

Patrones de diseño de software

Ingeniería de Software – IIC2143

Yadran Eterovic (yadran@ing.puc.cl)

2-2015

0

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

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

Hay 11 patrones GoF de comportamiento:

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

1

El sistema PdV y varios subsistemas calculadores de impuestos

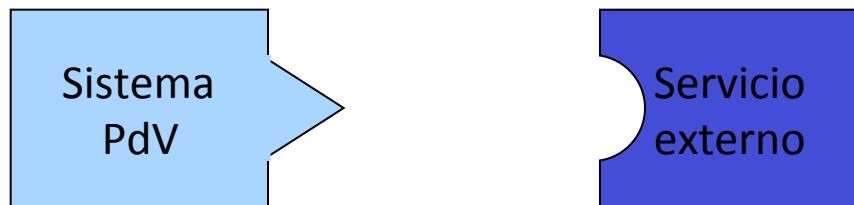
Un sistema PdV necesita poder conectarse a diversas aplicaciones para calcular impuestos —p.ej., Tax-Master y TopTaxPro— desarrolladas por terceros

... sólo que cada calculador de impuestos 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

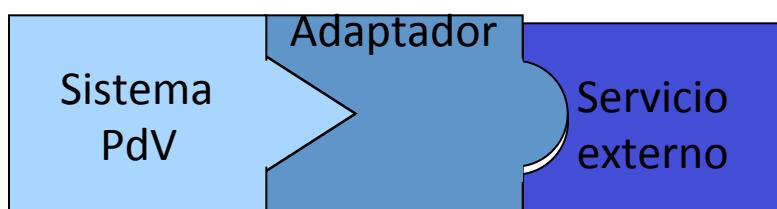
Convertimos la interfaz original en otra interfaz, a través de un objeto *adaptador* intermedio



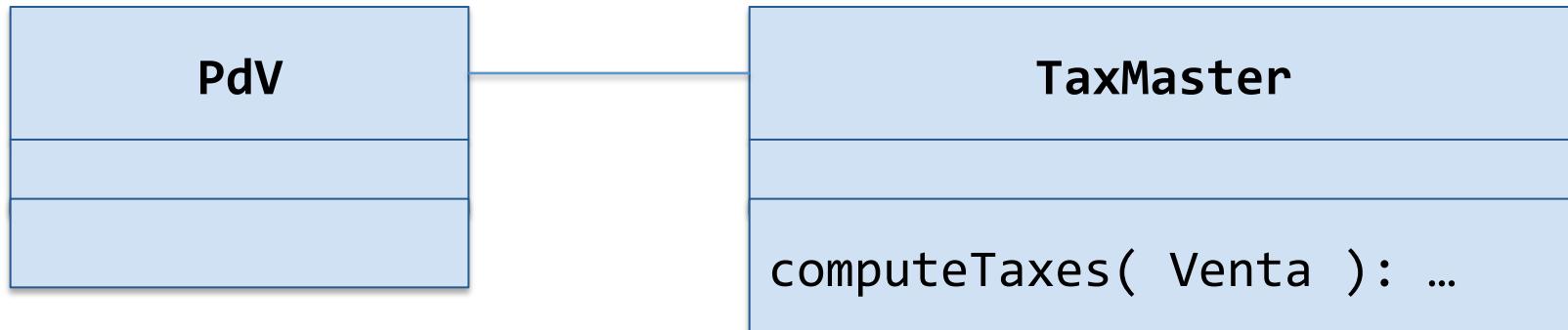
El código del sistema PdV no cambia



El código del servicio externo no cambia

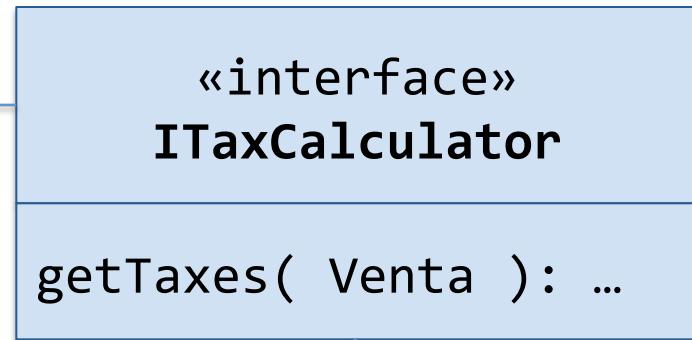
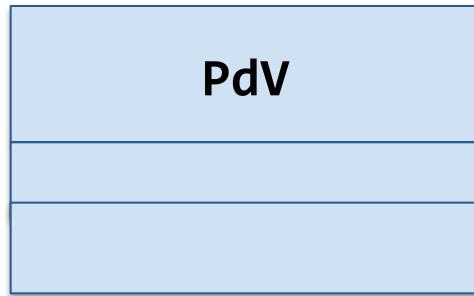


El nuevo código —lo que cambia— se centraliza en el adaptador



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()`

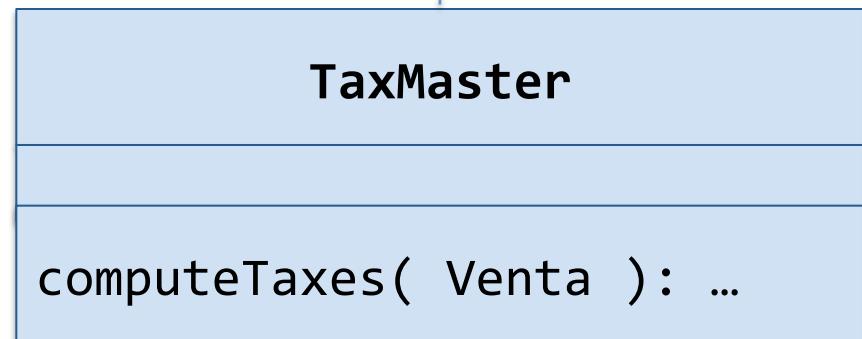
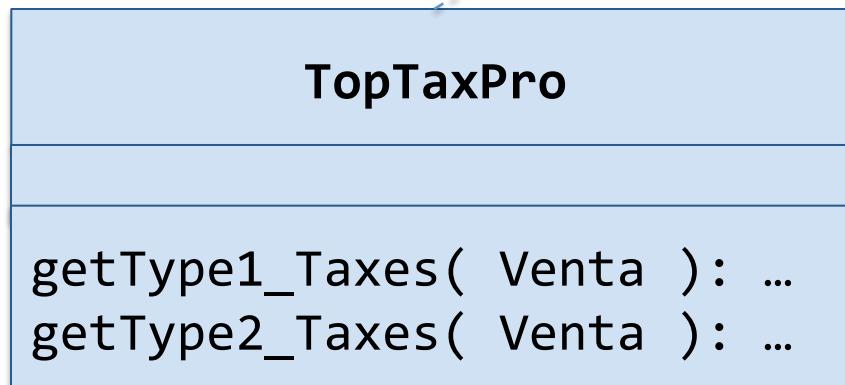


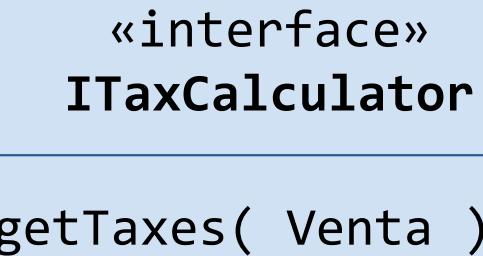
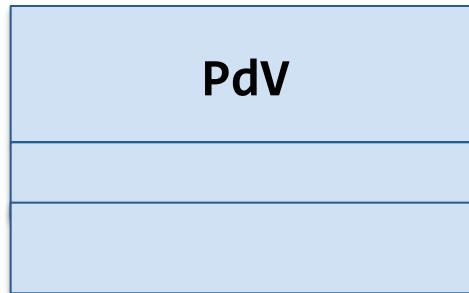


¿Cómo conseguimos que PdV pueda usar siempre la misma interfaz **ITaxCalculator** y llamar sólo al método `getTaxes()`?

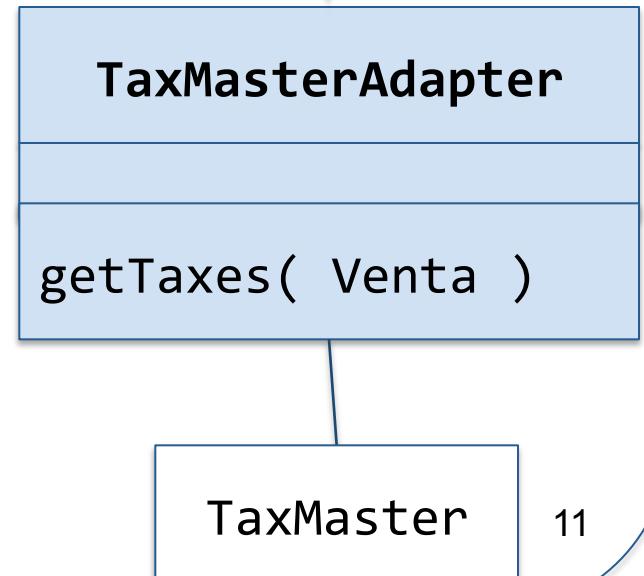
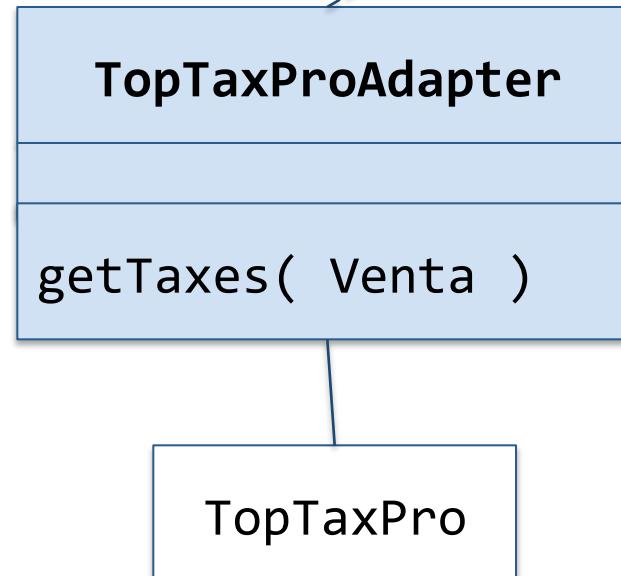
¿?

