The first part of the project what I did was write the code while testing it as I made it. It seemed to work well and only had a few issues that I had to fix later on. I tried to align it with the requirements by making each section of the code one at a time and checking it with what is wanted and needed in the program. I also realized later after submitting that I did not test everything that I should have, I forgot to test the code with too long imputes and making sure that they reacted the way they were supposed to. I went back and added more tests to the contact system in the code to test for too long of names and such as well now.

The second section, for creating tasks, I did much of the same, but I split up the code into more functions and methods for it to be easier to change and read in the future as well. I wrote the code for this section much the same as the last, where I tried to write the system in tandem with the tests. I believe this helps in not forgetting something because you have an idea of what you have to test for next being written in front of you. I believe this was the easiest part of the system to write as it had the fewest impute variables to verify and test.

The last section, for creating appointments, was also much of the same but I felt much more confident in writing it because it seemed similar to the previous section in the system being written. I wrote the tests for the code as I wrote it and this time also made it so that the ID can be made but the user, if not it is randomized. I felt that went along with the requirements a bit more as it seems in the last description that they want the ability to enter their own ID into the appointment.

While I wrote the code, I tried to check that it was technically sound by routinely checking for errors in the code as well as outputting some information to the console to

check for correctness as Junit tests for if the test passes but doesn’t give you any information, really, beyond that. I wasn’t too sure how to check for efficiency in the code other than making sure that it ran correctly and checking how long the test takes to complete, as most of them take under 0.04 seconds.

As for the testing techniques that I employed while working on this project, there are a few that I did, and many that I didn’t do. One of the main testing techniques that I used was whitebox testing, or unit testing. In this I tested that the code responds the way it is intended to specific inputs and instances. This was good for making sure that the code itself was sound. This along with static testing, where I just reviewed the code for easy to spot mistakes. I also ended up doing some manual testing where I input the data and received a response on the console as to what it did.

Each of these techniques has its own advantages and disadvantages. Unit testing is good for making sure that a specific part or section of the code or system is sound and reacts the way it is supposed to, but it doesn’t tell you if there are logic errors or the like. It could tell you that it worked when in reality the output was incorrect. Static testing is a good way to combat that in that it helps spot logic errors or simple typing errors in the code, but it takes time and patience to comb through all the code to check it visually. Then the last technique, manual testing, is good to check for both problems with the code and logic errors, but it is time consuming and prone to human error when making manual tests.

There are some other testing techniques that I did not get to in the project. For example, integration testing is not something I did, as it seemed that each part of the system ran independently from one another. There is also regression testing, as the system was made in a short amount of time there was no need to go back and make sure that it still worked the way it was supposed to.

While working on this project the mindset I tended to adopt was usually along the lines of, how can this program go wrong? Trying to think of different ways to input a variable for it to accept or reject as acceptable and that it was what was expected. Understanding the complexity of the system is also important because if you don’t know how everything interacts with one another then it would be much more difficult to test if they are interacting with each other correctly, or where the problem is in the code if there is one. This ended up happening to me, in that it was good I knew what parts interacted with what because at one point I copied part of the code from the other system as it was very similar, but I forgot to change one of the variables. Because I knew how everything worked with each other, when the test failed, I was able to quickly find the issue because I knew where to look.

When testing your own code, bias is definitely an issue that must be addressed. Many people, including myself, tend to have a sense of pride in their work, especially if they worked hard on it or it was something new, they were trying. So, when there is something wrong with it, it can sometimes be hard to not take it personally, so when testing your own code, you have to be careful not to overlook issues or problems just because you think it is fine.

When testing and writing code it is important to work hard and not cut corners because it can cause many problems for yourself and others later on. If something goes wrong or something needs to be changed, and it is found that corners were cut or there are problems with the code that where not addressed, then it can be much more difficult, if not impossible to fix depending on the problems created. This is important not just for yourself but for everyone that might also work on the system later as it could put everyone behind.