

Sumérgete en los patrones de diseño (\$6.69)



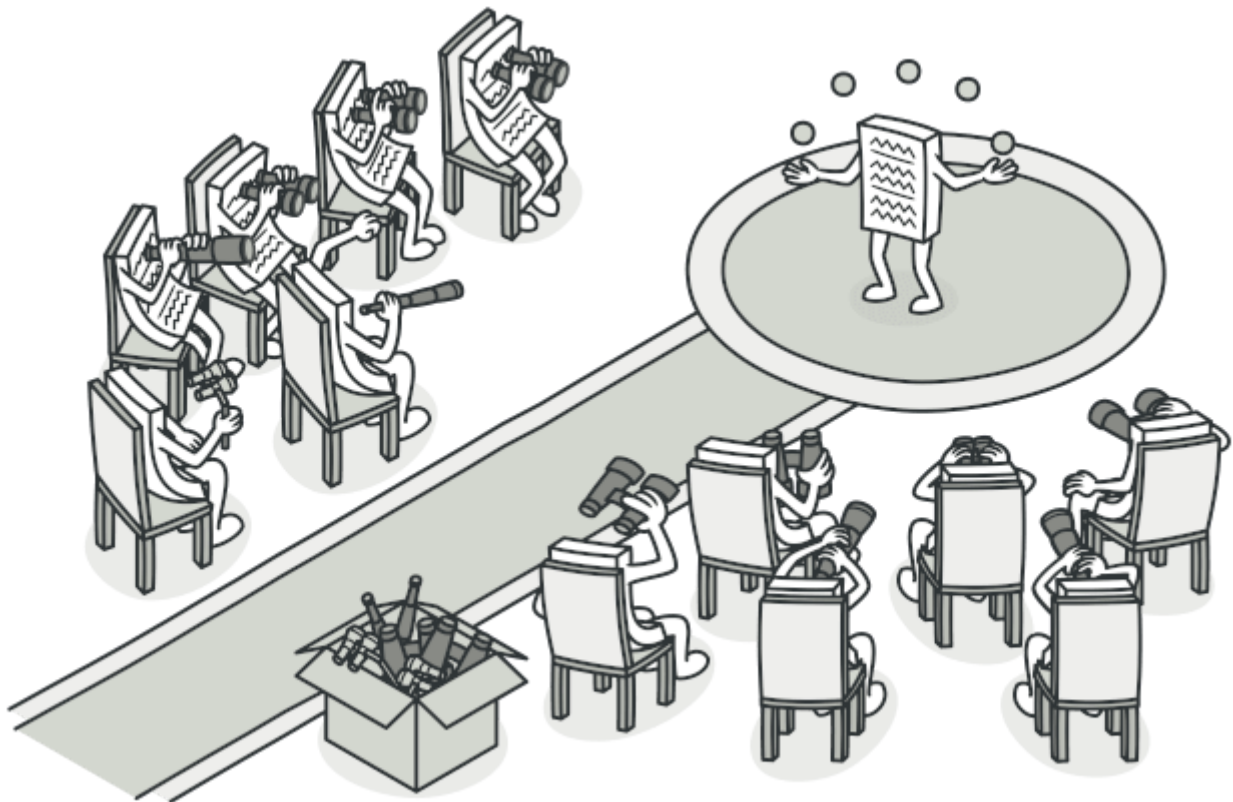
[🏠](#) / [Patrones de diseño](#) / [Patrones de comportamiento](#)

