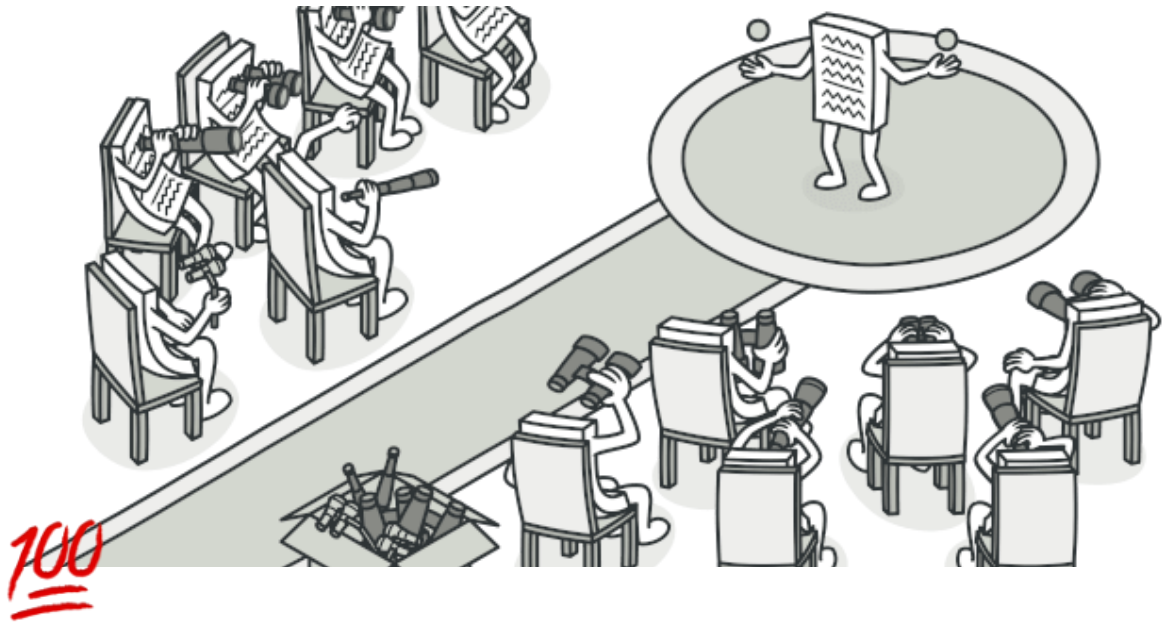


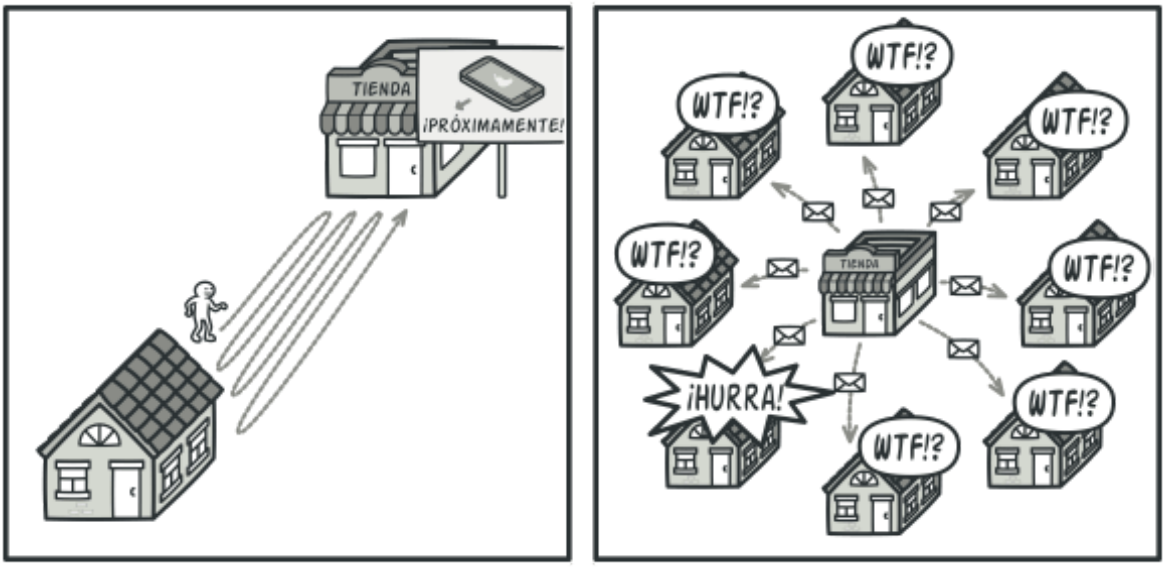
OBSERVER

- **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.

