**Reflection on Unit Testing Approach for Contact and Task Service**

1. **Testing Approach and Alignment with Requirements**I designed my testing method to validate the requirements through JUnit tests which checked both proper and improper input values. The testing process included scenarios where requirements were deliberately violated in order to confirm proper identification and handling of errors by the code. The testing approach helped verify that the implemented system followed all the documented software requirements.  
   For instance:  
   A screenshot of a computer program

   AI-generated content may be incorrect.  
   This test ensures that the input meets the requirement and also checks to see an in-case situation where each string field fails to meet requirement and throws an exception.
2. **Defending the Quality of JUnit Tests**My JUnit tests maintained a high coverage percentage (100%), which proved their effectiveness because they tested every possible path within the code. Clear assertions such as **assertEquals** validate expected behavior, but it can also make sure that throwable exception can occur if input doesn’t meet certain conditions. Another notable assertion, **assertThrows**, helps check if a certain piece of code throws the right exception. This is important for testing situations where things go wrong and making sure that the error-handling processes function properly. These were shown in my test cases on the Appointment Test file:  
   A screenshot of a computer screen

   AI-generated content may be incorrect.

Test Coverage Percentage **–** 100%  
A screenshot of a computer

AI-generated content may be incorrect.

1. **Ensuring Technically Sound Code**I applied technical soundness principles which included boundary testing and proper error handling and coding standards. The JUnit tests for the addTask() method conducted tests with inputs that make sure that a task can be added, with a fail-safe included.

For example:  
A computer screen shot of a program code

Description automatically generated

This test ensures that a task can be added successfully, and the system correctly duplicates task IDs and throws an appropriate exception.

1. **Ensuring Code Efficiency**Test case performance was maximized when the approach prevented unnecessary test logic repetition while setup methods handled repeated test actions. I implemented using precise assertions (such as **assertEquals, assertTrue, assertThrows**) to avoid redundant checks as shown below:

A computer screen shot of a program code

Description automatically generated