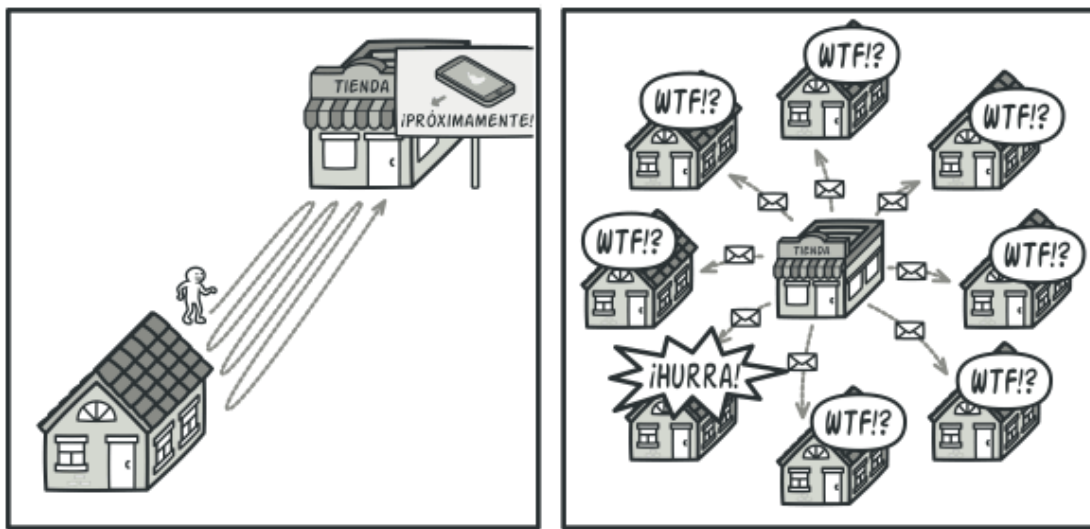


PATRÓN OBSERVER

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.

Analogía en el mundo real



Suscripciones a revistas y periódicos.

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.

Objetivo: «Definir una dependencia uno-a-muchos entre objetos de forma de que, cuando el estado de uno de ellos cambia, todos los objetos dependientes son notificados y actualizados de forma automática».

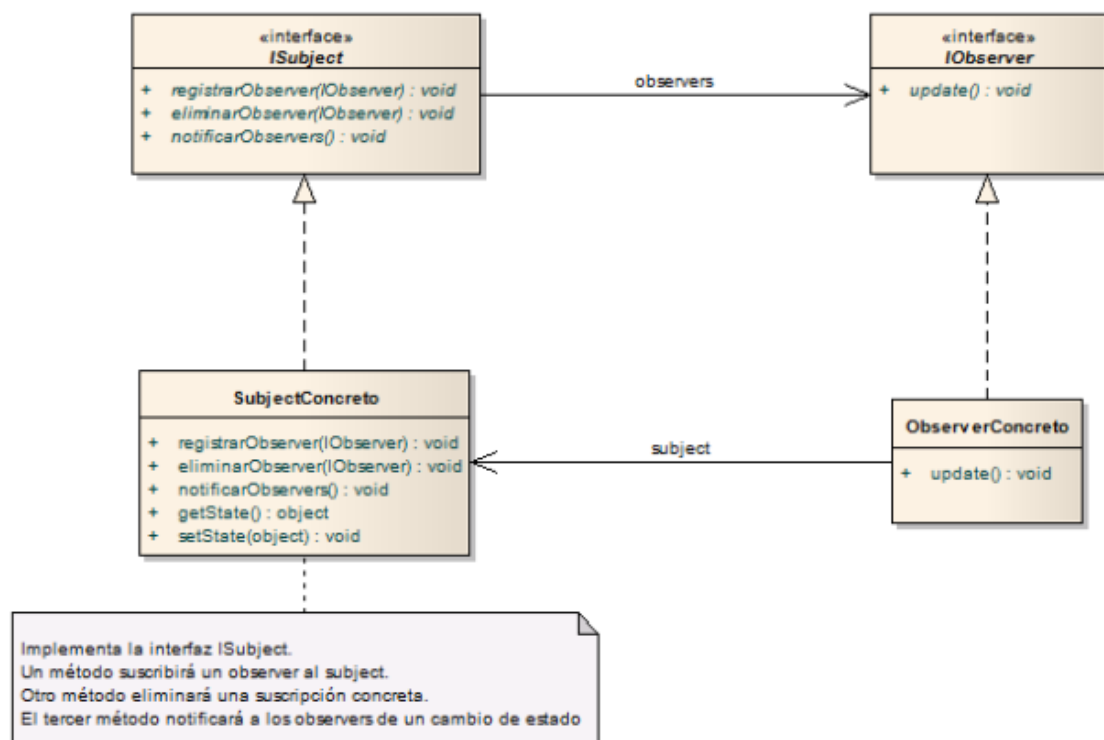
Podemos afirmar que el patrón *Observer* es uno de los más importantes (y utilizados) de todos los patrones de diseño vistos hasta el momento. Su filosofía es simple: un objeto, denominado *sujeto* (*Subject*) posee un estado. Cuando su estado cambia, es capaz de «avisar» a sus *subscriptores* (*Observers*) de este cambio de estado. De este modo, los objetos suscritos al objeto no tienen que preocuparse de cuándo se produce un cambio de estado: éste se encargará de informar de forma activa a todos aquellos objetos que hayan decidido suscribirse.

