2

(IV)

Decorator

(Patrones de diseño)

Diseño del Software

Grado en Ingeniería Informática del Software

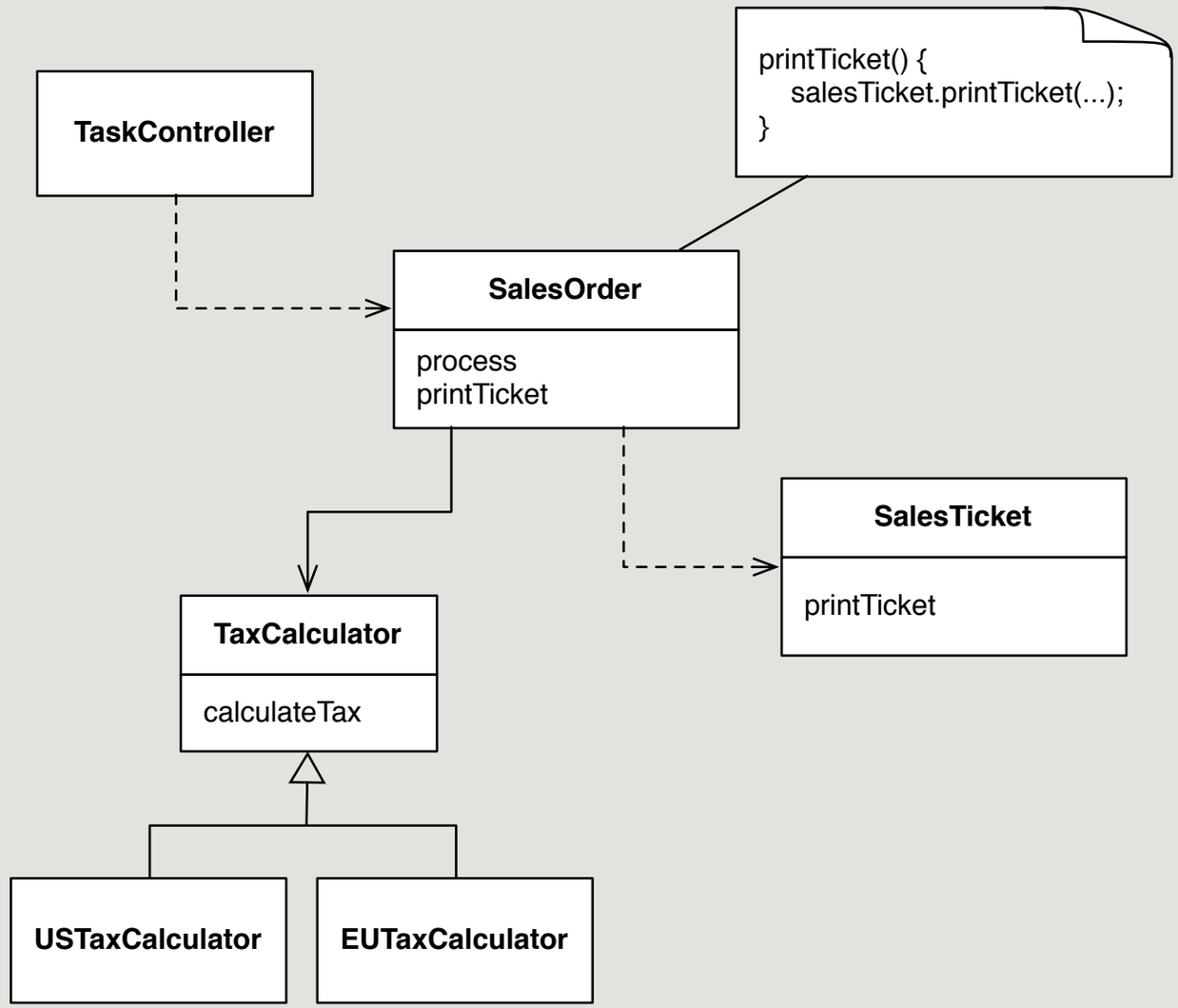
Curso 2017-2018

Escuela de Ingeniería Informática
Universidad de Oviedo

César Acebal

Ejemplo

- Una tienda necesita imprimir facturas de las compras realizadas por los clientes

