategy, such as one that uses multiple threads, multiple processes, or even multiple physical machines.

```
> qmtest create-target -a name=local -a group=local -a threads=2 a thread_target.ThreadTarget
```

This creates a 'thread target' where two threads are used to run the tests in parallel.

1.9 Hosts

Sometime it is necessary to execute a test application on a different machine than the one running the qmtest application itself. For example, if the test involves executing an application previously cross-compiled, the binary needs to be uploaded to an appropriate host and run there.

QMTTest provides a *host* abstraction for this purpose. To use this mechanism, test classes need to provide explicit support for it. A number of built-in test classes support cross-testing.

To run the `compile` test on a remote target machine, specify a `CompilationTest.target` variable in the context file to contain a host descriptor:

```
CompilationTest.compiler=/path/to/cross_compiler  
CompilationTest.target=ssh_host.SSHHost(host_name="192.168.0.100")
```

This will run the compiled executable on the machine with the IP address `192.168.0.100`, using `ssh` for communication.

Chapter 2

Invoking QMTest

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

2.1 qmtest

2.1.1 Synopsis

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

2.1.2 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**.

<code>-D path, --tdb path</code>	Use the test database located in the directory given by <i>path</i> . This flag overrides the value of the environment variable <code>QMTEST_DB_PATH</code> . If neither this flag nor the environment variable is specified, QMTest assumes that the current directory should be used as the database. See Section 1.4, "Test Database".
<code>-h, --help</code>	Display help information, listing commands and general options for the qmtest command.
<code>--version</code>	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 available options for that command.

2.2 qmtest create

2.2.1 Summary

Create a new extension instance.

2.2.2 Synopsis

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

2.2.3 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.

If the `--id` option is provided then the new instance is created in the test database. The argument to the `--id` option gives the name of the instance. Otherwise, the extension is written as XML to the filename specified by `--output` option, or to the standard output if no `--output` is specified.

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 extending extension or a QMTest class name in the form `module.class`. If the `--id` option has been provided, QMTest will look for an existing extension in the test database named `class`. If the `--id` option has not been provided, QMTest will look for an XML file named `class`. In either case, if an existing extension cannot be found, the `class` is interpreted as the name of an extension class.

The attributes used to construct the extension instance come from three sources: the attributes in the extant extension (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 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 **create** command accepts these options:

<code>-a name=value, --attribute name=value</code>	Set the target class argument <code>name</code> to <code>value</code> . The set of valid argument names and valid values is dependent on the extension class in use.
<code>-i id, --id id</code>	Add the extension instance to the database, using <code>id</code> as the name of the instance.
<code>-o file, --output file</code>	Write the extension instance to <code>file</code> .

2.2.4 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`.

This command:

```
> qmtest create --id=simple -a program=testprog test command.ExecTest
```

creates a test named "simple" that executes the program `testprog`:

This command:

```
> qmtest create --id=copy test simple
```

creates a copy of the "simple" test, naming the new version "copy".

2.3 qmtest create-target

2.3.1 Summary

Create a new target.

2.3.2 Synopsis

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

2.3.3 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:

<code>-a name=value, --attribute name=value</code>	Set the target class argument <i>name</i> to <i>value</i> . The set of valid argument names and valid values is dependent on the target class in use.
<code>-T file, --targets file</code>	Write the target description to the indicated <i>file</i> . If there are already targets listed in <i>file</i> , 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 <code>QMTest/targets</code> file in the test database directory.

2.4 qmtest create-tdb

2.4.1 Summary

Create a new test database.

2.4.2 Synopsis

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

2.4.3 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, “Writing Database Classes” for information about writing a test database class.)

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

<code>-a name=value, --attribute name=value</code>	Set the database attribute <i>name</i> to <i>value</i> . The set of attribute names and valid values is dependent on the database class in use. The default database class accepts no attributes.
<code>-c class, --class class</code>	Use the test database class given by <i>class</i> . The <i>class</i> may have the general form described in Section 2.2, “ qmtest create ”. 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.

2.5 qmtest gui

2.5.1 Summary

Start the graphical user interface.

