2

(X)

Patrón Observer

(Patrones de diseño)

Diseño del Software

Grado en Ingeniería Informática del Software

Curso 2022-2023

Escuela de Ingeniería Informática
Universidad de Oviedo

César Acebal

Observer (Observador)

- **Patrón de comportamiento (de objetos)**
- **Propósito:**

> *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:**
  - Publicar-Suscribir (*Publish-Subscribe*)

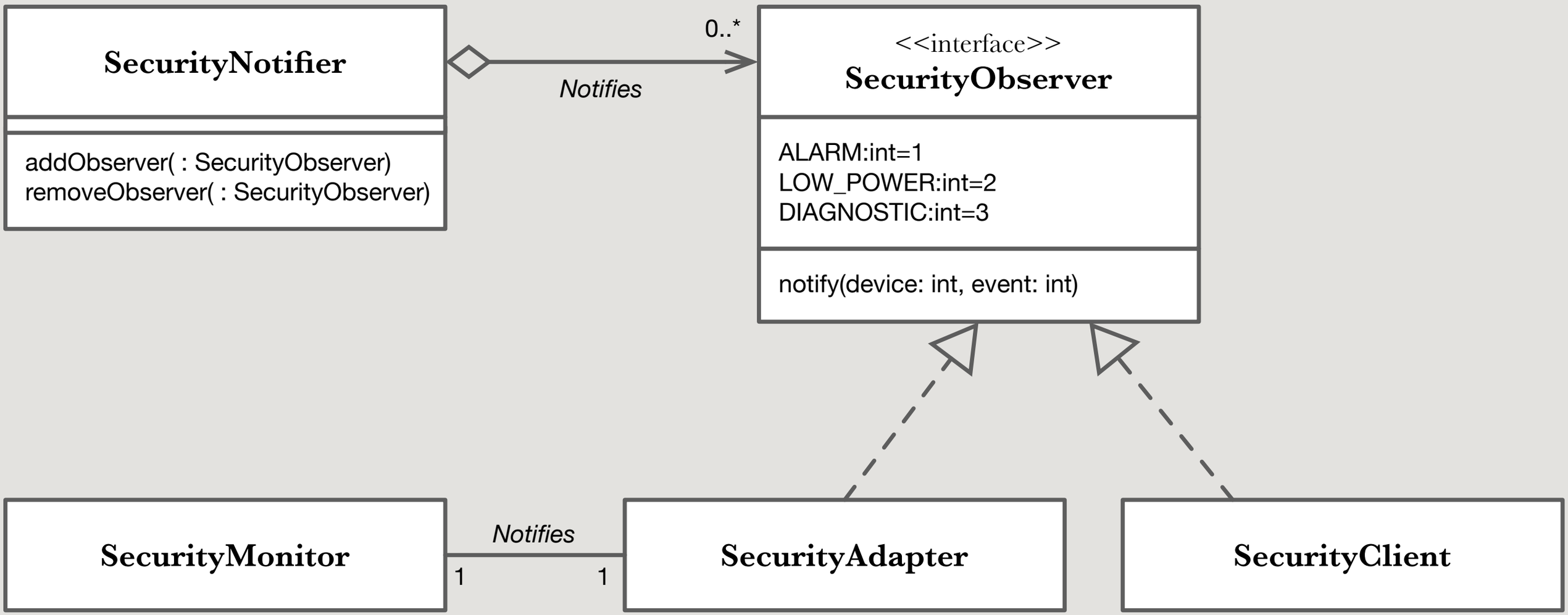
Motivación

- **Muchas veces un efecto lateral de partir un sistema en una colección de objetos relacionados es que necesitamos mantener la consistencia entre dichos objetos**
  - ¿Cómo hacerlo sin que sus clases estén fuertemente acopladas?

Ejemplo

- Fabricamos detectores de humo, sensores de movimiento y otros dispositivos de seguridad que pueden enviar una señal a una tarjeta de ordenador
- Queremos que las compañías que desarrollan programas de monitorización los integren en sus sistemas
- ¿Cómo sabemos a qué objetos avisar?

