9 Unit Tests

Our profession has come a long way in the last ten years. In 1997 no one had heard of Test Driven Development. For the vast majority of us, unit tests were short bits of throw-away code that we wrote to make sure our programs "worked." We would painstakingly write our classes and methods, and then we would concoct some ad hoc code to test them. Typically this would involve some kind of simple driver program that would allow us to manually interact with the program we had written.

I remember writing a C++ program for an embedded real-time system back in the mid-90s. The program was a simple timer with the following signature:

```
void Timer::ScheduleCommand(Command* theCommand, int milliseconds)
```

The idea was simple; the `execute` method of the `Command` would be executed in a new thread after the specified number of milliseconds. The problem was, how to test it.

I cobbled together a simple driver program that listened to the keyboard. Every time a character was typed, it would schedule a command that would type the same character five seconds later. Then I tapped out a rhythmic melody on the keyboard and waited for that melody to replay on the screen five seconds later.

"I … want-a-girl … just … like-the-girl-who-marr … ied … dear … old … dad."

I actually sang that melody while typing the "." key, and then I sang it again as the dots appeared on the screen.

That was my test! Once I saw it work and demonstrated it to my colleagues, I threw the test code away.

As I said, our profession has come a long way. Nowadays I would write a test that made sure that every nook and cranny of that code worked as I expected it to. I would isolate my code from the operating system rather than just calling the standard timing functions. I would mock out those timing functions so that I had absolute control over the time. I would schedule commands that set boolean flags, and then I would step the time forward, watching those flags and ensuring that they went from false to true just as I changed the time to the right value.

Once I got a suite of tests to pass, I would make sure that those tests were convenient to run for anyone else who needed to work with the code. I would ensure that the tests and the code were checked in together into the same source package.

Yes, we've come a long way; but we have farther to go. The Agile and TDD movements have encouraged many programmers to write automated unit tests, and more are joining their ranks every day. But in the mad rush to add testing to our discipline, many programmers have missed some of the more subtle, and important, points of writing good tests.

The Three Laws of TDD

By now everyone knows that TDD asks us to write unit tests first, before we write production code. But that rule is just the tip of the iceberg. Consider the following three laws:[1]

http://doi.ieeecomputersociety.org/10.1109/MS.2007.85

First Law You may not write production code until you have written a failing unit test.

Second Law You may not write more of a unit test than is sufficient to fail, and not compiling is failing.

Third Law You may not write more production code than is sufficient to pass the currently failing test.

These three laws lock you into a cycle that is perhaps thirty seconds long. The tests and the production code are written *together*, with the tests just a few seconds ahead of the production code.

If we work this way, we will write dozens of tests every day, hundreds of tests every month, and thousands of tests every year. If we work this way, those tests will cover virtually all of our production code. The sheer bulk of those tests, which can rival the size of the production code itself, can present a daunting management problem.

Keeping Tests Clean

Some years back I was asked to coach a team who had explicitly decided that their test code *should not* be maintained to the same standards of quality as their production code. They gave each other license to break the rules in their unit tests. "Quick and dirty" was the watchword. Their variables did not have to be well named, their test functions did not need to be short and descriptive. Their test code did not need to be well designed and thoughtfully partitioned. So long as the test code worked, and so long as it covered the production code, it was good enough.

Some of you reading this might sympathize with that decision. Perhaps, long in the past, you wrote tests of the kind that I wrote for that `Timer` class. It's a huge step from writing that kind of throw-away test, to writing a suite of automated unit tests. So, like the team I was coaching, you might decide that having dirty tests is better than having no tests.

What this team did not realize was that having dirty tests is equivalent to, if not worse than, having no tests. The problem is that tests must change as the production code evolves. The dirtier the tests, the harder they are to change. The more tangled the test code, the more likely it is that you will spend more time cramming new tests into the suite than it takes to write the new production code. As you modify the production code, old tests start to fail, and the mess in the test code makes it hard to get those tests to pass again. So the tests become viewed as an ever-increasing liability.

From release to release the cost of maintaining my team's test suite rose. Eventually it became the single biggest complaint among the developers. When managers asked why their estimates were getting so large, the developers blamed the tests. In the end they were forced to discard the test suite entirely.

But, without a test suite they lost the ability to make sure that changes to their code base worked as expected. Without a test suite they could not ensure that changes to one part of their system did not break other parts of their system. So their defect rate began to rise. As the number of unintended defects rose, they started to fear making changes. They stopped cleaning their production code because they feared the changes would do more harm than good. Their production code began to rot. In the end they were left with no tests, tangled and bug-riddled production code, frustrated customers, and the feeling that their testing effort had failed them.

