

QMTTest: User's Guide and Reference

CodeSourcery, LLC

QMTTest: User's Guide and Reference

by CodeSourcery, LLC

Copyright © 2002 by CodeSourcery LLC

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 Software Carpentry Open Publication License, which is available at <http://sc-archive.codesourcery.com/openpub-license.html>.

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

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).

Revision History

Revision 1.0
Initial revision.

Table of Contents

1. Introduction.....	7
2. Getting Started with QMTest	9
2.1. Setting Up	9
2.2. Starting the Graphical Interface	10
2.3. Running Tests.....	10
2.4. Setting Expectations.....	11
2.5. Examining Tests	11
2.6. Modifying and Creating Tests	12
2.7. Using the Command-Line Interface.....	13
2.8. Expectations on the Command Line	15
2.9. Reviewing Results.....	16
3. Using QMTest.....	19
3.1. QMTest Concepts.....	19
3.1.1. Tests.....	19
3.1.2. Resources.....	20
3.1.3. Context	20
3.1.4. Test Results.....	21
3.1.4.1. Outcomes	21
3.1.4.2. Annotations.....	21
3.1.5. Test Suite	22
3.1.5.1. Implicit Test Suites	22
3.1.6. Test Database.....	23
3.1.7. Targets	23
3.2. Running Tests.....	24
3.2.1. Ordering and Dependencies	24
3.2.2. The Context	25
3.3. Test Database Contents	25
3.4. Invoking QMTest	26
3.4.1. qmtest	26
3.4.1.1. Synopsis	26
3.4.1.2. Options.....	26
3.4.2. qmtest create-tdb	27
3.4.2.1. Summary	27
3.4.2.2. Synopsis	27
3.4.2.3. Description.....	27
3.4.3. qmtest gui	28
3.4.3.1. Summary	28

3.4.3.2. Synopsis	28
3.4.3.3. Description	28
3.4.4. qmttest run	29
3.4.4.1. Summary	30
3.4.4.2. Synopsis	30
3.4.4.3. Description	30
3.4.5. qmttest summarize	32
3.4.5.1. Summary	32
3.4.5.2. Synopsis	32
3.4.5.3. Description	33
3.4.6. Environment Variables	33
3.5. Test and Resource Classes	34
3.5.1. Test Classes	34
3.5.1.1. <code>command.ExecTest</code>	34
3.5.1.2. <code>command.ShellCommandTest</code>	35
3.5.1.3. <code>command.ShellScriptTest</code>	36
3.6. Test Targets	37
3.6.1. Target Specification	37
3.6.2. Target Files	37
3.6.3. Target Classes	38
3.6.3.1. Thread Target	38
3.6.3.2. Remote Shell Target	39
3.7. Writing Test and Resource Classes	40
3.7.1. Writing Test Classes	40
3.7.1.1. The <code>arguments</code> Attribute	40
3.7.1.2. The <code>Run</code> Function	41
3.7.1.3. Using the Context	41
3.7.1.4. Creating Results	41
3.7.2. Writing Resource Classes	42
3.7.3. Specifying a Test or Resource Class	43
3.8. Writing a Database Class	43
4. The QM Configuration File	45
4.1. Configuration Variables	45

Chapter 1. Introduction

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 site (<http://www.qmtest.com>).

Chapter 2. Getting Started with QMTest

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

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

2.1. Setting Up

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

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

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

```
> xcopy c:\progra~1\qm\share\qm\tutorial\test\tdb tdb\ /s
```

If you installed QMTest in another directory, substitute that directory for `/usr/local` or `c:\progra~1\qm` in the commands above.

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

```
> cd tdb
```

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

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

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

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

On Windows, use:

```
> PATH C:\program~1\qm\bin;%PATH%
```

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

2.2. Starting the Graphical Interface

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

```
> qmttest gui
```

You will see output similar to:

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

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

The page you see in your browser shows the contents of the test database. You can see that there are three tests in the database, named `exec0`, `exec1`, and `exec2`. You can always click on the **Home** link in the upper-right corner to return to this page.

QMTest creates a traditional graphical user interface using your web browser. The black bar across the top of the browser window is QMTest's menu bar.

