Patrones de comportamiento

Mediator
Memento
Observer

Dashiell Atencio - 8-971-955
Leanis Bello - 8-976-976
Ángel Fong - 8-971-496
Hiroshi Komatsu - 8-993-1985
Alejandro Samudio - 8-972-368

Grupo: 1IL143

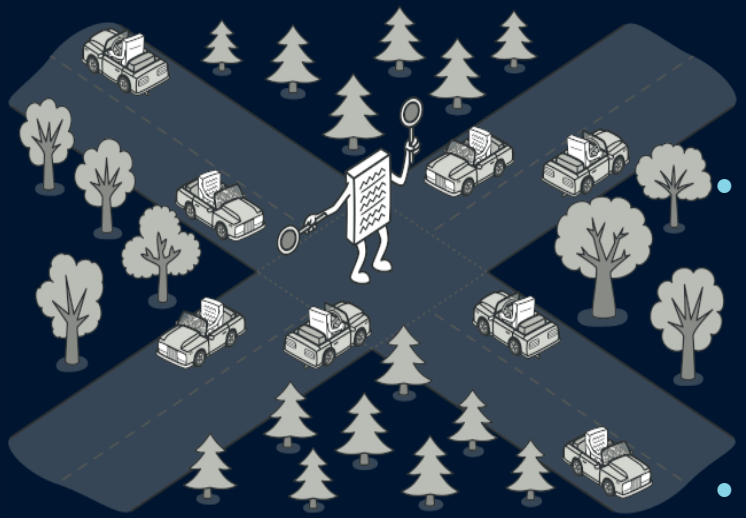
Mediator

01

El patrón de diseño de comportamiento Mediator se encarga de gestionar la forma en que un conjunto de clases se comunica entre sí. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador.

Es especialmente útil cuando tenemos una gran cantidad de clases que se comunican de forma directa, ya que mediante la implementación de este patrón podemos crear una capa de comunicación bidireccional, en la cual las clases se pueden comunicar con el resto de ellas por medio de un objeto en común que funge como un mediador o intermediario.

¿CUÁNDO USAR EL PATRÓN MEDIATOR?



- El patrón Mediador debería ser utilizado cuando sea difícil cambiar algunas de las clases porque están estrechamente acopladas a un montón de otras clases. Este patrón permite extraer todas las relaciones entre clases en una clase separada.
- También se podría utilizar cuando no se pueda reutilizar un componente en un programa diferente porque depende demasiado de otros componentes. Aplicando el Mediador, los componentes individuales no se dan cuenta de los otros componentes.
- Se usa cuando se estén creando demasiadas subclases de componentes solo para reutilizar algunos comportamientos básicos en varios contextos.

ESTRUCTURA

01

CLIENTE

Componente que inicia la comunicación con el resto de los componentes por medio del mediador.

02

COMPONENTES

Parte de la red de comunicación por medio del mediador.

03

MEDIADOR

Componente que sirve de mediador entre el resto de componentes.

