**Task 2**

In this task, I verified the unit tests given for a simple project. I found one discrepancy: I could not find the method “teardown” which was supposed to be annotated with the “@After” attribute. I learned that methods can be marked with “@Before” or “@After” attributes in order to run those methods before and after each unit test. This helps with any initial set up as well as any cleanup required for running the unit tests. The bug with the code was a simple one: the case statement in the Fibonacci class’s Fibonacci method returned 1 when the value was 0. I changed this method to return 0 when the value is 0.

**Task 3**

In this task, I found the error to be within the class Point, which set its x variable equal to y instead of x. This error surfaced when testing the method getArea() for the Rectangle class even though the error was in the Point class. In addition to this, the encapsulation for both the classes was incomplete. I fixed the code by first correcting Point’s constructor method. In order to improve the code, I refactored the classes Point and Rectangle so that they would use get methods to access inner variables. Finally, I also wrote tests for the constructor methods of Point and Rectangle. These tests ensure that the correct variables are assigned in the constructor method. With the new tests, the user should be able to easily point out that the initial error was with the Point class, and not within the Rectangle class. Given below are the corrected version of the getArea() and getDiagonal() methods which use the get methods for accessing x and y.

/\*\*

\* Gets the area.

\*

\* **@return** the area

\*/

**public** Double getArea() {

**return** Math.*abs*((point2.getX() - point1.getX()) \* (point2.getY() - point1.getY()));

}

/\*\*

\* Gets the diagonal.

\*

\* **@return** the diagonal

\*/

**public** Double getDiagonal() {

**return** Math.*sqrt*(Math.*pow*((point2.getX() - point1.getX()), 2) + Math.*pow*((point2.getY() - point1.getY()), 2));

}

**Task 5**

**What I learned**

I learned a lot about how to implement unit tests for a code-base via this project. I learned that the attributes @Before and @After can be used to run certain methods before and after each unit test. This allows developers to only write initialization and clean up code only once. This is very important since many of the unit tests rely on the same global variables. It becomes extremely important to clear out all global variables after a test and make sure that values from a previous test do not interfere with the next test.

Attributes such as @BeforeClass and @AfterClass can be used for initialization and clean up tasks which only need to be run once (instead of being run before and after each test). This allows any initialization and clean-up which maybe be used by multiple tests, but are not changed by the tests (such as database connections, etc.). The @Test attribute lets the compiler know that the method is a unit test. It can also be overloaded with an input variable called expected, which lets the compiler know that the method should throw and exception. This allows developers to test cases where invalid input should result in a specific type of exception.

Unit tests are important in order to verify the functionality of a code-base. They are also important as they allow quick testing after any refactoring or code changes. This allows developers to catch bugs as they maintain code and make changes, instead of finding them after all the changes are made. However, unit tests can take a long time to implement and maintain. Sometimes, changes in functionality of the code or refactoring may require changing some of the unit tests as well as writing new unit tests. It is also difficult to write unit tests which cover all possible scenarios.

**JUnit support for unit testing**

JUnit makes writing unit tests extremely simple. The attributes mentioned above (@Test, @Before, @After, @BeforeClass and @AfterClass) make writing unit tests extremely simple. It also allows for quick reusability of code. Assert statements are also very intuitive and decrease the amount of code required to write unit tests. This makes writing and maintaining unit tests much easier. It also makes unit tests more readable.

After running unit tests, JUnit shows whether or not all tests passed. JUnit makes running tests easy by providing UI options in Eclipse. It also allows the user to create Test Suites where the user can pick and choose the test classes they want to test. This allows the user to test all test cases at once instead of testing each test class one at a time. The ability to quickly test all unit tests allows developers to be test frequently and catch errors earlier.

JUnit also shows a list of all the tests which failed after a test run. This allows developers to quickly find the exact test which failed and the line where it failed. This makes for a great starting point to debug the error. By knowing exactly what went wrong, the amount of time spent debugging the error and looking for the root cause is significantly reduced. JUnit should provide a metric for coverage which would allow developers to know what proportion of their code base has unit tests written for it and which parts do not have and unit tests. This would allow developers to write more Agile code and make sure that most if not all of their code has some unit tests validating against it.