Decorator Pattern

CS356 Object-Oriented Design and Programming
http://cs356.yusun.io
November 7, 2014

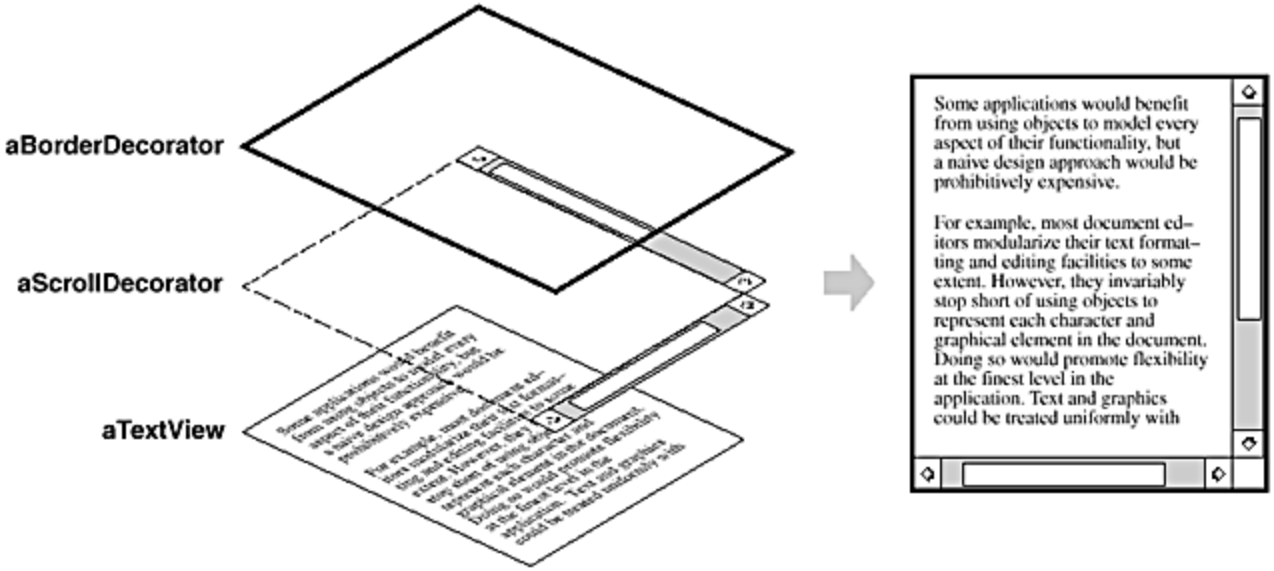
Yu Sun, Ph.D.
http://yusun.io
yusun@csupomona.edu

CAL POLY POMONA

Decorator

- *Intent*
  - Dynamically attach additional responsibilities to an object
  - Provide a flexible alternative to subclassing (static)
  - Decorating object is transparent to the core component
- *Also Known As* – Wrapper

Motivation



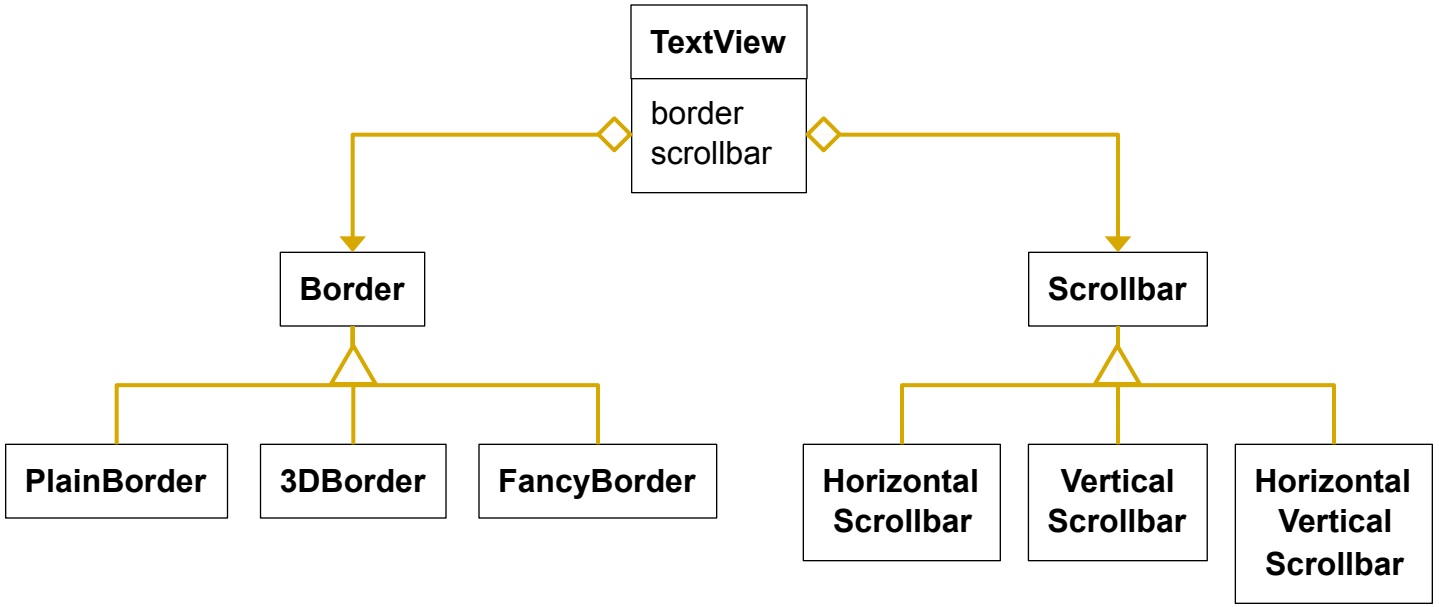
- We want to add different kinds of borders and/or scrollbars to a TextView GUI component
- *Borders* – Plain, 3D, or Fancy
- *Scrollbars* – Horizontal and/or Vertical

Motivation

- An inheritance solution requires 15 subclasses to represent each type of view

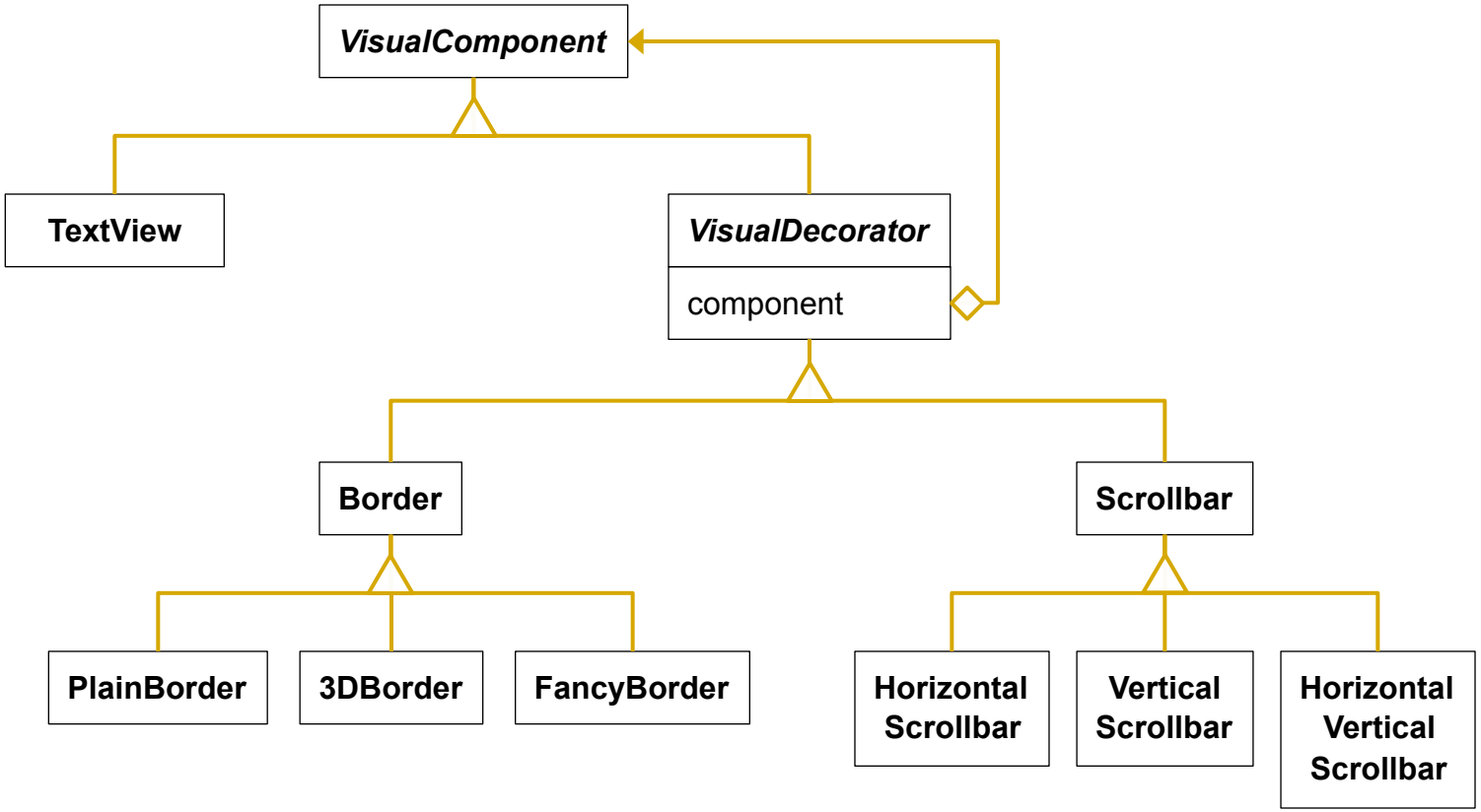
1. TextView-Plain
2. TextView-3D
3. TextView-Fancy
4. TextView-Horizontal
5. TextView-Vertical
6. TextView-Horizontal-Vertical
7. TextView-Plain-Horizontal
8. TextView-Plain-Vertical
9. TextView-Plain-Horizontal-Vertical
10. TextView-3D-Horizontal
11. TextView-3D-Vertical
12. TextView-3D-Horizontal-Vertical
13. TextView-Fancy-Horizontal
14. TextView-Fancy-Vertical
15. TextView-Fancy-Horizontal-Vertical