¿?

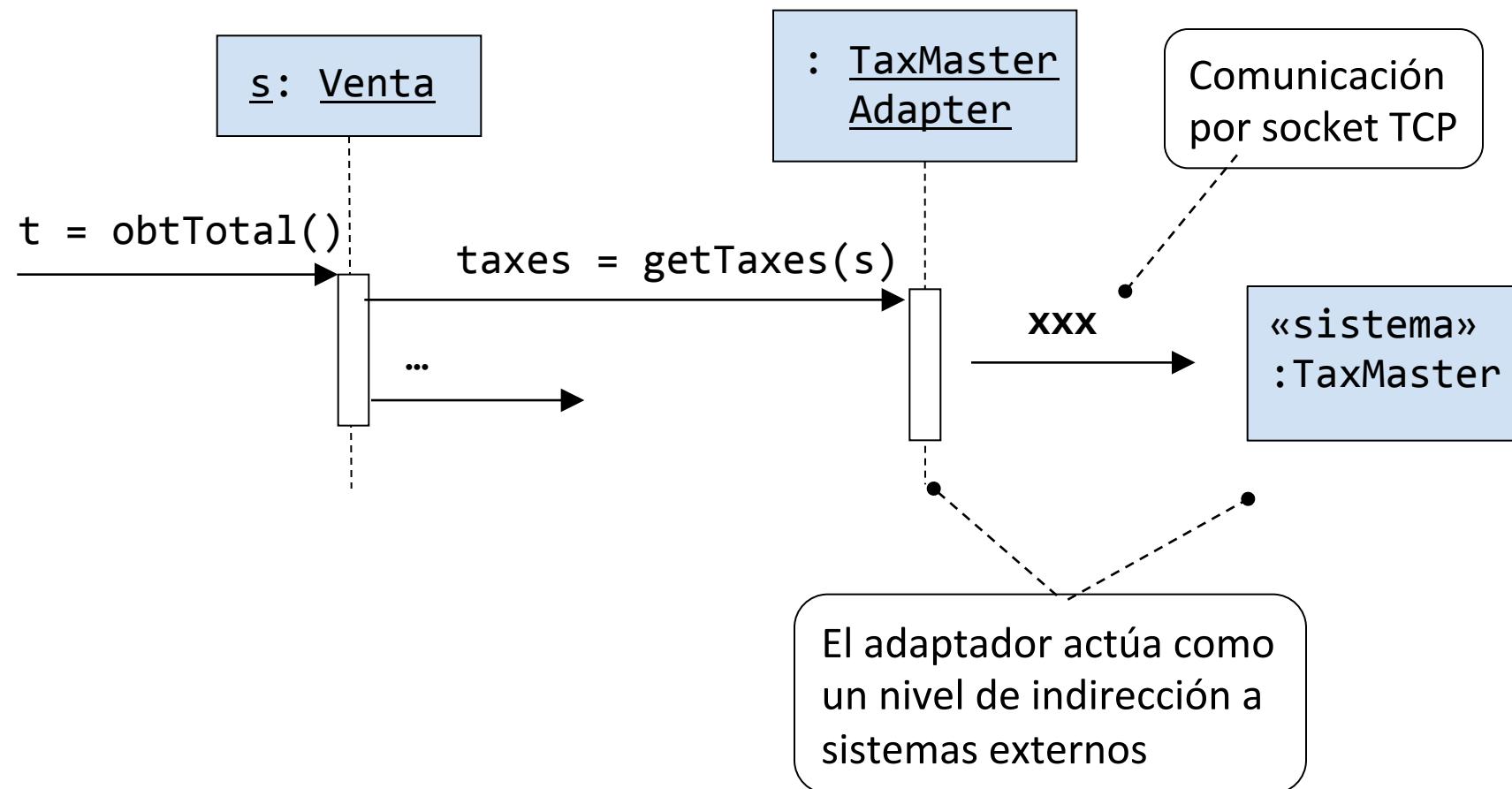




TopTaxProAdapter y TaxMasterAdapter implementan la interfaz ITaxCalculator y la *adaptan* a TopTaxPro y TaxMaster, respectivamente



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)

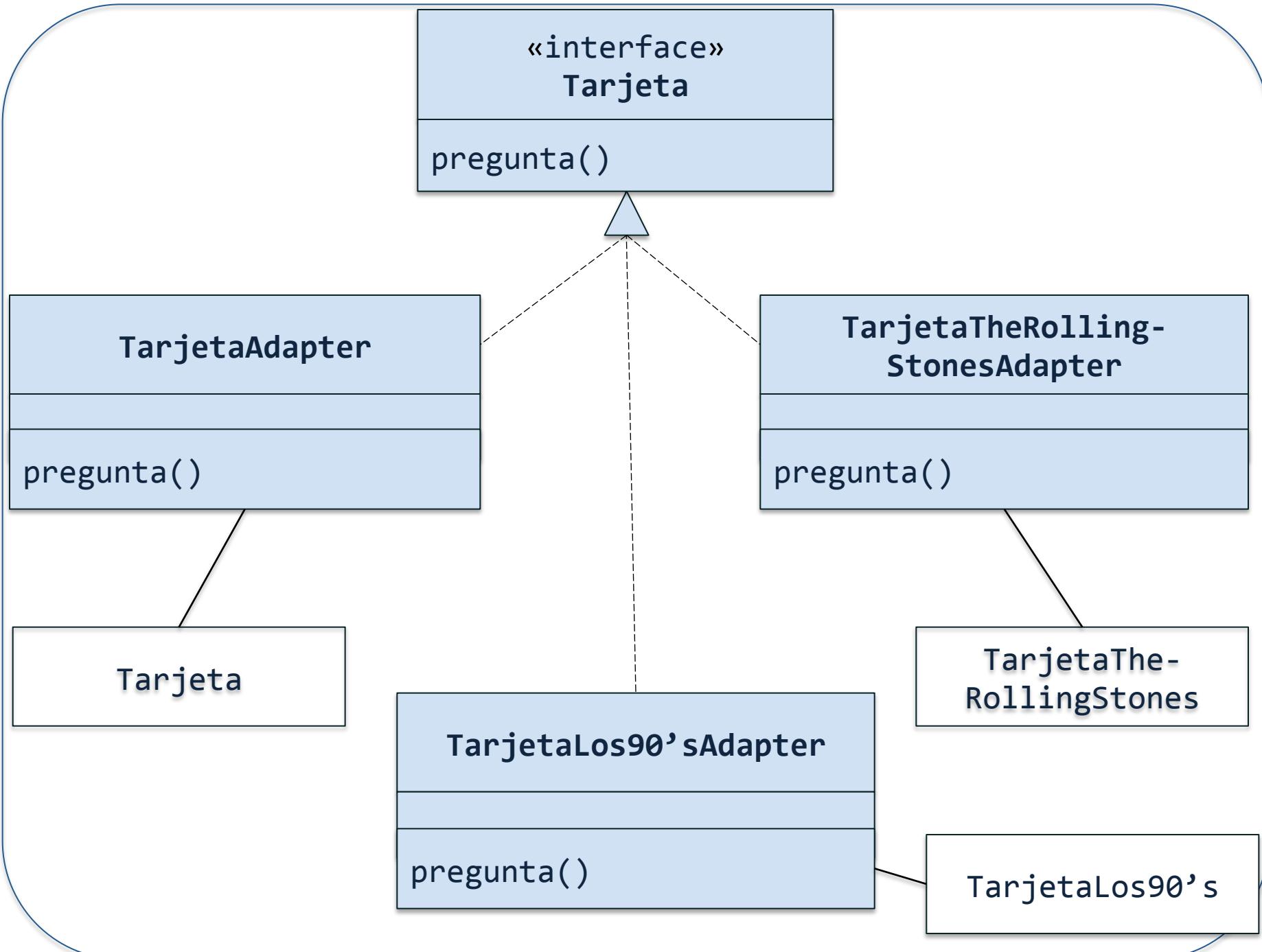
```
obtenerPregunta() {  
    —devuelve pregunta y respuesta ...}
```

