JUnit Test Infected: Programmers Love Writing Tests

Note: this article describes JUnit 3.8.x.

Testing is not closely integrated with development. This prevents you from measuring the progress of development- you can't tell when something starts working or when something stops working. Using *JUnit* you can cheaply and incrementally build a test suite that will help you measure your progress, spot unintended side effects, and focus your development efforts.

Contents

- The Problem
- Example
- Testing Practices
- Conclusions

The Problem

Every programmer knows they should write tests for their code. Few do. The universal response to "Why not?" is "I'm in too much of a hurry." This quickly becomes a vicious cycle- the more pressure you feel, the fewer tests you write. The fewer tests you write, the less productive you are and the less stable your code becomes. The less productive and accurate you are, the more pressure you feel.

Programmers burn out from just such cycles. Breaking out requires an outside influence. We found the outside influence we needed in a simple testing framework that lets us do a little testing that makes a big difference.

The best way to convince you of the value of writing your own tests would be to sit down with you and do a bit of development. Along the way, we would encounter new bugs, catch them with tests, fix them, have them come back, fix them again, and so on. You would see the value of the immediate feedback you get from writing and saving and rerunning your own unit tests.

Unfortunately, this is an article, not an office overlooking charming old-town Zürich, with the bustle of medieval commerce outside and the thump of techno from the record store downstairs, so we'll have to simulate the process of development. We'll write a simple program and its tests, and show you the results of running the tests. This way you can get a feel for the process we use and advocate without having to pay for our presence.

Example

As you read, pay attention to the interplay of the code and the tests. The style here is to write a few lines of code, then a test that should run, or even better, to write a test that won't run, then write the code that will make it run.

The program we write will solve the problem of representing arithmetic with multiple currencies. Arithmetic between single currencies is trivial, you can just add the two amounts. Simple numbers

suffice. You can ignore the presence of currencies altogether.

Things get more interesting once multiple currencies are involved. You cannot just convert one currency into another for doing arithmetic since there is no single conversion rate- you may need to compare the value of a portfolio at yesterday's rate and today's rate.

Let's start simple and define a class Money to represent a value in a single currency. We represent the amount by a simple int. To get full accuracy you would probably use double or java.math.BigDecimal to store arbitrary-precision signed decimal numbers. We represent a currency as a string holding the ISO three letter abbreviation (USD, CHF, etc.). In more complex implementations, currency might deserve its own object.

```
class Money {
    private int fAmount;
    private String fCurrency;

    public Money(int amount, String currency) {
        fAmount= amount;
        fCurrency= currency;
    }

    public int amount() {
        return fAmount;
    }

    public String currency() {
        return fCurrency;
    }
}
```

When you add two Moneys of the same currency, the resulting Money has as its amount the sum of the other two amounts.

```
public Money add(Money m) {
    return new Money(amount()+m.amount(), currency());
}
```

Now, instead of just coding on, we want to get immediate feedback and practice "code a little, test a little, code a little, test a little". To implement our tests we use the JUnit framework. To write tests you need to get the latest copy JUnit (or write your own equivalent- it's not so much work).

JUnit defines how to structure your test cases and provides the tools to run them. You implement a test in a subclass of TestCase. To test our Money implementation we therefore define MoneyTest as a subclass of TestCase. In Java, classes are contained in packages and we have to decide where to put MoneyTest. Our current practice is to put MoneyTest in the same package as the classes under test. In this way a test case has access to the package private methods. We add a test method testSimpleAdd, that will exercise the simple version of Money.add() above. A JUnit test method is an ordinary method without any parameters.

```
public class MoneyTest extends TestCase {
    //…
    public void testSimpleAdd() {
        Money m12CHF= new Money(12, "CHF");  // (1)
        Money m14CHF= new Money(14, "CHF");
        Money expected= new Money(26, "CHF");
        Money result= m12CHF.add(m14CHF);    // (2)
        Assert.assertTrue(expected.equals(result));     // (3)
    }
}
```

The testSimpleAdd() test case consists of:

1. Code which creates the objects we will interact with during the test. This testing context is commonly referred to as a test's *fixture*. All we need for the testSimpleAdd test are some Money objects.
2. Code which exercises the objects in the fixture.
3. Code which verifies the result.

