Mission Debrief: Unit Testing Summary and Reflections for Grand Strand Systems

Bradley D. Saucier

Southern New Hampshire University

CS 320: Software Testing, Automation, and Quality Assurance

Professor Wilson

February 15, 2026

Executive Summary

This report documents the unit testing approach used to verify the Contact, Task, and Appointment services for Grand Strand Systems. The objective was translating requirements into executable JUnit 5 verification controls, then defending quality through aligned assertions and structural coverage analysis. Across all three services, the protocol enforced strict input validation, immutability of unique identifiers, and predictable service state transitions. The result is an auditable, repeatable test suite resistant to regression.

Common Verification Pattern

All three services follow the same verification pattern: model classes enforce constraints at construction and on allowed updates, while service classes enforce uniqueness and lifecycle behavior. Tests were derived from requirements using specification-based techniques (Hambling et al., 2019) and executed as repeatable checks in the Maven test lifecycle and CI pipeline.

Contact Service

Primary risks include invalid identity data, inconsistent field constraints, and accidental mutation of identifiers. The Contact model enforces a non-null unique identifier with a maximum length and rejects invalid personal fields (names, phone, address). The Contact identifier is immutable, preventing downstream reference failures when the record is used as a stable key.

Alignment evidence

Model tests validate null handling and boundary conditions using equivalence partitioning and boundary value analysis (Hambling et al., 2019). Boundary value analysis is applied by testing at

the maximum allowed length and immediately beyond it (boundary value plus one, boundary+1). For example, ContactTest.contactIdNullThrows verifies rejection of a null contactId, and ContactTest.contactIdTooLongThrows verifies rejection beyond the maximum length. Service tests verify update behavior without allowing identifier mutation, which preserves referential integrity; ContactServiceTest.updateContactUpdatesFields demonstrates that allowed fields can change while the identifier remains stable.

Task Service

Primary risks include constraint drift across fields, invalid update paths, and accidental identifier mutation during edits. Task entities are more mutable than contacts because name and description updates are required, but the taskId remains immutable as the anchor for lifecycle operations.

Alignment evidence

Model tests validate null and boundary+1 failures for taskId and name constraints. TaskTest.taskIdNullThrows verifies rejection of a null taskId, and TaskTest.nameTooLongThrows verifies rejection beyond the name length boundary. Update tests confirm setters accept valid updates and reject invalid updates via exceptions; TaskTest.settersRejectInvalidValues validates that setter-based changes are re-validated rather than silently accepted. Immutability is preserved during updates, and TaskTest.settersUpdateFieldsAndIdStaysStable provides concrete evidence that fields can change without allowing taskId mutation.

Appointment Service

Primary risks include temporal validation errors, test instability over time, and reference-based mutation of dates. The Appointment service introduces the most operationally sensitive logic in this project: time. A correct design must reject dates in the past and protect the stored appointment date from external mutation.

Alignment evidence

Tests avoid test rot (tests failing over time due to stale hard-coded values) by generating relative dates at runtime for past and future conditions. AppointmentTest.appointmentDatePastThrows verifies that past dates are rejected, and AppointmentTest.descriptionTooLongThrows verifies rejection of boundary+1 descriptions. Defensive copying is verified by AppointmentTest.gettersReturnExpectedValuesAndProtectDate, which confirms the model copies the Date on assignment and on retrieval, and that mutating the original Date does not alter the stored value. This control is critical; because java.util.Date is mutable, shared references create state corruption risks.

Effectiveness and Coverage

Test effectiveness was established through requirement-aligned assertions and structural coverage analysis.

First, every requirement-driven constraint is defended by paired tests: at least one acceptance test at the valid boundary and at least one rejection test at boundary+1 or invalid partitions. This pairing exercises both outcomes of decision logic and targets boundaries where defects commonly occur (Hambling et al., 2019). Second, JaCoCo structural results report overall coverage of 98.8 percent and branch coverage of 95.5 percent, supporting the claim that most

decision points were executed under test. Where branch coverage is not 100 percent, the remaining gap should be limited to paths that do not represent requirement-driven decision logic, such as defensive fallback paths that are not reachable under valid construction and update protocols. Coverage is supporting evidence, not proof; meaningful effectiveness depends on requirement-mapped assertions, not just execution (Hambling et al., 2019). CI execution of the Maven test lifecycle ensures the same verification checks run on each change, reducing regression risk and keeping the evidence current.

