Table of Contents

Test Driven Development in LHCb

Test Driven Development (tdd) is a software development process that relies on the repetition of a very short development cycle, based around producing small and quick simple tests of problems and new features **before** fixing those problems. There is a nice wiki article on tdd here.

tdd can vastly improve the functionality and reduce the maintenance overhead of our software, provided that the shortest and simplest unit test is made to demonstrate the bug or missing feature, and provided it is added into our automated testing framework.

Table of Contents

The Metric of TDD

A good metric to tell how well we;re doing in terms of tdd in LHCb is to look at the ratio of lines of test code to lines of production code in an individual project. R = lines of test / lines of code

- R <<0.01 : virtually no tests, not good.
- R < 0.1 : A small amount of code is tested, not so good either.
- R ~ 1 : A good ratio for TDD, the target ratio, it takes approximately as many lines of test code to check the results as production code to make the fixes, not too much testing overhead, not too little testing.
- R > 1 : Too many tests or tests are too complicated.

Inside our software you can call `lb-project-testcoderatio` to measure this quantity for any LHCb project.

The existing testing framework

We use:

- **a custom extension of** QM-Test :
  - managed with **suites**. A suite (*.qms) can be an xml file with a list of tests in it (E.g. DaVinciTests), or a directory with a load of testfiles physically sitting below (E.g. lumialgs.qms). Tests can lead on from one another, and use custom validators in which we can run any arbitrary python code to check the results and decide on pass or fail.
  - Each suite contains *.qmt files, which are xml descriptions of how to run the test and how to validate the output.
- **A custom database of test files in** PRConfig :
  - PRConfig can hold options files for tests (PRConfig options), and also holds a database of different test files, TestFileDB (PRConfig TestFiles).
- **A central place to store input files for tests** :
  - Input files for tests (ROOT files, DSTs, Raw files, etc.) can/should be pinned permanently through Dirac, to the CERN-SWTEST storage element. This should mean they are pinned there **forever**, and are not linked to the life of any individual user account.
- **The nightly build system** :
  - Provided that your tests are in qmt form, they will be run every night in the LHCb nightlies, and so the release manager will know whenever a mistake is made.

Structure of the tests directory

Generally we follow the following structure:

- `/tests` top-level directory in the package, alongside src/python etc.
- `/tests/refs` directory in which we store any reference stdout or other files for comparison in the test results
- `/tests/options` any python files or steering options common to the tests. It's best to add them here to avoid having top-level-options and avoid messing up tests. If you don't want the firles to appear here, tehn they should be in PRConfig.
- `/tests/qmtest` this is the directory in which the tests will mostly physically run, so all relative paths should be relative to this location.
- `/tests/qmtest/<packagename>.qms` the automatic test suite run in the nightly builds
- `/tests/qmtest/<compatibility>.qms` the automatic test suite run in the nightly build slot called **lhcb-compatibility**

Large files (>10MB) used for testing should **not** appear in the package themselves, because it's not very efficient for SVN, instead they should appear in the standard testfiles location, CERN-SWTEST, and be in the TestFileDB in PRConfig.

Examples of existing test suites

Take a look at the directory structure and requirements files of these packages for examples:

- **A simple test suite with simple options:** IndependenceTests HltBase
- **Testing simple python modules:** XMLSummaryBase, RecSys
- **Multiple consecutive unit tests in a suite:** lumialgs.qms, DAQSys, raweventcompat
- **More complicated test suite with multiple inter-dependencies**: DaVinciTests.fsrs moore.physics.tck
- **A test suite with custom validators in the qmt file**: XMLSummaryKernel
- **A complicated test suite with custom validators stored elsewhere**: moore.physics.deferral

Video example, adding a new test, running a test

Best to copy a previous test suite, or get your release manager to create the intial structure for the qmtests for you. Adding a new test requires placing a new qmt file with some options into a test suite. See the video for an example, or read up more on GaudiTestingInfrastructure .

Adding a new input file for testing

Initially your test file will probably be a local file, or a file on castor given to you by somebody else. As soon as the test is showing what you want to show, though, the test file should be copied to the test location CERN-SWTEST and migrated into the TestFileDB framework.

