

# Patrones de diseño de software

Ingeniería de Software – IIC2143

Yadran Eterovic ([yadran@ing.puc.cl](mailto:yadran@ing.puc.cl))

2-2018

# 0

Los *patrones de diseño* de software que estudiaremos a continuación fueron presentados inicialmente en

... “*Design Patterns: Elements of Reusable Object-Oriented Software*”, de E. Gamma, R. Helm, R. Johnson y J. Vissides [1994]

Son conocidos como los **patrones GoF** (*gang of four*, por los cuatro autores del libro)

# Los patrones de diseño GoF son 23

... y están agrupados en tres categorías:

## **de creación**

crean objetos, en lugar de que los tenga que instanciar uno mismo

## **estructurales**

relativos a la composición de clases y objetos

## **de comportamiento**

relativos a la comunicación entre objetos

Hay 5 patrones Gof de creación:

- Fábrica abstracta
- Constructor
- Método de fábrica
- Prototipo
- *Singleton*

Dan al programa (software) flexibilidad para decidir qué objetos necesitan ser creados en un caso dado:

- instanciación de objetos, desacoplando un cliente de los objetos que necesita instanciar

Hay 7 patrones GoF estructurales:

- **Adaptador**
- **Puente**
- **Compuesto**
- **Decorador**
- **Fachada**
- ***Flyweight***
- ***Proxy***

Usan herencia para componer interfaces

... y para definir formas de componer objetos para obtener nueva funcionalidad —composición de clases u objetos

Hay 11 patrones GoF de comportamiento:

- Cadena de responsabilidad
- Comando
- Intérprete
- Iterador
- Mediador
- *Memento*
- **Observador**
- Estado
- **Estrategia**
- Método plantilla
- Visitante

Cómo las clases y los objetos interactúan y se distribuyen las responsabilidades

# 1

## Un sistema PdV y varios subsistemas para calcular impuestos

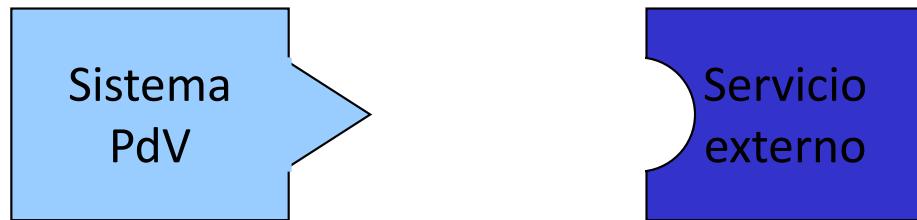
Un sistema PdV (*punto de venta*) necesita poder conectarse a diversas aplicaciones para calcular impuestos —p.ej., Tax-Master y TopTaxPro— desarrolladas por terceros

... sólo que cada una de estas aplicaciones tiene su propia API —conjunto de métodos públicos— que no puede ser cambiada

Una solución es agregar *un nivel de indirección*:

objetos que adapten las diversas interfaces externas (de los subsistemas) a una interfaz consistente usada por el sistema

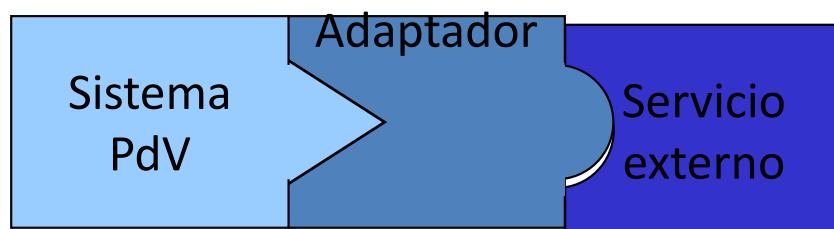
Convertimos la interfaz original en otra interfaz, a través de un objeto *adaptador* intermedio



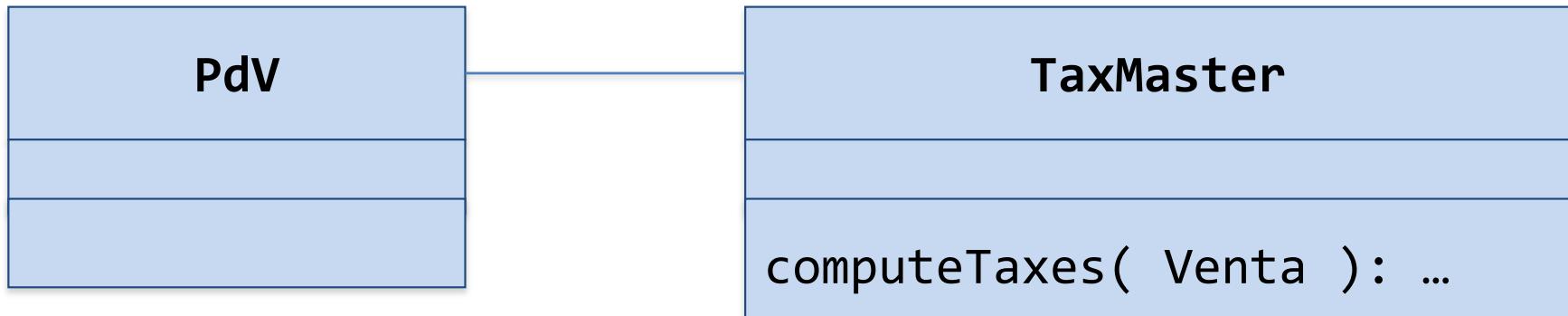
El código del sistema PdV no cambia



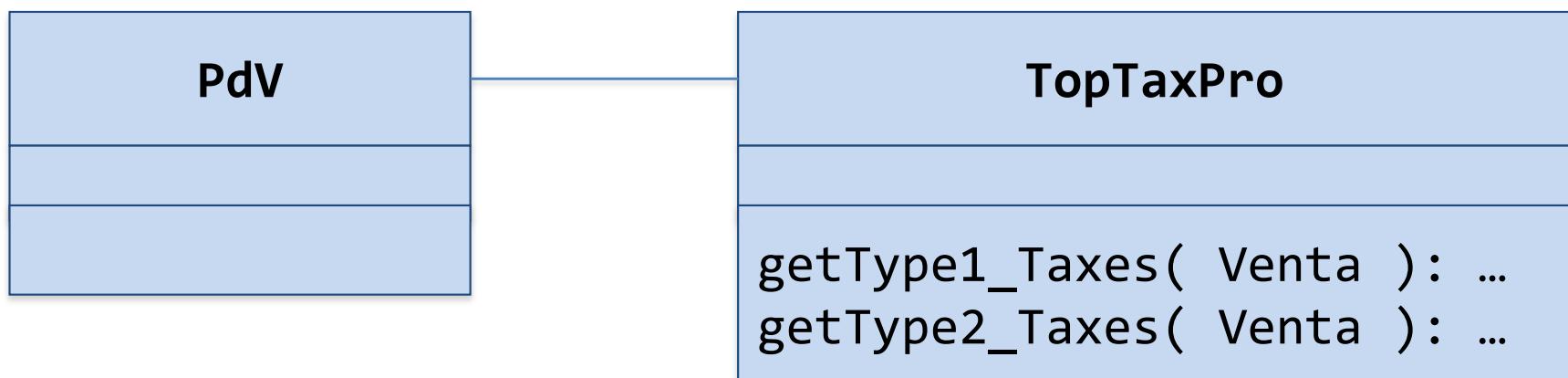
El código del servicio externo no cambia

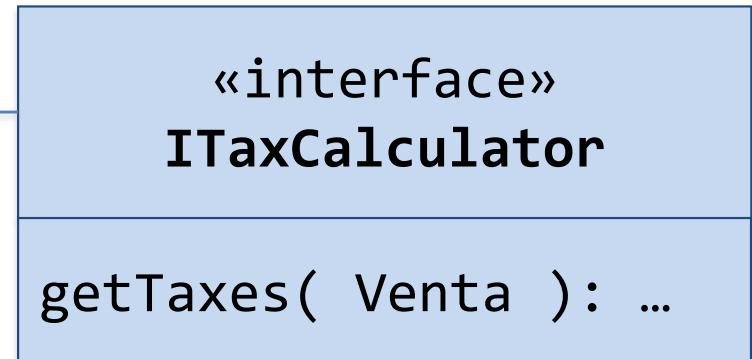
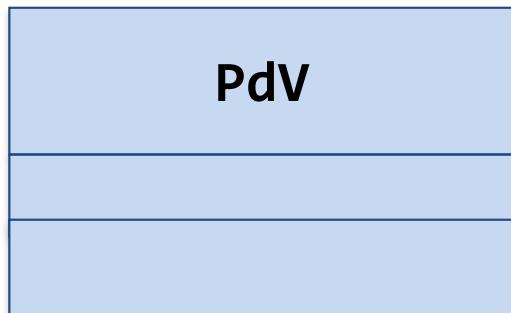


El nuevo código —lo que cambia— se centraliza en el adaptador



El sistema PdV tiene que cambiar los métodos a los que llama cada vez que cambia el calculador de impuestos: si está conectado a TaxMaster, tiene que llamar a `computeTaxes()`, pero si está conectado a TopTaxPro, tiene que llamar tanto a `getType1_taxes()` como a `getType2_Taxes()`

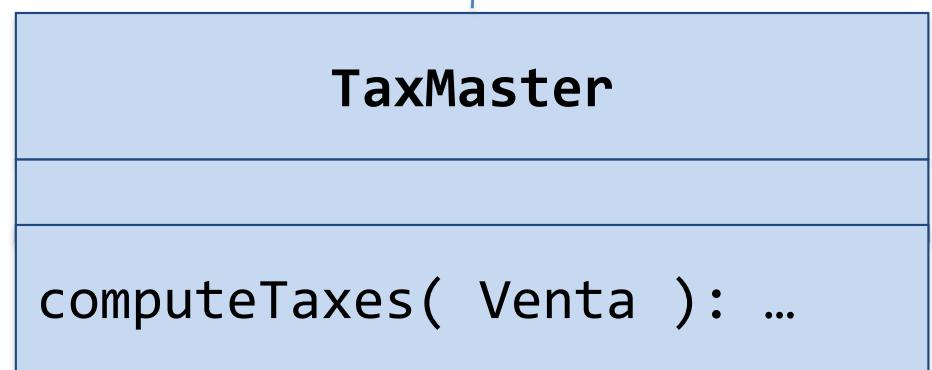
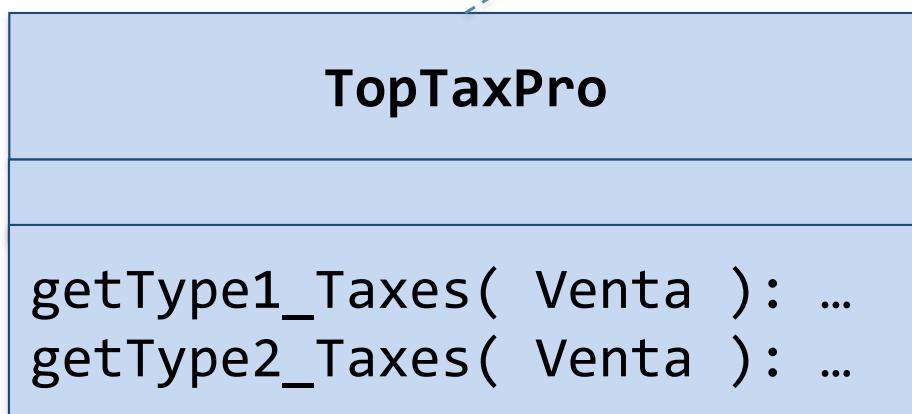


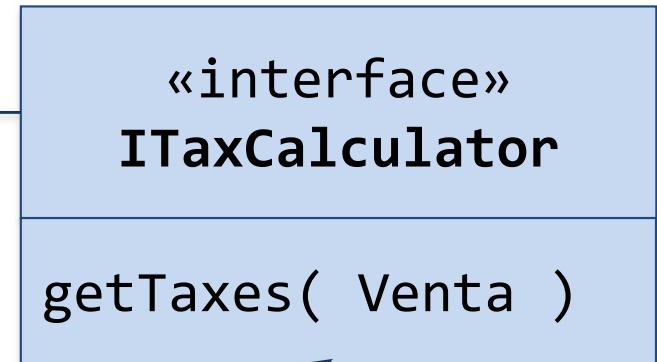


¿Cómo conseguimos que PdV pueda usar siempre la misma interfaz ITaxCalculator y llamar sólo al método getTaxes()?

¿?

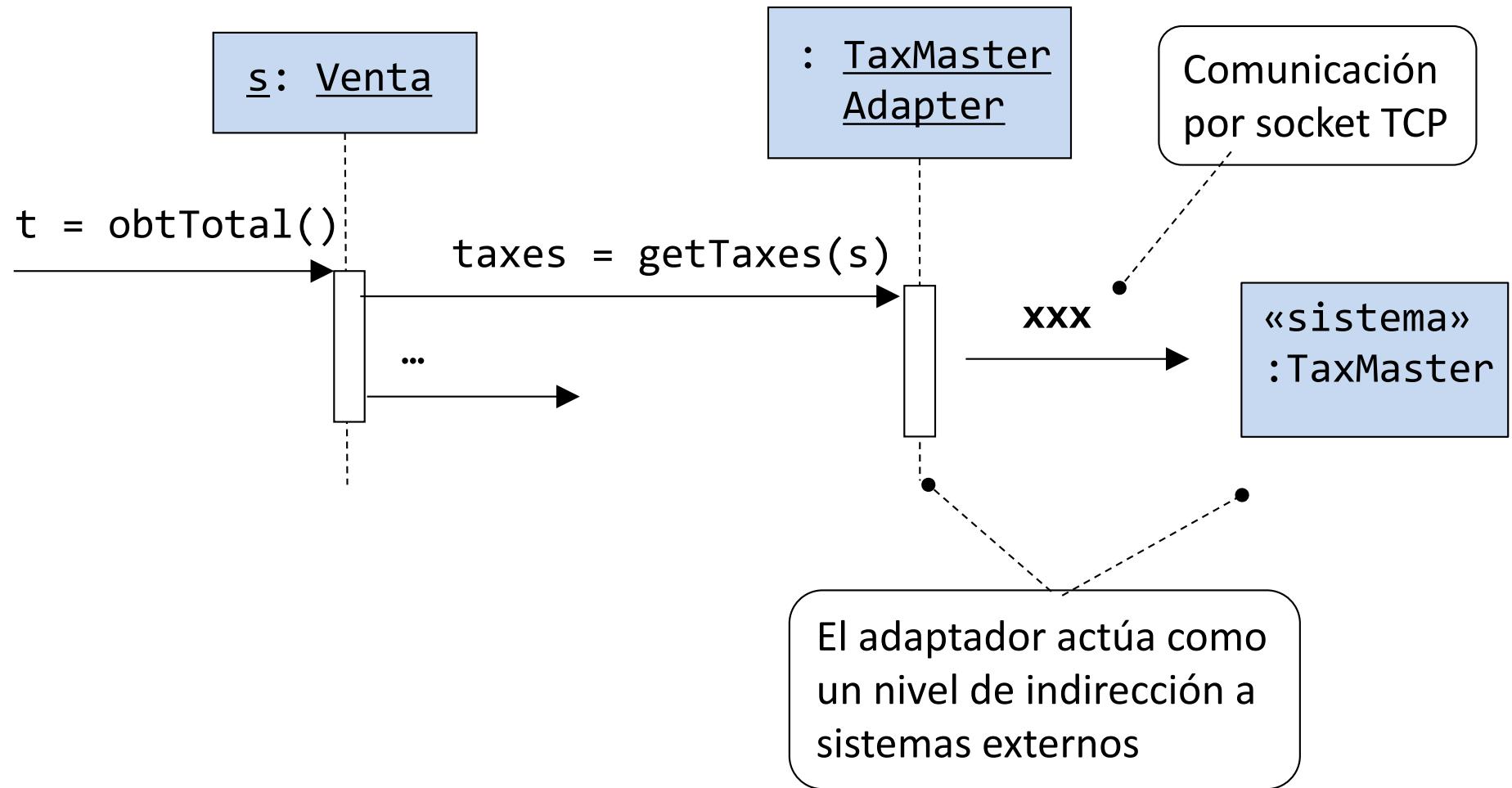
¿?