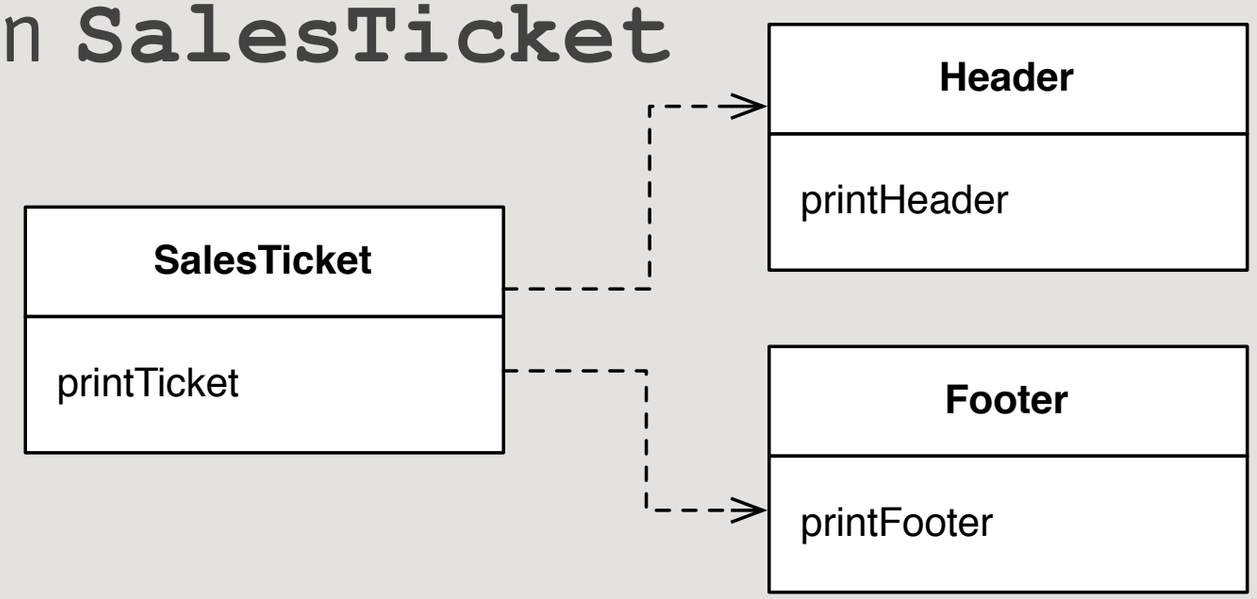


Nuevo requisito

- **Ahora algunas facturas necesitan una cabecera y un pie**
- **Primer enfoque**

- Usar condicionales en **`SalesTicket`**

```
printTicket:
if (lleva cabecera)
    header.printHeader(…);
if (lleva pie)
     footer.printFooter(…);
```



Primer enfoque

- La solución anterior funciona bastante bien si no hay que tratar con muchas opciones diferentes (muchos tipos de cabecera y pie)
- Si hubiera varios tipos, podríamos aplicar un patrón *Strategy* para las cabeceras y otro para los pies

Más combinaciones

- **Pero, ¿qué ocurre si hay que imprimir más de un tipo de cabecera o pie en una misma factura?**
- **¿Y si además el orden de éstos puede cambiar?**

HEADER 1
SALES TICKET
FOOTER 1

HEADER 1
HEADER 2
SALES TICKET
FOOTER 1

Decorator (Decorador)

◉ Intención

> *Permite añadir responsabilidades a un objeto dinámicamente. Los decoradores proporcionan una alternativa flexible a la herencia para extender la funcionalidad.*

◉ También conocido como

- Wrapper (Envoltorio)

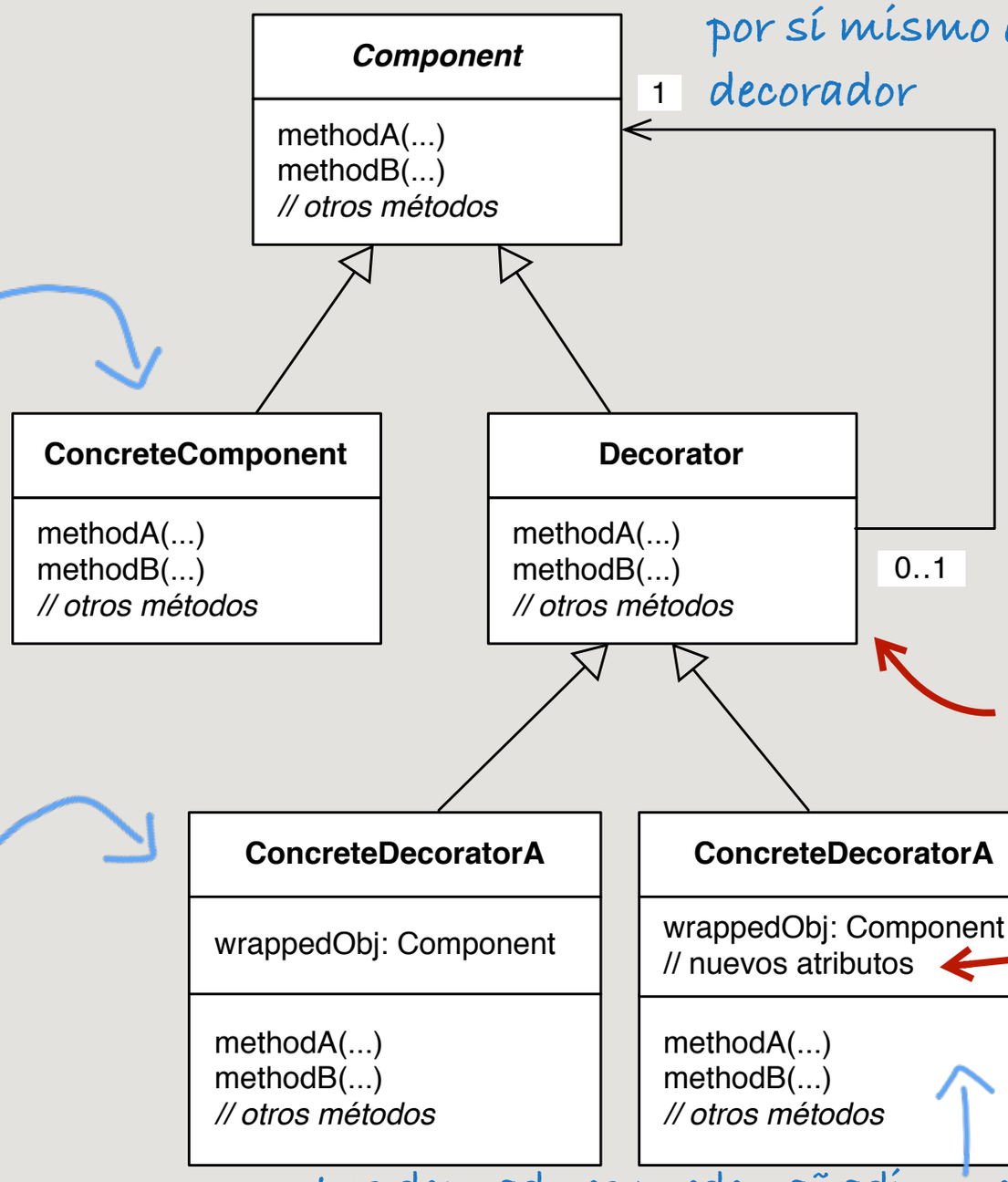
3 the Decorator Pattern

Decorating Objects

Just call this chapter "Design Eye for the Inheritance Guy."

