Over the past several weeks we attempted various approaches to unit testing, written JUnit tests, and were able to attempt different testing techniques and trial various mindsets for writing code. This document will serve as a summary of the work completed as well as a reflection on what was learned and how the information covered will be pertinent in the future. To touch on that last point, this will no doubt be invaluable in the future as these series of tests can serve as a first line of defense against potential code breaking errors that can turn a thousand-line software system to a crumbling mess all because a user entered a password more than 10 characters.

Let’s start this out with a summary of the work completed over the past few weeks. Let’s start with a programming quality that cannot be easily tested. How well does our program meet the requirements of the customer? In the case of this project the requirements of the customer have been adhered to almost to the letter. This was made easy as we had the benefit of not having to deal with the potential changing requirements of the customer as different stages of the project are met. The quality of our JUnit tests has to be covered as part of this summary. The tests were of a very high quality as seen by the coverage percentage. The coverage percentage is our ratio of the number of lines of code in comparison to the lines of code being actively used by the program.

This project has really opened me up to JUnit testing which I had not really even heard of before undertaking this project. The main ways that I deduced the soundness of my code was by really taking my time with the first object of each task to really ensure that the first was to the highest caliber and then utilizing it as a guide to ensure that the rest of the objects were held to that same degree. Also, to ensure that each test was held up to par I was able to enter in false inputs in order to ensure that those sorts of inputs would inform the user that their input is invalid and prevent those types of inputs causing a massive system failure. In order to keep the code efficient, I found that it was important to keep lines as simple as possible, not clash with one another and keep each object as separate as possible.

This document is also to reflect on the work completed on the project. First let’s reflect on the various software testing techniques that were employed throughout this project. Most of the testing I used both from the beginning to the end could be surmised as static testing. Meaning, a majority of the testing was done without requiring the full code to be completed and testing more for errors on my part as the programmer than how the various pieces of the software are able to interact with one another.

Towards the end of the project, however, I amended my testing strategy. At the beginning of the project, I was going through double or sometimes even triple checking each line of code which can be a good rule of thumb but really adds on to the time that the programming takes to complete. Towards the end of the project, I had the realization that a majority of the programming was very similar, so I got into the habit of taking the time and doing my due diligence on the first object of the code and then using it as a reference for the rest of the objects to ensure that each object was as successfully composed as the first. I did not use nearly as much dynamic testing as I wish I had for the project and know that in the future I will not be able to count on static testing alone to be successful in programming.

What proved to be very important for this project was mindset. The mindset that I found most useful for completing this project in the best possible manner was a sort of rendition of devil’s advocate. I chose to approach this project with the mindset of what can go wrong. I found this was wildly helpful as it allowed me to come up with creative ways that user inputs could break my program. Caution came into play just as much as it does with any program. I used the most caution when it came to ensuring that my code covered the requirements deemed by the customer to quell invalid user inputs from running rampant through my system.

Limiting bias is an incredibly hard thing to do as a software engineer. The best way that I have found to eliminate bias is by having a separate programmer test and review my code which was not really a possibility in this circumstance, but I plan to implement in the future when I have that opportunity. In this scenario, however I will admit I did succumb to some bias when programming. The way that struck me more often than any other was assuming that since I did the coding it is good and doesn’t need checking. Which is a very dangerous type of bias for a programmer. As previously mentioned, I did not as intensely check each object after the first one. When working on software for paying clients I can’t count on the first object being enough reference for the remainder of the project as I did in this scenario.

Discipline is one of the most important traits a programmer can have, and I am disappointed to report that my discipline was not up to par for this project. My main testing style was the testing method of a disciplined programmer. I was lucky this time, my lack of discipline did not bite me too badly, but I doubt I will be able to say the same next time. Discipline, and not cutting corners are crucial to keeping out of technical debt. It is best to test early, test often, and test for as many scenarios as possible.

Citations

GeeksforGeeks. (2020, February 27). *Difference between static and dynamic testing*. GeeksforGeeks. Retrieved April 16, 2023, from https://www.geeksforgeeks.org/difference-between-static-and-dynamic-testing/