Kyle Smith

CS-320 Software Test, Automation

7-2 Project Two

4/14/25

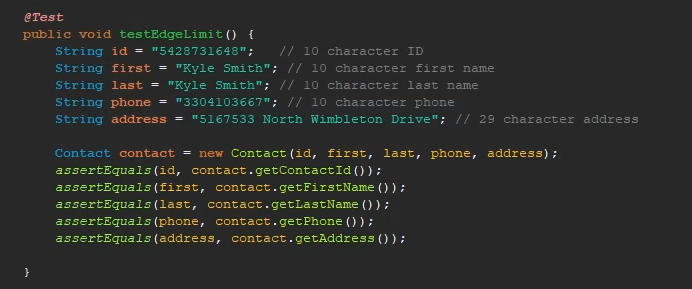
As part of the final project for Grand Strand Systems, the Contact, Appointment and Task services were designed and tested using a structured unit testing strategy. JUnit, a widely adopted Java testing framework, was used to ensure the correctness and reliability of the methods within each service class. The unit testing approach for each feature was closely aligned to the given software requirements, and the effectiveness of the tests was supported by high code coverage percentages across all services.

For the Contact Service, JUnit tests were designed to validate the creation of new contact entries and to ensure that constraints such as non-null values and character limits were respected. For example, the testInvalidContactCreation() ensures that each field is a non-null input, aligning with one of the key software requirements. The coverage report showed 81.6%, demonstrating a strong test suite that validated both expected behaviors and boundary conditions.

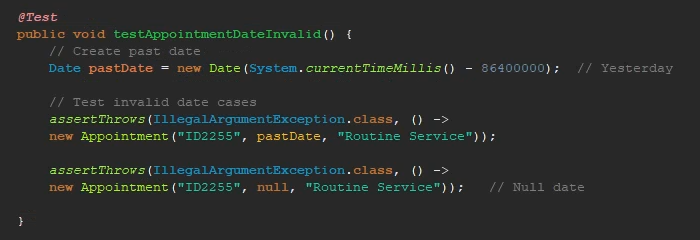
For the Appointment Service, the testing approach focused on validating appointment creation with future dates, description limits and unique identifiers. The AppointmentTest.java file reached 80.6% coverage, with tests such as testDescriptionInvalid() confirming that invalid input is correctly rejected. Another example is testDateInvalid(), which ensures that past-dated appointments are properly rejected, verifying that time-sensitive validation rules are enforced. The AppointmentServiceTest.java file added further coverage by testing operations like adding valid appointments, rejecting duplicates, handling deletions and checking for null inputs. These tests provided high confidence in the core logic, especially around time-sensitive fields like dates and proper error handling.

For the Task Service, tests were written to validate ID constraints, non-null names and descriptions and correct behavior when removing or updating tasks. The TaskTest.java file achieved 81.9% coverage, verifying the construction of valid tasks and input validation in setter methods. The TaskServiceTest.java file reached 86.1% coverage, with tests ensuring tasks could be added, updated and deleted properly. It also confirmed that duplicate or unknown task IDs triggered the expected exceptions. One important case is testSetNameNull(), which confirms that a null name input during task update triggers an exception, helping preserve data integrity. These tests ensured full coverage of the task management.

In writing JUnit tests, care was taken to write technically sound code by enforcing strong validation logic and avoiding assumptions about user input. For instance, in the ContactTest class, the test:



Shows how assertEquals was used to verify that edge-limit values were correctly handled. This helps confirm that the maximum allowed inputs are accepted, preventing off-by-one errors. These kinds of subtle issues can be overlooked without boundary testing, and they often lead to bugs that are hard to detect later. Another example of technical soundness can be seen in AppointmentTest.java, where testAppointmentDateInvalid() uses assertThrows to confirm that an appointment set in the past throws an exception. This line ensures time-sensitive validation logic is enforced, which is critical in real-world scheduling systems. Efficiency was addressed by keeping test cases focused, modular and independent, avoiding unnecessary setup and making sure each test had a clear purpose. Additionally, using literal values that clearly demonstrate edge limits improved the clarity and maintainability of the tests.



The primary test technique that was used was unit testing, which isolates individual methods to confirm their correctness. JUnit’s @Test annotations and assertions (assertEquals, assertThrows) were used to ensure methods behaved as expected under both normal and exceptional conditions. For example, assertThrows was used in both Task and Appointment services to confirm that invalid inputs, such as null values or past dates, were handled by throwing the appropriate exceptions. This technique is precise, fast to execute and ideal for the logic-heavy services implemented. Boundary value analysis was also used, particularly for string lengths, null validations and date constraints. This was critical in all three services, as requirements included strict lengths and formatting rules. Other testing strategies such as integration testing and system testing were not used in this project but are worth mentioning. Integration testing checks the interactions between modules, while system testing validates the software as a whole in its operational environment. These would be more relevant in a complete application that integrates a user interface or database. Unit testing is particularly effective during development and for test-driven development. It helps catch issues early and encourages clean, modular code. Integration testing is critical when multiple components interact, while system testing is often done before release to end users to ensure overall functionality. Choosing the right technique depends on the stage of development and the complexity of the system.

Working on this project, I approached the work with a cautious mindset. I recognized that errors in a simple field validation could spread through the application, so I ensured each input constraint was rigorously tested. For example, the TaskService’s refusal to add a null-named task was tested to prevent data corruption. I also made an effort to limit bias by writing tests before completing the service logic. This prevented me from validating my own assumptions. If I had only tested valid paths, I would have overlooked important failure conditions like invalid dates in the Appointment Service. Finally, discipline was central to my work. Cutting corners on testing could have resulted in bug-prone code. Instead, I stayed committed to validating every requirement, and I maintained a consistent naming convention and structure across all JUnit tests. As a future software engineer, I plan to avoid technical mistakes by continuing to use automated tests, maintaining documentation and revisiting older code for editing when necessary.

This project reinforced the importance of thorough unit testing and a disciplined mindset in software development, Through JUnit testing, I was able to meet software requirements, deliver high-quality services and adopt industry best practices. The experience also gave me hands-on exposure to how different test strategies fit into the software lifecycle and how automation can serve as a backbone for scalable, reliable development. The experience has strengthened my understanding of how different testing strategies can be applied depending on the context, and it has prepared me to write reliable, maintainable code in professional environments.

**Sources**

BrowserStack. (2024, October 29). *Software testing techniques: Explained with examples*. <https://www.browserstack.com/guide/software-testing-techniques>

Hambling, B. (2019). *Software testing: An ISTQB-BCS certified tester foundation guide*. BCS, The Chartered Institute for IT. [https://ebookcentral.proquest.com/lib/snhu-ebooks/detail.action?docID=5837074#](https://ebookcentral.proquest.com/lib/snhu-ebooks/detail.action?docID=5837074)

Stefan Bechtold, S. B. (n.d.). JUnit 5 user guide. <https://junit.org/junit5/docs/current/user-guide/>