Connor Martin

SNHU

Grand Strand System

10/12/2025

**Summary**

**Unit Testing Approach**

During Project One, I developed and tested three main features: the **ContactService**, **TaskService**, and **AppointmentService**. Each service was designed to add, update, and delete data stored in memory. My approach to testing each of these was consistent—write **JUnit 5 unit tests** that checked both valid and invalid behaviors for every method.

For the **ContactService**, I focused on verifying that contacts could be added correctly and that each contact ID was unique. I also tested that inputs like names and phone numbers followed the required length and format. For example, tests ensured a contact could not be created with an ID longer than ten characters or with a null phone number.

The **TaskService** tests checked that each task could be created, updated, and deleted as expected. I made sure that invalid data, such as an empty task name or a description longer than 50 characters, would trigger an exception. This matched the project’s business rules exactly.

The **AppointmentService** tests confirmed that appointments were properly created with valid IDs and dates. I included tests to ensure invalid dates were rejected and that updates reflected correctly when retrieving an appointment afterward. Each of these services was tested separately so that errors could be isolated easily.

**Alignment to Software Requirements**

My testing approach aligned directly with the software requirements for each feature. Each test case was written based on a specific requirement from the project description. For instance, one requirement stated that IDs must be unique, which I verified using assertions like:

assertThrows(IllegalArgumentException.class, () -> contactService.addContact(existingContact));

This confirmed that duplicates were not allowed. Another requirement limited input lengths, which I verified using:

assertThrows(IllegalArgumentException.class, () -> taskService.addTask("12345678901", "Test", "desc"));

By writing tests that corresponded to every functional rule, I could confirm that my code met the project’s expectations.

**Quality of JUnit Tests**

I measured the quality of my JUnit tests through both **test coverage** and **test reliability**. I used Eclipse’s coverage tool to check how much of my code was being tested, and most classes achieved **above 90% coverage**. I also included both positive and negative test cases to ensure the code behaved as expected in every scenario.

The tests were effective because they checked for all common sources of failure—null inputs, invalid IDs, and over-length fields. Each method had its own group of tests, which made it easy to verify that all paths were being executed.

**Experience Writing JUnit Tests**

Writing these JUnit tests helped strengthen my understanding of how testing supports good software design. I used annotations like @BeforeEach to reset the objects before each test, keeping all tests independent and repeatable. For example:

@BeforeEach  
void setUp() {  
 contactService = new ContactService();  
}

This ensured that one test’s data never interfered with another.

I used assertEquals, assertNotNull, and assertThrows to verify both success and failure conditions. For instance:

assertEquals("John", contactService.getContact("001").getFirstName());  
assertThrows(IllegalArgumentException.class, () -> contactService.deleteContact("999"));

These statements made the tests easy to read and maintain.

I kept the code efficient by minimizing duplication and grouping similar logic into helper methods. Since no database was involved, the tests ran very quickly, making it easy to rerun them after each code change.

**Reflection**

**Testing Techniques**

In this project, I primarily used **unit testing**, **black-box testing**, and **boundary testing**.

* **Unit Testing:** Each test focused on one small piece of functionality, such as verifying that the addTask() or deleteContact() methods worked correctly.
* **Black-Box Testing:** I treated the methods as a “black box,” focusing on inputs and expected outputs rather than how the methods were written internally.
* **Boundary Testing:** I tested edge cases—like input strings at their exact length limits—to make sure the validation rules were applied correctly.

Some techniques I did not use include **integration testing**, **system testing**, and **regression testing**.  
 Integration testing checks how different modules work together, which wasn’t necessary here because each service was self-contained. System testing is used to verify the behavior of a complete application, and regression testing is used to confirm that new code doesn’t break old features. These techniques would be valuable if the project expanded into a full mobile app.

Each of the methods I used helped ensure quality at different levels. Unit testing caught logic errors early, black-box testing confirmed that the services behaved correctly from a user’s point of view, and boundary testing helped ensure the input rules were strict but accurate.

**Mindset**

Throughout this project, I tried to approach testing with caution and attention to detail. My mindset was to **assume the code might fail** and prove otherwise through testing. I learned that even simple functions can cause unexpected errors when given invalid data. For example, testing invalid contact IDs revealed that I needed to improve my exception handling.

It was important to appreciate how the classes interacted. For example, changing how an appointment ID was validated could affect the add and update methods in subtle ways. Understanding these relationships helped prevent hidden bugs.

To limit **bias**, I made sure to write tests that tried to break my own code. It’s easy to assume your code will work, but testing requires a mindset that looks for failure, not success. By intentionally testing bad input and unexpected cases, I was able to verify the code was truly reliable.

Remaining disciplined in testing was also critical. Cutting corners might save time short-term, but it can lead to **technical debt**, where problems build up and become harder to fix later. I plan to avoid this by continuing to write full unit tests for new features, running all tests before merging code, and refactoring whenever code becomes hard to read or maintain.

**Conclusion**

This project taught me the value of a structured and consistent testing process. By writing clear and complete JUnit tests, I was able to confirm that all three services worked as intended and met their requirements. Testing was not just about finding bugs—it was about building confidence that the code was correct, efficient, and maintainable.  
 Working through this project helped me understand that a disciplined and unbiased testing approach leads to higher-quality software and better long-term results for both the developer and the client.