Danny Forte

CS-320: Software Testing Automation

Project Two: Summary and Reflections Report

Southern New Hampshire University

October 18, 2025

**Alignment to Requirements**

Throughout Project One, I developed and tested three core services—Contact, Task, and Appointment—for a mobile application. My testing strategy was tightly aligned with the functional requirements provided. Each requirement was mapped to one or more JUnit test cases using a traceability matrix. For example, the requirement that “task names must not exceed 20 characters” was validated with a boundary test using the line:

"assertThrows(IllegalArgumentException.class, () -> new Task("ThisNameIsWayTooLong", "Valid description"));" (Forte, 2025).

Similarly, I ensured that contact phone numbers were exactly 10 digits and appointment dates were set in the future. These validations were directly derived from the specification and confirmed through targeted test cases. This alignment ensured that every business rule was verified through automated testing before release.

**Effective Tests**

My JUnit tests covered both positive and negative scenarios across all services. I verified effectiveness by manually tracking which methods and requirements were exercised through my test suite in Codio. To further validate coverage, I integrated the JaCoCo (Java Code Coverage) library by uploading the necessary dependencies and executing a custom command to generate an HTML-based report. This report provided a visual breakdown of coverage across all classes and methods. For example, in the Appointment service, I ensured that dates were in the future using the line: "assertTrue(appointment.getDate().isAfter(LocalDate.now()));" (Forte, 2025).

The JaCoCo report confirmed 93% instruction coverage and 100% branch coverage for the Appointment class, with overall project coverage reaching 87% for instructions and 81% for branches. These metrics demonstrated that my test suite effectively exercised critical paths and validated key business rules.

**Technically Sound Code**

To ensure technically sound code, I followed best practices in test isolation, input validation, and exception handling. Each test class used @BeforeEach to reset state, preventing test interference. For example, I initialized the contact service before each test with the line:

"contactService = new ContactService();" (Forte, 2025).

I also validated constructor behavior with null inputs using:

"assertThrows(IllegalArgumentException.class, () -> new Task("TaskName", null));" (Forte, 2025).

These examples demonstrate how I proactively tested for failure conditions and ensured that the code responded appropriately. I added comments and author tags to support maintainability and academic integrity, and I structured my test classes to mirror the service logic for clarity.

**Efficient Code**

Efficiency was achieved by grouping similar tests using parameterized testing and minimizing redundancy. For instance, I validated multiple invalid inputs with the following line:

"@ParameterizedTest @ValueSource(strings = {"", " ", null})" (Forte, 2025).

This allowed me to test a range of invalid contact names without duplicating test logic. I also reused setup methods and helper functions to streamline test execution. By focusing on reusable patterns and concise assertions, I reduced boilerplate and improved maintainability. My test suite was organized for readability, with logical grouping and consistent naming conventions that made it easy to navigate and extend.

**Reflection**

**Techniques Employed**

In this project, I employed several foundational testing techniques to validate the mobile application's services. These included black-box testing, boundary value analysis, and equivalence partitioning. Black-box testing allowed me to focus on input/output behavior without needing to inspect internal logic, which was ideal for service-layer validation. Boundary value analysis helped me test edge cases such as maximum string lengths and null inputs. For example, I used the line

"assertThrows(IllegalArgumentException.class, () -> new Task("ThisNameIsWayTooLong", "Valid description"));" (Forte, 2025)

to confirm that overly long task names were rejected. Equivalence partitioning allowed me to group valid and invalid inputs efficiently, reducing redundant test cases while maintaining coverage.

**Other Techniques**

While I focused on unit testing and input validation, I also considered—but did not implement—white-box testing and mutation testing. White-box testing involves analyzing internal logic paths and branches, which can be useful for security-critical systems or performance-sensitive modules. Mutation testing introduces small changes to the code to ensure that tests catch those mutations, but it requires specialized tooling and time that were beyond the scope of this project. Exploratory testing, typically used in UI or system-level testing, was also not applicable here but remains valuable in agile environments where documentation may be limited.

**Uses and Implications of Techniques**

Each technique has practical implications depending on the project context. Black-box testing is ideal for validating APIs and services where internal logic is abstracted. Boundary analysis is critical for user-facing applications where input validation is essential. Equivalence partitioning is useful in large-scale systems to reduce test volume while maintaining effectiveness. Mutation testing is best suited for high-risk systems where test robustness must be proven. Exploratory testing is valuable in early-stage development or when requirements are evolving. Understanding when and how to apply these techniques ensures that testing remains both efficient and effective.

**Caution**

I approached testing with caution, treating each test as a safeguard against regressions. For example, when updating the Contact ID format, I retested appointment linkage logic to ensure no breakage occurred. This mindset helped me appreciate the interdependencies across services and the potential ripple effects of even small changes. I also validated that appointments were scheduled for future dates using the line

"assertTrue(appointment.getDate().isAfter(LocalDate.now()));" (Forte, 2025), which prevented logical errors in scheduling.

**Bias**

To limit bias, I adopted a test-first approach, writing failing tests before implementing features. This helped me avoid confirmation bias and ensured that my code was truly robust. For instance, I wrote a test for null task descriptions before coding the constructor using the line

"assertThrows(IllegalArgumentException.class, () -> new Task("TaskName", null));" (Forte, 2025).

By assuming that bugs were present, I was able to uncover edge cases that might have been overlooked if I had assumed the code was correct.

**Discipline**

Discipline was key to maintaining quality throughout the project. I avoided shortcuts like skipping null checks or hardcoding test data. I added comments, author tags, and citations to support maintainability and academic integrity. To avoid technical debt, I plan to use CI tools to automate testing, refactor tests alongside code changes, and maintain at least 80% test coverage as a baseline. These habits will help ensure that my code remains clean, reliable, and scalable in future development environments.

**References**

Ammann, P., & Offutt, J. (2016). Introduction to software testing (2nd ed.). Cambridge University Press.

Forte, D. (2025). JUnit test suite for mobile application services [Java source code]. Southern New Hampshire University.

JaCoCo. (n.d.). Java code coverage library. Retrieved October 18, 2025, from <https://www.jacoco.org/jacoco/trunk/doc/>

JUnit 5 Team. (n.d.). JUnit 5 user guide. Retrieved October 18, 2025, from <https://junit.org/junit5/docs/current/user-guide/>

Oracle. (n.d.). Java documentation. Retrieved October 18, 2025, from <https://docs.oracle.com/en/java/>