Patrones de diseño

Comportamiento

Ya hemos aprendido sobre cómo inicializar objetos de manera efectiva y sobre cómo estructurar nuestros proyectos. Los patrones de diseño creacionales y estructurales como Factory, Adapter, Singleton, etc. son una brújula en la programación para evitar caer en errores comunes y tácticas obsoletas.

Los patrones de diseño **de comportamiento** son aquellos que vienen a suplir el cómo hacer que las entidades que creamos en código se comporten y actúen. No hablan de la "estructura" de tu programa (eso lo hacen los patrones estructurales), sino de la **forma en que las piezas se comunican, colaboran y reaccionan**. Lo que buscan es que el código sea **más flexible, claro y fácil de mantener**, evitando acoplamientos innecesarios y dando margen para cambiar comportamientos sin romper todo lo demás.

Los patrones de comportamiento definen **protocolos de interacción, distribuyen responsabilidades y facilitan la extensibilidad**.
Establecen reglas claras para que objetos distintos puedan trabajar en conjunto. Organizan quién hace qué, evitando que un objeto cargue con demasiadas tareas. Permiten agregar nuevas formas de comportamiento sin modificar el código ya probado.

En esta clase vamos a trabajar con tres patrones: **Strategy, Observer** y **Command.**

Strategy

Strategy es un patrón de comportamiento que **permite intercambiar algoritmos (o "formas de hacer algo") en tiempo de ejecución** sin tocar el resto del código.

En lugar de tener un `if/else` o `switch` gigante para decidir “cómo” hacer una tarea, **encapsulás cada variante en una clase** y el **contexto** delega el trabajo a la estrategia elegida.

Te conviene usarlo cuando tenés múltiples maneras de resolver la misma operación (cálculo de precio/envío, validación, serialización, compresión, políticas de bonificación). Cuando el criterio cambia seguido (marketing, reglas del negocio, A/B testing) o querés agregar nuevas variantes sin romper lo existente.

Escenario de ejemplo: un checkout que puede calcular envío por **moto**, **correo**, o **retiro en tienda**. Cambiamos la estrategia sin tocar el resto.

```
using System;


//------------------------------------------------

// 1) CONTRATO DE LA ESTRATEGIA

//------------------------------------------------

// Esta interfaz define "qué sabe hacer" cualquier
estrategia de envío.

// Todas deben implementar el método Calcular() y
exponer un Nombre para mostrar.

// De esta manera, Checkout no depende de cómo se
calcula, solo de que exista.

public interface IEnvioStrategy

{

    decimal Calcular(decimal subtotal); // calcula el
costo de envío según el subtotal
```

```
    string Nombre { get; }              //
nombre descriptivo de la estrategia

}


//-------------------------------------------------

// 2) ESTRATEGIAS CONCRETAS

//-------------------------------------------------

// Cada clase implementa su "forma de calcular el
envío".

// Todas cumplen con el mismo contrato
(IEnvioStrategy), pero la lógica interna cambia.


public class EnvioMoto : IEnvioStrategy

{

    public string Nombre => "Moto (rápido en la
ciudad)";

    // Envío fijo: siempre cuesta 1200

    public decimal Calcular(decimal subtotal) =>
1200m;

}


public class EnvioCorreo : IEnvioStrategy

{

    public string Nombre => "Correo (a todo el país)";
```

```
    // Envío gratis si el subtotal supera
cierto monto (ej: $50.000)

    // Si no, cuesta 3500

    public decimal Calcular(decimal subtotal) =>
subtotal >= 50000m ? 0m : 3500m;

}


public class RetiroEnTienda : IEnvioStrategy

{

    public string Nombre => "Retiro en tienda (sin
costo)";

    // Retirar en tienda siempre es gratis

    public decimal Calcular(decimal subtotal) => 0m;

}


//------------------------------------------------

// 3) CONTEXTO

//------------------------------------------------

// Esta clase representa un "Checkout" de compras.

// No calcula el envío por sí misma, sino que DELEGA
ese cálculo en una estrategia.

// Lo importante: si quiero cambiar la forma de
calcular el envío,

// no tengo que modificar Checkout, solo inyectar otra
estrategia.
```

```
public class Checkout

{

    private IEnvioStrategy _envio; // referencia a la
estrategia actual



    // Se inyecta una estrategia inicial por
constructor

    public Checkout(IEnvioStrategy envioInicial)

    {

        _envio = envioInicial;

    }



    // Permite cambiar la estrategia en tiempo de
ejecución

    public void SetEnvio(IEnvioStrategy nuevoEnvio)

    {

        _envio = nuevoEnvio;

        Console.WriteLine($"Estrategia de envío
cambiada a: {_envio.Nombre}");

    }



    // Calcula el total sumando el subtotal + el costo
de envío

    public decimal Total(decimal subtotal)
```

```
    {
        var costoEnvio = _envio.Calcular(subtotal); // delega en la estrategia
        return subtotal + costoEnvio;
    }
}

//-----------------------------------------------------
// 4) DEMO SIMPLE
//-----------------------------------------------------
// Acá simulamos un escenario de compra y mostramos cómo cambiar de estrategia en caliente sin reescribir nada en Checkout.
public static class Program
{
    public static void Main()
    {
        decimal subtotal = 42000m;

        // Caso 1: inicializamos con Envío en Moto
        var checkout = new Checkout(new EnvioMoto());
        Console.WriteLine($"Total con {nameof(EnvioMoto)}:   {checkout.Total(subtotal)}");
```

```
        // Caso 2: cambiamos la estrategia
a Correo

        checkout.SetEnvio(new EnvioCorreo());

        Console.WriteLine($"Total con
{nameof(EnvioCorreo)}: {checkout.Total(subtotal)}");



        // Caso 3: cambiamos a Retiro en Tienda

        checkout.SetEnvio(new RetiroEnTienda());

        Console.WriteLine($"Total con
{nameof(RetiroEnTienda)}:
{checkout.Total(subtotal)}");

    }

}
```

