Victor Udeh  
SNHU CS320  
M4-Journal  
  
Reflecting on the development and testing of both the contact service in Module Three and the task service in the current module has been an enlightening journey. Throughout this process, my primary aim was to ensure that my unit testing approach was rigorously aligned with the software requirements specified for each service. This alignment was crucial for guaranteeing the functionality and reliability of the services I developed.

**Testing Approach Alignment with Software Requirements**

My testing approach was deeply intertwined with the software requirements from the outset. For example, the task service required tasks to have unique IDs, names within 20 characters, and descriptions no longer than 50 characters. To verify these requirements, I crafted specific JUnit tests. For instance, in the TaskTest class, I implemented the `goodConstructor()` test method to ensure that a task could be successfully created with valid parameters:

@Test

void goodConstructor() {

String id = "1";

String fullName = "Victor Udeh";

String description = "This is a good description";

Task tempTask = new Task(id, fullName, description);

assertEquals(id, tempTask.getUniqueID());

assertEquals(fullName, tempTask.getName());

assertEquals(description, tempTask.getDescription());

}

This method not only checks that a task object is correctly instantiated but also validates the adherence of the object's properties to the specified length constraints, thus directly aligning with the software requirements.

**Quality of JUnit Tests**

The overall quality of my JUnit tests for both services can be defended on the basis of coverage percentage. Through the use of tools like JaCoCo, I ensured that my tests achieved a high coverage percentage, touching upon critical paths and edge cases in the code. For example, I tested edge cases such as creating a task with a description that exceeds the maximum length:

@Test

void invalidConstructor() {

String description = "This description is too long ...";

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

new Task("1", "Task Name", description);

});

}

This approach ensured that my tests were effective in validating the robustness of the service functionalities against the requirements.

**Ensuring Technical Soundness**

To ensure the technical soundness of my code, I paid close attention to the structure and logic of my test cases. I focused on writing clear, concise, and meaningful tests that directly corresponded to the functionality being tested. For instance, the `testDeleteTask()` method in the `TaskServiceTest` class demonstrates this approach:

@Test

void testDeleteTask() {

TaskService tempTask = new TaskService();

tempTask.addUniqueTask("Task Name", "Description");

tempTask.deleteTasks("0");

assertFalse(TaskService.tasks.containsKey("0"));

}

This test method succinctly verifies the delete functionality by adding a task and then deleting it, ensuring the task is no longer present in the collection. It directly reflects the technical soundness by targeting the core functionality of the task deletion process.

**Ensuring Code Efficiency**

Efficiency in my code was achieved by optimizing both the logic within my service methods and the structure of my tests. I ensured that my code executed with minimal redundancy and maximum clarity. For example, the use of setup and teardown methods in JUnit tests helped maintain a clean state for each test, avoiding cross-test interference and ensuring that each test was as efficient and isolated as possible:

@AfterEach

void tearDown() {

TaskService.tasks.clear();

}

This line of code, by resetting the state after each test, exemplifies efficiency in preparing the test environment, ensuring no residual data affects subsequent tests.

In conclusion, my reflection on the unit testing of the contact and task services has reinforced the importance of a methodical approach that aligns closely with software requirements. By carefully designing tests that not only cover the breadth of functionalities but also delve into edge cases, I have honed a testing strategy that underscores the effectiveness, technical soundness, and efficiency of my code. This reflective practice will undoubtedly be beneficial as I prepare for the comprehensive summary and reflections report due in Module Seven.