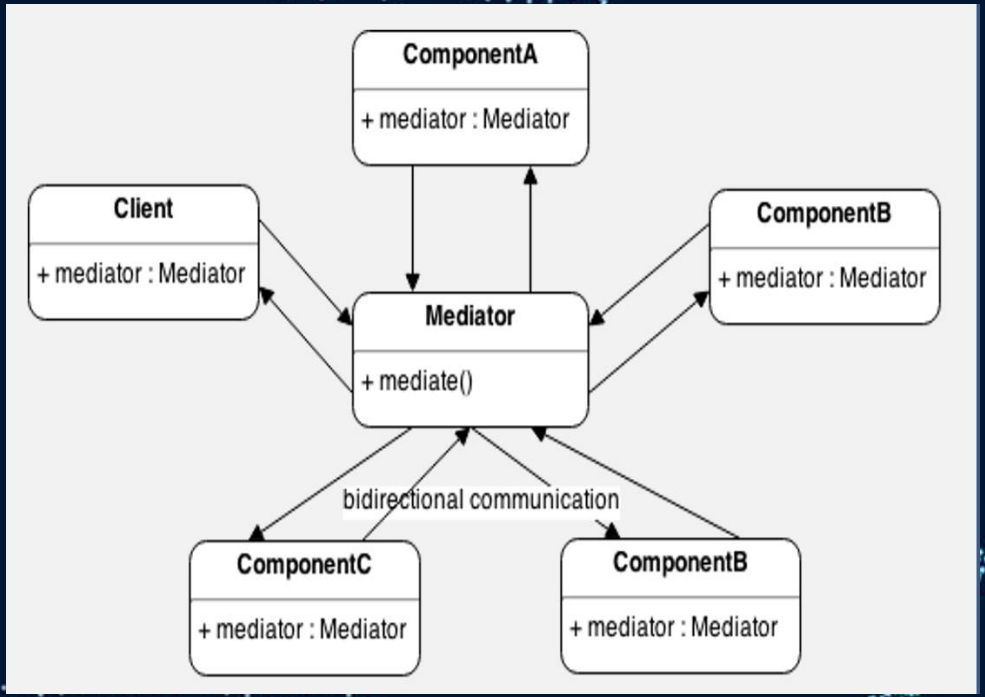
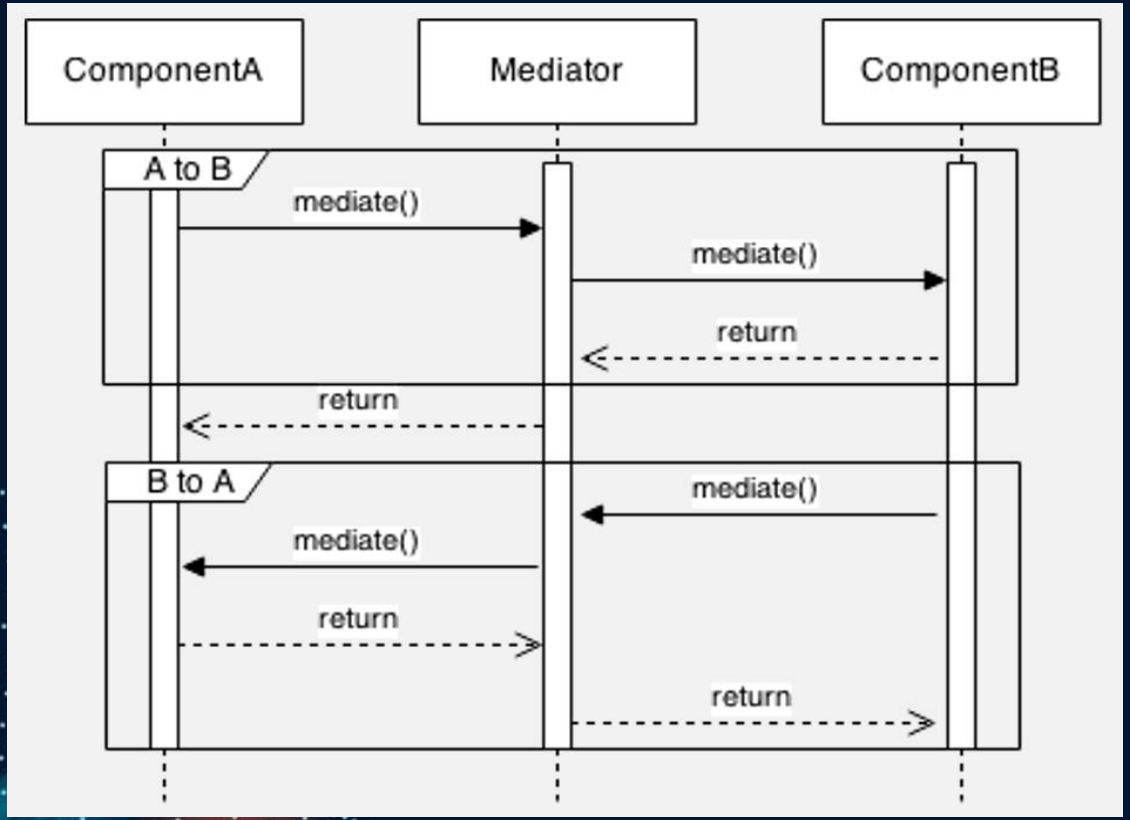


