

# 14 observer

## Diseño del Software

Grado en Ingeniería Informática del Software

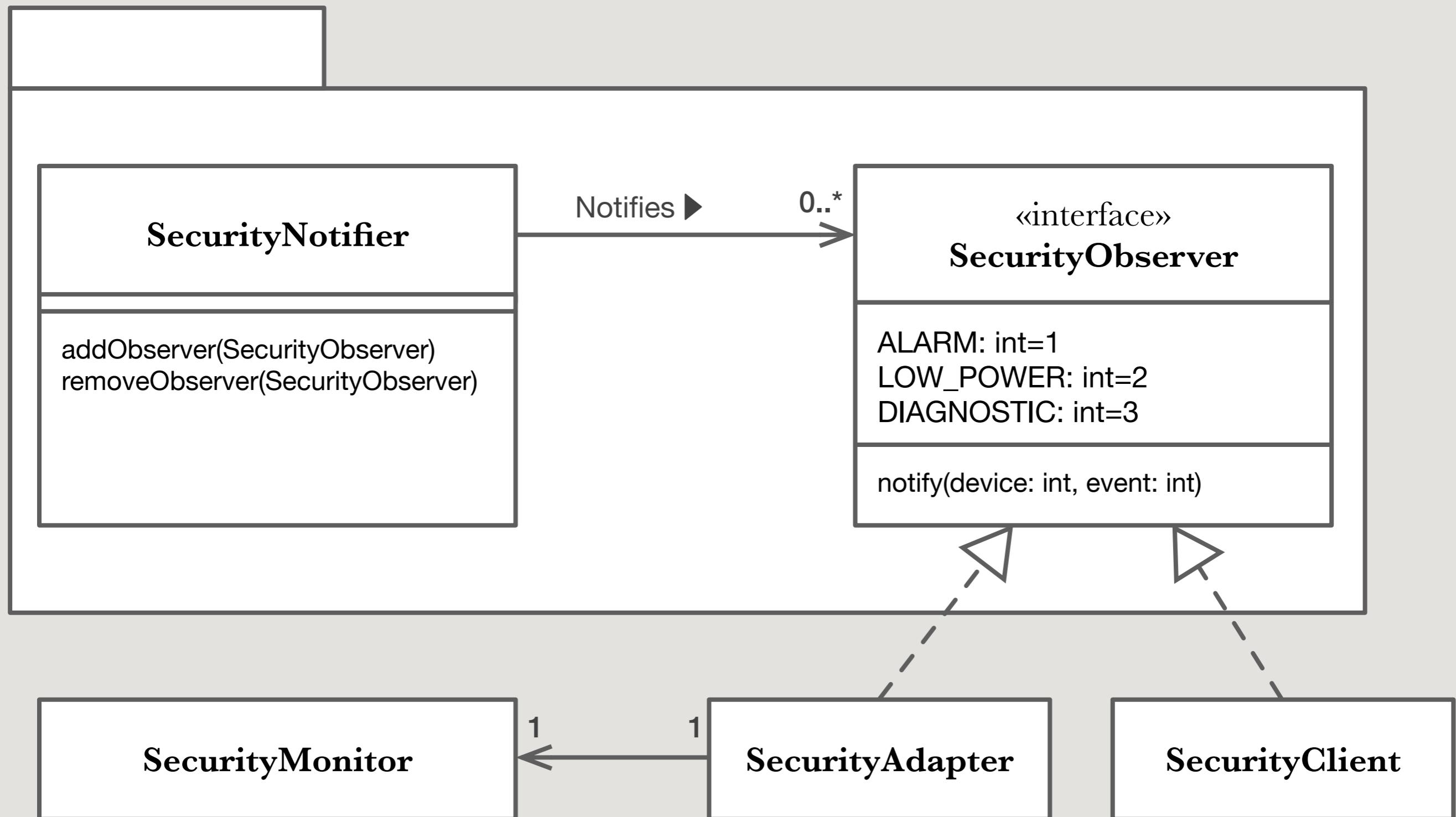
2024-2025

Supongamos que trabajamos para una empresa que fabrica detectores de humo, sensores de movimiento y otros dispositivos de seguridad.

Queremos que las compañías que desarrollan aplicaciones de domótica los integren en sus sistemas.

Nuestra tarea es crear una API fácil de usar que permita a los futuros clientes trabajar con nuestros dispositivos sin obligarles a alterar la arquitectura de su software actual.

¿Cómo sabemos a qué objetos avisar?



Un objeto **SecurityNotifier** pasa una notificación a un objeto **SecurityObserver** llamando a su método `notify`. Los parámetros que pasa a su método `notify` son un número que identifica de forma exclusiva el dispositivo de seguridad del que procede la notificación original y un número que especifica el tipo de notificación.

El resto de clases del diagrama no forman parte de la API. Son clases que ya existían o que se añadieron al software de supervisión de los clientes potenciales.

La clase indicada en el diagrama como **SecurityClient** corresponde a cualquier clase que un cliente añada a su software de supervisión que implemente la interfaz **SecurityObserver** para procesar las notificaciones de un objeto **SecurityNotifier**.

La clase indicada en el diagrama como **SecurityMonitor** corresponde a una clase existente en el software de monitorización de un cliente que no implementa la interfaz **SecurityObserver** pero sí tiene un método que debe ser llamado para procesar las notificaciones de los dispositivos de seguridad. El cliente puede escribir una clase adaptadora que implemente la interfaz **SecurityObserver** para que su método `notify` llame al método apropiado de la clase **SecurityMonitor**, sin modificar la clase.

# Observer

# **Patrón de comportamiento de objetos**

*Define una dependencia uno-a-muchos entre objetos, de modo que cuando un objeto cambia su estado todos los demás objetos dependientes se modifican y actualizan automáticamente.*