Problema

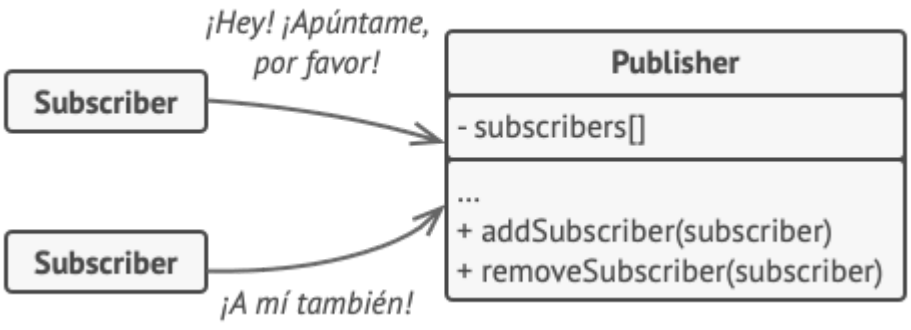
- Imagina que tienes dos tipos de objetos: un objeto `Cliente` y un objeto `Tienda`. El cliente está muy interesado en una marca particular de producto (digamos, un nuevo modelo de iPhone) que estará disponible en la tienda muy pronto.

El cliente puede visitar la tienda cada día para comprobar la disponibilidad del producto. Pero, mientras el producto está en camino, la mayoría de estos viajes serán en vano.



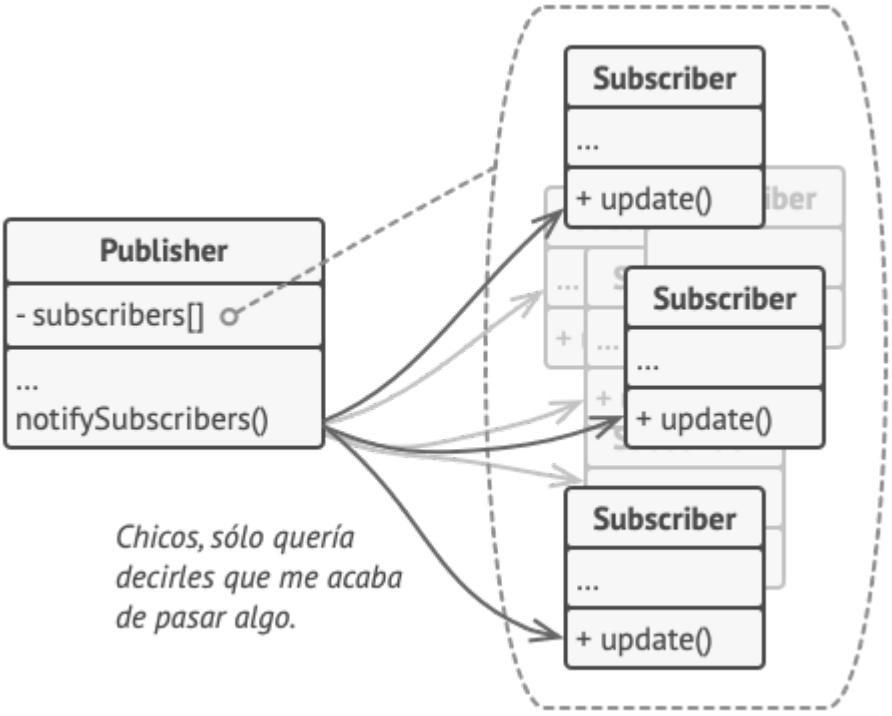
El problema descrito surge cuando una tienda podría enviar correos masivos a todos los clientes cada vez que hay un nuevo producto, lo que puede resultar en spam para algunos y ahorro de tiempo para otros. Esto genera un conflicto entre la pérdida de tiempo de los clientes al revisar productos y el desperdicio de recursos de la tienda al notificar a clientes no interesados.

La solución propuesta es el patrón Observer, en el que un objeto, llamado notificador, mantiene una lista de suscriptores interesados en sus actualizaciones. Los suscriptores pueden añadirse o eliminarse del flujo de notificaciones según lo deseen, evitando notificaciones innecesarias.