DIAGRAMA DE SECUENCIA DEL PATRÓN



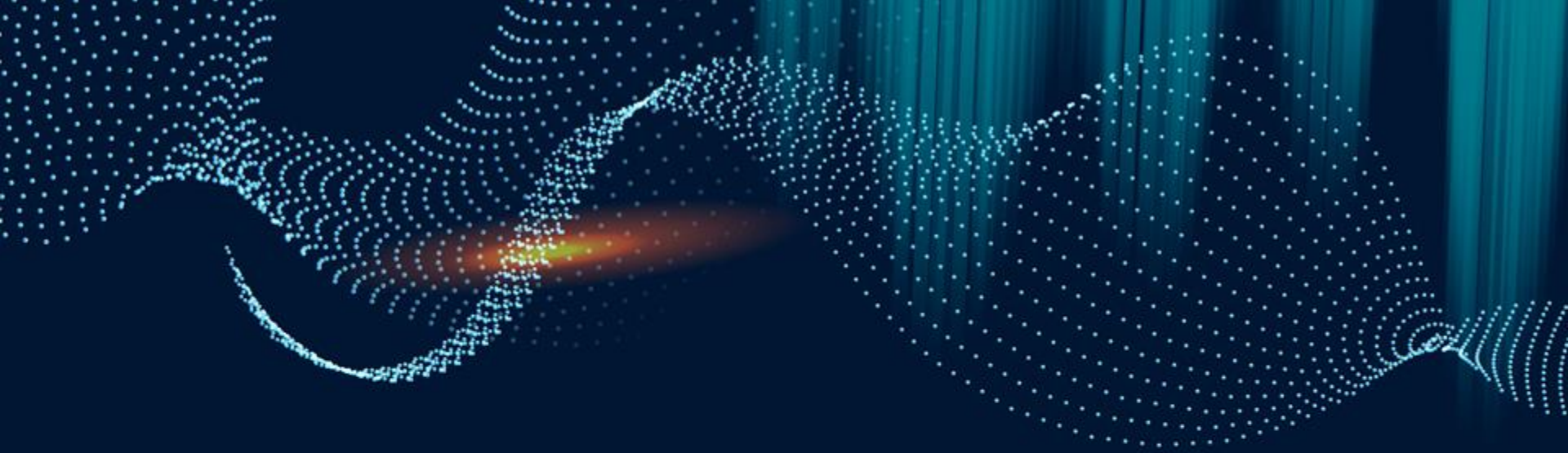
VENTAJAS Y DESVENTAJAS

VENTAJAS

- El patrón Mediator es especialmente útil cuando tenemos una gran cantidad de clases que se comunican de forma directa.
- Se hace responsable de los cambios necesarios en el sistema, dado que es el centro de control.
- Permite conectar y extraer componentes con facilidad.

DESVENTAJAS

- Proyectos grandes representan un problema de acoplamiento.
- El Mediator puede ser difícil de manejar ya que necesita ser indispensable para todas las diferentes clases.
- Reduce la herencia.



