# Patrones de diseño

# Patrones Comportamiento


Fernando Díaz-Alonso Dorado

# Patrones comportamiento

## ¿Qué son?

Los patrones de comportamiento tratan con algoritmos y la asignación de responsabilidades entre objetos.

# Patrones comportamiento

## Principales patrones comportamiento

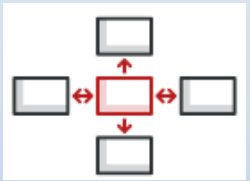
Algunos patrones comportamiento son:

- Chain of Responsability

- Command

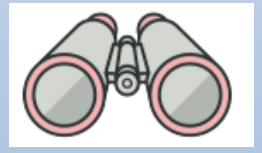
- Iterator

- Mediator



- Memento

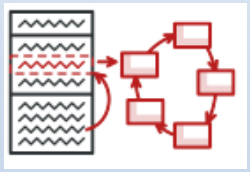
- Observer



# Patrones comportamiento

## Principales patrones comportamiento II

- State



- Strategy

- Template Method

- Visitor

# Chain of Responsability

## Cadena de responsabilidad, CoR, Chain of Command

Es un patrón de diseño de comportamiento que te permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.

**Aplicabilidad:**

- Cuando tu programa deba procesar distintos tipos de solicitudes de varias maneras, pero los tipos exactos de solicitudes y sus secuencias no se conozcan de antemano.
- Cuando sea fundamental ejecutar varios manejadores en un orden específico.
- Cuando el grupo de manejadores y su orden deban cambiar durante el tiempo de ejecución.

# Chain of Responsability

## Cadena de responsabilidad, CoR, Chain of Command II

**Pros y contras:**

- ✓ Puedes controlar el orden de control de solicitudes.
- ✓ *Principio de responsabilidad única.* Puedes desacoplar las clases que invoquen operaciones de las que realicen operaciones.
- ✓ *Principio de abierto/cerrado.* Puedes introducir nuevos manejadores en la aplicación sin descomponer el código cliente existente.
- ✗ Algunas solicitudes pueden acabar sin ser gestionadas.

# Command

## Comando, Orden, Action, Transaction

Es un patrón de diseño de comportamiento que convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación te permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar.

**Aplicabilidad:**

- Cuando quieras parametrizar objetos con operaciones.
- Cuando quieras poner operaciones en cola, programar su ejecución, o ejecutarlas de forma remota.
- Cuando quieras implementar operaciones reversibles.

# Command

## Comando, Orden, Action, Transaction II

**Pros y contras:**

- ✓ Puedes implementar deshacer/rehacer.
- ✓ Puedes implementar la ejecución diferida de operaciones.
- ✓ Puedes ensamblar un grupo de comandos simples para crear uno complejo.
- ✓ *Principio de responsabilidad única*. Puedes desacoplar las clases que invocan operaciones de las que realizan esas operaciones.
- ✓ *Principio de abierto/cerrado.* Puedes introducir nuevos comandos en la aplicación sin descomponer el código cliente existente.
- × El código puede complicarse, ya que estás introduciendo una nueva capa entre emisores y receptores.

# Iterator

## Iterator

Es un patrón de diseño de comportamiento que te permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.)

**Aplicabilidad:**

- Cuando tu colección tenga una estructura de datos compleja a nivel interno, pero quieras ocultar su complejidad a los clientes (ya sea por conveniencia o por razones de seguridad).
- Para reducir la duplicación en el código de recorrido a lo largo de tu aplicación.
- Cuando quieras que tu código pueda recorrer distintas estructuras de datos, o cuando los tipos de estas estructuras no se conozcan de antemano.

# Iterator

## Iterator II

**Pros:**

- ✓ *Principio de responsabilidad única*. Puedes limpiar el código cliente y las colecciones extrayendo algoritmos de recorrido voluminosos y colocándolos en clases independientes.
- ✓ *Principio de abierto/cerrado.* Puedes implementar nuevos tipos de colecciones e iteradores y pasarlos al código existente sin descomponer nada.
- ✓ Puedes recorrer la misma colección en paralelo porque cada objeto iterador contiene su propio estado de iteración.
- ✓ Por la misma razón, puedes retrasar una iteración y continuar cuando sea necesario.

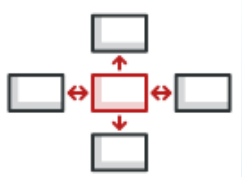
# Iterator

## Iterator II

**Contras:**

- Aplicar el patrón puede resultar excesivo si tu aplicación funciona únicamente con colecciones sencillas.
- Utilizar un iterador puede ser menos eficiente que recorrer directamente los elementos de algunas colecciones especializadas.

