Contents

Previous

Qt Test

Reference

Getting Started

Overviews

Qt Test Best Practices

We recommend that you add Qt tests for bug fixes and new features. Before you try to fix a bug, add a *regression test* (ideally automatic) that fails before the fix, exhibiting the bug, and passes after the fix. While you're developing new features, add tests to verify that they work as intended.

Conforming to a set of coding standards will make it more likely for Qt autotests to work reliably in all environments. For example, some tests need to read data from disk. If no standards are set for how this is done, some tests won't be portable. For example, a test that assumes its test-data files are in the current working directory only works for an in-source build. In a shadow build (outside the source directory), the test will fail to find its data.

The following sections contain guidelines for writing Qt tests:

General Principles

The following sections provide general guidelines for writing unit tests:

Verify Tests

Write and commit your tests along with your fix or new feature on a new branch. Once you're done, you can check out the branch on which your work is based, and then check out into this branch the test-files for your new tests. This enables you to verify that the tests do fail on the prior branch, and therefore actually do catch a bug or test a new feature.

For example, the workflow to fix a bug in the `QDateTime` class could be like this if you use the Git version control system:

1. Create a branch for your fix and test: `git checkout -b fix-branch 5.14`
2. Write a test and fix the bug.
3. Build and test with both the fix and the new test, to verify that the new test passes with the fix.
4. Add the fix and test to your branch: `git add tests/auto/corelib/time/qdatetime/tst_qdatetime.cpp src/corelib/time/qdatetime.cpp`
5. Commit the fix and test to your branch: `git commit -m 'Fix bug in QDateTime'`
6. To verify that the test actually catches something for which you needed the fix, checkout the branch you based your own branch on: `git checkout 5.14`
7. Checkout only the test file to the 5.14 branch: `git checkout fix-branch -- tests/auto/corelib/time/qdatetime/tst_qdatetime.cpp`

   Only the test is now on the fix-branch. The rest of the source tree is still on 5.14.
8. Build and run the test to verify that it fails on 5.14, and therefore does indeed catch a bug.
9. You can now return to the fix branch: `git checkout fix-branch`
10. Alternatively, you can restore your work tree to a clean state on 5.14: `git checkout HEAD -- tests/auto/corelib/time/qdatetime/tst_qdatetime.cpp`

When you're reviewing a change, you can adapt this workflow to check that the change does indeed come with a test for a problem it does fix.

Give Test Functions Descriptive Names

Naming test cases is important. The test name appears in the failure report for a test run. For data-driven tests

Naming test cases is important. The test name appears in the failure report for a test run. For data-driven tests, the name of the data row also appears in the failure report. The names give those reading the report a first indication of what has gone wrong.

Test function names should make it obvious what the function is trying to test. Do not simply use the bug-tracking identifier, because the identifiers become obsolete if the bug-tracker is replaced. Also, some bug-trackers may not be accessible to all users. When the bug report may be of interest to later readers of the test code, you can mention it in a comment alongside a relevant part of the test.

Likewise, when writing data-driven tests, give descriptive names to the test-cases, that indicate what aspect of the functionality each focuses on. Do not simply number the test-case, or use bug-tracking identifiers. Someone reading the test output will have no idea what the numbers or identifiers mean. You can add a comment on the test-row that mentions the bug-tracking identifier, when relevant.

Write Self-contained Test Functions

Within a test program, test functions should be independent of each other and they should not rely upon previous test functions having been run. You can check this by running the test function on its own with `tst_foo testname`.

Do not re-use instances of the class under test in several tests. Test instances (for example widgets) should not be member variables of the tests, but preferably be instantiated on the stack to ensure proper cleanup even if a test fails, so that tests do not interfere with each other.

Test the Full Stack

If an API is implemented in terms of pluggable or platform-specific backends that do the heavy-lifting, make sure to write tests that cover the code-paths all the way down into the backends. Testing the upper layer API parts using a mock backend is a nice way to isolate errors in the API layer from the backends, but it is complementary to tests that run the actual implementation with real-world data.

Make Tests Complete Quickly

Tests should not waste time by being unnecessarily repetitious, by using inappropriately large volumes of test data, or by introducing needless idle time.