02

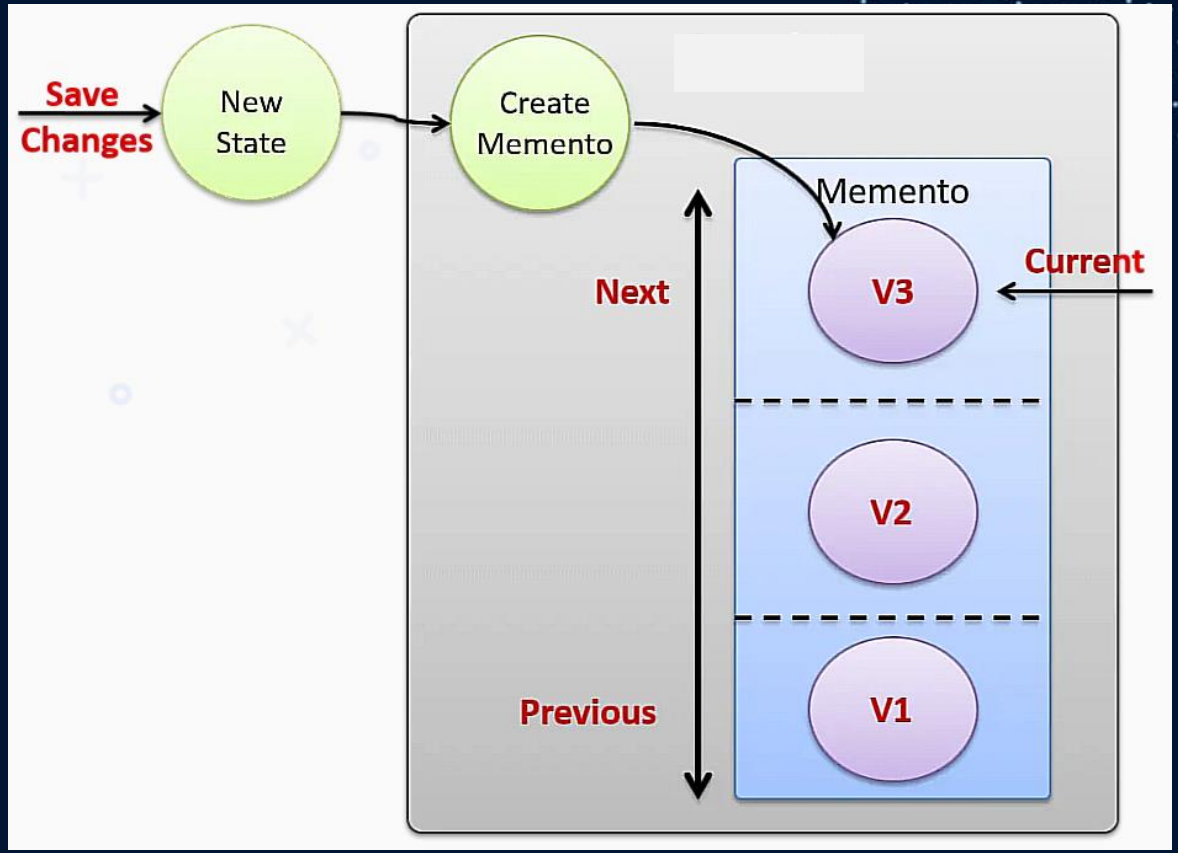
MEMENTO

Es un patrón de comportamiento que se encarga de capturar y almacenar el estado interno de un objeto en un momento dado, sin violar su encapsulación y proporciona la capacidad de restaurar ese estado en el futuro si es necesario.

¿QUÉ UTILIDAD TIENE EL PATRÓN MEMENTO?

La utilidad principal del patrón Memento es la implementación de mecanismos de "deshacer" o "revertir" en aplicaciones. Por ejemplo, en un editor de texto, podrías utilizar Memento para guardar el estado del texto antes de realizar una serie de cambios y luego restaurarlo si el usuario desea deshacer esos cambios. También se puede utilizar en juegos para guardar y cargar partidas en momentos específicos.

Ejemplo de implementación



¿Memento es lo mismo que una Copia de Seguridad?

