Sabrina Shaver

Professor Antoun

CS 320

19 October 2025

Summary and Reflections Report

For Project One, I developed and tested three key back-end services for a mobile application: the contact service, the task service, and the appointment service. My unit testing approach focused on ensuring that each service met its functional requirements using structured, methodical JUnit tests that targeted valid and invalid inputs, edge cases, and expected behaviors.

For the Contact service, my tests focused on verifying that contact objects could be created with valid fields and that invalid input, such as null values or fields that exceeded length requirements, would correctly trigger exceptions. I also wrote unit tests to verify that updates to mutable fields behaved as expected while the contact ID remained immutable.

For the Task service, I adopted a similar strategy. I tested valid task creation, updates to the name and description fields, and enforcement of length restrictions. I included negative test cases to ensure that attempts to create or update a task with invalid data would fail, and I verified that task IDs remained unique and unchangeable after creation.

For the Appointment service, I implemented unit tests that checked whether appointment objects could be created with valid IDs, future dates, and valid descriptions. I added tests for invalid IDs (null or exceeding 10 characters), invalid dates (null or in the past), and invalid descriptions (null or exceeding 50 characters). I also tested the behavior of the appointment service when adding, retrieving, and deleting appointments, including attempts to add duplicate IDs and delete non-existent appointments.

I can defend the overall quality of my JUnit tests through their coverage. Each test class covers both positive (valid) and negative (invalid) scenarios, as well as edge cases. For example, in the appointment tests, I checked both null and overlength inputs for IDs and descriptions, along with valid boundary values. This breadth of coverage provides strong assurance that the services behave as expected.

Writing the JUnit tests was a structured process. To ensure that my code was technically sound, I followed a consistent pattern of asserting expected outcomes using assertEquals for valid data and assertThrows for exceptions. For example:

assertThrows(IllegalArgumentException.class, () -> new Appointment(null, futureDate, "Desc"));

This line tests a requirement violation in a concise way. To ensure efficiency, I avoided redundant tests and used variables like futureDate to streamline setup. For example:

Date futureDate = new Date(System.currentTimeMillis() + 100000);

Appointment appt = new Appointment("001", futureDate, "Meeting");

This ensures that the tests remain readable and quick to execute while maintaining comprehensive coverage.

The primary software testing technique I used for this project was unit testing with black box testing characteristics. I focused on testing inputs and outputs without relying on the internal implementation details of the methods. Each test case examined whether a method produced the correct result given a certain input, aligning with the specifications.

Additionally, I applied boundary value testing for fields with character limits. For example, appointment IDs were tested at lengths of exactly 10 and 11 characters to ensure that the class correctly enforced its length constraints. I also used negative testing by intentionally passing null or invalid data to verify error handling.

Each testing technique has different practical uses. Unit testing is essential early in development to catch small, isolated bugs before they propagate. Integration testing is critical when combining multiple services to ensure they work together correctly, such as verifying that a scheduling service correctly interacts with a notification service. System testing would be used closer to deployment to verify that the application meets business requirements end-to-end, and performance testing would be valuable for ensuring reliability under load.

Throughout this project, I maintained a cautious and analytical mindset. Acting as a software tester required me to consider both expected and unexpected ways the code could fail. For example, in the appointment service, I tested deletion of both existing and non-existent appointments. By doing so, I anticipated potential edge cases and handled them gracefully with clear exceptions.

It was important to appreciate the complexity and interrelationships of the code because errors in one area, such as incorrectly allowing null appointment dates, could easily lead to failures in another, such as retrieval or sorting operations later. By maintaining strict validation rules in the appointment class, I helped ensure that the service layer could safely assume valid data, simplifying other tests.

To limit bias in my review of the code, I treated my own code as if it had been written by someone else. I intentionally tried to “break” the service methods by passing in malformed inputs. For example, I used strings far longer than the allowed limits to ensure that my length checks were robust.

As a developer, I recognize that bias can easily occur when testing one’s own code. Developers may subconsciously avoid testing edge cases they believe “should work,” leading to blind spots. For example, if I had assumed that IDs would always be well-formed, I might not have tested null IDs and missed a critical bug.

Finally, I understand the importance of discipline and avoiding technical debt. Cutting corners in testing can lead to subtle bugs that are much more expensive to fix later. For example, if I skipped null input checks in my tests now, a future change could break the application without detection. To avoid technical debt, I plan to maintain thorough test coverage, follow a test-driven development approach when possible, and document edge cases carefully. By doing so, I will help ensure that future enhancements and refactoring can be done with confidence.