Jacob Barker

CS 320

Professor Sam Haney

30 March 2025

**Module Four Journal:**

**Milestones Reflection**

Over the past two modules, I developed and validated both a contact service and a task service. My testing methodology was primarily guided by the software specifications, and I drew upon YouTube tutorials and examples from previous SNHU courses to structure my JUnit tests.

I ensured that my tests were in accordance with each specific requirement. For instance, in the Task class, the requirements stipulate that the task ID cannot exceed 10 characters or be null, that the task name cannot exceed 20 characters or be null, and that the description cannot exceed 50 characters or be null. In my TaskTest.java, I have tests such as:

java

CopyEdit

@Test

void testTaskIdTooLong() {

assertThrows(IllegalArgumentException.class, () -> {

new Task("12345678901", "Do Homework", "Complete math assignments");

});

}

This confirms that creating a task with an 11-character ID immediately triggers an exception. Similar tests verify that the name and description fields behave exactly as required. As observed in my coverage report, each line that checks these conditions is triggered by at least one test case.

The effectiveness of my JUnit tests is assessed based on both coverage percentage and the number of negative and positive cases tested. The coverage percentage for TaskTest.java is approximately 76.7%, while TaskServiceTest.java achieves around 94.5%. These metrics indicate that nearly all lines in both the Task class and the TaskService class are exercised by my tests. Uncovered lines usually handle rare corner cases or exceptions. For instance, in TaskServiceTest.java, I verify that adding a duplicate ID fails correctly.

java

CopyEdit

@Test

void testAddTaskDuplicateId() {

Task task1 = new Task("12345", "Buy Groceries", "Milk, Eggs, Bread");

Task task2 = new Task("12345", "Pay Bills", "Utilities");

service.addTask(task1);

assertThrows(IllegalArgumentException.class, () -> {

service.addTask(task2);

A screenshot of a graph

AI-generated content may be incorrect.

Given that this negative path is tested, the coverage tool registers that exception path as utilized, thereby enhancing my confidence in the enforcement of the requirement for unique task IDs.

To maintain technical sound code, I tested edge cases. For example, to ensure that null or out-of-bounds fields are rejected, I rely on lines such as:

java

CopyEdit

@Test

void testSetNameInvalid() {

Task task = new Task("OK123", "Valid Name", "Valid Description");

assertThrows(IllegalArgumentException.class, () -> {

task.setName(null);

});

}

This ensures that the setter fails if the name is null. These targeted tests clarify that my constructors and setters do not allow invalid data.

My tests also show that I’m not doing unnecessary calculations or making the code too complicated. For example, in Task.java, the constructor quickly checks for invalid input and throws an exception. This “fail fast” approach is more efficient than putting every operation inside multiple if/else statements. Additionally, the test methods themselves are concise, each focusing on a specific requirement or edge case. This approach keeps each test short, clear, and easy to maintain while ensuring that all requirements are met.

Overall, my approach to both the contact and task services includes unit tests that match the stated requirements. The coverage metrics show that I’m testing most of the code paths, and the targeted assertions confirm that I handle both valid and invalid data scenarios. Although I did encounter a few issues with Eclipse configuration that sometimes required restarting the entire project, the end result is a set of tests that aligns with the specified requirements and produces high-quality, efficient code.