Summary and Reflections Report

Emily Cruz Gutiérrez

Introduction

This report provides an analysis of the testing strategies used in Project One, where contact, task, and appointment services were developed for a mobile application. It reflects on the unit testing approach, the effectiveness of JUnit tests, and the mindset adopted as a software engineer to ensure quality and efficiency.

Summary

Unit Testing Approach

1. Contact Service Testing: I used a positive and negative test approach. For example, one test ensured that a valid contact object was created successfully, while another checked the system's behavior when invalid data was provided.
2. Task Service Testing: The testing focused on validating task creation and updates. I implemented edge case tests to confirm behavior when tasks had overlapping deadlines using assertThrows.
3. Appointment Service Testing: I focused on boundary testing to ensure that appointments could not be scheduled outside business hours.

Alignment with Software Requirements

The unit tests were aligned with the software requirements. For example, requirements specified that appointments could not overlap, and the task service requirement to validate deadlines was tested through various scenarios.

Quality of JUnit Tests and Coverage Percentage

The quality of the JUnit tests was ensured by achieving high code coverage. Coverage reports indicated 85-90% coverage, demonstrating that most logical branches were tested.

Experience Writing JUnit Tests

I ensured technical soundness by testing both normal conditions and edge cases. For example:

```java
@Test
void testInvalidAppointment() {
   assertThrows(IllegalArgumentException.class, () ->
      appointmentService.addAppointment(null));
}
```

I ensured efficiency by reusing setup code with @BeforeEach to avoid redundancy:

```java
@BeforeEach
void setUp() {
   appointmentService = new AppointmentService();
}
```

Reflection

Testing Techniques

1. Unit Testing: Isolates small components to ensure each behaves correctly.
2. Boundary Testing: Checks input range limits to validate behavior.
3. Exception Testing: Confirms the system handles invalid inputs gracefully.

Other Testing Techniques Not Used

1. Integration Testing: Ensures modules work together correctly.
2. Performance Testing: Evaluates system speed and responsiveness.
3. Regression Testing: Verifies new changes do not break existing functionality.

Mindset and Approach as a Tester

I employed caution by considering interdependencies within the services. For example, shared data fields between contact and appointment services required consistent validation.

To limit bias, I took breaks before reviewing my tests and requested peer reviews. For example, a peer review identified the need to test null inputs, leading to the following additional test:

```java
@Test
void testNullDeadline() {
   assertThrows(NullPointerException.class, () ->
      taskService.addTask("Task 1", null));
}
```

Importance of Commitment to Quality

Cutting corners can lead to technical debt, complicating future development. I plan to follow test-driven development (TDD) practices to maintain quality and prevent technical debt.

Conclusion

In Project One, I applied various testing strategies to ensure the quality of the services developed. My JUnit tests aligned with software requirements and achieved high coverage. Moving forward, I aim to maintain a disciplined approach to testing and embrace peer reviews.