# **Mobile Application Testing Project**

Hamad Alghaithi

Southern New Hampshire University

CS 320

06/22/2025

## Summary

### Unit Testing Approach and Requirements Alignment

My unit testing approach for the Contact, Task, and Appointment services strategically aligned with software requirements through comprehensive validation testing. Each service required specific validation rules that I tested systematically.

Contact service requirements specified contactId ≤10 characters, firstName/lastName ≤10 characters, phone exactly 10 digits, and address ≤30 characters. My tests directly validated each requirement with methods like testContactIdTooLong() and testPhoneExactly10Digits(). The test testContactIdExactly10Characters() validates boundary conditions where inputs are exactly at maximum allowed length.

Task service requirements included taskId ≤10 characters (immutable), name ≤20 characters, and description ≤50 characters. My boundary value testing with testNameExactly20Characters() ensured precise compliance while verifying update functionality and ID immutability.

Appointment service focused on appointmentId ≤10 characters (immutable), future dates only, and description ≤50 characters. Temporal validation through testAppointmentDateInPast() ensured the unique requirement that appointments cannot be scheduled in the past.

### Test Effectiveness and Technical Quality

My JUnit tests achieved over 80% code coverage, demonstrating effectiveness through comprehensive testing of all code paths. Coverage included all CRUD operations and error conditions like duplicate IDs and invalid updates. The testAddContactDuplicateId() method tests exception paths when adding contacts with existing IDs.

For technical soundness, I used assertThrows() for proper exception testing: assertThrows(IllegalArgumentException.class, () -> { new Contact(null, "John", "Doe", "1234567890", "123 Main St"); }); ensures null contact IDs are rejected appropriately.

For efficiency, I used @BeforeEach annotations: @BeforeEach void setUp() { contactService = new ContactService(); } ensuring clean initialization without code duplication. Each test method focuses on single functionality aspects for maintainable, efficient execution.

## Reflection

### Testing Techniques

**Employed Techniques:** Unit testing was primary, testing individual components in isolation through separate test classes for each service. Black-box testing designed test cases from requirements specifications without examining internal code structure. Boundary value analysis tested values at input domain boundaries where errors commonly occur, such as exact character limits.

**Unused Techniques:** Integration testing focuses on component interactions but wasn't needed since services were independent with in-memory data structures. System testing evaluates complete end-to-end systems but wasn't applicable for individual service components. Performance testing measures responsiveness under load but was unnecessary for simple functional validation.

**Practical Applications:** Unit testing is essential for agile development, enabling safe refactoring and continuous integration. Integration testing becomes critical in microservices architectures ensuring proper communication between services. System testing is indispensable for complete applications with user interfaces and complex workflows, particularly in healthcare systems requiring regulatory compliance.

### Professional Mindset

**Caution Employment:** I adopted a skeptical mindset, systematically questioning code behavior assumptions. When testing phone validation, I tested null values, incorrect lengths, and non-numeric characters rather than assuming single tests sufficed. I recognized validation logic interdependencies—constructor validation affects both object creation and service operations, requiring tests for both successful operations and various failure modes.

**Limiting Bias:** As both developer and tester, I generated test cases directly from requirements rather than implementation knowledge, preventing unconscious avoidance of difficult test scenarios. Developer bias is a significant concern when testing own code—developers naturally test expected successful cases while potentially overlooking edge cases that could reveal implementation flaws.

**Quality Discipline:** I maintained discipline by writing comprehensive tests even when minimal coverage would be easier, ensuring thorough requirement validation rather than superficial testing. Cutting corners creates technical debt that compounds over time—inadequately tested code leads to production bugs, emergency fixes, and reduced reliability. I plan to avoid technical debt through consistent coding standards, comprehensive test coverage, and regular code reviews.

Technical debt prevention requires long-term perspective. Well-tested code is easier to maintain, extend, and debug, leading to faster development cycles and higher customer satisfaction. In professional practice, I will advocate for adequate testing time in project planning and resist pressure to skip quality practices for short-term delivery goals.

## Conclusion

This project reinforced the importance of systematic testing approaches, comprehensive requirement validation, and disciplined quality practices. The experience highlighted how proper testing techniques ensure code reliability and maintainability while supporting business requirements. Moving forward, I will continue applying these testing principles and quality mindset to contribute to successful software development projects, understanding that initial investment in quality pays dividends through reduced maintenance costs and improved system reliability.