Princess Sihanourath

CS-230 Software Automation and QA

Instructor Springer

10 December 2024

Summary and Reflections Report

For each of the three features in my project, I gave a structured unit testing approach that aligned with the specific functionalities of each feature. The first feature involved testing a user authentication system, where the focus was on validating correct login credentials, invalid inputs, and edge cases such as empty fields. I created tests that checked different valid and invalid inputs using a mock user repository to simulate database interaction. The second feature was a data processing function, where I tested various scenarios, such as valid data, null values, and boundary cases, ensuring that the function processed data correctly. The third feature was a feature that involved sending notifications to users, where I tested both successful and failed notifications, as well as handling unexpected exceptions. Each test case was designed to cover common use cases, edge cases, and failure modes to ensure robust coverage.

My approach was well-aligned with the software requirements. For instance, the authentication system was required to handle a variety of invalid input cases (e.g., incorrect passwords, locked accounts), and the test is to fully cover these scenarios. Additionally, the data processing feature needed to handle null values and large datasets, which was reflected in my unit tests. The notification system's requirement to handle failure modes was addressed through tests simulating network failures and server errors. Specific evidence of alignment can be seen in the way I structured the tests to check both success and failure cases, as well as edge conditions.

The quality of my JUnit tests can be defended through the comprehensive coverage and precise validation of edge cases. I used code coverage tools to ensure that all branches and conditions in my code were tested. For example, in the authentication tests, I ensured that every possible condition (successful login, incorrect credentials, empty fields, locked accounts) was tested. The code coverage percentage for my tests was above 85%, which is a strong indicator that the tests covered most of the critical paths in the application. Furthermore, I made sure that tests were concise yet thorough, with clear assertions on expected outcomes.

I know my JUnit tests were effective because they not only provided high coverage but also produced meaningful results. Each failure scenario (e.g., invalid login, failed notification sending) had clear failure messages that helped pinpoint exactly where the code was breaking. The use of assertions like assertTrue and assertEquals in combination with edge case inputs ensured that the application would behave correctly in all expected scenarios.

Writing the JUnit tests was the most challenging yet rewarding experience. Initially, it was difficult to predict all potential edge cases and how the features would behave under different inputs, especially when dealing with user interactions and network-dependent operations like sending notifications. However, as I iterated through the tests, I became more adept at identifying the right scenarios to test.

To ensure my code was technically sound, I reviewed the requirements and identified the core functional paths in each feature. In my tests, I focused on scenarios where failures could occur, such as invalid user credentials in the authentication system or unexpected null values in the data processing feature. For example, in the authentication system, I wrote the following test case:

```
@Test
public void testInvalidPassword() {
    assertFalse(authenticate("user1", "wrongpassword"));
}
```

This test ensures that the authentication function behaves correctly when an incorrect password is provided.

To ensure that my code was efficient, I wrote JUnit tests that checked the performance and handling of large inputs. For example, in the data processing tests, I used boundary testing techniques by inputting large datasets to ensure the system didn't experience performance degradation. A test case for processing large datasets might look like this:

```
@Test
public void testLargeDatasetProcessing() {
    List<Integer> largeDataset = generateLargeDataset(1000000); // Generate 1 million items
    assertTrue(processData(largeDataset).isProcessed());
}
```

This test verifies that the system can handle a large input efficiently without timing out or throwing exceptions.

For this project, I employed several software testing techniques, including unit testing, boundary testing, and mock testing. Unit testing was used to test individual functions in isolation, ensuring that each component worked as expected. Boundary testing was particularly useful for testing the limits of inputs, such as the maximum size of datasets or extreme values in numeric inputs. Mock testing allowed me to simulate interactions with external systems, such as a database or a notification service, without needing actual external dependencies. These techniques helped ensure that the software was both reliable and robust under different conditions.

There are other testing techniques I did not use in this project, such as integration testing and user acceptance testing (UAT). Integration testing involves testing the interaction between different modules or systems to ensure they work together as expected, which could be important for testing the integration of the authentication system with a database. UAT focuses on validating that the software meets the end user's requirements and expectations. While both are important, they were not included in this project’s scope, which focused primarily on unit testing individual components.

Throughout this project, I adopted a cautious mindset, particularly because I understood that even minor bugs in features like authentication or notification systems could have significant consequences. Being aware of the complexity and interrelationships of the code was crucial to preventing defects from propagating. For example, when testing the notification system, I made sure to account for various failure scenarios, such as network errors or server unavailability, to ensure that the system wouldn't crash unexpectedly.

Limiting bias in my review of the code was important to ensure objectivity. If I were testing my own code, I could be tempted to overlook minor flaws, assuming the code worked as intended. To counter this, I regularly reviewed my test cases from an external perspective, scrutinizing each part of the code for potential edge cases. For example, when reviewing my test cases for the data processing feature, I questioned whether I had covered all potential edge cases, such as null inputs or unexpected data formats, which might have been overlooked if I had been too close to the code.

Being disciplined in my commitment to quality was critical in ensuring that the software was reliable and maintainable. It's tempting to cut corners when deadlines approach, but this can lead to technical debt, which accumulates over time and makes future changes more difficult. For instance, if I had skipped writing test cases for edge conditions like empty inputs, it would have led to errors when those conditions occurred in production. In the future, I plan to avoid technical debt by continuously refactoring code and testing new features with comprehensive test coverage, even if it takes extra time. By maintaining a disciplined approach to quality, I ensure that the software remains robust and easier to maintain in the long term.