Reflection Report: JUnit Testing for Mobile Application Services

# Testing Techniques

Throughout the development of ContactService, TaskService, and AppointmentService, I employed JUnit unit testing as the primary software testing technique. Unit testing focuses on verifying that individual components of the program perform as expected in isolation. Using JUnit, I created modular and repeatable test cases for each method, such as adding, updating, and deleting records.

In `ContactServiceTest.java`, I implemented tests such as `testAddContact()` and `testUpdateContact()` to ensure data constraints (e.g., contact ID length and non-null fields) were strictly enforced. These test cases used assertions like `assertEquals` and `assertThrows` to check whether methods correctly accepted valid inputs and rejected invalid ones. I also included tests for boundary values and illegal inputs, ensuring robust coverage of all branches of logic.

I tested edge cases, including null values, excessively long strings, and invalid ID formats. These cases ensured that the code responded gracefully to invalid input, an important aspect of building secure and stable software. My strategy followed test-driven development (TDD) principles where possible, where writing tests guided my implementation process.

Other software testing techniques I did not employ include integration testing, regression testing, and system testing. Integration testing examines the interaction between multiple components, such as the relationship between services or services and a database. Regression testing ensures that new code changes do not negatively impact existing functionality, typically used when applications are updated frequently. System testing focuses on the software's behavior in a complete environment. These testing strategies are essential when delivering full-stack applications, especially with external APIs or databases.

While these techniques didn’t apply to the scope of this project, I understand their importance and would incorporate them in future phases of development involving APIs, persistent data layers, or production-level deployments. Each technique plays a critical role in ensuring comprehensive quality assurance.

# Mindset

Working on this project required a mindset focused on caution, discipline, and curiosity. As a developer and tester, I approached each component with the belief that even simple logic might fail if untested. This caution led me to create test cases that evaluated not only expected behavior but also edge cases, exceptions, and invalid conditions. For example, in `TaskTest.java`, I tested null task names and invalid task IDs to verify the robustness of the input validation logic.

My disciplined approach helped ensure that no method or constructor was left untested. I made sure that each setter, getter, and service method had corresponding unit tests. Understanding how constructors influenced getter methods led me to carefully verify object state after creation. In `AppointmentServiceTest.java`, I validated that the created appointment’s fields accurately reflected the provided inputs, particularly date formats and unique IDs.

Bias can present challenges in testing, especially when reviewing one’s code. Developers tend to assume their logic is correct, which can lead to skipped validations or narrow test scenarios. To mitigate this, I deliberately created tests that were intended to fail. I implemented assertions that expected exceptions or invalid outputs to ensure my implementation wasn’t based solely on best-case scenarios. This helped me find bugs I might have otherwise overlooked.

By putting myself in the shoes of an end user—or a QA tester unfamiliar with the logic—I was able to question my assumptions and enhance test coverage. I found that adopting an adversarial mindset as a tester helped strengthen the integrity of my code and highlighted areas where defensive coding was required.

# Commitment to Quality and Avoiding Technical Debt

Writing high-quality tests isn’t just about catching bugs; it’s about building confidence in the software and promoting maintainability. One of the biggest threats to quality in software engineering is technical debt—short-term trade-offs made to speed up development, which often result in long-term instability.

In this project, I committed to avoiding technical debt by staying consistent in my coding and testing practices. For instance, I avoided duplicating logic by using helper methods when creating repeated test data. I documented each test case with clear naming and structure to ensure readability. Each test method was concise, targeted a single purpose, and provided meaningful feedback in case of failure.

Looking ahead, I plan to maintain this discipline by incorporating static code analysis tools, setting code coverage goals, and implementing code review workflows. I will also prioritize documentation to communicate test intentions to future developers. Quality is not just about passing tests—it’s about making sure the code is understandable, maintainable, and extendable.

In conclusion, this project has strengthened my belief that testing is both a technical necessity and a professional responsibility. The combination of clear testing techniques, a cautious mindset, and a commitment to avoiding technical debt will serve as a strong foundation for my future work in software engineering.