As a software engineer at Grand Strand Systems, I worked on developing and testing core services for a mobile application. These include ContactService, TaskService, and AppointmentService. Each one was responsible for handling a distinct kind of object. The project required attention to object constraints, CRUD functionality, and reliable validation. This report provides an overview of my testing approach and my experience writing JUnit tests. It also describes the strategies and reasoning I used during the project to ensure high-quality software.

My testing approach began with an analysis of the assignment requirements. Each class had specific constraints. The Contact class was limited to a 10-character ID, required a phone number of exactly 10 digits, and an address that could not be blank and was capped at 30 characters. The Task class required a unique ID no longer than 10 characters, a non-null name with a maximum of 20 characters, and a non-null description limited to 50 characters. The Appointment class required a unique ID no longer than 10 characters, a non-null description limited to 50 characters, and a date that must be in the future. I implemented a set of JUnit tests for each class and its corresponding service to verify correct behavior in normal cases. To test for improper inputs, such as null values or data that exceeded length limits, I followed the principle that “a unit test should be a small, isolated piece of code that verifies a single aspect of program behavior” (Appel, 2015, p. 22). These tests were directly aligned with the project’s guidelines. For example, I used assertThrows to ensure that invalid values correctly triggered exceptions, such as when a Task ID exceeded the 10-character maximum. I also ensured that service classes like ContactService and AppointmentService properly handled adding, updating, and deleting objects by ID, with tests that covered both valid and invalid operations.

The overall effectiveness of my JUnit tests was supported by strong coverage. All six test classes, two for each service, consistently produced coverage above 90 percent. This ensures that both normal and edge cases are verified. In the AppointmentServiceTest file, I tested not only future appointments but also ensured the class rejected past dates by creating a date in the past and verifying that the constructor threw an exception. This example, among others, shows how the tests enforced real-world rules embedded in the application logic. The consistent use of both positive and negative test cases helped me confirm that the application would behave reliably across a range of scenarios.

Writing the JUnit tests gave me valuable experience in designing technical and efficient test code. I utilized reusable values, logical structure, and setup methods to eliminate redundancy. For instance, to ensure a consistent future date in AppointmentTest, I declared a single date value and reused it across multiple test cases rather than creating a new date in each method. This approach minimized duplication and improved readability across the test suite. I also used clear and descriptive method names such as testUpdatePhoneFailsWithInvalidLength, which made the tests easier to understand and maintain. These small but intentional strategies contributed to both the technical quality and efficiency of the code.

For this project, I applied multiple testing methods, with unit testing and boundary value analysis being the most significant. Through unit testing, I verified each class’s functionality on its own, without interference from other components. Boundary value analysis helped me validate that fields like contact ID or phone number correctly enforced maximum lengths and format constraints, such as testing both exactly 10 digits and 11 digits for phone numbers. This aligns with research noting that “boundary value analysis remains a fundamental approach for verifying input constraints, especially when combined with automated test case generation” (Amarif, Alfitouri, & Allag, 2023, p. 313). Which I did not utilize techniques like integration testing, system testing, or mocking in this project. I recognize their importance in larger or more interconnected systems. For example, integration testing is useful for verifying that different service components interact correctly, while mocking is essential when simulating database or API behavior without actual dependencies.

Understanding when and how to use these techniques is key to professional development. Unit testing is most effective for checking logic within isolated classes. In contrast, integration and system testing are better suited for full-stack applications or systems with multiple services interacting together. In a production setting, applying the right strategy helps catch different classes of errors early and ensures robust quality at all levels of the application.

As both a developer and a tester on this project, I was being cautious and thorough. I avoided making assumptions about what would work and instead tried to break the code intentionally through malformed input. For example, I tested how ContactService behaved when asked to delete a nonexistent ID or update a contact that had not been added. These tests confirmed that error handling was functioning correctly and that the application would not crash under invalid operations. I also worked to avoid bias in my review by writing test cases that intentionally challenged my own assumptions. Instead of only testing the “happy path,” I included negative cases such as too-short phone numbers to validate that my code handled unexpected input gracefully.

This project reinforced the importance of discipline and commitment to quality in software engineering. It is easy to cut corners when tests seem repetitive or when the code already appears to be working. However, skipping thorough testing leads to technical debt problems that grow more difficult and expensive to fix later. To avoid this, I made sure every field validation and service method was covered by meaningful tests. As I continue my development career, I plan to maintain this mindset by practicing test-driven development and reviewing code. I will also use automated testing tools as part of continuous integration pipelines.

In conclusion, this project was an important opportunity to apply unit testing in a real-world context. I have worked with Java before, but this was my first time using JUnit testing. This broadened my knowledge of new aspects of the language. It strengthened my understanding of JUnit and quality assurance principles. As recent studies have noted, leveraging frameworks like JUnit with advanced methods such as genetic algorithms can “improve coverage and reduce manual effort in test design” (Amarif et al., 2023, p. 314). This reinforces the value of structured, methodical testing. Through test design and reflection, I was able to validate that the services worked as intended. I aim to carry forward these habits into my professional development as a software engineer.

**References**

Appel, F. (2015). *Testing with JUnit: Master high-quality software development driven by unit tests*. Packt Publishing.

Amarif, M., Alfitouri, A., & Allag, Z. (2023). An automatic generation of test cases in JUnit testing framework using genetic algorithm. *2023 IEEE 11th International Conference on Systems and Control (ICSC)*, 312–317. https://doi.org/10.1109/ICSC58660.2023.10449836