VENTAJAS Y DESVENTAJAS

Ventajas

1. Preserva la encapsulación: El patrón Memento permite almacenar y restaurar el estado interno de un objeto sin exponer sus detalles internos, lo que ayuda a mantener la encapsulación y la integridad del objeto.

2. Flexibilidad en la restauración: Permite restaurar el estado de un objeto a diferentes puntos en el tiempo, lo que brinda flexibilidad y control al programador o usuario.

3. Desacopla el origen y la gestión del estado: El patrón Memento separa la lógica del objeto que desea guardar su estado y la lógica del objeto que gestiona los estados almacenados, lo que mejora la modularidad y el mantenimiento del código.

Desventajas

1. Uso intensivo de memoria: Si se guardan múltiples estados en un objeto Memento, puede requerir una cantidad considerable de memoria, especialmente si los objetos son grandes.

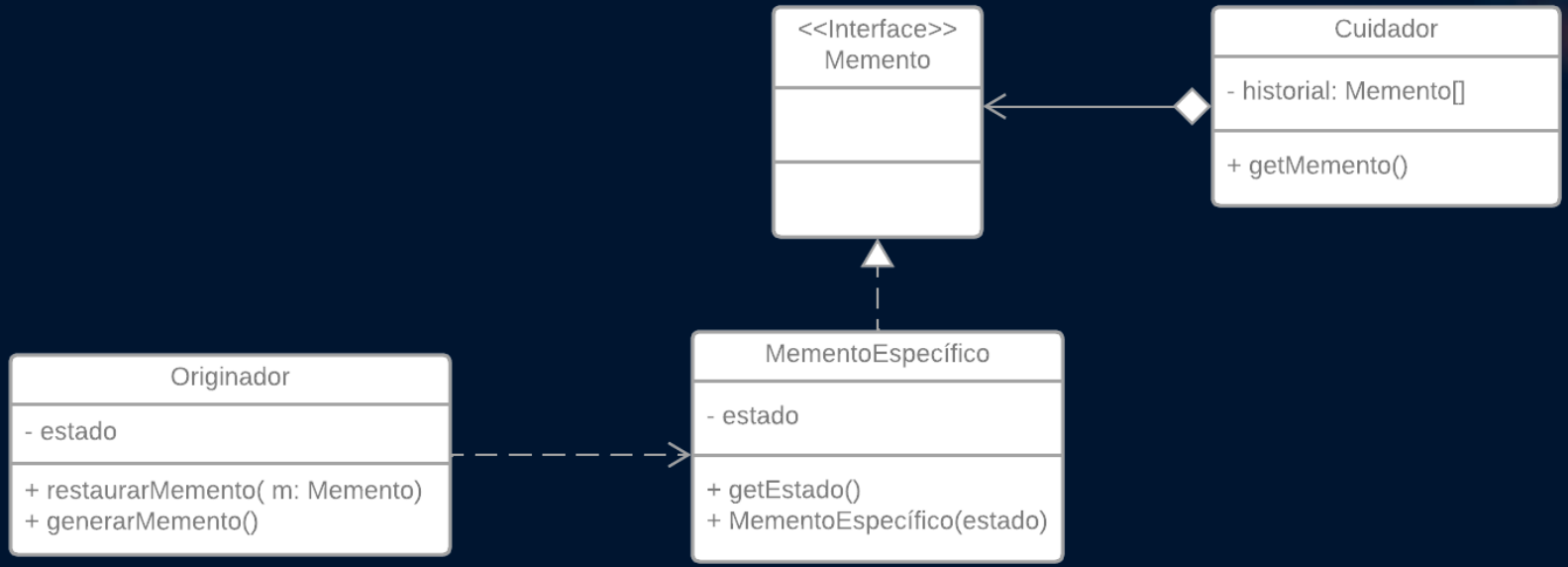
2. Sobrecarga de almacenamiento: El almacenamiento de múltiples estados puede resultar costoso en términos de almacenamiento en disco o en red, lo que puede afectar el rendimiento en aplicaciones que trabajan con grandes cantidades de datos.

