Metodología de Sistemas II

Clase 5

Patrones de Diseño: Estructurales

Jeremías Fassi

2025

Patrones de Diseño: Estructurales

Introducción a los Patrones Estructurales

Jeremías Fassi

2025

Introducción a los patrones estructurales

- ¿Qué son los patrones estructurales y por qué son importantes?

- Son un grupo de patrones de diseño cuyo objetivo principal es **organizar clases y objetos** para formar estructuras más grandes, eficientes y flexibles.

- Son importantes porque permiten conectar, organizar y simplificar la relación entre clases y objetos, haciendo que los sistemas sean más flexibles, reutilizables y fáciles de mantener, sin necesidad de modificar el código existente.

Introducción a los patrones estructurales

- Características
    - Se centran en la **relación entre clases y objetos**, más que en cómo se crean (creacionales) o cómo se comunican (comportamiento).
    - Permiten adaptar interfaces incompatibles, componer comportamientos de forma dinámica y simplificar interacciones complejas.
    - Son especialmente útiles cuando trabajamos en sistemas grandes, con muchas dependencias, librerías externas o código *legacy*.

Introducción a los patrones estructurales

- Analogía

  Piensen a un sistema como un auto:

  → Los patrones estructurales son como los **adaptadores** de piezas, es decir, los accesorios que se agregan y el tablero de control que muestra solo lo necesario, ocultando toda la mecánica compleja que hay detrás.

Patrón Adapter

Jeremías Fassi

2025

Definición

El patrón **Adapter** permite que objetos con interfaces incompatibles, puedan colaborar. Convierte la interfaz de una clase en otra interfaz que el cliente espera. Pensarlo como un traductor.

- Motivación

  Muchas veces nos enfrentamos con código ya existente (código *legacy* o librerías externas) que no podemos modificar, pero que necesitamos integrarnos con nuevas clases.

  → Ejemplo cotidiano: un adaptador de enchufe que permite usar un cargador “de 2 patitas” en un tomacorriente “de 3 patitas”.

Estructura

- **Cliente**: el que espera trabajar con cierta interfaz.
- **Interfaz objetivo**: la interfaz esperada por el cliente.
- **Adaptado** (*Adaptee*): la clase existente que no es compatible.
- **Adapter**: traduce llamadas del cliente hacia el adaptado.

Adapter

- Ventajas
  - Reutilización de código existente.
  - Aislamiento entre el cliente y las clases que no cumplen la interfaz.
  - Facilita la integración de sistemas heterogéneos.

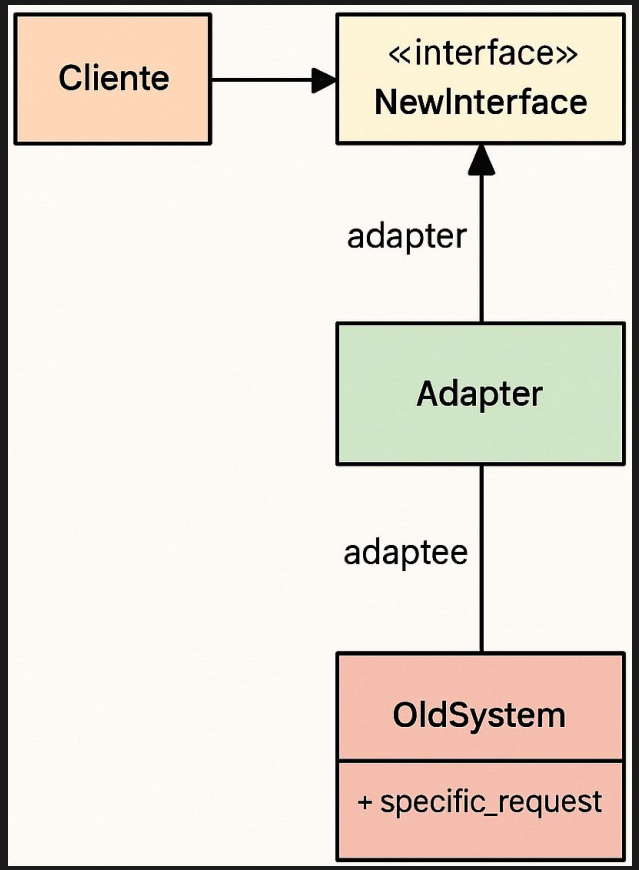
Adapter - Ejemplos

- Python
  - adapter.py
- TS
  - adapter.ts

Casos de uso real

- Una API antigua devuelve datos en XML, pero la app con la que estamos trabajando, utiliza JSON.
  Un Adapter convierte el XML a JSON antes de entregarlo al cliente.

