Anthony Minunni

10/12/21

Software Testing and Programming

Project Two

I had a fairly straight forward plan to ensuring my project adhered to each component’s software requirements. I would read each bullet point in order and code that portion of the assignment rather than jumping around the requirements aimlessly. At first I would code the general aspect of the bullet points, then I would edit my code to ensure further completion of the software requirements. An example of what I mean can be seen through the Contact class. When I created the initial Contact class, I chose what I saw was the relevant components and created my constructor with setters and getters. For the case of the ID variable, I saw the contact ID had to be a unique string with no more than 10 characters and could not be null, so I knew the Contact class needed a private ID variable and a getID method but not a set method; I would save the unique ID, not null, and character limitation requirements for the ContactService class, where I incorporated the remaining requirements of Class into the bullet points of ClassService.

I think overall, my Junit portion of my tests could have been better. I had not used Junit before to test my coding projects, so my knowledge of the topic was limited only to Boolean tests. I was able to use them for the Contact, Task, and Appointment classes, but for the service side classes of the three, I did not know how to use Junit to test those methods and resorted to manual testing. Overall, Junit technically accounted for fifty percent of my testing, but it was not as in-depth as I would like to have had it. If I were to use Junit testing in the future, I would like to further my knowledge for testing methods that require user input or have branching lines of code (switch statement, etc.).

I ensured my code was sound by using both Junit testing and manual testing to ensure my code was running correctly, as stated prior. An example of this is through my Appointment project, where I used Junit testing to ensure my Appointment class was holding and updating private variables correctly. It was especially important with Appointment because I was using the LocalDate Java utility, so I had to use Junit testing to make sure I was declaring and using LocalDate variables correctly. I ensured my code was efficient by using switch statements as opposed to solely using if-else statements. Switch statements are quicker for the computer to process and allows an opportunity for me to write multiple methods to break up my code. This was also seen in AppointmentService where a method would be called based on the user’s request, such as the user pressing “2” to add an appointment would have the code run addAppointment.

My testing done for this project falls under the dynamic testing techniques, which are used to test the dynamic behavior of the application under test, that is by execution of the code base (GeeksForGeeks, 2021). The main technique I used outside of the Junit testing was error guessing, where I acted as a user using the program and testing all possible inputs that could be entered at a given time to make sure the program reacted accordingly. An example I used across all classes was to only press enter at an input prompt without filling out actual information (as a side note, I also had all inputs be .nextLine instead of .next to prevent a user from hitting space and causing data to be incorrectly entered further into the program). The program should recognize the lack of data and inform the user the “input” was not accepted, and the program should point back to the menu without saving the invalid input. There was some static testing as well through my inspection of the code’s control flow, which is the structure of how program instructions get executed (GeeksForGeeks, 2021), through my use of loops and if-else statements.

There were other testing techniques that I did not use. Reviews as a whole is a static testing technique that I did not use because they involve a developer speaking to a relevant third party about how the code will be programmed; I did not converse with anybody on these projects outside of the bullet points. Each technique that I did and did not use have practical purposes. Error guessing is practical because one of the more emphasized aspects of testing is bug fixing, and specifically looking for obvious bugs helps developers create a better constructed and more professional project. Control flow is important because it helps find redundancy in code or areas where it can be optimized to run more efficiently. Reviews are important because it helps developers ensure requirements are considered and multiple methods of developing are discussed so the developer ends up planning and developing more efficiently.

For this project, the amount of caution I displayed for this project varied as I developed different blocks of code. For the main classes of Task, Contact, and Appointment, I did not have a great sense of caution when creating the constructor, setters, and getters because I was comfortable with those formats and making sure they worked correctly. The lines of code that were reading from Scanner, however, made me feel a greater sense of caution because of past projects I worked. I’ve had a few projects where the program had bugs because of how I implemented the Scanner and I was worried this project would have the same issues, hence why I use both .nextLine as well as a try-catch block. I did not think about bias as I tested the projects because the point of testing is to make sure the main aspects of the project work correctly to have a running program. It’s hard to have a bias because even if a developer tries to test their own code willingly testing around known bugs, the bugs will become apparent at some point in the future, and that will hurt the reputation of the developer, so it is best to not have bias. This also relates to a developer’s discipline and the importance of that. Developers should not cut corners when developing, not only because of the reputation loss once the issues become apparent, but for programs with more important uses, such as software within medical equipment or travel vehicles, cutting corners could have grave consequences in the form of failures that could cause injury or death.

**References**

*Software testing techniques*. GeeksforGeeks. (2021, March 1). Retrieved October 13, 2021, from https://www.geeksforgeeks.org/software-testing-techniques/.