CLEARENT Coding Challenge

I have uploaded the solution to GitHub which has the following classes in the solution:

Project: ClearentCodingChallenge

1. Card

* DiscoverCard.cs
* MasterCard.cs
* VisaCard.cs

1. Person

* Person.cs

1. Wallet

* Wallet.cs

I did create another project in the solution for unit testing called UnitTests. I have experience working with MSUnit Test, NUnit Test and xUnit Test. For this project I have used NUnit Test. All tests are running successfully. I was not sure if I had to implement mock for those unit tests, so I have not implemented that but in case that is needed I do have experience doing that. Based on my understanding, the task was to create a program to calculate credit card interest for a person and based on test cases write unit tests and explain how I used SOLID principles in the code. So, here is the explanation of SOLID principles and how those are implemented in the code.

SOLID Principles Explanation:

1. The Single Responsibility Principle : Every object should have a single responsibility and that responsibility should be entirely encapsulated by the class.

- The**single-responsibility principle** (SRP) states that each class, module, or function in the program should only do one job. In other words, each should have full responsibility for a single functionality of the program. The class should contain only variables and methods relevant to its functionality. In the code, we have two different classes called Person and Wallets which calculates the interest and sets the value. Wallet class is used to set card values whereas Person class is used to se wallet values. Thus, it is only taking one responsibility for one class and it is encapsulated by that class.

2. The Open / Close Principle : Open to extension but closed to modification. What is your pattern approach?

- This principle basically means that it should be open for extension but closed for modification. The code is written in such a way that the new functionality can be added only when new requirements are generated, and we have already developed a class that has gone through unit testing so we should not alter it and use that class for extension. In the code we have credit card interface which is inherited by different card types. If in future, if we support another credit card type for example: Citi Card then we do not need to make any modifications to CreditCard interface. We can just create another class called Citi Card and inherit CreditCard interface. Thus, the credit card class is open for extension but closed for modifications.

3. The Liskov Substitution Principle : Are your subtypes substitutable for your base types?

- This principle basically means that any class must be directly replaceable by any of its subclasses without error. The child class must be able to process all the same requests and complete all the same tasks as its parent class. In the code base this is how we satisfy this principle: We have a class called Credit Card which is further divided into Credit Card types: Discover, MasterCard and Visa. Now, any subtype of Credit Card can be replaced with the other subtype without any error. As a result, this style now achieves the Liskov substitution principle.

4. Interface Segregation Principle : Prefer small, cohesive interfaces to ‘fat’ interfaces.

- This rule means that when one class depends on another then the number of members in the interface that is visible to the dependent class should be minimized. Classes do not include behaviors they do not use. In the code this is how it is implemented: we have a class called Wallet which does not inherit Credit Card as some methods inside the CreditCard interface are not used in Wallet class like getBalance() and setBalance(). We are not forcing any class to have unnecessary methods that they do not use. We have three classes which inherits CreditCard interface, and it uses all those methods declared in there. In this code implementation we have small and cohesive interface instead of big ones.

5. Dependency Inversion Principle : Do you have hidden dependencies that prevent you from testing without fakes or mocks?

- No hidden dependencies are in the code that will prevent us from testing without fakes or mocks. The way dependency inversion principle is implemented in the code can be explained by the following diagram:

Inherits

References

* So, first we created an interface called CreditCard which has all the methods needed to calculate the interest. Then we created three types of credit card classes called: Discover, MasterCard and Visa which inherits CreditCard. By doing so, we achieve the second part of DIP principle i.e., abstraction should not depend on details. Details should depend on abstractions. This principle decouples high-level and low-level components and instead connects both to abstractions. The major advantage of using this is that decoupled programs require less work of change i.e., a single change can affect many separate parts. There is a class called Wallet which implements the interface CreditCard thus satisfying the principle. We are fully overhauling behind the scenes implementation of how the program achieves its behavior without the user’s knowledge.