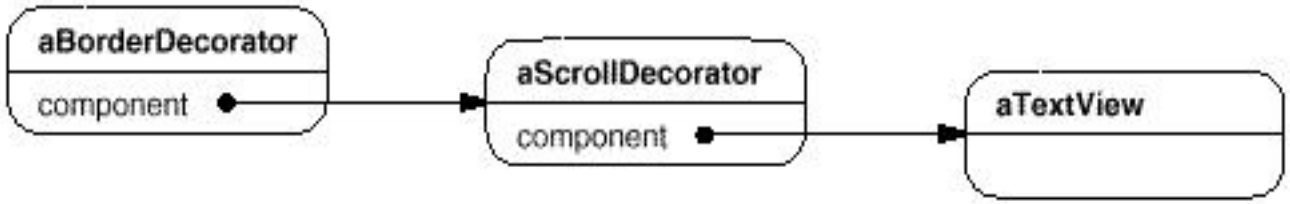
The Decorator Pattern

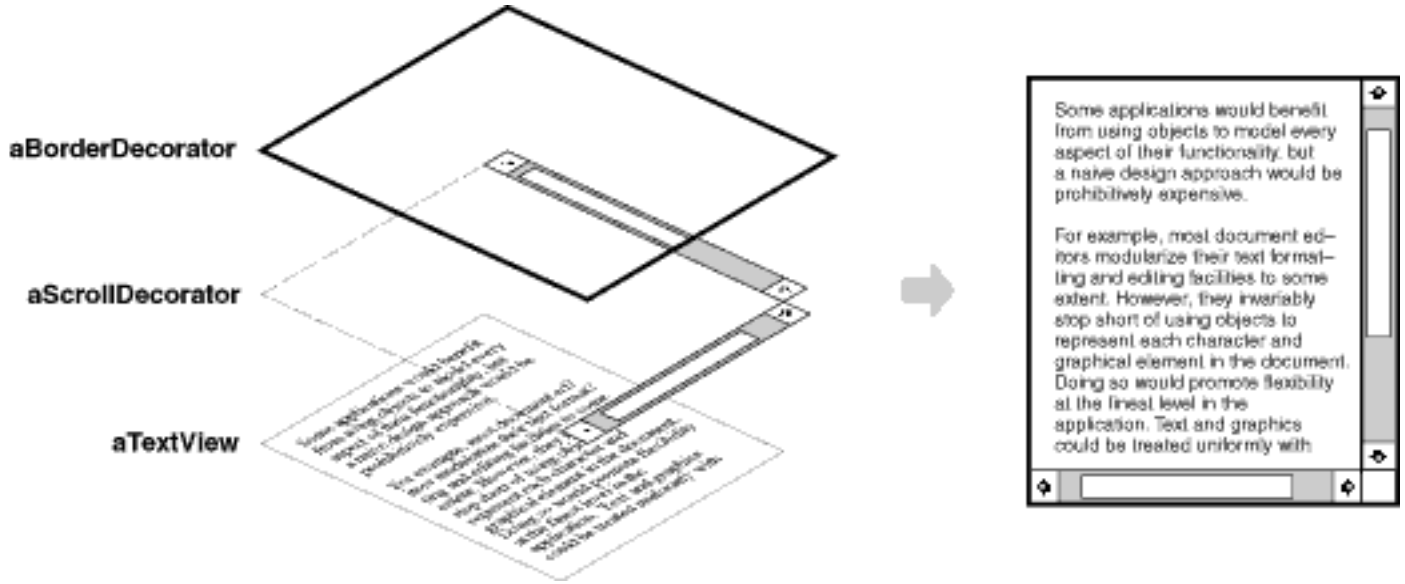
The Decorator Pattern

- Intent
  - ⇨ Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Also Known As
  - ⇨ Wrapper
- Motivation
  - ⇨ We want to add properties, such as borders or scrollbars to a GUI component. We can do this with inheritance (subclassing), but this limits our flexibility. A better way is to use composition!



