Design Pattern in .NET

Objectives

- Overview about Design Pattern
- Why must use Design Pattern?
- Explain about Singleton Pattern
- Explain about Factories Pattern
- Explain about Abstract Factory Pattern
- Explain about Builder Pattern
- Explain about Prototype Pattern
- Demo about Factories Method, Prototype Pattern and Singleton Pattern in .NET applications

Understanding Design Patterns

- A design pattern provides a general reusable solution for the common problems that occur in software design
- A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations
- Design patterns are programming language independent strategies for solving a common problem. That means a design pattern represents an idea, not a particular implementation

Understanding Design Patterns

- Design patterns are not meant for project development. Design patterns are meant for common problem-solving and whenever there is a need, we have to implement a suitable pattern to avoid such problems in the future
- The pattern typically shows relationships and interactions between classes or objects
- By using the design patterns we can make our code more flexible, reusable, and maintainable
- The Gang of Four (GoF) patterns are generally considered the foundation for all other patterns. They are categorized in three groups: **Creational**, **Structural**, and **Behavioral**

Why use Design Pattern?

- Design patterns can speed up the development process by providing tested, proven development paradigms
- Effective software design requires considering issues that may not become visible until later in the implementation
- Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns

Why use Design Pattern?

- Often, we only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem
- Patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs

Introduction to GoF Patterns

- **Creational Design Patterns**
  - These design patterns are all about class instantiation or object creation
  - These patterns can be further categorized into Class-creational patterns and object-creational patterns
  - While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done
  - Creational design patterns are the Singleton, Factory Method, Abstract Factory, Builder, and Prototype
- **Structural Design Patterns:** Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data, and Proxy
- **Behavioral Design Patterns:** Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method and Visitor

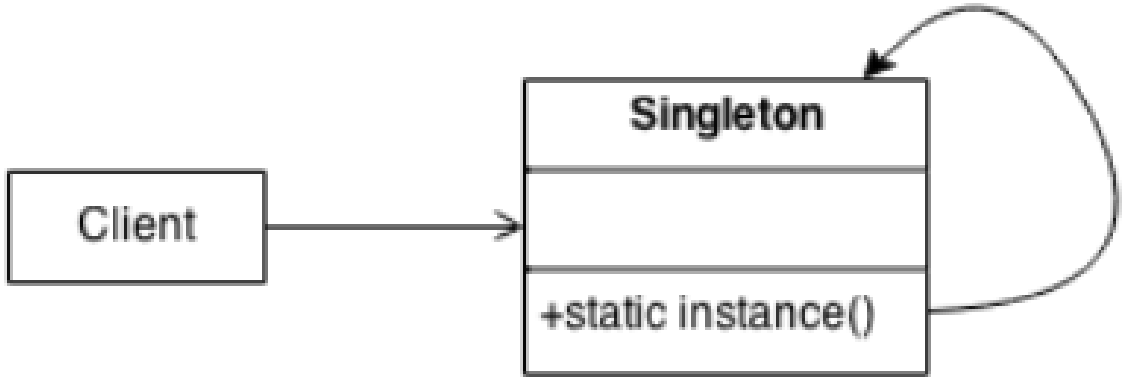
Creational Design Patterns

- **The following five patterns are concerned with the instantiation of objects:**
  - **Singleton**: A pattern for enforcing only one instance of a class
  - **Factory Method**: A pattern for creating objects derived from a class where the specific class is determined at runtime
  - **Abstract Factory**: A pattern for the creation of objects belonging to a family of classes. The specific object is determined at runtime
  - **Builder**: A useful pattern for more complex objects where the construction of the object is controlled externally to the constructed class
  - **Prototype**: A pattern for copying or cloning an object

Singleton Pattern

Singleton Pattern

- Defines an Instance operation that lets clients access its unique instance. Instance is a class operation
- Responsible for creating and maintaining its own unique instance



- Make the class of the single instance responsible for access and "initialization on first use". The single instance is a private static attribute. The accessor function is a public static method

