Aeriel Denmark

CS 320 - Software Test Automation & QA

Lopez – Project #2

\***due Thurs 6/18/23**\*

Summary and Reflections Report

1. **Summary**
2. Unit Testing Approach

The unit testing approach I used for the Contact Service involved writing JUnit tests that covered various scenarios such as creating a new contact, updating an existing contact, and deleting a contact. I ensured that all the required functionalities specified in the software requirements were tested. The JUnit tests for the Contact Service were aligned with the software requirements as they covered all the expected behaviors and edge cases. For example, "testGetContactId" verifies that the contact ID is not null, has a length of 10 characters, and matches the expected value. This aligns with the requirement of having a contact ID of up to 10 characters.

For the Task Service, I followed a similar unit testing approach, testing the creation, updating, and deletion of tasks. I also verified the integration of the task service with the Contact Service to ensure the association between tasks and contacts was working correctly. The JUnit tests for the Task Service were aligned with the software requirements as they covered all the expected functionalities and interactions between tasks and contacts. For instance, the test "testAddAppointment" ensures that adding an appointment with the same ID throws an IllegalArgumentException, which aligns with the requirement of not allowing duplicate appointment IDs.

The Unit Testing approach for the Appointment Service involved testing the addition, deletion, and retrieval of appointments. I also tested the validation rules for appointment attributes such as appointment ID, appointment date, and description. The JUnit tests for the Appointment Service were aligned with the software requirements as they covered all the necessary functionalities and validated the expected behaviors. For example, the test "testAddAppointment" ensures that adding a null appointment throws an IllegalArgumentException, which aligns with the requirement of not accepting null appointments.

1. JUnit Tests

To ensure that my code was technically sound, I followed best practices such as writing clear and concise test cases, using appropriate assertions, and verifying the expected outcomes.

In the AppointmentTest class, for example, "testGetAppointmentID" includes assertions that verify the appointment ID is not null, has a length of 10 characters, and matches the expected value. These assertions ensure that the tested code behaves correctly and adheres to the specified requirements.

In the AppointmentServiceTest class, I used the setup method to create and add appointments to the appointment service. This approach avoids redundant code and ensures that each test starts with a consistent initial state. Additionally, I used assertions to validate the expected outcomes, ensuring that the tests efficiently check the correctness of the service's behavior.

The overall quality of my JUnit tests for all three features was expected to cover a minimum of 80%. I aimed to achieve effective test coverage by addressing different scenarios and ensuring that the critical functionality was thoroughly tested. To evaluate the effectiveness of my JUnit tests, I reviewed the test cases I created. By doing so I was able to evaluate the variety and depth of my test cases. It was important that I designed a comprehensive set of tests that would cover different scenarios, input combinations, and edge cases. By analyzing this test suite, I believe I was able to determine if I had tested various aspects of the code. A high coverage percentage indicates that most of the code was exercised, increasing confidence in the effectiveness of the tests. For all three features, I strived to achieve a coverage percentage of at least 80%. This level of coverage ensures that a significant portion of the code is tested, reducing the likelihood of undiscovered defects. By reviewing the test cases I created, I was able to verify that the critical paths, edge cases, and boundary conditions were covered, providing a high level of confidence in the quality of the tests.

1. **Reflection**
2. Testing Techniques

In this project, I used several software testing techniques. I primarily used Unit Testing to verify the individual units of code, such as classes and methods, in isolation. By testing each unit separately, I could identify and fix issues early in the development cycle. I employed integration testing to validate the interactions and collaborations between different components or services. For example, in the TaskServiceTest class, I tested the integration between the TaskService and Task classes to ensure the correct behavior of updating task attributes.

In this project there were also software testing techniques I did not use. Performance testing was not employed in this project which assess the system's performance under specific workloads and determine its responsiveness and scalability. Security testing, which focuses on identifying vulnerabilities and ensuring data protection, was also not used in this project. Techniques like penetration testing or code review for security issues were not utilized.

Unit testing is crucial for ensuring the correctness of individual units of code. It helps detect defects early, promotes code maintainability, and provides a safety net for refactoring and future changes. Unit testing is beneficial for any software development project, regardless of its size or complexity. Integration testing is essential to validate the proper functioning and interaction of different components or services. It ensures that the integrated system meets the requirements and handles various scenarios. Integration testing is particularly important in projects that involve multiple modules or services working together.

1. Mindset

While working on this project, I adopted a cautious mindset as a software tester. I recognized the complexity and interrelationships of the code I was testing, which required careful consideration of inputs, edge cases, and potential failure scenarios. For example, in the ContactServiceTest class, when testing the "testAddContact" method, I carefully designed tests to handle different scenarios, such as adding a contact with an existing ID or a null contact. This caution allowed me to identify and handle potential issues and ensure the correctness of the service's behavior.

As a software developer responsible for testing my own code, bias can be a concern. To limit bias, I employed a systematic approach and followed established testing practices. I maintained a separation of concerns between development and testing phases, which involved designing test cases based on requirements and executing them objectively. For instance, in the TaskServiceTest class, I reviewed and tested the TaskService class's methods independently, verifying their behavior against the requirements. By adhering to a structured testing process, I aimed to minimize bias and ensure a thorough evaluation of the code.

Being disciplined in my commitment to quality as a software engineering professional is of utmost importance. Cutting corners when writing or testing code can lead to technical debt, which negatively impacts the overall quality and maintainability of the software. To avoid technical debt, I followed coding best practices, employed systematic testing approaches, and maintained a high level of code coverage. For example, in the TaskServiceTest class, I covered various scenarios and performed boundary value analysis to ensure robust test coverage. This discipline ensures a solid foundation for future development and reduces the likelihood of introducing defects. As a practitioner in the field, I plan to continue prioritizing quality, following established coding standards, performing thorough testing, and actively participating in code reviews to maintain high-quality software and prevent the accumulation of technical debt.

This report summarizes my unit testing approach, experience writing JUnit tests, and reflections on testing techniques and mindset during the development of Project One. The comprehensive unit testing, alignment with requirements, technical soundness, efficiency, and adoption of a cautious mindset contribute to the overall quality of the JUnit tests and the reliability of the developed mobile application.