### **Summary and Reflections Report**

#### **Summary**

#### **a. Unit Testing Approach**

For this project I created unit tests for the ContactService, TaskService, and AppointmentService features. Each one relied on memory data structures and followed specific functional requirements. I used JUnit to validate the integrity and Behavior of each feature by writing unit tests targeting both expected inputs and edge cases.

i. **To what extent was your approach aligned to the software requirements?** For each class, my tests were made based on the functional requirements. For the Contact class, I verified that the contact ID was non-null and no longer than 10 characters, the phone number was exactly 10 digits, and that the names and addresses followed their specified length constraints. Similarly, for the Task class, I enforced validation on the task ID, name, and description, ensuring length and null checks. For Appointment, tests ensured appointments IDs were unique, the date was not in the past, and that descriptions did not exceed 50 characters.

The service classes (Contact Service, TaskService, and appointmentService) were tested to confirm they supported adding, updating, and deleting objects with valid input. Each method was validated to ensure that invalid data could not bypass the rules from their associated classes.

ii**. Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were effective based on the coverage percentage?**My tests were effective because they covered the essential rules for each object and service. I used assetThrows to check for expected errors with invalid input, and assertEquals to confirm correct behavior with valid data. Both standard use cases and edge cases were tested, providing through coverage. This gave me confidence that the code met all required specifications.

#### **b. Experience Writing the JUnit Tests**

**i. How did you ensure that your code was technically sound?**To ensure my unit tests were technically sound, I aligned each one of the specific requirements from the projects and verified both expected and accepted behavior. I used meaningful assertions and followed best practices in test isolation and readability. For example, in AppointmentTest.Java, I wrote:

AssertThrows(IllegalArgumentException.class, () -> {

New Appointment(“12345”, new Date(0), “Past Appointment”);

});

This test checks whether the constructor throws an exception if an appointment date is set in the past, which is a direct requirement. I also verified that getters return correct values:

Appointment appt = new Appointment(“12345”, new Date(System.currentTimeMillis() + 100000), “Dentist Visit”);

assertEquals(“Dentist Visit”, appt.getDescription());

These kinds of tests demonstrate soundness by ensuring the object behaves exactly as expected under valid and invalid inputs. By checking each field and expected result, I ensured that my tests validated the accuracy of the implementation

ii. **How did you ensure that your code was efficient?**My tests were clean, simple and each one focused on just one function like adding a task or checking an invalid phone number, this made the test suite fast to run and easy to understand. I also avoided repeating the same code in multiple tests.

### **Reflection**

#### **a. Testing Techniques**

i. **What were the software testing techniques that you employed in this project?**

The two main testing strategies I implemented were boundary testing and exception testing. Boundary testing included values on the edge of valid input. For example, a name exactly 10 characters long or an address exactly 30 characters long. Exception testing ensured that violations of input rules (like null values or invalid dates) threw the correct exceptions and prevented invalid object creation or updates.

ii. **What are the other software testing techniques that you did not use for this project?**

I did not use parameterized testing, which could have allowed me to test multiple input combinations more efficiently. I also did not use mocking because the application didn't include external services or dependencies that would require simulated behavior.

iii. **For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.**

The testing techniques I used are particularly useful for small to medium scale applications with strict input validation. Boundary and exception testing are key for data integrity and user input handling. However in larger systems, using parameterized testing and mocking would improve reusability and scalability of test code, reduce redundancy, and also allow for better modular testing of components with external dependencies.

#### **b. Mindset**

i. **Caution as a Software Tester**Throughout the project I approached testing and development with a high degree of caution. As a software tester, I recognized that even small mistakes in data validation could lead to major issues, especially since each of the services(ContactService, TaskService, and AppointmentService) handled user provided information. I made sure to carefully check the interrelationships between objects and service. For example when writing tests for the AppointmentService, I had to ensure that appointment dates could not be in the past. Overlooking this logic could allow invalid appointments to be created, which would affect downstream scheduling and workflows. By taking the time to understand each requirement in detail and writing tests for both common and uncommon scenarios, I was able to ensure my code behaved predictably and safely.

ii. **Bias in Review**To avoid bias in reviewing my own code, I approached the project as if someone else had written it. I didn't assume that the constructor logic or service methods would work correctly just because I had implemented them. Instead I wrote tests designed to fail to validate exception handling. For example, passing null values or strings that exceeded the allowed character limits. This helped me verify that the proper exceptions were thrown and that my validation logic was sound. I also tried to imagine how an end user might accidentally issue the system and test for those edge cases. This mindset helped reduce confirmation bias which is common when developers test their own code and unconsciously avoid scenarios that might expose flaws.

iii. **Discipline and Avoiding Technical Debt**Discipline played an important role in maintaining the quality of both omy code and my tests. It would have been easy to skip certain checks like validating description length or handling null inputs once the basic functionality appeared to work. However I made sure to write thorough test cases for every rule. For example, I ensured the Task description followed its 50 character limit even though it might seem minor. This level of thoroughness helps prevent technical debt and ensures that the software can be maintained and scaled safely in the future