Lab Exercise 2
CS 2334

September 1, 2016

Introduction

Producing quality code requires us to take steps to ensure that our code actually performs as we expect it to. We must write careful specifications for each method that we implement. For a given method, this includes what the inputs are (i.e., the parameters and their expected values), and the results that are to be produced (return value and side effects). Once a method or group of methods is implemented, we must also perform appropriate testing. *Unit testing* is a formal technique that requires us to implement a set of tests that ensure that *each* piece of code is exercised and produces the correct results. In practice, each time a code base is modified, this set of tests is executed before the code is released for general use.

In this laboratory, we will use the *JUnit* tool to produce and evaluate a set of tests. We have provided a specification and implementation of a couple classes. Your task is to write a set of tests for one of these classes, and discover the bugs in our implementation and fix them.

Learning Objectives

By the end of this laboratory exercise, you should be able to:

1. Read and understand method-level specifications
2. Read and understand previously written code
3. Create JUnit test cases

4. Use JUnit tests to discover bugs and to ultimately verify correctness of the methods
5. Correct all bugs in the code base so that all tests pass successfully

Proper Academic Conduct

This lab is to be done individually. Do not look at or discuss solutions with anyone other than the instructor or the TAs. Do not copy or look at specific solutions from the net.

Preparation

1. Check to see if the EclEmma plug-in is installed:
   (a) *Help* menu: select *About Eclipse Platform / Installation Details*
   (b) You will see a list of installed plugins. If *EclEmma* is on the list, then you are done.
2. If not already installed, then install the EclEmma plug-in:
   (a) *Help* menu: select *Install new software*
   (b) *Work with* box: enter `http://update.eclemma.org`
   (c) Select *EclEmma* from the list. Click *Next*
   (d) Read and accept the license agreement. Click *Finish*
3. Create a new project called *lab2*
4. Download the lab 2 implementation and unzip the file into your file system: `http://www.cs.ou.edu/~fagg/classes/cs2334/labs/lab2/lab2.zip`
5. Import lab 2 into your Eclipse workspace:
   (a) In Eclipse, select your *lab2* project
   (b) Select *File/Import*
   (c) Select *General/File System*. Click *Next*
   (d) For *From Directory*: browse to the folder that contains the unzipped lab2, and then into *src*. Click *OK*.

(e) Select the checkboxes for all of the Java files.

(f) *Into folder*: browse to your new project and then into *src*. Click *OK*.

(g) Click *Finish*.

6. Import JUnit into the project

   (a) Right-click on lab2 and select *properties*

   (b) Select *Java Build Path*

   (c) Click *Libraries* tab

   (d) If this project is imported: you may need to select *JUnit4* and click *Remove*

   (e) Click *Add Library*

   (f) Select *JUnit*. Click *Next*

   (g) Select the most recent version (*JUnit 4*)

   (h) Click *Finish*

   (i) Click *OK*

7. Carefully examine the code for the *Item* and *Inventory* classes. Note that this code has a number of bugs in it, which we will find over the course of solving this lab

Unit Tests

Within the lab2.zip file, we have included the *ItemTest* class as an example:

```
import org.junit.Test;
import org.junit.Assert;

/**
 * Testing class for Item object
 *
 * @author Taner Davis
 * @version 20160827
 */
public class ItemTest
{

    /**
     * Test the empty Item constructor
     */
    @Test
    public void emptyItemConstructorTest()
    {
        //Use the default constructor
        Item item = new Item();

        //The name should be null, and the price and weight zero
        Assert.assertNull(item.getName());
        Assert.assertEquals(0, item.getPrice());
        Assert.assertEquals(0, item.getWeight(), 0.01);

    }

    /**
     * Test the Item constructor with only a String parameter
     */
    @Test
    public void singleParameterConstructorTest()
    {
        //Use the single-parameter constructor
        Item item = new Item("Portal Gun");

        /*
         * The name should match its initial parameter,
         * the price and weight should be zero
         */
        Assert.assertEquals("Portal Gun", item.getName());
        Assert.assertEquals(0, item.getPrice());
        Assert.assertEquals(0, item.getWeight(), 0.01);

    }
```

```
    /**
     * Test the Item constructor using a string parameter
     * and a double weight
     */
    @Test
    public void doubleParameterConstructorTest()
    {
        //Use the double-parameter constructor
        Item item = new Item("Battle Axe", 98.7);

        /*
         * The name should match its initial parameter,
         * the weight should equal its initial value,
         * and the price should be zero
         */
        Assert.assertEquals("Battle Axe", item.getName());
        Assert.assertEquals(0, item.getPrice());
        Assert.assertEquals(98.7, item.getWeight(), 0.01);

    }

    /**
     * Test full constructor and the getters
     */
    @Test
    public void fullConstructorTest()
    {
        // Use full constructor
        Item item = new Item("Whip", 10.1, 80);

        /*
         * The name should match its initial parameter,
         * and the weight and price should equal their
         * initial values
         */
        Assert.assertEquals(80, item.getPrice());
        Assert.assertEquals(10.1, item.getWeight(), 0.01);
        Assert.assertTrue(item.getName().equals("Whip"));
    }
```

```
    /**
     * Test all mutator methods
     */
    @Test
    public void allMutatorsTest()
    {
        // Use full constructor
        Item item = new Item("Change This", 999.9, 999);

        // Set name, price, and weight properties
        item.setName("Scythe");
        item.setPrice(125);
        item.setWeight(38.3);

        /*
         * The name should match the parameter passed
         * into the mutator methods, and the weight and
         * price should equal the values passed to their
         * respective mutator methods.
         */
        Assert.assertEquals(125, item.getPrice());
        Assert.assertEquals(38.3, item.getWeight(), 0.01);
        Assert.assertTrue(item.getName().equals("Scythe"));
    }

    /**
     * Test the String representation of an Item
     */
    @Test
    public void itemToStringTest()
    {
        Item item = new Item("MegaBuster", 27.9, 500);

        Assert.assertEquals("MegaBuster, 27.9 Fantasy Units, 500 Gold\n", item.↩
            toString());
    }
}
```