2.5.2 Synopsis

`qmtest gui [option ...]`

2.5.3 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 not work with QMTest.

The **gui** command accepts these options:

<code>-A address, --address address</code>	Bind the server to the indicated internet <i>address</i> , 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.
<code>-c name=value, --context name=value</code>	For details about this option, see the description of the qmtest run command.
<code>-C file, --load-context file</code>	For details about this option, see the description of the qmtest run command.
<code>--daemon</code>	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.
<code>-j count, --concurrency count</code>	For details about this option, see the description of the qmtest run command.

<code>--no-browser</code>	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.
<code>-O file, --outcomes file</code>	For details about this option, see the description of the qmtest run command.
<code>--pid-file path</code>	Specify the <i>path</i> 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 <i>path</i> is the empty string, QMTest will check the <code>.qmrc</code> configuration file for a <code>pid-file</code> entry. If there is no such entry, QMTest will use an appropriate platform-specific default value.
<code>--port port</code>	Specify the <i>port</i> on which the QMTest GUI will listen for connections. If this option is not provided, QMTest will select an available port automatically.
<code>-T file, --targets file</code>	For details about this option, see the description of the qmtest run command.

2.6 qmtest extensions

2.6.1 Summary

List available extension classes.

2.6.2 Synopsis

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

2.6.3 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:

<code>-k kind, --kind kind</code>	List the available extension classes of the indicated <i>kind</i> . The <i>kind</i> must be one of <code>test</code> , <code>resource</code> , <code>target</code> , <code>database</code> , <code>host</code> , or <code>run_database</code> .
-----------------------------------	---

2.7 qmtest describe

2.7.1 Summary

Describe an extension class.

2.7.2 Synopsis

```
qmtest describe [ option ...] kind name
```

2.7.3 Description

The **qmtest describe** displays a description of the extension *name*.

The **describe** command accepts these options:

<code>--kind kind</code>	Describe an extension class of the indicated <i>kind</i> . The <i>kind</i> must be one of <code>test</code> , <code>resource</code> , <code>target</code> , <code>database</code> , <code>host</code> , or <code>run_database</code> .
<code>-l, --long</code>	Provide a long (i.e. more detailed) description of the extension.
<code>-a name, --attribute name</code>	Describe the given attribute.

2.8 qmtest ls

2.8.1 Summary

List the contents of the test database.

2.8.2 Synopsis

```
qmtest ls [ option ...] [ name ...]
```

2.8.3 Description

The **qmtest ls** lists the contents of the database, just as the UNIX **ls** command lists the contents of the filesystem. If this command is used with no options, QMTest will list the names of the entries in the root directory of the test database. If one or more names are supplied, then QMTest will list those items, rather than the root directory. If a name refers to a directory, then the contents of that directory will be displayed.

The **ls** command accepts these options:

<code>-l, --long</code>	Use a detailed output format that displays the kind and extension class associated with each item.
<code>-d, --details</code>	Display details such as individual attributes for each item.
<code>-R, --recursive</code>	Recursively list the contents of directories.

2.9 qmtest register

2.9.1 Summary

Register an extension class.

2.9.2 Synopsis

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

2.9.3 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`.

2.10 qmtest run

2.10.1 Summary

Run tests or test suites.

2.10.2 Synopsis

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

2.10.3 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:

`-a name=value , --annotate name=value` Annotate the test run by inserting the named annotation *name* with the value *value* into all result streams.

This option may be specified multiple times.

`-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 combined 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`, in QMTest's machine-readable file format. Use a `"-"` (a hyphen) to write results to the 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` is usually a results file created either by using the **qmtest run** or by saving results in the graphical user interface. If `file` does not appear to be such a file, it is interpreted as an extension descriptor, as described in Section 2.2, “**qmtest create**”. 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 from 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 2.2, “**qmtest create**”.

`--seed integer`

If the `--random` is used, QMTest randomizes the order in which tests are run, subject to the constraints described in Section 1.1.3, “Ordering and Dependencies”. 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.

