My unit testing approach aligned with the requirements by following the required methods in the base code. To ensure the input met the length requirements and was not null I checked each section that would take input. For this specific example I’ll be referencing Appointment.java the appointment ID could not be longer than 10, the description could not exceed 30, and the date could not be in the past. My approach verified the inputs were within these constraints and were not null by testing various inputs that were designed to verify the correct input. “assertThrows(IllegalArgumentException.class, () -> appointment.SetDescription(null)); and assertThrows(IllegalArgumentException.class, () -> appointment.SetDescription(longDescription));” This tests the description to throw an illegal argument flag when the input is too long and the second portion throws if there is no input given. I tested the functionality was working to add, update, or delete appointments as well as verifying no two IDs matched. The same logic used for the appointment portion was used for Contact and Task.

Overall, I feel the quality was effective in these tests as the coverage percentage came out to 90.5% for the project. The quality of coverage has gradually improved with each milestone as the Contact file was originally 87.6%, Task was 90.2%, and Appointment was 93.1%. To ensure the code was technically sound I used data structures like an array to store the information for contacts, tasks, and appointments. “public static ArrayList<Appointment>appointments = new ArrayList<>();” I tried to use better coding practices such as implementing “equalsIgnoreCase” to avoid mistakes or duplicates from capitalization. I also made use of assertions such as “assertEquals”, “assertThrows”, and “assertTrue” as one example, to test the date in AppointmentTest.java “assertNotNull(appointment.GetDate());” was something I learned about to check the date was not null.

Originally, my code was not what you would call efficient at all, I had to work on it and make revisions to achieve clear efficient code. In Contact and Task milestones I was adding the input to test on each line, by the time I began the Appointment milestone I moved on to storing the information in variables to reuse and improve readability in the code. This can be seen from the original version of Contact and the later improvement of Appointment in this example “Contact test1 = new Contact("1111", "John", "Doe", "1111111111", "123 Any Street");” to “Appointment test1 = new Appointment(ID, futureDate, description);” after previously initializing the variables with their given input, both do the same thing but make the code more efficient with better readability.

Some of the software techniques that were implemented in this project are in black box and white box categories. Black box testing is specification based, used to find missing functions, initialization errors, and performance errors. I used this to test for valid and invalid inputs, test conditions and actions, and to test outputs. White box testing is structure based, it is used at unit, integration, and system levels of the testing process. This was used in coverage tests to check components, used to break down tests into sections for testing. Static and dynamic testing were used throughout this project. Static testing is used to find defects in the code before running the program and can help find errors in both syntax and logic by analyzing the code only. This works with error guessing as well, looking over the code to predict the expected outcome and whether that section will work as intended. Dynamic testing is when we can test the program by executing the code to check for errors. The focus of the testing on this project was for functional testing to verify functionality of the software as well as unit testing for each individual unit of the software and checking internal data structures.

Software testing techniques that were not utilized in this project include non-functional testing techniques, testing the usability, compatibility, and performance. Users were not part of the testing process for this project to check for usability. I was not able to test for compatibility across multiple systems nor performance as the tests conducted were for relatively low workloads. If this project were to eventually be released testing phases later would include load testing, stress testing, endurance testing, and spike testing to check the boundaries of the system.

Black box testing is typically used when code has clear functionality for outsourced testing. White box techniques are utilized when a specific outcome is thoroughly understood. Static and dynamic testing are used through the development process, before and after executing any code. The non-functional techniques, testing the usability, compatibility, and performance are for the phases where we check for user experience with the interface, across multiple platforms available to different users, and the performance of the system with varying workloads.

Through the development of this project my mindset shifted to a more analytical side. I had to experiment a lot with what worked and what did not, in the end I spent more time on these parts of the project than on any previous coding assignment. More than ever before, I had to make certain I had a full understanding of every portion of code to get the desired result. I was pushed to focus more on the complexity and interrelationships between the code with lots of trial and error to produce a finished product that met all requirements with adequate coverage. The tests need to correspond to each method for checking their functionality and I used caution when creating the tests so they would test exactly what they needed to.

I do feel that bias plays a role when it comes to anything you are creating yourself. It’s easy to become “too close” to a project you’re working on, sometimes just coming back to it the next day with a fresh set of eyes can help see more than you could when actively working on it. Tests could be skipped if you feel you are certain your code is doing what it is intended to do, but it’s good to remove yourself emotionally from the project to reduce any bias. For instance, when I was working on DeleteTask in TaskService.java within the if statement. I had set it to return false when a task was removed. I realized it was setting it to false no matter what and would show the task was not deleted even after it was removed. Simple mistakes can be overlooked when writing the code and I may have felt biased towards this section of code and confident it worked as intended resulting in skipping the test where it did not produce a full expected result.

It is important to be disciplined in the commitment to quality as a software engineering professional. Software engineers have a code of ethics when it comes to the development of software which could harm individuals and companies if you make the wrong mistakes. In previous classes, we have learned the importance of producing code that is in the best interest of the public. Cutting corners can lead to an unsafe or defective product and to avoid technical debt we should implement agile practices of testing code early and often, always pushing for high-quality code. Software quality safeguards end-users from using inferior software and cutting corners can compromise security or overall function of the program.

Works Cited

*Software engineering code - ACM ethics*. ACM Ethics - The Official Site of the Association for Computing Machinery's Committee on Professional Ethics. (2022, June 8). Retrieved June 17, 2022, from https://ethics.acm.org/code-of-ethics/software-engineering-code/

\*, N. (2020, February 11). *Testing techniques : Types, advantages and disadvantages*. ElProCus. Retrieved June 17, 2022, from https://www.elprocus.com/what-are-testing-techniques-types-advantages-disadvantages/

*Software testing techniques*. GeeksforGeeks. (2021, March 1). Retrieved June 17, 2022, from https://www.geeksforgeeks.org/software-testing-techniques/