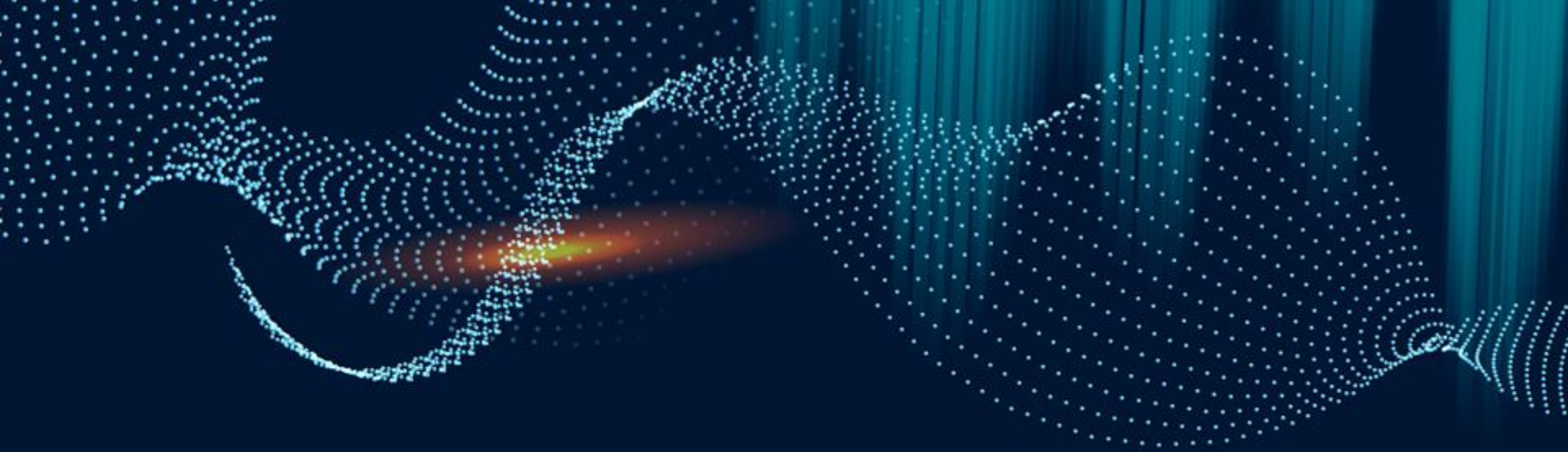
IMPLEMENTACIONES

PATRÓN MEMENTO

IMPLEMENTACIÓN BASADA EN CLASES ANIDADAS

IMPLEMENTACIÓN BASADA EN UNA INTERFAZ INTERMEDIA





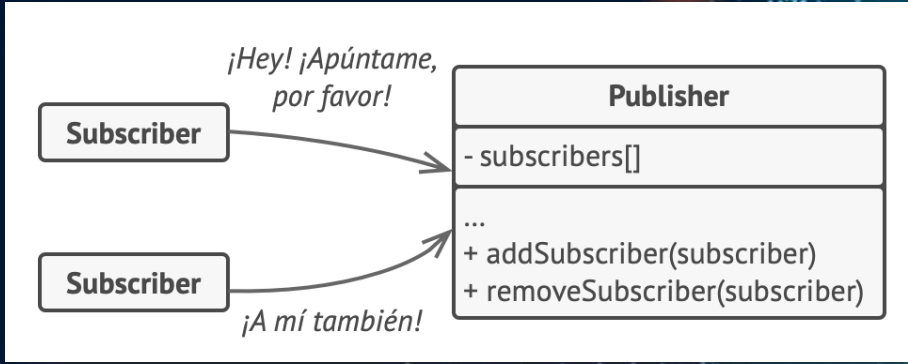
03

Observer

Observer 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.

