**Module Four Journal Reflection**

My testing approach was closely aligned with the software requirements provided for both the contact service and the task service. Each unit test was designed to ensure that all specified constraints and functionalities were rigorously verified.

For the contact service, I focused on testing the constraints set by the software requirements, such as ensuring that fields like **contactId**, **firstName**, **lastName**, **phone**, and **address** were validated correctly. For instance, the **testInvalidPhone()** method in ContactTest.java includes tests for various invalid phone numbers to ensure that they meet the requirement of being exactly 10 digits long:

**assertThrows(IllegalArgumentException.class, () -> {**

**new Contact("12345", "John", "Doe", "12345", "123 Main St");**

**});**

For the task service, I aligned my tests with the requirements by verifying that the **taskId**, **name**, and **description** fields adhered to length and nullability constraints. For example, the **testInvalidTaskName()** method in **TaskTest.java** checks that an exception is thrown for a task name that exceeds twenty characters:

**assertThrows(IllegalArgumentException.class, () -> {**

**new Task("123", "This name is definitely way too long for the task", "Task Description");**

**});**

The overall quality of my JUnit tests for both the contact service and task service can be defended by the high coverage percentage achieved. The tests were effective as they covered all possible paths and edge cases in the application logic. For example, the JUnit test coverage reports showed above 80% for both classes, indicating comprehensive test coverage. This high coverage percentage provides confidence that the core functionality and constraints of the application are thoroughly validated.

To ensure technically sound code, I incorporated exception handling and validations that align with the requirements. In **TaskServiceTest.java**, the **testAddDuplicateTask()** method confirms the duplicate tasks cannot be added, illustrating robust error handling:

**assertThrows(IllegalArgumentException.class, () -> {**

**service.addTask(task);**

**});**

This line of code demonstrates how the test checks for logical soundness by ensuring exceptions are thrown as expected when requirements are violated.

Efficiency was achieved by minimizing redundancy and optimizing the logic within the unit tests. For instance, using parameterized tests where possible allowed me to test multiple cases with minimal code. Also, the use of assertions effectively checks conditions without unnecessary additional logic, as shown in the following example from **TaskTest.java**:  
**assertNotNull(task);**

**assertEquals("12345", task.getTaskId());**

**assertEquals("Task Name", task.getName());**

**assertEquals("Task Description", task.getDescription());**

These assertions are concise and verify that the task attributes are initialized correctly, contributing to efficient testing.

My approach to unit testing in both the contact and task services ensured alignment with software requirements, robust quality verification through high coverage, and efficient testing through concise and well-structured test cases. This process provided a solid foundation for developing reliable and maintainable software components.