Jeffrey Flores

August 18, 2024

Summary and Reflections Report: Testing Approach for Project One

For Project One, which involved the development of a mobile application with contact, task, and appointment services, I employed a structured and methodical approach to unit testing. Each feature was thoroughly tested to ensure it met the defined software requirements. I created unit tests to validate the creation, updating, deletion, and retrieval of contacts. Each test case focused on ensuring that the service properly handled valid and invalid input scenarios. The tests were directly aligned with the requirement that each contact must have a unique ID, name, and phone number. For instance, a test was written to ensure that the system prevents the creation of contacts with duplicate IDs.

The tests for the task service were designed to ensure that tasks could be added, updated, and removed as required. The tests also covered scenarios where invalid data (e.g., overly long descriptions) might be provided. The tests validated that tasks adhered to the constraints of having a unique ID, a name, and a description. They ensured that tasks could not be updated or created with invalid data. For the appointment service, the tests were focused on ensuring that appointments could be created, deleted, and retrieved correctly, with particular attention to the date constraints (i.e., appointments cannot be scheduled in the past). The testing approach aligned with the requirements by ensuring appointments with invalid dates were not accepted and that each appointment had a unique ID.

The overall quality of the JUnit tests was demonstrated by the high coverage percentage achieved during testing. Coverage tools were used to ensure that all critical paths were tested. The effectiveness of the tests were observed in their ability to catch edge cases and prevent potential bugs from reaching production. The tests did not only cover the happy paths but also included scenarios where inputs were incorrect, missing, or close to being valid.

Writing the JUnit tests was an enlightening experience, reinforcing the importance of covering a broad range of scenarios, from basic functionality to edge cases. I ensured that the code was technically sound by writing comprehensive test cases, such as:

**Handling Null Inputs:**

@Test

void testAddNullContact() {

ContactService service = new ContactService();

assertThrows(IllegalArgumentException.class, () -> service.addContact(null));

}

The efficiency of the code was ensured by avoiding redundant operations within the test cases. For example, I avoided unnecessary re-instantiation of services within multiple test methods by reusing the same service object where appropriate.

Black-box testing focuses on testing the functionality of the software without knowing its internal implementation. In this project, I utilized this technique extensively by validating that each service met the functional requirements as defined by the client.Black-box testing is ideal for validating user-facing features where the internal codes are less relevant than the outputs generated based on specific inputs.Boundary testing involves testing the boundaries of input values, such as the minimum and maximum lengths of strings. This technique was crucial in testing features like the length constraints on task names and descriptions.Boundary testing is essential for detecting off-by-one errors and ensuring that the software behaves correctly at the extremes of input ranges.White-box testing involves testing the internal structures or workings of an application. This technique often requires a deep understanding of the codebase and is typically used to validate complex algorithms or ensure that all code paths are executed. This was a testing technique that I did not use in my project.White-box testing is particularly useful for validating internal logic, such as loop conditions or recursive algorithms, ensuring that all code branches are tested.

Throughout the project, I adopted a cautious mindset, understanding the potential complexities and interdependencies of the code. This caution was proven in how I approached testing the relationships between different classes (e.g., ensuring that an appointment could not be created if its date was invalid). Appreciating these complexities helped prevent potential bugs from slipping through.As a software developer, I was aware of the potential for bias, particularly when testing my own code. To limit this bias, I approached testing with a critical mindset, intentionally seeking out flaws in my logic rather than assuming correctness. For example, I wrote tests that intentionally passed invalid data to the services to see if they would handle it gracefully.

Being disciplined in my commitment to quality was a key focus throughout this project. I understood the importance of not cutting corners, especially when writing and testing code. Cutting corners could lead to technical debt, which would increase the long-term maintenance cost of the application. To avoid this, I ensured that every piece of functionality was thoroughly tested before considering it complete. For example, I didn’t give in t the temptation to skip testing edge cases, knowing that doing so could result in bugs that are harder to diagnose later.

In the future, I plan to continue this disciplined approach by adhering to rigorous testing standards and regularly reviewing code for potential improvements. This will help to maintain a high level of code quality and minimize technical debt over time.