Decorator

Christofer Fabián Chávez Carazas

Universidad Nacional de San Agustín

20 de octubre de 2016

Motivación

- A veces es necesario añadir responsabilidades a objetos individuales, no a una clase entera.
- Por ejemplo, cuando un conjunto de herramientas de interfaz gráfica de usuario debe permitir agregar propiedades a cualquier componente.
- Una forma de añadir responsabilidades es con la herencia, pero es poco flexible.
- La solución está en encapsular dentro de otro objeto, llamado Decorador, las nuevas responsabilidades.

Decorator

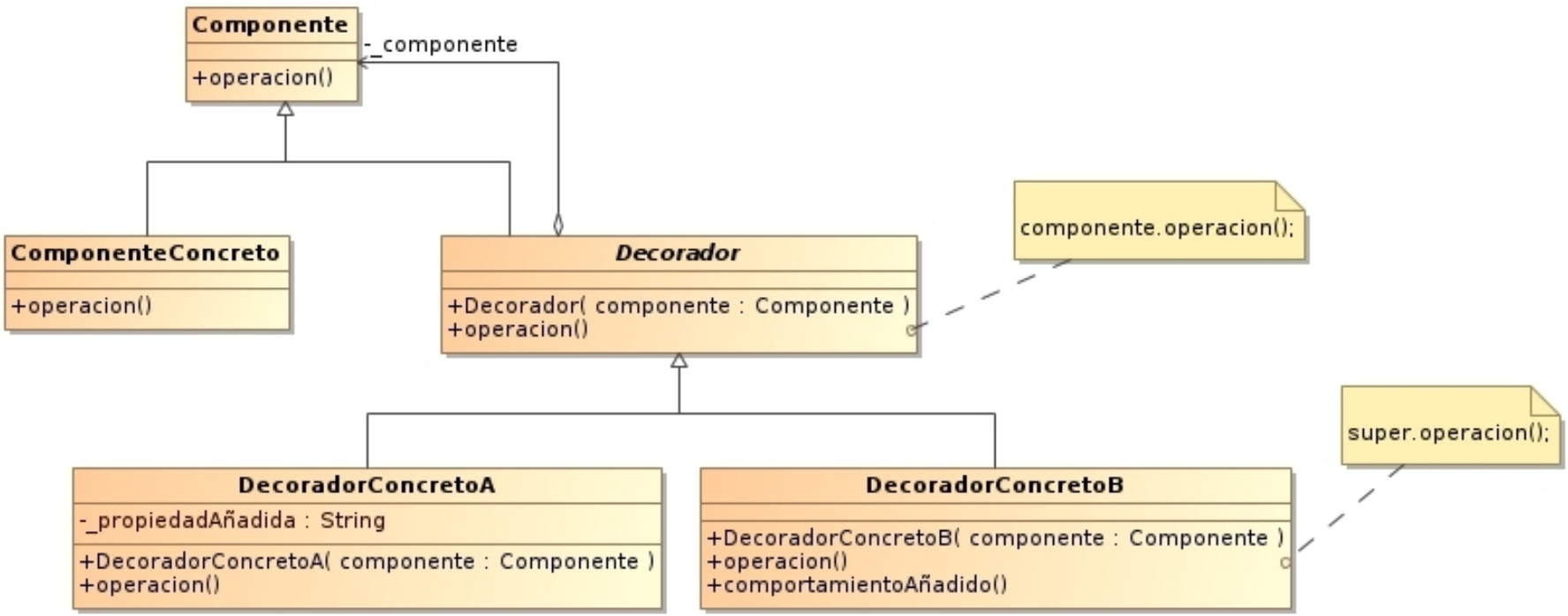
- El decorador redirige las peticiones al componente asociado.
- Opcionalmente puede realizar tareas adicionales antes y después de redirigir la petición.
- Es transparente para el cliente.

