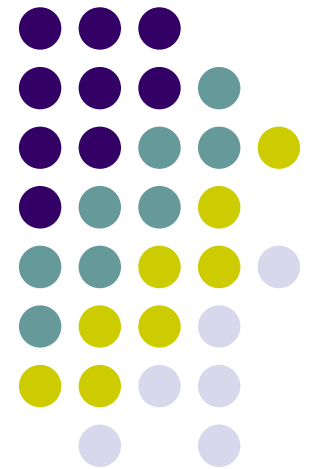


Patrones de Diseño

Patrón estructural *Decorator*



Decorator

Propósito



- Permite añadir responsabilidades extra a objetos concretos de manera dinámica
- Proporciona una alternativa flexible a la herencia para extender funcionalidad
- También conocido como *wrapper*

Decorator

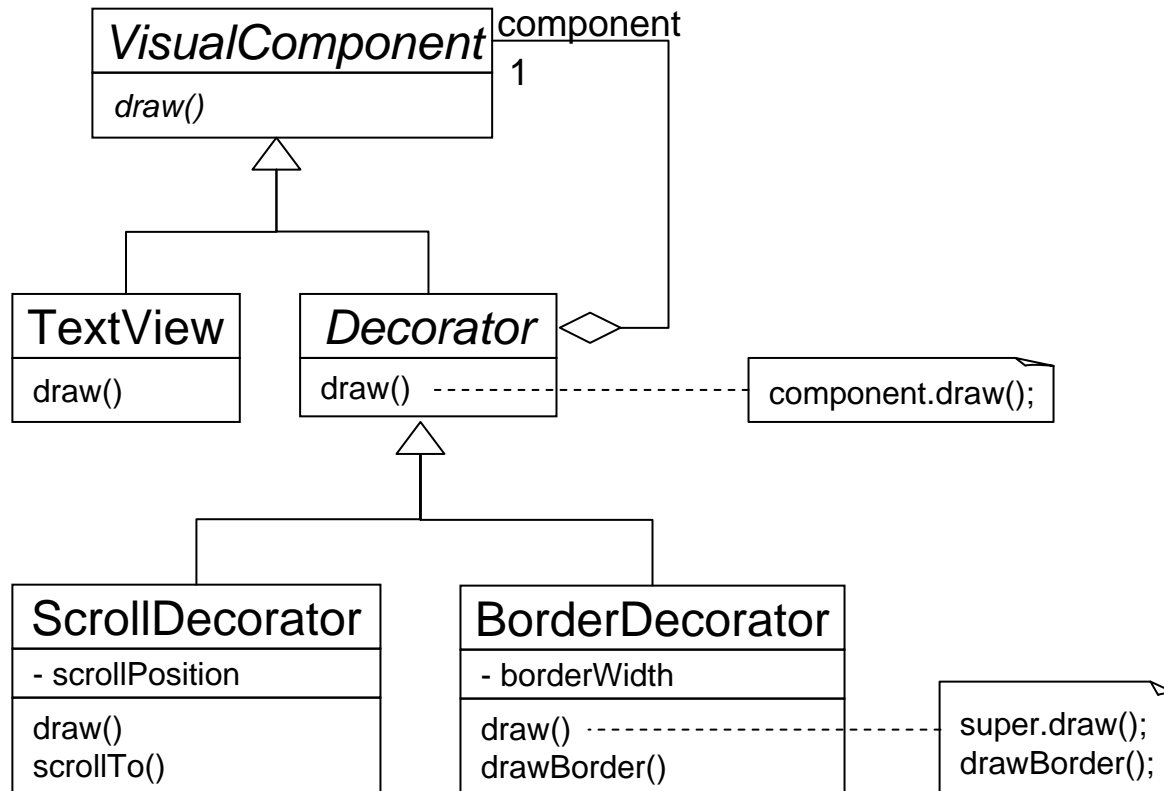
Motivación



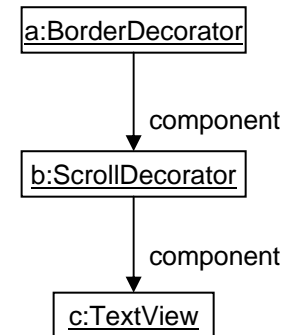
- A veces se quiere añadir funcionalidad a un objeto concreto, no a una clase entera
- Ej: Un toolkit para GUIs proporciona soporte para añadir marcos, barras de desplazamiento... a componentes
- Solución:
 - Herencia: no es flexible, la funcionalidad se añade estáticamente
 - Definir una clase “decoradora” que envuelve al componente, y le proporciona la funcionalidad adicional requerida: más flexible, transparente al cliente, se pueden anidar decoradores

Decorator

Estructura



Ejemplo de componente textual con scroll y marco:



Decorator

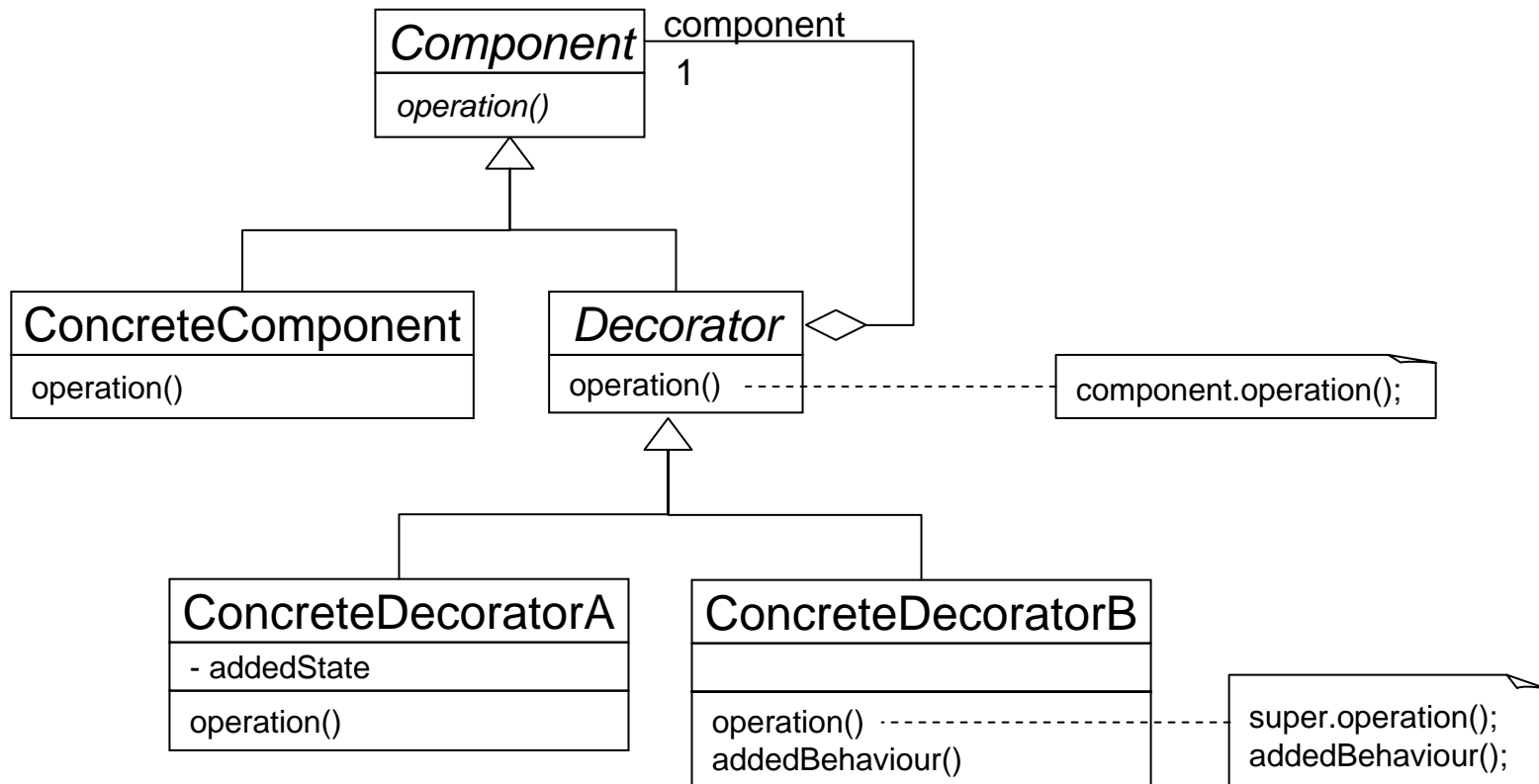
Aplicabilidad



- Usa el patrón *Decorator*.
 - Para añadir responsabilidades a objetos concretos de manera dinámica y transparente, esto es, sin afectar a otros objetos
 - Para responsabilidades que se pueden añadir y quitar
 - Cuando la herencia sea impracticable, porque implique crear múltiples subclases para todas las combinaciones posibles (ej. *TextViewScroll*, *TextViewScrollBorder*, ...)

