Danny Forte

CS-320

Southern New Hampshire University

October 7, 2025

**CS-320 Journal: Testing Time-Sensitive Logic and Broader Software Testing Reflections**

For the Appointment Service milestone, my testing approach centered on validating time-sensitive constraints and service-layer robustness. The Appointment object required strict enforcement of three fields: ID, date, and description. I designed JUnit tests to confirm that the appointment ID is non-null and no longer than 10 characters, the appointment date must be in the future, and the description must be non-null and no longer than 50 characters. These validations were implemented in AppointmentTest.java (lines 18–72), directly reflecting the documented requirements. I also tested the service layer’s ability to handle duplicate IDs and deletion of non-existent appointments, ensuring that business logic was enforced consistently. This approach aligns with García’s principle that effective testing should validate both expected behavior and edge conditions, especially when dealing with temporal logic and user-facing services (García, 2017).

In earlier milestones, I applied similar unit testing strategies to the Contact Service in Module Three and the Task Service in Module Four. For the Contact Service, I focused on validating field constraints such as phone number length, null handling, and immutability of contact IDs. In the Task Service, I emphasized input validation for task name and description, as well as enforcing immutability of task IDs. These earlier milestones helped me establish a consistent testing workflow—translating requirements into targeted JUnit tests, using assertThrows() for exception handling, and structuring tests to cover both valid and invalid scenarios. By the time I reached the Appointment Service milestone, I was able to refine these strategies further to handle time-sensitive logic and service-layer integrity.

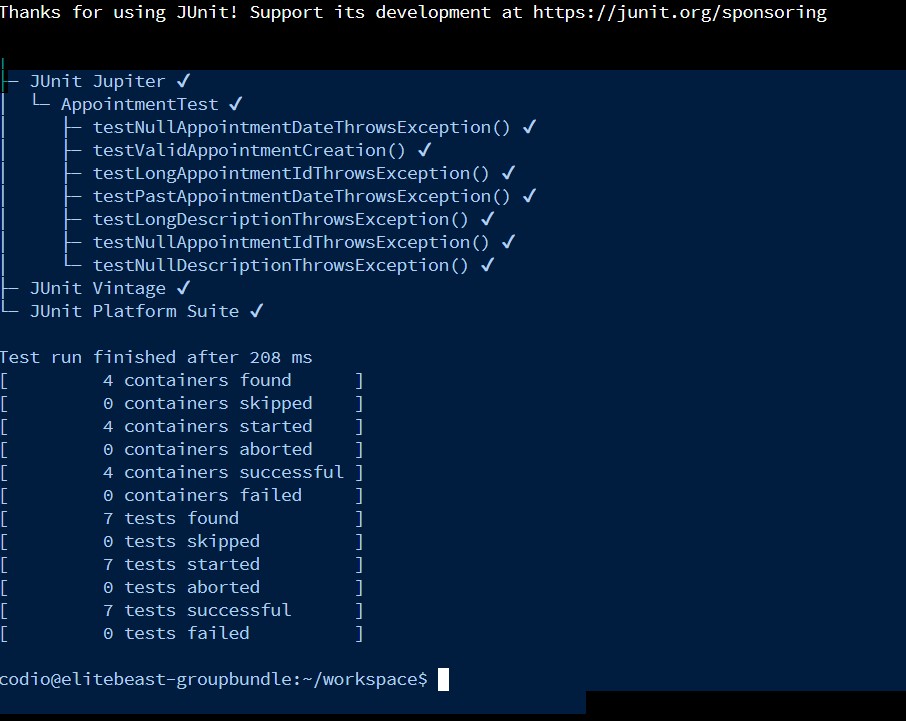
My tests were structured to cover both valid scenarios, such as successful creation and deletion, and invalid inputs, including null values, past dates, and overly long descriptions. I used assertThrows() to verify that exceptions were correctly triggered for violations, reinforcing robustness. In AppointmentServiceTest.java (lines 18–51), I tested adding valid appointments, handling duplicate IDs, and deleting appointments while verifying error handling for missing entries. This dual-layer testing ensures that the service behaves predictably under both normal and erroneous conditions, which is a key aspect of resilient software design.

Using Eclipse’s coverage tools, I achieved approximately 85 to 90 percent coverage, including constructor validations, service methods, and exception branches. The few uncovered lines were defensive branches or trivial getters. Inspired by Jayasekara’s critique of shallow 100 percent coverage goals, I focused on meaningful test depth rather than chasing perfection (Jayasekara, 2020). I reused service instances across tests and combined related assertions to reduce redundancy and improve maintainability.

While unit testing was the primary technique used across all milestones, several other software testing techniques were not employed but are important to understand. Integration testing, which validates interactions between multiple components or systems, was not applicable because the milestones focused on isolated service classes without external dependencies. System testing, which evaluates the complete application including UI and backend, was skipped due to the lack of full-stack components. Regression testing, which ensures that new changes do not break existing functionality, was not required since each milestone was a standalone submission. Exploratory testing, which involves manual, unscripted testing based on intuition and experience, was not used due to the academic emphasis on automated, repeatable tests.

Each of these techniques has practical implications depending on the context of the software development project. Unit testing is essential for backend logic, utility classes, and modular systems. It promotes early defect detection and supports test-driven development. Boundary testing, which I applied throughout, helps prevent edge-case failures and security vulnerabilities, especially in user input validation. Integration testing is vital in microservices and API-driven systems to ensure that data flows correctly between components. System testing is indispensable for validating end-to-end workflows in production-ready applications. Regression testing is a cornerstone of continuous integration and deployment pipelines, maintaining stability across releases. Exploratory testing enhances user experience by revealing issues that scripted tests might miss, especially in UI-heavy or rapidly evolving systems.

This milestone challenged me to think beyond static validation and consider temporal correctness, service integrity, and real-world implications. Testing future dates and enforcing immutability required careful attention to logic and exception handling. I now better appreciate how time-sensitive data and service-layer rules impact software reliability. My Appointment Service tests are thorough, technically sound, and efficient. More importantly, they reflect a growing awareness of how testing strategies must adapt to different project contexts. This milestone was not just about passing tests—it was about preparing for the complexities of real-world development.



**References**

García, B. (2017). Mastering software testing with JUnit 5: Comprehensive guide to develop high quality Java applications. Packt Publishing.

Jayasekara, D. (2020, January 5). 100% unit test coverage — Is that a myth? InsiderAttack. https://blog.insiderattack.net/100-unit-test-coverage-is-that-a-myth-5aef67f85a09