Maridelle Anne Gonzales

CS320

December 10, 2024

My approach to unit testing for the Appointment, Task, and Contact features was focused on validating all specified constraints and edge cases, closely aligning with the software requirements. For the Appointment feature, I created tests to ensure that the appointment ID length was correct, the appointment date was not in the past, and the description adhered to the allowed character limits. I also tested invalid scenarios, such as an appointment ID being too long or the date being in the past. For the Task feature, I validated that updates were only valid if the description was under 50 characters, and I ensured that attempts to update immutable fields like the task ID would raise an error. Lastly, for the Contact feature, I tested that the first name could not be null, the phone number was within the expected length, and duplicate contact IDs were not allowed. This approach ensured that each method handled valid inputs correctly and raised the appropriate errors when invalid data was provided, in line with the outlined requirements.

The quality of my JUnit tests can be defended by the fact that they covered the core functionality of the system, including typical use cases and edge cases, resulting in an 87% coverage percentage. This high coverage indicates that the majority of important functionality was thoroughly tested. The tests not only validated correct inputs but also handled edge cases, ensuring that any unexpected behavior would be caught. For example, in the Appointment tests, I used assertions like assertThrows to confirm that invalid inputs, such as a past appointment date, triggered the expected exceptions.

Writing the JUnit tests was an insightful experience as it allowed me to ensure the software met the specified requirements and handled various edge cases. I approached the task by first understanding the requirements and then writing tests for each constraint. During the process, I made sure to write clear and maintainable tests by using helper methods to reduce redundancy. For example, I created a method to generate valid appointments, so I didn't have to repeat the same setup code in every test.

To ensure that my code was technically sound, I made sure to validate constraints in the service methods and used assertions in my tests to verify that errors were thrown when needed. For example, in the Appointment service, I validated the appointment ID length and the appointment date, ensuring that the constraints were enforced from the beginning. To ensure efficiency, I reduced repetition in my tests by creating helper methods for common test setup. This made the tests more maintainable and easier to update when needed, without repeating code across multiple test cases.

For the milestones, I primarily used unit testing, edge case testing, exception testing, and functional testing. Unit testing allowed me to verify that individual components, like methods and classes, worked as expected in isolation. For example, I used JUnit to ensure that constraints such as appointmentId length and appointmentDate validity were enforced. Edge case testing was another key technique, as it focused on validating inputs at the boundaries of acceptable ranges, such as checking if descriptions exceeding 50 characters or null values triggered appropriate exceptions. Exception testing ensured that the application handled invalid inputs correctly, like throwing an IllegalArgumentException with accurate error messages when constraints were violated. Functional testing helped confirm that the core features met the specified requirements, such as adding, updating, and retrieving tasks or contacts.

There were other testing techniques that I didn’t employ during the milestones, but they have valuable applications. For instance, integration testing would have tested interactions between multiple components or services to ensure they worked together seamlessly. Similarly, system testing could validate the entire system as a whole, testing workflows and end-to-end scenarios. Each testing technique serves different purposes and can be applied depending on the project’s needs. Unit testing is essential for modular applications, where verifying the functionality of individual components can save debugging time. Edge case and exception testing are especially useful for input-sensitive systems, ensuring reliability and a smoother user experience. Integration testing becomes crucial in projects with multiple interacting modules, while system testing is ideal for validating end-user workflows. By strategically selecting testing techniques, I can ensure quality and reliability for any software development project.

My mindset as a software tester was focused on caution, ensuring I rigorously tested both valid inputs and edge cases. It was important to understand the complexity and interrelationships of the code, as one change could impact multiple areas. For example, when testing the Appointment feature, I validated edge cases like past dates and invalid appointment IDs, understanding how these could cause errors in appointment scheduling. This holistic approach helped identify issues early and ensured the system worked as intended.

To limit bias, I approached testing without assumptions, focusing on requirements and logic rather than familiarity with the code. For instance, I tested that appointment IDs followed length constraints, even though it seemed straightforward. If I were testing my own code, bias could be a concern, as I might overlook subtle flaws. For example, I could assume that a phone number validation in the Contact feature was correct without testing edge cases like invalid characters, which is why maintaining objectivity is critical in testing. Maintaining a disciplined approach to quality is essential for creating reliable software. Cutting corners in code or testing leads to technical debt, which accumulates and makes future development harder. For example, in the Task feature, testing description limits ensures that invalid inputs don't cause issues. To avoid technical debt, I plan to write comprehensive tests, refactor code regularly, and use continuous integration to maintain clean, scalable software. This commitment to quality ensures the longevity and stability of the project.