
# SHRI RAMDEOBABA COLLEGE OF ENGINEERING AND MANAGEMENT, NAGPUR - 440013

## *DESIGN PATTERNS*
## *V SEMESTER*

*COURSE COORDINATOR: RINA DAMDOO*

***DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING***

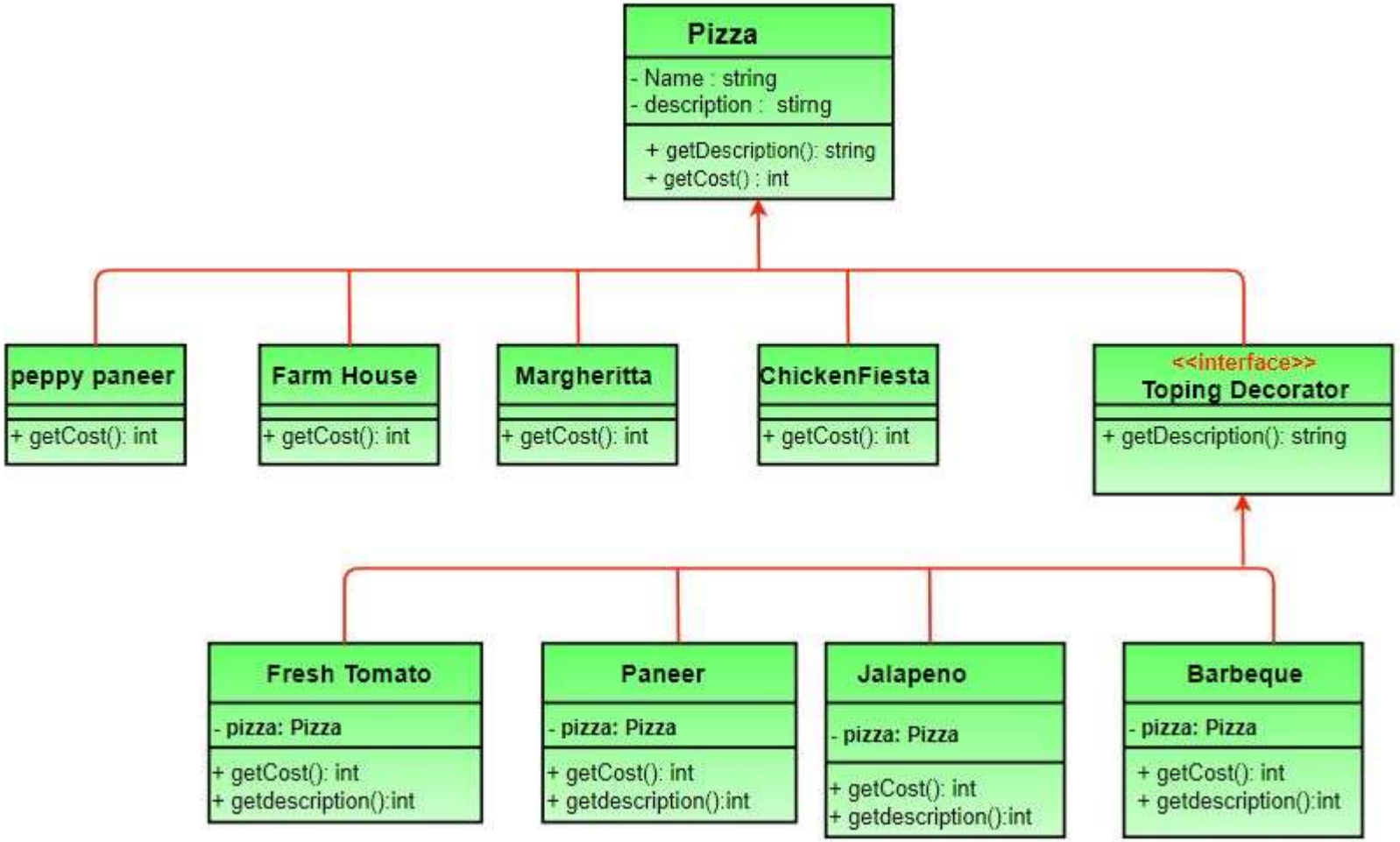
# DECORATOR DESIGN PATTERNS

- **Intent**

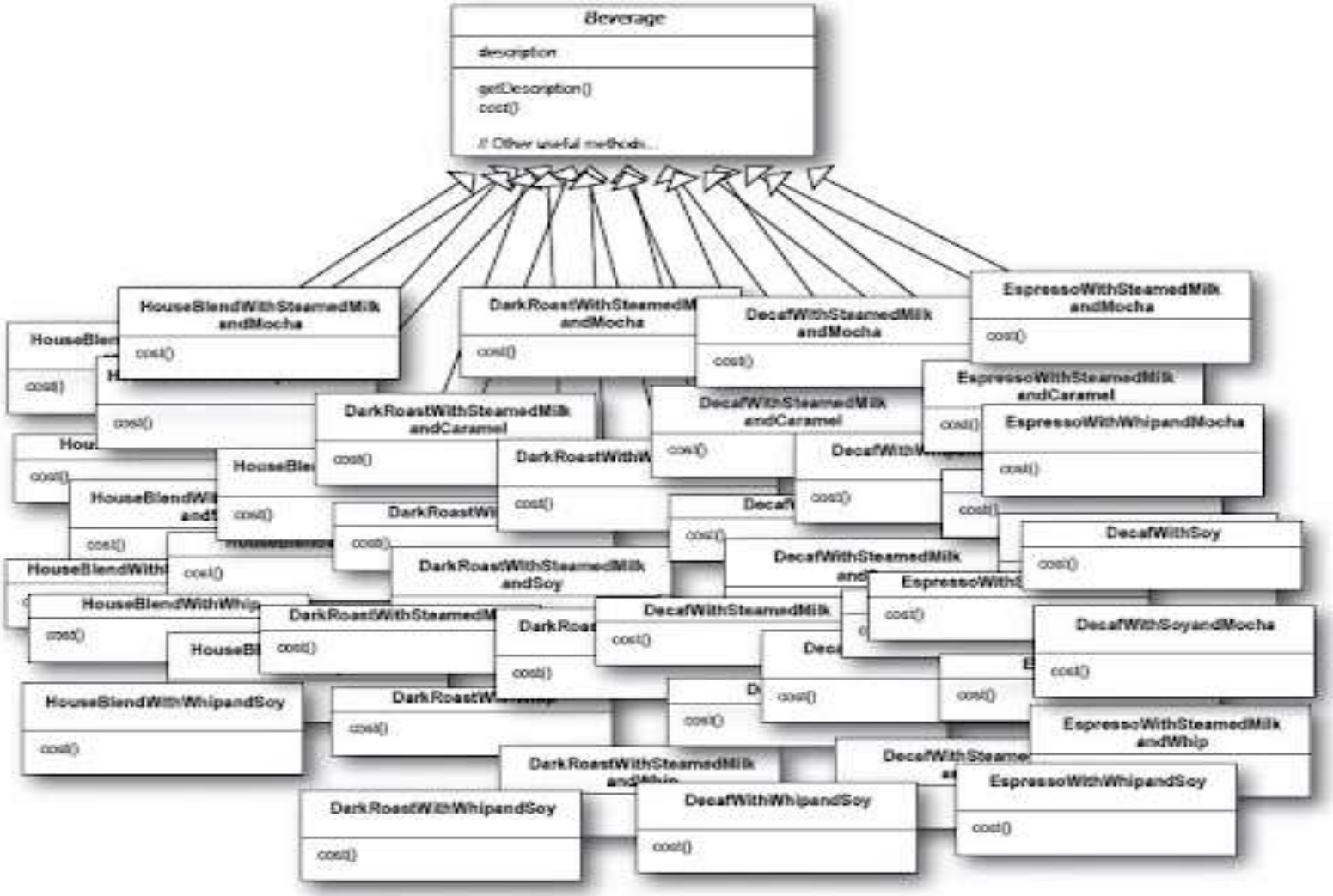
  Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

**Also Known As:** Wrapper

- The Decorator pattern demonstrates how to support many permutations of core and optional features in software
- All the features are mapped to classes that each implement a common interface, and then the user can mix and match any set of features