Este tipo de suscripción puede ser de dos tipos:

- **Suscripción *push*:** el objeto informa a sus suscriptores con sus valores, tan pronto como su estado cambie.
- **Suscripción *pull*:** el objeto es interrogado por sus suscriptores si su estado ha cambiado, desde la última vez que se tanteó.

Este esquema respeta al máximo el principio de **bajo acoplamiento**. El *Subject* y el *Observer* pueden interactuar entre ellos, pero apenas tienen conocimiento del uno sobre el otro. Dado que **hemos basado el diseño en abstracciones, no en concreciones**, no será necesario modificar el *Subject* para añadir nuevos *Observers* (basta con que implemente la interfaz *IObserver*). Del mismo modo, un *Observer* podrá suscribirse a más de un *Subject* si éste implementa la interfaz *ISubject*.

Ampliando un poco más, ¿cómo funciona realmente el patrón? Explicado de forma simple, podemos resumirlo de la siguiente manera:



- *SubjectConcreto* implementa la interfaz *ISubject*. Esta clase tendrá una funcionalidad propia, un estado (por ejemplo, una serie de variables) y una lista interna de objetos que implementan la interfaz *IObserver*. Esta lista se utilizará para realizar la «suscripción» a los cambios de estado, invocando el método *IObserver.update()* de cada objeto dentro de esta lista cada vez que se produzca un cambio de estado. Además, esta clase implementará los métodos de *ISubject*:
 - *registrarObserver(IObserver o)*: agrega el *observer* «o» pasado como parámetro a la lista de objetos que serán notificados.
 - *eliminarObserver(Observer o)*: elimina el *observer* «o» pasado como parámetro de la lista de objetos que serán notificados.
 - *notificarObservers()*: recorre la lista de *observers*, invocando el método *update()* de cada uno de ellos, provocando que se realice una

acción determinada cuando el estado del *Subject* cambie. Normalmente este método será invocado en el momento en el que el estado del *Subject* cambie (por ejemplo, al final del método *setState()*).

El método *update()* será implementado por cada *Observer* concreto, y se encargará de realizar las operaciones que el objeto suscrito quiera realizar cuando se entere de que algo ha cambiado dentro del *Subject*.

Implementando el patrón Observer

Realizaremos una implementación de este patrón basándonos en un sistema de sensores que realice una medición de los niveles de agua y aceite, así como de la presión de los neumáticos. Este sistema permitirá que los *observers* puedan suscribirse a los cambios en los valores de estos niveles, realizando las operaciones que estimen oportunas.

Comenzaremos creando la interfaz *IObserver* que incluirá el método *update* que tendrán que implementar todas aquellas clases que deseen recibir notificaciones. Este método recibirá un parámetro que contendrá información sobre la actualización. Para hacerla lo más genérica posible, haremos que sea de tipo *Object*, realizando el *casting* correspondiente a la hora de implementar la clase concreta.

