“GoF” Gang of Four, Design Patterns: Elements of Reusable Object-Oriented Software

In the context of C#, "**GoF**" refers to the "**Gang of Four**", which is a group of four authors who wrote the book "Design Patterns: Elements of Reusable Object-Oriented Software." The book, commonly referred to as the Gang of Four (GoF) book, was published in 1994 and is considered one of the seminal works on software design patterns.

Software design patterns are reusable solutions to common problems that arise during software development. They provide a way to organize code in a more maintainable and flexible manner, promoting good software design principles like encapsulation, modularity, and separation of concerns.

There are 23 design patterns in the GoF book, categorized into three groups:

Creational Patterns: These patterns deal with the process of object creation, providing more flexibility in creating instances of classes.

Structural Patterns: These patterns focus on how objects are composed to form larger structures, making it easier to design complex relationships between classes.

Behavioral Patterns: These patterns define communication patterns between objects, helping them work together effectively.

The **Gang of Four (GoF)** design patterns are a collection of **23 design patterns** that were introduced by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides in their book "Design Patterns: Elements of Reusable Object-Oriented Software," published in 1994. These patterns provide solutions to common software design problems and help in creating flexible, maintainable, and scalable software systems. The patterns are categorized into three groups: **Creational**, **Structural**, and **Behavioral** patterns. Here's a brief overview of each pattern:

Creational Patterns:

1. Singleton Pattern: Ensures that a class has only one instance and provides a global point of access to it.
2. Factory Method Pattern: Defines an interface for creating objects but allows subclasses to decide which class to instantiate.
3. Abstract Factory Pattern: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
4. Prototype Pattern: Creates new objects by copying existing ones, thus avoiding the need for complex initialization.
5. Builder Pattern: Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

Structural Patterns:

6. Adapter Pattern: Converts the interface of one class into another interface that clients expect.
7. Bridge Pattern: Decouples an abstraction from its implementation, allowing them to vary independently.
8. Composite Pattern: Treats individual objects and compositions of objects uniformly, forming a tree-like structure.
9. Decorator Pattern: Adds additional behaviors or responsibilities to objects dynamically, without affecting other objects.
19. Facade Pattern: Provides a simplified interface to a complex subsystem, making it easier to use.
11. Flyweight Pattern: Shares instances of lightweight objects to reduce memory usage for large numbers of objects.
12. Proxy Pattern: Represents another object and controls access to it, acting as a substitute or placeholder.

Behavioral Patterns:

13. Chain of Responsibility Pattern: Allows multiple objects to handle a request without specifying the receiver explicitly.
14. Command Pattern: Encapsulates a request as an object, allowing parameterization of clients with different requests, queuing of requests, and logging of requests.
15. Interpreter Pattern: Provides a way to evaluate language grammar or expressions.
16. Iterator Pattern: Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
17. Mediator Pattern: Defines an object that centralizes communication between other objects, promoting loose coupling.
18. Memento Pattern: Captures and restores an object's internal state, allowing undo/redo functionality.
19. Observer Pattern: Establishes a one-to-many dependency between objects, so that when one object changes state, its dependents are notified automatically.
20. State Pattern: Allows an object to change its behavior when its internal state changes.
21. Strategy Pattern: Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
22. Template Method Pattern: Defines the structure of an algorithm, allowing subclasses to override certain steps.
23. Visitor Pattern: Separates an algorithm from the objects on which it operates, allowing new operations to be added without modifying the objects.

Each of these design patterns addresses specific design challenges, and understanding and applying them appropriately can lead to more maintainable, modular, and extensible software systems.

Creational Patterns

1. Singleton Pattern

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. This is useful when you want to control the number of instances of a class and ensure that all parts of the application access the same instance.

C# Sample Application:

```
using System;

public class Singleton
{
   private static Singleton instance;

   // Private constructor to prevent external instantiation
   private Singleton() { }

   public static Singleton GetInstance()
   {
      if (instance == null)
      {
         instance = new Singleton();
      }
      return instance;
   }

   public string Function() // Renamed the method to start with a capital letter
   {
      return "Hola";
   }

   // Add other methods and properties here
}
```

```
class Program
{
   static void Main()
   {
      Singleton singleton = Singleton.GetInstance();
      Console.WriteLine(singleton.Function());
   }
}
```

2. Factory Method Pattern

The Factory Method pattern defines an interface for creating objects, but it allows subclasses to decide which class to instantiate. It promotes loose coupling by separating the client code from the object creation process.

