BUILDER DESIGN PATTERN

CREATIONAL PATTERN


```
public class MotorcycleBuilder : IVehicleBuilder
{
    private readonly Vehicle _vehicle = new();

    public void SetType() => _vehicle.Type = "Motorcycle";
    public void SetWheels() => _vehicle.Wheels = 2;
    public void SetEngine() => _vehicle.Engine = "V2";
    public void SetColor() => _vehicle.Color = "Black";

    public Vehicle Build() => _vehicle;
}
```

Karwan Essmat othman

Builder Design Pattern - Creational Pattern

1. What is a Builder Design Pattern?

The Builder design pattern is a way to construct complex objects step-by-step. It separates the construction process from the final representation, allowing ***the same construction process*** to create different representations.

2. Where and Why Do We Use It?

We use the Builder pattern when:

1. We need to create a complex object that requires multiple steps.
2. We want to separate the construction process from the final product.
3. We want to create different representations of an object using the same construction code.

3. What Problems Will We Encounter If We Do Not Use It?

Without the Builder pattern, constructing complex objects directly in a class can lead to:

1. Constructors with too many parameters.
2. Increased complexity and difficulty in managing the class.
3. Hard-to-read and maintain code.
4. Difficulty in creating different variations of the object.\

4. Real-World Scenario

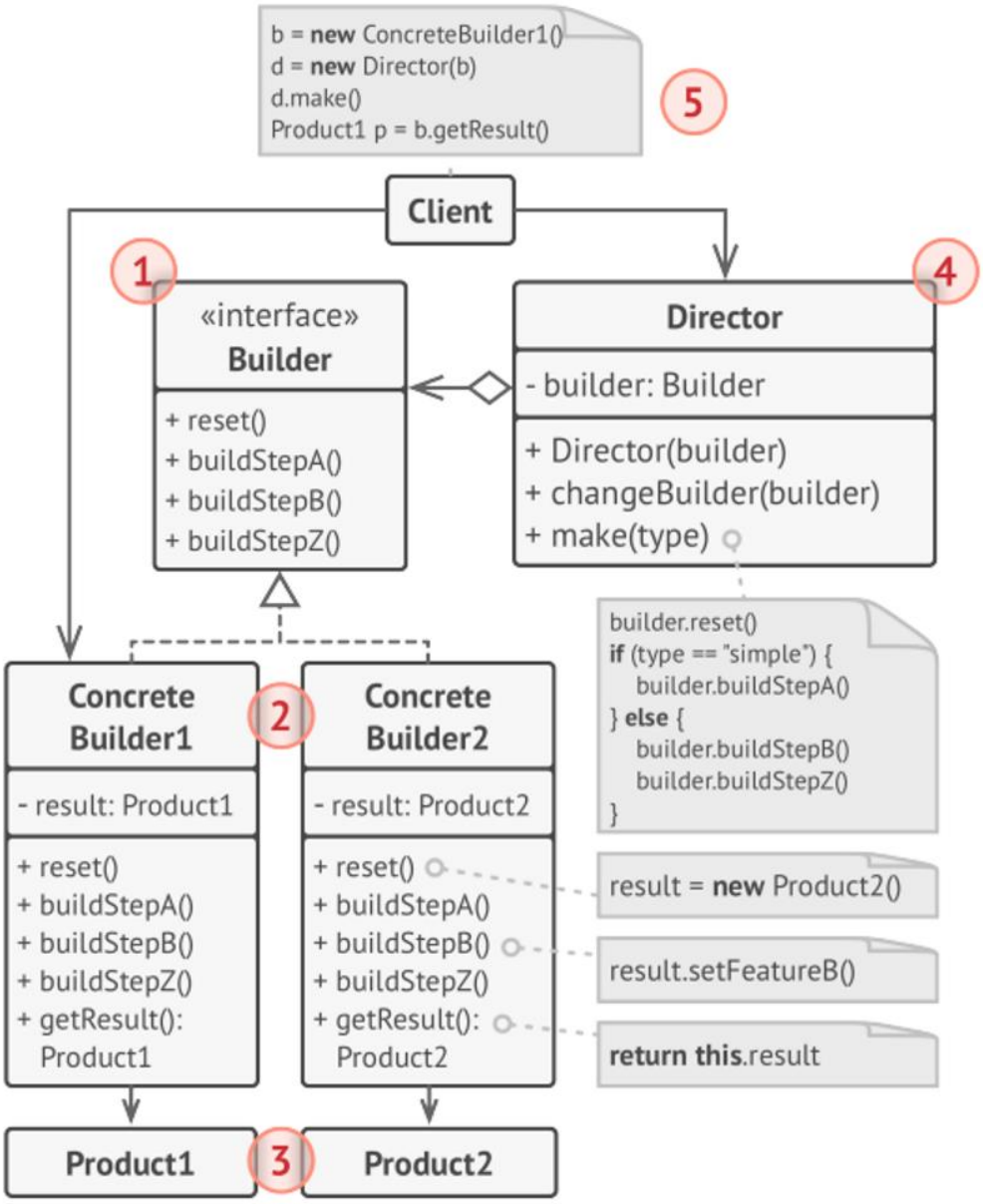
Imagine a car manufacturing plant where cars are customized based on customer specifications (e.g., different engines, interiors, colors). Using the Builder pattern, each car can be built step-by-step according to these specifications without needing multiple constructors for each possible combination.

