Arquitetura de Software

Padrões de Projetos – Padrões Estruturais

Nairon Neri Silva

Sumário

Padrões de Projeto Estruturais

1. *Adapter*
2. *Bridge*
3. *Composite*
4. ***Decorator***
5. *Façade*
6. *Flyweight*
7. *Proxy*

Decorator

Também conhecido como ***Wrapper***

Intenção

- Dinamicamente, agregar responsabilidades adicionais a um objeto

- Os ***Decorators*** fornecem uma alternativa flexível ao uso de subclasses para extensão de funcionalidades

Motivação

- Algumas vezes queremos acrescentar responsabilidades a objetos individuais, e não a toda uma classe

- Exemplo: um *toolkit* para construção de interfaces gráficas de usuário
  - deveria permitir a adição de propriedades, como bordas, ou comportamentos, como rolamento, para qualquer componente da interface dousuário

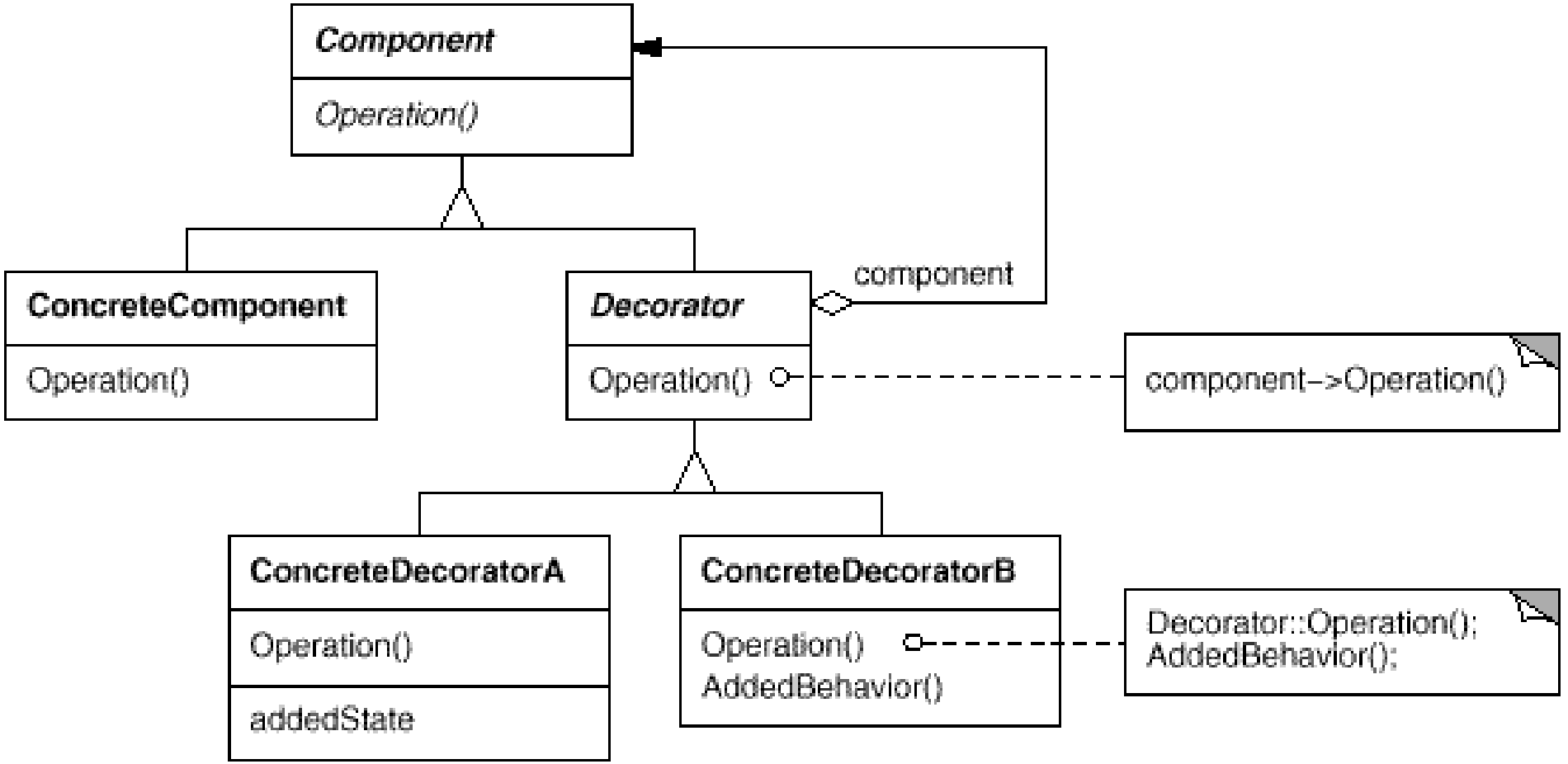
Motivação

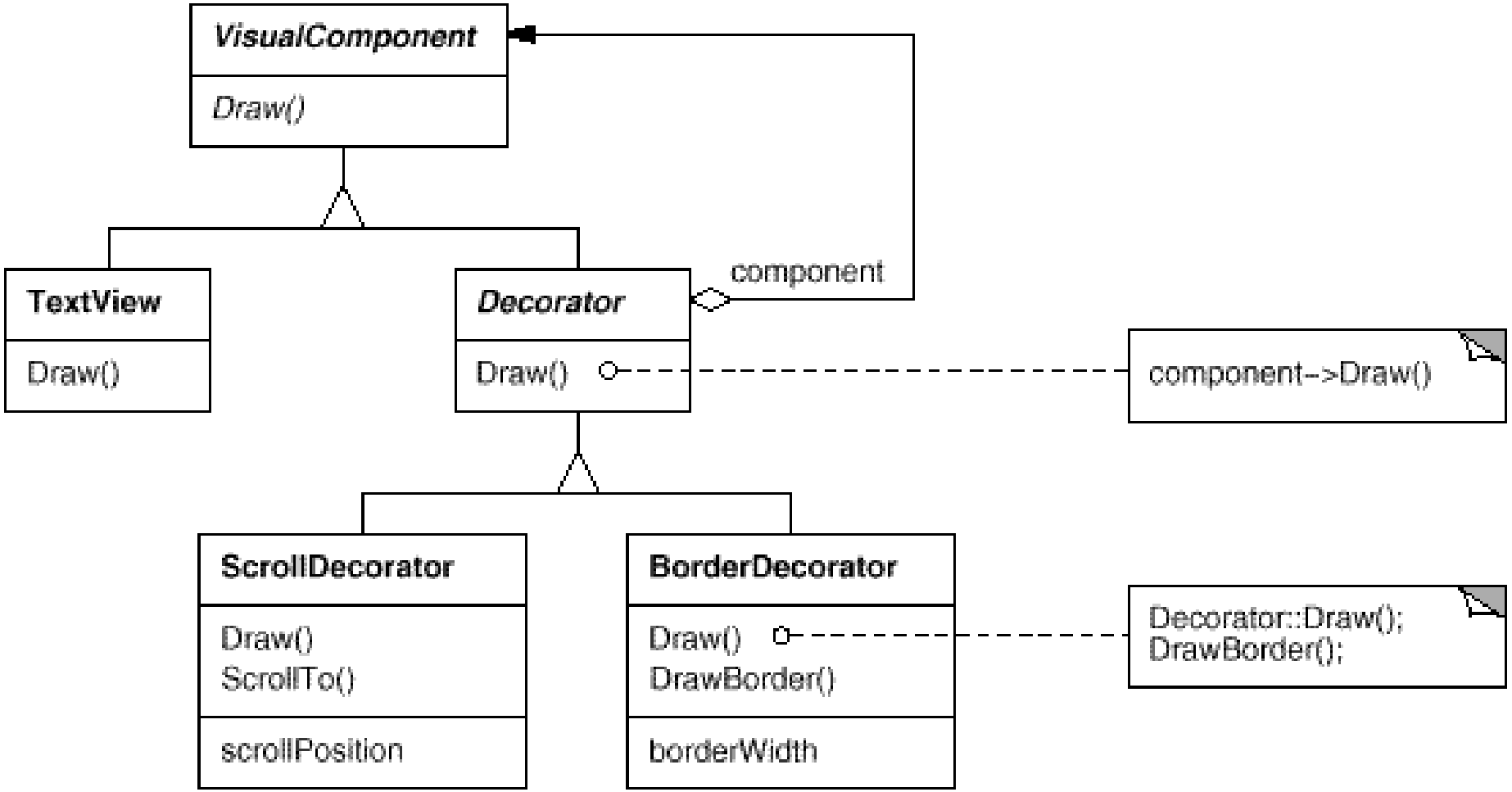
- Uma forma de adicionar responsabilidade é através da herança
  - Essa alternativa é inflexível – a escolha da borda é feita estaticamente – um cliente não pode decidir quando e como decorar o componente com uma borda

- Uma alternativa mais flexível seria embutir o componente em outro objeto que acrescenta a borda

