Chapter 4: Building Tests

If you want to do refactoring, the essential pre-condition is having solid tests. Even if you are fortunate enough to have a tool that can automate the refactorings, you still need tests. It'll be a long time before all possible refactorings could be automated in a refactoring tool.

I don't see this as a disadvantage. I've found that writing good tests greatly speeds up my programming, even if I'm not doing refactoring. This was a surprise for me and it is counter-intuitive for many programmers, so it's worth explaining why.

The Value of Self Testing Code

If you look at how most programmers spend their time, you'll find that writing code actually is quite a small fraction. Some time is spent figuring out what ought to be going on, some time is spent designing, but most time is spent debugging. I'm sure every reader can remember long hours of debugging, often long into the night. Every programmer can tell a story of a bug that took a whole day (or more) to find. Fixing the bug is usually pretty quick, but finding it is the nightmare. And then when you do fix it, there's always a chance that another one appears, which you might not even notice till much later, and then you spend ages finding that.

The event that started me on the road to self-testing code was a talk at OOPSLA in 1992. Someone (I think it was Dave Thomas) said off-handedly "classes should contain their own tests". That struck me as a good way to organize tests. I interpreted that as saying that each class should have its own method (called test) that can be used to test itself.

At that time I was also into incremental development so I tried adding test methods to classes as I completed each increment. The project I was working at that time was quite small so we put out increments every week or so. Running the tests was now fairly straightforward, but although they were easy to run they were still pretty boring to do. This was because every test produced output to the console which I had to check. Now I'm a pretty lazy person, and am prepared to work quite hard in order to avoid work. I realized that instead of me looking at the screen to see if it printed out some information from the model, I could get the computer to make that test. All I had to do was put the output I expected into the test code, and do a comparison. Now I could run each class's test method and it would just print "OK" to the screen if all was well. The class was now self-testing.

Make sure all tests are fully automatic and check their own results.

Now it was easy to run a test, indeed it was as easy as compiling. So I started to run tests every time I compiled. Shortly I began to notice my productivity had shot upward. I realized that I wasn't spending so much time debugging. If I added a bug that was caught by a previous test, then it would show up as soon as I ran that test. Since the test worked before that would then tell me that the bug was in the work I had done since I last tested. Since I ran the tests frequently, that was only a few minutes ago. I thus knew that the source of the bug was in the code I just wrote. Since that code was fresh in my mind and a small amount, the bug was easy to find. Bugs that would take an hour or more to find now took a couple of minutes at most. Not just had I built self-testing classes, but by running them frequently I had a powerful bug detector.

As I noticed this I got more aggressive about doing the tests. Instead of waiting for the end of increment, I would add the tests straight after writing a bit of function. Every day I would add a couple of new features and the tests to test them. These days I hardly ever spend more than a few minutes debugging.

A suite of tests is a powerful bug detector that decapitates the time it takes to find bugs.

Of course it is not so easy to convince others to follow this route. Writing the tests is a lot of extra code to write. Unless you have actually experienced the way it speeds up your programming, it does not make sense that it can do so. This is not helped by the fact that many people have never learned to write tests, to think about tests. When tests are manual they are gut-wrenchedly boring. But when they are automatic, they can actually be quite fun to write.

In fact one of the most useful times to write tests is before you start programming. When you need to add a feature, you begin by writing the test. This isn't as backwards as it sounds. By writing the test you are asking yourself what needs to be done to add the function. Writing the test also concentrates on the interface rather than the implementation (always a good thing). It also means you have a clear point at which you are done coding — the test works.

This notion of frequent testing is an important part of *Extreme Programming*. The name conjures up notions of programmers who are fast and loose hackers. But extreme programmers are very dedicated testers. They want to develop software as fast as possible, and they know that tests help you to go as fast as you possibly can.

> *Don't let the fear that tests can't catch every bug stop you from testing. If your tests only get half the bugs, they are still worthwhile, and you will usually do much better than that.*

That's enough of the polemic. While I think everyone would benefit by writing self-testing code, it is not the point of this book. This book is about refactoring, refactoring needs tests, so if you want to refactor you *have* to write tests. This chapter will give you a start in doing this for Java. This is not a testing book, so I'm not going to go into much detail. But with testing I've found that a remarkably small amount can have surprisingly big benefits.