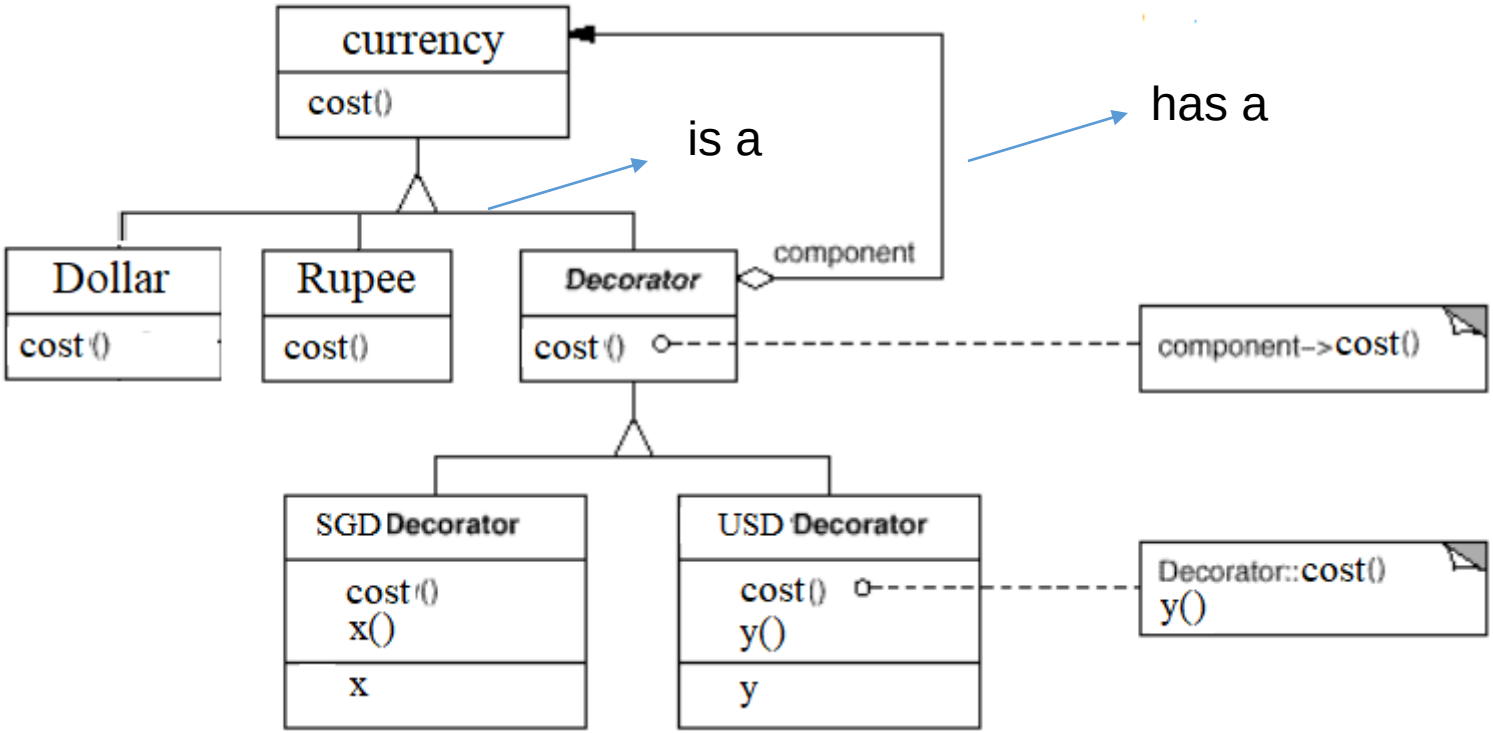

Pizza
- Name : string
- description : stirng
+ getDescription(): string
+ getCost() : int
peppy paneer
+ getCost(): int
Farm House
+ getCost(): int
Margheritta
+ getCost(): int
ChickenFiesta
+ getCost(): int
<<interface>>
Toping Decorator
+ getDescription(): string
Fresh Tomato
- pizza: Pizza
+ getCost(): int
+ getdescription():int
Paneer
- pizza: Pizza
+ getCost(): int
+ getdescription():int
Jalapeno
- pizza: Pizza
+ getCost(): int
+ getdescription():int
Barbeque
- pizza: Pizza
+ getCost(): int
+ getdescription():int

