# Grand Strand Systems Mobile Application Technical Retrospective

Zachariah Spencer

zachariah.spencer@snhu.edu

Southern New Hampshire University

During the development of the Grand Strand Systems mobile application, I implemented unit testing for three core features: task management, contact management, and appointment scheduling. My unit testing approach was designed to align closely with the software requirements by ensuring that all constraints and expected behaviors were thoroughly validated. For example, in testing the task management functionality, I created tests that verified whether task IDs, names, and descriptions adhered to the required constraints. Since task IDs needed to be unique and not exceed a certain length, I wrote tests that provided both valid and invalid inputs to confirm that the system handled them correctly. A similar approach was used for contact management, where I ensured that first names, last names, phone numbers, and addresses met their respective constraints. These tests helped validate that the system would reject incorrect data while allowing valid entries to pass.

The effectiveness of my JUnit tests can be measured by their ability to catch errors and the level of code coverage achieved. I ensured that my tests covered all critical functions, including object creation, validation, and service-layer operations such as adding, updating, and deleting records. For instance, in testing the task service, I confirmed that adding a task with an existing ID would result in a failure, preventing duplicate entries. Similarly, my tests verified that once a contact was deleted, retrieving it would return no results. By ensuring that different scenarios were accounted for, I could be confident that the application functioned as expected in various situations. High code coverage also indicated that most methods and condition branches were tested, reducing the risk of undetected bugs.

Writing JUnit tests required careful attention to detail to ensure technical accuracy. To maintain correctness, I structured assertions to validate expected outputs and included exception handling to confirm that invalid inputs triggered appropriate errors. For example, I wrote test cases that checked whether providing a null value for a required field would result in an exception, ensuring that the system enforced data integrity rules. Additionally, I focused on efficiency by avoiding redundant object creation and unnecessary computations. Instead of repeatedly initializing test objects in multiple test cases, I set up a reusable test object before each test execution to maintain consistency and improve efficiency.

For this project, I primarily used unit testing and black-box testing. Unit testing focused on verifying the behavior of individual methods in isolation, ensuring that each component worked as expected without relying on other parts of the system. Black-box testing allowed me to validate inputs and outputs without considering the internal workings of the code, making it useful for checking system behavior from a user’s perspective. Additionally, I used parameterized testing, which allowed me to test multiple input scenarios with a single test method, improving efficiency. However, some other testing techniques were not used in this project, such as integration testing and system testing. Integration testing would have been beneficial for verifying how different components interacted with each other, while system testing would have evaluated the entire application’s behavior as a whole. Although these techniques were not part of the unit testing phase, they are essential in ensuring that a fully integrated system functions correctly.

Each testing technique serves different practical purposes depending on the project. Unit testing is ideal for ensuring that individual components work correctly before integration, making it particularly useful for modular applications. Black-box testing is commonly used when validating APIs or user interfaces where internal implementation details are abstracted. Integration testing is essential for applications with multiple interacting components, such as systems that rely on external databases or services. System testing, on the other hand, is crucial for verifying overall application behavior before deployment to catch any potential issues that may arise in real-world use.

Throughout the project, I maintained a cautious approach to testing, understanding that code complexity often leads to unexpected interactions. Recognizing how changes in one part of the application could affect others was crucial to ensuring stability. For example, when implementing task deletion, I verified that attempting to retrieve a deleted task would return no result, ensuring that data was properly removed. This prevented issues where deleted items might still appear due to caching or improper reference handling.

To minimize bias in reviewing my own code, I assumed that errors existed rather than expecting correctness by default. By taking this mindset, I was able to approach testing with a critical perspective and avoid overlooking potential issues. If I had only tested my own code with the expectation that it worked correctly, I might have unintentionally ignored edge cases or subtle logic errors. This highlights the importance of independent testing, where someone unfamiliar with the code can identify issues that the original developer might overlook.

Maintaining discipline in software quality is crucial for avoiding technical debt and ensuring long-term maintainability. Cutting corners during testing can lead to unstable software that requires extensive fixes later. To prevent this, I plan to adopt test-driven development practices, writing tests before implementing functionality to ensure that requirements are met from the start. Additionally, I will use automated testing tools to catch errors early and conduct regular code reviews to identify potential improvements. By maintaining a strong commitment to quality, I can contribute to building reliable and maintainable software while reducing future maintenance costs.

**References**

Garcia, B. (2017). *Mastering JUnit 5*. Packt Publishing.

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing: An ISTQB-BCS certified tester foundation guide* (4th ed.). BCS Learning & Development Limited.