Jimmy Reynolds

CS-320-11035-M01 Software Test, Automation QA

Southern New Hampshire University

Omar Toledo Lopez

8-12-2025

In Project Two, I developed and tested the ContactService, TaskService, and AppointmentService components of a mobile application using JUnit. Initially, I found it challenging to understand the purpose of JUnit beyond simply “running tests.” However, through hands-on experience and reviewing the course tutorial, I came to appreciate JUnit as a structured framework that allows developers to verify that individual units of code such as methods perform as expected. JUnit helped me confirm that my services met the functional requirements and handled edge cases reliably. It also gave me a way to simulate real-world usage scenarios and catch errors early before integration or deployment.

**Summary**

For each of the three features in the project, I adopted a targeted unit testing strategy that aligned closely with the software requirements. For the ContactService, I focused on validating field-specific updates and input constraints. For example, the test method updateFirstName\_validInput\_success() confirmed that a valid first name updated correctly while other fields remained unchanged. In TaskService, I emphasized status toggling and null input handling. One test, toggleTaskStatus\_validId\_statusUpdated(), ensured that tasks could be marked complete or incomplete reliably. AppointmentService required careful attention to date logic, so I wrote tests like createAppointment\_pastDate\_throwsException() to enforce the rule that appointments must be scheduled in the future.

My testing approach was closely aligned with the software requirements provided by the customer. For instance, the requirement that contact fields reject null or empty values was validated through tests such as updateLastName\_emptyInput\_throwsException(). Similarly, the need for robust task status management was addressed through targeted status toggle tests. The requirement that appointments be future-dated was enforced through exception-based tests that simulated invalid input. These examples demonstrate that my unit tests were not just technical exercises, they were directly tied to business logic and user expectations. I achieved over 95% code coverage across all three services, which I verified using Eclipse’s coverage plugin. This high coverage percentage indicates that my tests exercised close to all conditional paths and public methods. More importantly, the tests were descriptive and focused. For example, updateEmail\_invalidFormat\_throwsException() clearly communicated its purpose and validated that malformed email inputs were rejected. This level of clarity and precision supports the overall quality of the test suite and ensures maintainability for future developers.

Writing the JUnit tests helped me refine both my implementation and my understanding of the domain logic. To ensure technical soundness, I used assertions that validated both the updated field and the unchanged state of other fields. In updatePhoneNumber\_validInput\_success(), for instance, I confirmed that the phone number was updated while the first name remained intact. To improve efficiency, I used parameterized tests to reduce redundancy. One example is updateTaskName\_invalidInputs\_throwsException(), which tested multiple invalid inputs using a single method. This approach minimized boilerplate code and made the test suite easier to maintain.

**Reflection**

Throughout the project, I employed several software testing techniques, including black-box testing, white-box testing, and mocking. Black-box testing focused on verifying inputs and outputs without considering internal logic, which was particularly useful for user-facing features. White-box testing allowed me to examine internal logic paths and ensure that all branches were covered, which was essential for complex validation and processing logic. Mocking was used to isolate units from external dependencies, enabling me to test service behavior independently of database or API responses. There were also techniques I did not use, such as exploratory testing, mutation testing, and integration testing. Exploratory testing, which involves manual and unscripted testing, was not applicable in this automated unit testing context. Mutation testing, which introduces small changes to code to test the robustness of the test suite, could have added rigor but was outside the scope of this project. Integration testing, which verifies interactions between modules, was deferred to later stages of development.

Each of these techniques has practical implications depending on the project. Black-box testing is ideal for validating user inputs and outputs, while white-box testing is crucial for ensuring internal logic correctness. Mocking is indispensable when testing services that rely on external systems. Mutation testing is particularly valuable in safety-critical applications where test robustness is paramount, and integration testing is essential for verifying end-to-end workflows in larger systems.

My mindset throughout the project was cautious and detail oriented. I recognized the importance of understanding the complexity and interrelationships within the codebase. For example, validating null inputs in the validate() method prevented downstream crashes and ensured system stability. I also made a conscious effort to limit bias by reviewing my code after breaks and seeking peer feedback. Bias can be a concern when developers test their own code, as they may overlook edge cases or assume correctness. Initially, I missed testing for empty strings until a peer pointed it out, which highlighted the value of external review.

Finally, I believe that discipline and a commitment to quality are essential traits for any software engineering professional. Cutting corners in writing or testing code can lead to technical debt, which accumulates over time and becomes costly to resolve. To avoid this, I plan to use continuous integration tools like Jenkins or GitHub Actions to enforce test coverage thresholds and automate quality checks. For example, I have configured GitHub Actions to fail builds if coverage drops below 85%, ensuring that quality remains a priority throughout development.