Jlaja Pandora Podier

Robert Bathmann

COSC 603: Software Testing and Maintenance

Spring 2016

Project 3 Unit Testing with JUnit

**Task 2 Getting Started – Fibonacci**

The Fibonacci JUnit test case failed on the assertion equality test case for the zero (0) test case in which a zero (0) was expected but the value returned was one (1). There was an error found in the Fibonacci class in the switch statement code. The JUnit Test code is as follows: *assertEquals*("0", 0, fibonacci.fibonacci(0)); but the switch statement code for calculating and returning the nth Fibonacci number code for the zero case had a return of one (1) as oppose to returning zero in which that threw an error because it was expecting a return value of zero but the code had it returning one (1). In order to correct the error, the switch statement had to be corrected for the zero (0) case to return a value of zero (0) as opposed to returning a value of one (1).

**public** **int** fibonacci(**int** n) {

**switch** (n) {

//case 0: return 1; //This case statement is in error

**case** 0: **return** 0;

**case** 1: **return** 1;

**default**: **return** (fibonacci(n - 1) + fibonacci(n - 2));

}

}

}

**Task 3 – A Little More Advanced – Rectangle**

When the RectangleTest JUnit test case was executed, there was an error that occurred in the software with the GetArea and GetDiagonal in which the value returned was not the expected value for the calculation for the area or the diagonal. After investigating the source code, we determined that the error was in the Point class in which the instantiation of the new point set the values for both x and y coordinates to the ‘y’ coordinate. The statement that had to be changed was the **this**.x = y to **this**.x = x because the ‘x’ coordinate had to be set for ‘x’ and the y coordinate for y.

Point(Double x, Double y) {

//this.x = y; //This is the error in the code because it sets the x value to the y coordinate

**this**.x = x;

**this**.y = y;

}

}

The methods to get the area and the diagonal as follows:

/\*\*

\* Gets the area.

\*

\* **@return** the area

\*/

**public** Double getArea() {

**return** Math.*abs*((p2.x - p1.x) \* (p2.y - p1.y));

}

/\*\*

\* Gets the diagonal.

\*

\* **@return** the diagonal

\*/

**public** Double getDiagonal() {

**return** Math.*sqrt*(Math.*pow*((p2.x - p1.x), 2) + Math.*pow*((p2.y - p1.y), 2));

}

In the JUnit Test case two rectangle points are being passed in the test case. In rectangle 1 (rect1), two points passed are ((2, 2) (4, 7)) and in rectangle 2, (rec2) ((2, 6) and (4, 3)). When these values are passed through the getArea and getDiagonal methods the actual output would be as follows:

**getArea**

rect1: ((4-2)\*(7-2)) = 10

rect2: ((4-2) \* (3-6) = -6 but you have to take the absolute value, so abs(-6) = 6

**getDiagonal**

rect1: sqrt( (4-2)2 \* + (7-2)2) = sqrt (4+25) = sqrt(29)=5.385164807 ~ 5.3852

rect2: sqrt( (4-2) 2 + (3-6) 2)=sqrt (4+9) = sqrt (13) = 3.605551275 ~ 3.6056

Once the x and y coordinates were properly set, the expected results for the getArea and getDiagonal were actually returned as anticipated and the JUnit test case successfully passed with no errors.

**Task 4 – On Your Own – A Vending Machine**

We were unable to find any syntactical or logical bugs in the code. However, there is room for improvement to better model how a vending machine works. The bugs.txt file contains the improvements to the Vending Machine source code that could be made. Utilizing the JUnit allowed us to determine ways to improve the source code and make it more robust.

**Task5 – Summing it All Up**

* **A description (2-3 paragraphs) of what you learned from this project (particularly Task 4)**

Developing JUnit test cases can be cumbersome as you have to think what you want to test against, in which you have to develop cases that would test for expected results versus actual results to ensure the expected results are what is returned by the application. In this project, we learned that as you develop unit test cases using JUnit, it can assist with identifying improvements to be made in the source code to make the application more robust while testing for bugs in the source code.

While developing the unit test cases, we also learned to create test cases for each possible branch in the code. Each time an if statement was encountered, we added unit tests for each condition. Since this application is very small, this was easily achieved and did exponentially increase the number of test cases.

Additionally, we learned the syntax for testing thrown exceptions from the source code. It does not follow the typical assert pattern that is primarily used for JUnit testing. The syntax used in the exception testing allowed us to verify that our source code throws a specific exception for the conditions set for the test case. For example, we tested for the condition if price was a negative value as the price could not be negative. If a negative value was returned, an exception was to be thrown.

* **A description (2-3 paragraphs) of what you liked and didn’t like about JUnit’s support for unit testing**

There are some limitations in what JUnit can and cannot do with regards to unit testing based upon the type of test cases it executes. JUnit does not check each method developed in a class because JUnit testing only test for what the developer develops the test case for it to test against. The preconditions are set for the unit tests, therefore the JUnit test cases does not test for preconditions that were not anticipated. When we ran the JUnit test cases and it successfully passed and there were no errors found, we made the assumption that everything was written correctly and there were no issues in the source code. However, there could potentially be improvement made to the code.

Although, JUnit can be utilized to test individual components of the source code, it is most helpful when used to test the application as a whole rather than just testing the individual components. To ensure that the application ran as expected the individual components had to work together so generating a JUnit test case that tests the entire application was beneficial for identifying bugs.

In this assignment we were required to write unit tests for trivial getter methods. This created a minor problem because the getter cannot be independently tested – instead the constructor or a setter method must first be invoked. Since we were also required to test a constructor that set a value, it was not possible to write tests for both that wouldn't be the same. This is why the VendingMachineItem constructor test only tests for the price exception. Setting the value is tested in the two getter tests. Reflection can be used to create tests that are more independent, but that also tightly couples the test to the implementation of the class and not its public interface. If JUnit tests could be included within the class itself, it might help to solve the problem (but can create others).

Finally, we did not feel the support for exception testing in JUnit was adequate. Using the recommended approach, we are only able to test the type of exception thrown and not any of its other properties such as message. We could have created a try-catch block inside the unit test and write an assert in the catch that tests for some other condition in the exception, but that is quite cumbersome. Junit simplified the syntax for testing exceptions, but did so at the cost of promoting potentially inadequate.