Before we can verify the result we have to digress a little since we need a way to test that two Money objects are equal. The Java idiom to do so is to override the method *equals* defined in Object. Before we implement equals let's a write a test for equals in MoneyTest.

```
public void testEquals() {
    Money m12CHF= new Money(12, "CHF");
    Money m14CHF= new Money(14, "CHF");

    Assert.assertTrue(!m12CHF.equals(null));
    Assert.assertEquals(m12CHF, m12CHF);
    Assert.assertEquals(m12CHF, new Money(12, "CHF")); // (1)
    Assert.assertTrue(!m12CHF.equals(m14CHF));
}
```

The equals method in Object returns true when both objects are the same. However, Money is a *value object*. Two Monies are considered equal if they have the same currency and value. To test this property we have added a test (1) to verify that Monies are equal when they have the same value but are not the same object.

Next let's write the equals method in Money:

```
public boolean equals(Object anObject) {
    if (anObject instanceof Money) {
        Money aMoney= (Money)anObject;
        return aMoney.currency().equals(currency())
            && amount() == aMoney.amount();
    }
    return false;
}
```

Since equals can receive any kind of object as its argument we first have to check its type before we cast it as a Money. As an aside, it is a recommended practice to also override the method hashCode whenever you override method equals. However, we want to get back to our test case.

With an equals method in hand we can verify the outcome of testSimpleAdd. In JUnit you do so by a calling Assert.assertTrue, which triggers a failure that is recorded by JUnit when the argument isn't true. Since assertions for equality are very common, there is also an Assert.assertEquals convenience method. In addition to testing for equality with equals, it reports the printed value of the two objects in the case they differ. This lets us immediately see why a test failed in a JUnit test result report. The value a string representation created by the toString converter method. There are other asertXXXX variants not discussed here.

Now that we have implemented two test cases we notice some code duplication for setting-up the tests. It would be nice to reuse some of this test set-up code. In other words, we would like to have a common fixture for running the tests. With JUnit you can do so by storing the fixture's objects in instance variables of your TestCase subclass and initialize them by overridding the setUp method. The symmetric operation to setUp is tearDown which you can override to clean up the test fixture at the end of a test. Each test runs in its own fixture and JUnit calls setUp and tearDown for each test so that there can be no side effects among test runs.

```
public class MoneyTest extends TestCase {
    private Money f12CHF;
    private Money f14CHF;
```

```
    protected void setUp() {
        f12CHF= new Money(12, "CHF");
        f14CHF= new Money(14, "CHF");
    }
}
```

We can rewrite the two test case methods, removing the common setup code:

```
public void testEquals() {
    Assert.assertTrue(!f12CHF.equals(null));
    Assert.assertEquals(f12CHF, f12CHF);
    Assert.assertEquals(f12CHF, new Money(12, "CHF"));
    Assert.assertTrue(!f12CHF.equals(f14CHF));
}

public void testSimpleAdd() {
    Money expected= new Money(26, "CHF");
    Money result= f12CHF.add(f14CHF);
    Assert.assertTrue(expected.equals(result));
}
```

Two additional steps are needed to run the two test cases:

1. define how to run an individual test case,
2. define how to run a *test suite*.

JUnit supports two ways of running single tests:

- static
- dynamic

In the static way you override the runTest method inherited from TestCase and call the desired test case. A convenient way to do this is with an anonymous inner class. Note that each test must be given a name, so you can identify it if it fails.

```
TestCase test= new MoneyTest("simple add") {
    public void runTest() {
        testSimpleAdd();
    }
};
```

A template method[1] in the superclass will make sure runTest is executed when the time comes.