Ejercicio práctico: “Promo flexible del finde”

Sos dev de un e-commerce local. Marketing cambia las promos cada fin de semana. Necesitás un Carrito que aplique **distintas estrategias de descuento** sin tocar el Carrito.

Pasos:

1. **Contrato de estrategia**
   Creá IDescuentoStrategy { decimal Aplicar(decimal subtotal); string Nombre { get; } }.

2. **Estrategias concretas (mínimo 3):**

   - SinDescuento: devuelve el subtotal tal cual.
   - Porcentaje: aplica un % (ej.: 10%).

- TopeBanco: descuenta un monto fijo hasta un **tope** (ej.: hasta $5.000).

3. **Contexto Carrito:**

- Recibe la estrategia por constructor.
- Método Total(subtotal) que delega el cálculo a la estrategia.
- Método SetDescuento(IDescuentoStrategy s) para cambiar en runtime.

4. **Menú de consola (feedback inmediato):**

- Pedí subtotal.
- Mostrá opciones: 1) SinDescuento, 2) Porcentaje, 3) TopeBanco.
- Según elección, **instanciá** la estrategia y calculá el total.
- Permití **cambiar** de estrategia y recalcular sin salir del programa.

5. **Extras:**

- Factory simple que, dado un código ("WKND", "BANK"), devuelve la estrategia.

- Loguear qué estrategia se usó y cuánto se ahorró.

Observer

Supongamos que tenés un **Sujeto** (quien "tiene la data" o sufre un cambio) y un conjunto de **Observadores** (interesados) que quieren enterarse **automáticamente** cuando algo cambie.

El Sujeto **no conoce** los detalles de cada Observador: solo **avisa** (“atención, cambió X”). Cada Observador decide **qué hacer** cuando recibe el aviso.

¿Para qué sirve?

