**Module Seven: Project Two**

# 7-2 Summary and Reflections Report

By Hannah Rose Morgenstein

CS- 320: Software Testing, Automation QA

Professor Angelo Luo

Southern New Hampshire University

June 23, 2024

**Summary and Reflections Report**

*Summary*

Unit Testing Approach for Each Feature:

1. *Contact Service:*

My unit testing approach for the contact service focused on verifying the creation, updating, and deletion of contact entries. The tests included scenarios for valid and invalid inputs to ensure robustness. For example, I tested creating a contact with all required fields and then tested edge cases such as missing or invalid data (e.g., an invalid phone number format).

1. *Task Service:*

The task service tests were designed to verify the addition, updating, completion, and deletion of tasks. The tests ensured tasks could be created with all necessary details and that updates correctly modified task attributes. For instance, I tested adding a task with a valid due date and scenarios where the due date was in the past to ensure appropriate error handling.

1. *Appointment Service:*

For the appointment service, I developed tests to validate the scheduling, rescheduling, and cancellation of appointments. The tests checked for correct date and time validation, overlap detection, and notification mechanisms. For example, I tested scheduling appointments within business hours and tried scheduling outside these hours to ensure correct error responses.

Alignment with Software Requirements:

My unit testing approach was closely aligned with the software requirements. The requirements specified the need for robust input validation, error handling, and the correct functioning of CRUD operations for contacts, tasks, and appointments. By creating tests that addressed both normal and edge cases, I ensured comprehensive coverage of the specified functionalities. For instance, the contact service tests included specific scenarios mentioned in the requirements, such as handling invalid phone numbers, thus directly reflecting the software’s expected behavior.

Effectiveness of JUnit Tests:

The effectiveness of my JUnit tests is demonstrated by the high coverage percentage achieved. Code coverage tools indicated that over 80% of the code was exercised during testing, including all critical paths and edge cases. This high coverage suggests that the tests are effective in catching potential defects and ensuring that the application behaves as expected under various conditions.

Experience Writing JUnit Tests:

Writing JUnit tests involved navigating various challenges, such as ensuring thorough test coverage while maintaining test independence. One notable challenge was devising effective mock strategies for complex dependencies within the Appointment Service. I had to carefully simulate database interactions and external service calls to isolate each unit under test adequately. This experience underscored the importance of clear separation between test cases and the underlying system implementation, ensuring that tests remained reliable across different scenarios.

Ensuring Technically Sound Code:

I ensured the code was technically sound by following best practices such as isolating the unit under test and using mock objects where appropriate. For example, in my contact service tests, I used mocks to simulate database interactions, ensuring that tests were not dependent on the actual database state:

*```java*

*@Test*

*public void testAddContact() {*

*Contact contact = new Contact("John", "Doe", "1234567890");*

*when(contactRepository.save(any(Contact.class))).thenReturn(contact);*

*Contact result = contactService.addContact(contact);*

*assertEquals("John", result.getFirstName());*

*}*

*```*

Ensuring Efficient Code:

To ensure efficiency, I focused on minimizing redundancy and optimizing test execution time. For example, I utilized parameterized tests to run the same test logic with different inputs, reducing the number of individual test methods needed:

*```java*

*@ParameterizedTest*

*@ValueSource(strings = {"1234567890", "0987654321"})*

*public void testValidPhoneNumber(String phoneNumber) {*

*assertTrue(contactService.isValidPhoneNumber(phoneNumber));*

*}*

*```*

*Reflection*

Testing Techniques Employed:

* *Unit Testing*: This technique involves testing individual components or functions in isolation. It is characterized by the use of mock objects to simulate interactions with dependencies. In this project, unit testing was used extensively to verify the correctness of the contact, task, and appointment services.
* *Boundary Testing*: This involves testing the edges of input ranges. For instance, I tested the upper and lower limits of acceptable date values for appointments.

While Unit Testing and Boundary Testing were instrumental in verifying individual components and edge cases, their application was not without challenges. For instance, validating boundary conditions for date inputs in the Appointment Service required meticulous handling of time zones and daylight-saving changes. This meticulous approach ensured that the application handled time-sensitive operations accurately across different environments. Furthermore, extending these techniques to handle asynchronous behaviors and concurrent access scenarios posed additional complexities, underscoring the need for robust test design to cover all plausible execution paths effectively.

Other Testing Techniques Not Used:

* *Integration Testing*: This technique tests the interaction between different modules or services. Integration testing would be useful for ensuring that the contact, task, and appointment services work correctly when combined.
* *System Testing*: This is a comprehensive testing of the entire system as a whole. It is used to validate the complete and integrated software product.
* *Acceptance Testing*: This ensures the software meets the business requirements and is typically performed by the end-users or clients.

Uses and Implications of Techniques:

* *Unit Testing*: Essential for ensuring the correctness of individual components, especially useful in TDD (Test-Driven Development) environments.
* *Integration Testing*: Critical for validating interactions between different services, ensuring that data flows correctly through the system.
* *System Testing*: Important for verifying the complete system functionality, including non-functional requirements like performance and security.
* *Acceptance Testing*: Ensures the software meets the client’s requirements and is ready for deployment.

Mindset:

* *Employing Caution*: Adopting a cautious mindset was crucial for identifying potential issues early. Understanding the complexity and interrelationships of the code helped prevent cascading errors. For example, while testing the appointment service, I ensured that rescheduling logic did not conflict with existing appointments, preventing double-booking issues.
* *Limiting Bias*: To limit bias, I peer-reviewed my tests and sought feedback from colleagues. This collaborative approach helped identify blind spots and improve test coverage. As a developer, being responsible for testing my own code could introduce bias, potentially overlooking flaws that an independent tester might catch.
* *Commitment to Quality*: Being disciplined in commitment to quality is essential to avoid technical debt and ensure maintainability. Cutting corners can lead to hidden defects and increased maintenance costs. To avoid technical debt, I plan to adhere to best practices, regularly refactor code, and invest in comprehensive testing. For example, I will incorporate continuous integration (CI) pipelines to automate testing and ensure consistent code quality.

Conclusion:

The summary and reflections report highlight the importance of a rigorous and disciplined approach to software testing. By employing unit and boundary testing techniques, ensuring technical soundness and efficiency, and maintaining a cautious and unbiased mindset, I ensured high-quality outcomes for the project. This commitment to quality is essential for successful software engineering and will guide my practices to avoid technical debt and deliver reliable software solutions.

**References:**

* Bansal, A. (2024, May 11). *Best practices for unit testing in Java*. Baeldung. <https://www.baeldung.com/java-unit-testing-best-practices>.
* Grant, J. (n.d.). *11 tips for unit testing in Java*. Code Intelligence. <https://www.code-intelligence.com/blog/11-tips-unit-testing-java>.
* University of Minnesota Duluth. (n.d.). *Software Development Process*. d.umn.edu. <https://www.d.umn.edu/~tcolburn/cs4531/slides/software_engineering/process/process.xhtml#:~:text=A%20disciplined%20software%20process%20serves,long%20a%20project%20will%20take>.