Ejemplo



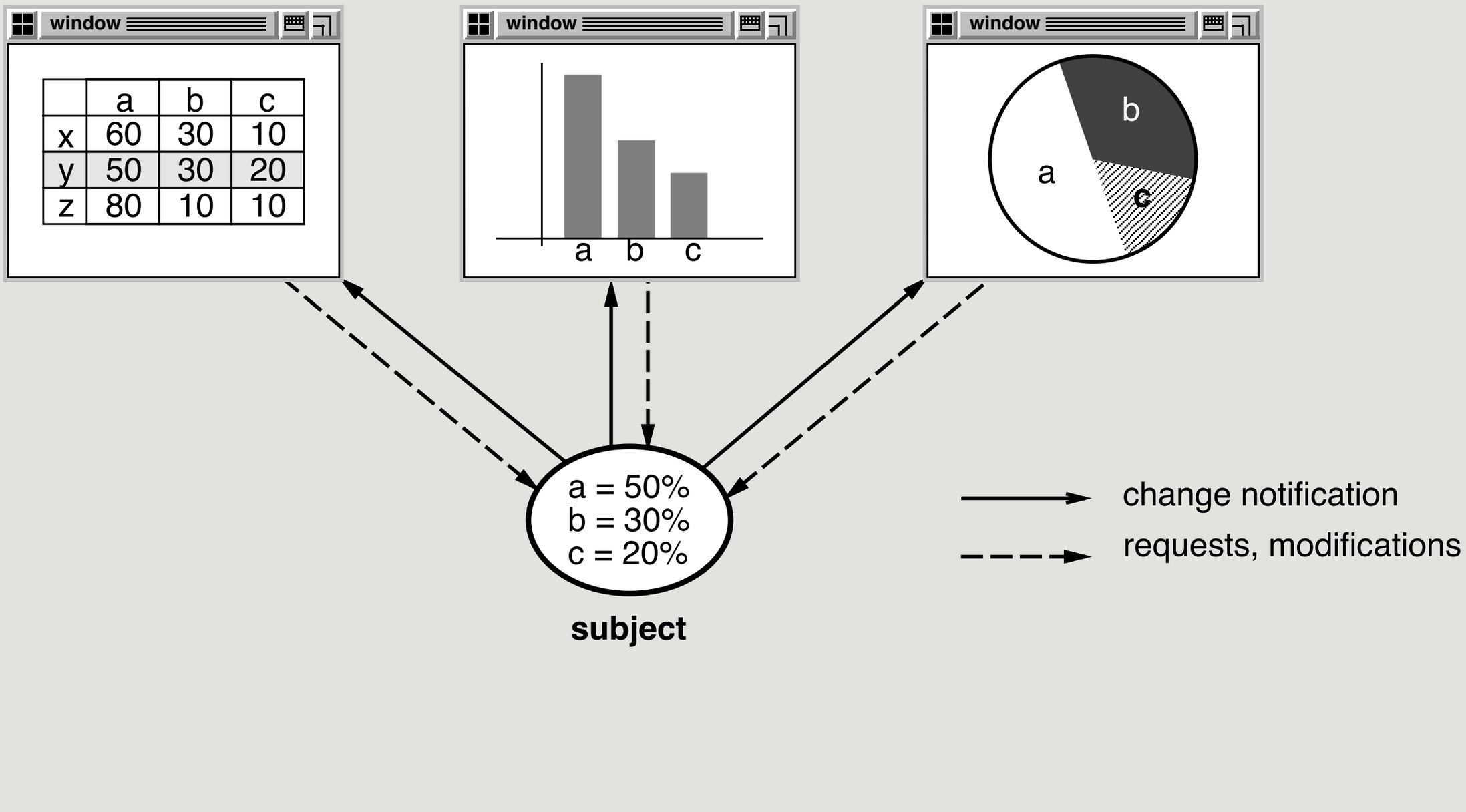
Ejemplo extraído de *Patterns in Java. Volume 1*
(M. Grand, John Wiley & Sons, 1998)

Otro ejemplo

- **Fórmula en una hoja de cálculo que contenga referencias a otras celdas**

	A	B	C	D	E	F	G	H
1	Ingresos y gastos	Enero	Febrero	Marzo	Abril	Mayo	Junio	Julio
2	Balance del mes previo		€ 3.305	€ 6.610	€ 9.915	€ 13.220	€ 16.525	€ 21.830
3	Efectivo disponible	€ 7.000	€ 7.000	€ 7.000	€ 7.000	€ 7.000	€ 7.000	€ 7.000
4	Ingresos adicionales	€ 0	€ 0	€ 0	€ 0	€ 0	€ 2.000	€ 0
5	Gastos mensuales	€ 3.695	€ 3.695	€ 3.695	€ 3.695	€ 3.695	€ 3.695	€ 3.695
6	Gastos planificados	€ 0	€ 0	€ 0	€ 0	€ 0	€ 0	€ 880
7	Ahorros	=SUMA(B2:B4)-Enero Gastos mensuales-Enero Gastos planificados						€ 24.255