- Para **desacoplar**: el Sujeto no está atado a “quién escucha”.
- Para **reaccionar en cadena**: UI que se refresca, notificaciones, logs, métricas, etc.
- Para **evitar “polling”** (andar preguntando cada tanto). En lugar de: “¿ya cambió?”, el Sujeto te **avisa**.

```
using System;

//
============================================================
======================
// Un servicio de stock avisa cuando cambia la
cantidad de un SKU.
// Múltiples observadores reaccionan: Email, Dashboard
y Auditoría.
//
============================================================
======================

//
------------------------------------------------------------
--------------------
// 1) Datos del evento: qué info viaja con la
notificación
//    - Usamos EventArgs (patrón típico de .NET) para
encapsular el "payload".
//
------------------------------------------------------------
--------------------
public class StockChangedEventArgs : EventArgs
{
    public string Sku { get; }          //
```

```
Identificador del producto
    public int NuevaCantidad { get; }    // Cantidad
después del cambio

    public StockChangedEventArgs(string sku, int
nuevaCantidad)
        => (Sku, NuevaCantidad) = (sku,
nuevaCantidad);
}

//
------------------------------------------------------
--------------------
// 2) SUJETO (publicador): el que tiene el estado y
"dispara" el evento
//    - Expone un evento .NET (event EventHandler<T>)
que otros pueden suscribirse.
//    - Cuando cambia el stock, invoca el evento y
notifica a todos los observadores.
//
------------------------------------------------------
--------------------
public class StockService
{
    // Evento que notifica cambios de stock.
    // EventHandler<StockChangedEventArgs> define la
firma del callback de los observadores.
    public event EventHandler<StockChangedEventArgs>?
StockCambiado;

    // Simula una operación que cambia el stock y
dispara el evento
    public void Descontar(string sku, int
cantidadADescontar)
    {
        // En una app real, acá consultás BD, restás
stock y persistís.
        // Para demo, imaginemos que quedó en 3 (valor
fijo ilustrativo).
        int nuevaCantidad = 3;

        Console.WriteLine($"[STOCK] Descontado
{cantidadADescontar} unidad(es) de {sku}.
```

```
NuevaCantidad={nuevaCantidad}");

        // Si hay suscriptores, se les avisa. El "?"
evita NullReference si no hay ninguno.
        StockCambiado?.Invoke(
            this,
// sender: quien dispara (el sujeto)
            new StockChangedEventArgs(sku,
nuevaCantidad) // args con los datos del cambio
        );
    }
}

//
------------------------------------------------------
--------------------
// 3) OBSERVADORES (suscriptores): reaccionan cuando
se dispara el evento
//    - Cada uno hace su laburo sin que el sujeto los
conozca.
//
------------------------------------------------------
--------------------

// Observador A: envía un "email" (acá, simulamos con
Console)
public class NotificadorEmail
{
    // Método que "engancha" la función manejadora al
evento del sujeto
    public void Suscribir(StockService s) =>
s.StockCambiado += OnStockCambiado;
    public void Desuscribir(StockService s) =>
s.StockCambiado -= OnStockCambiado;

    // Este método se ejecuta cuando el sujeto dispara
el evento
    private void OnStockCambiado(object? sender,
StockChangedEventArgs e)
    {
        // Reacción concreta del observador
        Console.WriteLine($"[EMAIL] Aviso: {e.Sku}
tiene {e.NuevaCantidad} unidades.");
```

```
        if (e.NuevaCantidad < 5)
        {
            Console.WriteLine("[EMAIL] Sugerencia:
activar reposición urgente.");
        }
    }
}

// Observador B: actualiza un "dashboard" (acá,
simulamos con Console)
public class PanelDashboard
{
    public void Suscribir(StockService s) =>
s.StockCambiado += OnStockCambiado;
    public void Desuscribir(StockService s) =>
s.StockCambiado -= OnStockCambiado;

    private void OnStockCambiado(object? sender,
StockChangedEventArgs e)
    {
        // Semáforo simple según umbrales
        string color = e.NuevaCantidad >= 10 ?
"VERDE" : e.NuevaCantidad >= 5 ? "AMARILLO" : "ROJO";
        Console.WriteLine($"[DASH] SKU {e.Sku} ->
Semáforo {color} (cant: {e.NuevaCantidad})");
    }
}

// Observador C: auditoría/log (guarda historial, acá
solo imprime)
public class AuditorLog
{
    public void Suscribir(StockService s) =>
s.StockCambiado += OnStockCambiado;
    public void Desuscribir(StockService s) =>
s.StockCambiado -= OnStockCambiado;

    private void OnStockCambiado(object? sender,
StockChangedEventArgs e)
    {
        // En un sistema real: grabar en archivo/BD
con timestamp
        Console.WriteLine($"[AUDIT]
```

```
{DateTime.Now:HH:mm:ss} -> {e.Sku} =
{e.NuevaCantidad}");
    }
}

//
------------------------------------------------------------
--------------------
// 4) DEMO: armamos todo, suscribimos, disparamos
eventos y mostramos desuscripción
//
------------------------------------------------------------
--------------------
public static class Program
{
    public static void Main()
    {
        var stock = new StockService();

        var email = new NotificadorEmail();
        var dash  = new PanelDashboard();
        var audit = new AuditorLog();

        // SUSCRIPCIÓN: los tres observadores se
enganchan al evento del sujeto
        email.Suscribir(stock);
        dash.Suscribir(stock);
        audit.Suscribir(stock);

        Console.WriteLine("\n--- Primera operación:
todos suscriptos ---");
        stock.Descontar("SKU-ABC123", 1);

        // DESUSCRIPCIÓN: por ejemplo, el dash se
cierra (evitamos fugas de eventos)
        dash.Desuscribir(stock);

        Console.WriteLine("\n--- Segunda operación:
Dashboard desuscripto ---");
        stock.Descontar("SKU-ABC123", 2);

        // Nota: si más tarde quisiera volver a
escuchar, puede suscribirse otra vez.
        // dash.Suscribir(stock);
```

```
        }
    }
```

