Design Patterns

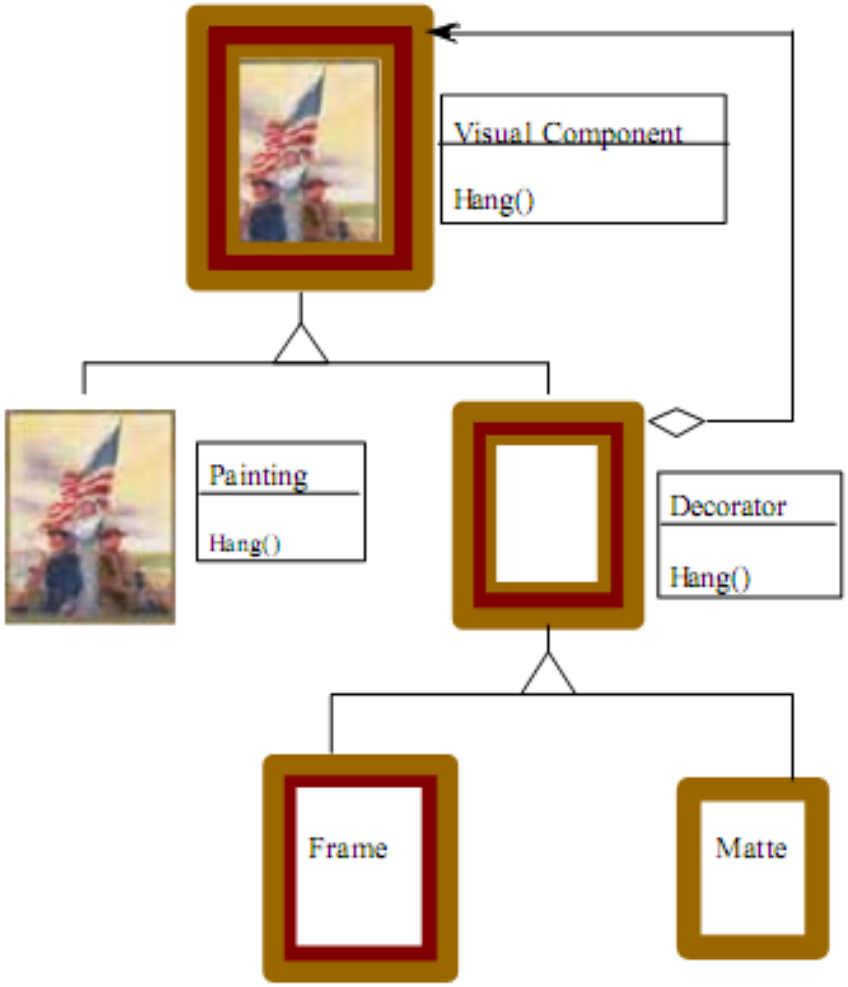
宋 杰
Song Jie
东北大学 软件学院
Software College, Northeastern University

