CS032 2009 Lab 01 b
Testing and Debugging with JUnit

1 Introduction

In your coding experiences so far, you've likely gone straight from design to coding and only dealt with debugging when your code crashes or does not compile. In CS032, we're going to stress design and thinking about testing and debugging even before you code. In this lab, we'll look at a debugging tool called JUnit[1] that will allow you to create test cases for your code without running the entire program, which will come in more useful than simply vomiting code and then praying that it works on the first try. If that ever happens to you, savor the moment, because it will never happen again.

2 Background

JUnit is an application that allows you to build a suite of test cases. A test case consists of certain method calls and assertions that they are behaving the way you'd expect. A "successful" test case is one that finds a bug! Use this way of thinking when writing your test cases.

For more information on JUnit, you can visit www.junit.org. Of particular use are the "Getting Started" and "JavaDoc" links. The version of JUnit we are using is 3.8.1.

3 Program

You'll be testing two classes, one which represents a currency, and one which represents a bank that can deal with accounts in various currencies. We've provided these classes, `Money` and `Bank`, and an interface, `Constants`, that simply has some enumerated constants. These are hidden in the Jar file, and you don't really need to worry about them.

4 During the Lab

4.1 Formulate your test cases

The first thing you need to do is write down in pseudocode/English two test cases for all of the non-trivial methods in the two classes found in the stencil code. Each should contain:

a) What it is testing for (can be multiple things) – e.g. correct behavior
b) What the test case consists of (function + inputs; sequence of functions; ...) **AND** what the expected output is

Please don't be long winded (for your sake and ours) – a few words in a text file is enough for each method.

We have already implemented a buggy version of the code. The classes are called `MoneyImpl(Currency, double)` and `BankImpl(Currency)`. As an example, we might expect the following prepared for testing the `Money - double worldValue()` method:

```
a) The test will make sure that Money.worldValue() behaves properly for different currencies
when the rates are just initialized and also for when after the rates have been modified.

b) Pseudocode/English might look like this:
```

[1] JUnit takes its name from "unit testing," a common term for testing "units" – the smallest testable parts of your program – from inside your code. See http://en.wikipedia.org/wiki/Unit_test/ for more information.

```
usd2 = new MoneyImpl(USD, 2)
can2 = new MoneyImpl(CAN, 2)
euro2 = new MoneyImpl(EURO, 2)
make sure can2.worldValue() == usd2.worldValue() == euro2.worldValue()
// expected output for all is 2.0

EURO.setRate(3.0);
USD.setRate(1.5);
CAN.setRate(1.0);
usd4 = new MoneyImpl(USD, 4)
can6 = new MoneyImpl(CAN, 6)
make sure can6.worldValue() == usd4.worldValue() == euro2.worldValue()
// expected output for all is 6.0 since 6*1.0 == 4*1.5 == 2*3.0

usd6 = new MoneyImpl(USD, 6)
make sure can6.worldValue() != usd6.worldValue()
// expected output is 6.0 for can6 and 9.0 for usd6
```

Once you've done something similar for all the rest of the methods, show your test cases to a TA, and you've now reached **CHECKPOINT 1**.

4.2 Implement a testing class using JUnit

First create your own directory for this lab, i.e. `course/cs032/edu/brown/cs/cs032/junitlab/`. In accordance with the coding conventions, keep this all lowercase. This should mirror the name of your package (`edu.brown.cs.cs032.junitlab`). Remember, the top of every file should have the code:

```
package edu.brown.cs.cs032.junitlab;
```

Note that the support code `Constants` file uses enumerated constants, a feature of Java 1.5. Therefore, you'll need to compile and run your files with Java 1.5. This should automatically work for you, but if it doesn't, see a TA.

Copy the code for this lab, located in `/course/cs032/pub/labs/01_junit/buggyCode`, into your own directory and open up Eclipse. Create a new project for this lab and import the code into the IDE (see Eclipse lab for help; use Import then File System). After, right click on your project, select properties, Java Build Path, Libraries tab, Add External Jar, navigate to /course/cs032/lib/junitlab/support.jar, and select Add/OK. `Important:` In order to find out what all of the methods in the support code actually do, expand the support.jar file, and then the package, and double-click on the class files. They contain comments that would normally be included in a javadoc, and it's important that you read these to figure out what these methods are actually supposed to do.

4.3 Testing

`junit` runs its tests through the use of a `TestCase` class that you must extend. There will be a `JunitlabTest.java` file in your edu/brown/cs/cs032/junitlab/ directory. Note how `JunitlabTest` extends from `junit.framework.TestCase` and has the following after the package declaration:

```
import static edu.brown.cs.cs032.junitlab.Constants.Currency.*;
```

The `static import` line is another feature of Java 1.5. It brings into the namespace all the static members of the Currency class, so that you can just type `USD` instead of the fully-qualified name: `Currency.USD`.