2.3. Running Tests

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

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

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

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

2.4. Setting Expectations

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

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

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

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

2.5. Examining Tests

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

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

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

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

2.6. Modifying and Creating Tests

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

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

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

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

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

Now, QMTest will present you with a form that looks just like the form you used to edit `exec1`, except that the arguments are different. The arguments are different because you're creating a different kind of test. Enter `echo` in the **Program** field. Click on the **Add** button to add a program argument and enter `test` in the popup dialog box. At this point, you've told qmtest that you want to run the command **echo test**. This command will produce an output (the word `test`) as output, so find the **Standard Output** box and enter `test` in this box. Make sure to hit the **Return** key after you type `test`; the **echo** command will output a carriage return after it prints the word `test`, so you must indicate that you expect a carriage return. When you are done, click the **OK** button at the bottom of the form.

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

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

2.7. Using the Command-Line Interface

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

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

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

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

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

```
> cd ..
> rm -rf tddb
```

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

On a Windows system, use these commands instead:

```
> cd ..
> rmdir /s tdb
> xcopy c:\progra~1\qm\share\qm\tutorial\test\tdb tdb\ /s
> cd tdb
```

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

```
> qmtest run
```

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

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

exec2                                     : PASS

exec0                                     : PASS

exec1                                     : FAIL
    Expression evaluates to false.

--- STATISTICS -----

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

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

exec1                                     : FAIL
    Expression evaluates to false.
```

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

2.8. Expectations on the Command Line

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

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

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

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

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

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

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

exec2                               : PASS
exec0                               : PASS
exec1                               : XFAIL
    Expression evaluates to false.

--- STATISTICS -----

    3      tests total
    3 (100%) tests as expected

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

None.
```

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

2.9. Reviewing Results

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

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

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

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

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

exec2                                     : PASS
exec0                                     : PASS
exec1                                     : FAIL
    Expression evaluates to false.

--- STATISTICS -----

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

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

exec1                                     : FAIL
    Expression evaluates to false.

ExecTest.expr:
    x + x == 5

ExecTest.value:
    0

qmtest.target:
```



```
local
```

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

Notes

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

Chapter 3. Using QMTest

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

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

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

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

3.1. QMTest Concepts

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

3.1.1. Tests

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

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

particular *directory*. For example, the test name `a.b.c` names a test named `c` in the directory `a.b`. The directory `a.b` is a subdirectory of the directory `a`.

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

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

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

3.1.2. Resources

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

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

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

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

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

3.1.3. Context

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

A *context* gives users a way of conveying this kind of information to tests. The context is a set of key/value pairs. The keys and values are always strings. In general, all tests in a given use of QMTest will have the same context. However, when a resource is set up, it may place additional information in the context of those tests that depend upon it; the tests can use this information to locate the resource.

3.1.4. Test Results

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

3.1.4.1. Outcomes

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

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

This outcome may also indicate a defect in QMTest or in the test class.

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

3.1.4.2. Annotations

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

3.1.5. Test Suite

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

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

3.1.5.1. Implicit Test Suites

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

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

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

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

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

```
> qmtest run .
```

is equivalent to:

```
> qmtest run
```

Both commands run all tests in the database.

3.1.6. Test Database

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

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

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

3.1.7. Targets

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

By default, QMTest uses only one target: the machine on which you are running QMTest. You may specify other targets by creating a target file, which lists the available targets and their attributes, and specifying the target file when you invoke **qmtest**. See Section 3.6.2 for details on writing and using target files.

Each target is a member of a single *target group*. All machines in the same target group are considered equivalent. A target group is specified by a string. If you are testing software on multiple platforms at once, the target group might correspond to machines running the same operating system. For example,

all Intel 80386 compatible machines running GNU/Linux might be in the “i386-pc-linux-gnu” target group.

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

3.2. Running Tests

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

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

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

3.2.1. Ordering and Dependencies

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

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

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

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

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

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

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

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

3.2.2. The Context

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

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

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

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

3.3. Test Database Contents

The default QMTest test database implementation stores the database as a directory hierarchy containing XML files. Each QMTest subdirectory is represented by a subdirectory in the filesystem. A test, suite, or

resource is represented by an XML file. These files have file extensions `.qmt`, `.qms`, and `.qma`, respectively.

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

3.4. Invoking QMTest

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

3.4.1. qmttest

3.4.1.1. Synopsis

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

3.4.1.2. Options

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

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

`-D path`

`--tdb path`

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

-h
--help

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

-v
--verbose

Generate progress and status messages while executing. This option may be specified more than once; the more times it is specified, the higher the verbose level, and the more messages are printed.

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.

3.4.2. qmtest create-tdb

3.4.2.1. Summary

Create a new test database.

3.4.2.2. Synopsis

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

3.4.2.3. 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. The path you specify for the new test database must not exist. (If no database path is specified, QMTest assumes that the current directory is the test database. Since the current directory already exists, QMTest will issue an error. Therefore, when using the **create-tdb** command, the database directory must be explicitly specified.)

By default, QMTest creates a new test database that uses the standard XML-based implementation. (See Section 3.8 for information about writing a test database class.)

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

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

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

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

Use the test database class given by *class*. Once you create a test database, you cannot change the test database implementation it uses. If you do not use this option, QMTest will use the default test database implementation, which uses an XML file format to store tests.

3.4.3. qmtest gui

3.4.3.1. Summary

Start the graphical user interface.

3.4.3.2. Synopsis

