LH3 C++ Unit Testing

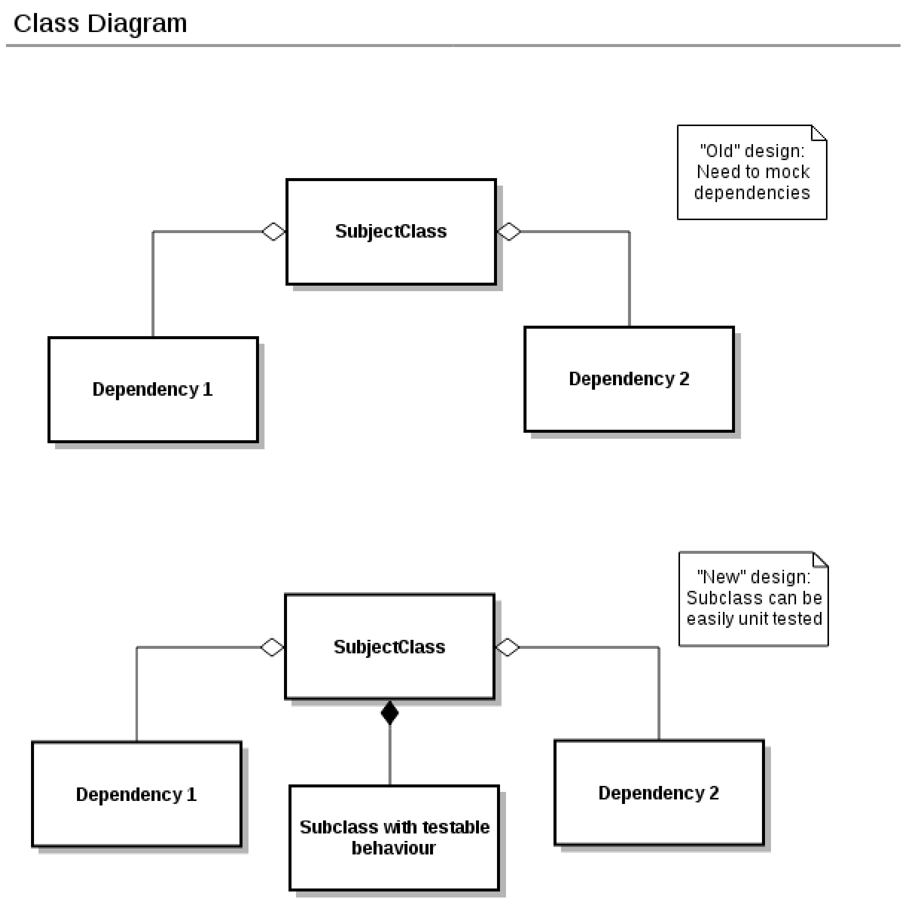
Introduction

This document describes the way we're going to do unit testing on the LightHouse III project.

Basic Approach

We're using QTest as our test framework because it's lightweight and integrates nicely with Qt Creator

To avoid the Mock problems we had on the Atom project we're going to try to create independent subclasses and unit test them. These testable classes won't call out directly to other classes - this means we don't have to mock dependencies. Data input should be pushed via function calls, and output via parameters passed by reference, return values and possibly Qt signals:


Class Diagram
"Old" design: Need to mock dependencies
SubjectClass
Dependency 1
Dependency 2
"New" design: Subclass can be easily unit tested
SubjectClass
Dependency 1
Subclass with testable behaviour
Dependency 2

QTest Design

- For each testable module create a subdirectory under apollo.app/Tests
- Create a module .pro file (see ChartTests.pro for an example)
- Add your new test module to the Tests.pro file
- Add the Tests/Common/UnitTestMain class
- Add each class unit test file to the ChartTests.pro file
- Add a runTests() function to your test.cpp and register this with UnitTestMain so it gets run (see ExampleTest.cpp)
- Your class unit test file should declare the unit test class and a set of private Qt slots, with a test in each slot (this is the way QTest works).

- If you don't have a header file for your class unit test file, you may need to manually run qmake to generate the moc file for your class.

Running The Tests

Given that each module's test.pro file defines an executable console app that runs the unit tests, there are a number of ways of running this.

1. Add the project as a run target in QtCreator. Then run the tests by pressing the green arrow icon, or Build --> Run. This is the normal way for a developer to work, while maintaining code, or using TDD to write new code.Tests can be run in debug mode using the green arrow with a bug or Debug --> Start Debugging but QML debugging must be disabled. To disable QML debugging open Projects -> Run Settings then in Debugger Settings clear the Enable QML check box.
2. Run all the unit tests by invoking apollo.app/run_unit_tests.sh
3. Run the executable from a command prompt. This usually results in library load errors. Qt provides a target_wrapper.sh script to overcome this (in each build directory) but this also seems not to work.

