Juan Rodriguez

CS 320 Project Two

In developing the application, I created three critical features: the contact, task, and appointment services. Throughout Project One, I employed a comprehensive unit testing approach to ensure these services' functionality, reliability, and efficiency. This report delves into my unit testing strategy, experience writing JUnit tests, and reflections on the process.

For the contact service, I meticulously tested each method in the `ContactService` and `Contact` classes to validate their behavior. This approach was meticulously aligned with the software requirements, ensuring that contacts could be added, deleted, and updated as expected. For instance, consider the test case below:

@Test

public void testAddContact() {

    ContactService contactService = new ContactService();

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

    contactService.addContact(contact);

    assertEquals(1, contactService.getContacts().size());

}

In this test, I verify that adding contact results in the expected size of the contacts list provides concrete evidence of alignment with the requirements. Similar to the contact service, I rigorously tested the `TaskService` and `Task` classes, ensuring correct behavior for adding, deleting, and updating tasks. Here's an example:

@Test

public void testDeleteTask() {

    TaskService taskService = new TaskService();

    Task task = new Task("T1", "Task 1", "Description for Task 1");

    taskService.addTask(task);

    taskService.deleteTask("T1");

    assertEquals(0, taskService.getTasks().size());

}

This test validates that deleting a task results in an empty task list, showcasing the effectiveness of my testing approach. For the appointment service, I continued the trend of systematic testing, ensuring that appointments could be managed accurately. Here's a snippet from one of the tests:

@Test

public void testAddAppointment() {

    AppointmentService appointmentService = new AppointmentService();

    Appointment appointment = new Appointment("1234567890", new Date(), "Test Appointment");

    appointmentService.addAppointment(appointment);

    assertNotNull(appointmentService.getAppointment("1234567890"));

}

This test verifies that an appointment added to the service can be retrieved successfully, assuring correct functionality.

To ensure the technical soundness of my tests, I meticulously designed test cases covering various scenarios, including edge cases and error conditions. Consider the following example from the contact service tests:

@Test

public void testAddContactNullId() {

    ContactService contactService = new ContactService();

    assertThrows(IllegalArgumentException.class, () -> {

        contactService.addContact(new Contact(null, "John", "Doe", "1234567890", "Sample Address"));

    });

}

In this test, I verify that attempting to add a contact with a null ID results in an `IllegalArgumentException` being thrown, ensuring robust error handling.

My testing approach prioritized efficiency, and I aimed to minimize redundancy and optimize test coverage. For instance, I avoided duplicating test cases for similar scenarios in the task service tests by reusing existing test methods with different input parameters. Additionally, boundary value analysis helped identify potential issues at input extremes, ensuring comprehensive test coverage.

While effective, I did not utilize mutation testing in this project. This technique involves introducing small changes to the source code and checking whether the tests detect these changes. While powerful, it can be time-consuming. Boundary value analysis is a versatile technique applicable to various software development projects. They help identify defects early in the development cycle, ensuring robust and reliable software.

Adopting a cautious mindset was crucial to test each feature and consider all possible scenarios thoroughly. Appreciating the complexity and interrelationships of the code being tested helped identify potential areas of failure. For example, understanding how appointments were stored and retrieved aided in designing compelling test cases.

Consciously limiting bias in my code review was essential to approaching testing objectively and focusing on functionality verification. Bias would indeed be a concern if I were responsible for testing my own code, as familiarity might lead to overlooking potential issues. For instance, I might unconsciously skip testing specific paths I assume are correct. Maintaining discipline and commitment to quality is paramount for a software engineering professional. Cutting corners when writing or testing code can lead to technical debt, resulting in future issues and increased development costs. By prioritizing thorough testing, code reviews, and refactoring, I plan to deliver bug-free code.