Testing Philosophy

10 May 2007

Table of Contents

1. Introduction

The purpose of this document is to put forth a set of guidelines for good testing. We will cover some of the basic underlying principles of testing followed by good practices and common misconceptions.

2. Guidelines For Writing Good Tests

In this section we discuss the principles and end goals for writing tests. We will later take these principles and evaluate different approaches to writing tests.

- Write tests to enable refactoring of the production code. The value proposition here is that a well designed test suite will allow us to make changes to our production code (*without changing the test code*) and have good confidence that the new production code does not break existing functionality. The important characteristic here is the emphasized phrase in the last sentence. If a change to production code requires a change to a test, either the test wasn't as good as it could be or there was a real change to the functionality. In either case, we have no reason to believe the new code does not break existing functionality.

- Write tests to increase the maintainability of the production code. This is subtly different than the first point. The main thrust here is maintainability. It is not (even theoretically) possible to write tests for all scenarios. For example, when a bug is found in production code, it should be reproduced with a test and then fixed. Such a test increases maintainability by ensuring that future changes will not cause the same problem.

- Do not write tests for trivial code. Do not test the compiler. The justification here is simple: if we can't trust the compiler, we can't trust the test code, so there is no point in writing any tests at all. From another perspective, the reason we write tests is because we're not certain our production code is correct. If we write a piece of test code, how can we be certain that the test is correct? Clearly, there is no magic pixie dust we can sprinkle on the test code to somehow make it correct, when we can't do the same with the production code. If we can't be certain about the correctness of our test code, then we should write tests for our tests. The cycle is infinite, unless we break it somehow. The only reasonable way of breaking it is by making the statement: code that is correct by *inspection* does not need any tests. Having come to this conclusion, we must apply it uniformly to all our code. A typical example of a trivial piece of code would be a getter method.

- Use coverage as a guide to write more or better tests. Coverage won't tell us if our tests are good tests, but it will tell us what code is not all executed. That code can be a target for writing a test.

- Do not re-implement production code in the tests. A test that repeats the production code (typically in an assert) is not testing anything at all (other than the compiler). Trivial example of bad piece of test code:

`assertEquals(2+3, sum(2,3)`. The correct way to write that would be: `assertEquals(5, sum(2,3))`. The worst incarnations of this are tests that build entire (parallel) code to the production code just to test the production code. Having two implementations of the same concepts is even worse for maintenance. Steven Wright, a comedian, expressed this idea thus: “For my birthday I got a humidifier and a de-humidifier... I put them in the same room and let them fight it out.”

- Every line of code, including test code, costs time, effort and money. Two test suites that cover the same code can be evaluated based on this: all other thing being equal, the suite that is smaller is the better suite.

3. Common Misconceptions About Tests

There are two categories of misconceptions about tests at the two extremes. Some people don't like tests at all and don't understand their utility. We will not discuss those here; our assumptions is that tests are good. The other extreme is giving too much emphasis and importance to tests. It is unfortunately all too easy to try to make tests into something they are not. In this section, we'll explore a few of the common misconceptions about tests in the latter category.

- Coverage is a direct measure of how good the tests (and therefore the code) are. There is no evidence for such a claim. In fact, there is evidence that this claim is not only wrong but it can be misleading. It is misleading because coverage fails to show two important causes of failure: missing code and varied input conditions. If a bit of code is missing, for example, an if statement, there is no way for coverage to highlight that. The code could be a 100% covered and totally buggy. Alternatively, the code could be tested with one set of inputs (and therefore have high coverage), but fail badly for another set of inputs which are not tested. The correlation between coverage and quality, at best, should be taken very lightly. The only guideline we can give here is based on experience with a large project (Reladomo) that has had coverage reports for several years: between 0 and 40%, the there is significant correlation between coverage and quality. From 40% to 60% the correlation drops significantly. Above 60%, there is practically no correlation between coverage and quality. Put differently, the bugs that get fixed in a code base with more than 60% coverage tend to be mostly of the variety that coverage doesn't catch. Unit tests added for these bugs does not change the coverage percentage.

