**Summary and Reflections Report**

Tyler Skinner

SNHU

CS-320

Professor Norman

02/23/2025

**Introduction**

As part of the development procedure software testing verifies that applications fulfill functionality standards and performance requirements and maintain high-quality specifications. Unit testing became the main approach to verify the validity of the essential features which included contact management, task management, and appointment scheduling. JUnit tests allowed complete accuracy checking for software requirements compliance while detecting potential issues during the initial development phase of each feature.

The report explains the procedure of my unit testing while assessing the JUnit quality and reviewing applied testing methods. Additionally, the report highlights essential testing mentality key to effective software testing by stressing caution and bias reduction and demanding quality excellence. This reflection demonstrates why formal testing practices together with best practices are necessary to avoid technical debt while building software reliability.

**Unit Testing Approach**

My implementation used JUnit as the formal platform for testing the three mobile app components including contact service and task service and appointment service. Each unit test within my plan validated expected system behavior against the predefined specifications and tested extreme cases simultaneously.

1. I performed a unit test of contact service functions that included testing contact addition as well as contact modification and contact removal through simulated user interactions. The test cases specifically checked for correct assignment of unique identifiers while also validating proper invalid input handling together with data integrity preservation.
2. The testing process included units which verified proper execution of task creation and modification and task deletion through ensuring correct handling of character restrictions and null value processing.
3. Appointment Service: Testing included scheduling, rescheduling, and canceling appointments with valid date-time formats and non-conflicting appointments.

**Alignment with Software Requirements**

My approach was tightly linked to the software requirements, as evidenced by the large test coverage percentage. For instance, I ensured that each business requirement had at least one test case to represent it (Pressman, 2020). Test cases included boundary value analysis, equivalence partitioning, and negative testing for robust validation.

**Quality of JUnit Tests**

The JUnit test quality was evaluated overall by coverage analysis. I was able to achieve an average of 90% test coverage, adequately exercising major components. High coverage ensured main logic paths were tested, and possible bugs were minimized (Junit, nd). The use of assertions (assertEquals, assertNotNull, assertThrows) checked expected results effectively.

**Experience Writing JUnit Tests**

**Technical Soundness**

To be technically correct, I adhered to best practices while structuring test cases. For example, in the ContactServiceTest, I did the following:

@Test

void testAddContact() {

ContactService service = new ContactService();

Contact contact = new Contact("1", "John", "Doe", "1234567890");

service.addContact(contact);

assertEquals("John", service.getContact("1").getFirstName());

}

This test confirmed that the addContact method was successfully saving contact information and returning anticipated values.

**Code Efficiency**

To be efficient, I utilized setup methods and parameterized tests to avoid duplicating code. For instance:

@BeforeEach

void setUp() {

service = new ContactService();

contact = new Contact("1", "John", "Doe", "1234567890");

service.addContact(contact);

}

This reduced code duplication and ensured that each test ran in a sandboxed environment with predefined data.

**Reflection**

**Testing Techniques**

**Techniques Used**

* Black-box Testing: Focuses on testing outputs with respect to given inputs without examining internal code structure.
* White-box Testing: Used coverage analysis to test all logical paths, including exception handling.
* Boundary Testing: Tested proper handling of boundary values, i.e., minimum and maximum input lengths.

**Techniques Not Used**

* Integration Testing: While unit tests guaranteed individual components, they did not consider interactions between different services.
* Performance Testing: Load and stress testing were beyond the project scope but are crucial for scalability.
* Security Testing: Not specifically requested, but security tests such as input validation for SQL injections would provide strength.

Each approach has practical uses in developing software. While unit tests ensure correctness at the micro level, integration and performance tests provide confidence in system-wide behavior.

**Mindset**

**Caution and Complexity Considerations**

I followed a strict methodology to uncover edge cases and defects early on. I needed to understand the complexity of dependent services. For example, I anticipated data persistence problems, ensuring that the deletion of a contact also removed tasks and appointments related to it.

**Limiting Bias**

To prevent bias in reviewing my own code, I employed peer review and automated test verification. If I had been alone testing my own code, confirmation bias could have led me to overlook defects. As an example, a test I initially wrote assumed updateContact would always succeed, but external review raised the issue of testing failed updates when contact IDs are absent.

**Commitment to Quality**

Code quality discipline is extremely crucial in software engineering. Shortcuts in testing can lead to technical debt, which will cause maintenance to be more expensive in the future. To avoid this, I will:

1. 1.Utilize Test-Driven Development (TDD): Prewritten tests ensure well-defined objectives.
2. Use Automated Testing Pipelines: Continuous Integration/Continuous Deployment (CI/CD) ensures regular verification.
3. Refactor Code Regularly: Reducing complexity and improving readability prevents long-term issues.

By emphasizing robust testing, I will assist in creating stable and maintainable software systems.

**Conclusion**

This assignment cemented my understanding of unit test methods with an emphasis on black-box, white-box, and boundary value testing. Developing JUnit tests not only strengthened the application's reliability but also my understanding of code quality and software development discipline (Beck, 2003). Through thorough testing, being unbiased, and adhering to best practices, I delivered solid, efficient, and quality code—a philosophy that I will uphold throughout my software engineering future.

References

Beck, K. (2003). *Test-Driven Development: By Example.* Addison-Wesley.

Junit.org. (n.d.). *JUnit 5 User Guide.* Retrieved from https://junit.org/junit5/docs/current/user-guide/

Pressman, R. S., & Maxim, B. R. (2020). *Software Engineering: A Practitioner's Approach* (9th ed.). McGraw-Hill.