What sort of tests should I add?

Many different things are testable, but each individual test should ideally only test one thing at a time. This might be **can I instantiate my class** or **does my specific branch appear in this ntuple**. Testing too many things at once is a typical bad habit, of course the applications themselves need to get as much coverage as possible, but when looking at one bug or feature, try and implement tests which are only sensitive to what **you** yourself have done.

How long can my tests take?

The ideal unit test finishes in a few seconds. The ideal functionality test finishes in under 5 minutes. The ideal integration or application test finishes in under an hour. Tests taking more than 5 minutes should really only

be run in the Performance and Regression weekly testing framework which is currently in development, but until that is provided, then anything under an hour is OK. Remember that the shorter your test is, the quicker your development cycle will be.

When should I add tests?

In tdd you add tests **before** you solve a problem. This may be problematic in the case you haven't obtained yet the minimum requirements for reproducing the problem, but as soon as you have a reproducible problem, then a test is required. You should also add tests for each major new feature, especially if it involves any repackaging of existing components.

Where should I add the tests?

Add a test **as close as possible** to the package which causes the bug. This may force you to write simpler test cases. If it is a problem in the integration between two otherwise independent packages, you will probably need to add a test at the application level.

Can I run the test myself with plain gaudirun?

Yes, provided you have called SetupProject correctly, `gaudirun.py somefile.qmt` is now supported.

Common Mistakes:

- **Testing too much:** Often tests are overly sensitive to small changes in other miscellaneous code which was not related to this particular bug or feature. This can very much over-complicate your test validation procedure. Better to reduce what's running in your test as much as possible, such that it is just the algorithm you actually need.
- **Testing too late (time):** Often tests are constructed a long time after a bug is noticed or to test a feature which was been added a long time ago. This makes it very difficult to tell if there are problems with the bug or feature in question, or some other ancillary software which is unrelated and now works slightly differently. Testing should ideally start **before** development, and a new qmtest should itself validate that the development which was made is included correctly before the release..
- **Testing too late (packaging):** Often testing is done in a much much higher level package for low-level problems. For example testing an individual decoder inside Moore, when the decoders live in the LHCb project LHCb->Lbcom->Rec->Hlt->Moore. So, if a problem is spotted later it requires patching at a much lower level, and so is a much larger release overhead. Place tests as close to the code as possible for a faster develpment cycle.
- **Forgetting custom validators:** custom validators are very powerful. Any generic python can be added into the qmtest validation step to check output files of any type. Writing a custom validator can avoid sensitivity to other random features of the data and can be specific to checking the problem you are actually dealing with. LHCb already has a large suite of custom validators you can make use of, such as the countErrorLines validator.
- **Adding any upper case letters:** If a test contains an upper case letter in the name of the suite or the name of the qmt file, it will not be run.
- **Not declaring prerequisites:** Often several tests are interlinked since they test gradually more features of the software. If the first test fails because of some new underlying bug or feature, it is often very much not necessary to continue with the remaining tests. All that does is slow the testing process down. Declaring the correct dependencies can save time in development when a new problem surfaces.
- **Ignoring test results or deleting failing tests:** a lot of the time old tests are simply removed, which is poor practise. If they were testing for a specific bug or feature, then just removing the test is dangerous. Similarly editing the validator such that it just passes, or updating the reference files

without checking what has really changed is dangerous and bad practise, to be avoided.

- **Deleting or moving input files**: a lot of problems ensue when you accidentally move or delete an input file for a test. That's why we have the CERN-SWTEST storage element.
- **Forgetting to make declarations in the requirements file:** Packages have run-time dependencies which are often necessary to run the tests you want. Forget about them and the test will fail.
- **Too much copy-paste:** too much copying from one test to another most often ends up with overly complicated tests which are not sensitive to the problem they are trying to detect, but instead overly sensitive to small changes in underlying code. Well-designed tests which are more specific to the task will get around this problem.

This topic: LHCb > TestDrivenDevelopment
Topic revision: r3 - 2014-01-20 - RobLambert