### **Summary and Reflections Report**

1. **Summary**
   1. Describe your unit testing approach for each of the three features.
      1. To what extent was your approach **aligned to the software requirements**? Support your claims with specific evidence.  **<** In my unit testing approach, I followed a test-driven development style to make sure that my tests were aligned with the software requirements for the Contact, Task, and Appointment services. I validated that fields such as contact ID, task ID, and appointment ID were immutable and adhered to the length constraints of 10 characters. I also made sure that descriptions in both the Task and Appointment classes adhered to their limits of 50 characters and that invalid descriptions threw exceptions. For the Appointment class, I checked that the appointment date could not be in the past, per the requirement, and any attempts to assign a date in the past threw an IllegalArgumentException. In the TaskServiceTest class, I wrote tests such as testUpdateInvalidTaskDescriptionLength(), which verified that trying to update a task description to more than 50 characters resulted in an exception. In the AppointmentServiceTest class, the testUpdateInvalidDescriptionLength() method made sure that description updates that exceed the specified limit triggered the correct validation error. **>**
      2. 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? **<** I made sure that the quality of my tests was strong by covering edge cases, invalid inputs, and valid inputs. My code coverage was 86.3%, meaning that a significant majority of my code was tested. This coverage included both normal operational paths and exception cases. I also used assertions like assertTrue(), assertThrows(), and assertEquals() to verify expected outcomes in both success and failure scenarios. For instance, I covered different lengths of inputs, null checks, and boundary conditions, making sure that all parts of the software worked as expected. In the ContactServiceTest class, the test testAddDuplicateID() verified that adding a duplicate contact ID correctly threw an exception. The AppointmentServiceTest class included a test testUpdateInvalidDescriptionLength(), making sure that descriptions exceeding the limit threw the appropriate exception.  **>**
   2. Describe your experience writing the JUnit tests.
      1. How did you ensure that your code was **technically sound**? Cite specific lines of code from your tests to illustrate. **<** I focused on validating the correctness of my code by reusing the validateInput() method to ensure consistency across all the services in validating fields. I also ensured immutability for certain fields like contactID, taskID, and appointmentID as required by the specifications. This method was used across different classes to maintain consistent validation logic, thus avoiding potential issues from repetitive or incorrect validation.

private boolean validateInput(String item, int length) {

return (item != null && item.length() <= length);

} **>**

* + 1. How did you ensure that your code was **efficient**? Cite specific lines of code from your tests to illustrate. **<** I improved the efficiency of the services by switching from an ArrayList to a HashMap in the ContactService and AppointmentService classes. This allowed for faster lookups and deletions by using containsKey() to quickly check for existing objects.I also refactored repetitive code into helper methods like validateInput(), reducing code duplication and improving maintainability. For example, by using a HashMap, the lookup time for contacts and appointments was reduced from O(n) in an ArrayList to O(1) in a HashMap. This made the code more efficient and scalable. **>**

1. **Reflection**
   1. Testing Techniques
      1. What were the **software testing techniques** that you employed in this project? Describe their characteristics using specific details. **<** I used black box testing to focus on verifying that the system met the predefined functional requirements without delving into the internal structure of the code. For example, I tested the behavior of the services by inputting valid and invalid data as outlined by the software specifications. I also used equivalence partitioning to divide input data into valid and invalid partitions and tested representative values from each. For instance, the description fields had a length limit of 50 characters, so I tested descriptions that were exactly 50 characters long to represent valid input and more than 50 characters long to represent invalid input. Another technique used was boundary value analysis to focus on testing the boundaries of the allowed input values, such as making sure that a task name of 20 characters was valid while 21 characters was not. This technique was particularly useful in making sure that my system handled edge cases correctly. **>**
      2. What are the **other software testing techniques** that you did not use for this project? Describe their characteristics using specific details. <One software technique that I did not use was decision testing. It could have been implemented more thoroughly, particularly for areas like conditional branching in the update methods. This technique would verify that all possible decision points such as success, failure, and exception-throwing paths were fully tested. Another technique is regression testing that could have been applied after code refactoring. This would make sure that changes in one part of the code, such as switching from an ArrayList to a HashMap, didn’t unintentionally break functionality in other parts. >
      3. For each of the techniques you discussed, explain the **practical uses and implications** for different software development projects and situations. **<** Equivalence partitioning allowed me to test a wide range of input conditions without creating redundant tests. For example, testing one invalid phone number string, like a phone number of only 5 digits, was sufficient to confirm that any other phone number that failed the validation would also throw an exception. Boundary value analysis is important for identifying off-by-one errors and making sure that edge cases are handled. For instance, verifying that a task description exactly 50 characters long was valid, while a description of 51 characters threw an exception, makes sure the system worked correctly at the boundary limits. **>**
   2. Mindset
      1. Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ **caution**? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims. **<** Throughout the project, I approached testing with caution, particularly around validation and exception handling. I made sure to test every possible input scenario to prevent edge cases from causing failures in production. This was particularly important for ensuring that immutability and validation logic, such as making sure that the appointment date wasn't in the past, worked as expected. **>**
      2. Assess the ways you tried to limit **bias** in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims. **<** To limit bias, I focused on making sure that my tests didn’t just confirm that the "happy path" worked, but that invalid data was properly rejected as well.I also took your feedback to heart by testing both valid and invalid inputs for fields like description and making sure that length constraints were respected.By using assertThrows() for exception handling, I confirmed that my code raised the correct exceptions when faced with invalid data, ensuring unbiased testing of both success and failure scenarios. **>**
      3. Finally, evaluate the importance of being **disciplined** in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims. **<** Discipline was key in making sure that I didn’t cut corners, especially when it came to edge case testing and input validation. In this project, I maintained best practices by using helper methods for validation and efficient data structures like a HashMap to ensure both readability and performance. Avoiding technical debt is important because poorly written or untested code can cause significant issues down the road. By adhering to best practices in testing, I avoided accumulating technical debt and made sure my code was both efficient and maintainable. **>**