Implement Singleton Pattern

```
//sealed class : this class cannot be inherited
public sealed class Singleton {
    private static readonly Singleton Instance;
    private static int TotalInstances=0;
    /*
     Private constructor is used to prevent
     creation of instances with 'new' keyword outside this class.
    */
    private Singleton()=> Console.WriteLine("--Private constructor is called.");
    //Using Static constructor
    static Singleton() {
        // Printing some messages before create the instance
        Console.WriteLine("--Static constructor is called.");
        Instance = new Singleton();
        TotalInstances++;
        Console.WriteLine($"--Singleton instance is created. Number of instances:{ TotalInstances}");
        Console.WriteLine("--Exit from static constructor.");
    }
    public static Singleton GetInstance => Instance;
    public int GetTotalInstances => TotalInstances;
    public void Print() => Console.WriteLine("Hello World.");
}
```

Implement Singleton Pattern

```
class Program {
    static void Main(string[] args){
        Console.WriteLine("#1.Trying to get a Singleton instance, called firstInstance.");
        Singleton firstInstance = Singleton.GetInstance;
        Console.Write("--Invoke Print() method : ");
        firstInstance.Print();
        Console.WriteLine("#2.Trying to get another Singleton instance, called secondInstance.");
        Singleton secondInstance = Singleton.GetInstance;
        Console.WriteLine($"--Number of instances:{secondInstance.GetTotalInstances}");
        Console.Write("--Invoke Print() method : ");
        secondInstance.Print();
        if (firstInstance.Equals(secondInstance)) {
            Console.WriteLine("=> The firstInstance and secondInstance are the same.");
        }
        else{
            Console.WriteLine("=> Different instances exist.");
        }
        Console.Read();
    }
}
```

```
D:\Demo\FU\Basic.NET\Slot_07_DesignPattern\Demo_Singleton_Pattern\bin\Debug\net5.0\Demo_Singleton_Pattern.exe
#1.Trying to get a Singleton instance, called firstInstance.
--Static constructor is called.
--Private constructor is called.
--Singleton instance is created. Number of instances:1
--Exit from static constructor.
--Invoke Print() method : Hello World.
#2.Trying to get another Singleton instance, called secondInstance.
--Number of instances:1
--Invoke Print() method : Hello World.
=> The firstInstance and secondInstance are the same.
```