# Explosion of classes



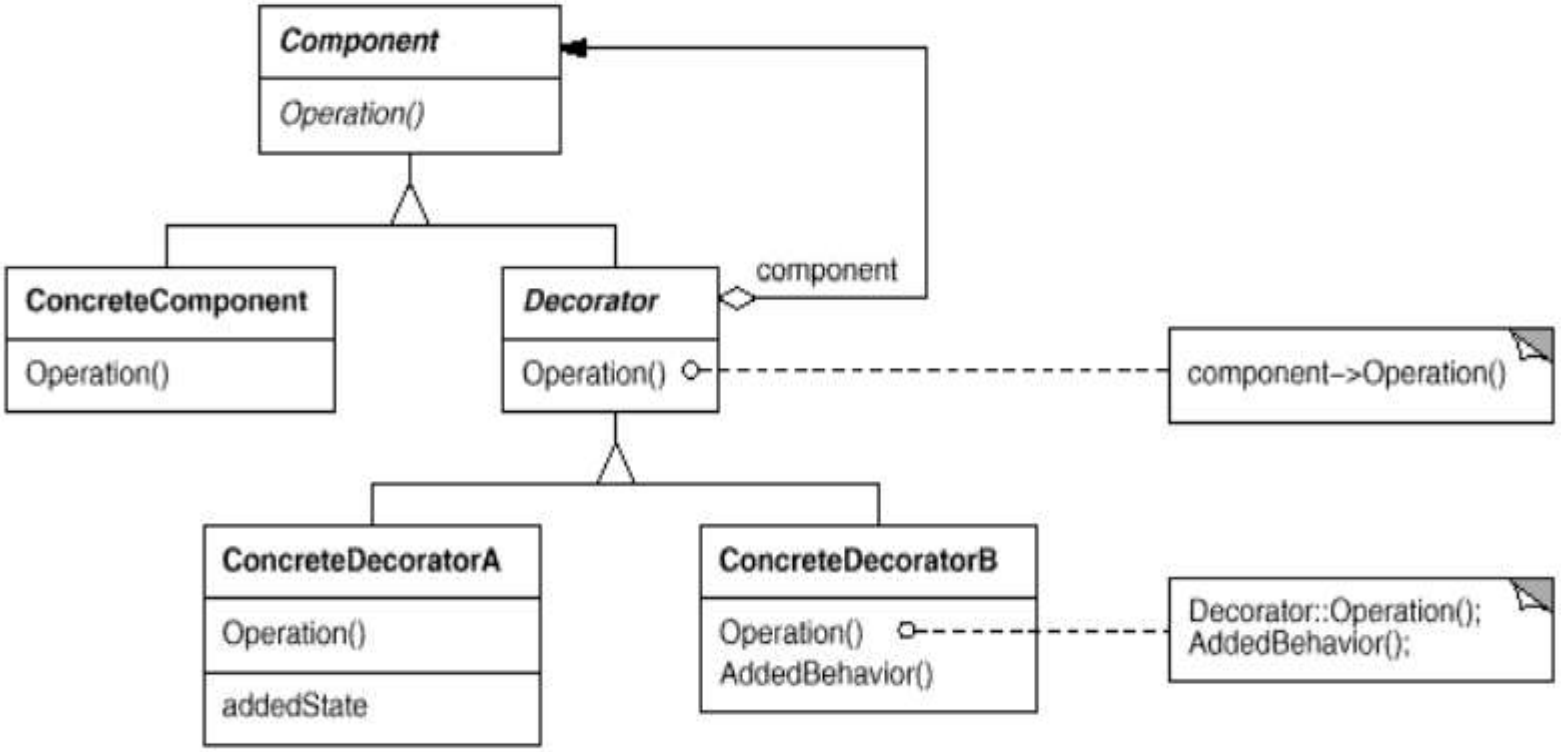
# DECORATOR DESIGN PATTERNS

Problem Statement:
An application has to create product as basic currency decorated in different ways as a USD or SGD as shown and print cost with new decoration as and when demanded by client.