Adapter - Diagrama



Patrón Decorator

Jeremías Fassi

2025

Definición

El patrón **Decorator** permite añadir responsabilidades adicionales o nuevos comportamientos, a un objeto en tiempo de ejecución, de manera transparente, es decir, sin modificar el código. La idea es colocar estos objetos dentro de otros objetos llamados *wrappers* que son los que efectivamente contienen los **comportamientos**.

- Motivación

A veces necesitamos agregar funcionalidades a una clase, pero:

- No queremos o podemos modificar el código original (puede ser código de terceros).
- No queremos aplicar herencia (porque puede limitar la flexibilidad).

Motivación

- Problema que resuelve

Imaginen una librería de notificaciones que originalmente sólo manda emails. Con el tiempo, comienza a ser necesario el envío por SMS, Facebook, Slack, etc. Así, se crean subclases del `Notificador` para cada tipo de notificación.
Sumado a esto, aparece el problema de combinaciones: si una persona quiere varios medios de envío (email + SMS + Slack), se necesitarían combinaciones de subclases (ej: `EmailSMSNotifier`, `EmailSlackNotifier`, etc.), lo cual que escala rápidamente. Esto puede crecer mucho en número de clases, volverse muy difícil de mantener.

Estructura

- **Componente**: interfaz común para el componente base y los decoradores.
- **Componente concreto**: la clase que define el comportamiento básico.
- **Decorador base**: almacena la referencia al objeto “envuelto” (*wrappee*), declara dicha referencia de tipo interfaz del Componente. Y delegará todas (o casi todas) las operaciones al objeto “envuelto”.
- **Decoradores concretos** (Concrete Decorators): extienden del decorador base, añaden comportamiento adicional antes o después de delegar al objeto interior.
- **Cliente**: ensambla los objetos (componente base + uno o más decoradores) según lo necesite, y trabaja siempre mediante la interfaz común, sin importar qué capas de decorador haya.

Aplicabilidad (cuándo usarlo)

- Cuando necesitan asignar comportamientos extra a objetos en tiempo de ejecución y no romper el código que los usa.
- Cuando la herencia sola no sea suficiente o sea poco práctica (por ejemplo demasiadas combinaciones posibles).
- Cuando no puedan modificar la clase base pero necesitan extender su funcionalidad.

→ Regla para decidir

- Pocas variantes y no es necesario combinarlas → herencia simple puede alcanzar.
- Muchas variantes con combinaciones → Decorator da composición sin explosión de clases.

Decorator

- Ventajas
    - Extiende comportamiento sin tener que crear muchas subclases.
    - Permite añadir o quitar responsabilidades de un objeto en tiempo de ejecución.
    - Se pueden combinar varios decoradores para lograr comportamientos complejos de forma flexible.
    - Favorece el Principio de Responsabilidad Única, porque puedes dividir comportamientos en clases más pequeñas.

Decorator

- Desventajas
    - Es difícil eliminar un decorador específico de la pila de decoradores una vez que está aplicado.
    - El orden en que se aplican decoradores puede afectar mucho el resultado; puede no ser trivial garantizar que el comportamiento sea independiente del orden.
    - El código que configura los decoradores puede volverse complicado si hay muchas combinaciones posibles.

Relación con otros patrones

- **Adapter**: cambia la interfaz, **Decorator** la mantiene o extiende.
- **Proxy**: estructura similar pero distinto objetivo. **Proxy** controla acceso, Decorator añade comportamiento extra.
- **Strategy**: cambia la manera interna de hacer algo, mientras que **Decorator** cambia “lo que hace” pero manteniendo la interfaz.