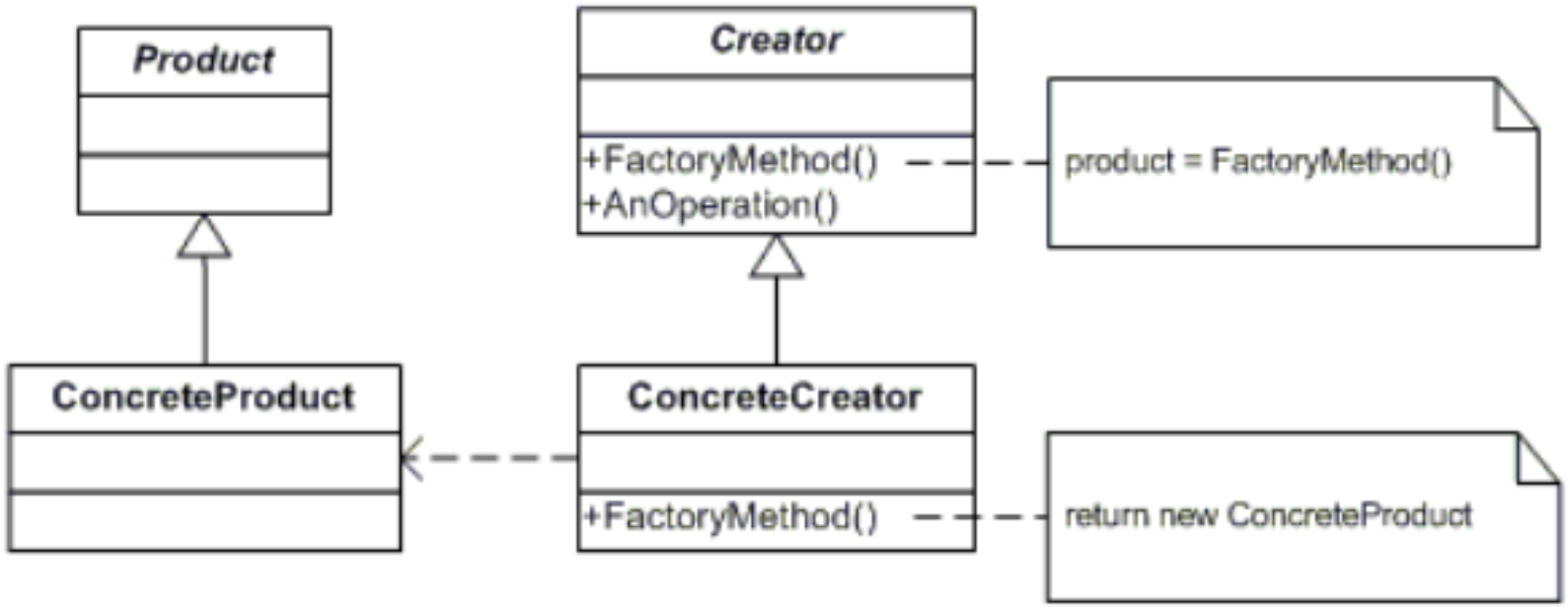
Factories Method Pattern

Factories Method Pattern

- Factory method related to object creation. Define an interface for creating an object, but let subclasses decide which class to instantiate
- Factory Method lets a class defer instantiation to subclasses
- In Factory pattern, we create object without exposing the creation logic to client and the client use the same common interface to create new type of object
- The idea is to use a (static)member-function or (static)factory method which creates and returns instances, hiding the details of class modules from user