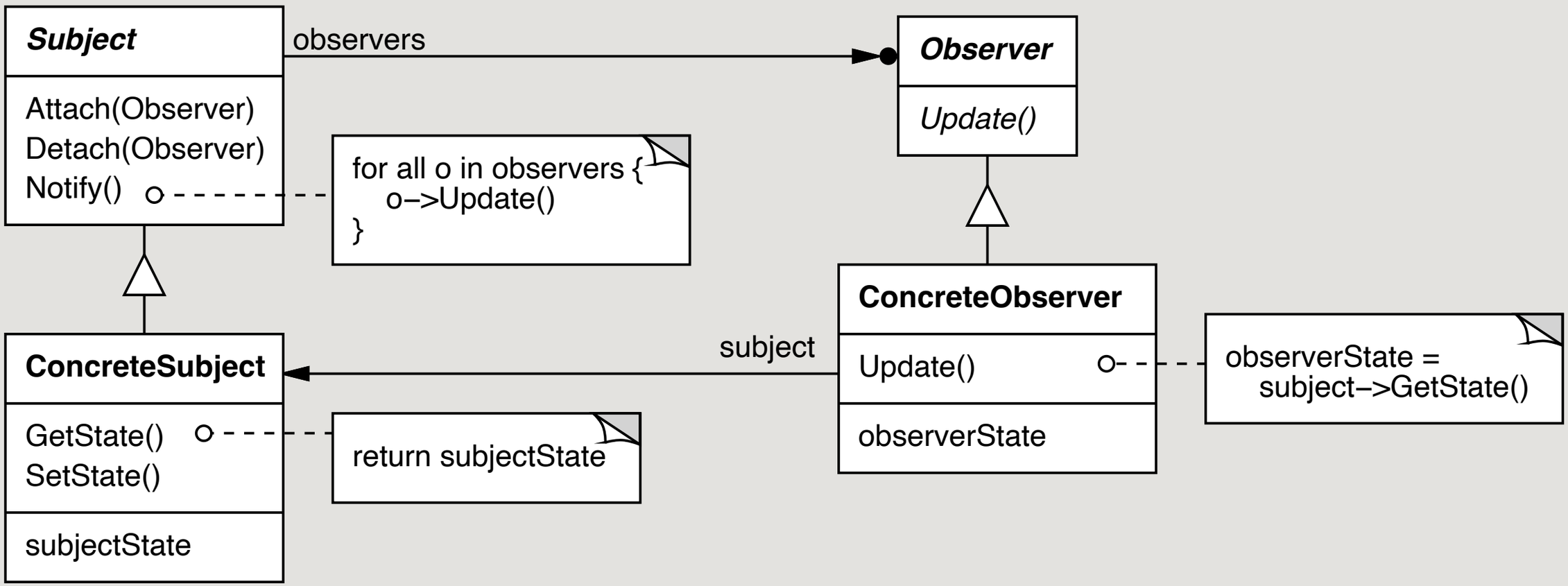
Otro más


observers
window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
window
a b c
window
b
a
c
a = 50%
b = 30%
c = 20%
subject
change notification
requests, modifications

Aplicabilidad

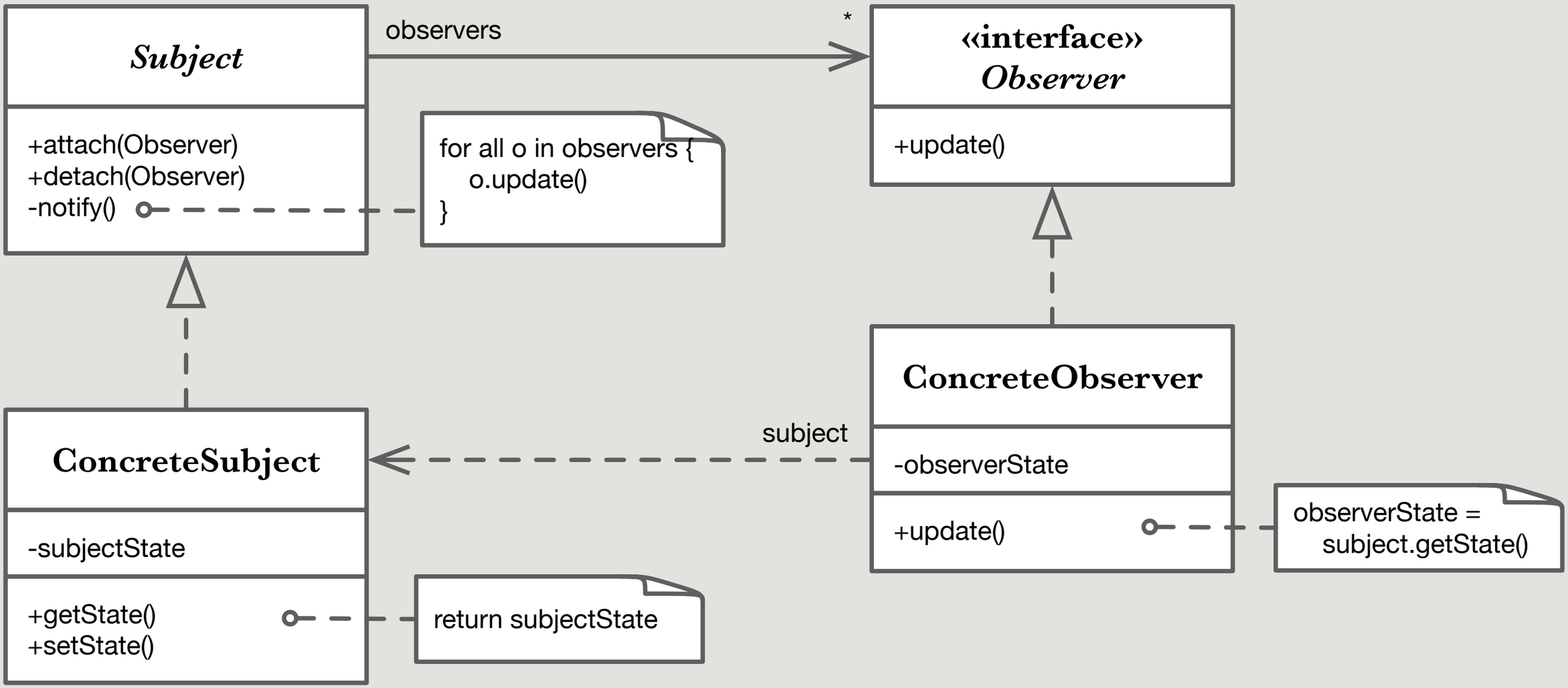
- **Una abstracción tiene dos aspectos, uno de los cuales depende del otro**
  - Encapsular estos aspectos en objetos separados permite que los objetos varíen (y puedan ser reutilizados) de forma independiente
- **Un cambio en un objeto requiere que cambien otros**
  - Y no sabemos a priori cuáles ni 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