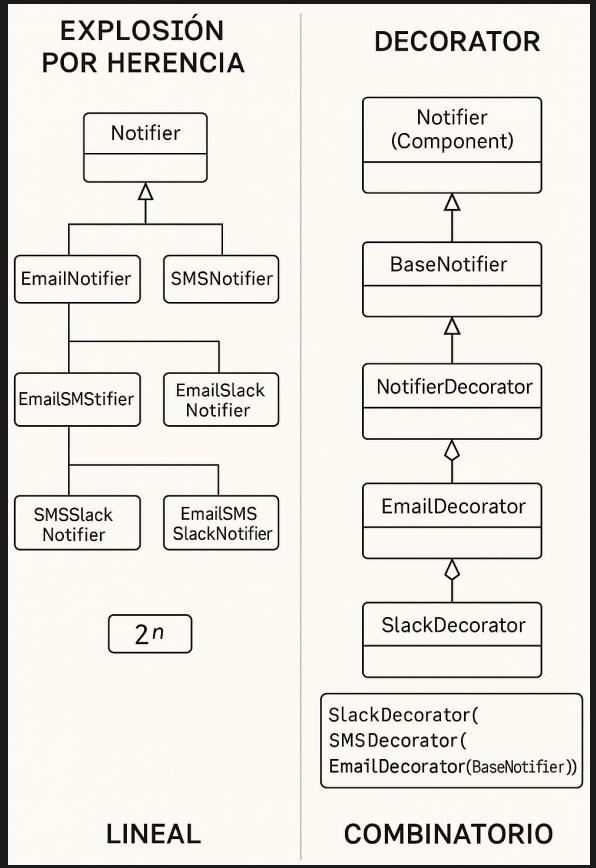
Decorator - Ejemplos

- Python
  - decorator.py
- TS
  - decorator.ts

Casos de uso

- Una clase Logger que ya funciona.
  Decoradores permiten enviar logs a un archivo, a una base de datos, a consola, o a un servicio externo, sin cambiar el código original.

Decorator – Diagrama



Patrón Facade

Jeremías Fassi

2025

Definición

El patrón **Facade** proporciona una interfaz unificada y simplificada a un conjunto complejo de clases o subsistemas, por ejemplo: librerías o frameworks.
Su objetivo es reducir el acoplamiento del cliente con los detalles internos y hacer más fácil el uso del sistema.

- Ejemplo abstracto: la fachada de un edificio.
  El usuario ve una entrada sencilla, pero detrás hay una estructura compleja de paredes, caños y cables.

Motivación

En proyectos grandes hay muchos módulos interconectados, y no siempre queremos exponer esa complejidad al cliente.

- Problema que resuelve

  Cuando el código debe tratar con muchas clases de una librería (inicialización, dependencias, orden correcto de llamadas, etc.), el **código de negocio** se vuelve difícil de entender y mantener por el fuerte acoplamiento.

Motivación

- Solución

  Así como en el ejemplo del edificio, la idea es introducir una clase *Facade* que exponga sólo lo esencial para el cliente y delegue en las clases internas del subsistema. El cliente trabaja con un método sencillo, mientras la fachada gestiona toda la complejidad.

Estructura (roles)

- **Facade**: interfaz simple; orquesta llamadas y puede gestionar ciclo de vida de objetos internos.
- **Subsistemas**: clases complejas reales (no conocen a la fachada).
- **Cliente**: usa sólo la fachada.

Si conviene, suele existir más de una fachada por módulo.

Facade

Señales de que conviene usar Facade

- El equipo se queja de que “usar la librería X es un lío”.
- Alta rotación en la API de terceros: es necesario “escudar” o proteger el dominio propio.
- La capa de aplicación repite siempre el mismo ritual de pasos para lograr algo: centralizarlo en la fachada.

Facade

- Ventajas
    - Simplifica el uso del subsistema; mejora legibilidad y usabilidad.
    - Aísla al cliente de cambios internos.
    - Facilita testing (es posible mockear la fachada).
- Desventajas
    - Puede convertirse en una especie de “Dios” si tiene demasiada lógica; recomendación, fachadas delgadas y, si hace falta, varias fachadas por áreas.