C# Sample Application:

```
using System;

public class Program
{
   public static void Main()
   {
      // Create ConcreteCreatorA and ConcreteCreatorB instances
      Creator creatorA = new ConcreteCreatorA();
      Creator creatorB = new ConcreteCreatorB();

      // Use the FactoryMethod to create products
      IProduct productA = creatorA.FactoryMethod();
      IProduct productB = creatorB.FactoryMethod();

      // Display the products
      productA.Display();
      productB.Display();
   }
}
```

```
public interface IProduct
{
   void Display();
}

public class ConcreteProductA : IProduct
{
   public void Display()
   {
      Console.WriteLine("Concrete Product A");
   }
}

public class ConcreteProductB : IProduct
{
   public void Display()
   {
      Console.WriteLine("Concrete Product B");
   }
}

public abstract class Creator
{
   public abstract IProduct FactoryMethod();
}

public class ConcreteCreatorA : Creator
{
   public override IProduct FactoryMethod()
   {
      return new ConcreteProductA();
   }
}

public class ConcreteCreatorB : Creator
{
   public override IProduct FactoryMethod()
```

```
    {
        return new ConcreteProductB();
    }
}
```

3. Abstract Factory Pattern

Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

The Abstract Factory Pattern is a creational design pattern that provides an interface for creating families of related objects without specifying their concrete classes directly. It allows you to create objects that follow a specific theme or variation. Let's create a simple C# sample for the Abstract Factory Pattern:

In this example, we'll create a simple UI component factory that can generate buttons and checkboxes for both Windows and macOS operating systems.

```
using System;

// Abstract Product: Button
interface IButton
{
    void Render();
}

// Concrete Product: Windows Button
class WindowsButton : IButton
{
    public void Render()
    {
        Console.WriteLine("Rendering a Windows button.");
    }
}

// Concrete Product: macOS Button
class MacOSButton : IButton
```

```
{
    public void Render()
    {
        Console.WriteLine("Rendering a macOS button.");
    }
}

// Abstract Product: Checkbox
interface ICheckbox
{
    void Render();
}

// Concrete Product: Windows Checkbox
class WindowsCheckbox : ICheckbox
{
    public void Render()
    {
        Console.WriteLine("Rendering a Windows checkbox.");
    }
}

// Concrete Product: macOS Checkbox
class MacOSCheckbox : ICheckbox
{
    public void Render()
    {
        Console.WriteLine("Rendering a macOS checkbox.");
    }
}

// Abstract Factory
interface IUIComponentFactory
{
    IButton CreateButton();
    ICheckbox CreateCheckbox();
}
```

```
// Concrete Factory for Windows
class WindowsUIComponentFactory : IUIComponentFactory
{
    public IButton CreateButton()
    {
        return new WindowsButton();
    }

    public ICheckbox CreateCheckbox()
    {
        return new WindowsCheckbox();
    }
}

// Concrete Factory for macOS
class MacOSUIComponentFactory : IUIComponentFactory
{
    public IButton CreateButton()
    {
        return new MacOSButton();
    }

    public ICheckbox CreateCheckbox()
    {
        return new MacOSCheckbox();
    }
}

// Client Code
class Program
{
    static void Main(string[] args)
    {
        // Depending on the operating system, create the appropriate factory
        IUIComponentFactory uiFactory;
        if (IsWindows())
```

```
        {
            uiFactory = new WindowsUIComponentFactory();
        }
        else
        {
            uiFactory = new MacOSUIComponentFactory();
        }

        // Create UI components using the factory
        IButton button = uiFactory.CreateButton();
        ICheckbox checkbox = uiFactory.CreateCheckbox();

        // Render the UI components
        button.Render();
        checkbox.Render();
    }

    // A simple method to determine the operating system (for demonstration purposes)
    static bool IsWindows()
    {
        return Environment.OSVersion.Platform == PlatformID.Win32NT;
    }
}
```

4. Prototype Pattern

Creates new objects by copying existing ones, thus avoiding the need for complex initialization.

The Prototype Pattern is a creational design pattern in software development that allows you to create new objects by copying existing ones, thus avoiding the need for complex initialization. The key idea is to clone an existing object and customize it as needed rather than creating a new object from scratch.

In C#, you can implement the Prototype Pattern by using the ICloneable interface, which defines a method called Clone(). This interface allows you to create a copy of an object.

