Experience GoogleTest Framework for an ATLAS Package

P Sherwood

8 June 2016

Context

- 'New package' - TrigHLTJetHypo.Replaces many Hypo Algos from TrigJetHypo with a single hypo + helper classes written in plain C++.
- Simple helper classes are combined to perform complex algorithms.
- Helper classes are used in more than one hypo algorithm (code reuse).
- Having simple (non-framework) classes makes instantiation and this testing easier.
- Test package TrigHLTJetHypoUnitTests - code updates manifestly independent of TrigHLTJetHypo

Unit Tests

- Test the detailed behaviour of functions.
- Test: a short piece of code that runs the function under test, supplying it with the necessary arguments. Verifies expected behaviour via asserts.
- White box testing with GoogleTest and Google Moc
- $\approx$ 50 tests written for TrigHLTJetHypo
- Integrated into the cmake builds

Google Test

Takes care of the chore of looking after tests

- Runs tests independently - a test failure does no prevent later tests from running
- allows grouping of tests into suites - allows eg switching on and off groups of tests
- Collects and reports results
- Shared setup and teardown functions - efficiency gain

Documentation describes-

- many details on codeing strategies that facilitate testing
- how to setup and run the framework

Personal Impressions

Easy to use, quick to add tests, similar to other frameworks enciuntered (eg PyUnit)

Mock Objects

- When a function takes an instance as an argument pass it an alternative (mock) with the same interface.
- The mock object may be required to respond to method calls. This can be explicitly programmed to provide the response desired to test the functions behaviour.
- Mock objects can note which of its methods are called, and in which order.
- Polymorphism (eg interfaces or templates) is required - the functions need to be able to accept real and mock objects.

Black box testing?

- Black box testing - test only by varying inputs, and observing outputs
- Polymorphism not needed
- High code coverage is more difficult to achieve.

Google Mock

- provides mechanisms for writing mock object given an interface.
- wide range of monitoring functionality available
- responses of the mock object are programmable
- monitoring performed is programmable
- This style of testing without such tools is extremely tedious.
- C++ is very expressive, so tricky cases can arise.Plenty of help on the web.
- Works well with Google Test

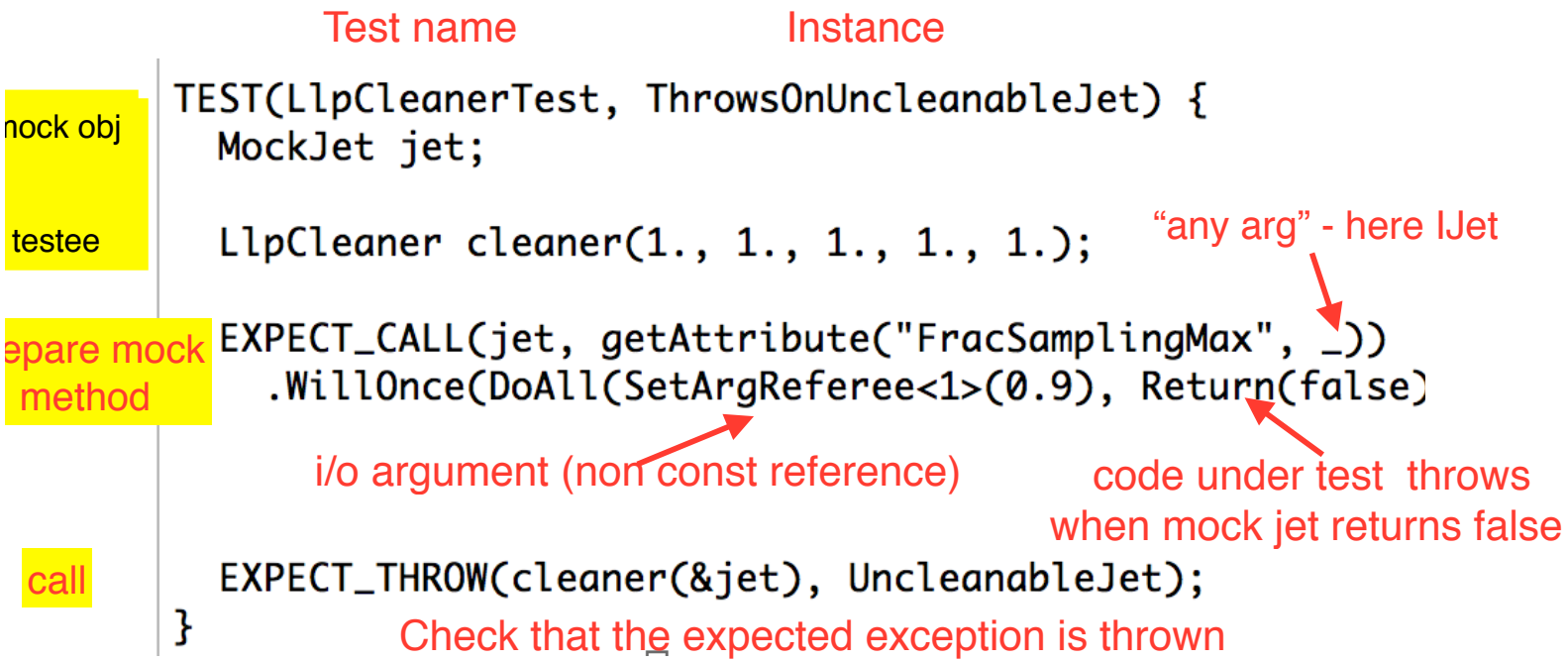
Code Example - a jet cleaner (really a jet rejector)

```
bool LlpCleaner::operator()(const pHypoJet& jet) const {
 /* make cuts on jet attributes to select jets from long-lived p
  float fsmJet;
  float neJet;
  if(jet -> getAttribute("FracSamplingMax", fsmJet)){
    if(jet -> getAttribute("NegativeE", neJet)){
         if(fsmJet > m_fSampMaxLlpThreshold &&
         std::abs(neJet) > m_negELlpThreshold){isClean = false;}
    } else {
      throw UncleanableJet("Cleaner Cannot retrieve NegativeE");
    }
  } else {
    throw UncleanableJet("Cleaner Cannot retrieve FracSamplingMa
  }
```