7. Decorator Pattern

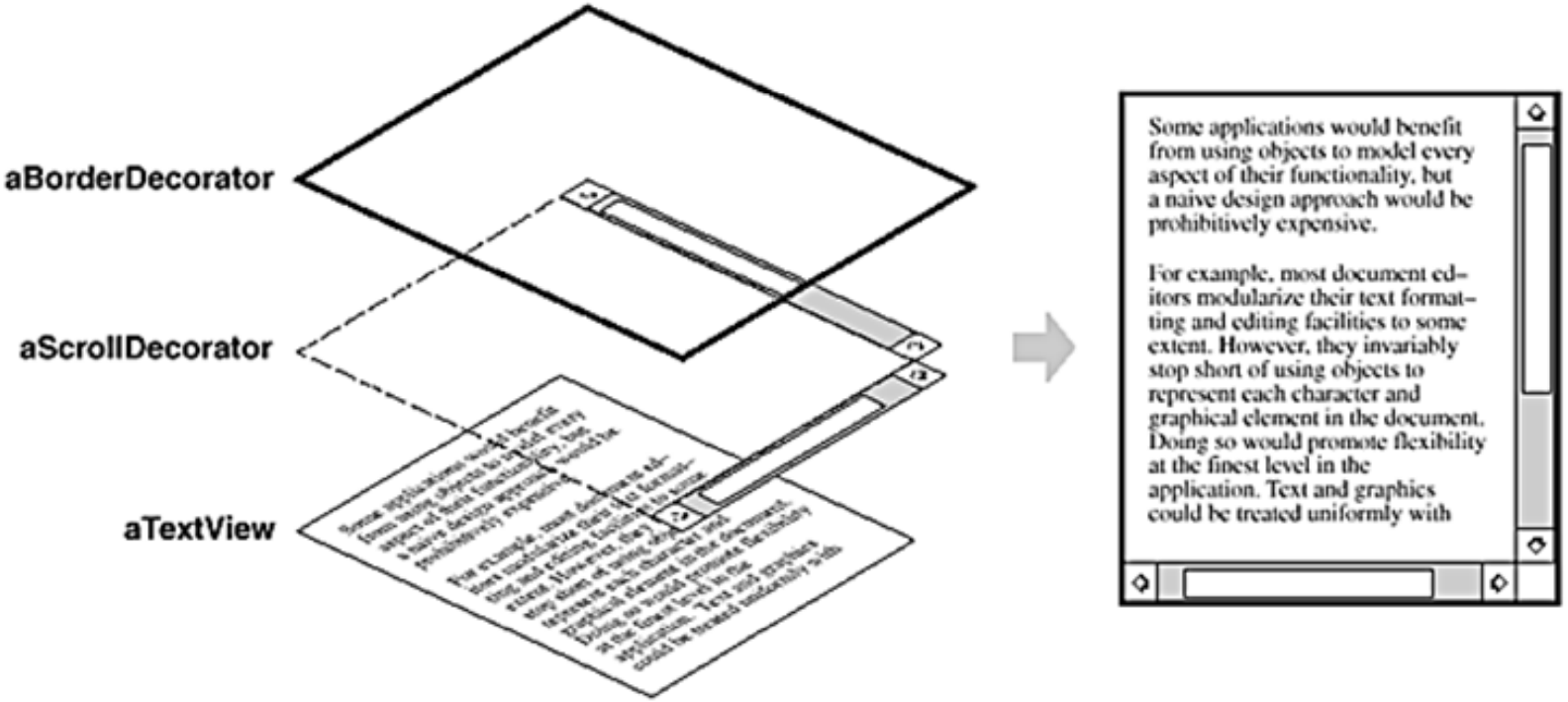
Intent

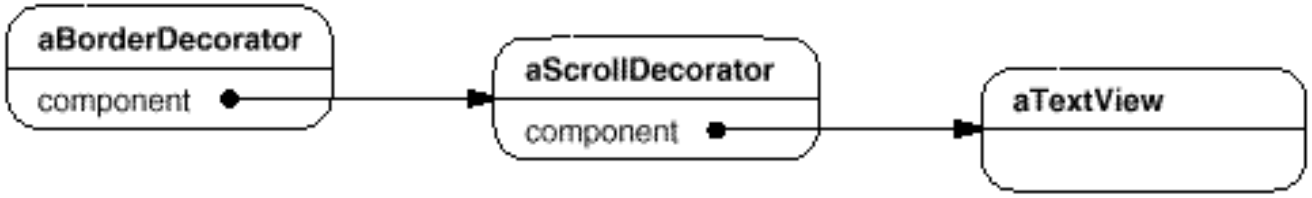
- Attach additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to subclassing for extending functionality.
  - Dynamically extension;
  - Better than inheritance;
- Wrapper;

