

Lab 2: Unit Testing

CS 2334

August 31, 2017

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 newly modified 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. Import the initial lab 2 implementation into your Eclipse workspace:
 - (a) Download the lab 2 implementation:
`http://www.cs.ou.edu/~fagg/classes/cs2334/labs/lab2/lab2.zip`
 - (b) *File* menu: *Import*
 - (c) Select *General/Existing Projects into Workspace* and then click *Next*
 - (d) *Select archive file*: browse to the lab2.zip file
 - (e) Click *Finish*
 - (f) Once you create the new project, it may not initially know where to find the standard Java libraries (it varies depending on your configuration).

- i. *Project* menu: Select *properties*
 - ii. Select *Java Build Path / Libraries*
 - iii. If the *JRE System Library* is not listed, then select *Add library: JRE System Library* and click *Next*. Select *Workspace default*. Click *OK*
 - iv. If the *JUnit Library* is not listed, then select *Add library: JUnit* and click *Next*. Select *JUnit 4*. Click *Finish*
 - v. Click *Apply* and then *OK*
4. Carefully examine the code for the *Gadget*, *GadgetTest* and *UtilityBelt* classes. The implementations of the *Gadget* and *GadgetTest* classes are correct and complete. However, the *UtilityBelt* class 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 *GadgetTest* class as an example. Here is a piece of this class:

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

/**
 * Testing class for Gadget object
 *
 * @author Monique Shotande
 * @version 08/31/17
 */
public class GadgetTest
{
    /**
     * Test the empty Gadget constructor
     */
    @Test
    public void testEmptyGadgetConstructor()
    {
        Gadget gadget = new Gadget();

        // Check that the name is null, and the price and weight are zero
        Assert.assertNull("Empty constructor: NULL name", gadget.getName());
        Assert.assertEquals("Empty constructor: no weight",
            0, gadget.getWeight(), 0.00001);
        Assert.assertEquals("Empty constructor: no price",
            0, gadget.getPrice(), 0.00001);
    }

    /**
     * Test the Gadget constructor with only a String parameter
     */
    @Test
    public void testStringParameterConstructor()
    {
        String testGadgetName = "Gappling gun";
        Gadget gadget = new Gadget(testGadgetName);

        // Check that the name should match the initial parameter,
        // and the price and weight should be zero
        Assert.assertEquals("String Parameter Constructor: name",
            testGadgetName, gadget.getName());
        Assert.assertEquals("String Parameter Constructor: no weight",
            0, gadget.getWeight(), 0.00001);
        Assert.assertEquals("String Parameter Constructor: no price",
            0, gadget.getPrice(), 0.00001);
    }
}
```

```

/**
 * Test the Gadget constructor using a string parameter and a double parameter
 * for the name and weight, respectively
 */
@Test
public void testDoubleParameterConstructor()
{
    String testGadgetName = "Lock pick";
    double testGadgetWeight = .57;
    Gadget gadget = new Gadget(testGadgetName, testGadgetWeight);

    // Check that the name and weight should match the initial
    // parameters and that the price should be zero
    Assert.assertEquals("Double parameter constructor: name",
        testGadgetName, gadget.getName());
    Assert.assertEquals("Double parameter constructor: weight",
        testGadgetWeight, gadget.getWeight(), 0.00001);
    Assert.assertEquals("Double parameter constructor: no price",
        0, gadget.getPrice(), 0.00001);
}

/**
 * Test the complete constructor and the getters
 */
@Test
public void testFullConstructor()
{
    String testGadgetName = "Gas pellets";
    double testGadgetWeight = 2.05;
    double testGadgetPrice = 12.53;
    Gadget gadget = new Gadget(testGadgetName, testGadgetWeight,
        testGadgetPrice);

