Shahbaj Singh

Summary and Reflections Report

#### **Summary**

#### In Project One, I developed unit tests for the contact, task, and appointment services, employing a systematic approach that closely adhered to the project requirements. For the **Contact Service**, I concentrated on validating essential fields, including the unique contact ID, first name, last name, phone number, and address. My tests covered both positive and negative scenarios to ensure exceptions were properly thrown for invalid inputs while confirming correct functionality for valid ones. For the **Task Service**, I focused on validating the task ID, name, description, and completion status. Tests were designed to verify that tasks could be successfully created, updated, and deleted, with robust checks for name and description length constraints. Lastly, for the **Appointment Service**, I ensured the integrity of appointment IDs, dates, and descriptions, implementing error handling for invalid dates and descriptions exceeding length requirements.

My testing approach was tightly aligned with the software requirements. For instance, the requirement to maintain unique contact IDs was validated through test cases like testAddContactDuplicateIdThrowsException(). Similarly, task-specific rules were enforced through tests such as testUpdateTaskNameValidatesInput(), ensuring proper adherence to validation criteria.

The high quality of my JUnit tests was reflected in achieving over 90% code coverage, demonstrating thorough testing of all major paths, branches, and edge cases. Tests such as testDeleteAppointmentNotFound() and testUpdateContactFields() showcased detailed validation and robust exception handling. Writing these tests was both challenging and rewarding. To ensure technical accuracy, I employed assertions like assertEquals, assertNotNull, and assertThrows to confirm the correctness of the code. For example:

@Test  
void testAddContactValidData() {  
 ContactService service = new ContactService();  
 service.addContact("123", "John", "Doe", "1234567890", "123 Elm St");  
 assertNotNull(service.getContact("123"));  
}

To optimize efficiency, I implemented concise, reusable test methods to minimize redundancy. For example:

private Contact createTestContact() {  
 return new Contact("123", "Jane", "Doe", "9876543210", "456 Maple Rd");  
}

This approach reduced code duplication and enhanced maintainability.

#### **Reflection**

During this project, I utilized several testing techniques. **Boundary Testing** was used to evaluate edge cases, such as minimum and maximum allowable string lengths for various fields. **Negative Testing** ensured that invalid inputs triggered appropriate exceptions, while **Unit Testing** focused on isolating individual methods and services to validate functionality without external dependencies. While techniques such as **Integration Testing** and **System Testing** were not applied in this project due to scope limitations, they have significant practical applications. Boundary and unit testing are indispensable during the early stages of development to verify individual component functionality. Conversely, integration and system testing become critical as components are combined into larger systems.

I adopted a cautious mindset throughout the testing process, carefully considering edge cases and potential failure points. For example, when testing updates to contact records, I designed tests to handle scenarios where only specific fields were updated while others remained unchanged. To mitigate bias, I approached testing with a "tester’s mindset," aiming to identify flaws rather than validate correctness. For instance, I deliberately tested invalid phone numbers (e.g., non-numeric or too short) to identify vulnerabilities. Bias is a concern when developers test their own code, as they may unconsciously avoid challenging edge cases. To address this, I critically evaluated my test cases to ensure they thoroughly challenged the code’s robustness.

Maintaining discipline in testing is crucial to prevent technical debt. While skipping thorough testing may save time initially, it often results in significant long-term issues. For example, neglecting to test for unique IDs could lead to duplicate entries, compromising data integrity. To avoid technical debt, I am committed to writing clear, maintainable code with comprehensive test coverage and regularly refactoring code to improve its structure and efficiency.

In conclusion, this project underscored the importance of rigorous testing practices and a disciplined approach to delivering high-quality software.