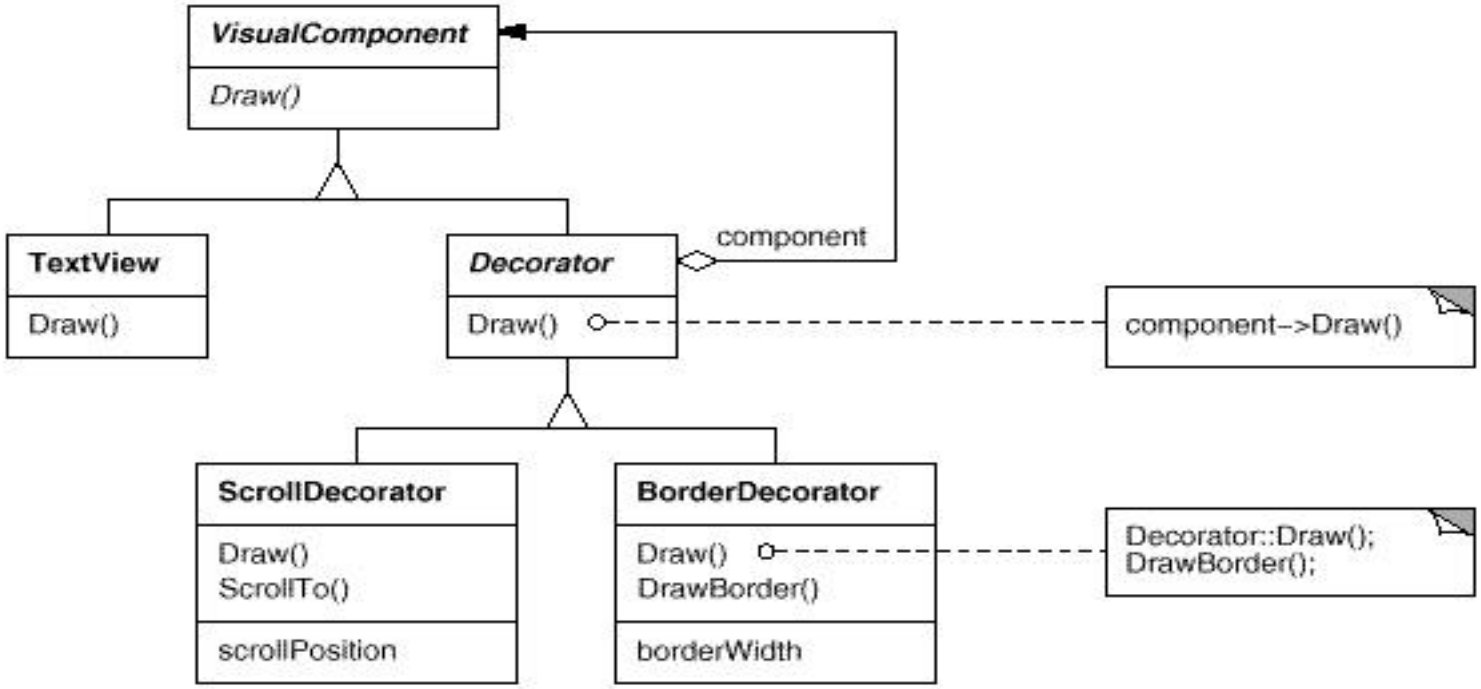
The Decorator Pattern

- Motivation



The Decorator Pattern

- Motivation

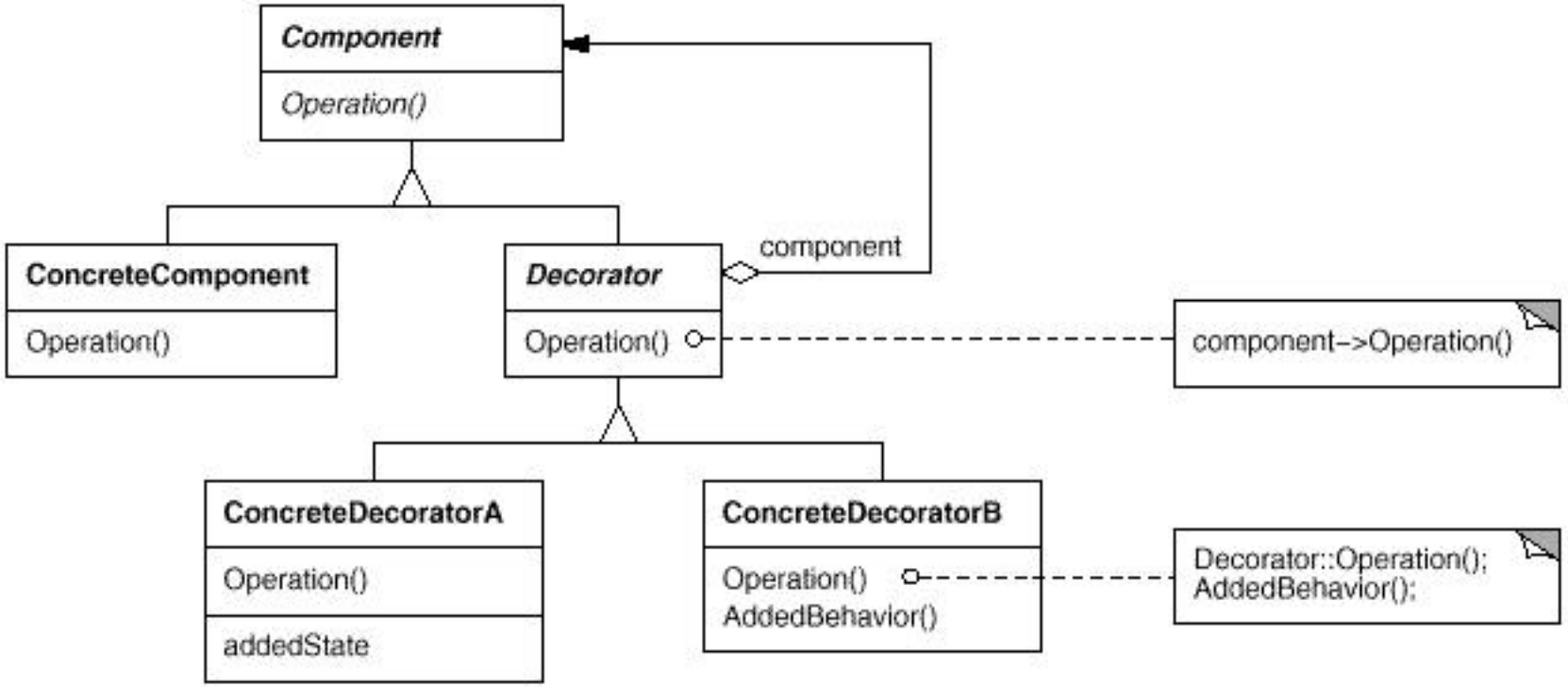


The Decorator Pattern

- Applicability
  - Use Decorator:
    - To add responsibilities to individual objects dynamically without affecting other objects.
    - 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.

The Decorator Pattern

- Structure



Decorator Example 1

- Let's look at the motivation for the Decorator pattern in a little more detail. Suppose we have a TextView GUI component and we want to add different kinds of borders and scrollbars to it.
- Suppose we have three types of borders:
  - ⇨ Plain, 3D, Fancy
- And two types of scrollbars:
  - ⇨ Horizontal, Vertical
- Solution 1: Let's use inheritance first. We'll generate subclasses of TextView for all the required cases. We'll need the 15 subclasses:
  - ⇨ TextView-Plain
  - ⇨ TextView-Fancy
  - ⇨ TextView-3D

Decorator Example 1 (Continued)

