**Summary**

My JUnit tests were created mostly in the same way. Firstly, the code was created for the requirements of the project. This is the code that is going to be tested. After that, a checklist was created to make sure I wouldn’t miss anything that needed to be tested. Then, I created the code to test the functions of the project, while consulting the checklist to make sure I didn’t miss anything. All of this helped me keep in light with software requirements. For example, this section of code in the Contact class:

Graphical user interface, text, application

Description automatically generated

is the code that set the ID string for an individual contact. This simple function tests whether the ID is null or too long. The JUnit test for this piece of code consists of two blocks:

Graphical user interface, text, application

Description automatically generated

The first block tested the ID being too long, or the second catch in the setContactID function. The next block tested the ID being null, or the first catch in the setContactID function. Most of my JUnit tests were based like this. For each code block in the Contact class, one or more JUnit tests were created to test for each possibility that could be entered. This is how I met the software requirements.

The quality of my tests were, in my opinion, high. This is because of the simplicity of the tests and what they were testing. Most of the JUnit testing blocks are, effectively, one line of code testing a specific outcome or catch for the function in the respective class. So, there is little to go wrong in the testing phase. The examples above show that. I, also, made sure that all the tests that needed to be done, to cover the percentage, were done. This is why the checklist helped.

My code was technically sound and efficient because it was simple, which means there is little to go wrong. For example, the list search, which was one area that was improved. Although the line is a little long, it is still one line of code.

A picture containing text

Description automatically generated

Here, the code is looking for a task with the same taskID it was given. If it finds the corresponding task, then returns the task in the same line of code. This makes the code efficient as well because the compiler has less to press to get the same amount of work done. Comparing this to the previous search that I had created, shown below, it is leagues better.

Text, letter

Description automatically generated

**Reflection**

There were a few different testing techniques used in this project. One main one was the Regression Testing. This is mainly what JUnit is based around, testing different blocks of code before all the blocks are tested together. This is a great way to test things because you’d like to find out if, before the rest of your code depends on it, the block actually works. This helps prevents errors from being costly after discovering them later on in the project.

Other testing techniques I didn’t use were performance testing and security testing. The reason for not using these techniques is that we weren’t testing for performance or security, we were testing for the functionality of the code. However, that doesn’t mean we weren’t thinking about performance or security. The above code shows that, especially for performance wise.

Performance testing is used to make sure that the program doesn’t create memory leaks, use up too many resources, or create unneeded variables. In performance testing, it is ideal to change code if it can do the same job but use less resources. (See example above) This means, if the change gives the program a 0.001% boost by changing just a few lines of code, it would be ideal, depending on the project.

Security testing is even more important, though. With security testing, you want to make sure data is encrypted when it needs to be, that people don’t have access to the program in restricted areas, and that certain inputs or keystrokes don’t break the program, creating unwanted results. This means, just like performance testing, if changing just a few lines of code prevents users from seeing just a restricted letter, just a letter, it would be ideal. The reason being that, with seeing that letter, hackers can try to break into the program, knowing it has at least one vulnerability.

My mindset was different when creating the code and when I was testing it. Knowing I’m the one who solely created the program, I would have to take responsibility when the program breaks. Therefore, when I was testing it, I wanted to make sure every possibility for an error was covered. I can imagine that a bias would be concerning when testing your own code, but for me, it’s of no concern. Imagine that you built software, having created and tested it, knowing there’s issues but because of bias, you released it. Then, multiple companies or people you’ve distributed it to get angry and mad with you because of said errors. Your image is tarnished and destroyed, simply because you didn’t want to fix your code. This is the same way with being disciplined. If there’s errors, they need to be fixed. For me, there is no bias, as I want to maintain a self-image of being able to test and fix my code.