Fabiola Soyoy

**Summary and Reflections Report**

In Project One, I created unit tests for three features of a mobile application: the contact service, the task service and the appointment service. Each service was developed and tested individually using the JUnit framework to verify that the application met customer requirements. My unit testing approach was to write both success and failure cases for each functionality, which helped confirm that validation and logic were implemented correctly.

For the Contact Service, my unit tests focused on validating the contact fields and ensuring the service properly added and deleted contacts. I used assertEquals() to verify that contact values were correct after creation and assertThrows() to catch invalid inputs. One key test ensured that a null contact ID triggered an exception:

For the Task Service, I tested creating tasks with valid input, handling duplicates, updating names and descriptions and deleting tasks. Each test reflected the requirements such as ensuring that the name did not exceed 20 characters and the description did not exceed 50. For example:

For the Appointment Service, I tested for valid appointments and rejected those scheduled in the past or with overly long descriptions. A future date was created using System.currentTimeMillis() to ensure the test passed:

Each test was directly based on the software requirements provided. For example, appointment dates were validated to ensure they were not in the past. The following constructor in Appointment.java enforced this:



I made sure my test suite confirmed this by attempting to create an appointment with a past date:



I know that my JUnit tests were effective because they covered both valid and invalid inputs, producing correct results and catching exceptions where expected. All test cases ran successfully in Eclipse’s JUnit window and no functionality was left untested. I structured the tests to include meaningful names and grouped similar ones together to improve readability and maintainability.

Initially, I made mistakes like repeating object instantiations or forgetting to assert values. But over time, I learned to set up reusable variables using @BeforeEach with JUnit 5 which cleaned up my code and ensured each test had a consistent starting point (GeeksforGeeks, 2024) like this:



To ensure my code was technically sound, I included strict validation rules in each class constructor. For example, in Contact.java I made sure the phone number was exactly 10 digits, not null and contained only numeric characters. This was implemented with the condition:



I verified this logic in my JUnit test by attempting to create a contact with an invalid phone number ensuring the exception was thrown as expected.

To make the code efficient, I used a HashMap to store objects like contacts, allowing quick access and updates using unique IDs. I also used the @BeforeEach annotation in my test classes to initialize service instances, which helped avoid repeated code and kept my tests consistent and clean.

The main testing technique I employed was unit testing, which focuses on testing individual components in isolation. I tested each class (Contact, Task, Appointment) independently and validated all methods through assertions. I did not use integration testing or system testing in this project. Integration testing would involve verifying how components interact together, while system testing validates the complete functionality of an application with user input and output. But since this project had no UI or database unit testing was sufficient.

In real world development, unit testing is vital for catching bugs early. When writing enterprise software, testing every class and method before deployment ensures fewer production issues. If this mobile app had launched without testing, invalid tasks or appointments could easily break the app or frustrate users.

I adopted a cautious mindset by thoroughly validating inputs and ensuring my tests didn’t just confirm that "something worked" but that it failed when it was supposed to. This mindset is especially important in fields where accuracy matters. This perspective helped me appreciate how even a small logic bug could break functionality.

It’s easy to miss your own mistakes, so I created test cases designed to challenge my assumptions. I included edge cases I didn’t expect to fail, just to be thorough. This approach helped me catch issues in my validation logic early. Bias can definitely be a concern when you're only testing scenarios where everything goes right, so I made sure to test beyond just the happy path.

Cutting corners might seem easier at first, but skipping tests only leads to bugs and frustration later. I made a point to stay disciplined by writing all the required tests and ensuring they were accurate. For example, even though checking for name length might seem straightforward it was important to include a test to meet the project requirements and validate proper error handling. In TaskTest.java, I added the following test to confirm that task names longer than 20 characters were correctly rejected:



Moving forward, I plan to document all requirements and confirm that each is tested before release. This habit will help me avoid technical debt and maintain high quality software.

Completing this project helped me understand the importance of unit testing and writing efficient, readable test cases. I improved my ability to analyze requirements, convert them into meaningful test scenarios and reflect on how testing techniques apply to real world systems. I will carry these lessons forward and continue to focus on writing sound, tested code that meets user expectations and minimizes bugs.

**References**

GeeksforGeeks. (2024). *JUnit 5 @BeforeEach annotation with examples*. <https://www.geeksforgeeks.org/junit-5-beforeeach/>