Lab Exercise 4
CS 2334

February 7, 2018

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. What is correct is determined by our specification. 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 a mimic of the *JUnit* tool to produce and evaluate a set of tests. We have provided a specification and full implementation of the previous labs. Your task is to write a set of tests for each class to ensure that its execution is correct.

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 unit tests to verify code correctness

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. Download lab 4 from Zylabs.
2. Copy files into your eclipse workspace.
3. Carefully examine the code for the classes. Note the specification for the code detailed in the previous lab writeups and in the code commentary.

Unit Tests

Within the lab4 files, we have included the *AnimalTest* class (partially implemented) as an example:

```
/**
 * Testing class for Product object
 *
 * @author Stephen Thung, references previous code from Dr.Fagg, Taner Davis
 * @version 2018-08-25
 */
public class AnimalTest
{

    /**
     * Test the empty Animal constructor and the getters
     */
    public void emptyConstructorTest() throws AssertException
    {
        // Use the default constructor
        Animal animal = new Animal();

        // The name should be "noname", color "unknown", height and weight 0.
        Assert.assertEquals("unknown", animal.getColor());
        Assert.assertEquals("noname", animal.getName());
        Assert.assertEquals(0, animal.getHeight(), 0.01);
        Assert.assertEquals(0, animal.getWeight(), 0.01);
    }

    // TODO: test full constructor, getters, and toString
}
```

A unit test file is a class in its own right, containing one or more methods (often named in a convention similar to *emptyConstructorTest*, *animalToStringTest*, etc.). For the Zylabs, you will use a custom-made framework for your unit tests. It mimics the well-established Junit framework (the methods are named the same and act similarly).

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. These assertions are used to determine if the code is acting correctly. The purpose of these assertions is to make sure that the code's behaviour lines up

with the specification. For example, if the function "AddFive()" should add 5 to a private variable *num*, an assertion might be to check that the value of *num* has increased by 5 after calling the function.

In *fullConstructorTest()* in *AnimalTest.java*, an Animal object is created with the name "Tony", a color "Orange", and a weight (in lbs) and height (in inches) of 200.1 and 20, respectively. This test method confirms that each of these three properties is set correctly during the construction of the object. For example:

```
Assert.assertEquals(200.1, animal.getWeight(), 0.01);
```

queries the object's weight through the weight getter method and compares it to the expected value of 200.1 (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(animal.getName().equals("Tony"));
```

states the the name must be exactly equal to "Tony" (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 run your tests with our .java class. Make sure that your tests are named "AnimalTest", "EquipmentTest", and "ZooTest". Running this program will report to you whether or not your tests work.

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. That is, you should understand from the documentation what the function should do and write your tests to this specification; you should prioritize this over what your function actually does. This way, your test is independent of your implementation.

E.g. you have a toString method in a class. You should create a string that is the expected output and compare it to what you get from the toString call. A good test might look like:

```
@Test
public void Test()
{
    Foo f = new Foo();
    String expected = "I am a foo!";
    String actual = f.toString();

    Assert.equals(expected, actual);
}
```

Whereas a bad test might look like:

```
@Test
public void Test()
{
    Foo f = new Foo();
    expected = f.toString();

    Assert.equals(expected, f.toString());
}
```

The second test does nothing of use. It sets the actual value to the output of toString, but also sets the expected value to the output of toString. This is like asserting A == A. Clearly, this should be true. Instead, we want to provide an expected value - "I am a foo!" in this case. As a programmer, you will define what the input/steps for a test should be and what the corresponding output should be. In this case, There is no input to the Foo constructor or the toString method, but the steps are:

1. Create the Foo object
2. call toString on the Foo object to get the actual result

Given these steps and the specification, we decide that the result should be "I am a foo!" - the expected value. When we run the code, we use the test to compare this expected value to the actual value returned by toString.

Additional Unit Tests

Your task for this lab is to write a set of unit tests for the methods in the **Zoo, Animal, and Equipment** classes. Here is the procedure:

1. Fill in code at the TODOs of an existing JUnit test class OR
2. Create a new test class (for classes without an existing test class)
3. Write a set of tests that confirm that all methods of the *Animal and Equipment* classes perform correctly. Note that in these tests, you will need to create some objects. For example, in the Zoo class you will need to create at least one Zoo object and populate it with a number of Animal objects of various names, colors, weights and heights. 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).
4. For the Zoo class you only need to make tests for the getTotalHeight, getCapacity, and toString methods.

equals() Method

You will finish the implementation of the equals() method for the *Animal and Equipment* classes. Java does not inherently know how to compare two objects, but this can be defined with the equals() method. For example: intuitively, two animals should be equal if their name, color, weight, and height are the same, but Java does not do this automatically. It must be defined. One could also define that two Animals are equal if their name and color are the same, regardless of the weight.

Submission Instructions

You will submit to Zylabs. Before submission, finish testing your program by editing and executing your tests. You should submit when your program behaves as you expect it to.

Grading Rubric

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

Zylabs Submission: 40 points

The Zylabs grader 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 Zylabs server will inform you as to which tests your code passed/failed. You may submit up to 15 times to pass more tests and improve your score.

Grader Scoring: 40 points

A TA grader will assess your unit tests. Unit tests should be meaningful (i.e. they actually run the code with different values) and thorough (i.e. multiple conditions are checked).

Github: 10 points

You will need to use github when developing your project. The grader will check that you have made several commits (¿= 5) while developing your lab. Commits should have good messages. You will link your github repo on canvas.

Lab Plan: 10 points

Your todo.txt file will be checked by the zylabs unit test to verify its format. A grader will also asses the quality of your lab plan. Your plan should have at least 5 objectives that should be relatively granular (e.g. an objective of "finish lab" is much too large and not a good tasks in a plan). The grader will ensure that your predicted and actual times are reasonable (e.g. not 20000 minutes).