Java Unit Testing Guide

Introduction

What do you think of when you hear "a unit of work" ? In the sense of programming, a unit of work could loosely be put as the smallest logical functional piece of code. As an example, consider a method that returns a list of squares of a given list of integers:

```
public List<Integer> squares(List<Integer> numbers){
     List<Integer> squares = new ArrayList<Integer>();
     for (int i = 0; i < numbers.length; i++) {
        squares.add(numbers.get(i)*numbers.get(i));
     }
     return squares;
}
```

This is a piece of code that only has one purpose, as defined above.

Very much related to the concept of a "unit piece of code", what is a unit test ? A unit test is a piece of code that tests a single assumption about another logical piece of code. A formal written unit test is characterized by a known input and an expected output, which is worked out before the test is executed. As an example, for testing a single assumption about the above piece of code, a test could take the form:

```
assert(squares(Arrays.asList(1,2,3).equals(Array.asList(1,4,9));
```

This test asserts an assumption about the output given a certain input.

Testing small pieces of functionality using unit tests can help isolate bugs much more effectively than looking at the larger framework of a project.

JUnit is Java's unit testing framework. We'll guide you on using this framework by introducing the idea of "test-driven development" i.e set up test data for a piece of code and then implement it.

Example

We'll take you through the test-driven design recipe using a simple example of implementing an `add` method in a class `Calculator.java`. This method should take in a list of integers and return their sum

The Design Recipe

So how exactly would we go about employing ”test-driven development” in this case ? The basic steps are for our recipe are:

1. Writing **examples** of the data your method will process. For instance, an example of `add` would be:

   ```
   Input: [1,2]
   Output: 3
   ```

   Think of all sorts of edge cases your method may encounter and what the expected output should be and write more examples.

2. Outline the **method signature** using header comments on each method. Understand and write the relationship between the input and output in words in the header comment. For instance, for `add` we have,

   ```
   /**
    * Takes in a list of integers and outputs their sum
    * @param numbers - a list of integers to add
    * @return - a single number giving the sum of the integers
    */
    public static int add(List<Integer> numbers)
   ```

3. Use the method signature and your comments to write **test cases** in `CalculatorTest.java`. These tests must follow the method specifications and input/output relationship as described. For example:

   ```
   public testAddOneNumber(){
        int actualSum = Calculator.add([3]);
        int expectedSum = 3;

        // checking program state
        assertThat(actualSum, is(expectedSum));
   }

   public testAddSeveralNumbers(){
        int actualSum = Calculator.add([3, 5, 7, 9]);
        int expectedSum = 24;

        // checking program state
        assertThat(actualSum, is(expectedSum));
   }
   ```

Once you're used to the format for asserting assumptions in your tests, you can shorten your test code:

```
public testAddSeveralNumbers(){
     assertThat(Calculator.add([3, 5, 7, 9]), is(24));
}
```

4. **Implement** the method now!

```
/**
 * Takes in a list of integers and outputs their sum
 * @param numbers - a list of integers to add
 * @return - a single number giving the sum of the integers
 */
 public static int add(List<Integer> numbers){
     if(numbers == null || numbers.isEmpty() {
         throw new InvalidInputException();
     }
     int sum = 0;
         for (int i = 0; i < numbers.length(); i++) {
             sum += numbers.get(i);
         }
         return sum;
 }
```

   Think of more tests covering all unique cases you can think of.

5. **Run** your test cases and verify that your method works!

Syntax, Syntax, Syntax !!

If you are working on one of the java projects, and are confused by the syntax in the unit testing files, don't worry! Help is at hand. The JUnit framework uses **annotations** to tell the compiler that a certain method is a test method. Here are the two annotations you will need:

1. `@Test`: Identifies a method as a unit test
2. `@Ignore`: Will ignore the test method. This is useful when you change the underlying code and the test case has not yet been adapted to test for the changed code. Ignored tests shouldn't show up in anything that you hand in, but this annotation can be useful to you while you're working on projects and testing them.

As you will notice in the stencil files, annotatations appear at the top of a method, example:

```
@Test
public testAddOneNumber(){
     int actualSum = Calculator.add([3]);
     int expectedSum = 3;

     // checking program state
     assertThat(actualSum, is(expectedSum));
}
```

Feel free to take a look at other potentially useful annotations at `http://junit.sourceforge.net/javadoc/org/junit/package-summary.html` although, you shouldn't need to use other annotations for projects in this class.

But what about the `assertThat, assertTrue` lines ? Help! I've only ever seen `assert(something)` in java code. Fret not, these work quite like the assert statements you may have already seen and used in your java code. There are however, several types of `asserts` offered by the junit package that can help make your test cases as fluid to read as plain english! Some common ones are outlined below for your reference:

1. `assertTrue([optional message string], boolean)`: Asserts that the boolean condition is true, and if not, gives an assertion failure with the specified message string (if any).
2. `assertEquals([optional message string], expected, actual)`: Tests that two values are the same. If not, gives an assertion failure with the given message string (if any). **Note:** When using this on two arrays, this will pass only if the references point to the same array object in memory. This will **not** check the array contents.
3. `assertNull([optional string message], object)`: These assertion names are really quite like english aren't they ?

For even more asserts, take a look at this awesome javadoc: `http://junit.sourceforge.net/javadoc/org/junit/Assert.html`

There are more fancy asserts from another supporting library called hamcrest. We've already imported hamcrest for you, so don't worry about what it is, but one of the useful asserts you might want to use and that can be seen in one of the `add` examples as well is `assertThat(actual, is(expected))`. A nice, readable way of stating your test expectations! If you are curious to learn more about and/or use other hamcrest asserts, check out: `http://hamcrest.org/JavaHamcrest/javadoc/1.3/org/hamcrest/MatcherAssert.html` Though the plain old junit asserts will suffice for this course.

Summary of useful resources

Here's a short list of all online resources mentioned at various places in this document:

1. **JUnit annotations:** `http://junit.sourceforge.net/javadoc/org/junit/package-summary.html`
2. **JUnit assertions:** `http://junit.sourceforge.net/javadoc/org/junit/Assert.html`
3. **Hamcrest assertions:**
   `http://hamcrest.org/JavaHamcrest/javadoc/1.3/org/hamcrest/MatcherAssert.html`