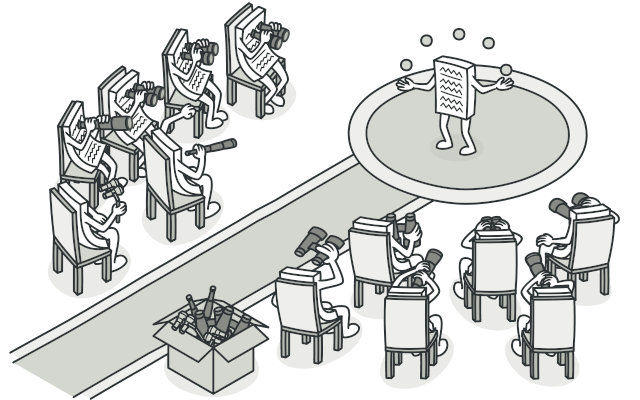
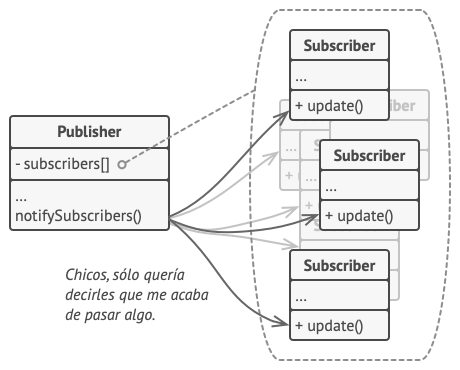
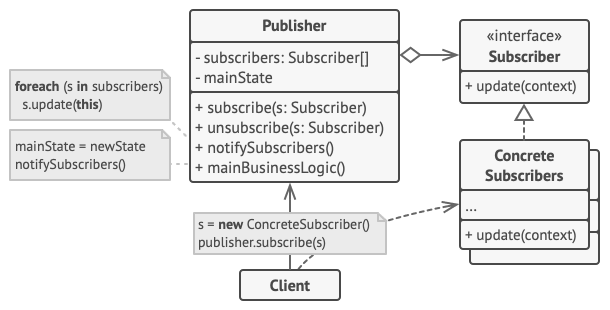
**Patrón Observer**



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.





* 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.
* 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.
* 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.
* 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.
* 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.
* El Cliente crea objetos tipo notificador y suscriptor por separado y después registra a los suscriptores para las actualizaciones del notificador.

## Patrón Observer: Desarrollo Detallado

## **1. Explicación del Patrón de Arquitectura**

El patrón Observer es un patrón de diseño de comportamiento que define una dependencia uno-a-muchos entre objetos. Cuando un objeto (el sujeto) cambia su estado, todos sus dependientes (observadores) son notificados y actualizados automáticamente. Esto permite un acoplamiento débil entre el sujeto y los observadores, promoviendo la reutilización y la flexibilidad del código.

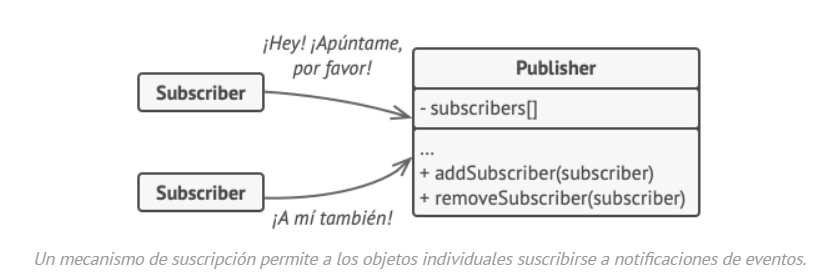
Ejemplo: Imagina un termómetro (sujeto) que notifica a diferentes pantallas (observadores) la temperatura actual. Cuando la temperatura cambia, el termómetro notifica a todas las pantallas para que se actualicen.