```
qmtest gui [ option ...]
```

3.4.3.3. 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:

`-A address`

`--address address`

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

`-c name=value`

`--context name=value`

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

`-C file`

`--load-context file`

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

`-j count`

`--concurrency count`

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

`--no-browser`

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

`--port port`

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

`-T file`

`--targets file`

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

3.4.4. qmtest run

3.4.4.1. Summary

Run tests or test suites.

3.4.4.2. Synopsis

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

3.4.4.3. Description

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

The **run** command accepts these options:

```
-c name=value
```

```
--context name=value
```

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

This option may be specified multiple times.

```
-c file
```

```
--load-context file
```

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

The file should be a text file with one context property on each line, in the format *name=value*.

Leading and trailing whitespace on each line are ignored. Also, blank lines and lines that begin with "#" (a hash mark) are ignored as comments.

This option may be specified more than once, and used in conjunction with the `--context` option.

`-f format`

`--format format`

Control the format used when displaying results. The format specified must be one of `full`, `brief`, `stats`, or `none`. The `brief` format is the default. 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 `none` mode, no results are displayed, but a results file is still created, unless the `--no-output` option is also provided.

`-j count`

`--concurrency count`

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

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

`--no-output`

Do not produce a test results file.

`-o file`

`--output file`

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

`-O file`

`--option file`

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

`-s file`

`--summary file`

Write a summary of the test run to *file*. Specify "-" (a hyphen) to write results to standard output

(the default).

QMTest prints a summary of test results, including statistics and the names of tests that did not pass. If expected outcomes were specified, the the names of tests that had unexpected results are printed, instead of the names of tests that did not pass.

`-S`

`--no-summary`

Do not produce a summary of the test run.

`--seed integer`

For each test run, QMTest randomizes the order in which tests are run, subject to the constraints described in Section 3.2.1. The random number generator is seeded using the system time. This maximizes the chance of detecting unanticipated dependencies among tests across multiple test runs.

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 on targets with a concurrency greater than one, 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*. See Section 3.6.2 for a description of the target file syntax.

3.4.5. qmtest summarize

3.4.5.1. Summary

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

3.4.5.2. Synopsis

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

3.4.5.3. Description

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

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

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

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

