SNHU CS-320

Oct.19, 2025

Project II || Aiden Villanueva

My unit testing approach for the contact, task, and appointment features was centered on a requirement-driven strategy. The core principle was to ensure that every functional and non-functional requirement specified for the software was explicitly verified by one or more dedicated tests. This involved creating a comprehensive suite of tests for each class, covering not only the successful operations but also all foreseeable failure scenarios, edge cases, and invalid inputs.

This method ensured a very high level of alignment between the software and the requirements. For every feature, I could draw a direct line from a requirement to a specific test case that validated it. For instance:

Requirement: The ContactService must be able to add a contact with a unique ID.

Alignment: This was validated by testAddContactSuccess(), which confirms a valid contact is added correctly, and testAddDuplicateContactId(), which confirms the system properly throws an IllegalArgumentException when a duplicate ID is used.

Requirement: All fields in the Task object must meet specific validation criteria (i.e not null, within a certain character limit).

Alignment: This was validated by a series of tests for the Task class constructor. For example, testTaskCreationSuccess() validated a correct instantiation, while testTaskIdTooLong() and testTaskNameNull() confirmed that the object's validation logic correctly rejected data that violated the specified constraints.

This pattern was consistently applied across all three services - Contact, Task, and Appointment; creating a robust testing framework where every piece of application logic was directly tied to and validated against a documented requirement.

Quality and Effectiveness of JUnit Tests

The overall quality of the JUnit tests is, in my opinion, high, which is directly shown by their thoroughness and the resulting code coverage. An effective test suite must do more than confirm that code works under ideal conditions. It must also ensure the application behaves predictably and gracefully when encountering errors or invalid data.

The effectiveness of these tests is best measured by their impact on code coverage. While a specific percentage from a tool was not formally part of this project's deliverables, the systematic approach taken guarantees very high line and branch coverage. For the model classes (Contact, Task, Appointment), the test suites for their constructors cover every single validation rule for every attribute. This includes tests for: Successful object creation. Null values for every required field. Values that exceed the maximum character length for every string field. Invalid date inputs, for example, a past date for an appointment.

That methodical testing of every conditional path within the constructors ensures virtually 100% of the code was executed and validated. This high level of coverage provides strong confidence that the data model is stable, reliable, and secure against invalid data.

Experience Writing JUnit Tests

My experience writing the JUnit tests was focused on two key factors: ensuring the code was technically sound and that the test suite itself was efficient and maintainable.

To ensure the code was technically sound, I relied heavily on JUnit's assertion methods to rigorously verify the application's state after every action. Assertions act as objective proof that the code is behaving exactly as designed. For example, in the testUpdateContactSuccess() method within ContactServiceTest, after the update operation is called, I don’t just check that the object still exists, I verify that the specific fields were changed to their new values:

// After updating contact "ID002"

Contact updatedContact = service.getContact("ID002");

assertEquals("Smith", updatedContact.getLastName());

assertEquals("1112223333", updatedContact.getPhoneNumber());

These lines provide concrete, programmatic proof that the updateContact method is functioning correctly and is free of defects like partial updates or data corruption.

To ensure the test code was efficient, I utilized JUnit annotations to reduce redundancy and improve readability. The @Before annotation was particularly useful. By placing common setup logic in a setUp() method, I could create a fresh, predictable test environment before each test method was executed. This adheres to the Don't Repeat Yourself (DRY) principle.

Reflection

The primary software testing techniques employed in this project were Unit Testing and White-Box Testing.

Unit Testing: This technique involves testing individual software components or modules in isolation to verify their correctness. In this project, each JUnit test file (ContactTest.java, ContactServiceTest.java, etc.) represented a dedicated test suite for a single unit. For example, the Appointment class was tested in complete isolation to validate its constructor logic before we ever tested its integration with the AppointmentService. The primary benefit of this is early bug detection, which makes defects significantly cheaper and easier to fix. Unit tests also serve as a critical safety net for future refactoring.

White-Box Testing: In this method, the tester has full knowledge of the internal structure, logic, and implementation of the code. My approach was a clear example of this. When writing a test like testAppointmentIdTooLong, I did so with the explicit knowledge that the Appointment constructor contained a conditional check for an ID length greater than 10. White-box testing is excellent for achieving high code coverage and is essential for developers to verify their own logic, including internal pathways, conditional branches, and error-handling routines.

There are several other software testing techniques that were not used for this project but are important in the broader software development lifecycle:

Black-Box Testing: This is the inverse of white-box testing, where the tester has no knowledge of the internal code and focuses only on inputs and outputs. A black-box tester would interact with the application through a UI or API endpoint, testing if a new appointment can be created with a past date, without knowing how the date validation is implemented. Its practical use is to validate the software from a user's perspective and ensure it meets functional requirements.

Integration Testing: This technique focuses on verifying the interaction between two or more integrated software components. For example, we would use integration testing to verify that the AppointmentService can correctly save and retrieve data from a database. While our unit tests used an in-memory map, an integration test would use a real database connection to expose issues in data serialization, network communication, or transaction handling.

System Testing: This involves testing the complete, fully integrated software application as a whole. Its purpose is to evaluate the system's compliance with all specified requirements, including functional, performance, and security requirements. For our project, this would involve testing the mobile application's UI, the back-end services, and the database all working together.

Mindset

As a software tester, I employed a high degree of caution. This meant constantly thinking not about how the code should work, but how it could fail. It was crucial to appreciate the complexity of the code because even a small, seemingly isolated component can have downstream effects. For instance, if the validation logic in the Contact class constructor were to change, it could break the ContactService's ability to add or update contacts. By creating cautious, defensive tests for every validation rule in the Contact class, we ensure that any such breaking change would be caught immediately by a failing unit test, preventing the bug from ever reaching a later stage of development.

I actively worked to limit bias in my review of the code. Developer bias is a significant concern; it's natural for a developer to believe their own code works well and to test it in ways that subconsciously confirm that belief. To counteract this, I relied on the software requirements as an objective checklist. Rather than testing what I thought the code did, I tested what the requirements document said it must do. This forced me to write tests for failure cases like duplicate IDs or null names. This objective approach is crucial for effective testing.

Finally, I believe a disciplined commitment to quality is non-negotiable for a software engineering professional. Cutting corners on testing is a direct path to accumulating technical debt - a collection of poor design choices and bugs that make future development slower and more expensive. I have personally lost several projects to this exact issue. It is far more efficient to spend an hour writing thorough tests today than to spend a week debugging a critical failure in production six months from now. To avoid technical debt, my plan is to continue this disciplined practice, to never commit a new feature without comprehensive unit tests, to maintain high test coverage on all projects, and to use the test suite as a safety net to continuously refactor and improve the codebase.