**Summary and Reflections Report**

**Summary**

**Unit Testing Approach:**

1. **Task Class:**
   * In my approach to testing the Task class, I focused on validating the input constraints when creating and updating task objects. I aimed to ensure that all necessary validations were enforced according to the specified requirements, such as limits on the length of the taskId, name, and description.

|  |
| --- |
| @Test  public void testTaskIdLength() {  Task task = new Task("12345678901", "Task Name", "Task Description");  assertNull(task.getTaskId());  }  @Test  public void testTaskIdBoundary() {  Task task = new Task("123", "Task Name", "Task Description"); // Minimum length  assertNotNull(task.getTaskId());  }  @Test  public void testTaskIdNull() {  Task task = new Task(null, "Task Name", "Task Description");  assertThrows(IllegalArgumentException.class, () -> task.setTaskId(null));  } |

1. **TaskService Class:**
   * For the TaskService class, I concentrated on testing the correct management of tasks, including the ability to add, delete, and update tasks. My goal was to verify that the service handled these operations as expected, even when dealing with edge cases, such as adding tasks with duplicate IDs.

|  |
| --- |
| @Test  public void testDeleteTask() {  TaskService service = new TaskService();  Task task = new Task("1", "Task1", "Description1");  service.addTask(task);  assertTrue(service.deleteTask("1"));  assertNull(service.getTask("1"));  }  @Test  public void testTaskServiceInteraction() {  TaskService service = new TaskService();  Task task = new Task("1", "Task1", "Description1");  service.addTask(task);  assertTrue(service.getTaskList().contains(task));  } |

**Alignment with Software Requirements:**

* I ensured that my testing approach was closely aligned with the software requirements by carefully considering the constraints and behaviors specified in the project guidelines. For example, the limits on the length of taskId, name, and description were directly reflected in my tests. This alignment was crucial in confirming that the implementation met the customer's needs.

**Overall Quality of JUnit Tests:**

* I believe that my JUnit tests were effective based on the high coverage percentage achieved. The tests I wrote covered various scenarios, including both valid and invalid inputs, which helped me identify and fix potential issues early in the development process.

**Experience Writing JUnit Tests:**

* My experience writing the JUnit tests was both challenging and rewarding. To ensure that my code was technically sound, I followed best practices in test writing, such as focusing on small, isolated tests that target specific functionalities. Additionally, I made sure my tests were efficient by minimizing redundant setups and directly addressing the core functionality I wanted to verify.

**Reflection**

**Testing Techniques Employed:**

* Throughout the project, I employed several software testing techniques, including Black-Box Testing and Boundary Value Analysis. Black-Box Testing allowed me to focus on testing the functionality based on inputs and outputs, without being influenced by the internal code structure. Boundary Value Analysis helped me test the edges of input ranges to ensure that the system handled boundary cases correctly.

**Other Testing Techniques:**

* There were also some testing techniques I did not use in this project, such as White-Box Testing and Integration Testing. White-Box Testing involves examining the internal logic and paths of the code, which could have provided deeper insights into the code coverage. Integration Testing, which focuses on interactions between different system components, might have been useful if the project had involved more complex interactions between different services or modules.

**Practical Uses and Implications:**

* Each testing technique has its practical uses depending on the project. For instance, Black-Box Testing is particularly useful for validating user-facing features, while White-Box Testing is essential for ensuring complete coverage of internal logic. Integration Testing is vital in projects where multiple components need to work together seamlessly.

**Mindset:**

**Caution:**

* I approached the testing process with caution, recognizing the importance of thoroughly checking edge cases and potential failure points. This careful approach helped me identify and resolve issues that might have gone unnoticed if I had taken a more superficial approach to testing.

**Bias:**

* To limit bias in my review of the code, I made an effort to step back and consider different perspectives, including how end users might interact with the application. This was important in ensuring that the tests were not solely focused on scenarios I anticipated during development. I also recognized that if I were responsible for testing my own code without this mindset, there might be a risk of overlooking potential issues due to familiarity with the code.

**Discipline:**

* Being disciplined in my commitment to quality was crucial throughout the project. I avoided cutting corners by rigorously applying testing standards and not rushing through the testing phase. This discipline is essential for preventing technical debt, which can accumulate when quality is compromised. Going forward, I plan to continue this disciplined approach by regularly reviewing and refactoring code to maintain high standards.