Project Two

Drew Heard

10/20/2023

SNHU

**Summary**

In terms of testing approaches, the first feature was the only approach that was in any way unique, at least at first. As I was learning about JUnit tests, using exceptions, and just general testing principles, I found myself changing my code quite frequently. When I first began with the Contact service and its tester, I was approaching it in a manual brute force testing way as I have any programs that I have created in the past with lots of variables and commented out variable checks in different areas. After more learning and research, I changed my approach and basically started from scratch. My new approach more reflected the same approach I would take with the two future features which is creating a test for each requirement. These tests would target each input variable and every possible type of input that it could be. The types were empty, valid, or invalid, with each state being exclusive. Even down to the method names, each requirement was addressed by its own test, such as the contact test called “testEmptyID” which ensures that the variable “ID” isn’t null or empty, which is one of the programs requirements.

When planning these tests, I wrote down each requirement and branched out each possible test that came from that requirement. That is how I ensured that every aspect of every requirement was addressed. This is how I approached a one hundred percent test coverage and ensured that my tests were effective. Since I was so confident in my planning, I trusted my Junit tests when they all passed. The way I ensured my code was effective was through the use of illegal argument exceptions. They do a lot of the heavy lifting within the code. They catch the failed requirements and make sure the tests see them. An example of this can be seen in the method “testAddressLong” which tries to pass an input that is too long to the variable “address”. Similarly, a way I ensured that the code was efficient was by making sure to reuse established variables and other code. This can be seen throughout the appointment feature where in line seventeen I begin to create dates for the past, future, and present to be used in future methods and tests.

**Reflection**

I would define my testing technique as systematic, or more specifically, white box testing. Since I had to develop the program and the accompanying tests, I knew pretty much every aspect of the code. I had to import very little and I did my due research on everything imported. If I had used more concrete testing, such as building a GUI and trying to manually input different test cases, that could have been considered more experience-based testing. If I had worked with a team of developers, and we had passed each other different functions without having to know the entire inner workings of what we were using, that would’ve been more black boxed based testing. Experience testing would be a good way to showcase progress to less technically experienced clients and black box testing with a team would be much faster and efficient.

The main areas I exercised caution in development were in planning phases and in post reviews. I knew that with proper planning, I could easily achieve proper execution. Also, after my code and tests were done, I knew it was going to be the last time I look at my code before deploying it, so I had to be meticulous in my checking that requirements were met. With my cautious planning and review, I knew that the complicated systems would work together as I intended. It is important to respect complex systems because these systems are what you build your own system on top of. One example of this during my own development was the use of Java dates. I had to learn to treat them as their own object so that they would not clash with any other variables or aspects of my classes. It was in my reviews that I also had to keep my bias in check. Without team members to double check me, I was responsible for keeping my own ego in check. With my high level of confidence in my planning and testing, I was sure that I had not missed a possible test case. I can see this being a problem in a much bigger, complicated system.

Discipline to quality should be the foundation of all testing, no matter what testing techniques one uses. You want not only high-quality product but also high-quality tests. Low quality tests mean more bugs go missed, more phases of development and testing need to get added, more resources get wasted, and your image and reputation as a developer gets hurt as well. In other words, high quality tests yield high quality products which in the long run gives a higher quality developer.