Example Test - one of many needed for full coverage

LlpCleaner

- LlpCleaner: one of a number of jet cleaner function objects.
- test single functionality: does the cleaner throw an exception if a an non-existent jet moment is requested?
- passes an instrumented mock jet via an interface



Screen shot - end of output

```
[       OK ] EtaEtConditionTest.artefacts (0 ms)
[ RUN      ] EtaEtConditionTest.accepts
[       OK ] EtaEtConditionTest.accepts (0 ms)
[ RUN      ] EtaEtConditionTest.belowEtaMinCut
[       OK ] EtaEtConditionTest.belowEtaMinCut (0 ms)
[ RUN      ] EtaEtConditionTest.aboveEtaMaxCut
[       OK ] EtaEtConditionTest.aboveEtaMaxCut (0 ms)
[ RUN      ] EtaEtConditionTest.belowEtCut
[       OK ] EtaEtConditionTest.belowEtCut (0 ms)
[----------] 5 tests from EtaEtConditionTest (0 ms total

[----------] 2 tests from TightCleanerTest
[ RUN      ] TightCleanerTest.SimpleThresholds
[       OK ] TightCleanerTest.SimpleThresholds (0 ms)
[ RUN      ] TightCleanerTest.ThrowsOnUncleanableJet
[       OK ] TightCleanerTest.ThrowsOnUncleanableJet (1
[----------] 2 tests from TightCleanerTest (1 ms total)

[----------] 1 test from FlowNetworkTest
[ RUN      ] FlowNetworkTest.SimpleTest
[       OK ] FlowNetworkTest.SimpleTest (0 ms)
[----------] 1 test from FlowNetworkTest (0 ms total)

[----------] 1 test from FlowEdgeTest
[ RUN      ] FlowEdgeTest.DefaultConstructor
[       OK ] FlowEdgeTest.DefaultConstructor (0 ms)
[----------] 1 test from FlowEdgeTest (0 ms total)

[----------] Global test environment tear-down
[==========] 49 tests from 11 test cases ran. (8 ms tota
[  PASSED  ] 49 tests.
```

Why write unit tests?

- they are very effective for identifying certain kinds of bugs
- the tests are close to the code, as opposed to integration tests
- my package: take milliseconds to run - instantaneous turn around
- allow responsible refactoring of code - integration tests are too coarse

Down side?

More code to write and maintain. The tools used help reduce this load.

Things that make function and class testing easier

(Very) short functions and testability

- Devising inputs to explore all code paths easier.
- Determining the correct output is easier.
- Code written as several short functions rather than a single long function is often more flexible - *ie* easier to modify in the future (overriding, reuse, template pattern....)
- BUT have more functions that interact.

The 7 ± 2 rule becomes natural, rather than only a Guru admonition.

Things that make unit testing more difficult

- many paths through the function
- use of globals
- object creation within a function:
  - Instantiation within a function body may lead to more code paths
  - Makes using mock objects more difficult.
- prefer: object creation in a separate function, pass in object
- prefer: passing in an object to be used as an attribute rather than instantiating in the constructor body.

Cmake integration

- Building the tests in Cmake has been added by A Krasznahorkay
- instructions for building with cmake can be found at https://twiki.cern.ch/twiki/bin/viewauth/AtlasComputing/CMakeTe
- very easy instructions to follow
- Look at $SVN-ROOT/Trigger/TriggerHypothesis/TrigHLTJetHypoUnitTests for the cmakelist.txt
- after running make, make test runs the tests - little output if all pass
- the command 'ctest –verbose' shows full output

Final Comments

- Unit tests provide fine scale testing.
- with very short functions, feels a bit like checking code with a debugger
- could be used to catch many bugs *much* faster than with integration tests
- maybe difficult to see bug effects in integration tests
- do not require the resources (machine and human) of integration tests
- does *not* replace integration tests!
- coverage measurements are needed to guide design of further tests
- the test package has integrated into the CMAKE builds - very convenient + automatable.