**Summary and Reflections Report (Project Two)**

1. **Summary**
   1. **Describe your unit testing approach for each of the three features.**
      1. For each contact, task, and appointment services, I used unit testing to make sure everything worked correctly. My approach followed the software requirements by creating specific tests for each requirement. For example, in the contact service, I tested that the contact ID was unique, not null, and no longer than 10 characters, just as the requirements specified. Similarly, in the appointment service, I checked that the appointment date wasn't in the past and that the description stayed within 50 characters.
      2. I believe my JUnit tests were effective because they covered all the important aspects of the features. The tests included both positive and negative scenarios, meaning I tested for both correct and incorrect inputs. For example, in the task service, I tested with both valid and invalid task IDs, names, and descriptions.
   2. **Describe your experience writing the JUnit tests.**
      1. To make sure my code was technically sound, I included checks in both the classes and the tests. For example, in AppointmentTest.java, I wrote the following code to check for invalid appointment dates:  
         assertThrows(IllegalArgumentException.class, () -> {  
          new Appointment("12345", pastDate, "Doctor's appointment")});  
         This code made sure that invalid inputs like past dates were correctly rejected, which kept the system reliable.
      2. I made my code efficient by writing tests that focused on specific tasks without unnecessary steps. For example, in TaskServiceTest.java, I tested the update function with a simple approach:  
         service.modifyTask("12345", "NewName", "New description.");  
         assertEquals("NewName", task.getName());
2. **Reflection**
   1. **Testing Techniques**
      1. The main testing technique I used in this project was unit testing. Unit testing focuses on testing small parts of the software independently to make sure each part works correctly. It involves testing specific functions or methods to make sure they behave as expected in different situations.
      2. There are other testing techniques I didn't use in this project, like integration and system testing. Integration testing checks how different parts of the system work together which makes sure they interact properly. System testing looks at the entire system to ensure it meets all the requirements and works well in real world conditions.
      3. Each of these techniques is useful depending on the project. Unit testing is great for catching problems early in development, especially in projects where high reliability is important. Integration testing is important for projects with many interacting components, as it makes sure they work together. System testing is crucial for making sure the entire application works correctly before deployment.
   2. **Mindset**
      1. While working on this project, I focused on being cautious and detailed. As a software tester, it was important to understand the complexity of the code and how different parts interacted. For example, when testing the appointment service, I was especially careful with date validation, knowing that handling past dates incorrectly could cause problems in a real world application. By being cautious, I was able to avoid potential issues.
      2. To limit bias when reviewing my code, I approached testing with a mindset of finding flaws instead of assuming the code was correct. As a developer, it's easy to trust your own work, which can lead to missing mistakes. To counter this, I deliberately looked for ways the code might fail. For example, I tested the appointment date logic by intentionally using incorrect dates to see how the system would react. This approach helped me identify and fix issues that I might have overlooked if I hadn't questioned my own assumptions.
      3. Being disciplined in maintaining quality is essential as a software engineer. Cutting corners in writing or testing code can lead to technical debt, which can be costly and time consuming to fix later. I plan to avoid technical debt by following coding best practices and carefully reviewing code before considering it complete. For instance, I will continue to apply strict validation checks, as I did with the appointment dates, to make sure all inputs are handled correctly and prevent unexpected issues down the road.