# CS 320 Project Two: Summary and Reflections Report

**Matthew Biletnikoff**

## Summary

This report provides a comprehensive evaluation of the unit testing strategies, technical soundness, and professional reflections applied during the development of the ContactService, TaskService, and AppointmentService components for the mobile scheduling system. Each of these services was tested using structured, modular unit testing principles aligned with software engineering best practices and the requirements of the CS 320 project.  
  
**Unit Testing Approach:**  
 Each service followed a structured unit testing methodology using JUnit 5, focusing on verifying both positive and negative test cases. The ContactService tests verified that each contact object retained a unique identifier and enforced strict data validation for first name, last name, phone number, and address fields. The TaskService tests validated the creation, modification, and deletion of tasks while enforcing unique task IDs. The AppointmentService tests verified the integrity of appointment IDs, start and end times, and deletion logic, ensuring no overlap or invalid dates occurred. Each service was tested in isolation to eliminate cross-dependencies and ensure test independence.  
  
**Alignment With Requirements:**  
 The testing approach was tightly aligned with the system requirements, ensuring that each business rule was implemented as specified. For example, the ContactServiceTest validated constraints such as maximum field lengths and phone number formatting, reflecting direct alignment with functional requirements. The TaskService and AppointmentService tests validated that IDs were immutable and that null or empty values were rejected. Assertions such as assertThrows and assertEquals were consistently used to verify expected behaviors, ensuring that input validation and error handling matched requirement specifications.  
  
**Quality and Effectiveness of JUnit Tests:**  
 The tests achieved high coverage through multiple assertions per test and comprehensive validation of both valid and invalid inputs. Coverage analysis using JaCoCo demonstrated nearly full line and branch coverage across all service classes. Each JUnit test method followed descriptive naming conventions such as addContact\_enforces\_unique\_id and deleteTask\_removes\_entry, which enhanced readability and traceability. The tests also validated exception handling, confirming system resilience under erroneous conditions.  
  
**Experience Writing JUnit Tests:**  
 Developing the test suite provided valuable experience in planning and implementing structured test cases. The @BeforeEach annotations ensured test isolation, preventing data leakage between tests. Logical grouping of tests improved organization and debugging efficiency. Writing JUnit tests for this project emphasized the need to anticipate edge cases, such as null input or duplicate identifiers, and confirmed the importance of testing both valid and invalid operations to fully ensure application reliability.  
  
**Ensuring Technical Soundness and Efficiency:**  
 The code maintained technical soundness by adhering to the principles of separation of concerns and minimal redundancy. Efficiency was improved by reusing initialized objects and employing lightweight assertions. Code such as service = new ContactService(); in the @BeforeEach method ensured that every test began from a clean, reproducible state. The combination of assertEquals, assertTrue, and assertThrows validated both correctness and reliability without introducing unnecessary computation overhead.

## Reflection

**Testing Techniques Employed:**  
 The project employed unit testing, boundary value analysis, and equivalence partitioning as the core testing techniques. Unit testing verified each service component independently, ensuring correctness at the method level. Boundary value analysis tested data at its limits, such as maximum character counts for names and addresses. Equivalence partitioning organized inputs into valid and invalid sets to optimize coverage without redundant tests. Together, these techniques ensured accuracy, scalability, and maintainability.  
  
**Testing Techniques Not Used:**  
 This project did not include integration testing, system testing, or user acceptance testing (UAT), which would typically validate module interactions and end-to-end workflows. Integration testing ensures correct communication between classes (e.g., ContactService interacting with AppointmentService), while UAT confirms that system functionality aligns with user expectations. These tests would be implemented in later stages once the individual components had passed all unit-level validations.  
  
**Practical Uses and Implications:**  
 The testing techniques used in this project are ideal for early-stage validation in an Agile development environment. They provide confidence in core functionality and enable continuous integration by detecting regressions quickly. In professional contexts, these tests can be extended into CI/CD pipelines to enforce quality gates before deployment. For larger enterprise systems, incorporating integration and stress testing further ensures that tested modules scale and perform under load.  
  
**Mindset and Approach to Testing:**  
 A meticulous and analytical mindset guided the testing process. Acting as both developer and tester required caution, curiosity, and skepticism toward one’s own code. Each method was validated against both expected and unexpected conditions. Recognizing the complexity and interdependence between services was essential, for example, ensuring appointment validity depended on consistent date management in related task records. This awareness fostered a deeper understanding of software reliability and modular design principles.  
  
**Limiting Bias in Testing:**  
 Bias was mitigated through deliberate test design and negative scenario validation. Tests were not written to confirm correctness alone but to intentionally expose potential weaknesses. For instance, intentionally attempting to add duplicate contacts or appointments with invalid time ranges helped uncover potential oversights. Testing from the mindset of an external evaluator reduced confirmation bias and encouraged a more objective evaluation of code behavior.  
 **Commitment to Quality and Avoiding Technical Debt:**  
 Maintaining discipline in testing is critical for ensuring software quality. Cutting corners in test creation often results in latent defects that accumulate as technical debt. By emphasizing thorough and maintainable JUnit testing, this project reinforced the principle of building quality into each development phase. To avoid technical debt in future projects, best practices will include continuous testing, regular refactoring, and automated build verification. A culture of disciplined testing not only prevents costly defects but also ensures long-term project sustainability and client satisfaction.

**References**

Beizer, B. (1995). *Black-box testing: Techniques for functional testing of software and systems* (2nd ed.). John Wiley & Sons.

Jorgensen, P. C. (2018). *Software testing: A craftsman’s approach* (5th ed.). CRC Press.

Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing* (3rd ed.). John Wiley & Sons.

Somerville, I. (2020). *Software engineering* (10th ed.). Pearson.

IEEE. (2013). *IEEE Standard for Software and System Test Documentation (IEEE Std 829-2008)*. Institute of Electrical and Electronics Engineers.