Martin Granados

CS320

12/10/2023

**Project Two**

First, I divided the requirement list into more manageable, smaller chunks before starting to create my test cases. To do this, I took advantage of the modules' resources. This tactic ensured that all requirements were satisfied. In the AppointmentClass lines 12-20, you can see how I was able to implement requirements of length and dates into the code. The code implemented would throw an illegal argument exception if the length went over a certain limit, null was entered, or if a past date was entered. I developed a test for every criterion after developing the classes for the Contact, Task, and Appointment to make sure they were all being fulfilled. For instance, the contact class contained a number of constraints that prevented null from being an option and restricted the length of the string variables to a specific number of characters. I created tests that, if the conditions weren't fulfilled, would raise illegal argument. When creating the tests, I had to ensure that the code performed as intended and that if a user attempted to do something that was beyond the parameters or scope of the requirements, it would either throw an illegal argument or return false. The ContactTest had a coverage of 84.6%, ContactServiceTest had a coverage of 100%, TaskTest had a coverage of 81.8%, TaskServiceTest had a coverage of 100%, the AppointmentTest has a coverage of 100%, and the AppointmentServiceTest has a coverage of 100%. Overall, all of the tests reached a coverage level of at least 80% for all the tests and the tests were generated to make sure that the intended outcome occurred along with aligning with the requirements.

My code is sound since the test and variable names are meaningful and comprehensible to all readers. Additionally, the exams cover the particular needs of the assignment and are straightforward in what they are testing. If a new task is created with the necessary variables, it is easy to understand lines 12 through 17 of the Task Test as a test of the tests. Because there is no repetitive code, the code is efficient. It also executes quickly and operates more effectively now that the hashmap is being used. To make the code more legible and understandable, I introduced different tests and names in TaskServiceTest than in the Contact Service. Lines 10 through 30 demonstrate this. I included two tests to verify that a new job had been added and to examine what would happen if a duplicate task were added. This enhanced the functionality of the Contact Class service. I had tests identical to this in the Contact Class Test service, however I combined the assertions into a single test called testAddContact. The new method makes things easier to grasp and offers more information.

Static testing was one type of testing that was used for every milestone. Static testing detects errors in software without running the program's code. J Unit tests were utilized for this. Developers may utilize JUnit, a Regression Testing Framework, to expedite development, enhance code quality, and apply unit testing in Java. Three modules of white-box testing were conducted. Understanding the inner workings of an application's code is the cornerstone of white-box testing, also known as structural testing. It determines if the program's logic and code structure are faulty. White-box test scenarios are only dependable if the tester is cognizant of the application's intended functionality (García, 2017).

Given that we were the ones who wrote the code and were aware of the requirements for its functionality, both of the aforementioned testing techniques were employed. J unit tests were designed to evaluate the results of code generated under specific circumstances. The code itself was never executed, though. The code was enhanced and made more intelligible by the ability to view the result of the function under certain circumstances. Because the code was not run, static testing was employed, and the J unit test aided in the usage of White box testing.

For every module, there was no usage of dynamic testing. Dynamic testing is used to look at the code's dynamic behavior. It comprises assessing the program's suitability for analysis in light of the input and output data. The main objective of dynamic testing is to characterize the dynamic behavior of code. It clarifies how the system physically reacts to factors that change over time (Hambling, 2019). Black-box testing is an additional testing method that I did not employ. Black-box testing is done using requirements alone; it doesn't take into account the data or software structure. It is also known as behavioral or functional testing at times. Test cases for black-box testing are created using the system or component's requirements as a guide. Since the system is a "black box," the only method to comprehend how it functions is to look at the inputs and related outputs (García, 2017).

While all testing methods are necessary for every project, there are a few reasons you could choose to use some over others. It would be wonderful to apply every testing approach in a scenario if there was no project timeframe, but that is not realistic. Static and white box testing are useful methods for testing code without actually running it, as demonstrated by our modules. When you create a product and provide it to the client to test, you may employ blackbox testing. Although the consumer is unaware of the code's inner workings, they may verify that it satisfies their needs.

When first generating the code and classes with the requirements I was employing caution and moving diligently when generating the code. I did this to ensure that I was implement all the requirements within the code. This was even more important when generating the test classes because I had to make sure that I was able to test most of the possible inputs from the consumer end, to make sure that program would run as predicted. In lines 13-20, of the AppointmentServiceTest, one can see that even though we are testing the Appointment Service class, we still need the Appointment class to test our code, showing just how interconnected our code was. I believe that there has to be some kind of bias in the testing of the code as it is impossible to get rid of all bias, but following the requirements can be useful to limit the bias. Also, having an instructor look over the code and provide corrections, can help limit bias.

Having an instructor review the code can help one generate quality code. Generating quality code is vital to create a program that will be able to do what it is intended to go and hopefully only need patches throughout its life. Cutting corners might help you generate code faster, but it won’t help you improve your skills for future assignments, especially as technology continues to improve. One way that I continue to learn more is I continue to watch videos about software engineering even while we have a break from school. I have found that if I don’t watch any videos/examples during my break, I will need some time to refresh my mind of what I have learned. I also like going over the assignments, once the course is finished, and see if I can remember the steps I took, and how I could have improved my work.

**References**

GarcíaB. (2017). *Mastering software testing with JUnit 5 : comprehensive guide to develop high quality Java applications*. Packt Publishing.

Hambling, B. (2019). *SOFTWARE TESTING : an istqb-bcs certified tester foundation guide.* Bcs, The Chartered Instit.