**Project Two**

**Joseph Wilfong**

**SNHU**

**CS320**

**Professor Jacks**

**06/22/25**

**Project Two**

**Summary**

For Project One, I wrote solid unit tests for ContactService, TaskService, and AppointmentService. For ContactService, I tested that contact IDs are unique and can’t be changed, while checking that first name, last name, phone, and address follow the length and format rules. I included tests for valid inputs and also cases where the inputs are null or too long.

For TaskService, I made sure task IDs were unique and immutable. I tested the name and description fields for length and null values, and checked the add, update, and delete methods, including tests that expect exceptions for invalid data.

In AppointmentService, I tested that appointment IDs are unique and fixed, dates aren’t null or in the past, and descriptions meet length requirements. I also tested setters and confirmed exceptions are thrown when inputs are invalid. The add and delete operations were covered too, including tests for duplicate IDs and missing appointments.

**Aligned to the Software Requirements**

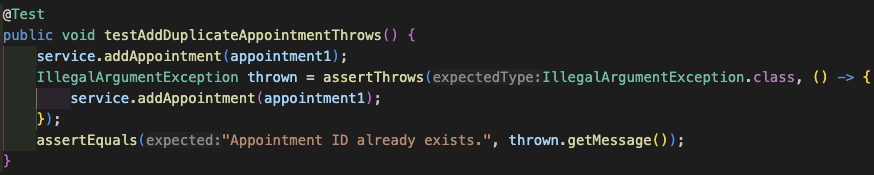
My tests map directly to the software requirements. For example, the test testNullAppointmentDateThrows() clearly checks that the appointment date can’t be null. The test testAddDuplicateAppointmentThrows() proves that duplicate appointment IDs aren’t allowed. Every requirement has one or more tests validating it, so the coverage is tight and focused on what the customer asked for.

**Defending the Quality of JUnit Tests**

The JaCoCo reports show my tests cover more than 80% of the code in all three of the services. My tests don’t just cover the positive paths; they also test invalid inputs and exception handling. This means the code is verified not only for normal operation but also for handling errors gracefully, which makes the tests effective.

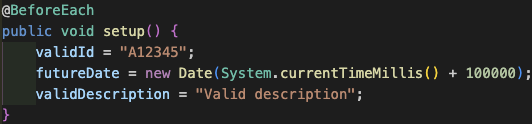
**Experience Writing the JUnit Tests**

Writing these tests was straightforward. I used @BeforeEach to set up common data so I didn’t need to repeat code. The tests have clear names that describe what they check, like:

This makes it easy to understand the purpose of each test. I kept the tests clean and focused on one thing at a time.

**Ensuring Technically Sound Code**

I kept tests efficient by reusing setup code with @BeforeEach:



This avoids duplication and keeps tests simple and easy to maintain.

**Reflection**

I used unit testing to check each class and method in isolation, making sure they do what they’re supposed to do. I also used exception testing by deliberately giving bad input to see if the code correctly throws errors. These tests are automated, repeatable, and fast, which is great for catching problems early on.

**Software Techniques Not Used**

I didn’t use integration testing, which checks how different parts work together, or system testing tests the whole application end-to-end. I also skipped performance testing and user acceptance testing since this project focused on backend logic without a UI. Those other tests are important for bigger or later stage projects.

**Practical Uses and Implications**

Unit and exception testing are perfect for early development and catching bugs quickly in individual pieces. Integration and system testing matter more when your app has many parts that interact or a user interface. Performance testing is great when you expect lots of users or heavy loads. User acceptance testing makes sure the app actually meets user needs before release.

**Assess Adopted Mindset**

I worked carefully to test both expected and edge cases, knowing the validation rules could have tricky corner cases. For example, appointment dates can’t be in the past, so I tested that carefully with different date values. That helped me catch bugs that could be easily overlooked otherwise.

**Limit Bias**

To avoid bias, I treated the code like a black box; I only cared about inputs and outputs, not how the code was written inside. Since I was testing my own code, it was important to write tests that challenged it pretty thoroughly, like adding invalid data on purpose, to make sure errors were caught. That helped me avoid the “it works because I wrote it” mindset.

**Importance of Discipline**

Being disciplined means not rushing or skipping tests even when under pressure. Cutting corners risks bugs slipping through and costs more time fixing them later on. For example, skipping validation tests could allow bad data to cause crashes in production. I plan to avoid tech debt by writing thorough tests upfront, regularly reviewing coverage, and updating tests as the code evolves.