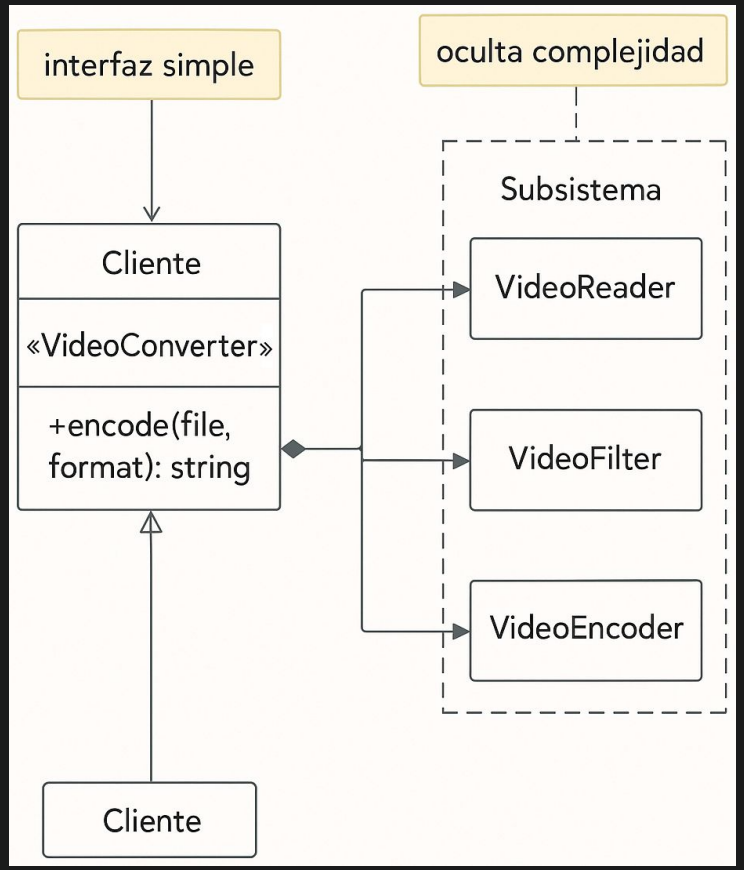
Relación con otros patrones

- Adapter vs Facade: **Adapter** cambia interfaz para compatibilidad; **Facade** simplifica sin cambiar el contrato del subsistema.
- **Decorator** añade responsabilidades; **Facade** oculta complejidad.
- **Proxy** controla acceso; **Facade** reduce complejidad visible.

Facade - Ejemplos

- Python
  - facade.py
- TS
  - facade.ts

Facade - Diagrama



Comparación de Patrones Estructurales

Patrón	Concepto	Motivación	Cuándo	Caso de Uso
Adapter	Convierte la interfaz de una clase en otra esperada por el cliente (actúa como "traductor").	Integrar componentes con interfaces incompatibles (código *legacy* o lib externas) sin tocarlos.	Cuando es necesario usar una clase existente que no cumple la interfaz que espera nuestro sistema.	Comunicamos con una librería que entrega XML y nuestra app espera JSON: un Adapter convierte la salida antes de entregarla al cliente.
Decorator	Añade responsabilidades extra a un objeto dinámicamente, sin modificar su código.	Evitar herencias rígidas y explosión de subclases para agregar comportamientos transversales (logging, cache, auth).	Cuando es necesario extender funcionalidades de forma selectiva y combinable en tiempo de ejecución.	Un `Notifier` base al que le agregamos decoradores de Email, SMS y Slack para enviar por múltiples canales.
Facade	Provee una interfaz unificada y simple a un conjunto de clases/subsistemas complejos.	Ocultar complejidad interna y reducir acoplamiento del cliente con múltiples APIs o módulos.	Cuando se quiere exponer una API sencilla para tareas comunes y desacoplar al cliente de detalles internos.	Una clase `SystemFacade` que inicializa servicios (DB, cache, colas de mensajes, etc) y ofrece métodos simples como `generateDataEntry()` o `runWorkflow()`.

Comparación de Patrones Estructurales

Resumen intuitivo

- Adapter → “Hace de traductor entre interfaces que no encajan.”
- Decorator → “Agrega capas extra de funcionalidad como si fueran adornos, sin tocar el objeto original.”
- Facade → “Muestra una puerta de entrada simple que oculta toda la complejidad detrás.”

Recursos:

- Patrones Estructurales

- DigitalOcean Community – GoF Patterns Explained

Muchas Gracias

Jeremías Fassi

Javier Kinter