Aplicabilidade

- Para acrescentar responsabilidade a objetos individuais de forma dinâmica e transparente
- Para responsabilidades que podem ser removidas
- Quando a extensão através do uso de subclasses não é prática – devido a possibilidade de explosão de subclasses para suportar cada combinação


Component
Operation()
ConcreteComponent
Operation()
Decorator
Operation()
component
component->Operation()
ConcreteDecoratorA
Operation()
addedState
ConcreteDecoratorB
Operation()
AddedBehavior()
Decorator::Operation();
AddedBehavior();


VisualComponent
Draw()
TextView
Draw()
Decorator
Draw()
component
component->Draw()
ScrollDecorator
Draw()
ScrollTo()
scrollPosition
BorderDecorator
Draw()
DrawBorder()
borderWidth
Decorator::Draw();
DrawBorder();

Exemplo/Participantes

1. ***Component (VisualComponent)***
   - Define a interface para objetos que podem ter responsabilidades acrescentadas dinamicamente
2. ***CnocreteComponent (TextView)***
   - Define um objeto que recebera novas responsabilidades
3. ***Decorator***
   - Mantem uma referência para um ***Component***
4. ***ConcreteDecorator (BorderDecorator, ScrollDecorator)***
   - Acrescenta responsabilidades ao componente

