

13

Decorator

Diseño del Software

Grado en Ingeniería Informática del Software

2024-2025

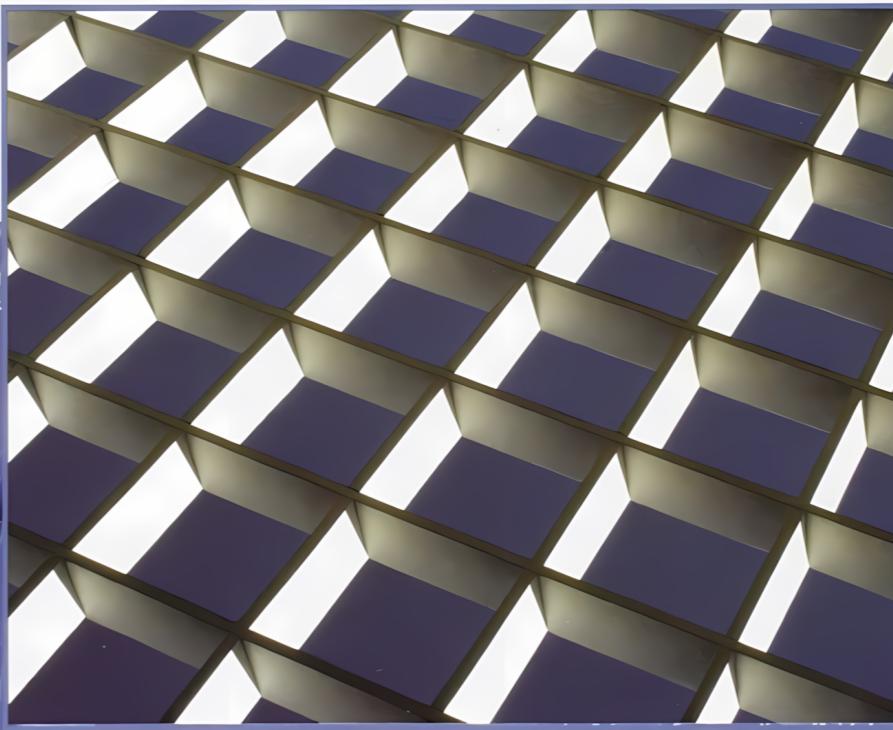
Una tienda necesita imprimir facturas de las compras realizadas por los clientes.



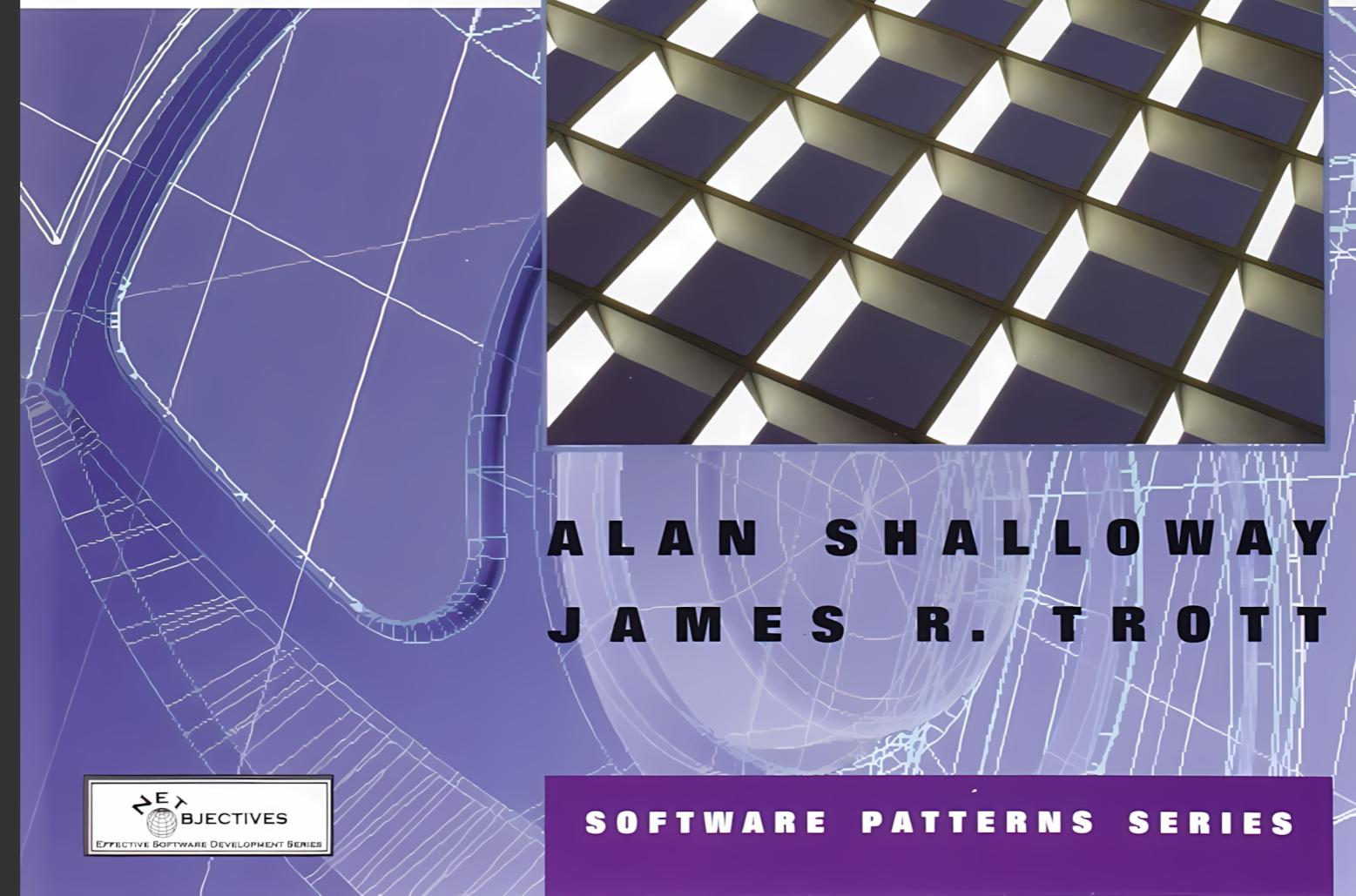
DESIGN PATTERNS EXPLAINED

A New Perspective on Object-Oriented Design

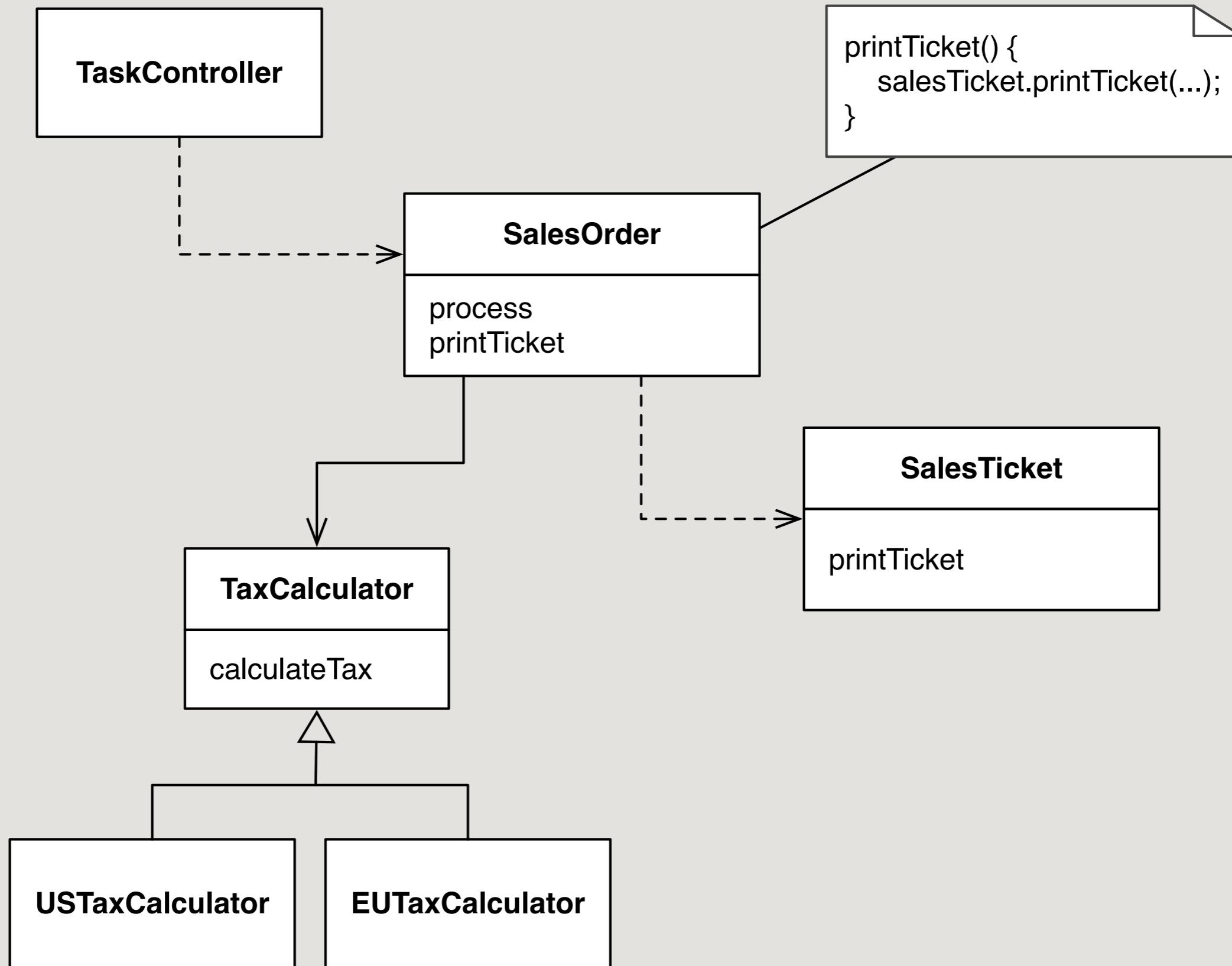
SECOND EDITION



**ALAN SHALLOWAY
JAMES R. TROTT**

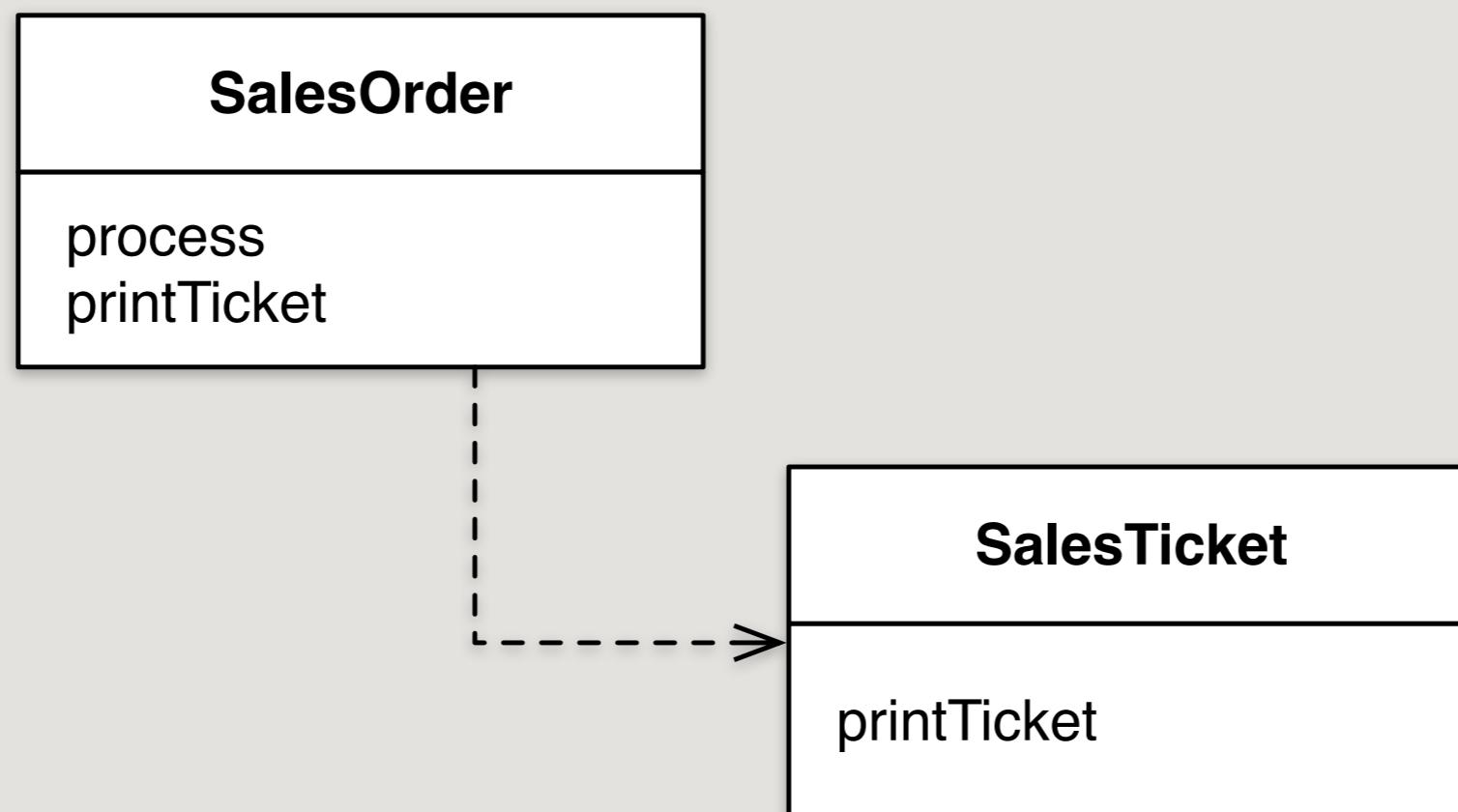


SOFTWARE PATTERNS SERIES



Ahora aparece un nuevo requisito: algunas facturas necesitan que lleven una cabecera y un pie.