TopTaxProAdapter y TaxMasterAdapter implementan la interfaz ITaxCalculator y la *adaptan* a TopTaxPro y TaxMaster, respectivamente

# Uso del adaptador TaxMasterAdapter



# El patrón de diseño *Adaptador*

**Problema:** ¿Cómo conciliar interfaces incompatibles?

... ¿Cómo proporcionar una interfaz estable a componentes similares, pero con interfaces diferentes?

**Solución:** Convertir la interfaz original de un componente en otra interfaz,

... a través de un **objeto adaptador intermediario**

En el caso de *The Trivium*, ¿ómo permitimos variaciones en la API de las tarjetas?

Queremos permitir cambiar los temas, y las preguntas y respuestas:

- las preguntas de “The Trivium” original; las preguntas de “The Trivium —The Rolling Stones”; las preguntas de “The Trivium —los 90’s”; etc.
- distintos proveedores de preguntas podrían ofrecer conjuntos de tarjetas con distintas api’s —los métodos para tener acceso a las preguntas y respuestas pueden llamarse diferentemente
- no es conveniente que las casillas tengan que cambiar su manera de tener acceso a las tarjetas para cada nuevo conjunto de tarjetas

## Tarjeta (original)

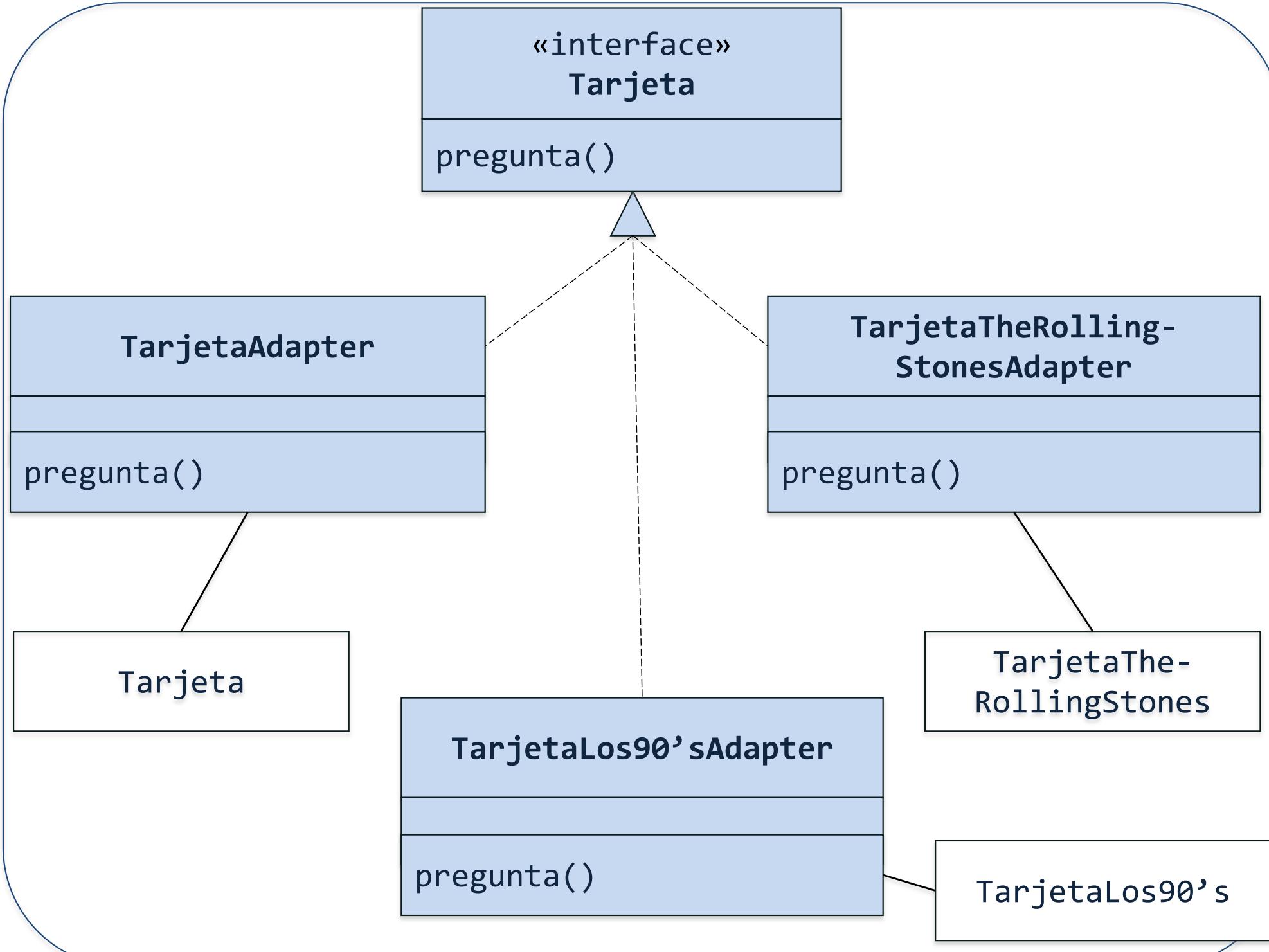
```
obtenerPregunta() {  
    —devuelve pregunta y respuesta ...}
```

## TarjetaTheRollingStones

```
pregunta() {  
    —devuelve pregunta ... }  
respuesta() {  
    —devuelve respuesta ... }
```

## TarjetaLos90's

```
pregunta&respuesta() {  
    —devuelve pregunta y respuesta ... }
```



## 2

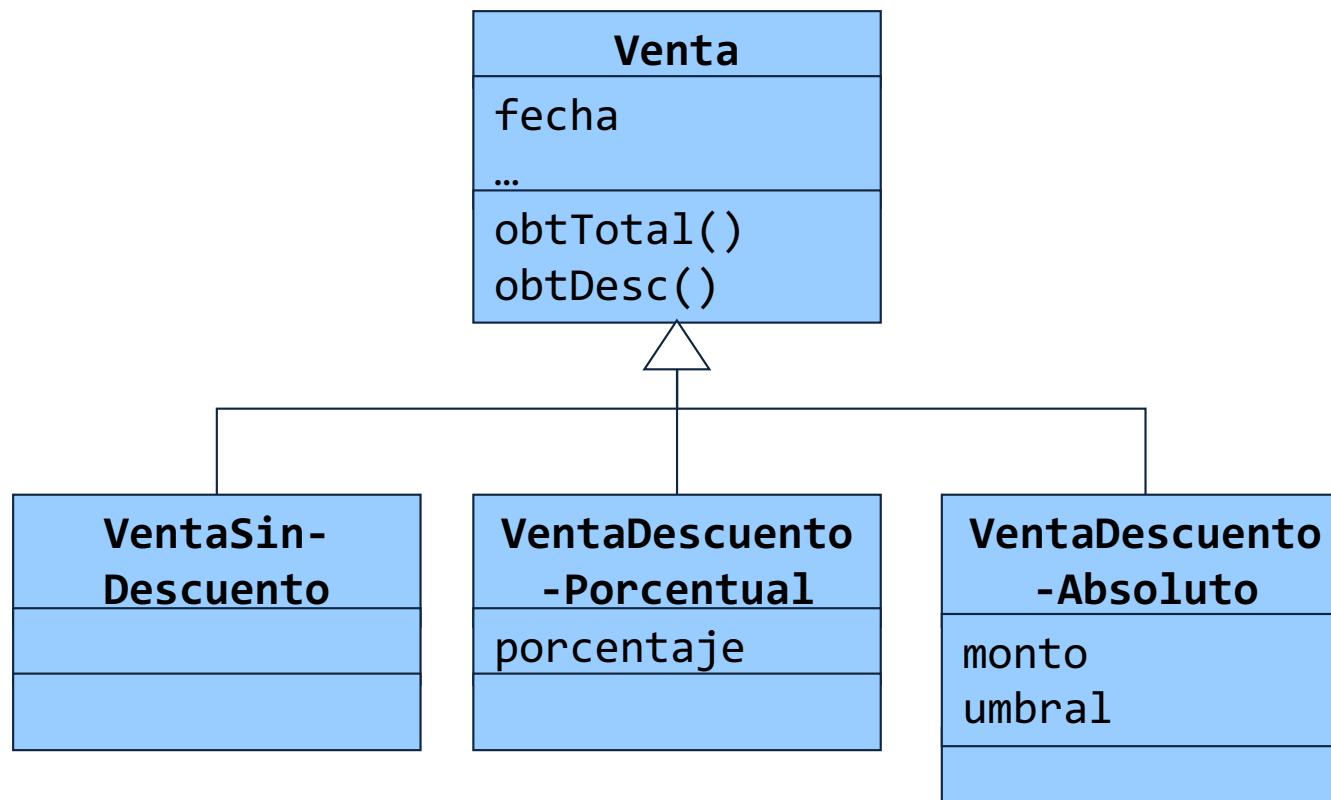
# El sistema PdV y las políticas de descuento

En el sistema PdV, la política de descuento para una venta puede variar:

- durante cierto período, puede aplicarse un descuento del 10% sobre el total de la venta
- más tarde, puede descontarse un monto fijo, p.ej., \$5000 o \$10000, si el total de la venta supera un cierto umbral
- finalmente, no hay descuentos

Una posibilidad sería usar herencia:

- hay tres tipos de ventas
- definimos una subclase para cada una
- pero ... ¿qué hacemos con el método obtDesc() en la clase VentaSinDescuento?



El sistema PdV sólo sabe de la clase Venta:

- sólo puede llamar a los métodos definidos en la clase Venta y que son comunes a todos los tipos de ventas
- en la práctica, va a estar aplicando el método a una venta de un tipo particular (ya sea con descuento porcentual, o con descuento absoluto, o sin descuento, etc.)
- si dos tipos particulares de venta comparten un método, p.ej., `obtDesc()`, que otro tipo de venta no comparte, este método debe escribirse en cada tipo de venta que lo comparte y no en la clase Venta
- si se hace cualquier cambio en la clase Venta, p.ej., si se agrega un método, el cambio afecta a todos los tipos particulares de venta

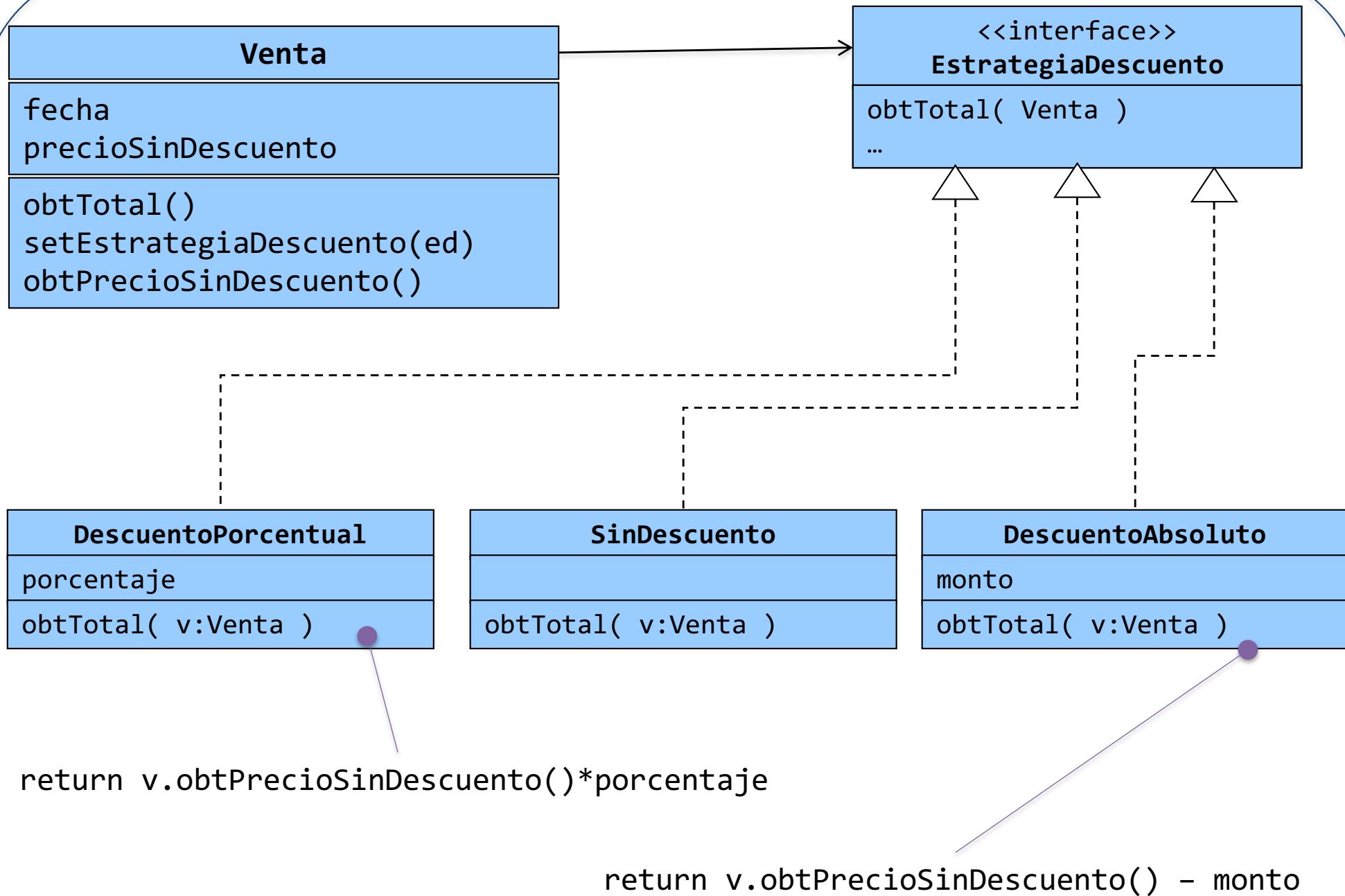
La herencia no permite flexibilidad en los cambios y duplica código

