Test It!

Unit testing for lazy developers

Andreas Cord-Landwehr
cordlandwehr@kde.org
Next up...

1. **Part 1: Test Theory**

2. **Part 2: Test Frameworks in KDE**

3. **Part 3: KDE Build & Test Infrastructure**

4. **The End**
Hey, I did test it and it looks fine!

Good reasons why you want automatic tests:

1. You do not waste time with testing manually. – Again. – And again.
2. Errors are easier to find & to fix if the test is small.
3. Refactoring without tests is THE WAY that leads to regressions.
4. Tests assert behavior that is only written down in documentation (if written down at all).
5. When writing integration tests for a library, you are using the API yourself and see if it is usable.
White Box vs. Black Box

**White Box Test**  You are looking into the class and know its implementation:
- unit tests are typical white box tests
- “you know what might cause trouble, you add checks”
- test scope: only one/a few tightly coupled classes

**Black Box Test**  You are looking at the (public) API and features:
- focus is on the features that are promised
- asserts the contract to the outside world
Test Types and Goals
My Definition for System under test

Unit Tests for yourself, to ensure that implementation works
Integration Tests for API user, to ensure that promised functionality works
Subsystem Tests for end-user, to ensure that feature works

System under test
Test-Driven Development (TDD)

- first write a test, then write the code
- then write just enough code that the test passes and repeat
- thus every method is covered by at least one test
- usually results in better code:
  - **Code Quality** you have to think about testability and simplifying the code when writing it
  - **Architecture** leads to better decoupling of classes by interfaces and compositions
  - **Refactoring** you document your assumptions via tests and tests will tell you if still everything works
Next up...

1. Part 1: Test Theory

2. Part 2: Test Frameworks in KDE

3. Part 3: KDE Build & Test Infrastructure

4. The End
QTest

- Lightweight and easy to use framework!
- Every test is an application, which may contain several test cases.
- Test class is a QObject and every Q SLOT is interpreted as a test case.
- Provides many convenient test methods:
  - QVERIFY: Test if expression is true.
  - QCOMPARE: Test if expressions are equal (and state differences if not).
  - QSignalSpy: Test if a signal is received and allows to check its parameters.

Documentation
- All macros: https://doc.qt.io/qt-5/qtest.html
# QTest: Basic Example

```cpp
#include <QTest>

class SimpleTest : public QObject {
    Q_OBJECT

private Q_SLOTS:
    void initTestCase() { /* called before anything else */ }
    void myTest() {
        QVERIFY(true);
        QCOMPARE(1, 1);
    }
}

QTEST_MAIN(SimpleTest)
```

**Special Slots:**
- `initTestCae():` called **before first** test function
- `cleanupTestCae():` called **after last** test function
- `init():` called **before each** test function
- `cleanup():` called **after each** test function
QTest: Test Emitted Signals

Listing 1: Simple

1. `QPushButton myPushButton;
2. QSignalSpy spy(&myPushButton, &QPushButton::clicked);
3. QVERIFY(spy.isValid());
4. myPushButton.click();
5. QCOMPARE(spy.count(), 1);
6. QList<QVariant> arguments = spy.takeFirst();
7. QCOMPARE(arguments.at(0).toBool(), false);

Listing 2: Concurrency

1. QVERIFY(spy.wait(1000)); // start event loop with 1s timeout
#include <QTest>

class SimpleTest : public QObject
{
    Q_OBJECT

private Q_SLOTS:

    void myTest_data()
    {
        QTest::addColumn<QString>("original");
        QTest::addColumn<QString>("target");
        QTest::newRow("first case") << "foo" << "foo_target";
        QTest::newRow("second case") << "baa" << "baa_target";
    }

    void myTest()
    {
        QFETCH(QString, original);
        QFETCH(QString, target);
        Q_EXPECT_FAIL();
        QCOMPARE(original, target);
    }

