Yoel Marquez

9-24-24

CS-320 Module 4-2

Journal

In Module Three, I worked on the contact service, and in this module, I developed the task service. Both services involved creating Java classes and validating their functionality using JUnit tests. Since I’m still getting comfortable with these concepts, the process helped me understand how unit testing plays a crucial role in verifying that the software meets specific requirements.

When I began working on the unit tests, I wasn’t entirely sure how to structure them to match the requirements. However, as I progressed, I learned how important it is to ensure that my tests directly align with the given specifications. For example, in the task service, the task ID was required to be unique, no longer than 10 characters, and not null. I wrote tests to confirm these rules, like the one below:

assertThrows(IllegalArgumentException.class, () -> {

new Task(null, "TaskName", "Task Description", new Date(), "In Progress", "High");

});

This test ensures that a null task ID triggers an exception, as specified in the requirements. Writing this kind of test helped me realize how important it is to account for every condition mentioned in the project guidelines.

For the contact service in Module Three, I applied the same approach by checking conditions such as field length and null values for contact ID, first name, and last name. By the end of the project, I felt more confident about aligning my tests with the software requirements.

Since I’m still learning about testing, I wasn’t sure how to judge the quality of my JUnit tests at first. I relied heavily on running multiple test cases to ensure that every requirement was covered. For example, in the task service, I made sure to write tests for all the required fields—task name, description, and ID—to validate that they functioned as expected.

assertEquals("UpdatedName", taskService.getTask("12345").getName());

This simple test confirmed that the task name was updated correctly, ensuring that my tests captured essential functionality. By running tests with various inputs, I learned how effective JUnit can be in finding errors early on.

As I’m still getting used to writing clean code, I found it helpful to focus on using simple and reliable techniques to ensure technical soundness. One key element I learned was using the assertThrows method to ensure that exceptions were handled correctly, especially for invalid inputs. This helped me make sure the code wasn’t just technically sound but also followed the error-handling requirements.

For instance, testing whether tasks could be updated without affecting the task ID taught me about immutability in Java:

assertThrows(UnsupportedOperationException.class, () -> {

task.setId("newID");

});

This test confirmed that the task ID remained unchanged after initialization, which is a requirement I wasn’t fully aware of when I first started coding.

Efficiency wasn’t something I was thinking about initially, but as I worked more with the task service, I realized how important it is to ensure that the code performs well. I used a HashMap to store the tasks because I learned that it allows fast access to tasks by their ID. Writing tests for task addition and deletion helped me understand why efficiency matters.

taskService.addTask(task);

taskService.deleteTask("12345");

assertNull(taskService.getTask("12345"));

This test made sure that tasks were quickly deleted from the HashMap, reinforcing that the structure I used was efficient for handling tasks in the application.

Through developing the contact service and task service, I’ve learned the importance of aligning tests with requirements, ensuring technical soundness, and thinking about efficiency. Writing JUnit tests was initially challenging, but I now understand how valuable it is for identifying issues early and ensuring the software behaves as expected. These experiences have given me a better foundation to build on for future projects.