TarjetaTheRollingStones

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

TarjetaLos90's

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



2

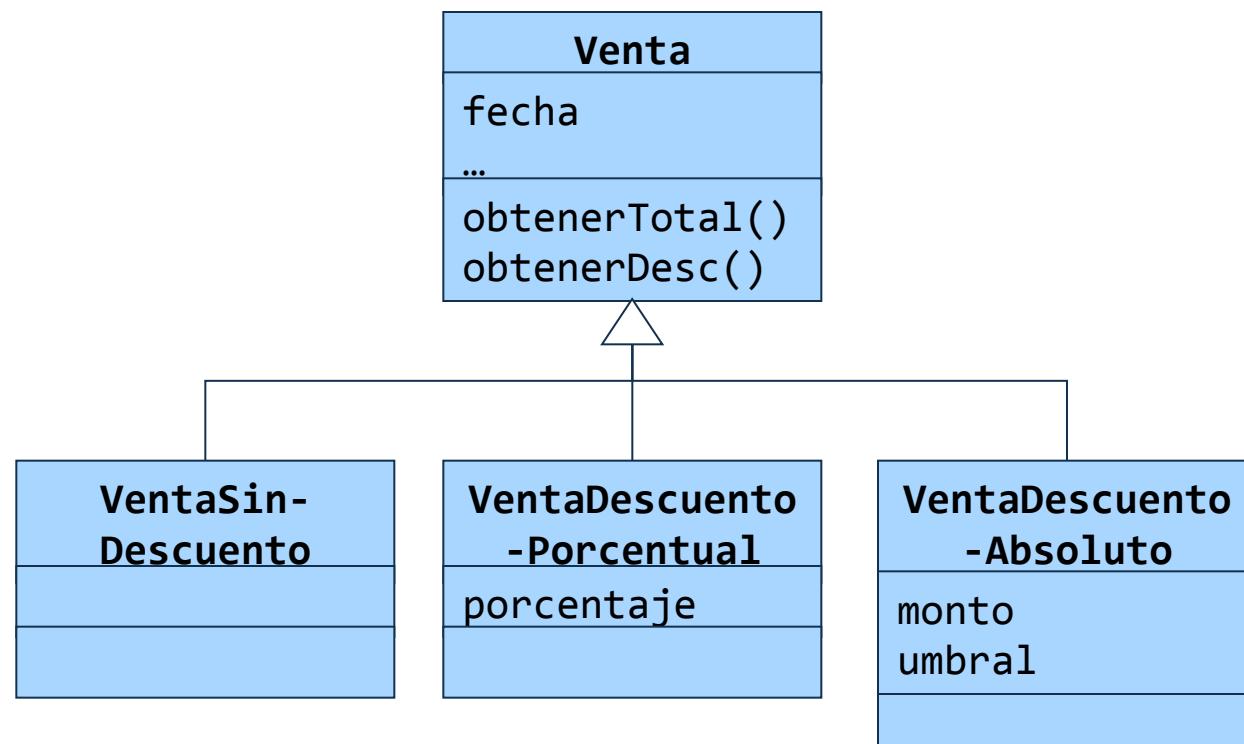
El sistema PdV y las políticas de descuento

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 obtenerDesc() 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., obtenerDesc(), 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

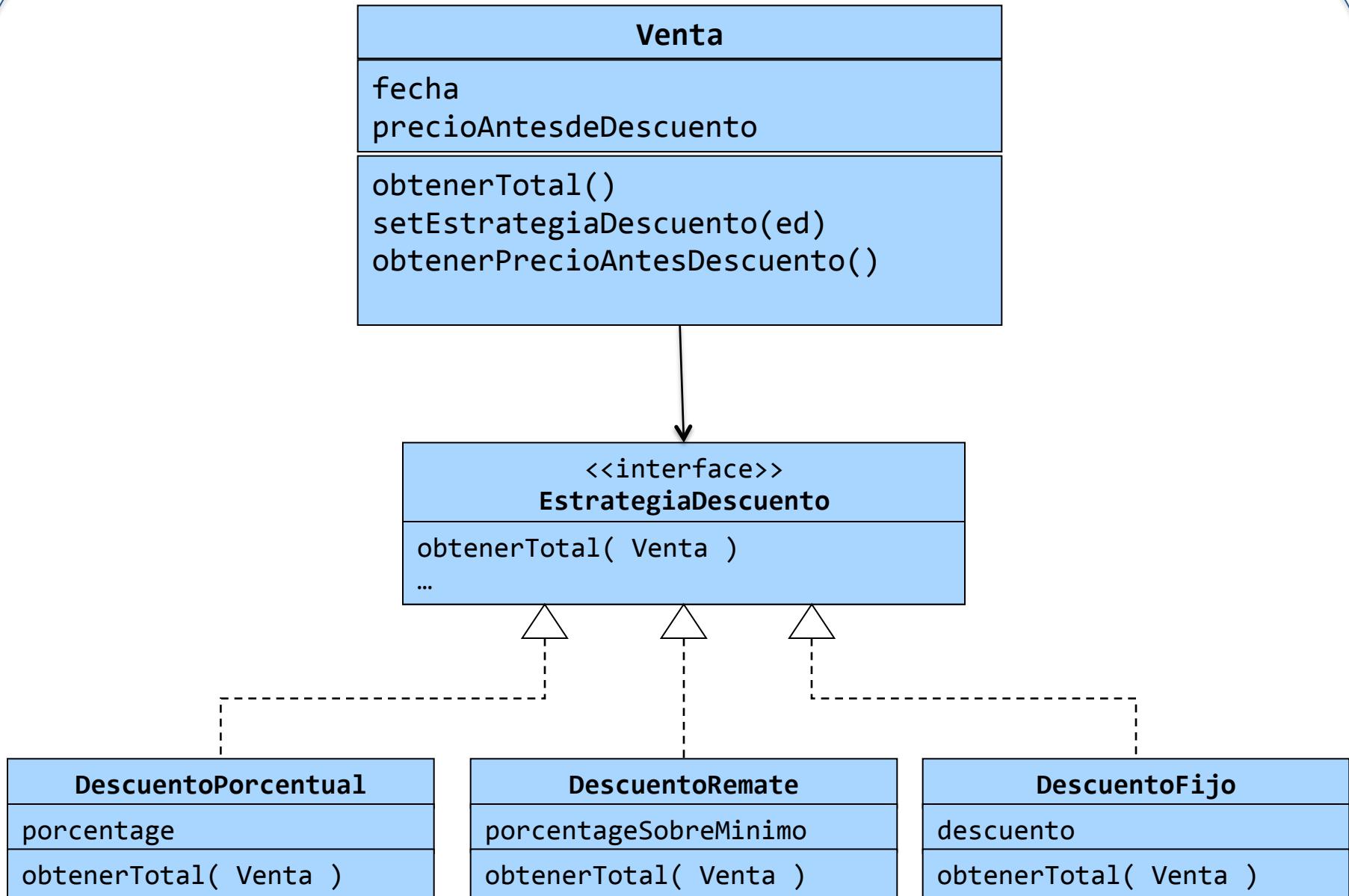
La herencia no permite flexibilidad en los cambios y duplica código

Solución: Separar las políticas de descuento de las ventas

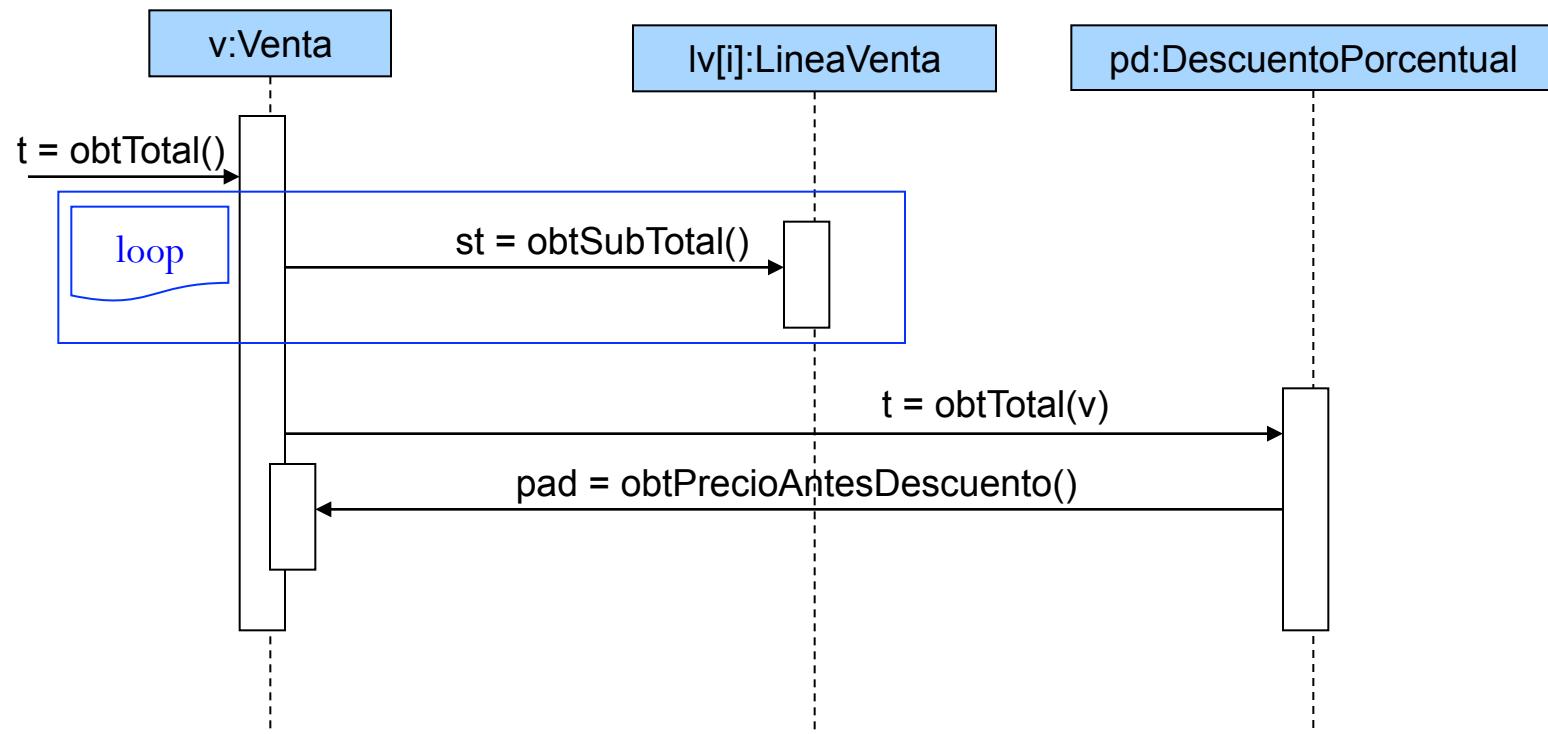
Creamos una *interfaz* que representa la *estrategia de descuento*

