John Rosario

CS 320

Project Two

**JUnit Testing Approach**

For Project One, I implemented unit tests for three major features of the application: the Contact Service, the Task Service, and the Appointment Service. Each of these features represented a distinct business requirement, and the goal of my testing approach was to confirm that the functionality worked correctly under both normal and edge-case conditions.

For the **Contact Service**, the JUnit tests verified that each contact could be created with a valid unique ID, name, phone number, and address. I tested boundary conditions such as maximum lengths (e.g., names limited to 10 characters, phone numbers, exactly 10 digits) and invalid inputs (such as null or empty values). This ensured that the system enforced data integrity.

For the **Task Service**, I wrote tests to confirm that tasks required valid IDs, names, and descriptions. I also tested cases where a user attempted to create a task with missing or overly long fields. These tests validated the requirement that all tasks must be concise, well-defined, and correctly stored in the system.

Finally, for the **Appointment Service**, my JUnit tests ensured that appointments could not be scheduled in the past, that IDs were unique, and that valid dates and descriptions were required. I tested scenarios where a user attempted to book an appointment with a past date or missing description. This confirmed compliance with the business rule that all appointments must be forward-looking and properly described.

By covering normal, boundary, and invalid input scenarios, my test case aligned directly with the functional requirements of the system. This gave me confidence that the code met the intended design and would behave predictably in real-world use.

**Experience Writing JUnit Tests**

My experience writing the JUnit tests gave me insight into the importance of precision and clarity in test design. I ensured the tests were technically sound by targeting specific conditions tied to requirements. For example, in the Contact Service tests, I included the following case:  
@Test  
public void testPhoneNumberTooLong() {  
 Assertions.assertThrows(IllegalArgumentException.class, () -> {  
 new Contact(“12345”, “John”, “Doe”, “1234567890123”, “123 Main Street”);  
 });  
}

This test explicitly validated that a phone number longer than 10 digits would trigger an exception, which aligned with the requirement that phone numbers must contain exactly 10 digits.

I also made sure my tests were efficient by avoiding redundancy. For example, instead of creating new object instances repeatedly across different test methods, I used helper methods to generate sample contact or task objects. This reduced duplication in the code and made the tests easier to maintain. Efficiency also came from grouping related test cases into logical classes so that I could quickly evaluate all constraints of a single feature.

**Reflection**

The main software testing technique I used in this project was unit testing, where I isolated and tested individual components of the application. Within unit testing, I relied heavily on boundary value testing to ensure fields like phone numbers and task descriptions met their length limits, and on equivalence partitioning to check that inputs fell into valid and invalid groups.

For example, with boundary value testing, I confirmed that a 10-digit phone number was valid while a 9-digit or 11-digit number caused an exception. With equivalence partitioning, I verified that valid strings of 1-50 characters worked for task descriptions, while empty or overly long strings were rejected.

Techniques I did not use in this project included integration testing and system testing, which are generally performed after unit testing. Integration testing checks whether different compononets work together properly, while system testing validates the entire system against requirements. While valuable, these were outside the scope of the project. If applied in the future, integration testing could verify that the Contact, Task, and Appointment services all function together without data conflicts, while system testing could simulate full user scenarios.

The practical implication of my chosen techniques is that they provide early feedback at the component level, which helps catch bugs before they propagate. This is especially important in agile environments, where catching issues early reduces rework and costs.

**Mindset**

The mindset I adopted for this project was one of caution, discipline, and professionalism. When designing tests, I considered both typical user behavior and possible misuse of the system. For example, I deliberately tested invalid cases such as null Ids or scheduling an appointment in the past. This cautious mindset helped ensure the code was robust against errors.

I also worked to avoid bias when testing my own code. It is tempting to only test cases that confirm the code works, but I made sure to include negative tests that might break it. This helped me remain objective and ensure that the tests were truly meaningful.

Finally, I stayed disciplined by not cutting corners, even when the test cases became repetitive. Instead, I looked for ways to write tests more efficiently without sacrificing thoroughness. This helped me avoid technical debts, where shortcuts taken during development create problems later. By maintaining quality and thoroughness in my unit tests, I ensured that the foundation of the system was strong enough to support future enhancements.

**Conclusion**

Overall, this project reinforced the importance of unit testing as a foundation for high-quality software development. My JUnit tests validated the system’s functional requirements through a combination of normal, boundary, and invalid input testing. By applying boundary value and equivalence partitioning techniques, I was able to confirm that each component behaved correctly. The mindset I carried throughout helped me deliver test cases that were both effect and maintainable.