Note that, depending on your particular OS/Java installation, the import lines at the top of the testing class may be different. If the *Assert* class is undefined, then the simple thing to do is to first delete these two import lines. Then, mousing over one of the undefined *Assert* references, Eclipse will give you the option of importing the correct class.

A unit test file is a class in its own right, containing one or more methods (often named in a convention similar to *allMutatorsTest*, *singleParameterConstructorTest*, etc.). Each of these methods are preceded by the *@test* tag. This tells the compiler to configure this method as one of the tests to be executed.

Each unit test contains three sections of code (which may be intertwined):

1. Creation of a set of objects that will be used for testing

2. Calling of the methods to be tested, often storing their results

3. A set of *Assertions* that test the results returned by the method calls. Each assertion is a declaration by the test code of some condition that must hold if the code is performing correctly. A typical test will have several such assertions.

In *fullConstructorTest()* in *ItemTest.java*, an item object is created with the name "Whip" and a weight and price (in gold pieces) of 10.1 and 80, respectively. This test method confirms that each of these three properties is set correctly during the construction of the object. For example:

```
Assert.assertEquals(10.1, item.getWeight(), 0.01);
```

queries the object's weight through the weight getter method and compares it to the expected value of 10.1 (expected since this is the value that was used in the constructor). Remember that it is not appropriate to simply test the equality of two doubles (since two values can be arbitrarily close to one-another and still not be exactly equal). Instead, this double version of *assertEquals()* asks whether the two values are within 0.01 of one another. If this is the case, then this assertion will pass. On the other hand, if the returned price is very different than the expected value, then the test will fail.

The *assertTrue()* method will test an arbitrary condition. For example:

```
Assert.assertTrue(item.getName().equals("Master Sword"));
```

states the the name must be exactly equal to "Master Sword" (remember that *String.equals()* requires an exact string match in order to return true). Through the use of this type of assertion, one can check any Boolean condition. A link to the full *Assert* class is given below.

Within Eclipse, you can execute a unit test by pressing the  button (upper tool pane) and selecting your unit test. A JUnit window pane will appear on the left-hand-side of the interface and show you how many tests passed/failed. If a test fails, you will be able to click on it to see exactly which line resulted in the failure. A failure indicates a bug in the implementation of your class (or in the test itself).

EclEmma will highlight your tested code to indicate how well it is *covered* by the unit tests. In particular, Eclipse will highlight each line of code accordingly:

- Green: the line has been "touched" by one or more tests

- Red: the line hs not been touched

- Yellow: only one part of the branch (if, while, for, switch) has been touched by the tests

Your goal is to have all of your tested class be highlighted in green.

When you are writing tests, you should not rely on your implementation to produce the expected values. Instead, you should work out by hand what the expected values should be. This way, your test is independent of your implementation. Also, it is good practice to write your tests *before* you write your methods.