We'll re-examine the typical overuse of inheritance and you'll learn how to decorate your classes at runtime using a form of object composition. Why? Once you know the techniques of decorating, you'll be able to give your (or someone else's) objects new responsibilities *without making any code changes to the underlying classes.*

Estructura



Cada componente se puede usar por sí mismo o envuelto en un decorador

Cada decorador TIENE-UN (envuelve) componente (un atributo que es una referencia a un objeto de tipo Component)

El componente concreto es el objeto al que añadimos dinámicamente nuevo comportamiento.

Los decoradores implementan la misma interfaz o clase abstracta (son del mismo tipo) que el componente al que decoran

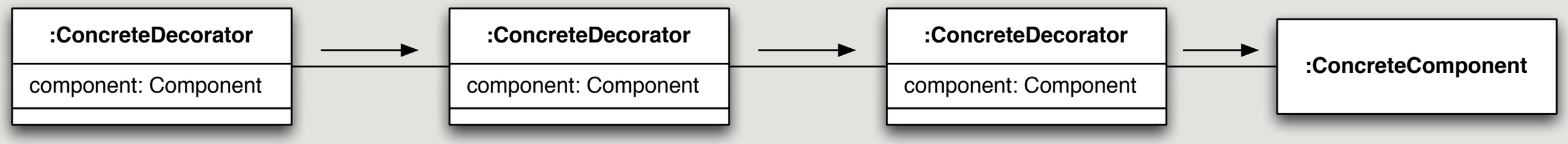
El decorador concreto tiene un atributo que guarda una referencia al objeto decorado

Los decoradores pueden ampliar el estado del componente

Los decoradores pueden añadir nuevos métodos, aunque el comportamiento nuevo normalmente se añade haciendo algo juto antes o después de llamar a un método existente en el componente

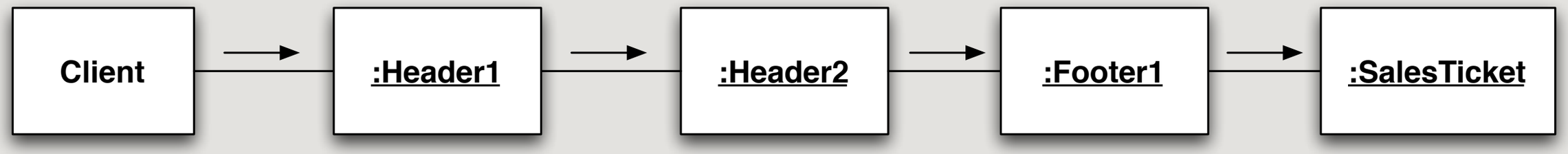
Composición de objetos

- En tiempo de ejecución, tendríamos la siguiente cadena de decoradores:

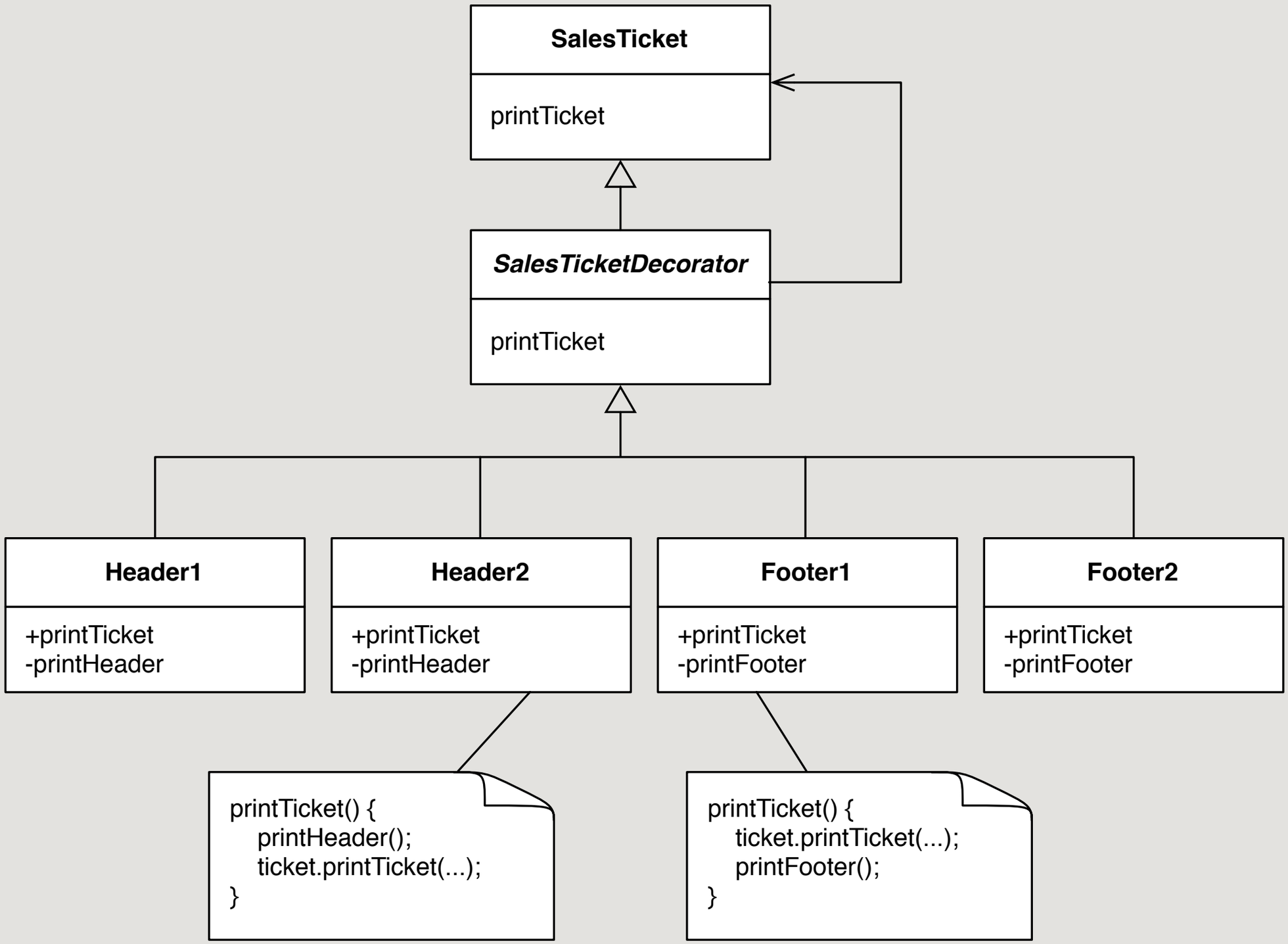


Solución para las facturas

- **¿Cómo sería aplicando el patrón Decorator?**
  - Tened en cuenta que en ejecución hemos de poder hacer cosas como ésta:



SalesTicketDecorator



Aspectos clave

- **Los decoradores tienen el mismo tipo que los objetos que decoran**
  - Por tanto, podemos pasar un decorador en vez del objeto original
- **Se puede usar uno o más decoradores para envolver un objeto**
- **El decorador añade su propio comportamiento antes o después de delegar al objeto decorado el resto del trabajo**

Aspectos clave

- **Los objetos pueden ser decorados en cualquier momento**
  - Podemos decorar objetos dinámicamente en tiempo de ejecución con tantos decoradores como queramos y en cualquier orden

El patrón Decorator según el GoF

Decorator (Decorador)

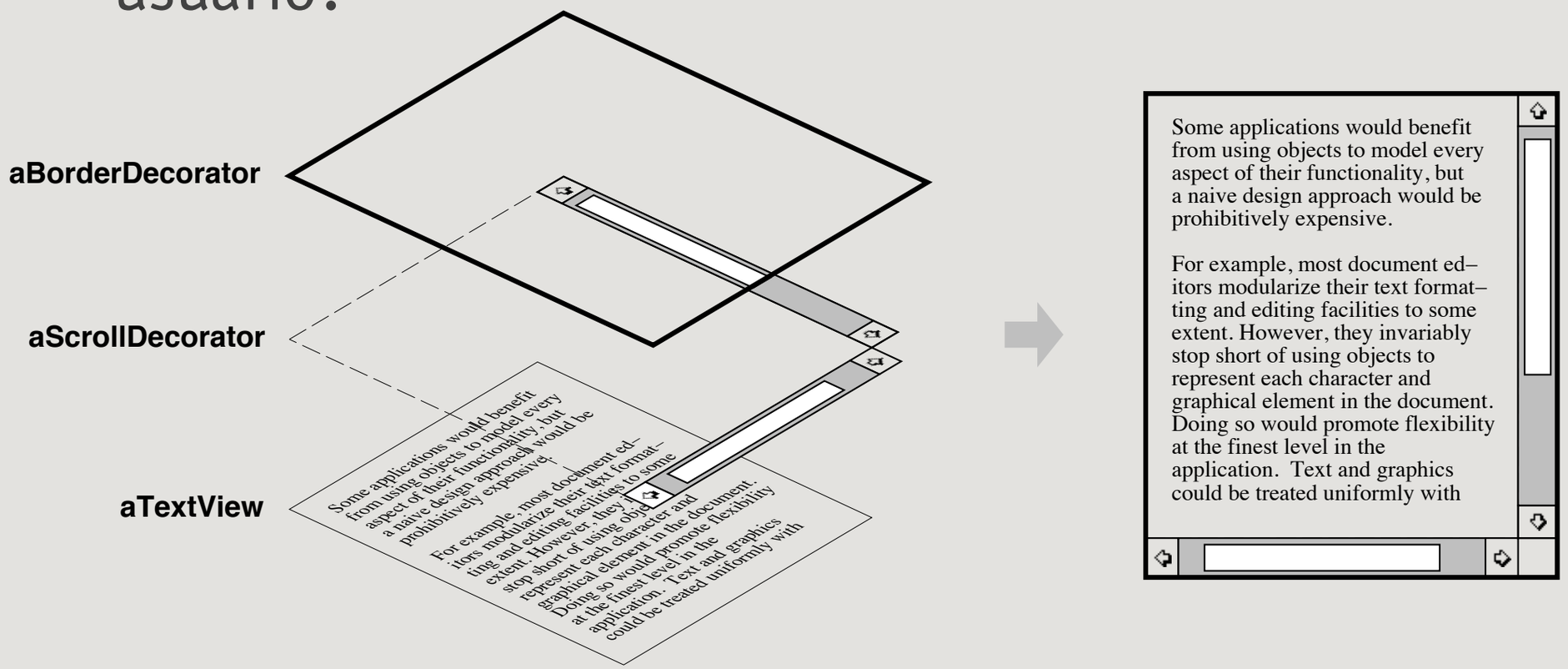
- **Patrón estructural de objetos**
- **Propósito:**