As with everything else in this book, I'll describe the testing approach using examples. When I develop code I write the tests as I go, but often when I'm working with people on refactoring we have a body of non-self-testing code to work on. So first we have to make it self-testing before we refactor.

The standard java idiom for testing is the testing main. The idea is that every class should have a main function that tests that class. That's a reasonable convention (although not honored much) but can get awkward. The problem is that such a convention makes it tricky to run lots of tests easily. Another approach is to build separate test classes that work in a framework to make testing easier.

The JUnit Testing Framework

The testing framework I use is JUnit, an open-source testing framework developed by Erich Gamma and Kent Beck [JUnit]. The framework is very simple, yet allows you to do all the key things you need

to do in order to do some testing. In this chapter I'll use this framework to develop some tests for some io classes.

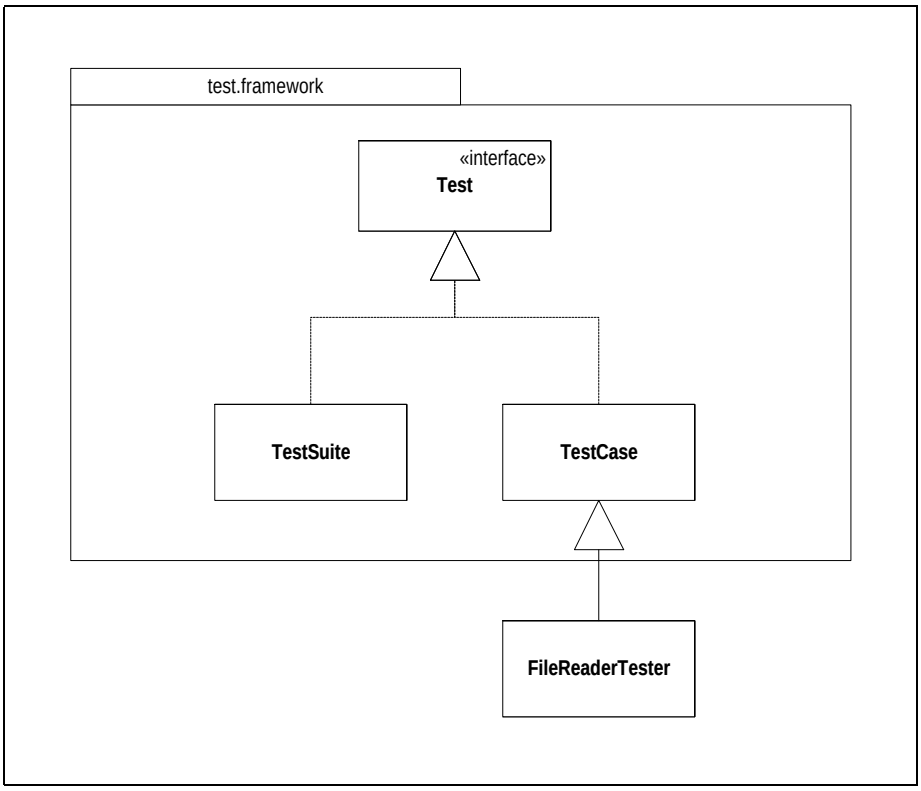

Figure 4.1: The composite structure of tests

To begin I'll create a FileReaderTester class to test the file reader. Any class that contains tests must subclass the test case class from the testing framework. The framework uses the composite pattern [Gang of Four] that allows you to group tests into suites. These suites can contain the raw test cases or other suites of test cases. This makes it easy to build a range of large test suites and run the tests automatically.

```
class FileReaderTester extends TestCase {
    public FileReaderTester (String name) {
        super(name);
    }
}
```

The new class has to have a constructor. After this I can start adding some test code. My first job is to set up the test fixture. A test fixture is essentially those objects which act as samples for testing. Since I'm reading a file I need to set up a test file.

```
Bradman     99.94  52   80   10   6996  334   29
Pollock     60.97  23   41   4    2256  274   7
Headley     60.83  22   40   4    2256  270*  10
Sutcliffe   60.73  54   84   9    4555  194   16
```

