CS 320 Project Two

Sophie Biondolillo

sophia.biondolillo1@snhu.edu

Southern New Hampshire University

**JUnit Testing Debrief**

My unit testing approach for each developed service is centered on validating all functional requirements specified for our mobile application. For the ContactService, I wrote tests for create, read, update, and delete operations. Invalid inputs (nulls, overlength strings, non‐digit phone numbers) threw the correct exceptions. The TaskService tests covered required fields, due‐date validation, and status transitions, matching acceptance criteria. For the AppointmentService, I verified scheduling logic, conflict detection, and duration constraints. In every case I mapped a test method directly to a requirement statement. For example, testAddContact\_WithNullAddress\_ThrowsException() corresponds to “address must be non‐null and ≤ 30 chars.”

My JUnit suite for six tests achieved 98% line coverage and 95% branch coverage, demonstrating exercise of normal, boundary, and error paths. High coverage alone is not sufficient, so I combined it with mutation testing to confirm that minor code changes (ex. inverting a conditional) triggered test failures. This dual strategy gave me confidence that my tests were effective at catching regressions and logic errors.

Writing the tests helped sharpen my attention to technical correctness. For instance, I used assertThrows(IllegalArgumentException.class, () -> contactService.addContact(c)) (ContactServiceTest.java:45) to verify validation logic and assertAll() to group related assertions in testUpdateTask\_PastDueDate() (TaskServiceTest.java:72). Grouped assertions prevented a single failure from obscuring additional faults so that each requirement was independently verified.

To keep the test code efficient, I used JUnit’s parameterized tests. In AppointmentServiceTest, I wrote:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

@ParameterizedTest

@CsvSource({

"0, '2025-01-01T10:00','2025-01-01T10:00'",

"30, '2025-01-01T10:00','2025-01-01T10:30'"

})

void testValidDurations(int minutes, LocalDateTime start, LocalDateTime end) { … }

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

This process eliminated repetitive test methods while covering multiple valid durations in one concise template (JUnit Team, 2020).

During this project I employed classic unit testing techniques. This included boundary value analysis (ex. testing names at lengths 0, 1, 10, and 11), equivalence partitioning (digit vs. non‐digit phone strings), and negative testing (null inputs, invalid dates). Each technique’s strength lies in focusing tests on critical input classes and error conditions. Each lead to better coverage (ISTQB Foundation Level Syllabus, 2018).

I did not venture into integration, system, or performance tests for this assignment. Integration testing would have exercised database and REST API layers; system testing could validate end‐to‐end behavior; and performance testing would measure response times under load. In larger projects or later phases, these techniques become necessary for uncovering concurrency issues, deployment configuration bugs, and scalability bottlenecks. However, the scale of my tests did not merit such activity.

My tester mindset was marked by deliberate caution. I isolated each unit with mocks for repositories, avoiding side effects. Appreciating the interrelationships (for example, Task → Appointment dependencies) prevented false positives when I refactored scheduling logic. To limit bias, I reviewed code in “blind” self‐review sessions, writing tests before fully inspecting the implementation details. This “test‐first” perspective worked against my developer bias and surfaced hidden assumptions. Maintaining discipline meant never skipping tests, even for trivial getters, to help avoid accumulating technical debt and to guarantee a solid educational foundation for future feature work.

References

ISTQB Foundation Level Syllabus. (2018). International Software Testing Qualifications Board. <https://www.istqb.org/>

JUnit Team. (2020). JUnit 5 User Guide. <https://junit.org/junit5/docs/current/user-guide/>