Mediator es un patrón de diseño en el cual se define un objeto que encapsula cómo interactúan entre sí un conjunto de objetos y puede alterar el comportamiento del programa en ejecución, por lo cual se considera como un patrón de comportamiento.
La característica principal de este patrón es que los objetos no se comunican entre sí sino que se comunican a través del objeto mediator o mediador, lo cual reduce las dependencias entre los objetos que se comunican y por consecuencia la dependencia de código.

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main(string[] args)
    {
        ChatMediator chatMediator = new ChatMediator();

        Usuario usuario1 = new Usuario("Usuario1", chatMediator);
        Usuario usuario2 = new Usuario("Usuario2", chatMediator);

        chatMediator.AgregarUsuario(usuario1);
        chatMediator.AgregarUsuario(usuario2);

        usuario1.EnviarMensaje("¡Hola a todos!");
        usuario2.EnviarMensaje("¡Hola! ¿Cómo están?");
    }
}

class ChatMediator
{
    private List<Usuario> usuarios = new List<Usuario>();

    public void AgregarUsuario(Usuario usuario)
    {
        usuarios.Add(usuario);
    }

    public void EnviarMensaje(string mensaje, Usuario remitente)
    {
        foreach (var usuario in usuarios)
        {
            if (usuario != remitente)
            {
                usuario.RecibirMensaje(mensaje);
            }
        }
    }
}

class Usuario
{
```

```
    public string Nombre { get; private set; }
    private ChatMediator chatMediator;

    public Usuario(string nombre, ChatMediator mediator)
    {
        Nombre = nombre;
        chatMediator = mediator;
    }

    public void EnviarMensaje(string mensaje)
    {
        Console.WriteLine($"{Nombre} envía: {mensaje}");
        chatMediator.EnviarMensaje(mensaje, this);
    }

    public void RecibirMensaje(string mensaje)
    {
        Console.WriteLine($"{Nombre} recibe: {mensaje}");
    }
}
```

Chain of Responsibility

En este patrón de diseño el emisor de una petición a una cadena de objetos receptores que pueden optar por recibir la petición, o sea, manejar la solicitud entrante o pasarla al siguiente objeto de la cadena, lo que permite que varios objetos tengan la posibilidad de manejar una solicitud sin acoplar el emisor de la petición a su receptor. Cuando un objeto decide manejar la solicitud la cadena se detiene. Además, se pueden agregar nuevos objetos a la cadena o cambiar su orden sin que esto afecte al resto del sistema.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Handler manager = new Manager();
        Handler director = new Director();
        Handler ceo = new CEO();

        manager.SetNextHandler(director);
        director.SetNextHandler(ceo);

        manager.HandleRequest(1000);
        manager.HandleRequest(5000);
        manager.HandleRequest(15000);
    }
}

abstract class Handler
```

```
{
    protected Handler nextHandler;

    public void SetNextHandler(Handler handler)
    {
        nextHandler = handler;
    }

    public abstract void HandleRequest(int amount);
}

class Manager : Handler
{
    public override void HandleRequest(int amount)
    {
        if (amount <= 10000)
            Console.WriteLine("El Manager ha aprobado el gasto.");
        else if (nextHandler != null)
            nextHandler.HandleRequest(amount);
    }
}

class Director : Handler
{
    public override void HandleRequest(int amount)
    {
        if (amount <= 50000)
            Console.WriteLine("El Director ha aprobado el gasto.");
        else if (nextHandler != null)
            nextHandler.HandleRequest(amount);
    }
}

class CEO : Handler
{
    public override void HandleRequest(int amount)
    {
        Console.WriteLine("El CEO ha aprobado el gasto.");
    }
}
```

Strategy

El patrón strategy permite definir un conjunto de algoritmos llamados estrategias que se encapsularan cada uno en una clase, los objetos pueden cambiar su comportamiento intercambiando estos algoritmos estrategias durante la ejecución. Esto facilita la selección y aplicación de un algoritmo adecuado sin cambiar el código cliente.

Es posible y sencillo aplicar nuevas estrategias ya que cada una se implementa en su propia clase.

```
using System;
```

```
interface IEstrategiaPago
{
    void ProcesarPago(double monto);
}

class PagoTarjetaCredito : IEstrategiaPago
{
    public void ProcesarPago(double monto)
    {
        Console.WriteLine($"Se ha realizado un pago de {monto} con
tarjeta de crédito.");
    }
}

class PagoPayPal : IEstrategiaPago
{
    public void ProcesarPago(double monto)
    {
        Console.WriteLine($"Se ha realizado un pago de {monto} con
PayPal.");
    }
}

class ProcesadorPagos
{
    private IEstrategiaPago estrategia;

    public ProcesadorPagos(IEstrategiaPago estrategia)
    {
        this.estrategia = estrategia;
    }

    public void RealizarPago(double monto)
    {
        estrategia.ProcesarPago(monto);
    }
}

class Program
{
    static void Main(string[] args)
    {
        var estrategiaTarjeta = new PagoTarjetaCredito();
        var estrategiaPayPal = new PagoPayPal();
```

```
        var procesadorTarjeta = new ProcesadorPagos(estrategiaTarjeta);
        procesadorTarjeta.RealizarPago(100);

        var procesadorPayPal = new ProcesadorPagos(estrategiaPayPal);
        procesadorPayPal.RealizarPago(50);
    }
}
```

Template Method

Es un patrón que define la estructura de un algoritmo dentro de una clase base, pero permitiendo que las subclases implementen pasos específicos de ese algoritmo. Se encapsula la estructura del algoritmo y se delega la implementación de las diferentes partes a las subclases.

```
using System;

abstract class Receta
{
    public void Cocinar()
    {
        PrepararIngredientes();
        CocinarPlato();
        ServirPlato();
    }

    protected abstract void PrepararIngredientes();
    protected abstract void CocinarPlato();
    protected abstract void ServirPlato();
}

class RecetaPasta : Receta
{
    protected override void PrepararIngredientes()
    {
        Console.WriteLine("Preparar pasta, salsa y queso.");
    }

    protected override void CocinarPlato()
    {
        Console.WriteLine("Cocinar la pasta y mezclar con la salsa.");
    }

    protected override void ServirPlato()
```

```
    {
        Console.WriteLine("Servir la pasta en un plato.");
    }
}

class RecetaTorta : Receta
{
    protected override void PrepararIngredientes()
    {
        Console.WriteLine("Preparar harina, azúcar y huevos.");
    }

    protected override void CocinarPlato()
    {
        Console.WriteLine("Hornear la mezcla en el horno.");
    }

    protected override void ServirPlato()
    {
        Console.WriteLine("Servir la torta en un plato.");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Receta recetaPasta = new RecetaPasta();
        Receta recetaTorta = new RecetaTorta();

        Console.WriteLine("Cocinando receta de pasta:");
        recetaPasta.Cocinar();

        Console.WriteLine("\nCocinando receta de torta:");
        recetaTorta.Cocinar();
    }
}
```