Solution 1: Use Object Composition



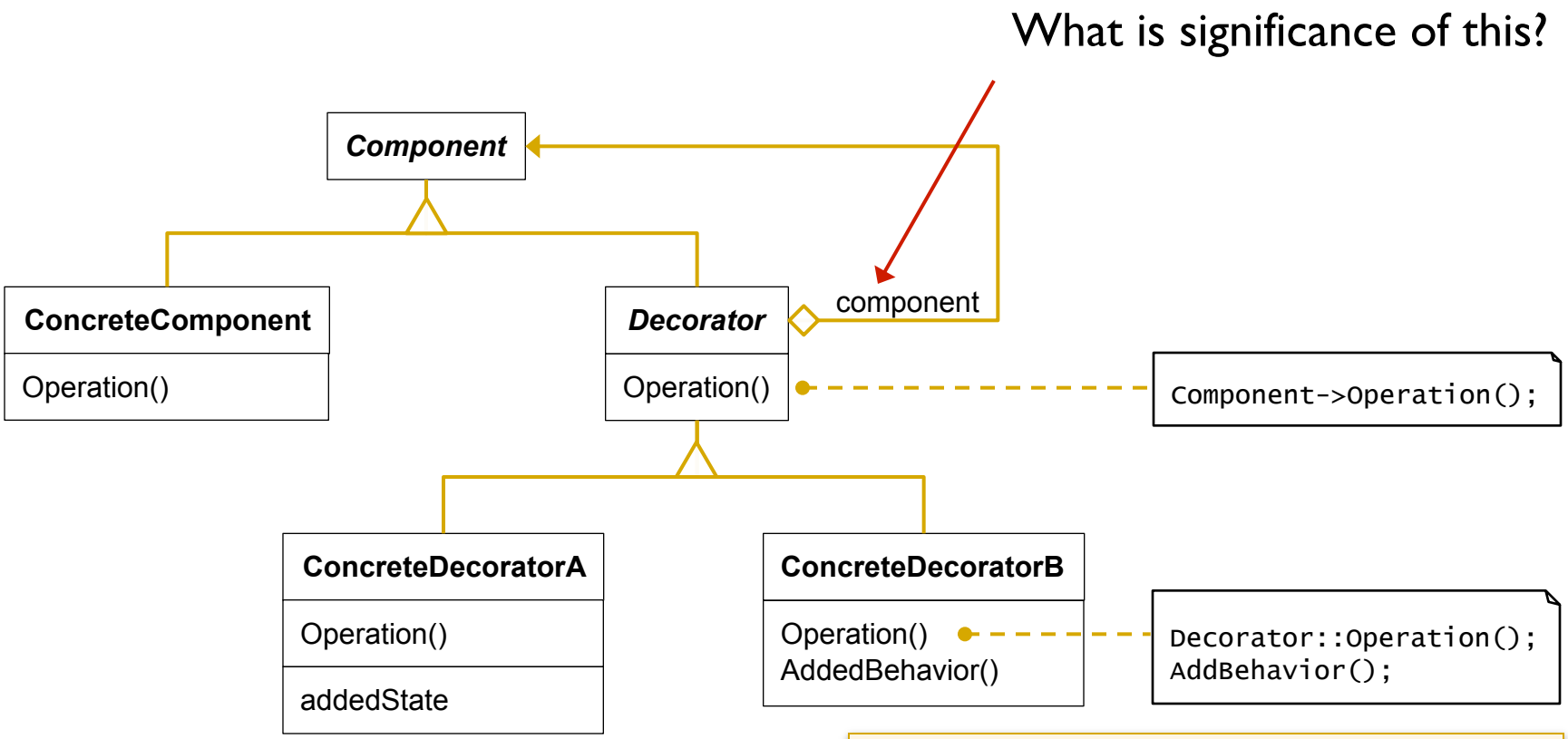
- Is it Open-Closed?
- Can you add new features without affecting TextView?
  - e.g., what about adding sound to a TextView?

Decorator Pattern Solution



- Change the Skin, not the Guts!
- TextView has no borders or scrollbars!
- Add borders and scrollbars on top of a TextView

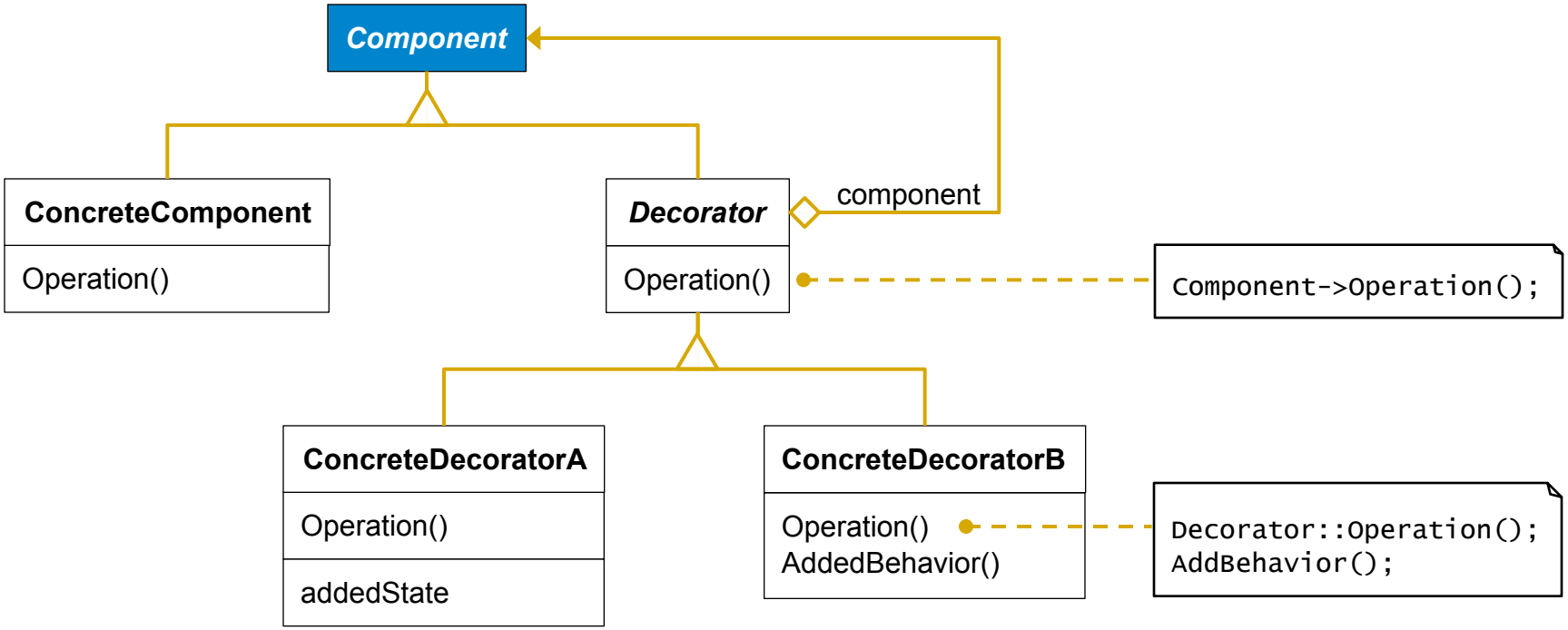
Structure



The decorator forwards requests to the component and may perform additional actions (such as drawing a border) before or after any forwarding

