Software Unit Test Policy

Introduction

Unit testing validates the implementation of all objects from the lowest level defined in the detailed design (classes and functions) up to and including the lowest level in the architectural design (equivalent to the WBS components). The next layer of testing, known as integration testing, tests the interfaces between the architectural design objects (i.e. WBS components). This Policy only addresses Unit Testing. Refer to Introduction into testing or why testing is worth the effort (John R. Phillips, 2006) for a short, but good discussion on Unit Testing and its transition into Integration Testing.

Types of Unit Tests

Unit tests should be developed from the detailed design of the baseline, i.e. from either the structure diagrams or the class/function definitions. The type of tests performed during unit testing include:

White-box Tests
These tests are designed by examining the internal logic of each module and defining the input data sets that force the execution of different paths through the logic. Each input data set is a test case.

Black-box Tests
These tests are designed by examining the specification of each module and defining input data sets that will result in different behavior (e.g. outputs). Black-box tests should be designed to exercise the software for its whole range of inputs. Each input data set is a test case.

Performance Tests
If the detailed design placed resource constraints on the performance of a module, compliance with these constraints should be tested. Each input data set is a test case.

The collection of a module's white-box, black-box, and performance tests is known as a Unit Test suite. A rough measure of a Unit Test suite's quality is the percentage of the module's code which is exercised when the test suite is executed.

