**Summary**

First, when developing the Contact and Contact Service classes, I ensured that they both met the functional requirements. When running the JUnit tests for these classes, since it was early in the course, my JUnit tests were functional, but did not meet an efficient coverage percentage, meaning that certain instances or code paths were not taken care of. Since this was the first course I had taken that covered JUnit testing, I did not know the importance of the coverage percentages. For the Task and Task Service classes, I had the same approach but came to the realization when doing the journal reflection that I did not have the right coverage percentages yet. In our resources, I read through a piece that had covered percentages but I had already submitted my assignment for the week. After coming to this realization, I decided to go back through and edit my code to fit the requirements and boost my coverage percentages to at least 80%. To make sure my code was technically sound, I made sure to implement functions that met the correct requirements. For example, a requirement for the AppointmentService class was to be able to add or delete appointments, so implementing

**public void addAppointment(Appointment appointment)**… as well as

**public void deleteAppointment(string appointmentID)**… was necessary for functionality.

To ensure that my code was efficient, I used data structures such as the array list for Appointments, checked for existing appointments, etc. This helps optimize the code so it operates efficiently. EX:

**if (existingAppointment.getAppointmentID().equals(appointment.getAppointmentID())) {**

**throw new IllegalArgumentException(“Appointment ID already exists.”);**

**Reflection**

The software testing techniques I used in this project were positive and negative testing. In using positive and negative testing techniques, positive tests will verify that it is functioning properly and negative tests verify that the function fails. I did not use stress testing in this project to analyze the stability of high loads.

For practical uses and implications, Positive testing helps ensure that the positive outcomes are achieved when a software functions as intended. An example for this could be the shopping cart in an online shopping website. Making sure that the correct amount is charged when a purchase is made would be positive testing. Negative testing consists of testing the systems response when invalid inputs are provided. An example of this would be the if a website requires a 10 character minimum for setting a password, inputting a 9 character password and seeing the outcome. For Stress testing, this involves testing the system’s performance when it is under a high load such as checking for bottlenecks. Understanding this helps us understand the system’s scalability and limitations. An example of this would be accounting software that stores data like inbound and outbound transactions, payroll etc.

The mindset I adopted when working on this project focused on the complexity of the project. I was cautious when it came to testing to ensure I was meeting the requirements. Appreciating the complexity of the code being tested helps ensure I met the specific requirements needed so the software worked as it should. For example, understanding that the system had a requirement to be able to add appointments was important because a it not only needed to have that functionality, but it also needed the exceptions for when an appointmentID being added already existed. This was crucial for ensuring that duplicate appoinments were not added.

Biases in code review can effect the overall outcome, so when reviewing my code, I attempted to review it as if it was someone elses code to be as objective as possible. I would imagine that if the developers were also the testers, this would cause issues for the future. Many developers most likely have a lot of pride in their work, so if they were to test their own work, they could oversee some issues.

Cutting corners in software engineering is something that helps no one. Initially it may seem that it could cut down on time but in the long run it can lead to many issues that could take up more time than if it was done right the first time. Whether you are writing the code or testing it, both duties are crucial for the overall functionality of the software, so if the discipline is lacking, issues are bound to occur. To avoid technical debt as a practitioner, it would be beneficial to breakdown code into smaller bits so that it is more maintainable and scailable. For example, in game development, specifically live ops games, aspects of it change very frequently as more content is released, bugs are reported, etc. Ensuring that the code is maintainable helps ensure that these changes can be managed effectively.