**CS 320 Project Two**

Samuel Alexander Black

Southern New Hampshire University

CS-320-11663-M01 Software Test, Automation QA 2025

Professor Jeff Phillips

June 22nd, 2025

Throughout Project One, I implemented unit testing for three key features of the mobile application: contact service, task service, and appointment service. For each feature, I developed a suite of JUnit tests to validate functionality according to the specified requirements. My unit testing approach was methodical and requirement-driven. For the contact service, I ensured all fields such as contact ID, first name, last name, phone number, and address were tested for their constraints—non-nullity and maximum character limits. The task and appointment services were tested similarly to verify the correct handling of creation, updating, and deletion logic. My approach directly aligned with the functional requirements outlined in the project rubric. For instance, the contact ID was confirmed to be immutable and unique through assertions that validated no duplicate IDs could be added, and attempts to modify the ID failed safely.

The effectiveness of the tests was validated by comprehensive test coverage. Each unit test suite covered both valid and invalid inputs, and I confirmed that all major code branches were exercised. My tests covered all CRUD operations and edge cases such as null values, overly long strings, and invalid formats. Although we were not required to submit coverage reports, a manual review confirmed the JUnit tests accounted for all functional paths. This level of coverage supports the confidence that the application could be safely promoted to production.

My experience writing JUnit tests was smooth, largely due to consistent naming conventions and reusable code blocks. I followed a pattern of testAddValidContact(), testAddContactWithNullID(), and so on to make the intent of each test clear. To ensure technical soundness, I used assertions like assertEquals, assertNotNull, and assertThrows. For example, in testUpdatePhoneNumber(), I used assertEquals("5551234567", contact.getPhoneNumber()) to confirm proper data flow. Exception handling was verified using assertThrows(IllegalArgumentException.class, () -> contact.setPhoneNumber(null)), which guaranteed the application responded appropriately to invalid inputs.

To ensure efficiency, I avoided redundant code by using setup methods where possible and focusing on atomic test cases. In my @BeforeEach methods, I initialized common test objects so each test could remain isolated and concise. For instance, instead of repeating contact creation logic in every test, I used a helper method to reduce redundancy. This improved maintainability and readability without sacrificing clarity or coverage.

In terms of software testing techniques, I primarily employed black-box testing and boundary value analysis. Black-box testing allowed me to validate the expected behavior of each method based on input/output, rather than internal implementation. For example, the tests for addTask() and deleteAppointment() evaluated whether the service behaved correctly given a set of inputs, without knowledge of its internal list structure. Boundary value analysis helped verify that string inputs such as contact names were handled correctly at their length limits (e.g., exactly 10 characters vs. 11). I also applied equivalence partitioning by testing representative values within valid and invalid input sets.

Techniques not used in this project included white-box testing, exploratory testing, and automated UI testing. White-box testing, which requires knowledge of internal logic paths, was not applicable since our focus was strictly on unit tests for backend service logic. Exploratory testing, where testers dynamically probe the system based on experience and curiosity, is better suited for more complete software systems with user interfaces or unpredictable behavior. Automated UI testing was irrelevant, as the project did not include a graphical interface. Each of these techniques has value, however. For example, exploratory testing is useful for usability-focused applications, and white-box testing is essential for validating logic-heavy algorithms such as encryption modules.

Adopting a tester’s mindset was critical to the project’s success. I approached each class not just as its developer but as its skeptic. For every method I wrote, I considered how it could fail and how someone else might try to break it. I took extra caution when implementing constraints such as string lengths and null checks, since user input is often the root of software bugs. A specific example was the updateAppointmentDate() method. I ensured that invalid date formats triggered exceptions and that null dates were rejected, because failure to validate date input could lead to downstream scheduling bugs.

Bias was a concern, especially since I wrote both the application logic and its tests. To minimize this bias, I reviewed the test cases independently from the logic I had written and mentally adopted the role of a hostile tester. I wrote tests assuming that the method implementations would fail or be misused. For instance, I attempted to update a task description with an empty string, even though I knew my implementation would likely prevent it. This helped uncover hidden flaws and made my code more robust.

As a future software engineer, I recognize the importance of maintaining discipline in coding and testing practices. Cutting corners—whether by skipping validation checks, avoiding edge-case tests, or leaving code unreviewed—leads directly to technical debt. This debt may not surface immediately, but it inevitably causes long-term instability and higher maintenance costs. To avoid technical debt in the field, I plan to continue enforcing test-driven development practices, adopt continuous integration with automated testing pipelines, and conduct regular code reviews. By treating software quality as a non-negotiable part of the development process, I can contribute to more secure, reliable, and maintainable software.

**References**

Eckstein, R. (2017). *JUnit 5 Basics*. O’Reilly Media.  
Garg, P. (2020). *Effective Unit Testing with JUnit 5*. Springer.

Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson Education.