**Grand Strand Systems: Unit Testing Summary and Reflection Report**

**Justin Turner**

**Southern New Hampshire University**

**CS 320: Software Test Automation & QA**

**October 2025**

### **Unit Testing Approach**

During the development of the mobile application for Grand Strand Systems, I implemented a structured and requirement-driven unit testing strategy for all three major services: Contact, Task, and Appointment. Each service was tested using JUnit 5 to ensure validation of rules, exception handling, and logical operations functioned as intended.

For the ContactService, tests such as testAddJustinContact() and testPreventDuplicateContactIds() verified that new contacts could be added successfully while preventing duplicate IDs. The testDeleteNonExistentContact() and testUpdateJustinContactInformation() methods confirmed correct behavior when removing or updating existing records. These tests align with requirements such as enforcing unique identifiers, validating phone formats, and limiting name and address lengths. For example, in ContactServiceTest.java, line 83 throws an exception when a duplicate ID is added, directly enforcing system requirements.

For the TaskService, tests were written to validate creation, updates, and deletions of tasks. The testAddTaskWithDuplicateId() method ensures each task ID remains unique, while testUpdateTaskNameTooLong() and testUpdateTaskDescriptionTooLong() confirm that field length constraints (20 and 50 characters, respectively) are enforced. The testCompleteWorkflow() method demonstrates comprehensive functionality by adding, updating, and deleting a task within one sequence, verifying proper integration between all service operations.

The AppointmentService tests ensured time-sensitive and descriptive integrity of appointments. The testAddAppointmentPastDate() method confirmed that appointments scheduled in the past were rejected, while testAddAppointmentDescriptionTooLong() checked input constraints. The overall logic aligns perfectly with requirements that mandate unique appointment IDs, non-null fields, and future dates only.

#### **Alignment With Requirements**

Each testing approach directly reflects the project's requirements. The constraints on ID length, non-null fields, and maximum character limits were validated through a mixture of positive (successful creation) and negative (invalid data rejection) tests. For instance, the TaskTest.java class includes testTaskDescriptionTooLong() and testTaskNameCannotBeNull(), providing direct enforcement of business rules from the requirements document. By writing both success and failure cases, the testing approach ensures complete coverage and compliance.

#### **Overall Quality of JUnit Tests**

I measured quality primarily through code coverage and assertion of diversity. The test suits all three modules achieved near-total line and branch coverage by validating valid and invalid inputs, boundary conditions, and exception cases. Assertions such as assertThrows(), assertEquals(), and assertFalse() ensure comprehensive validation. For example, in TaskServiceTest.java line 67, assertThrows(IllegalArgumentException.class, ...) confirms proper exception handling for duplicate tasks, strengthening test reliability.

#### **Experience Writing JUnit Tests**

Developing these tests helped solidify my understanding of modular and testable code design. I ensured technical soundness by validating each function’s preconditions and postconditions. For instance, in ContactServiceTest.java, initialization in the @BeforeEach method created a clean testing environment before every test. Efficiency was maintained by using reusable test data, minimizing redundancy, and isolating tests to focus on single functions at a time. This design pattern improved clarity and reduced potential interference between tests.

### **Reflection**

#### **Testing Techniques**

Throughout the project, I primarily employed black-box testing, boundary value analysis, and negative testing. Black-box testing was used to validate outputs against expected results without inspecting the internal logic. Boundary testing ensured that edge cases (such as maximum character limits or null fields) were properly handled. Negative testing was vital in confirming that invalid data triggered exceptions as expected.

Techniques not employed included integration testing, system testing, and acceptance testing. These methods would be more appropriate once multiple modules or user interfaces are connected. For example, integration testing could verify how the Contact and Appointment modules interact when deployed in the full mobile application environment.

Each technique has practical applications. Black-box testing is ideal for validating requirements in isolation, while boundary value testing is crucial in domains where constraints are strict, such as financial systems or safety-critical software. Meanwhile, integration and acceptance testing are essential for real-world deployment readiness.

#### **Mindset**

Adopting a software tester’s mindset requires caution and objectivity. I approached each feature as if attempting to break it, simulating how users might enter invalid or unexpected data. Appreciating the complexity and interrelationships of the code under test was critical; for example, understanding that updating a contact phone number should not impact unrelated contact fields.

Limiting bias was another focus. Testing my own code requires separating developer's intent from test validation. I achieved this by writing tests before completing implementation whenever possible (a light version of test-driven development). For example, by predefining failure conditions for duplicate contact IDs, I avoided overlooking this critical requirement later.

Maintaining discipline in testing is fundamental to software quality. Real-world failures such as the Ariane 5 explosion (European Space Agency, 1996) underscore the cost of inadequate validation in that case; a numeric overflow error led to a $500 million loss. Similarly, Apple’s premature release of Apple Maps without thorough validation demonstrated how incomplete testing damages credibility (When Coding Goes Wrong, n.d.). Finally, the Chinook helicopter disaster illustrated in Clear and Present Danger: Why We Refused to Give Up highlights the ethical responsibility developers hold in ensuring safety through rigorous testing. These examples reaffirm that quality is not optional, it is a professional obligation.

To avoid technical debt as a practitioner, I plan to continue emphasizing unit test coverage, code reviews, and documentation before feature delivery. By not cutting corners and maintaining automated testing as part of continuous integration, I can ensure both immediate reliability and long-term maintainability.

### **References**

European Space Agency. (1996). *The explosion of Ariane 5.* Retrieved from Southern New Hampshire University Shapiro Library.

Microsoft. (n.d.). *When coding goes wrong.* Retrieved from Southern New Hampshire University Shapiro Library.

Shapiro Library. (n.d.). *Clear and present danger: Why* *we refused to give* *up.* Southern New Hampshire University.