7-2 Project Two

Euntak Jang

CS 320

Dr. Lewis

**Summary**

**Describe your unit testing approach for each of the three features.**

I used a thorough unit testing strategy to construct the mobile application's three core features: Contact, Task, and Appointment services. The Contact service tests were created to validate the validity of adding, updating, and removing contacts, with extra checks for unique IDs and contact information validation. Similarly, the Task service tests were designed to guarantee that tasks could be created, altered, and removed, with an emphasis on unique task IDs and task attribute validation. Finally, the Appointment service was tested to ensure that appointments could be made, changed, and withdrawn, with special emphasis on unique appointment IDs, verifying appointment dates were not in the past, and validating the description.

**To what extent was your approach aligned to the software requirements?**

My testing strategy was precisely matched with the software requirements, ensuring that all declared criteria were covered. There was a unit test for each criterion, such as unique IDs, field lengths, and field limitations. In the AppointmentTest class, for example, there are tests like testValidAppointmentCreation and testPastAppointmentDate that check the appointment date and guarantee it is not set in the past, precisely corresponding with the specified criteria.

**How do you know your JUnit tests were effective based on the coverage percentage?**

The complete nature of the JUnit tests, which target both positive and negative test scenarios, may be used to assess their overall quality. Effective JUnit tests do more than simply check feature execution; they also anticipate and manage potential failures. Using coverage tools, it was clear that the tests covered a large percentage of the program, with over 80% coverage. This high percentage of coverage guarantees that the majority of the code's branches and lines were run during testing, confirming the code's functionality and resilience against the requirements.

**How did you ensure that your code was technically sound?**

Writing the JUnit tests was both difficult and illuminating. It forced me to think like a user, to anticipate how they could interact with the program and to anticipate potential pitfalls. It wasn't only about confirming the happy route; it was also about ensuring that the program handled erroneous inputs or edge situations graciously. Several boundary checks and input validations were included to ensure the code's technical soundness. For example, in the AppointmentTest, the method testInvalidAppointmentId uses the assertThrows function to ensure that an exception is triggered when an excessively lengthy appointment ID is supplied, assuring the system's resilience against incorrect inputs.

**How did you ensure that your code was efficient?**

By streamlining data structures and ensuring procedures had clear and succinct logic, efficiency was preserved. The AppointmentService, for example, used a HashMap to store appointments, allowing for O(1) time complexity for add, get, and delete operations. The AppointmentServiceTest test function testAddDuplicateAppointment tests the efficiency and validity of this technique by ensuring that duplicate appointments are not created, demonstrating the system's rapid reaction to such instances.

**Reflection**

**What were the software testing techniques that you employed in this project?**

In this project, I mostly used White Box Testing, concentrating on the application's internal architecture. This method allowed for a thorough examination of the code's logic, ensuring that all branches and routes were checked. For instance, I built tests to verify both the successful establishment of appointments and the errors thrown when the parameters were not satisfied. Boundary Testing was another approach used. The system handled edge situations gracefully by focusing on the limits of input domains, such as the maximum length of a string or the validity of a date.

**What are the other software testing techniques that you did not use for this project?**

While Black Box Testing was mentioned, it was not the main approach. Black Box Testing assesses system functioning without taking into account internal architecture. Techniques such as Equivalence Partitioning or Decision Table Testing might have been used to enhance test coverage by classifying input data into equivalent classes. Another approach, Performance Testing, which would test the system's performance under stress, was not used.

**For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.**

* White Box Testing: Designed for comprehensive validation and is frequently utilized throughout the early stages of development. While thorough, it can be time-consuming.
* Boundary testing is essential for systems with stringent input boundaries. It guarantees that both the inner and outside borders are examined.
* Black Box Testing: This type of testing is useful for acceptability and system-level testing. It aids in the detection of missing requirements.
* Performance testing is essential for apps that are anticipated to handle numerous users concurrently or those that require real-time processing.

**Assess the mindset that you adopted working on this project…**

It was critical to adopt the attitude of a careful software tester. "What could go wrong?" was the query that guided every line of code and function. For example, when developing the AppointmentService, it was critical to consider instances in which appointments may be entered with multiple IDs. I assured the system's robustness by understanding the interrelationships, such as how the services interacted with the data structures and the limitations on the data. The testAddDuplicateAppointment test was created as a direct result of this cautious approach, verifying that the system handled such instances graciously.

**Assess the ways you tried to limit bias in your review of the code…**

I frequently took breaks between developing the service classes and the tests to reduce bias. This aided in approaching the tests with a fresh perspective, mimicking a scenario in which a different person produces tests. There is an inherent bias in development, a notion that "my code is correct." This can lead to possible faults being overlooked. For example, while developing the updateAppointmentDate function, the initial assumption was that users would always provide future dates. However, by using an impartial technique, past date validation was introduced, confirming the application's resilience.

**Finally, evaluate the importance of being disciplined in your commitment to quality as a software engineering professional…**

In software engineering, quality is non-negotiable. Cutting shortcuts may provide short-term benefits, but it will cost you in the long run. A hurriedly developed feature may work initially, but it may subsequently have problems that undermine user trust. Skipping validation tests, for example, may speed up development, but if users enter erroneous data, the system may crash, resulting in data loss or corruption. To minimize technological debt, it is critical to follow best practices, refactor often, and provide complete test coverage. Regular code reviews and adherence to design concepts such as SOLID can help to guarantee that the codebase is manageable and scalable.