- A piece of code with no unit tests is not a high quality piece of code. We can easily disprove this with an example: the Linux Kernel. The Linux Kernel is possibly the most mission critical and high quality piece of code on the planet for its size (over 4.2 million lines of code). Every bit that gets stored on a disk, goes onto the network or is put into memory goes through the kernel. It has no unit tests. That doesn't mean it's not tested, but there really are no unit tests. Developers don't write tests and therefore don't run them before commit or release. And still, the Linux Kernel is the highest quality mission critical piece of code on the planet. In short, unit tests are not the only way to achieve quality. They are not even a necessary component of achieving quality. (This is not a claim that unit tests are bad, just that they can made into more than they really are from a myopic point of view).

- A test is proof that the code is correct. This is wrong on many levels. It is wrong on a trivial level: no test suite can test all scenarios. It is wrong on a conceptual level: a proof requires a lot more than a test suite. Here is a 58 page proof of the RSA-PSS algorithm: Formal Proof for the Correctness of RSA-PSS [http://eprint.iacr.org/2006/011.pdf]. The RSA-PSS algorithm can be implemented in about 60 lines of code. Put differently, unit tests can only show the presence of errors. They cannot prove the absence of errors.

To summarize: tests are good, but we should not get caught in the hype and make them into something they are not.

4. What's a Unit?

It can be difficult to decide what exactly a test case should look like in terms of its interaction with the production code. We will show in this section that a unit should loosely be defined as the public contract of the code under test. This is in contrast to narrow definitions of a unit (e.g. a unit is a method).

At a high level we can contrast two different approaches: testing the public interface of a the code versus testing every method. We will go into a specific example later in this section.

Testing at the method level has these characteristics:

- Pro: These tests acts as a development aid. While there is certainly nothing wrong with using a variety of tools to aid in the development effort, it's not clear these types of tests should become part of the system. We use many tools in our development (ide, debugger, javadoc, tutorials, conversations, etc). We don't check in those development aids, why should we check in tests-as-aids?
- Con: these tests tend to test the specific implementation. This makes the code more brittle and makes the tests less useful because these tests do not enable refactoring.
- Con:they often add contracts to the object that has nothing to do with the object's responsibility. These extra contracts are then used throughout the code (after all, there is a test case for it, it must behave this way).

Testing the public contract has these characteristics:

- Pro: is by nature at the interface level and much harder to test the specific implementation. These tests tend to be much more resilient to change and can enable refactoring.
- Pro: since there is no emphasis on per-method tests (just the public contract), these tests tend to be fewer and cover more code. This is in line with our "shorter test is better" principle.
- Con: their utility as a development time aid is not as much a method level test.

As an illustration of the two methodologies, we will implement a simple class (a stack) in both ways and then compare the results. Developer A is given the task to write a stack class. He's been told to write tests for every method in isolation. There will be no private methods in his class; he's been instructed to mark any would-be private method as protected in order to facilitate testing. There is no need to write tests for getter/setter methods. He writes the following code:

```
package test;

public class StackA
{
    private Node head;

    public void push(Object o)
    {
        Node n = createNode(o, head);
        head = n;
    }

    protected Node getHead()
    {
        return head;
```

```
    }

    protected void setHead(Node head)
    {
        this.head = head;
    }

    public Object pop()
    {
        if (head == null) return null;
        Node n = head;
        head = n.getNext();
        return n.getObj();
    }

    protected Node createNode(Object o, Node next)
    {
        return new Node(o, next);
    }

    protected static class Node
    {
        private Object obj;
        private Node next;

        public Node(Object obj, Node next)
        {
            this.obj = obj;
            this.next = next;
        }

        public Object getObj()
        {
            return obj;
        }

        public Node getNext()
        {
            return next;
        }
    }
}
```