Example

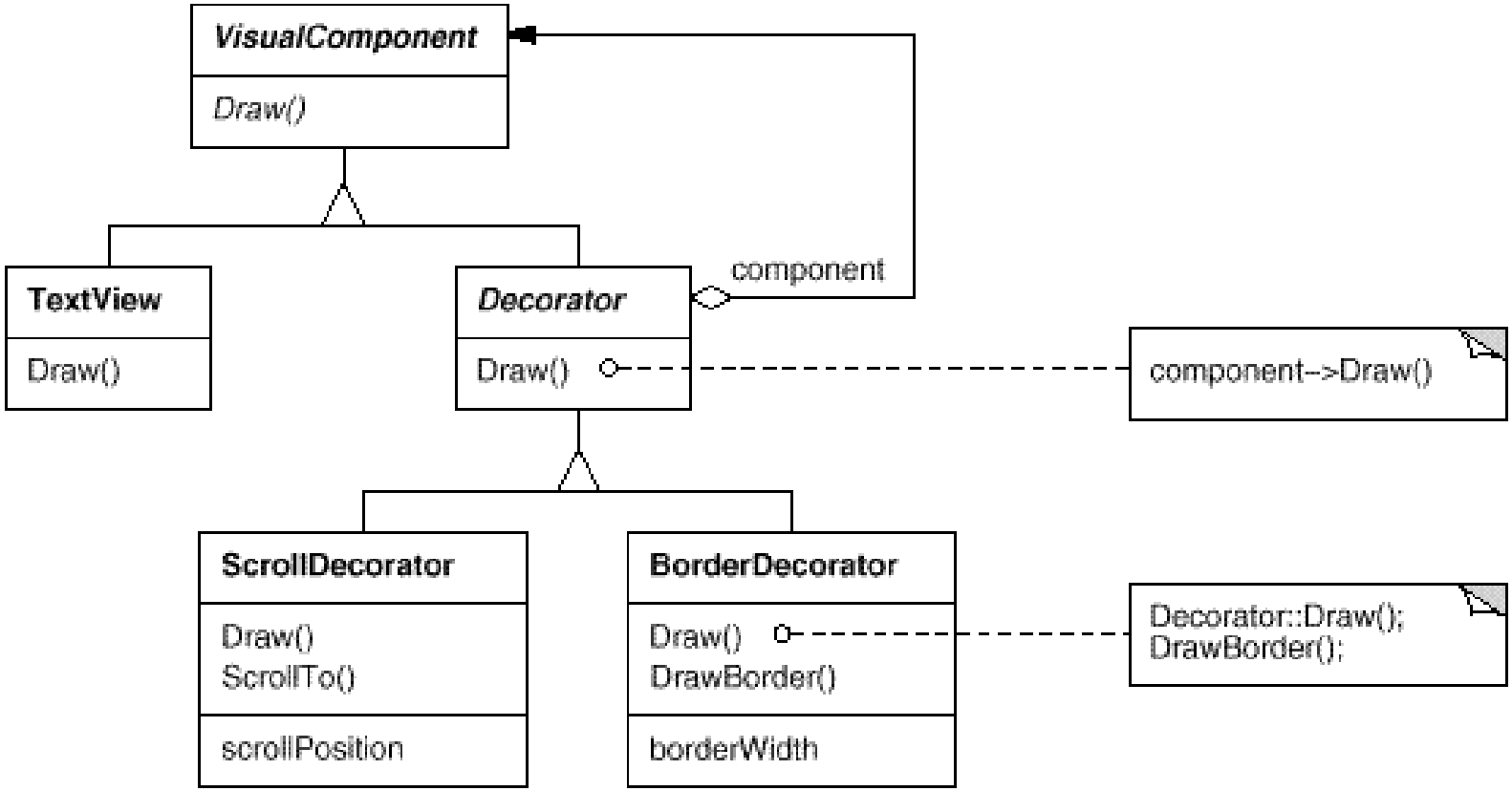

Visual Component
Hang()
Painting
Hang()
Decorator
Hang()
Frame
Matte

Example


aBorderDecorator
aScrollDecorator
aTextView
Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.
For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with


aBorderDecorator
component
aScrollDecorator
component
aTextView

Example


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

```java
interface VisualComponent {
    public void draw();
}

class TextView implements VisualComponent {
    public void draw() {
        // do something
    }
}

abstract class Decorator implements VisualComponent {
    protected VisualComponent component;
    public Decorator(VisualComponent component) {
        this.component = component;
    }
    public abstract void draw();
}
```

