Summary and Reflection Report

As I have mentioned various times, testing code is an essential task that must be done in order to ensure that the code is sound and secure. In this course I have done various testing to ensure that my code worked as intended. When I was setting up my JUnit testing, I had the requirements right next to me as I began coding. Even before I wrote up the tests, I was ensuring that my code met the requirements because nonetheless if the requirements are not met the project, is a failure.

When I first started the milestones, I was very confused as to what was needed and what was required. Once I started the second milestone, I had a better idea of what needed to be done and how I can achieve the requirements. For example, when I started to code the contact side of the project, I had no idea how to make the name not be null and not be longer than 10 characters. It wasn’t until I started on the Task class and started to write up the JUnit tests that I realized that before I set the objects to their respective variables, I can create those If statements so that it can be checked before it gets registered into the system. Understanding the requirements is one thing but understanding how the requirements can be met is another very important aspect. When writing the JUnit tests understanding coverage is how I made sure my testing was effective. In eclipse we can run our program as a JUnit test and when we run it as coverage it will show us how much of our code was covered and tested. This is very helpful and effective since it will show us how much of the actual code was ran and tested. If your coverage is not above eighty percent, then we know not all your code has been covered and maybe more tests need to be written up. There were things that I have noticed was even after I achieved coverage above eighty percent some of the code will still come up highlighted red to show that it was not covered. However, that part of the code was embedded with the test itself, so I wonder if the test is not counted in the coverage. Either way, when I wrote up my tests, I made sure to try and cover as many different inputs as possible that can arise. I as the developer know how the code should function and what is required from inputs, but if given to a normal consumer will the do the same? I wrote up most of my tests with that mindset at hand, to compensate for those users that may not understand what is required. I ensured my code was efficient by the coverage use and making sure that every part of my code was tested. However, to make sure that my code was technically sound was just to keep it short and simple. There is no need to over complicate anything especially with code. The less code there is the better. Like I said, there is no real way to test and make sure your code is technically sound, but simplifying things helps not only yourself but others when reading your code.

There are many different techniques used for testing code, and I only used a total of three. The ones that I used were assertTrue, assertFalse, and some assertThrows to test my code. AssertTrue was used for when we wanted to test the code in use will return a Boolean value of true. For example, in my code I wrote “assertTrue(con.getContactID().equals("123456"));.” This would test to see if my contact ID equal what I was looking for, the “123456”. Like assertTrue, assertFalse did the opposite and was used for when we wanted to test the code to see if it would return a Boolean value of false. For both, if the Boolean value came back to the respective test, the test would be successful or fail depending on the Boolean value requested. For the last one, the assertThrow was used for throwing illegal arguments to see if an invalid input will be caught. In my code I used that to test all the different ways the input could come up incorrect. If the value is null, if the value was too long, whatever is chosen if that illegal argument is thrown, the test is a success. If for some reason the test would fail, that would tell me that the input for whatever variable made was not caught and I would have to adjust my code to compensate for that variable. There are many other different types of JUnit tests for a numerous number of different situations. One that I will mention is assertEquals. Just like the other assert tests that I mentioned, this type of test checks whether two objects equal one another. You will put object one and object two separated with a comma in between, and the test will be a successful if both objects are the same. If they are different than the test will come back as a failure. Regardless of what test we use it all is dependent on the application that is being built. Choosing the right type of test is what will allow the tests to be more effective.

Testing out code that you have written really reveals a lot about how one codes. It is easy to overlook simple code because you have done it numerous times. Then when you go to test, and one of those tests fails, you look back and wonder why you even misspelled the line of code, or you forgot one simple thing. It happens to me a good number of times and it is something that becomes more apparent as I continue to test out my own code. One thing that I have noticed myself doing is I mentally test my code as I write it. It is good practice to take frequent breaks and test your code after a few lines instead of waiting until all your code is written. It is to have that mindset of testing your code as its being developed that will mitigate those errors. Of course, it is easy to bias of your own code. Many will think “I’ve done this a line of code many times” and don’t erven bother questioning themselves. I don’t trust myself which is why I always go over every line of code that I write after I write it. That way I minimize the more chances of those biased simple mistakes from making their way through. Having that discipline of triple checking your work even if you have done it a million times, will save you and your code from many avoidable bugs and issues that can arise.

Testing is a very fundamental aspect of software development it should be taken as serious as other aspects of the field. There are many things we do in our daily lives that get quick results. For example, basketball. In basketball we shoot the ball and try to make it in the hope. If we try one method and fail, we know something went wrong and we adjust. With code we don’t have that quick feedback unless we ask ourselves and test the code out to make sure it works. So, testing often and making sure your tests are as efficient as possible are only going to benefit you in the long run.