**MINNU JOHN**

**06/24/2024**

**Summary and Reflections Report**

**Unit Testing Approach**

1. Contact Service:

- Approach: I implemented unit tests to validate the functionality of adding, deleting, and updating contact information. Tests were created to cover both valid and invalid inputs.

- Alignment to Requirements: My approach was closely aligned with the software requirements. For instance, the requirements specified that contact names must be non-null and non-empty, and the tests included scenarios for null and empty names.

- Evidence: A specific test case for the Contact Service was:

@Test

public void testAddContactWithNullName() {

Exception exception = assertThrows(IllegalArgumentException.class, () -> {

contactService.addContact(null, "Doe", "1234567890");

});

assertEquals("First name cannot be null", exception.getMessage());

}

2. Task Service:

- Approach: I focused on testing the creation, update, and deletion of tasks, ensuring tasks have valid titles and descriptions.

- Alignment to Requirements: The approach ensured tasks met all specified requirements, such as title length and non-null descriptions.

- Evidence: A sample test for the Task Service was:

@Test

public void testUpdateTaskWithEmptyTitle() {

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

Exception exception = assertThrows(IllegalArgumentException.class, () -> {

taskService.updateTask("1", "", "New Description");

});

assertEquals("Title cannot be empty", exception.getMessage());

}

3. Appointment Service:

- Approach: Tests were developed to ensure appointments could be created, updated, and deleted with valid date and time ranges.

- Alignment to Requirements: My tests adhered to the requirements, ensuring appointments were not set in the past and end times were after start times.

- Evidence: An example test for the Appointment Service was:

@Test

public void testCreateAppointmentWithInvalidDateRange() {

Exception exception = assertThrows(IllegalArgumentException.class, () -> {

appointmentService.createAppointment("1", LocalDateTime.now(), LocalDateTime.now().minusHours(1));

});

assertEquals("End time must be after start time", exception.getMessage());

}

**Quality of JUnit Tests**

- Effectiveness: The quality of the JUnit tests was measured by the code coverage percentage, which was consistently above 90% for each service. This high coverage indicates that the tests effectively checked a wide range of scenarios and edge cases.

- Coverage Evidence: For example, in the Contact Service, tests covered various scenarios including null inputs, valid inputs, and boundary cases:

@Test

public void testDeleteContact() {

contactService.addContact("John", "Doe", "1234567890");

boolean result = contactService.deleteContact("1234567890");

assertTrue(result);

}

**Experience Writing JUnit Tests**

- Technical Soundness: Ensured by verifying each function against multiple scenarios, including edge cases. For example:

@Test

public void testUpdateContactWithValidData() {

contactService.addContact("John", "Doe", "1234567890");

boolean result = contactService.updateContact("1234567890", "Jane", "Doe", "0987654321");

assertTrue(result);

}

- Efficiency: Achieved by minimizing redundant code and using setup methods to initialize common objects:

@BeforeEach

public void setUp() {

contactService = new ContactService();

contactService.addContact("John", "Doe", "1234567890");

}

**Reflection**

**Testing Techniques**

1. Techniques Employed:

- Black Box Testing: Focused on input-output validation without considering internal code structure. This technique was suitable for testing the services' public methods and ensuring they met the specified requirements.

- White Box Testing: Employed to ensure all code paths and logic branches were tested. This involved writing tests to cover conditional statements and loops within the methods.

2. Other Techniques:

- Integration Testing: Not used as the focus was on unit testing individual services. This technique would be valuable for testing the interaction between different services.

- System Testing: Also not used, but crucial for validating the entire application flow from end to end, ensuring all components work together as expected.

3. Practical Uses and Implications:

- Black Box Testing: Useful for validating user-facing functionalities and ensuring compliance with user requirements.

- White Box Testing: Critical for identifying logical errors and ensuring comprehensive code coverage.

- Integration Testing: Essential for complex applications with multiple interacting components.

- System Testing: Important for verifying the complete application, especially in production-like environments.

**Mindset**

1. Caution in Testing:

- Complexity Appreciation: Recognizing the complexity of interrelated methods and their potential impact on each other was crucial. For instance, modifying the contact information could affect task assignments and appointments linked to the contact.

- Example: Ensuring that updating a contact’s phone number didn't inadvertently affect task assignments linked by the phone number required cautious and comprehensive testing.

2. Limiting Bias:

- Code Review Practices: Employed peer reviews to get an unbiased perspective on the tests and code.

- Example: Asking a colleague to review the test cases helped identify missed edge cases or logical flaws.

3. Commitment to Quality:

- Avoiding Technical Debt: By writing comprehensive tests and adhering to coding standards, I aimed to minimize future maintenance issues.

- Example: Ensuring all methods had corresponding tests with high coverage helped detect potential bugs early, reducing the likelihood of accumulating technical debt.

**Conclusion**

Submitting this summary and reflections report has allowed me to critically analyze and document my approach to unit testing for the mobile application project. The use of various testing techniques, along with a disciplined mindset towards quality, has ensured that the delivered product meets the required standards and is robust against potential issues.