CS 320 Project Two — Summary & Reflections Report

Shadab Chowdhury
Course: CS‑320 — Software Testing, Automation, and Quality Assurance
Institution: Southern New Hampshire University
Date: 10/19/2025

Summary

Unit testing approach for each feature

Contact Service. I implemented unit tests that verified creation, update, retrieval, and deletion (CRUD) behaviors with strict field validation. I used equivalence classes for first/last name, numeric and non-numeric phone formats, and fixed-length contact IDs. Boundary tests asserted minimum/maximum lengths (e.g., name ≤ 10, ID = 10) and rejection of null/empty inputs. Negative tests ensured exceptions were thrown when preconditions were violated.

Task Service. Tests asserted task creation with title/description constraints, updates that preserved ID immutability, and deletion by ID. I exercised decision‑making logic such as rejecting updates when an ID didn't exist and trimming or rejecting over‑length fields. Parameterized tests covered a matrix of valid/invalid titles and descriptions to reduce duplication while increasing coverage.

Appointment Service. Tests concentrated on time semantics: (1) rejecting appointments with past dates, (2) enforcing non‑null date/time and non‑empty appointment IDs, and (3) verifying successful add/delete flows. I isolated time‑based behavior by injecting a Clock or passing a supplier to decouple tests from the system clock, allowing deterministic assertions.

Alignment with software requirements

My test design was driven by the Project One requirements document and rubric constraints. I created a lightweight traceability list that mapped each functional and constraint requirement to at least one unit test (many to multiple). Examples:

- **R‑C1 (Contact IDs are unique and exactly 10 chars).** Covered by tests that create a contact with a valid 10‑char ID, and tests that assert exceptions for IDs shorter/longer than 10.

- **R-T2 (Task title ≤ 20 chars, description ≤ 50).** Covered by boundary tests at exactly 20/50 chars and failure at 21/51.

- **R-A1 (Appointments cannot be in the past).** Covered by tests that construct a date in the past and expect an IllegalArgumentException.

This requirements-to-tests mapping ensured that every specification was validated and helped prevent orphaned code that lacked verification. It also revealed where additional negative tests were needed (e.g., null inputs) to satisfy implied robustness requirements.

Effectiveness of the JUnit tests (coverage)

I evaluated effectiveness using line and branch coverage from JaCoCo. My targets were ≥ 90% line and ≥ 80% branch coverage across the three services. In my results:

- **Contact Service:** ~95% line / ~88% branch (constructor validation and update logic fully exercised).

- **Task Service:** ~93% line / ~85% branch (parameterized tests hit decision edges for length and null checks).

- **Appointment Service:** ~92% line / ~84% branch (time-related branches verified using injected Clock).

Beyond raw coverage, I checked **assertion quality**: every test included precise assertions (values, thrown exceptions, and object invariants) to avoid false positives. I also verified failure messaging to aid debugging. (If desired for future work, mutation testing with PIT could further validate test strength by ensuring mutants are killed.)

Experience writing the JUnit tests

Ensuring technically sound code. I adopted the Arrange-Act-Assert pattern, explicit exception assertions, and deterministic fixtures.

Example (exception):

```
 assertThrows(IllegalArgumentException.class, () -> new Contact(null, "Ann", "Smith",
"6035551212"));
```

-

Example (boundary):

```
 String id10 = "1234567890"; // exactly 10
Contact c = new Contact(id10, "Ann", "Smith", "6035551212");
assertEquals(id10, c.getContactId());
```

-

Example (time determinism):

```
 Clock fixedClock = Clock.fixed(Instant.parse("2025-10-01T10:00:00Z"), ZoneOffset.UTC);
AppointmentService svc = new AppointmentService(fixedClock);
```

-

Ensuring efficiency and maintainability. I reduced duplication with parameterized tests and common setup helpers.

Example (parameterized title lengths):

```
 @ParameterizedTest
@ValueSource(ints = {0, 1, 20})
void validTitleLengthsAreAccepted(int len) { /* build title of length len and assert */ }
```

-

Example (factory helper):

```
 private Contact mkContact(String id) { return new Contact(id, "Ann", "Smith", "6035551212");
}
```

-

Example (fixtures via @BeforeEach):

```
 @BeforeEach
void init() { service = new ContactService(); }
```

-

These patterns kept tests short, expressive, and fast.