His code is accompanied by the following test class:

```
package test;

import junit.framework.TestCase;


public class TestStackA extends TestCase
{
    public void testCreateNode()
```

```
    {
        StackA s = new StackA();
        Integer o = new Integer(10);
        StackA.Node n = s.createNode(o, null);
        assertSame(o, n.getObj());
        assertNull(n.getNext());

        Integer o2 = new Integer(2);
        StackA.Node secondNode = s.createNode(o, n);
        assertSame(o2, secondNode.getObj());
        assertSame(n, secondNode.getNext());
    }

    public void testPush()
    {
        StackA s = new StackA();
        Integer o = new Integer(10);
        s.push(o);
        assertSame(s.getHead().getObj(), o);
        assertNull(s.getHead());
    }

    public void testPop()
    {
        StackA s = new StackA();
        assertNull(s.pop());

        Integer o = new Integer(10);
        StackA.Node n = new StackA.Node(o, null);
        s.setHead(n);

        assertSame(o, s.pop());
    }

}
```

Developer B is given the same task, but he's instructed to only test the public contract of the object. He's told not to change the object's encapsulation. He writes the following class:

```
package test;

public class StackB
{
    private Node head;

    public void push(Object o)
    {
        Node n = createNode(o, head);
        head = n;
    }

    public Object pop()
    {
```

```
        if (head == null) return null;
        Node n = head;
        head = n.getNext();
        return n.getObj();
    }

    private Node createNode(Object o, Node next)
    {
        return new Node(o, next);
    }

    private static class Node
    {
        private Object obj;
        private Node next;

        public Node(Object obj, Node next)
        {
            this.obj = obj;
            this.next = next;
        }

        public Object getObj()
        {
            return obj;
        }

        public Node getNext()
        {
            return next;
        }
    }
}
```

His code is accompanied by the following test class:

```
package test;

import junit.framework.TestCase;

public class TestStackB extends TestCase
{
    public void testOnePushPop()
    {
        StackB s = new StackB();
        assertNull(s.pop());
        Integer o = new Integer(10);
        s.push(o);
        assertSame(o, s.pop());
        assertNull(s.pop());
    }

    public void testTwoPushPop()
```

```
    {
        StackB s = new StackB();
        Integer o = new Integer(10);
        s.push(o);

        Integer o2 = new Integer(2);
        s.push(o2);
        assertSame(o2, s.pop());
        assertSame(o, s.pop());
        assertNull(s.pop());
    }
}
```

There are simple differences between the two pieces of code. Overall, Developer B wrote less code. He has not exposed the internal implementation of his class and his tests are just about the public contract of the object. Developer A had to add extra methods to his class to facilitate his testing (get/setHead) and mark several things as protected. Developer A's test methods are very specific to the implementation and they test just the method by itself.

To illustrate the badness of the write-a-test-for-a-method approach, let's consider changing the implementation. Suppose we find that the stack creates too much garbage (it does) and would like to replace the internal data structure with an array list. Developer A's tests are completely useless: they will fail, signaling that the changes we make are not right. His tests are showing a false positive, which is the worst outcome for a unit tests. Unit tests that have false positives undermine the mindshare of developers writing tests: if tests can't be trusted to evaluate the correctness of a change, why should we bother with them at all?

The same change (going from a linked list to an array list) can be done to Developer B's code *without changing a single line of his tests*. His tests are a complete success: they enabled us to refactor the code and be confident that our changes do not affect existing users and functionality.

The second ill effect of Developer A's methodology was the exposure of the internals of the class. It is now simple to subclass StackA and change the behavior of the internal methods. For example, we could override createNode and return a subclass of Node that stores the time it was pushed. This is a side effect of making createNode protected, and is not something we intended. When we change the implementation to use an array list, the subclass will break, potentially unknown to us, as it may be in a different source tree. Overall, Developer A's code has been extremely brittle. His object's encapsulation has been entirely broken; his tests will break with the slightest change to the implementation.