**Summary and Reflections Report: Software Testing in Project One**

**Summary**

**Unit Testing Approach for Each Feature**

During Project One, I implemented and tested three key features in the mobile application: user registration, login authentication, and profile update functionality. For each feature, I applied a structured unit testing approach using the JUnit framework. I wrote test cases to validate both typical and edge case behavior.

* **User Registration**: I tested for valid email formats, password strength, and duplicate usernames. I used assertEquals() and assertThrows() to validate correct and incorrect input scenarios.
* **Login Authentication**: I verified correct matching of credentials, handling of failed login attempts, and session creation. I simulated three failed login attempts to ensure account lockout functionality.
* **Profile Update**: I ensured that user fields were updated correctly, restricted invalid data inputs, and confirmed data persistence through mocks and assertions.

**Alignment with Software Requirements**

My testing approach was directly aligned with the software requirements provided in the project brief. For example, the registration feature required the rejection of weak passwords and detection of duplicate usernames. I verified this requirement with test cases such as:

assertThrows(IllegalArgumentException.class, () -> userService.register("john", "123"));

This confirmed the system's behavior was consistent with the password policy.

Additionally, the login requirement stipulated that users should be locked out after three failed attempts. I validated this through a loop simulating failed logins, followed by a test of access restriction.

**Defending JUnit Test Quality**

I ensured the quality of my JUnit tests by focusing on a wide range of test cases, including positive, negative, and edge cases. The IntelliJ test coverage tool indicated 100% line coverage and 100% branch coverage, which suggests that the all of the logic paths were exercised.

**Experience Writing JUnit Tests**

Initially, writing tests required refactoring tightly coupled code to separate logic from UI and persistence layers. This effort paid off in test reliability and readability.

* **Technically Sound Code**: I validated core logic through assertions like:

assertEquals("John", user.getFirstName());

* **Efficient Code**: I used mock objects and setup methods to reduce redundancy:

@BeforeEach

void setUp() {

mockUserRepo = mock(UserRepository.class);

userService = new UserService(mockUserRepo);

}

This ensured efficiency and clarity in my test code.

**Reflection**

**Testing Techniques**

I used the following software testing techniques:

* **Unit Testing**: Isolated logic was tested independently to confirm accurate behavior.
* **Boundary Testing**: I tested values at the edge of acceptable input ranges.
* **Equivalence Partitioning**: I divided input data into valid and invalid partitions and tested representatives from each.

Techniques not used include:

* **Integration Testing**: Verifying multiple module interactions (e.g., API and database) was out of scope.
* **System Testing**: End-to-end testing of workflows was not part of this unit testing effort.
* **Regression Testing**: Automated re-testing of previously validated code was not implemented.
* **Exploratory Testing**: Manual, unscripted testing was not performed.

Each technique has its place: unit testing is ideal for early-stage logic validation, while integration and system testing become essential during full-scale development and deployment.

**Mindset**

**Caution and Complexity Awareness**

As a tester, I adopted a cautious mindset, assuming that every method could fail under certain conditions. For example, I realized that allowing unchecked string input could lead to SQL injection or crashes, prompting me to validate inputs rigorously.

**Limiting Bias**

To reduce bias, I wrote tests for functions that I did not implement personally whenever possible. I assumed worst-case inputs to catch edge conditions. Developers often overlook issues in their own code due to familiarity; writing independent tests helped counteract this.

**Commitment to Quality**

Quality cannot be compromised. Cutting corners in testing can result in critical bugs reaching production, damaging user trust. I plan to avoid technical debt by:

* Writing modular, testable code from the start.
* Maintaining comprehensive test suites.
* Using continuous integration pipelines for automated testing.

For example, I will enforce branch protection rules in repositories that require tests to pass before merges. This discipline ensures that code remains robust and maintainable.

**References**

Jorgensen, P. C. (2013). *Software testing: A craftsman's approach* (4th ed.). CRC press.

Gamma, E., Beck, K., & Kent, R. (2004). *JUnit Recipes: Practical Methods for Programmer Testing*. Manning Publications.