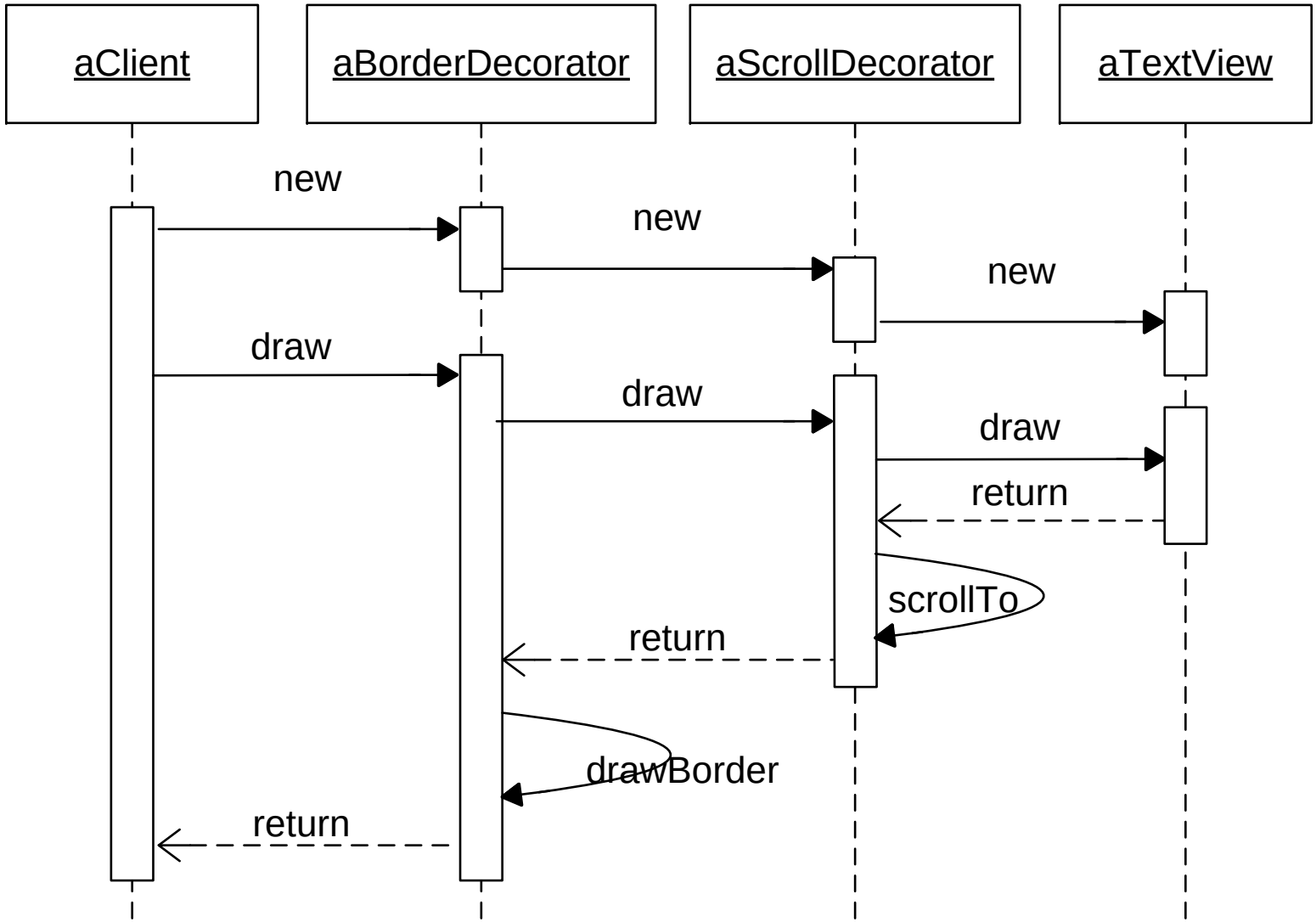
```java
class ScrollDecorator extends Decorator {
    public ScrollDecorator(VisualComponent component) {
        super(component);
    }
    public void draw() {
        component.draw();
        scrollTo();
    }
    public void scrollTo() {
        // do something
    }
}
```

```java
class BorderDecorator extends Decorator {
    public BorderDecorator(VisualComponent component) {
        super(component);
    }
    public void draw() {
        component.draw();
        drawBorder();
    }
    public void drawBorder() {
        // do something
    }
}
```