1. Vea el código de la interfaz [IObserver.cs](#)

Lo siguiente que haremos será codificar una interfaz que exponga los métodos propios de un notificador:

- Un método para registrar un *observer*, que aceptará un *IObserver* como argumento.
- Un método para eliminar la suscripción de un *observer*, que también aceptará un *IObserver* como argumento.
- Un método para realizar la notificación a todos los *observers* que se encuentren suscritos al *Subject*.

2. Vea el código de la interfaz [ISubject.cs](#)

A continuación, codificaremos el *Subject* en sí:

- una clase que implementará la interfaz *ISubject* y que incluirá los métodos declarados en ésta. Además, incluirá una serie de variables que almacenarán

- los distintos niveles (aceite, agua y presión de los neumáticos) junto a sus respectivas *Properties*, que permitirán modificar sus valores.
- En el método *set* de estas *Properties* será donde se realice la invocación del método *NotificarObservers()*, siempre y cuando el valor cambie, es decir, **se comprueba si el nuevo valor es igual al anterior**, y en caso contrario, se actualiza el estado y se notifica a todos aquellos *Observers* que se encuentren suscritos.
 - El constructor de la clase recibirá como parámetros los valores iniciales de los niveles de aceite, agua y presión de los neumáticos, además de instanciar el *ICollection* que almacenará los *Observers* suscritos a las notificaciones.
 - Los métodos *RegistrarObserver* y *EliminarObserver* serán los encargados de agregar y eliminar suscriptores a la lista (previa comprobación de si éstos están ya en la lista o no).
 - El método *NotificarObservers()* será tan sencillo como un ciclo *foreach* que recorra todos los *IObserver* suscritos a las notificaciones, invocando su método *update()* pasándole el estado del objeto como parámetro.

Previamente mencionamos que existen dos versiones del patrón:

- Una versión *push*, que es precisamente la que definimos aquí: el propio *Subject* inyecta el estado en los *Observers*, pasándolo como parámetro al método *Update()*.
- Una versión *pop*, que como alternativa ofrece simplemente informar al *Observer* que se ha producido un cambio de estado, siendo responsabilidad de éste solicitar al *Subject* los nuevos valores.

3. Vea el código de la clase MedidorSensores.cs

El último paso será la creación de las clases que implementen la interfaz *IObserver*. Crearemos dos clases, una de ellas simbolizará el *display* del vehículo, es decir, la pantalla de cristal líquido que muestra la información al conductor, además de la velocidad, revoluciones, llenado del tanque del combustible...

Este *Observer* mostrará **siempre** los cambios producidos en los niveles, ya que su misión será la de informar al usuario los niveles actuales en todo momento. Es el tipo más simple de *Observer*, ya que se limita a presentar la información recibida del *Subject*.

4. Vea el código de la clase ObserverDisplay.cs

Como segundo *Observer* codificaremos una clase *ObserverAlerta* que, en lugar de informar siempre de los niveles actuales, sea capaz de enviar una alerta (sonora, remota, visual o de cualquier otro tipo) en caso de que los niveles superen ciertos

umbrales superiores o inferiores. En caso contrario, no realizarán ninguna acción. Esto servirá para mostrar que un *Observer* realizará cualquier tipo de operación con estos datos, y también la suscripción de más de uno de estos elementos.