The dynamic way to create a test case to be run uses reflection to implement runTest. It assumes the name of the test is the name of the test case method to invoke. It dynamically finds and invokes the test method. To invoke the testSimpleAdd test we therefore construct a MoneyTest as shown below:

```
TestCase test= new MoneyTest("testSimpleAdd");
```

The dynamic way is more compact to write but it is less static type safe. An error in the name of the test case goes unnoticed until you run it and get a NoSuchMethodException. Since both approaches have advantages, we decided to leave the choice of which to use up to you.

As the last step to getting both test cases to run together, we have to define a test suite. In JUnit this requires the definition of a static method called suite. The suite method is like a main method that is specialized to run tests. Inside suite you add the tests to be run to a TestSuite object and return it. A TestSuite can run a collection of tests. TestSuite and TestCase both implement an interface called Test which defines the methods to run a test. This enables the creation of test suites by composing arbitrary TestCases and TestSuites. In short TestSuite is a Composite [1]. The code below illustrates the creation of a test suite with the dynamic way to run a test.

```
public static Test suite() {
    TestSuite suite= new TestSuite();
    suite.addTest(new MoneyTest("testEquals"));
    suite.addTest(new MoneyTest("testSimpleAdd"));
    return suite;
}
```

Since JUnit 2.0 there is an even simpler dynamic way. You only pass the class with the tests to a TestSuite and it extracts the test methods automatically.

```
public static Test suite() {
    return new TestSuite(MoneyTest.class);
}
```

Here is the corresponding code using the static way.

```
public static Test suite() {
    TestSuite suite= new TestSuite();
    suite.addTest(
        new MoneyTest("money equals") {
            protected void runTest() { testEquals(); }
        }
    );

    suite.addTest(
        new MoneyTest("simple add") {
            protected void runTest() { testSimpleAdd(); }
        }
    );
    return suite;
}
```

Now we are ready to run our tests. JUnit comes with a graphical interface to run tests. Type the name of your test class in the field at the top of the window. Press the Run button. While the test is run JUnit shows its progress with a progress bar below the input field. The bar is initially green but turns into red as soon as there is an unsuccessful test. Failed tests are shown in a list at the bottom. Figure 1 shows the TestRunner window after we run our trivial test suite.

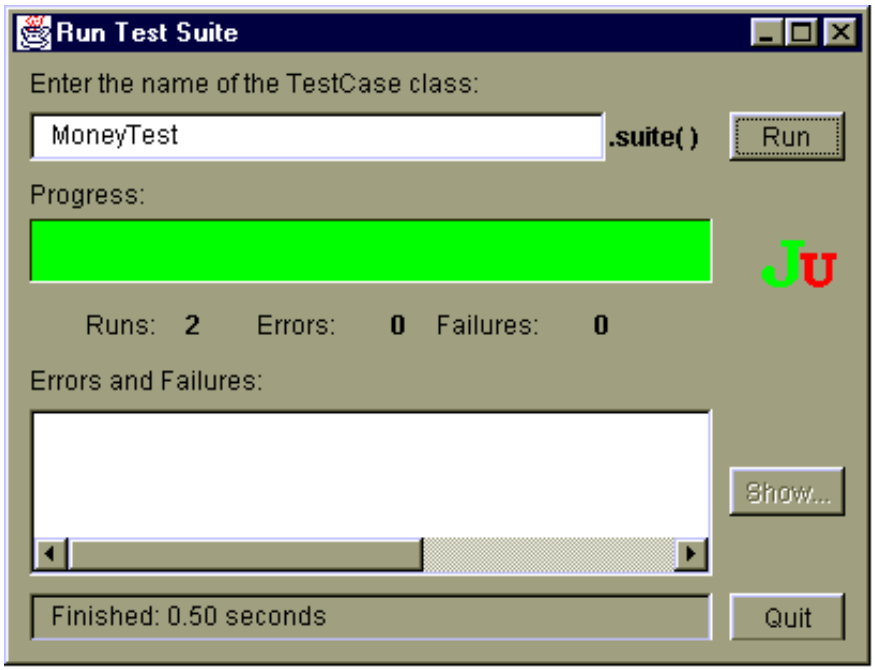

Figure 1: A Successful Run

After having verified that the simple currency case works we move on to multiple currencies. As mentioned above the problem of mixed currency arithmetic is that there isn't a single exchange rate. To avoid this problem we introduce a MoneyBag which defers exchange rate conversions. For example

adding 12 Swiss Francs to 14 US Dollars is represented as a bag containing the two Monies 12 CHF and 14 USD. Adding another 10 Swiss francs gives a bag with 22 CHF and 14 USD. We can later evaluate a MoneyBag with different exchange rates.