Consequências

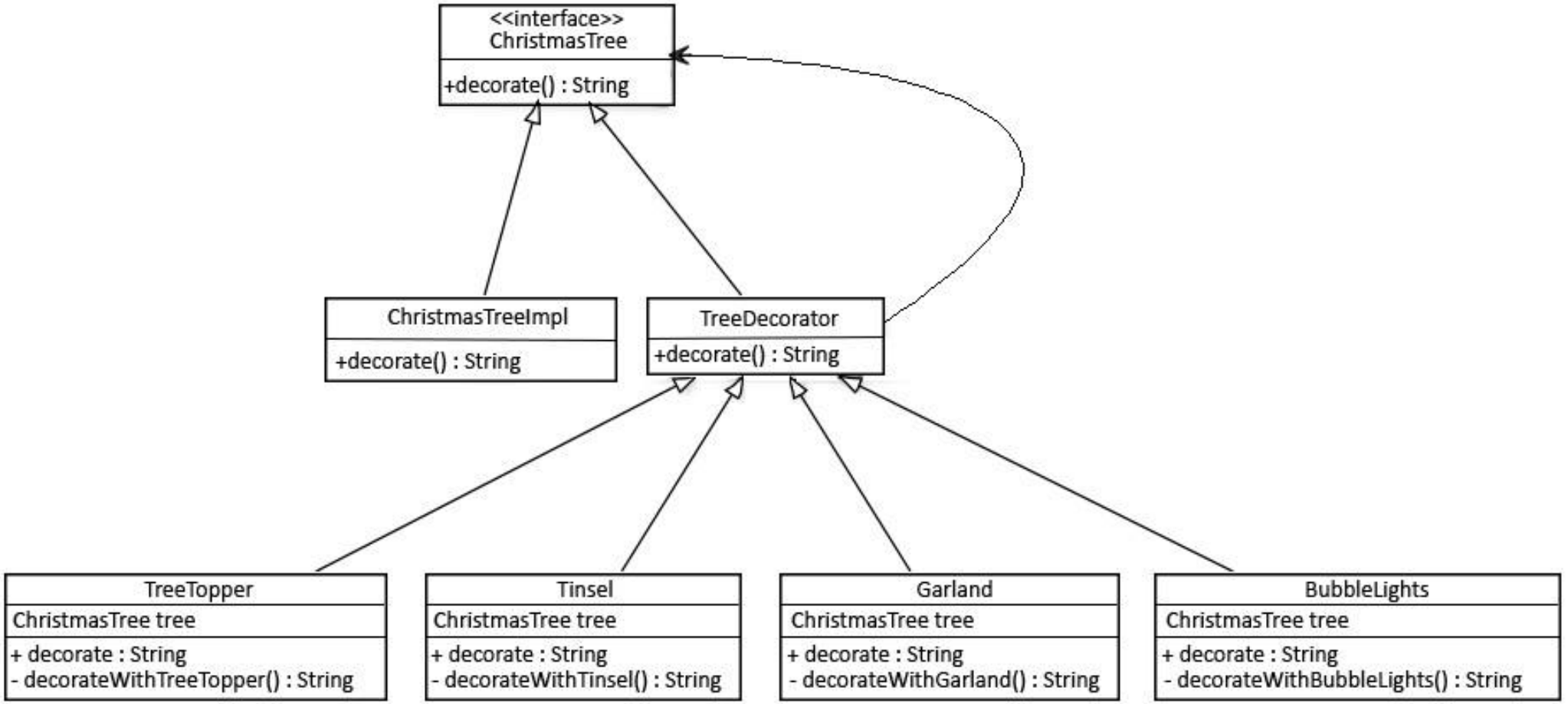
1. Mais flexibilidade do que a herança estática
2. Evita classes sobrecarregadas de características na parte superior da hierarquia
3. É possível decorar um objeto já decorado, adicionando ainda mais funcionalidades
4. Um *Decorator* e o seu componente não são idênticos
5. Grande quantidade de pequenas classes (quanto mais decorators em camadas, mais complexo ficará o código)

Exemplo prático:

- Suponha que temos um objeto de árvore de Natal e queremos decorá-lo.
- A decoração não altera o próprio objeto, ou seja, além da árvore de Natal, estamos adicionando alguns itens de decoração, como guirlanda, enfeites, enfeites de árvore, luzes de bolha, etc.
- Vamos ver como ficará o código desse exemplo aplicando o padrão de projeto Decorator.

Exemplo retirado do site: https://www.baeldung.com/java-decorator-pattern

Exemplo prático:



Exemplo retirado do site: https://www.baeldung.com/java-decorator-pattern

Saiba mais...

- https://www.youtube.com/watch?v=p3Dh7VjxudE
- https://refactoring.guru/pt-br/design-patterns/decorator
- https://www.baeldung.com/java-decorator-pattern

Acesse os endereços e veja mais detalhes sobre o padrão Decorator