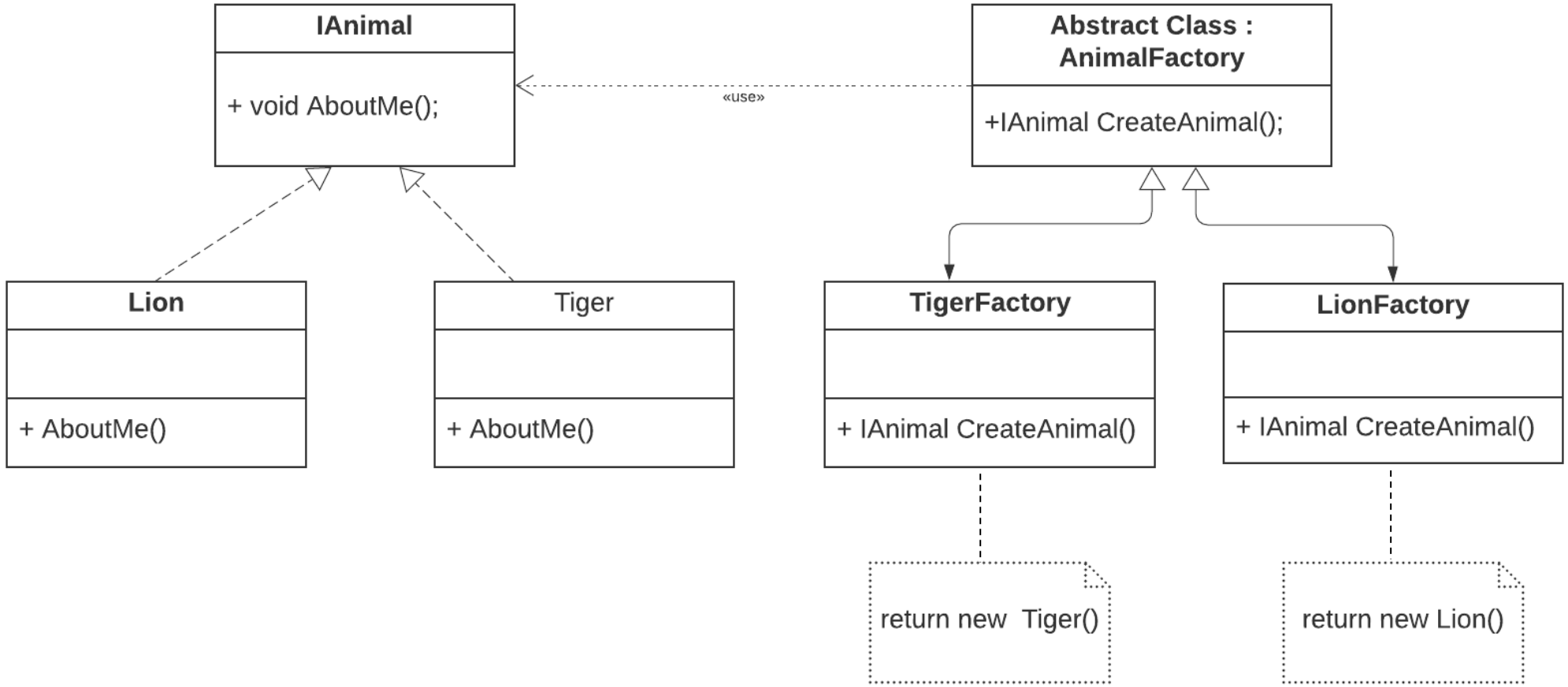
Factories Method Pattern

- A factory pattern is one of the core design principles to create an object, allowing clients to create objects of a library in a way such that it doesn't have tight coupling with the class hierarchy of the library



UML class diagram

Implement Factories Method



UML class diagram

Implement Factories Method

```
using System.Collections.Generic;
using static System.Console; //using static directive
namespace Demo_Factories_Pattern
{
    //Both the Lion and Tiger classes will
    //implement the IAnimal interface method.
    public interface IAnimal{
        void AboutMe();
    }
    //Lion class
    public class Lion : IAnimal{
        public void AboutMe() => WriteLine("This is Lion.");
    }
    //Tiger class
    public class Tiger : IAnimal{
        public void AboutMe()=>WriteLine("This is Tiger.");
    }
```

Implement Factories Method

```
//Both LionFactory and TigerFactory will use this.
public abstract class AnimalFactory{
    /*
    Factory method lets a class defer instantiation to subclasses.
    The following method will create a Tiger or a Lion,
    but at this point it does not know whether it will get a Lion or a tiger.
    It will be decided by the subclasses i.e.LionFactory or TigerFactory.
    So, the following method is acting like a factory(of creation).
    */
    public abstract IAnimal CreateAnimal();
}
//LionFactory is used to create Lions
public class LionFactory : AnimalFactory {
    //Creating a Lion
    public override IAnimal CreateAnimal()=>new Lion();
}
```

Implement Factories Method

```
//TigerFactory is used to create tigers
public class TigerFactory : AnimalFactory{
    //Creating a Tiger
    public override IAnimal CreateAnimal()=>new Tiger();
}
class Program {
    static void Main(string[] args){
        Console.WriteLine("***Factory Method Pattern Demo.***\n");
        //Create a list AnimalFactory included TigerFactory and LionFactory
        List<AnimalFactory> animalFactoryList = new List<AnimalFactory>
        {
            new TigerFactory(),
            new LionFactory()
        };
        foreach (var animal in animalFactoryList){
            animal.CreateAnimal().AboutMe();
        }
        ReadLine();
    }
}
}
```

```
D:\Demo\FU\Basic.NET\Slot_07_DesignPattern\Demo_Factories_Pattern\
***Factory Method Pattern Demo.***

This is Tiger.
This is Lion.
```