# Mediator



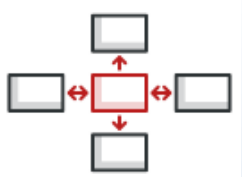
## Mediador, Intermediary, Controller

Es un patrón de diseño de comportamiento que te permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador.

**Aplicabilidad:**

- Cuando resulte difícil cambiar algunas de las clases porque están fuertemente acopladas a un puñado de otras clases.
- Cuando no puedas reutilizar un componente en un programa diferente porque sea demasiado dependiente de otros componentes.
- Cuando te encuentres creando cientos de subclases de componente sólo para reutilizar un comportamiento básico en varios contextos

# Mediator



## Mediador, Intermediary, Controller II

**Pros y Contras:**

- ✓ *Principio de responsabilidad única.* Puedes extraer las comunicaciones entre varios componentes dentro de un único sitio, haciéndolo más fácil de comprender y mantener.
- ✓ *Principio de abierto/cerrado.* Puedes introducir nuevos mediadores sin tener que cambiar los propios componentes.
- ✓ Puedes reducir el acoplamiento entre varios componentes de un programa.
- ✓ Puedes reutilizar componentes individuales con mayor facilidad.
- ✗ Con el tiempo, un mediador puede evolucionar a un objeto todopoderoso.

# Memento

## Recuerdo, Instantánea, Snapshot

Es un patrón de diseño de comportamiento que te permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación.

**Aplicabilidad:**

- Cuando quieras producir instantáneas del estado del objeto para poder restaurar un estado previo del objeto.
- Cuando el acceso directo a los campos, consultores o modificadores del objeto viole su encapsulación.

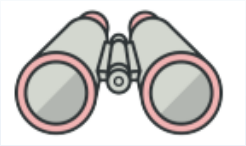
# Memento

## Recuerdo, Instantánea, Snapshot II

**Pros y contras:**

- ✓ Puedes producir instantáneas del estado del objeto sin violar su encapsulación.
- ✓ Puedes simplificar el código de la originadora permitiendo que la cuidadora mantenga el historial del estado de la originadora.
- × La aplicación puede consumir mucha memoria RAM si los clientes crean mementos muy a menudo.
- × Las cuidadoras deben rastrear el ciclo de vida de la originadora para poder destruir mementos obsoletos.
- × **La mayoría de los lenguajes de programación dinámicos, como PHP, Python y JavaScript, no pueden garantizar que el estado dentro del memento se mantenga intacto.**

# Observer



## Observador, Publicación-Suscripción, Modelo-patrón, Event-Subscriber, Listener

Es un patrón de diseño de comportamiento que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.

**Aplicabilidad:**

- Cuando los cambios en el estado de un objeto puedan necesitar cambiar otros objetos y el grupo de objetos sea desconocido de antemano o cambie dinámicamente.
- Cuando algunos objetos de tu aplicación deban observar a otros, pero sólo durante un tiempo limitado o en casos específicos

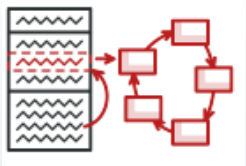
# Observer

## Observador, Publicación-Suscripción, Modelo-patrón, Event-Subscriber, Listener II

**Pros y contras:**

- ✓ *Principio de abierto/cerrado.* Puedes introducir nuevas clases suscriptoras sin tener que cambiar el código de la notificadora (y viceversa si hay una interfaz notificadora).
- ✓ Puedes establecer relaciones entre objetos durante el tiempo de ejecución.
- ✗ Los suscriptores son notificados en un orden aleatorio.

# State



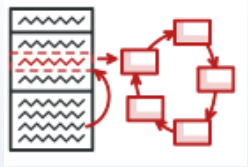
## Estado

Es un patrón de diseño de comportamiento que permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiara su clase.

**Aplicabilidad:**

- Cuando tengas un objeto que se comporta de forma diferente dependiendo de su estado actual, el número de estados sea enorme y el código específico del estado cambie con frecuencia.
- Cuando tengas una clase contaminada con enormes condicionales que alteran el modo en que se comporta la clase de acuerdo con los valores actuales de los campos de la clase.
- Cuando tengas mucho código duplicado por estados similares y transiciones de una máquina de estados basada en condiciones.

# State



## Estado II

**Pros y contras:**

- ✓ *Principio de responsabilidad única.* Organiza el código relacionado con estados particulares en clases separadas.
- ✓ *Principio de abierto/cerrado.* Introduce nuevos estados sin cambiar clases de estado existentes o la clase contexto.
- ✓ Simplifica el código del contexto eliminando voluminosos condicionales de máquina de estados.
- ✗ Aplicar el patrón puede resultar excesivo si una máquina de estados sólo tiene unos pocos estados o raramente cambia.