Inventory Unit Test

Your task for this lab is to write a set of unit tests for the methods in the **Inventory** class. Here is the procedure:

1. Create a new JUnit test class:
   (a) In the package explorer, right-click on *Inventory.java*
   (b) Select *New/JUnit Test Case*
   (c) Select *New JUnit 4 test*
   (d) The source folder should be *lab2/src*
   (e) The name should be *InventoryTest*
   (f) Class under test should be *Inventory*
   (g) Click *Finish*. This will create and open a new class called *InventoryTest*
2. Write a set of tests that confirm that all methods of the *Inventory* class perform correctly. Note that in these tests, you will need to create at least one Inventory object and populate it with a number of Item objects of various names, weights and prices. The set of tests that you write must *cover* all of the cases in the methods in this class. This means that you must test all possible paths through the code (e.g., every *if* and *else* branch).
3. As you execute your tests, you will discover a number of errors in the Inventory implementation. Fix these errors and confirm that all of the bugs are resolved using your unit tests.

Final Steps

1. Generate Javadoc using Eclipse.
   - Select *Project/Generate Javadoc...*
   - Make sure that your project is selected, as are the Driver, Item and Inventory classes
   - Select *Private* visibility
   - Use the default destination folder
   - Click *Finish*
2. Open the *lab2/doc/index.html* file using your favorite web browser or Eclipse (double clicking in the package explorer will open the web page). Check to make sure that that both of your classes are listed and that all of your documented methods have the necessary documentation.
3. If you complete the above instructions during lab, you may have your implementation checked by one of the TAs.

Submission instructions

Before submission, finish testing your program by executing your unit tests. If your program passes all tests, then you are ready to attempt a submission. Here are the details:

- All required components (source code and compiled documentation) are due at 11:59pm on Friday, September 2. **Submission must be done through the Web-Cat server.**
- Method 1: Submit through Eclipse
  1. Select your project.
  2. From the Project menu, select *Submit Assignment*.
  3. Under *Select the assignment to submit*, select *CS 2334/Lab 02 (2334): Inventory*.
  4. Click *Change Username or Password...*. Enter your Web-Cat username and password. Click *OK*. You should only need to do this step once per session.

5. Click *Finish*.
6. Your browser should automatically open a Web-Cat page that shows your submission being graded. After a short wait, the page will show a report of your submission. See the main class web page for a link that describes the Web-Cat output.

- Method 2: Submit directly to the Web-Cat server

1. From the File menu, select *Export*.
2. Select *Java/JAR File*. Click *Next*.
3. Select and expand your project folder.
4. Select your *src* and *doc* folders.
5. Select *Export Java source files and resources*.
6. Select an export destination location (e.g., your *Documents* folder/directory). This file should end in *.jar*
7. Select *Add directory entries*.
8. Click *Finish*.
9. In your web browser, login to the Web-Cat server.
10. Click the *Submit* button.
11. Browse to your jar file.
12. Click the *Upload Submission* button.
13. The next page will give you a list of all files that you are uploading. If you selected the correct jar file, then click the *Confirm* button.
14. Your browser will then open a Web-Cat page that shows your submission being graded. After a short wait, the page will show a report of your submission. See the main class web page for a link that describes the Web-Cat output.

References

- The API of the *Assert* class can be found at:
`http://junit.sourceforge.net/javadoc/org/junit/Assert.html`
- JUnit tutorial in Eclipse:
`https://dzone.com/articles/junit-tutorial-beginners`

Rubric

The project will be graded out of 100 points. The distribution is as follows:

Correctness/Testing: 45 points

The Web-Cat server will grade this automatically upon submission. Your code will be compiled against a set of tests (called *Unit Tests*). These unit tests will not be visible to you, but the Web-Cat server will inform you as to which tests your code passed/failed. This grade component is proportional to the fraction of tests that your code passes (so 22.5 points means that your code passed half of the tests)

Style/Coding: 20 points

The Web-Cat server will grade this automatically upon submission. Every violation of the *Program Formatting* standard described in Lab 1 will result in a subtraction of a small number of points (usually two points). Looking at your submission report on the Web-Cat server, you will be able to see a notation for each violation that describes the nature of the problem and the number of subtracted points.

Design/Readability: 35 points

This element will be assessed by a grader (typically sometime after the lab deadline). Any *errors* in your program will be noted in the code stored on the Web-Cat server, and two points will be deducted for each. Possible errors include:

- Non-descriptive or inappropriate project- or method-level documentation
- Missing or inappropriate inline documentation
- Inappropriate choice of variable or method names
- Inefficient implementation of an algorithm
- Incorrect implementation of an algorithm
- Incomplete coverage of your Unit Tests. We expect that your unit tests will test all lines of your code

If you do not submit compiled javadoc for your lab, 5 points will be deducted from this part of your score.

Note that the grader may also give *warnings* or other feedback. Although no points will be deducted, the issues should be addressed in future submissions(where points may be deducted).