**7-2 Project Two Submission: Grand Strand Systems**

**Daryl Murtha**  
**Department of Computer Science, Southern New Hampshire University**  
**CS 320:** [**Software Test, Automation QA**](https://learn.snhu.edu/d2l/home/1969331)  
**Desmond Springer**  
**August 15, 2025**

**Unit Testing Approach**

For Project One, I developed three core services: ContactService, TaskService, and AppointmentService. My unit testing approach was designed to validate input constraints, ensure data integrity, and verify exception handling using JUnit. However, due to a missing testing library dependency in my project setup, the tests were unable to execute properly. This configuration oversight impacted my ability to validate the services through automated testing.

Despite this, I structured my test classes to reflect a requirements-driven approach. For example, the requirement that “contacts must have a valid phone number of exactly 10 digits” was addressed in a test method using:

assertThrows(IllegalArgumentException.class, () -> contact.setPhone("123"));

Similarly, I wrote tests to verify that tasks and appointments could be added and deleted by ID, and that fields such as name and address were updatable. However, I later discovered that I had incorrectly declared all fields as final, which prevented updates and contradicted the requirement that only the ID should be immutable.

**Alignment to Requirements**

My testing approach was conceptually aligned with the software requirements. Each test method was mapped to a specific functional requirement, such as:

* “The contact ID must be unique.”
* “Appointments must not be scheduled in the past.”
* “Tasks must allow updates to name and description.”

While the implementation did not fully reflect these requirements due to the incorrect use of final and lack of unique ID generation, the test structure was intended to validate them. This experience highlighted the importance of reviewing both the requirements and the implementation before writing tests.

**Effectiveness of JUnit Tests**

Due to the missing JUnit dependency, my tests did not run and coverage remained below 20%. This was a significant learning moment. I now understand that without proper configuration, even well-written test methods cannot contribute to code quality. In future projects, I will ensure that my build system includes the necessary testing libraries and that coverage tools like JaCoCo are properly integrated.

**Coverage Improvement Plan**

To improve test effectiveness, I plan to:

* Integrate JUnit 5 via Maven using the junit-jupiter dependency.
* Use JaCoCo to monitor line and branch coverage.
* Refactor service classes to remove unnecessary final modifiers.
* Implement unique ID generation using UUIDs.
* Write tests for all CRUD operations, exception paths, and boundary conditions.

My goal is to achieve at least 85% line and branch coverage and validate all functional requirements through automated tests.

**Technically Sound Code**

While my test code was not executable, I followed best practices in its structure. I used assertThrows to validate exception handling and wrote assertions to confirm expected values. For example:

assertEquals("John", contact.getFirstName());

was intended to verify correct object initialization. I also used @BeforeEach to isolate test cases and avoid side effects. These practices align with recommendations from JUnit documentation and industry standards (Gamma et al., 1995).

**Efficient Code**

Efficiency was achieved through:

* Parameterized tests using @ValueSource to test multiple invalid inputs.
* Helper methods for object creation to reduce duplication:

private Contact createValidContact() {

return new Contact("123", "John", "Doe", "1234567890", "123 Main St");

}

* Focused test methods that validated single behaviors.

While these strategies were not validated through execution, they reflect an intent to write maintainable and scalable test code.

**Reflection of Project One**

In retrospect, submitting only the individual code classes for the Contact, Task, and Appointment services limited the ability of my professor to fully evaluate the scope and functionality of each application. A more effective approach would have been to submit the complete application libraries for each service, including all source files, test classes, configuration files, and build scripts.

Providing the full project structure ideally as Maven-based libraries—would have allowed for proper compilation, dependency resolution, and execution of JUnit tests. This would have enabled my professor to verify not only the correctness of the code but also the coverage and robustness of the test cases. Additionally, it would have showcased the architectural decisions, modular design, and integration points across the services, offering a more comprehensive view of the work completed.

By submitting the full libraries, I could have demonstrated a clearer understanding of software packaging, build automation, and test-driven development practices. It also would have made it easier to validate encryption logic, certificate handling, checksum verification, and suppression configuration within a realistic runtime environment.

Moving forward, I will ensure that all future submissions include complete, runnable project structures to support thorough evaluation and reflect the full scope of the technical effort invested.

**Testing Techniques Employed**

I employed several software testing techniques:

* **Black-box testing**: Focused on validating inputs and outputs without relying on internal implementation.
* **White-box testing**: Structured tests to verify internal logic paths and exception handling.
* **Boundary testing**: Designed tests to validate field length constraints and date/time logic.

These techniques ensured that both functional and structural aspects of the code were considered, even if not fully executed.

**Techniques Not Used**

Some techniques were not applicable or not implemented:

* **Integration testing**: Not used, as services were tested in isolation.
* **System testing**: Would require full application context, including UI and persistence layers.
* **Exploratory testing**: Not formally applied due to the structured nature of the assignment.

**Uses and Implications**

Each technique has practical implications:

* Black-box: Useful for validating user-facing behavior and regression testing.
* White-box: Essential for logic-heavy modules and exception handling.
* Boundary testing: Prevents off-by-one errors and ensures robustness.
* Integration testing: Important when services interact, such as syncing contacts with appointments.
* Exploratory testing: Valuable in agile environments for uncovering unexpected behaviors.

Choosing the right technique depends on project scope, risk level, and development stage (Myers et al., 2011).

**Mindset: Caution**

I approached the project with caution, anticipating edge cases and writing defensive code. For example, I tested null inputs even when not explicitly required:

assertThrows(IllegalArgumentException.class, () -> taskService.addTask(null));

This helped prevent silent failures and ensured the code was resilient. However, I learned that caution must extend beyond code logic to include environment setup and dependency management.

**Bias**

To limit bias, I reviewed my own code critically and wrote tests before implementation. This helped avoid confirmation bias. I also stepped away from the code before testing to gain a fresh perspective. If I were testing someone else’s code, I would be more objective—but testing my own required extra vigilance. In future projects, I plan to incorporate peer review and test-first development to further reduce bias.

**Discipline**

Discipline is essential in software engineering. While I was disciplined in writing test methods, I failed to verify their execution. This led to low coverage and missed opportunities for validation. To avoid technical debt, I will:

* Validate test execution early.
* Use CI tools to automate testing.
* Refactor code based on test results.
* Document edge cases and test strategies.

Quality cannot be compromised, and every step from configuration to execution must be treated with equal importance.

**Future Testing Workflow**

To ensure consistent quality in future projects, I will adopt the following workflow:

1. Ensure that added testing dependencies are present and functional during initial project setup.
2. Write unit tests before implementing features.
3. Run tests with each commit using a CI pipeline.
4. Monitor coverage with JaCoCo and address gaps.
5. Conduct peer reviews of both code and tests.

This disciplined approach will help me avoid technical debt and deliver robust, maintainable software.

**References**

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Myers, G. J., Sandler, C., & Badgett, T. (2011). *The Art of Software Testing* (3rd ed.). Wiley.