Abstract Factory Pattern

Abstract Factory Patterns

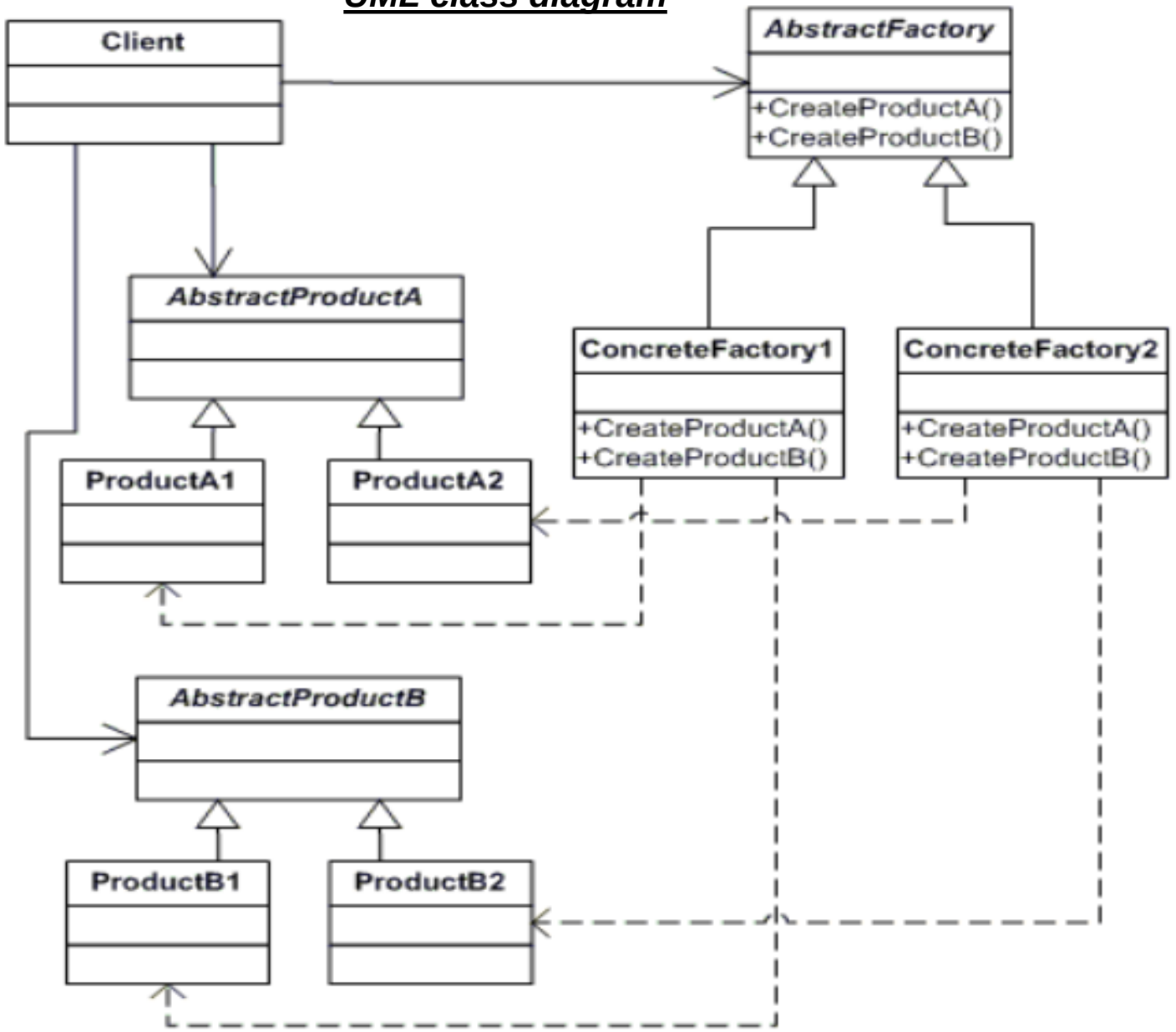
- Abstract Factory patterns work around a super-factory which creates other factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object

- In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern

Abstract Factory Design Pattern Participants

- **AbstractFactory**: declares an interface for operations that create abstract products
- **ConcreteFactory**: implements the operations to create concrete product objects
- **AbstractProduct**: declares an interface for a type of product object
- **Product**: defines a product object to be created by the corresponding concrete factory and implements the AbstractProduct interface
- **Client**: uses interfaces declared by AbstractFactory and AbstractProduct classes

UML class diagram



Builder Pattern

Builder Pattern

- Builder pattern aims to “Separate the construction of a complex object from its representation so that the same construction process can create different representations.”
- It is used to construct a complex object step by step and the final step will return the object
- The process of constructing an object should be generic so that it can be used to create different representations of the same object