También conocido  
como

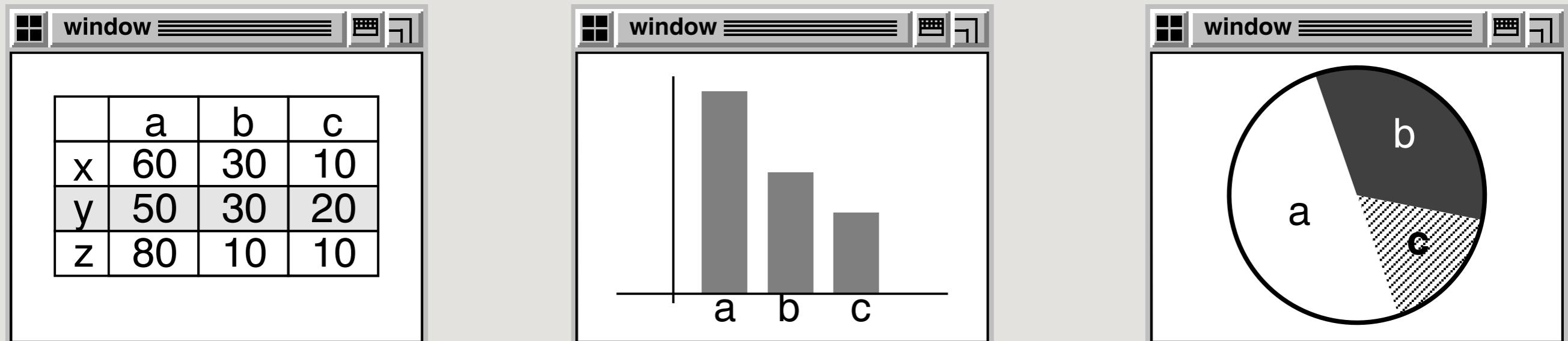
Dependientes, Publicar-Suscribir

# Motivación

Muchas veces un efecto lateral de partir un sistema en una colección de objetos colaborativos es que necesitamos mantener la consistencia entre los objetos relacionados.

**Pero no queremos que al hacerlo las clases pasen a estar fuertemente acopladas, porque eso reduciría su reutilización.**

## views

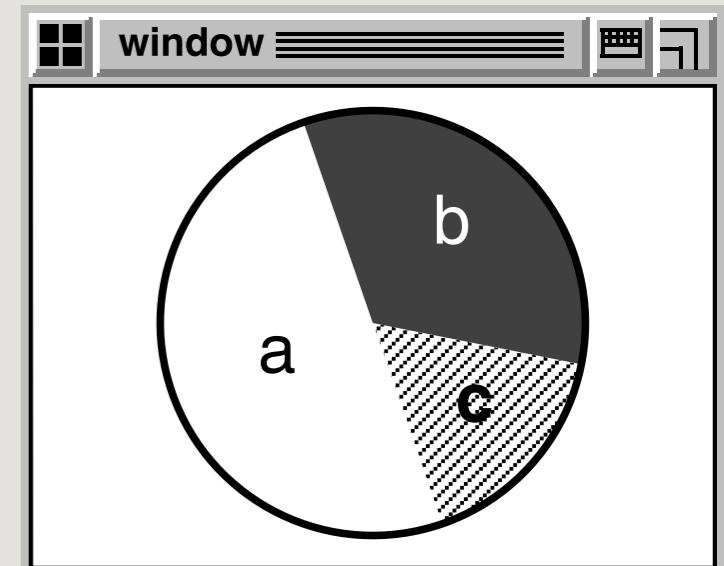
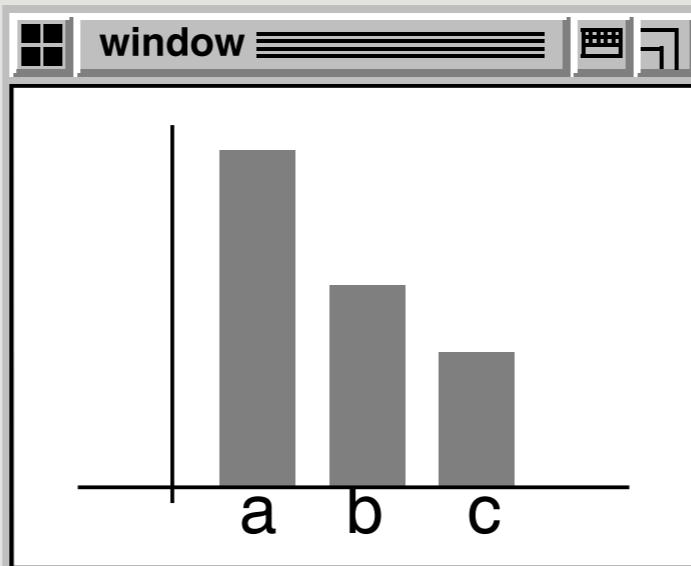


## model

→ change modification  
..... → request, modification

## observers

	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10



a = 50%  
b = 30%  
c = 20%

subject

→ change modification  
..... → request, modification

**Los elementos clave de este patrón son el sujeto observado y el observador.**

Un sujeto puede tener cualquier número de observadores dependientes. Todos los observadores reciben una notificación cada vez que el sujeto sufre un cambio de estado. Como respuesta, cada observador consultará al sujeto para llevar a cabo alguna acción y sincronizar su estado con el de aquél.

A este tipo de interacción también se la conoce como **publicar-suscribir**.

El sujeto es quien publica las notificaciones. Envía dichas notificaciones sin tener que saber quiénes son sus observadores. Se puede suscribir cualquier número de observadores para recibir notificaciones.