Decorator

Estructura



Decorator

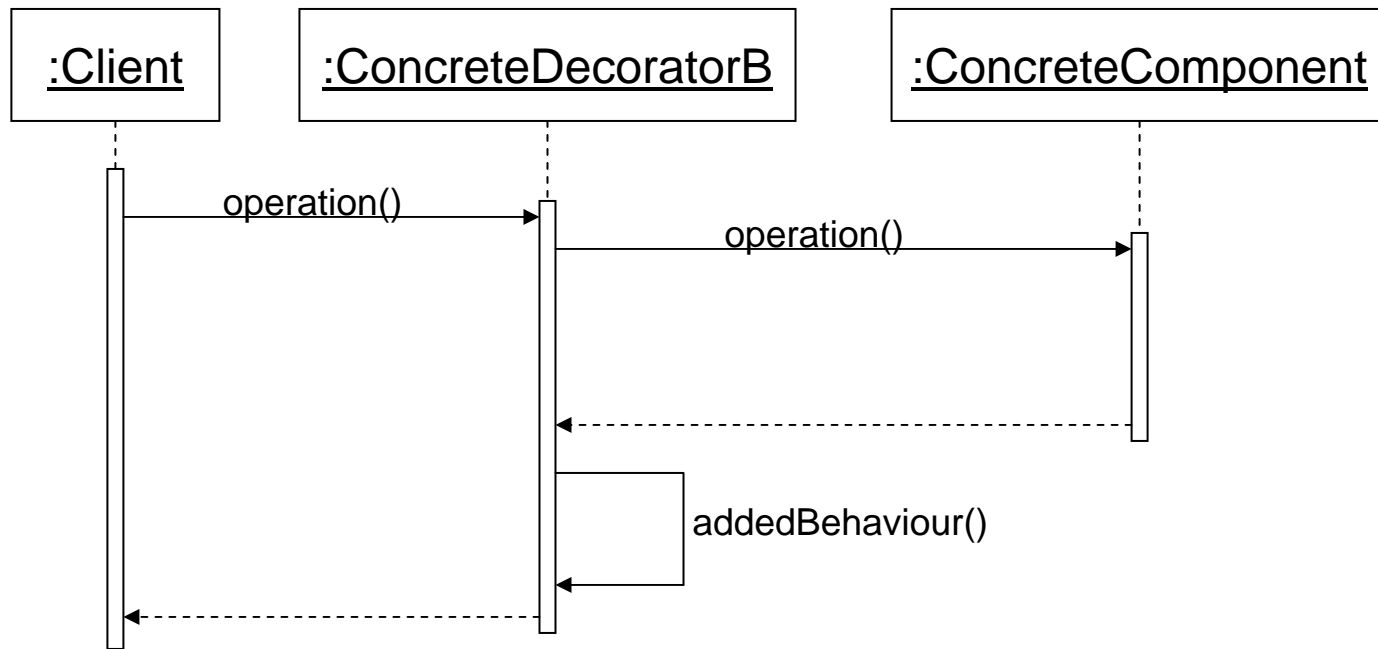
Participantes



- **Component** (*VisualComponent*): define la interfaz de los objetos a los que se puede añadir responsabilidades de manera dinámica
- **ConcreteComponent** (*TextView*): define un objeto al que añadir responsabilidades de manera dinámica
- **Decorator**: mantiene una referencia al objeto componente y define una interfaz conforme a la del componente
- **ConcreteDecorator** (*BorderDecorator, ScrollDecorator*): añade responsabilidades al componente al que referencia

Decorator

Colaboraciones



Decorator

Consecuencias



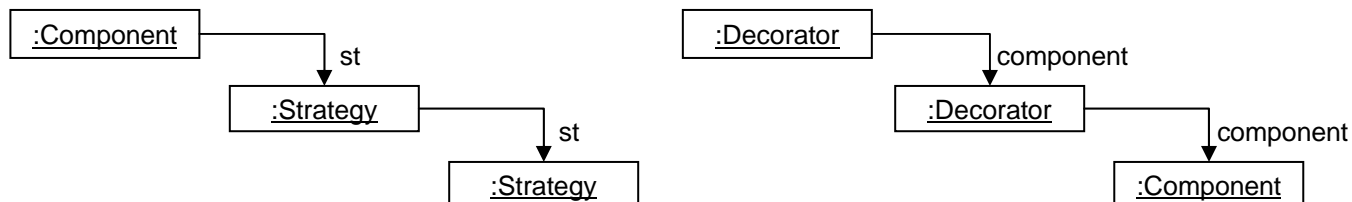
- Es más flexible que la herencia estática
 - Las responsabilidades se añaden y eliminan dinámicamente
 - Facilita definir una propiedad varias veces (ej. doble marco)
- Evita que las clases más altas en la jerarquía estén demasiado cargadas de funcionalidad y sean complejas
 - No hay precio que pagar por propiedades que no se usan
 - Facilita la definición de nuevos decoradores
- Un decorador y el componente al que se refiere no son idénticos (esto es, tienen distinto identificador)
- Provoca la creación de muchos objetos pequeños parecidos y encadenados, complicando la depuración

Decorator

Implementación



- Un componente y su decorador deben compartir la misma interfaz
- Se puede omitir la clase abstracta *Decorator* si sólo se va a definir una responsabilidad
- Mantener una clase *Component* ligera (definición de la interfaz, no almacén de datos). En caso contrario se incrementa la probabilidad de que las subclases hereden características que no necesitan
- ¿Cuál es la diferencia entre *Decorator* y *Strategy*?
 - Strategy: acceso al componente, el componente cambia
 - Decorator: acceso al decorador, el componente no cambia



Decorator

Código de ejemplo



```
public interface VisualComponent { // component
    public void draw();
}
public class TextView { // concrete component
    public void draw () { ... }
}
public abstract class Decorator implements VisualComponent { // decorator
    protected VisualComponent _component;
    public Decorator (VisualComponent vc) { _component = vc; }
    public void draw () { _component.draw(); }
}
public class BorderDecorator extends Decorator { // concrete decorator
    public BorderDecorator (VisualComponent vc) { super(vc); }
    public void draw() {
        super.draw();
        drawBorder();
    }
    public void drawBorder() { ... }
}
public class Test { // client
    public static void main (String args[]) {
        VisualComponent vc = new BorderDecorator(new TextView());
        vc.draw();
    }
}
```