Ahora aparece un nuevo requisito: algunas facturas necesitan que lleven una cabecera y un pie.



¿Cómo lo hacemos?

Un primer enfoque sería llevar el control de la cabecera y el pie directamente en la clase SalesTicket, mediante lógica condicional que nos dijese si hay que imprimirlos o no.

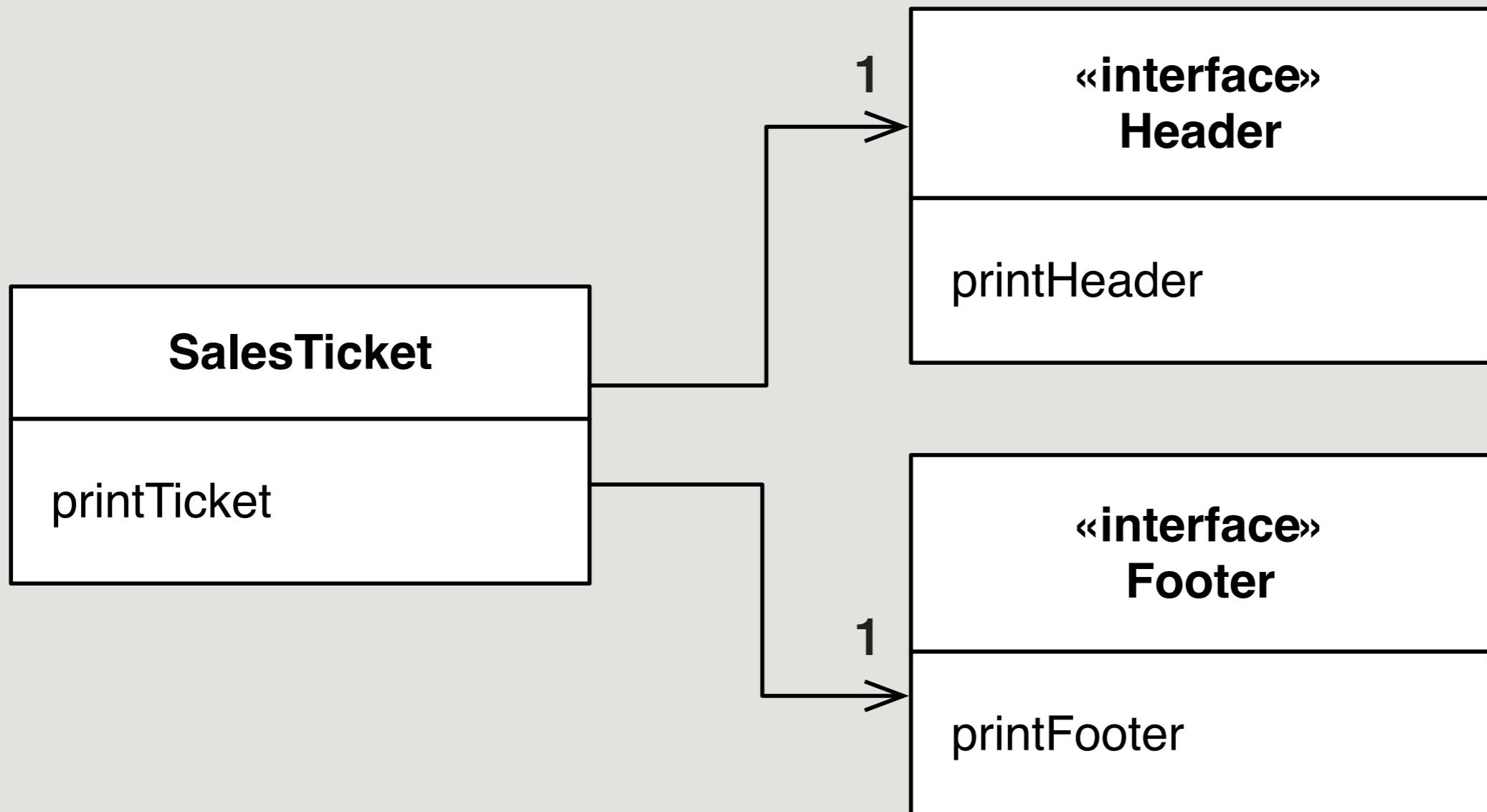


SalesTicket.java

```
printTicket {  
    if (has header)  
        header.printHeader();  
    // print ticket here  
    if (has footer)  
        footer.printFooter();  
}
```

La solución anterior funciona bastante bien si no hay que tratar con tipos distintos de cabeceras y pies y si el orden de estos no cambia.

Si hubiera que tratar con muchos tipos distintos de cabeceras y pies, imprimiendo solo uno cada vez, podríamos pensar en usar un patrón **Strategy** para las cabeceras y otro para los pies.



Pero, ¿qué ocurre si hay que imprimir más de un tipo de cabecera o pie en una misma factura?

¿Y si el orden de estos puede cambiar?

Pero, ¿qué ocurre si hay que imprimir más de un tipo de cabecera o pie en una misma factura?

¿Y si el orden de estos puede cambiar?

HEADER 1

SALES TICKET

FOOTER 1

HEADER 1

HEADER 2

SALES TICKET

FOOTER 1

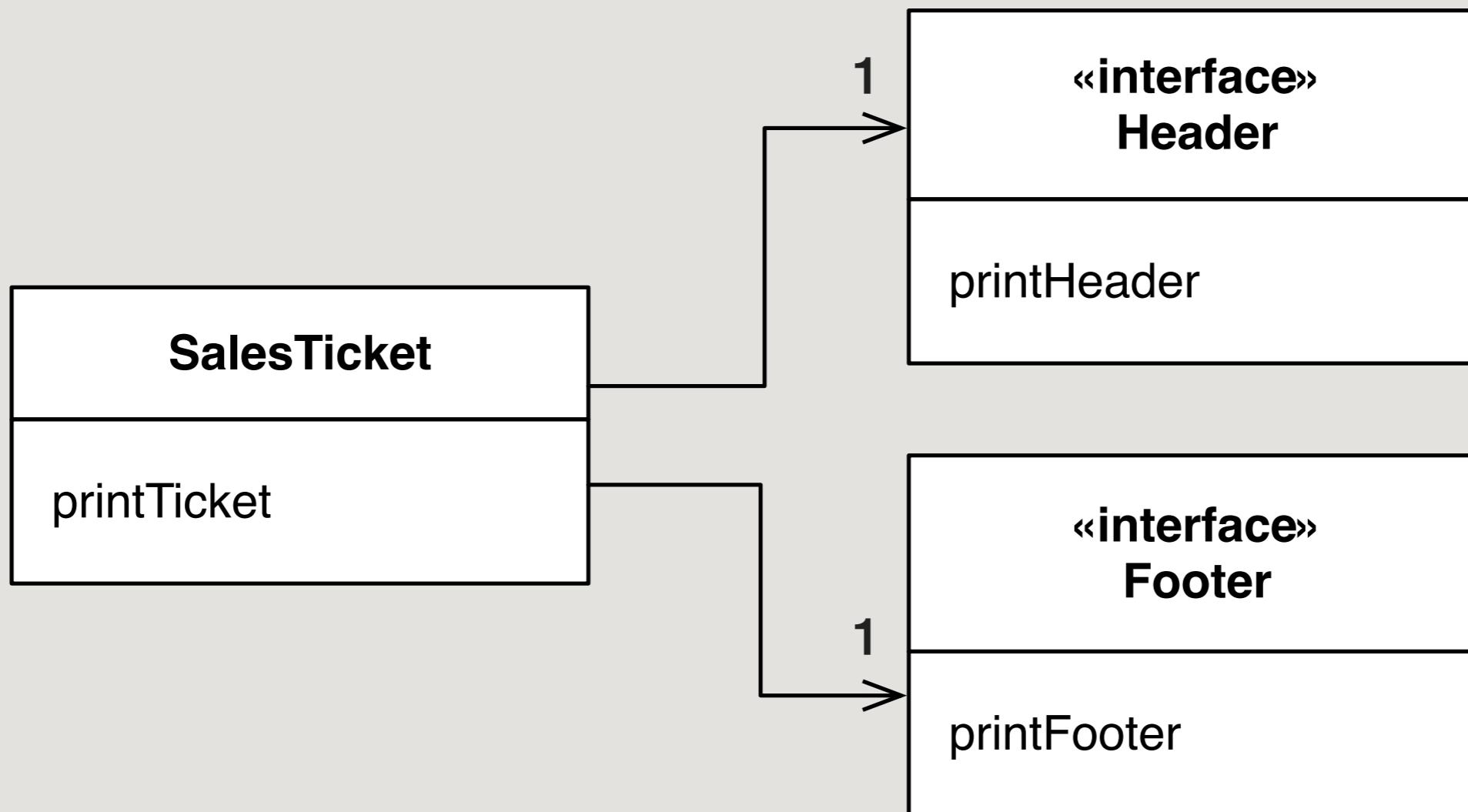
El número de combinaciones
empieza a ser desmesurado.

De hecho, son infinitas.