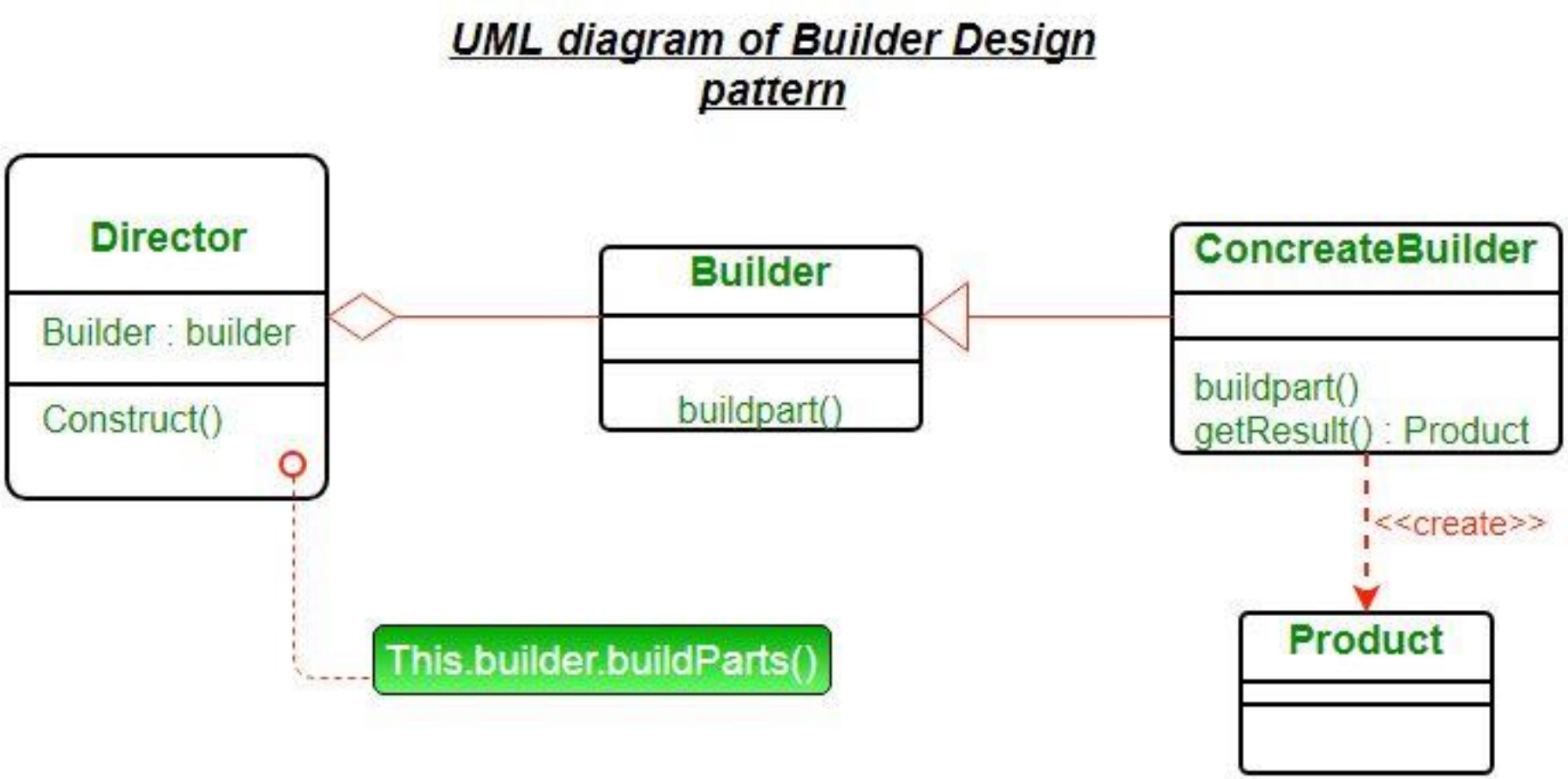
Builder Design Pattern Participants

- **Product** – The product class defines the type of the complex object that is to be generated by the builder pattern.
- **Builder** – This abstract base class defines all of the steps that must be taken in order to correctly create a product. Each step is generally abstract as the actual functionality of the builder is carried out in the concrete subclasses. The GetProduct method is used to return the final product. The builder class is often replaced with a simple interface
- **ConcreteBuilder** – There may be any number of concrete builder classes inheriting from Builder. These classes contain the functionality to create a particular complex product