> Cuando ocurre un evento importante en el notificador, este recorre su lista de suscriptores y llama al método de notificación correspondiente. En aplicaciones reales, puede haber muchas clases diferentes interesadas en estos eventos, y no sería práctico acoplar la notificadora a todas ellas. Además, algunas clases suscriptoras podrían ser desconocidas.

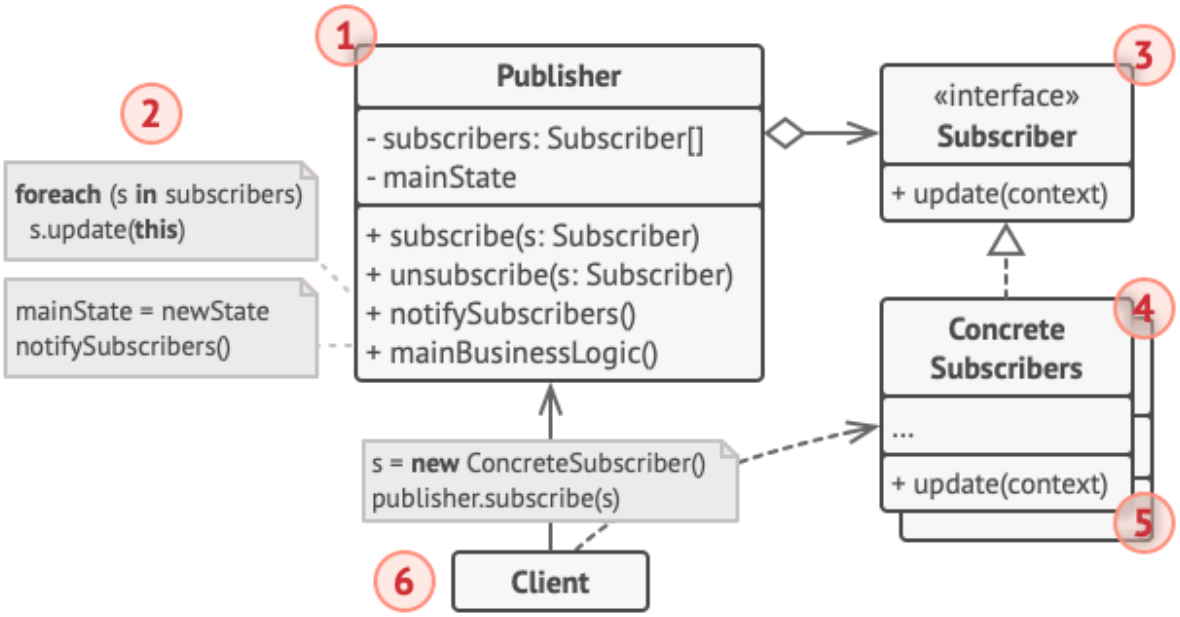
Por ello, todos los suscriptores deben implementar la misma interfaz, permitiendo que el notificador se comunique con ellos de manera uniforme. Esta interfaz define el método de notificación y los parámetros necesarios para transmitir información relevante.



- Si te suscribes a un periódico o una revista, ya no necesitarás ir a la tienda a comprobar si el siguiente número está disponible. En lugar de eso, el

notificador envía nuevos números directamente a tu buzón justo después de la publicación, o incluso antes.

- El notificador mantiene una lista de suscriptores y sabe qué revistas les interesan. Los suscriptores pueden abandonar la lista en cualquier momento si quieren que el notificador deje de enviarles nuevos números.



