For each of the three features, I took each of the requirements and tried to implement them as best as I could using the specific code as needed. Such as creating else if statements for new appointments, this way they could create new appointments as well as store them along with existing appointments. Hash maps were created for contacts to store new and existing contact information. This way it made it cleaner and easier to store. Then I created a string for tasks to store needed information there as well.

To check my JUnit tests, I used assert methods in order to check to make sure that the code was running properly. An assert method tests the expected output of the code that I had created. My JUnit tests show the expected output that I wanted, such as showing new appointments or showing an error if there was duplications or errors. This showed me that it was working efficiently and effectively. It also helped to throw an error if a name was misspelled, or wrong information was entered.

To ensure that the code was technically sound I did the same kind of checking and test that I did for effectiveness. I used assert methods and throws and does not throw. These allowed me to check and test for different errors and to make sure it does not throw an exception. For example, I added in the code to throw an error if I added a contact that already existed or added nothing, the does not throw runs the code and checks to see that it does not throw an exception. This is an example of how I tested a throw : assertThrows(IllegalArgumentException.class, () -> { Appointment a = new Appointment(null, null, null); }); (AppointmentTest.Java line 23)

To ensure that my code was efficient I used HashMap's, strings and else if statements. This is an example of the HashMap I used: private HashMap<String, Appointment> appointments; (AppointmentService.Java line 4)

These check the code efficiently going line by line of the code and implementation. By using a throw statement, it helps clean up the code by throwing exceptions, in other words if there is invalid data the throw statement throws an exception and keeps the code clean.

When it came to software testing, I used a static technique, instead of a dynamic technique. By using the static technique, I pretty much ran the code and tested as I went along. This was more efficient and was much more hands on for me and helped me to learn as I went. Especially since it was just me working on the project. With a dynamic technique that would have required me to have more planning and more anticipation of certain problems that I might foresee. Such as having charts and a meeting to anticipate any issues that might come up. Considering that it was a smaller project and just myself I felt that static technique was better because it was faster. As a static technique is just swiftly hands on. If it were a larger project with more people working on it, then I would see the need for a dynamic technique and strategy.

When it came to the mindset that I had when working on this I took the cautious approach of being suspect as to whether the code would work or not. I kept with the mindset of the code may not do as it should, and I might have to fix something. This is an important mindset because arrogance can lead to assumptions and little room for error or improvement. Whereas siding along with caution allows for improvements or changes as needed. It also helps when testing as I can go task after task and run the code to ensure that it is functional. This is shown in the code that I wrote that had else if statements, the else if statements basically work on the side of caution by saying in case this thing happens than this other thing happens. As I created them with the thought that perhaps someone might enter someone with the same information, or a misspelled name or not have all the information needed to access a client’s information.

I tried to avoid making assumptions regarding what the code would do. Bias would be a huge concern because it would lead to an assumption that the code would work as it should without feeling the need to test or check for bugs. An example of this would be assuming that my code was flawless and did as it should without testing, and it ended up loaded with bugs that enabled the code to not run as it should. Then it would make my work even more difficult because if I hadn’t tested, especially line by line, if I had a bug I would have no choice but to start from the beginning and go from there. This would impact the time of which the work is expected.

It is vital to being disciplined in software quality as well as not cutting corners when writing or testing code. Lazy and or sloppy code makes it harder for others to work with you, and not testing or not testing as you should make debugging code even harder because you would have to go over line by line to find out where something went wrong which adds more time to a project that could be spent elsewhere. To avoid technical debt such as backed up untested code I plan to keep testing and checking code as I go. To be efficient but to also be cautious and keep checking and running the code. To also try to anticipate that things might go wrong, things I might not see coming. Which is why I will also make sure that someone else does look over my work to make sure it is working properly.