**Tatum Scott**

**2/23/2025**

**CS-320**

**Southern New Hampshire University**

**Module 7 Project II**

**Describe your unit testing approach for each of the three features.**

In this project, I focused on unit testing three core features of the mobile application: login functionality, data validation, and user interface behavior.

* **Login Functionality:** I tested the login functionality by simulating various inputs such as valid credentials, invalid credentials, and blank input fields. These tests ensured that the login system properly handles authentication errors and prompts users for valid input.
* **Data Validation:** The data validation feature tested the proper format of user inputs, such as email addresses and phone numbers, using regular expressions. I ensured that only correctly formatted data would be accepted, while invalid inputs triggered validation errors.
* **User Interface Behavior:** For the UI, I created tests to check if buttons, text fields, and other UI elements functioned correctly. These tests simulated user interactions to ensure that the app responded as expected when users clicked buttons or filled out forms.

**To what extent was your approach aligned to the software requirements? Support your claims with specific evidence.**

My unit testing approach was closely aligned with the software requirements. For example, the login feature was tested to meet the security requirement of rejecting invalid credentials. The data validation feature was tested to ensure that only correctly formatted user inputs were accepted, addressing the requirement for data integrity. The UI tests verified that the app’s interface functioned as specified, ensuring a user-friendly experience.

**Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were effective based on the coverage percentage?**

The quality of my JUnit tests was verified through the use of code coverage tools like JaCoCo.My tests aimed for at least 85% coverage, ensuring that a significant portion of the application was tested. The tests covered edge cases and normal scenarios for all features, providing a reliable indication that the system would perform correctly in different conditions.

**Describe your experience writing the JUnit tests.**

Writing the JUnit tests was an iterative process. I began by testing basic functionality and progressively added more complex cases to account for edge scenarios. This approach helped ensure thorough testing, especially for boundary conditions like very long inputs in the data validation feature.

**How did you ensure that your code was technically sound? Cite specific lines of code from your tests to illustrate.**

To ensure technical soundness, I wrote focused, isolated tests that addressed specific behaviors. For example, in the login test for valid credentials, I wrote the following code to verify the expected behavior:

java

Copy

@Test

public void testLogin\_withValidCredentials\_returnsSuccess() {

User user = new User("testuser", "password123");

boolean result = loginService.login(user);

assertTrue(result);

}

This code verifies that the login system correctly authenticates valid users, ensuring the login functionality is technically sound.

**How did you ensure that your code was efficient? Cite specific lines of code from your tests to illustrate.**

I ensured efficiency by mocking dependencies to avoid unnecessary resource usage. For example, when testing the login feature, I mocked the user database:

java

Copy

UserDatabase mockDb = mock(UserDatabase.class);

when(mockDb.getUser("testuser")).thenReturn(new User("testuser", "password123"));

Mocking the database allowed the test to run quickly without involving external systems, ensuring both speed and resource efficiency.

**Reflection**

**Testing Techniques**

What were the software testing techniques that you employed in this project? Describe their characteristics using specific details.

I employed two primary software testing techniques in this project: **unit testing** and **boundary testing**.

* **Unit Testing:** This technique tests individual methods or functions in isolation. I used unit tests to validate the core features, ensuring that each unit of functionality worked correctly in isolation, such as validating input formats or checking authentication logic.
* **Boundary Testing:** This testing technique focuses on testing the extremes of input values. I used boundary testing to verify that inputs outside expected ranges, such as very long or malformed data, were properly handled.

**What are the other software testing techniques that you did not use for this project? Describe their characteristics using specific details.**

Other testing techniques I did not use for this project include **integration testing** and **system testing**.

* **Integration Testing:** This type of testing focuses on checking how different modules or components work together. For example, integration testing could have been useful to verify that the login service and the database communicate properly.
* **System Testing:** System testing tests the entire application as a whole, ensuring that all components interact correctly in a real-world environment. I did not employ system testing, as my focus was on individual feature validation.

**For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.**

* **Unit Testing:** Ideal for smaller, isolated components of the system. This technique is helpful in large projects with complex systems where developers need to ensure that individual pieces work correctly before integration.
* **Boundary Testing:** Practical for projects where inputs can vary significantly in size or type, such as applications that accept user-generated data. This ensures that the system doesn’t fail in extreme situations.
* **Integration Testing:** Useful for projects where different modules or systems interact. It’s necessary to verify that integrated systems work as expected and to detect issues that arise when combining components.
* **System Testing:** Crucial for projects where the end-to-end functionality must be validated before deployment. It ensures that all components work together as a whole and that the application meets overall user requirements.

**Mindset**

**Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ caution? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.**

Throughout the project, I adopted a mindset of caution and thoroughness. I appreciated the complexity of the code and the interrelationships between features, especially when testing login and data validation. For instance, when testing the login functionality, I was cautious not to overlook edge cases, such as special characters or empty input fields, because these cases could lead to unexpected behaviors and security vulnerabilities.

**Assess the ways you tried to limit bias in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims.**

To limit bias, I reviewed the code from the perspective of an external tester, focusing on different input scenarios and potential weaknesses. When testing my own code, I made a conscious effort to challenge my assumptions and verify that all edge cases were covered. If I were testing my own code, bias could arise from familiarity with the code, possibly leading to overlooking flaws or assuming the code worked perfectly. For instance, I might assume the login functionality works as expected without testing special cases like extremely long usernames or incorrect password formats.

**Finally, evaluate the importance of being disciplined in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.**

Being disciplined in writing and testing code is crucial to maintaining the quality and reliability of software. Cutting corners, such as skipping tests or failing to write clear code, can lead to bugs, poor performance, and security vulnerabilities down the road. For instance, skipping unit tests could result in critical features failing in production, negatively impacting users. I plan to avoid technical debt by consistently writing comprehensive tests, adhering to coding standards, and regularly refactoring code to keep it maintainable. By doing so, I can ensure the application remains robust and flexible for future updates.