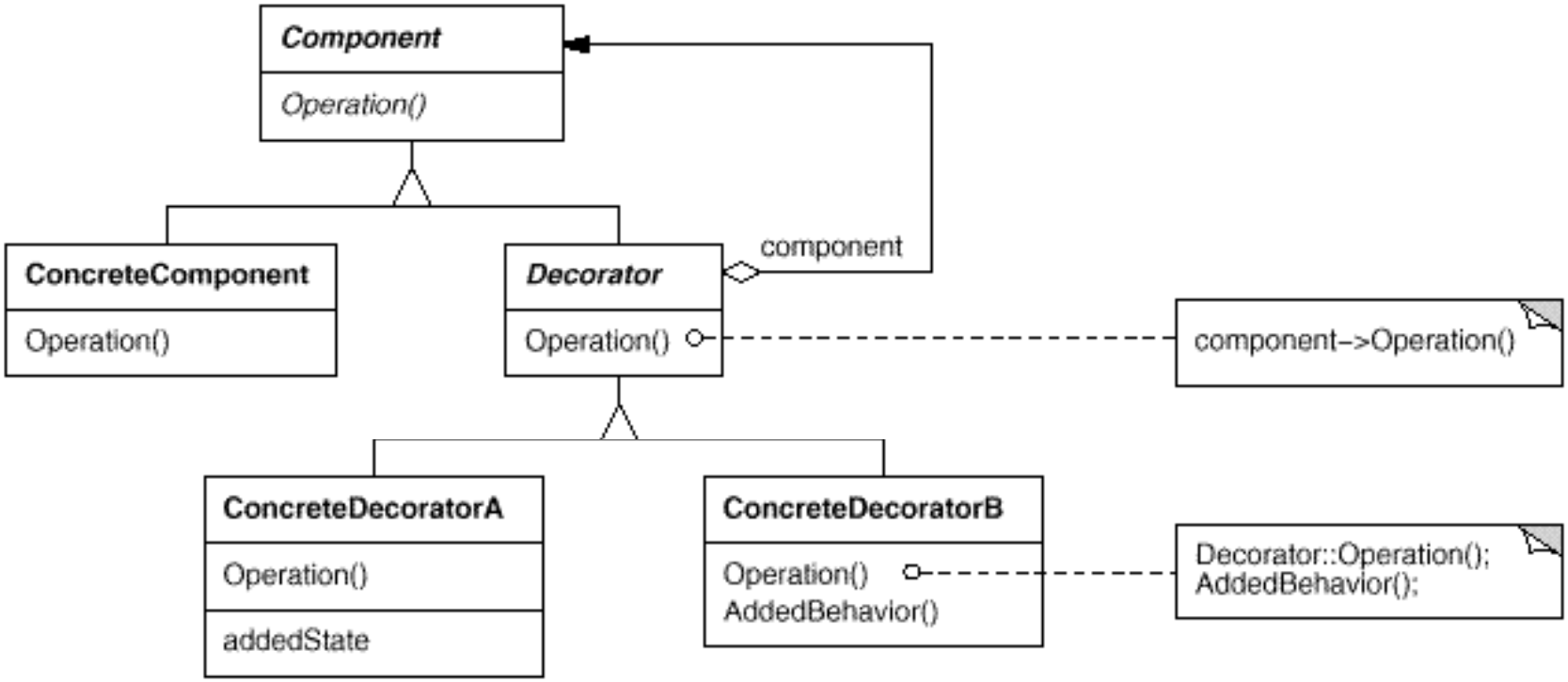
```java
class DecoratorClient {
    public void decoratorClient() {
        VisualComponent component =
            new BorderDecorator(
                    new ScrollDecorator(
                            new TextView()));

//      VisualComponent border = new BorderDecorator();
//      VisualComponent scroll = new ScrollDecorator();
//      VisualComponent text = new TextView();
//      border.setComponent(scroll);
//      scroll.setComponent(text);

        component.draw();
    }
}
```