Aplicabilidad

Este patron se utiliza cuando:

- se necesita añadir responsabilidades a objetos individuales de forma dinámica y transparente.
- se necesita retirar responsabilidades a otros objetos.
- la extensión mediante la herencia no es viable.
- hay una necesidad de extender la funcionalidad de una clase, pero no hay razones para extenderlo a través de la herencia.
- existe la necesidad de extender dinámicamente la funcionalidad de un objeto y quizás quitar la funcionalidad extendida.

Participantes



Ventajas

- Mayor flexibilidad que la herencia estática.
- Evita las clases cargadas de características, ya que permite la adición de responsabilidades.
- Son fáciles de personalizar por aquellos que entienden su estructura.

Desventajas

- Un decorador y su componente no son idénticos. Un decorador actúa como una caja transparente, pero desde el punto de vista de la identidad del objeto un componente decorado no es idéntico a la del propio componente.
- Un diseño que utiliza Decorator a menudo resulta en sistemas compuestos por gran cantidad de pequeños objetos parecidos. Los objetos sólo se diferencian en la manera en que están interconectados. Esto puede resultar difícil de aprender y depurar.

Implementación

- La interfaz de un objeto decorador debe ajustarse a la interfaz del componente que decora (por herencia).
- Omitiendo la clase Decorador abstracta. No hay necesidad de definir una clase Decorador abstracta cuando sólo tiene que añadir una responsabilidad.

Implementación

- Componentes livianos. Es importante mantener la clase Component liviana para evitar que los decoradores resulten demasiado cargados, es decir, debe centrarse en la definición de una interfaz, no en el almacenamiento de datos (quienes deberían ser tratados en subclases).
- Decoratos vs Strategy: son dos formas alternativas de cambiar un objeto. El patrón Decorador sólo cambia un componente desde el exterior, de modo que los decoradores son transparentes para el componente. Con Strategy, el componente conoce las posibles extensiones, teniendo que hacer referencia y mantener las correspondientes estrategias.

Código

Visual Component:

```
class VisualComponent{
public:
  VisualComponent();
  virtual void Draw();
  virtual void Resize();
 //...
};
```

Decorator:

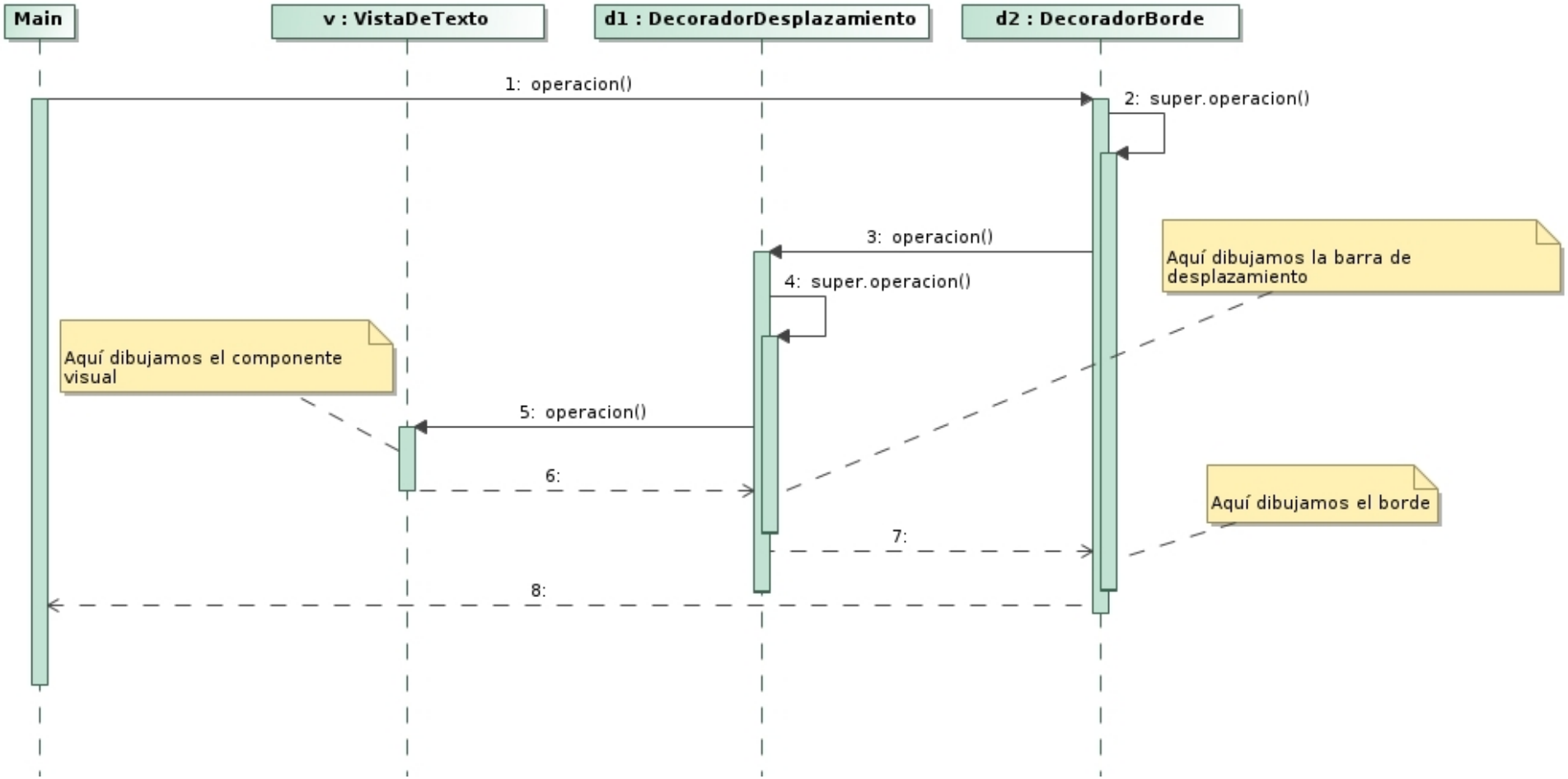
```
class Decorator:public VisualComponent{
public:
   Decorator(VisualComponent*);
   virtual void Draw();
   virtual void Resize();
   //...
private:
   VisualComponent* component;
};
```