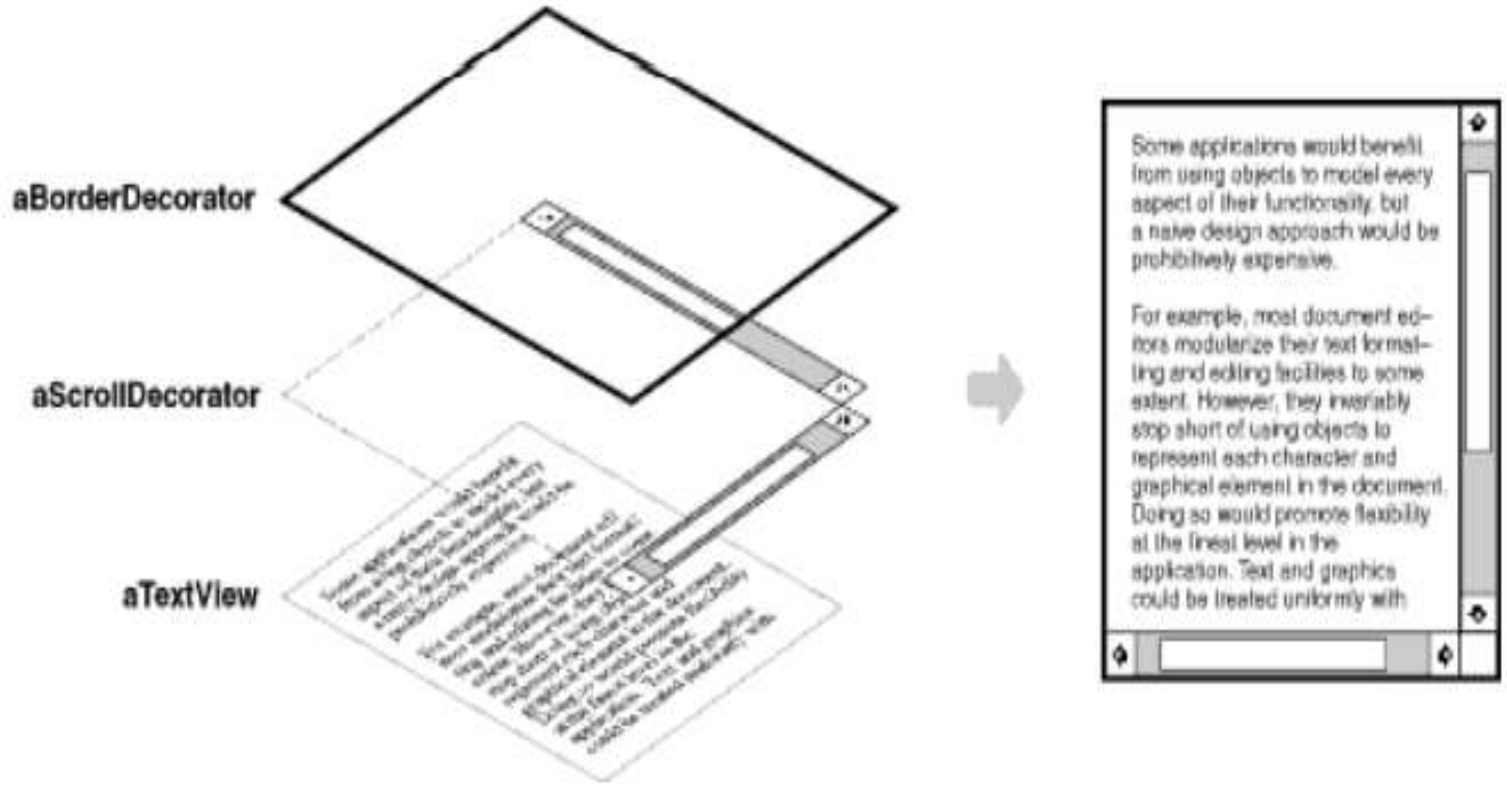


# DECORATOR DESIGN PATTERNS

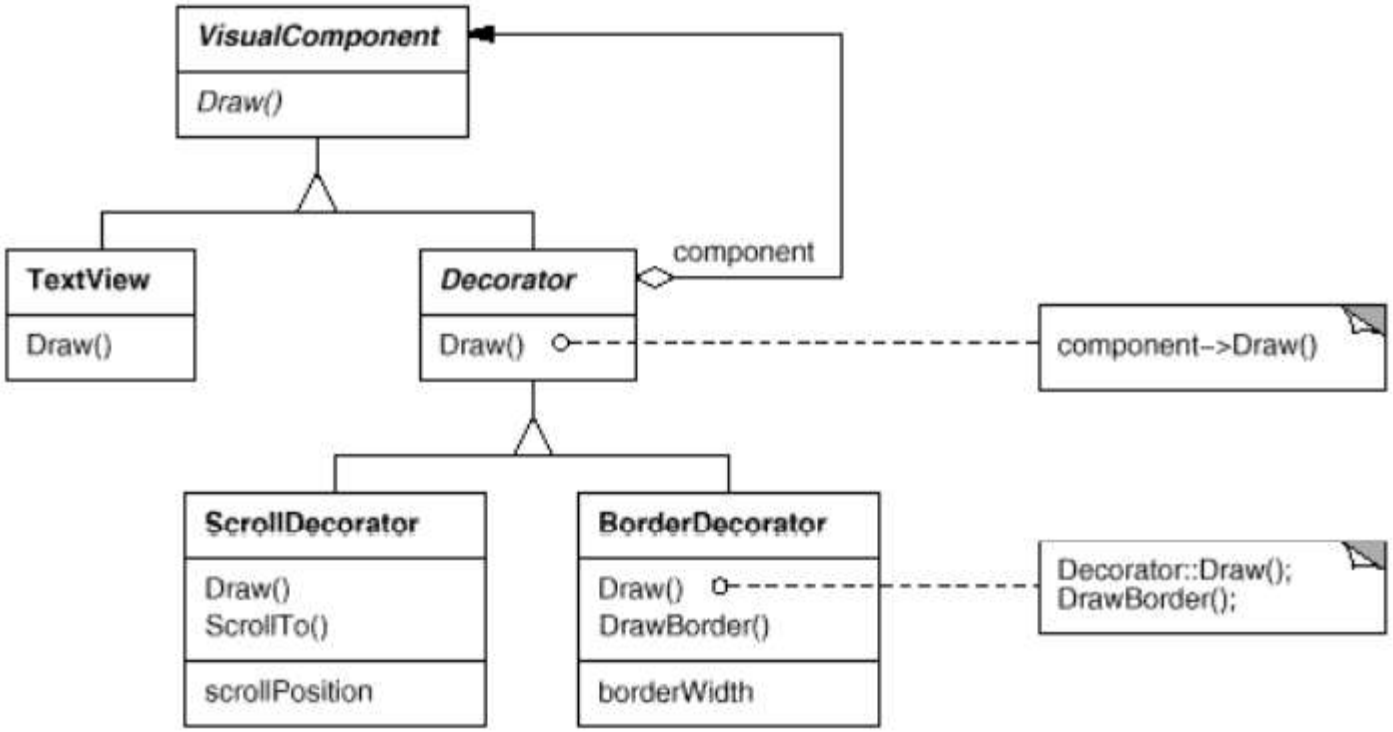
## Structure



# MOTIVATION




More example: Displaying same information in 2D, 3D, Black and white, colored bar charts




The important aspect of this pattern is that it lets decorators appear anywhere a VisualComponent can. That way clients generally can't tell the difference between a decorated component and an undecorated one, and so they don't depend at all on the decoration