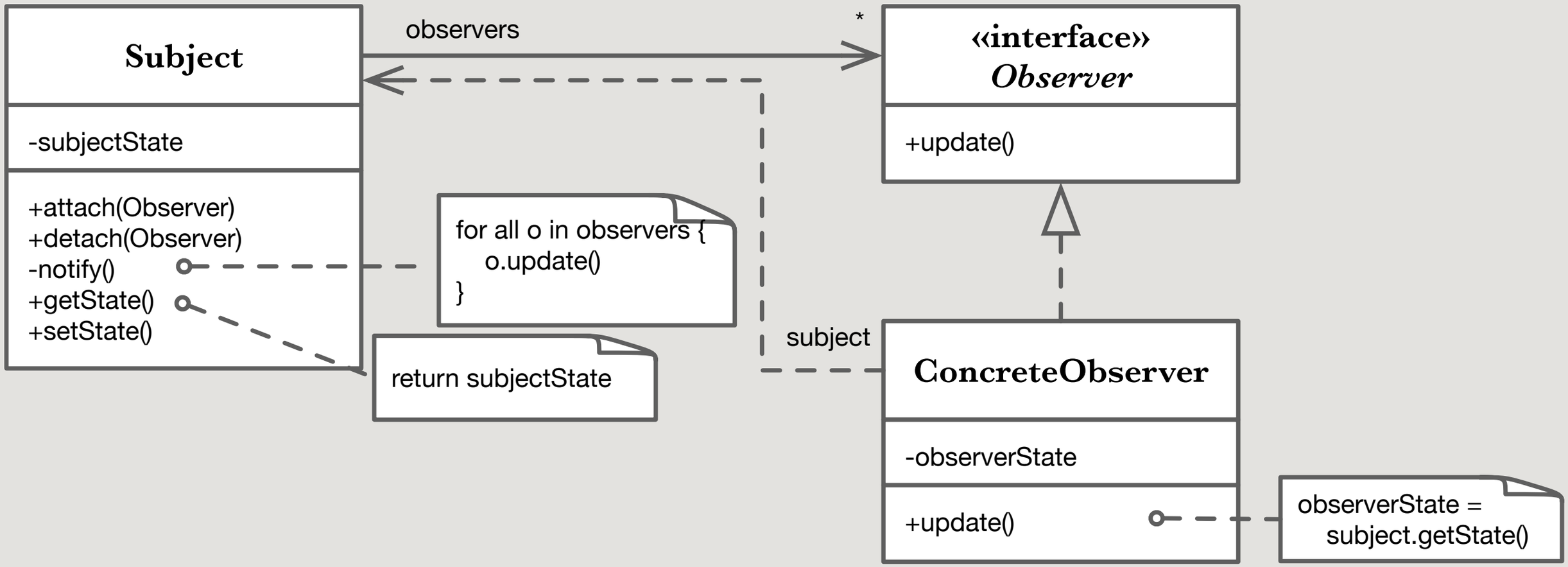
La estructura del patrón *Observer*, tal como aparece en el GoF