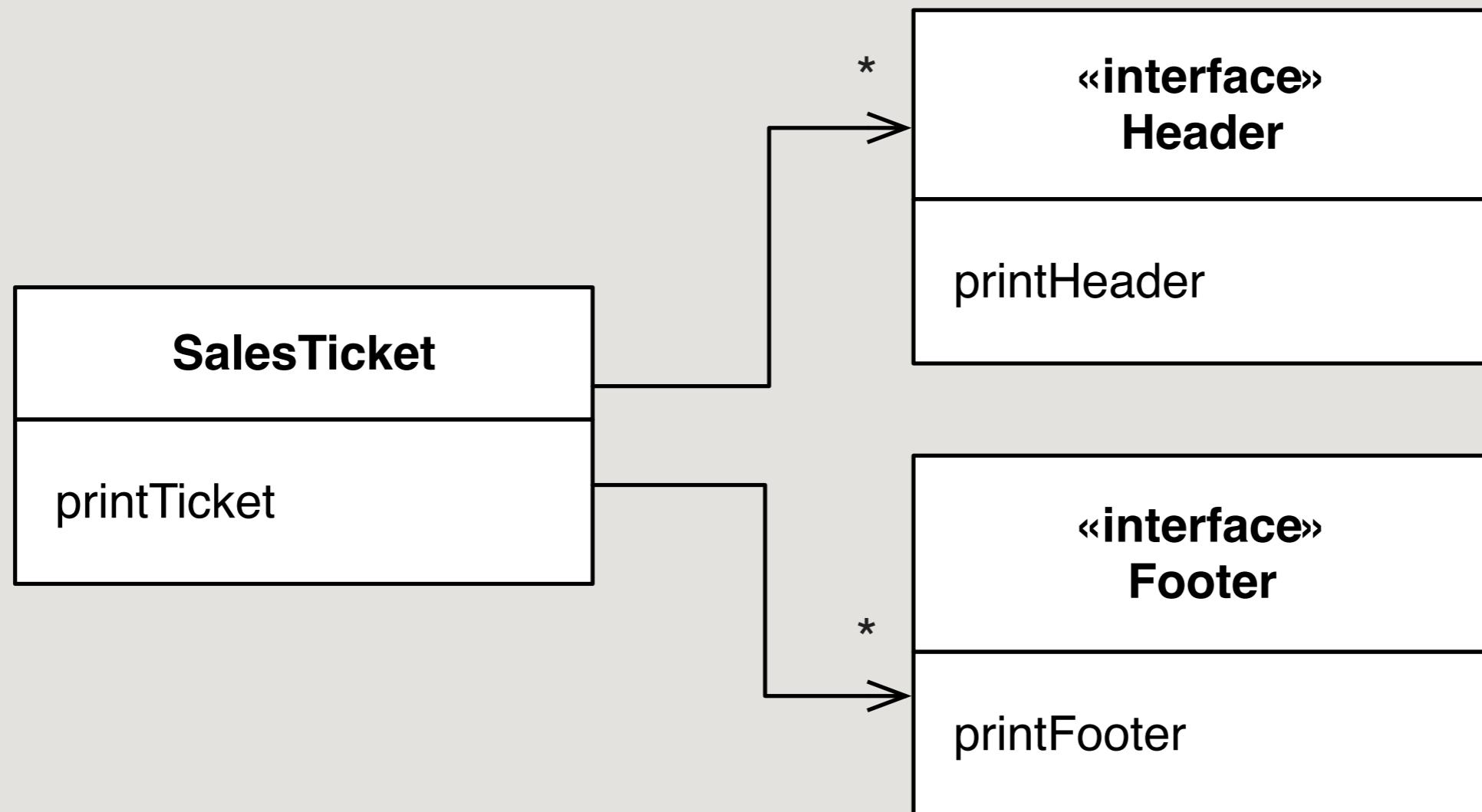
Si bien el libro del que se ha extraído este ejemplo utiliza el requisito anterior como argumento para introducir el patrón **Decorator**, existe otra opción perfectamente válida en este caso.

¿Se os ocurre cómo?

Podríamos haber utilizado, en vez de una estrategia, una lista de estrategias (dos, en este caso).



Podríamos haber utilizado, en vez de una estrategia, una lista de estrategias (dos, en este caso).



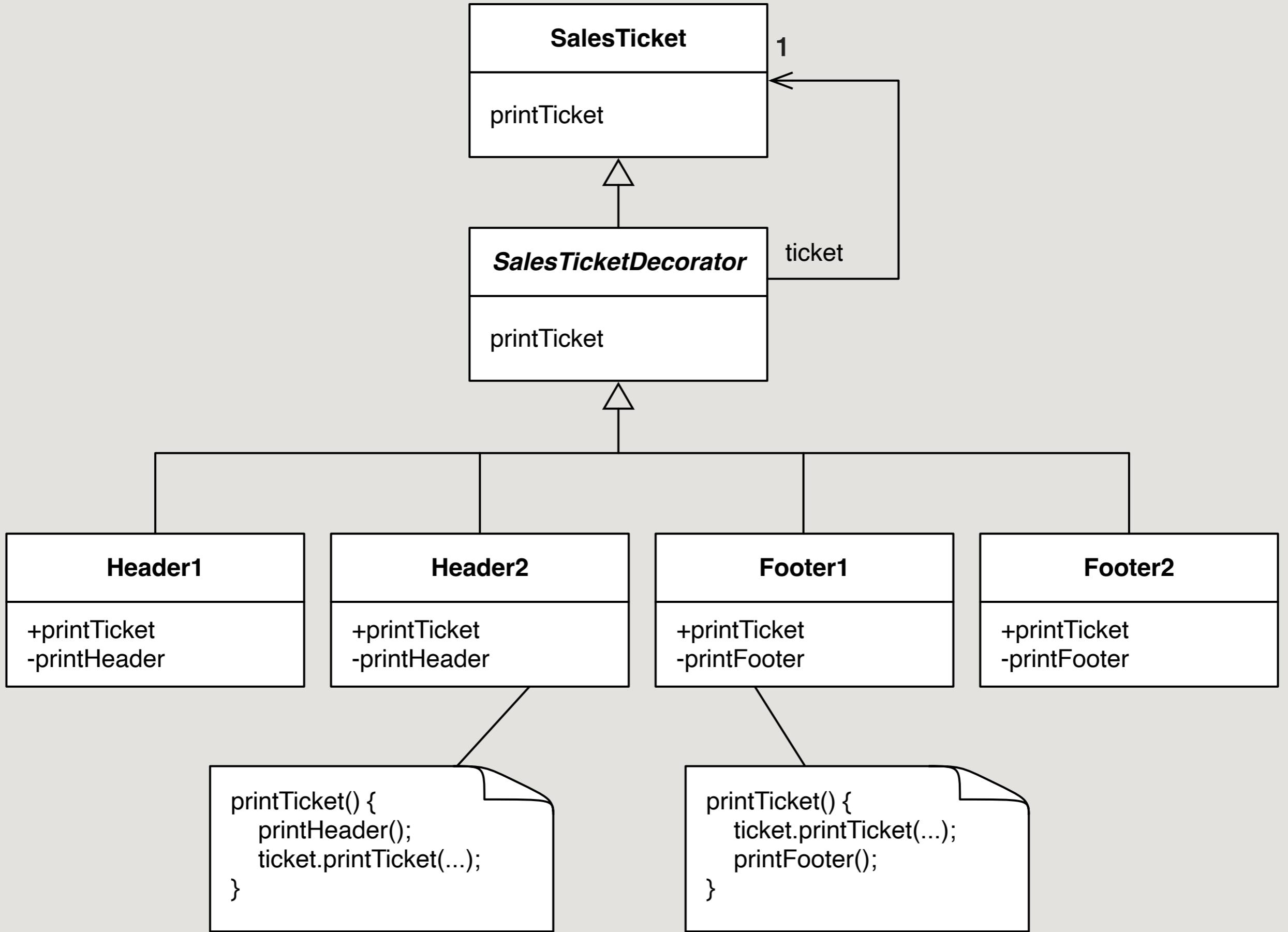


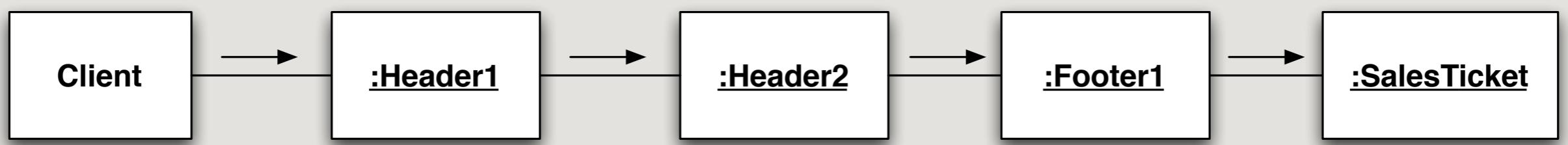
SalesTicket.java

```
printTicket {  
    for (Header header : headers) {  
        header.printHeader();  
    }  
    // print ticket here  
    for (Footer footer : footers) {  
        footer.printFooter();  
    }  
}
```

Usando dos listas de estrategias

Solución con decoradores





Una estructura de objetos típica en tiempo de ejecución del patrón Decorator

En vez de añadir funcionalidad a través de un método centralizado de control, el patrón Decorator lo hace construyendo una cadena, envolviendo las funciones (objetos) deseadas unas en otras en el orden adecuado.

**El patrón Decorator separa la creación de
la cadena del cliente que la usa.**

En este caso, SalesOrder.

Decorator

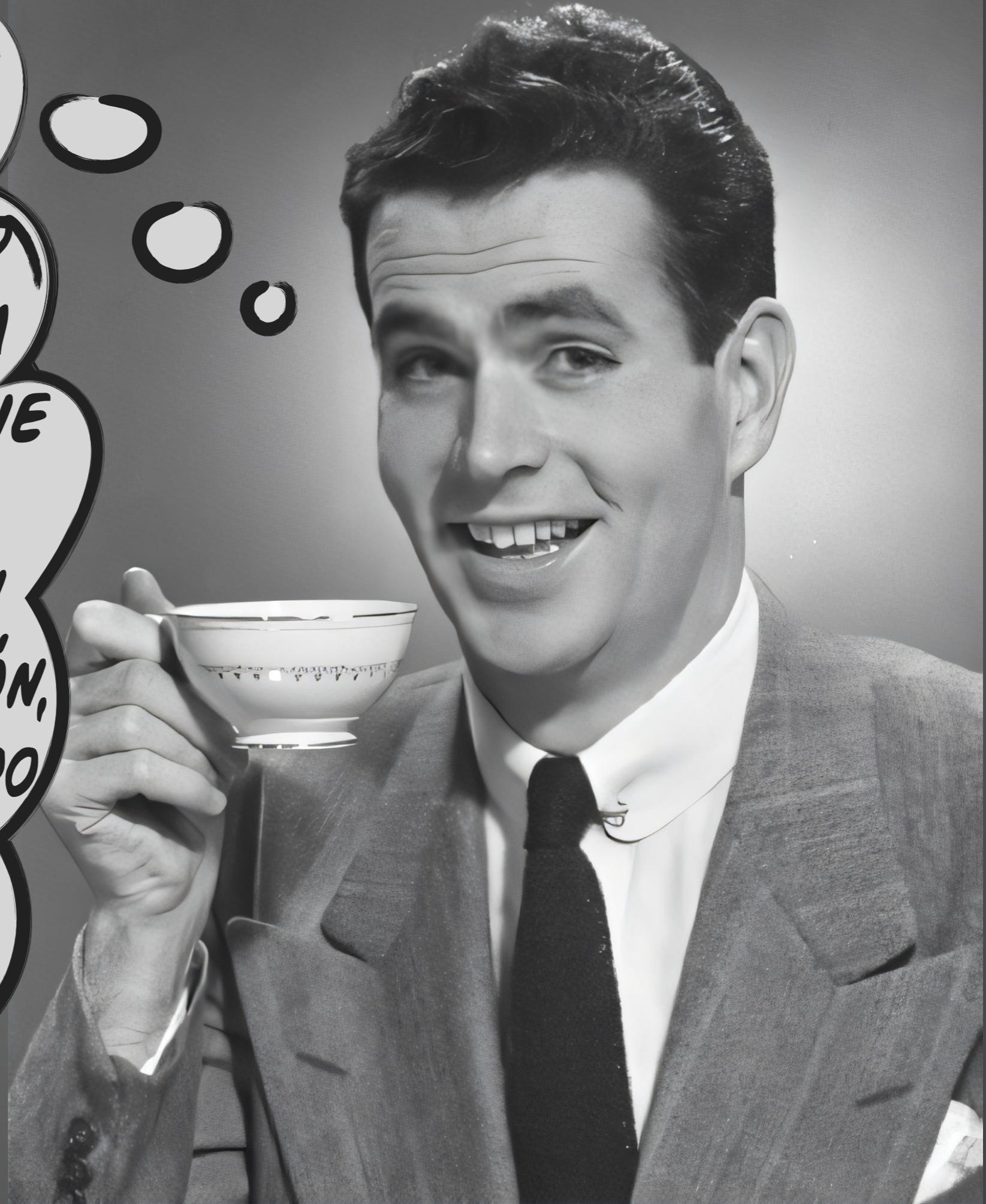
Patron estructural de objetos

Aunque en el libro lo clasifican como patrón estructural, tal vez debería haber sido más bien uno de comportamiento.

Puesto que añade responsabilidades en tiempo de ejecución.

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

PENSABA QUE LOS
HOMBRES DE VERDAD
USABAN LA HERENCIA
PARA TODO, HASTA QUE
DESCUBRÍ EL PODER
DE LA EXTENSIÓN EN
TIEMPO DE EJECUCIÓN,
EN VEZ DE EN TIEMPO
DE COMPILACIÓN. ¡Y
MÍRAME AHORA!



**También conocido
como
Wrapper (Envoltorio)**

Motivación

A veces queremos añadir responsabilidades a objetos individuales, no a toda la clase.