## Solución: Separar las políticas de descuento de las ventas propiamente tales

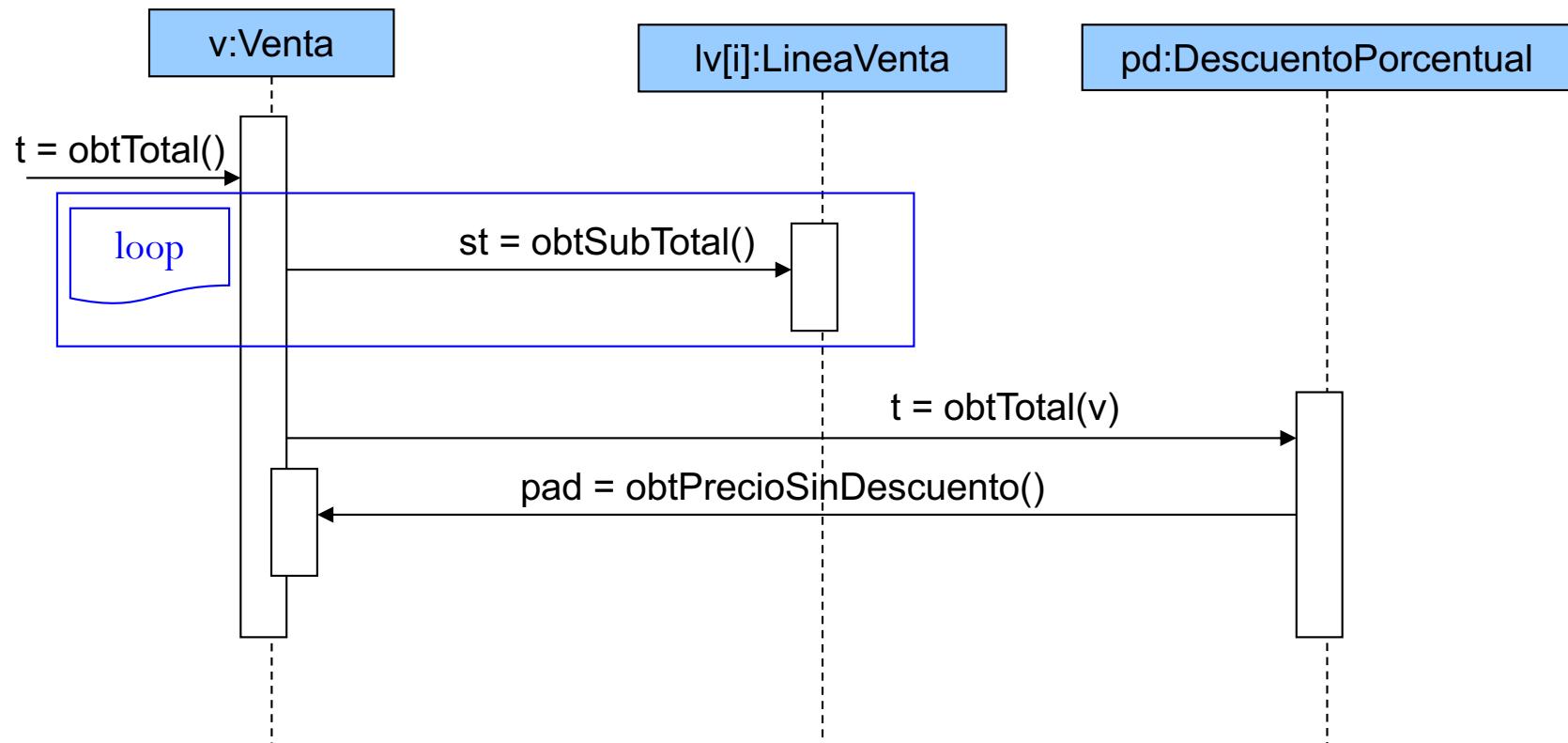
Creamos una *interfaz* («interface») que representa la *estrategia de descuento*

... asociamos las ventas a esta interfaz

... y hacemos que todas las estrategias de descuento concretas implementen la interfaz



Uso de la estrategia: En la práctica, la venta se comunica directamente con una estrategia de descuento particular



# El patrón de diseño *Estrategia*

**Problema:** ¿Cómo diseñar para permitir algoritmos, o políticas, diferentes pero relacionados?

... ¿Cómo diseñar para tener la facilidad de cambiar estos algoritmos o políticas?

**Solución:** Definamos cada algoritmo, política o estrategia en una clase separada,

... con una **interfaz común**

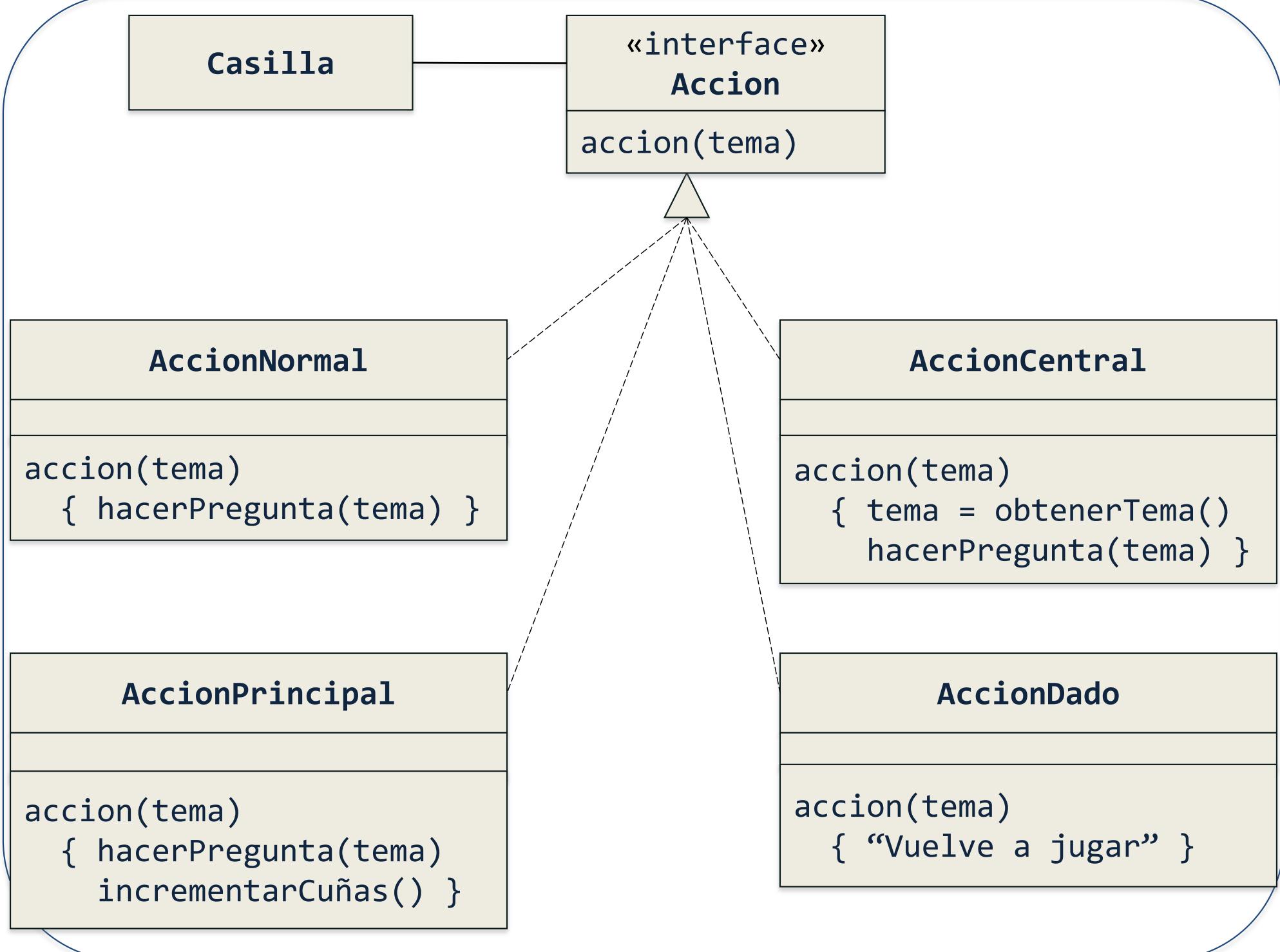
Un **objeto estrategia** (p.ej., una estrategia de descuento) está asociado a un **objeto de contexto** —el objeto al que se le aplica la estrategia (o algoritmo), en el ejemplo, una Venta

Cuando este objeto de contexto recibe un mensaje particular, delega parte del trabajo que tiene que hacer a su objeto estrategia

Es común —y usualmente necesario— que el objeto de contexto pase una referencia a sí mismo al objeto estrategia

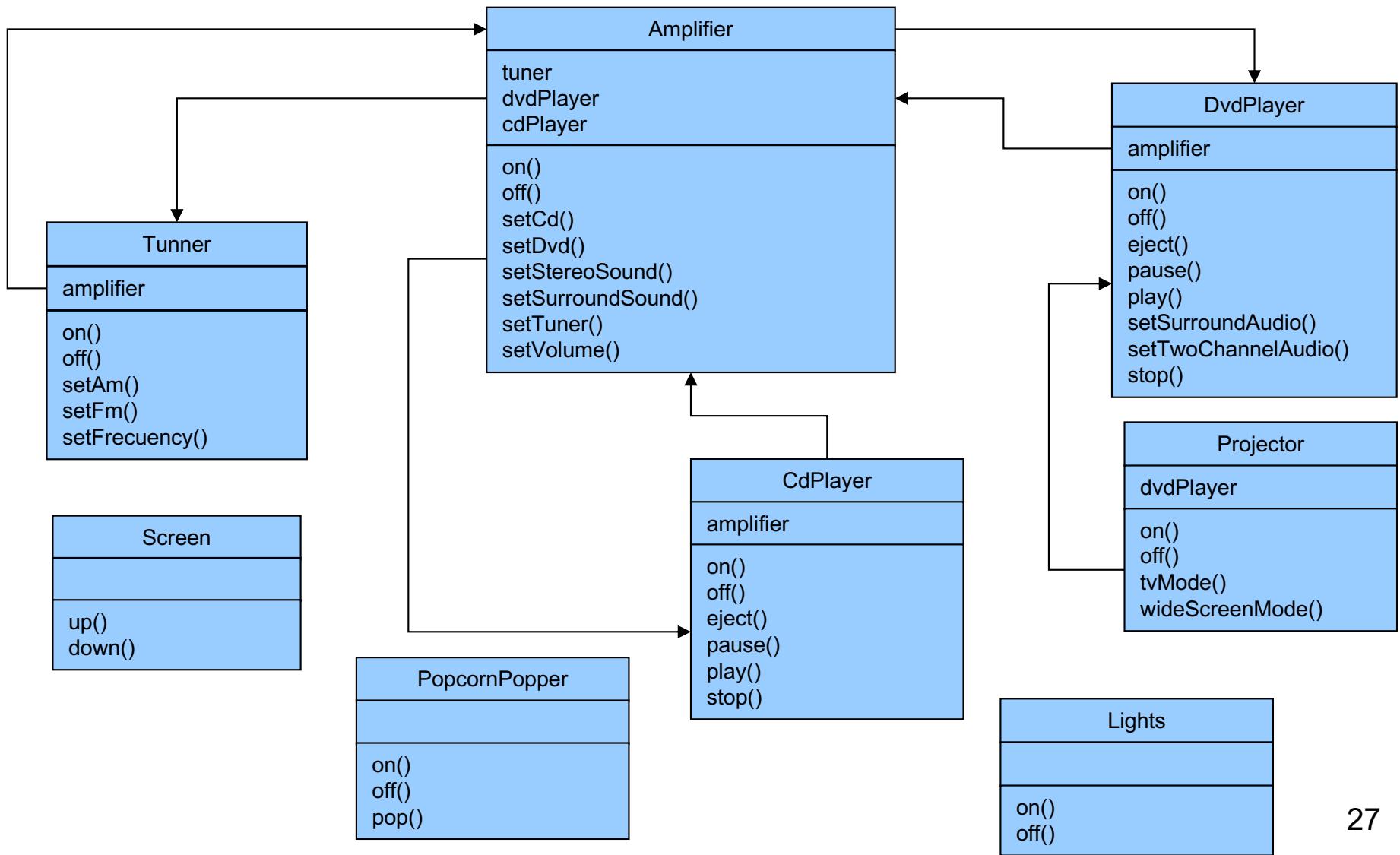
En el caso de *The Trivium*, ¿cómo enfrentamos la variabilidad en la conducta de las casillas?

- las casillas *normales*: hacen una pregunta, dependiendo del tema que les corresponde; si el jugador contesta correctamente, juega de nuevo
- las casillas *principales*: hacen una pregunta, dependiendo del tema que les corresponde; si el jugador contesta correctamente, gana una cuña y juega de nuevo
- las casillas “*dados*”: no hacen preguntas; el jugador simplemente juega de nuevo
- la casilla *central*: hace una pregunta, pero el tema lo elige el jugador, si tiene 5 o menos cuñas, o sus contrincantes, si el jugador ya tiene las 6 cuñas



## 3

## El caso de “cine en tu casa”

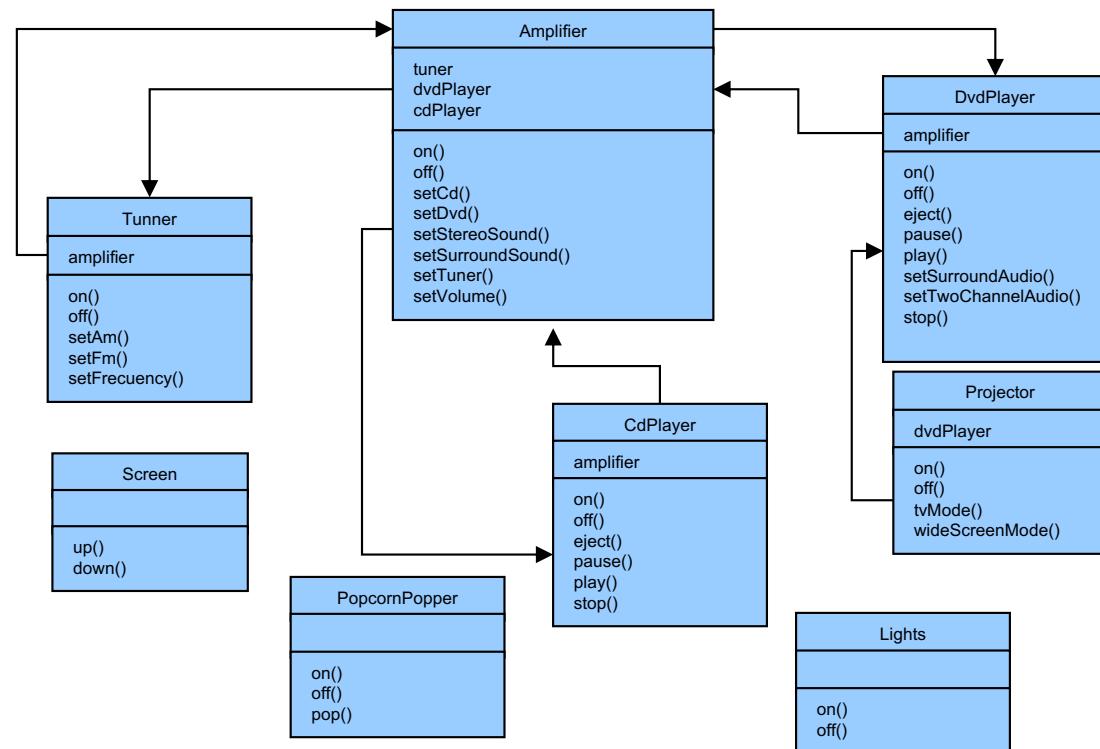


## Observaciones:

- hay muchas clases
- hay diferentes interacciones entre las clases
- cada clase tiene su interfaz que hay que usar correctamente

