My approach for software testing was heavily reliant on the software requirements. The requirements essentially acted as a guide for me to be able to write effective tests and make sure the coverage was sufficient. As an example, the contact class needed to be able to create a contact with specific attributes like character limits in names or addresses and those had to be tested. The limits I put in for testing were directly written by looking at the requirements. Sometimes I wrote the requirements in a comment above the class as I wrote it for guidance.

My coverage percentage was extremely high, this means that the tests were effective at making sure the code was well tested. The coverage report allows a developer to see how much of the code has been verified to work via a Junit test.

I was extremely proud of my Junit tests. I know that they were technically sound because I had strong coverage, as well as all of my tests met the requirements of the project.

*@DisplayName*("Testing Task Name")

void testTaskName() {

*testTask1*.setTaskName("");

*assertEquals*("NULL",*testTask1*.getTaskName(),"Failed to null");

*testTask1*.setTaskName("abcdefghijklmnopqrstuvwxyz");

*assertEquals*("abcdefghijklmnopqrst",*testTask1*.getTaskName(),"20 Character limit failed.");

This is an excerpt from my TaskTest.class file I had submitted a few weeks back. This code demonstrates that the task name is only supposed to be 20 characters long. If the task name is longer, it will only display the first 20 characters. This test has a good name, a good error message, and it runs properly. This is an example of a great Junit test.

I am also happy with the efficiency of my tests. For the majority of my service tests like TaskService I used one instance of the class Task.class. I find this more efficient than creating a new instance of the Task class for every single test if that can be avoided. For a project this size, I feel the impact would have been small, but when I start doing projects at scale, efficiency becomes critical.

The software testing techniques that I used were mostly straight forward. I tested things based on the requirements and made sure that my programs would set strings to an appropriate length and for the appointment classes, I made sure the date was working properly. I did negative testing as well as positive testing to make sure that my classes worked as intended. I didn’t do any system testing.

Positive testing is great for making sure that the system is performing as expected with proper input. If my Task Class has a 20 character limit for taskName, It clearly shouldn’t be getting task names longer than 20 characters. It is good practice to make sure that it could take a task name such as (“Go for a walk”) since that name is within 20 characters.

Negative testing on the other hand is making sure that the class can take input that the class undesired input and won’t freak out. Setting the task name to something like null or to a string that is longer than 20 characters is helpful because it will test the classes ability to function correctly under improper conditions.

This course gave me a new appreciation for software testing that I had not had before. Software testing was only something I did when a program wasn’t working for me. Now when I write software, I make a point to include tests as an important part of the programming process. Testing makes me more cautious as a developer and makes me appreciate functioning code a little more. I went from writing basic if statements in the main code to test basic issues all the way to writing quality Junit tests. I am extremely proud of the work I have done in this class.

It is certainly important to limit bias when writing tests. There were times when my tests were going haywire because they were running out of order because I hadn’t imported the correct libraries and I desperately wanted to just create new instances of each class and make the tests less efficient to finish the assignment. I chose to figure out my issue and I feel as though I became a better developer for it.

As a developer, I have a responsibility to write quality code that is as bug free as I can make it. It is imperative that I as a developer take testing seriously and make an effort to have strong code and strong tests. Eventually, I will have a career where my code will impact people's workflow and potentially people’s lives and it is a good practice to start getting comfortable with the idea of testing now.