Ejercicio practico: Seguimiento del estado de un pedido con observadores

En un e-commerce, cada **Pedido** cambia de estado: Recibido → Preparando → Enviado → Entregado. Distintos equipos quieren enterarse **al instante**:

- **Cliente**: recibir un aviso (simular con Console.WriteLine).
- **Logística**: actualizar un tablero con el estado.
- **Auditoría**: registrar cada transición.

Pasos guiados

1. **Enum de estados**
   Creá enum EstadoPedido { Recibido, Preparando, Enviado, Entregado }.

2. **EventArgs del evento**
   PedidoChangedEventArgs : EventArgs con: int PedidoId, EstadoPedido NuevoEstado, DateTime Cuando.

3. **Sujeto PedidoService**

   ○ Propiedad/índice de estados por PedidoId (puede ser un Dictionary<int, EstadoPedido> simple).

   ○ Evento: public event EventHandler<PedidoChangedEventArgs>? EstadoCambiado;

   ○ Método CambiarEstado(int pedidoId, EstadoPedido nuevo) que actualiza el diccionario y hace

EstadoCambiado?.Invoke(this, new PedidoChangedEventArgs(...)).

4. **Observadores (mínimo 3)**

○ NotificadorCliente → imprime: "Tu pedido X ahora está: Y".

○ PanelLogistica → imprime una línea estilo tablero: "Pedido X => Y (HH:mm:ss)".

○ AuditoriaPedidos → imprime o guarda un log con timestamp.

Cada uno debe tener Suscribir(PedidoService s) y Desuscribir(PedidoService s).

5. **Programa de consola**

○ Instanciá PedidoService y los observadores.

○ **Suscribí** a los tres.

Simulá un flujo:

Pedido 101: Recibido -> Preparando -> Enviado -> Entregado

○ Llamando a CambiarEstado(101, ...) entre cada transición.

○ Mostrá en consola cómo cada observador reacciona.

6. **Desuscripción**

○ Desuscribí PanelLogistica antes del último cambio y comprobá que ya no imprime.

7. **Extras**

○ Filtro: que un observador se suscriba solo a un PedidoId.

- Validación: impedir saltos inválidos (por ejemplo, de Recibido directo a Entregado).

- Buffer: guardar las últimas N transiciones en memoria.

Command

Cuando querés ejecutar una acción (por ejemplo “agregar texto”, “borrar”, “pagar”, “enviar mail”), en vez de llamar directo una función, **empaquetás esa acción dentro de un objeto**: un **Comando**. Ese objeto sabe **ejecutarse** y (si corresponde) **deshacerse**.

¿Para qué sirve?

- **Desacoplar** quién pide la acción (**Invocador**) de quién la hace (**Receptor**).
- **Historial/Undo/Redo**: como cada acción es un objeto con Execute() / Undo(), podés deshacer y rehacer.
- **Colas, job runners y reintentos**: guardar comandos y ejecutarlos después.
- **Macros**: agrupar varios comandos y ejecutarlos en secuencia.