## Problema:

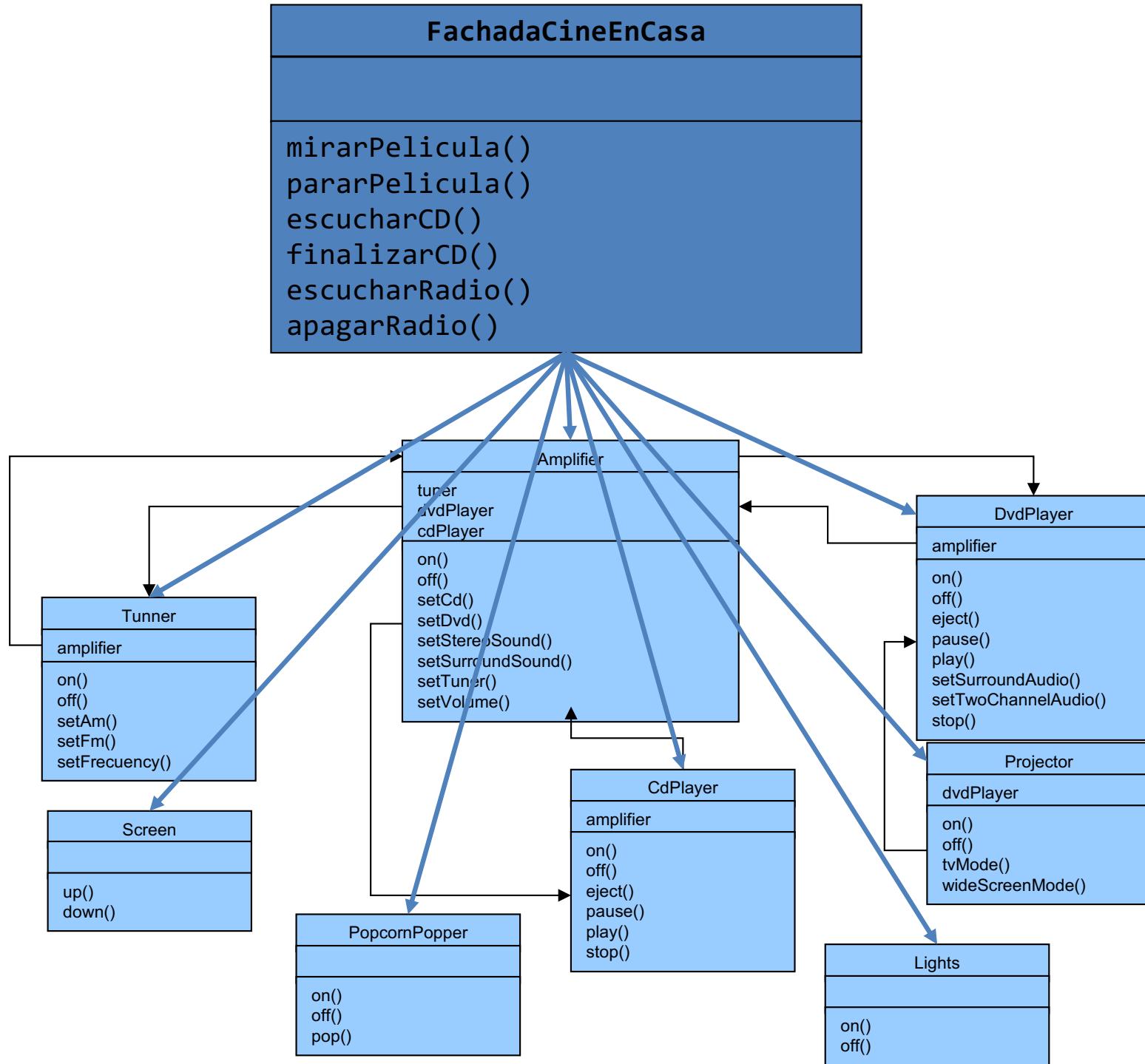
- ¿Cómo simplificar el uso del sistema integralmente?



## Solución: El control remoto universal

Para gestionar todos estos dispositivos necesitaríamos un único “mando a distancia” que nos permitiera

- mirar una película
- escuchar un cd
- escuchar la radio



## Propiedades:

- proporciona una interfaz unificada hacia un conjunto de interfaces de varios subsistemas
- define una interfaz de alto nivel que facilita el uso de los subsistemas

## Principio de diseño:

- “encapsular lo que cambia”

## Beneficios:

- el cliente se mantiene simple y flexible
- se puede actualizar los subsistemas sin afectar al cliente

# El patrón de diseño *Fachada*

**Problema:** Unificar (y simplificar) el acceso a un conjunto de subsistemas

... considerando que la implementación de los subsistemas puede cambiar

... y que se quiere tener un bajo acoplamiento con ellos

**Solución:** Definir un único punto de contacto mediante un objeto — una *Fachada*— que encapsule los subsistemas

... y que tenga la responsabilidad de colaborar con los componentes de los subsistemas

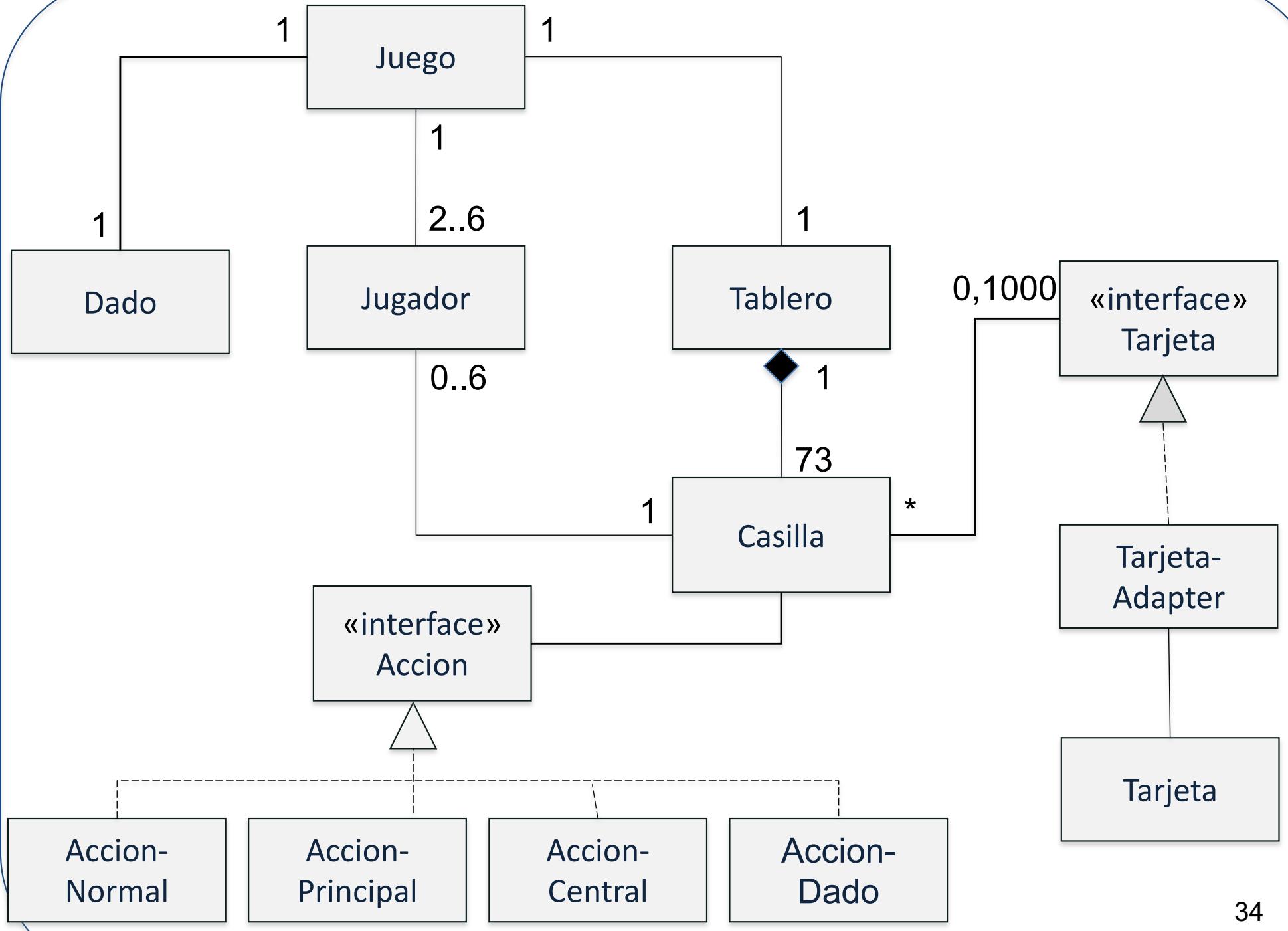
## El :Juego es una *fachada* en “The Trivium”

El mensaje `lanzarDado()`, generado por el jugador, podría ir directamente desde la gui al :Dado, en lugar de pasar por el :Juego

El mensaje `moverFicha()`, generado por el jugador, podría ir directamente desde la gui al :Tablero, en lugar de pasar por el :Juego

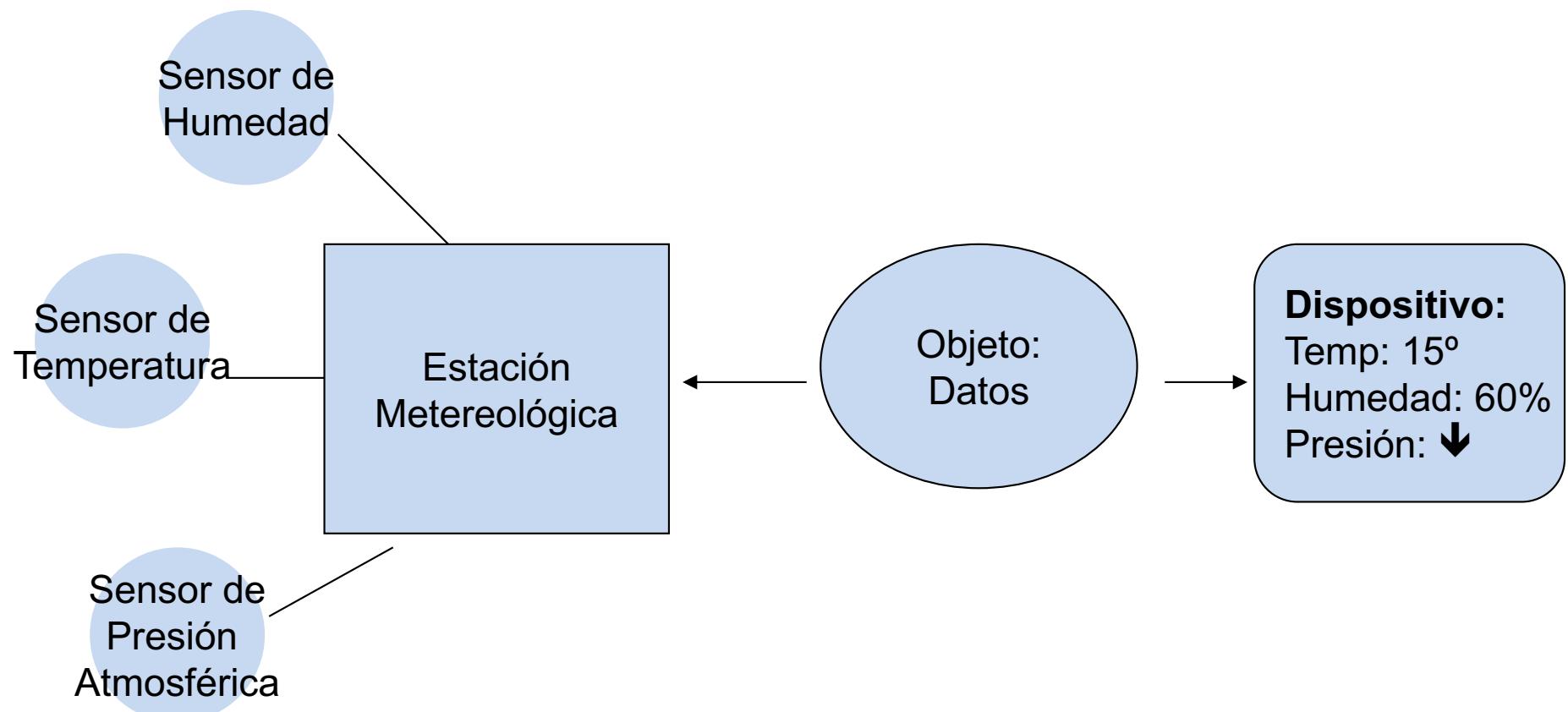
Otros mensajes del jugador que podrían ir directamente desde la GUI a algún otro objeto de la aplicación:

p.ej., cuando el jugador pide mostrar la respuesta correcta o cuando avisa que su respuesta es incorrecta, el mensaje podría ir directamente a la :Casilla o a la :Tarjeta



## 4

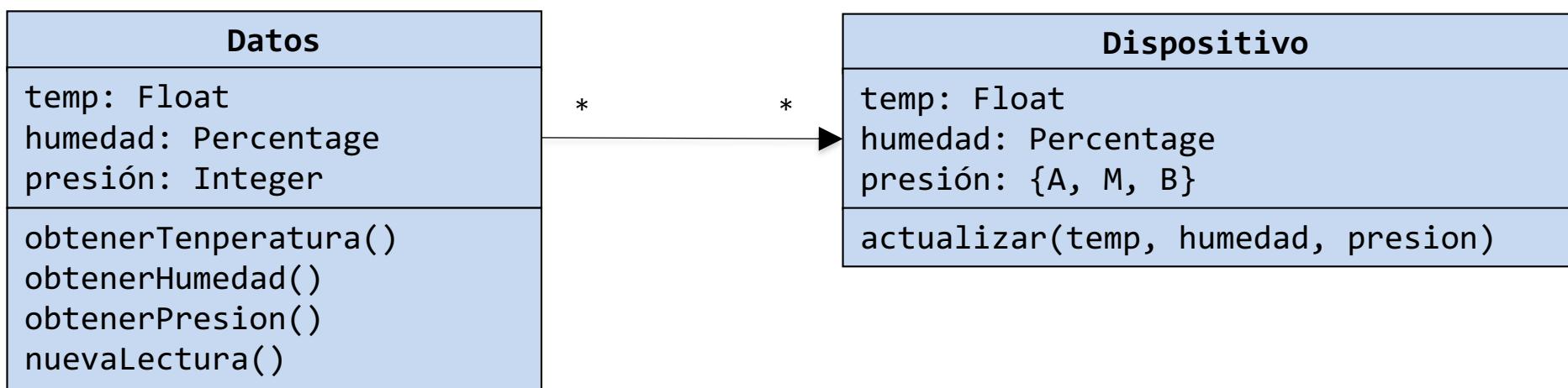
# La app del informe del tiempo



*este no es un diagrama UML*

La clase **Datos** tiene asociada la clase **Dispositivo**:

- **Datos** tiene la operación **nuevaLectura()**
- **Dispositivo** tiene la operación **actualizar(...)**
- cada vez que el método **nuevaLectura()** es llamado (no sabemos ni nos preocupa cómo), el objeto de tipo **Datos** primero realiza una nueva lectura de la temperatura, humedad y presión
- ... y luego llama a **actualizar()** del dispositivo



primero, leemos los valores más recientes de temperatura, humedad y presión

```
class Datos:  
    Dispositivo dispActual  
  
    nuevaLectura():  
        temp = obtenerTemperatura()  
        humedad = obtenerHumedad()  
        presión = obtenerPresión()  
        dispActual.actualizar(temp, humedad, presión)
```

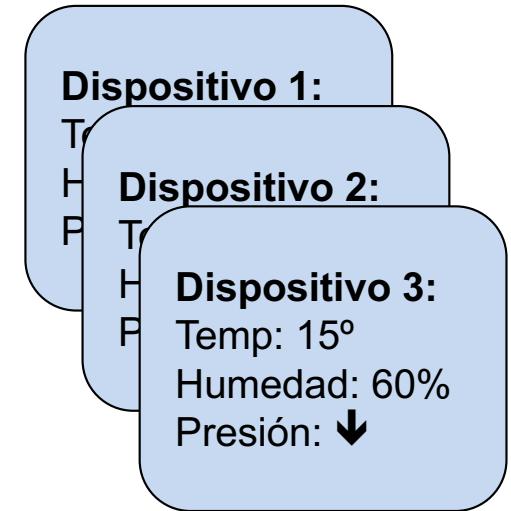
... y luego actualizamos el dispositivo

**Dispositivo:**  
Temp: 15°  
Humedad: 60%  
Presión: ↓

¿Y si fuera necesario actualizar más dispositivos?