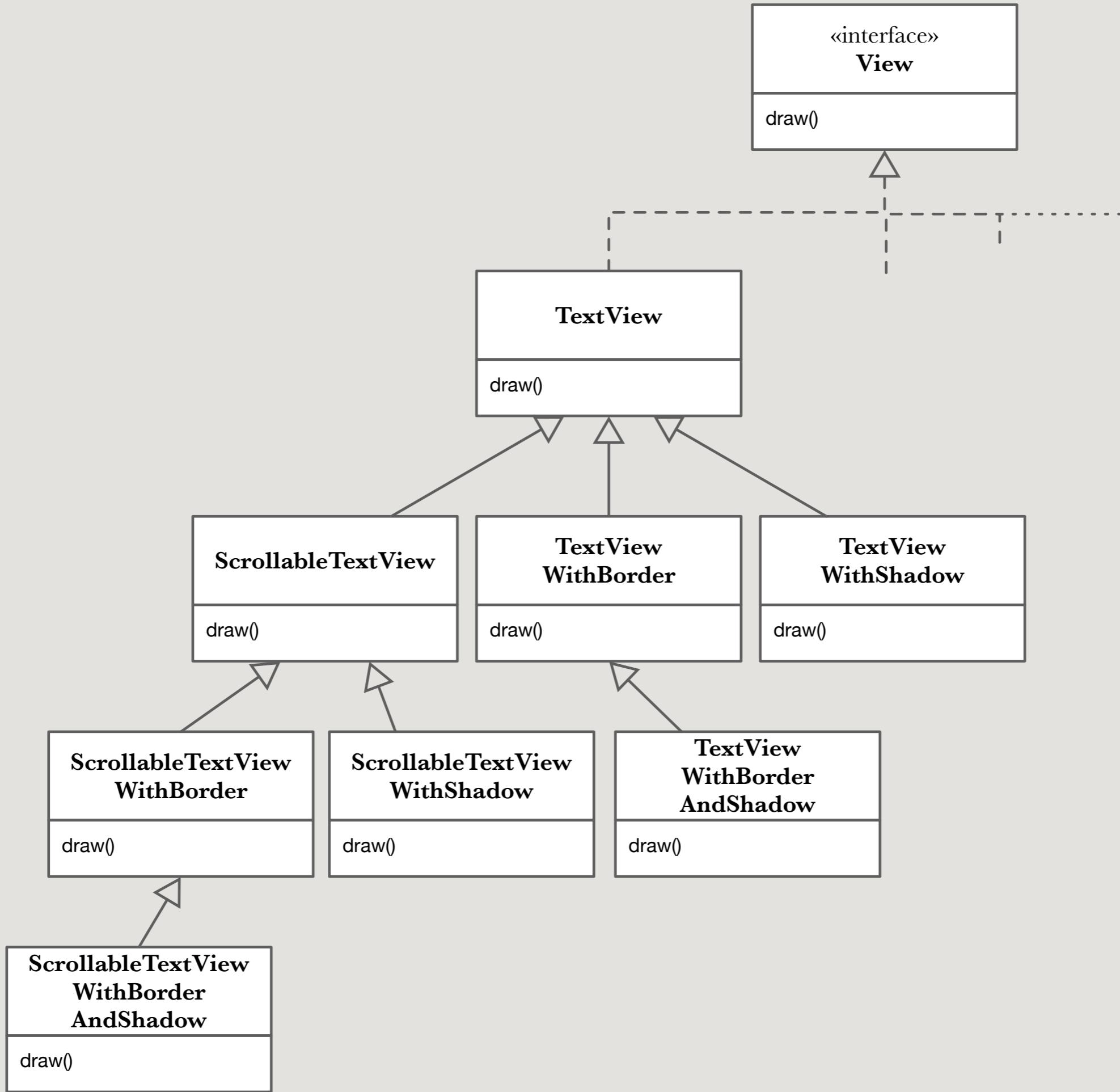
Una biblioteca gráfica, por ejemplo, debería permitir añadir bordes o barras de desplazamiento a cualquier componente de la interfaz de usuario.

Una forma de añadir responsabilidades es mediante la herencia. Así, heredar de un componente con borde añadiría el borde a cada instancia de dicha subclase.

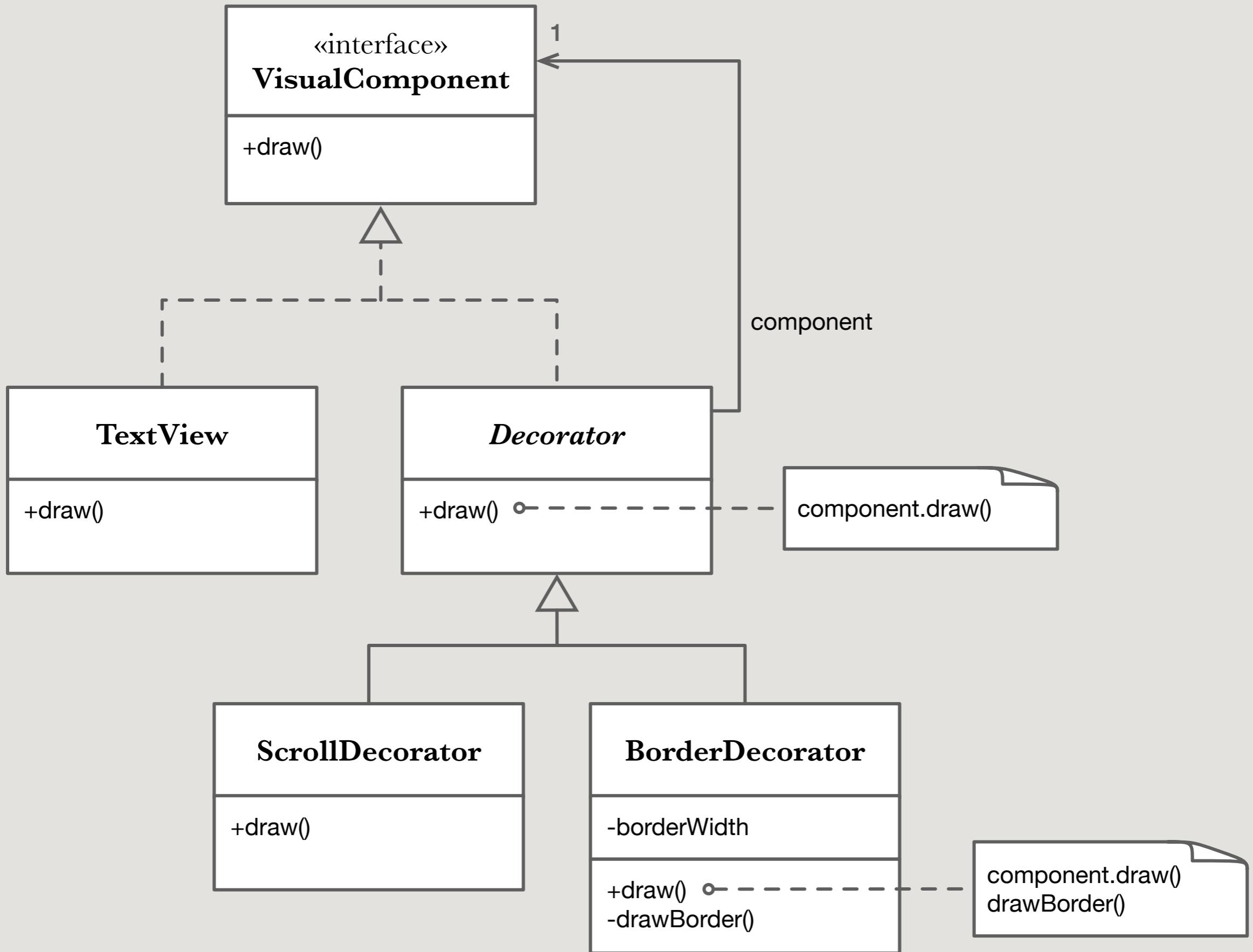
Sin embargo esto es inflexible, ya que la elección de si lleva borde o no se hace **estáticamente**, en el **momento de la creación**.

Un cliente no puede controlar **cómo** y **cuándo** decorar el componente con un borde.

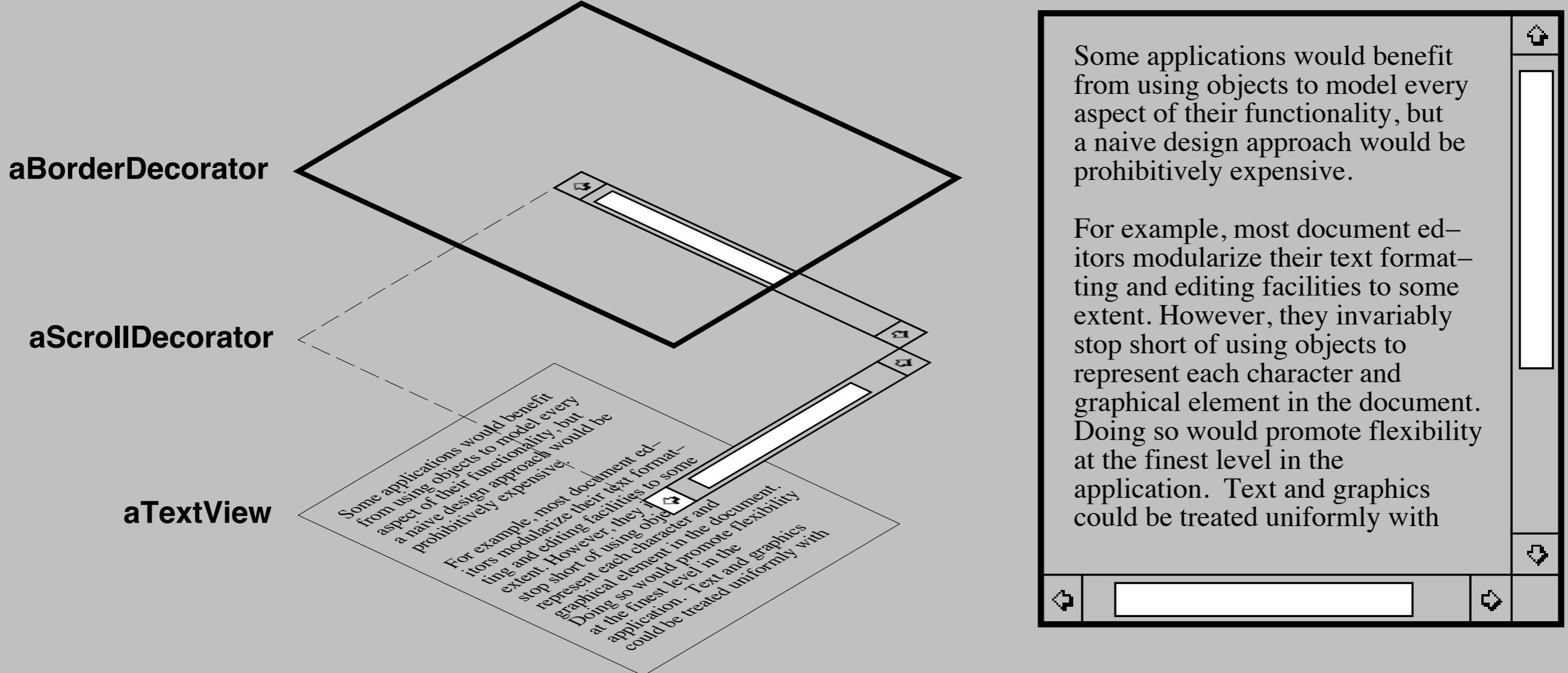
Por otro lado, ¿qué ocurriría si quisieramos poder añadir también sombras, bordes...?

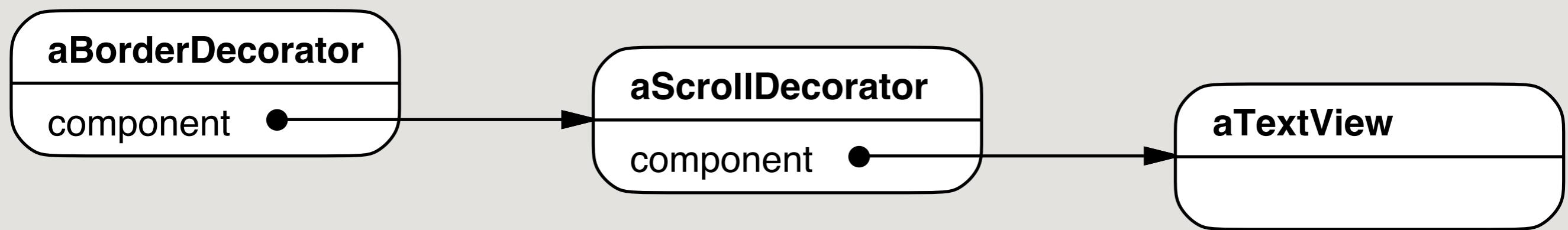


*¡Explosión de
subclases!*



El patrón Decorator nos permite hacer este tipo de cosas, incluso en tiempo de ejecución.