2.11 qmtest summarize

2.11.1 Summary

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

2.11.2 Synopsis

```
qmtest summarize [ option ... ] [ {results-file} [ test-name | suite-name ... ] ]
```

2.11.3 Description

The **qmtest summarize** extracts information stored in the *results-file* (or *results.qmr*, if no *results-file* is specified) and displays this information on the console. The information is formatted just as if the tests had been run with **qmtest run**, but, instead of actually running the tests, QMTest reads the results from the *results-file*.

If the *results-file* is not a valid results file, it is interpreted as an extension descriptor, as described in Section 2.2, “**qmtest create**”. You can use the descriptor syntax to read results stored in formats that are not “built-in” to QMTest.

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

<code>-f format, --format format</code>	For details about this option, see the description of the qmtest run command.
<code>-o file, --output file</code>	Write full test results to <i>file</i> , in QMTest's machine-readable file format. Use a “-” (a hyphen) to write results to the standard output.
<code>-O file, --outcomes file</code>	For details about this option, see the description of the qmtest run command.
<code>--result-stream descriptor</code>	Specify an additional output result stream. The descriptor is in the format described in Section 2.2, “ qmtest create ”.

2.12 qmtest report

2.12.1 Summary

The **qmtest report** generates an xml report from a set of test result files.

2.12.2 Synopsis

```
qmtest report [-O output] [-f] [report-file [-e expectation-file] ...]
```

2.12.3 Description

The **qmtest report** extracts information stored in one or more result files and generates an xml report file from it. This report file is then typically processed using xslt to generate html or pdf versions of the report.

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

<code>-f, --flat</code>	Generate a flat listing of test results, instead of reproducing the database directory tree in the report.
<code>-o output file, --output output-file</code>	The name of the file to write the report into.

2.13 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 <code>PATH</code> 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 <code>--tdb (-D)</code> option is used. If this environment variable is not set and the <code>--tdb</code> option is not used, the current directory is used as the test database.

2.14 Configuration Variables

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

pid-file	The default path to use when creating a PID file with the <code>--pid-file</code> option. (See Section 2.5, “ qctest gui ” for more information about this option.) If this entry is not present, an appropriate platform-specific default value is used.
----------	--

2.15 Return Value

If QMTest successfully performed the action requested, QMTest returns 0. For the **qctest run** or **qctest 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), **qctest** returns 1.

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

Chapter 3

Customizing QMTest

The previous chapter introduced the concepts underlying QMTest's design. These were demonstrated with particular types of tests, resources, databases, etc. . These types are designed to be extensible, i.e. QMTest can operate with arbitrary instances of these types. In fact, QMTest provides a large number of such extensions.

This chapter discusses how to use them, i.e. how to find out what extensions are available, and how to use them to construct custom test databases.

3.1 Extensions

In the previous chapter we have created a test database containing a number of tests. These tests were instances of test types, such as `python.ExecTest`. When creating a test, a user chooses a test type, and assigns test-specific parameters to it.

The same extension mechanism is used for other aspects of QMTest's functionality. All these types are extensions, which provide a generic interface for QMTest to query.

To find out what extensions are available, use **qmtest extensions**. To list all available test types, use:

```
> qmtest extensions -k test
** Available test classes **

- command.ExecTest

  Check a program's output and exit code.

- command.ShellCommandTest

  Check a shell command's output and exit code.

- python.ExecTest

  Check that a Python expression evaluates to true.

...
```

To find out more about a particular extension type, use the **describe** command:

```
> qmtest describe test command.ExecTest

class name: python.ExecTest
  Check that a Python expression evaluates to true.

class attributes:
  prerequisites      The tests on which this test depends.
  source             The source code.
  target_group       The targets on which this test can run.
  expression         The expression to evaluate.
  resources          Resources on which this test or resource depends.
```

To create a test of that type, you can call:

```
> qmtest create --id=my_test -a expression='True' test python.ExecTest
```

where the extension parameters are specified with the `-a` option.

QMTest searches for extensions in particular places, in the following order:

1. in the paths contained in the `QMTEST_CLASS_PATH` environment variable.
2. in the class-path associated with a particular test database (typically the `QMTest/` subdirectory).
3. in the site-extension path associated with a particular QMTest installation.
4. in the set of built-in extensions that come with a particular QMTest installation.