tres dispositivos de distintos tipos (distintas clases)

```
class Datos:  
    Dispositivo dispActual  
    NuevoDispositivo dispActual2  
    OtroDispositivo dispActual3  
  
    nuevaLectura():  
        temp = obtenerTemperatura()  
        humedad = obtenerHumedad()  
        presión = obtenerPresión()  
  
        dispActual.actualizar(temp,humedad,presión)  
        dispActual2.actualizar(temp,humedad,presión)  
        dispActual3.actualizar(temp,humedad,presión)
```



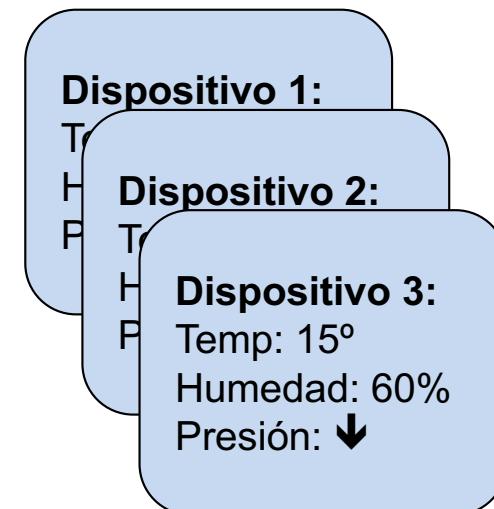
al menos todos tienen un método `actualizar(...)` con los mismos parámetros (pero podría no haber sido así)

## Observaciones:

- los dispositivos reciben los datos cuando los sensores actualizan sus valores
- en el futuro, podría ser necesario añadir nuevos dispositivos
- estos nuevos dispositivos podrían ser diferentes
- se podría necesitar añadir/quitar los dispositivos en tiempo de ejecución
- cada dispositivo podría necesitar datos diferentes

## Problema:

- la gestión de dispositivos se realiza en el código de la clase **Datos** y no es flexible



```
class Datos:  
    Dispositivo dispActual  
NuevoDispositivo dispActual2  
OtroDispositivo dispActual3  
  
    nuevaLectura():  
        temp = obtenerTemperatura()  
        humedad = obtenerHumedad()  
        presión = obtenerPresión()  
  
        dispActual.actualizar(temp, humedad, presión)  
dispActual2.actualizar(temp, humedad, presión)  
dispActual3.actualizar(temp, humedad, presión)
```

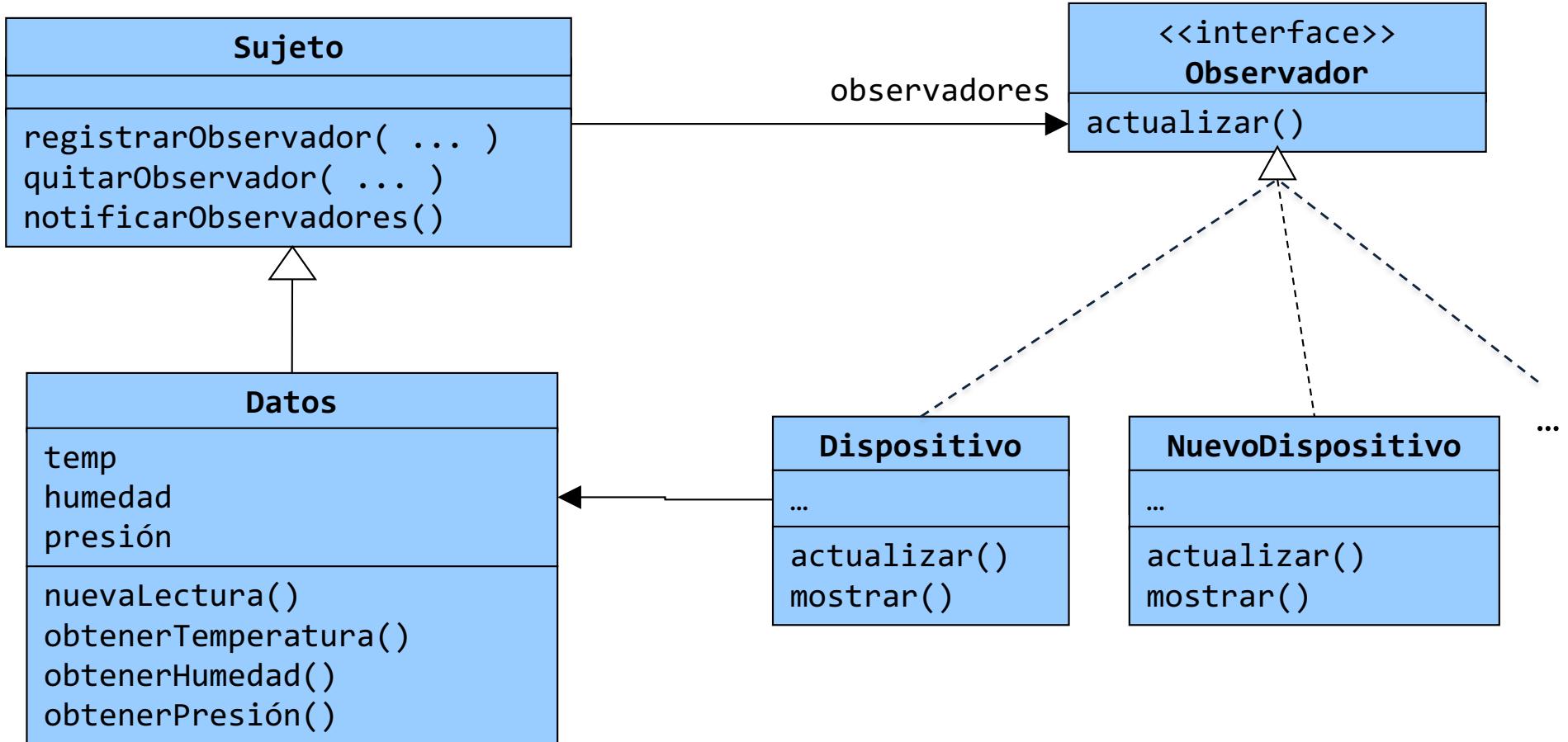
código escrito con respecto a implementaciones concretas;  
no hay cómo añadir o quitar un dispositivo sin tener que hacerle  
cambios al programa

área de cambios;  
hay que encapsularla

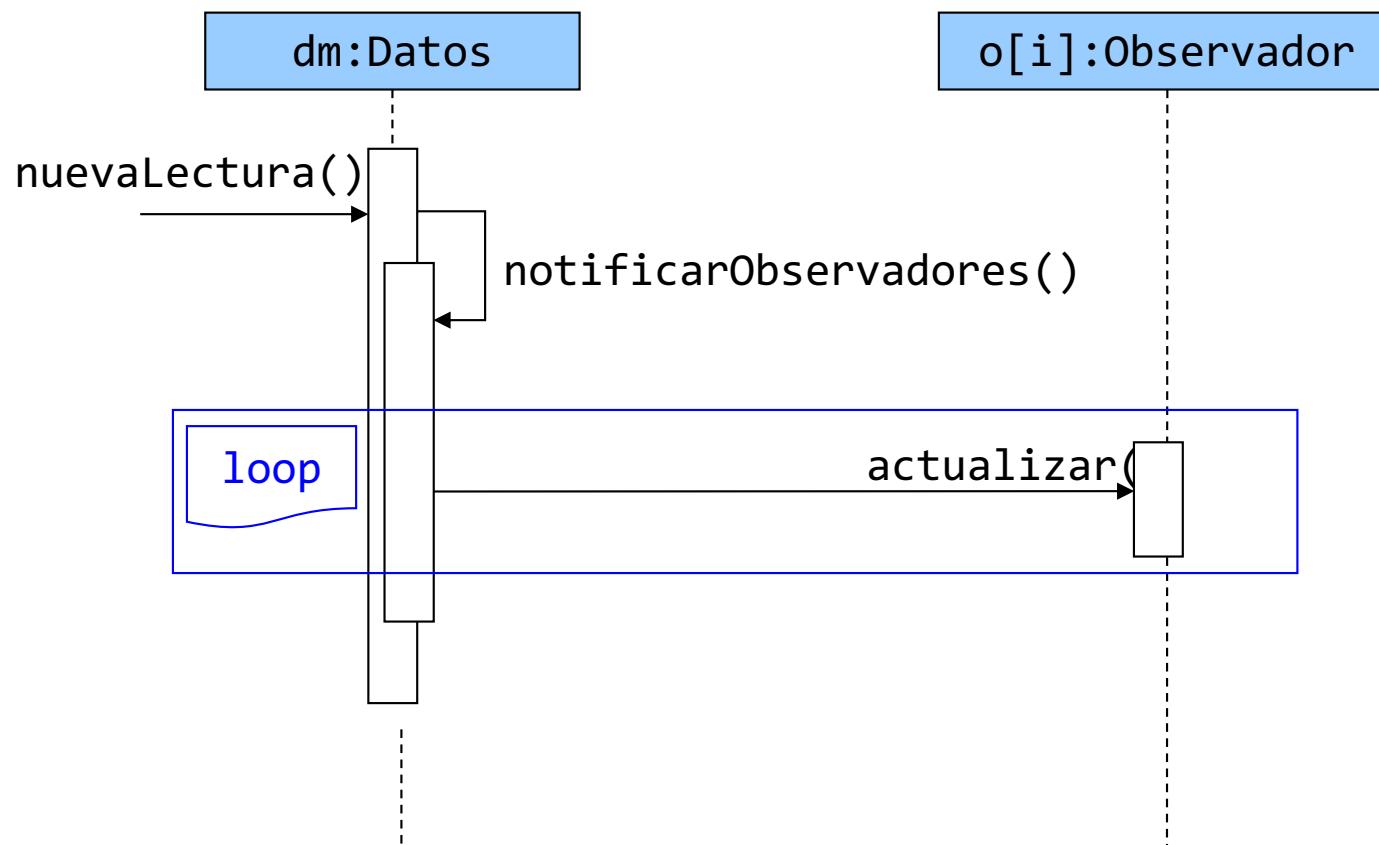


Una solución es emplear el mecanismo de *publicación y suscripción*:

- un objeto *publica* la información —el **Sujeto**
- otros objetos *se subscriben* a la información —los observadores, que implementan la interfaz **Observador**
- un observador, p.ej., un :Dispositivo, se suscribe al sujeto, p.ej., un :Datos, llamando a la operación **registrarObservador()** de la clase **Datos** (heredada de **Sujeto**)
- cuando :Datos quiere avisar a sus suscriptores de un cambio en su estado, ejecuta **notificarObservadores()**, que recorre la lista de observadores suscritos y para cada uno ejecuta polimórficamente la operación **actualizar()**
- un observador también puede “desuscribirse” (terminar su suscripción) llamando a la operación **quitarObservador()** del sujeto



Cada vez que el objeto :Datos recibe un mensaje con un nuevo de valor de temperatura, humedad o presión, avisa a todos sus suscriptores/observadores



## Propiedad:

- define una dependencia entre objetos —los observadores dependen del sujeto— tal que cuando un objeto (sujeto observable) cambia su estado, sus observadores son notificados y actualizados automáticamente

## Principios de diseño:

- polimorfismo
- variaciones protegidas: el sujeto no sabe ni la clase ni el número de sus observadores

## Beneficios:

- los observadores están poco acoplados con el sujeto observado

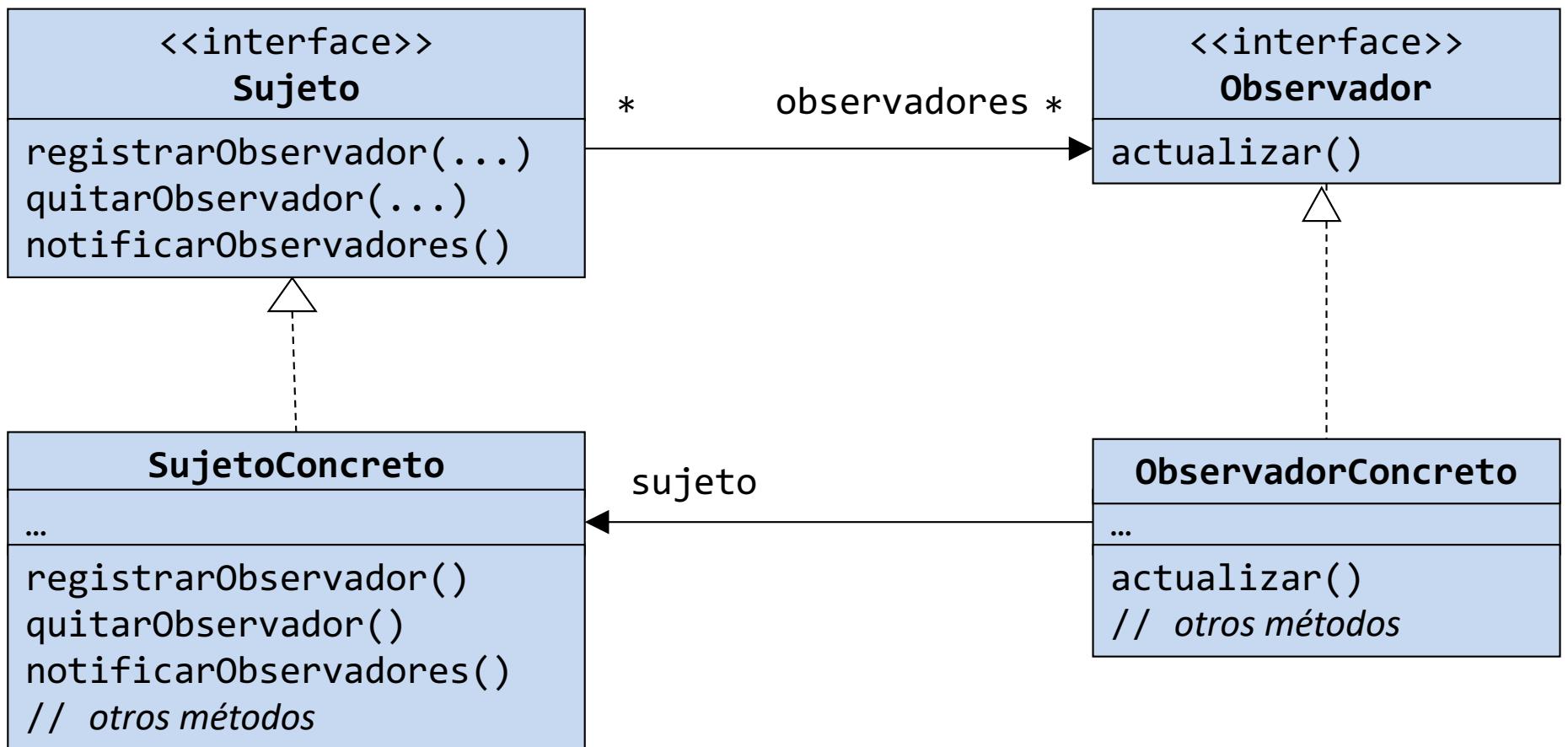
# El patrón de diseño ***Observador***

**Problema:** Diferentes objetos (observadores) están interesados en los cambios producidos en otro objeto (sujeto);

... el sujeto, o publicador, quiere tener poco acoplamiento con sus suscriptores (observadores)

**Solución:** Definir una interfaz Observador que sea implementada por los diferentes observadores o suscriptores;

... los observadores pueden subscribirse para ser notificados cuando ocurre un evento