Here's a simple example demonstrating the Prototype Pattern in C#:

```
using System;

class Program
{
    static void Main(string[] args)
    {
        // Create an original prototype
        var originalPrototype = new ConcretePrototype(1);

        // Shallow copy using ICloneable (MemberwiseClone)
        var shallowCopy = (ConcretePrototype)originalPrototype.Clone();

        // Deep copy using custom Clone method
        var deepCopy = originalPrototype.Clone();

        // Modify the cloned objects
        shallowCopy.Id = 2;
        ((ConcretePrototype)deepCopy).Id = 3;

        // Output the results
        Console.WriteLine($"Original Prototype ID: {originalPrototype.Id}");
```

```
        Console.WriteLine($"Shallow Copy ID: {shallowCopy.Id}");
        Console.WriteLine($"Deep Copy ID: {((ConcretePrototype)deepCopy).Id}");
    }
}

// The prototype interface
public interface IPrototype
{
    IPrototype Clone();
}

// A concrete prototype class
public class ConcretePrototype : IPrototype, ICloneable
{
    public int Id { get; set; }

    public ConcretePrototype(int id)
    {
        Id = id;
    }

    // Shallow copy implementation using ICloneable
    public object Clone()
    {
        return this.MemberwiseClone();
    }

    // Deep copy implementation using custom Clone method
    public IPrototype Clone()
    {
        return new ConcretePrototype(this.Id);
    }
}
```

In this example, we have the IPrototype interface defining the Clone() method, which is then implemented by the ConcretePrototype class. We demonstrate two types of cloning: shallow copy and deep copy.

Shallow copy is implemented using the ICloneable interface, which performs a shallow copy of the object using the MemberwiseClone() method. This means that the reference-type members are copied, but their values are still pointing to the same objects.

Deep copy is implemented using a custom Clone() method, which creates a new instance of ConcretePrototype and copies the value of the Id property. This results in a completely separate object with its own Id value.

When you run the program, you'll see the following output:

rust
Copy code
Original Prototype ID: 1
Shallow Copy ID: 2
Deep Copy ID: 3

This demonstrates how the Prototype Pattern allows you to create new objects by copying existing ones while avoiding complex initialization.

5. Builder Pattern

Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

The Builder Pattern is a creational design pattern that allows you to separate the construction of a complex object from its representation. This pattern is useful when you want to create an object with multiple parts or configurations, but you don't want to expose the details of the object's construction to the client code. It provides a step-by-step approach to building an object and allows different representations of the object to be created using the same construction process.

Here's a simple example of implementing the Builder Pattern in C#:

Let's say we want to create a Car object with the following properties:
Brand
Model
Color
Engine Capacity
Number of Doors

First, we'll define the Car class representing the complex object:

```
public class Car
{
    public string Brand { get; set; }
    public string Model { get; set; }
    public string Color { get; set; }
    public double EngineCapacity { get; set; }
    public int NumberOfDoors { get; set; }

    public override string ToString()
    {
        return $"Brand: {Brand}, Model: {Model}, Color: {Color}, Engine: {EngineCapacity}L, Doors: {NumberOfDoors}";
    }
}
```

Next, we'll create an interface called ICarBuilder to define the steps to build a car:

```
public interface ICarBuilder
{
   void SetBrand(string brand);
   void SetModel(string model);
   void SetColor(string color);
   void SetEngineCapacity(double engineCapacity);
   void SetNumberOfDoors(int numberOfDoors);
   Car Build();
}
```

Now, we'll implement the CarBuilder class that implements the ICarBuilder interface:

```
public class CarBuilder : ICarBuilder
{
   private Car car;

   public CarBuilder()
   {
      car = new Car();
   }

   public void SetBrand(string brand)
   {
      car.Brand = brand;
   }

   public void SetModel(string model)
   {
      car.Model = model;
   }

   public void SetColor(string color)
   {
      car.Color = color;
```

```
    }

    public void SetEngineCapacity(double engineCapacity)
    {
        car.EngineCapacity = engineCapacity;
    }

    public void SetNumberOfDoors(int numberOfDoors)
    {
        car.NumberOfDoors = numberOfDoors;
    }

    public Car Build()
    {
        return car;
    }
}
```

Now, we can use the CarBuilder to create different representations of Car objects:

```
class Program
{
    static void Main(string[] args)
    {
        ICarBuilder carBuilder = new CarBuilder();

        carBuilder.SetBrand("Toyota")
                  .SetModel("Camry")
                  .SetColor("Red")
                  .SetEngineCapacity(2.5)
                  .SetNumberOfDoors(4);

        Car car1 = carBuilder.Build();
        Console.WriteLine("Car 1: " + car1);

        // Let's create another representation of a car
        carBuilder.SetBrand("Honda")
```

```
            .SetModel("Civic")
            .SetColor("Blue")
            .SetEngineCapacity(1.8)
            .SetNumberOfDoors(2);

        Car car2 = carBuilder.Build();
        Console.WriteLine("Car 2: " + car2);
    }
}
```

In this example, we used the CarBuilder to construct two different Car objects with different properties. The construction process remains the same, but we can create different representations of the Car object with various configurations.