In a way they were right. Their testing effort *had* failed them. But it was their decision to allow the tests to be messy that was the seed of that failure. Had they kept their tests clean, their testing effort would not have failed. I can say this with some certainty because I have participated in, and coached, many teams who have been successful with *clean* unit tests.

The moral of the story is simple: *Test code is just as important as production code.* It is not a second-class citizen. It requires thought, design, and care. It must be kept as clean as production code.

Tests Enable the -ilities

If you don't keep your tests clean, you will lose them. And without them, you lose the very thing that keeps your production code flexible. Yes, you read that correctly. It is *unit tests* that keep our code flexible, maintainable, and reusable. The reason is simple. If you have tests, you do not fear making changes to the code! Without tests every change is a possible bug. No matter how flexible your architecture is, no matter how nicely partitioned your design, without tests you will be reluctant to make changes because of the fear that you will introduce undetected bugs.

But *with* tests that fear virtually disappears. The higher your test coverage, the less your fear. You can make changes with near impunity to code that has a less than stellar architecture and a tangled and opaque design. Indeed, you can *improve* that architecture and design without fear!

So having an automated suite of unit tests that cover the production code is the key to keeping your design and architecture as clean as possible. Tests enable all the -ilities, because tests enable *change*.

So if your tests are dirty, then your ability to change your code is hampered, and you begin to lose the ability to improve the structure of that code. The dirtier your tests, the dirtier your code becomes. Eventually you lose the tests, and your code rots.

Clean Tests

What makes a clean test? Three things. Readability, readability, and readability. Readability is perhaps even more important in unit tests than it is in production code. What makes tests readable? The same thing that makes all code readable: clarity, simplicity, and density of expression. In a test you want to say a lot with as few expressions as possible.

Consider the code from FitNesse in Listing 9-1. These three tests are difficult to understand and can certainly be improved. First, there is a terrible amount of duplicate code [G5] in the repeated calls to `addPage` and `assertSubString`. More importantly, this code is just loaded with details that interfere with the expressiveness of the test.

Listing 9-1 `SerializedPageResponderTest.java`

```
  public void testGetPageHieratchyAsXml() throws Exception
{

  crawler.addPage(root, PathParser.parse(“PageOne”));
  crawler.addPage(root, PathParser.parse(“PageOne.ChildOne”));
  crawler.addPage(root, PathParser.parse(“PageTwo”));
```

```
    request.setResource(“root”);
    request.addInput(“type”, “pages”);
    Responder responder = new SerializedPageResponder();
    SimpleResponse response =
      (SimpleResponse) responder.makeResponse(
        new FitNesseContext(root), request);
    String xml = response.getContent();

    assertEquals(“text/xml”, response.getContentType());
    assertSubString(“<name>PageOne</name>”, xml);
    assertSubString(“<name>PageTwo</name>”, xml);
    assertSubString(“<name>ChildOne</name>”, xml);
  }
  public void testGetPageHieratchyAsXmlDoesntContainSymbolicLinks()
  throws Exception {

                        WikiPage    pageOne    =    crawler.addPage(root,
PathParser.parse(“PageOne”));
    crawler.addPage(root, PathParser.parse(“PageOne.ChildOne”));
    crawler.addPage(root, PathParser.parse(“PageTwo”));

    PageData data = pageOne.getData();
    WikiPageProperties properties = data.getProperties();
                                WikiPageProperty        symLinks        =
properties.set(SymbolicPage.PROPERTY_NAME);
    symLinks.set(“SymPage”, ”PageTwo”);
    pageOne.commit(data);

    request.setResource(“root”);
    request.addInput(“type”, ”pages”);
    Responder responder = new SerializedPageResponder();
    SimpleResponse response =
      (SimpleResponse) responder.makeResponse(
        new FitNesseContext(root), request);
    String xml = response.getContent();
    assertEquals("text/xml", response.getContentType());
    assertSubString("<name>PageOne</name>", xml);
    assertSubString("<name>PageTwo</name>", xml);
    assertSubString("<name>ChildOne</name>", xml);
    assertNotSubString("SymPage", xml);
  }

  public void testGetDataAsHtml() throws Exception
  {
    crawler.addPage(root, PathParser.parse("TestPageOne"), "test page");

    request.setResource("TestPageOne");
    request.addInput("type", "data");
    Responder responder = new SerializedPageResponder();
    SimpleResponse response =
      (SimpleResponse) responder.makeResponse(
        new FitNesseContext(root), request);
    String xml = response.getContent();

    assertEquals("text/xml", response.getContentType());
    assertSubString("test page", xml);
    assertSubString("<Test", xml);
  }
```

