Besker Telisma

CS320

Summary and Reflections Report

For Project One, I employed a systematic unit testing approach tailored to validate each specific functionality detailed in the requirements. Here's how I approached unit testing for each service:

1. **Contact Service:**
    - Tests validated object creation, field validation, and operations such as add, delete, and update. For instance, testAddContact ensured contacts could be added uniquely without duplication (assertEquals(expectedContact, actualContact);).
2. **Task Service:**
    - Focused on task manipulation, including adding, updating, and deleting tasks, ensuring each operation respected service constraints (assertThrows(IllegalArgumentException.class, () -> taskService.addTask(duplicateTask));).
3. **Appointment Service:**
    - Tested date validations and uniqueness of appointment IDs. For example, testAppointmentDateConstraints checked that appointment dates were not set in the past (assertThrows(IllegalArgumentException.class, () -> new Appointment("123", pastDate, "Checkup"));).

Alignment with Software Requirements:

My approach was closely aligned with the software requirements, ensuring comprehensive coverage. For instance, in the appointment service, tests like testAppointmentConstructorValidData directly addressed the requirement that no appointment could be scheduled in the past, thus validating the application's adherence to business logic.

Quality of JUnit Tests:

The effectiveness of the JUnit tests was demonstrated by their high coverage percentage, which consistently ranged above 90% across services. This coverage ensured that nearly all code paths and error scenarios were tested, thereby minimizing the risk of undetected bugs.

The experience Writing JUnit Tests:

Writing JUnit tests was an iterative and enlightening process that not only improved my understanding of the application's functional dynamics but also sharpened my skills in anticipating potential failure points in the software.

How I ensure the test was technically sound

Technical soundness was ensured by:

- Validating inputs and outputs rigorously (assertEquals("Expected Result", service.methodCall());).
- Handling exceptions effectively (assertThrows(ExpectedException.class, () -> service.erroneousMethodCall());).

Code Efficiency:

Code efficiency was optimized by:

- Reusing setup configurations using @BeforeEach, which minimized repetitive setup code (setUp() { initializeObjects(); }).
- Using parameterized tests where applicable to cover multiple scenarios with less code.

Reflection

Software Testing Techniques:

1. Unit Testing: The primary technique used, focusing on small, manageable sections of code to ensure each function performs as expected.
2. Boundary Value Testing: Integrated within unit tests to check the robust handling at the edges of input ranges.

Other Software Techniques Not Used:

1. Integration Testing: Not used in this project but would involve testing the interactions between connected services to ensure combined operations succeed.
2. Performance Testing: Would be useful in future projects to ensure the application performs well under load.

Implications for Different software dev Projects:

- Unit Testing is universally applicable, especially beneficial early in development to catch bugs before integration.

- Integration Testing becomes critical in multi-service architectures where services depend on each other.

Mindset:

While working on Project One, adopting a cautious mindset was crucial, especially given the complexity of interrelated services. Appreciating this complexity meant thoroughly testing not just individual units but considering how an error in one unit could propagate.

To limit bias, especially as the code's author, I adopted rigorous peer review practices and iterated through test scenarios that a user might encounter, ensuring no assumptions influenced the test outcomes. Testing one's code can often lead to overlooking subtle bugs due to familiarity; hence, peer reviews are essential.

The commitment to quality is fundamental in software engineering. Cutting corners can lead to technical debt, a reduction in code maintainability, and potentially costly fixes later in the project lifecycle. As a practitioner, I plan to maintain rigorous testing standards and continuous learning to keep abreast of best practices in code quality and testing.