Jeffrey Karl

CS-320

10 December 2023

Project Summary and Reflections Report

My unit testing approach was meticulously aligned with the software requirements detailed in the requirements document. For each of the three features I developed, I referred to the requirements document to determine the necessary tests to validate the code’s functionality. For instance, I created a test case to verify the correct operation of the address field. Subsequently, I designed another test case with an invalid address exceeding ten characters to ensure the code appropriately triggered an error. The effectiveness of my JUnit tests was evident, with my code’s test coverage exceeding eighty percent and the project’s total coverage reaching approximately ninety percent. This high coverage indicates that the tests I wrote were executed on ninety percent of the project’s code lines.

To ascertain the technical robustness and efficiency of my code, I devised tests to identify scenarios where the code might fail, thereby ensuring its proper functioning. To enhance test efficiency and avoid code repetition, I implemented a ‘before’ function to execute prior to each test, significantly reducing redundancy in the test files.

Throughout each development phase, I relied heavily on unit testing as my primary strategy. This technique assesses individual components of a software program and is frequently used by developers to ensure that separate software units operate as expected. Unit tests, typically automated, are designed to scrutinize specific code segments, such as a unique function or method. Conducted at the most basic level of the software development cycle, unit testing involves the separate examination of individual code units. To verify that the code functioned in accordance with the requirements document, I isolated and scrutinized various code components.

While I didn’t employ them, there are other testing methodologies available, such as Blackbox testing and non-functional testing. Blackbox testing entails the tester validating the software’s functionality across all scenarios, without insight into the software’s internal structure, backend architecture, components, or business/technical requirements. The objective of black-box testing is to uncover performance issues, missing functions, initialization errors, and potential complications that could occur when interfacing with an external database. On the other hand, non-functional testing assesses the software’s non-functional attributes, such as performance, usability, security, and compatibility. The tester checks if the software meets the non-functional requirements and offers a user-friendly experience. The goal of non-functional testing is to ensure the software aligns with user expectations and delivers a seamless user experience.

Blackbox, unit, and non-functional testing are distinct methodologies, each tailored to specific project requirements. Blackbox testing is advantageous when the tester lacks knowledge of the software’s internals and is primarily used for verifying functionality and user requirements in web or mobile applications. In contrast, unit testing is a strategy to improve code quality by identifying errors and bugs early in the development cycle, thereby reducing the cost of bug fixes at a later stage. It expedites the development process and can be applied to any project, including API and desktop application builds, to test for bugs. Lastly, non-functional testing is employed to ensure the software meets non-functional aspects like performance, usability, security, and compatibility, and is often used for performance testing of web or mobile applications.

In software testing, it’s crucial to adopt a problem-oriented mindset over a solution-oriented one. When testing the code, my emphasis was on uncovering defects and breaking the features. This approach aids in identifying any potential issues that the developers might have overlooked during code writing.

In software testing, it's vital to comprehend the complexity and interconnectedness of the code. Code complexity, which refers to the challenge in understanding and maintaining a piece of software code, is influenced by various factors. These include the number of lines of code, conditional statements, loops, nested structures, methods/functions, the volume of data processed, and the level of abstraction used in the code. By grasping the complexity and interrelationships within the code, it becomes easier to pinpoint potential issues stemming from the interactions between different code parts.

To minimize bias, I crafted my tests using only the provided functions in the code, treating the test as an interface to the features. This approach ensures that I'm not writing tests in a way that intentionally makes them pass. I also adopted a mindset that these tests are designed to enhance the code I wrote. Therefore, I was stringent in attempting to break it to improve the code, as a developer's ego can significantly contribute to bias and hinder their testing.

Automated tests are inherently biased by their author, typically a developer. In most teams, the developer who constructs a feature is also the one who writes the automated tests for that feature. It's all too easy for this developer, irrespective of their level or experience, to write tests that are overly tied to the specific code they wrote. Ideally, automated tests should assess some part of a software system's behavior from the perspective of someone without knowledge of the system's inner workings. In development, this is often referred to as an interface. Our applications have user interfaces, and our software has logical interfaces.

As a software engineering professional, maintaining a disciplined commitment to quality is paramount. It's crucial to ensure that the software you develop is of high quality and meets the end-users' requirements. Cutting corners when writing or testing code can lead to technical debt, which can be detrimental to the project's success. Technical debt, the cost of maintaining and fixing poorly written code, can arise when developers take shortcuts to meet deadlines or when they disregard best practices. Technical debt can result in increased costs, reduced productivity, and diminished quality. Therefore, it's critical to avoid technical debt as much as possible. To prevent technical debt, I plan to adhere to the best practices of software development. For instance, I will write clean and maintainable code, follow coding standards, and use automated testing to verify that the code is functioning correctly. I will also regularly refactor the code to keep it current and eliminate any accumulated technical debt.

Work Cited

GeeksforGeeks. (n.d.). Types of Software Testing. Retrieved December 10, 2023, from [https://www.geeksforgeeks.org/types-software-testing/?ref=lbp](https://www.geeksforgeeks.org/types-software-testing/)

Bose, S. (2023, May 25). Software Testing Techniques: Explained with Examples. BrowserStack. Retrieved December 10, 2023, from <https://www.browserstack.com/guide/software-testing-techniques>

Moradov, O. (2023, July 26). Unit Testing: Definition, Examples, and Critical Best Practices. Bright Security. Retrieved December 10, 2023, from <https://brightsec.com/blog/unit-testing/>

Kumar, N. (2021, November 26). What is Code Complexity and How to Measure it? Hatica. Retrieved December 10, 2023, from [What is Code Complexity and How to Measure it? - Hatica](https://www.hatica.io/blog/code-complexity/)

McConnell, J. (2018, March 8). Mozilla experiment aims to reduce bias in code reviews. The Mozilla Blog. Retrieved December 10, 2023, from [Mozilla experiment aims to reduce bias in code reviews](https://blog.mozilla.org/en/mozilla/gender-bias-code-reviews/)