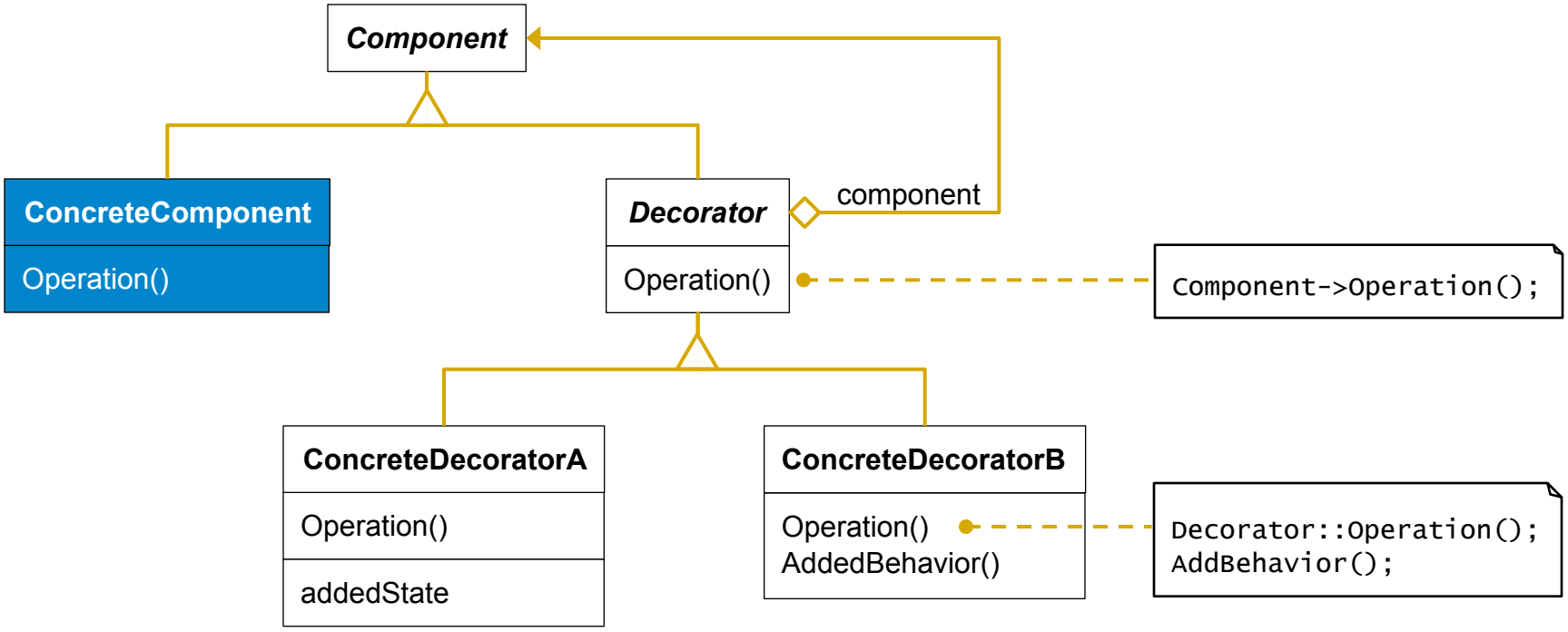
Component

- Defines the interface for objects that can have responsibilities added dynamically



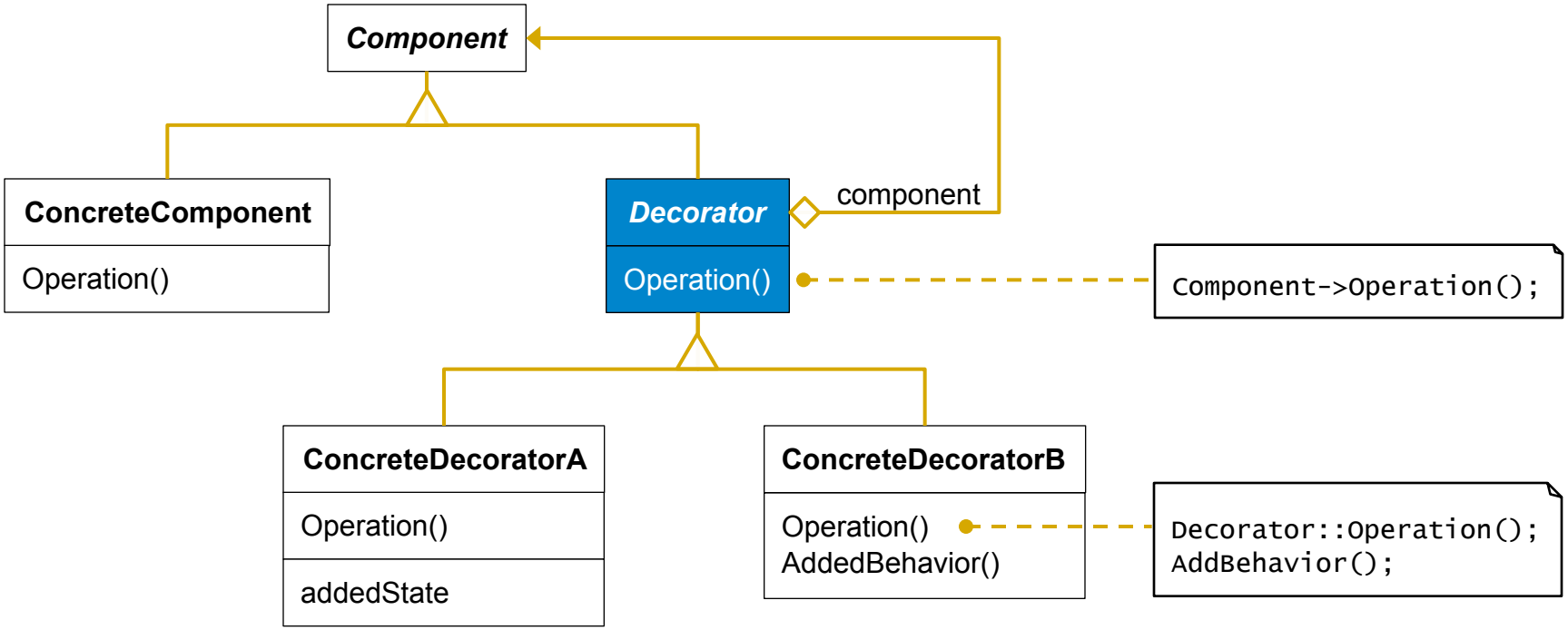
ConcreteComponent

- The "base" object to which additional responsibilities can be added



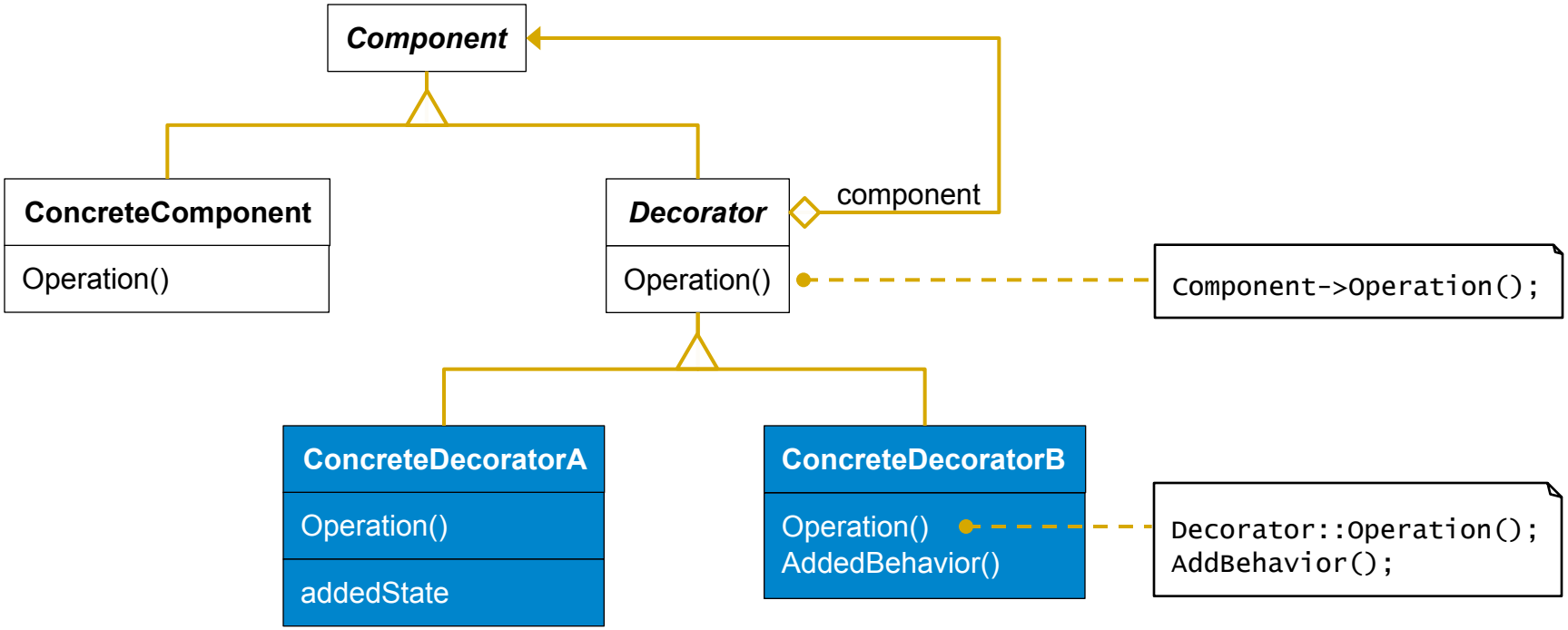
Decorator

- Maintains a reference to a Component object
- Defines an interface conformant to Component's interface



