Summary and Reflections Report

Christopher Kwame Prempeh

CS-320

June 22, 2025

**Summary and Reflections Report**

**Unit Testing Approach**

For Project One, I created JUnit tests for the contact, task, and appointment services based on clearly defined requirements. Each of these services involved strict constraints, such as character limits on IDs, validation of field inputs, and rules governing update methods. My testing approach focused on enforcing those rules precisely.

Each object, Contact, Task, and Appointment was tested with both valid and invalid inputs. For example, in ContactTest.java, I tested for invalid ID lengths and null values using this line:

**assertThrows(IllegalArgumentException.class, () -> {**

**new Contact("12345678901", "John", "Doe", "1234567890", "123 Main St");**

**});**

This was directly aligned to the requirement that contact IDs must be 10 characters or fewer. I applied a similar method across all test classes to ensure the constraints were strictly followed. My tests included coverage of constructors, getters, setters, and update methods.

**Alignment to Software Requirements**

My tests mapped directly to the project specifications. The requirements stated that all fields must be validated and updates must affect only specific attributes. In response, I wrote separate unit tests for every service method, confirming both expected functionality and edge cases. By doing this, I demonstrated not only that the application met the functional requirements, but also that it handled exceptions reliably.

I also included helper methods such as getContact() and getTask() to check object states after update operations. This allowed me to confirm that changes occurred as intended and that no unintended side effects took place. In short, every method required by the specification was accompanied by at least one meaningful test.

**Quality and Effectiveness of JUnit Tests**

To gauge the quality of my tests, I measured code coverage using IntelliJ’s built-in coverage tools. Every service class maintained above 80% test coverage, indicating that a majority of code paths had been exercised.

More importantly, the tests went beyond surface-level validation. I included both positive and negative test cases. For instance, I tested not only that tasks could be created with valid data, but also that invalid data, such as an empty task name or a due date in the past, would be rejected.

This comprehensive approach ensured the robustness of the system. When I ran into a case that passed unexpectedly, it indicated a missing validation check in the source code, which I promptly fixed. That feedback loop proved the value of thorough unit testing.

**Writing Technically Sound and Efficient Code**

One strategy I used to maintain technically sound tests was to validate each public method with corresponding assertions. For example:

**@Test**

**public void testUpdateTaskNameSuccessfully() {**

**service.updateTaskName("001", "Updated Task");**

**assertEquals("Updated Task", service.getTask("001").getName());**

**}**

This test confirms both that the update method does not throw errors and that it results in the correct state change. For efficiency, I reused object instances when testing multiple behaviors of the same entity. Instead of creating new objects in each test, I set up shared resources using @BeforeEach to isolate test environments and avoid repetition. This helped keep the codebase lean while ensuring each test was independent and reliable.

Grouping related assertions together also improved efficiency. For example, when testing input constraints, I bundled several invalid cases into a single test method, using a parameterized structure to keep the file concise without losing clarity.

**Reflection**

In this project, I used two main testing techniques: unit testing and boundary testing. Unit testing focused on validating small, isolated methods, each constructor, getter, setter, and updater was tested on its own. These tests ensured that each method performed as expected under specific conditions.

Boundary testing was applied to catch off-by-one errors and data format violations. For example, I tested 10-character IDs (valid) and 11-character IDs (invalid), as well as phone numbers that contained letters or symbols. I did not use integration testing or mock testing in this project. Integration testing, which examines how multiple modules interact, was not necessary since each service was self-contained. Mock testing, which simulates external dependencies, also was not applicable because all services operated entirely in memory.

Had this been a full-stack application, other techniques would be important. Integration testing would confirm that data sent from a UI component reaches the backend correctly. End-to-end testing would simulate user interactions from start to finish. Mock testing would allow testing of code that depends on external APIs, databases, or payment processors without relying on those systems being available or functional. In professional software development, unit testing forms the base of a testing strategy, but the other techniques become vital as the application grows. Integration and mock tests help catch defects that only occur in real-world environments with multiple components interacting.

In a microservices environment, for example, unit testing ensures each service behaves correctly. Integration testing ensures that services talk to each other correctly. Mock testing helps simulate interactions with services that are still in development. Boundary testing catches edge cases that could crash a system or expose security flaws. Each technique has a role, and choosing the right one depends on the context. Over-relying on unit tests can lead to missed bugs if components fail at their connection points.

Throughout the project, I approached testing with caution. For instance, in ContactService, I wrote individual update methods such as updateFirstName and updatePhone, and I tested each method to make sure only the intended field changed. I deliberately introduced bad inputs such as nulls, long strings, and invalid formats to ensure the validation code held up. Appreciating the complexity of the system meant recognizing that even a simple field update could affect the application’s integrity. If I changed a phone number but allowed invalid input, the whole contact system could become unreliable. This awareness guided how carefully I wrote and reviewed my tests.

Bias is a real risk when developers test their own code. To counter this, I pretended I was an outsider trying to break the system. I did not assume my code would work. I wrote tests with the mindset that something would go wrong, and I needed to find where. For example, I tested update methods not just for success but also for failure: empty strings, overly long inputs, and illegal characters. I also looked for cases I had not thought of initially. By approaching the code from multiple angles, I reduced the chance of blind spots. If I had assumed everything was fine just because the code compiled, I would have missed critical issues.

Discipline was essential to maintaining quality. It was tempting to skip tests for minor methods or assume defaults would handle edge cases, but I resisted that urge. Every shortcut introduces risk. Technical debt builds up when developers skip validation or leave out tests, and that debt has to be paid later, usually at the worst time. To avoid that, I plan to continue writing tests alongside the code, not after. This test-driven mindset forces me to think more clearly about what the code should do and reduces the chance of bugs slipping into production. It is a practice I will carry forward as a professional developer.