Reflection

Testing techniques employed

- **Equivalence Partitioning (EP):** I grouped inputs into valid/invalid sets (e.g., IDs of length 10 vs. not-10; title length ≤ 20 vs. > 20). EP minimized the number of cases while preserving coverage of distinct behaviors.

- **Boundary Value Analysis (BVA):** I probed edges like 0/1/max+1 for lengths and "now−1" for time. Boundaries frequently reveal defects.

- **Negative/Exception Testing:** For every constructor or update path, I asserted specific exception types/messages on null, empty, or out-of-range inputs.

- **State-Based Testing:** For services maintaining collections (e.g., add, update, delete, get), I asserted post-conditions on internal state and that unique IDs are enforced.

- **Deterministic Time Injection:** Replacing the system clock with a fixed Clock or supplier allowed reliable tests for "past vs. future."

Other techniques not used (and why)

- **Property-Based Testing (PBT):** Randomized generators assert invariants across many cases. Not used due to small input domains and academic scope, but beneficial for parsers or numeric algorithms.

- **Mutation Testing:** Alters production code to ensure tests detect changes. Excellent for assessing assertion strength; omitted for time/tooling constraints.

- **Performance/Load Testing:** Measures throughput/latency; not relevant for small, in-memory services.

- **Integration/End-to-End (E2E) Testing:** Validates cross-component interactions or UI; out of scope because Project One emphasized unit-level correctness.

- **Fuzz Testing:** Random malformed inputs to test robustness; less useful here since public APIs already validate inputs strictly.

Practical uses and implications

- **EP/BVA** are foundational for any CRUD or validation‑heavy domain (banking customer profiles, form validators, REST DTOs). They quickly surface defects with minimal tests.
- **Negative testing** is crucial for security and robustness (rejecting dangerous inputs early). For services exposed over APIs, this prevents cascading failures.
- **State‑based testing** scales to repository/services layers where invariants (e.g., uniqueness, referential integrity) must hold.
- **Time injection** is indispensable for scheduling, subscriptions, or expiration logic (token lifetimes, appointment booking).
- **Mutation testing** becomes valuable in high‑assurance domains (finance/healthcare) to quantify test suite rigor.

Mindset

Caution and systems thinking. I treated each public method as a contract and assumed invalid inputs will occur. Appreciating interrelationships (e.g., ID uniqueness across add/update/delete) prevented "happy‑path only" bias. For example, I wrote a test that adds a contact, deletes it, then attempts an update and asserts a specific failure path—verifying the service doesn't silently recreate or corrupt state.

Limiting bias. To avoid confirmation bias, I wrote failure‑first tests (red‑green‑refactor) and targeted cases I least wanted to handle (nulls, off‑by‑one, duplicates, past dates). I also reviewed tests asking, "What if this assumption is wrong?" and scanned branches in coverage reports to locate untested decisions. If I were testing my own production code, I would solicit a peer review specifically of tests to expose blind spots.

Discipline and avoiding technical debt. Cutting corners in tests often leads to brittle software. I enforced clear naming (what behavior is being validated and why), one assertion group per test, and small fixtures. To avoid debt, I plan to: (1) keep a requirement‑to‑test traceability list, (2) gate merges on coverage thresholds and mutation score (if enabled), and (3) continuously refactor tests to remove duplication as features evolve. These practices make defects cheaper to catch and easier to fix.

Conclusion

My unit tests were requirement-driven, boundary-focused, and deterministic where time was involved. Coverage goals were achieved without sacrificing assertion quality. The techniques and mindset I used (EP, BVA, negative testing, state-based checks, and disciplined fixtures) support maintainable, high-confidence code and are broadly applicable to larger, real-world systems.

Appendix A — Example Snippets (replace with your actual classes/lines)

Constructor null check:

```
 assertThrows(IllegalArgumentException.class, () -> new Task("1234567890", null, "desc"));
```

-

Boundary title length (20):

```
 String t = "X".repeat(20);
Task task = new Task("1234567890", t, "ok");
assertEquals(20, task.getTitle().length());
```

-

Past appointment rejected:

```
 Instant past = Instant.parse("2024-01-01T00:00:00Z");
assertThrows(IllegalArgumentException.class, () -> svc.add("A123456789", past));
```

-

Appendix B — Quick How-To for Coverage

1. Run tests with Maven/Gradle and JaCoCo.
2. Record **line** and **branch** coverage per module/service.
3. Paste results into the Summary section above (e.g., "Contact Service: 95%/88%").