QTEST_MAIN(SimpleTest)
QTest: Test QtQuick Bindings

Example for test method with focus on QML Engine interaction:

```cpp
void bindingTest() {
    // test.qml has root object with property "testProperty"
    QUrl input = QUrl::fromLocalFile(QFINDTESTDATA("test.qml"));
    QQmlEngine engine;
    QQmlComponent component(&engine, input, QQmlComponent::PreferSynchronous);
    QObject *object = component.create();
    if (!object) {
        qDebug() << "errors:" << component.errors();
    }
    QVERIFY(object);
    QVERIFY(!component.isLoading());
    QCOMPARE(object->property("testProperty").toString(), "foo");
}
```

There is also QtQuickTest for interactive tests, but can be tricky to use.
Example for test method with focus on QML Engine interaction:

```c++
void bindingTest() {
    // test.qml has root object with property "testProperty"
    QUrl input = QUrl::fromLocalFile(QFINDTESTDATA("test.qml"));
    QQmlEngine engine;
    QQmlComponent component(&engine, input, QQmlComponent::PreferSynchronous);
    QObject *object = component.create();
    if (!object) {
        qDebug() << "errors:" << component.errors();
    }
    QVERIFY(object);
    QVERIFY(!component.isLoading());
    QCOMPARE(object->property("testProperty").toString(), "foo");
}
```

There is also QtQuickTest for interactive tests, but can be tricky to use.
Test Design

Never ever...

1. make one unit test depending on another one
2. test production code, rather than setting up a fake
3. make your tests slow
4. create tests suits to test third party code (if you do not trust, don’t use it)
5. create a test that need dozens of cpp files compiled into it, because then you are missing interfaces and mocks

Good Advice

- If you never spent time learning about software patterns, do it now!
- There is a good reason, why there are mocks, stubs and fakes :)
Test Design

Never ever...

1. make one unit test depending on another one
2. test production code, rather than setting up a fake
3. make your tests slow
4. create tests suits to test third party code (if you do not trust, don’t use it)
5. create a test that need dozens of cpp files compiled into to, because then you are missing interfaces and mocks

Good Advice

- If you never spent time learning about software patterns, do it now!
- There is a good reason, why there are mocks, stubs and fakes :)
Next up...

1. Part 1: Test Theory
2. Part 2: Test Frameworks in KDE
3. Part 3: KDE Build & Test Infrastructure
4. The End
Integrate Tests into your Build System

CMake has its own testing tool: CTest

- CTest executes your QTest tests and reports results

Steps to Integrate:

1. in your main CMakeLists.txt: include(ECMAddTests)
2. use ecm_add_test macro:

```cpp
ecm_add_test(<sources> LINK_LIBRARIES <library> [...][[<library> [...] [[TEST_NAME <name>]]]
[[NAME_PREFIX <prefix>]]
[GUI])
```

[https://api.kde.org/ecm/module/ECMAddTests.html](https://api.kde.org/ecm/module/ECMAddTests.html)
Executing CTest

Execute in your build directory:

- ctest -N: List all available tests
- ctest -R: Run all tests
- ctest -R -V: Run all tests and print information on problems
- ctest -R --output-on-failure: Runs all tests and gives output for failed tests
- ctest -R foo --output-on-failure: Runs all tests with “foo” in their name and gives output for failed tests

make test: Runs: /usr/bin/ctest --force-new-ctest-process
We have some great CI tooling:

- all (non-playground) projects run on the CI
- building is checked against various architectures
- tests are run on most of this architectures
- this gives an important safety net to see if everything works outside of your own system!
- task for today: check the status of your project! ;)

Keep in mind: Never merge a test that fails on your system.
Test Coverage Computation GCov

Sometimes it is good to not only rely on your feelings...

- GCov is a tool to log which code is executed during a test
- results are generated in CI
- it shows you areas that are forgotten by tests
Next up...

1. Part 1: Test Theory
2. Part 2: Test Frameworks in KDE
3. Part 3: KDE Build & Test Infrastructure
4. The End
The End

Question Time

Contact

mail: cordlandwehr@kde.org
irc: CoLa