Builder Pattern

- **Director** – The director class controls the algorithm that generates the final product object. A director object is instantiated and its Construct method is called. The method includes a parameter to capture the specific concrete builder object that is to be used to generate the product. The director then calls methods of the concrete builder in the correct order to generate the product object. On completion of the process, the GetProduct method of the builder object can be used to return the product

Builder Pattern



Advantages of Builder Design Pattern

- The parameters to the constructor are reduced and are provided in highly readable method calls
- Builder design pattern also helps in minimizing the number of parameters in constructor and thus there is no need to pass in null for optional parameters to the constructor
- Object is always instantiated in a complete state
- Immutable objects can be build without much complex logic in object building process

Disadvantages of Builder Design Pattern

- The number of lines of code increase at least to double in builder pattern, but the effort pays off in terms of design flexibility and much more readable code
- Requires creating a separate ConcreteBuilder for each different type of Product

Prototype Pattern

Prototype Pattern

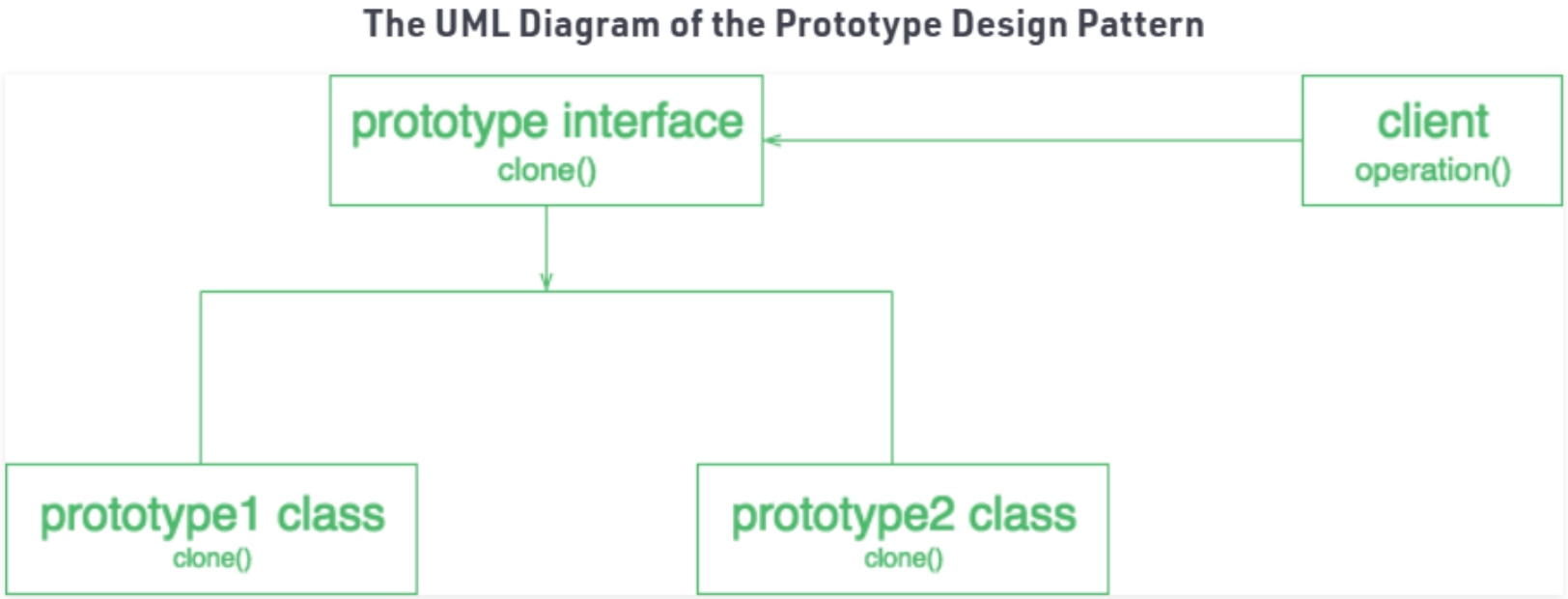
- The prototype allows us to hide the complexity of making new instances from the client
- The concept is to copy an existing object rather than creating a new instance from scratch, something that may include costly operations
- The existing object acts as a prototype and contains the state of the object
- The newly copied object may change the same properties only if required. This approach saves costly resources and time, especially when object creation is a heavy process

