CODIGN
SOFTWARE, LLC

Unit Testing Recommended Best Practices

This cheat sheet provides a set of Codign recommended best practices when creating unit tests. All the best practices defined are used in Codign's software product, CoView.

Definition of Unit Testing

Unit testing is the execution and validation of a block of code in isolation that a developer created. Because an object's behavior is determined by its method logic, Codign Software identifies a unit as a method within a class.

Write tests for methods that have the fewest dependencies first, and work your way up. If you start by testing a high-level method, your test may fail because a subordinate method may return an incorrect value to the method under test. This increases the time spent finding the source of the problem, and you will still have to test the subordinate method anyway to be sure it does not contain other bugs.

Tests should be logically simple as possible, preferably with no decisions at all. Every decision added to a test method increases the number of possible ways that a test method can be executed. Testing is meant to be a controlled environment, the only thing you want to change is the input to the method under test, not the test method itself.

Where possible, use constant expected values in your assertions instead of computed values. Consider these two assertions:

```
returnVal = methodUnderTest(input);
assertEquals(returnVal, computeExpectedReturnValue(input));
assertEquals(returnVal, 12);
```

In order for computeExpectedReturnValue() to return the proper result it probably has to implement the same logic as the method under test, which is what you are trying to test for in the first place! Further, suppose you fixed a defect in the method under test. Now you have to change computeExpectedReturnValue() in the same manner. The second assertion is easier to understand and maintain.

Each unit test should be independent of all other tests. A unit test should execute one specific behavior for a single method. Validating behavior of multiple methods is problematic as such coupling can increase refactoring time and effort. Consider the following example:

```
void testAdd(){
    int return1 = myClass.add(1,2);
    int return2 = myclass.add(-1,-2);
    assertTrue (return1 - return2 == 0);
}
```

If the assertions fails in this case, it will be difficult to determine which invocation of add() caused the problem.

Each unit test should be clearly named and documented. The name of the test method should clearly indicate which method is being tested because improperly named tests increase maintenance and refactoring efforts. Comments should also be used to describe the test or any special conditions.

All methods, regardless of visibility, should have appropriate unit tests. Public, private and protected methods that contain data and decisions should be unit tested in isolation.

All unit tests should use appropriate coverage techniques to measure their effectiveness. Isolating coverage for each unit test independently of all other unit test is the only way to verify actual results. It is not appropriate to look at coverage numbers by themselves. Think of the situation where a unit test for Method A may increase the coverage for Method B as well. Even if Method B has coverage, unless the specific behavior was verified via an assertion, it should not be considered unit tested.

Strive for one assertion per test case. Multiple assertions in one test case can cause performance issues and bottlenecks. For example, if a test case with five assertions fails on the first assertion, then the remaining four are not executed. Only when the first assertion is corrected will the other assertions execute. If assertion two fails, then that also must be corrected before the remaining three assertions are verified.

One major benefit of unit testing is to find the root cause of defects. Unit tests should be designed with this in mind, thus it is more economical to have one assertion per test case. Although the unit test file may be large, it will be easier to find and fix defects.

Create unit tests that target exceptions. Exceptions are thrown when a method finds itself in a state it cannot handle. These cases should be tested just like a method's normal state of operations.

If a method is declared to throw one or more exceptions, then unit tests must be create that simulate the circumstances under which those exceptions could be thrown. The unit tests should assert that the exceptions are thrown as expected.

516 N. Charles St. Ste 405
Baltimore, MD 21201

p (410) 244-8660
f (410) 244-8661

www.codign.com