104222099 – Phan Vu

TASK 2

1. **Describe the principle of polymorphism and how it was used in Task 1.**

One of the most fundamental concepts in Object-Oriented Programming is polymorphism, which allows objects of different types to be viewed as belonging to the same category, making the code more versatile and extendable. This makes it easier to write code that can handle a wide range of objects without having to be changed or rewritten for each type. Additionally, extensibility lets you add new features to the code by adding new classes without changing the code that depends on the superclass or interface.

MinMaxSummary and **AverageSummary** are two different strategies in Task 1. They are both from the **SummaryStrategy** class. To put it another way, there are two types of Summary Strategies: **MinMaxSummary** (minmax strategy (local variable)) and **AverageSummary** (averagestrategy(local variable)). For example, the DataAnalyzer class can call either

```
DataAnalyser analyser = new DataAnalyser(new MinMaxSummary(), numbers);
```

Or

```
analyser.Strategy = new AverageSummary();
```

The Summarize is executed in different ways based on the strategy

2. **Using an example, explain the principle of abstraction. In your answer, refer to how classes in OO programs are designed.**

Abstraction in object-oriented programming simplifies complicated systems by concentrating on property roles and avoiding implementation specifics. It lets you make models or representations of things or processes that exist in the real

world, with only the most important data and behaviors included for each situation. By hiding the details, the system is easier to handle and can be broken down into smaller parts. It also has a more straightforward way to deal with things. Here is an example:

```
public abstract class Vehicle
{
    public abstract void Move();
}

public class Car : Vehicle
{
    public override void Move()
    {
        Console.WriteLine("The car is moving.");
    }
}

public class Bike : Vehicle
{
    public override void Move()
    {
        Console.WriteLine("The bike is moving.");
    }
}
```

The Vehicle class in this case is an abstract parent class that defines the Move method. The Concrete class Vehicle is where the Car and Bike classes come from, and they both implement the Move method. Now I'm gonna make a list of Vehicle objects and move each one to show how abstraction works in more detail.

```
public static void Main(string[] args)
{
    List<Vehicle> vehicles = new List<Vehicle>();
    vehicles.Add(new Car());
    vehicles.Add(new Bike());

    foreach (Vehicle vehicle in vehicles)
    {
        vehicle.Move();
    }
}
```

The Main method generates a Vehicle list of Car and Bike objects. Abstraction lets us invoke the Move method on any Vehicle object regardless of its kind. We

can operate with a collection of objects without caring about their implementation.

3. **What was the issue with the original design in Task 1? Consider what would happen if we had 50 different summary approaches to choose from instead of just 2.**

The code in the original design lacks OOP structure, making the program insufficient and potentially becoming more complex as more approaches are added.
If there were 50 alternative summary ways instead of just two, more memory would be needed at initially than if the OOP structure was employed. Also, the writer has to use the same if/else statements over and over, which can make the code hard to read and make it harder to fix bugs. However, using OOP structure (ideas like inheritance, polymorphism, encapsulation, and abstraction) as we did in Task1 means that common methods can be used again and again, making it easier to add to or change code without worrying about how easy it is to read.