Summary and Reflections Report

During the development of the mobile application, I tried to use various unit testing approaches to ensure the reliability and functionality of each service—Contacts, Tasks, and Appointments. For the Contact Service, I utilized boundary value analysis and equivalence partitioning to test edge cases and representative values. The tests were formulated with the requirements in mind, verifying functionalities such as adding, updating, and deleting contacts. For instance, the test testAddSingleContact verified that a contact could be successfully added and retrieved with valid data. Boundary value analysis was applied to ensure the contactID field did not exceed the maximum length, such as in the test for invalid contact ID length. Similarly, for the phone field, boundary value analysis confirmed the exact length requirement. Equivalence partitioning was used for fields like firstName to test both valid and invalid inputs, covering cases where firstName is null or exceeds 10 characters.

In developing the Task Service, I adopted data-driven testing to manage different task states and transitions comprehensively. This method allowed me to cover various scenarios, including task creation, updating, and deletion. The tests were aligned with the project requirements, confirming that all functionalities were accurately tested. For example, the testAddTask method ensured tasks could be added with unique IDs, while testUpdateTaskName verified that task names could be correctly updated. Boundary value analysis and equivalence partitioning were also applied to ensure the task ID and name met the requirements. The test for task ID constraints checked the upper boundary for task ID length, while testNameConstraints ensured the name did not exceed 20 characters.

The Appointment Service combined exploratory testing with automated unit tests to test for hidden issues. The tests were designed to cover the creation, updating, and deletion of appointments, confirming alignment with the specified requirements. For example, the testAddAppointment method confirmed that appointments could be added with valid data, and the testInvalidAppointmentDate method ensured that invalid dates were correctly handled. This thorough testing approach ensured that all aspects of the appointment management functionalities were validated and met the project specifications.

The overall quality of the JUnit tests achieved over 86% code coverage. This demonstrated that the tests effectively covered most of the codebase, ensuring that most functionality and edge cases were adequately tested. The effectiveness of these tests was further validated by the detection of potential bugs during unit testing. Writing these tests involved adhering to best practices, ensuring clarity, conciseness, and technical soundness. For example, the testAddSingleContact method was written to ensure that a single contact could be added and retrieved correctly, following best practices for unit test development. Efficient test writing was achieved by minimizing setup code and reusing test data where possible, as demonstrated in the testUpdateTaskName method.

Reflecting on the project, I employed several software testing techniques, including boundary value analysis, equivalence partitioning, data-driven testing, and exploratory testing. Boundary value analysis and equivalence partitioning were particularly useful for testing input validation logic, while data-driven testing proved valuable for applications with complex state transitions. Exploratory testing allowed for the discovery of hidden bugs that scripted tests might miss. Other techniques that were not used in this project include mutation testing and static code analysis. Mutation testing involves making small changes to the code (mutations) and checking if the existing tests detect the changes. For instance, introducing a mutation like changing a conditional operator and verifying that the tests fail appropriately. Static code analysis involves examining the code for potential issues without executing it. Analyzing the codebase for vulnerabilities and adherence to coding standards maintains code quality from the early stages of development by identifying issues that might not be immediately apparent through runtime testing.

Throughout the project, I adopted a cautious mindset, understanding the complexity and interrelationships of the code being tested. Although each service was developed independently of the other, it was important to keep in mind how changes in one service could negatively impact others. Bias can be a concern when testing one's own code, as familiarity may lead to overlooking certain issues. During one of the reviews of my code I discovered edge cases I initially missed. This to me highlights the importance of diverse perspectives and different eyes reviewing code.

Maintaining discipline in the commitment to quality is essential to avoid cutting corners, which could introduce defects and cause significant issues later. For instance, instead of skipping unit tests for a last-minute change, I ensured all tests were updated and passed before submitting the code. Avoiding technical debt involved adhering to coding standards, thorough testing, and regular refactoring. This will save lots of time and effort in having to back and correct errors that could have been fixed earlier in the development process.

Sources:

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). Software testing : An istqb-bcs certified tester foundation guide - 4th edition. BCS Learning & Development Limited.

GeeksforGeeks. (2024, October 7). *Software Testing Boundary Value Analysis vs Equivalence Partitioning*. GeeksforGeeks. https://www.geeksforgeeks.org/software-testing-boundary-value-analysis-vs-equivalence-partitioning/#