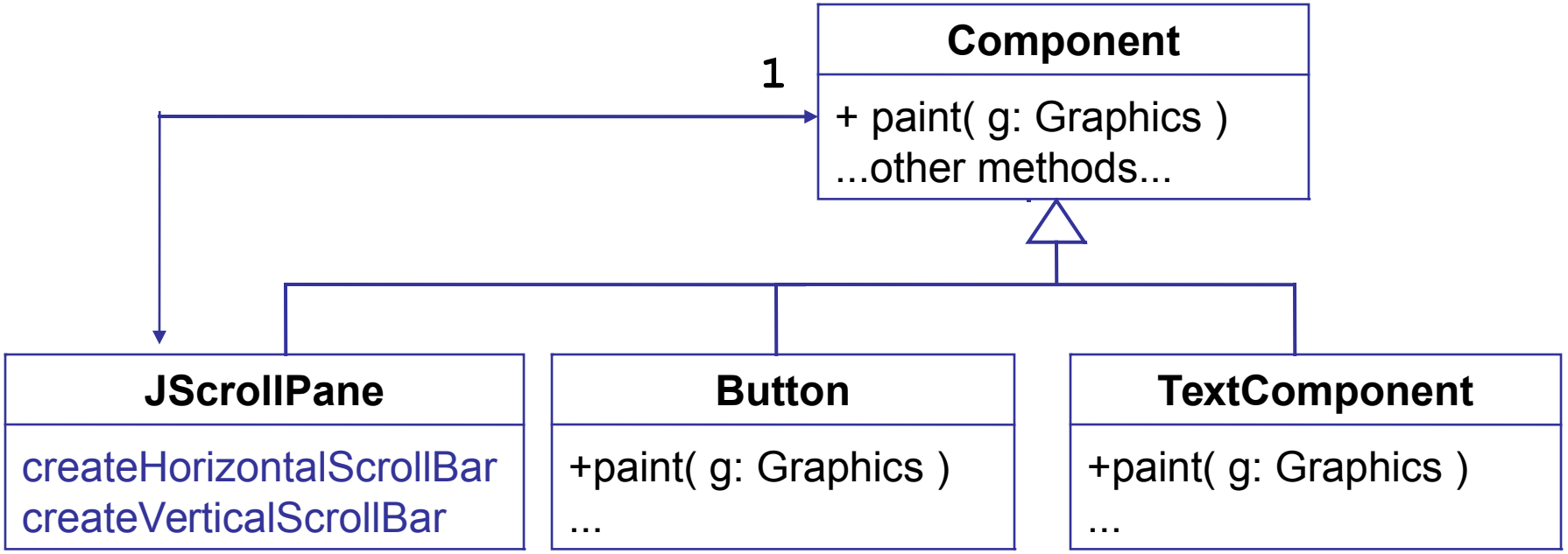
Decorator Pattern

Context: We want to *enhance* the behavior of a class, and there may be many (open-ended) ways of enhancing the class.

The *enhanced* class can be used the same as the base class.

Solution: Create an interface for the base class. The base class implements the interface. Create a *decorator* that implements the interface and wraps the plain class, "decorating" its behavior.



Decorator Example

Purpose: create a TextArea with scrollbars so that text will scroll when larger than the viewport.

```
// create TextArea with 5 rows, 40 columns
JTextArea textArea = new JTextArea( 5, 40 );
// decorate with JScrollPane to add scrollbars
JScrollPane pane = new JScrollPane( textArea );
pane.setVerticalScrollBarPolicy(
   JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED );

// Add the decorator to the contentpane.
// Don't add the textArea!
contentPane.add( pane );
```

Advantage of Using Decorators (1)

- We can write a behavior one time and apply it to many different kinds of objects.

Example: a JScrollPane can be applied to any component, not just JTextArea.

Advantage of Using Decorators (2)

- Improves the *cohesion* of objects, by not adding responsibility that isn't part of the object's main purpose.

Example: the purpose of a TextArea is to display text!

Not to manage scrolling.

Advantage of Using Decorators (3)

- New decorators can be added in the future, *extending* the behavior of the class.

Example: a *zoom decorator* to zoom a component.

Open-Closed Principle

A class should be open for extension but closed for modification.

Disadvantage of Decorators

Lots of pass-through methods:

Any method the decorator doesn't "decorate" itself, it must pass to the decorated object.