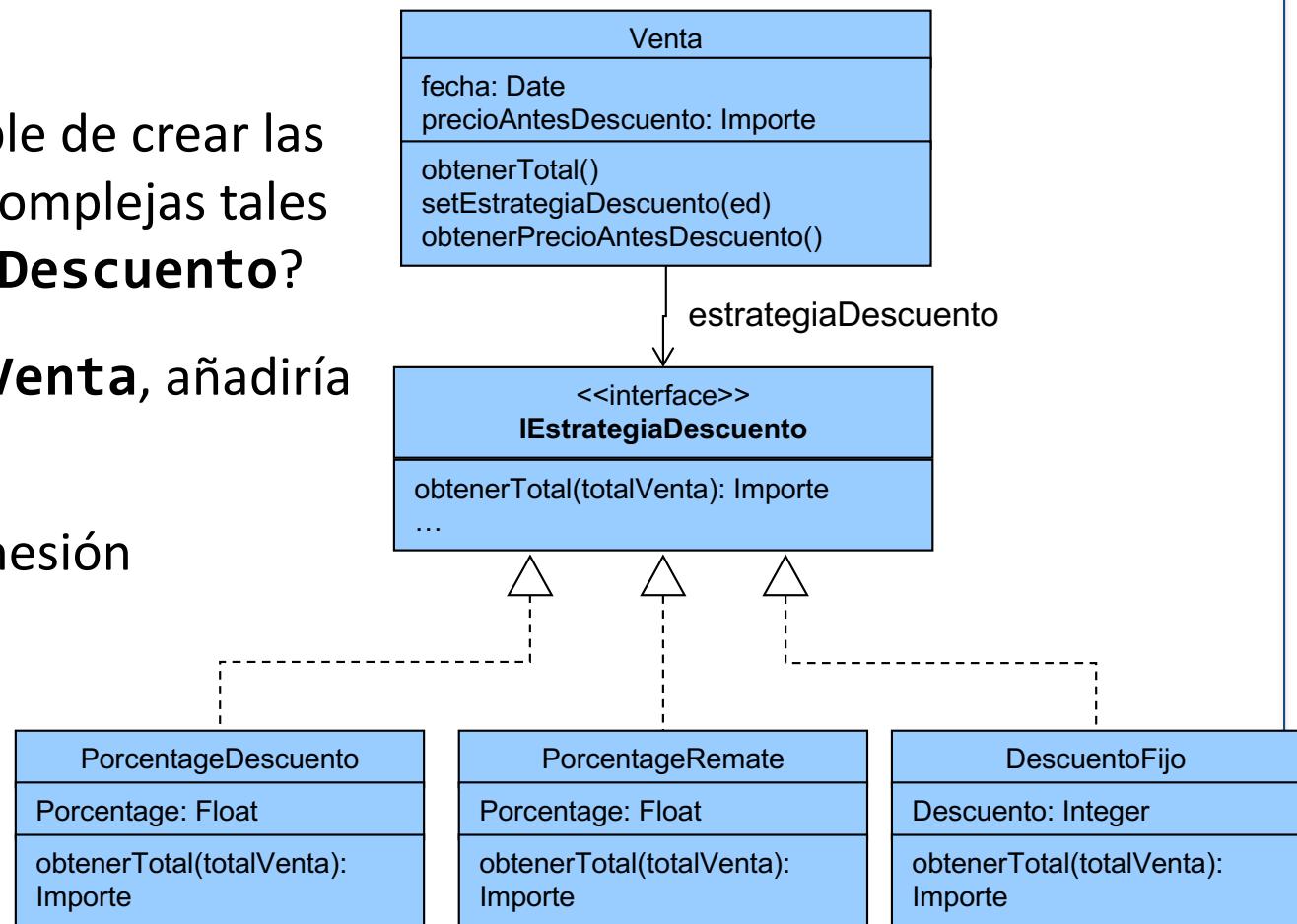
# 5

## Construcción de Estrategia-Descuento

¿ Quién es responsable de crear las instancias de clases complejas tales como **Estrategia-Descuento**?

Asignarla al experto **Venta**, añadiría acoplamiento

... y disminuiría la cohesión



Solucionamos este problema mediante una fábrica (o *fábrica*) de objetos complejos

**FabricaEstrategiaDescuento:**

- es responsable de crear todos los objetos, que respondan a la interfaz `IESTRATEGIADESCUENTO`, que necesite la aplicación
- toma el nombre de la clase que va a construir de las propiedades del sistema (o fuente de datos externa), ya que `IESTRATEGIADESCUENTO` es una interfaz

### **FabricaEstrategiaDescuento**

`getDescuentoPorcentual(): IESTRATEGIADESCUENTO`  
`getDescuentoFijo(): IESTRATEGIADESCUENTO`  
`getDescuentoRemate(): IESTRATEGIADESCUENTO`

## Tipos de fábricas:

- *Abstracta*: Proporciona una interfaz para construir familias de objetos (relacionados o dependientes), sin especificar sus clases concretas; se apoya en clases de fábricas concretas, que implementan la interfaz
- *Concreta*: Proporciona una interfaz para construir un objeto, y entrega a las clases que implementan la interfaz la decisión de qué clases instanciar

## Principios de diseño:

- **Experto**, aunque se inventa una clase experta específica

## Propiedades:

- separa la responsabilidad de una creación compleja en objetos cohesivos
- esconde una lógica de creación que puede ser compleja

# El patrón de diseño *Fábrica*

**Problema:** ¿Quién es el responsable de construir objetos cuando

- ... los objetos son especiales? (la lógica de creación es compleja, hay varias clases involucradas, etc.)
- ... queremos separar la responsabilidad de la construcción? (para mejorar la cohesión)

**Solución:** Creamos un nuevo objeto —una *Fábrica*— cuyo único propósito es construir otros objetos

En el caso de los adaptadores (por aplicación del patrón *Adaptador*):

- ¿quién los crea?
- ¿y cómo determinamos qué clase de adaptador crear?

Si ponemos la responsabilidad de creación en un objeto del dominio, este objeto perderá cohesión al tener ahora responsabilidades —relativas a conectividad con componentes externas— que van más allá de la lógica de la aplicación

Además, por su propia naturaleza, los adaptadores pueden ser de diversos tipos —según las particularidades de los objetos que adaptan— e ir apareciendo y desapareciendo con el tiempo

Empleamos el patrón *Fábrica*, que nos sugiere definir un objeto *FabricadeAdaptadores*, constructor de adaptadores:

- su único trabajo en la vida es crear adaptadores
- separamos la responsabilidad de una creación compleja poniéndola en objetos cohesivos especializados en creación
- escondemos la lógica potencialmente compleja de creación
- permitimos un manejo más eficiente de la memoria, p.ej., reciclando objetos o usando una cache de objetos

```
class FabricadeAdaptadores:

    Adaptador construirAdaptador(string tipo):

        Adaptador adaptador = null

        if tipo == “toptaxproAdapter”:
            adaptador = new TopTaxProAdapter()
        else if tipo == “taxmasterAdapter”
            adaptador = new TaxMasterAdapter()
        else if tipo == ... —adaptador para otro calculador

        return adaptador
```

```
class TiendadeAdaptadores: —este es el cliente de la fábrica  
    FabricadeAdaptadores fabrica  
  
    TiendadeAdaptadores(FabricadeAdaptadores fabrica):  
        this.fabrica = fabrica  
  
    Adaptador pedirAdaptador(string tipo):  
        Adaptador adaptador  
        adaptador = fabrica.construirAdaptador(tipo)  
        ... —una vez construido el adaptador, llamamos a los  
(posibles) métodos de la interfaz Adaptador (p.ej., los métodos de la  
interfaz ITaxCalculator en la diap.#11)  
  
        return adaptador
```

¿Y si nuestra tienda de adaptadores empieza a recibir solicitudes de adaptadores para validadores de medios de pago (además de adaptadores para calculadores de impuestos)?

Le cambiamos el nombre a nuestra fábrica de adaptadores, para que se sepa que es una fábrica de adaptadores para calculadores de impuestos

... y definimos una (nueva) fábrica de adaptadores para validadores de medios de pago

# 6

## Varios tipos de cafés y sus condimentos

El software de una tienda que vende café está estructurado como una clase abstracta **Bebida**, con un atributo, descripción, y dos métodos, obtDescripción() y precio(), todos abstractos

... de la cual heredan directamente cada una de las múltiples clases concretas que representan los diferentes tipos de cafés con los diferentes tipos de condimentos posibles (y que implementan descripción, obtDescripción() y precio())

A estas alturas, en que las combinaciones suman más de 40, esta estructura se ha convertido en una pesadilla a la hora de hacer mantenimiento, p.ej., si sube el precio de la leche semidescremada o si agregan un nuevo condimento

Una posibilidad es agregar atributos (variables de instancia) de tipo boolean a la clase abstracta **Bebida**, uno por cada condimento:

- ... junto a sus respectivos métodos get() y set()

... y hacer que el método precio() ya no sea abstracto, sino que calcule el precio correspondiente a la combinación de condimentos para una instancia particular de bebida:

- las subclases —una por cada tipo de café— igual tienen que reescribir sus propios métodos precio(),
- ... pero también van a llamar a precio() de **Bebida** para poder calcular el precio total de un café, que depende del tipo de café y del precio correspondiente a la combinación de condimentos

Ahora tenemos solo la (super)clase **Bebida** más una clase heredera por cada tipo de café, digamos menos de 10 clases en total

# El principio de diseño Abierto-Cerrado (nuestro quinto principio)

*Las clases deberían estar abiertas para extensiones, pero cerradas para modificaciones*

Clases abiertas:

- puedes extender las clases con cualquier comportamiento nuevo que te parezca
- si tus necesidades o requisitos cambian, haz tus propias extensiones

... pero al mismo tiempo cerradas:

- lo sentimos, pero hemos invertido mucho tiempo consiguiendo que nuestro código funcione correctamente y no tenga *bugs*, y no podemos permitir que tú lo cambies
- debe permanecer cerrado para cambios

La estructura del software basada en herencia no funciona bien:

- explosión de clases y diseños rígidos
- ... o funcionalidad en la clase base que no es apropiada para algunas de las subclases

En cambio, podemos tomar una bebida básica y la vamos *decorando* en tiempo de ejecución:

- p.ej., si un cliente quiere un café moca con crema
  - ... primero, tomamos un objeto de tipo **Espresso**
  - ... lo decoramos con un objeto de tipo **Moca**
  - ... lo decoramos con un objeto de tipo **Crema**
  - ... y llamamos al método `precio()` y usamos *delegación* para agregar los costos de los condimentos

¿Cómo?

Partimos con un Espresso, que hereda de Bebida, y tiene un método precio()

Luego, creamos un objeto Moca y lo usamos para cubrir (o envolver) el espresso:

- el moca es un decorador del mismo tipo que el objeto al que está decorando
- también tiene un método precio() y por polimorfismo podemos tratar cualquier bebida envuelta en un moca como una bebida

Luego, creamos un objeto Crema y lo usamos para envolver el moca:

- la crema también es un decorador del mismo tipo que el espresso e incluye un método precio()
- luego, un espresso envuelto en moca y en crema es una bebida, por lo que podemos llamar a su método precio()

## ¿Cómo calculamos el precio?

- Llamamos al método `precio()` del decorador más externo, **Crema**,
  - ... el cual delega el cálculo del precio en el objeto que está decorado —en este caso, el decorador **Mocha**
  - ... y cuando recibe un precio de vuelta, le agrega el costo de la crema

## ¿Qué hace el decorador **Mocha** cuando recibe por delegación la instrucción del cálculo de su precio?

- delega a su vez el cálculo en el objeto que está decorando —el **Espresso**
  - ... y cuando recibe la respuesta, p.ej., 1,500, le agrega su propio precio, p.ej., 200,
  - ... y devuelve  $1,500 + 200 = 1,700$  a **Crema**

Los decoradores tienen el mismo supertipo que los objetos que decoran:

- ... podemos pasar un objeto decorado en lugar del objeto original
- el decorador agrega su propio comportamiento antes y/o después de delegar al objeto decorado el resto del trabajo
- los objetos pueden ser decorados en cualquier momento, p.ej., dinámicamente en tiempo de ejecución

# El patrón *Decorador*

Agrega responsabilidades adicionales a un objeto de manera dinámica

... ofreciendo una alternativa flexible al mecanismo de subclases para extender funcionalidad

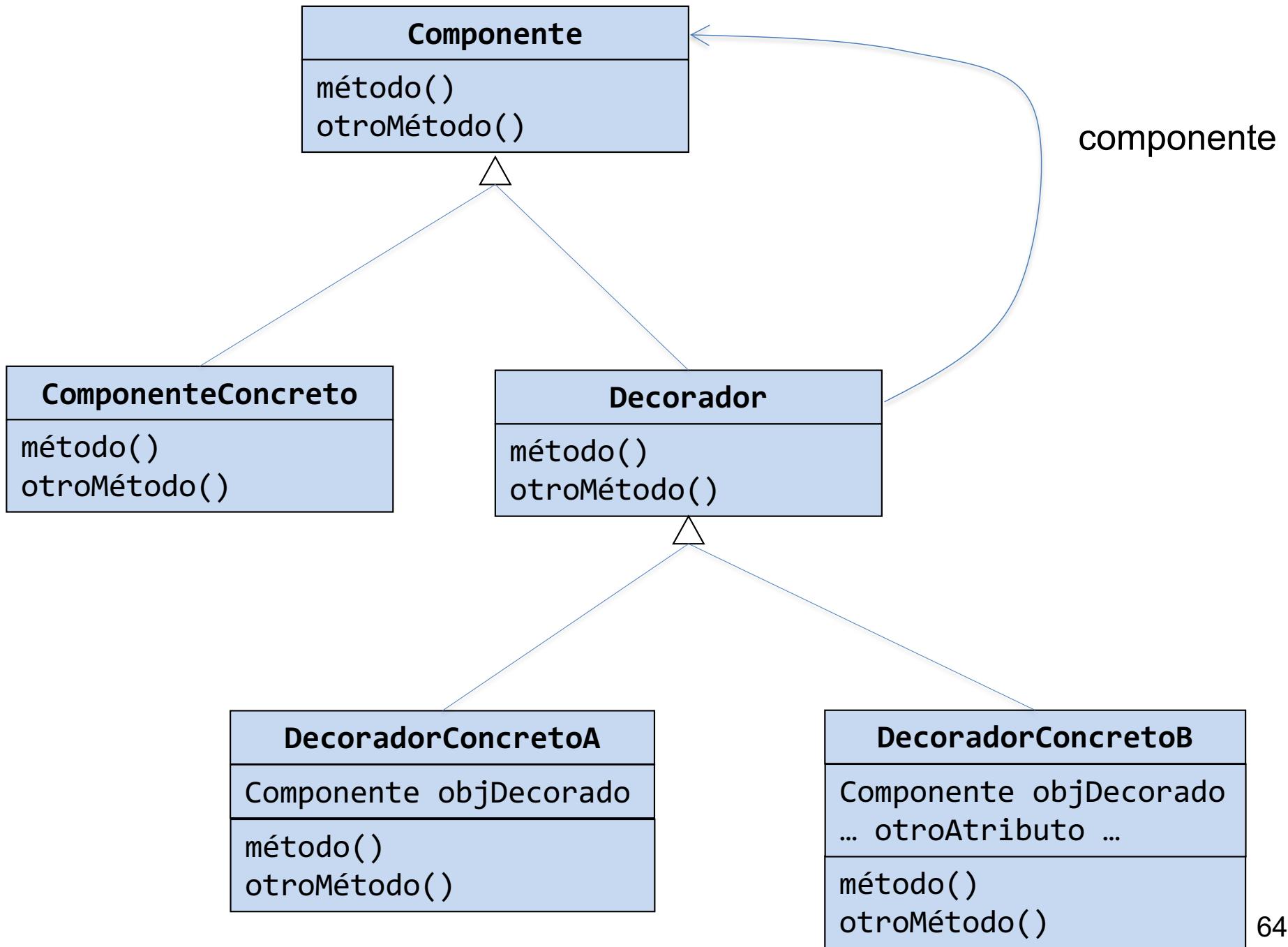
Cada componente puede ser usado individualmente, o decorado

