Kileigh Adams

SNHU

CS 320

October 19, 2025

**Project Two**

According to the customer requirements specification, IDs must be unique, non-null, and under ten characters, while names, addresses, and phone numbers each have their own length and format constraints. The AppointmentService also required that appointment dates be scheduled in the future. My tests were built directly around these requirements. For example, the Contact class enforces a ten-character limit on names, a strict ten-digit phone number format, and address validation for length and null values. My test suite validated every one of these rules, verifying that exceptions were thrown for invalid input and that valid objects were successfully created. I designed unit tests to confirm that each validation rule functioned exactly as described in the requirements document.

Eclipse reported around ninety to ninety-five percent line coverage across all classes. This coverage level demonstrates effectiveness because it validates all decision branches that handle user input and enforce business rules. The uncovered lines consisted only of auto-generated methods like getters and hashCode implementations, which are deterministic and require no validation. Every functional branch, conditional check, and exception pathway was tested through positive and negative test cases, confirming that the software handled data correctly under realistic scenarios and that business rules were met before integration.

My test code was technically sound because it followed three consistent strategies: structured exception handling, purposeful assertion selection, and strict test isolation. I used assertThrows whenever an invalid input should raise an exception, ensuring that the software failed safely. For example, in the Appointment tests, I confirmed that past dates raised exceptions through assertThrows(IllegalArgumentException.class, () -> new Appointment("A1", pastDate, "Checkup"));. I also made sure each test focused on a single behavior using assertEquals, assertTrue, and assertFalse. The tests were isolated by reinitializing service objects before each run to prevent data leakage or shared state, ensuring that results were reproducible and failures could be traced to a single cause.

I ensured efficiency through several strategies. First, each test class used @BeforeEach to initialize a fresh service instance, which avoided redundant setup code across more than fifteen test methods. Second, I chose HashMap as the data structure for all services, providing O(1) lookup and insertion time (line 8 of ContactService.java). Third, I minimized object creation by reusing test fixtures within each method. Finally, I wrote focused assertions that checked one behavior per method, such as testInvalidPhoneNumberLength() verifying only phone validation. Together, these strategies made the code efficient and maintainable while keeping tests lightweight and logically independent.

The main testing techniques I used were unit testing, boundary testing, and exception testing. Unit testing isolated individual methods and verified logic without external dependencies. Boundary testing focused on input limits, such as the ten-character maximum for contact names and date validation between today and future dates. Exception testing ensured that invalid data triggered controlled exceptions rather than causing runtime crashes. These techniques gave me fine-grained visibility into how each method behaved and helped confirm that data validation and error handling worked as expected (JUnit 5 Documentation, n.d.).

In a larger project, I would also apply integration testing, system testing, user acceptance testing, regression testing, and performance testing. Integration testing checks how multiple components interact, such as verifying that ContactService and TaskService work together when a user profile needs associated tasks. System testing validates complete workflows, such as verifying the entire appointment scheduling process from input through data persistence. User acceptance testing confirms that software meets end-user needs, such as testing from the perspective of a medical office manager scheduling appointments. Regression testing re-runs existing test suites after code changes to confirm that updates do not introduce new errors. Performance testing measures response times and scalability under load. Although these methods were not used here, understanding their purpose allows me to select the right testing strategy for future projects.

Each testing approach has practical uses depending on the project phase. Unit testing is ideal during development because it provides immediate feedback and isolates logic-level issues before integration. For example, in an API library that processes payments, unit tests could validate individual calculation methods before connecting to external systems. Integration testing becomes critical when multiple modules or microservices interact, such as verifying that an inventory system and payment service coordinate correctly in an e-commerce platform. System testing ensures that complete workflows meet design expectations, while user acceptance testing validates that the product fulfills business needs. Regression testing is essential in agile environments to ensure stability as new features are introduced. If these stages are skipped, issues can accumulate and cause system failures that are difficult to diagnose later (Pressman & Maxim, 2020).

I approached this project with caution because small validation errors can create large cascading failures. For example, I tested deletion operations carefully because a null-pointer exception in a delete method could affect all downstream checks for object existence. Another example was date validation in AppointmentService; a missed condition allowing past dates could lead to corrupted records that would later break sorting logic or business workflows. Appreciating these interdependencies made me more methodical while designing test coverage. I learned that thorough testing is about understanding how errors in one area can propagate through a system.

Testing my own code required constant awareness of bias. It is easy to assume that your logic is correct and unconsciously avoid writing tests that might disprove it. I realized this while testing the update methods in ContactService. Initially, I only verified successful updates because I was confident in my logic. When I forced myself to write failure tests first, I discovered that updating with a null field did not throw an exception as expected. Instead, it silently failed. That discovery reminded me that bias often hides in the "happy path." Writing tests that focused on breaking my code made me more objective and helped reveal subtle flaws.

Discipline in testing is critical to avoiding technical debt. Cutting corners creates problems that multiply over time. For instance, if I had skipped testing null inputs to save thirty minutes, the AppointmentService might have passed initial review but crashed once connected to a mobile UI where users could leave fields blank. The cost of fixing that later would outweigh the time saved. Another example is phone number validation; testing only valid numbers would be easier, but ignoring edge cases like 9-digit or alphanumeric input could break SMS features in production. To avoid this kind of debt, I plan to integrate automated testing into CI/CD pipelines so tests run on every commit, track coverage metrics to detect gaps, and use static analysis tools like SonarQube to maintain code quality. Test-driven development will continue to guide my approach, ensuring that tests define expected behavior before code is written. Quality starts with disciplined habits at the development stage.

**References**

JUnit 5 Documentation. (n.d.). Assertions and test lifecycle methods. https://junit.org/junit5/docs/current/user-guide/

Oracle. (n.d.). Java Platform, Standard Edition 21 API Specification. https://docs.oracle.com/en/java/javase/21/docs/api/

Pressman, R. S., & Maxim, B. R. (2020). *Software Engineering: A Practitioner's Approach* (9th ed.). McGraw-Hill Education.