Unit Testing Guidelines

These are unit testing guidelines created and maintained by GeoSoft.

The guidelines are applied to all GeoSoft software projects.

1. Keep unit tests small and fast

Ideally the entire test suite should be executed before every code check in. Keeping the tests fast reduce the development turnaround time.

2. Unit tests should be fully automated and non-interactive

The test suite is normally executed on a regular basis and must be fully automated to be useful. If the results require manual inspection the tests are not proper unit tests.

3. Make unit tests simple to run

Configure the development environment so that single tests and test suites can be run by a single command or a one button click.

4. Measure the tests

Apply coverage analysis to the test runs so that it is possible to read the exact execution coverage and investigate which parts of the code is executed and not.

5. Fix failing tests immediately

Each developer should be responsible for making sure a new test runs successfully upon check in, and that all existing tests runs successfully upon code check in.

If a test fails as part of a regular test execution the entire team should drop what they are currently doing and make sure the problem gets fixed.

6. 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.

7. Start off simple

One simple test is infinitely better than no tests at all. A simple test class will establish the target class test framework, it will verify the presence and correctness of both the build environment, the unit testing environment, the execution environment and the coverage analysis tool, and it will prove that the target class is part of the assembly and that it can be accessed.

The *Hello, world!* of unit tests goes like this:

```
void testDefaultConstruction()
{
  Foo foo = new Foo();
  assertNotNull(foo);
}
```

8. Keep tests independent

To ensure testing robustness and simplify maintenance, tests should never rely on other tests nor should they depend on the ordering in which tests are executed.

9. Keep tests close to the class being tested

If the class to test is `Foo` the test class should be called `FooTest` (*not* `TestFoo`) and kept in the same package (directory) as `Foo`. Keeping test classes in separate directory trees makes them harder to access and maintain.

Make sure the build environment is configured so that the test classes doesn't make its way into production libraries or executables.

10. Name tests properly

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

11. Test public API

Unit testing can be defined as *testing classes through their public API*. Some testing tools makes it possible to test private content of a class, but this should be avoided as it makes the test more verbose and much harder to maintain. If there is private content that seems to need explicit testing, consider refactoring it into public methods in utility classes instead. But do this to improve the general design, not to aid testing.

12. Think black-box

Act as a 3rd party class consumer, and test if the class fulfills its requirements. And try to tear it apart.

13. Think white-box

After all, the test programmer also wrote the class being tested, and extra effort should be put into testing the most complex logic.

14. 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:

```
private double weight_;
private double x_, y_;

public void setWeight(int weight)
{
  weight = weight_;  // error
}

public double getX()
{
  return x_;
}

public double getY()
{
  return x_;  // error
}
```

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

15. Focus on execution coverage first

Distinguish 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 acheive 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.

16. 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.

17. 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);
  }
}
```

18. 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.

19. 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.

20. 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 behavior 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
}
```

21. Design code with testing in mind

Writing and maintaining unit tests are costly, and minimizing public API and reducing cyclomatic complexity in the code are ways to reduce this cost and make high-coverage test code faster to write and easier to maintain.

Some suggestions:

- Make class members immutable by establishing state at construction time. This reduce the need of setter methods.

- Restrict the use of excessive inheritance and virtual public methods.
- Reduce the public API by utilizing friend classes (C++), internal scope (C#) and package scope (Java).
- Avoid unnecessary branching.
- Keep as little code as possible inside branches.
- Make heavy use of exceptions and assertions to validate arguments in public and private API's respectively.
- Restrict the use of convenience methods. From a black-box perspective every method must be tested equally well. Consider the following trivial example:

```
public void scale(double x0, double y0, double scaleFactor)
{
  // scaling logic
}

public void scale(double x0, double y0)
{
  scale(x0, y0, 1.0);
}
```

  Leaving out the latter simplifies testing on the expense of slightly extra workload for the client code.

22. Don't connect to predefined external resources

Unit tests should be written without explicit knowledge of the environment context in which they are executed so that they can be run anywhere at anytime. In order to provide required resources for a test these resources should instead be made available by the test itself.

Consider for instance a class for parsing files of a certain type. Instead of picking a sample file from a predefined location, put the file content inside the test, write it to a temporary file in the test setup process and delete the file when the test is done.

23. Know the cost of testing

Not writing unit tests is costly, but writing unit tests is costly too. There is a trade-off between the two, and in terms of execution coverage the typical industry standard is at about 80%.

The typical areas where it is hard to get full execution coverage is on error and exception handling dealing with external resources. Simulating a database breakdown in the middle of a transaction is quite possible, but might prove too costly compared to extensive code reviews which is the alternative approach.

24. Prioritize testing

Unit testing is a typical bottom-up process, and if there is not enough resources to test all parts of a system priority should be put on the lower levels first.

25. 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. In general rewrite as follows:

```
Handle handle = manager.getHandle();
assertNotNull(handle);
if (handle == null) return;

String handleName = handle.getName();
assertEquals(handleName, "handle-01");
```

26. Write tests to reproduce bugs

When a bug is reported, write a test to reproduce the bug (i.e. a failing test) and use this test as a success criteria when fixing the code.

27. Keep it simple

Unit tests must be simple in order to be effective, they should not contain comprehensive complexity on their own. A sure smell is if the unit test is duplicating some of the logic in the code being tested, or if it otherwise seems that the test code *itself* needs unit testing.

28. Know the limitations

Unit tests can never prove the correctness of code!!

A failing test may indicate that the code contains errors, but a succeeding test doesn't prove anything at all.

The most useful appliance of unit tests are verification and documentation of requirements at a low level, and *regression testing*: verifying that code invariants remains stable during code evolution and refactoring.

Consequently unit tests can never replace a proper up-front design and a sound development process. Unit tests should be used as a valuable supplement to the established development methodologies.

And perhaps most important: The use of unit tests forces the developers to think through their designs which in general improve code quality and API's.