Prototype Pattern

- Prototype patterns are required, when object creation is a time-consuming, and costly operation, so we create an object with the existing object itself
- One of the best available ways to create an object from existing objects is the clone() method
- Clone is the simplest approach to implement a prototype pattern. However, it is our call to decide how to copy existing object based on your business model

Prototype Design Pattern Participants

- **Prototype**: This is the prototype of the actual object
- **Prototype registry**: This is used as a registry service to have all prototypes accessible using simple string parameters
- **Client**: The client will be responsible for using the registry service to access prototype instances



Implement Prototype Pattern

```
public abstract class Car {
    protected int basePrice = 0, onRoadPrice = 0;
    public string ModelName { get; set; }
    public int BasePrice{
        set => basePrice = value;
        get => basePrice;
    }
    public int OnRoadPrice
    {
        set => onRoadPrice = value;
        get => onRoadPrice;
    }
    public static int SetAdditionalPrice(){
        Random random = new Random();
        int additionalPrice = random.Next(200_000, 500_000);
        return additionalPrice;
    }
    public abstract Car Clone();
}//end Car
```

Implement Prototype Pattern

```
public class Mustang : Car {
    public Mustang(string model)=>(ModelName, BasePrice) = (model, 200_000);
    // Creating a shallow copy and returning it.
    public override Car Clone()=> this.MemberwiseClone() as Mustang;
}//end Mustang
public class Bentley : Car{
    public Bentley(string model) => (ModelName, BasePrice) = (model, 300_000);
    // Creating a shallow copy and returning it.
    public override Car Clone()=>this.MemberwiseClone() as Bentley;

}//end Bentley
class Program
{
    static void Main(string[] args){
        Console.WriteLine("***Prototype Pattern Demo***\n");
        //Base or Original Copy
        Car mustang = new Mustang("Mustang EcoBoost");
        Car bentley = new Bentley("Continental GT Mulliner");

        //Console.WriteLine("Before clone, base prices:");
        Console.WriteLine($"Car is: {mustang.ModelName}, and it's base price is Rs. {mustang.BasePrice}");
        Console.WriteLine($"Car is: {bentley.ModelName}, and it's base price is Rs. {bentley.BasePrice}");
```

Implement Prototype Pattern

```
            Car Car;
            Car = mustang.Clone();
            // Working on cloned copy
            Car.OnRoadPrice = Car.BasePrice + Car.SetAdditionalPrice();
            Console.WriteLine($"Car is: {Car.ModelName}, and it's price is Rs. {Car.OnRoadPrice}");

            Car = bentley.Clone();
            // Working on cloned copy
            Car.OnRoadPrice = Car.BasePrice + Car.SetAdditionalPrice();
            Console.WriteLine($"Car is: {Car.ModelName}, and it's price is Rs. {Car.OnRoadPrice}");
            Console.ReadLine();
        }//end Main
}//end Program
```

```
D:\Demo\FU\Basic.NET\Slot_07_DesignPattern\Demo_Prototype_Pattern\bin\Debug\net5.0\Demo_Prototype_Pattern.exe
***Prototype Pattern Demo***

Car is: Mustang EcoBoost, and it's base price is Rs. 200000
Car is: Continental GT Mulliner, and it's base price is Rs. 300000
Car is: Mustang EcoBoost, and it's price is Rs. 445159
Car is: Continental GT Mulliner, and it's price is Rs. 559240
```

Summary

- Concepts were introduced:
  - Overview about Design Pattern in .NET
  - Why must use Design Pattern?
  - Explain about Singleton Pattern
  - Explain about Factories Pattern
  - Explain about Abstract Factory Pattern
  - Explain about Builder Pattern
  - Explain about Prototype Pattern
  - Demo about Factories Method, Prototype Pattern and Singleton Pattern in .NET applications