# Aplicabilidad

Úsese el patrón Observer en cualquiera de las situaciones siguientes

**Una abstracción tiene dos aspectos, uno de los cuales depende del otro.**

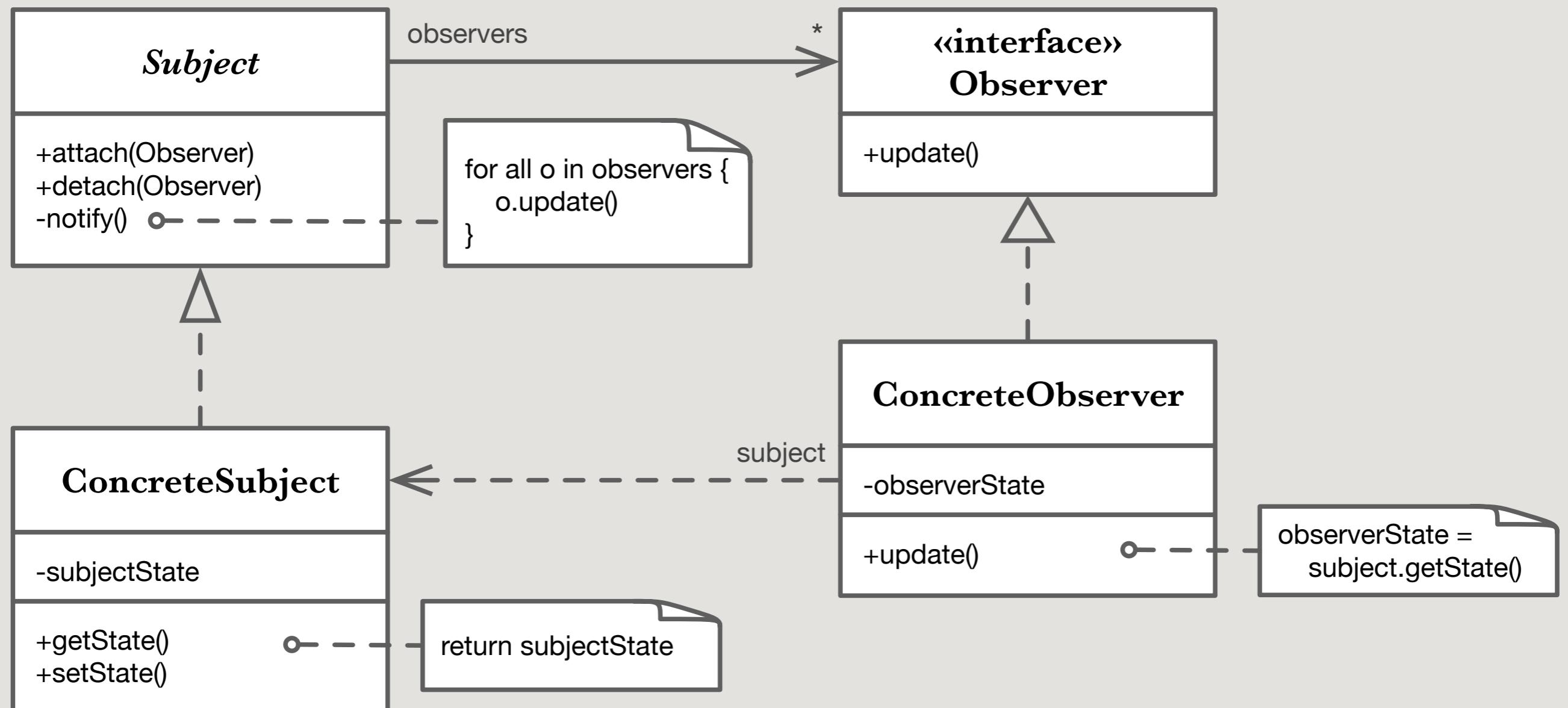
Encapsular estos aspectos en objetos separados permite que los objetos varíen (y puedan ser reutilizados) de forma independiente.

**Un cambio en un objeto requiere  
que cambien otros (y no sabemos  
cuántos).**

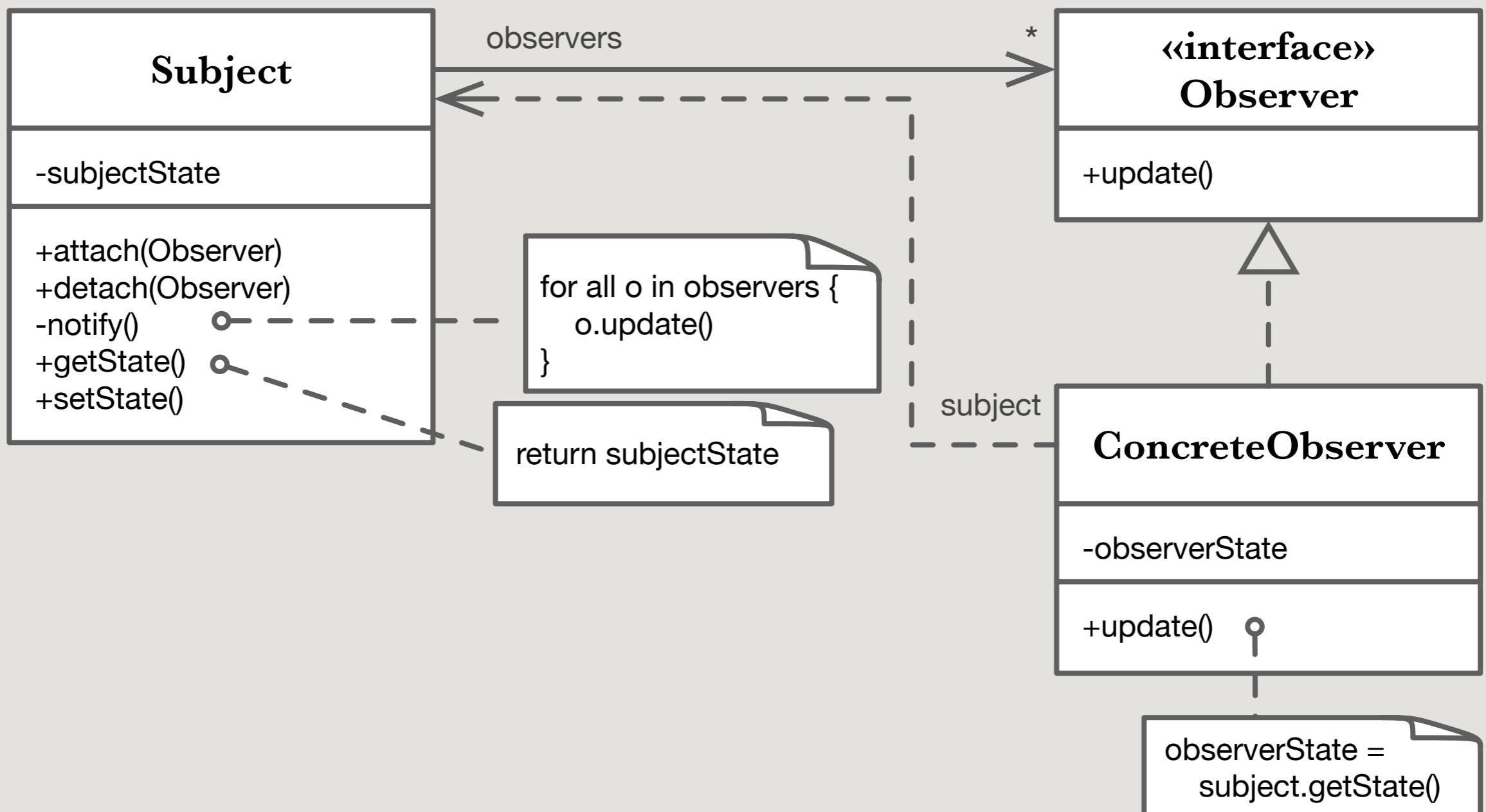
**Un objeto necesita notificar a otros cambios en su estado sin hacer presunciones sobre quiénes son dichos objetos**

Es decir, cuando no queremos que estén fuertemente acoplados.

# Estructura



La estructura del patrón de diseño Observer, tal como aparece en el GoF



La estructura del patrón Observer, como es más común en la práctica (sin una interfaz separada o clase abstracta específica para el sujeto, sino incorporando esas responsabilidades directamente en el objeto observado).