This search allows users to provide their own extension types. See Chapter 4, *Extending QMTest* for a detailed discussion of how to create new extensions.

The rest of this chapter will discuss the various extension kinds that can be used to customize QMTest.

3.2 Tests

3.2.1 `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. <code>command.ExecTest</code> attempts to locate the program executable in the path specified by the <code>path</code> 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. <code>command.ExecTest</code> 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 " <code>VARIABLE=VALUE</code> ". <code>command.ExecTest</code> 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 " <code>QMV_</code> " and with any dots (".") replaced by a double underscore (" <code>__</code> "). For example, the value of the context property "Com-

	<code>pilerTable.c_path</code> " is available as the value of the environment variable <code>"QMV_CompilerTable__c_path"</code> .
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 <code>command.ExecTest</code> performs a bitwise 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 <code>command.ExecTest</code> performs a bitwise comparison to the expected text or data. If they do not match, the test fails.

3.2.2 `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.
----------------------	---

3.2.3 `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.

3.2.4 `CompilationTest`

`compilation_test.CompilationTest` compiles a set of source files and optionally runs the compiled executable.

The compiler executable's name, as well as global compilation parameters are queried from these context variables:

`CompilationTest.compiler_path` The name of the compiler executable.

`CompilationTest.compiler_options` Compiler options.

`CompilationTest.compiler_ldflags` Linker options.

The `CompilationTest` takes the following parameters.

options (set field) Test-specific options to pass to the compiler.

ldflags (set field) Test-specific linker flags to pass to the compiler.

source_files (set field) Source files to be compiled..

executable (text field) The name of the executable to be compiled.

execute (boolean field) Whether or not to run the compiled executable.

3.3 Test Suites

3.4 Test Resources

3.4.1 `TempDirectoryResource`

An instance of this resource creates a temporary directory during setup, and deletes it during cleanup. The full path to the directory is available to tests via a context property.

`dir_path_property` (text field) The name of the context property which is set to the path to the temporary directory.

3.4.2 `CompilerTable`

A `CompilerTable` resource creates a set of compiler objects according to context variables. These compiler objects are then made available to test instances through the `compilers` context variable that maps language names to compiler objects.

The context variable `CompilerTable.languages` should be a whitespace-separated list of programming language names. Then, for each language `<lang>`, the following variables are looked up and used for the creation of compiler objects:

CompilerTable.<lang>_kind (text field)	The kind of compiler (e.g., "GCC" or "EDG").
CompilerTable.<lang>_path (text field)	The path to the compiler.
CompilerTable.<lang>_options (text field)	A whitespace-separated list of command-line options to provide to the compiler.

3.5 Test Databases

3.5.1 XMLDatabase

The XMLDatabase stores its content as a directory hierarchy containing XML files. It is the default test database used by QTest. Each QTest 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 QTest 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 QTest is running with that test database.

3.5.2 CompilationTestDatabase

The CompilationTestDatabase associates source files with CompilationTest instances. The mapping uses file extensions to determine the programming language, and thus, what compiler and compilation flags to use. To create a new test, simply add a source file with an appropriate file extension to the test database.

All tests use the CompilerTable resource, and therefore require the appropriate context variables to be provided.

Additionally, more compiler options can be added per test id or even subdirectory id. Given the following context file:

```
CompilerTable.languages=c
CompilerTable.c_kind=GCC
CompilerTable.c_path=gcc
CompilerTable.c_options=-O2
a.b.c_options=-I /a/b/common
a.b.c.c_options=-I /a/b/c
```

a test `a/b/c/test.c` will be compiled with `gcc -O2 -I /a/b/common -I /a/b/c` while a test `a/b/d/test.c` will be compiled with `gcc -O2 -I /a/b/common`.

The CompilationTestDatabase takes the following parameters.

srkdir (text field)	The root of the test suite's source tree.
excluded_subdirs (set field)	A set of directory names not to be parsed as subdirectories.

test_extensions (dictionary field) The mapping from file extensions to programming languages.