To further use the file I then prepare the fixture. The test case class provides two methods to manipulate the test fixture: setUp creates the objects and tearDown removes them. Both are implemented as null methods on test case. Most of the time you don't need to do a tear down (the garbage collector can handle it) but it is wise to use it here to close the file.

```
class FileReaderTester...
    protected void setUp() {
        try {
            _input = new FileReader("data.txt");
        } catch (FileNotFoundException e) {
            throw new RuntimeException ("unable to open test file");
        }
    }

    protected void tearDown() {
        try {
            _input.close();
        } catch (IOException e) {
            throw new RuntimeException ("error on closing test file");
        }
    }
```

Now I have the test fixture in place I can start writing some tests. The first test I'll do is to test the read method. To test this I'll read a few characters and then check that the character I read next is the right one.

```
    public void testRead() throws IOException {
        char ch = '&';
        for (int i=0; i < 4; i++)
            ch = (char) _input.read();
        assert('d' == ch);
    }
```

The automatic test is the assert method. If the value inside the assert is true, all is well. Otherwise we signal an error. I'll come to how the framework does that in a moment. First I'll describe how we run the

test. The first step is to create a test suite. To do this you create a method called suite.

```
public static Test suite() {
    TestSuite suite= new TestSuite();
    suite.addTest(new FileReaderTester("testRead"));
    return suite;
}
```

This test suite contains just one test case object, an instance of FileReaderTester. When I create a test case I give the constructor a string argument, which is the name of the method I'm going to test. This will create one object that will test that one method. The test is bound to the object by using Java's reflection capability. You can take a look at the downloaded source code to figure out how it does it. I just treat it as magic.

To run the tests I use a separate TestRunner class. There are two versions of TestRunner: one uses a cool GUI, the other a simple character interface. I can call the character interface version in the main.

```
class FileReaderTester
    public static void main (String[] args) {
        test.textui.TestRunner.run (suite());
    }
```

The code creates the test runner and gives it the suite to test. When I run it I see

```
.
Time: 0.110

OK (1 tests)
```

JUnit prints a period for each test that runs (so you can see progress). It tells you how long the tests have run for. It then says "OK" if nothing goes wrong and tells you how many tests ran. I could run a thousand tests and if all goes well I'll see that OK. This simple feedback is essential to self-testing code. Without it you'll never run the tests often enough. With it you can run masses of tests, go off for lunch (or some meeting) and see the results when you go back.

Run your tests frequently. Localized tests whenever you compile, every test at least every day.

In refactoring you only run a few tests that exercise the code you are working on. You can only run a few because they must be fast, otherwise they'll slow you down and you'll be tempted to not run them. (Don't give in to that temptation — retribution *will* follow.)

Essentially testing is all about this. Write a fixture, write some tests (with asserts) and add them to a suite. I typically build a suite for each tester class and then build larger suites for packages, and larger suites for the whole system.

What happens if something goes wrong? I'll demonstrate by putting in a deliberate bug.

```
public void testRead() throws IOException {
    char ch = '&';
    for (int i=0; i < 4; i++)
        ch = (char) _input.read();
    assert('2' == ch);    //deliberate error
}
```

Now the result looks like this.

```
.F
Time: 0.220

!!!FAILURES!!!
Test Results:
Run: 1 Failures: 1 Errors: 0
There was 1 failure:
1) FileReaderTester.testRead
test.framework.AssertionFailedError
```

The framework alerts me to the failure and tells me which test failed. The error message isn't particularly helpful though. I can make the error message better by using another form of assert.

```
public void testRead() throws IOException {
    char ch = '&';
    for (int i=0; i < 4; i++)
        ch = (char) _input.read();
    assertEquals('m',ch);
}
```

Most of the asserts you do are comparing two values to see if they are equal. So the framework includes an assertEquals. Not just is this convenient (it uses equals() on objects and == on values — which I always forget to do), it also allows a more meaningful error message.

```
.F
```

```
Time: 0.170

!!!FAILURES!!!
Test Results:
Run: 1 Failures: 1 Errors: 0
There was 1 failure:
1) FileReaderTester.testRead "expected:"m"but was:"d""
```