> *Añade responsabilidades adicionales a un objeto dinámicamente. Los decoradores proporcionan una alternativa flexible a la herencia para extender la funcionalidad.*

- **También conocido como:**
  - Wrapper (Envoltorio)

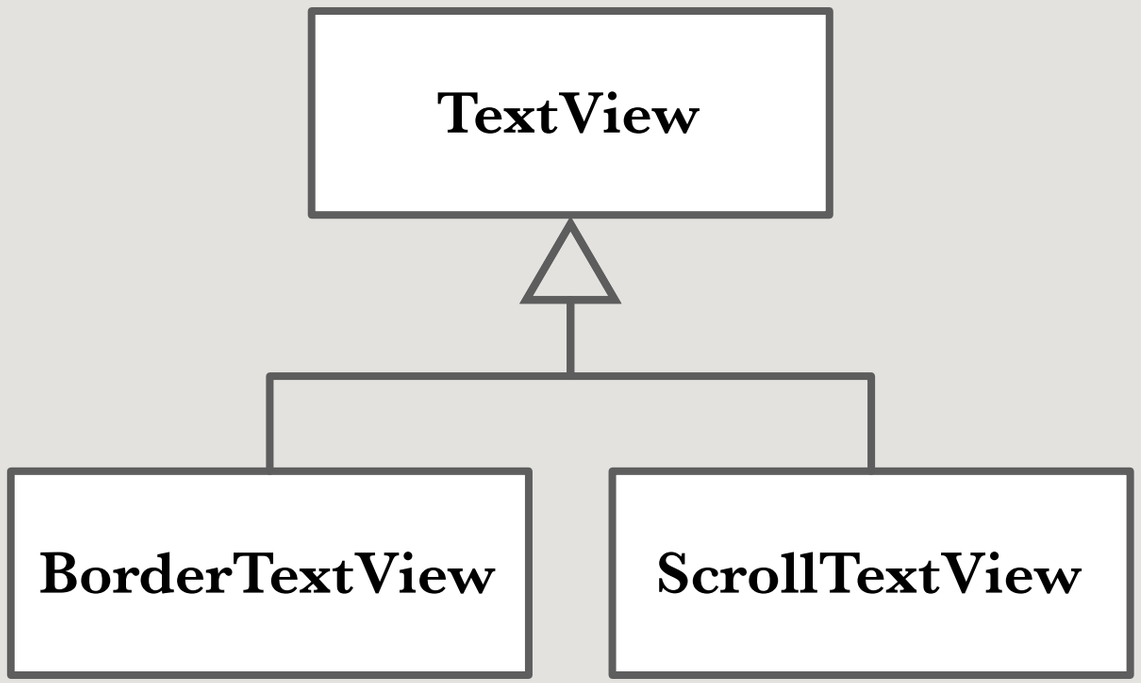
Motivación

- **A veces queremos añadir responsabilidades a objetos individuales, no a toda una clase**
  - Por ejemplo, ¿cómo añadiríamos un borde o una barra de desplazamiento a un componente de interfaz de usuario?



Motivación

- Primera alternativa: mediante la herencia



Motivación

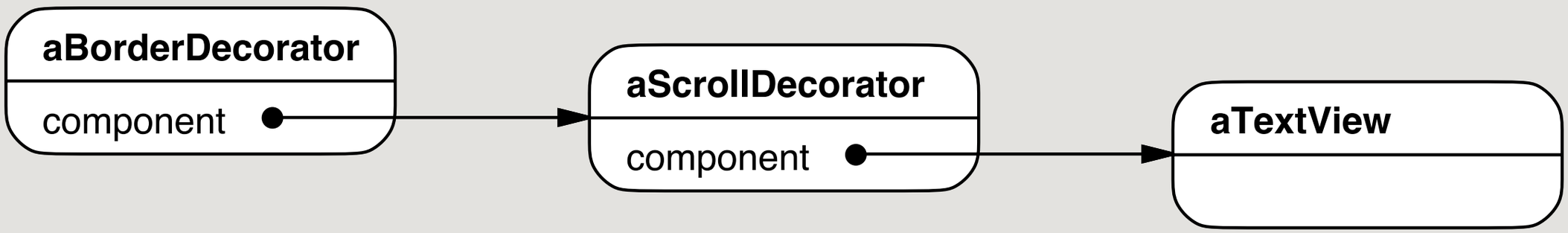
- **¿Problemas?**
  - Inflexible: la elección del borde se hace estáticamente, no la puede hacer el cliente
  - Explosión de clases: ¿qué pasaría si queremos un **`TextView`** con borde y barra de desplazamiento?

