**Summary and Reflections Report**

During Project One, I developed and tested three core features of a mobile application for a client: the ContactService, TaskService, and AppointmentService. My approach to unit testing was rooted in understanding the specific software requirements for each of these components and ensuring that my JUnit tests would verify both core functionality and boundary conditions.

For each feature, I created comprehensive unit tests using JUnit 5. The ContactService tests verified that each field (first name, last name, phone number, address, and contact ID) was initialized correctly and followed the given constraints (e.g., 10-digit phone number, non-null fields). The TaskService tests validated that tasks were created with valid IDs, names, and descriptions, and that exceptions were thrown when any invalid input was provided. The AppointmentService had slightly more complex logic due to the date handling, so I paid special attention to validating date formats and ensuring null values were rejected.

My approach was highly aligned with the software requirements. For instance, in the TaskService requirements, it was stated that the task name must not exceed 20 characters. My JUnit test explicitly included:

assertThrows(IllegalArgumentException.class, () -> new Task("1234567890", "ThisNameIsWayTooLong", "Description"));

This validated that the software correctly rejected oversized input, which directly matched the business rule. I applied similar logic to every constraint across the services.

The overall quality of my JUnit tests can be defended through both test coverage and variety. I didn’t just test "happy path" scenarios; I also tested edge cases, such as passing null values or strings exceeding allowed lengths. I used a code coverage tool (built into my IDE) and verified that my test coverage was close to 100% for each service class. That gave me confidence that I wasn’t leaving any major logic unchecked.

Writing the JUnit tests was a learning experience. At first, I struggled a bit with structuring test methods clearly, but I eventually adopted a consistent naming format like testValidContactCreation() or testNullPhoneThrowsException(), which made it easier to understand each test’s purpose. To ensure the code was technically sound, I used assert statements like:

assertEquals("1234567890", contact.getContactId()); assertThrows(IllegalArgumentException.class, () -> new Contact(null, "Doe", "1234567890", "123 Main St"));

These lines made sure that both proper and improper inputs were handled as expected.

To keep my test code efficient, I reused setup logic whenever possible instead of duplicating object creation. For example, I used @BeforeEach to initialize commonly used variables. This reduced repetition and made my tests easier to manage as the project grew.

**Reflection**

**Testing Techniques**

In this project, I mainly used unit testing and boundary testing. Unit testing allowed me to test small, independent blocks of code (like constructors and getter methods) in isolation. This was important because it helped catch bugs at the class level before they could escalate. Boundary testing was another essential technique I applied—ensuring that inputs at or near the limits (like 10-character phone numbers or 20-character task names) behaved correctly.

I did not use integration testing or system testing in this project. Integration testing would have helped me test how different components (like ContactService and TaskService) interact with each other, but that was outside the project scope. System testing, which evaluates the whole application as a unified system, would be more appropriate in later development stages or in a QA environment.

Each testing technique serves different purposes. Unit testing is ideal for catching logic bugs early and is most useful during development. Integration testing is helpful when several components must work together, such as in a microservices environment. System testing is essential when verifying that user requirements are met end-to-end. Understanding when to apply each method is key to maintaining a reliable software product.

**Mindset**

Throughout the project, I adopted a cautious and detail-oriented mindset. As a software tester, I had to think not just like a developer, but also like a user and a potential critic. I asked myself questions like, “What happens if this field is left blank?” or “What if someone enters 11 digits for a phone number?” This mindset helped me anticipate potential issues and write more effective tests.

Appreciating the complexity and interrelationships in the code was important. For example, in the AppointmentService, date validation relied on ensuring the appointment ID and description were valid simultaneously. Testing one without the other would not have been sufficient. A failure in date validation could potentially make the whole appointment invalid, so it was crucial to test that each element was validated in context, not just in isolation.

Bias was a real concern, especially because I was writing tests for code I had also written. To reduce this bias, I tried to approach testing with a “break-it” mentality. I asked myself, “How would someone misuse this class?” and wrote tests to simulate those scenarios. If I had only tested with valid data, I would have missed the opportunity to strengthen the program’s robustness. For example, testing what happens when null is passed into the Contact constructor helped me uncover missing validation logic early.

As a software engineering student, I’m learning how critical it is to be disciplined when it comes to code quality. Cutting corners may save time initially, but it can lead to technical debt—something I want to avoid as I grow in this field. For example, if I had skipped writing tests for edge cases, I might not have caught issues with string length validation, which could have led to bugs later on.

To avoid technical debt in the future, I plan to follow test-driven development (TDD) practices whenever possible, where I write tests before implementing features. I’ll also aim to review code more frequently and use tools like static analyzers and linters to enforce consistency and catch early problems.

**Conclusion**

Overall, working on Project One taught me the importance of a thoughtful, strategic approach to unit testing. By aligning my JUnit tests with software requirements, applying sound testing techniques, and maintaining a disciplined mindset, I was able to create reliable and maintainable code. This project has given me a strong foundation in software testing that I will carry into future work as a developer and tester.