A MoneyBag is represented as a list of Monies and provides different constructors to create a MoneyBag. Note, that the constructors are package private since MoneyBags are created behind the scenes when doing currency arithmetic.

```
class MoneyBag {
    private Vector fMonies= new Vector();

    MoneyBag(Money m1, Money m2) {
        appendMoney(m1);
        appendMoney(m2);
    }

    MoneyBag(Money bag[]) {
        for (int i= 0; i < bag.length; i++)
            appendMoney(bag[i]);
    }
}
```

The method appendMoney is an internal helper method that adds a Money to the list of Moneys and takes care of consolidating Monies with the same currency. MoneyBag also needs an equals method together with a corresponding test. We skip the implementation of equals and only show the testBagEquals method. In a first step we extend the fixture to include two MoneyBags.

```
protected void setUp() {
    f12CHF= new Money(12, "CHF");
    f14CHF= new Money(14, "CHF");
    f7USD=  new Money( 7, "USD");
    f21USD= new Money(21, "USD");
    fMB1= new MoneyBag(f12CHF, f7USD);
    fMB2= new MoneyBag(f14CHF, f21USD);
}
```

With this fixture the testBagEquals test case becomes:

```
public void testBagEquals() {
    Assert.assertTrue(!fMB1.equals(null));
    Assert.assertEquals(fMB1, fMB1);
    Assert.assertTrue(!fMB1.equals(f12CHF));
    Assert.assertTrue(!f12CHF.equals(fMB1));
    Assert.assertTrue(!fMB1.equals(fMB2));
}
```

Following "code a little, test a little" we run our extended test with JUnit and verify that we are still doing fine. With MoneyBag in hand, we can now fix the add method in Money.

```
public Money add(Money m) {
    if (m.currency().equals(currency()) )
        return new Money(amount()+m.amount(), currency());
    return new MoneyBag(this, m);
}
```

As defined above this method will not compile since it expects a Money and not a MoneyBag as its return value. With the introduction of MoneyBag there are now two representations for Moneys which we would like to hide from the client code. To do so we introduce an interface IMoney that both representations implement. Here is the IMoney interface:

```
interface IMoney {
    public abstract IMoney add(IMoney aMoney);
    //
```

```
    ,,…
}
```

To fully hide the different representations from the client we have to support arithmetic between all combinations of Moneys with MoneyBags. Before we code on, we therefore define a couple more test cases. The expected MoneyBag results use the convenience constructor shown above, initializing a MoneyBag from an array.

```
public void testMixedSimpleAdd() {
    // [12 CHF] + [7 USD] == {[12 CHF][7 USD]}
    Money bag[]= { f12CHF, f7USD };
    MoneyBag expected= new MoneyBag(bag);
    Assert.assertEquals(expected, f12CHF.add(f7USD));
}
```

The other tests follow the same pattern:

- testBagSimpleAdd - to add a MoneyBag to a simple Money
- testSimpleBagAdd - to add a simple Money to a MoneyBag
- testBagBagAdd - to add two MoneyBags

Next, we extend our test suite accordingly:

```
public static Test suite() {
    TestSuite suite= new TestSuite();
    suite.addTest(new MoneyTest("testMoneyEquals"));
    suite.addTest(new MoneyTest("testBagEquals"));
    suite.addTest(new MoneyTest("testSimpleAdd"));
    suite.addTest(new MoneyTest("testMixedSimpleAdd"));
    suite.addTest(new MoneyTest("testBagSimpleAdd"));
    suite.addTest(new MoneyTest("testSimpleBagAdd"));
    suite.addTest(new MoneyTest("testBagBagAdd"));
    return suite;
}
```