# Participantes

# Subject

**Conoce a sus observadores.**

Puede haber un número indeterminado de ellos.

**Proporciona una interfaz para que se suscriban o  
se borren los objetos observadores.**

# Observer

Define una interfaz para actualizar los objetos que deben ser notificados de cambios en el objeto Subject.

# ConcreteSubject

Guarda el estado de interés para los observadores concretos.

Envía una notificación a sus observadores cada vez que cambia su estado.

# ConcreteObserver

Implementa la interfaz Observer para ser notificado de los cambios en el estado del sujeto.

Mantiene una referencia al sujeto observado.

Lleva a cabo alguna acción en respuesta a los cambios de estado del sujeto.

Normalmente, los observadores necesitan cierta información de contexto para procesar la actualización correctamente. Por esta razón, el sujeto suele pasar algunos datos de contexto como parámetros del método de notificación. El sujeto también puede pasarse a sí mismo como parámetros, permitiendo al observador consultar su estado directamente.

# Colaboraciones

**Los observadores concretos se suscriben a los cambios del sujeto en los que están interesados.**

Pueden hacerlo los propios observadores concretos o un cliente.

**El objeto observado notifica a sus  
observadores cada vez que  
ocurre un cambio.**

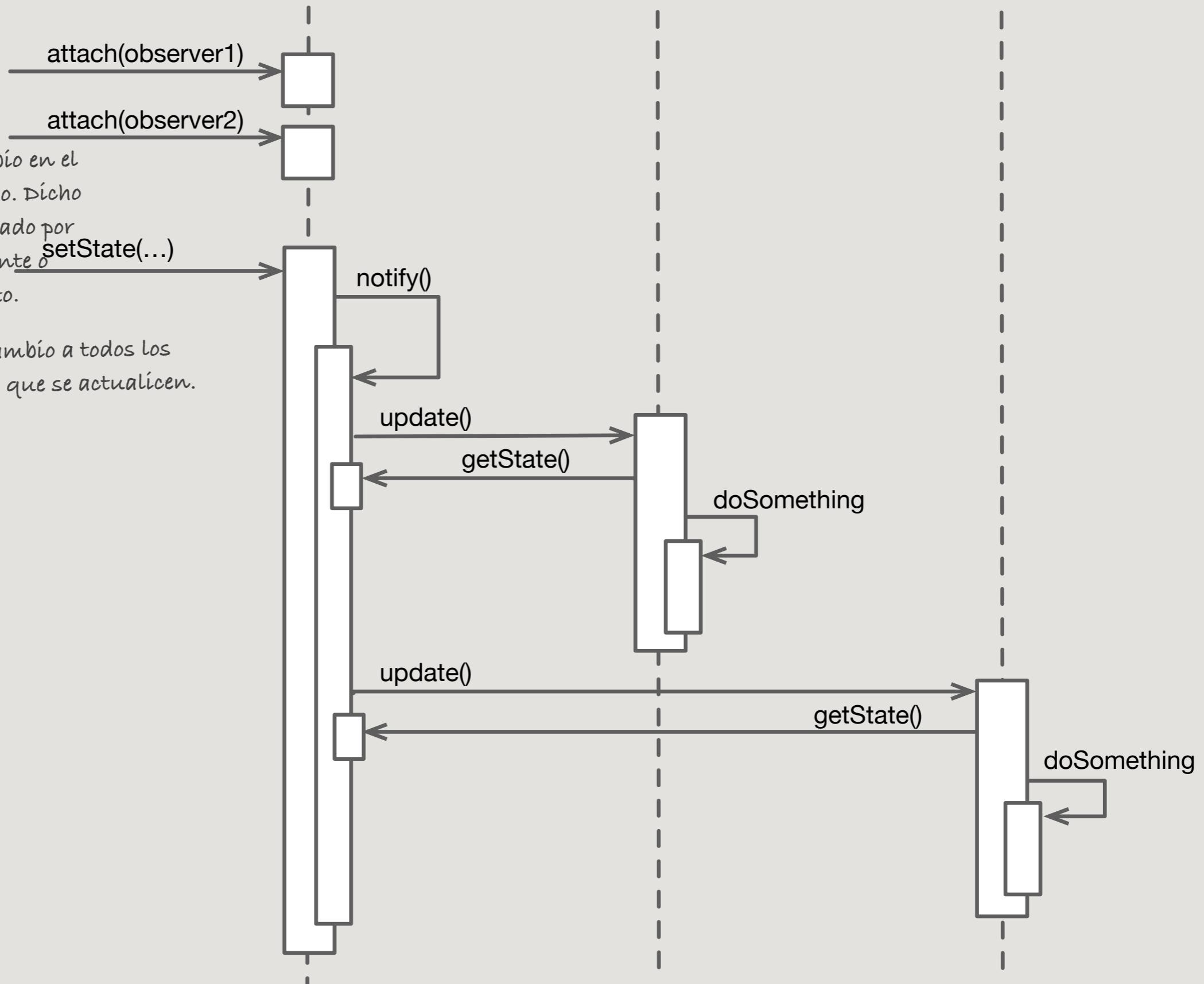
**Después de ser informado de un cambio en el objeto observado, cada observador concreto puede pedirle la información que necesita.**

**Los observadores concretos usan dicha información para realizar alguna acción y reconciliar su estado con el de aquél.**

Paso 1: O bien los propios observadores concretos se suscriben a los cambios de un sujeto o los suscribe un cliente.

2. Se produce algún cambio en el estado del sujeto observado. Dicho cambio puede estar motivado por alguna acción de un cliente o por un observador concreto.

3. Se notifican las actualizaciones a todos los observadores, pidiéndoles que se actualicen.



# Consecuencias

**Permite variar objetos observados y observadores independientemente.**

Se puede reutilizar los objetos observados (Subject) sin sus observadores, y viceversa.

Se pueden añadir nuevos observadores sin modificar ninguna de las clases existentes.

# **Acoplamiento abstracto entre Subject y Observer**

**Todo lo que un objeto sabe de sus observadores es que tiene una lista de objetos que satisfacen la interfaz Observer.**

**El sujeto no conoce sus clases concretas: el acoplamiento entre el sujeto y los observadores es abstracto y mínimo.**

Con lo que podrían pertenecer a dos capas distintas de la arquitectura de la aplicación o incluso a aplicaciones distintas.