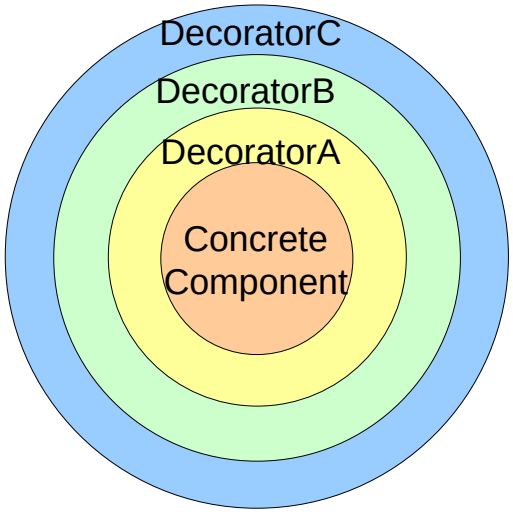

aClient
aBorderDecorator
aScrollDecorator
aTextView
new
new
new
draw
draw
draw
return
scrollTo
return
drawBorder
return

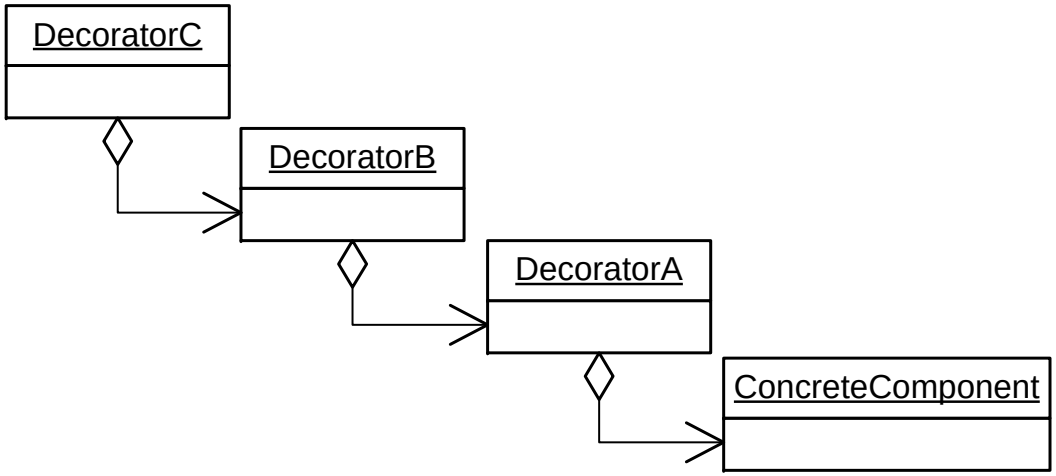
Structure


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

Participants

- Component: defines the interface for objects that can have responsibilities added to them dynamically.
- ConcreteComponent: defines an object to which additional responsibilities can be attached.
- Decorator: maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- ConcreteDecorator: adds responsibilities to the component.






```
Component component =
  new DecoratorC(
      new DecoratorB(
          new DecoratorA(
              new ConcreteComponent()));
```

Collaborations

- Decorator forwards requests to its Component object. It may optionally perform additional operations before and/or after forwarding the request.

Consequences

- More flexibility than static inheritance.
  - With Decorators, responsibilities can be added and removed at run-time simply by attaching and detaching them. Decorators also make it easy to add a property twice. For example, to give a TextView a double border, simply attach two BorderDecorators.
- Avoids feature-laden(过多特性的) classes high up in the hierarchy.
  - Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects. Functionality can be composed from simple pieces.
- By permutation and combination, lots of behavioral combinations can be created.

Consequences

- A decorator and its component aren't identical. shouldn't rely on object identity or true type when you use decorators.
- A design that uses Decorator often results in systems composed of lots of little objects that all look alike. The objects differ only in the way they are interconnected.
- Remove the Decorator from component is very difficult rather than recreating a new one.

Applicability

- To add responsibilities to individual objects dynamically and transparently.
- For responsibilities that can be withdrawn. (difficult to implement)
- When extension by subclassing is impractical.
  - Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or
  - A class definition may be hidden or otherwise unavailable for subclassing.