# APPLICABILITY

Use the Decorator pattern when:

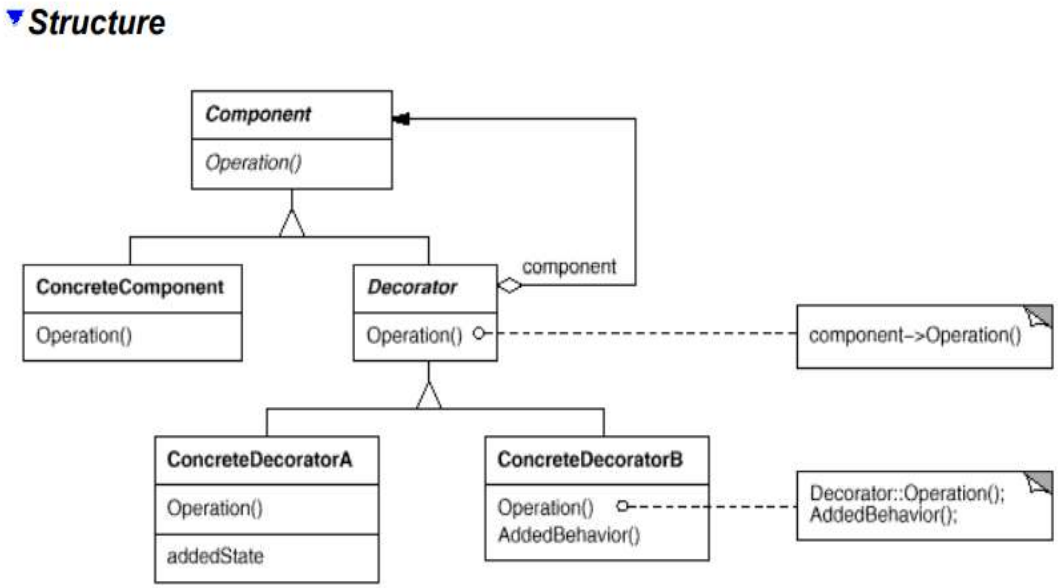
- to add responsibilities to individual objects dynamically and transparently, without affecting other objects.
- for responsibilities that can be withdrawn.
- when extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination.
- a class definition may be hidden or otherwise unavailable for subclassing.