5. Solution to the Problem

The Builder pattern separates the construction of a complex object from its representation. It allows you to use the same construction process to create different representations. The construction logic is encapsulated in a separate builder object, guided by a director object that controls the construction process.

6. Steps to Structure and Implement

1. **Define a Builder Interface:** Specifies methods for creating different parts of a product.
2. **Create Concrete Builder Classes:** Implement the builder interface with specific implementations.
3. **Define a Director Class:** Manages the order of construction steps.
4. **Client Interaction:** The client uses the director to construct an object using a builder instance.
5. **Retrieve the Final Product:** Get the constructed product from the builder.



7.1 Code Without Builder Pattern (Encountering Issues)

In this example, we'll create the Vehicle objects directly without using the Builder pattern. This can lead to issues such as complex constructors and repetitive code.

```
namespace WithoutBuilderPattern
{
    public class Vehicle
    {
        public string Type { get; set; }
        public int Wheels { get; set; }
        public string Engine { get; set; }
        public string Color { get; set; }

        public override string ToString()
        {
            return $"{Type} with {Wheels} wheels, Engine: {Engine}, Color: {Color}";
        }
    }

    public class Program
    {
        public static void Main()
        {
            // Creating vehicles directly
            var car = new Vehicle
            {
                Type = "Car",
                Wheels = 4,
                Engine = "V6",
                Color = "Red"
            };
            Console.WriteLine(car);

            var motorcycle = new Vehicle
            {
                Type = "Motorcycle",
                Wheels = 2,
                Engine = "V2",
                Color = "Black"
            };
            Console.WriteLine(motorcycle);

            var scooter = new Vehicle
            {
                Type = "Scooter",
                Wheels = 2,
                Engine = "Electric",
                Color = "White"
            };
            Console.WriteLine(scooter);
        }
    }
}
```

Explanation:

- **Complex Constructors:** If Vehicle had more properties or different configurations, constructors would become overly complex.
- **Repetitive Code:** Each time a new Vehicle is created, the same properties need to be set manually, leading to repetitive code.

7.2 Code With Builder Pattern

Now, let's implement the same functionality using the Builder pattern.

Step 1: Define a Vehicle Class

```
public class Vehicle
{
    public string Type { get; set; }
    public int Wheels { get; set; }
    public string Engine { get; set; }
    public string Color { get; set; }

    public override string ToString()
    {
        return $"{Type} with {Wheels} wheels, Engine: {Engine}, Color: {Color}";
    }
}
```

Step 2: Define a Builder Interface

```
public interface IVehicleBuilder
{
    void SetType();
    void SetWheels();
    void SetEngine();
    void SetColor();
    Vehicle Build();
}
```

Step 3: Create Concrete Builder Classes

- ❖ ***CarBuilder:***

```
public class CarBuilder : IVehicleBuilder
{
    private readonly Vehicle _vehicle = new();

    public void SetType() => _vehicle.Type = "Car";
    public void SetWheels() => _vehicle.Wheels = 4;
    public void SetEngine() => _vehicle.Engine = "V6";
    public void SetColor() => _vehicle.Color = "Red";

    public Vehicle Build() => _vehicle;
}
```

- ***MotorcycleBuilder:***

```
public class MotorcycleBuilder : IVehicleBuilder
{
    private readonly Vehicle _vehicle = new();

    public void SetType() => _vehicle.Type = "Motorcycle";
    public void SetWheels() => _vehicle.Wheels = 2;
    public void SetEngine() => _vehicle.Engine = "V2";
    public void SetColor() => _vehicle.Color = "Black";

    public Vehicle Build() => _vehicle;
}
```

- ***ScooterBuilder:***

```
public class ScooterBuilder : IVehicleBuilder
{
    private readonly Vehicle _vehicle = new();

    public void SetType() => _vehicle.Type = "Scooter";
    public void SetWheels() => _vehicle.Wheels = 2;
    public void SetEngine() => _vehicle.Engine = "Electric";
    public void SetColor() => _vehicle.Color = "White";

    public Vehicle Build() => _vehicle;
}
```

Step 4: Define a Director Class

```
public class VehicleDirector
{
    private readonly IVehicleBuilder _builder;

    public VehicleDirector(IVehicleBuilder builder)
    {
        _builder = builder;
    }

    public Vehicle Construct()
    {
        _builder.SetType();
        _builder.SetWheels();
        _builder.SetEngine();
        _builder.SetColor();
        return _builder.Build();
    }
}
```

Step 5: Client Code

```
public class Program
{
    public static void Main()
    {
        var carDirector = new VehicleDirector(new CarBuilder());
        Vehicle car = carDirector.Construct();
        Console.WriteLine(car);

        var motorcycleDirector = new VehicleDirector(new MotorcycleBuilder());
        Vehicle motorcycle = motorcycleDirector.Construct();
        Console.WriteLine(motorcycle);

        var scooterDirector = new VehicleDirector(new ScooterBuilder());
        Vehicle scooter = scooterDirector.Construct();
        Console.WriteLine(scooter);
    }
}
```

Explanation:

- **Decoupling:** The construction of a Vehicle is decoupled from its representation.
- **Flexibility:** It is easy to create different types of vehicles using different builders.
- **Maintainability:** The code is easier to maintain and extend.

Summary

By using the Builder pattern, you can create complex objects in a more controlled and flexible manner. It separates the construction process from the object representation, providing better readability, maintainability, and scalability of the code.