Collin Mann

CS320

Project 2 Submission

**Summary**

**Unit Testing Approach for Each Feature**

For this project, I created and tested the Contact, Task, and Appointment services using JUnit 5. My approach was to write individual tests that directly matched each requirement for the object classes and service methods. Each test focused on specific inputs or actions, so I could clearly tell which requirement was being verified and if something failed, I’d know exactly why.

**Alignment With Software Requirements**

My testing strategy was tightly aligned with the requirements. For example, the Contact object required the phone number to be exactly 10 digits and not null. I wrote this test to confirm that constraint:

assertThrows(IllegalArgumentException.class, () -> {

new Contact("123", "John", "Doe", null, "123 Main St");

});

This kind of targeted validation was used across all services, checking ID length, null inputs, and updating fields only when allowed. I also tested success paths to ensure valid data worked correctly.

**Defending JUnit Test Quality (Coverage %)**

All my test classes reached over 80% code coverage. I confirmed this by checking that every method, including edge cases like invalid inputs, was directly exercised in the test suite. Since I used in-memory data and had no dependencies like databases, it was easier to test all logic thoroughly.

**Experience Writing JUnit Tests**

Writing JUnit tests helped me think more deeply about the behavior of my code. At first, it felt repetitive, but I realized that each test gave me more confidence in the stability of the app. One example is how I structured my service update tests, short and focused:

assertTrue(service.updatePhone("1", "0987654321"));

assertEquals("0987654321", contact.getPhone());

This pattern helped me write clean, efficient tests that ran quickly and clearly told me if something broke.

**How I Ensured Technical Soundness**

I stuck to good testing habits like testing for both valid and invalid data. I always assumed something might fail and tried to prove it wouldn’t. For instance, in the appointment tests, I made sure no appointment could be scheduled in the past using:

new Date(System.currentTimeMillis() - 1000);

If my code didn’t throw an exception, I’d know something was off.

**How I Ensured Efficiency**

To keep my tests efficient, I avoided unnecessary duplication and made sure each test had a clear purpose. I didn’t test the same logic twice unless the situation was different. Also, I kept setup simple: creating objects right in the test method when possible.

**Reflection**

**Software Testing Techniques Used**

The main techniques I used were black-box testing and boundary value testing. Black-box testing helped me focus on what the methods were supposed to do without worrying about internal logic. Boundary testing was especially helpful for string length limits (like 10-character IDs or 50-character descriptions). I pushed inputs to those edges to make sure limits were enforced.

**Techniques Not Used**

I didn’t use mocking or integration testing. Since these services didn’t rely on external systems or data, unit testing was enough. I also didn’t use white-box testing because the code was straightforward and didn’t have complicated branches or loops.

**Practical Uses of These Techniques**

For simple, self-contained services like this project, black-box and boundary testing are fast and effective. But in a bigger app, mocking might be needed to simulate a database or API, and integration testing would help make sure modules work together correctly. Each technique has its place depending on the complexity of the system.

**Mindset During Testing**

I approached testing with caution and curiosity. I always asked myself, “What could go wrong?” For example, I tested null inputs and overly long strings even if I thought they wouldn’t happen. It helped me catch any missing validations early. I also paid attention to how changes in one field could affect the whole object.

**How I Limited Bias**

Since I wrote the code and the tests, I tried to “break” my own code by thinking like someone else. I included tests I hoped wouldn’t fail, just to be sure. I didn’t assume that everything would work just because I wrote it.

**Discipline and Avoiding Technical Debt**

I took the mindset that cutting corners now would cost me later. Skipping a null check or length test could mean trouble down the line if a user input slipped through. To avoid technical debt in the future, I plan to stick with TDD or at least write tests right after coding. I also plan to use code reviews and test coverage tools to keep myself honest.

**Conclusion**

This project showed me how important unit testing is not just for catching bugs, but for building confidence in the code. I feel like my testing process helped me deliver reliable services that meet the customer’s expectations. Going forward, I’ll keep using these techniques to build high-quality, well-tested code.