3.6 Expectation Databases

3.6.1 PreviousTestRun

The PreviousTestRun queries expectations from a results file. Thus, running QMTest twice, the second time using the results of the first test run as expectations, will result in no unexpected results:

```
> qmtest run -o my_results.qmr
--- TEST RESULTS -----
exec0                               : PASS
exec1                               : FAIL
...
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 run -e previous_testrun.PreviousTestRun(file_name="my_results.qmr")
--- TEST RESULTS -----
exec0                               : PASS
exec1                               : XFAIL
...
exec2                               : PASS
--- TESTS WITH UNEXPECTED OUTCOMES -----
None.
--- STATISTICS -----
      3 (100%) tests as expected
```

Since taking previous test runs as expectations is a common use case, the second command above may be expressed in a more compact form as:

```
qmtest run -O my_results.qmr
```

The PreviousTestRun takes the following parameters.

file_name (text field) The filename of the results file.

3.6.2 XMLExpectationDatabase

The XMLExpectationDatabase stores expectations in an XML file. Instead of containing expectations for all tests, individual expectations are computed from rules by matching test-ids as well as test-run annotations against specific rules.

```
<expectations>
  <expectation outcome="fail" test_id=".*">
    <annotation name="a" value="b"/>
  </expectation>
  <expectation outcome="fail" test_id="first*" />
</expectations>
```

The above little expectation database thus contains the following rules (subsequent matching rules override previous matching rules):

1. All tests are expected to fail if the annotation *a* has value *b*.
2. All tests whose test-ids start with *first* are expected to fail.

The XMLExpectationDatabase takes the following parameters.

`file_name` (text field) The filename of the xml expectations file.

3.7 Test Targets

Test targets represent entities that QMTest uses to run tests. See Section 1.8, “Targets” for an overview of how QMTest uses targets.

3.7.1 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 3.7.2, “Target Classes” 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.

3.7.2 Target Classes

QMTest includes these target class implementations.

3.7.2.1 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.

3.7.2.2 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.

3.7.2.3 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.

3.7.2.4 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.

3.8 Hosts

A number of test classes delegate the execution of executables to a dedicated `Host` class, for example to allow parts of the test to be run on a different platform (such as when cross-compiling and cross-testing). To achieve that, a number of `Host` subclasses are provided that can be used to execute code in different ways. Typically, a test class will query the `Host` instance to use from a context variable.

3.8.1 `local_host.LocalHost`

A `LocalHost` is the machine on which Python is running.

3.8.2 `ssh_host.SSHHost`

An `SSHHost` is accessible via `ssh` or a similar program. The `SSHHost` host uses the following parameters.

<code>host_name</code> (text field)	The name of the remote host.
<code>ssh_program</code> (text field)	The path to the remote shell program.
<code>ssh_args</code> (set field)	The arguments to the remote shell program.
<code>scp_program</code> (text field)	The path to the remote copy program.
<code>scp_args</code> (set field)	The arguments to the remote copy program.
<code>default_dir</code> (text field)	The default directory on the remote system.

3.8.3 `ssh_host.RSHHost`

An `RSHHost` is a `SSHHost` that uses `rsh` instead of `ssh`.

3.8.4 `simulator.Simulator`

A `Simulator` is a semi-hosted simulation environment. The `Simulator` host uses the following parameters.

<code>simulator</code> (text field)	The simulation program.
<code>simulator_args</code> (set field)	Arguments to the simulation program.

3.9 Result Streams and Result Readers

Result streams and result readers are the means that allow QMTest to externalize and internalize test results.

3.9.1 `text_result_stream.TextResultStream`

A `TextResultStream` displays information textually, in human readable form. It is used when QMTest is run without a graphical user interface.

3.9.2 `xml_result_stream.XMLResultStream`

An `XMLResultStream` writes out results as XML. The resulting file can be read back in using an `XMLResultReader`.

3.9.3 `pickle_result_stream.PickleResultStream`

A `PickleResultStream` writes out results as Python pickles. The resulting file can be read back in using a `PickleResultReader`.

3.9.4 `sql_result_stream.SQLResultStream`

An `SQLResultStream` writes results out to an SQL database. To read results from an SQL database use `SQLResultReader`.