# **Envío simultáneo**

**La notificación que envía el objeto observado no especifica ningún receptor concreto.**

**Se transmite automáticamente a todos los objetos interesados que se hayan suscrito a él.**

Al sujeto no le importa cuántos objetos interesados existen, ni de qué tipo son; su única responsabilidad es notificar a sus observadores.

Esto nos permite añadir y quitar observadores en cualquier momento.

Depende del observador procesar o ignorar una notificación.

# Actualizaciones inesperadas

Dado que los observadores no tienen conocimiento de la presencia de los demás, pueden ser ajenos al coste final de cambiar el sujeto.

Una operación aparentemente sencilla sobre el sujeto puede dar lugar a una sucesión de actualizaciones en cascada muy ineficientes.

# Implementación

# Correspondencia entre objetos observados y observadores

La forma más sencilla de que un sujeto sepa a qué observadores debe notificar es almacenar referencias a ellos explícitamente en el sujeto.

Lo anterior puede resultar costoso en términos de espacio cuando hay muchos sujetos y pocos observadores.

Una posible solución sería mantener mediante una tabla hash que relacionase a un sujeto con sus observadores.

# Observar más de un objeto

Cuando un observador dependa de más de un objeto, es necesario ampliar la información de la operación update para que el observador sepa qué sujeto está enviando la notificación.

Por ejemplo, incluyéndose el objeto observado a sí mismo como parámetro.

# Quién lanza la actualización

- a) Que las propias operaciones del sujeto llamen a notify cada vez que cambien su estado.

La ventaja de este enfoque es que los clientes no tienen que hacer nada con respecto a las notificaciones.

Tiene el inconveniente de que cuando hay varias operaciones consecutivas, cada una lanzará una actualización, lo que puede resultar ineficiente.

## Quién lanza la actualización

- b) La otra opción sería que los clientes llamen a notificar cuando corresponda.

La ventaja es que un cliente puede esperar para lanzar la actualización hasta que se hayan terminado de realizar todos los cambios de estado, evitando así actualizaciones intermedias innecesarias.

El inconveniente es que estos tienen una responsabilidad añadida, por un lado, y que podrían olvidarse de llamar a notificar, por otro.

# Protocolos de actualización: los modelos push y pull

Normalmente la implementación del patrón requiere que el sujeto envíe información adicional sobre el cambio como un parámetro del método update.

¿Cuánta? Dependerá de cada caso concreto.

En un extremo, que denominamos modelo push, el sujeto envía a los observadores información detallada sobre el cambio, la necesiten o no.

En el otro extremo está el modelo pull: el sujeto solo avisa de que cambió (tal vez pasándose a sí mismo como parámetro); son los observadores quienes le piden los datos que necesiten para actualizarse.

El modelo pull hace hincapié en el desconocimiento del sujeto con respecto a sus observadores, mientras que el modelo push parte de la base de que los sujetos saben algo sobre las necesidades de estos.

El modelo push puede dificultar la introducción de nuevos observadores, ya que podrían requerir datos que no estaban previstos en la firma del método update.

# Especificar explícitamente los cambios de interés

El sujeto puede permitir que los observadores se registren únicamente a aquellos eventos que les interesan.

Cuando se produzca un cambio, el sujeto notificará únicamente a los observadores que se hayan registrado a ese cambio concreto.



## Subject

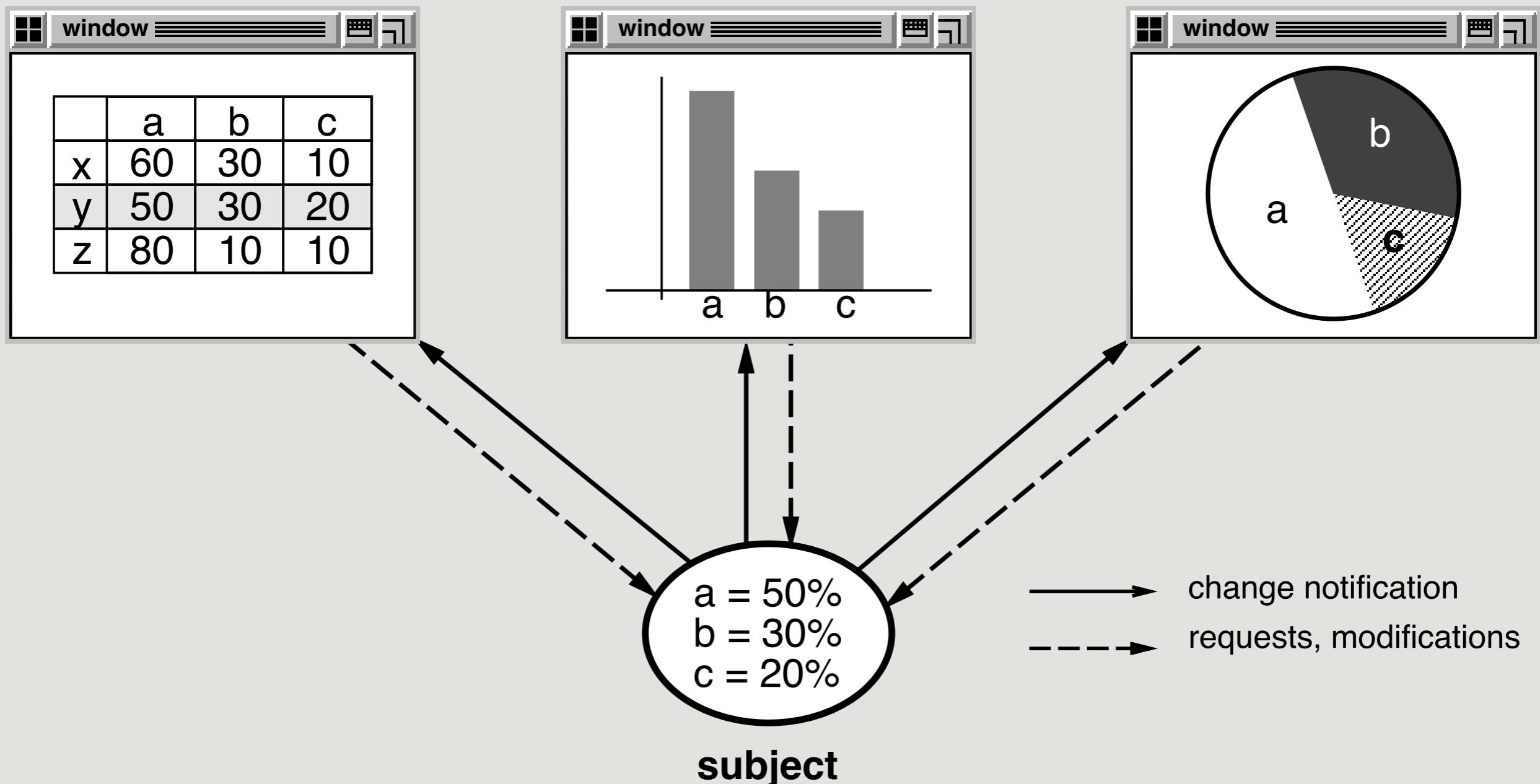
```
void addObserver(Observer observer, Event eventType) {  
    if (observers.containsKey(eventType)) {  
        observers.get(eventType).add(observer);  
    } else {  
        List<Observer> observerList = new ArrayList<>();  
        observerList.add(observer);  
        observers.put(eventType, observerList);  
    }  
}
```

