Summary and Reflections Report

Felix A. Carela

Southern New Hampshire University

CS-320 Software Test, Automation, and Quality Assurance

Summary and Reflections Report

**Summary: Unit Testing Approach**

The unit testing for the Contact Service was structured to validate each specified functionality, including the uniqueness, non-nullability, and immutability of the contact ID. Tests were conducted to handle scenarios such as attempting to add contacts with duplicate IDs and ensuring appropriate exceptions were thrown, aligning with the stringent requirements set forth for data validation.

In the Task Service, unit tests focused on confirming the functionality for adding, deleting, and updating tasks, emphasizing the handling of unique and immutable task IDs, and adhering to character limits for task names and descriptions. The tests were designed to reject duplicate IDs and input exceeding the character limits by raising exceptions, thus ensuring compliance with the defined requirements.

The Appointment Service testing was aimed at verifying CRUD operations and enforcing business rules such as the rejection of past appointment dates. Tests like checking the immutability and uniqueness of appointment IDs, and ensuring no past dates are accepted, directly supported the requirements that appointment dates must be current or future-dated and each ID must be unique.

The quality of these JUnit tests was assessed primarily through coverage metrics, with a target of achieving 80% code coverage or higher. This metric is indicative of the tests covering a substantial portion of the codebase, including critical use cases and potential edge cases. Tools such as JaCoCo were used to measure coverage and identify lines of code that were not tested, assisting in refining the tests further. This coverage ensures a robust verification against the defined requirements and reduces the likelihood of defects in production. The approach to testing both expected operational scenarios and potential error conditions provided a comprehensive validation of the system's adherence to its specifications.

**Summary: Experience with JUnit Tests**

In the development of the JUnit tests for the Contact, Task, and Appointment Services, ensuring technical soundness and efficiency was essential. Technical soundness was upheld by crafting test cases that validated each requirement and edge case to ensure that the code under test behaved as expected under various conditions. For instance, in the Contact Service, the testAddDuplicateContact() method plays a critical role by including lines of code that attempt to add a contact with a duplicate ID, as shown in the test: service.addContact("1234567890", "John", "Doe", "1234567890", "123 Main St"); service.addContact("1234567890", "Jane", "Doe", "0987654321", "456 Elm St");. This action triggers an IllegalArgumentException, demonstrating the system's robustness in handling errors as expected when a duplicate ID is introduced. The corresponding assertion assertThrows(IllegalArgumentException.class, () -> { service.addContact("1234567890", "Jane", "Doe", "0987654321", "456 Elm St"); }); confirms that the exception is caught and processed correctly, verifying that the application's backend services adhere strictly to the requirement that each contact must have a unique ID and behave consistently under error conditions. This approach in the JUnit tests ensures a high level of reliability and compliance with specified requirements.

Efficiency in testing was achieved by designing test cases that maximized code coverage while minimizing redundancy. Each test case was carefully crafted to target a specific aspect of functionality, avoiding duplication of test scenarios wherever possible. For example, in the Task Service, individual tests were created to validate the addition, deletion, and updating of tasks, each focusing on a distinct aspect of the service's functionality. This approach ensured that the tests remained concise and efficient, while still providing coverage of the codebase.

By adhering to these principles of technical soundness and efficiency, the JUnit tests developed for the project were able to effectively validate the functionality of the Contact, Task, and Appointment Services, providing confidence in their correctness and reliability.

**Reflection: Testing Techniques**

In this project, I employed several software testing techniques tailored to verify the functionality and robustness of the Contact, Task, and Appointment Services within the mobile application. The primary technique used was **unit testing**, which involved writing small, independent tests for individual components or functions. This method allowed for isolating specific areas of the application to validate their correct behavior independently of other parts. The characteristics of unit testing include its focus on small units of code, which makes identifying faults easier and helps in testing specific functionalities thoroughly.

Another key technique was **boundary value analysis**, particularly useful in ensuring that all possible edge cases were covered. For example, testing the maximum and minimum acceptable lengths for contact names and task descriptions ensured that boundary conditions were properly handled. This technique is characterized by its effectiveness in identifying errors that occur at the edges of input ranges, which are common areas for bugs to appear.

While these techniques were central to my approach, there are other methodologies I did not employ extensively, such as **integration testing** or **system testing**. Integration testing involves combining individual software modules and testing them as a group, which is crucial for identifying issues that occur when units interact. System testing, on the other hand, tests the complete and integrated software to evaluate its compliance with the requirements. Both are characterized by their broader scope compared to unit testing, focusing on interactions between components and the overall behavior of the system.

Each of these techniques has practical uses depending on the project requirements and development phase. For instance, unit testing is most beneficial during the development phase to catch bugs early, while integration and system testing are crucial as the project nears completion to ensure that all parts work together seamlessly and meet the original specifications. Employing these varied techniques in appropriate development stages enhances the software's reliability and functionality, addressing different levels of the software architecture.

**Reflection: Mindset**

In this project, while developing the unit tests for the Contact, Task, and Appointment services, I adopted a detail-oriented mindset, which is crucial for a software tester. This mindset allowed me to approach each test with caution, understanding that even small segments of code could have complex interactions and implications within the larger system. For instance, testing the immutability of the contact ID not only involved checking that the ID could not be updated once set but also ensuring that such a property did not inadvertently affect the ability to update other contact attributes like the phone number or address. Recognizing these interrelationships was essential for thorough testing and ensured that the unit tests could catch potential regressions or unintended side-effects from code changes.

Another aspect of my approach was actively trying to limit bias in reviewing the code. Testing one’s own code can often lead to overlooking errors due to familiarity or assumptions about its functionality. To mitigate this, I adopted practices such regularly revisiting the test cases with fresh perspectives after some time. This approach helped in identifying oversights and improving the test coverage. For example, initial tests for the Task service might have missed edge cases for task description updates, which were later added after revisiting the test scenarios the next day.

Finally, the importance of maintaining discipline in the commitment to quality was highlighted throughout this project. In software development, cutting corners can lead to significant technical debt, making future modifications more difficult and error-prone. By ensuring comprehensive test coverage and adhering to best practices in test development, I worked to maintain high standards of quality. For example, maintaining a consistent structure for test cases and ensuring each function met its test coverage target before proceeding helped avoid technical debt. Looking ahead, continuing to prioritize these practices will be essential in my development as a software engineer, ensuring that the software not only meets current requirements but is also maintainable and scalable in the future.