# Observer

**También llamado:** Observador, Publicación-Suscripción, Modelo-patrón, Event-Subscriber, Listener

## Propósito

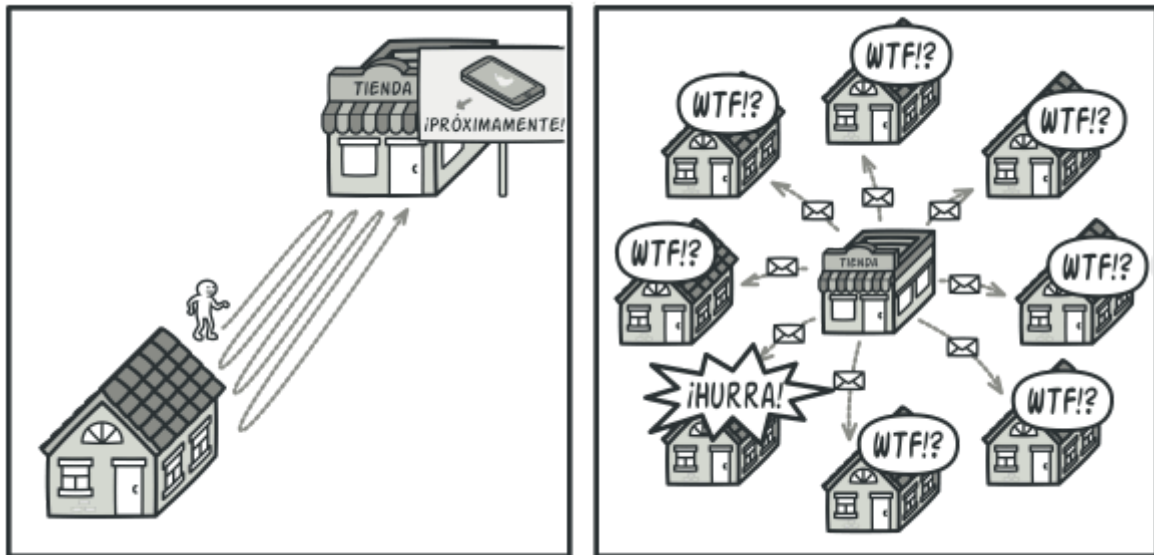
**Observer** es un patrón de diseño de comportamiento que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.



## 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.



*Visita a la tienda vs. envío de spam*

Por otro lado, la tienda podría enviar cientos de correos (lo cual se podría considerar spam) a todos los clientes cada vez que hay un nuevo producto disponible. Esto ahorraría a los clientes los interminables viajes a la tienda, pero, al mismo tiempo, molestaría a otros clientes que no están interesados en los nuevos productos.

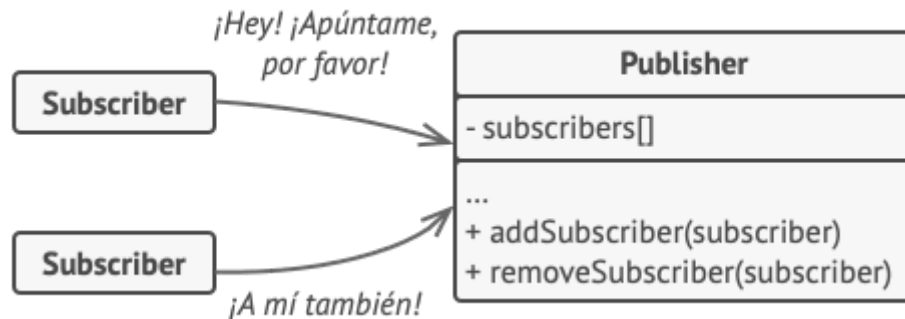
Parece que nos encontramos ante un conflicto. O el cliente pierde tiempo comprobando la disponibilidad del producto, o bien la tienda desperdicia recursos notificando a los clientes equivocados.

## 😊 Solución

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*.

El patrón Observer sugiere que añadas un mecanismo de suscripción a la clase notificadora para que los objetos individuales puedan suscribirse o cancelar su suscripción a un flujo de

eventos que proviene de esa notificadora. ¡No temas! No es tan complicado como parece. En realidad, este mecanismo consiste en: 1) un campo matriz para almacenar una lista de referencias a objetos suscriptores y 2) varios métodos públicos que permiten añadir suscriptores y eliminarlos de esa lista.



*Un mecanismo de suscripción permite a los objetos individuales suscribirse a notificaciones de eventos.*

Ahora, cuando le sucede un evento importante al notificador, recorre sus suscriptores y llama al método de notificación específico de sus objetos.

Las aplicaciones reales pueden tener decenas de clases suscriptoras diferentes interesadas en seguir los eventos de la misma clase notificadora. No querrás acoplar la notificadora a todas esas clases. Además, puede que no conozcas algunas de ellas de antemano si se supone que otras personas pueden utilizar tu clase notificadora.

