**7-2 Project Two**

Summary:

For the Contact class, the JUnit tests covered various scenarios, including adding contacts, deleting contacts, and updating existing contact fields. These tests aligned with the software requirements for this project. The quality of these JUnit tests is adequate, as they provide good coverage of the contact class and contact service class, including success and error cases.

*Example from code:*

@Test

public void testAddContact() {

Contact contact = new Contact("1", "John", "Doe", "1234567890", "Sample Address");

boolean result = contactService.addContact(contact);

assertTrue(result);

}

***This test validates that adding a valid contact return ‘true’, which is one of the software requirements of the contact class.***

For the Task and Task Service classes, the JUnit tests are comprehensive, they covered the adding, deleting, and updating of tasks, which aligned with the software requirements for these classes. The corresponding JUnit tests for the classes are of high quality, they provide a strong coverage of the test cases and include boundary cases. These tests ensure that the Task and Task Service class can handle the scenarios described in the requirements effectively.

*Example from code:*

@Test

public void testAddTask() {

Task task = new Task("1", "Task 1", "Description 1");

taskService.addTask("1", "Task 1", "Description 1");

List<Task> tasks = taskService.getTasks();

assertEquals(1, tasks.size());

}

*This test verifies that adding valid tasks results in an updated list, which aligns with the requirement of adding tasks to task service class.*

The appointment and appointment service classes had JUnit tests that covered adding appointments, deleting appointments, and checking for duplicate appointments, which aligned with the software requirements for these classes. These JUnit tests were good as they provided adequate coverage of both the appointment and appointment service classes.

*Example from code:*

@Test

public void testAddAppointment() {

Date appointmentDate = new Date(System.currentTimeMillis() + 100000); // Future date

Appointment appointment = new Appointment("A123456789", appointmentDate, "Sample Description");

appointmentService.addAppointment(appointment);

List<Appointment> appointments = appointmentService.getAppointments();

assertNotNull(appointments);

assertEquals(1, appointments.size());

}

*This test validates adding an appointment result in the appointment list being updated, which follows the requirement of the appointment service being able to add a new appointment.*

My experience writing JUnit tests started with this class, I have learned they are helpful in ensuring that the code is behaving according to the software requirements by verifying the functionality of the classes based on the requirements. For example, testing each method with both valid and invalid input, using assertions to validate these results. I ensured that my code was efficient by testing only the necessary code paths, which focused on critical functionality while avoiding redundant tests. By creating concise and clear test cases I was able to cover all essential aspects of the classes.

Reflection:

Some of the software testing techniques I employed in this project were Black box testing and boundary value testing. Black box testing is done by examining the behavior of a system without looking at the internal structure or implementation. The focus is testing the functionality based on its requirements, for instance, when testing the ‘addContact’ method, I examined whether it correctly handles adding contacts and returns the expected results based on the requirements. Similarly, in the ‘TaskService’ class, I tested the ‘addTask’, ‘deleteTask’, and ‘updateTask’ methods based on their external behavior. I did this by verifying that tasks can be added, deleted, and updated without delving into the internal logic. As for boundary testing, it’s a technique that focuses on testing values at the boundary of input domains to help identify potential issues. For instance, in the contact class, I used boundary testing on the ‘firstName’ and ‘lastName’ fields by including test cases where the input string are exactly 10 characters long. This ensures that the contact class correctly handles values at the upper boundary of the allowed length. One of the testing techniques not employed in my project was load and performance testing as it really didn’t seem to apply for this specific project.

As a tester, I tried to exhibit caution by validating the behavior of the classes according to their requirements. I considered various input scenarios and boundary conditions to avoid overlooking potential issues in the classes. It was important to appreciate the complexity of the code and its interrelationships, as a change in one part of the code could impact other parts. I made sure my tests considered these interdependencies, especially in the case of updating fields. To limit bias in my classes, I made sure to approach testing objectively by focusing on verifying compliance with requirements. I did not make assumptions about the code's behavior but relied on test cases. Bias can be a concern if you are responsible for testing your own code, as you might unintentionally overlook issues or assume certain behaviors. This can be solved by independent testing by another party, which can help identify issues you might miss. By using comprehensive test cases to demonstrate your commitment to quality by ensuring the correctness of your software components, such as the ‘TaskService’ and ‘ContactService’. Avoiding shortcuts and conducting code reviews is crucial to maintaining code quality and minimizing technical debt.