Implementation

- Interface conformance.
  - A decorator object's interface must conform to the interface of the component it decorates
- Keeping Component classes lightweight.
  - To ensure a conforming interface, components and decorators must inherit from a common Component class.
  - The complexity of the Component class might make the decorators too heavyweight to used.
  - Putting a lot of functionality into Component also increases the probability that concrete subclasses will pay for features they don't need.

Think about it

- Changing the skin of an object versus changing its guts. We can think of a decorator as a skin over an object that changes its behavior. An alternative is to change the object's guts.
- The Strategy pattern is a good example of a pattern for changing the guts.

Example

- Ticket of an supermarket

萬寧 HEALTH & BEAUTY mannings

安兒寶A+助長奶粉900克	特價	187.70
安兒健A+兒童成長奶粉900克	特價	153.80
惠氏金裝健兒樂較大嬰兒奶粉	特價	186.50
安嬰寶A+高蛋白奶粉900克	特價	212.50
惠氏金裝幼兒樂幼兒成長奶粉	特價	164.90
2 X 105.00		
安佳全脂奶粉1800克	特價	210.00

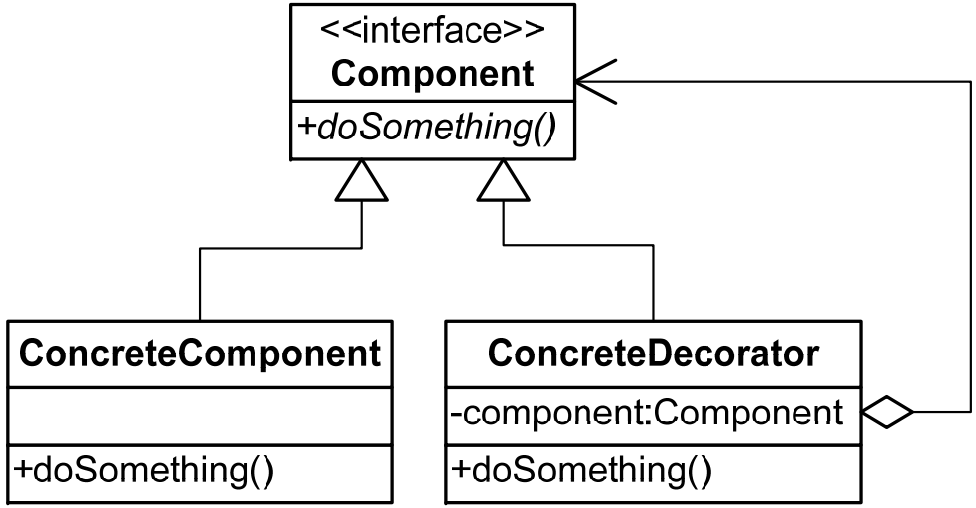
合計(港幣) 1115.40
現金 1125.50
找贖 10.10
交易印花 = 22
額外印花 = 18

總印花 = 40
14天購物保障(憑單據可享退/換貨保障,
印花須一併退回,詳情請參閱店內海報)
新文華中心 電話: 23687759
18-03-2010 15:59:33 R#01 C:3895 MAY T#26142296

索取塑膠購物袋須繳付每個港幣五毫環保徵費,
徵費不設退款

Variation: Decorator is omitted

- It is unnecessary to providing a Decorator if there is only one ConcreteDecorator to decorate the Component.



Extension: Semi-transportation of Decorator

- Transparent decorator pattern: the transparency of decorator pattern requires the ConcreteDecorator do not contains the public methods which are not defined in Component, or clients do not required such methods. (DIP, 针对接口编程)
- Decorator pattern could be semi-transportation. The intent of decorator pattern is adding behaviors dynamically without modifying the interface and introducing the subclasses. But sometimes the public method is defined in ConcreteDecorator when new behaviors are introduced.

Extension: Semi-transportation of Decorator

```
class DecoratorClient {
    public void decoratorClient() {
        VisualComponent component =
            new BorderDecorator(
                    new ScrollDecorator(
                            new TextView()));

        component.draw();
        BorderDecorator borderComponent
                = (BorderDecorator)component;
        borderComponent.drawBorder();
    }
}
```

Let’s go to next…