Estructura



La estructura del patrón *Observer*, en UML

Estructura



La estructura del patrón *Observer*, como suele ser más habitual en la práctica (sin una interfaz o clase abstracta aparte específicas sólo para el «Subject», sino añadiéndole dichas responsabilidades directamente al objeto observado)

Participantes

- **Subject**
  - Conoce a sus observadores
  - Proporciona una interfaz para que se suscriban los objetos Observer (o que se borren)
- **Observer**
  - Define una interfaz para actualizar los objetos que deben ser notificados de cambios en el objeto Subject

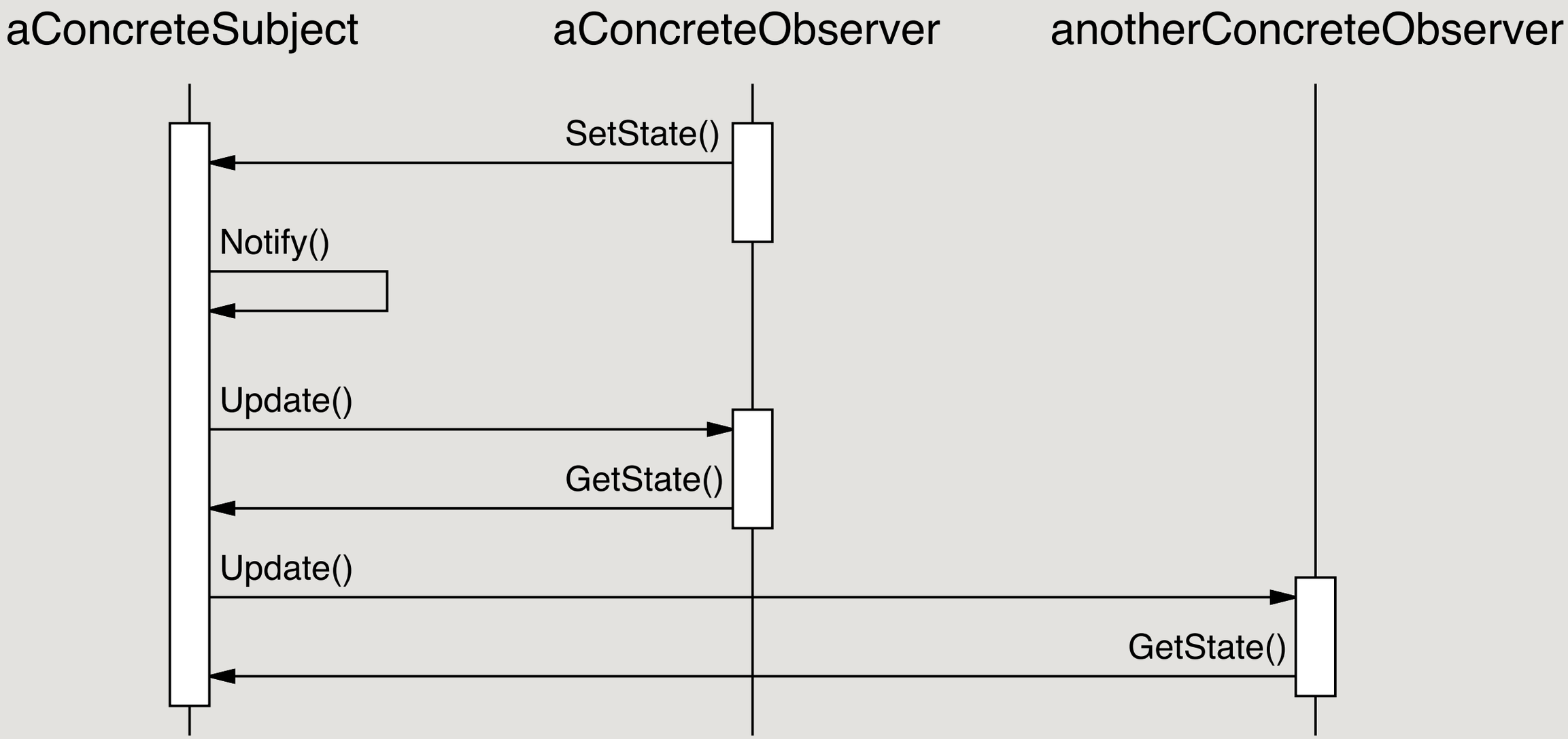
Participantes

- **ConcreteSubject**
  - Guarda el estado de interés para los objetos ConcreteObserver
  - Envía una notificación a sus observadores cuando cambia su estado
- **ConcreteObserver**
  - Mantiene una referencia a un objeto ConcreteSubject
  - Guarda el estado que debería permanecer sincronizado con el objeto observado
  - Implementa la interfaz Observer para mantener su estado consistente con el objeto observado

