Patrones de diseño

Estructurales

Como ya vimos dentro de los patrones de diseño existen tres categorías primarias. En la clase anterior pudimos aprender sobre los creacionales, por ende en la clase de hoy aprenderemos sobre los estructurales.

Este tipo de patrones son cruciales cuando queremos armar estructuras complejas dentro de nuestro código, es decir, cuando vamos a tener un sinfín de clases, interfaces y carpetas. Para que no sea un desastre organizacional aplicamos un patrón que simplifique el entendimiento de nuestro proyecto. Existen muchísimos tipos y aca te dejo un repaso rápido por los más populares:

1. **Adapter (Adaptador):** Permite que interfaces incompatibles trabajen juntas. Es ideal cuando debes integrar nuevos componentes que no son compatibles con las interfaces existentes.
2. **Bridge (Puente):** Separa la abstracción de su implementación, permitiendo que ambas evolucionen de forma independiente. Es útil cuando deseas evitar un enlace permanente entre una abstracción y su implementación.
3. **Composite (Compuesto):** Permite a los clientes tratar objetos individuales y composiciones de objetos de manera uniforme. Es útil cuando quieres representar jerarquías parte-todo de manera consistente.
4. **Decorator (Decorador):** Añade responsabilidades adicionales a un objeto dinámicamente. Es útil cuando necesitas extender las capacidades de objetos individuales sin alterar su estructura.

5. **Facade (Fachada):** Proporciona una interfaz unificada para un conjunto de interfaces en un subsistema. Es útil cuando deseas simplificar la interfaz de un grupo de clases o componentes.
6. **Flyweight (Peso Ligero):** Minimiza el uso de memoria o recursos compartiendo lo máximo posible con objetos similares. Es útil cuando manejas un gran número de objetos que comparten características comunes.
7. **Proxy (Proxy):** Controla el acceso a un objeto proporcionando un sustituto o representante. Es útil cuando deseas controlar el acceso a un objeto o agregar funcionalidades adicionales.

Sin embargo, no vamos a aprender sobre todos. Solo nos vamos a enfocar en tres tipos: **adapter, decorator y facade**.

Vamos a desarrollarlos:

1. **Adapter**: Tenés dos piezas que no encajan porque sus interfaces no coinciden. Adapter es el "enchufe universal" que traduce una interfaz a otra sin tocar el código original. *"Dejalo como está, yo traduzco"*
    a. ¿Cuando usar? Librerías viejas (o de terceros) que no podés modificar. Migraciones graduales: tu código nuevo habla "A" y el legacy habla "B". Querés **bajar acoplamiento**: el cliente solo conoce la interfaz objetivo, no sabe ni le importa cómo es la clase adaptada.

```
using System;

namespace Patrones.Estructurales.Adapter
{
    // 1) Interfaz OBJETIVO que el cliente espera
usar
```

```
    //    Nuestro sistema "moderno"
entiende IAudioPlayer con Play(string ruta)
    public interface IAudioPlayer
    {
        void Play(string ruta);
    }

    // 2) Clase EXISTENTE/LEGACY (no la podemos
tocar)
    //    Su interfaz es distinta: reproduce con otro
método y hasta otro nombre
    public class ReproductorMp3Legacy
    {
        public void ReproducirMp3(string archivo)
        {
            Console.WriteLine($"[LEGACY]
Reproduciendo MP3: {archivo}");
        }
    }

    // 3) ADAPTADOR: implementa la interfaz objetivo,
    //    pero por dentro usa la clase legacy y
traduce la llamada.
    public class Mp3ToAudioAdapter : IAudioPlayer
    {
        private readonly ReproductorMp3Legacy
_legacy;

        public
Mp3ToAudioAdapter(ReproductorMp3Legacy legacy)
        {
            _legacy = legacy;
        }

        // Traducción: el cliente llama Play(ruta),
        // y nosotros por dentro llamamos
ReproducirMp3(ruta).
```

```
        public void Play(string ruta)
        {
            // Acá podríamos validar formato,
normalizar ruta, etc.
            _legacy.ReproducirMp3(ruta);
        }
    }

    // 4) Cliente
    public static class Demo
    {
        public static void Run()
        {
            // El cliente SOLO conoce IAudioPlayer
            IAudioPlayer player = new
Mp3ToAudioAdapter(new ReproductorMp3Legacy());
            player.Play("temazo.mp3");

            // Si mañana cambiara el backend, el
cliente ni se entera:
            // solo reemplazamos el adapter o su
implementación interna.
        }
    }
}
```