This is particularly true for unit testing, where every second of extra unit test execution time makes CI testing of a branch across multiple targets take longer. Remember that unit testing is separate from load and reliability testing, where larger volumes of test data and longer test runs are expected.

Benchmark tests, which typically execute the same test multiple times, should be located in a separate `tests/benchmarks` directory and they should not be mixed with functional unit tests.

Use Data-driven Testing

Data-driven tests make it easier to add new tests for boundary conditions found in later bug reports.

Using a data-driven test rather than testing several items in sequence in a test saves repetition of very similar code and ensures later cases are tested even when earlier ones fail. It also encourages systematic and uniform testing, because the same tests are applied to each data sample.

Use Coverage Tools

Use a coverage tool such as Froglogic Coco Code Coverage or gcov to help write tests that cover as many statements, branches, and conditions as possible in the function or class being tested. The earlier this is done in the development cycle for a new feature, the easier it will be to catch regressions later when the code is

refactored.

Select Appropriate Mechanisms to Exclude Tests

It is important to select the appropriate mechanism to exclude inapplicable tests: QSKIP(), using conditional statements to exclude parts of a test function, or not building the test for a particular platform.

Use QSKIP() to handle cases where a whole test function is found at run-time to be inapplicable in the current test environment. When just a part of a test function is to be skipped, a conditional statement can be used, optionally with a `qDebug()` call to report the reason for skipping the inapplicable part.

Test functions or data rows of a data-driven test can be limited to particular platforms, or to particular features being enabled using `#if`. However, beware of moc limitations when using `#if` to skip test functions. The `moc` preprocessor does not have access to all the `builtin` macros of the compiler that are often used for feature detection of the compiler. Therefore, `moc` might get a different result for a preprocessor condition from that seen by the rest of your code. This may result in `moc` generating meta-data for a test slot that the actual compiler skips, or omitting the meta-data for a test slot that is actually compiled into the class. In the first case, the test will attempt to run a slot that is not implemented. In the second case, the test will not attempt to run a test slot even though it should.

If an entire test program is inapplicable for a specific platform or unless a particular feature is enabled, the best approach is to use the parent directory's `.pro` file to avoid building the test. For example, if the `tests/auto/gui/someclass` test is not valid for macOS, add the following line to `tests/auto/gui.pro`:

```
mac*: SUBDIRS -= someclass
```

Avoid Q_ASSERT

The Q_ASSERT macro causes a program to abort whenever the asserted condition is `false`, but only if the software was built in debug mode. In both release and debug-and-release builds, Q_ASSERT does nothing.

Q_ASSERT should be avoided because it makes tests behave differently depending on whether a debug build is being tested, and because it causes a test to abort immediately, skipping all remaining test functions and returning incomplete or malformed test results.

It also skips any tear-down or tidy-up that was supposed to happen at the end of the test, and might therefore leave the workspace in an untidy state, which might cause complications for further tests.

Instead of Q_ASSERT, the QCOMPARE() or QVERIFY() macro variants should be used. They cause the current test to report a failure and terminate, but allow the remaining test functions to be executed and the entire test program to terminate normally. QVERIFY2() even allows a descriptive error message to be recorded in the test log.

Writing Reliable Tests

The following sections provide guidelines for writing reliable tests:

- Avoid Side-effects in Verification Steps
- Avoid Fixed Timeouts
- Beware of Timing-dependent Behavior
- Avoid Bitmap Capture and Comparison

Avoid Side-effects in Verification Steps

When performing verification steps in an autotest using QCOMPARE(), QVERIFY(), and so on, side-effects should be avoided. Side-effects in verification steps can make a test difficult to understand. Also, they can easily break a test in ways that are difficult to diagnose when the test is changed to use QTRY_VERIFY(), QTRY_COMPARE() or QBENCHMARK(). These can execute the passed expression multiple times, thus repeating any side-effects.

When side-effects are unavoidable, ensure that the prior state is restored at the end of the test function, even if the test fails. This commonly requires use of an RAII (resource acquisition is initialization) class that restores state when the function returns, or a `cleanup()` method. Do not simply put the restoration code at the end of the test. If part of the test fails, such code will be skipped and the prior state will not be restored.

