Final Project

Summary and Reflection

CS-320

SNHU

Caroline Wilson

June 2020

**Summary and Reflection**

With testing procedures in mind, I set out to ensure that the software requirements would be up to par for the client. I went through a series of steps to test features of the program as well as the sub-categories therein. Through this process I was able to recognize a number of errors that would affect the readability of the code as well as the functionality. With careful and concise testing, I was able to report these findings and pinpoint where there were issues in the original source code.

In order for each of the required features of the program to work successfully, it was important to test each of the building blocks that lead up to them. I started this process by individually testing each of the models to ensure accuracy and functionality. It was here that I started finding many of the problems. Some of these issues were just a question of readability. However, in some cases, because of the inconsistencies we are able to see the waterfall effect that would happen as a result. This was aided in my decision making early on to test classes and calls individually instead of creating testing suits for organizational purposes. I believe that, while easy access and organization are paramount to a successful test, having the code be recognized for potential syntax errors, spelling errors, and overall sloppiness of code lends itself more to an overall quality product in the long run. Very often it would be easier to compile a test suite in order to have better organization, but with the number of easily overlooked errors I found, I decided it best to compile my tests in a piece wise fashion.

The next level of importance is of course to present the client with the product that they are looking for. In order to do this, it is important to test each of their required features and be ready to present them with exactly what they asked for. I found using a ‘map’ and pseudocode allowed me to create a checklist of sorts so that I could check off the features functionality as I moved along in my testing process. Before anything could be done, it is important to set the groundwork. For example, in adding a doctor, not only was it important to test each of the classes that allowed a doctor to be “set and get”, but it was also by request that there be no duplicate doctor ID’s. There were lines of code in reading that original source code that allowed for this, but that feature, too, needed to be tested.

It was important to the client that there be the ability to add patients and records to the program. This, too, had a stipulation in that there could be no data entered under a patient without a treatment initiated. Each of these features required models that would allow for the construction of the records so that they could be called upon. These, too, needed to be tested to ensure that the information was able to be added to the lists as the entry went on. In addition to the models, the lists themselves needed to be tested to ensure that they could store data and retrieve data when asked.

This concept of lists brings us to the next topic of testing, the allergy lists. Within this level of testing, a test needed to be created that would ensure that not only a list was created of known allergies, but that this information could be pulled up either by patient or by allergen list as a whole. With these tests, we had to check each line of code to ensure that everything functioned properly when called, stored information as it was supposed to, and was accessible to the doctors. The list functions were paramount to this step in the code, and it was important to ensure that the ‘Allergey’ model was available to access.

In my time coding up until now, I have learned one mantra that keeps repeating itself. Test early, and test often. It may seem redundant to think that there is a need to test the code that is testing the code, but it was paramount to my success in these assignments. It is not always easy to visualize how exactly lines of code are going to pan out, even less so to understand without some sort of guidelines to understanding what variables, list, and classes need to be initiated in order for the tests to occur. With this in mind, I used a pseudocode that allowed for me to paint a roadmap to what I was trying to do with each line of code that I was writing. Starting from the very bottom in creating the variables, I was very careful to ensure that every call was answered. Each time I wrote a few lines of code, I would stop and run it as a jUnit test. This allowed for me to pinpoint issues before they got to be too big and too many, and it allowed me to find direction in the errors I was encountering.

Through the course of my testing I was able to pinpoint a number of errors. There were some small errors, that would account for code cleanliness and readability, but there were also cases in which these errors cascaded into something much more. There were a couple of instances, as well, that I was able recognize and notate errors in the execution of the code. With my initial testing, I was able to find eight errors that would need to be corrected in order to create a better code.

The most glaring was the incorrect spelling of ‘Allergy’. Unfortunately, this error caused need for the misspelling to be carried out not only throughout the entire original source code, but also through the testing phase as well. Secondly, there is another error in the naming of “MedicalRecordService”, there is an extra S in Record. Because of this, and the calls that had to be made in my testing, I had to use the same spelling to be carried out repeatedly. There were two more misspellings of “allergy” throughout the source code. One was in PatientHistory when “getalergies” is called, and the other in the PatientHistory Model when, though not a typo, allergy is inexplicably spelled correctly. Though the spelling is correct it doesn’t allow for the code to run as every other instance is incorrect.

