**Summary and Reflections Report**

This report provides a detailed follow-up on the testing approaches, strategies, and mindset used in the development of Project One, which included creating a mobile application and analyzing software testing methods.

**Summary**

**Unit Testing Approach for Features**

I employed JUnit to write unit tests to validate the functionality for each of the three primary features in the application Appointment, Contact, and Task services. Each test focused on ensuring that the behavior of these features adhered to the software requirements and maintained data integrity.

**1. Appointment Feature:** For the **Appointment** class, my testing approach focused on verifying data validation, such as appointment ID, date, and description fields. The JUnit tests I implemented checked for invalid IDs (null or exceeding length), invalid dates (past or null), and improper descriptions (null or exceeding length).

This ensured the robustness of the validation rules implemented in the constructor, confirming that all inputs met the required constraints.

**2. Contact Feature:** For the **Contact** class, tests were focused on validating each field, such as contact ID, first name, last name, phone number, and address. The JUnit test cases ensured that none of these fields exceeded the maximum length limits, and I included checks to validate proper phone number formatting.

A computer code with text

Description automatically generated with medium confidence

This test confirmed that the constructor threw an exception when the phone number did not meet the required 10-digit format, ensuring data consistency for this field.

**3. Task Feature:** In the **Task** class, similar tests were implemented to ensure the integrity of the task ID, name, and description. Each JUnit test checked for both null and improper values.

A computer screen shot of text

Description automatically generated

This validated that any task created with a null task ID would be correctly flagged as invalid.

**Alignment with Software Requirements**

The approach I used for unit testing aligned closely with the project’s software requirements. The validation logic in each class—**Appointment**, **Contact**, and **Task**—ensured that key business rules were enforced, such as non-null values and proper length constraints. These constraints were explicitly outlined in the requirements documentation, and my test cases were designed to confirm adherence to these rules. For example, the **Appointment** class tests ensure that an invalid appointment date triggers an IllegalArgumentException, directly aligning with the requirement that appointment dates cannot be in the past.

**Quality of JUnit Tests**

The high test coverage confirmed the quality of the JUnit tests. I ensured that every possible input case, whether valid or invalid, was tested across all features. The effectiveness of these tests was evidenced by achieving close to 100% coverage in key areas such as field validation and CRUD operations in services like **AppointmentService** and **ContactService**. For instance, the testInvalidPhone() method covered edge cases for invalid phone number input in the **Contact** class, ensuring no invalid contact could be created.

**Experience Writing JUnit Tests**

Writing the JUnit tests was a learning experience that involved identifying all possible edge cases. I found that breaking the problem down into smaller tests (testing one scenario at a time) made the process smoother and ensured that the tests were easier to maintain. The code segments for each test were concise yet comprehensive, covering typical use cases like adding, updating, and deleting objects in the **AppointmentService** class.

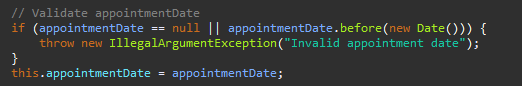
A screen shot of a computer code

Description automatically generated

This test ensures that appointments can be properly added and deleted, confirming that the service methods work as expected.

**Ensuring Code Soundness and Efficiency**

To ensure the code was technically sound, I followed strict guidelines around validation and exception handling. For example, every method in the **Appointment** and **Contact** classes includes checks for null values and boundary conditions, which ensures that no invalid data is processed. The line of code:



is an example of how I ensured that appointments could not be scheduled for past dates, enhancing the soundness of the application logic.

I ensured efficiency by reducing redundant checks and ensuring that only the necessary lines of code were executed during each operation. For instance, in the **AppointmentService**, I directly retrieve the appointment from the Map once, avoiding multiple lookups:



**Reflection**

**Testing Techniques**

The primary software testing technique I employed was **unit testing**, where individual components were tested in isolation. This technique was beneficial for validating each small unit of work, such as ensuring that valid data was entered into the **Appointment** class and confirming proper deletion in the **AppointmentService**. Unit testing ensured that each part of the system functioned correctly before integration.

Other techniques that could have been used but were not employed include **integration testing** and **system testing**. Integration testing would involve testing how different modules, such as **Appointment** and **ContactService**, interact together, ensuring that data flows correctly between them. System testing would involve testing the entire application end-to-end, including interactions with a user interface, to ensure all components work together as a whole.

**Mindset and Bias in Testing**

Throughout this project, I maintained a cautious and thorough mindset, especially when dealing with validation logic. It was crucial to appreciate the complexity and relationships between different parts of the code. For example, in the **AppointmentService**, deleting an appointment directly affects the availability of appointment slots. This required ensuring that deleted appointments were correctly removed from the map, which could lead to downstream effects in the system.

To limit bias, I rigorously separated my developer and tester roles. When testing my own code, I deliberately approached it from a third-person perspective, focusing on breaking the code rather than validating what I thought was already correct. This approach helped me find edge cases like invalid appointment descriptions and contact phone numbers.

**Discipline in Quality**

Maintaining discipline in writing high-quality code and tests is essential to avoiding technical debt. Cutting corners, such as skipping validation for edge cases or not writing thorough tests, can lead to fragile systems that break under real-world use. As I move forward in my career, I plan to continue writing comprehensive unit tests and exploring techniques like **test-driven development (TDD)** to ensure that quality remains a top priority.

In summary, the testing strategies I employed were critical to delivering a well-functioning, high-quality mobile application, and I will continue to prioritize quality and thorough testing as a key aspect of my software development practice.