Project Two

Brian Chmura

CS 320

For each of the three features (contact, task, and appointment services) in Project One, I adopted a comprehensive unit testing approach. For the contact service, I designed tests to ensure proper creation, updating, and deletion of contacts. Similarly, for task and appointment services, the tests focused on verifying the correct execution of related functionalities. The unit testing approach was meticulously aligned with the software requirements, ensuring that the critical functionalities of the mobile application were thoroughly tested.

The overall quality of the JUnit tests can be defended through a combination of factors. Firstly, the tests achieved high coverage percentages by addressing various scenarios, edge cases, and user interactions. Code coverage analysis tools indicated that the majority of the codebase was exercised by the tests, providing confidence in the robustness of the testing suite. Additionally, frequent code reviews and collaboration with team members helped in identifying potential gaps in test coverage, ensuring a well-rounded approach.

In writing the JUnit tests, I focused on maintaining technically sound and efficient code. For technical soundness, I adhered to best practices, ensuring clear and expressive test names, using relevant assertions, and structuring tests to be easily maintainable. An example of this is evident in the contact service tests, where I validated the creation of a contact with specific attributes:

javaCopy code@Test  
public void createContact\_ValidAttributes\_Success() {  
 Contact contact = new Contact("John Doe", "[john.doe@example.com](mailto:john.doe@example.com)");  
 contactService.createContact(contact);  
 assertEquals("John Doe", contact.getName());  
 [assertEquals("john.doe@example.com](mailto:assertEquals("john.doe@example.com)", contact.getEmail());  
}

Efficiency was ensured by avoiding unnecessary redundancy in test code. In the task service tests, I maintained efficiency by reusing setup code for different test cases:

javaCopy code@Before  
public void setUp() {  
 taskService = new TaskService();  
 user = new User("user123”  
}  
 I employed various testing techniques such as boundary value analysis, equivalence partitioning, and exploratory testing to identify defects and ensure the reliability of the application. These techniques allowed me to cover a wide range of scenarios, ensuring that the software behaved as expected under different conditions. However, I did not extensively use mutation testing and chaos engineering for this project.

Mutation testing and chaos engineering could be beneficial for introducing faults and validating the system's resilience. These techniques are particularly useful in identifying weaknesses in error-handling mechanisms and understanding how the system behaves under stressful conditions.

As a software tester, I adopted a cautious mindset, understanding the complexity and interrelationships of the code. For instance, in the appointment service tests, I considered scenarios where overlapping appointments could occur, testing the system's ability to handle such complexities:

javaCopy code

@Test  
public void scheduleAppointment\_OverlappingAppointments\_ExceptionThrown() {  
 // Test logic for overlapping appointments  
 // ...  
 assertThrows(AppointmentConflictException.class, () -> appointmentService.scheduleAppointment(appointment));  
}  
 Limiting bias in code review involved maintaining objectivity and avoiding assumptions about the code's correctness. If I were responsible for testing my own code as a developer, bias could creep in, potentially overlooking flaws due to familiarity with the implementation. For instance, in the contact service tests, I actively sought diverse input values to prevent biased testing:

javaCopy code

@Test  
public void createContact\_InvalidEmail\_ExceptionThrown() {  
 Contact contact = new Contact("Invalid Email", "invalid\_email");  
 assertThrows(InvalidEmailException.class, () -> contactService.createContact(contact));  
}  
 Being disciplined in commitment to quality is crucial in software engineering. Cutting corners in writing or testing code can lead to technical debt, impacting long-term maintainability. In the task service tests, I prioritized readability and clarity:

javaCopy code@Test  
public void completeTask\_ValidTask\_TaskMarkedAsCompleted() {  
 Task task = new Task("Complete Assignment", user);  
 taskService.completeTask(task);  
 assertTrue(task.isCompleted());  
}

Avoiding technical debt involves thorough testing, adherence to coding standards, and continuous refactoring. Planning for regular code reviews, automated testing, and periodic refactoring sessions will be essential to maintain code quality and prevent the accumulation of technical debt.