**CS 320: Project Two - Summary and Reflections Report**  
**Author:** Gonzalo Ramos  
**Date:** April 11, 2025

**Summary**

Unit Testing Approach  
For Project One, I developed unit tests for the Contact, Task, and Appointment services using JUnit 5 within a Maven-managed structure. Each service was tested independently using an in-memory data structure without a user interface or database layer. My approach centered around verifying each business rule from the specification through method-level testing, such as adding, updating, and deleting data, and validating input constraints like character length and null checks (Tutorials Point, 2024).

The testing approach was closely aligned with the software requirements. For instance, the Contact class required validation for fields like contactId being non-null and under 10 characters. I validated these in the test testInvalidId() in ContactTest.java by asserting exceptions when constraints were violated. Similarly, phone number validation was enforced using assertThrows(IllegalArgumentException.class, ...) to ensure exactly 10 digits were required (Tutorials Point, 2024).

Test Effectiveness and Coverage  
To evaluate the effectiveness of the JUnit tests, I used Eclipse's built-in test coverage tool. All 12 Java files were tested, and the screenshot ContactServiceCoverage.png shows full test execution with 100% success and no errors. This level of test coverage gives me confidence that the services are robust enough for release (Oracle, 2024).

Technically Sound and Efficient Code  
I ensured technical soundness by writing concise, isolated test methods with clear assertions. For example, ContactServiceTest.java includes a test called testUpdateFields() which checks all updatable fields by creating a contact, performing updates, and then asserting their values with assertEquals(). This structure ensures that each business rule is tested directly and independently.

Efficiency was achieved through reuse of setup logic and the use of @BeforeEach to initialize services before tests. For example, the method initializeServiceBeforeEachTest() in TaskServiceTest.java prevents duplication and ensures consistent preconditions for each test case.

**Reflection**

Techniques Employed  
I primarily employed unit testing, which involves testing individual components in isolation. Its strengths include early bug detection, fast execution, and clear fault localization (Tutorials Point, 2024). I also used exception-based testing to ensure that invalid inputs are properly rejected, which is essential for input validation logic.

Other Techniques Not Used  
I did not use integration testing or system testing in this project. Integration testing focuses on verifying how components work together, which wasn’t necessary due to the modular structure. System testing verifies end-to-end functionality, typically with UI or external systems, which was not applicable for this backend service-only application (Tutorials Point, 2024).

Uses and Implications  
Unit testing is ideal for validating isolated logic such as data validation, CRUD operations, and enforcing constraints. In contrast, system testing would be more suitable in a full-stack application to ensure workflows like submitting forms or processing appointments work across layers (Tutorials Point, 2024).

Mindset and Caution  
Throughout the project, I approached each test with caution, expecting that the code might fail. For example, I wrote testDeleteNonExistentAppointment() in AppointmentServiceTest.java to explicitly test an edge case where deletion is attempted on an unknown ID. This mindset of "expecting bugs" helped ensure I didn't overlook negative paths.

Limiting Bias  
To limit bias in my code review and testing, I treated my code as if it was written by someone else. I intentionally wrote tests to break the application and forced it to handle unexpected inputs. Had I assumed the code was flawless, I might have skipped writing assertThrows() tests that uncovered gaps in validation.

Discipline and Technical Debt  
Maintaining discipline in both writing and testing code is vital. Cutting corners may seem efficient in the short term but leads to technical debt. For example, if I had skipped field-length checks in the Task object, a later integration may have failed when exceeding limits. To avoid this, I plan to enforce automated testing in future projects through continuous integration and always write meaningful unit tests as part of my development routine.

**References**

Oracle. (2024). *JUnit 5 User Guide*. https://junit.org/junit5/docs/current/user-guide/

Tutorials Point. (2024). *Software Testing - Unit Testing*. https://www.tutorialspoint.com/software\_testing/software\_testing\_quick\_guide.htm