Joseph Dumke

SNHU

CS-320

June 16, 2025

**7-2 Project Two Submission**

To ensure high-quality code and alignment with functional requirements, I developed a comprehensive suite of unit tests using the JUnit framework for the three main features of the mobile application: Appointment, Task, and Contact. For each of these features, I implemented a dedicated service class to manage core operations and corresponding unit tests to validate the behavior of each method under expected and edge-case scenarios.

For the **Appointment** feature, I validated that appointment IDs were unique, appointment dates were not in the past, and descriptions were limited to 50 characters The as assert True(appointmentService.addAppointment(a)); which confirmed that valid appointments could be successfully added. In addition, I also tested invalid inputs by asserting exceptions with lines like assertThrows(IllegalArgumentException.class, () -> new Appointment("01234567891", new Date(), "Description"));, ensuring the ID limit was respected.

The **Task** feature followed a similar structure, validating task ID uniqueness, ensuring names did not exceed 20 characters, and confirming descriptions were within a 50-character limit. In TaskServiceTest, I wrote tests to ensure the addition, update, and deletion functionalities worked as expected. For instance, assertNotNull(taskService.getTask("12345")); helped confirm that tasks were correctly stored and retrieved.

The **Contact** feature had the most fields to validate, including first name, last name, phone number, and address. I ensured each field met its length requirement and was non-null. In ContactServiceTest, I tested both successful creation and invalid input handling with assertions such as assertThrows(IllegalArgumentException.class, () -> new Contact("12345678901", "John", "Smith", "123456789", "123 Street Blvd.")); to ensure the phone number format was enforced.

Each of these approaches was directly tied to the documented software requirements. For example, the requirement that contact phone numbers be exactly 10 digits long was enforced both in the Contact class constructor and confirmed through targeted unit tests. Similarly, the 50-character constraint for descriptions in the Appointment and Task classes was verified through test cases that deliberately exceeded the limit.

I believe the overall quality of the JUnit tests is strong due to their scope and precision. Every requirement has at least one test case that validates correct behavior and one that tests for expected failure. While I did not generate a formal code coverage report, the breadth of test cases provides informal assurance that most if not all functional paths were tested. This includes edge cases and invalid input scenarios, improving confidence in robustness.

Writing these tests improved my ability to think from both a developer and user perspective. In addition, I found it effective to use descriptive method names and assertions to trace test intentions. For instance, in TaskTest, the method testSetNameTooLong clearly describes the failure condition being validated, helping both readability and maintainability.

To ensure technical soundness, I leveraged exception assertions throughout the test files. Lines like assertThrows(IllegalArgumentException.class, () -> new Appointment(null, new Date(), "desc"));ensured that my constructors and setters were defensive and reliable. Efficiency was promoted by grouping related assertions and avoiding redundant tests. For example, assertAll() was used to combine multiple related validations, which optimized test execution and reduced repetition.

In this project, I primarily used **unit testing** with the JUnit framework, combined with **boundary value analysis** and **negative testing**. Unit testing allowed me to isolate each class’s logic and validate its behavior independently from the UI or backend systems. Boundary value analysis was useful for enforcing constraints like the 10-digit phone number, the 50-character description, and the ID length, ensuring both ends of the allowed range were tested. Negative testing helped validate the program’s robustness by providing null values, overly long strings, and invalid formats.

Other common software testing techniques not used in this project include **integration testing**, **system testing**, and **automated UI testing**. Integration testing involves checking how multiple classes or services interact, such as ensuring AppointmentService correctly integrates with a persistence layer. System testing validates the entire application’s behavior under realistic conditions, typically through end-to-end scenarios. Automated UI testing, using tools like Selenium, verifies that front-end components behave correctly from a user perspective.

These techniques each serve different needs. Unit testing is best for logic-heavy classes in early development. Integration and system tests are critical as the codebase grows and modules interact. UI testing is important in consumer-facing apps where layout and input behavior are essential. In professional environments, using a combination of these methods ensures robust

Throughout the project, I approached testing with a cautious and thorough mindset. Since each class enforced specific constraints, it was crucial to understand how a change in one part could affect the system. For example, modifying the way an Appointment object handles date validation could inadvertently affect service logic or cause test failures in downstream methods. This complexity required careful attention to interdependencies between constructors, setters, and service-layer logic.

To minimize bias while testing my own code, I deliberately created test cases I hoped would fail. For instance, I intentionally passed null values, strings that exceeded allowed lengths, and duplicate IDs to expose flaws in my logic. This helped simulate a third-party reviewer’s perspective and prevented me from assuming my code worked perfectly.

Bias in testing one's own code is a real concern. Developers may unintentionally avoid creating tests that challenge their assumptions. For example, if I had only written tests for successful cases, I might have missed bugs that would occur during actual use. To overcome this, I also wrote tests that ensured my service classes handled duplicate IDs and removal of nonexistent entries correctly, which often reveals edge-case issues.

Maintaining a disciplined mindset is essential in software engineering. Cutting corners on validation or skipping tests may save time in the short term but introduces **technical debt** that can lead to defects, rework, or customer dissatisfaction. To avoid this, I plan to follow **test-driven development (TDD)** where applicable, document all business logic rules, and regularly run coverage tools to catch gaps early. For instance, I will integrate tools like JaCoCo for coverage metrics and SonarQube for static analysis to support code quality enforcement.

In conclusion, this project enhanced my appreciation for testing and quality assurance. Writing robust, maintainable JUnit tests not only confirmed my logic was sound but also made future debugging and feature additions more manageable. This disciplined approach to software testing will remain a central part of my development practice going forward.