Nicole Penner

CS-320-T1025: Project Two

The unit testing approach I took for all three features of Project Two was to use Junit tests to check for completion. When looking at the software requirements for each java file needed, they were all aligned with the requirements needed. The first component involved the contact and contact service java files. The contact java file was created to have variables for the contact ID, first name, last name, phone number, and address. The contact ID specifications were that it cannot be null or greater than 10 characters, the first name and last name fields also could not be null or greater than 10 characters, the phone number variable could not be void or not equal to 10 characters, and the address field cannot be null or greater than 30 characters. If any field did not meet these qualifications, it would throw an exception and say “Invalid” to whatever field was incorrect. The contact service java file needed to have functions to add a contact (which also checked if it already existed), remove a contact with the contact ID, or use the contact ID to find a contact and update the other fields. This was all written into the code, and the Junit tests that followed had an 86% completion percentage. Therefore, the software requirements were met, and the quality of the Junit tests were effective. The coverage percentage is what is used to show how effective a Junit test is. The next pair of files were for the task and task service requirement. Tasks needed to have a task ID that was not null or greater than 10 characters, a task name that was not null or greater than 20 characters, and a task description that was not null or greater than 50 characters. All of these requirements were written into the task logic, and if the conditions were not met an exception would be thrown for and invalid field. The task service java file needed to have the functionality to add a new task with a unique ID, update a task using the unique task ID, and finally to remove a task using the unique task ID. Functions were written to ensure all the functionalities existed under addTask, updateTask, and remove. The Junit test completion percentage was 96% which ensures that the Junit tests written were effective. Finally, the java files were written for appointment and appointment service. Appointment needed to have variables for appointment ID which could not be null or greater than 10 characters, appointment date which could not be null or before the current date, and appointment description which be neither null or greater than 50 characters. All of these qualifications were written into the appointment variable in an if logic, and if the parameters were not an exception will be thrown of an invalid field. In the appointment service file, there needed to but functionality to add a new appointment and delete an appointment using the appointment ID – both were included in this file. After spending sometime rewriting the Junit tests for appointment, I was able to get the completion percentage up to 86% which means the Junit tests written were effective because it was 86% or higher. Writing the Junit tests for each piece of the code I did everything possible to help ensure that the code was both technically sound and efficient. To ensure the code was technically sound I followed the examples of how to write a Junit test and based all tests on this, because this was my first experience writing Junit tests. I ensured that I followed the best practices that I am aware of. I ensured my code was technically sound by making sure there were no errors in the code, and that the code ran and the output produced was what I intended. This was the case for all the files I have previously mentioned. I also correctly used if-else statements in contact service, task service, and appointment service to help ensure my code was technically sound and ran on the correct parameters. I also made use of for with an if statement to remove contacts, appointments, and tasks in their service files. All of the examples and reasons I gave help to paint the picture of how my code is technically sound. I also attempted to make my code as efficient as possible. I think the best example of this is the way I wrote the tests for contact, contact service, task, task service, appointment, and appointment service. To ensure my code was efficient I separately tested each part of the code that would throw an exception to make sure the tests of code were thorough. I will not cite specific lines of code by I made sure to test errors that would come back as null, and that would break the length maximum requirements. The other way I made my code efficient was to strictly lay out the requirements the different components of task, contact, and appointment needed to follow. If they didn’t an error would be thrown and read as invalid. My code is also efficient because there is no unneeded “fluff” to my code, it is all necessary and has a specific purpose.

The software testing technique that I used for Project Two was automation testing – but more specifically, I made use of Junit testing. Junit tests are a type of automated testing that is used to test software without human intervention. Less human intervention helps to increase success rates and provides for easier retesting. Unit testing is a form of white box testing which means that it tests the internal structure of the code to help ensure that methods and classes function correctly, and perform the functionalities they are supposed to. Junit tests are written typically per class or function to help pinpoint errors so the developer can easily identify and fix the errors. A type of testing that was not used in this project was manual testing. This has the same goal as automated testing, but it involves manually executing all test cases. To do this type of testing software requirements would first need to be analyzed to then write a test plan to ensure all software functionalities are covered. Finally, the testing is carried out a human going through all test cases and ensuring the functionalities perform as they are expected to. All software requirements are still tested, unless by human error a test case isn’t written or carried out by the tester. Automated testing can be used to save time and money during the software development process, and can easily be repeated multiple times within a short time. Automated tests take less time to create, and can be run as many times as needed to ensure the software’s functionality. Automated tests can be used in any project, whether it has a small or large budget. Time does not need to be spent meticulously writing test cases and then running the test cases all separately. Automated tests also help to keep accuracy high, since there is less room for human error, and can be used to easily increase the scope and coverage of tests. Therefore, automated tests would be especially beneficial for long-term and more complex projects. Manual testing would be beneficial for simple short-term projects. Manual tests are also useful to observe what an end user will see, and be aware of what the user experience will be like.

To work on this project, I tried to adopt the mindset of a software developer and software tester. This means thinking of how to take the requirements and create code that follows them. This needs to be done while considering the technical side, and how that can be translated to code. I worked to understand priorities, analyze the data, and tried to be flexible when my initial plan for the code didn’t work as I thought. The mindset of a software tester should be able to think critically, practically, technically and somewhat creatively. So I tried to adopt that type of mindset and consider all facets of the code. I employed caution when using Junit tests because I have never used them before, and it took some work to be able to use them correctly and make sure they were thorough. It was important to appreciate the complexity of the code and how the relationships of the files, because I made use of the same variables for example in contact and contact service, appointment and appointment service, and finally task and task service. Since the information was stored in an array, and these files needed to work together all of that needed to be considered and handled appropriately. I tried to eliminate bias by keeping a rational mindset, and understanding how I initially write code may not always be the best way. I can imagine a bias getting in the way because it may lead to a developer/tester not following all code requirements and test guidelines. This likely would be in the form of personal bias, and maybe would hinder a developer making sure there is proper test coverage and all requirements being met. It is important to be committed to quality as a software engineer because if you aren’t you may have errors in your code that could cause a problem later. If the code is quality is good it would be easier to scale the code up as needed, also if the quality is poor their may be unforeseen errors later in running code as its intended. If code needs to be reworked this will cost more money later, which incurs more technical debt. I plan to avoid technical debt just by writing quality code, because if corners are cut it can cost more money down the line to rework code if it is written not well. The same can be said if the automated tests are not written adequately. It will cost more money because the team will have to spend more time and money to fix the code, therefore this is good reasoning (besides wanting to provide a quality product) to write code right well the first time. I would try to cut technical debt further by having others review my code and tests to ensure I didn’t miss anything and so my code can be high quality and as technically sound as possible.