Motivación

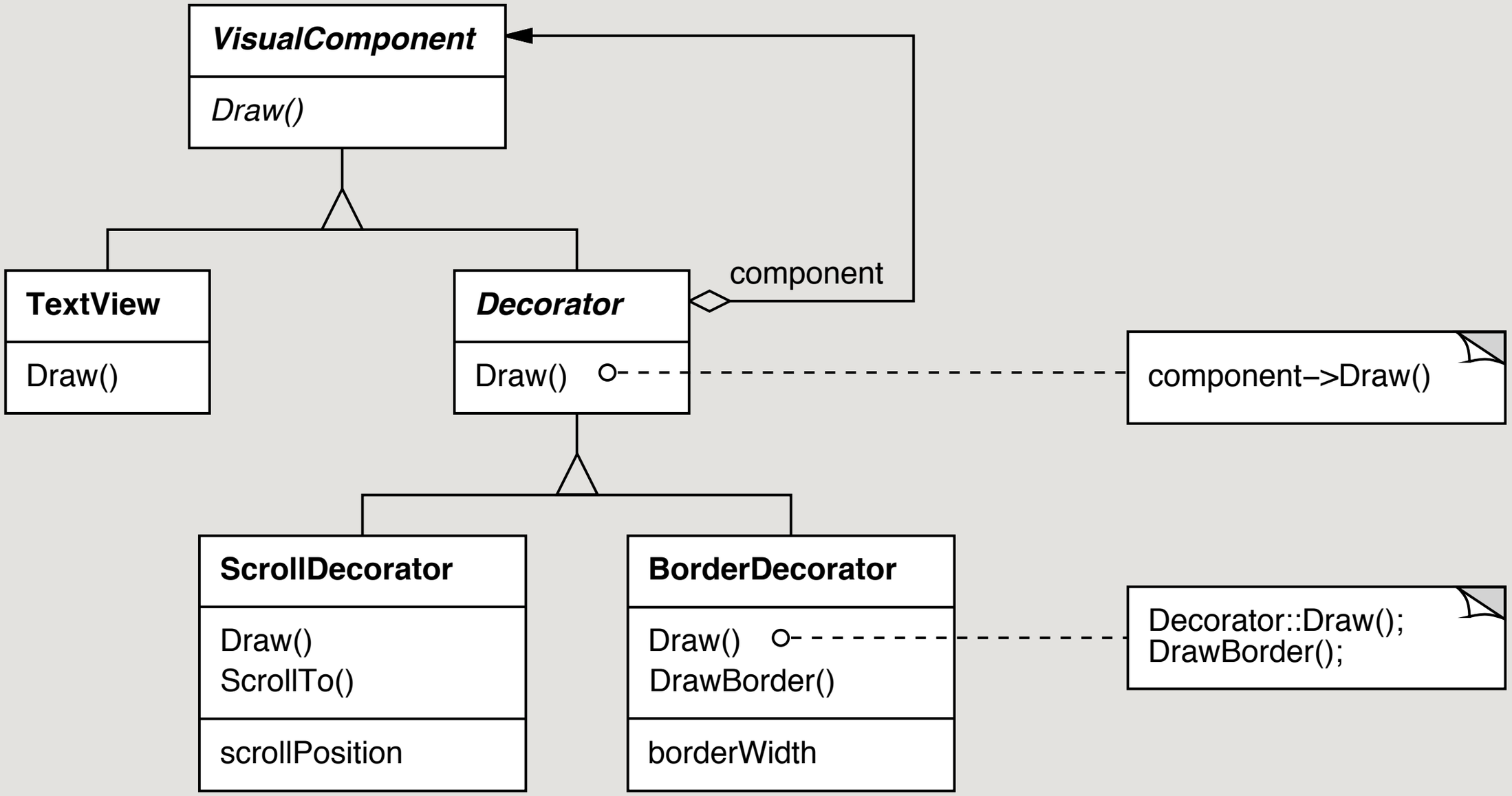
- **Una solución más flexible es envolver el componente en otro objeto que sea quien añada el borde**
  - Este objeto envoltorio es el decorador
- **El decorador sigue cumpliendo la interfaz del objeto original, así que su presencia es transparente para los clientes del componente**
  - El decorador delega las peticiones al componente y puede llevar a cabo acciones adicionales
  - La transparencia permite anidar decoradores de forma recursiva

Motivación: ejemplo

- El siguiente diagrama de objetos muestra un `TextView` con borde y barra de desplazamiento