3.10 The QMTest Configuration File

QMTest 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 QMTest 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
```

3.10.1 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 <code>mozilla</code> . 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 <code>/bin/bash -norc -noprofile -c</code> . On other UNIX-like systems, the default is <code>/bin/sh -c</code> .
click_menus	If this option is not present, or has the value <code>0</code> , menus in the GUI are activated by moving the mouse over the menu name. If this option has the value <code>1</code> , 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 <code>rsh</code> 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 <code>/usr/bin/ssh</code> .
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 <code>/bin/bash -norc -noprofile</code> . On other UNIX-like systems, the default is <code>/bin/sh</code> .

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`¹. 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 *parameter* attributes. When an `Extension` instance is written out as XML, the value of each parameter is encoded in the output. Similarly, when an `Extension` instance is read back in, the parameter values are decoded. Conceptually, two `Extension` instances are the same if they are instances of the same derived class and their parameters have the same values.

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

Each parameter is represented by an instance of `Field`². 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` parameters are instances of `Field` that makes it possible to represent `Extension` instances as XML. Similarly, it is the use of the `Field` class that allows the user to edit tests in the QMTest GUI.

Each class derived from `Extension` may contain attributes that are instances of `Field`.

For example, after the following class definitions:

```
class A(Extension):
    x = TextField()

class B(A):
    y = IntegerField(default_value = 42)
    z = TextField(default_value = "a value")
```

`A` has one parameter (`x`) and `B` has three parameters (`x`, `y`, and `z`).

During construction of extensions you may provide arguments to set the values of these parameters (and thus overriding default values):

¹ <http://www.codesourcery.com/public/qmtest/qmtest-2.4.1/internals/qm/extension/Extension.html>

² <http://www.codesourcery.com/public/qmtest/qmtest-2.4.1/internals/qm/fields/Field.html>

```
a = A(x = "argument")
b = B(x = "another argument", z = "value")
```

The serialized form of A will be equivalent to

```
<extension class="__main__.A">
  <argument name="x"><text>argument</text></argument>
</extension>
```

and for B

```
<extension class="__main__.B">
  <argument name="x"><text>another argument</text></argument>
  <argument name="y"><integer>42</integer></argument>
  <argument name="z"><text>value</text></argument>
</extension>
```

Extension instances hold appropriately typed attributes for all fields. A `TextField` translates to a `str` instance, while a `IntegerField` translates to an `int`, etc.

```
> a = A(x = "argument")
> print type(a.x), a.x
<type 'str'> argument
```

4.2 Field Classes

`Field`³ instances are used to represent extension parameters. 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`.

A `Field` may 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 a `host` parameter 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 argument is provided for the field when the extension is initialized.

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

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

³ <http://www.codesourcery.com/public/qmtest/qmtest-2.4.1/internals/qm/fields/Field.html>

```
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`⁴ for complete interface documentation.

4.2.1 Built-In Field Classes

QMTest comes with several useful field classes:

- `IntegerField`⁵ stores integers.
- `TextField`⁶ stores strings.
- `EnumerationField`⁷ stores one of a set of (statically determined) possible values.
- `ChoiceField`⁸ stores one of a set of (dynamically determined) possible values.
- `BooleanField`⁹ stores a boolean value.
- `TimeField`¹⁰ stores a date and time.
- `AttachmentField`¹¹ stores arbitrary data.
- `SetField`¹² stores multiple values of the same type.
- `TupleField`¹³ 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`.

⁴ <http://www.codesourcery.com/public/qmtest/qmtest-2.4.1/internals/qm/fields/Field.html>

⁵ <http://www.codesourcery.com/public/qmtest/qmtest-2.4.1/internals/qm/fields/IntegerField.html>

⁶ <http://www.codesourcery.com/public/qmtest/qmtest-2.4.1/internals/qm/fields/TextField.html>

⁷ <http://www.codesourcery.com/public/qmtest/qmtest-2.4.1/internals/qm/fields/EnumerationField.html>

⁸ <http://www.codesourcery.com/public/qmtest/qmtest-2.4.1/internals/qm/fields/ChoiceField.html>

⁹ <http://www.codesourcery.com/public/qmtest/qmtest-2.4.1/internals/qm/fields/BooleanField.html>

