Dan Stull

CS320

12/05/2022

Reflections Report

1. **Summary**

For this project, I used Junit to individually test the code (individual functions or scenarios presented through the requirements). From there, I used Junit to test entire modules or components of the project. In this way, I could write a little code, check it to make sure it was correct, and from there, complete the entire requirement and then test that. I feel doing it this way, testing incrementally, I was better able to feel out the project and its requirements, which put me on the right path toward a complete and operational program.

This was my first attempt at using Junit. In previous classes, we worked with maven and how to set up a program using it, but never once did we delve into Junit in any real capacity. So, it’s safe to say I was pretty new to the experience. But once I realized how Junit, maven, and the java program all worked together, I found the process, in its own way, interesting and fun.

To make sure the code was technically sound, I followed the guidelines for Junit, which I learned about from our textbook and other sources online. Below is an image of a requirement and the resulting Junit test. It also shows concise and efficient code:

Text

Description automatically generated

Text

Description automatically generated

1. **Reflection**

First, I used module code development, which is to say, I created blocks of complete code and ran it to ensure there weren’t any errors. From there, I created Junit tests and reran the code. I employed this technique throughout the milestones. As for Junit software testing techniques, I used assertThrows() and assertEquals() quite often, as they seemed the easiest to use, and most times, the testing called for their use. I used assertNull() and assertNotNull(), which were nice when checking to see if something was successfully deleted or added to an array or list.

As for other testing techniques, there is line-by-line testing, better known as static testing. It works just how it sounds, parsing each line of code while looking for syntax or spelling errors. I did read over my code, but I can’t say I parsed through each line.

There are many others I didn’t use, such as Boundary Value Analysis, Equivalence Partitioning, Decision Table testing, and many others. Each of these uses some form of comparing, ensuring the output of the functions or methods is between two set values or equals (or doesn’t equal) a specific value. And each is valuable in its own way and can be used very similarly to how we used Junit testing. The benefit of Junit testing is that it is built into the maven build, and much of the work is done for us.

When first stepping into the shoes of a developer for this project, I had a less-careful approach. I went at the first milestone with a bit of recklessness and an unfocused point of view of where I needed to end up. But as I kept running into obstacles, I realized my approach was the cause, not the program itself. That said, I realized I needed to take a different approach. Using more caution and breaking things down into smaller portions, I could build each section of the program better. And as I built each section, I could better see how they all fit together. And from there, how the testing would fit into the mix.

It would have been easy enough to simply assume my code was correct and only test things in one broad stroke. But taking a step back and realizing I don’t know everything and that errors can happen to even the more advanced developer, I decided to give myself a day to think before applying any Junit tests. In this way, I was better able to think of ways to test the code that I otherwise wouldn’t have. I made sure to test for the obvious but also the obscure. This way, I could see the project from a new angle. And starting this process from the first milestone allowed me to practice and get better at the testing, doing my best to step away from personal bias.

As I said before, I did cut corners in the beginning. I rushed through the first milestone, certain I could finish it fast and have time for other homework. But I quickly found that this was a recipe for disaster. I ran into many roadblocks. I was frustrated. And more than anything, the code simply didn’t work as intended. This taught me to practice patience. And from patience, I was able to slow down and work on the code modularly. I worked one line at a time, and once one requirement was complete, I created the test and ran it. If it worked as expected, I checked that one off the list and then moved on to the next. I adopted this plan for the entirety of the course, and it made me a better developer—one who builds with discipline rather than speed. Speed leads to issues and many re-works on the code. Slower, more methodical work may seem tedious at the moment, but it limits mistakes, allows for reflection, and creates an environment where creativity and thoughtfulness can blossom.