# Project #3 – Unit Testing with JUnit

## Task 2 – Getting Started – Fibonacci

When I ran the junit test, the test obviously failed. The message was “AssertionError: 0 expected: <0>, but was 1”, which means when 0 was used for the input, the function returned 1, when it should have returned 0. The second line of the Failure trace shows which line of code failed in the test case. I fixed this error in the Fibonacci method by changing the return for case 0: from 1 to 0 at line 17. Re-running the JUnit test revealed no errors.

## Task 3 – A Little More Advanced – Rectangle

The JUnit test revealed there was an error with the getArea() method. The expected output was 10, however the actual output was 25. Looking at the test coordinates (2,2) & (4,7) means that the x-axis width should be 2 and the y axis height should be 5 (x2-x1) and (y2-y1), which is why we use 10 for the asserts equal test. A result of 25 means that a length and width of both 5 are being calculated in the area method. This is because the Y axis height is being set to the x axis width of 5. This led me to discover the bug in the Point class, where the x field was being Instantiated as the y field because of a typo. I corrected by setting the x field to the x parameter.

I made the code a little more readable by moving the calculation of the length and width of the rectangle from the getArea() method to the constructor, so that a length and width field are set for all methods to use.

/\*\*

\* The Class Rectangle.

\*/

**public** **class** Rectangle {

/\*\* The p2. \*/

**private** Point p1, p2;

/\*\* The x axis length. \*/

**private** **double** xAxisLength;

/\*\* The y axis width. \*/

**private** **double** yAxisWidth;

/\*\*

\* Instantiates a new rectangle, setting the length and width

\*

\* **@param** p1 the p1

\* **@param** p2 the p2

\*/

Rectangle(Point p1, Point p2) {

**this**.p1 = p1;

**this**.p2 = p2;

xAxisLength = Math.*abs*(p2.x - p1.x);

yAxisWidth = Math.*abs*(p2.y - p1.y);

}

/\*\*

\* Gets the area.

\*

\* **@return** the area

\*/

**public** Double getArea() {

**return** xAxisLength \* yAxisWidth;

}

/\*\*

\* Gets the diagonal.

\*

\* **@return** the diagonal

\*/

**public** Double getDiagonal() {

**double** valueToBeSquared = Math.*pow*((xAxisLength), 2) + Math.*pow*((yAxisWidth), 2);

**return** Math.*sqrt*(valueToBeSquared);

}

}

## Task 4 – On Your Own – A vending Machine

I did not find any bugs in the VendingMachine Class. I thought that the methods dealt with bad inputs well. A GUI that prevents users from picking options that are out of bounds would prevent most bad inputs from reaching the Vending Machine object. I did find an issue with the VendingMachineItem Class. I thought that VendingMachineItem constructor should also make sure that all items have names in addition to checking that the price is not set to a negative number. I modified the code to catch when no name was set. I added a “No name assigned” exception message, so that the user creating the items will see the error and correct it.

## Task 5 – Summing it All Up.

I learned or rather reinforced that finding bugs is a tricky endeavor. The first two projects, Fibonacci and Rectangles, showed me how a simple typo can mess up the correctness of a program. The IDE identifies most syntax errors, which is great, however careless typing or copy and pasting can let through simple errors, like setting a field to the wrong value. It was also easier to find the bugs in the first two projects because the program operations failed using normal input. The vending machine project was more difficult to identify bugs, because there were no errors with normal input and most bad inputs had catch statements. I also found myself having to reinforce my domain knowledge to help me identify why something wasn’t working (refreshing my knowledge of Coordinate Geometry).

I also found it very time consuming to make sure I had 100% test coverage for the vending machine project. The documentation greatly helped with creating test cases. The preconditions and post conditions comments were particularly useful for coming up with good and bad inputs. In addition to the documentation, the simplicity of the methods also helped in coming up with test cases for the Vending Machine project. It wasn’t difficult to decipher what was going on. Unsurprisingly, maintainable code is easier to test.

I really liked JUnit’s support for testing. Before I knew about JUnit, I would use print statements to check state and debug issues. The built in methods were very helpful for testing methods that returned values. For void methods, I had to rely on the class’ get methods to check that the fields being utilized were being modified correctly.

I didn’t find anything that I really disliked about JUnit’s support. I noticed JUnit also helps people write cleaner more maintainable code. It is much easier to test simple methods than convoluted ones. This definitely helps prevent the design smell of long methods. If each method only does one thing, then the automated creation of the unit test class really helps speed up developing tests.