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

//
============================================================
====================
// Un mini editor de texto con acciones "agregar
texto", "borrar último"
// y "reemplazar", con soporte para Undo/Redo.
//
============================================================
====================

//
------------------------------------------------------------
--------------------
```

```
// 1) CONTRATO DEL COMANDO
//    - Todo comando debe poder ejecutarse.
//    - Si la acción es reversible, también debe poder
deshacerse.
//
-------------------------------------------------------
--------------------
public interface ICommand
{
    void Execute();
    void Undo(); // Si alguna acción no se puede
deshacer, dejá un No-Op y documentalo.
}

//
-------------------------------------------------------
--------------------
// 2) RECEIVER (RECEPTOR): la entidad que hace el
trabajo "real".
//    - Este es el "Documento" que mutamos con los
comandos.
//
-------------------------------------------------------
--------------------
public class Documento
{
    public string Texto { get; private set; } = "";

    public void Append(string s) => Texto += s;

    public void RemoveLast(int n)
    {
        if (n <= 0) return;
        if (n >= Texto.Length) { Texto = ""; return; }
        Texto = Texto[..^n]; // recorta los últimos n
caracteres
    }

    public void Replace(string nuevo)
    {
        Texto = nuevo ?? "";
    }
}
```

```
//
------------------------------------------------------------
--------------------
// 3) COMANDOS CONCRETOS
//    - Cada uno guarda lo necesario para ejecutar y
poder deshacer.
//
------------------------------------------------------------
--------------------

// 3.a) Agregar texto
public class AppendCommand : ICommand
{
    private readonly Documento _doc;
    private readonly string _toAppend;

    public AppendCommand(Documento doc, string
toAppend)
    {
        _doc = doc;
        _toAppend = toAppend;
    }

    public void Execute()
    {
        _doc.Append(_toAppend);
    }

    public void Undo()
    {
        // Para deshacer, removemos lo que agregamos
        _doc.RemoveLast(_toAppend?.Length ?? 0);
    }
}

// 3.b) Borrar últimos N caracteres
public class RemoveLastCommand : ICommand
{
    private readonly Documento _doc;
    private readonly int _n;
    private string _backup = ""; // guardamos lo
borrado para poder deshacer
```

```
    public RemoveLastCommand(Documento
doc, int n)
    {
        _doc = doc;
        _n = n;
    }

    public void Execute()
    {
        // Guardamos un backup de lo que vamos a
borrar:
        if (_n > 0 && _n <= _doc.Texto.Length)
        {
            _backup = _doc.Texto[^_n..]; // últimos n
caracteres
        }
        else
        {
            _backup = _doc.Texto; // si borramos todo
o más de lo que hay
        }
        _doc.RemoveLast(_n);
    }

    public void Undo()
    {
        // Simplemente volver a agregar lo borrado
        _doc.Append(_backup);
    }
}

// 3.c) Reemplazar todo el texto
public class ReplaceCommand : ICommand
{
    private readonly Documento _doc;
    private readonly string _nuevo;
    private string _anterior = "";

    public ReplaceCommand(Documento doc, string nuevo)
    {
        _doc = doc;
        _nuevo = nuevo ?? "";
    }
```

```
    public void Execute()
    {
        _anterior = _doc.Texto; // guardamos para Undo
        _doc.Replace(_nuevo);
    }

    public void Undo()
    {
        _doc.Replace(_anterior);
    }
}

//
------------------------------------------------------------
--------------------
// 4) INVOCADOR (Invoker): maneja la ejecución y el
historial para Undo/Redo.
//    - Guarda pilas de comandos ejecutados y comandos
deshechos.
//
------------------------------------------------------------
--------------------
public class EditorInvoker
{
    private readonly Stack<ICommand> _undo = new();
    private readonly Stack<ICommand> _redo = new();

    public void Run(ICommand cmd)
    {
        cmd.Execute();
        _undo.Push(cmd);
        _redo.Clear(); // al ejecutar un comando nuevo,
el futuro (redo) se descarta
    }

    public void Undo()
    {
        if (_undo.Count == 0) return;
        var cmd = _undo.Pop();
        cmd.Undo();
        _redo.Push(cmd);
    }
```

```
    public void Redo()
    {
        if (_redo.Count == 0) return;
        var cmd = _redo.Pop();
        cmd.Execute();
        _undo.Push(cmd);
    }
}