Colaboraciones

- **El objeto observado notifica a sus observadores cada vez que ocurre un cambio**
  - A fin de que el estado de ambos permanezca consistente
- **Después de ser informado de un cambio en el objeto observado, cada observador concreto puede pedirle la información que necesita para reconciliar su estado con el de aquél**

Colaboraciones



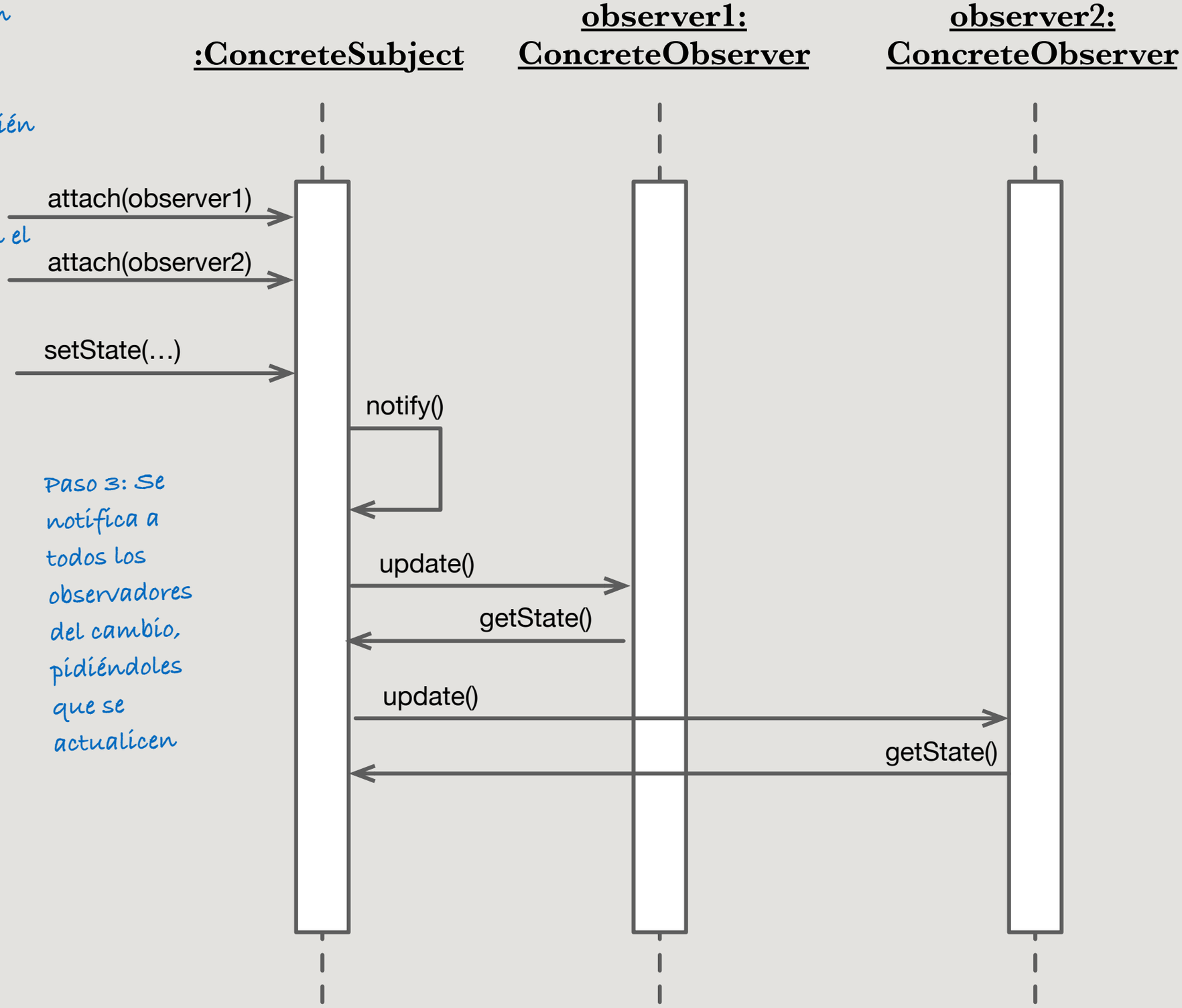
En este diagrama de secuencia (tal como aparece en el libro) faltaría el paso previo de suscripción de los observadores; se completa en el más detallado de la siguiente diapositiva

Colaboraciones

Paso 1: los observadores se suscriben al sujeto (esto se puede hacer desde fuera, no es necesario que sean los propios observadores; aunque también se podría hacer que fueran ellos mismos, al crearse, en los casos en que reciben al sujeto observando en el constructor)

Paso 2: Cambia el estado del sujeto, por cualquier causa (algún cliente de la clase, algún observador...)