2. **Decorator**: Querés agregar responsabilidades (comportamientos extra) a un objeto en tiempo de ejecución y sin heredar mil veces ni modificar la clase original. *"Agregá features como si fueran capas de cebolla, sin romper el núcleo."*
    a. ¿Cuando usar? Tenés una funcionalidad base y "toppings" opcionales (logging, cache, compresión, encriptado, métricas, validación…). Queres componer varios extras en cadena (stackeables). Evitar una jerarquía de herencia gigante por combinaciones de features.

```
using System;

namespace Patrones.Estructurales.Decorator
{
    // 1) Interfaz común (contrato)
    public interface INotificador
    {
        void Enviar(string mensaje);
    }

    // 2) Implementación base (la simple)
    public class NotificadorEmail : INotificador
    {
        public void Enviar(string mensaje)
        {
            // Acá iría la lógica real de envío por
email
            Console.WriteLine($"[EMAIL] Enviando:
{mensaje}");
        }
    }

    // 3) Clase abstracta Decorador: implementa la
interfaz y guarda una referencia
    //    al componente que envuelve (composición)
    public abstract class NotificadorDecorator :
INotificador
    {
        protected readonly INotificador _wrappee;

        protected NotificadorDecorator(INotificador
wrappee)
        {
            _wrappee = wrappee;
        }

        public virtual void Enviar(string mensaje)
```

```
        {
            _wrappee.Enviar(mensaje);
        }
    }

    // 4) Decorador concreto: agrega LOG
antes/después
    public class NotificadorConLog :
NotificadorDecorator
    {
        public NotificadorConLog(INotificador
wrappee) : base(wrappee) {}

        public override void Enviar(string mensaje)
        {
            Console.WriteLine("[LOG] Preparando
envío...");
            base.Enviar(mensaje);
            Console.WriteLine("[LOG] Envío OK.");
        }
    }

    // 5) Otro decorador concreto: agrega firma al
mensaje
    public class NotificadorConFirma :
NotificadorDecorator
    {
        private readonly string _firma;

        public NotificadorConFirma(INotificador
wrappee, string firma)
            : base(wrappee)
        {
            _firma = firma;
        }

        public override void Enviar(string mensaje)
```

```
        {
            // Sumamos la firma sin tocar
NotificadorEmail
            var conFirma = $"{mensaje}\n\n--
{_firma}";
            base.Enviar(conFirma);
        }
    }

    // 6) Cliente: apila decoradores a gusto
    public static class Demo
    {
        public static void Run()
        {
            // Base
            INotificador noti = new
NotificadorEmail();

            // Le agrego log
            noti = new NotificadorConLog(noti);

            // Le agrego firma (se apilan en runtime)
            noti = new NotificadorConFirma(noti,
"Equipo de Soporte");

            // Disparo
            noti.Enviar("Tu ticket fue recibido.
Gracias por escribirnos.");

            // Salida esperada:
            // [LOG] Preparando envío...
            // [EMAIL] Enviando: Tu ticket fue
recibido. Gracias por escribirnos.
            //
            // -- Equipo de Soporte
            // [LOG] Envío OK.
        }
```

```
    }
}
```