The first method you should implement is `protected void setUp()`. Here is where you can instantiate objects that might get used in multiple test cases. `junit` calls this `setUp()` before it runs its tests; optionally you can also define a `tearDown()` method to clean up between test runs. We are overriding a method here, so exact spelling and letter case are necessary (if your test cases don't work, **check this first**).

We will now run through an example to give you an idea of how to code a test case, using the earlier pseudocode as an example. We'll create a `public void testWorldValue()` method to test whether `Money.worldValue()` behaves properly.

```
public void testWorldValue() {
        // assuming all variables were instantiated in setUp()...
        Assert.assertTrue(can2.worldValue() == usd2.worldValue());
        Assert.assertTrue(usd2.worldValue() == euro2.worldValue());
        Assert.assertTrue(can2.worldValue() == euro2.worldValue());

        EURO.setRate(3.0);
        USD.setRate(1.5);
        CAN.setRate(1.0);

        Assert.assertTrue(can6.worldValue() == usd4.worldValue());
        Assert.assertTrue(usd4.worldValue() == euro2.worldValue());
        Assert.assertTrue(can6.worldValue() == euro2.worldValue());

        // thus, we've now completed the earlier pseudocoded tests.
        // additional examples of tests that could be performed now are:
        Assert.assertFalse(usd2.worldValue() == usd4.worldValue());
        Assert.assertFalse(can2.worldValue() == can6.worldValue());

}
```

This `Assert` call is what makes up the core of debugging with JUnit. While it may seem like a simple test at this point, when classes begin to build up and projects become more complicated, you will be able to quickly and easily use JUnit to run a series of tests to see in which methods your bug lies. Look at the JUnit javadocs (`http://www.junit.org/junit/javadoc/3.8.1/index.htm`) to see other `Assert` calls you can make. When any of these asserts fail, JUnit will tell you in which test the failure occurred.

The

```
public static Test suite()
```

method sets up JUnit so that when it is run, all methods which start with `test` are executed.

Once JUnit pops up, inside "Test class name" put the full package name of your test class and hit Run. In this case, the ant script should automatically cause the class to appear (click the "...", but if not, make sure it is `edu.brown.cs.cs032.junitlab.JunitlabTest`. JUnit should run all your tests and tell you which ones have failed. JUnit should fail for the buggy code. A failure means that one of your tests didn't pass and an error usually means that an exception was thrown (i.e. a `NullPointerException`). In the lower window, you can see on which line the failure or error is stemming from. Show this to a TA, and you have now reached **CHECKPOINT 2**.

4.4 Implement remaining test cases.

You should now implement your remaining test cases as separate functions (i.e. create a testBankTotal() method, testAddMoney(), etc.). Then show your junit running on the buggy program to a TA. **CHECKPOINT 3.**

4.5 Fix the bugs

Edit the code until all the test cases pass. Use JUnit to help debug the code. You can keep JUnit open if you want; you'll just need to recompile your java code and JUnit will automatically reload class files for you. You'll know it's in working order when all the JUnit tests have passed. There are **three main bugs**

in the code and also a few spots where we **should throw a NullPointerException, but didn't**. Try to find them all.

This is the essence of using JUnit: if you have a huge library you're working on or you have multiple people editing functions, you can always make sure that your code is still functioning the way you want it to at the end of the day. Remember JUnit when you're working on your final projects.

Show a TA once you have all methods working (the bar is green). Point out where the bugs were and discuss if there are any additional test cases that you might create now that you've seen the code. You have now reached **CHECKPOINT 4**.

Note that you have several different ways of running the tests. The suite method isn't strictly needed. Look at the docs or ask a TA for more information. Visit http://www.junit.org or http://junit.sourceforge.net/doc/testinfected/testing.htm for an article similar to this lab, but in more detail. Also note that in the future `junit.jar` must be in your classpath (ask a TA; editing the classpath can be dangerous) before you can compile the JUnit test files. We have a copy in /course/cs032/lib/junit.jar.

Remember, "If the bar is green, your code is clean."

5 Checkpoints

1. Write pseudocode for test cases for the main methods of the Bank and Money interfaces.

2. Create the `JunitlabTest` class that compiles and has the `setUp()`, `testWorldValue()`, and `suite()` methods defined. Run JUnit for the three backends. The only version that should pass all your tests should be the clean version.

3. Fill in the remaining test cases in `JunitlabTest`. Run `junit` for just the buggy, empty, and clean backends. The only version that should pass all the tests should be the clean version.

4. Copy the buggy code and use JUnit to debug it until it passes all your tests.