Experience Writing JUnit Tests

Technical Soundness

Technical soundness was achieved through isolation, precision, and actionable diagnostics. Test isolation is enforced with @BeforeEach, ensuring each test begins in a clean state and preventing order-dependent failures. Precise exception testing is implemented with assertThrows so failures are verified at the exact point of invalid input, rather than passing due to unrelated exceptions (Garcia, 2017). Valid-boundary acceptance is also explicitly verified using assertDoesNotThrow in tests such as TaskTest.taskIdLengthTenAccepted and AppointmentTest.descriptionLengthFiftyAccepted, which confirms that correct inputs are accepted at the edge of the requirement envelope. Diagnostics were improved with grouped assertions using assertAll when validating multiple fields in a single operation. This provides a more complete failure picture in one run and reduces churn during debugging by reporting multiple discrepancies at once (Garcia, 2017). Test readability was further supported with @DisplayName annotations that label each test by requirement and expected behavior, reducing time-to-triage when failures occur (Garcia, 2017).

Efficiency

The suite achieved efficiency by minimizing redundancy while preserving verification quality. Each test method is short, single-purpose, and scoped to one requirement or one failure mode, which keeps failures easy to localize and supports fast iteration during development. In CI, short-running tests provide rapid feedback on each push, which lowers the cost of verification and makes regression checks routine rather than deferred (Garcia, 2017). For example, ContactTest.contactIdNullThrows and ContactTest.contactIdTooLongThrows isolate two distinct invalid partitions without coupling multiple failure causes in one test. The suite favors clarity over overly abstracted test frameworks, but it is structured so that parameterized tests could be adopted later for repeating boundary patterns, such as using @CsvSource to iterate invalid lengths and payloads in a single reusable test body (Garcia, 2017). This preserves maintainability as the constraint set grows while keeping the current suite readable at an undergraduate scope level.

Reflection

Techniques Employed

The primary techniques employed were equivalence partitioning, boundary value analysis, and state transition testing as defined in the ISTQB framework (Hambling et al., 2019). Equivalence partitioning was used to select representative inputs from valid and invalid classes rather than attempting exhaustive testing. Boundary value analysis was applied by testing at the maximum allowed length and immediately beyond it (boundary+1), which targets off-by-one errors that commonly appear at constraint edges (Hambling et al., 2019). State transition testing was applied

to confirm lifecycle behavior: after add operations, records exist and can be retrieved; after delete operations, records are removed and the system returns to a stable state. This combination is appropriate for Project One as the dominant risks are boundary violations and predictable lifecycle behavior under create and delete operations.

Techniques Not Employed

Several techniques were excluded as the project scope lacks the dependencies or interfaces they evaluate.

Integration testing was not performed because the services use in-memory structures rather than external persistence or networked dependencies. In a deployed environment, integration tests would verify boundaries such as database writes, concurrent access, and serialization of contact, task, and appointment records (Hambling et al., 2019).

System testing was not performed because the project does not include a full application stack with UI workflows and end-to-end scenarios. In production, system tests would validate user-facing flows such as creating a contact, adding tasks for that contact, scheduling appointments, and confirming that the combined behaviors remain correct across realistic usage sequences (Hambling et al., 2019).

Security testing was not performed because the scope was unit-level validation of constraints rather than threat-driven evaluation. In a deployed system, negative security testing would be required to harden input-handling against malformed payloads, injection-style strings, and abuse of boundary conditions, especially for any fields that transit UI, APIs, logs, or storage (Hambling et al., 2019).

Practical Uses and Implications

These techniques scale directly into real engineering practice because they provide a defensible way to select tests under time constraints. Equivalence partitioning reduces the test set while still covering distinct categories of risk, which is valuable when requirements are strict but schedules are real (Hambling et al., 2019). Boundary testing increases confidence at the edges, which is where requirement disputes and user input errors concentrate. State transition testing protects lifecycle integrity, which becomes critical as features expand, such as detecting that a delete operation fails to remove the record from the backing store and leaves the system in an inconsistent state. Unit testing remains necessary as the base layer of verification, and it provides the evidence needed to refactor safely as the code evolves and complexity grows.

Mindset: Caution, Complexity, Bias, and Discipline

Caution means assuming the system can fail and designing tests to prove where it holds. The suite consistently tests nulls, boundary values, boundary+1 violations, and past dates rather than relying on valid-input tests alone. This treats invalid input as expected operating pressure and verifies that failures are rejected early and predictably.

Complexity emerges through interrelationships, even in small systems. If a model accepts invalid data, the service may function with corrupted state, breaking uniqueness checks, update logic, and downstream operations. Verifying constraints at the model layer and confirming lifecycle behavior at the service layer reduces the chance that a defect in one layer silently degrades another.

Bias mitigation required intentional separation between what the code does and what the requirements demand. Because the developer and tester roles are combined in this course, confirmation bias can lead to tests that only prove the happy path. Deriving failure-path tests from requirements effectively counters this bias and increases defect detection (Hambling et al., 2019).

Discipline is the refusal to accept unverified assumptions. The test suite functions as an operational control: it defines the validation envelope, exposes boundary failures, and prevents regression by running the same checks repeatedly in CI. This satisfies academic requirements while establishing verification habits transferable to professional practice.

References

Garcia, B. (2017). Mastering software testing with 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.