ConcreteDecorator

- Adds responsibilities to the component



Decorator

- *Applicability*
  - Dynamically and transparently attach responsibilities to objects
  - Responsibilities that can be withdrawn
  - Extension by subclassing is impractical
    - May lead to too many subclasses

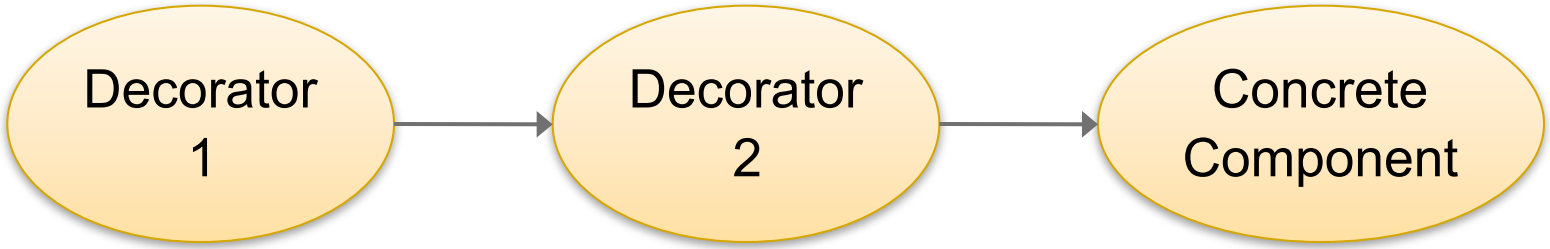
Example – Sales Ticket Printing

Example: Decorate Sales Ticket Printing

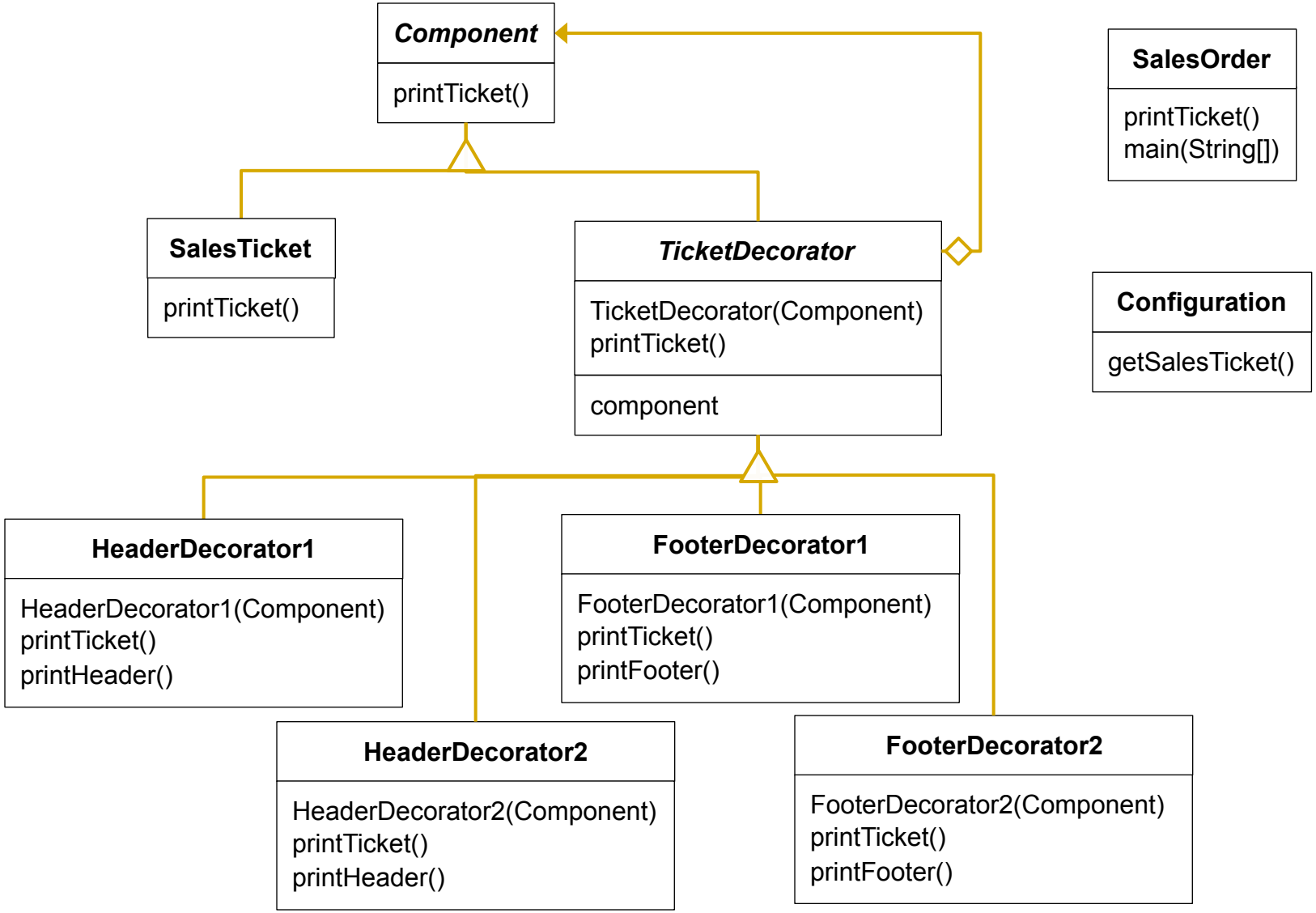
- Assume the SalesTicket currently creates an html sales receipt for an Airline Ticket
- New Requirements
  - Add header with company name
  - Add footer that is an advertisement
  - During the holidays add holiday relevant header(s) and footer(s)
  - We're not sure how many such things
- One solution
  - Place control in SalesTicket
    - Then you need flags to control what header(s) get printed