Una posibilidad

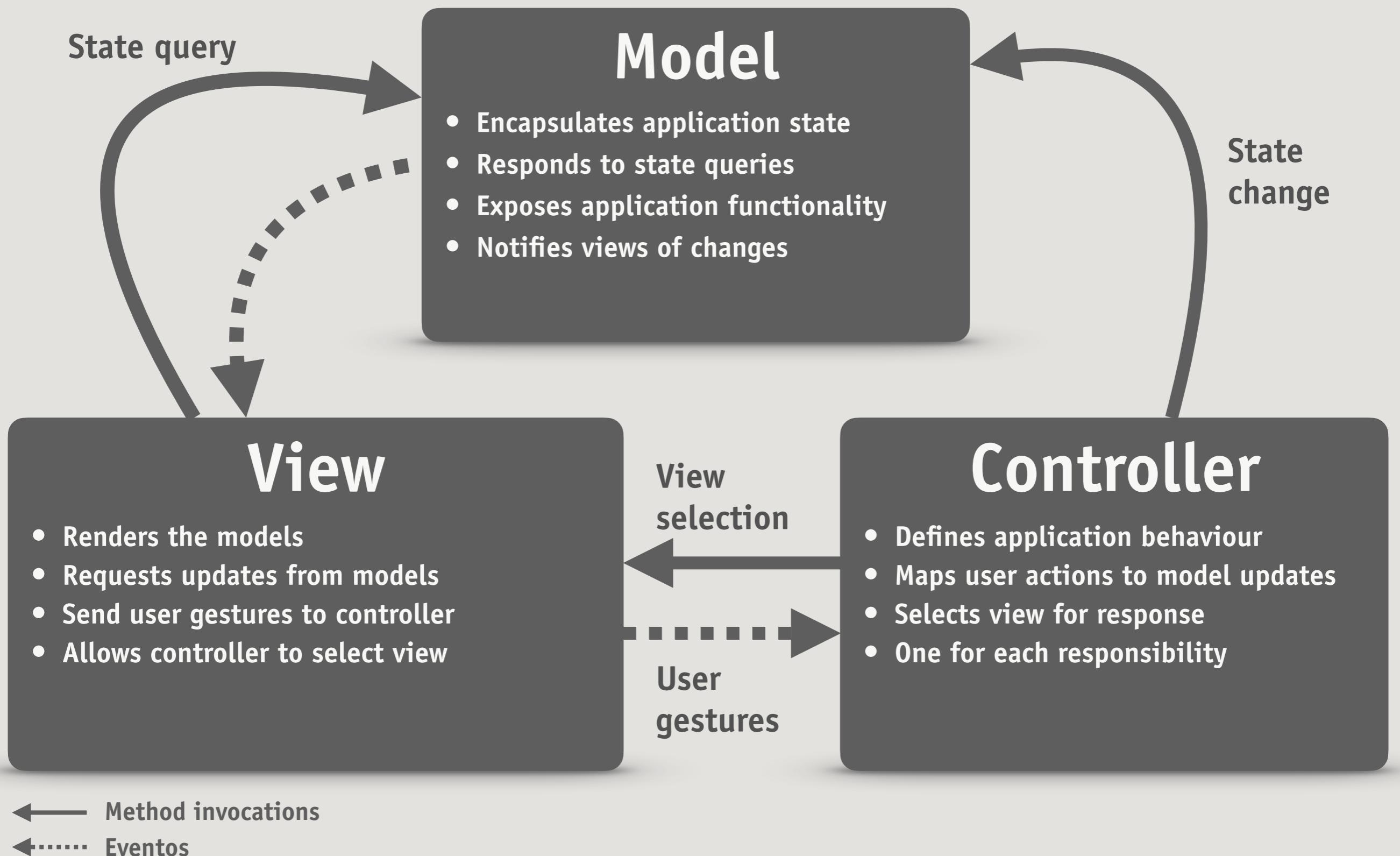


# Usos conocidos

## observers



# The MVC Triad



# Código de ejemplo



## How to Program in Unity: Observer Pattern Explained



iHeartGameDev  
80,6 K suscriptores

Unirme

Suscribirme

6,1 K



Compartir

Gracias

Clip

Guardar



115 K visualizaciones hace 1 año MANHATTAN

Learn the fundamentals of the Observer Pattern in this new video break down and create a dynamic narration system just like in the game "Bastion" by SuperGiant Games!

This tutorial will teach you how to program and design systems in unity that communicate seamlessly while remaining easy to debug, scale, and maintain. Learn the power of decoupled code and e ...más