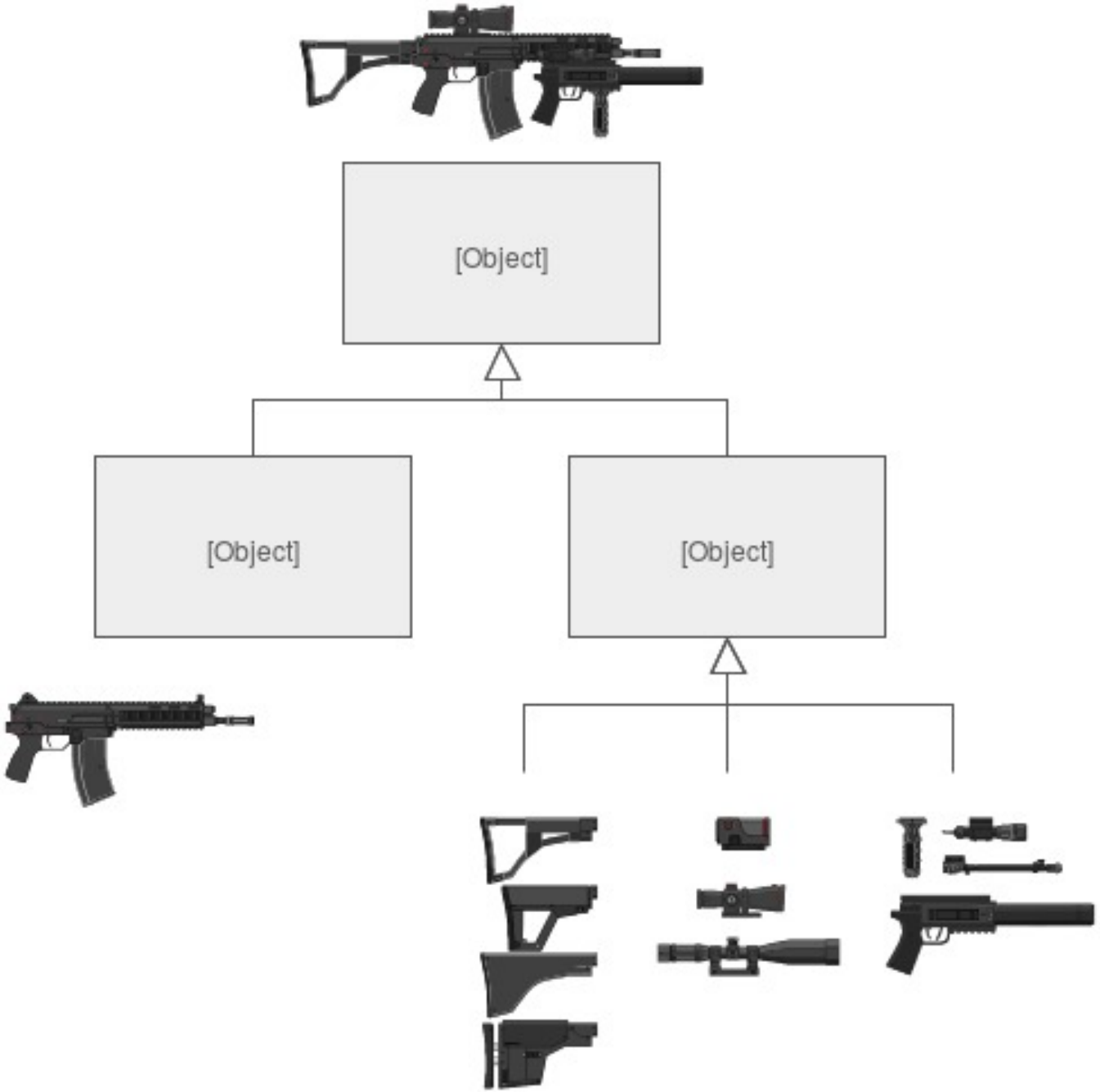
Código

```
void Decorator::Draw(){
 component->Draw();
}

void Decorator::Rezise(){
 component->Rezise();
}
```

Border Decorator

```
class BorderDecorator:public Decorator{
public:
 BorderDecorator(VisualComponent*, int borderWidth);
 virtual void Draw();
private:
 int _width;
 void DrawBorder(int);
};

void BorderDecorator::Draw(){
 Decorator::Draw();
 DrawBorder(_width);
}
```


Main
v : VistaDeTexto
d1 : DecoradorDesplazamiento
d2 : DecoradorBorde
1: operacion()
2: super.operacion()
3: operacion()
4: super.operacion()
Aquí dibujamos la barra de desplazamiento
Aquí dibujamos el componente visual
5: operacion()
6:
Aquí dibujamos el borde
7:
8:


[Object]
[Object]
[Object]

Patrones Relacionados

- **Adapter:** un Decorator sólo cambia las responsabilidades de un objeto, mientras que un adaptador dará a un objeto una interfaz nueva.
- **Composite:** un Decorator puede ser visto como un composite de un solo componente. Sin embargo, un decorador añade responsabilidades adicionales.
- **Strategy:** un Decorator permite cambiar la "piel"de un objeto, mientras que un Strategy le permite cambiar las "tripas"(por dentro).

Conclusiones

- El Decorator es un patron de diseño fácil de aprender.
- Ayuda mucho en crear dinamismo.
- Puede hasta llegar a eliminar la herencia multiple.

Bibliografía

 Ralph Johnson, Erich Gamma, John Vlissides, Richard Helm

Design Patterns: Elements of Reusable Object-Oriented Software

Addison-Wesley, 1995