PROTOTYPE DESIGN PATTERN

CREATIONAL PATTERN

01 What is it?

02 Where and Why Do We Use It?

03 Real-World Scenario

04 Solution to the Problem

05 Steps to Structure and Implement

06 Code Without Pattern

07 Code With Pattern

```
public interface IPrototype
{
    IPrototype Clone();
    string GetDetails();
}
```

Karwan Essmat othman

Prototype Design Pattern - Creational Pattern

1. What is a Prototype Design Pattern?

The Prototype pattern is a way to create new objects by copying existing ones. Instead of making new objects from scratch, you make a copy of an existing object.

2. Where and Why Do We Use It?

- Where to use:
    1. When creating new objects is complicated or expensive.
    2. When you need new objects quickly and easily.

- Why use it:
    1. It saves time and effort by copying existing objects.
    2. It reduces the need for many different subclasses.
    3. It makes object creation more flexible.

3. Real-World Scenario

Analogy: The Cookie Cutter

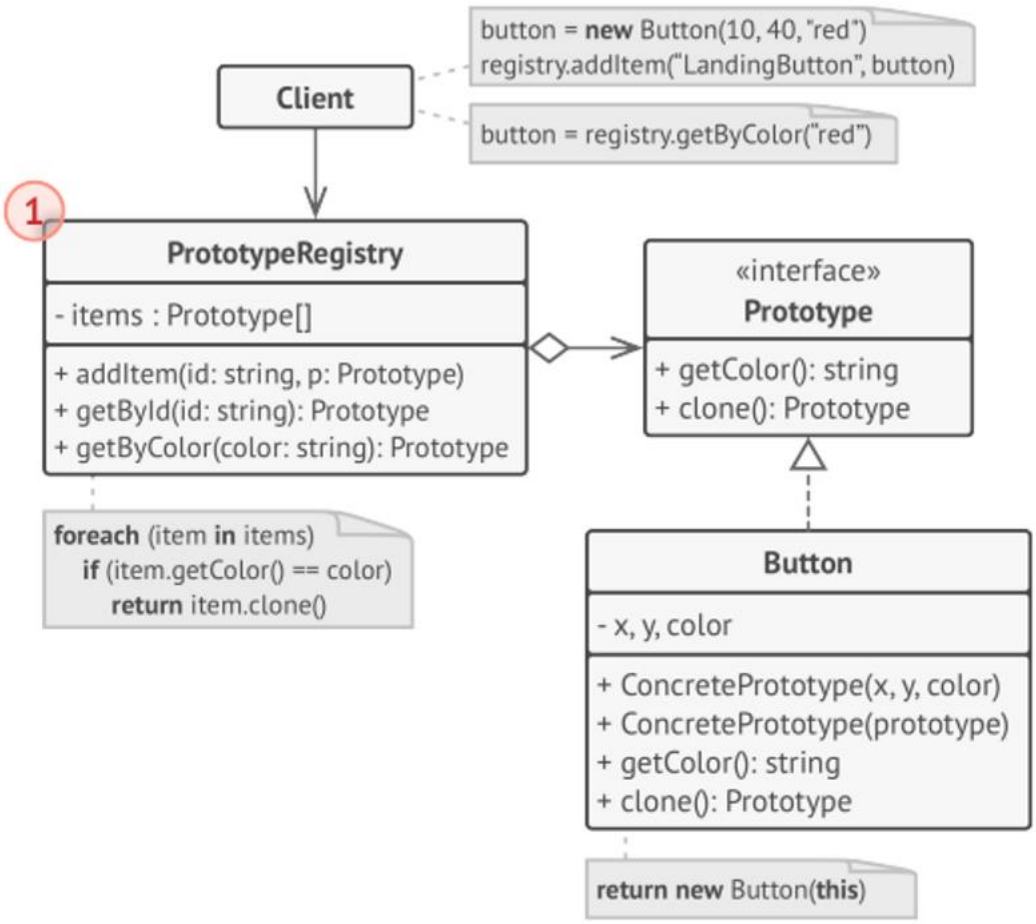
Think of making cookies with a cookie cutter. Instead of shaping each cookie by hand, you use a cookie cutter to make identical cookies quickly. Similarly, in programming, you clone objects to make new ones efficiently.

