CS 320 Project 2

Brandon Ellis

For my unit testing approach, I focused on testing the main features of three services: Contact, Task, and Appointment management. For **Contact management**, I wrote tests for the methods that add, delete, update, and retrieve contacts. These tests checked whether the system handled things like duplicate contact IDs, missing information, and invalid phone numbers. For **Task management**, I tested methods that add, delete, update, and retrieve tasks. These tests ensured that tasks could be properly managed and that the system would handle errors like deleting a non-existent task. In **Appointment management**, I focused on testing the methods for adding and removing appointments, making sure the system correctly handled invalid appointment IDs, dates in the past, or descriptions that were too long.

My testing approach matched the software requirements. The services required unique IDs for contacts, tasks, and appointments, and my tests made sure these IDs were checked for duplicates. I also tested that all the fields (like names, phone numbers, and descriptions) were validated according to the rules, such as length limits and specific formats. I made sure to test both the normal cases and edge cases, such as invalid or missing data, to fully cover the required features.

The quality of my tests was strong because they covered all the key features of the services and included both normal and exceptional cases. I used a code coverage tool to track how much of the code my tests covered, achieving about 90% coverage for each service. For example, my tests for adding contacts checked both the correct functionality and error handling, like rejecting duplicate contact IDs. This thorough testing helped ensure the code would work correctly and handle unexpected situations.

Writing the tests was an ongoing process. I started by testing the most common use cases and then added tests for edge cases, such as trying to delete a task that doesn’t exist. I made sure the tests were technically sound by verifying that the system handled all input validation and correctly threw errors when necessary. The tests were also efficient; I focused on checking the most important aspects of each method without repeating unnecessary checks.

For testing techniques, I mainly used **boundary testing**, **negative testing**, and **equivalence partitioning**. **Boundary testing** ensured that input values stayed within valid limits, such as making sure contact names were no longer than 10 characters and phone numbers were exactly 10 digits. **Negative testing** focused on making sure the system rejected invalid inputs, like duplicate IDs or incorrectly formatted data. **Equivalence partitioning** allowed me to test a smaller set of representative values instead of every possible input, which helped reduce redundancy in the tests. Techniques I didn’t use included **mutation testing**, which could have helped evaluate the quality of my tests by introducing small changes to the code, and **state-based testing**, which wasn’t necessary for this project since the services were simple.

Each of these techniques has its uses. **Boundary testing** is helpful when the system has strict input limits, such as when entering data in forms. **Negative testing** is essential for handling unexpected user inputs, making sure the system reacts properly to invalid data. **Equivalence partitioning** is useful when you have a lot of possible inputs, as it allows you to test a smaller, more manageable set of inputs instead of testing every possibility.

In terms of mindset, I was careful when working on the project. I made sure to understand the complexity of the code and how different parts of the system interacted. This helped me avoid missing errors, like when testing the deleteContact method to ensure it correctly handled cases where a contact didn’t exist. I also made sure to limit bias in my testing by using automated tests and getting peer feedback. It’s easy to miss problems in your own code, so getting a fresh perspective helped ensure the tests were accurate.

Being disciplined about code quality is important because cutting corners can lead to problems later on. If I didn’t test thoroughly or ignored input validation, the system could break or be harder to maintain. By writing good tests and validating inputs, I made sure the system was both reliable and easy to update. Moving forward, I plan to avoid technical debt by writing clean, maintainable code, running thorough tests, and following good practices like continuous integration. This will help me build software that is sustainable and less likely to cause problems down the road.