El decorador sigue implementando la interfaz del objeto original, por lo que su presencia es transparente para los clientes del componente.

Delega peticiones al componente y puede realizar acciones adicionales.

Se pueden anidar de forma recursiva.

Aplicabilidad

Usaremos el patrón Decorator

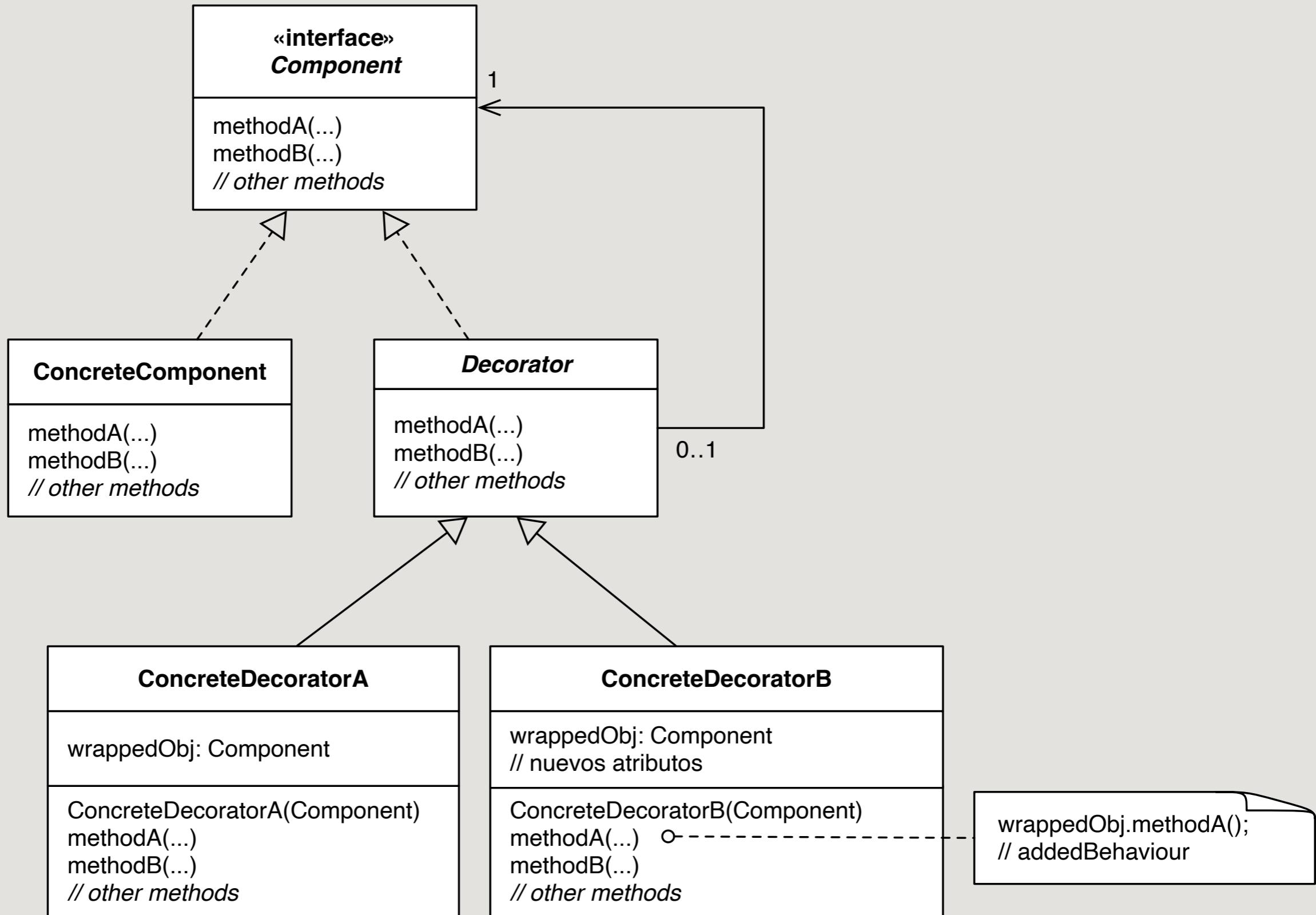
**Para añadir responsabilidades a
otros objetos dinámicamente y
de forma transparente**

Es decir, sin afectar a otros objetos ni que los clientes se percaten de ello.

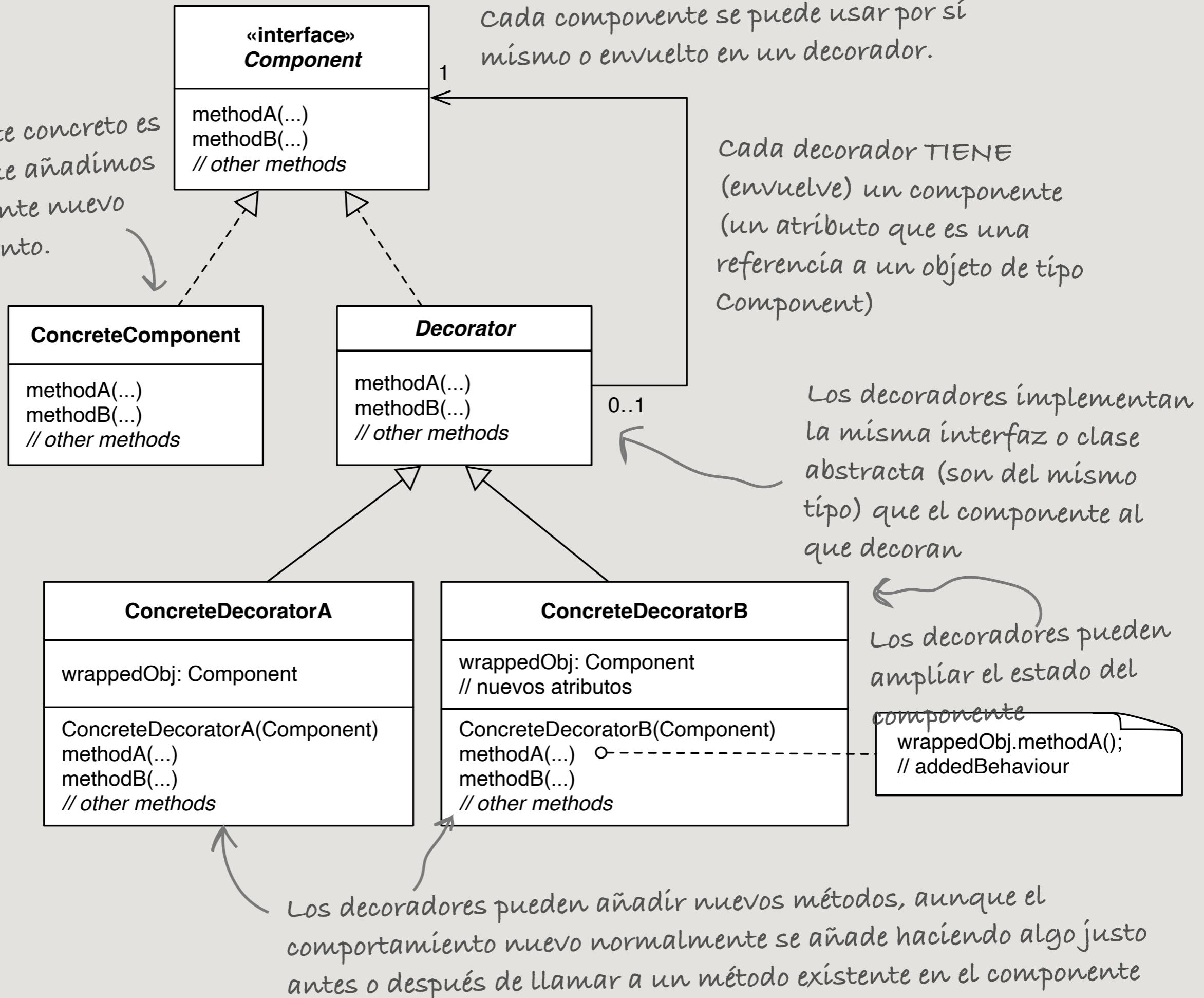
**Cuando no se puede heredar o
no resulta práctico.**

Por ejemplo, cuando se produzca una **explosión de subclases** para permitir todas las combinaciones posibles.

Estructura



La estructura del patrón Decorator



Participantes

Component

(VisualComponent)

Define la interfaz de los objetos a los que se les puede añadir responsabilidades dinámicamente.

ConcreteComponent

(TextView)

Define un objeto al que se le puede añadir responsabilidades dinámicamente.

Decorator

(Decorator)

Contiene una referencia a un objeto
Component y tiene su misma interfaz.

ConcreteDecorator

(BorderDecorator, ScrollDecorator)

Añade responsabilidades al componente.

Colaboraciones

**El decorador envía las peticiones
a su componente.**

Adicionalmente, puede llevar a cabo acciones adicionales antes o después de delegar en él.

Consecuencias

Más flexibilidad que la herencia estática

El patrón Decorator proporciona una forma más flexible de añadir responsabilidades a los objetos que la que se puede tener con la herencia estática (múltiple).

Con los decoradores, las responsabilidades pueden **añadirse y eliminarse en tiempo de ejecución**. Por el contrario, la herencia requiere crear una nueva clase para cada combinación posible (**explosión de subclases**).

BorderedScrollView, BorderedTextView...

Más flexibilidad que la herencia estática

Permite incluso añadir la misma funcionalidad varias veces.

Aunque normalmente dicha flexibilidad es una de las ventajas del patrón, puede que no siempre sea deseable.

¿Tiene sentido añadir un borde dos veces? ¿Y una barra de desplazamiento?

Evita que las clases de arriba de la jerarquía estén repletas de funcionalidades.

En vez de definir una clase compleja, configurable, para tratar de dar cabida a todas ellas, la funcionalidad se logra incrementalmente, añadiendo decoradores a una clase simple.

La funcionalidad puede componerse a partir de elementos simples.

También resulta fácil definir nuevas clases de decoradores independientemente de las clases de objetos que decoran, incluso para ampliaciones imprevistas.

Los decoradores son del mismo tipo que el objeto al que decoran.

Por lo tanto, podemos pasar un decorador en lugar del objeto original en cada lugar donde se espere un componente.

Es decir, son transparentes para los clientes.

Se pueden usar uno o varios decoradores para envolver un componente.

El decorador añade su propio comportamiento antes o después de delegar el resto del trabajo al objeto decorado.

Son también transparentes para el componente al que decoran.

Implementación

Se puede omitir la clase abstracta Decorator.