... 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**

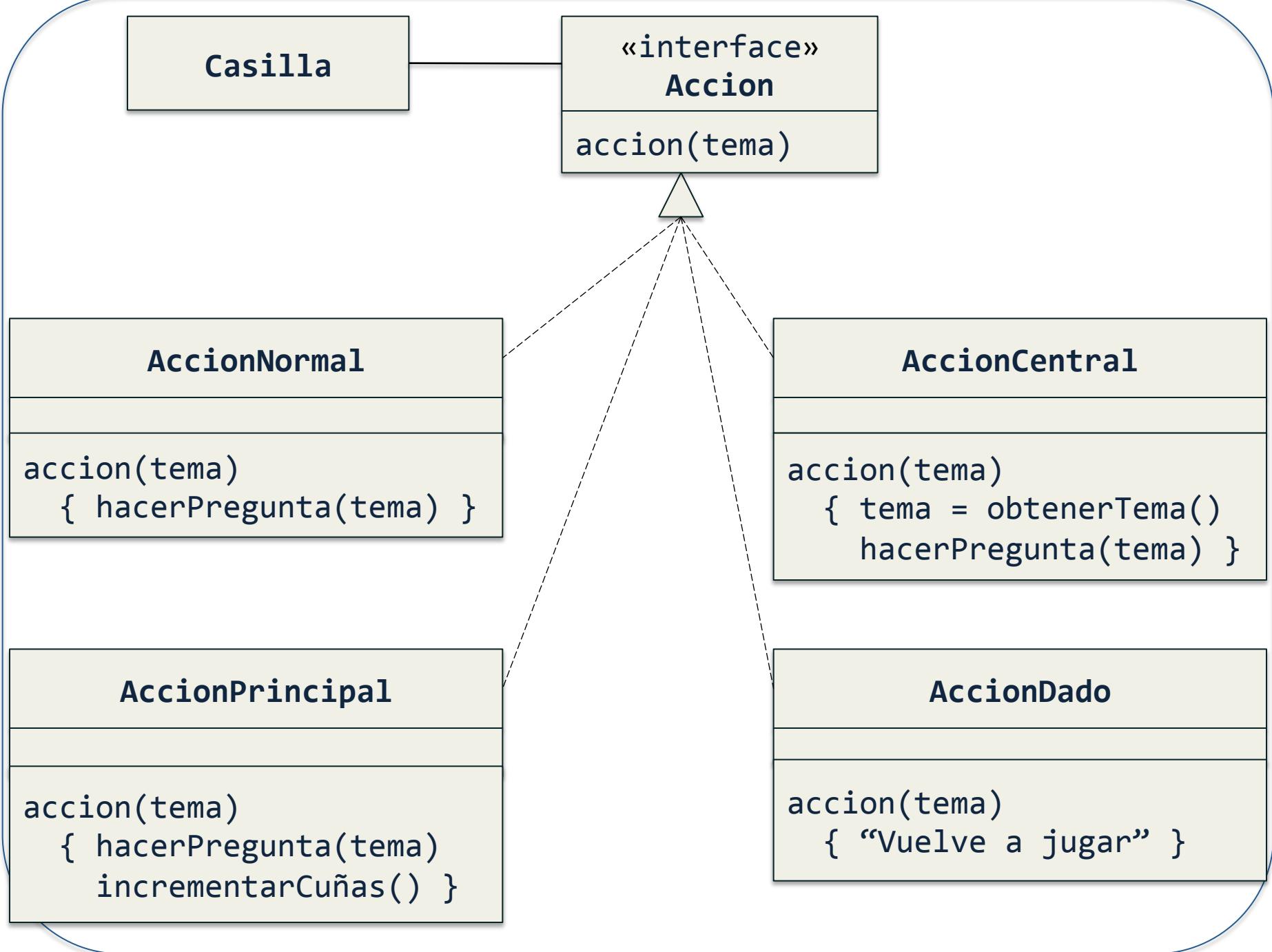
Un **objeto estrategia** (p.ej., una estrategia de descuento) está vinculado a un **objeto de contexto** —el objeto al que se le aplica la estrategia (o algoritmo), en el ejemplo, una Venta

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

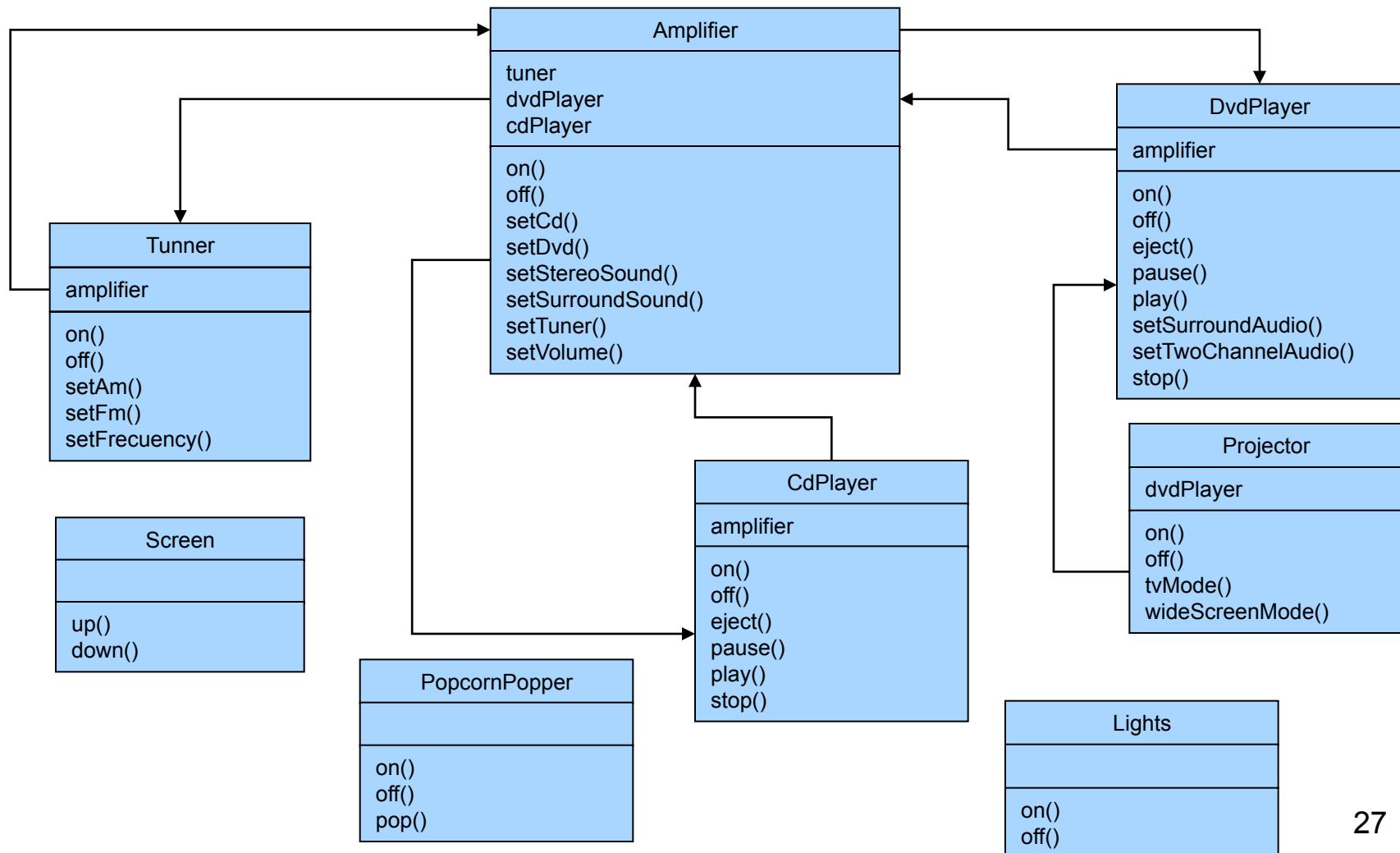
En el caso de *The Trivium*, ¿cómo enfrentamos la variabilidad en la conducta de las casillas?

- 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”

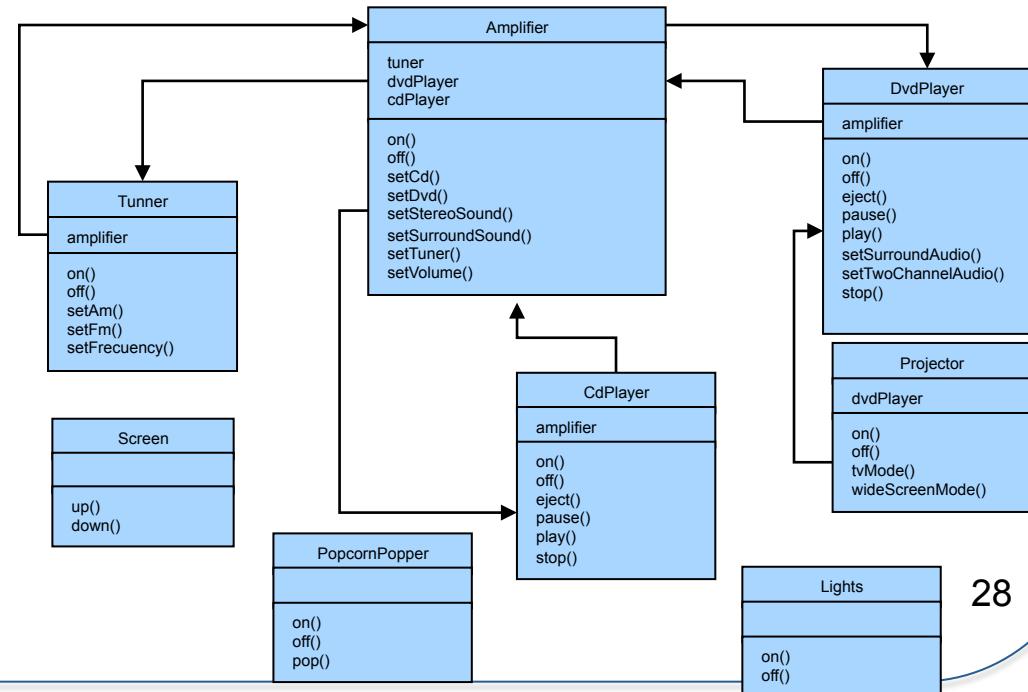


Observaciones:

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

Problema:

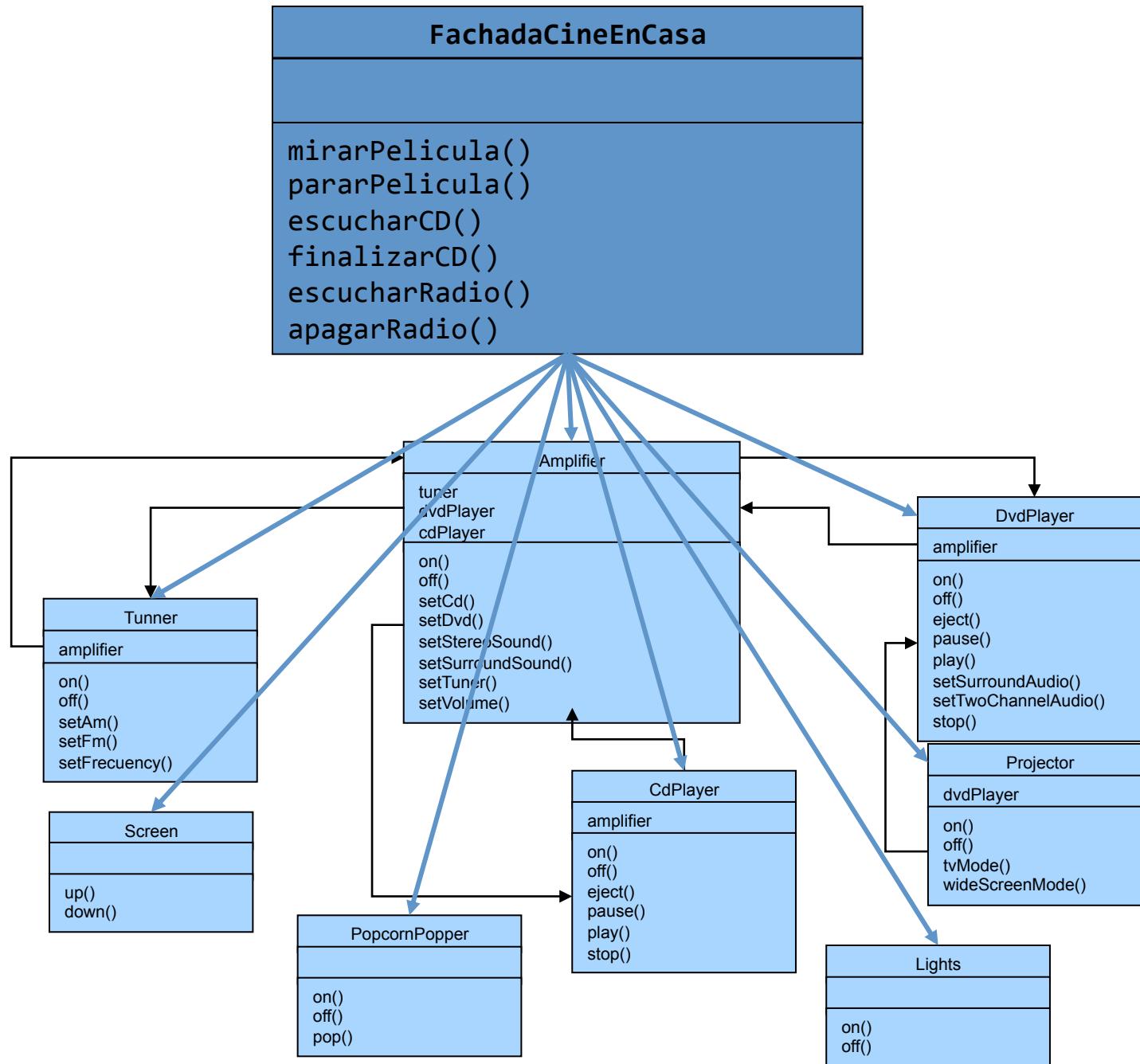
- ¿Cómo simplificar el uso del sistema integralmente?



Solución: El control remoto universal

Para gestionar todos estos aparatos 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

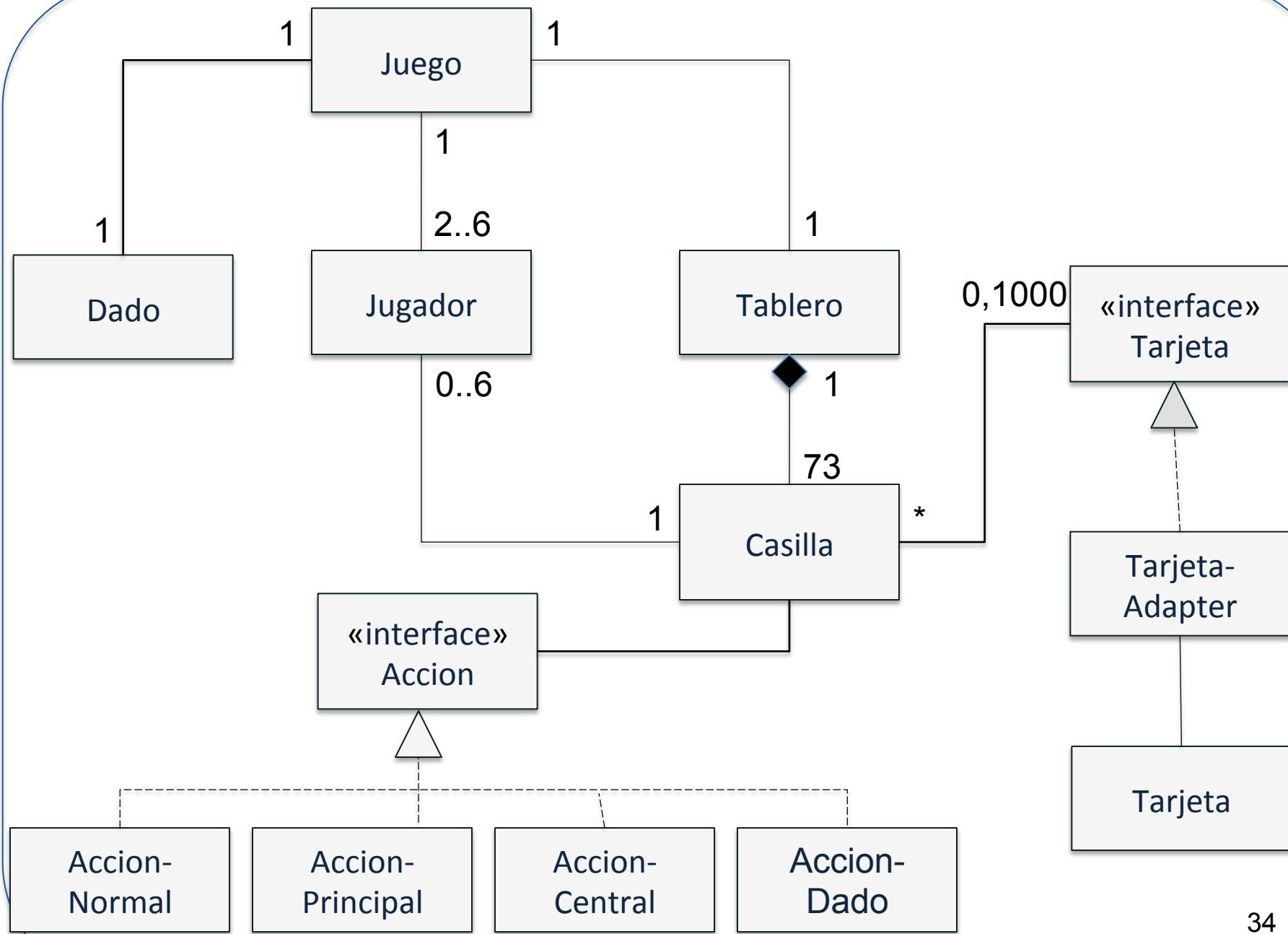
El :Juego es una *fachada* en “The Trivium”