Avoid Fixed Timeouts

Avoid using hard-coded timeouts, such as QTest::qWait() to wait for some conditions to become true. Consider using the QSignalSpy class, the QTRY_VERIFY() or QTRY_COMPARE() macros, or the `QSignalSpy` class in conjunction with the `QTRY_` macro variants.

The `qWait()` function can be used to set a delay for a fixed period between performing some action and waiting for some asynchronous behavior triggered by that action to be completed. For example, changing the state of a widget and then waiting for the widget to be repainted. However, such timeouts often cause failures when a test written on a workstation is executed on a device, where the expected behavior might take longer to complete. Increasing the fixed timeout to a value several times larger than needed on the slowest test platform is not a good solution, because it slows down the test run on all platforms, particularly for table-driven tests.

If the code under test issues Qt signals on completion of the asynchronous behavior, a better approach is to use the QSignalSpy class to notify the test function that the verification step can now be performed.

If there are no Qt signals, use the `QTRY_COMPARE()` and `QTRY_VERIFY()` macros, which periodically test a specified condition until it becomes true or some maximum timeout is reached. These macros prevent the test from taking longer than necessary, while avoiding breakages when tests are written on workstations and later executed on embedded platforms.

If there are no Qt signals, and you are writing the test as part of developing a new API, consider whether the API could benefit from the addition of a signal that reports the completion of the asynchronous behavior.

Beware of Timing-dependent Behavior

Some test strategies are vulnerable to timing-dependent behavior of certain classes, which can lead to tests that fail only on certain platforms or that do not return consistent results.

One example of this is text-entry widgets, which often have a blinking cursor that can make comparisons of captured bitmaps succeed or fail depending on the state of the cursor when the bitmap is captured. This, in turn, may depend on the speed of the machine executing the test.

When testing classes that change their state based on timer events, the timer-based behavior needs to be taken into account when performing verification steps. Due to the variety of timing-dependent behavior, there is no single generic solution to this testing problem.

For text-entry widgets, potential solutions include disabling the cursor blinking behavior (if the API provides that feature), waiting for the cursor to be in a known state before capturing a bitmap (for example, by subscribing to an appropriate signal if the API provides one), or excluding the area containing the cursor from the bitmap comparison.

Avoid Bitmap Capture and Comparison

While verifying test results by capturing and comparing bitmaps is sometimes necessary, it can be quite fragile and labor-intensive.

For example, a particular widget may have different appearance on different platforms or with different widget styles, so reference bitmaps may need to be created multiple times and then maintained in the future as Qt's set of supported platforms evolves. Making changes that affect the bitmap thus means having to recreate the expected bitmaps on each supported platform, which would require access to each platform.

Bitmap comparisons can also be influenced by factors such as the test machine's screen resolution, bit depth, active theme, color scheme, widget style, active locale (currency symbols, text direction, and so on), font size, transparency effects, and choice of window manager.

Where possible, use programmatic means, such as verifying properties of objects and variables, instead of capturing and comparing bitmaps.

Improving Test Output

The following sections provide guidelines for producing readable and helpful test output:

- Explicitly Ignore Expected Warnings
- Avoid Printing Debug Messages from Autotests
- Write Well-structured Diagnostic Code

Explicitly Ignore Expected Warnings

If a test is expected to cause Qt to output a warning or debug message on the console, it should call QTest::ignoreMessage() to filter that message out of the test output and to fail the test if the message is not output.

If such a message is only output when Qt is built in debug mode, use QLibraryInfo::isDebugBuild() to determine whether the Qt libraries were built in debug mode. Using `#ifdef QT_DEBUG` is not enough, as it will only tell you whether the test was built in debug mode, and that does not guarantee that the Qt libraries were also built in debug mode.

Avoid Printing Debug Messages from Autotests

Autotests should not produce any unhandled warning or debug messages. This will allow the CI Gate to treat new warning or debug messages as test failures.

Adding debug messages during development is fine, but these should be either disabled or removed before a test is checked in.

Write Well-structured Diagnostic Code

Any diagnostic output that would be useful if a test fails should be part of the regular test output rather than being commented-out, disabled by preprocessor directives, or enabled only in debug builds. If a test fails during continuous integration, having all of the relevant diagnostic output in the CI logs could save you a lot of time compared to enabling the diagnostic code and testing again. Epecially, if the failure was on a platform that you don't have on your desktop.