Decorator Approach

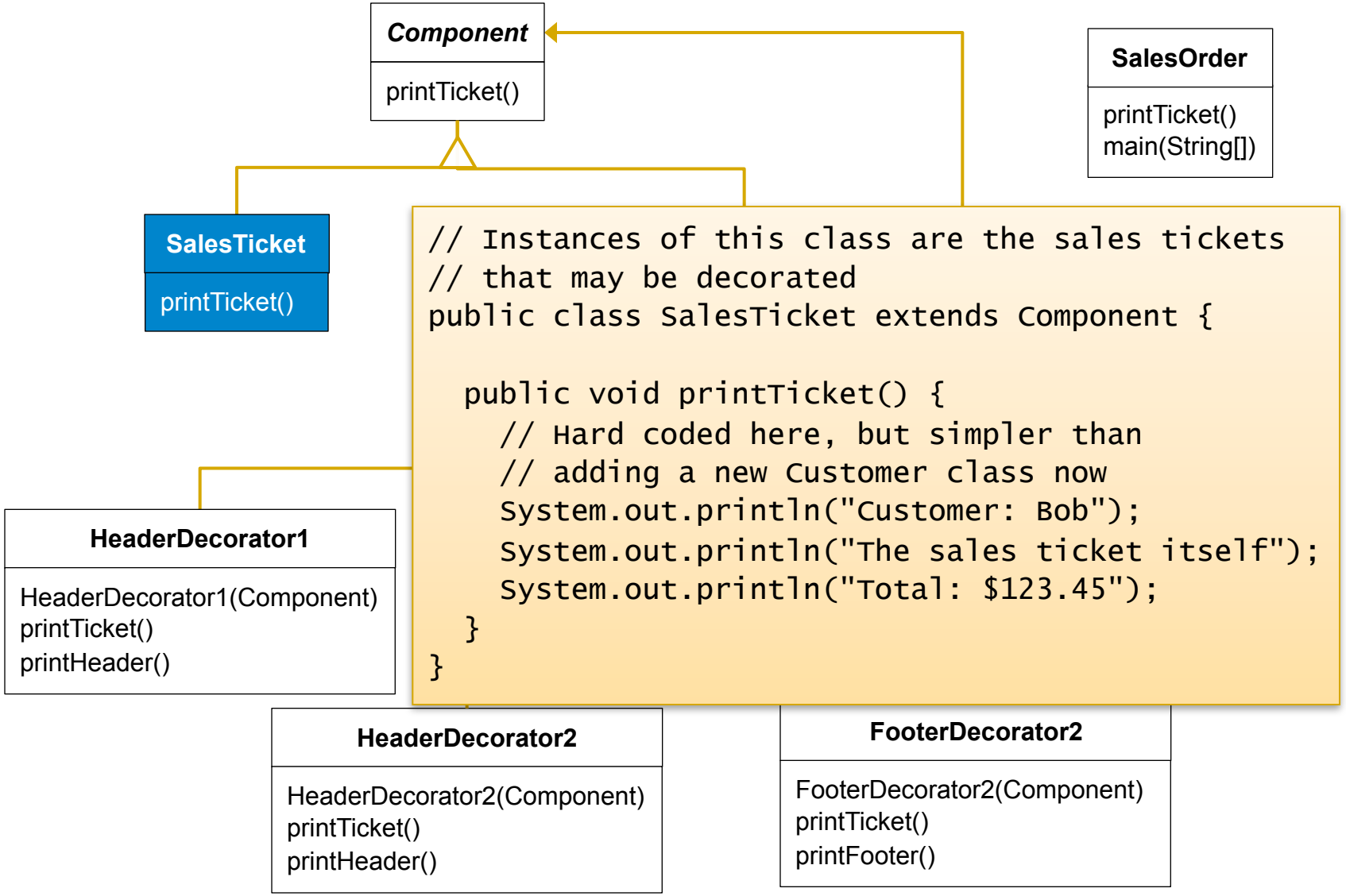
- A layered approach
  - Start chain with decorators
  - End with original object