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

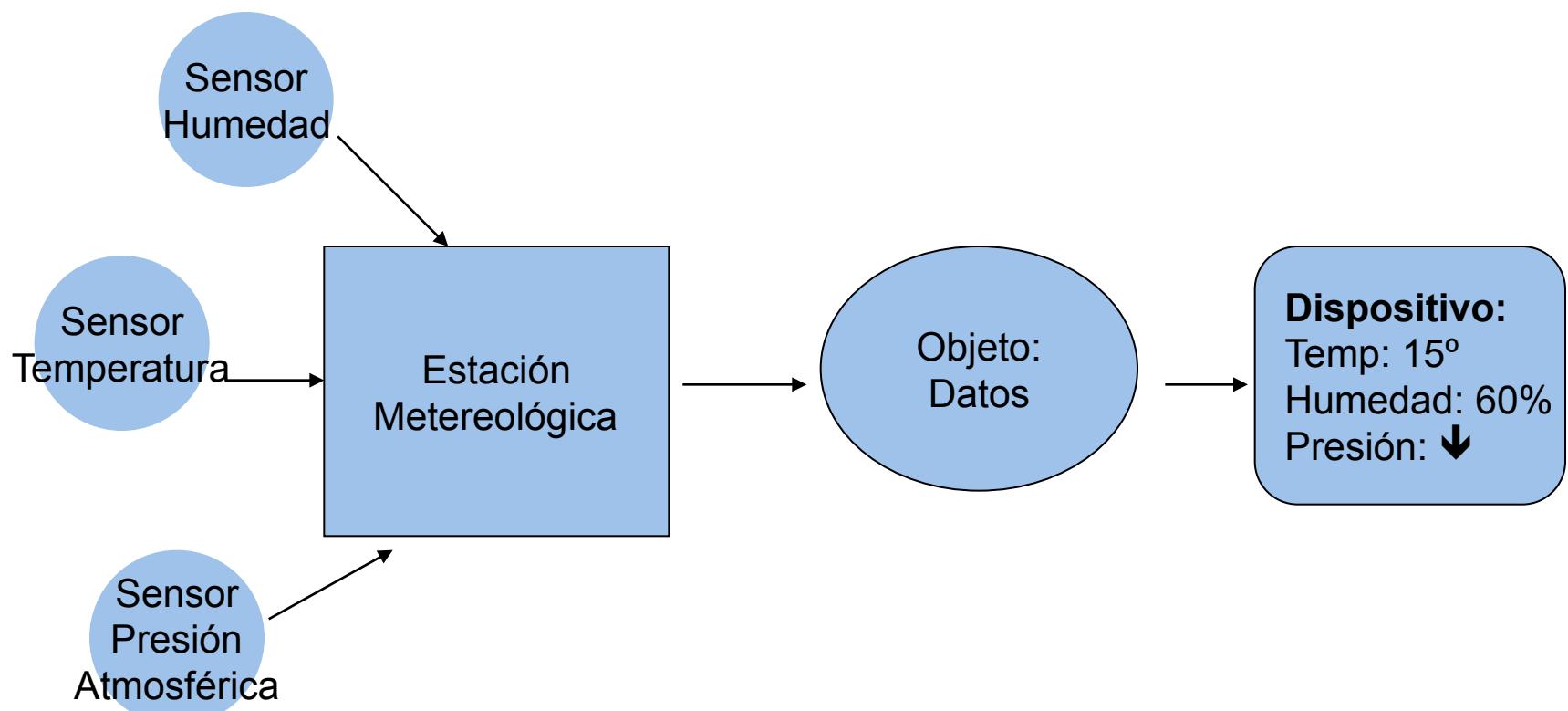
Otros mensajes del jugador que podrían ir directamente desde la GUI a algún otro objeto de la aplicación:

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



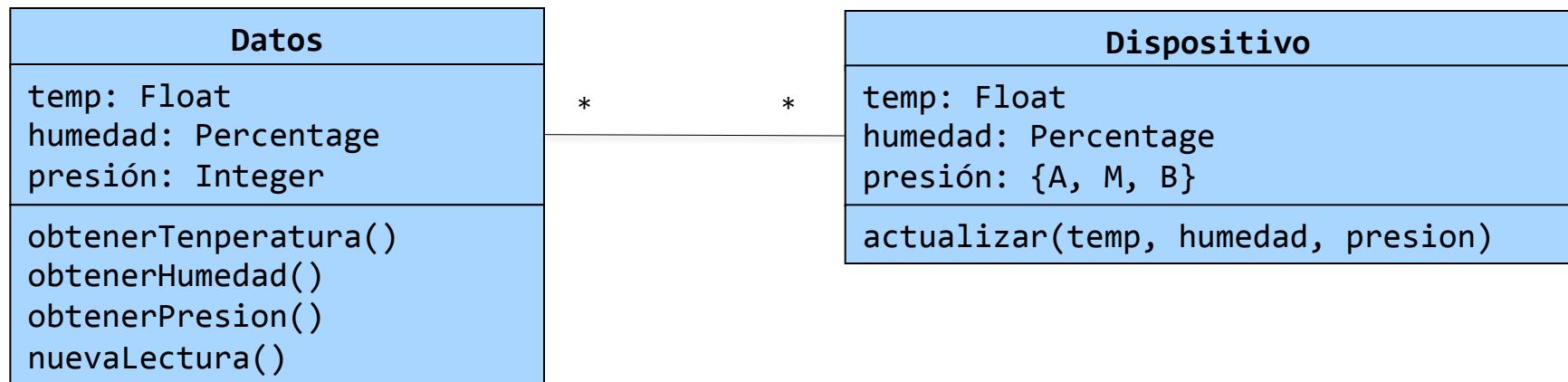
4

El caso de la estación metereológica



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

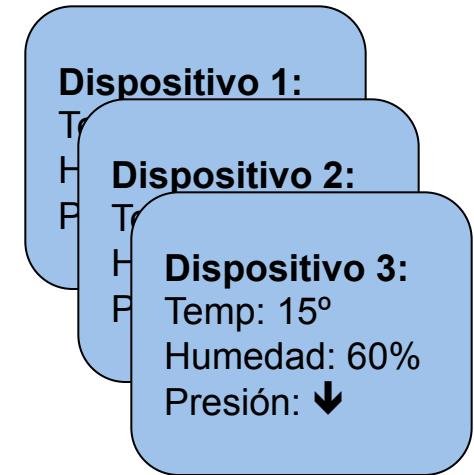
- **Datos** tiene la operación **nuevaLectura()**
- **Dispositivo** tiene la operación **actualizar(...)**
- cada vez que el método **nuevaLectura()** es llamado, el objeto de tipo **Datos** realiza una nueva lectura de la temperatura, humedad y presión
- ... y luego llama a **actualizar()** del dispositivo



```
public class Datos {  
  
    Dispositivo dispActual;  
  
    public void nuevaLectura() {  
        temp = obtenerTemperatura();  
        humedad = obtenerHumedad();  
        presión = obtenerPresión();  
  
        dispActual.actualizar(temp, humedad, presión);  
    }  
}
```

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

```
public class Datos {  
  
    Dispositivo dispActual;  
NuevoDispositivo dispActual2;  
OtroDispositivo dispActual3;  
  
    public void nuevaLectura() {  
        temp = obtenerTemperatura();  
        humedad = obtenerHumedad();  
        presión = obtenerPresión();  
  
        dispActual.actualizar(temp, humedad, presión);  
dispActual2.actualizar(temp, humedad, presión);  
dispActual3.actualizar(temp, humedad, presión);  
    }  
}
```



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

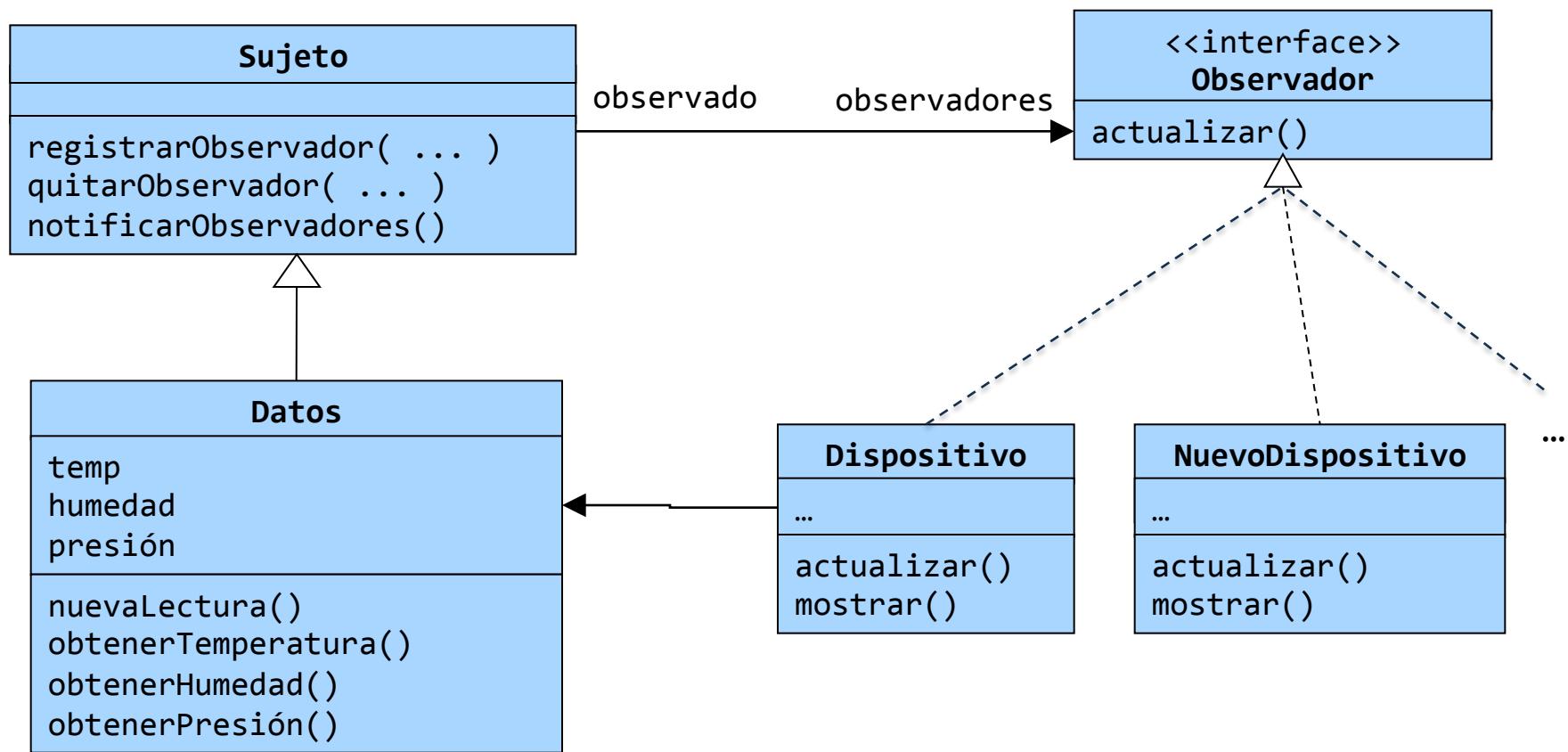
Problema:

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

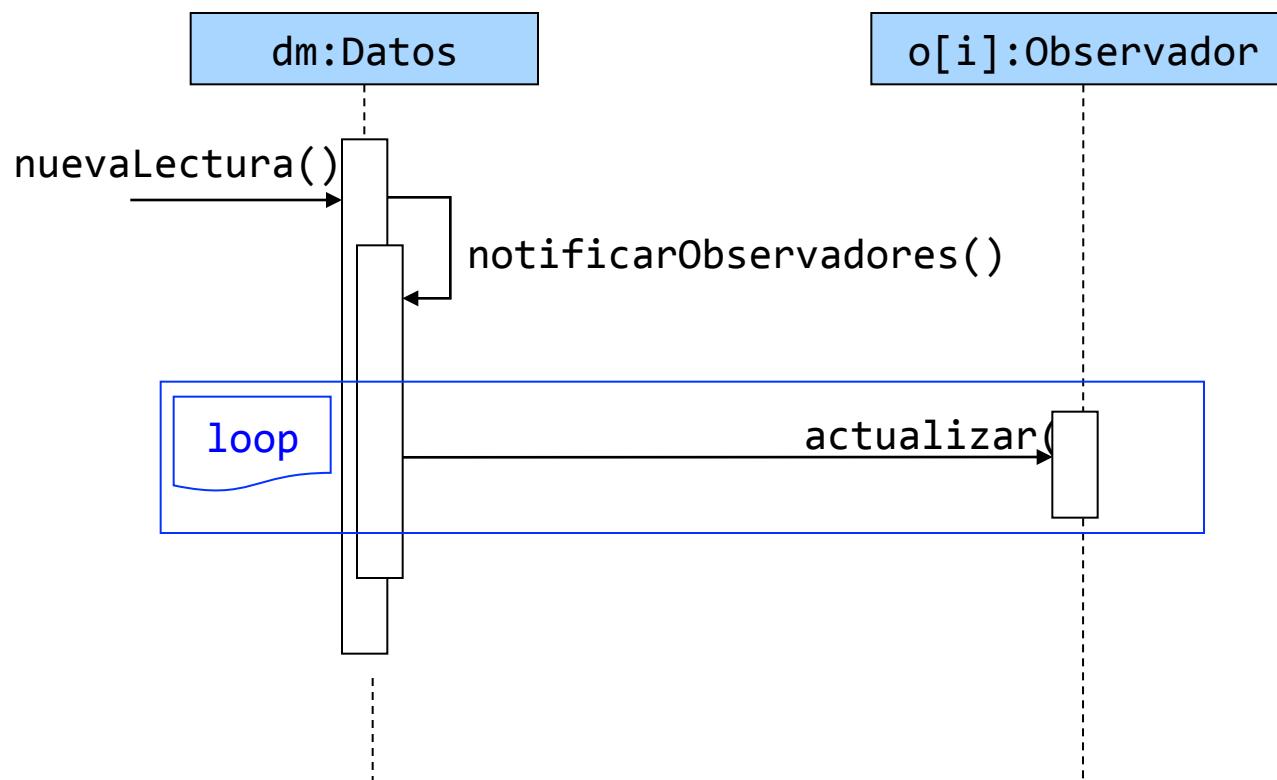


Una solución es emplear el mecanismo de *publicación y suscripción*:

- 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, ejecuta **notificarObservadores()**, que recorre la lista de observadores suscritos y para cada uno ejecuta polimórficamente la operación **actualizar()**



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



Propiedad:

- define una dependencia entre objetos , 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

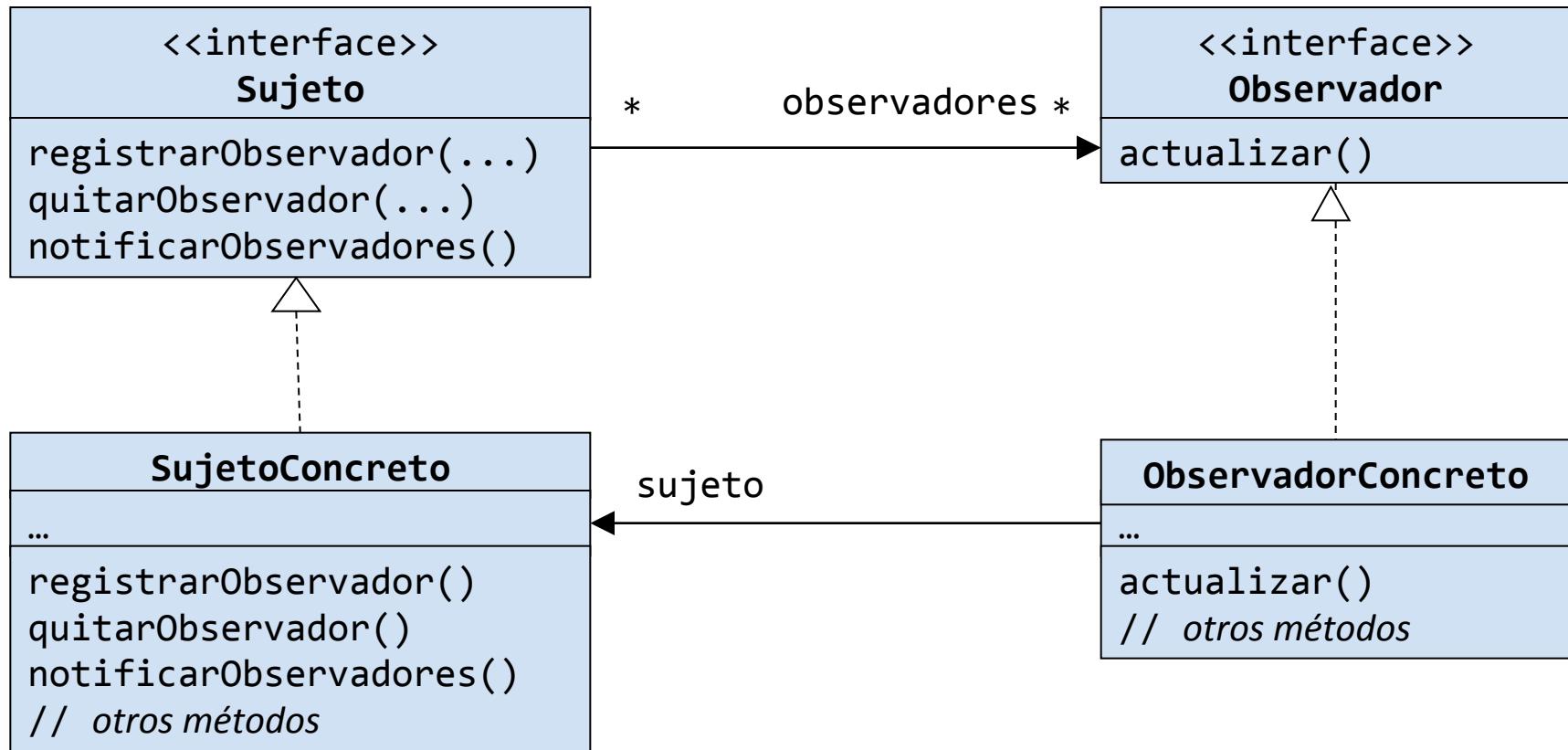
El patrón de diseño **Observador**

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 contenga los diferentes observadores o suscriptores;

