**Summary and Reflections Report**

**1. Summary**

**a. Unit Testing Approach for the Three Features**

1. **Alignment with Software Requirements**  
   For each of the three services (Contact, Task, and Appointment), my unit tests were designed to ensure they met all the requirements listed in the project. For instance, in the **ContactService**, I made sure that the contact ID was unique and unchangeable. This was crucial as it was explicitly mentioned in the requirements. My unit tests included cases where I tried to modify the contact ID, which correctly failed, demonstrating that the system behaved as expected. Similarly, for the **AppointmentService**, I created tests to confirm that the appointment date couldn't be set in the past, a key part of the project's requirements. By addressing these specific requirements, I ensured my tests aligned closely with what was needed.
2. **Defending the Quality of JUnit Tests**  
   I ensured my JUnit tests covered all the required operations, such as adding, updating, and deleting entries. For example, in **TaskServiceTest**, I validated how the system handled tasks with names longer than 20 characters and descriptions exceeding 50 characters, which were outside the specified limits. This comprehensive test coverage, which included both valid and invalid inputs, gave me confidence that my tests effectively caught potential errors.

**b. Experience Writing JUnit Tests**

1. **Ensuring Technically Sound Code**  
   To make sure the code was technically sound; I wrote specific tests for each service. For instance, in **ContactTest**, I validated that all fields were within the required character limits and ensured that null or invalid inputs triggered appropriate errors. By focusing on these details, I ensured that the code behaved correctly under all circumstances, as expected by the project’s specifications.
2. **Ensuring Code Efficiency**  
   I aimed to keep my tests efficient by avoiding redundancy. In **AppointmentServiceTest**, I grouped related test cases, such as those testing the appointment date, to reduce duplication while maintaining thorough coverage. This approach helped streamline the testing process and ensured that the tests ran efficiently without sacrificing coverage.

**2. Reflection**

**a. Testing Techniques**

1. **Techniques Used**  
   For this project, I primarily relied on **unit testing**. This was the best approach because I needed to isolate each service (Contact, Task, and Appointment) and ensure that their specific functionalities worked as intended. I also used **boundary testing** in cases like the **TaskService** to ensure the system handled edge cases, such as tasks with overly long names or descriptions.
2. **Techniques Not Used**  
   I didn’t use **integration testing**, which would have been useful to check how the three services (Contact, Task, and Appointment) interacted together. Additionally, I didn’t implement **performance testing** to check how the services would perform under heavy load. While these were not required for this project, they would be helpful in a more complex system where different components rely on one another.
3. **Practical Uses and Implications**  
   The unit and boundary testing I used are common in many software development projects, particularly for catching small bugs early. However, integrating unit testing with **integration testing** and **performance testing** is crucial in larger systems, especially when different services or components are heavily interconnected. These techniques ensure that systems not only function as expected but also scale well and perform efficiently under load.

**b. Mindset**

1. **Employing Caution**  
   I was cautious throughout the project by carefully reviewing the requirements and making sure all edge cases were accounted for. For example, in **AppointmentService**, I made sure to test not only valid appointment dates, but also scenarios where a date in the past was entered. This attention to detail ensured that the system behaved correctly in all situations.
2. **Avoiding Bias**  
   When reviewing my code, I worked hard to avoid assuming that everything would work perfectly. Instead, I tried to approach the system as a user who might make mistakes or enter incorrect data. This helped me write better tests that focused on catching errors rather than assuming the system would always behave correctly.
3. **Discipline and Commitment to Quality**  
   I remained focused on delivering quality code by sticking to best practices and thoroughly testing each feature. Rather than cutting corners, I made sure to follow all the guidelines and test requirements. Moving forward, this commitment to quality will continue to guide my approach in future projects, as ensuring thorough testing is essential to delivering reliable software.