4. Solution to the Problem

The Prototype pattern solves the problem of creating complicated objects by allowing you to clone existing objects. This makes it easier and faster to create new objects.

5. Steps to Structure and Implement

1. **Define the Prototype Interface:** Create an interface that includes a method for cloning objects.
2. **Create Concrete Prototypes:** Implement the interface in classes whose objects need to be cloned.
3. **Use the Prototype:** During runtime, the client can create new objects by cloning the prototype.
4. **Manage Prototypes:** Optionally, maintain a prototype registry to manage different prototypes, making them easily accessible by the rest of the application.



6. Code Without Builder Pattern (Encountering Issues)

Without the Prototype pattern, creating objects with complex configurations might involve extensive subclassing or manual copying of each property.

```
01  public class Rectangle
02  {
03      public int Width { get; set; }
04      public int Height { get; set; }
05      public string Color { get; set; }
06
07      public Rectangle(int width, int height, string color)
08      {
09          Width = width;
10          Height = height;
11          Color = color;
12      }
13  }
14
15  public class Circle
16  {
17      public int Radius { get; set; }
18      public string Color { get; set; }
```

```
19
20      public Circle(int radius, string color)
21      {
22          Radius = radius;
23          Color = color;
24      }
25  }
26
27  // Creating objects manually
28  Rectangle rect1 = new Rectangle(10, 20, "Red");
29  Rectangle rect2 = new Rectangle(10, 20, "Blue");
30  Circle circle1 = new Circle(15, "Green");
```

Explanation:

This approach involves manually creating objects and setting their properties, which can be tedious for complex objects.

7. Code With Builder Pattern

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

Step 1: ***Prototype Interface***

```
1 public interface IPrototype

2 {
3     IPrototype Clone();
4     string GetDetails();
5 }
```

Step 2: ***Concrete Prototypes***

```
01 public class Rectangle : IPrototype
02 {
03     public int Width { get; set; }
04     public int Height { get; set; }
05     public string Color { get; set; }
06
07     public Rectangle(int width, int height, string color)
08     {
09         Width = width;
10         Height = height;
11         Color = color;
12     }
13
14     public IPrototype Clone()
15     {
16         return new Rectangle(Width, Height, Color);
17     }
18
```

```
    public string GetDetails()
    {
        return $"Rectangle [Width: {Width}, Height: {Height}, Color: {Color}]";
    }
}

public class Circle : IPrototype
{
    public int Radius { get; set; }
    public string Color { get; set; }

    public Circle(int radius, string color)
    {
        Radius = radius;
        Color = color;
    }

    public IPrototype Clone()
    {
        return new Circle(Radius, Color);
    }

    public string GetDetails()
    {
        return $"Circle [Radius: {Radius}, Color: {Color}]";
    }
}
```

Step 3: *Client Code*

```
class Program
{
    static void Main(string[] args)
    {
        // Creating prototypes
        IPrototype rectanglePrototype = new Rectangle(10, 20, "Red");
        IPrototype circlePrototype = new Circle(15, "Green");

        // Cloning prototypes
        IPrototype clonedRectangle = rectanglePrototype.Clone();
        IPrototype clonedCircle = circlePrototype.Clone();

        // Displaying details
        Console.WriteLine(clonedRectangle.GetDetails());
        Console.WriteLine(clonedCircle.GetDetails());
    }
}
```

Step 4: *Prototype Registry (Optional)*

A prototype registry can manage frequently used prototypes, providing a convenient way to clone objects.

```
01 public class PrototypeRegistry
02 {
03     private Dictionary<string, IPrototype> _prototypes = new Dictionary<string, IPrototype>();
04
05     public void AddPrototype(string key, IPrototype prototype)
06     {
07         _prototypes[key] = prototype;
08     }
09
10     public IPrototype GetPrototype(string key)
11     {
12         return _prototypes[key].Clone();
13     }
14 }
```

Summary

The Prototype design pattern is a powerful tool in scenarios where object creation is costly or complex, and where objects need to be instantiated dynamically. It leverages the cloning mechanism to make object creation more flexible and efficient, reducing the need for rigid class hierarchies and simplifying the code.