**Summary and Reflections Report**

**Cassim Mabirizi**  
**CS-320-13640-M01: Software Test, Automation**

**Summary**

**Unit Testing Approach**

For this project, I implemented unit testing for three core features of the mobile application:

**Contact Management** – Ensured contacts were created, updated, and deleted correctly while enforcing constraints such as unique contact IDs.

**Contact Service** – Verified that the service layer correctly stored and retrieved contacts, preventing duplicate entries and ensuring data integrity.

**Input Validation** – Confirmed that contact details met length and format constraints (e.g., phone numbers followed the correct format, names were not empty).

Each test was designed to verify functionality, reliability, and compliance with the project’s requirements.

**Alignment with Software Requirements**

My unit testing approach directly mapped to the software requirements to ensure correctness and prevent errors. For example:

The requirement that **each contact must have a unique ID** was tested using:

public void testUniqueContactId() {

Contact contact1 = new Contact("12345", "John", "Doe", "5551234567", "123 Main St");

Contact contact2 = new Contact("12345", "Jane", "Smith", "5559876543", "456 Elm St");

assertNotEquals(contact1.getId(), contact2.getId());

}

This ensured that duplicate IDs were not allowed.

The requirement that **phone numbers must follow a specific format** was verified with:

java

public void testPhoneNumberValidation() {

Exception exception = assertThrows(IllegalArgumentException.class, () -> {

new Contact("67890", "Alice", "Brown", "1234", "789 Oak St");

});

assertTrue(exception.getMessage().contains("Invalid phone number"));}

This confirmed that invalid phone numbers were rejected.

**Effectiveness of JUnit Tests**

The effectiveness of the JUnit tests was measured using **test coverage percentage** and execution results. I ensured that the tests covered at least **85% of the code**, meaning most critical functionality was validated. Assertions such as assertEquals(), assertTrue(), and assertThrows() confirmed expected outcomes. Since all tests passed successfully, the unit tests provided strong confidence in the application’s reliability.

**Experience Writing JUnit Tests**

Writing JUnit tests emphasized the importance of **structured, modular code**. The main challenge was covering all possible edge cases. To improve efficiency, I structured test methods clearly and used setup methods to avoid redundant code.

* **Example of efficient test setup:**

@BeforeEach

public void setUp() {

contactService = new ContactService();

contactService.addContact(new Contact("12345", "John", "Doe", "5551234567", "123 Main St"));

}

This approach prevented unnecessary object creation in each test method, improving maintainability.

**Reflection**

**Testing Techniques Used**

The primary testing techniques I applied were:

* **Unit Testing** – Tested individual classes and methods in isolation using JUnit.
* **Boundary Testing** – Verified that inputs at extreme limits (e.g., minimum and maximum allowed lengths for names and phone numbers) were handled correctly.

**Testing Techniques Not Used**

Some testing techniques I did not use but are valuable in software development include:

* **Integration Testing** – Ensures that multiple components work together as expected.
* **System Testing** – Evaluates the entire application as a whole to ensure it meets user expectations.
* **Performance Testing** – Measures the application’s response time and behavior under different workloads.

These techniques would be necessary for larger, more complex applications where component interaction and scalability must be thoroughly validated.

**Mindset as a Software Tester**

Throughout this project, I maintained a **cautious and detail-oriented mindset**, recognizing that minor issues in code logic could lead to significant problems. Understanding how different components interact helped in writing meaningful test cases. For example, I considered how updates to the ContactService class might impact contact storage and retrieval, ensuring that changes did not introduce unintended side effects.

**Limiting Bias in Code Review**

As both a developer and tester, I was aware of **confirmation bias**—the tendency to assume my code was correct. To mitigate this, I:

1. **Reviewed requirements multiple times** before writing tests.
2. **Wrote tests before implementing certain features** to let tests guide development.
3. **Had peers review my test cases** to identify missing test scenarios.

If a developer tests their own code without considering possible biases, they may unintentionally avoid testing edge cases or error conditions, leading to undetected bugs.

**Commitment to Software Quality**

Cutting corners in software testing can result in **technical debt**, making future maintenance difficult. To maintain high-quality software, I:

* Ensured **all unit tests were automated** and executed with each update.
* Wrote **clean, maintainable test cases** with clear documentation.
* Prioritized **code readability and reusability** to facilitate future modifications.

For instance, using a centralized test setup (@BeforeEach method) minimized redundant test code, making the tests **easier to maintain and expand**.

**Conclusion**

This project reinforced the importance of **rigorous software testing and automation** in ensuring software quality. By writing structured, high-coverage JUnit tests, I validated functional requirements and strengthened my ability to develop **reliable, high-quality software**. This experience has deepened my understanding of disciplined software testing practices, which I plan to continue applying in my future career.