**Unit Testing Approach**

For this project, I developed and tested three main features for the mobile application: the ContactService, Task Service, and Appointment Service. My unit testing approach centered on verifying the creation, modification, and deletion of objects, as well as ensuring that all data validation rules were enforced.

For the Contact Service, I wrote JUnit tests that verified correct contact creation, updates, and deletions. For example, in ContactServiceTest, the testAddContact() method ensures that a contact can be successfully added and retrieved, while testDeleteContact() confirms proper removal from the collection. Invalid inputs such as null or oversized values were also tested in ContactTest to confirm defensive programming practices.

For the Task Service, my tests ensured that tasks were correctly added, deleted, and updated. For example, testAddDuplicateTaskThrowsException() in TaskServiceTest verified that attempting to add a task with an existing ID raised an exception, enforcing data integrity.

For the Appointment Service, I tested both valid and invalid appointment creation in AppointmentTest. For example, testInvalidAppointmentDate() ensures that past or null dates are rejected, while testValidAppointment() confirms that future appointments are accepted. These tests validated the business logic that prevents users from creating appointments in the past.

**Alignment to Software Requirements**

My testing approach was closely aligned to the software requirements, which emphasized robust data validation and proper handling of edge cases. Each JUnit test directly corresponds to a functional or validation requirement. For example, the requirement that phone numbers must be exactly 10 digits long is enforced in the Contact class constructor and verified through tests such as testInvalidPhoneThrowsError() in ContactTest.

**Quality and Coverage**

I ensured the overall quality of my JUnit tests by focusing on code coverage and logical path testing. Each test method was designed to cover both valid and invalid input scenarios. The high number of assertion statements (assertEquals, assertNull, and assertThrows) across the tests demonstrates comprehensive coverage of both positive and negative cases. If measured with a coverage tool such as JaCoCo, my test coverage would likely approach 100% for model and service classes, since every method and validation condition is explicitly exercised.

**Experience Writing JUnit Tests**

Writing JUnit tests was a straightforward yet insightful experience. To ensure my code was technically sound, I incorporated assertions that confirmed the expected state after each operation. For example, in ContactServiceTest, the line

assertEquals("Jane", service.getContact("1").getFirstName());

ensures that the updateFirstName() method correctly modifies the stored object.

To ensure efficiency, I avoided redundant setup code by creating reusable Contact, Task, and Appointment instances within each test, and by limiting the scope of data stored in service collections. Additionally, I used helper methods like getFutureDate() and getPastDate() in AppointmentTest to reduce repetitive date logic.

**Reflection**

**Testing Techniques**

The main testing technique I employed was unit testing, supported by black-box testing and boundary value analysis. Unit testing focused on individual methods and classes, verifying that each unit behaved as expected in isolation. Black-box testing was used when verifying input validation—tests were designed without concern for internal implementation, only for inputs and outputs. Boundary value analysis ensured proper handling of limits, such as maximum string lengths (e.g., 10 characters for contactId, 50 for description) and date constraints (future vs. past).

Other techniques not used include integration testing, system testing, and performance testing. Integration testing would verify interactions between services (for example, linking tasks to appointments), while system testing would assess the entire application end-to-end. Performance testing would measure scalability and response time under load. Although not necessary for this project, those techniques are valuable for larger, production-level systems.

In practical terms, unit and boundary testing are essential early in the development lifecycle to catch logical and validation errors quickly, while integration and system testing are better suited for later phases to ensure cross-component reliability.

**Mindset**

While working on this project, I adopted a cautious and detail-oriented mindset. I recognized the importance of thoroughly testing all possible input scenarios to avoid hidden defects. For instance, in the AppointmentTest, I deliberately tested both null and past dates to confirm the robustness of date validation logic.

As a tester, I made an effort to limit bias by viewing my code from an external user’s perspective. Even though I wrote both the implementation and tests, I avoided assuming that my methods would work correctly. For example, I deliberately tested invalid conditions such as duplicate IDs and overlong field values to challenge my own assumptions.

Bias can be a concern when developers test their own code because they may subconsciously avoid testing failure cases. Writing tests immediately after coding each class helped me maintain objectivity and discover issues early.

Finally, I learned the importance of maintaining discipline and commitment to quality. Cutting corners in testing can introduce technical debt that becomes costly to fix later. For instance, if I ignored validation testing for phone numbers or appointment dates, those errors could cascade into larger system issues. To avoid technical debt in the future, I plan to maintain strict testing standards, conduct code reviews, and continue practicing test-driven development (TDD) whenever possible.