Diagnostic messages in tests should use Qt's output mechanisms, such as `qDebug()` and `qWarning()`, rather than `stdio.h` or `iostream.h` output mechanisms. The latter bypass Qt's message handling and

prevent the `-silent` command-line option from suppressing the diagnostic messages. This could result in important failure messages being hidden in a large volume of debugging output.

Writing Testable Code

The following sections provide guidelines for writing code that is easy to test:

- Break Dependencies
- Compile All Classes into Libraries

Break Dependencies

The idea of unit testing is to use every class in isolation. Since many classes instantiate other classes, it is not possible to instantiate one class separately. Therefore, you should use a technique called *dependency injection* that separates object creation from object use. A factory is responsible for building object trees. Other objects manipulate these objects through abstract interfaces.

This technique works well for data-driven applications. For GUI applications, this approach can be difficult as objects are frequently created and destructed. To verify the correct behavior of classes that depend on abstract interfaces, *mocking* can be used. For example, see Googletest Mocking (gMock) Framework.

Compile All Classes into Libraries

In small to medium sized projects, a build script typically lists all source files and then compiles the executable in one go. This means that the build scripts for the tests must list the needed source files again.

It is easier to list the source files and the headers only once in a script to build a static library. Then the `main()` function will be linked against the static library to build the executable and the tests will be linked against the static libraries.

For projects where the same source files are used in building several programs, it may be more appropriate to build the shared classes into a dynamically-linked (or shared object) library that each program, including the test programs, can load at run-time. Again, having the compiled code in a library helps to avoid duplication in the description of which components to combine to make the various programs.

Setting up Test Machines

The following sections discuss common problems caused by test machine setup:

- Screen Savers
- System Dialogs
- Display Usage
- Window Managers

All of these problems can typically be solved by the judicious use of virtualisation.

Screen Savers

Screen savers can interfere with some of the tests for GUI classes, causing unreliable test results. Screen savers should be disabled to ensure that test results are consistent and reliable.

System Dialogs

System Dialogs

Dialogs displayed unexpectedly by the operating system or other running applications can steal input focus from widgets involved in an autotest, causing unreproducible failures.

Examples of typical problems include online update notification dialogs on macOS, false alarms from virus scanners, scheduled tasks such as virus signature updates, software updates pushed out to workstations, and chat programs popping up windows on top of the stack.

Display Usage

Some tests use the test machine's display, mouse, and keyboard, and can thus fail if the machine is being used for something else at the same time or if multiple tests are run in parallel.

The CI system uses dedicated test machines to avoid this problem, but if you don't have a dedicated test machine, you may be able to solve this problem by running the tests on a second display.

On Unix, one can also run the tests on a nested or virtual X-server, such as Xephyr. For example, to run the entire set of tests on Xephyr, execute the following commands:

```
Xephyr :1 -ac -screen 1920x1200 >/dev/null 2>&1 &
sleep 5
DISPLAY=:1 icewm >/dev/null 2>&1 &
cd tests/auto
make
DISPLAY=:1 make -k -j1 check
```

Users of NVIDIA binary drivers should note that Xephyr might not be able to provide GLX extensions. Forcing Mesa libGL might help:

```
export LD_PRELOAD=/usr/lib/mesa-diverted/x86_64-linux-gnu/libGL.so.1
```

However, when tests are run on Xephyr and the real X-server with different libGL versions, the QML disk cache can make the tests crash. To avoid this, use `QML_DISABLE_DISK_CACHE=1`.

Alternatively, use the offscreen plugin:

```
TESTARGS="-platform offscreen" make check -k -j1
```

Window Managers

On Unix, at least two autotests (`tst_examples` and `tst_gestures`) require a window manager to be running. Therefore, if running these tests under a nested X-server, you must also run a window manager in that X-server.

Your window manager must be configured to position all windows on the display automatically. Some windows managers, such as Tab Window Manager (twm), have a mode for manually positioning new windows, and this prevents the test suite from running without user interaction.

Note: Tab Window Manager is not suitable for running the full suite of Qt autotests, as the `tst_gestures`

autotest causes it to forget its configuration and revert to manual window placement.