In addition to the numerous typos, there were also instances in which readability was affected and the output would no look as it was supposed to. In two separate instances, in both the medication and doctor models, when the output is presented the spacing is incorrect within the quotation marks. The tests that I created are meant to test for the correct output, and thus the test fails in both of these instances.

In addition to syntax, grammatical and spelling errors, I was able to locate two errors in the functionality of the source code. In the Doctor Services, there are two instances in which the requirements of the client were not met. By using Boolean checker in addition to a while loops, it is possible to see, first, that the requirement that there be no duplicate doctor ID’s it not met. It allows for more than one doctor to be entered with the same id. In addition to this, the client asked for the ability to add a doctor to the list with blank space in the name. According to the tests run, this is not possible.

**Mindset**

In working on this project, I was afforded the opportunity to take a peek inside the mind of a software tester. It is an important step in understanding the different facets of the development teams. Testing, as I found, is much more intricate a practice than I had previously lent to it. With the discretion that must be used, best practices that must be called upon and the general skill set that is required to be effective, the testing team is not without their talents.

I found that it was important throughout my role as a tester to ensure two things. The first was that I was testing the pertinent lines of code, and secondly that I was paying attention to what I was doing along the way. One area in which I noticed that I needed to exercise caution was that in my creation of the test suites. If I wasn’t careful, I could see how the wrong lines of code could be edited. With the .src file being the same location as both the test and the original source code, while flipping between the two to ensure that every important aspect was covered. It could be a large setback if a section of code was accidentally overwritten.

It is also important to practice caution in the creation of tests in general. It is important that you are careful in your creation of classes and variables, especially in the initial methods. If the incorrect verbiage is used, incorrect results could be given and the tests could be nullified. In this case they would do more harm than good in the production process. As with any business time is money, and with ‘small’ mistakes in the initial framework, it can send the whole testing code on its tailspin. This could lead to subpar production, duplicate testing and errors, and more time needed over all to produce what the client is actually looking for.

Additionally, it is important to ensure that none of the code is rushed or overlooked. There are strict guidelines that must be adhered to. If there happens to be an error in an important aspect of the code, as in the case of the Doctor Services in this course example, and that test was missed and given the greenlight, then the product would not be what the client was looking for. This could cause longer production times as well as a tarnished reputation. In a competitive field, there is always someone waiting in the wings to take your place. If you don’t put your best work forward, its very possible that the client would take their business elsewhere on their next project. In addition, bad reviews and a lack of commendable recommendations could be the line between overall success and failure.

It is important to limit bias in testing because of the pure fact that every coder has their own style. Stylistic coding is something that an individual develops over years of practice and refinement. Just because a developer may have a different approach to a problem, doesn’t mean that it is incorrect. At the same time, it is important to remain impartial. If a developer were to test their own code, they would be looking at it through the same eyes by which it was created. It would be possible to overlook errors that another set of eyes would pickup on. A developer testing their own code would have the ability to pass over things that they believe to be best practices based on their style, but in reality are not the best for the code at hand. This is one of the reasons that there is a division between the teams. While one team may have the ability to do the other’s jobs, their specialties and the separation allow for further scrutiny and constructive criticism.

As I stated before, as in any business time is money. The less errors, the more dedication to excellence, the more profitable a team will be overall. Getting a production done right in a timelier manner allows for more projects to be tackled. In addition to this, having realistic expectations and best practices, a team can work to their fullest potential without having made promises that are unobtainable and unrealistic. It is important to set a wider scope of deadlines and be finished early that present a late rushed product that falls behind in expectations.

Cutting corners and taking the easy way out in coding is a quick way to lead to failure. While its true that very often there is more than one way to complete a task at hand, following best practices will allow for more flexibility, more modularity and concise replication of the clients wants and needs. It is always better to do more than expected with a quality product and to wow a client than it is to make promises that can’t be kept. To submit a program to a client that does not meet their expectations is a proverbial nail in the coffin. In any business, reputation is everything. It is better to be known as an overachiever with quality reusable products that to be known as someone who will do the bare minimum to present a less than stellar product.