Terry Jung (tjung6)
Difficulties During Software Development
(Design, implementation, documentation, testing, maintenance)

Learning new coding languages while mixing it with old coding languages

I worked on a kiosk sign in system with a friend, it was written using JavaScript, PHP, HTML, SQL, and CSS. Before working on this project, it was already a working project in production written by someone else a few years ago. My skill in these coding languages were very little and I had no experience at all in PHP. We took on this project to try to optimize the sign in process and make the UI friendlier.

Reading the code became difficult because there's no specific standard how different coding languages should be intermingled. People have to try to use their best judgement and preferences to write clean, and concise code. There came situations where we were not sure whether to code a specific implementation in one language or do it another. Many times we came across editing the UI and debating whether we should use JavaScript or PHP to implement the functionality of the buttons, selections, and text boxes. Each language had their own approach and syntax to solve the problem, and even though performance wise it wasn't a big issue, it came down to clarity of the code. My partner preferred to write code in JavaScript but my preference was with the new PHP that I learned. In the end, we decided to stick mainly with PHP because there was very little JavaScript in the original code compared to the amount of PHP. We spent a lot of time reading documentation, since we believed this was the best way we could learn PHP will still make progress on the project. We could have taken some time to develop some small projects in PHP before messing with any code but we expected we could handle it. When I look back now, I would still argue there was no correct way to approach these problems.

Restructuring working code that had no definite structure

Since we weren’t the original authors of the code we wrote, it was really tempting for us to continue adding and changing the code with the current structure. We know we were given working code, so trying to change it seemed irrational. "Why fix something that isn't broken." However, we understood for any future changes or updates, it would make it better if we restructured everything. If we were ever to come back and fix any bugs or if someone else were to work with the code, we wanted it to be structured in a way that was more readable and clean than the current version.

This brought up an issue about correctness. It was difficult for us to define how the code should be structured especially if we were not the original authors. It was difficult for us to understand why he wrote code in a certain way, and sometimes even the author admitted he wasn’t sure about certain parts since he had written it long ago. Our best solution was to base our structure off how we believed how parts could be broken into modules. However, this was easier said than done. Many parts of the code felt unnecessary, but it wasn’t our call to deem the code useless if the kiosk system could potentially use it in the future. We had used the sign in system many times ourselves, so we tried to make decisions based on what a user would find friendly and simple to use. We wrote up all the current problems or issues that we thought about from past experience and tried to shape it to what we thought would be “cool” and “easy”.

This introduced a lot of bias into our code. My partner had a very specific way he liked structuring his code, and I had specific preferences that weren’t always the same as his. Neither of our methods

seemed incorrect, even sometimes the old code had nothing wrong, but sometimes felt like we had to change it slightly just to satisfy what we thought was “better”.

Creating false data to test our program

In order to see our sign in system work we had to create realistic data to work with in order to see the kiosk in action. However, sometimes whenever we found a bug, we weren't sure because it was the way we entered the data, the data itself, or our code. It wasn't always the case that our code was the issue, however many programmers will probably assume it has to do code, and our immediate thought was to assume that we had a bug in our program.

We had to retrieve our data from a database so the way we inserted data became very specific. We ended up having to create tests for our tests just to ensure that our "data" was properly recorded. We ended up only creating the basic necessity of data to test our code, which was bad practice on our part. While implementing code, it sometimes feels wasteful to try to create test cases for every possible path and outcome. We have previously used the sign in system before, so we assumed that certain situations could never occur and would never arise to avoid writing specific checks or test our code in specific ways in order to speed up our process and push our code in production. It made sense that if our program was to be used for a very specific purpose, it wasn’t practical to create and handle data that would never appear in the database. As a result, when we tried to demo our program for others to see before pushing it into production, we ran into many bugs, bugs we never expected to occur when we previously tested.

Looking back now, I would blame a lot of these last minute issues due to the lack of depth of our testing. It made sense to test our code in a way such that it would work when a normal student tried to sign in, but we lacked the many tests to make the program robust if someone tried to break the program. Seeing the code break in front of us while it was demoing made us realize how important it was to make sure that any student who was using the program would not end up in a broken HTML page, or having their personal classes exposed to the public. Even though these cases were ever documented or explicitly required, it made sense for us to be responsible to make sure our program handles misused situations. This brings up the issue about how programmers should be prepared to handle cases that were never explicitly listed in the specification, and make smart choices about how to handle them.