```
-o file  
--option file
```

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

3.4.6. Environment Variables

QMTest recognizes the following environment variables:

QM_PYTHON

If this environment variable is set, QMTest uses it as the path to the Python interpreter. If this environment variable is not set, QMTest looks for a file named `python` in the `bin` directory where QM is installed. If this file does not exist, but `/usr/bin/python2` exists, QMTest will use that path. Otherwise, QMTest searches for `python` in the directories listed in the `PATH` environment variable.

QMTEST_CLASS_PATH

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

QMTEST_DB_PATH

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

3.5. Test and Resource Classes

This section describes test classes and resource classes included with QMTest. Section 3.7.1 provides instructions for writing your own test classes, Section 3.7.2 for resource classes.

3.5.1. Test Classes

3.5.1.1. `command.ExecTest`

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

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

Program (text field)

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

Argument List (set of strings)

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

Standard Input (text field)

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

Environment (set of strings)

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

`command.ExecTest` adds additional environment variables automatically. Each context property is accessible as an environment variable; the name of the environment variable is the name of the context property, prefixed with "QMV_".

Expected Exit Code (integer field)

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

Expected Standard Output (text field)

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

3.5.1.2. `command.ShellCommandTest`

`command.ShellCommandTest` is very similar to `command.ExecTest`, except that it runs a program via the shell rather than directly. Instead of specifying an executable to run and the elements of its

argument list, a test provides a single command line. The shell is responsible for finding the executable and constructing its argument list.

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

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

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

Command (text field)

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

3.5.1.3. `command.ShellScriptTest`

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

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

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

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

Script (text field)

The text of the script to run.

3.6. Test Targets

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

3.6.1. Target Specification

Each target specification includes the following:

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

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

QMTest includes several target class implementations. See Section 3.6.3 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. A *concurrency* value, which must be a positive integer. Most target classes support concurrent execution of multiple tests on the target. This number allows the target specification to control how many tests are executed simultaneously.
5. Optionally, a target specification may include additional properties. Properties are named and have string values. Some target classes may use property information to control their configuration. For instance, a target class which executes tests on a remote computer would extract the network address of the remote computer from a target property.

3.6.2. Target Files

In a *target file*, you specify the computers or other test targets on which QMTest runs tests. Use the `--targets (-T)` option to the **qmttest run** command to specify the target file.

A target file is an XML document containing elements representing target specifications. The document type is `--Software Carpentry//QMTest Target V0.1//EN`, and the document element is `targets`.

Each target specification is represented by a target element, and includes name, class, group, and concurrency elements. Additional property elements may provide target properties.

For example, the following target file specifies two targets for running tests.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE targets PUBLIC "-//Software Carpentry//QMTest Target V0.1//EN" "http://www.softwa
carpentry.com/qm/xml/target.dtd">
<targets>
  <target>
    <name>local</name>
    <class>thread_target.ThreadTarget</class>
    <group>i386-pc-linux-gnu</group>
    <concurrency>1</concurrency>
  </target>
  <target>
    <name>remote</name>
    <class>rsh_target.RSHTarget</class>
    <group>sparc-sun-solaris2.5.1</group>
    <concurrency>2</concurrency>
    <property name="host">sunshine</property>
    <property name="arguments">-l test</property>
  </target>
</targets>
```

The first target is the local GNU/Linux computer. The second target is a remote computer running SPARC Solaris. QMTest invokes tests on the remote target using a remote shell invocation. The remote computer's name is "sunshine", and two tests are run concurrently on that target.

3.6.3. Target Classes

QMTest includes these target class implementations.

3.6.3.1. Thread Target

The `thread_target.ThreadTarget` target class runs tests in one or more threads on the local computer (the computer on which the **qmttest** command is invoked). The number of threads is the degree of concurrency specified for the target.

3.6.3.2. Remote Shell Target

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 **qmttest** 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.

The concurrency value of the target specification controls the number tests that may be run concurrently on the remote host.

The `RSHTarget` target class takes its configuration from the following target properties:

- The `remote_shell` property specifies the path to the remote shell program. If omitted, the configuration variable `remote_shell` is used instead. If neither is specified, the default is `/usr/bin/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.

- QMTest uses the remote shell to invoke the **qmttest_remote** program on the remote host. This program is installed by default in `/usr/local/bin/qmttest`. You may override this and use a version in a different place by specifying the `qmttest_remote` property.

3.7. Writing Test and Resource Classes

You can implement many tests using the test classes supplied with QMTest. However, expert test implementors may wish to create new test classes to customize existing tests or implement new testing behavior. In addition, implementors will wish to create resource classes, to add customized setup and cleanup behavior to tests and groups of tests.

3.7.1. Writing Test Classes

In QMTest, a test class is represented by a Python class. The class must inherit from `qm.test.test.Test`. The class must include two things: an `arguments` attribute, whose value is a sequence of field objects, and a `Run` function.

3.7.1.1. The `arguments` Attribute

The test class must include an `arguments` attribute, indicating the types of the test class's parameters. The `arguments` attribute's value should be a sequence consisting of field objects. A field object is an instance of a subclass of `qm.fields.Field`. The names of the arguments (specified by the "name" attribute of the field object) are the names of the parameters of the test class.

