Evviee Evjen

6/20/2025

CS-320 Project two

For this project I implemented unit tests for three main features Contact Service, Task Service and Appointment Service. I used Junit to verify each class and service independently. I created objects with valid data verifying immutability for fields like IDs testing update and deleting methods and checking that invalid inputs throw the appropriate exceptions. For Contact Service tests focused on constraints such as name length phone number formatting and address length. For Task service I tested ID uniqueness valid lengths for name and description and proper behavior of update operations. For the Appointment Service I emphasized checking the dates were not in the past and that the description met its length constraint.

My unit testing approach closely followed the functional requirements provided. The requirements that the contact ID must be no more than 10 characters was testing using

assertThrows(IllegalArgumentException.class, () -> new Contact("12345678901", "John", "Doe", "1234567890", "123 Main St"));

The appointment date constraint was tested using

Date pastDate = new Date(System.currentTimeMillis() - 10000);

assertThrows(IllegalArgumentException.class, () -> new Appointment("001", pastDate, "Past appointment"));

The quality of my Junit test is supported by the high coverage percentage achieved over 80% for each service class. I validated all the possible paths object creation updates deletions and invalid operation. The tests included valid and invalid constructor inputs, setter validations, exception cases for operations like updating nonexistent items, and attempted violations of constraints.

Writing these tests gave me a better insight into how software requirements translate into real world functionality. It also made me more conscious of error handling and the importance of writing defensive code. To ensure technical soundness I validiated that every functional method worked as expected and threw the appropriate exceptions when misused, assertThrows(IllegalArgumentException.class, () -> contactService.updatePhone("badId", "1234567890"));

I made sure it was efficient by using HashMap for constant time access in service classes minimizing duplicate object creation and using before each setup methods to reused initialization logic.

@BeforeEach

void setup() {

taskService = new TaskService();

task = new Task("1", "Initial Task", "Initial Description");

taskService.addTask(task);

}

This structure made my tests cleaner faster and easier to maintain.

In this project I used a couple software testing techniques to ensure correctness, I used Unit testing which was the primary technique where I tested each class in isolation using Junit. Each method constructor and field constraint were verified independently. This helped ensure that individual components meet their requirements without depending on others. Boundary testing I validated that all string fields stay within their defined length limits and that date values were not in the past. Exception testing I used assert throws to verify that the system correctly rejected invalid inputs, such as null values or duplicate IDs.

Techniques that I didn’t use were Integration testing, System testing, and Mocking. I didn’t use integration testing as the services were independent and there was no interaction between them at this stage. Integration testing is useful when validating how multiple components or services work together. System testing since there was no UI or full application layer provided full end to end system testing was not applicable. It would have been more relevant in a more complete application with interfaces and user flows. I didn’t use mocking tools since the services were self-contained and didn’t rely on external APIs or databases. Mocking is most useful when simulating dependencies or external systems.

Unit testing is best for catching bugs early and isolating problems in individual classes. Integration testing is critical when multiple services depend on each other and must be tested as a workflow. Mocking is useful when a class depends on a third-party service or external system that’s unavailable or expensive to call during testing. System testing ensures that the complete system behaves correctly from a user or business perspective.

Throughout this project I created my code as if it had potential flaws, like I made sure to test scenarios like trying to update a task or contact that doesn’t exist.

assertThrows(IllegalArgumentException.class, () -> contactService.updateAddress("nonexistent", "New Address"));

Being cautious helped me uncover issues early and avoid assumptions that can lead to bugs.

Cutting corners in testing might save time now but can lead to major issues later, being disciplined about quality is essential in software development. If I skipped testing appointments dates I could have introduced bugs where appointments were allowed in the past something that would cause logical errors.