Wesston Mccollum

April 16, 2023

CS320 Software Test Automation

Prof. Prasad

My unit testing approach was founded on the idea that interactions should be tested on both general usage and edge case. With this approach, I focused first on what the features should not be able to do, then ensured that they can do what they should be able to do with some random examples. This can be seen with my ContactServiceTest file, which had the following tests: testContactListUpdateFirstNameIdNotFound, testContactListUpdateFirstNameTooLong, and testContactListUpdateFirstName. These tests all serve to test the update function of the Contact Service class. My first two tests check for the edge cases of updating a first name, being that the length of the new name cannot be longer than ten characters long and that the Contact ID is in the list. It then proceeds to try a first name update with proper parameters. This ensures that the class will not do what it should and that the class will do what it should. When testing with this approach, I received coverage rates above 80%, which shows that I tested a large portion if not all of the code I had developed.

These high coverage rates are due to my work in ensuring that the code I developed was sound. It covered the areas specified by the software requirements while also remaining within the confines of the requested limits. Such examples of these limitations were ensuring that unique instances had contacts, addresses, and tasks with unique attributes such as first names, last names, IDs, and more. Even these attributes had their requirements as they could not initially be set to null nor be over a certain number of characters in length. Many of these constraints were dealt with at variable creation, allowing me to handle them as necessary more efficiently.

For this project, I mainly focused on white box testing, which is, “… an approach that allows testers to inspect and verify the inner workings of a software system—its code, infrastructure, and integrations with external systems” (White Box Testing, 2020). My approach was to use the code itself to ensure that all requirements were met. This is different from something like black box testing, which entails giving a function an input that should have a given output, but if it does not it is noted and worked on later after all functions are tested (Black box testing, n.d.). Black box testing is most useful when there are a finite number of options to choose from to test, but in cases where options are only slightly constrained, white box testing is more efficient.

When working on this project, I felt very confident in what I was setting out to accomplish. There was not much room for doubt as the requirements set forth a clear idea of what was desired, and I knew I could deliver them with a high degree of success. I also understood where I needed to approach certain aspects of my code with caution. The ability to set new information into a variable is always a risky capability to give users, which I knew I had to restrict carefully. While the current version of the project may not have very many issues with this that could affect it, later versions might run into problems. One such factor is not allowing a variable to be null. This could have serious consequences in a database if allowed so it is important to try to be as preventative as possible. This also came into play when starting to test my code. There were some areas in my code such as getter functions where I did not inherently feel the need to write tests as I saw them as simple matter-of-fact functions. This is a very improper perspective on testing, as anything could be a security risk. I had to overcome my assumptions that the small things work on their own and do not need testing. It is important not to cut corners when testing software. Even the slightest oversight can have massive repercussions, such as allowing a null value in a program. If set into an active program can stop a program from running entirely. It is important to try to cover as many details as possible in testing to ensure that technical debt is kept to a minimum wherever possible.

References

Imperva. (2020, September 24). *What is white box testing: Types & Techniques for code coverage: Imperva*. White Box Testing. Retrieved April 16, 2023, from <https://www.imperva.com/learn/application-security/white-box-testing/>

Java Point. (n.d.). *Software testing: Black box testing - javatpoint*. Black box testing. Retrieved April 16, 2023, from https://www.javatpoint.com/black-box-testing