Unit Testing Guidelines

Ina Schieferdecker, Edzard Höfig

August 24, 2012

This document describes unit testing guidelines which have been adapted from the web page at `http://geosoft.no/development/unittesting.html`.

1 Keep testing at unit level

Unit testing is about testing classes. There should be one test class per ordinary class and the class behaviour should be tested in isolation. Avoid the temptation to test an entire work-flow using a unit testing framework, as such tests are slow and hard to maintain. Work-flow testing may have its place, but it is not unit testing and it must be set up and executed independently.

2 Name tests properly

Make sure each test method tests one distinct feature of the class being tested and name the test methods accordingly. The typical naming convention is `testWHAT` such as `testSaveAs()`, `testAddListener()`, `testDeleteProperty()` etc.

3 Test the trivial cases, too

It is sometimes recommended that all non-trivial cases should be tested and that trivial methods like simple setters and getters can be omitted. However, there are several reasons why trivial cases should be tested too:

- Trivial is hard to define. It may mean different things to different people.
- From a black-box perspective there is no way to know which part of the code is trivial.
- The trivial cases can contain errors too, often as a result of copy-paste operations.

The recommendation is therefore to test everything. The trivial cases are simple to test after all.

4 Focus on execution coverage first

Differentiate between execution coverage and actual test coverage. The initial goal of a test should be to ensure high execution coverage. This will ensure that the code is actually executed on some input parameters. When this is in place, the test coverage should be improved. Note that actual test coverage cannot be easily measured (and is always close to 0% anyway). Consider the following public method:

```
void setLength(double length);
```

By calling `setLength(1.0)` you might get 100% execution coverage. To achieve 100% actual test coverage the method must be called for every possible double value and correct behaviour must be verified for all of them. Surly an impossible task.

5 Cover boundary cases

Make sure the parameter boundary cases are covered. For numbers, test negatives, 0, positive, smallest, largest, NaN, infinity, etc. For strings test empty string, single character string, non-ASCII string, multi-MB strings etc. For collections test empty, one, first, last, etc. For dates, test January 1, February 29, December 31 etc. The class being tested will suggest the boundary cases in each specific case. The point is to make sure as many as possible of these are tested properly as these cases are the prime candidates for errors.

6 Provide a random generator

When the boundary cases are covered, a simple way to improve test coverage further is to generate random parameters so that the tests can be executed with different input every time.

To achieve this, provide a simple utility class that generates random values of the base types like doubles, integers, strings, dates etc. The generator should produce values from the entire domain of each type. If the tests are fast, consider running them inside loops to cover as many possible input combinations as possible. The following example verifies that converting twice between little endian and big endian representations gives back the original value. As the test is fast, it is executed on one million different values each time.

```
void testByteSwapper() {
  for (int i = 0; i < 1000000; ++i) {
      double v0 = Random.getDouble();
      double v1 = ByteSwapper.swap(v0);
      double v2 = ByteSwapper.swap(v1);
      assertEquals(v0, v2);
    }
}
```

7 Test each feature once

When being in testing mode it is sometimes tempting to assert on *everything* in every test. This should be avoided as it makes maintenance harder. Test exactly the feature indicated by the name of the test method. As for ordinary code, it is a goal to keep the amount of test code as low as possible.

8 Use explicit asserts

Always prefer `assertEquals(a, b)` to `assertTrue(a == b)` (and likewise) as the former will give more useful information of what exactly is wrong if the test fails. This is in particular important in combination with random value parameters as described above when the input values are not known in advance.

9 Provide negative tests

Negative tests intentionally misuse the code and verify robustness and appropriate error handling. Consider this method that throws an exception if called with a negative parameter:

```
void setLength(double length) throws IllegalArgumentException;
```

Testing correct behaviour for this particular case can be done by:

```
try {
  setLength(−1.0);
  fail(); // If we get here, something went wrong
} catch (IllegalArgumentException exception) {
  // If we get here, all is fine
}
```

10 Prepare test code for failures

Consider the simple example:

```
Handle handle = manager.getHandle();
assertNotNull(handle);
String handleName = handle.getName();
assertEquals(handleName, "handle−01" );
```

If the first assertion is false, the code crashes in the subsequent statement and none of the remaining tests will be executed. Always prepare for test failure so that the failure of a single test doesn't bring down the entire test suite execution.