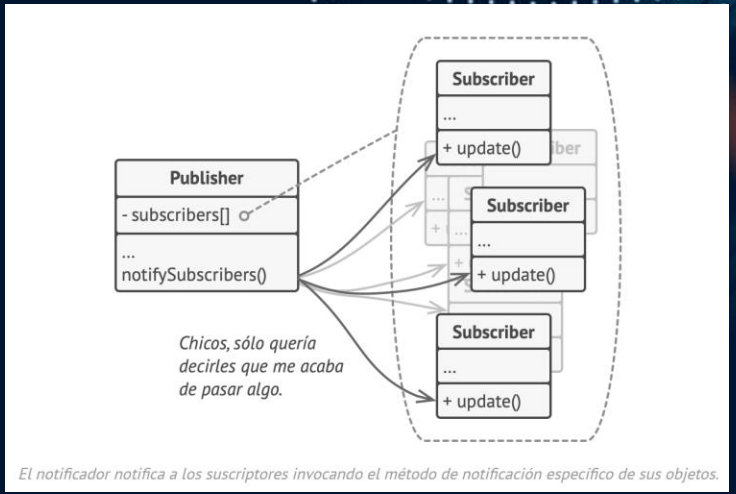
¿En qué consiste Observer?

- El objeto que tiene un estado interesante suele denominarse ***sujeto***, pero, como también va a notificar a otros objetos los cambios en su estado, le llamaremos ***notificador*** (en ocasiones también llamado publicador).
- El resto de los objetos que quieren conocer los cambios en el estado del notificador, se denominan suscriptores.
- Cuando le sucede un evento importante al notificador, recorre sus suscriptores y llama al método de notificación específico de sus objetos.



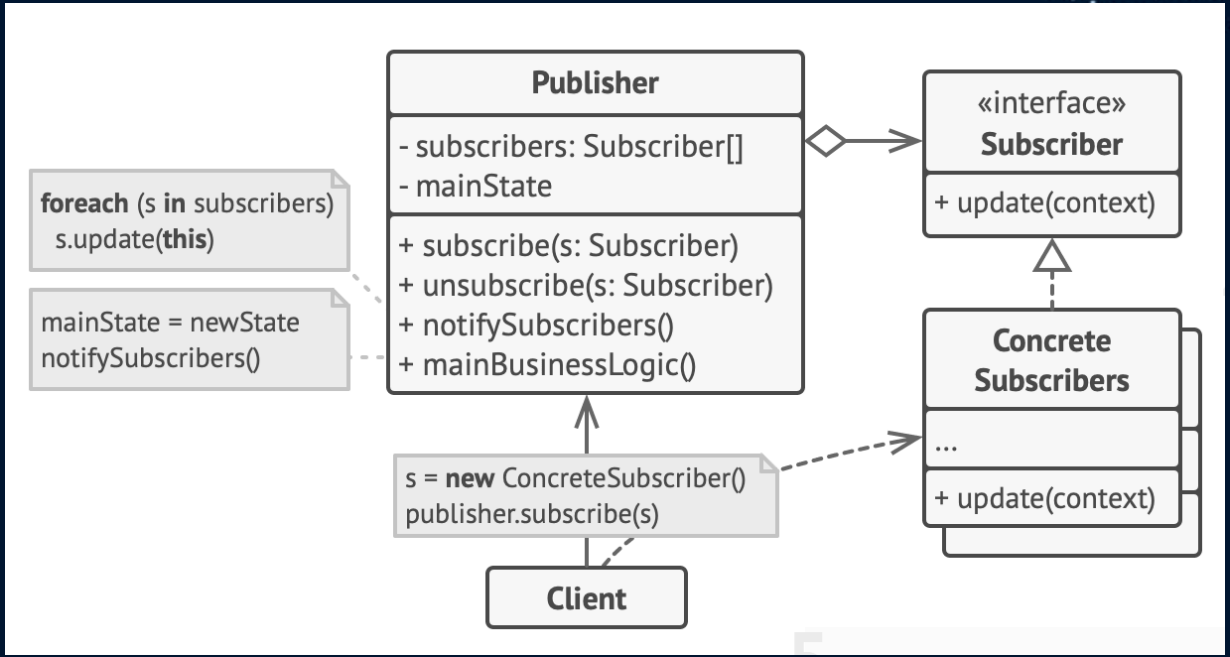
¿Cómo funciona?