    // Check that the name, weight, and price should match the
    // initial parameters
    Assert.assertEquals("Full constructor: name", testGadgetName,
        gadget.getName());
    Assert.assertEquals("Full constructor: weight",
        testGadgetWeight, gadget.getWeight(), 0.00001);
    Assert.assertEquals("Full constructor: price",
        testGadgetPrice, gadget.getPrice(), 0.00001);
}
:
:
}

```

A unit test class is named to indicate which class it is responsible for testing (e.g., *GadgetTest* will test the *Gadget* class). A unit test class contains one or more methods. These methods are often named to indicate which parts of a class that the method tests (e.g., *allMutatorsTest* and *singleParameterConstructorTest*). Each test method is preceded by the *@Test* tag, to inform the compiler to configure the method as one of the tests to be executed. This tag must be placed between the method-level documentation and the method prototype.

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.

The *fullConstructorTest()* test method in *GadgetTest.java*, creates a *Gadget* object with the name “Gas pellets”, weight 2.05 oz, and price 12.53 dollars. This test confirms that each of these properties are set correctly during the construction of the object instance. For example:

```
String testGadgetName = "Gas pellets";
double testGadgetWeight = 2.05;
double testGadgetPrice = 12.53;
Gadget gadget = new Gadget(testGadgetName, testGadgetWeight,
    testGadgetPrice);
:
:
Assert.assertEquals("Full constructor: weight",
    testGadgetWeight, gadget.getWeight(), 0.00001);
```


queries the gadget object’s weight property via the weight getter method and then compares it to the expected value of 2.05 (expected since this is the value that was used in the call to the constructor). The first parameter to *assertEquals* (the String) is a general description of the test that is used to report the specific tests that have failed. The second parameter is the expected value, and the third parameter is the test value. Remember that it is not appropriate to simply test the equality of two doubles, since two double values can be arbitrarily close (or close enough) 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 (specified by the fourth parameter). If this is the case, then this assertion will pass. On the other hand, if the returned weight is different than the expected value, then the test will fail.

The *assertEquals()* method works for ints, doubles and Strings.

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

```
String testGadgetName = "Rebreather";
Assert.assertTrue(gadget.getName().equals(testGadgetName));
```

states the the name must be exactly equal to “Rebreather” (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 documentation code of the *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 have 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 has 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 (in fact, your grade will depend on this). An exception to this is that some code will only execute when failures occur. In some cases, we might choose to forego tests on these cases, but it is rare for us to decide this.

When you are writing tests, you should not rely on your implementation to produce the expected values. Instead, you should work out by hand (or some independent implementation) 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.

UtilityBelt Unit Test

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

1. Create a new JUnit test class:
 - (a) In the package explorer, right-click on *UtilityBelt.java*
 - (b) Select *New/JUnit Test Case*

- (c) Select *New JUnit 4 test*
 - (d) The source folder should be *lab2/src* (the default package)
 - (e) The name should be *UtilityBeltTest*
 - (f) The class under test should be *UtilityBelt*
 - (g) Click *Finish*. This will create and open a new class called *UtilityBeltTest*
2. Write a set of tests that confirm that all methods of the *UtilityBelt* class perform correctly. Note that in each of these tests, you will need to create at least one *UtilityBelt* object and populate it with a number of *Gadget* objects with 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 *UtilityBelt* 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*, *Gadget*, *UtilityBelt* and *Test* classes
 - Select *Private* visibility
 - Use the default destination folder (this should be inside of your Eclipse workspace, in *lab2/doc*)
 - 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 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 and the *UtilityBelt* class is covered completely by your *UtilityBeltTest* class, then you are ready to attempt a submission. Here are the details:

- All required components (source code and compiled documentation) are due 48 hours from the end of your lab section (on Saturday, September 2). **Submission must be done through the Web-Cat server.**
- Use the same submission process as you used in lab 1. You must submit your implementation to the *Lab 2: Unit Testing* area on the Web-Cat server.

References

- The API of the *Assert* class can be found at:
<http://junit.sourceforge.net/javadoc/org/junit/Assert.html>
- JUnit tutorials:
<https://www.tutorialspoint.com/junit/index.htm>
<https://dzone.com/articles/junit-tutorial-beginners> (setup in Eclipse)

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 a product of the fraction of **our tests** that your code passes and the fraction of **your code** that is covered by *your tests*. In other words, your submission must perform well on both metrics in order to receive a reasonable grade.

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 (up to 10 points)
- Missing or inappropriate inline documentation (2 points per violation; up to 10 points)
- Inappropriate choice of variable or method names (2 points per violation; up to 10 points)
- Inefficient implementation of an algorithm (minor errors: 2 points each; up to 10 points)
- Incorrect implementation of an algorithm (minor errors: 2 points each; up to 10 points)

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).