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



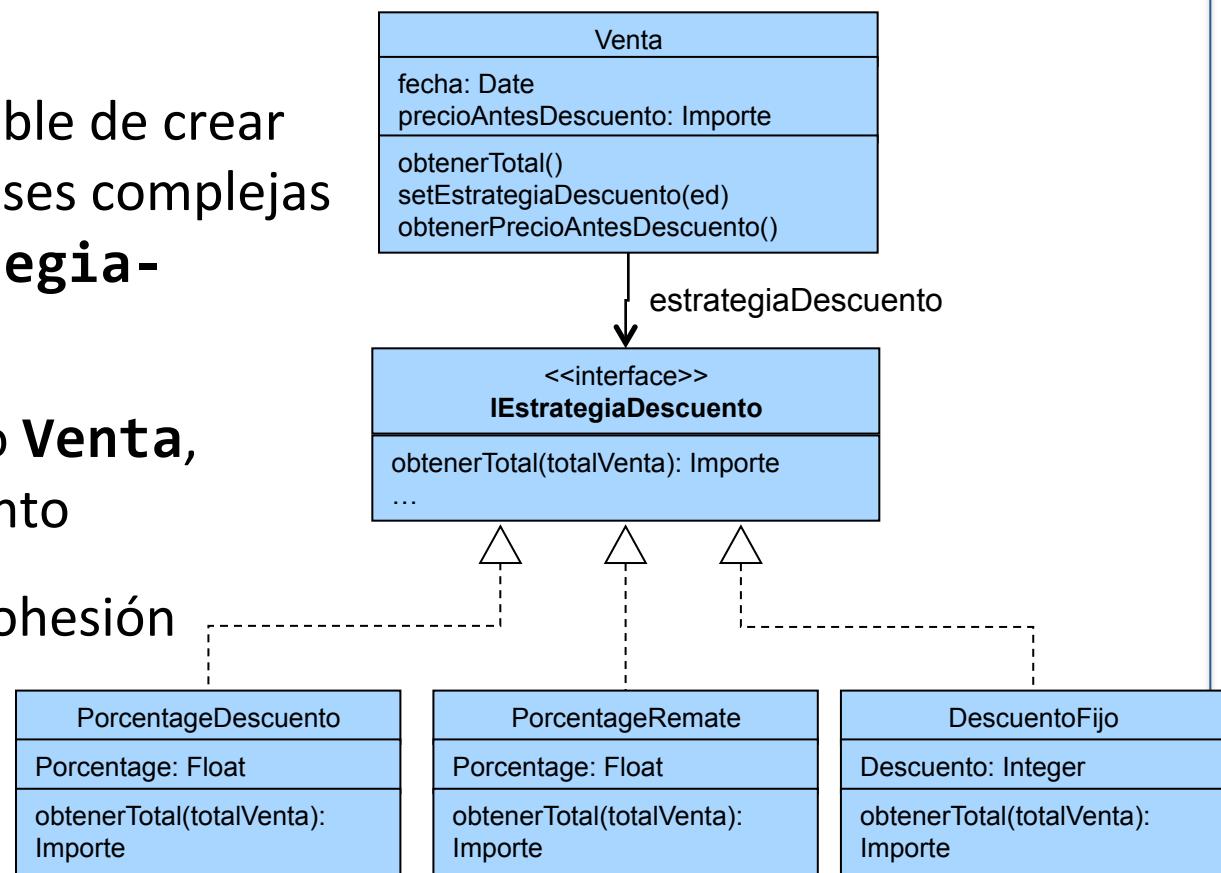
5

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 *factoría*, de objetos complejos

FactoríaEstrategiaDescuento:

- 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

FactoríaEstrategiaDescuento

```
getDescuentoPorcentual(): IEstrategiaDescuento  
getDescuentoFijo(): IEstrategiaDescuento  
getDescuentoRemate(): IEstrategiaDescuento
```

Tipos de Factorías:

- *Abstracta*: Proporciona una interfaz para construir familias de objetos (relacionados o dependientes), sin especificar sus clases concretas; se apoya en clases de factorías 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 Factoría

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 *Factoría*— prácticamente inventado, cuyo único propósito es que realice la construcción de los otros objetos

6

El patrón de diseño **Singleton**

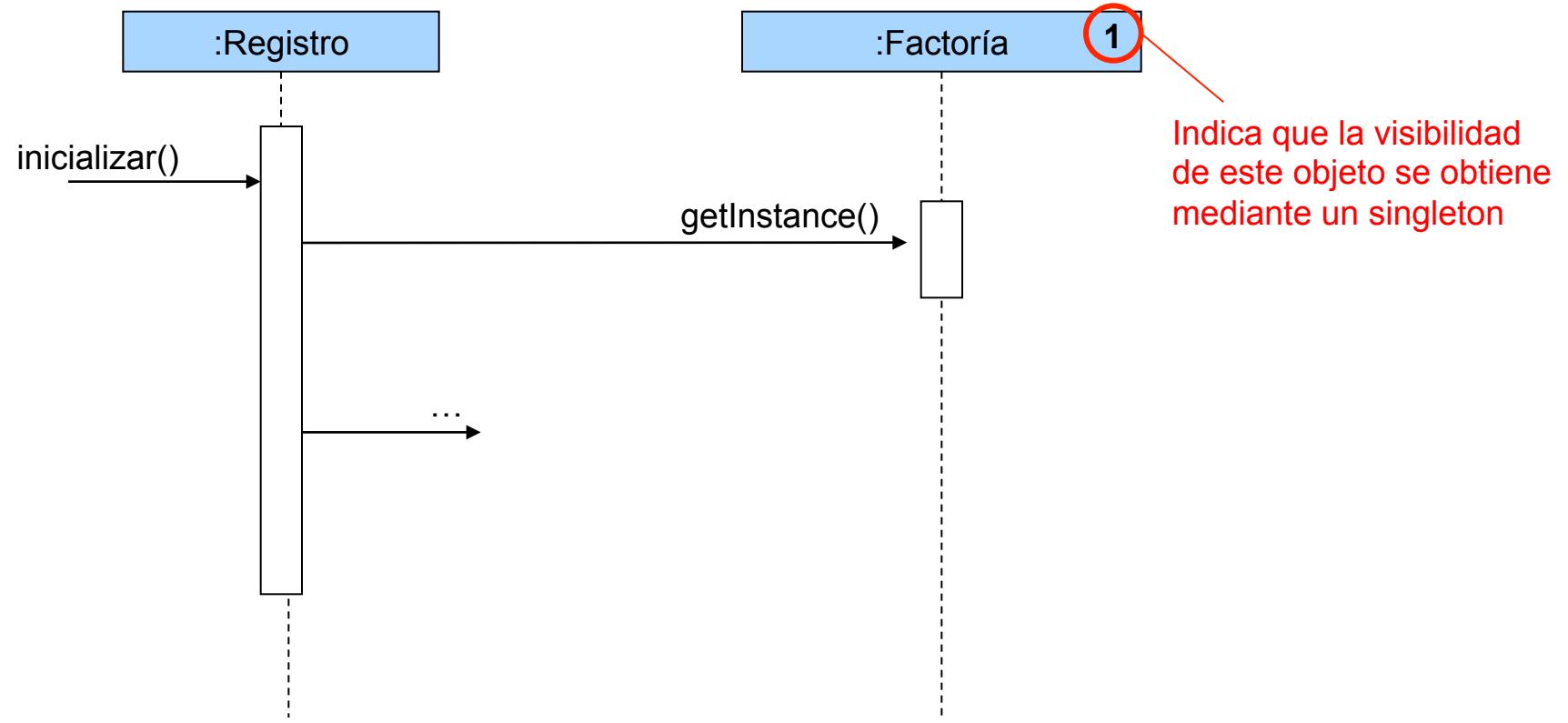
Asegura que una clase tiene una única instancia,

... 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

7

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

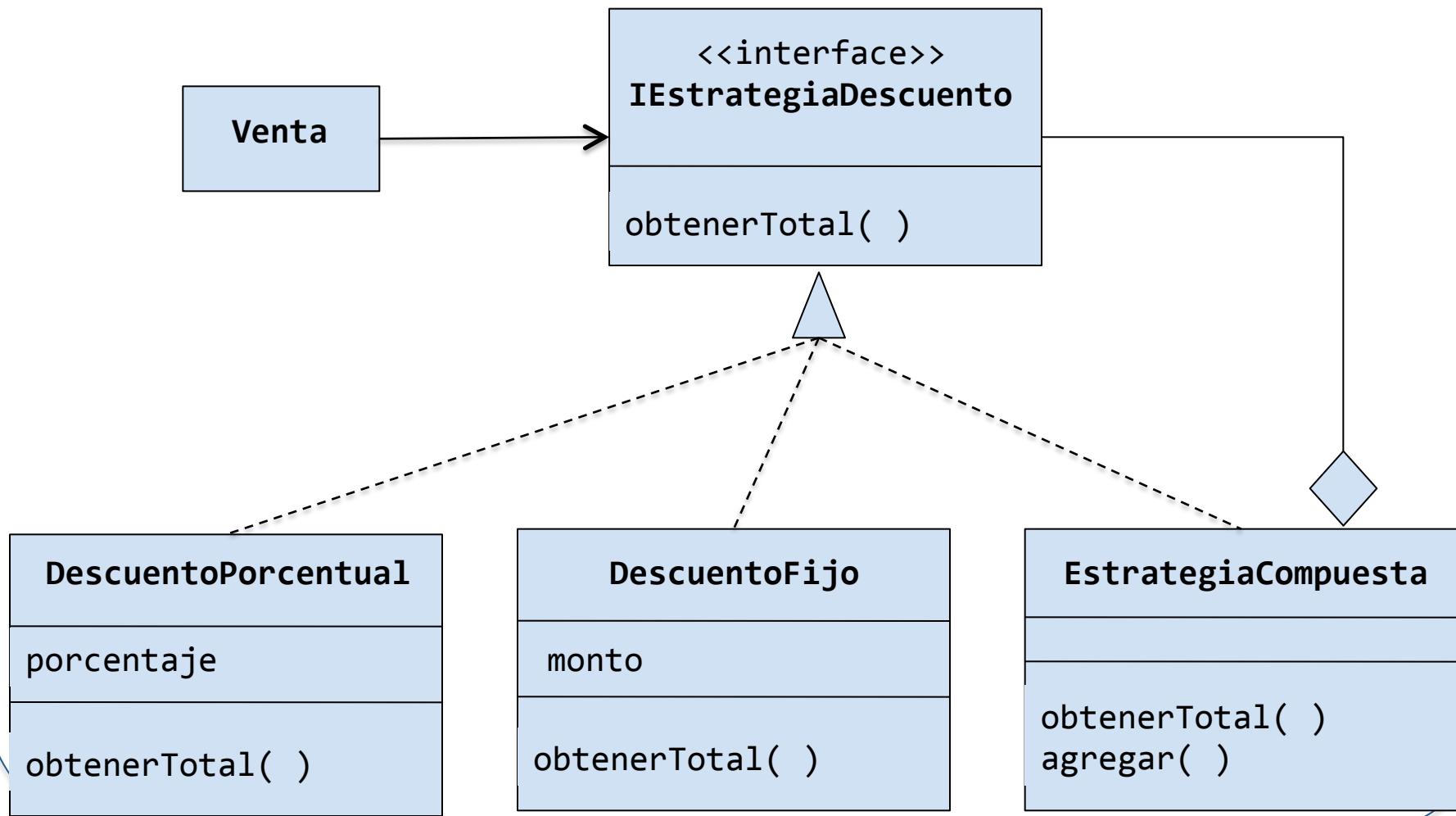
Dos desafíos importantes:

- 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**

