Computer Science 507
Software Engineering
The College of Saint Rose
Spring 2014

Lab 6: Unit Testing with JUnit

Due: 6:00 PM, Monday, March 17, 2014

This week, we will work with the basics of the JUnit unit testing system.

You may work alone or in groups of up to size 4 for these exercises.

Unit Testing

Unit testing is a standard technique for many software development projects. The idea is to develop test cases along with the actual code to make sure the code not only works correctly when first written (at least as far as these tests are concerned) but also that it continues to work (again, at least in so much as it still passes the unit tests) when it is later modified.

There are many techniques, including libraries and frameworks, to support unit testing in nearly any production programming language. As the majority of you are most likely familiar with Java, we will experiment with unit testing in that language. If you are not a Java programmer, be sure to team up with someone who does have some Java experience. Fortunately, the techniques are similar with other languages and unit testing frameworks.

JUnit Introduction and Setup

JUnit is a framework to manage test cases for Java classes. The idea is that for a Java class you wish to test, you write an additional class to perform the unit tests.

JUnit is often used with the Eclipse IDE, but can be used on its own or with other IDEs as well. You are welcome to complete this lab with any IDE you wish, but the instructions will assume you are running Java from the command line on `mogul.strose.edu`. If you wish to use JUnit in a different environment, you will need to set it up as appropriate.

To set up the Java environment on mogul to include the JUnit jar files. There is a script you can run to do this:

```
. /home/cs507/junit/junit.bashrc
```

Note: in the above, the period and space are part of the command and are important!

You can either enter this each time you log into mogul and wish to use JUnit, or you can add it to the end of the `.bashrc` file in your home directory.

Please copy the files `SuperSimple.java` and `SuperSimpleTest.java` from `/home/cs507/junit` into a directory for this lab. These are an incredibly simple Java class and its corresponding JUnit

test class. Take a look at these files and make sure everything in them makes sense (there's not much).

To compile these, the following commands should work:

```
javac SuperSimple.java
javac SuperSimpleTest.java
```

You can then run the unit test with the following:

```
java org.junit.runner.JUnitCore SuperSimpleTest
```

Lab Question 1:
What output do you get when you run the "SuperSimple" unit test? (1 point)

Now, change the `isItSimple` method so it no longer "works" (*i.e.*, it doesn't return `true`) and recompile and rerun the test.

Lab Question 2:
What output do you get now? (1 point)

This example uses one kind of JUnit assertion, but there are many more.

Lab Question 3:
Write a new unit test class that creates a Java `ArrayList`, adds two items to the array list with the `add` method. It should then have assertions to verify that the two items are in the appropriate locations and that the `size` method returns 2. Include this test class and the output when you run the test in your submission. (5 points)

These kinds of tests work for many simple situations.

More Substantial Tests

Thinking about that `ArrayList` example, you will quickly realize that many tests will require some "set up" before and/or "tear down" after each test case. JUnit provides this capability with the `@Before` and `@After` annotations. The test method annotated with `@Before` will run before *each* test method (*i.e.*, those annotated with `@Test`) and the `@After` method will run after each (whether the test succeeds or fails).

Note that you will need an additional `import` for each of these:

```
import org.junit.After;
import org.junit.Before;
```

Lab Question 4:

Make a copy of your `ArrayList` test from the previous section and modify it so that the construction of the `ArrayList` and the addition of the two items are in a method annotated with `@Before` (note that you will likely now need to declare your `ArrayList` as an instance variable), and separate out your `ArrayList` tests into 3 separate tests: one to check that the first element is correct, a second to check that the second element is correct, and a third to check that the `size` method returns 2. Include this test class and the output when you run the test in your submission. (6 points)

Lab Question 5:

Develop a Java class that implements a non-trivial data structure and/or algorithm – nothing more complicated than you'd see in a typical undergraduate data structures course – and a JUnit test class that tests the important features. Possibilities include a sorting algorithm, a list or tree data structure, or a class that performs some mathematical calculations that you can check easily. You must use at least two JUnit assertion methods that were not used in the previous examples or lab questions. In your submission, include the Java code for both the implementation and JUnit test class. (12 points)

Submission

Before 6:00 PM, Monday, March 17, 2014, submit your lab for grading. Package up all required files into an appropriate archive format (`.tar.gz`, `.zip`, and `.7z` are acceptable) and upload a copy of the using Submission Box at `http://sb.teresco.org` under assignment "JUnit".

Grading

This lab will be graded out of 25 points.