5. Vea el código de la clase ObserverAlerta.cs

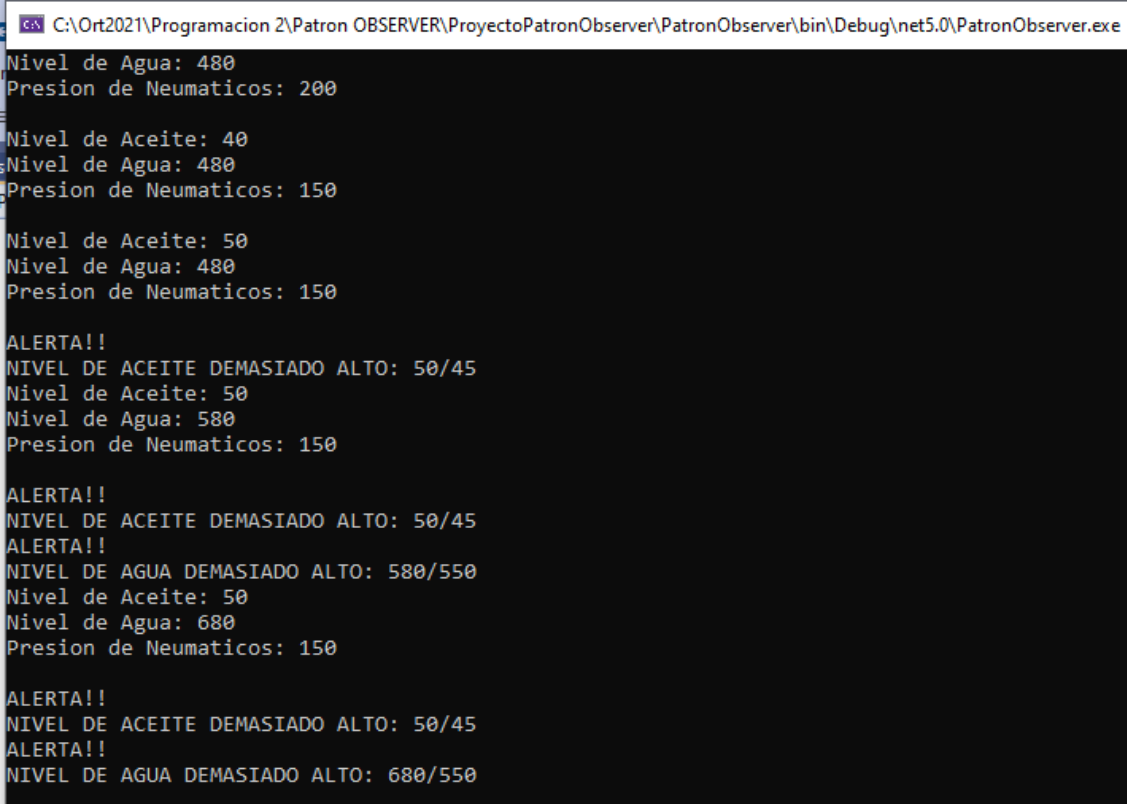
Finalmente, vemos el código necesario para mostrar el funcionamiento del patrón.

Comenzaremos instanciando un *Subject* con sus niveles iniciales, para luego instanciar dos *Observers* (uno de display y otro de alerta) que recibirán como parámetro el *Subject* al cual se suscribirán de forma automática.

A continuación alteraremos los valores del medidor de sensores y comprobaremos cómo los *Observers* son informados y actúan en consecuencia, mostrando **siempre** los cambios en los valores en el caso del display y mostrando únicamente las alertas cuando los valores se encuentran fuera de los límites para el caso del observer *ObserverAlerta*.

6. Vea el código del Program.cs PRIMERA PARTE

Como vemos, siempre que se produce un cambio de estado, el display recibe la notificación y actúa en consecuencia. Cuando esta notificación llega a la alerta y los valores se encuentran fuera de los límites aceptables, además se emitirá una alerta.



```
C:\Ort2021\Programacion 2\Patron OBSERVER\ProyectoPatronObserver\PatronObserver\bin\Debug\net5.0\PatronObserver.exe
Nivel de Agua: 480
Presion de Neumaticos: 200

Nivel de Aceite: 40
Nivel de Agua: 480
Presion de Neumaticos: 150

Nivel de Aceite: 50
Nivel de Agua: 480
Presion de Neumaticos: 150

ALERTA!!
NIVEL DE ACEITE DEMASIADO ALTO: 50/45
Nivel de Aceite: 50
Nivel de Agua: 580
Presion de Neumaticos: 150

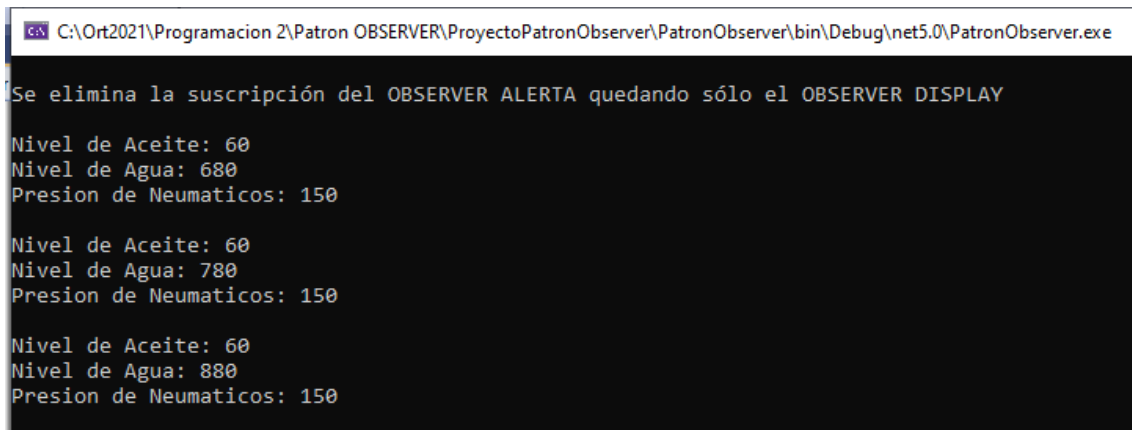
ALERTA!!
NIVEL DE ACEITE DEMASIADO ALTO: 50/45
ALERTA!!
NIVEL DE AGUA DEMASIADO ALTO: 580/550
Nivel de Aceite: 50
Nivel de Agua: 680
Presion de Neumaticos: 150

ALERTA!!
NIVEL DE ACEITE DEMASIADO ALTO: 50/45
ALERTA!!
NIVEL DE AGUA DEMASIADO ALTO: 680/550
```