As well as catching failures (assertions coming out false), the framework also catches errors (unexpected exceptions). So if I write

```
public void testRead() throws IOException {
    char ch = '&';
    _input.close();
    for (int i=0; i < 4; i++)
        ch = (char) _input.read();// will throw exception
    assertEquals('m',ch);
}
```

I get

```
.E

Time: 0.110

!!!FAILURES!!!
Test Results:
Run: 1 Failures: 0 Errors: 1
There was 1 error:
1) FileReaderTester.testRead
java.io.IOException: Stream closed
```

It is useful to distinguish between failures and errors, since they tend to turn up differently and the debugging process is different.

JUnit also includes a nice graphical user interface. The color shows green if all tests pass and red if there are any failures. In some environments you can leave the GUI up all the time and the environment automatically links in any changes to your code. If that's the case this is a very convenient way to run the tests.

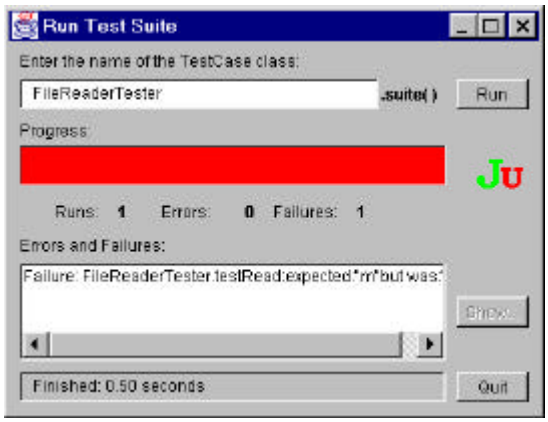

Figure 4.2: The Graphical User Interface of JUnit.

Unit and Functional Tests

This framework is used for unit tests, so I should mention the difference between unit tests and functional tests. These tests I'm talking about are unit tests. I write them to improve my productivity as a programmer. If they make the QA department happier, that's just a side effect. They are also very localized. Each test class works within a single package. It tests the interfaces to other packages, but beyond that it assumes the rest just works.

Functional tests are a different animal. They are written to ensure the software as a whole works. They provide quality assurance to the customer, and don't care about programmer productivity. They should be developed by a different team who delight in finding bugs. They will use heavy-weight tools and techniques to help them do this.

For refactoring purposes, it is the unit tests - the programmer's friend - that I count on.

Adding More Tests

Now we should continue adding more tests. The style I follow is to look at all the things the class should do, and test each one of them, looking for any conditions that might cause the class to fail. This is not the same as "test every public method" that some advocate. Testing should be risk driven; remember you are trying to find bugs, now or in the future. So I don't test accessors that just read and write a field. Since they are so simple, I'm not likely to find a bug there.

At the moment I'm looking at the read method. What else should it do? Well one thing it says is that it returns -1 at the end of the file (not a very nice protocol in my view, but I guess that makes it more natural for C programmers). So let's test it. My text editor tells me there's 141 characters in the file, so here's the test.

```
public void testReadAtEnd() throws IOException {
  int ch = -1234;
  for (int i = 0; i < 141; i++)
      ch =  _input.read();
  assertEquals(-1, ch);
}
```

To get the test to run, I have to add it to the suite.

```
public static Test suite() {
  TestSuite suite= new TestSuite();
  suite.addTest(new FileReaderTester("testRead"));
  suite.addTest(new FileReaderTester("testReadAtEnd"));
  return suite;
}
```

When this suite is run it tells each of its component tests (the two test cases) to run. Each test case executes setUp, then the body of the test code in the testing method, and finally tearDown. It is important to run setUp and tearDown each time so that the tests are isolated from each other. That means we can run them in any order and it doesn't matter.

