Unit testing can be a valuable resource to ensure that code has been written error free. By implementing JUnit tests, we can streamline the testing process and make sure that any changes made to the code still allow the system to meet requirements. However, JUnit tests are only useful if they are written to check as many lines of code as possible. This can be checked by running coverage tests on any systems written. The closer the coverage test gets to 100% coverage, the more sound the testing is considered. Lastly, it is important to have efficiently written code so when the JUnit tests are run, we aren’t testing the same lines of code repeatedly.

For this project, it was important to make sure that if invalid inputs were made, they were dealt with properly. This could be entering a name that is too long or any input that is a null value. When writing my tests, I also made sure to test that each requirement was met in all possible ways. For example, in my contact tests, I made sure to check that the length requirements for the variables (first name, last name, phone number, etc.) were met both when the variable was initially setup, and if the variable was changed. This was something I didn’t put in my first test classes, but in future iterations of my test file, I made sure this was accounted for. Both are examples of making sure that the software met the requirements that the client needed them to meet.

Another important aspect of the testing I did was to make sure that the tests were effective. The main way I made sure that the tests I wrote were effective was by checking the coverages of each of the tests. In all but one of my test classes, my coverage was at 100 percent. The appointment class was the only exception to this 100% coverage. The appointment class only had 99% coverage, because I checked to make sure that the date entered by the user matched the date that was produced by the appointment object. Only having 99% coverage means that there is a possibility that the section of code that wasn’t checked could produce an error. This is something that, given more time, I would have tried to see if I could make the system fail in that specific spot to better protect the software.

To be technically sound with our unit tests, we need to follow certain rules and normal practices. This includes naming my JUnit tests by capitalizing each word in the name and each individual test name is written in camel case. This can be seen in all my JUnit files for this project.

Lastly, we need to be efficient. This is something I wish I would have done a little better. I made sure that all the coverages were at 100% but found myself inadvertently testing some things multiple times. For example, in my appointment test, I make sure that everything works properly when creating an appointment object, but I also inadvertently test that it is created properly when I test the appointment service class as well. In stead of testing the perfect case in the appointment test, I could just test this when creating an appointment service object.

When working on each of the milestones in the previous modules, I struggled to find a pattern that worked best. I first tried writing all the source code at the beginning and then writing all the test classes but didn’t think that it was as efficient as it could have been. After that, I tried writing the class and then the test that went with that class, but it didn’t work the best either. Finally, I tried a hybrid of the two where I wrote some of the source code then worked on the coinciding test case for that section of code, but this didn’t seem to be as efficient as needed either. In the end, I think that the most efficient pattern was the last.

In these milestones, I found that we used a combination of white-box and black-box testing. I used white-box testing to make sure that every branch of my source code was properly tested. This was done specifically when I had errors that were thrown or when I had if/then statements in my code. On the other hand, we used black-box testing to make sure that the client’s requirements are all met. These tests were typically done first to make sure that the strings (ID Strings or description strings) were not too long and that they were valid entries (not null).

Though these are a lot of testing techniques, there are many more that I didn’t end up using for my milestones. Another testing technique is using control flow graphs. I didn’t end up doing this in my milestone, but it could help for systems that are more complex than simple classes like we made in our milestones. Though I used static testing slightly in these milestones, I don’t think I used it enough in these projects.

All the above techniques are useful in more applications than the one we were working with in this course. For example, black-box testing can be done to make sure that client requirements are properly met. This can be a simple as making sure that a person of certain age gets their senior discount or as complex as making sure that banks are calculating interest properly. We can also use white-box testing to make sure that all errors are properly thrown and caught. Using control flow graphs can help make sure that all branches of the system are checked for errors. Static testing is very important to the testing process as we can catch errors before we run tests on the code. This can help make sure that you have your code written properly and make these appropriate changes before writing all the accompanying test code.

Lastly, we need to consider our own personal biases in testing. Our own biases can creep in, and we can end up creating tests and requirements that the client didn’t want. You may think that a certain requirement needs to be met, but if the client hasn’t told you that requirement needs to be there you could cause further issues. Maybe a new test that you create limits the range that a user can input in, but the client wanted the user to be able to input data outside the new range you have made. Therefore, it is important to practice discipline with our internal biases when developing systems.

In this term we learned a lot about testing and how it can be useful for creating stronger, more accurate code. JUnit tests are a powerful tool that can be used to ensure that code is written accurately but can’t be the only tool to ensure that code is written accurately. No matter which testing techniques need to be used, the lessons that I learned in this term will greatly help me in my future career choices.