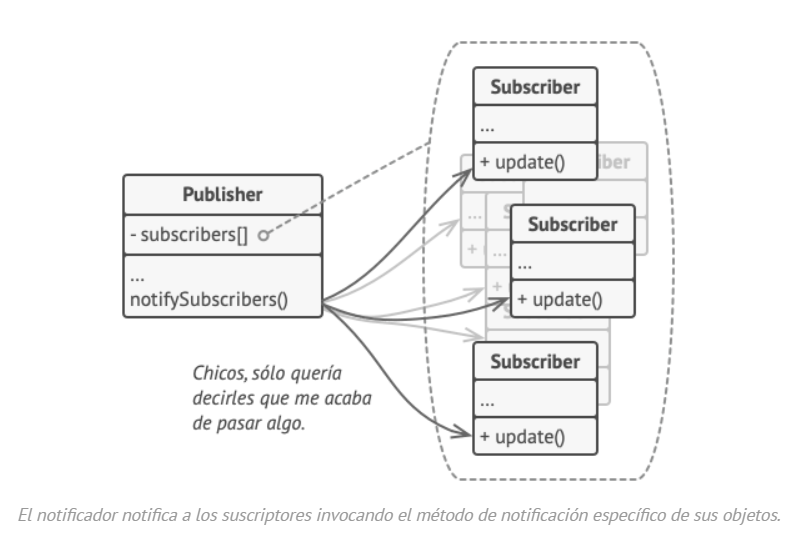
## **2. Diagrama Representativo**

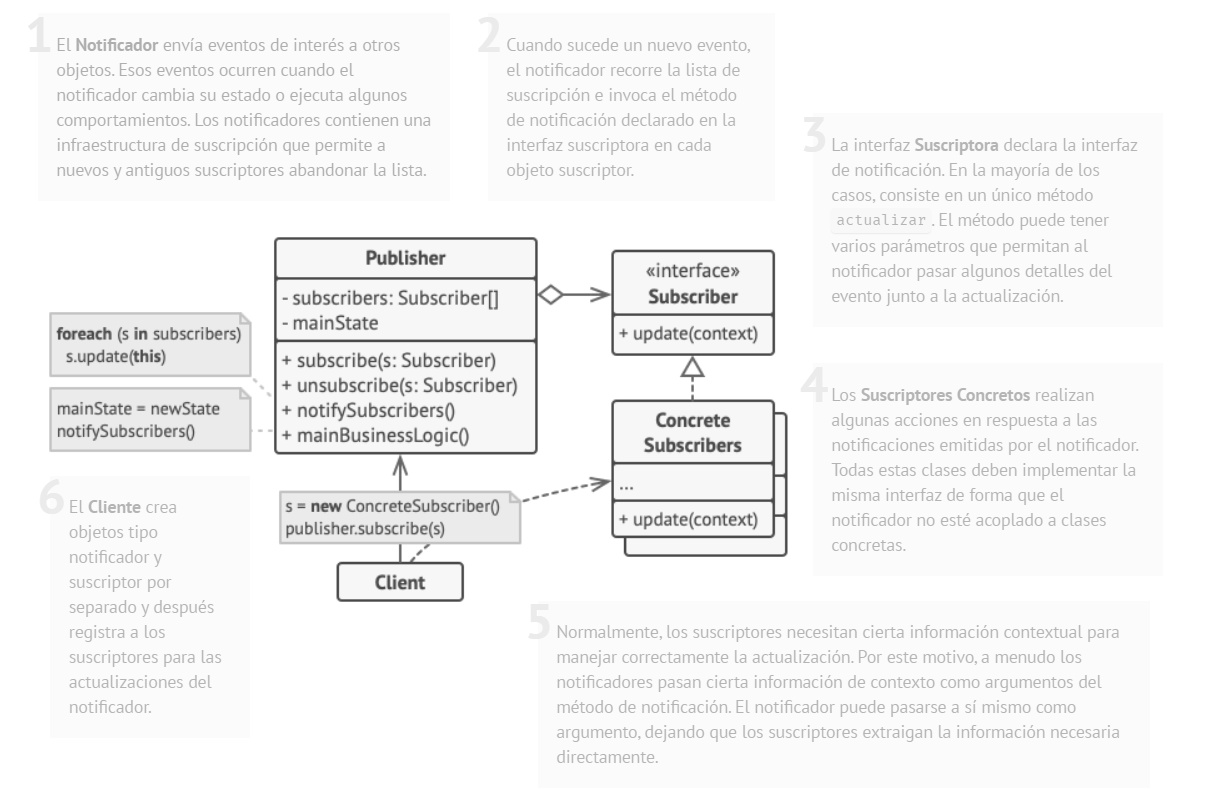
(Considera incluir un diagrama UML que muestre las clases Sujeto, Observador, Sujeto Concreto, y Observador Concreto, con las relaciones correspondientes).