# Patrones relacionados

**Un  
comentario  
final**

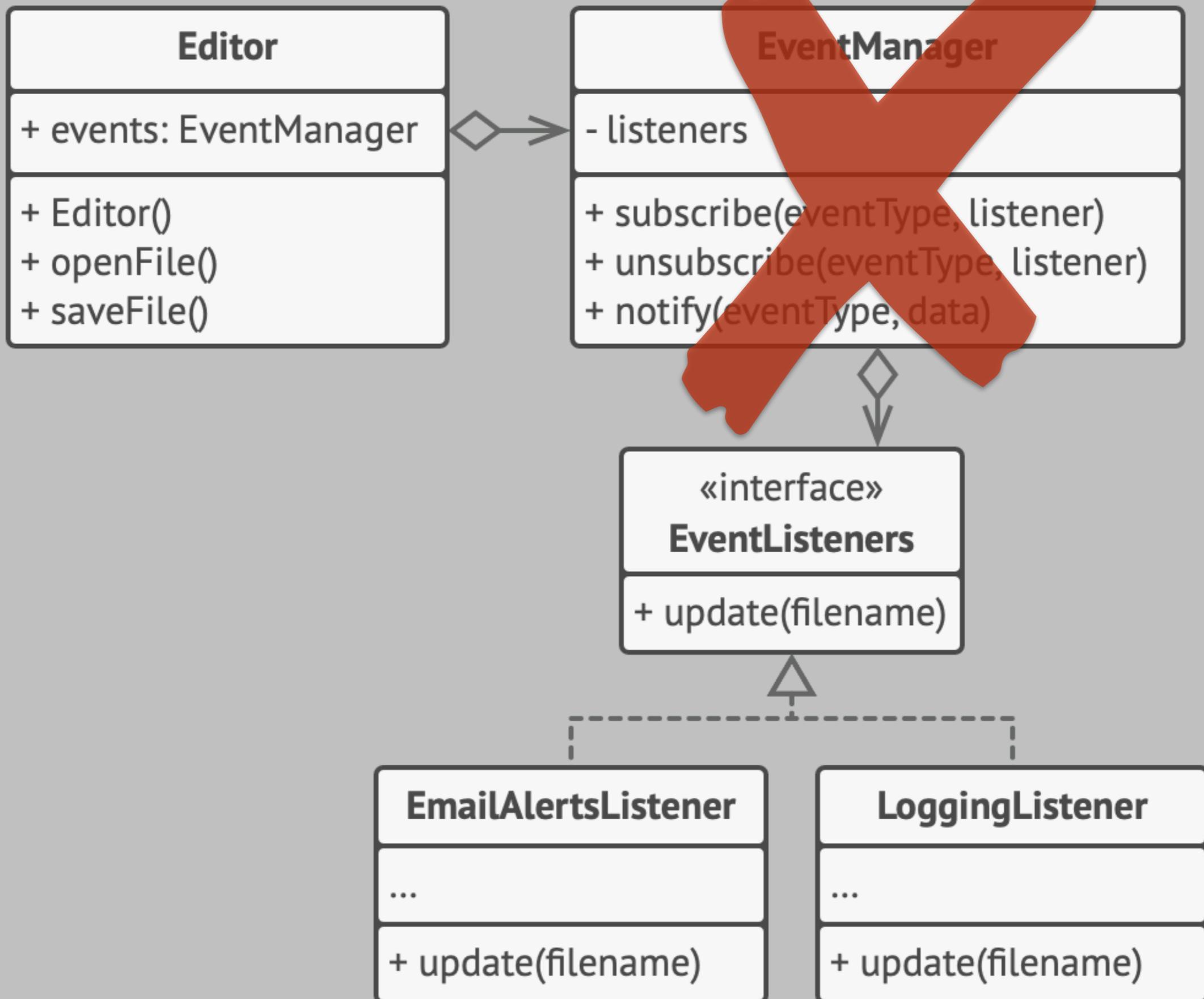
*¡Cuidado!*

*Dive Into*

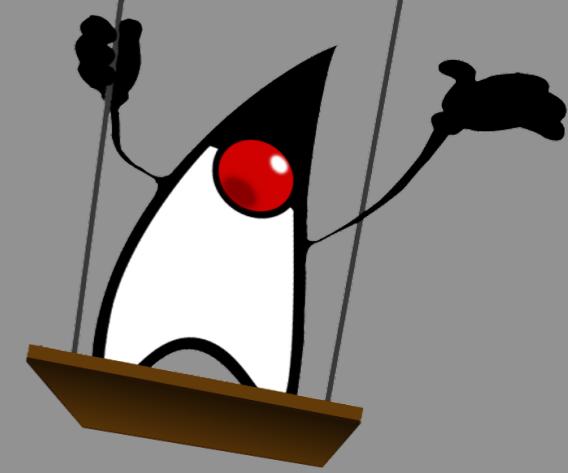
# DESIGN PATTERNS



Alexander Shvets



# Eventos en Java



# En el JDK 1.0, el modelo de eventos de AWT (Abstract Window Toolkit) se basaba en la herencia.

Para que un programa pudiera capturar y manejar los eventos de la interfaz de usuario, tenía que crear una subclase de los componentes gráficos de la biblioteca y redefinir sus métodos `action` o `handleEvent`.

Si uno de tales métodos devuelve `verdadero`, significa que el evento ya se procesó; de lo contrario, este se propagará por toda la jerarquía de objetos de la interfaz gráfica hasta que se consuma o se alcance la raíz de la jerarquía.

**Los programas tenían dos opciones para estructurar su código de gestión de eventos:**

**Heredar de cada componente de la interfaz para manejar específicamente sus eventos.**

Lo que da como resultado un sinfín de clases.

**Que un componente concreto se encargue de los eventos de toda la jerarquía (o una parte de ella).**

En ese caso el método `action` o `handleEvent` redefinido debe contener lógica condicional para poder procesar dichos eventos.

# Esto daba lugar a los siguientes problemas:

No había una separación limpia entre el código de la aplicación y la interfaz de usuario, porque la lógica de la aplicación tenía que integrarse directamente en las subclases de los componentes gráficos, de un modo u otro.

Dado que TODOS los tipos de eventos se manejan a través de los mismos métodos, la lógica para procesar los diferentes tipos de eventos es compleja y propensa a errores.

This becomes an even greater problem as new event types were added to the AWT.

**No había filtrado de eventos.**

Los eventos se envían siempre a los componentes, independientemente de que estos los traten o no.

**El origen del evento se pasaba normalmente como una cadena.**

**A partir de JDK 1.1, Java sigue el modelo de delegación de eventos definido en la especificación JavaBeans.**

Un modelo de gestión de eventos desacoplado que facilita el mantenimiento y la flexibilidad mediante la adopción del patrón Observer.

En este nuevo modelo, los eventos se gestionan a través de oyentes y objetos de eventos, lo que permite una mejor separación entre el origen del evento y los oyentes que lo gestionan.



---

## *Sun Microsystems*

---

*JavaBeans<sup>TM</sup>*

---

This is the JavaBeans™ API specification. It describes the core specification for the JavaBeans component architecture.

This version (1.01) of the specification describes the JavaBeans APIs that are present in JDK 1.1. These included some very minor API additions to the 1.00-A specification of December 1996. This version of the spec has also been updated to include various minor clarifications of various sections of the 1.00 spec, and to include some additional guidance for component developers. See the change history on page 112.

Because of the volume of interest in JavaBeans we can't normally respond individually to reviewer comments, but we do carefully read and consider all reviewer input. Please send comments to [java-beans@java.sun.com](mailto:java-beans@java.sun.com).

To stay in touch with the JavaBeans project, visit our web site at:

<http://java.sun.com/beans>

# Aspectos fundamentales

Las notificaciones de eventos se propagan desde los objetos originarios a los oyentes mediante invocaciones de métodos en dichos objetos.

