Reflections Report

During the development of the mobile application for our customer, I employed a comprehensive unit testing approach to ensure the quality and functionality of the contact, task, and appointment services. Each feature was thoroughly tested using JUnit tests, aligning closely with the software requirements. For contact service, I wrote JUnit tests to validate the creation, retrieval, updating, and deletion of contacts. These tests covered various scenarios, such as testing the handling of invalid input and ensuring the correct execution of CRUD operations.

For task service, JUnit tests were developed to verify the functionality of task creation, assignment, completion, and deletion. The tests examined edge cases, input validation, and the interaction between different task attributes. Lastly for appointment service, JUnit tests were created to validate the scheduling, rescheduling, and cancellation of appointments. The tests included scenarios with overlapping appointments, ensuring proper handling of conflicts. The unit testing approach was closely aligned with the software requirements. Each requirement was carefully analyzed, and test cases were designed to verify the expected behavior. For instance, in the contact service, a requirement specified that duplicate contacts should not be allowed. To meet this requirement, I wrote a JUnit test that ensured duplicate contacts were not created by checking for existing entries with the same information. The overall quality of the JUnit tests was ensured through several measures including coverage percentage. By using code coverage tools, I assessed the extent to which the JUnit tests covered the codebase. The coverage percentage was consistently high, indicating that the tests exercised a significant portion of the code. This provided confidence that critical paths and edge cases were adequately tested.

Quality was also ensured through technical soundness. The JUnit tests were carefully crafted to validate specific functionality. For example, in the TaskTest class the @Test annotation marks the test methods, assertEquals() asserts the expected values, and the @Rule annotation defines the ExpectedException rule for exception handling. Specific lines that showcase technical soundness in TaskTest.java include:

**ExpectedException rule = ExpectedException.none();** - Declaring the ExpectedException rule to handle assertion errors.

**assertEquals(ID, testTask.getId());** - Asserting that the retrieved ID matches the expected ID, ensuring correct behavior.

Code efficiency in unit tests can refer to the avoidance of unnecessary operations and the use of appropriate test data. In my tests, each test focuses on specific functionality and verifies it individually, allowing for targeted testing. For example, in the testName() method, the test sets a new name and then asserts its correctness. This approach minimizes unnecessary operations and ensures efficient testing.

In this project, I employed the following software testing techniques. Black-box testing: this technique was used to validate the externally visible behavior of the application without considering its internal structure. By focusing solely on inputs and outputs, I could test the features from a user's perspective, ensuring they functioned as intended.

Boundary testing: This technique involves testing the extreme boundaries of input values to uncover potential issues related to range, limits, or edge cases. By deliberately selecting values near or at the boundaries, I aimed to identify any unexpected behavior or vulnerabilities.

Equivalence partitioning: this technique involved dividing the input domain into equivalence classes and selecting representative test cases from each class. It helped ensure adequate coverage while minimizing the number of test cases needed. For example, when testing the contact service, I used equivalence partitioning to validate the handling of different types of input data (valid, invalid, and edge cases).

Although not employed in this project, there are other software testing techniques.

Mutation testing: this technique involves introducing intentional defects into the code to evaluate the effectiveness of the test suite in detecting those defects. It helps identify weaknesses in the test coverage by assessing how well the tests capture changes in the code.

Performance testing: this technique focuses on evaluating system performance under different load conditions. It helps identify bottlenecks, measure response times, and ensure the application can handle expected user volumes.

The choice of testing technique depends on the project's specific needs and goals. Black box testing and boundary testing are practical for most projects as they cover a wide range of scenarios. Equivalence partitioning is particularly useful when there are input variations that can be grouped into classes, enabling efficient test case selection.

Throughout the project, I adopted a cautious mindset as a software tester. It was important to appreciate the complexity and interrelationships of the code being tested to ensure comprehensive coverage. For example, in the appointment service, I considered scenarios where the cancellation of one appointment should trigger an update in related appointments, ensuring consistency across the system. To limit bias in code review, I followed established best practices. I approached the code objectively, focusing on functionality, readability, and adherence to coding standards. By treating the code as an external entity, I aimed to identify potential issues or areas for improvement without personal bias. As a software developer responsible for testing my own code, bias could be a concern. To address this, code reviews by colleagues or employing pair programming can provide fresh perspectives and mitigate potential biases. By involving others in the review process, I can gain valuable insights and ensure a more objective evaluation of my code. Being disciplined in commitment to quality is of utmost importance for a software engineering professional. Cutting corners when writing or testing code can lead to technical debt, which accumulates over time and hampers the maintainability and scalability of the software. To avoid technical debt, I ensure thorough testing, documentation, and adherence to best practices during development. By investing time and effort to write clean and efficient code, I can reduce the likelihood of bugs and facilitate future enhancements or modifications. For example, in the contact service, I included comments in the tests to provide clarity and enable easier maintenance and future updates. In conclusion, the unit testing approach I adopted for the mobile application demonstrated a strong alignment with the software requirements. The JUnit tests exhibited high quality, as evidenced by their extensive coverage and technical soundness. Employing various testing techniques, adopting a cautious mindset, and maintaining discipline in commitment to quality enabled me to deliver a reliable and robust product.