CS 320

Ryan O’Toole

October 23, 2025

**7-2 Project Two**

Throughout the development of the contact, task, and appointment services for the mobile application, I focused on writing reliable and thorough unit tests using JUnit 5. My approach emphasized testing each class’s core functionality in isolation to confirm that the code met the project requirements and behaved as expected under both normal and invalid conditions. For each service, I used assertion-based testing with methods like assertEquals, assertTrue, assertFalse, and assertThrows to verify outputs, enforce input constraints, and confirm that exceptions were handled correctly.

For the ContactService, I tested the ability to add, update, and delete contacts while verifying that each contact’s name, phone number, and ID met required validation rules. This included ensuring that IDs were unique and non-null, and that names and phone numbers followed the correct format and length restrictions. These tests aligned closely with the project’s requirement that all user information must be consistently valid and accessible at runtime.

The TaskService tests focused on validating that each task could be created, retrieved, and deleted correctly, and that invalid or overly long task names or descriptions were rejected. I confirmed that each new task generated a unique identifier and that deletion methods properly removed entries without affecting unrelated ones. These tests ensured the service handled edge cases effectively, which matched the project requirement for data integrity and user reliability.

For the AppointmentService, I tested the creation and validation of appointments to confirm that each appointment had a valid date and description. The tests confirmed that past or null dates were rejected and that all valid appointments were correctly stored and retrievable. This directly aligned with the functional requirement that users could only schedule legitimate appointments with accurate details.

Overall, my unit testing approach was highly aligned with the project requirements. Each test method was designed to verify that user-facing functionality behaved according to the specifications in the design documents. The coverage report confirmed that the tests touched all critical branches and scenarios. Because every test passed successfully, I was confident in the stability and reliability of the final codebase.

The effectiveness of the tests was reflected in the coverage percentage and the absence of unexpected behavior during execution. Every logical path in the service classes was tested at least once, providing confidence that the system would behave predictably in production. Writing these JUnit tests helped me better understand how detailed and structured testing improves the overall quality of software. To ensure technically sound code, I kept each test independent, used clear naming conventions like testAddContact and testDeleteAppointment, and followed the Arrange-Act-Assert pattern. For efficiency, I reused setup methods where possible and eliminated redundant code across test classes.

**Reflection**

**Testing Techniques**  
 The main testing techniques I employed were unit testing and boundary testing. Unit testing allowed me to isolate and test individual components without dependencies, which made debugging and validation easier. Boundary testing helped confirm that validation rules worked correctly at their limits, such as ensuring string lengths or date constraints were enforced. I also used negative testing to confirm that invalid inputs were properly rejected and that error messages or exceptions were triggered as expected. These methods helped ensure that the system was both functional and resilient.

**Other Techniques**  
 Other testing techniques that could have been used but were not necessary for this project include integration testing, system testing, and regression testing. Integration testing focuses on verifying how different modules interact, while system testing checks the entire application as a whole. Regression testing ensures that changes do not break existing functionality. Although these were beyond the scope of this assignment, they would be important in a larger project to confirm that combined modules perform correctly together and that updates do not introduce new bugs.

**Uses and Implications of Techniques**  
 Unit testing is useful in nearly every software development context, especially during early development stages when ensuring correctness of individual components is crucial. Boundary testing is particularly valuable in user-facing applications where invalid inputs could lead to crashes or security issues. Negative testing helps identify weak points in input handling and improves application robustness. Integration and regression testing, while not used here, are critical in larger enterprise projects where multiple services interact. Together, these techniques form a layered testing strategy that increases code confidence, reduces maintenance costs, and supports long-term scalability.

**Mindset** Throughout the project, I learned how important it is to approach software testing with caution and discipline. Employing caution meant being careful not to make assumptions about code behavior and ensuring that every test was based on clear expectations. For example, when testing appointment creation, I made sure to check not only valid dates but also invalid and null inputs to confirm that the system handled errors properly. Appreciating the complexity and interrelationships of the code made me more mindful of how small changes could impact other components, encouraging me to test thoroughly before committing updates.

Limiting bias was another important aspect of my approach. As the developer of the code I was testing, it was easy to fall into the habit of assuming the code worked as intended. To counteract that bias, I tested my methods from the perspective of an external user. For instance, I deliberately entered unexpected or invalid data to challenge the assumptions I had made when writing the logic. If I had skipped that step, I might have missed potential edge cases or unhandled exceptions.

Lastly, discipline was central to maintaining quality throughout this project. Writing and running tests consistently required focus and attention to detail, especially when debugging or refactoring existing code. It was tempting at times to take shortcuts or skip tests for “obvious” logic, but doing so would have increased the risk of hidden errors. I also learned that maintaining disciplined testing habits helps prevent technical debt, which can build up when developers defer testing or documentation to meet short-term deadlines. Moving forward, I plan to avoid technical debt by continuing to follow a test-driven approach, documenting functionality clearly, and maintaining modular, reusable code structures.

**Conclusion**

This project reinforced the importance of methodical testing and thoughtful reflection in software development. Writing and running JUnit tests for the contact, task, and appointment services taught me how structured testing contributes directly to software reliability and maintainability. The combination of unit, boundary, and negative testing provided confidence that each component met its requirements, while reflection on bias, caution, and discipline strengthened my mindset as a software engineer. Testing is not just about finding bugs—it is about proving that the code functions as intended, and doing so with care, integrity, and precision.