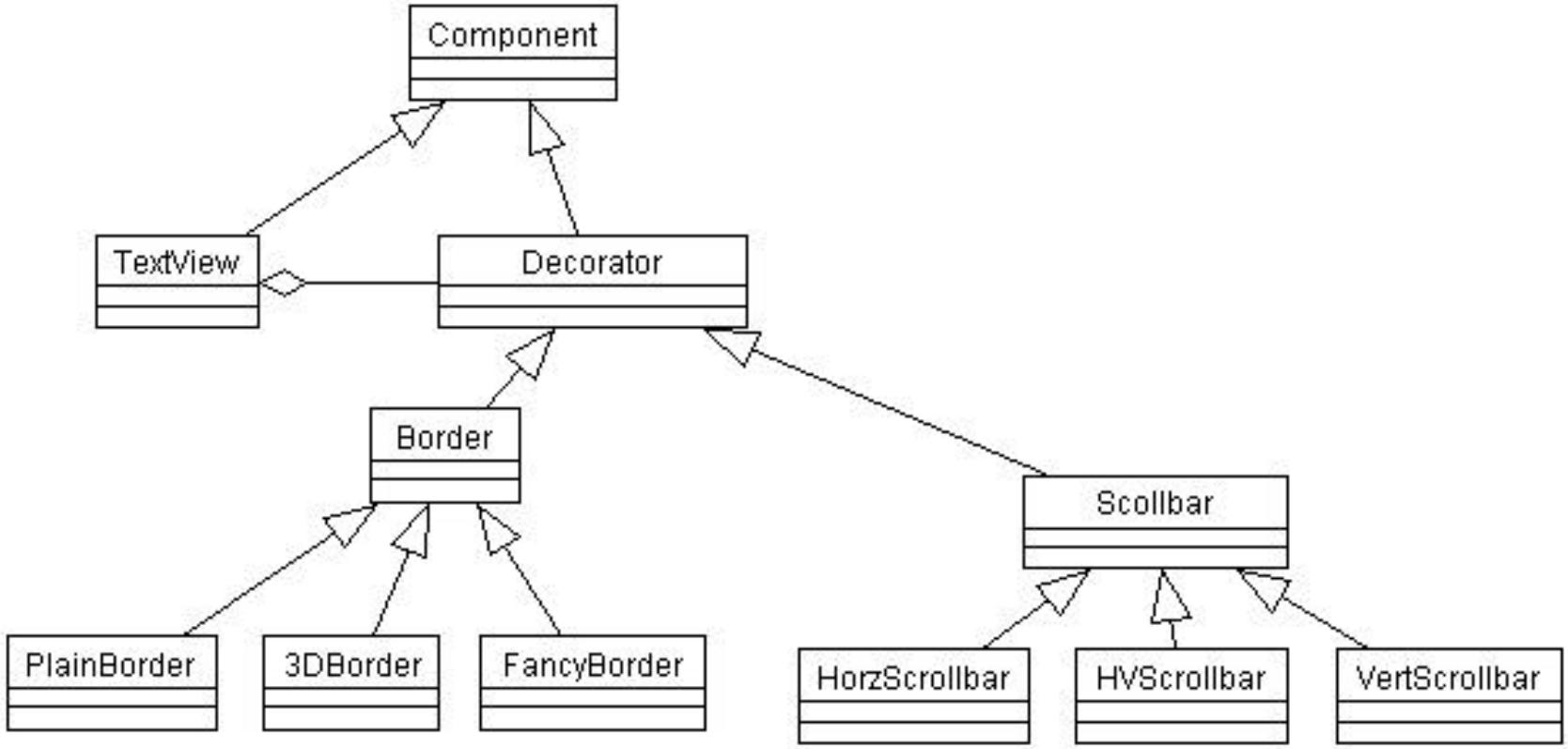
- TextView-Horizontal
- TextView-Vertical
- TextView-Horizontal-Vertical
- TextView-Plain-Horizontal
- TextView-Plain-Vertical
- TextView-Plain-Horizontal-Vertical
- TextView-3D-Horizontal
- TextView-3D-Vertical
- TextView-3D-Horizontal-Vertical
- TextView-Fancy-Horizontal
- TextView-Fancy-Vertical
- TextView-Fancy-Horizontal-Vertical

Decorator Example 1 (Continued)

- There are several disadvantages to this technique:
  - We already have an explosion of subclasses. What if we add another type of border? Or an entirely different property?
  - We have to instantiate a specific subclass to get the behavior we want. This choice is made statically and a client can't control how and when to decorate the component.

Decorator Example 1 (Continued)

- Solution 2: Let's use the Strategy pattern!



Decorator Example 1 (Continued)

- Now the TextView Class looks like this:

