Patrón de Diseño

Un **patrón de diseño** es una solución general y reutilizable a problemas comunes que surgen durante el desarrollo de software. Actúa como una plantilla que se puede aplicar en diferentes contextos para resolver problemas específicos de diseño. Los patrones de diseño no son fragmentos de código, sino más bien descripciones o modelos que guían a los desarrolladores en la creación de software más eficiente, mantenible y escalable

Importancia de los Patrones de Diseño

- **Estandarización**: Proporcionan un vocabulario común que facilita la comunicación entre desarrolladores.
- **Reutilización**: Permiten a los desarrolladores evitar reinventar soluciones ya probadas.
- **Mejora en la Mantenibilidad**: Fomentan un código más limpio y organizado, lo que facilita su mantenimiento a largo plazo.

Tipos de Patrones de Diseño

Los patrones de diseño se clasifican en tres categorías principales:

1. **Patrones Creacionales**: Se centran en la creación de objetos y proporcionan mecanismos para crear instancias de clases.
    - Ejemplos: *Singleton, Factory Method, Abstract Factory.*
2. **Patrones Estructurales**: Se ocupan de cómo se componen las clases y objetos para formar estructuras más grandes.
    - Ejemplos: *Adapter, Composite, Decorator.*
3. **Patrones de Comportamiento**: Se centran en la comunicación entre objetos y cómo se distribuyen las responsabilidades.
    - Ejemplos: *Observer, Strategy, Command.*

Ejemplos de Patrones de Diseño en C#

1. Singleton

Los patrones creacionales se centran en la forma en que se crean los objetos. Estos patrones abstraen el proceso de instanciación, lo que permite que el sistema sea independiente de cómo los objetos son creados, compuestos y representados.

Capture anexo:

```
public class Singleton
{
    private static Singleton instance;

    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            if (instance == null)
            {
                instance = new Singleton();
            }
            return instance;
        }
    }
}
```

Código:

```
public class Singleton
{
   private static Singleton instance;

   private Singleton() { }

   public static Singleton Instance
   {
      get
      {
         if (instance == null)
         {
            instance = new Singleton();
```

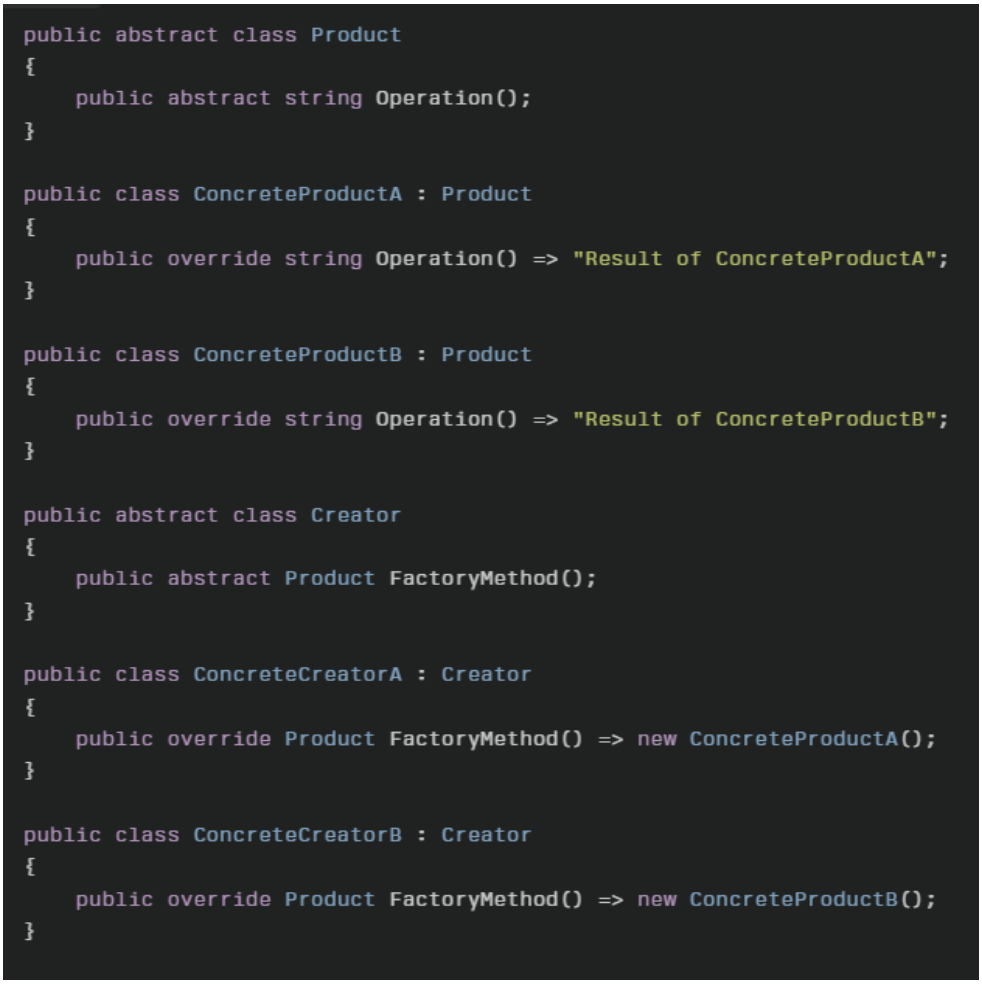
```
    }
        return instance;
      }
   }
}
```