¹⁰ <http://www.codesourcery.com/public/qmtest/qmtest-2.4.1/internals/qm/fields/TimeField.html>

¹¹ <http://www.codesourcery.com/public/qmtest/qmtest-2.4.1/internals/qm/fields/AttachmentField.html>

¹² <http://www.codesourcery.com/public/qmtest/qmtest-2.4.1/internals/qm/fields/SetField.html>

¹³ <http://www.codesourcery.com/public/qmtest/qmtest-2.4.1/internals/qm/fields/TupleField.html>

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`¹⁴, which itself is derived from `Extension`. It may define parameters as discussed in Section 4.1, “Extension Classes”. The test class must provide a `Run` method that implements the way the test is performed and results are validated.

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` method 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 1.6, “Context”); that is an input to the testing system that varies depending on the user's environment.

The `Run` method takes two arguments: the context and the result. The context object is an instance of `Context`¹⁵. The result object is an instance of `Result`¹⁶. The result is initialized with the `PASS` outcome. Therefore, if the `Run` method does not modify the result, the test will pass. If the test fails, the `Result.Fail` method should be called to indicate failure.

The `Result.Annotate` method 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" })
```

¹⁴ <http://www.codesourcery.com/public/qmtest/qmtest-2.4.1/internals/qm/test/test/Test.html>

¹⁵ <http://www.codesourcery.com/public/qmtest/qmtest-2.4.1/internals/qm/test/context/Context.html>

¹⁶ <http://www.codesourcery.com/public/qmtest/qmtest-2.4.1/internals/qm/test/result/Result.html>

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-existent 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` method, you must provide two methods named `SetUp` and `CleanUp`. The `SetUp` method must have the same signature as a test class's `Run`. The `CleanUp` method is similar, but does not take a `context` parameter.

The `SetUp` method may add additional properties to the context by assigning to its `context` parameter. These additional properties will be visible only to tests that require this resource.

The example below shows the `SetUp` and `CleanUp` from the standard QMTest `TempDirectoryResource` class. This resource creates a temporary directory for use by the tests that depend on the resource. The `SetUp` method creates the temporary directory and records the path to the temporary directory in the context so that tests know where to find the directory. The `CleanUp` method removes the temporary directory.

```
def SetUp(self, context, result):

    # Create a temporary directory.
    self.__dir = qm.temporary_directory.TemporaryDirectory()
    # Provide dependent tests with the path to the new directory.
    context["TemporaryDirectoryResource.temp_dir_path"]
        = self.__dir.GetPath()

def CleanUp(self, result):

    # Remove the temporary directory.
    del self.__dir
```

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`¹⁷, which is itself derived from `Extension`. To create a new database class, you must define methods that read and write `Extension` instances.

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 `GetExtension` method which retrieves an instance of `Extension` given the name of the instance. If your database is modifiable, you must also provide `WriteExtension` and `RemoveExtension` methods. For historical reasons, your database class must also set the class variable `_is_generic_database` to `true`.

4.6 Registering and Distributing Extension Classes

To use your `Extension` class, you must place the Python module file containing it in a directory where QMTest can find it. QMTest searches for extensions in particular places, in the following order:

1. in the paths contained in the `QMTEST_CLASS_PATH` environment variable.
2. in the class-path associated with a particular test database (typically the `QMTest/` subdirectory).
3. in the site-extension path associated with a particular QMTest installation.
4. in the set of built-in extensions that come with a particular QMTest installation.

You should generally place module files containing extension 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.

To make your extension classes sharable by other test database instances, you should install them into the site-extension path associated with a qmtest installation.

To facilitate this installation, QMTest provides code that can be used in conjunction with python's distutils to install, and even distribute them. The following `setup.py` script will allow you to install (and even package !) all extension modules from the `extensions` subdirectory:

¹⁷ <http://www.codesourcery.com/public/qmtest/qmtest-2.4.1/internals/qm/test/database/Database.html>

```
from qm.dist.distribution import Distribution
from distutils.core import setup

setup(distclass=Distribution,
      qmtest_extensions="extensions")
```

Then, the following command installs extension modules from `extensions`:

```
> python setup.py install
```