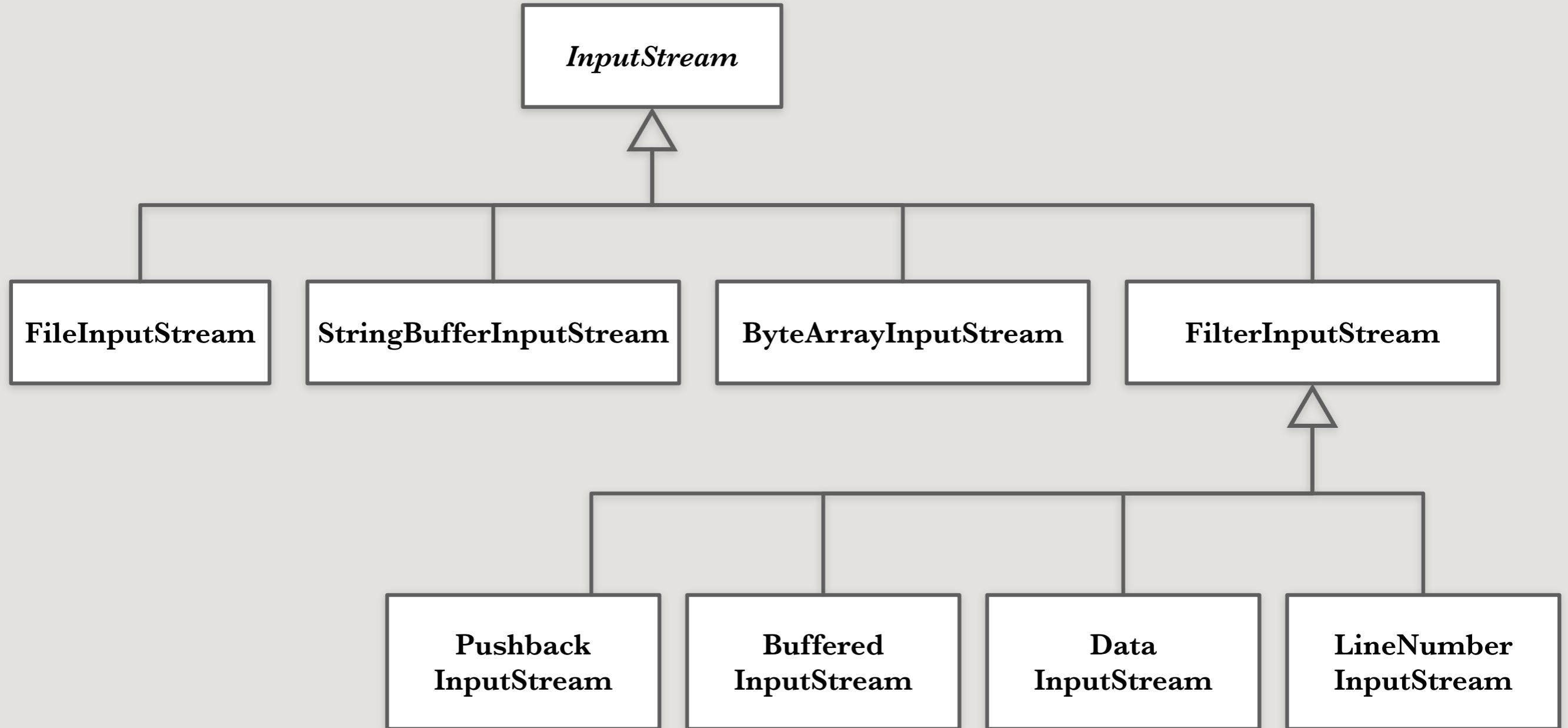
**No hay necesidad de crearla salvo que haya
código repetido.**

Resulta útil, no obstante, cuando solo se necesita decorar algunos métodos del componente, o para proporcionar una implementación por omisión que, para todos los métodos, simplemente llame al mismo método de su componente.

Usos conocidos



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



Patrones relacionados

Decorator frente a Strategy

Decorator frente a Strategy

Decorator frente a lista de estrategias

En cuanto a su intención

Un decorador añade funcionalidad, mientras que la estrategia es otra forma de hacer lo mismo (cambia la funcionalidad existente).

En cuanto a su estructura

Un decorador cambia «la piel» de un objeto; la estrategia cambia «las tripas».

Podemos pensar en un decorador como una piel sobre un objeto que cambia su comportamiento.

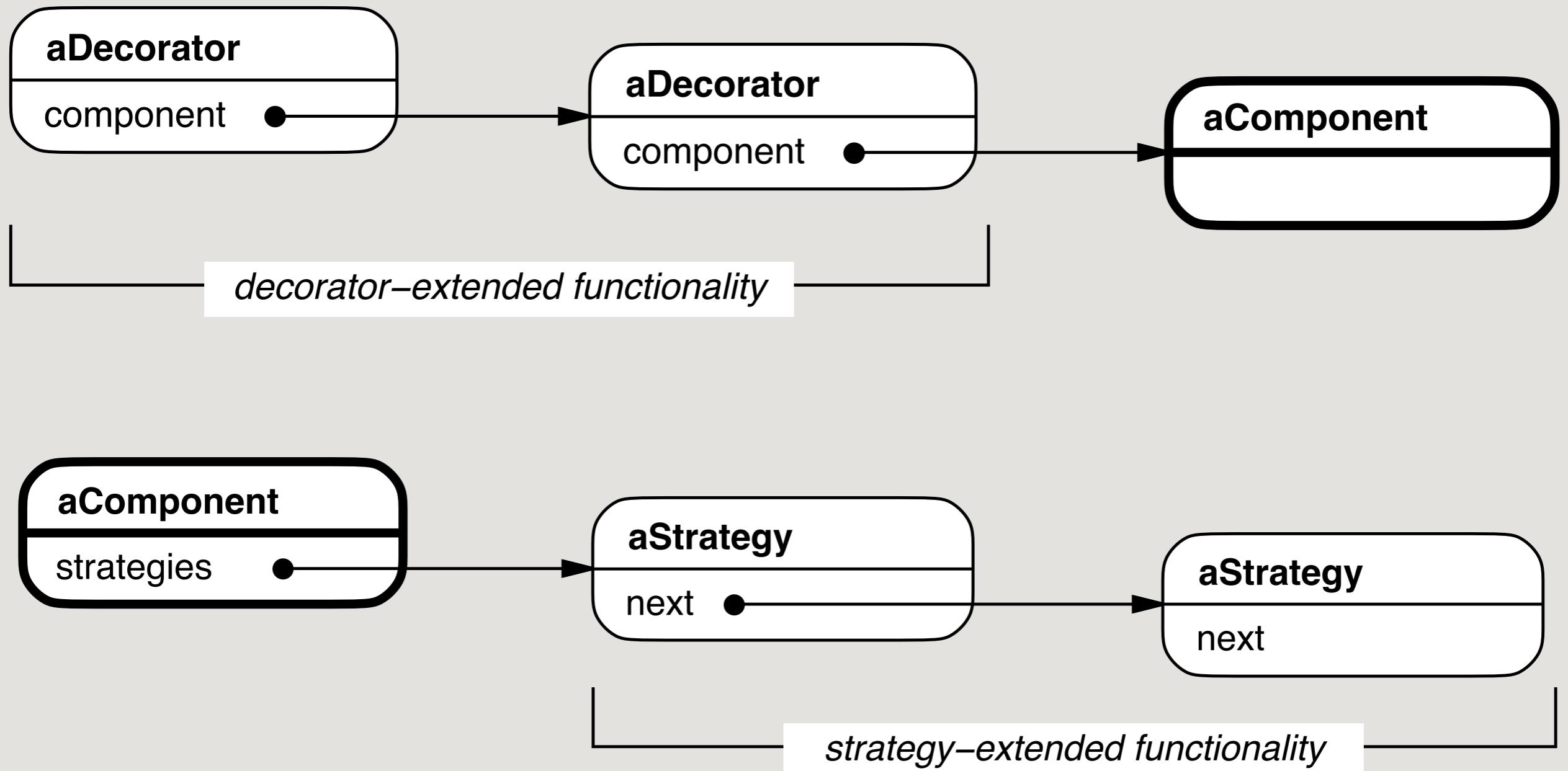
Una alternativa sería cambiar las tripas del objeto, como hace la estrategia.

En el patrón Strategy, el componente delega parte de su comportamiento en un objeto estrategia aparte. Nos permite modificar la funcionalidad del componente cambiando una estrategia por otra.

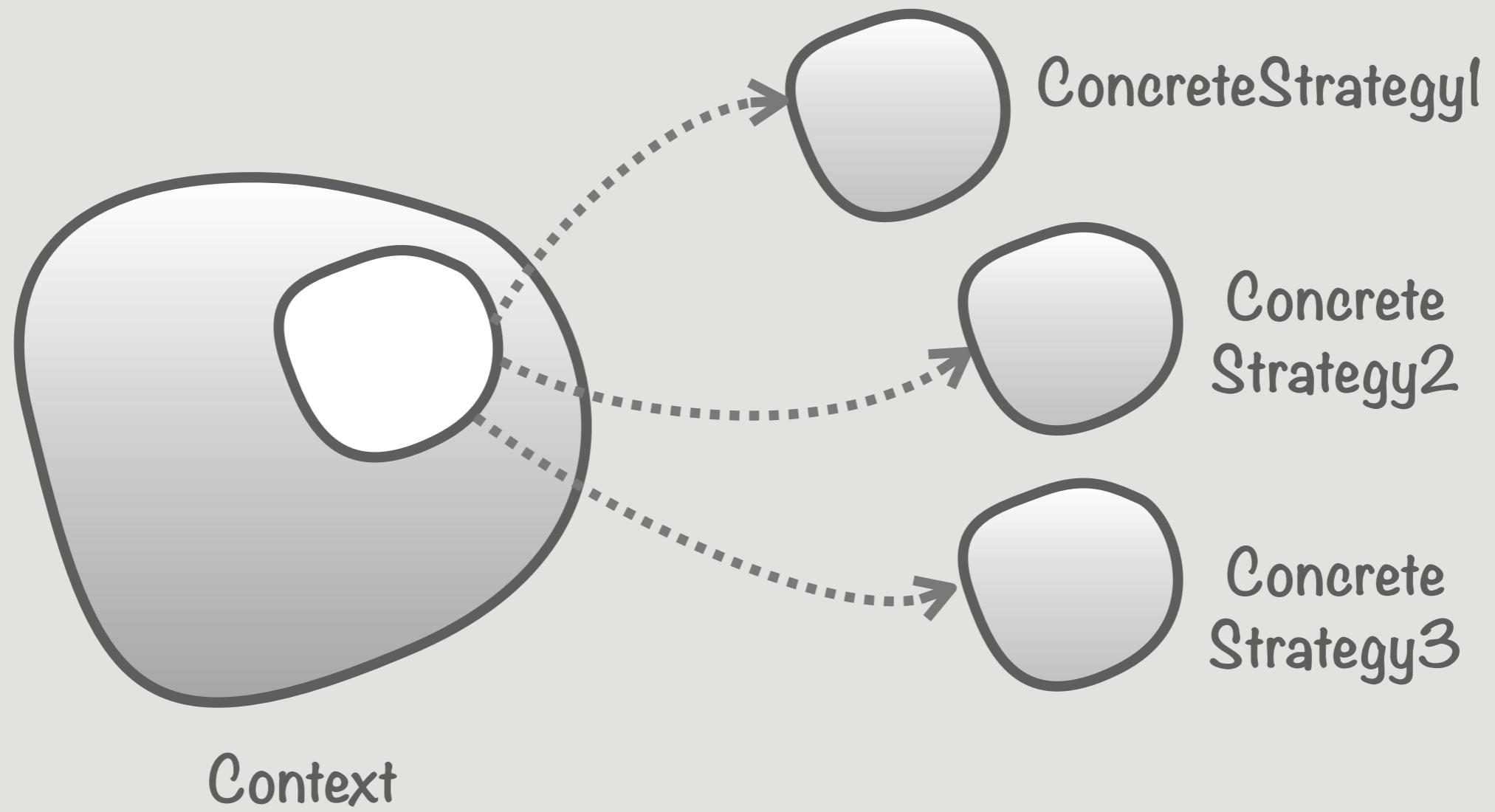
Dado que el patrón Decorator solo modifica el componente desde fuera, este no sabe nada de sus decoradores.

Son transparentes para el componente.