Paso 3: Se notifica a todos los observadores del cambio, 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

Consecuencias

- **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
    - Con lo que podrían incluso pertenecer a dos capas distintas de la arquitectura de la aplicación
- **No se especifica el receptor de una actualización**
  - Se envía a todos los objetos interesados

Consecuencias

- **Actualizaciones inesperadas**
  - Se podrían producir actualizaciones en cascada muy ineficientes

Implementación

Correspondencia entre objetos observados y observadores

- **En vez de mantener una colección con referencias explícitas a los observadores en el objeto observado, sería posible hacerlo con una tabla hash que relacionase ambos**
  - Útil cuando hay muchos objetos a observar y pocos observadores, para reducir los costes de almacenamiento

Observar más de un objeto

- **Cuando un observador dependa de más de un objeto, es necesario ampliar la información de la operación `update`**
  - Por ejemplo, incluyéndose el objeto observado a sí mismo como parámetro, para que el observador pueda discriminar

¿Quién lanza la actualización?

- **Es decir, ¿quién se encarga de llamar a `notify`?**
  - El objeto observado, cada vez que cambia su estado
    - Puede dar lugar a actualizaciones ineficientes
  - Los clientes
    - Puede eliminar actualizaciones intermedias innecesarias
    - Más propenso a errores: los clientes pueden olvidarse de llamar a **`notify`**

Protocolos de actualización

- **Modelo push**
  - El objeto observado envía información detallada a sus observadores sobre el cambio producido
    - (La necesiten o no)
- **Modelo pull**
  - Tan sólo avisa de que cambió
    - Los observadores le solicitan la información que necesiten

Especificar explícitamente el aspecto que varía

- Podemos extender la interfaz de registro de observadores para que éstos indiquen los eventos que les interesan
- Cuando se produzca un evento, el objeto observado informará sólo a los observadores interesados en ese evento

```
Void Subject::Attach(Observer*, Aspect& interest);

Void Observer::Update(Subject*, Aspect& interest);
```

Eventos en Java

JDK 1.0

- **Clase Observable e interfaz Observer**
  - (Paquete java.util)
- **Se basa en la herencia**
- **Lo mismo en AWT:**
  - Había que heredar de los componentes gráficos y redefinir los métodos action o handleEvent
    - Si devolvían true, se consumía el evento; si no, se propagaban hacia arriba en la jerarquía de componentes
    - (Hasta que se consumía o se llegaba al contenedor raíz)
    - O se creaba una subclase de cada componente para que manejase los eventos o se dejaba esa tarea para el contenedor principal

Problemas del modelo de eventos de AWT en el JDK 1.0

- Se mezclaba el código de aplicación (el tratamiento de los eventos) con el de la interfaz de usuario
- Problemas para añadir los nuevos tipos de eventos
- No había filtrado de eventos: todos se propagaban a todos los componentes
- Dificultad para conocer el origen del evento (a través de String)