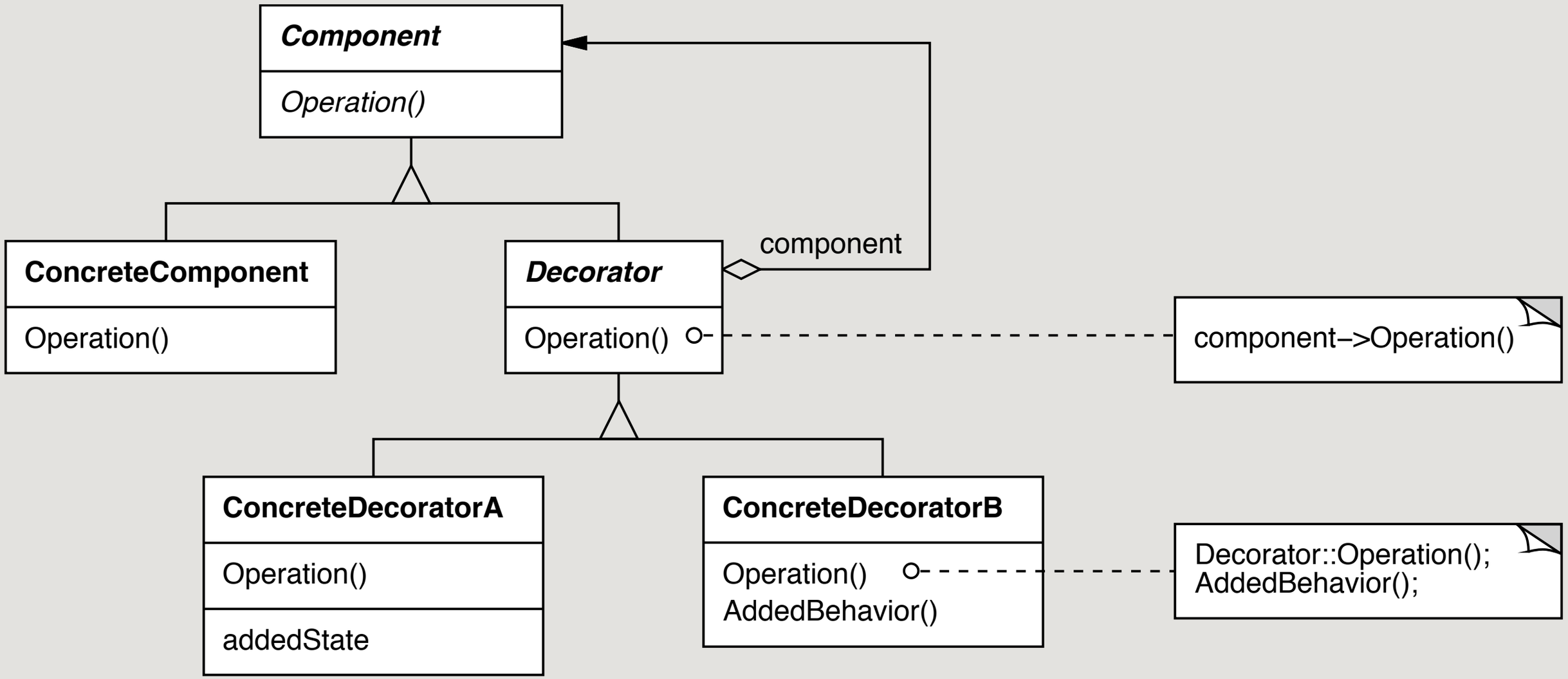
Motivación: ejemplo



Aplicabilidad

- Para añadir responsabilidades a otros objetos dinámicamente y de forma transparente
- Cuando no se puede heredar o no resulta práctico (explosión de subclases para permitir cada combinación posible)

Estructura



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

Participantes

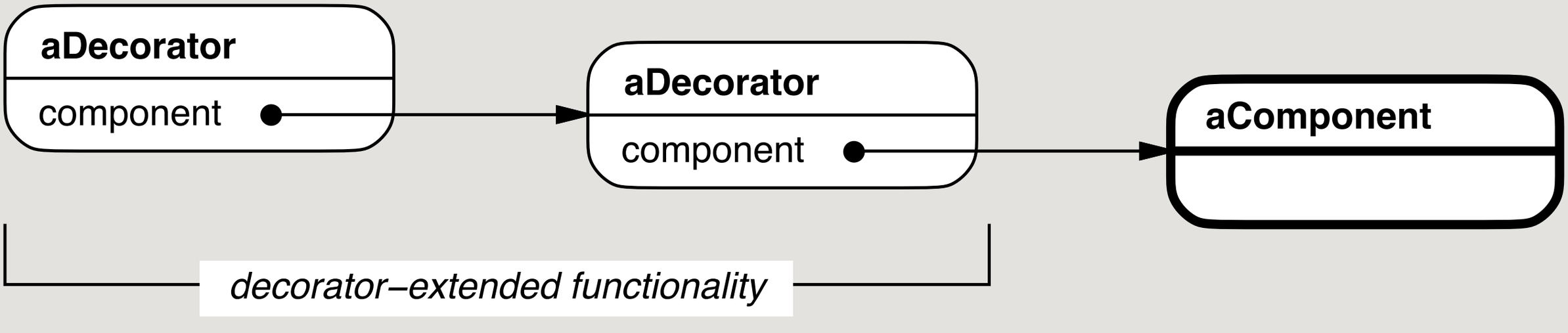
- **Component (VisualComponent)**
  - Define la interfaz de los objetos a los que se les puede añadir responsabilidades dinámicamente
- **ConcreteComponent (TextView)**
- **Decorator**
  - Mantiene una referencia a un objeto Component y tiene su misma interfaz
- **ConcreteDecorator (BorderDecorator, ScrollDecorator)**
  - Añade responsabilidades al componente

Consecuencias

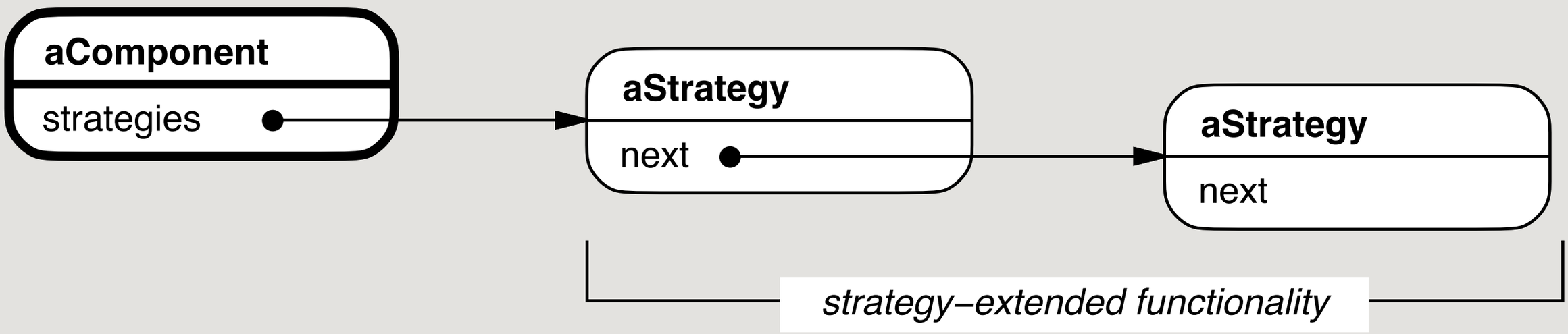
- **Ventajas**
  - Más flexibilidad que la herencia estática
  - Evita que las clases de arriba de la jerarquía estén repletas de funcionalidades
    - En vez de definir una clase compleja para tratar de dar cabida a todas ellas, la funcionalidad se logra añadiendo decoradores a una clase simple
- **Inconvenientes**
  - Un decorador y sus componentes no son idénticos
    - Desde el punto de vista de la identidad de objetos
  - Muchos objetos pequeños
    - El sistema puede ser más difícil de aprender y de depurar

