CS-320-T5520 - Project Two – Lance Cain

**Summary**

The unit testing methodology I used when creating the three entities and their corresponding services, consisted of testing for both valid and invalid operation results. I did this by testing functionality to ensure that it matched the given specifications for software requirements while also trying to be comprehensive and analyze the requirements for potentially desired requirements that were not listed.

I designed the unit tests to reflect the requirements as can be seen in any of the entity tests or the services tests. The assertions in these tests specifically match the limits given by testing each parameter to verify that the length constraints, and in some cases is not null or is unique. In some cases characters maybe included that should not be so some limitations on characters exist instead of simply not null and length being the only requirements. Doing so can avoid potential future issues.

The quality of the Junit tests would be regarded as high as they comprehensively cover the requirements for the classes created whether entities or services. When creating these tests I included creating, updating, and deleting tests to verify that all necessary functions worked as expected. One example of this can be seen in analysis of my Appointment Service Test as it includes assertions for adding, updating and deleting the entity.

Writing tests for any language is a technical challenge but a logical one as well because you have to think comprehensively about all possibilities that code can be used for or ways it may malfunction. In the case of the entities and the testing I used a single responsibility mindset to insure that testing could be done on each function and entity and all requirements individually to pinpoint errors and assist in code stability. This can be seen in any of the testing written as each requirement is individually tested as to avoid any odd behavior between multiple incorrect parameters allowing for errors to remain hidden behind other errors.

For efficiency in testing I used one test method and used assertThrows method to validate multiple scenarios in one test without sacrificing the detail and information necessary to properly debug errors.

**Reflection**

Testing Techniques

I used both white-box and black-box testing techniques by testing the parameters to each of their limits while also testing the internal logic of the code. Tests such as minimum and maximum lengths for a parameter input.

The techniques not implemented were primarily around scalability as well as more complex stress testing around input limits as in a real world scenario I would likely have inquired for further details as to how the entities were to be used to more comprehensively analyze possible methods of attack of the code to prevent cyber security issues as well as unexpected bugs.

White-box testing is great for finding hidden errors and can sometimes expose code inefficiencies as well. Black-box testing is great for validating that the user requirements are being met as well as potential integration issues can be discovered at this point as well. Testing for boundary level issues such as injection attack vulnerabilities is also crucial to stable and correctly operating code. Stress testing in scalable loads can also help to find code inefficiencies and unexpected instability.

Mindset

Understanding the complexity of the code is paramount for the test writer, without understanding the relationships between the different classes and functions within them, you run the risk of not comprehensively testing the different ways the code interacts with the system it ultimately becomes a part of. Understanding how the entities and the service for that entity interact with each other and how one changing can affect the other, is necessary to write tests to validate the requirements and stability of that function.

When creating functions or classes, and especially even larger pieces of code, sometimes you can overlook a scenario that could cause your code to malfunction or to be vulnerable to threat actors. Proper testing techniques and comprehensive analysis can help alleviate this possibility but having someone else at minimum review the code and tests to further back that you haven’t missed any testing that would be necessary for good quality code, is still a solid plan. I try to be thorough and analyze code from a logical point of view instead of an opinionated view. When looking at code from a logical point of view it helps to remove many bias’s that our own personal opinions would likely affect the outcome.

Having an almost perfectionist perspective is truly required for writing good quality code as you have to write solidly functioning code that remains bug free and stable in all the expected operating conditions. Cutting corners and making assumptions can be detrimental to any part of a code base or really any part of any project. In the case of software development, not properly addressing issues with validated and good quality solutions can result in technical debt that will later cause more and more issues and in some cases, can derail entire projects. Adhering to coding standards and being thorough in test cases and in general following best practices for all parts of the development process is paramount. Using the same practices and level of commitment throughout a project regardless of obstacles or adversities is vital as well, for example using the same patience and effort level during creating all three of these entities and their services and tests for all of those, is vital for the project to function smoothly and efficiently.