3. **Facade:** Tenés un sub-sistema complejo con muchas clases y pasos. Fachada te da una puerta de entrada simple para tareas comunes, escondiendo la complejidad. "*Una ventanilla única que te resuelve el trámite*"
    a. ¿Cuando usar? Módulos con varias APIs internas (pagos, inventario, envíos, validaciones…). Necesitás simplificar la vida del cliente: "hace todo esto con un solo método". Querés **reducir acoplamiento** entre el cliente y el sub-sistema.

```
using System;

namespace Patrones.Estructurales.Facade
{
    // 1) Subsistemas (simples para el ejemplo)
    public class InventarioService
    {
        public bool HayStock(string sku, int
cantidad)
        {
            Console.WriteLine($"[Inventario]
Verificando stock de {sku} x{cantidad}...");
            return true; // Simulación
        }

        public void DescontarStock(string sku, int
cantidad)
        {
            Console.WriteLine($"[Inventario]
Descontando stock de {sku} x{cantidad}...");
        }
    }
```

```
    public class PagoService
    {
        public bool Cobrar(string medioPago, decimal
monto)
        {
            Console.WriteLine($"[Pago] Cobrado
{monto:C} con {medioPago}.");
            return true; // Simulación
        }
    }

    public class EnvioService
    {
        public void GenerarGuia(string sku, int
cantidad, string direccion)
        {
            Console.WriteLine($"[Envío] Generando
guía a {direccion} para {sku} x{cantidad}...");
        }
    }

    public class NotificacionService
    {
        public void EnviarConfirmacion(string email,
string sku, int cantidad)
        {
            Console.WriteLine($"[Notif] Confirmación
enviada a {email} por {sku} x{cantidad}.");
        }
    }

    // 2) Fachada: ofrece un método simple que
orquesta todo el baile
    public class TiendaFacade
    {
        private readonly InventarioService
_inventario = new InventarioService();
```

```
        private readonly PagoService _pago
= new PagoService();
        private readonly EnvioService _envio = new
EnvioService();
        private readonly NotificacionService _noti =
new NotificacionService();

        // Caso de uso "Comprar"
        public bool Comprar(string sku, int
cantidad, decimal precioUnitario,
                            string medioPago, string
direccion, string emailCliente)
        {
            Console.WriteLine("[Fachada] Iniciando
compra...");

            // 1) Validaciones y stock
            if (!_inventario.HayStock(sku,
cantidad))
            {
                Console.WriteLine("[Fachada] Sin
stock suficiente.");
                return false;
            }

            // 2) Cobro
            var total = precioUnitario * cantidad;
            if (!_pago.Cobrar(medioPago, total))
            {
                Console.WriteLine("[Fachada] Error
de pago.");
                return false;
            }

            // 3) Descontar stock
            _inventario.DescontarStock(sku,
cantidad);
```

```
            // 4) Envío
            _envio.GenerarGuia(sku, cantidad,
direccion);

            // 5) Notificación
            _noti.EnviarConfirmacion(emailCliente,
sku, cantidad);

            Console.WriteLine("[Fachada] Compra
finalizada con éxito.");
            return true;
        }
    }

    // 3) Cliente: usa UNA sola puerta (la fachada) y
listo
    public static class Demo
    {
        public static void Run()
        {
            var tienda = new TiendaFacade();

            bool ok = tienda.Comprar(
                sku: "SKU-ABC-123",
                cantidad: 2,
                precioUnitario: 14999.99m,
                medioPago: "Tarjeta",
                direccion: "Av. Siempre Viva 742",
                emailCliente: "cliente@ejemplo.com"
            );

            Console.WriteLine($"Resultado: {(ok ?
"OK" : "FALLO")}");
        }
    }
}
```

Ahora, para poder aprender estos tres patrones bien como siempre debemos **practicar**. Para esto al igual que en la guia anterior, te dejo tres ejercicios narrados y con un paso a paso para que puedas armar tus propias soluciones aplicando los tres patrones vistos hoy.