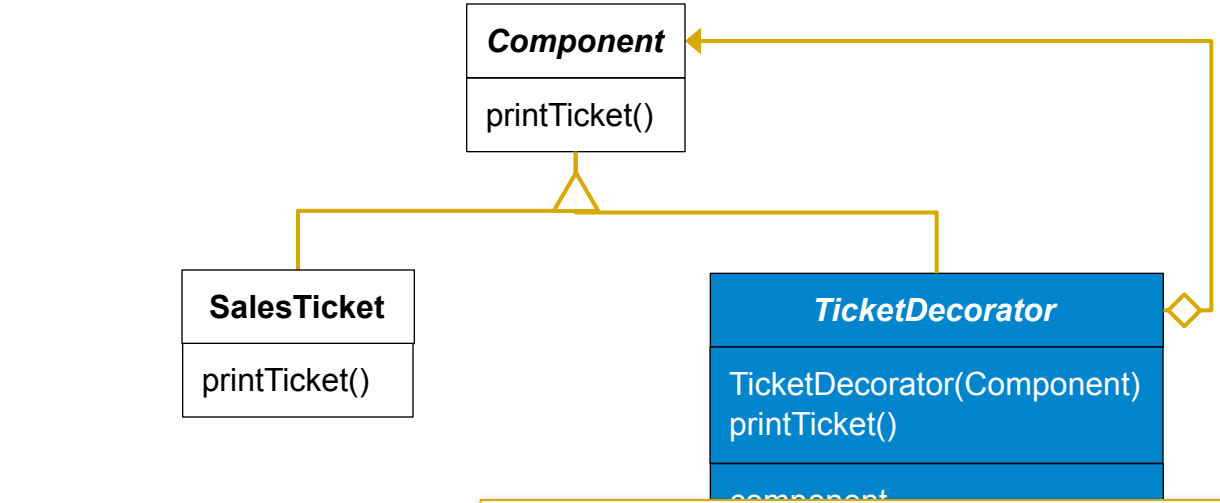
Example – Sales Ticket Printing



A SalesTicket Implementation

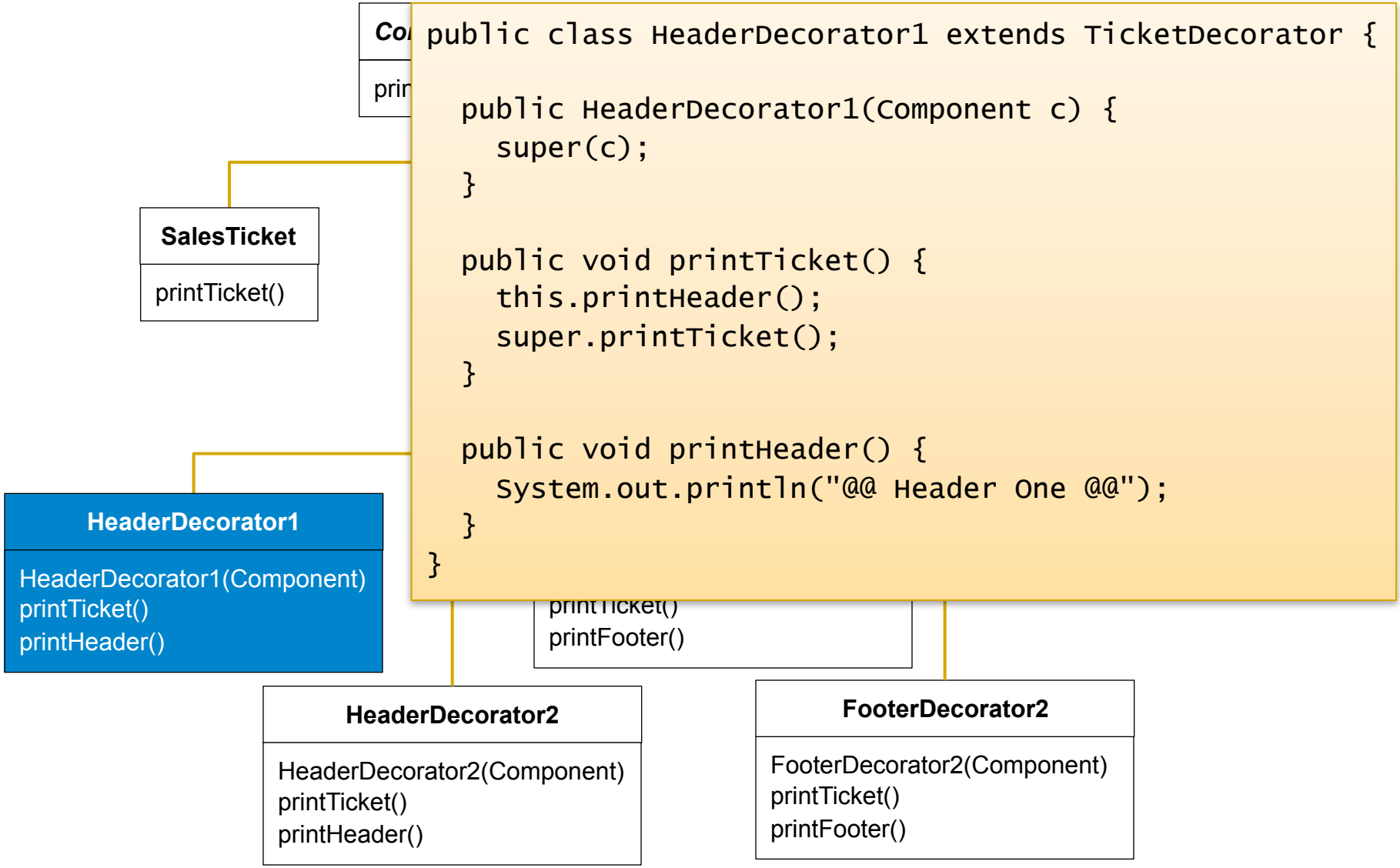


TicketDecorator



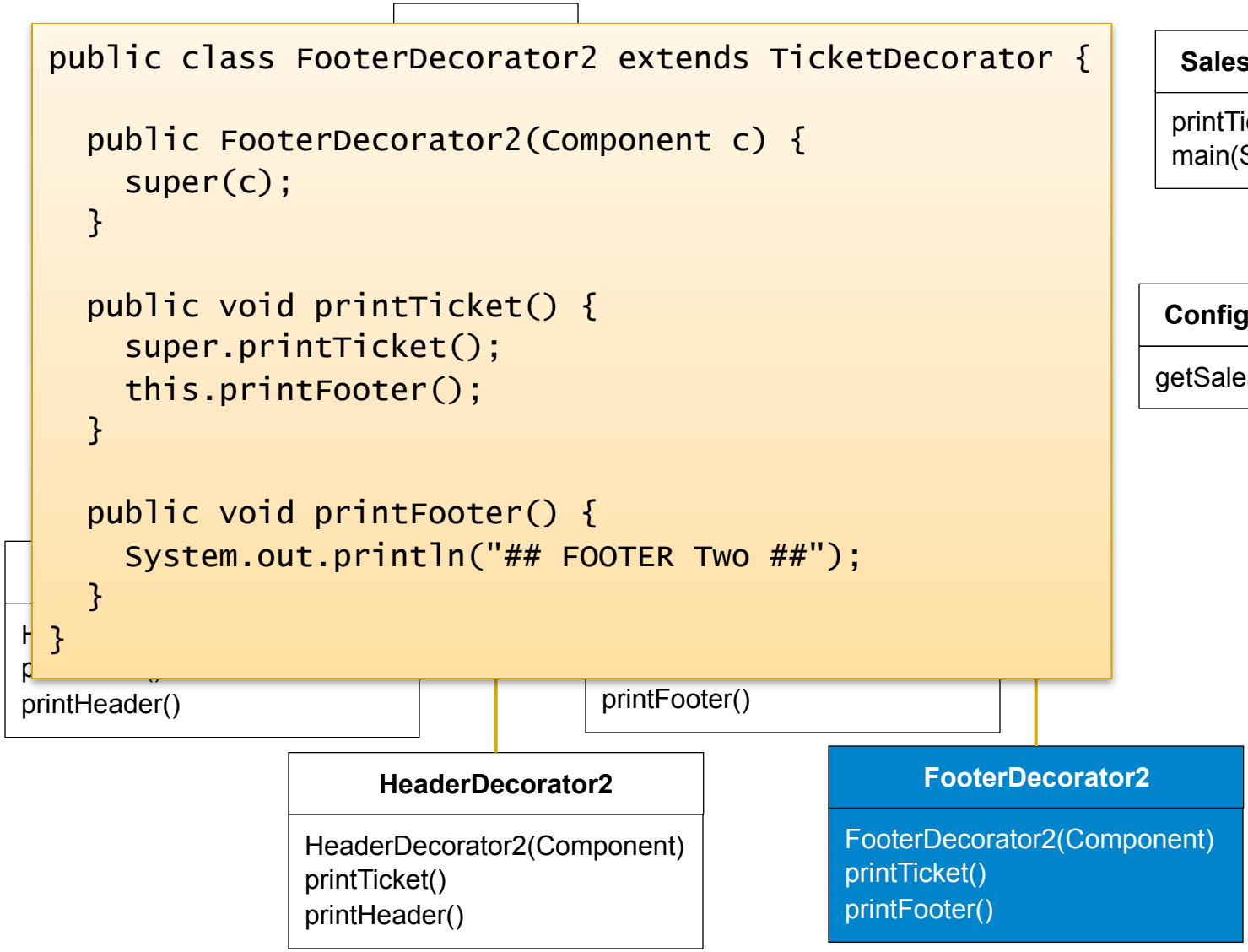

```
public abstract class TicketDecorator extends Component {

  private Component component;

  public TicketDecorator(Component c) {
    component = c;
  }

  public void printTicket() {
    if(component != null)
      component.printTicket();
  }
}
```

A Header Decorator

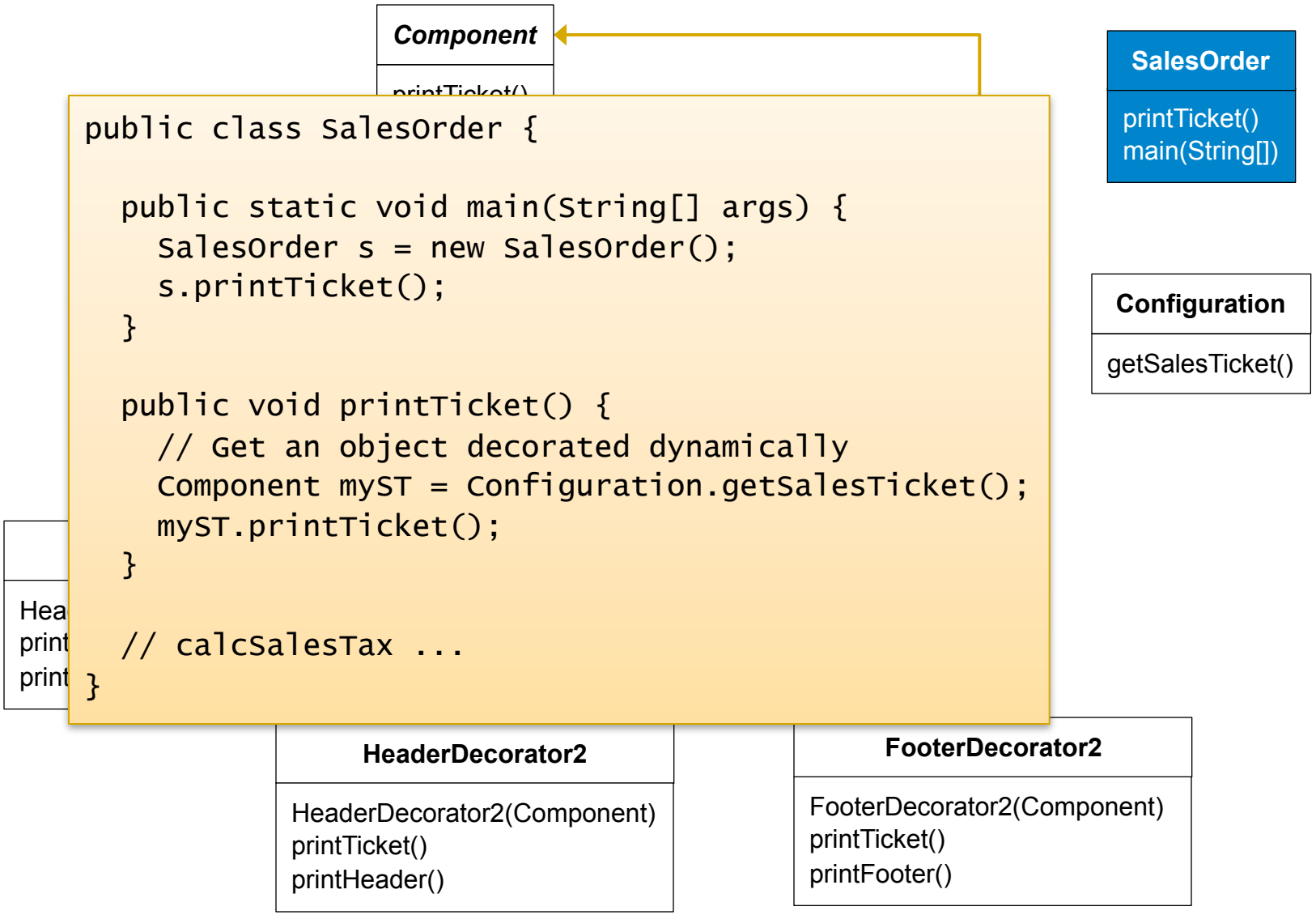


Example – Sales Ticket Printing

```
public class FooterDecorator2 extends TicketDecorator {

  public FooterDecorator2(Component c) {
    super(c);
  }

  public void printTicket() {
    super.printTicket();
    this.printFooter();
  }

  public void printFooter() {
    System.out.println("## FOOTER Two ##");
  }
}
```