Having defined the test cases we can start to implement them. The implementation challenge here is dealing with all the different combinations of Money with MoneyBag. Double dispatch [2] is an elegant way to solve this problem. The idea behind double dispatch is to use an additional call to discover the kind of argument we are dealing with. We call a method on the argument with the name of the original method followed by the class name of the receiver. The add method in Money and MoneyBag becomes:

```
class Money implements IMoney {
    public IMoney add(IMoney m) {
        return m.addMoney(this);
    }
    //…
}

class MoneyBag implements IMoney {
    public IMoney add(IMoney m) {
        return m.addMoneyBag(this);
    }
    //…
}
```

In order to get this to compile we need to extend the interface of IMoney with the two helper methods:

```
interface IMoney {
//…
    IMoney addMoney(Money aMoney);
    IMoney addMoneyBag(MoneyBag aMoneyBag);
}
```

To complete the implementation of double dispatch, we have to implement these methods in Money and

MoneyBag. This is the implementation in Money.

```
public IMoney addMoney(Money m) {
    if (m.currency().equals(currency()) )
        return new Money(amount()+m.amount(), currency());
    return new MoneyBag(this, m);
}

public IMoney addMoneyBag(MoneyBag s) {
    return s.addMoney(this);
}
```

Here is the implemenation in MoneyBag which assumes additional constructors to create a MoneyBag from a Money and a MoneyBag and from two MoneyBags.

```
public IMoney addMoney(Money m) {
    return new MoneyBag(m, this);
}

public IMoney addMoneyBag(MoneyBag s) {
    return new MoneyBag(s, this);
}
```

We run the tests, and they pass. However, while reflecting on the implementation we discover another interesting case. What happens when as the result of an addition a MoneyBag turns into a bag with only one Money? For example, adding -12 CHF to a Moneybag holding 7 USD and 12 CHF results in a bag with just 7 USD. Obviously, such a bag should be equal with a single Money of 7 USD. To verify the problem let's implement a test case and run it.

```
public void testSimplify() {
    // {[12 CHF][7 USD]} + [-12 CHF] == [7 USD]
    Money expected= new Money(7, "USD");
    Assert.assertEquals(expected, fMB1.add(new Money(-12, "CHF")));
}
```

When you are developing in this style you will often have a thought and turn immediately to writing a test, rather than going straight to the code.

It comes to no surprise that our test run ends with a red progress bar indicating the failure. So we fix the code in MoneyBag to get back to a green state.

```
public IMoney addMoney(Money m) {
    return (new MoneyBag(m, this)).simplify();
}

public IMoney addMoneyBag(MoneyBag s) {
    return (new MoneyBag(s, this)).simplify();
}

private IMoney simplify() {
    if (fMonies.size() == 1)
        return (IMoney)fMonies.firstElement()
    return this;
}
```

Now we run our tests again and voila we end up with green.

The code above solves only a small portion of the multi-currency arithmetic problem. We have to represent different exchange rates, print formatting, and the other arithmetic operations, and do it all with reasonable speed. However, we hope you can see how you could develop the rest of the objects one test at a time- a little test, a little code, a little test, a little code.

In particular, review how in the development above:

- We wrote the first test, testSimpleAdd, immediately after we had written add(). In general, your development will go much smoother if you write tests a little at a time as you develop. It is at the moment that you are coding that you are imagining how that code will work. That's the perfect time to capture your thoughts in a test.
- We refactored the existing tests, testSimpleAdd and testEqual, as soon as we introduced the common setUp code. Test code is just like model code in working best if it is factored well. When you see you have the same test code in two places, try to find a way to refactor it so it only appears once.
- We created a suite method, then extended it when we applied Double Dispatch. Keeping old tests running is just as important as making new ones run. The ideal is to always run all of your tests. Sometimes that will be too slow to do 10 times an hour. Make sure you run all of your tests at least daily.
- We created a new test immediately when we thought of the requirement that a one element MoneyBag should just return its element. It can be difficult to learn to switch gears like this, but we have found it valuable. When you are struck by an idea of what your system should do, defer thinking about the implementation. Instead, first write the test. Then run it (you never know, it might already work). Then work on the implementation.

