Michael Gontarek

CS-320- Software Test Automation

Southern New Hampshire University

February 18, 2022

Project Two

My testing approach was pretty aligned to the software requirements. This is shown with the following example.

Text

Description automatically generatedText

Description automatically generated

I made sure that for every method I made that there was a corresponding test. This was to ensure that the method ran properly. I did this for all the methods, some were covered 100% and others were in the high 80s. There is some room for improvement, but I am happy with this for now at my skill level.

While the quality of my JUnit testing was not perfect it did have pretty good coverage. In my contact service I achieved 79.7% coverage and in task service I achieved 78.4% coverage. I looked ahead in the class and the final project you need 80% coverage for the top tier grade. This leads me to believe I did pretty good considering my lack of experience and that I’m learning JUnit still.

The following picture is a great example of how I used technically sound code and good coding practices to keep my code clean and understandable. It has comments letting others know what each section of code is meant to do.

Text

Description automatically generated

I made sure to limit the amount of extra code when possible. The following code only has six lines of code. I made sure to create the test, then create a new task service, add the service to the list, delete that then go try and retrieve the task service to show that it was gone. There may be a quicker way to achieve this, this was just the quickest way I could think of. Text

Description automatically generated

For this project I found myself using some black-box testing. In specific, I used equivalence Partitioning. This is when you take code that sets a limit to a variable and test the limit. For example, the appointmentId had to be less than ten characters long. To test this code all I did was create a new appointment with an appointmentId longer than 10 characters long and then said to throw an illegal argument if it was longer. So, it threw the illegal argument and the code worked properly.

A screenshot of a computer

Description automatically generated with medium confidence

I also used white-box testing with me reading and interpreting code. The first week was full of stress and self-doubt. After hours and hours of pouring over code, I was able to understand what we were doing. I then started to look at code from the first week to help me when creating new code. This allowed me to learn from my mistakes and see what the code was trying to achieve.

There are several types of testing I did not use. I did not use decision table testing. I did not make a table to list all the input conditions that could occur and what the results could be in a table format. I also did not use flow charts to help me figure out what I wanted to test. When I wrote the code, I put the conditions for each variable in my if statement. This allowed me to just go back and see the parameters I had set up and build tests from there. I used these parameters and built tests one at a time. From there I tested one at a time until I had all my testing done.

Partitioning testing is great for when you have a limit for a variable. From there you can simply break the rule you have and make sure that your test throws an error. This is great for things like the project we worked on when you want to limit the number of characters that a variable may have so they don’t get out of control. Reading and interpreting code is one of the more valuable testing methods that I have used throughout school without knowing it is a testing method. In daily practical use, you will look at code to see if it looks right and the logic makes sense. This can later be verified with testing; however, it is also helpful to do this to learn new techniques especially if you are looking over someone else’s code. Decision table testing would be valuable if you had several possible decisions to choose from within the code. This would let you see what to test and in what order you should test the code.

When working on my tests I had to make sure that I used caution. In the beginning, I would make several changes to my tests and have issues. This came in the form of me copying and pasting my code. The issue stemmed from me not fixing all of the code to correlate with the current test, this caused havoc on my testing and code. I then started running my coverage after I created one test.

The way I was able to eliminate bias is simple, I know I am nowhere near a great programmer at this stage in my journey. So, I made sure to test my code early and often. I didn’t care how great I felt a line of code was, if it didn’t run like I was hoping I was ready to scrap it all and start over.

Being disciplined with the quality of code that I produce is going to be a major key in my future. This is how I will get promotions and try to avoid imposter syndrome down the road. If I write good code and can stay committed to being my best every day, then I can be proud of myself and help my teams in the future. If I started to try and cut corners when writing this code, I ran into a ton of errors. I learned my lesson that I needed to test early and often. Also, when working as a team I want to ensure that I am a strong member of the team that can be relied upon.

In the future, I plan on avoiding technical debt by making sure that I test early and often. I need to ensure that I code a little at a time and keep testing. This will ensure that I do not have dozens of lines of code that don’t work. If I were to just code and code with no testing… I could have major errors that would be more costly to fix later with time and money. For example, If I would have written all of my code for any of my tests without checking coverage and pushed it through, it may have had major issues and it may have been too close to release for us to go back and fix the errors to the degree that were needed.