- Las aplicaciones reales pueden tener decenas de clases suscriptoras diferentes interesadas en seguir los eventos de la misma clase notificadora.
- Es fundamental que todos los suscriptores implementen la misma interfaz y que el notificador únicamente se comunique con ellos a través de esa interfaz.
- Esta interfaz debe declarar el método de notificación junto con un grupo de parámetros que el notificador puede utilizar para pasar cierta información contextual con la notificación.



El notificador notifica a los suscriptores invocando el método de notificación específico de sus objetos.

Estructura


Publisher
- subscribers: Subscriber[]
- mainState
+ subscribe(s: Subscriber)
+ unsubscribe(s: Subscriber)
+ notifySubscribers()
+ mainBusinessLogic()
«interface»
Subscriber
+ update(context)
foreach (s in subscribers)
s.update(this)
mainState = newState
notifySubscribers()
Concrete
Subscribers
...
+ update(context)
s = new ConcreteSubscriber()
publisher.subscribe(s)
Client

¿Cómo aplicarlo?

1. Repasa la lógica de negocio e intenta dividirla en dos partes: la funcionalidad central, independiente del resto de código, actuará como notificador; el resto se convertirá en un grupo de clases suscriptoras.

2. Declara la interfaz suscriptora. Como mínimo, deberá declarar un único método actualizar.

3. Declara la interfaz notificadora y describe un par de métodos para añadir y eliminar de la lista un objeto suscriptor

4. Decide dónde colocar la lista de suscripción y la implementación de métodos de suscripción.

5. Si estás aplicando el patrón a una jerarquía de clases existentes, considera una solución basada en la composición: coloca la lógica de la suscripción en un objeto separado y haz que todos los notificadores reales la utilicen.

6. Crea clases notificadoras concretas.

7. Implementa los métodos de notificación de actualizaciones en clases suscriptoras concretas.
8. Pero hay otra opción. Al recibir una notificación, el suscriptor puede extraer la información directamente de ella.
9. El cliente debe crear todos los suscriptores necesarios y registrarlos con los notificadores adecuados.

Ejemplo 1 de aplicación - sistema de notificaciones para una aplicación móvil de noticias.

- **El publicador:** sería la fuente de las noticias.
- **Los observadores:** serían los usuarios que se han suscrito a las actualizaciones de las noticias.
- **Uso del patrón:** Cuando se publique una nueva noticia, el publicador notificará a los observadores (los usuarios suscritos), informándoles de la nueva noticia y permitiendo que puedan actualizar la información en sus dispositivos.

Ejemplo 2 de aplicación - sistema de alarmas para detectar fallos en un sistema informático.

- **El publicador:** sería el sistema informático.
- **Los observadores:** los sensores de fallos.
- **Uso del patrón:** Cuando se detecte un fallo en el sistema, el sujeto notificará a los observadores (los sensores de fallos), permitiendo que se tomen medidas para corregir el problema antes de que cause daños serios en el sistema.

En resumen...

- el patrón Observer es útil en situaciones donde se necesite notificar a múltiples objetos sobre cambios en un sujeto, permitiendo que los objetos observadores puedan tomar las medidas necesarias en un momento oportuno.

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.
- x Los suscriptores son notificados en un orden aleatorio.

Muchas gracias