Como alternativa, podemos probar a eliminar la suscripción a las alertas antes de que éstas rebasen los límites:

7. Vea el código del Program.cs SEGUNDA PARTE

En este caso, únicamente el display realizará su trabajo, ya que hemos decidido eliminar la suscripción de *ObserverAlerta* a lo largo del programa, lo que implicará que este objeto no reciba a partir de ese momento las notificaciones del *Subject*.



```
C:\Ort2021\Programacion 2\Patron OBSERVER\ProyectoPatronObserver\PatronObserver\bin\Debug\net5.0\PatronObserver.exe
Se elimina la suscripción del OBSERVER ALERTA quedando sólo el OBSERVER DISPLAY
Nivel de Aceite: 60
Nivel de Agua: 680
Presion de Neumaticos: 150
Nivel de Aceite: 60
Nivel de Agua: 780
Presion de Neumaticos: 150
Nivel de Aceite: 60
Nivel de Agua: 880
Presion de Neumaticos: 150
```

¿Cuándo utilizar este patrón? Ejemplos reales

Este patrón es útil en multitud de supuestos, pero el principal será cuando la modificación del estado de un objeto deba provocar cambio en otros objetos sin que sea necesario conocer nada acerca de la cantidad ni el tipo de dichos objetos.

Este patrón es la base de los **eventos** de la mayor parte de las interfaces de usuario: los *Event Handlers* se corresponderían con los *Observers* que esperan la notificación de un cambio de estado. Así, en un botón, definir un método para el evento *OnClick* hará que el control *Button* cambie su estado (por ejemplo, de *no pulsado* a *pulsado*), notificando a todos aquellos objetos que estén suscritos a él. De este modo, el *Event Handler* será notificado de este cambio de estado y ejecutará el código que haya sido programado para responder a dicho cambio de estado (por ejemplo, mostrar un mensaje cuando se pulsa un botón, o lo que es lo mismo, cuando el botón pasa de *no pulsado* a *pulsado*).

El patrón *Observer* también es la base del **enlace de datos** (*data binding*), por ejemplo, en un control *GridView*, que al obtener los datos de una fila de datos se encargará de dibujar la plantilla previamente configurada para mostrarle los datos al usuario.

Este patrón *Observer*, es la base de uno de los patrones compuestos más utilizados hoy en día: el **patrón MVC o Modelo Vista Controlador**. El papel del patrón *Observer* en este patrón compuesto (el patrón MVC no es un patrón en sí, sino que es un patrón

compuesto de varios patrones, al igual que una clase es una estructura de datos formada por otras estructuras de datos), será actuar de núcleo del mismo implementándose en el modelo. De este modo, los objetos interesados se mantienen actualizados cuando se produce un cambio de estado, haciendo que tanto la vista como el controlador estén débilmente acoplados con el modelo. MVC hace uso de otros dos patrones ya vistos anteriormente: Composite (vista) y Strategy (controlador).