Lab Exercise 2
CS 2334

September 4, 2015

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 result that are 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

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 as individuals. 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. Import the existing lab2 implementation into your eclipse workspace.
   (a) Download the lab2 implementation:
   `http://www.cs.ou.edu/~fagg/classes/cs2334/labs/lab2/lab2.zip`
   (b) In Eclipse, select *File/Import*
   (c) Select *General/Existing projects into workspace*. Click *Next*
   (d) Select *Select archive file*. Browse to the lab2.zip file. Click *Finish*
2. Carefully examine the code for the *Fruit* and *FruitBasket* classes.
3. Import JUnit into the project (depending on configuration, this may already be done for you for this lab)
   (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*

Unit Tests

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

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

public class FruitTest {

  /**
  * Test full constructor and the getters
  */
  @Test
  public void test1() {
    // Use full constructor
    Fruit f1 = new Fruit("Apple", 0.2, 0.5);

    // The getters must return the correct values
    Assert.assertEquals(f1.getPrice(), 0.5, 0.00001);
    Assert.assertEquals(f1.getWeight(), 0.2, 0.00001);
    Assert.assertTrue(f1.getName().equals("Apple"));
  }

  /**
  * Test name-only constructor and the getters
  */
  @Test
  public void test2() {
    // Name-only constructor
    Fruit f1 = new Fruit("Orange");

    // Set price and weight properties
    f1.setPrice(1.2);
    f1.setWeight(42.0);

    // The getters must return the correct values
    Assert.assertEquals(f1.getPrice(), 1.2, 0.00001);
    Assert.assertEquals(f1.getWeight(), 42.0, 0.00001);
    Assert.assertTrue(f1.getName().equals("Orange"));
  }
}
```

A unit test file is a class in its own right, containing one or more methods (often named *test1*, *test2*, etc.). Each of these methods is 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 *test1()* in *FruitTest*, a fruit object is created with the name "Apple" and a weight and price of 0.2 and 0.5, respectively. This test method confirms that each of these three properties is set correctly during the construction of the object. For example:

```
Assert.assertEquals(f1.getPrice(), 0.5, 0.00001);
```

queries the object's price through the price getter method and compares it to the expected value of 0.5 (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 0.00001 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(f1.getName().equals("Orange"));
```

states the the name must be exactly equal to "Orange" (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.

Within Eclipse, you can execute a unit test by first selecting the java file for the unit test and then clicking the *Run* button. 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).

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.

FruitBasket Unit Test

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

1. Create a new JUnit test class:
   (a) In the package explorer, right-click on *FruitBasket.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 *FruitBasketTest*
   (f) Class under test should be *FruitBasket*
   (g) Click *Finish*. This will create and open a new class called *FruitBasketTest*
2. Write a set of tests that confirm that all methods of the *FruitBasket* class perform correctly. Note that in these tests, you will need to create at least one FruitBasket object and populate it with a number of Fruit 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 FruitBasket 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, Fruit and FruitBasket 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

- All required components (source code and compiled documentation) are due at 11:59pm on Friday, September 4th.

- Prepare your submission file:

    1. Select the project in the *Package Explorer* window.
    2. Right-click. Select *Export*
    3. Select *General/Archive File*
    4. Expand your project and verify that both the *src* and *doc* folders are selected
    5. Enter the archive file name: *lab2.zip* (note that you may want to browse to a different destination folder)
    6. Select *Save in zip format*
    7. Click *Finish*

- Submit your zip file to the lab2 folder on D2L.

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:

Implementation: 45 points

Program formatting: 10 points

(10) The program is properly formatted (including indentation, curly brace and semicolon locations).

(5) There is one problem with program formatting.

(0) The program is not properly formatted.

Unit Tests: 15 points

(15) A complete set of unit tests has been implemented.

(10) One key unit test is missing.

(5) Two key unit tests are missing.

(0) Three or more key unit tests are missing.

Bug Fixes: 20 points

(20) All of the methods with bugs have been fixed.

(15) One bug has not been fixed.

(10) Two bugs have not been fixed.

(5) Three bugs have not been fixed.

(0) Four or more bugs have not been fixed.

Proper Execution: 30 points

Output: 15 points

(15) The program passes all unit tests (these are unit tests that we provide).

(10) The program fails one test.

(5) The program fails two tests.

(0) The program fails three or more tests.

Execution: 15 points

(15) The program executes with no errors.

(8) The program executes, but there is one minor error.

(0) The program does not execute.

Documentation and Submission: 25 points

Project Documentation: 5 points

(5) The java file contains all of the required documentation elements at the top of the file.

(3) The java file is missing one of the required documentation elements.

(2) The java file is missing two of the required documentation elements.

(0) The java file is missing more than two of the required documentation elements.

Inline Documentation: 10 points

(10) Every method contains appropriate inline documentation.

(7) There is one missing or incorrect line of inline documentation.

(3) There are two missing or incorrect lines of inline documentation.

(0) There are more than two missing or incorrect lines of inline documentation.

Submission: 10 points

(10) The correct zip file name is used and has the correct contents.

(5) The correct zip file name is used, but one required component is missing.

(0) An incorrect zip file name is used or more than one required component is missing.