**CS 320 Project 2**

For project 1, I employed a systematic approach to unit testing to ensure that all the features met the proper requirements. The primary testing focused on Contact Service, Task Service, and Appointment Service. When it comes to my unit testing with the contact and contact service class, I included basic CRUD operations which I verified through the Junit tests for each operation I wrote tests to verify that a contact could be added, retrieved, updated, and deleted successfully. Specific tests ensured invalid input, like exceeding name length or missing values, threw appropriate exceptions. For task service, I focused on testing task creation, validation, and task management. I specifically tested for valid/invalid task IDs, names, and descriptions, ensuring they would adhere to the required constraints. Lastly for appointment service, like Task Service, the Appointment Service required testing the creation, retrieval, and deletion of appointments, along with validation of input, such as appointment date and description. I tested edge cases like creating appointments in the past or exceeding character limits for descriptions.

The unit tests were directly aligned with the software requirements, ensuring that input validation, constraints, and operations worked as expected. For example, in the Task Service, I specifically ensured that the task ID was no longer than 10 characters, and in the Appointment Service, the appointment date could not be in the past. These validations were integral to meeting the functional requirements of the application.

The quality of the JUnit tests can be evaluated based on the extent to which they cover all possible scenarios, including edge cases and invalid inputs. The tests included positive cases and negative cases, ensuring a comprehensive evaluation of the feature behavior. For example, in the Task creation tests, both valid tasks and invalid tasks were tested, ensuring that the system handled errors appropriately. I used coverage analysis tools to measure the effectiveness of the tests, which provided a high level of code coverage. This ensured that the tests validated most of the logic in the services, such as the creation, updating, and deletion of records.

Writing JUnit tests involved creating a range of tests that addressed both valid and invalid inputs. The tests were written in a structured and organized way to ensure clarity and maintainability. For instance, in the TaskServiceTest class, I created tests for task creation, deletion, and updating, ensuring that the code would behave as expected in all situations. The code was made technically sound by adhering to test-driven development principles, ensuring all scenarios were tested before moving to production. In specific lines, this test shows that a task is properly created and added to the service:

@Test

void testAddTask() {

TaskService taskService = new TaskService();

Task task = new Task("12345", "Name", "Description");

taskService.addTask(task);

// Ensures the task is correctly added and retrieved by its Task ID

assertEquals(task, taskService.getTask("12345"));

}

The tests were designed to minimize unnecessary operations. For instance, by ensuring that the task ID was unique before adding a new task, I avoided unnecessary duplicate checks. One of the codes I used to maximize efficiency by only performing necessary validation was:

@Test

void testAddDuplicateTaskId() {

TaskService taskService = new TaskService();

Task task1 = new Task("12345", "Name1", "Description1");

Task task2 = new Task("12345", "Name2", "Description2");

taskService.addTask(task1);

// Efficiently checks if an exception is thrown when trying to add a duplicate task ID

assertThrows(IllegalArgumentException.class, () -> taskService.addTask(task2));

}

The primary testing technique used in this project was unit testing, which focuses on testing individual units of code. This technique is essential for verifying that each component works as expected before being integrated with other parts of the application. Additionally, I used boundary testing to ensure inputs, such as task IDs or appointment dates, stayed within acceptable limits. I did not employ integration testing or system testing in this phase, as the focus was on validating individual services. Integration testing would be useful when combining different parts of the application, while system testing would be important for testing the entire application’s functionality. Unit testing is most useful in the early phases of development when validating individual components. Integration testing is beneficial when verifying that different parts of the system work together, while system testing helps ensure the entire system functions correctly from the user's perspective.

I approached the project with caution, particularly when writing tests for edge cases like invalid task IDs or past appointment dates. It was crucial to think about how users might interact with the system and ensure that the software would handle these cases correctly. I tried to limit bias by imagining how a user might use the application and by testing features under different scenarios. Testing my code after it was written also allowed me to view it from a fresh perspective, ensuring that I did not overlook potential issues. Throughout the project, I emphasized the importance of writing tests that covered all possible cases. Cutting corners in testing could lead to undetected bugs, affecting the overall user experience. To avoid technical debt, I made sure that all tests were clear, reusable, and focused on the functionality being tested. By consistently checking for invalid task IDs and appointment dates, I ensured that even edge cases were covered. This attention to detail in testing helps prevent issues in the later stages of the project, ensuring that the software remains reliable and maintainable.