CS-320: Software Test, Automation QA

16 August 2025

Professor Butu

**Grand Strand Systems**

As a part of my work at Grand Strand Systems, I just finished Project One, which required creating and testing backend services for a mobile application. The services were ContactService, TaskService, and AppointmentService. My unit testing strategy centered on isolating each method within these services and confirming its behavior under both standard and edge-case scenarios. I used JUnit 5 to arrange the tests, making sure that each test case was independent, repeatable, and in line with the program requirements. For ContactService, I evaluated methods for adding, updating, and deleting contacts, as well as checking for constraints like non-null fields and unique contact IDs. In TaskService, I worked on task creation, deadline validation, and status updates, as well as boundary tests for date inputs and string lengths. AppointmentService required scheduling logic validation, including time conflicts, as well as proper date-time string formatting.

My testing strategy was closely aligned with the software requirements. For example, the requirement that contact names not exceed 10 characters was enforced using assertions that threw exceptions when the limit was reached. This alignment was obvious in test cases like assertThrows(IllegalArgumentException.class, () -> service.updateContactName("12345678901"));, which confirmed constraint compliance and avoided runtime issues. My JUnit tests' general quality was validated by strong coverage percentages, which averaged more than 90% for all three services. I utilized IntelliJ's built-in coverage tools to ensure that all branches and conditions were tested. For example, in TaskServiceTest, I added tests like assertEquals("Complete report", task.getDescription()); to ensure that the setter and getter functions worked as intended.

Writing the JUnit tests was a systematic yet gratifying experience. I started by assigning each requirement to a test case and then increased coverage by include negative tests and edge cases. To verify technical soundness, I utilized unambiguous naming conventions and setup methods, such as @BeforeEach void setUp() { service = new TaskService(); }, keeping the tests simple and isolated. To preserve efficiency, redundant assertions were avoided and relevant tests were logically grouped. For example, a single test, testAddTaskAndRetrieve(), validated both addition and retrieval, eliminating overhead and increasing clarity.

In terms of testing techniques, I primarily employed unit testing and boundary testing. Unit testing allowed me to isolate and validate individual methods, while boundary testing ensured that inputs at the edge of acceptable ranges were handled correctly. Techniques I did not use included integration testing, system testing, and regression testing. Integration testing, which verifies interactions between services or with external systems, was not applicable in this project. System testing, which ensures the application meets user expectations through end-to-end scenarios, was outside the scope. Regression testing, which automates checks for previously fixed bugs, was not implemented but remains a vital practice for future iterations.

Each technique has a practical application based on the project. Unit testing is helpful for early-stage development for detecting logical problems. Integration testing is critical when services rely on APIs or databases. System testing guarantees that the application performs as intended from the user's perspective, while regression testing ensures stability during upgrades. Understanding when and how to use these strategies is critical for developing trustworthy software.

Throughout the process, I maintained a careful and analytical approach. I viewed each approach as a potential source of failure and recognized the interdependence of services. For example, in AppointmentService, I evaluated how overlapping appointments would influence the user experience and included tests to avoid scheduling conflicts. To avoid prejudice, I analyzed my code as if it had been created by someone else, questioning assumptions and validating edge cases. One example was to include a test to ensure that task descriptions were not empty: assertThrows(IllegalArgumentException.class, () -> service.addTask("002", "", LocalDate.now())). This helped me identify neglected cases and enhance robustness.

Software engineering requires strict quality control. Cutting corners during testing can result in flaws that are costly to address later. To avoid technological debt, I intend to thoroughly document test cases, employ automated coverage monitoring tools, and rewrite tests as code evolves. By adhering to these practices, I hope to maintain high standards while delivering stable, maintainable software. This project emphasized the necessity of rigorous testing, smart design, and a quality and accountability-oriented mindset.

**References:**

1. GeeksforGeeks. (2025, July 26). *Software testing techniques*. Retrieved from <https://www.geeksforgeeks.org/software-testing/software-testing-techniques/>
2. BrowserStack. (2025, July 7). *Software testing techniques: Explained with examples*. Retrieved from <https://www.browserstack.com/guide/software-testing-techniques>
3. Luo, L. (n.d.). *Software testing techniques: Technology maturation and research strategies*. Carnegie Mellon University. Retrieved from <https://www.cs.cmu.edu/~luluo/Courses/17939Report.pdf>
4. IBM. (n.d.). *Unit testing best practices*. Retrieved from <https://www.ibm.com/think/insights/unit-testing-best-practices>
5. AlgoCademy. (2025). *The importance of unit testing in software development: A comprehensive guide*. Retrieved from <https://algocademy.com/blog/the-importance-of-unit-testing-in-software-development-a-comprehensive-guide/>
6. Scrum Alliance. (n.d.). *Avoiding technical debt with these "Core Four" practices*. Retrieved from <https://resources.scrumalliance.org/Article/avoiding-technical-debt-core-four-practices>
7. Built In. (2024, October 29). *10 ways to minimize technical debt*. Retrieved from <https://builtin.com/articles/tips-to-manage-technical-debt>
8. JUnit 5 User Guide. (n.d.). Retrieved from https://junit.org/junit5/docs/current/user-guide/ Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley.