**Aws Albayati**

**CS-320-T3329 Software Test Automation & QA 23EW3**

**ID: 2079977**

**Summary**

To succeed in unit testing we have to design our code in the beginning, to be unit-test friendly. My main rule is each method must perform only one function. I always ask myself this question: How many lines of code it will take to perform a unit test on this method? If the answer is more than 5 lines, it means the code design is wrong or maybe the method needs to be broken down into smaller functions. In other words, the unit test must be accounted on the early stages of code design. It's not an afterthought that can be dealt with later. The best way to accomplish a clean unit-testable code is by designing the code based on requirements where each method corresponds to one requirement, this is what is known as a white-box test. This is the part where I check that the logic is technically sound. Also, it's important to write the unit tests based on the order of the constructor to achieve efficiency. Each method in the object has its unit test that covers positive and negative testing.

In the contact and task base classes, I used getter and setter methods. Each setter checks for requirements and returns an exception on failure. This will allow me to check for any rise exception on each setter (Method) and compare the result with each getter in the object using the unit test. On the other hand, in the service class, I designed the object based on the requirements where each requirement has its method that can be unit-tested easily. In our case Add, Delete, and Update functions have their method that can be tested separately using the unit test.

**Reflection**

The testing technique I used in this project is white-box testing (also known as clear-box testing). White-box testing is all about being knowledgeable about the inner functionalities of the code. Hence why it's also called clear-box testing. This type of testing must be performed by the developers themselves. To implement white-box testing, I designed each object and method to be unit testable. This design will allow me to inspect and verify the inner functionality of the software and give me the ability to achieve complete code coverage. The only consequence of this method of testing is the loss of time that the developer has to spend writing these unit tests.

There are other test techniques that I did not implement in this project. For example, black-box testing is an important test to ensure the test result is a bias-free. Black-box requires testing the application from the user's perspective without knowing the code. This means it requires a QA tester other than myself to perform the test. Another testing technique that I could perform in this project is Static Testing (performing a code inspection without executing the application). This test method keeps the application secure by catching all software vulnerabilities in the early stages of the project. The consequence of this method is we often get false positive vulnerability results which can be time-consuming.

When it comes to the mindset, as I mentioned earlier in the summary, it's all about being cautious when designing the code. Breaking down the code into smaller unit-testable functions is what keeps my code clean and limits my bias when reviewing it. Making the methods as simple as possible will almost eliminate developer biases. It's easy to illuminate the errors when each method does one function at a time. In my opinion, taking the time to design the code is the key to developing high-quality secure software that can be easily maintained in the future.

**References**

* Cser, T. (2023, January 30). *Black-box testing: Definition, process.* Black-Box Testing: Definition, Process. Retrieved February 19, 2023, from https://www.functionize.com/automated-testing/black-box-testing
* Hamilton, T. (2023, January 21). *White box testing – what is, techniques, example & types.* Guru99. Retrieved February 19, 2023, from https://www.guru99.com/white-box-testing.html