- el componente concreto es el objeto al que vamos a agregar dinámicamente un nuevo comportamiento

Cada decorador tiene (envuelve) un componente: tiene un atributo que tiene una referencia a un componente

Los decoradores implementan la misma interfaz que el componente que van a decorar:

- los decoradores concretos tienen un atributo para el objeto que decoran



```
class Bebida:  
    string descripcion = “desconocida”  
    string obtDescripcion():  
        return descripcion  
    double precio()  
  
class Decorador extends Bebida:  
    string obtDescripcion()  
  
class Espresso extends Bebida:  
    Espresso():  
        descripcion = “espresso”  
    double precio():  
        return 1,500  
  
class Cappuccino extends Bebida:  
    Cappuccino():  
        descripcion = “cappuccino”  
    double precio():  
        return 2,000
```

```
class Mocha extends Decorador:  
    Bebida bebida  
    Mocha(Bebida bebida):  
        this.bebida = bebida  
    string obtDescripcion():  
        return bebida.obtDescripcion() + ", Mocha"  
    double precio():  
        return bebida.precio() + 200  
  
class Tienda:  
    ordenarCafés():  
        Bebida cafe1 = new Espresso()  
        print(cafe1.obtDescripcion(), cafe1.precio())  
  
        Bebida cafe2 = new Espresso()  
        cafe2 = new Mocha(cafe2)  
        print(cafe2.obtDescripcion(), cafe2.precio())  
  
        Bebida cafe3 = new Cappuccino()  
        cafe3 = new Mocha(cafe3)  
        cafe3 = new Crema(cafe3)  
        print(cafe3.obtDescripcion(), cafe3.precio())
```

Además de clasificar los patrones como *de creación*, *estructurales* o *de comportamiento*, los podemos clasificar dependiendo de si involucran clases u objetos:

Patrones relativos a clases:

- cómo se definen las relaciones entre clases vía herencia
- *Método Plantilla, Método de Fábrica, Adaptador, Intérprete*

Patrones relativos a objetos:

- relaciones entre objetos, principalmente mediante composición, típicamente creadas en tiempo de ejecución; son más dinámicas y flexibles
- todos los otros

## 7

## El patrón de diseño *Singleton*

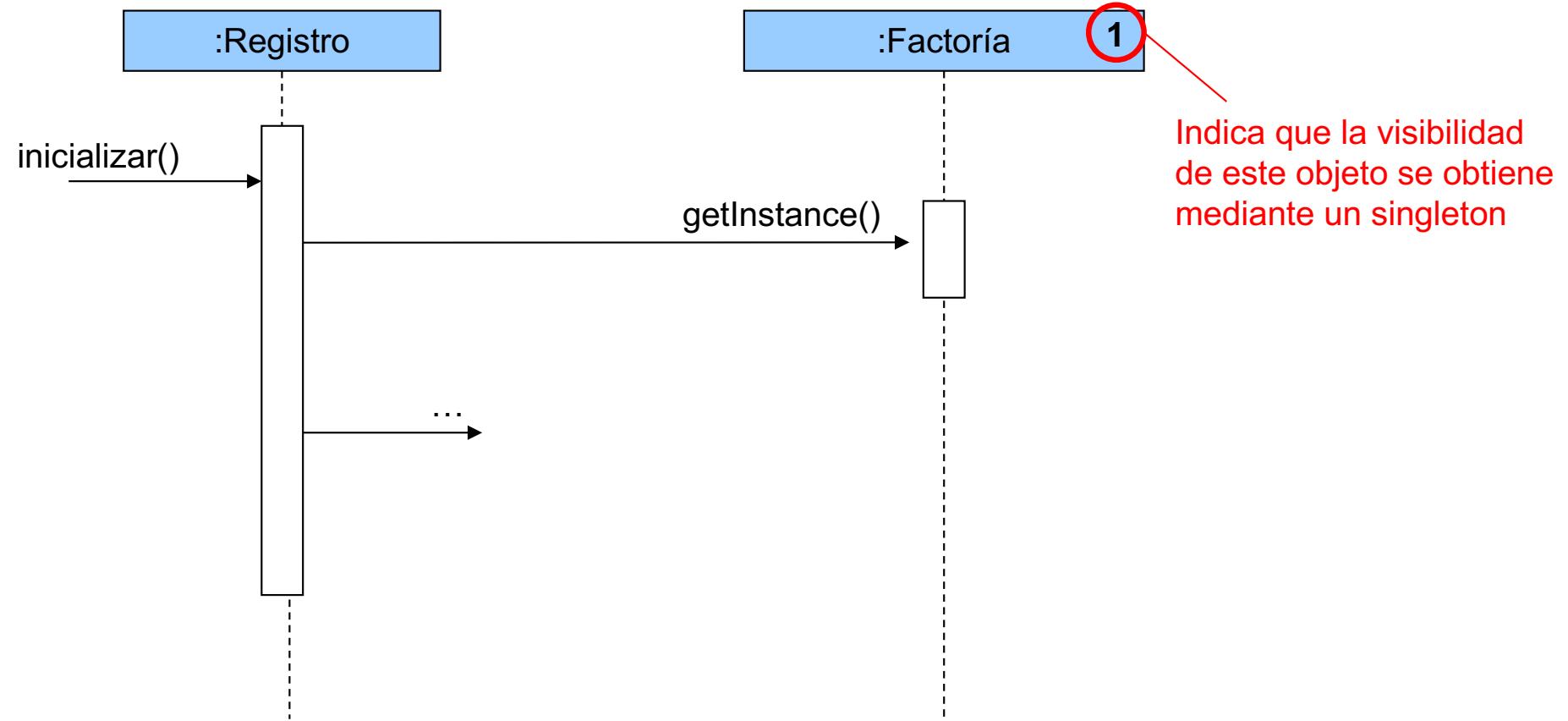
Asegura que una clase tiene una única instancia,

... y proporciona un único punto de acceso a esta instancia

( *Singleton* es una palabra inglesa que significa una cosa única del tipo del que se está hablando )

```
ClaseSingleton
static instanciaUnica
// Otros atributos del Singleton
static getInstance(): Singleton
// Otras operaciones del Singleton
```

# Uso de un objeto *Singleton*



## Propiedades:

- una clase tiene una única instancia,  
... y ofrece una único punto de acceso a esta instancia

## Uso:

- para administrar objetos de tipo *Factoría* y de tipo *Fachada*
- es posible definir subclases y refinamientos de una clase *singleton*  
... pero hay que tener cuidado con las facilidades que ofrecen (o no ofrecen) los lenguajes de programación en este sentido

# El patrón de diseño ***Singleton***

**Problema:** Queremos permitir una única instancia (esto es, un *singleton*) de una clase,

... p.ej., cuando otros objetos necesitan un único punto global de acceso

**Solución:** Definimos un método (estático) de la clase, que retorna la instancia única (el *singleton*)

Ejemplo: Un programa con varias clases que necesitan generar números aleatorios:

- para propósitos de *debugging*, no conviene construir varios generadores independientes de números aleatorios
- la secuencia de números que un GNA produce no es verdaderamente aleatoria, sino el resultado de un cálculo determinista —los números generados se llaman *pseudo aleatorios*
- en la mayoría de los algoritmos, se comienza con un valor semilla que es transformado para producir el primer valor de la secuencia
- luego, la transformación es aplicada nuevamente para producir el siguiente valor, y así sucesivamente

# 8

Definimos un **compuesto** como una colección de objetos

... en que algunos objetos pueden ser a su vez colecciones de objetos:

- algunos objetos representan colecciones de objetos
- otros, representan ítems individuales, u hojas

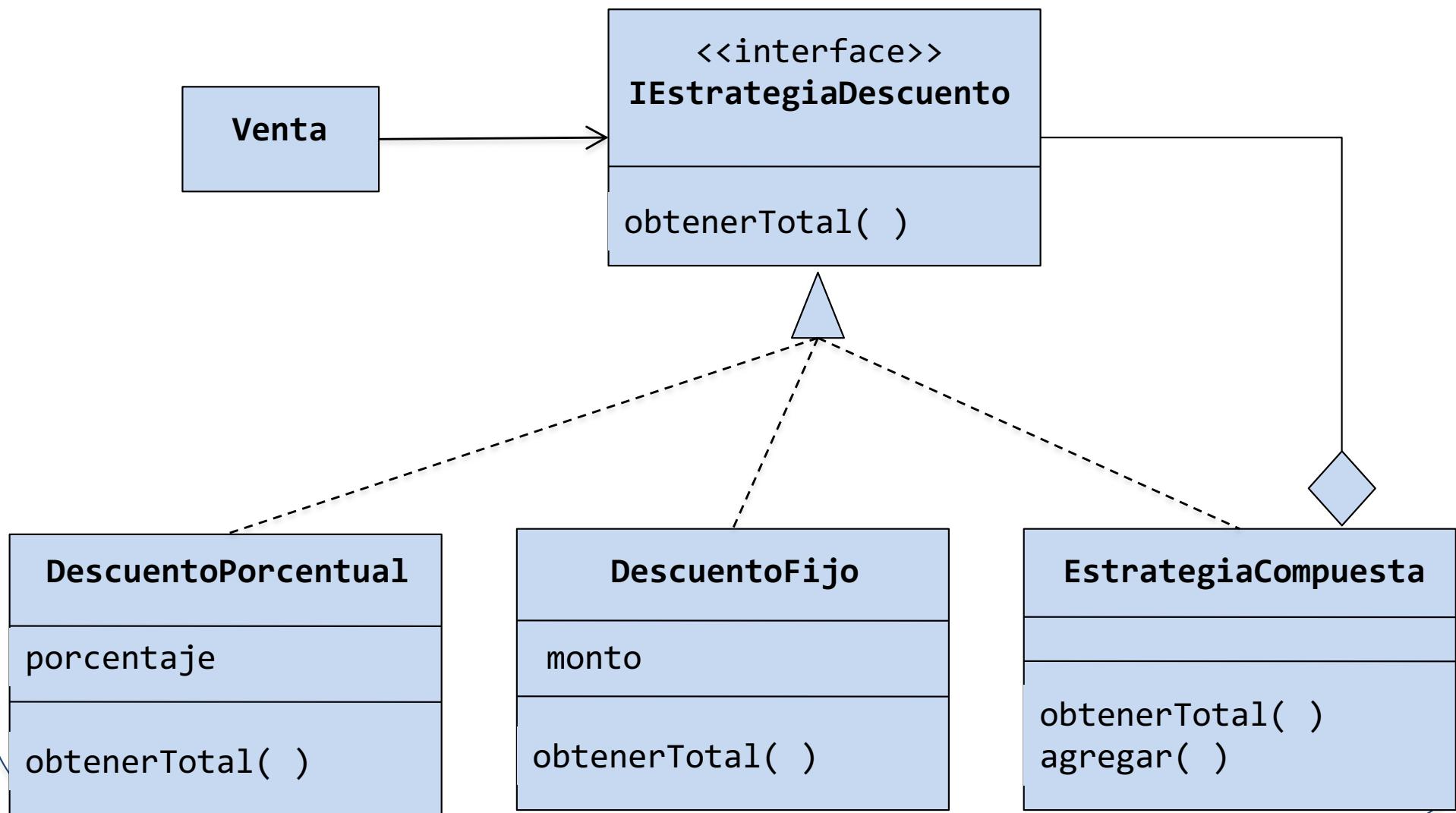
## Dos desafíos importantes:

- hay que poder diseñar las colecciones de objetos de manera que puedan contener ya sea ítems individuales
  - ... u otras colecciones similares
- hay que definir comportamientos comunes para objetos individuales
  - ... y para objetos compuestos

Definimos una interfaz común para colecciones e ítems,

... y modelamos las colecciones de manera que contengan una colección de objetos de este tipo

P.ej., la estrategia de descuento puede ser descuento porcentual, descuento fijo u otros, o también puede ser una *estrategia compuesta* por varios tipos de descuentos



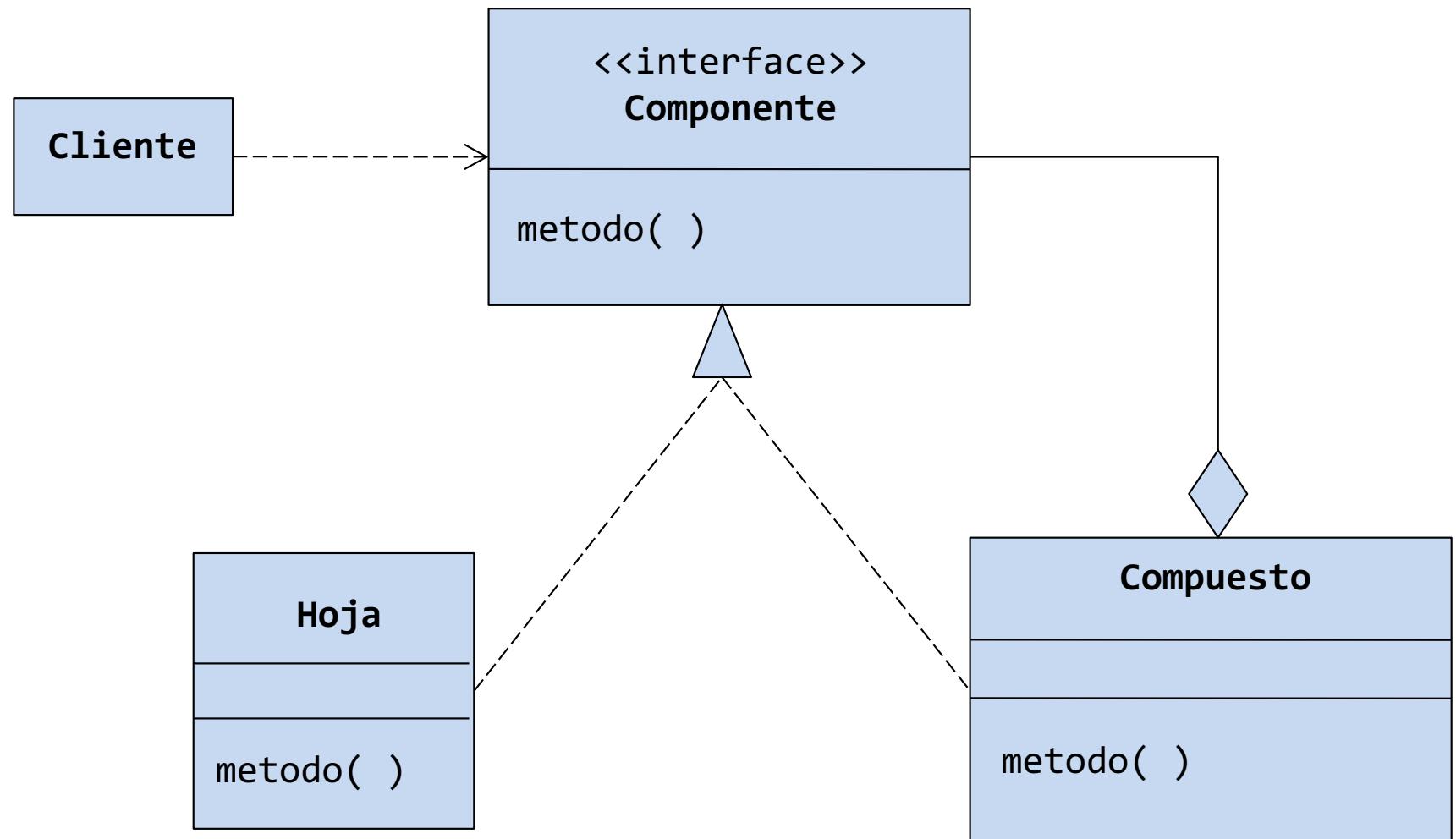
## Contexto:

- objetos simples —hojas— pueden ser combinados en objetos compuestos
- los clientes tratan el objeto compuesto como simple