That's good so far. A key trick with tests is to look for boundary conditions. For the read the boundaries would be the first character, the last character, and the character after the last character

```
public void testReadBoundaries()throwsIOException {
      assertEquals("read first char",'B', _input.read());
```

```
        int ch;
        for (int i = 1;i <140; i++)
            ch =  _input.read();
        assertEquals("read last char",'6',_input.read());
        assertEquals("read at end",-1,_input.read());
    }
```

Notice how you can add a message to the assert which will be printed if the test fails.

> *Think of the boundary conditions where things that might go wrong and concentrate your tests there*

Another part of looking for boundaries is looking for what special conditions could cause the test to fail. For files, empty files are always a good choice.

```
public void testEmptyRead() throws IOException {
  File empty = new File ("empty.txt");
  FileOutputStream out = new FileOutputStream(empty);
  out.close();
  FileReader in = newFileReader (empty);
  assertEquals (-1, in.read());
}
```

In this case I'm creating a bit of extra fixture just for this test. If I need an empty file for later I can move it into regular fixture by moving the code to setup.

```
protected void setUp(){
  try  {
      _input = new FileReader("data.txt");
      _empty = newEmptyFile();
  } catch(IOException e){
      throw new RuntimeException(e.toString());
  }
}

private FileReader newEmptyFile() throws IOException {
  File empty = new File ("empty.txt");
  FileOutputStream out = new FileOutputStream(empty);
  out.close();
  return newFileReader(empty);
}

public void testEmptyRead() throws IOException {
  assertEquals (-1, _empty.read());
}
```

So what happens if you read after the end of the file? Again -1 should be returned, and I'll just augment one of the other tests to probe that.

```
  public void testReadBoundaries()throwsIOException {
    assertEquals("read first char",'B', _input.read());
    int ch;
    for (int i = 1;i <140; i++)
        ch =  _input.read();
    assertEquals("read last char",'6',_input.read());
    assertEquals("read at end",-1,_input.read());
    assertEquals ("readpast end", -1, _input.read());
}
```

When you are doing tests, don't forget to check that expected errors occur properly. If you try to read a stream after it is closed, you should get an IOException. This too should be tested.

```
public void testReadAfterClose() throwsIOException{
  _input.close();
  try {
      _input.read();
      assert(false);// we should not get here
  } catch (IOException io) {}
}
```

Any other exception than the IOException will produce an error in the normal way.

Don't forget to test that exceptions are raised when things should go wrong

---

Fleshing out the tests continues along these lines. It takes a while to go through the interface to some classes to do this, but in the process you get to really understand the interface of the class. In particular it helps to think about error conditions and boundary conditions. That's another advantage for writing tests as you write code, or even before you write the production code.

When do you stop? I'm sure you have heard many times that you cannot prove a program has no bugs by testing. That's true, but does not affect the ability of testing to speed up programming. I've seen various proposals for rules to ensure you have tested every combination of everything. It's worth taking a look at these, but don't let them get to you. There is a point of diminishing returns on testing, and there is a

danger that by trying to write too many tests, you get discouraged and end up not writing any. You should concentrate on where the risk is. Look at the code and see where it gets complex, look at the function and consider the likely areas of error. Your tests will not find every bug, but as you refactor you will understand the program better and thus find more bugs. Although I always start refactoring with a test suite, I invariably add to it as I go along.

One of the tricky things about objects is that all the inheritance and polymorphism can make testing harder, as there are all these combinations to test. If you have three abstract classes with that collaborate where each has three subclasses, you have nine alternatives but twenty-seven combinations. I don't always try to test all the combinations possible, but I do try and test each alternative. It boils down to the risk in the combinations. If the alternatives are reasonably independent of each other, I'm not likely to try each combination. There's always a risk that I'll miss something, but it is better to spend a reasonable time to catch most bugs than to spend ages trying to get them all.

A difference between test code and production code is that it is okay to copy and edit test code. In fact when dealing with combinations and alternatives I often do that. First take regular pay event, now with seniority and disabled before the end of the year, now without seniority and disabled before the end of the year, and so on. With simple alternatives like that on top of a reasonable fixture I can generate tests very quickly.

I hope that's given you a feel for writing tests. There is a lot more I could say on this topic, but that would obscure the key message. Build up a good bug detector and run it frequently. It is a wonderful tool for any development, and an essential pre-condition for refactoring.