For example, look at the `PathParser` calls. They transform strings into `PagePath` instances used by the crawlers. This transformation is completely irrelevant to the test at hand and serves only to obfuscate the intent. The details surrounding the creation of the `responder` and the gathering and casting of the `response` are also just noise. Then there's the ham-handed way that the request URL is built from a `resource` and an argument. (I helped write this code, so I feel free to roundly criticize it.)

In the end, this code was not designed to be read. The poor reader is inundated with a swarm of details that must be understood before the tests make any real sense.

Now consider the improved tests in Listing 9-2. These tests do the exact same thing, but they have been refactored into a much cleaner and more explanatory form.

Listing 9-2 `SerializedPageResponderTest.java (refactored)`

```
  public void testGetPageHierarchyAsXml() throws Exception {
    makePages("PageOne", "PageOne.ChildOne", "PageTwo");

    submitRequest("root", "type:pages");

    assertResponseIsXML();
    assertResponseContains(
              "<name>PageOne</name>", "<name>PageTwo</name>",
"<name>ChildOne</name>"
    );
  }

  public void testSymbolicLinksAreNotInXmlPageHierarchy() throws
Exception {
    WikiPage page = makePage("PageOne");
    makePages("PageOne.ChildOne", "PageTwo");

    addLinkTo(page, "PageTwo", "SymPage");

    submitRequest("root", "type:pages");

    assertResponseIsXML();
    assertResponseContains(
      "<name>PageOne</name>", "<name>PageTwo</name>",
         "<name>ChildOne</name>"
    );
    assertResponseDoesNotContain("SymPage");
  }

  public void testGetDataAsXml() throws Exception {
    makePageWithContent("TestPageOne", "test page");

    submitRequest("TestPageOne", "type:data");

    assertResponseIsXML();
    assertResponseContains("test page", "<Test");
  }
```

The BUILD-OPERATE-CHECK[2] pattern is made obvious by the structure of these tests. Each of the tests is clearly split into three parts. The first part builds up the test data, the second part operates on that test data, and the third part checks that the operation yielded the expected results.

Notice that the vast majority of annoying detail has been eliminated. The tests get right to the point and use only the data types and functions that they truly need. Anyone who reads these tests should be able to work out what they do very quickly, without being misled or overwhelmed by details.

Domain-Specific Testing Language

The tests in Listing 9-2 demonstrate the technique of building a domain-specific language for your tests. Rather than using the APIs that programmers use to manipulate the system, we build up a set of functions and utilities that make use of those APIs and that make the tests more convenient to write and easier to read. These functions and utilities become a specialized API used by the tests. They are a testing *language* that programmers use to help themselves to write their tests and to help those who must read those tests later on.

This testing API is not designed up front; rather it evolves from the continued refactoring of test code that has gotten too tainted by obfuscating detail. Just as you saw me refactor Listing 9-1 into Listing 9-2, so too will disciplined developers refactor their test code into more succinct and expressive forms.

A Dual Standard

In one sense the team I mentioned at the beginning of this chapter had things right. The code within the testing API *does* have a different set of engineering standards than production code. It must still be simple, succinct, and expressive, but it need not be as efficient as production code. After all, it runs in a test environment, not a production environment, and those two environment have very different needs.

Consider the test in Listing 9-3. I wrote this test as part of an environment control system I was prototyping. Without going into the details you can tell that this test checks that the low temperature alarm, the heater, and the blower are all turned on when the temperature is "way too cold."

Listing 9-3 **`EnvironmentControllerTest.java`**

```
 @Test
public void turnOnLoTempAlarmAtThreashold() throws Exception {
 hw.setTemp(WAY_TOO_COLD);
 controller.tic();
 assertTrue(hw.heaterState());
 assertTrue(hw.blowerState());
 assertFalse(hw.coolerState());
 assertFalse(hw.hiTempAlarm());
 assertTrue(hw.loTempAlarm());
}
```

There are, of course, lots of details here. For example, what is that `tic` function all about? In fact, I'd rather you not worry about that while reading this test. I'd rather you just worry about whether you agree that the end state of the system is consistent with the temperature being "way too cold."