Utilidad: Este patrón es útil cuando necesitas controlar el acceso a recursos compartidos, como conexiones a bases de datos o configuraciones globales.

2. Factory Method

Los patrones estructurales se ocupan de la composición de clases y objetos para formar estructuras más grandes y complejas. Estos patrones ayudan a asegurar que si una parte del sistema cambia, las demás partes no necesiten cambiar también.

Capture anexo:



```
public abstract class Product
{
    public abstract string Operation();
}

public class ConcreteProductA : Product
{
    public override string Operation() => "Result of ConcreteProductA";
}

public class ConcreteProductB : Product
{
    public override string Operation() => "Result of ConcreteProductB";
}

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

public class ConcreteCreatorA : Creator
{
    public override Product FactoryMethod() => new ConcreteProductA();
}

public class ConcreteCreatorB : Creator
{
    public override Product FactoryMethod() => new ConcreteProductB();
}
```

public abstract class Product

```
{
    public abstract string Operation();
}
public class ConcreteProductA : Product
{
    public override string Operation() => "Result of Concrete ProductA";
}
public class ConcreteProductB : Product
{
    public override string Operation() => "Result of Concrete ProductB";
}
public abstract class Creator
{
    public abstract Product Factory Method();
}
public class Concrete CreatorA : Creator
{
    public override Product FactoryMethod() => new ConcreteProductA();
}
public class ConcreteCreatorB : Creator
{
    public override Product FactoryMethod() => new ConcreteProductB();
}
```

Utilidad: Este patrón permite que el código sea más flexible y escalable al desacoplar la creación de objetos del uso real.

3. Observer

Define una dependencia uno a muchos entre objetos, de modo que cuando un objeto cambie su estado, todos sus dependientes sean notificados y actualizados automáticamente.

Capture anexo:

```
public interface IObserver
{
    void Update(string message);
}

public class ConcreteObserver : IObserver
{
    public void Update(string message)
    {
        Console.WriteLine($"Observer received message: {message}");
    }
}

public class Subject
{
    private List<IObserver> observers = new List<IObserver>();

    public void Attach(IObserver observer) => observers.Add(observer);

    public void Notify(string message)
    {
        foreach (var observer in observers)
        {
            observer.Update(message);
        }
    }
}
```

public interface IObserver
{
    void Update(string message);
}
public class ConcreteObserver : IObserver
{
    public void Update(string message)
    {
        Console.WriteLine($"Observer received message: {message}");
    }
}
public class Subject
{

```
    private List<IObserver> observers = new List<IObserver>();


    public void Attach(IObserver observer) => observers.Add(observer);


    public void Notify(string message)
    {
        foreach (var observer in observers)
        {
            observer.Update(message);
        }
    }
```

Utilidad: Este patrón es ideal para implementar sistemas donde múltiples componentes necesitan reaccionar a cambios en otros componentes, como en interfaces gráficas o sistemas de eventos.