- **Adapter:** Imaginá que estás trabajando en un sistema de impresión de tickets. Tu sistema nuevo espera que todas las impresoras tengan un método `Imprimir(string texto)`, pero te dieron una librería vieja de impresoras térmicas que usa `PrintTicket(string data)`. No podés modificar esa librería.

  Necesitás **crear un adaptador** que permita que el sistema trate a cualquier impresora vieja como si fuera moderna.

  Vamos con el paso a paso:

  1. **Definí la interfaz estándar del sistema nuevo:**
     - `IImpresora` con método `Imprimir(string texto)`.
  2. **Simulá la clase vieja:**
     - `ImpresoraTermicaVieja` con método `PrintTicket(string data)`.
  3. **Crea el Adaptador:**
     - `AdaptadorTermica` que **implemente `IImpresora`** pero internamente use `ImpresoraTermicaVieja`.
     - Su `Imprimir()` debe llamar a `PrintTicket()`.
  4. **Crea un cliente que use el adaptador:**

- Instanciá una `ImpresoraTermicaVieja`, envolvela con `AdaptadorTermica` y llamá a `Imprimir("Texto del ticket")`.

5. **Prueba:**
   - Asegurate que al llamar a `Imprimir()`, se ejecute el método viejo de `PrintTicket()`.

Extra: Podés crear otra impresora ficticia (`ImpresoraLaser`) que ya implemente `IImpresora`, para mostrar que el cliente no nota la diferencia.

- **Decorator:** Vas a armar un sistema de mensajería instantánea. Hay un objeto base `MensajeSimple` que envía texto plano. Queremos agregar filtros opcionales como decoradores:

  `MensajeEncriptado` para enviar texto en mayúsculas (simulando encriptación).

  `MensajeConEmoji` para agregar un emoji al final.Los filtros deben poder apilarse dinámicamente.

1. **Definí la interfaz `IMensaje`:**
   - Método `Enviar(string texto)`.
2. **Clase base:**
   - `MensajeSimple` que implemente `IMensaje`, simplemente imprime el texto.

3. **Crea un decorador abstracto:**
    - `MensajeDecorator` que implemente `IMensaje` y guarde un `IMensaje` interno.
4. **Crea dos decoradores concretos:**
    - `MensajeEncriptado` (convierte el texto a mayúsculas).
    - `MensajeConEmoji` (agrega "😊" al final del texto).
5. **Cliente:**
    - Instanciá un `MensajeSimple`.
    - Decoralo primero con `MensajeEncriptado`, después con `MensajeConEmoji`.
    - Llamá a `Enviar("Hola mundo")`.

Extra: Probar diferentes órdenes de decoradores para ver cómo cambia la salida.

- **Facade**: Vas a diseñar una **fachada para gestionar usuarios**. Actualmente, registrar un usuario implica:

  Guardar datos en base de datos (`DBService`).
  Validar email (`EmailValidator`).
  Enviar correo de bienvenida (`EmailSender`).

  Todo esto es molesto de coordinar en el cliente. Tu misión es **crear una fachada `UserFacade`** que tenga un método `RegistrarUsuario(nombre, email)` y que orqueste todo automáticamente.

1. **Crea las clases del subsistema:**
    - `DBService` con método `GuardarUsuario(nombre, email)`.
    - `EmailValidator` con método `EsValido(email)`.
    - `EmailSender` con método `EnviarBienvenida(email)`.
2. **Crea la fachada `UserFacade`:**
    - Método `RegistrarUsuario(nombre, email)`:
        1. Valida el email.
        2. Si es válido, guarda el usuario.
        3. Envía correo de bienvenida.
        4. Muestra mensajes en consola.
3. **Cliente:**
    - Desde `Main()`, crear `UserFacade` y llamar a `RegistrarUsuario()`.

Extra: Podés agregar más pasos (ej. generar un ID único o loguear) para que vean cómo la fachada centraliza el flujo.