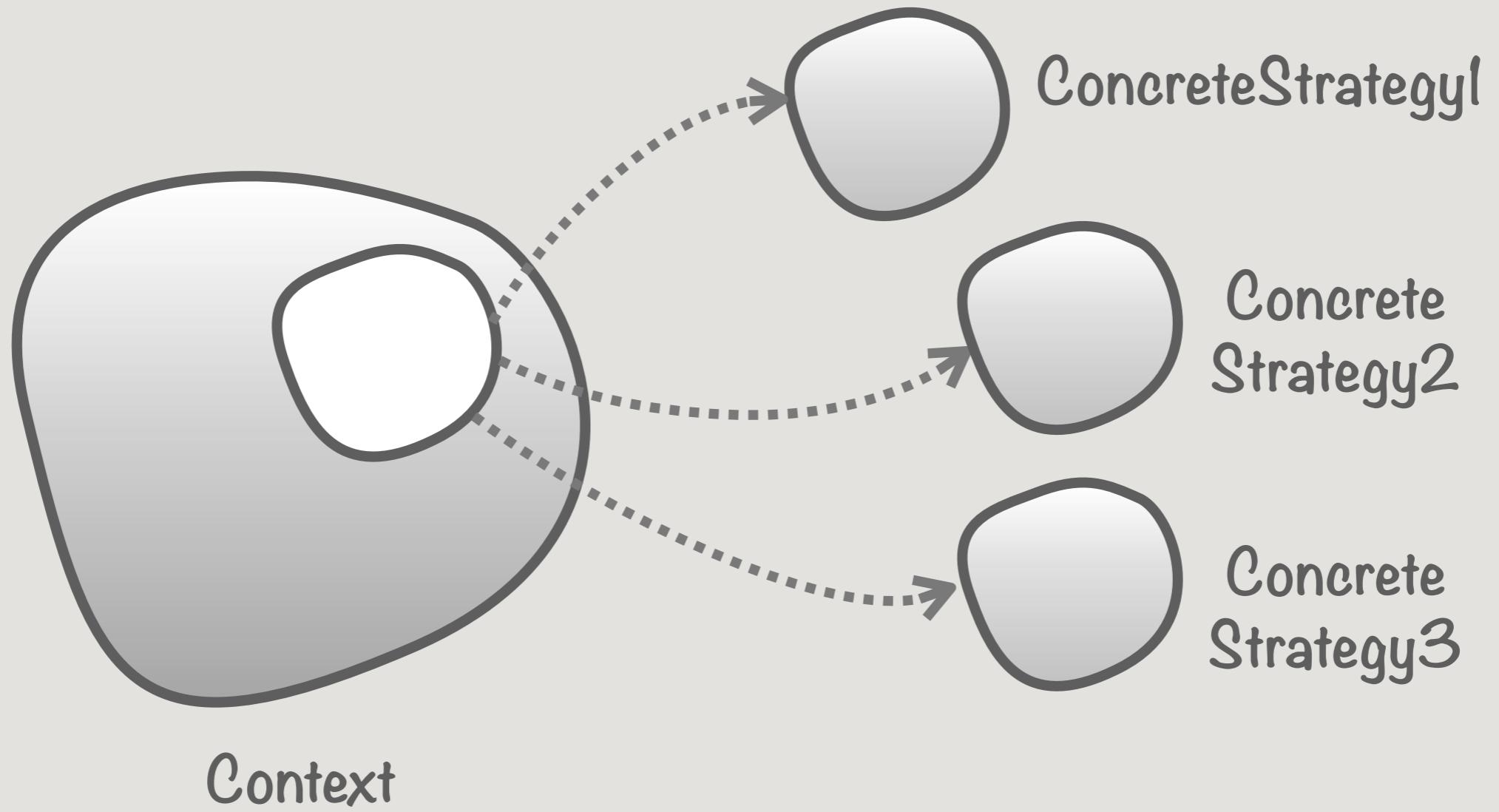
Por el contrario, el contexto conoce a sus estrategias, tiene una referencia a ellas.



¡Ojo!, porque, aunque conceptualmente podemos pensar en que tenemos una cadena como la anterior, y de hecho el libro hace un dibujo similar al comparar ambos patrones, lo anterior no refleja la estructura de objetos real que tendríamos en tiempo de ejecución.



Estructura en tiempo de ejecución correcta de una lista de estrategias

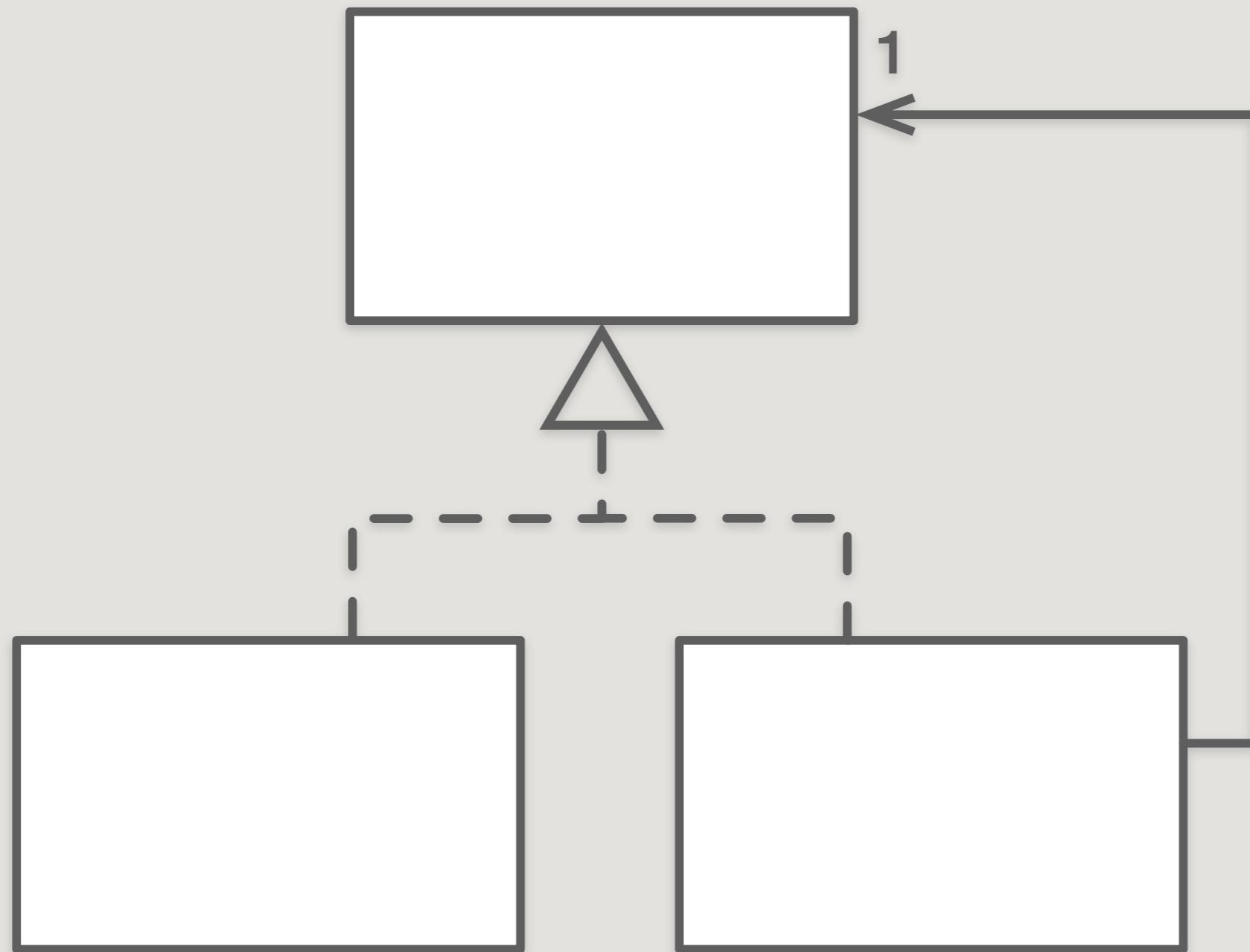


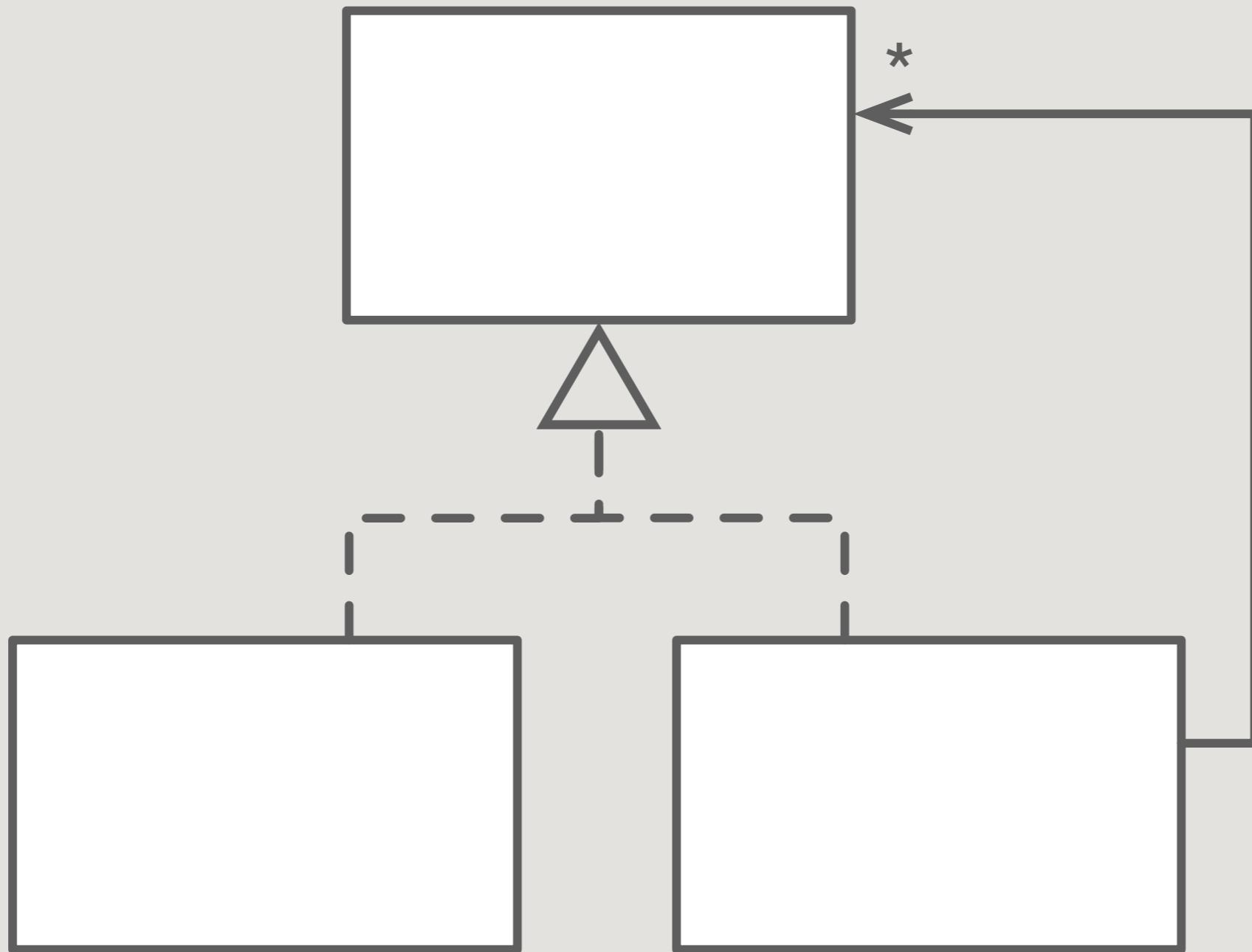
Es el contexto quien debe recorrer las distintas estrategias, llamando a la operación de cada una (posiblemente, pasándole el resultado de la anterior).