Notice, as you read the test, that your eye needs to bounce back and forth between the name of the state being checked, and the *sense* of the state being checked. You see `heaterState`, and then your eyes glissade left to `assertTrue`. You see `coolerState` and your eyes must track left to `assertFalse`. This is tedious and unreliable. It makes the test hard to read.

I improved the reading of this test greatly by transforming it into Listing 9-4.

Listing 9-4 **`EnvironmentControllerTest.java (refactored)`**

```
 @Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
 wayTooCold();
```

```
  assertEquals("HBchL", hw.getState());
 }
```

Of course I hid the detail of the `tic` function by creating a `wayTooCold` function. But the thing to note is the strange string in the `assertEquals`. Upper case means "on," lower case means "off," and the letters are always in the following order: `{heater, blower, cooler, hi-temp-alarm, lo-temp-alarm}`.

Even though this is close to a violation of the rule about mental mapping,[3] it seems appropriate in this case. Notice, once you know the meaning, your eyes glide across that string and you can quickly interpret the results. Reading the test becomes almost a pleasure. Just take a look at Listing 9-5 and see how easy it is to understand these tests.

Listing 9-5 `EnvironmentControllerTest.java (bigger selection)`

```
 @Test
public void turnOnCoolerAndBlowerIfTooHot() throws Exception {
 tooHot();
 assertEquals("hBChl", hw.getState());
}

@Test
public void turnOnHeaterAndBlowerIfTooCold() throws Exception {
 tooCold();
 assertEquals("HBchl", hw.getState());
}

@Test
public void turnOnHiTempAlarmAtThreshold() throws Exception {
 wayTooHot();
 assertEquals("hBCHl", hw.getState());
}
@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
 wayTooCold();
```

```
  assertEquals(“HBchL”, hw.getState());
}
```

The `getState` function is shown in Listing 9-6. Notice that this is not very efficient code. To make it efficient, I probably should have used a `StringBuffer`.

Listing 9-6 **`MockControlHardware.java`**

```
public String getState() {
  String state = ””;
  state += heater ? “H” : “h”;
  state += blower ? “B” : “b”;
  state += cooler ? “C” : “c”;
  state += hiTempAlarm ? “H” : “h”;
  state += loTempAlarm ? “L” : “l”;
  return state;
}
```

`StringBuffer`s are a bit ugly. Even in production code I will avoid them if the cost is small; and you could argue that the cost of the code in Listing 9-6 is very small. However, this application is clearly an embedded real-time system, and it is likely that computer and memory resources are very constrained. The *test* environment, however, is not likely to be constrained at all.

That is the nature of the dual standard. There are things that you might never do in a production environment that are perfectly fine in a test environment. Usually they involve issues of memory or CPU efficiency. But they *never* involve issues of cleanliness.

One Assert per Test

There is a school of thought[4] that says that every test function in a JUnit test should have one and only one assert statement. This rule may seem draconian, but the advantage can be seen in Listing 9-5. Those tests come to a single conclusion that is quick and easy to understand.

But what about Listing 9-2? It seems unreasonable that we could somehow easily merge the assertion that the output is XML and that it contains certain substrings. However, we can break the test into two separate tests, each with its own particular assertion, as shown in Listing 9-7.

Listing 9-7 **`SerializedPageResponderTest.java (Single Assert)`**

```
  public void testGetPageHierarchyAsXml() throws Exception {
  givenPages("PageOne", "PageOne.ChildOne", "PageTwo");

  whenRequestIsIssued("root", "type:pages");

  thenResponseShouldBeXML();
}
public void testGetPageHierarchyHasRightTags() throws Exception {
  givenPages("PageOne", "PageOne.ChildOne", "PageTwo");

  whenRequestIsIssued("root", "type:pages");

  thenResponseShouldContain(
          "<name>PageOne</name>", "<name>PageTwo</name>",
"<name>ChildOne</name>"
  );
}
```

Notice that I have changed the names of the functions to use the common given-when-then[5] convention. This makes the tests even easier to read. Unfortunately, splitting the tests as shown results in a lot of duplicate code.

We can eliminate the duplication by using the TEMPLATE METHOD[6] pattern and putting the *given/when* parts in the base class, and the *then* parts in different derivatives. Or we could create a completely separate test class and put the *given* and *when* parts in the `@Before` function, and the *when* parts in each `@Test` function. But this seems like too much mechanism for such a minor issue. In the end, I prefer the multiple asserts in Listing 9-2.