Por eso es fundamental que todos los suscriptores implementen la misma interfaz y que el notificador únicamente se comuniquen 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.*

Si tu aplicación tiene varios tipos diferentes de notificadores y quieres hacer a tus suscriptores compatibles con todos ellos, puedes ir más allá y hacer que todos los notificadores sigan la misma interfaz. Esta interfaz sólo tendrá que describir algunos métodos de suscripción. La interfaz permitirá a los suscriptores observar los estados de los notificadores sin acoplarse a sus clases concretas.

## Analogía en el mundo real

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.

## Estructura

1. El **Notificador** envía eventos de interés a otros objetos. Esos eventos ocurren cuando el notificador cambia su estado o ejecuta algunos comportamientos. Los notificadores contienen una infraestructura de suscripción que permite a nuevos y antiguos suscriptores abandonar la lista.
2. Cuando sucede un nuevo evento, el notificador recorre la lista de suscripción e invoca el método de notificación declarado en la interfaz suscriptora en cada objeto suscriptor.
3. La interfaz **Suscriptora** declara la interfaz de notificación. En la mayoría de los casos, consiste en un único método `actualizar`. El método puede tener varios parámetros que permitan al notificador pasar algunos detalles del evento junto a la actualización.
4. Los **Suscriptores Concretos** realizan algunas acciones en respuesta a las notificaciones emitidas por el notificador. Todas estas clases deben implementar la misma interfaz de forma que el notificador no esté acoplado a clases concretas.
5. Normalmente, los suscriptores necesitan cierta información contextual para manejar correctamente la actualización. Por este motivo, a menudo los notificadores pasan cierta información de contexto como argumentos del método de notificación. El notificador puede pasarse a sí mismo como argumento, dejando que los suscriptores extraigan la información necesaria directamente.
6. El **Cliente** crea objetos tipo notificador y suscriptor por separado y después registra a los suscriptores para las 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.

*Notificar a objetos sobre eventos que suceden a otros objetos.*

La lista de suscriptores se compila dinámicamente: los objetos pueden empezar o parar de escuchar notificaciones durante el tiempo de ejecución, dependiendo del comportamiento que desees para tu aplicación.

En esta implementación, la clase editora no mantiene la lista de suscripción por sí misma. Delega este trabajo al objeto ayudante especial dedicado justo a eso. Puedes actualizar ese objeto para que sirva como despachador centralizado de eventos, dejando que cualquier objeto actúe como notificador.

Añadir nuevos suscriptores al programa no requiere cambios en clases notificadoras existentes, siempre y cuando trabajen con todos los suscriptores a través de la misma interfaz.

```
// La clase notificadora base incluye código de gestión de
// suscripciones y métodos de notificación.
class EventManager is
    private field listeners: hash map of event types and listeners

    method subscribe(eventType, listener) is
        listeners.add(eventType, listener)

    method unsubscribe(eventType, listener) is
        listeners.remove(eventType, listener)

    method notify(eventType, data) is
        foreach (listener in listeners.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 lógica de
// la suscripción con composición, como hicimos aquí.
class Editor is
    public field events: EventManager
    private field file: File
```

```

constructor Editor() is
    events = new EventManager()

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

method saveFile() is
    file.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.

interface EventListener is
    method update(filename)

// Los suscriptores concretos reaccionan a las actualizaciones
// emitidas por el notificador al que están unidos.
class LoggingListener implements EventListener is
    private field log: File
    private field message: string

    constructor LoggingListener(log_filename, message) is
        this.log = new File(log_filename)
        this.message = message

    method update(filename) is
        log.write(replace("%s", filename, message))

class EmailAlertsListener implements EventListener is
    private field email: string
    private field message: string

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

    method update(filename) is
        system.email(email, replace("%s", filename, message))

// Una aplicación puede configurar notificadores y suscriptores
// durante el tiempo de ejecución.
class Application is