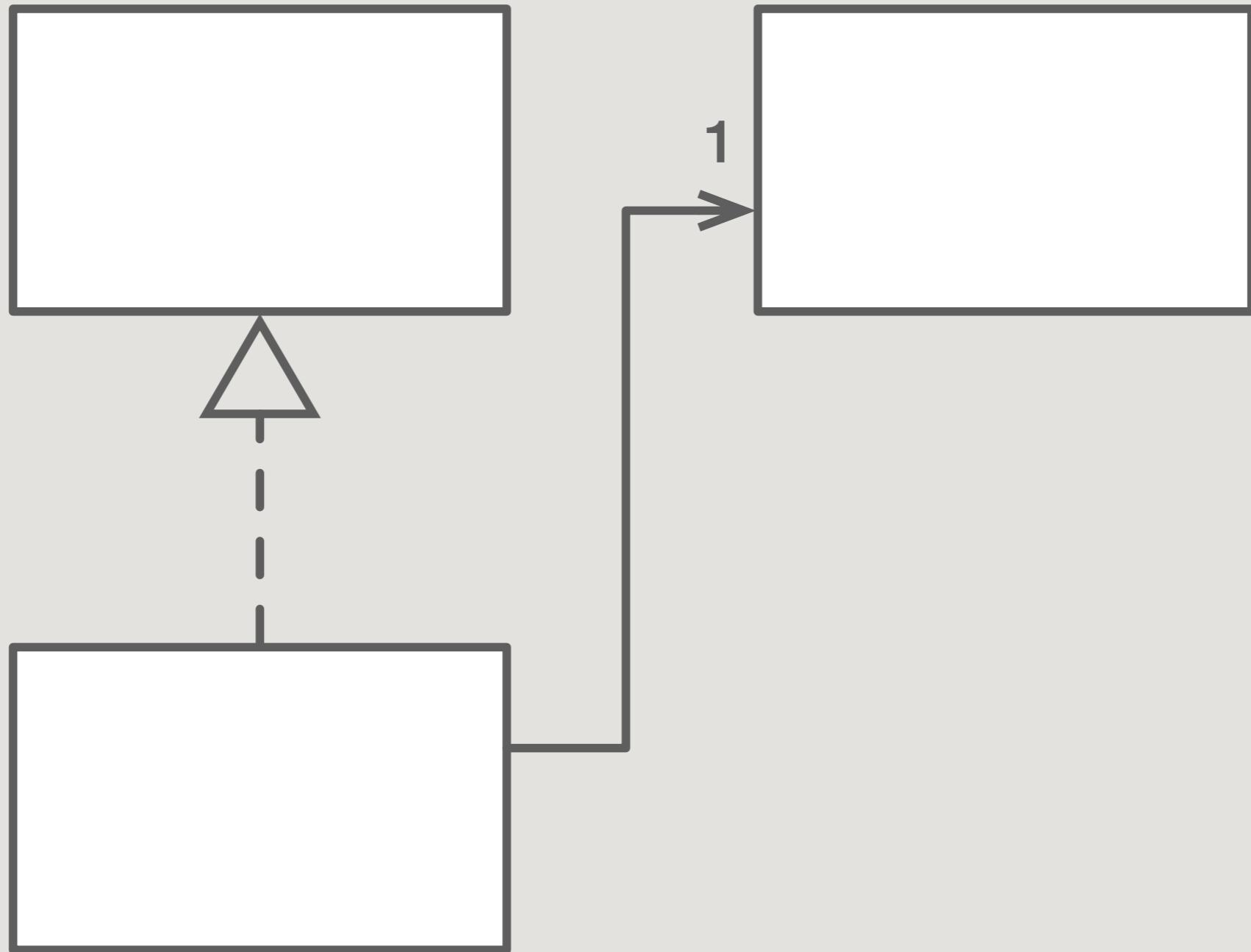
Y esta es, precisamente, su gran diferencia

Mientras que el decorador es consciente de que forma parte de una cadena y sabe quién le sigue, pudiendo añadir cualquier funcionalidad antes o después de llamar al siguiente (o «cortar» la cadena y directamente no llamarlo), las estrategias, por el contrario, no se conocen entre sí, es el contexto quien debe encadenar una tras otra.

Decorator, Adapter y Composite







Adapter

Un decorador se diferencia de un adaptador en que el decorador solo cambia las responsabilidades de un objeto, no su interfaz.

El adaptador proporciona una nueva interfaz a un objeto.

Composite

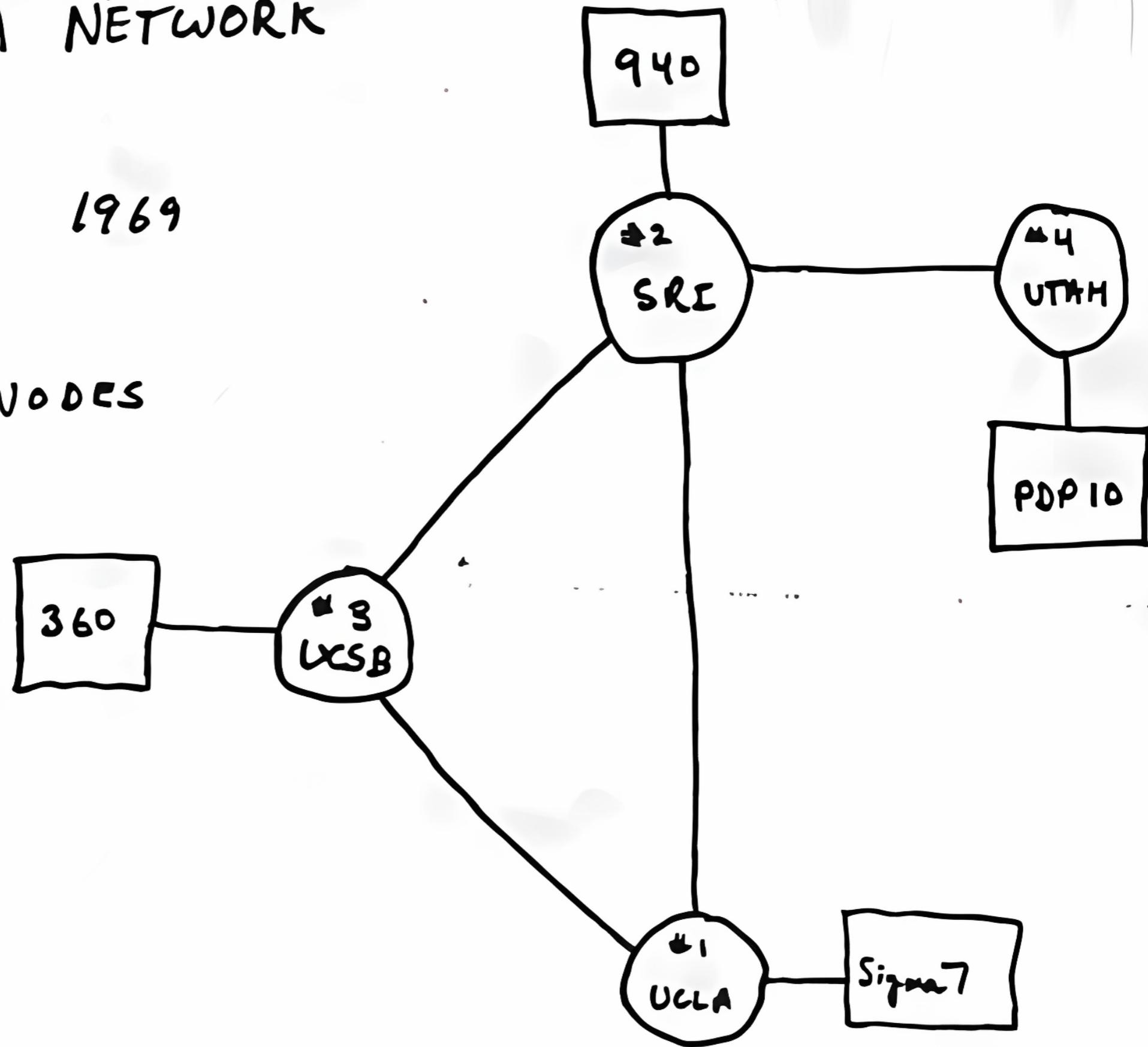
Aunque podríamos ver a un decorador como un Composite degenerado de un solo componente, el decorador **añade nuevas responsabilidades** (no está pensado para la mera composición de objetos).

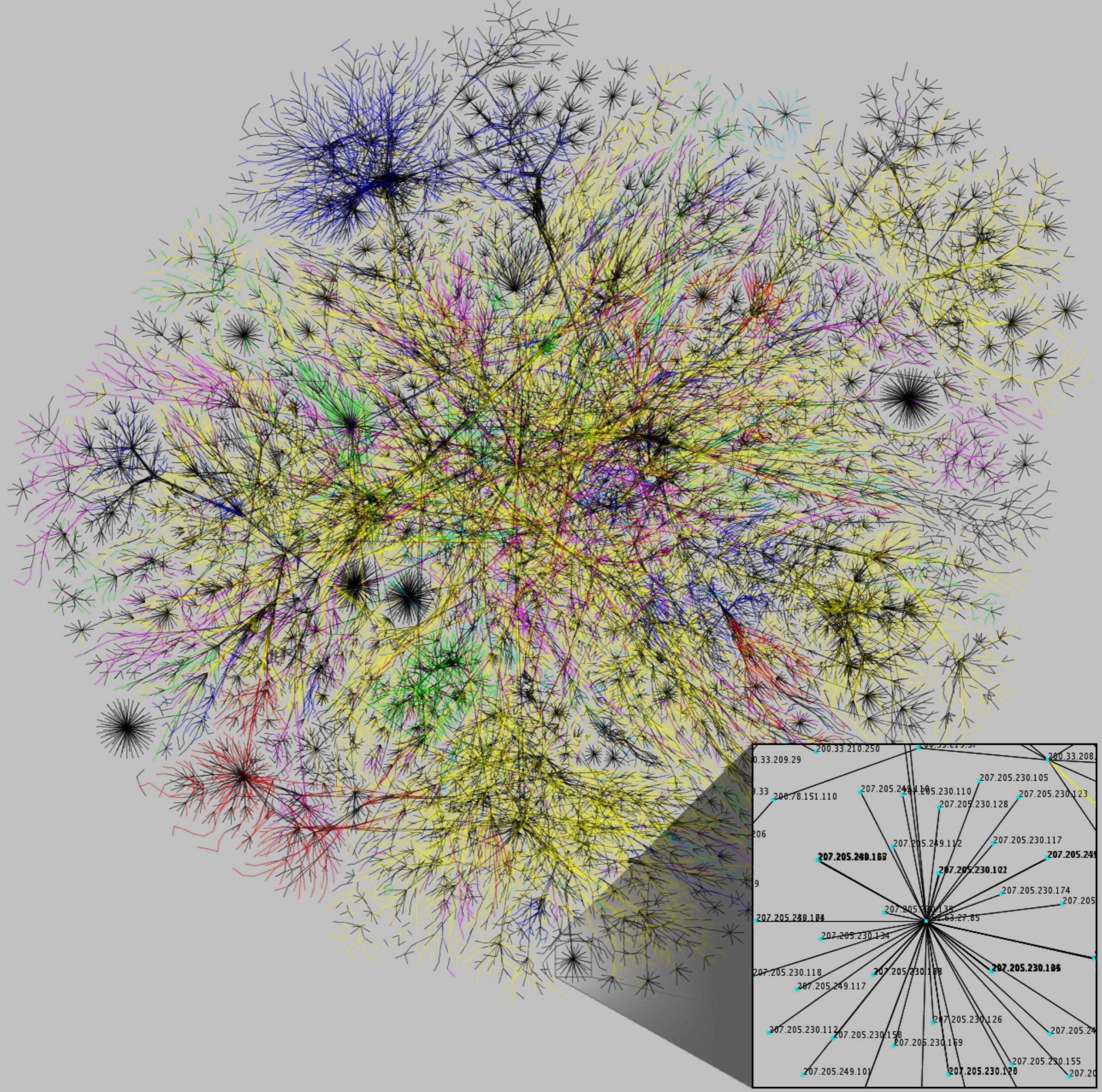
Epílogo

THE ARPA NETWORK

DEC 1969

4 NODES





Some Internet Architectural Guidelines and Philosophy

Status of this Memo

This memo provides information for the Internet community. It does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (2002). All Rights Reserved.

Abstract

This document extends [RFC 1958](#) by outlining some of the philosophical guidelines to which architects and designers of Internet backbone networks should adhere. We describe the Simplicity Principle, which states that complexity is the primary mechanism that impedes efficient scaling, and discuss its implications on the architecture, design and engineering issues found in large scale Internet backbones.

Table of Contents

1. Introduction	2
2. Large Systems and The Simplicity Principle	3
2.1. The End-to-End Argument and Simplicity	3
2.2. Non-linearity and Network Complexity	3
2.2.1. The Amplification Principle	4
2.2.2. The Coupling Principle	5
2.3. Complexity lesson from voice	6
2.4. Upgrade cost of complexity	7

Document type

[RFC - Informational](#)
December 2002

[View errata](#) [Report errata](#)

Updates [RFC 1958](#)

Was [draft-ymbk-arch-guideline-tsv](#) area)

Select version

05 [RFC 3439](#)

Compare versions

[draft-ymbk-arch-guidelines-05](#)

[RFC 3439](#)

[Side-by-side](#) [Inline](#)

Authors

[David Meyer](#) [Randy Bush](#)

[Email authors](#)

RFC stream



I E T F

Other formats

[txt](#) [html](#) [pdf](#) [w/](#)

[Report a datatracker bug](#)

Al mantener una interfaz mínima en el componente, el patrón Decorator promueve el principio de segregación de interfaces.

El patrón Decorator pattern encarna de manera elegante la esencia del principio abierto-cerrado.