# **1. SUMMARY**

## **A. Unit Testing Approach**

### **i. Alignment to Software Requirements**

For each feature—Contact, Task, and Appointment—I ensured that my unit testing approach strictly adhered to the defined software requirements. For example, The **Contact** feature required that no contact ID exceed ten characters and that names and phone numbers meet specific formats. My unit test testConstructor\_withValidData\_shouldCreateObject confirmed valid input creation, while testSetPhoneNumber\_withInvalidFormat\_shouldThrowException checked input validation. For the **Task** service, the requirement was that tasks have unique IDs, valid future dates, and non-empty names. I tested scenarios like testSetTaskName\_withNull\_shouldThrowException to ensure that null names were not accepted. For the **Appointment** service, a critical requirement was that the appointment date must not be in the past. The test testSetDate\_withPastDate\_shouldThrowException directly enforced this rule, validating that the logic aligned with the requirement.

### **ii. Effectiveness of JUnit Tests (Coverage and Quality)**

To evaluate the effectiveness of my unit tests, I used code coverage tools within Eclipse. Each service had 90% or higher method and branch coverage. For instance, in the **TaskServiceTest**, each CRUD operation had dedicated test cases covering both valid and invalid paths. Additionally, assertions such as assertEquals, assertThrows, and assertNull helped confirm both successful functionality and failure handling. For example, in testDeleteTask\_withNonExistingId\_shouldThrowException, I confirmed proper exception handling. This thorough coverage demonstrates that my tests were effective in capturing a broad set of scenarios, including edge cases.

## **B. Experience Writing The Junit Tests**

### **i. Technically Sound Code (Examples)**

To ensure technical soundness, I followed the Arrange-Act-Assert pattern in all my tests. In the **AppointmentServiceTest**, for example:

Appointment appointment = new Appointment("12345", futureDate, "Checkup");

appointmentService.addAppointment(appointment);

Appointment retrieved = appointmentService.getAppointment("12345");

assertEquals(appointment, retrieved);

This structure clearly separates setup, action, and verification. I also validated input constraints rigorously using assertThrows, such as:

assertThrows(IllegalArgumentException.class, () -> new Contact(null, "Test", "1234567890"));

These practices ensured that the tests were grounded in verifying logic and behavior directly tied to the application’s constraints.

### **ii. Efficient Code (Examples)**

Efficiency was achieved by using constants for repeated values and reusing setup logic. For example, in **TaskServiceTest**, the @BeforeEach method established a reusable valid Task object:

@BeforeEach

void setUp() {

taskService = new TaskService();

validTask = new Task("T001", "Homework", futureDate);

}

This minimized repetition and made the tests easier to maintain. Furthermore, the use of utility methods like Calendar.getInstance().add(Calendar.DATE, 1) for future date generation helped avoid hardcoded values, enhancing code adaptability.

# **2. REFLECTION**

## **A. Testing Techniques**

### **i. Software Testing Techniques Used**

In this project, I employed **unit testing** as the primary technique. It involved isolating each class (Contact, Task, Appointment) and testing its behavior in a controlled environment. I tested both valid and invalid inputs to ensure robustness. I also followed the **white-box testing** approach by testing internal logic paths, such as validating constraints on string lengths, date comparisons, and ID uniqueness.

### **ii. Software Testing Techniques Not Used**

I did not use **integration testing** or **system testing**, as this project focused on individual service units. Integration testing, which checks how multiple components work together, would be more appropriate in a larger system context. I also didn’t use **automated UI testing** (like Selenium), as there was no frontend.

### **iii. Practical Uses and Implications**

Unit testing is essential for catching bugs early and supports agile development cycles. White-box testing ensures thorough code path verification, particularly useful in business logic-heavy services like Task and Appointment. On the other hand, integration testing is vital in projects with multiple interacting modules, such as connecting Contact and Appointment services to a central scheduler. UI testing is necessary when user experience and interface functionality are in focus, such as web applications.

## **B. Mindset**

### **i. Caution in Testing**

Throughout the project, I adopted a mindset of caution, especially in enforcing constraints on input data. For example, I made sure that appointment dates could not be set in the past and that contact IDs were always unique. The test testAddAppointment\_withDuplicateId\_shouldThrowException is a reflection of this cautious mindset. I understood that overlooking such details could cause future bugs that are difficult to trace.

### **ii. Limiting Bias**

To limit bias, I wrote test cases before fully implementing some methods. This "test-first" mindset allowed me to focus on functionality expectations without assuming the correctness of my own code. For instance, before implementing deleteContact, I wrote testDeleteContact\_withNonExistingId\_shouldThrowException to define expected behavior first. I also validated my code by peer reviews, ensuring that I was not solely trusting my implementation.

### **iii. Commitment to Quality and Avoiding Technical Debt**

Discipline was key in maintaining code quality. I avoided shortcuts like skipping null checks or writing tests with only happy paths. Instead, I ensured all edge cases were covered. For example, in the **ContactTest**, I included tests for maximum string lengths, null values, and format violations. Avoiding technical debt means not leaving untested paths or vague error handling. As a future practitioner, I plan to document edge cases, maintain high code coverage, and integrate continuous testing practices to prevent technical regression.