Sarah Wagner

12/15/24

Project Two Reflection

For the contact, task, and appointment services, I focused on unit testing to make sure that the functionality of the code matched the provided requirements. I broke down each feature into small, testable components and developed JUnit tests for every specific requirement. For example, the contact service required a unique contactID that could not be null, exceed 10 characters, or be modified after creation. To make sure of this, I created test cases that validated these constraints. One test checked that an IllegalArgumentException was thrown when attempting to create a contact with an invalid or null ID, and another verified that an attempt to modify the contactID would fail. Similarly, in the task service, tests confirmed that tasks could be added, deleted, and updated, with each update scenario ensuring only the allowed fields were modified. For the appointment service, I tested constraints such as ensuring the appointmentDate was not in the past and that the description field did not exceed 50 characters. Each test case directly targeted a specific requirement, ensuring that my approach was closely aligned with the software specifications.

I checked that each path in the code was tested, achieving 100% coverage for all features. For instance, in the task service, I tested invalid scenarios like attempting to add a task with a duplicate ID, and edge cases such as trying to delete a task that did not exist. This comprehensive approach ensured that all requirements were met and that unexpected inputs were handled gracefully. Tools like code coverage reports validated that all branches and methods were tested, which gave me confidence in the reliability of my tests. The effectiveness of these tests was further demonstrated during development when they caught several potential issues early, such as failing to validate task descriptions properly.

To make sure my code was technically sound, I incorporated validation logic within the constructors and methods and tested these thoroughly. For example, in the appointment service, the appointmentDate field was validated to ensure it could not be null or in the past. I tested this with a specific case where a date in the past was provided, and the program correctly threw an IllegalArgumentException. This validation ensured that no invalid dates could be set, keeping the code robust. Efficiency was another priority. I avoided redundancy by combining related scenarios into single tests when appropriate. For instance, in the task service, I tested both the successful addition of a task and the failure of adding another task with the same ID in a single test. This approach reduced redundancy in my test cases while maintaining thorough coverage. Overall, writing JUnit tests not only ensured the functionality of my code but also improved my understanding of how to anticipate and handle potential issues.

The primary technique I employed for this project was unit testing, which focuses on testing individual components of the application in isolation. Unit testing allowed me to verify that each feature worked correctly according to the requirements. This technique is particularly useful during early development because it identifies bugs in specific components before they are integrated into a larger system. Other techniques, such as integration testing, system testing, and acceptance testing, were not used but are worth considering for larger projects. Integration testing ensures that different components work together seamlessly. For instance, if my services had interacted with an external database or API, integration tests would have been critical to verify that data flowed correctly between components. System testing validates the entire application, simulating real-world use cases. While I did not use system testing here because the services were isolated, this technique would be valuable for applications with user interfaces or complex workflows. Acceptance testing focuses on confirming that the application meets the end user's requirements. While my unit tests indirectly ensured this by verifying requirements, formal acceptance testing would involve stakeholder feedback to confirm satisfaction with the final product. Each technique has its place depending on the project. Unit testing is ideal for early-stage development or when working on specific, isolated features, as I did for this project. Integration testing becomes important when combining multiple components, such as services, databases, or APIs. System testing is critical for validating the functionality of the entire application, particularly in production-like environments. Acceptance testing ensures that the final product aligns with the customer's expectations, which is essential for customer-facing applications.

Throughout this project, I approached testing with a cautious and thorough mindset. I recognized the complexity and interdependencies of the code and took care to anticipate edge cases. For example, in the appointment service, I considered scenarios where users might input past dates, null descriptions, or overly long descriptions. Testing these cases helped me prevent errors that could have been overlooked. To limit bias while testing my own code, I relied on the requirements rather than my assumptions. Instead of assuming that a feature worked as intended, I wrote tests to explicitly confirm it. For instance, I tested each updatable field in the task service individually, ensuring that I did not overlook any potential issues. Bias can be a concern when testing your own work because familiarity with the code might lead to assumptions that everything is functioning correctly. To counteract this, I adopted the mindset of trying to "break" the application by providing invalid inputs and unexpected scenarios.

Maintaining discipline and commitment to quality is crucial as a software engineer. Cutting corners during development or testing might save time initially but can lead to long-term issues, such as bugs or poor performance. For example, skipping validation for the appointmentDate field in the appointment service could have allowed invalid dates, leading to unreliable functionality. To avoid technical debt, I plan to adhere to best practices, such as thorough testing, regular code reviews, and documenting code clearly. These steps will ensure that my code remains maintainable and reliable in the long term.

This project provided valuable experience in applying and reflecting on different software testing techniques. By focusing on unit testing, I ensured that the contact, task, and appointment services were robust, efficient, and aligned with their requirements. While other testing techniques were not used in this project, I now have a better understanding of their importance and when to apply them in future projects.