En esencia, el diagrama mostraría:

* Sujeto: Interfaz o clase abstracta con métodos para adjuntar/desadjuntar observadores y para notificar.
* Observador: Interfaz o clase abstracta con un método actualizar() que se llama cuando el sujeto cambia de estado.
* Sujeto Concreto: Implementa el sujeto, mantiene el estado y la lista de observadores, y notifica a los observadores cuando el estado cambia.
* Observador Concreto: Implementa el observador y realiza acciones específicas cuando es notificado por el sujeto.







## **3. Código de Implementación en C# (Ejemplo)**

csharp

*// Interfaz Observador*

public interface IObserver

{

void Update(ISubject subject);

}

*// Interfaz Sujeto*

public interface ISubject

{

void Attach(IObserver observer);

void Detach(IObserver observer);

void Notify();

string State { get; set; }

}

*// Sujeto Concreto*

public class ConcreteSubject : ISubject

{

private List<IObserver> \_observers = new List<IObserver>();

private string \_state;

public string State

{

get { return \_state; }

set { \_state = value; Notify(); }

}

public void Attach(IObserver observer)

{

\_observers.Add(observer);

}

public void Detach(IObserver observer)

{

\_observers.Remove(observer);

}

public void Notify()

{

foreach (var observer in \_observers)

{

observer.Update(this);

}

}

}

*// Observador Concreto*

public class ConcreteObserver : IObserver

{

private string \_name;

public ConcreteObserver(string name)

{

\_name = name;

}

public void Update(ISubject subject)

{

Console.WriteLine($"Observer {\_name} ha sido notificado. Nuevo estado: {subject.State}");

}

}

*// Uso*

public class Program

{

public static void Main(string[] args)

{

ConcreteSubject subject = new ConcreteSubject();

ConcreteObserver observer1 = new ConcreteObserver("Observador 1");

ConcreteObserver observer2 = new ConcreteObserver("Observador 2");

subject.Attach(observer1);

subject.Attach(observer2);

subject.State = "Estado cambiado!";

}

}

## **4. Ventajas del Patrón**

* Acoplamiento Débil: El sujeto y los observadores no están fuertemente acoplados, lo que permite cambios independientes.
* Flexibilidad: Se pueden añadir nuevos observadores sin modificar el sujeto.
* Reutilización: El mismo sujeto puede ser observado por múltiples observadores diferentes.
* Abierto/Cerrado: Permite añadir nuevos observadores sin modificar la clase del sujeto.

## **5. Desventajas del Patrón**

* Notificaciones Aleatorias: El orden de notificación de los observadores no está garantizado.
* Potencial de Cascadas: Un gran número de observadores pueden causar problemas de rendimiento o comportamientos inesperados si una actualización desencadena otra.
* Dificultad de Depuración: Las actualizaciones en cascada pueden ser difíciles de rastrear y depurar.
* Posibles Fugas de Memoria: Si no se gestionan correctamente, los observadores desuscriptos pueden permanecer en la lista, causando fugas de memoria.



## 

## **6. Casos de Uso en la Vida Real**

* Interfaces Gráficas de Usuario (GUI): Los componentes de la GUI (botones, campos de texto) actúan como observadores, reaccionando a eventos generados por el usuario (el sujeto).
* Hojas de Cálculo: Cuando una celda cambia, las celdas dependientes se actualizan automáticamente.
* Sistemas de Noticias: Los suscriptores (observadores) reciben actualizaciones cuando se publican nuevas noticias (sujeto).
* Middleware de Mensajería: En sistemas distribuidos, un objeto puede publicar mensajes (sujeto) y varios servicios (observadores) pueden suscribirse para recibir estos mensajes.
* Model-View-Controller (MVC): El patrón Observer es fundamental en el patrón MVC, donde el Modelo (sujeto) notifica a las Vistas (observadores) sobre los cambios en los datos.

## **7. Fuentes Consultadas y Referencias**

* Refactoring.Guru: <https://refactoring.guru/es/design-patterns/observer>
* "Design Patterns: Elements of Reusable Object-Oriented Software" (GoF Book)
* Microsoft Documentation: Para la implementación en C#.