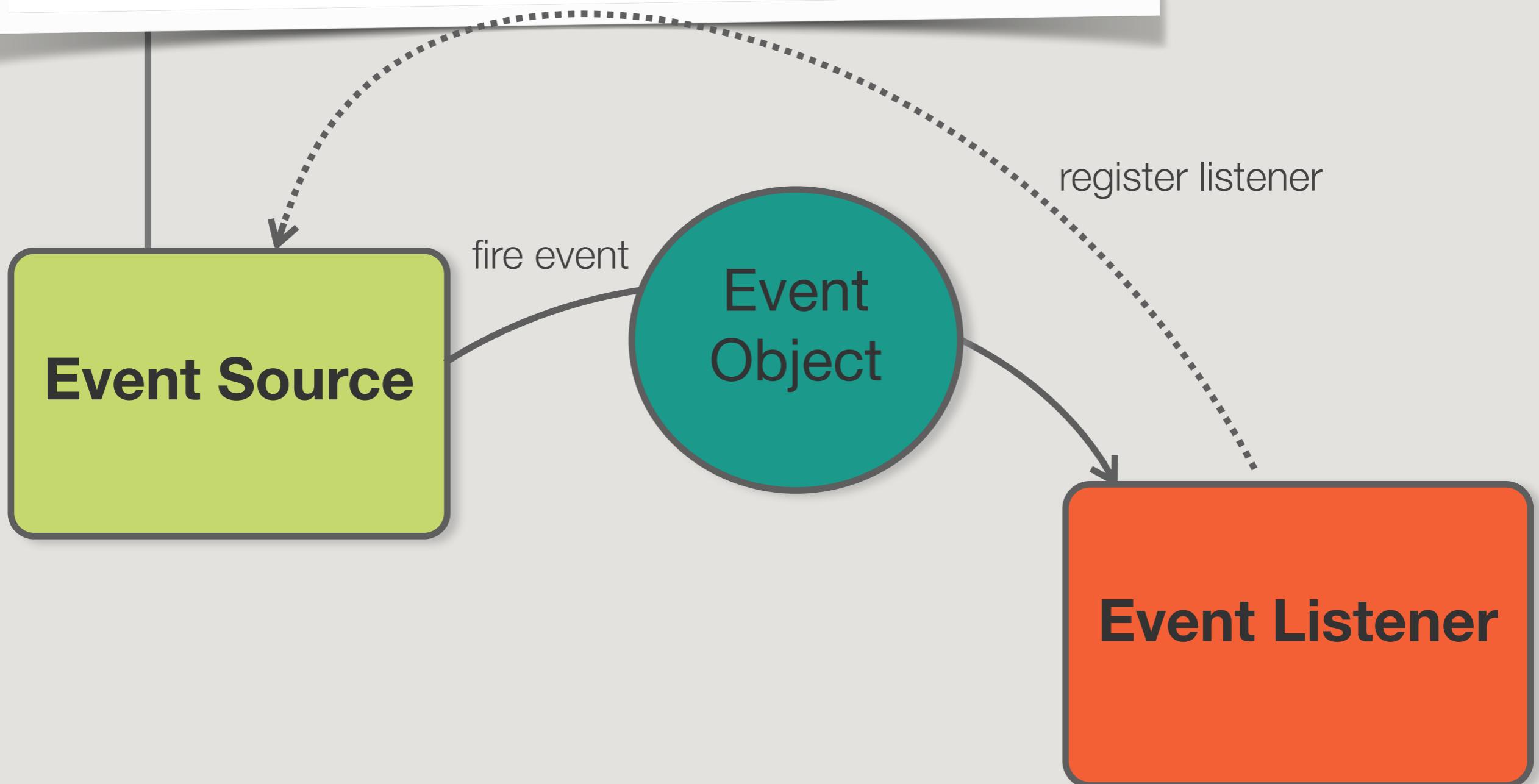
Hay un método distinto por cada tipo de evento.

Luego, estos métodos se agrupan en interfaces de «oyentes» que extienden la interfaz `EventListener`, definida en `java.util`.

La información asociada a una notificación de evento concreta se encapsula normalmente en un objeto para dicho evento.

Una subclase de la clase `EventObject` definida en `java.util`.

```
public synchronized  
FooListener addFooLister(FooListener listener);
```



# Objetos de eventos

java.util.EventObject

**Encapsula la información asociada a una notificación de evento concreta.**

Un evento de clic de ratón debe contener la posición del puntero, un cambio de temperatura incluirá ésta, etcétera.

**Además, guardan una referencia al objeto que dio origen al evento.**

Por convenio, a estas clases que representan eventos se les da un nombre terminado en -Event.



## TemperatureChangeEvent.java

```
public class TemperatureChangeEvent extends  
java.util.EventObject {  
  
    private double temperature;  
  
    TemperatureChangeEvent(Object source, double temperature) {  
        super(source);  
        this.temperature = temperature;  
    }  
  
    public double getTemperature() {  
        return temperature;  
    }  
}
```

# Interfaces de oyentes

java.util.EventListener

**Un oyente es un objeto al que se le notifican los eventos.**

Para lanzar un evento, el código fuente debe saber a qué método llamar. Esta información está contenida en una interfaz EventListener.

**Las notificaciones de eventos se llevan a cabo a través de invocaciones de métodos en el objeto receptor, con el objeto de evento pasado como parámetro.**

Por convenio, el nombre de todas las interfaces de oyentes termina en -Listener.

Un «oyente» puede contener cualquier número de métodos, cada uno correspondiente a evento distinto (se presupone que relacionados).



### TemperatureChangeListener.java

```
public interface TemperatureChangeListener  
extends java.util.EventListener {  
    void temperatureChanged(TemperatureChangeEvent evt);  
}
```

# Generadores de eventos

**Los generadores de eventos son objetos que los lanzan.**

Estos objetos implementan métodos que permiten al oyente registrarse, si está interesado en los eventos que genera.

El programador de un objeto que está interesado en los eventos de un determinado objeto debe implementar la interfaz EventListener apropiada y registrarse para los eventos que le interesan.



## TemperatureSensor.java

```
public class TemperatureSensor {  
    private List<TemperatureChangeListener> listeners;  
    // ...  
    public void addTemperatureChangeListener(  
        TemperatureChangeListener listener) {  
        listeners.add(listener);  
    }  
  
    public void removeTemperatureChangeListener(  
        TemperatureChangeListener listener) {  
        listeners.remove(listener);  
    }  
    // ...  
}
```