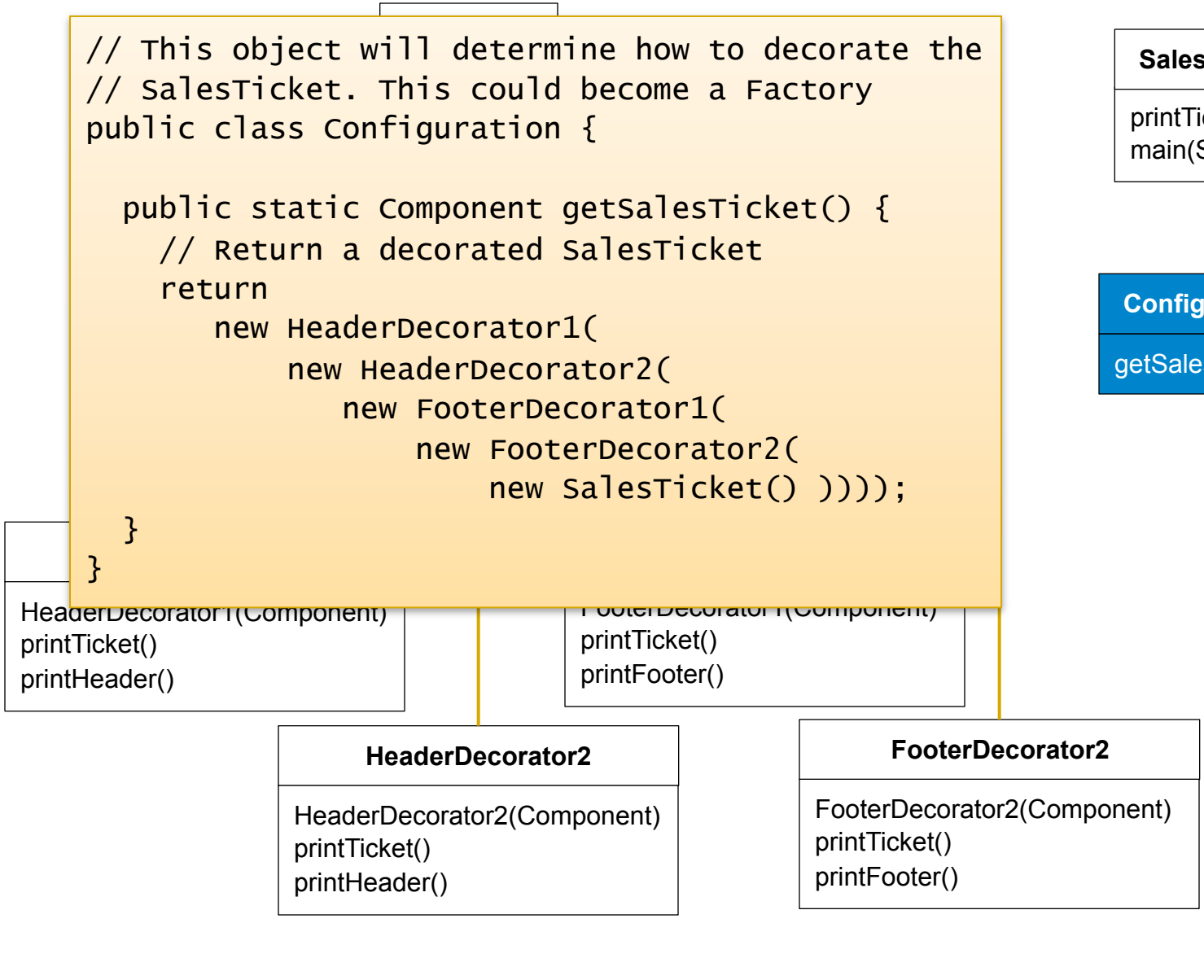


SalesOrder (Client)



Example Configuration

```
// This object will determine how to decorate the
// SalesTicket. This could become a Factory
public class Configuration {

  public static Component getSalesTicket() {
    // Return a decorated SalesTicket
    return
       new HeaderDecorator1(
           new HeaderDecorator2(
              new FooterDecorator1(
                  new FooterDecorator2(
                      new SalesTicket() ))));
  }
}
```



Output with Current Configuration

- Output:

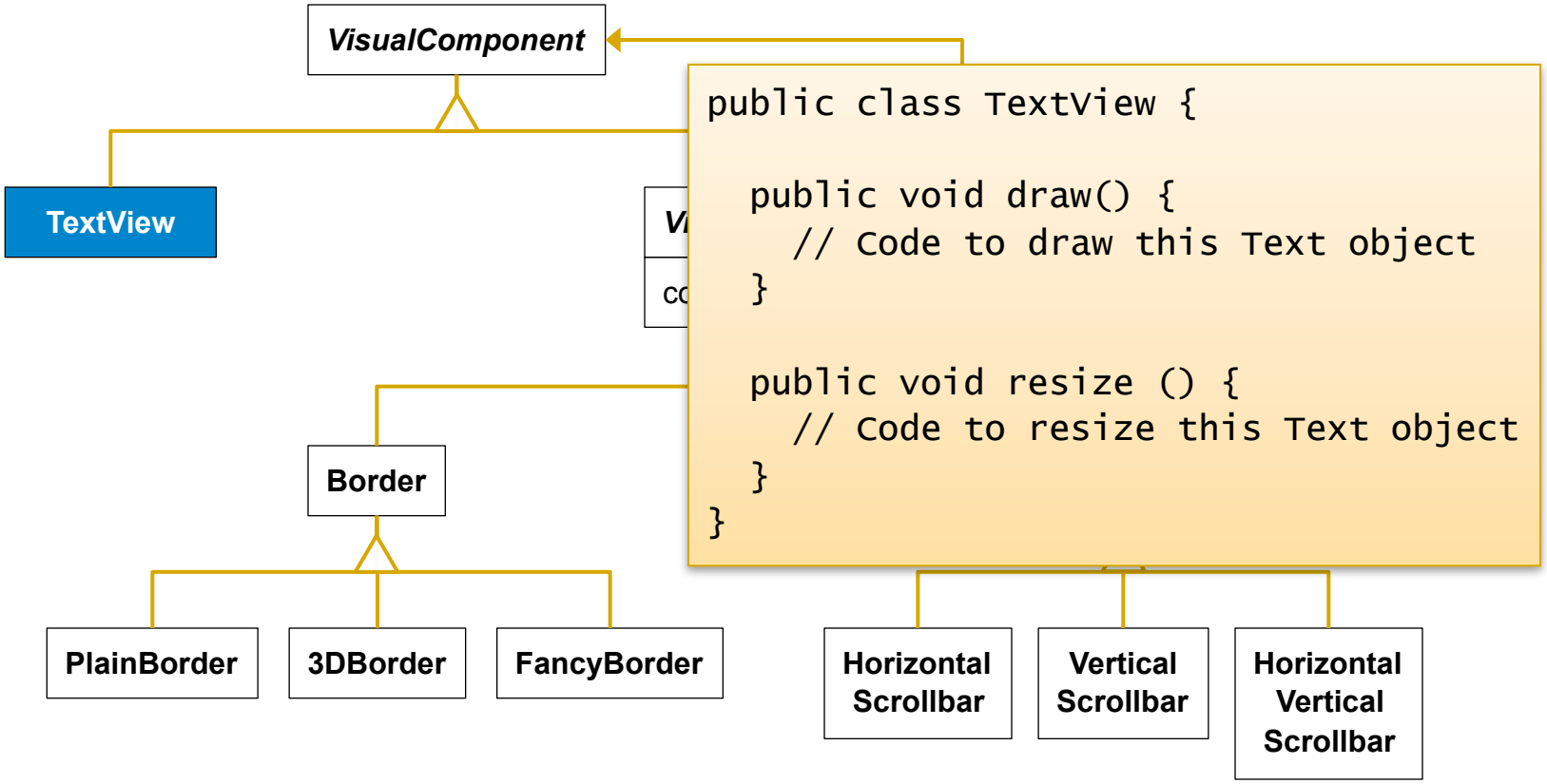
```
@@ Header One @@
>> Header Two <<
Customer: Bob
The sales ticket itself
Total: $123.45
%% FOOTER One %%
## FOOTER Two ##
```

Implementation Issues

- Keep Decorators lightweight
  - Don't put data members in Component
  - Use it for shaping the interface
- Omitting the abstract Decorator class
  - If only one decoration is needed
  - Subclasses may pay for what they don't need