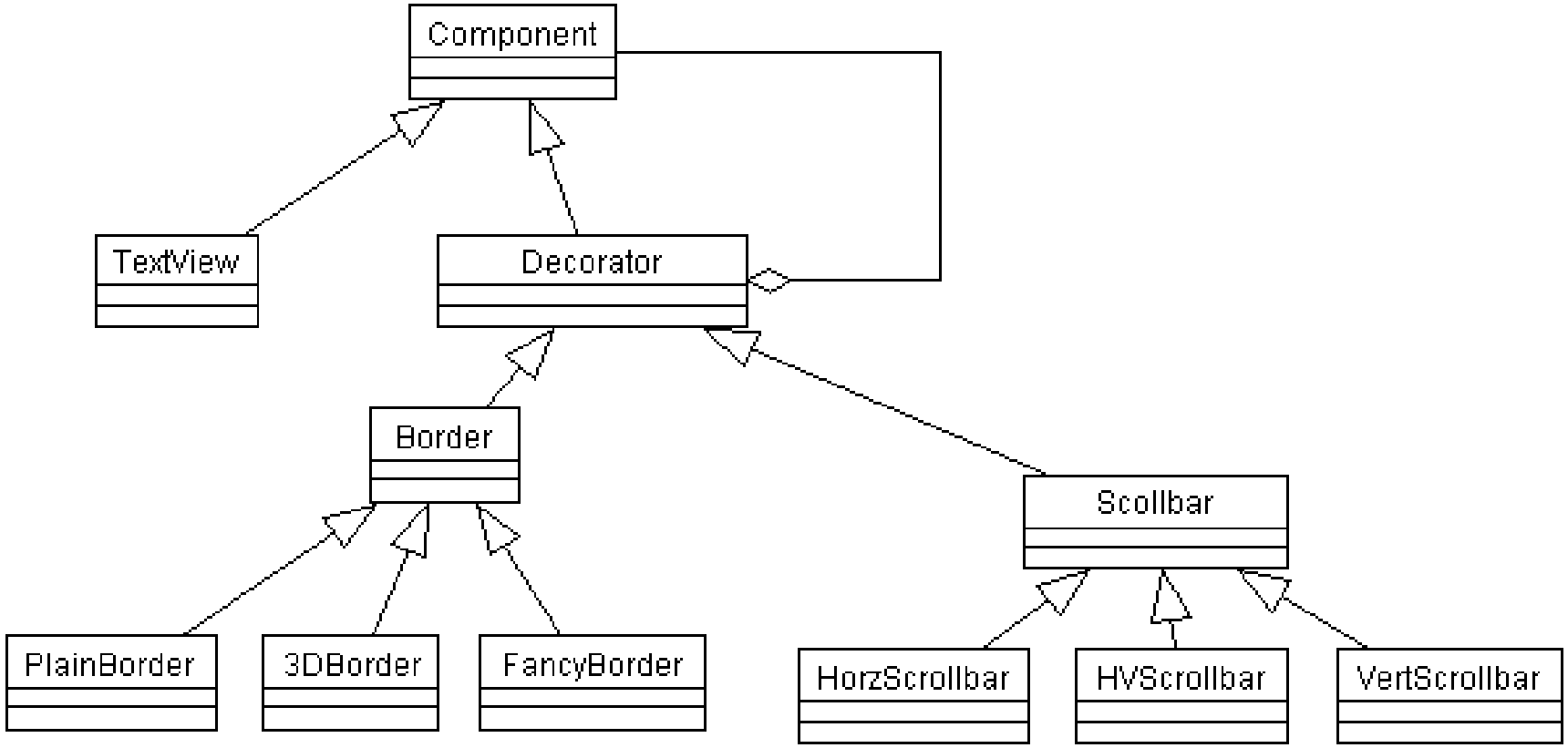
```
public class TextView extends Component {
  private Border border;
  private Scrollbar sb;

  public TextView(Border border, Scrollbar sb) {
    this.border = border;
    this.sb = sb;
  }
  public void draw() {
    border.draw();
    sb.draw();
    // Code to draw the TextView object itself.
  }
}
```

Decorator Example 1 (Continued)

- Using the Strategy pattern we can add or change properties to the TextView component dynamically. For example, we could have mutators for the border and sb attributes and we could change them at run-time.
- But note that the TextView object itself had to be modified and it has knowledge of borders and scrollbars! If we wanted to add another kind of property or behavior, we would have to again modify TextView.

Decorator Example 1 (Continued)

- Solution 3: Let's turn Strategy inside out to get the Decorator pattern!



Decorator Example 1 (Continued)

- Now the TextView class knows nothing about borders and scrollbars:

```
public class TextView extends Component {
  public void draw() {
    // Code to draw the TextView object itself.
  }
}
```

Decorator Example 1 (Continued)

- But the decorators need to know about components:

```
public class FancyBorder extends Decorator {
  private Component component;

  public FancyBorder(Component component) {
    this.component = component;
  }

  public void draw() {
    component.draw();
    // Code to draw the FancyBorder object itself.
  }

}
```

Decorator Example 1 (Continued)

- Now a client can add borders as follows:

```
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: Changing the skin of an object
- Strategy: Changing the guts of an object

Decorator Example 2

- Java I/O classes use the Decorator pattern
- The basic I/O classes are InputStream, OutputStream, Reader and Writer. These classes have a very basic set of behaviors.
- We would like to add additional behaviors to an existing stream to yield, for example:
  - ⇨ Buffered Stream - adds buffering for the stream
  - ⇨ Data Stream - allows I/O of primitive Java data types
  - ⇨ Pushback Stream - allows undo operation
- We really do not want to modify the basic I/O classes to achieve these behaviors, so we use decorator classes, which Java calls filter classes, to add the desired properties using composition

Decorator Example 2 (Continued)

- Some examples of the decorator (filter) classes are:
  - BufferedInputStream
  - DataInputStream
  - PushbackInputStream
- The constructors for these classes take an InputStream object

Decorator Example 2 (Continued)

- Here is an example of the use of these classes:

```
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 an unbuffered, data InputStream.
    DataInputStream din = new DataInputStream(in);
    // Create a buffered, pushback, data InputStream.
    PushbackInputStream pbdbin = new PushbackInputStream(dbin);
  }
}
```