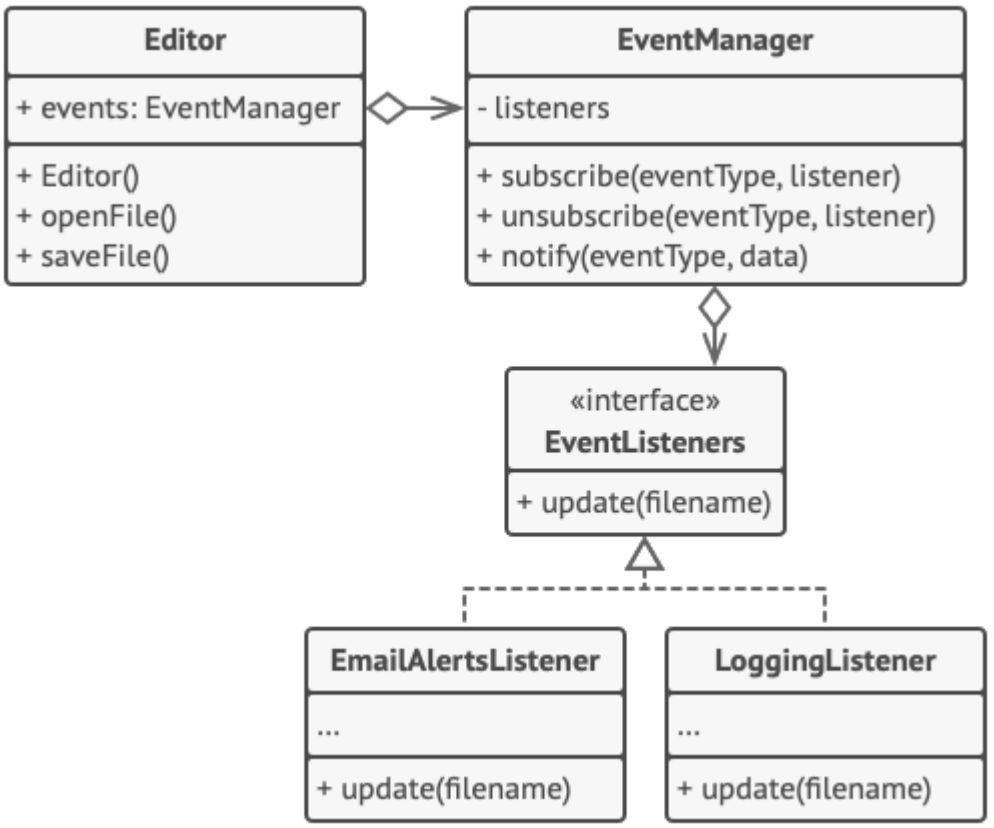
El notificador envía eventos a suscriptores cuando cambia su estado o realiza alguna acción, gestionando una lista de suscripción que permite agregar o eliminar suscriptores. Cuando ocurre un evento, recorre la lista y llama al método de notificación en cada suscriptor.

La interfaz de los suscriptores, que generalmente incluye un método `Actualizar`, define cómo reciben las notificaciones. Los suscriptores concretos responden a los eventos implementando esta interfaz, lo que evita acoplar el notificador a clases específicas.

El cliente crea notificadores y suscriptores, y registra a estos últimos para recibir actualizaciones del notificador.

#Pseudocódigo

En este ejemplo, el patrón **Observer** permite al objeto editor de texto notificar a otros objetos tipo servicio sobre los cambios en su estado.



La lista de suscriptores se gestiona dinámicamente, permitiendo que los objetos se suscriban o se den de baja durante la ejecución, según sea necesario. La clase editora delega la gestión de suscriptores a un objeto especial que centraliza el despacho de eventos, permitiendo que cualquier objeto actúe como notificador. Agregar nuevos suscriptores no requiere modificar las clases notificadoras, siempre que se comuniquen mediante la misma interfaz.

```
// La clase notificadora base incluye código de gestión de
// suscripciones y métodos de notificación.
classEventManagerisprivatefield listeners: hash map of even
t typesand listeners

method subscribe(eventType, listener)islisteners.add(eventT
ype, listener)
```

```
method unsubscribe(eventType, listener)islisteners.remove(e
ventType, listener)

method notify(eventType, data)isforeach (listener in listen
ers.of(eventType)) do
            listener.update(data)

// El notificador concreto contiene lógica de negocio real,
de
// interés para algunos suscriptores. Podemos derivar esta
clase
// de la notificadora base, pero esto no siempre es posible
en
// el mundo real porque puede que la notificadora concreta
sea
// ya una subclase. En este caso, puedes modificar la lógic
a de
// la suscripción con composición, como hicimos aquí.
classEditorispublicfield events: EventManager
privatefield file: File

constructor Editor()isevents =new EventManager()

    // Los métodos de la lógica de negocio pueden notificar
los
    // cambios a los suscriptores.
method openFile(path)isthis.file =new File(path)
        events.notify("open", file.name)

method saveFile()isfile.write()
        events.notify("save", file.name)

    // ...


// Aquí está la interfaz suscriptora. Si tu lenguaje de
// programación soporta tipos funcionales, puedes sustituir
toda
```

```
// la jerarquía suscriptora por un grupo de funciones.


interfaceEventListenerismethod update(filename)

// Los suscriptores concretos reaccionan a las actualizacio
nes
// emitidas por el notificador al que están unidos.
classLoggingListenerimplements EventListenerisprivatefield
log: File
privatefield message: string

constructor LoggingListener(log_filename, message)isthis.lo
g =new File(log_filename)
this.message = message

method update(filename)islog.write(replace('%s',filename,me
ssage))

classEmailAlertsListenerimplements EventListenerisprivatefi
eld email: string
privatefield message: string

constructor EmailAlertsListener(email, message)isthis.email
= email
this.message = message

method update(filename)issystem.email(email, replace('%s',f
ilename,message))


// Una  aplicación puede configurar notificadores y suscrip
tores
// durante el tiempo de ejecución.
classApplicationismethod config()iseditor =new Editor()

        logger =new LoggingListener(
            "/path/to/log.txt",
```

```
            "Someone has opened the file: %s")
        editor.events.subscribe("open", logger)

        emailAlerts =new EmailAlertsListener(
            "admin@example.com",
            "Someone has changed the file: %s")
        editor.events.subscribe("save", emailAlerts)
```