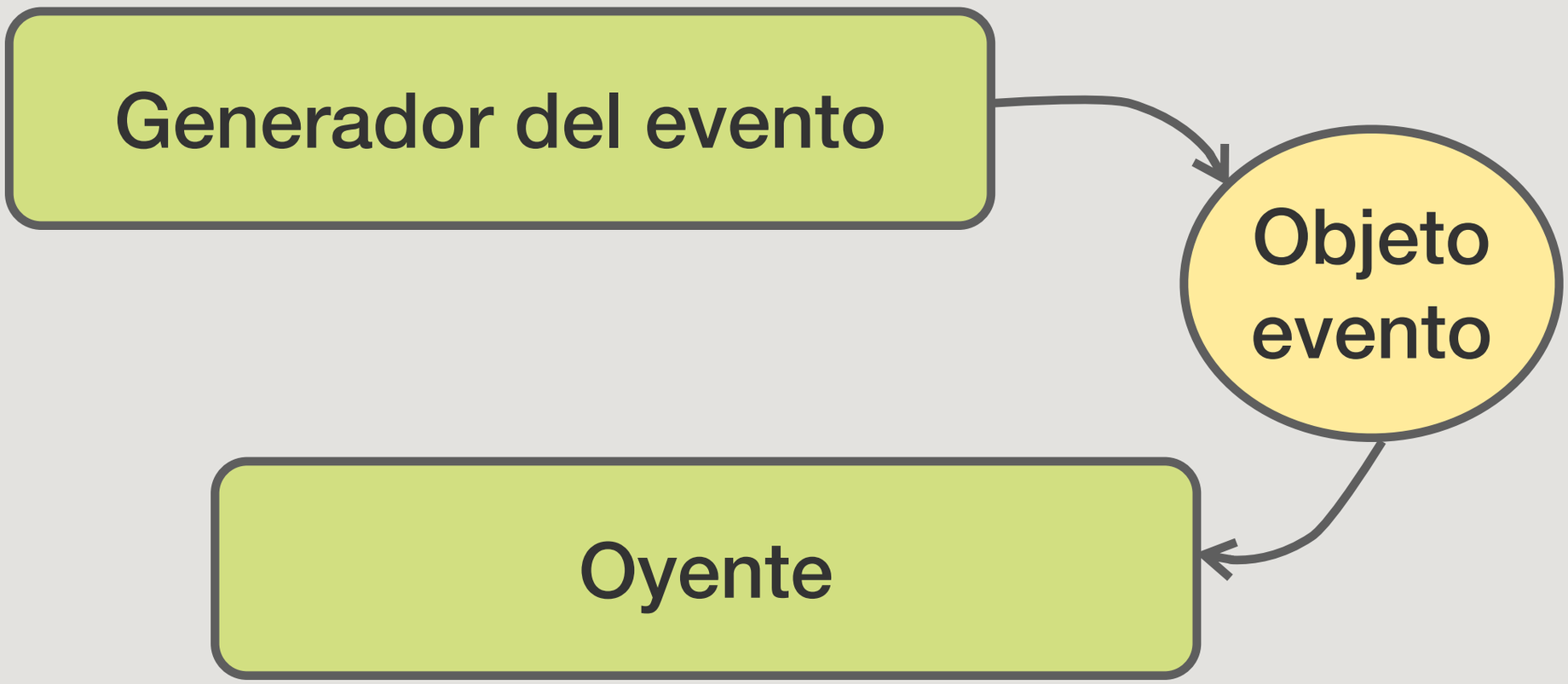
El modelo de delegación de eventos

- **A partir de JDK 1.1, Java sigue el modelo de delegación de eventos que se define en la especificación de los JavaBean**
  - http://java.sun.com/products/javabeans/docs/spec.html

    http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html

Introducción al modelo de eventos

- **El paquete `java.util` provee soporte básico para el modelo de eventos**
  - Proporciona una clase base para los eventos, y una interfaz para los «oyentes»



Eventos

- **Clase `java.util.EventObject`**
- **Un objeto evento encapsula la información que es específica a una instancia de un evento**
  - 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**

Eventos

- Se debe crear una subclase de `EventObject` para generar una clase particularizada.
- Por convención, el nombre de estas clases termina en `Event`

```
public class SimpleEvent extends
java.util.EventObject
{
    SimpleEvent(Object source)
    {
        super(source);
    }
}
```

Ejemplo

- Un sistema de control de la temperatura podría incluir un evento similar a éste:

```
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;
    }
}
```

Receptores de eventos (listeners)

- **Un oyente es un objeto al que se le notifican los eventos**
- **Las notificaciones de eventos se realizan a través de invocaciones a métodos del objeto receptor, con el objeto evento pasado como parámetro**
- **Para disparar un evento, el código fuente debe saber a qué método llamar**
  - Esta información está contenida en una interfaz EventListener que define tales métodos
- **Cualquier clase que quiera recibir notificaciones del evento deberá implementar dicha interfaz**

Interfaz EventListener

- **Todas las interfaces `EventListener` heredan del interfaz base `java.util.EventListener`**
  - Esta interfaz no define ningún método
- **Por convención, todas las interfaces de «oyentes» finalizan con la palabra `Listener`**
- **Un «oyente» puede contener cualquier número de métodos, cada uno correspondiente a evento distinto**
  - (Se supone que relacionados)

Objetos oyentes

- Si un objeto quiere escuchar los eventos provistos por un objeto emisor, tiene que implementar la interfaz asociada

```
public interface TemperatureChangeListener extends
EventListener
{
    // Este método se llama siempre que la temperatura
    // ambiente cambie
    void temperatureChanged(TemperatureChangeEvent evt);
}
```

Convenio

- La especificación de JavaBean define la siguiente forma estándar para los métodos a los que se notifican eventos:

```
void <nombreEvento>(<TipoDeEvento> evt);
```

Generadores de eventos

- Las fuentes de eventos son objetos que los disparan
- 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 asociados con un objeto fuente debe implementar la interfaz `EventListener` apropiada y registrar los eventos que le interesan

Métodos de registro

- **Por convenio:**

```
public void add<TipoListener>(<TipoListener> listener)
public void remove<TipoListener>(<TipoListener> listener)
```