I think the single assert rule is a good guideline.[7] I usually try to create a domain-specific testing language that supports it, as in Listing 9-5. But I am not afraid to put more than one assert in a test. I think the best thing we can say is that the number of asserts in a test ought to be minimized.

Single Concept per Test

Perhaps a better rule is that we want to test a single concept in each test function. We don't want long test functions that go testing one miscellaneous thing after another. Listing 9-8 is an example of such a test. This test should be split up into three independent tests because it tests three independent things. Merging them all together into the same function forces the reader to figure out why each section is there and what is being tested by that section.

Listing 9-8

```
  /**
* Miscellaneous tests for the addMonths() method.
*/
public void testAddMonths() {
  SerialDate d1 = SerialDate.createInstance(31, 5, 2004);

  SerialDate d2 = SerialDate.addMonths(1, d1);
  assertEquals(30, d2.getDayOfMonth());
  assertEquals(6, d2.getMonth());
  assertEquals(2004, d2.getYYYY());

  SerialDate d3 = SerialDate.addMonths(2, d1);
  assertEquals(31, d3.getDayOfMonth());
  assertEquals(7, d3.getMonth());
  assertEquals(2004, d3.getYYYY());

    SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1,
d1));
  assertEquals(30, d4.getDayOfMonth());
  assertEquals(7, d4.getMonth());
  assertEquals(2004, d4.getYYYY());

}
```

The three test functions probably ought to be like this:

- *Given* the last day of a month with 31 days (like May):

  1. *When* you add one month, such that the last day of that month is the 30th (like June), *then* the date should be the 30th of that month, not the 31st.

  2. *When* you add two months to that date, such that the final month has 31 days, *then* the date should be the 31st.

- *Given* the last day of a month with 30 days in it (like June):

  1. *When* you add one month such that the last day of that month has 31 days, *then* the date should be the 30th, not the 31st.

Stated like this, you can see that there is a general rule hiding amidst the miscellaneous tests. When you increment the month, the date can be no greater than the last day of the month. This implies that incrementing the month on February 28th should yield March 28th. *That* test is missing and would be a useful test to write.

So it's not the multiple asserts in each section of Listing 9-8 that causes the problem. Rather it is the fact that there is more than one concept being tested. So probably the best rule is that you should minimize the number of asserts per concept and test just one concept per test function.

F.I.R.S.T.[8]

Clean tests follow five other rules that form the above acronym:

Fast Tests should be fast. They should run quickly. When tests run slow, you won't want to run them frequently. If you don't run them frequently, you won't find problems early enough to fix them easily. You won't feel as free to clean up the code. Eventually the code will begin to rot.

Independent Tests should not depend on each other. One test should not set up the conditions for the next test. You should be able to run each test independently and run the tests in any order you like. When tests depend on each other, then the first one to fail causes a cascade of downstream failures, making diagnosis difficult and hiding downstream defects.

Repeatable Tests should be repeatable in any environment. You should be able to run the tests in the production environment, in the QA environment, and on your laptop while riding home on the train without a network. If your tests aren't repeatable in any environment, then you'll always have an excuse for why they fail. You'll also find yourself unable to run the tests when the environment isn't available.

Self-Validating The tests should have a boolean output. Either they pass or fail. You should not have to read through a log file to tell whether the tests pass. You should not have to manually compare two different text files to see whether the tests pass. If the tests aren't self-validating, then failure can become subjective and running the tests can require a long manual evaluation.

Timely The tests need to be written in a timely fashion. Unit tests should be written *just before* the production code that makes them pass. If you write tests after the production code, then you may find the production code to be hard to test. You may decide that some production code is too hard to test. You may not design the production code to be testable.

Conclusion

We have barely scratched the surface of this topic. Indeed, I think an entire book could be written about *clean tests*. Tests are as important to the health of a project as the production code is. Perhaps they are even more important, because tests preserve and enhance the flexibility, maintainability, and reusability of the production code. So keep your tests constantly clean. Work to make them expressive and succinct. Invent testing APIs that act as domain-specific language that helps you write the tests.

If you let the tests rot, then your code will rot too. Keep your tests clean.

Bibliography

[RSpec]: *RSpec: Behavior Driven Development for Ruby Programmers*, Aslak Hellesøy, David Chelimsky, Pragmatic Bookshelf, 2008.

[GOF]: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.