Randy Phimmasen

Professor Parul Hirpara

Southern New Hampshire University

October 20th, 2024

CS 320 Project Two

When developing the contact, task, and appointment services, I approached unit testing by focusing on key features and edge cases for each service. For the contact service, I tested the main functions of adding, updating, and deleting contacts, ensuring proper handling of both valid operations and invalid inputs. Similarly, the task service was tested for creating tasks, updating statuses, and setting deadlines. The appointment service required tests for time scheduling and preventing conflicts between appointments. Each test aimed to reflect how users would actually use the features, balancing efficiency and thoroughness.

My testing approach was aligned with the project’s software requirements. For example, one requirement specified that users must set deadlines when creating tasks. My tests verified that the system rejected tasks without deadlines, ensuring proper validation. Another requirement was to prevent overlapping appointments, which I addressed by writing tests that ensured the system detected scheduling conflicts and prevented double bookings. By frequently referencing the requirements, I ensured that the tests covered both expected and edge cases as specified.

I am confident in the quality of my JUnit tests because they provided high coverage, with the majority of the code paths tested. This included edge cases like ensuring tasks couldn't be created without descriptions or appointments couldn't be scheduled in the past. Coverage analysis confirmed that the tests thoroughly examined the system's functionality and identified potential issues early. Writing these tests allowed me to ensure correctness and performance by focusing on how the system handled both positive and negative cases.

While writing the tests, I made sure to consistently review the logic to keep the code technically sound. For example, I tested appointment conflicts by ensuring that the system prevented overlapping time slots, using specific methods to check for errors when necessary. I also focused on keeping the tests efficient by reducing redundant setup code, which made the tests cleaner and easier to maintain.

During the project, I used unit testing and mocking. Unit testing helped ensure that each part of the system worked independently, while mocking simulated external systems like databases. I didn’t use integration or system testing in this project. Integration testing would have checked how different services worked together, while system testing would have tested the entire system from the user’s perspective. Both of these techniques would be more useful in later stages when verifying the overall system.

In terms of mindset, I was cautious to ensure thorough testing, especially with complex features like scheduling, where small mistakes could cause major issues. It was important to understand how different parts of the system affected each other, such as how appointment conflicts could affect user experience. I tried to limit bias by testing failure scenarios like invalid inputs rather than assuming my code would always work correctly. This helped ensure my tests were objective and comprehensive.

Quality and thoroughness in testing are important for avoiding issues down the line. Such as failing to properly test appointment conflicts could have caused problems in real-world usage. Moving forward, I plan to maintain a strong commitment to quality by continuing to write comprehensive tests and using continuous testing throughout the development process. This approach will help catch issues early and reduce the risk of technical debt.