Jenna Case

6/18/2023

Project Two

I try to align my testing approach directly to the software requirements. The software requirements tell you what the criteria are for the input given by the user. This is shown in the project as limiting how long the input can be and the fact that it cannot be null. This allows you to create test cases that fall outside of what is acceptable in order to test that your software will not allow those cases in when the program is running.

I knew that my Junit tests were effective on the basis of coverage percentage because I follow the software requirements. I used those to guide what tests needed to be created and was able to ensure that the cases were covered. Since the requirements were straightforward and didn’t have a lot of stipulations, it was easy to make sure that every case was covered. These cases being anything that was too long or anything that was null. Then it was just a matter of checking that the functions work as well.

I ensured that my code was technically sound and efficient by testing any and all cases that could occur through user input. I did this first by ensuring that data going in to the program follows the requirements set in the software requirements. I made test cases that tested if the length was too long of if the input was null:

//id too many characters

Assertions.*assertThrows*(IllegalArgumentException.**class**, ()->{

**new** Task("1234567890B", "TaskName", "TaskDescription");

});

//null id

Assertions.*assertThrows*(IllegalArgumentException.**class**, ()->{

**new** Task(**null**, "TaskName", "TaskDescription");

});

I also tested the methods the same way. I looked at cases that would not be able to work but I also looked at a case that would work. This ensures that my code is both technically sound and efficient because it performs well and how it’s supposed to. I tested both on bad cases but good cases as well for the functions:

//update description

taskService.UpdateTask(task.GetID(), "This is a new description", 2);

*assertEquals*("This is a new description", taskService.GetTaskList().elementAt(0).GetTaskDescription());

//update bad description

Assertions.*assertThrows*(IllegalArgumentException.**class**, ()->{

taskService.UpdateTask(task.GetID(), "This description is too long anf will throw an error", 2);

});

The software testing techniques I used the most during the module milestones were manual testing and functional testing. I did manual testing by checking my code as I was going and checking how it ran at the end of my coding. I also did some manual testing while writing the tests in the junit if there were any errors in the test. Some of the errors came from the actual code and some were in the test themselves. Looking back at what I coded to figure out which was wrong was part of the manual testing.

Functional testing came with the actual tests that I wrote. Functional testing looks to ensure that the functional requirements of the software were met. I did this by going over the software requirements set in the modules and writing tests to make sure that my code followed that. This is seen better in the input side where I made sure that the input was not too long or null per the module requirements.

I did not look at security testing in these modules. I also did not look at non-functional testing. I was more focused on the functional aspects of the code since that it what was required. However, in larger scale projects or when putting multiple things together non-functional testing will become critical. Another test that would become more important that I did not use would be integration testing. There were not a lot of different components in these exercises, so I did not focus on integration testing.

All these different techniques have specific purposes when you are testing code. Some of these are a lot better for unit testing and looking at the code in broken down units. I think this is where the manual testing and functional testing are best as I followed in my own work. When the projects are larger scale and have a lot more components in them, integration testing becomes very important to make sure that all of it can work together and testing that they work well together as one system. Security testing is crucial for real world applications because you need to ensure that your information is safe. You also need to make sure that the security in place functions to the requirements set by the customer.

It is important to employ caution as a software tester because there are a lot of parts working together that need to function smoothly. You have to know the code that has been created, but you also have to understand how the code is going to be used. I used cation when testing my code by making sure that I tested all of the software requirements and by checking how much of the code was covered.

It can be difficult to limit bias in the review of code, especially when it is code that you have developed. I tried to eliminate bias by making sure that I was testing as much of the code as I could. I did not assume that my code was perfect because I had created it. It allowed me to double check myself in all aspects from following the requirements to the actual functions of the code. This can be seen by the test coverage as this shows that I tested 78.5% of the code.

It is important to be disciplined in the commitment to quality. Ensuring quality in code makes sure that no defects or bugs are going out to consumers. Defects in code can be detrimental and have dire effects on individuals. These defects could be anything from private information being leaked to failures in mechanical software. This is why cutting corners can be detrimental. In order to avoid technical debt, I will plan out the tests I want to create for code and ensure that I’m covering at least 75% of the code with my tests.

*Software Testing Techniques*. (2021, February 22). GeeksforGeeks. https://www.geeksforgeeks.org/software-testing-techniques/