```

```
method config() is
    editor = 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

-  Utiliza el patrón Observer 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.
-  Puedes experimentar este problema a menudo al trabajar con clases de la interfaz gráfica de usuario. Por ejemplo, si creaste clases personalizadas de botón y quieres permitir al cliente colgar código cliente de tus botones para que se active cuando un usuario pulse un botón.

El patrón Observer permite que cualquier objeto que implemente la interfaz suscriptor pueda suscribirse a notificaciones de eventos en objetos notificadores. Puedes añadir el mecanismo de suscripción a tus botones, permitiendo a los clientes acoplar su código personalizado a través de clases suscriptoras personalizadas.

---

-  Utiliza el patrón cuando algunos objetos de tu aplicación deban observar a otros, pero sólo durante un tiempo limitado o en casos específicos.
-  La lista de suscripción es dinámica, por lo que los suscriptores pueden unirse o abandonar la lista cuando lo deseen.

## Cómo implementarlo

1. Repasa tu 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. Recuerda que los notificadores deben trabajar con suscriptores únicamente a través de la interfaz suscriptora.
4. Decide dónde colocar la lista de suscripción y la implementación de métodos de suscripción. Normalmente, este código tiene el mismo aspecto para todos los tipos de notificadores, por lo que el lugar obvio para colocarlo es en una clase abstracta derivada directamente de la interfaz notificadora. Los notificadores concretos extienden esa clase, heredando el comportamiento de suscripción.  
  
Sin embargo, 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.
5. Crea clases notificadoras concretas. Cada vez que suceda algo importante dentro de una notificadora, deberá notificar a todos sus suscriptores.
6. Implementa los métodos de notificación de actualizaciones en clases suscriptoras concretas. La mayoría de las suscriptoras necesitarán cierta información de contexto sobre el evento, que puede pasarse como argumento del método de notificación.  
  
Pero hay otra opción. Al recibir una notificación, el suscriptor puede extraer la información directamente de ella. En este caso, el notificador debe pasarse a sí mismo a través del método de actualización. La opción menos flexible es vincular un notificador con el suscriptor de forma permanente a través del constructor.
7. El cliente debe crear todos los suscriptores necesarios y registrarlos con los notificadores adecuados.

## 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.



## ⇔ Relaciones con otros patrones

- Chain of Responsibility, Command, Mediator y Observer abordan distintas formas de conectar emisores y receptores de solicitudes:
  - *Chain of Responsibility* pasa una solicitud secuencialmente a lo largo de una cadena dinámica de receptores potenciales hasta que uno de ellos la gestiona.
  - *Command* establece conexiones unidireccionales entre emisores y receptores.
  - *Mediator* elimina las conexiones directas entre emisores y receptores, forzándolos a comunicarse indirectamente a través de un objeto mediador.
  - *Observer* permite a los receptores suscribirse o darse de baja dinámicamente a la recepción de solicitudes.
- La diferencia entre Mediator y Observer a menudo resulta difusa. En la mayoría de los casos, puedes implementar uno de estos dos patrones; pero en ocasiones puedes aplicarlos ambos a la vez. Veamos cómo podemos hacerlo.

La meta principal del patrón *Mediator* consiste en eliminar las dependencias mutuas entre un grupo de componentes del sistema. En su lugar, estos componentes se vuelven dependientes de un único objeto mediador. La meta del patrón *Observer* es establecer conexiones dinámicas de un único sentido entre objetos, donde algunos objetos actúan como subordinados de otros.

Hay una implementación popular del patrón *Mediator* que se basa en el *Observer*. El objeto mediador juega el papel de notificador, y los componentes actúan como suscriptores que se suscriben o se dan de baja de los eventos del mediador. Cuando se implementa el *Mediator* de esta forma, puede asemejarse mucho al *Observer*.

Cuando te sientas confundido, recuerda que puedes implementar el patrón *Mediator* de otras maneras. Por ejemplo, puedes vincular permanentemente todos los componentes al mismo objeto mediador. Esta implementación no se parece al *Observer*, pero aún así será una instancia del patrón *Mediator*.

Ahora, imagina un programa en el que todos los componentes se hayan convertido en notificadores, permitiendo conexiones dinámicas entre sí. No hay un objeto mediador centralizado, tan solo un grupo distribuido de observadores.

## </> Ejemplos de código