Return to TextView Example

- The TextView class knows nothing about Borders and Scrollbars



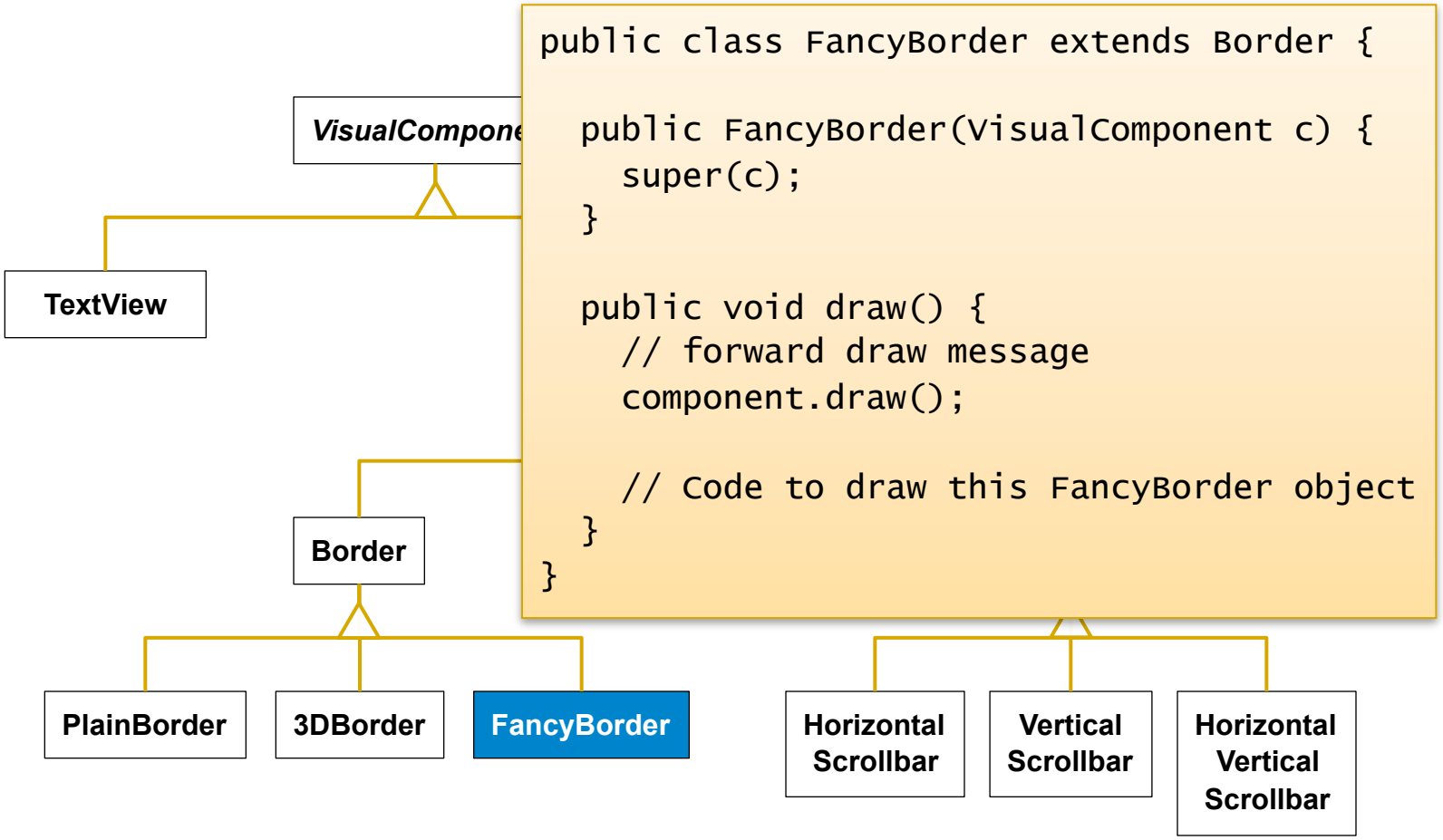
A New Class

- The new ImageView class knows nothing about Borders and Scrollbars

```
public class ImageView {

  public void draw() {
    // Code to draw this Image Object
  }

  public void resize () {
    // Code to resize this Image Object
  }
}
```

Decorators Contain Components

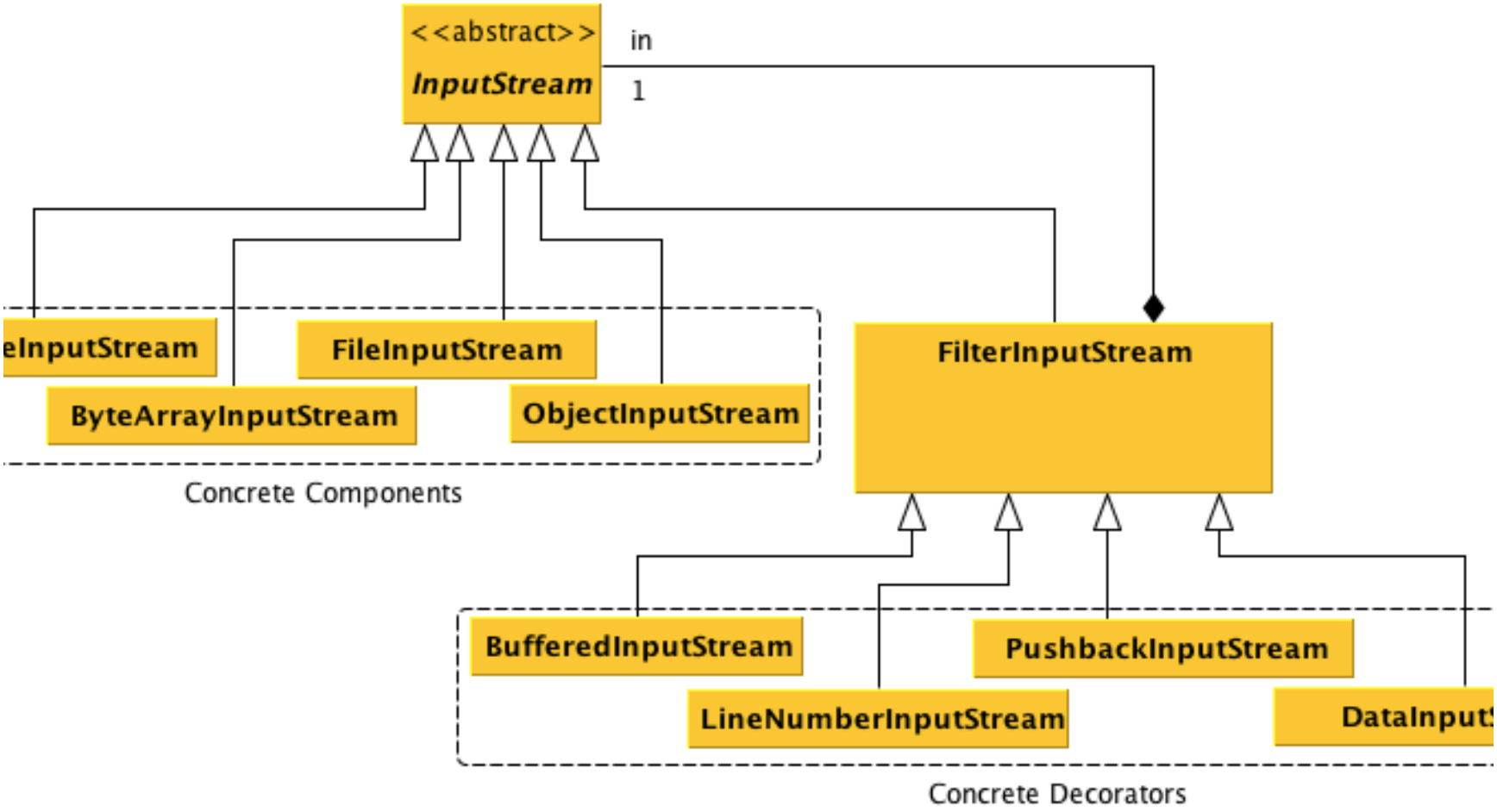
- The decorators don't need to know about components



How to Use Decorators

```
public class Client {

  public static void main(String[] args) {
   TextView data = new TextView();

   Component borderData = new FancyBorder(data);

   Component scrolledData = new VertScrollbar(data);

   Component borderAndScrolledData = new HorzScrollbar(borderData);
  }
}
```

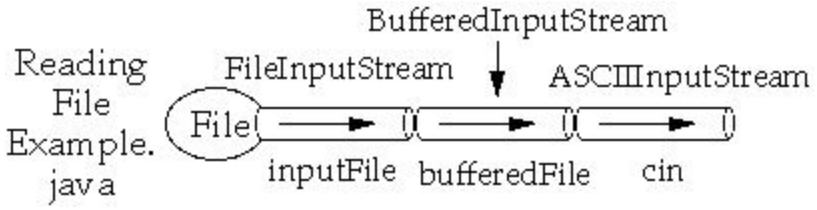
Decorator Pattern in Java



Decorator Pattern in Java

```
public class JavaIO {

  public static void main(String[] args) {
    // Open an InputStream.
    FileInputStream in = new FileInputStream("test.dat");
    // Create a buffered InputStream.
    BufferedInputStream bin = new BufferedInputStream(in);
    // Create a buffered, data InputStream.
    DataInputStream dbin = new DataInputStream(bin);
    // Create a buffered, pushback, data InputStream.
    PushbackInputStream pbdbin = new PushbackInputStream(dbin);
  }
}
```

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



Java Streams

- With > 60 streams in Java, you can create a wide variety of input and output streams
  - This provides flexibility (good)
  - It also adds complexity (bad)
  - Flexibility made possible with inheritance and classes that accept many different classes that extend the parameter
- You can have an InputStream instance or any instance of a class that extends InputStream

```
public InputStreamReader(InputStream in)
```

Consequences

- + Transparency – very good
- + More flexibility than static inheritance
  - Allows to mix and match responsibilities
  - Allows to apply a property twice
- + Avoid feature-laden classes high-up in the hierarchy
  - "Pay-as-you-go" approach
  - Easy to define new types of decorations

- × A decorator and its component aren't identical
- × Lots of little objects
  - Easy to customize, but hard to learn and debug