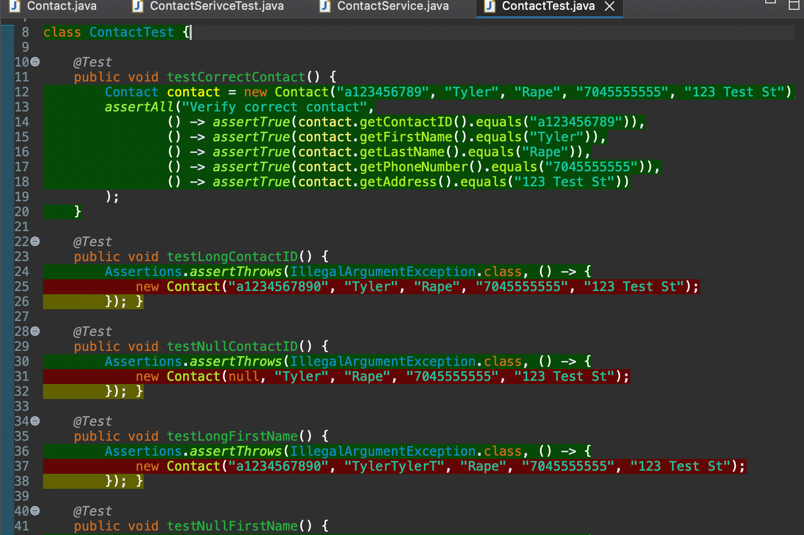
**Project Two**

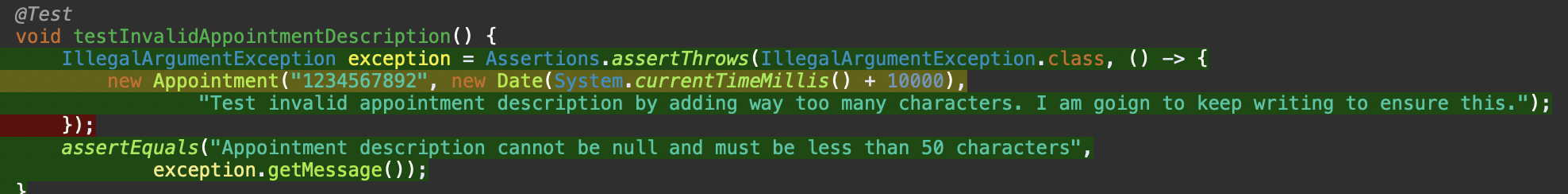
Summary

For each of the three features, I focused on testing exactly what was in the requirements. For each test, I would start with a test case that complied with all inputted requirements. All three asked that we have the functionality to add, delete, and update components of each service. To satisfy this, I added tests for all. Each also required a unique ID. In this case, I tested a case of trying to add a duplicate name. Each component required that nothing be null. In each case, I tested to ensure that it would not accept null feedback. Throughout there were random requirements that had differing amounts of character allotment. In these cases, I also tested that it not only accepted appropriate feedback but that it tested against edge cases that would exceed these limits.

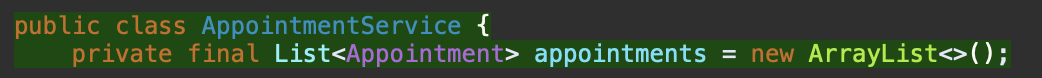
Defending the quality of my JUnit testing through coverage will be a bit difficult. I am sure that I need a better grasp on why my coverage was so low when testing exactly what was required. I found that many of the things that were dragging down my average was the input lines for each test case. I feel that I might need to work with testing a bit more in general to get a better understanding of this. See the picture below as an example:



To ensure that my code was technically sound, I later added assertions that the error message would also be verified. This not only verifies that the program is only accepting appropriate feedback, but also that the error message to the user is correct. Below you can see this take place when I am verifying the appointmentDescription will only accept 50 characters or less.



In previous projects, I learned that writing efficient code can rely heavily on data structures. For this project, I chose to store the data in an array. This was because of the simplicity of the project being assigned and because there wasn’t much data to put into this structure. Below, you will see where I created an Array List to store the appointments in the program.



Reflection

For this project, I employed equivalence some black box testing techniques, including partitioning and boundary testing. Black box testing techniques focus on creating tests from the specifications. These specifications could be created from requirements and or otherwise (Hambling, et al., 2019). In the case of equivalence partitioning, I tested valid and invalid data. In boundary testing, I was focusing on edge cases, like testing over the maximum character limit, or dates in the past, in the case of the appointment settings.

Other black box testing techniques that I did not employ were decision table testing, and performance testing. Decision table testing will test a variation of valid and invalid input will give the correct output (Hambling, et al., 2019). As a program becomes more complicated, this is especially useful as one would wish a program would give the desired output with combinations of both. Performance testing would test how the program works under a heavy load. This would be especially helpful in preventing bottle-necks.

Equivalence partitioning is useful to ensure that valid and invalid data perform properly in a program. Boundary testing is especially useful in projects like these because it identifies the strict boundaries that we have placed on all of the inputs. Decision table testing would provide better insight if the program had many business rules to follow. If this were a final, public facing application, performance testing would be useful for identifying ways that the system could be overloaded.

This project was interesting because we were not only the tester, but we were the developer. It gave a unique perspective. When writing the code, I was thinking about not only how it would work but how it would test. I am sure many developers work this way, but it was my first time doing so. Understanding this complexity and interrelationships of testing and coding is important because not only do you want the program to function, but you want the program to perform in the way it should under rigorous testing conditions.

Identifying your own bias is a near constant struggle in every way of life. This is also true in coding and testing. When you create your code, you want to be sure of what you are doing and this can lead to feeling infallible, which is not the best stance to have. We need to know, as developers, that what we have created could not work properly when it comes to testing. I have learned from a good friend that is a software developer that he starts with writing the tests, and then writes code until the tests perform properly. While he is testing his own code from the beginning, he knows that even his tests might not be up to par. Learning this helps keep your ego in check. For this reason, when I received feedback from our instructor, I remembered to not take it personally, but instead use it as a critical feedback to make a better program.

Before this project, I would typically code a program, and then run the program, manually testing the feedback. Now, that I have learned to employ unit testing, I am more careful in my coding. I am thinking about how the code will perform under this testing. An example that comes to mind is when I was creating the IDs for each section. These were supposed to be immutable. I would have originally written that code without making it final, then later adding that detail. Because I knew I was going to be testing against it, I made sure it was written in the code to begin with. This, I feel, is common laziness. Learning testing has led to me being more disciplined in my commitment to quality, which helps avoid technical debt, like simple misses.

References

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing : An istqb-bcs certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.