# Strategy

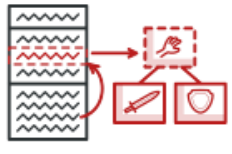
## Estrategia

Es un patrón de diseño de comportamiento que te permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.

**Aplicabilidad:**

- Cuando quieras utiliza distintas variantes de un algoritmo dentro de un objeto y poder cambiar de un algoritmo a otro durante el tiempo de ejecución.
- Cuando tengas muchas clases similares que sólo se diferencien en la forma en que ejecutan cierto comportamiento.

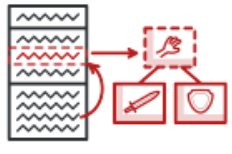
# Strategy



## Estrategia II

- Para aislar la lógica de negocio de una clase, de los detalles de implementación de algoritmos que pueden no ser tan importantes en el contexto de esa lógica.
- Cuando tu clase tenga un enorme operador condicional que cambie entre distintas variantes del mismo algoritmo.

# Strategy

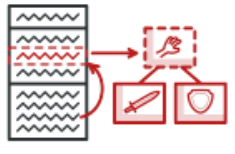


## Estrategia III

**Pros:**

- ✓ Puedes intercambiar algoritmos usados dentro de un objeto durante el tiempo de ejecución.
- ✓ Puedes aislar los detalles de implementación de un algoritmo del código que lo utiliza.
- ✓ Puedes sustituir la herencia por composición.
- ✓ *Principio de abierto/cerrado.* Puedes introducir nuevas estrategias sin tener que cambiar el contexto.

# Strategy



## Estrategia IV

**Contras:**

- Si sólo tienes un par de algoritmos que raramente cambian, no hay una razón real para complicar el programa en exceso con nuevas clases e interfaces que vengan con el patrón.
- Los clientes deben conocer las diferencias entre estrategias para poder seleccionar la adecuada.
- Muchos lenguajes de programación modernos tienen un soporte de tipo funcional que te permite implementar distintas versiones de un algoritmo dentro de un grupo de funciones anónimas. Entonces puedes utilizar estas funciones exactamente como habrías utilizado los objetos de estrategia, pero sin saturar tu código con clases e interfaces adicionales.

# Template Method

## Método plantilla

Es un patrón de diseño de comportamiento que define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura.

**Aplicabilidad:**

- Cuando quieras permitir a tus clientes que extiendan únicamente pasos particulares de un algoritmo, pero no todo el algoritmo o su estructura.
- Cuando tengas muchas clases que contengan algoritmos casi idénticos, pero con algunas diferencias mínimas. Como resultado, puede que tengas que modificar todas las clases cuando el algoritmo cambie.

# Template Method

## Método plantilla II

**Pros y contras:**

- ✓ Puedes permitir a los clientes que sobrescriban tan solo ciertas partes de un algoritmo grande, para que les afecten menos los cambios que tienen lugar en otras partes del algoritmo.
- ✓ Puedes colocar el código duplicado dentro de una superclase.
- × Algunos clientes pueden verse limitados por el esqueleto proporcionado de un algoritmo.
- × Puede que violes el *principio de sustitución de Liskov* suprimiendo una implementación por defecto de un paso a través de una subclase.
- × Los métodos plantilla tienden a ser más difíciles de mantener cuantos más pasos tengan.

# Visitor

## Visitante

Es un patrón de diseño de comportamiento que te permite separar algoritmos de los objetos sobre los que operan.

**Aplicabilidad:**

- Cuando necesites realizar una operación sobre todos los elementos de una compleja estructura de objetos (por ejemplo, un árbol de objetos).
- Para limpiar la lógica de negocio de comportamientos auxiliares.
- Cuando un comportamiento solo tenga sentido en algunas clases de una jerarquía de clases, pero no en otras.

# Visitor

## Visitante II

**Pros :**

- ✓ *Principio de abierto/cerrado.* Puedes introducir un nuevo comportamiento que puede funcionar con objetos de clases diferentes sin cambiar esas clases.
- ✓ *Principio de responsabilidad única.* Puedes tomar varias versiones del mismo comportamiento y ponerlas en la misma clase.
- ✓ Un objeto visitante puede acumular cierta información útil mientras trabaja con varios objetos. Esto puede resultar útil cuando quieras atravesar una compleja estructura de objetos, como un árbol de objetos, y aplicar el visitante a cada objeto de esa estructura.

# Visitor

## Visitante II

**Contras:**

- Debes actualizar todos los visitantes cada vez que una clase se añada o elimine de la jerarquía de elementos.
- Los visitantes pueden carecer del acceso necesario a los campos y métodos privados de los elementos con los que se supone que deben trabajar.

# Principios del diseño de software

## Bibliografía

- Patrones de Diseño – Alexander Shvets