Tests.pro file

```
TEMPLATE = subdirs

SUBDIRS += \
    ChartTests \
    LightHouseTests \
```

ChartTests.pro

```
QT += testlib

QT -= gui

TARGET = ChartTests
CONFIG += warn_on console
CONFIG -= app_bundle

# for 'make check'
# http://doc.qt.io/qt-5/qmake-common-projects.html#building-a-testcase
CONFIG += testcase

TEMPLATE = app

HEADERS *= ../Common/UnitTestMain.h
SOURCES *= ../Common/UnitTestMain.cpp
SOURCES *= ../Common/Rx5ChartClassFactoryStub.cpp # break dependency on Rx5libs

HEADERS *= ExampleUUT.h
SOURCES *= ExampleUUT.cpp

SOURCES *= ExampleTest.cpp
SOURCES *= ChartRangeIndicatorTest.cpp
SOURCES *= ChartRangeRingTableTest.cpp
SOURCES *= VesselSourceImageTableTest.cpp
SOURCES *= ChartGridCalcsTest.cpp

INCLUDEPATH += ../../Applications/Chart # allow us to include UUT directly
INCLUDEPATH += ../Common
INCLUDEPATH += ../..

LIBS *= -lChart -L../../Applications/Chart
LIBS *= -lSystemFunctions -L../../SystemFunctions
LIBS *= -lSettings -L../../Settings
LIBS *= -lFramework -L../../Framework
LIBS *= -lCommonPresentation -L../../CommonPresentation
```

