**Project Two**

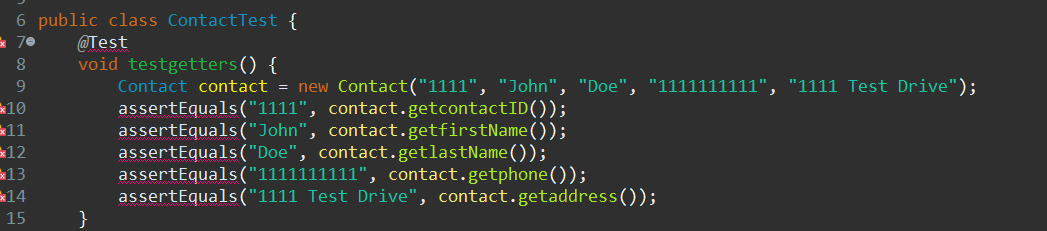
**SNHU: CS-320**

**Nathan Book**

**8/17/24**

When developing Junit tests for the code you are writing, it is important to always approach the development with the requirements in mind. How I achieved this was going through the requirements, one-by-one, and writing code that would accomplish them. This was a progressive approach which was not the most efficient because I was having to go back to previously written code and make amendments based on requirements further down the list. Most of the requirements were about specific data fields and restricting input. An example would be: 

I was certain my Junit tests were effective because after executing the code with the tests implemented, I received all the correct feedback from my test parameters. This was also important in determining if my code was technically sound. If a result did not appear in the console, then I knew the code was ineffective. An example of an effective Junit test is:



The IDE is showing syntax errors for some of those lines, but this is due to not having Junit4 active in the project. It was then later added to permit the execution of the code. In order to accomplish the task of writing efficient code, I made an effort to avoid any redundancies within the code. Meaning, there were not two classes that performed the same function, inheritance was applied to avoid excess code. I also grouped “getters” and “setters” together in the same order as the declared variables. This helped with organization and allowed me to jump around without losing my place.

One of the software testing techniques that I did not use during this project was integration testing because I did not need the different milestones to work together. The other techniques that were not used just did not apply to the testing that was being asked of me, which is why they were not used.

According to the Junit Tutorial resource that was provided for the class, I tested with success cases and behavior cases. The success case was when the input data was within the predefined margins and the code spits the correct input back out. This was demonstrated in the second image in the paper. The second type was the behavioral cases where I tested the input limits. I was looking for the handled exceptions that were implemented into the code if the input were to be outside the requirements.

My mindset while working these projects was focused on fulfilling the requirements. I was cautious at first, double checking each line of code that a wrote and the variables/classes being called. Once I figured out the template for creating these classes, the caution slipped away and I double checked less and less. I believe it is important to appreciate the complexity and interrelationships of the code because these assignments are likely to be like the kind of work I will be doing as a software engineer. If I am being completely honest, for years, I been practicing the art of withholding personal bias in objective situations. I did not reach a roadblock in which my bias was in the way because I know how it can complicate problem solving. It is important to be a disciplined developer and not cut corners because the stakeholders and company I’ll be working for would not continue to employ a developer who repeatedly releases defective code. I take pride in my work, so if I were to hand someone a project of mine, I would ensure that it will do what I described without fault. It is my reputation that will be tarnished if I slack or cut corners.