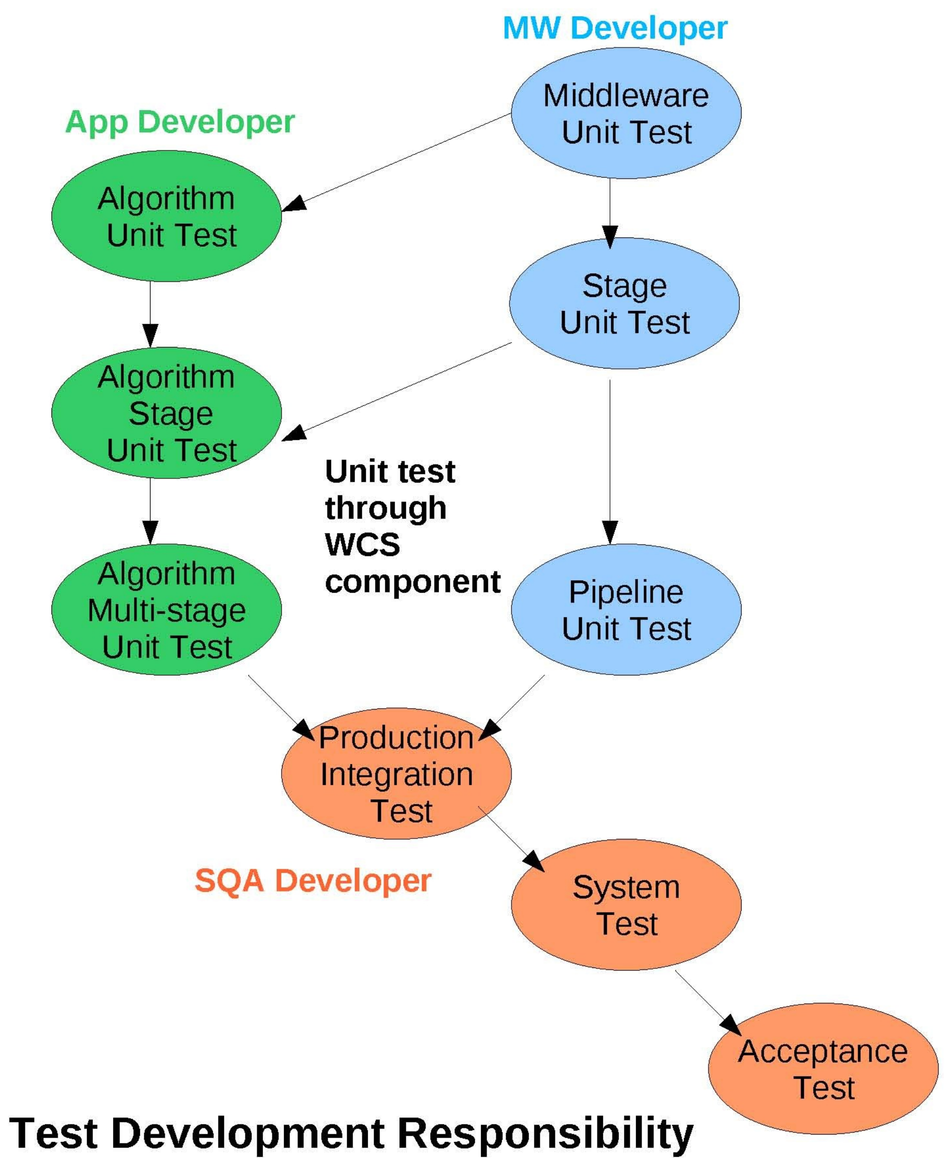
Responsibility for Implementation

Due to the nature of white box testing, it is best if the original developer of an object creates the test suite validating that object. During later object modification, the current developer should update the test suite to validate the changed internals.

Important

LSST DM developers are responsible for creating test suites to unit test all objects they implement. Additionally, they are responsible for updating an object's test suite after modifying the object's source code.

The division of responsibility between DM developer groups is highlighted in the diagram below. In essence, the Applications group, which is responsible for all science algorithm development, also develops the unit testers for: algorithm components, a stage wrapped algorithm, and the sequences of stage wrapped algorithms comprising a simple-stage-tester algorithm pipeline (i.e. without parallel processing job management). The Middleware group, so named because it is responsible for all low level framework software, develops the unit testers for those framework modules. Finally, the SQA team is responsible for the higher level testers for integration, system performance, and acceptance testing.


MW Developer
Middleware Unit Test
App Developer
Algorithm Unit Test
Stage Unit Test
Algorithm Stage Unit Test
Unit test through WCS component
Algorithm Multi-stage Unit Test
Pipeline Unit Test
Production Integration Test
SQA Developer
System Test
Acceptance Test
Test Development Responsibility

Testing Frameworks

Test suite execution should be managed by a testing framework, also known as a test harness, which monitors the execution status of individual test cases.

C++: boost.test
LSST DM developers should use the single-header variant of the Boost Unit Test Framework. When unit testing C++ private functions using the Boost util test macros, refer to the standard methods described in private function testing.

Python: unittest
LSST DM developers should use Python's unittest framework.

`lsst.utils.tests` provides several utilities for writing Python tests that developers should make use of. In particular, `lsst.utils.tests.MemoryTestCase` is used to detect memory leaks in C++ objects. `MemoryTestCase` should be used in all tests, even if C++ code is not explicitly referenced.

This example shows the basic structure of an LSST Python unit test module, including `lsst.utils.tests.MemoryTestCase`:

```
import unittest

import lsst.utils.tests as utilsTests


class DemoTestCase(utilsTests.TestCase):
    """Demo test case."""

    def testDemo(self):
        assert True


def suite():
    """Returns a suite containing all the test cases in this module."""
    utilsTests.init()

    suites = []
    # Test suites for this module here
    suites += unittest.makeSuite(DemoTestCase)
    # MemoryTestCase to find C++ memory leaks
    suites += unittest.makeSuite(utilsTests.MemoryTestCase)
    return unittest.TestSuite(suites)


def run(exit=False):
    """Run the tests"""
    utilsTests.run(suite(), exit)


if __name__ == "__main__":
    run(True)
```

Note that `MemoryTestCase` must always be the final test suite.

Unit Testing Composite Objects

Data Management uses a bottom-up testing method where validated objects are tested with, then added to, a validated baseline. That baseline, in turn, is used as the new validated baseline for further iterative testing. When developing test suites for composite objects, the developer should first ensure that adequate test suites exist for the base objects.

Automated Nightly and On-Demand Testing

Jenkins is a system which automates the compile/load/test cycle required to validate code changes. In particular, Jenkins automatically performs unit builds and unit tests expedites the module's repair and, hopefully, limits the time other developers are impacted by the failure. For details, refer to the workflow documentation on Testing with Jenkins.

Verifying Test Quality

Since Unit Tests are used to validate the implementation of detailed design objects through comprehensive testing, it's important to measure the thoroughness of the test suite. Coverage analysis does this by executing an instrumented code which records the complete execution path through the code and then calculating metrics indicative of the coverage achieved during execution.

See Coverage Analysis for more information.