Dylan Kusick

Dr. Angel D. Cross

CS-320

August 12th, 2023

Aligning my unit testing approach was achieved by working on each small requirement once at a time. It was easier to do my unit testing to make sure that a task, contact, or appointment could be added, making sure that feature worked fully, and then moving on to the next, like the deletion of a task, contact, or appointment. My unit testing was similar for all three features, as they all used similar functions. For example, the addition of a task for the task service had logic that was similar to the addition of an appointment for the appointment service. The difference between the two is that they had different parameters and required logic.

I was able to reach a high percentage of coverage in the Service testing for the TaskService, ContactService, and AppointmentService, as they all reached over 80 percent. This means that nearly all of the code I used was tested and used. A lower percentage would indicate that the code being written was not actually testing correctly.

Ensuring that the code was technically sound really came from using clear and consistent code. I was able to use functions from different testing libraries that allowed me to check to make sure the code being entered into the program was checking boolean values. I was able to work with these boolean values and throw errors in the testing if the value was supposed to be false but was instead true, or vice versa.

I was able to make sure that my code was efficient by using the least amount of code possible. This was a bit more difficult when working with the function that evaluates time, as we had a requirement that should not allow an appointment to be placed in the past. I ran into some issues and added some code required for the program to compile, but I was not particularly familiar with it or how it was used. I am not great at handling exceptions so checking for past dates in the AppointmentService.java file might have had code that could have been condensed for more efficiency.

Testing techniques that I used are equivalence partitioning and boundary testing. The equivalence testing was something I was able to use so that I could test to make sure certain entries were not more than ten characters long. Boundary testing was used to ensure that null values were not entered. This is because a null value would be out of the valid range of values that a user should be able to enter.

I did not test the interconnectedness between the different files, this could have been the reason some of my files had a lower coverage than others. I just recently started to understand the importance of coverage percentage, and why higher coverage means the code is functioning more effectively and efficiently. Integration testing would be something that could have brought up my coverage percentage, but I did not realize this until it was too late into the project development.

All these techniques have practical uses and implications and can be used in a lot of real-world applications. For example, different testing techniques can be used to make sure a system is not overloaded with information as part of an attack. This is something I learned from a software security class previously. Input validation is the first line of defense to secure software, so testing to ensure it works is essential for the program's success.

Working as a software tester in this class made the class unlike anything else I had ever done before. There was a lot of caution in test annotations as they were all so new to me. In other classes, we used functions that would only allow certain input but never tested if it would work exactly the way we intended it to like we did with JUnit testing. It was important to understand how some of the programs were connected because you needed to understand that all this input was being tested in a similar way. A specific example was how there was an ID for all the files, the contact, task, and appointments. The complexity of the files was something that you needed to understand so that you knew what you would be validating. It is difficult to validate something if you do not have a clue as to what you are validating.

I needed to limit bias to ensure I was testing the code in the most effective way possible. If I were to be testing my own code, bias could cause me to overlook some of the code that needed to be tested. As we have seen in different modules throughout CS320, code errors can lead to massive amounts of loss, and sometimes it is the loss of life, not just the loss of assets or things of monetary value. If I had not checked to make sure a user was entering in information that was less than 10 characters long for their name, there could be a way to overload that field in the program, which could lead to a system breach.

Working back to what happens when bugs pop up in software post-release, it is detrimental to many involved. When working with systems that could be placed in automobiles, aircraft, or anything that can be driven, a bug that breaks the system could lead to loss of life and assets. When software does not work the way it is intended to, the user ends up in inconvenient situations and problems. I plan to avoid technical debt in the field by assessing the testing needs early and testing them often. For example, when writing the code for the contact service, it was essential to write out what the contact service should do, and then immediately follow up with the testing for the different fields of input that would be presented to the user. This is because not every user will know the input parameters, and things do not always work the way the developers intended them to work.