Testing Practices

Martin Fowler makes this easy for you. He says, "Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead." At first you will find that you have to create a new fixtures all the time, and testing will seem to slow you down a little. Soon, however, you will begin reusing your library of fixtures and new tests will usually be as simple as adding a method to an existing TestCase subclass.

You can always write more tests. However, you will quickly find that only a fraction of the tests you can imagine are actually useful. What you want is to write tests that fail even though you think they should work, or tests that succeed even though you think they should fail. Another way to think of it is in cost/benefit terms. You want to write tests that will pay you back with information.

Here are a couple of the times that you will receive a reasonable return on your testing investment:

- During Development- When you need to add new functionality to the system, write the tests first. Then, you will be done developing when the test runs.
- During Debugging- When someone discovers a defect in your code, first write a test that will succeed if the code is working. Then debug until the test succeeds.

One word of caution about your tests. Once you get them running, make sure they stay running. There is a huge difference between having your suite running and having it broken. Ideally, you would run every test in your suite every time you change a method. Practically, your suite will soon grow too large to run all the time. Try to optimize your setup code so you can run all the tests. Or, at the very least, create special suites that contain all the tests that might possibly be affected by your current development. Then, run the suite every time you compile. And make sure you run every test at least once a day: overnight, during lunch, during one of those long meetings….

Conclusion

This article only scratches the surface of testing. However, it focuses on a style of testing that with a remarkably small investment will make you a faster, more productive, more predictable, and less

stressed developer.

Once you've been test infected, your attitude toward development is likely to change. Here are some of the changes we have noticed:

There is a huge difference between tests that are all running correctly and tests that aren't. Part of being test infected is not being able to go home if your tests aren't 100%. If you run your suite ten or a hundred times an hour, though, you won't be able to create enough havoc to make you late for supper.

Sometimes you just won't feel like writing tests, especially at first. Don't. However, pay attention to how much more trouble you get into, how much more time you spend debugging, and how much more stress you feel when you don't have tests. We have been amazed at how much more fun programming is and how much more aggressive we are willing to be and how much less stress we feel when we are supported by tests. The difference is dramatic enough to keep us writing tests even when we don't feel like it.

You will be able to refactor much more aggressively once you have the tests. You won't understand at first just how much you can do, though. Try to catch yourself saying, "Oh, I see, I should have designed this thus and so. I can't change it now. I don't want to break anything." When you say this, save a copy of your current code and give yourself a couple of hours to clean up. (This part works best you can get a buddy to look over your shoulder while you work.) Make your changes, all the while running your tests. You will be surprised at how much ground you can cover in a couple of hours if you aren't worrying every second about what you might be breaking.

For example, we switched from the Vector-based implementation of MoneyBag to one based on HashTable. We were able to make the switch very quickly and confidently because we had so many tests to rely on. If the tests all worked, we were sure we hadn't changed the answers the system produced at all.

You will want to get the rest of your team writing tests. The best way we have found to spread the test infection is through direct contact. The next time someone asks you for help debugging, get them to talk about the problem in terms of a fixture and expected results. Then say, "I'd like to write down what you just told me in a form we can use." Have them watch while you write one little test. Run it. Fix it. Write another. Pretty soon they will be writing their own.

So- give JUnit a try. If you make it better, please send us the changes so we can spread them around. Our next article will double click on the JUnit framework itself. We will show you how it is constructed, and talk a little about our philosophy of framework development.

We would like to thank Martin Fowler, as good a programmer as any analyst can ever hope to be, for his helpful comments in spite of being subjected to early versions of JUnit.

References

1. Gamma, E., et al. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995
2. Beck, K. Smalltalk Best Practice Patterns, Prentice Hall, 1996