ExampleTest.cpp

```
// Include the UUT at the very top of the file to test that it stands alone
// (i.e. all its own include dependencies are correct.
#include "ExampleUUT.h"

// Include the UUT *twice* to check its include guards.
#include "ExampleUUT.h"


#include <QtTest>
#include <QDebug>

/// Example test class
class ExampleTest : public QObject
{
    Q_OBJECT
public:
    //ExampleTest() {}

    ExampleUUT * uut;

private Q_SLOTS:

    /// Automatically run by QTest before the first test case
    ///
    /// You would not normally new() and delete() the UUT unless construction
    /// is complex. Normally the UUT can be created locally in the test method.
    /// Note that if you do create the UUT here, its state will persist from
    /// one test to the next which can be troublesome.
    void initTestCase()
    {
        qDebug() << "initTestCase";
        QObject* parent(nullptr);
        uut = new ExampleUUT(parent);
    }

    /// Automatically run by QTest after the last test case
    void cleanupTestCase()
    {
        qDebug() << "cleanupTestCase";
        delete uut;
    }

    /// Automatically run by QTest before each test case
    void init()
    {
        qDebug() << "init";
        // e.g. reset the state of the UUT you created in initTestCase()
    }

    /// Automatically run by QTest after each test case
    void cleanup()
    {
        qDebug() << "cleanup";
        const bool noChildObjects = uut->children().size() == 0;
        QVERIFY2(noChildObjects, "Test did not add any children to the UUT");
    }

    //---------------------------------------------------------------------------

    /// Test names should use lower case [1] with underscore separators, so the
    /// functions are easily read in the test output. Clauses within the test
    /// name can be separated by two underscores.
    /// [1] method names should appear unchanged.
    ///
    /// Test names should be descriptive and self documenting. Remember that
    /// when a test fails in the future, the first indication will be the test
    /// name and the QVERIFY/QCOMPARE failure message, and no more.
    /// The goal when naming a test is for this information, by itself, to fully
    /// describe the problem to someone familiar with the code. No further
    /// investigation should be necassary.
    /// Look at the output of badExample (below).
```

```
///
/// Name tests according to "given when then"
/// http://martinfowler.com/bliki/GivenWhenThen.html
/// Separate the three clauses with double underscores for clarity.
///
/// Alternatively, a more concise style can be adopted if any of the three
/// given/when/then clauses are not clear.
/// @see motionMode_defaults_to_active
///
/// The body of the test should be broken down into arrange, act, assert:
/// http://c2.com/cgi/wiki?ArrangeActAssert
/// These blocks do not need to be commented - the comments here are just
/// for illustration
///
/// Tests should be short and focus on a single logical concept.
/// Ideally this will lead to a single assertion. Often it is useful to
/// verify preconditions etc. to prove that the test action actually happens
/// rather than the test passing by chance. This may well lead to further
/// assertions in the test.
/// @see setCentre__no_presenter_set__ignores_the_value
///
/// Always make your test fail so you can look at the failure output.
/// Consider whether the output will be meaningful in the future.
/// Learn about the different QVERIFY and QCOMPARE macros, and the different
/// ways they report failure. Some simply repeat the assertion, but some
/// output the actual and expected values. This can be very useful when
/// testing maths and logic functions.
///
void for_integers__adding_one_and_one__yields_two()
{
    // arrange
    constexpr int one = 1;

    // act
    const int one_plus_one = one + one;

    // assert
    QVERIFY(one_plus_one == 2);
}

/// Poor test name and meaningless failure message from the QVERIFY.
void bad_example()
{
    int * x = 0;
    QVERIFY(x);
}

void example_using_pre_created_UUT_with_self_documenting_verify()
{
    // deliberately wrong to cause a fail to demonstrate the self documentation.
    const bool uutHasNoChildren = uut->children().size() != 0;
    QVERIFY(uutHasNoChildren);
}

void example_skipped_test()
{
    // loads of complicated test setup
    QSKIP("This keeps failing - needs investigation");
    QVERIFY2(1 == 2, "This is the failing test that gets skipped");
}

void example_using_scoping_to_control_UUT_lifetime()
{
    // create (and destroy) an instance
    {
        ExampleUUT uut;
        (void)uut;
    }

    QVERIFY2(1 + 1 == 2, "Destroying an ExampleUUT doesn't violate the laws of physics");
}
```

```
    // Constructor tests -------------------------------------------------------

    void new_instance__after_construction__has_size_zero()
    {
        // Making the UUT const makes the compiler test that the gettter is
        // const, i.e. if it is not const we get a compilation error.
        const ExampleUUT uut;

        const float size = uut.getSize();

        QCOMPARE(size, 0.0F);
    }

    // Alternative way of naming the same test
    void size_defaults_to_zero()
    {
        const ExampleUUT uut;
        QCOMPARE(uut.getSize(), 0.0F);
    }

    // Setter tests ------------------------------------------------------------

    void setSize_changes_size()
    {
        ExampleUUT uut;

        const float sizeBefore = uut.getSize();
        constexpr float newSize = 8.8;
        uut.setSize(newSize);
        const float sizeAfter = uut.getSize();

        QVERIFY(sizeBefore != sizeAfter);
        QCOMPARE(sizeAfter, newSize);
    }
    void setSize_ignores_values_greater_than_10()
    {
        ExampleUUT uut;

        const float sizeBefore = uut.getSize();
        constexpr float newSize = 10.1;
        uut.setSize(newSize);
        const float sizeAfter = uut.getSize();

        QVERIFY(sizeBefore == sizeAfter);
        QVERIFY(sizeAfter != newSize);
    }

    // init() tests ------------------------------------------------------------

    // Alternative, more verbose, test method name
    void after_calling_init__getSize_returns_pi()
    {
        init_sets_size_to_pi();
    }

    void init_sets_size_to_pi()
    {
        ExampleUUT uut;
        const float sizeBefore = uut.getSize();
        uut.init();
        const float sizeAfter = uut.getSize();
        QVERIFY(sizeBefore != sizeAfter);
        QCOMPARE(sizeAfter, 3.14151F); // example of fuzzy float compare
    }

    // doSomething() tests -----------------------------------------------------
    // These demonstrates the use of QSignalSpy

    void before_init__calling_doSomething__emits_zero()
    {
        ExampleUUT uut;
        QSignalSpy goodSignal(&uut, &ExampleUUT::GoodSignal);

        uut.doSomething();

        QList<QVariant> arguments = goodSignal.takeFirst();
        const int signalValue = arguments.at(0).toInt();
        QVERIFY(signalValue == 0);
    }

    void after_init__calling_doSomething__emits_one()
    {
        ExampleUUT uut;
        QSignalSpy goodSignal(&uut, &ExampleUUT::GoodSignal);

        uut.init();
        uut.doSomething();

        QList<QVariant> arguments = goodSignal.takeFirst();
        const int signalValue = arguments.at(0).toInt();
        QCOMPARE(signalValue, 1);
    }

    void doSomething_only_emits_GoodSignal()
    {
        ExampleUUT uut;
        QSignalSpy goodSignal(&uut, &ExampleUUT::GoodSignal);
        QSignalSpy badSignal(&uut, &ExampleUUT::BadSignal);

        uut.doSomething();

        QCOMPARE(goodSignal.count(), 1);
        // check the other didn't fire
        QCOMPARE(badSignal.count(), 0);
    }
};


// This allows our test class to be run simply by calling this as an external
// method. The test class need not be included by the caller.
static int runTests(int argc, char **argv)
{
    ExampleTest uut;
    return QTest::qExec(&uut, argc, argv);
}


#include "UnitTestMain.h"

// This static variable is a trick to cause AddTestFunction() to be called
// during startup. This causes all the tests get added to the vector in
// UnitTestsMain.cpp before main() runs.
static bool registered = RegisterTestFunction(runTests);

// This is needed because the test class doesn't have a header file.
#include "ExampleTest.moc"
```

Examples

ExampleTest.cpp is a fully documented example test. This explains the test framework, and good unit test practice.

For a practical example see the class ChartRangeIndicator. This behaviour could be inside ChartViewService, but by separating it out, it's easy to unit test without needing to mock dependencies like ChartMotionService, ChartCameraControl, Settings, etc.