# PARTICIPANTS

| Participant | Responsibility |
| --- | --- |
| **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. |

# COLLABORATIONS

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



# CONSEQUENCES

- More flexibility than static inheritance: **Inheritance** requires creating a new class for each additional responsibility. This gives rise to many classes and increases the complexity of a system. Providing different **Decorator** classes for a specific Component class lets you mix and match responsibilities. Decorators also make it easy to add a property twice. For example, to give a TextView a double border, simply attach two BorderDecorators. Inheriting from a Border class twice is error-prone at best.
- Avoids feature-laden classes high up in the hierarchy. Decorator offers a pay-as you-go approach to adding responsibilities. 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.

# CONSEQUENCES

- A decorator and its component aren't identical. A decorator acts as a transparent enclosure. But from an object identity point of view, a decorated component is not identical to the component itself. Hence you shouldn't rely on object identity 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, not in their class or in the value of their variables. Although these systems are easy to customize by those who understand them, they can be hard to learn and debug.

# IMPLEMENTATION

Consider the following issues when applying the Decorator pattern:

• **Interface conformance.** A decorator object's interface must conform to the interface of the component it decorates. ConcreteDecorator classes must therefore inherit from a common class.

• **Omitting the abstract Decorator class.** There's no need to define an abstract Decorator class when you only need to add one responsibility. That's often the case when you're dealing with an existing class hierarchy rather than designing a new one.

# IMPLEMENTATION

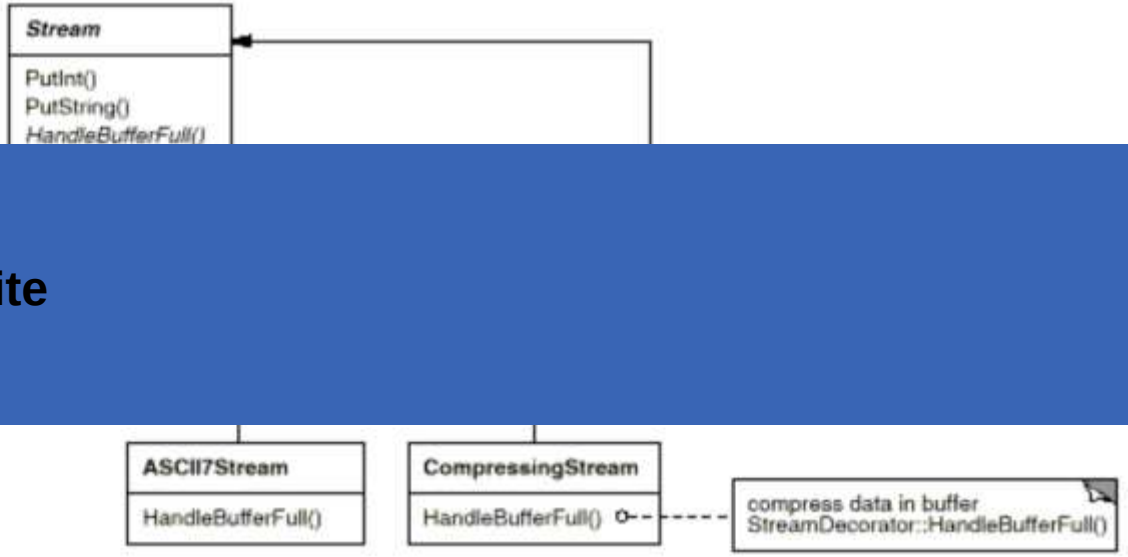
Consider the following issues when applying the Decorator pattern:

• **Keeping Component classes lightweight.** To ensure a conforming interface, components and decorators must descend from a common Component class. It's important to keep this common class lightweight; that is, it should focus on defining an interface, not on storing data.

• **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.

# KNOWN USES & RELATED PATTERNS

- CompressingStream subclass compresses the data, and the ASCII7Stream converts the data into 7-bit ASCII.
- Now, to create a FileStream that compresses its data and converts the compressed binary data to 7-bit ASCII, we decorate a FileStream with a CompressingStream and an ASCII7Stream:




- **Adapter**
- **Composite**
- **Strategy**