APLICABILIDAD

- El patrón Observer se utiliza cuando los cambios en el estado de un objeto deben reflejarse en otros objetos, pero no se conoce de antemano qué objetos serán afectados o estos cambian dinámicamente.

Es común en interfaces gráficas, permitiendo que los clientes añadan código personalizado a eventos, como un botón que activa acciones al ser pulsado. Este patrón permite que cualquier objeto que implemente la interfaz de suscriptor se suscriba a notificaciones.

- La lista de suscriptores es dinámica, por lo que pueden unirse o abandonar en cualquier momento.

COMO IMPLEMENTARLO

1. **Divide la lógica**: Separa la funcionalidad central, que actuará como notificador, de otras partes del código, que serán las clases suscriptoras.
2. **Define la interfaz suscriptora**: Debe incluir al menos un método `actualizar`.
3. **Define la interfaz notificadora**: Incluye métodos para añadir y eliminar suscriptores, trabajando solo a través de la interfaz suscriptora.
4. **Implementa la suscripción**: Coloca la lista de suscripción y la implementación en una clase abstracta derivada de la interfaz notificadora,

que los notificadores concretos extenderán. Si ya tienes una jerarquía de clases, considera usar composición para manejar la suscripción.

5. **Crea clases notificadoras concretas**: Estas deben notificar a todos los suscriptores cuando ocurran eventos importantes.
6. **Implementa métodos de actualización en los suscriptores**: Los suscriptores concretos deben manejar la información del evento, que puede ser pasada como argumento o extraída directamente del notificador.
7. **Configura suscriptores**: El cliente debe crear y registrar todos los suscriptores con los notificadores adecuados.

PROS	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).	Los suscriptores son notificados en un orden aleatorio.
Puedes establecer relaciones entre objetos durante el tiempo de ejecución.	

RELACIONES CON OTROS PATRONES

Los patrones **Chain of Responsibility**, **Command**, **Mediator** y **Observer** ofrecen diferentes métodos para conectar emisores y receptores:

- **Chain of Responsibility**: Pasa una solicitud a lo largo de una cadena de receptores potenciales hasta que uno la maneja.
- **Command**: Establece conexiones unidireccionales entre emisores y receptores.
- **Mediator**: Elimina conexiones directas entre emisores y receptores, forzando la comunicación a través de un objeto mediador.
- **Observer**: Permite que los receptores se suscriban o se den de baja dinámicamente para recibir notificaciones.

Mediator y **Observer** pueden parecer similares, pero tienen diferencias clave. Mediator centraliza la comunicación entre componentes mediante un objeto mediador, mientras que Observer establece conexiones dinámicas unidireccionales entre objetos, donde algunos actúan como notificaciones de cambios.

A menudo, el **Mediator** puede ser implementado usando el patrón **Observer**, donde el mediador actúa como notificador y los componentes se suscriben a sus eventos. Sin embargo, también se puede usar Mediator de otras maneras, como vinculando permanentemente componentes al mismo objeto mediador.

En algunos casos, puedes tener un sistema distribuido donde cada componente actúa como un notificador, eliminando la necesidad de un mediador centralizado, lo que hace que todos los componentes actúen como observadores entre sí.