//
------------------------------------------------------------
--------------------
// 5) DEMO: uso del patrón en un mini flujo
//
------------------------------------------------------------
--------------------
public static class Program
{
    public static void Main()
    {
        var doc = new Documento();
        var inv = new EditorInvoker();

        Console.WriteLine("=== DEMO Command: mini
editor con Undo/Redo ===");

        // Agregamos "Hola"
        inv.Run(new AppendCommand(doc, "Hola"));
        Console.WriteLine(doc.Texto); // Hola

        // Agregamos " mundo"
        inv.Run(new AppendCommand(doc, " mundo"));
        Console.WriteLine(doc.Texto); // Hola mundo

        // Borramos los últimos 6 caracteres (" mundo")
        inv.Run(new RemoveLastCommand(doc, 6));
        Console.WriteLine(doc.Texto); // Hola

        // Reemplazamos todo por "Chau!"
        inv.Run(new ReplaceCommand(doc, "Chau!"));
        Console.WriteLine(doc.Texto); // Chau!

        // Hacemos Undo (vuelve a "Hola")
        inv.Undo();
```

```
            Console.WriteLine(doc.Texto); //
Hola

            // Hacemos Redo (vuelve a "Chau!")
            inv.Redo();
            Console.WriteLine(doc.Texto); // Chau!

            Console.WriteLine("=== Fin de la demo ===");
        }
}
```

Ejercicio práctico: Carrito de Compras con Undo/Redo y Macro Comandos

Estás armando un carrito. El usuario puede **agregar producto**, **quitar producto** y **cambiar cantidad**. Además, querés que pueda **deshacer** la última acción, **rehacerla**, y ejecutar **promos** que son **varias acciones juntas** (macro).

Modelo:

- **Receiver**: Carrito, con una lista interna de Item { string Sku; string Nombre; decimal Precio; int Cantidad; }

  - Métodos: Agregar(Item), Quitar(string sku), CambiarCantidad(string sku, int nuevaCantidad), Total().

- **Invoker**: CarritoInvoker con pilas _undo/_redo (igual que el ejemplo).

- **Commands**:

  - AgregarItemCommand(Carrito, Item)

- QuitarItemCommand(Carrito, string sku) (guardar backup del item para Undo)

- CambiarCantidadCommand(Carrito, string sku, int nuevaCantidad) (guardar cantidad anterior)

- MacroCommand(List<ICommand>) (ejecuta varios y hace undo en orden inverso)

Pasos guiados

1. **Crear Carrito (Receiver)**

- Estructura interna: Dictionary<string, Item> para acceso rápido por Sku.

- Implementar Agregar, Quitar, CambiarCantidad, y Total.

2. **Definir ICommand** con Execute() y Undo().

3. **Implementar comandos concretos**

- AgregarItemCommand: en Execute() llama carrito.Agregar(item); en Undo() llama carrito.Quitar(item.Sku).

- QuitarItemCommand: en Execute() hace backup del Item existente y llama Quitar(sku); en Undo() re-agrega el backup.

- CambiarCantidadCommand: en Execute() guarda cantidadAnterior, aplica la nueva; en Undo() restaura la anterior.

4. **Invoker con historial**

- Igual al EditorInvoker del ejemplo: Run(cmd), Undo(), Redo().

5. **MacroCommand**

- Implementá un comando que reciba una **lista de comandos**.
- Execute(): ejecuta cada uno en orden.
- Undo(): deshace en **orden inverso**.

6. **Menú de consola**

- Opciones:
 1) Agregar item, 2) Quitar item, 3) Cambiar cantidad, 4) Undo, 5) Redo, 6) Total, 7) Promo (Macro).
- Para Promo (Macro): por ejemplo, agregar dos ítems y setear cantidad especial; todo en un solo "click".

7. **Pruebas manuales**

- Agregar un ítem, ver total.
- Cambiar cantidad, Undo, Redo.
- Ejecutar Promo (Macro), verificar que un Undo **revierte todas** las acciones de la promo.

8. **Extras**

- Validar que no haya cantidades negativas.
- Log de acciones con timestamp.
- Persistir el historial (serializar comandos ejecutados) y "replay".