Implementación

Cambiar la «piel» del objeto en vez de sus «tripas»

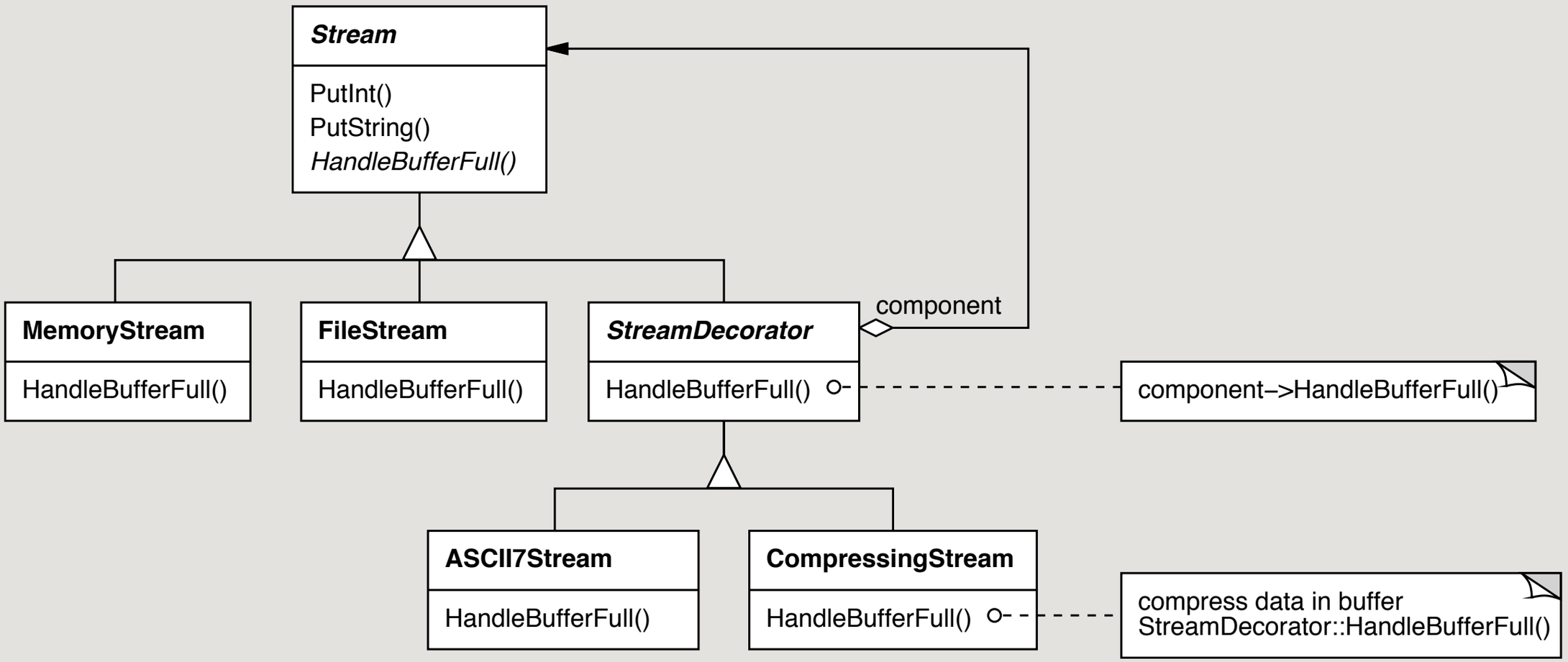


- **Una alternativa sería emplear el patrón** *Strategy*
  - Por ejemplo, el componente podría permitir distintos tipos de borde delegando el dibujado de éste a un objeto **`Border`** aparte (la estrategia)



Usos conocidos

- Muchas bibliotecas de interfaces gráficas de usuario
- Los flujos en las bibliotecas de entrada/salida



Patrones relacionados

- **Adapter**
  - El decorador sólo cambia las responsabilidades del objeto, no su interfaz
- **Composite**
  - Un decorador puede verse como un «composite» de un solo componente
  - Pero el decorador añade responsabilidades adicionales (no está pensado para la agregación de objetos)
- **Strategy**
  - El decorador cambia la «piel» del objeto; una estrategia cambia sus «tripas»

Ejemplo real

Java I/O

```
Reader in = new BufferedReader
    (new InputStreamReader(System.in));
```

- **Ejemplos:**
  - BufferedInputStream
  - LineNumberInputStream
  - DataInputStream
  - PushbackInputStream

¿Qué tienen en común estas clases?

Sí, son decoradores concretos.
Pero, ¿qué clase de la API de Java haría las veces del decorador en sí?

Decoradores en Java I/O