## Solución:

- definir una interfaz que sea una abstracción de los objetos simples
- un objeto compuesto contiene objetos simples
- tanto las clases simples como las clases compuestas implementan la (misma) interfaz
- al implementar un método de la interfaz, la clase compuesta lo aplica primero a sus objetos simples, y luego combina los resultados

# El patrón de diseño ***Compuesto***



# 9

## El patrón de diseño *Puente*

Se concentra en el diseño de una *abstracción*:

- una clase con un conjunto de operaciones abstractas —es decir, en la clase sólo tenemos los nombres de las operaciones
- hay varias implementaciones distintas posibles de estas operaciones abstractas

Un diseño posible:

- una jerarquía de clases, con la clase abstracta en la cúspide (define las operaciones abstractas)
- cada subclase en la jerarquía proporciona una implementación diferente para las operaciones abstractas

P.ej., supongamos que tenemos clases contenedoras, tales como Mapa y Cola:

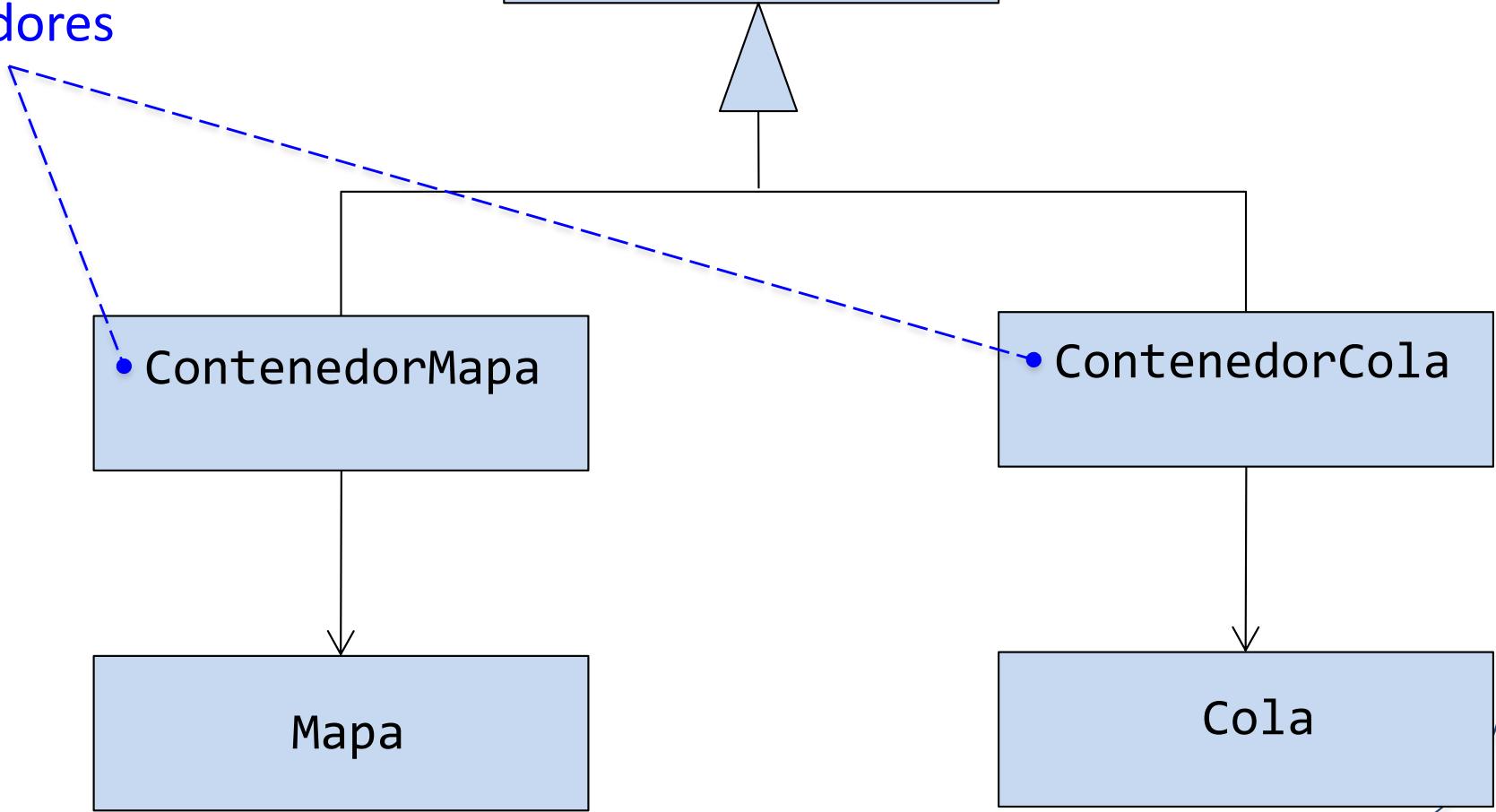
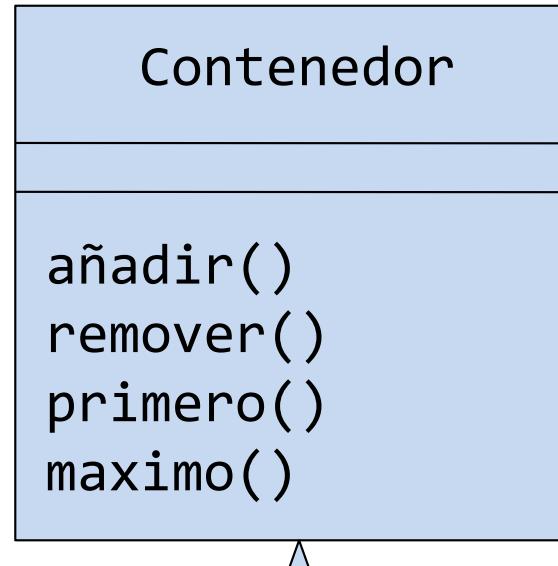
- Mapa tiene los métodos pertenece(), agregar() y eliminar()
- Cola tiene llegar(), salir(), primero()

Queremos crear una operación abstracta que haga lo mismo en cualquier contenedor, p.ej., maximo():

- para simplificar la escritura de maximo(), estandariza-mos los nombres de los métodos de las clases contenedoras: añadir(), remover(), primero()  
... pero no podemos cambiar los nombres de los métodos en las clases Mapa y Cola

# Abstracción

Son verdaderos adaptadores

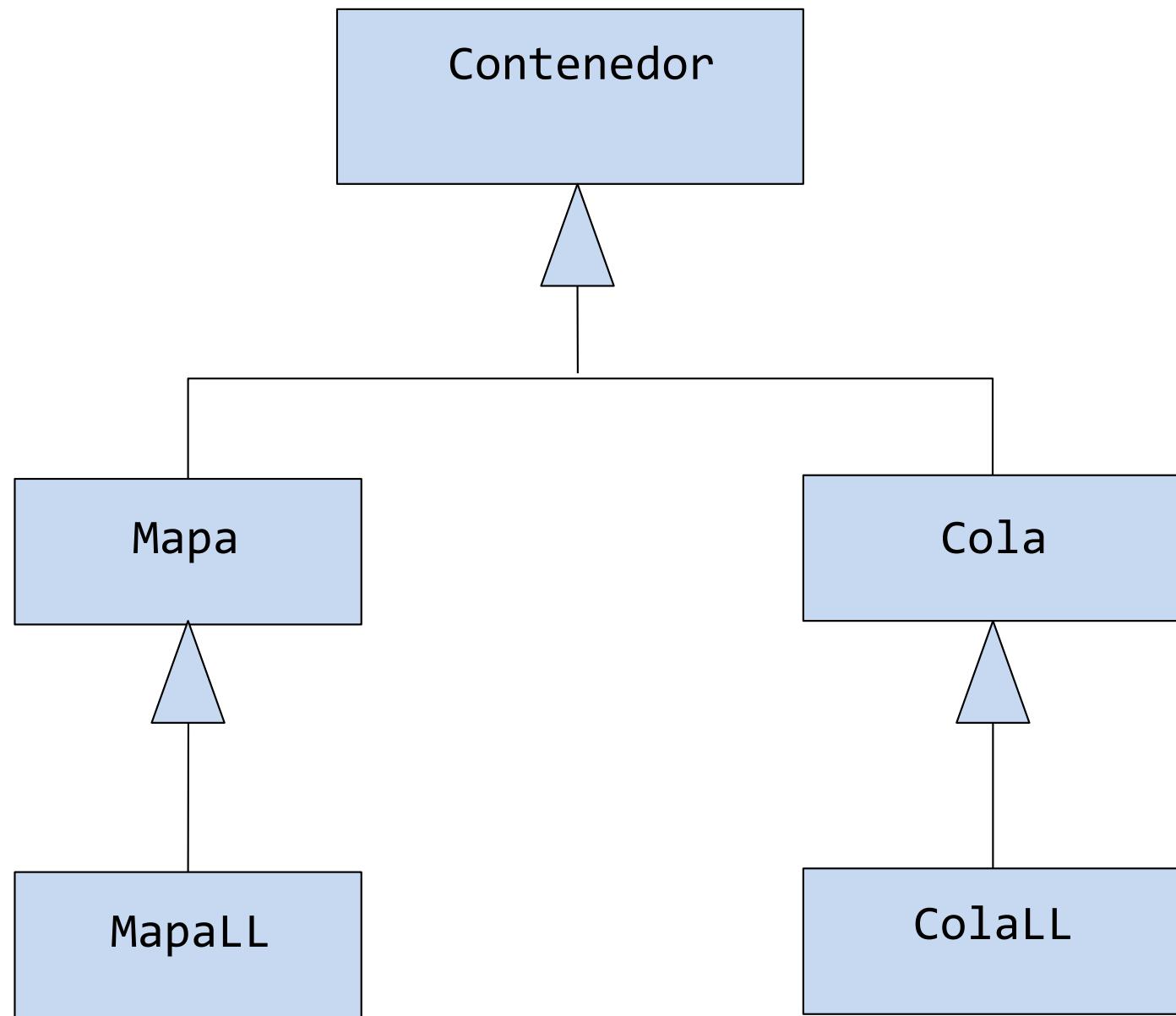


En la diap. anterior, cada tipo particular de contenedor debe ser usado por una subclase diferente de la clase abstracta Contenedor:

- ContenedorMapa para mapas
- ContenedorCola para colas

¿Qué pasa si necesitamos hacer subclases por alguna otra razón?

- p.ej., si queremos distinguir los contenedores implementados mediante listas ligadas
  - ... necesitamos otra subclase de Contenedor —un Contenedor implementado mediante una lista ligada
  - ... pero aún necesitamos contenedores tipo Mapa y tipo Cola



La diap. anterior muestra una posibilidad de tener mapas y colas implementados mediante listas ligadas

Sin embargo, las clases para listas ligadas —MapaLL y ColaLL— no podrían compartir código

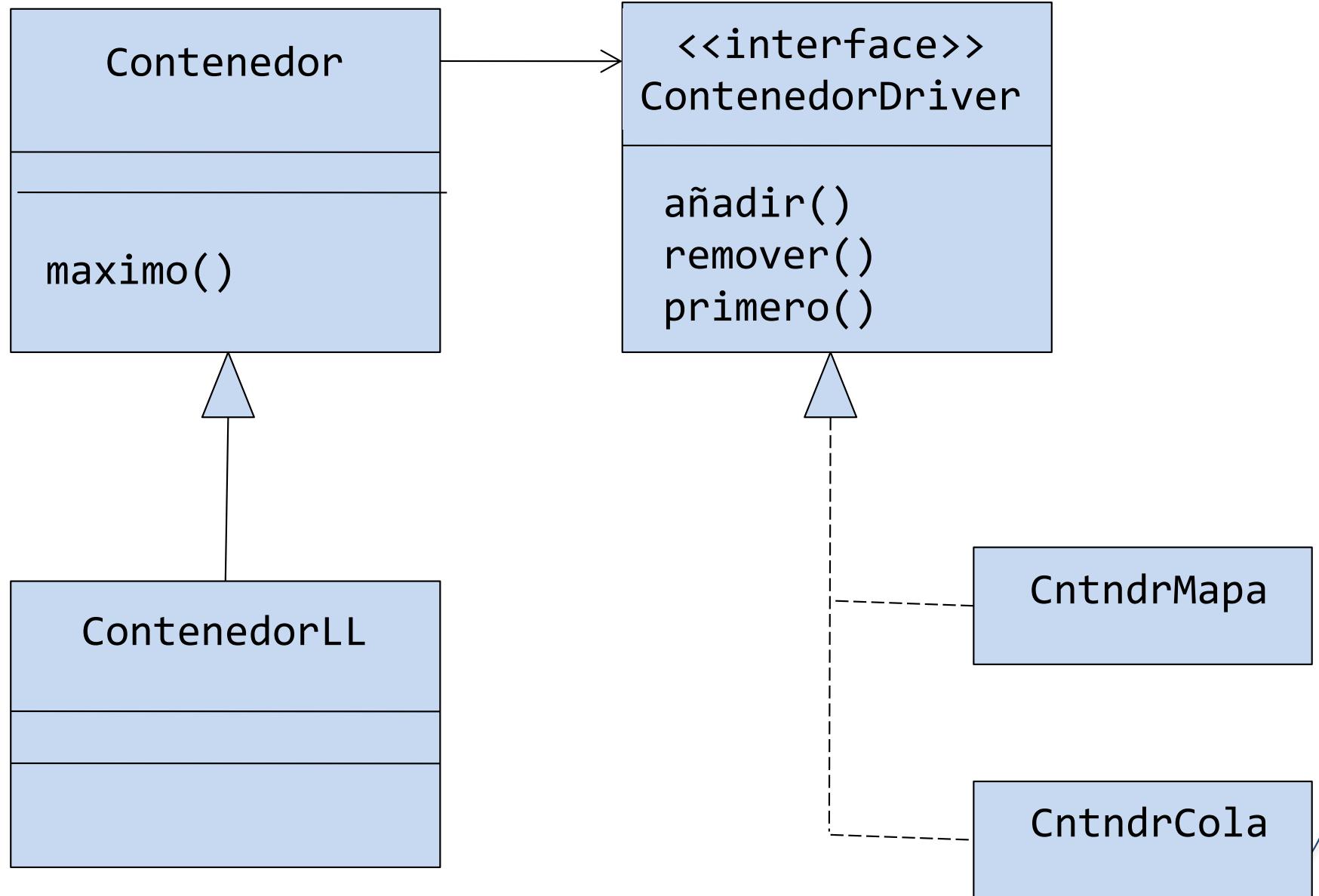
... ya que no hay una superclase para listas ligadas:

- si hay muchos contenedores y queremos cambiar la implementación de un método, tenemos un problema

La solución es crear un *puntoe*:

- llevamos el conjunto de operaciones abstractas a una interfaz — ContenedorDriver
- ahora, una abstracción dependerá de una implementación de la interfaz
- el propósito es desacoplar una abstracción de la implementación de sus operaciones abstractas —ambas pueden variar independiente-mente

# El patrón *Puente*



## Principios de orientación a objetos (ya vistos y adicionales):

- encapsular lo que cambia
- evitar código duplicado, abstrayendo las cosas que son comunes y poniéndolas en un único lugar
- programar con respecto a una interfaz y no con respecto a una implementación (*bajo acoplamiento*)
- cada clase debería tener sólo una razón para cambiar (*alta cohesión*)
- las clases deberían estar abiertas para extenderlas, pero cerradas para modificarlas (*abierto-cerrado*)
- cada objeto debería tener una sola responsabilidad, y todos los servicios del objeto deberían concentrarse en llevar adelante esa responsabilidad (nuevamente, *alta cohesión*)
- las subclases deberían ser adecuadas para sus superclases (*sustituibilidad*)