For instance, this definition of the `arguments` attribute declares two parameters for the test class. One parameter, called "input_text", takes a text value. The other parameter, called "value_list", takes a set of integers.

```
arguments = [
    qm.fields.IssueFieldText(
        name="input_text"),
    qm.fields.IssueFieldSet(qm.fields.IssueFieldInteger(
        name="value_list")),
]
```


Test classes should also initialize the `title` and `description` attributes of each field. The values of these attributes help users of the test class identify the purpose and semantics of each of the test class's parameters.

3.7.1.2. The `Run` Function

The heart of the test class is the `Run` function. This function runs the test and produces a test result.

The `Run` function takes two arguments: the context and the result. The context object satisfies the interface of the `qm.test.context.Context` Python class (though it may in actuality be an instance of a different Python class).

The result object is an instance of `qm.test.result.Result`. By default, the result will indicate that the test passes. If the test fails, the test class should call the `Fail` method on the result to indicate failure.

If the `Run` raises an unhandled exception, QMTest creates a result for the test with the outcome `ERROR`. Test classes should be designed so that they do not raise unhandled exceptions in the course of normal use (including test failures). An unhandled exception should reflect an internal error in the implementation of the test class.

3.7.1.3. Using the Context

A context object is simply a dictionary of properties. Use Python's map syntax to access a context's properties. A property key is always a string composed of letters, digits, hyphens, underscores, and periods. Property values are strings.

A test's `Run` function only sees context properties added by QMTest itself and properties added by the setup functions of required resources.

3.7.1.4. Creating Results

In case of a `FAIL` result, it is conventional to assign the `Result.CAUSE` property a string value providing a description of why the test failed. These two equivalent examples demonstrate how to indicate test failure (both assume that the `qm.test.result.Result` class has been imported into the module's global namespace).

```
result.Fail()
result[Result.CAUSE] = "Unexpected end of input."
```

or

```
result.Fail("Unexpected end of input.")
```

Sometimes, a Run function implementation detects a failure by catching an exception. The method `Result.NoteException` provides a convenient mechanism for creating a result that includes information about the exception. For example:

```
try:
    # ... run test code here ...
except EndOfFileError:
    result.NoteException()
```

The `NoteException` method will automatically add annotations describing the cause of the exception.

3.7.2. Writing Resource Classes

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

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

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

```
def SetUp(self, context, result):
    try:
        handle = create_my_resource()
        self.__handle = handle
    except:
        result.NoteException()
    else:
        context["resource_handle"] = str(handle)

def CleanUp(self, result):
    try:
```

```

        destroy_my_resource(self.__handle)
    except:
        result.NoteException()

```

3.7.3. Specifying a Test or Resource Class

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

- If the environment variable QMTEST_CLASS_PATH is defined, QMTest first checks any directories listed in it. This value of this environment variable should be a list of directories to check for the module file, in the same format as the standard PATH environment variable.
- A test database may specify locations to check for module files when loading a class. QMTest test database implementations generally check the subdirectory named QMTest in the top test database directory for test and resource classes. (Note that QMTest may place other files in this directory; you should not disturb them.) Module files containing test and resource classes may be placed in this directory.
- Finally, QMTest checks a standard directory. This directory, installed with QMTest, contains modules with the standard test classes described in Section 3.5.

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

Assuming the Python module file containing the test class is located in one of the directories specified above, you can refer to it using the syntax `module.Class`, where `module` is the name of the module and `Class` is the name of the class.

3.8. Writing a Database Class

The 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 `qm.test.database.Database`. You must define methods that handle retrieving a test from the database, writing a test to the database, and other related tasks. Read the code for `qm.test.database.Database` to see what methods need to be overridden.

To use your new database class, create a new test database using the `--class (-c)` option.

Chapter 4. The QM Configuration File

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

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

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

Here is a sample QM configuration file:

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

4.1. 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 makes an attempt to locate a browser on your computer.

command_shell

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

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

script_shell

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

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

sendmail (UNIX-like platforms only)

The path to the `sendmail` program, or a compatible replacement. This program is used for sending email messages. The default value is `/usr/lib/sendmail`.

remote_shell (UNIX-like platforms only)

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