Decorator Pattern – Structural

• Intent
  » Attach additional responsibilities to an object dynamically
  » Provide a flexible alternative to sub-classing for extending functionality

• Also known as
  » Wrapper
Motivation

- Need to add responsibility to individual objects not to entire classes

  Add properties like border, scrolling, etc. to any user interface component as needed

- Enclose object within a decorator object for flexibility

  Nest recursively for unlimited customization
Example Text Decoration

- Compose a border decorator with a scroll decorator for text view.

```
a_border_decorator

component

a_scroll_decorator

component

a_text_view
```
Abstract Architecture

COMPONENT *
  method *

CONCRETE_COMPONENT +
  method+

DECORATOR *

CONCRETE_DECORATOR_A +
  other_method+
  method+

CONCRETE_DECORATOR_B +
  method+
  other_method+

pre_actions
component.method
post_actions

Recursive do
method in chain

© Gunnar Gotshalks
Participants

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

- Concrete component
  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

- Concrete decorator
  Add responsibilities to the component
Applicability

• Add responsibilities to individual objects dynamically and transparently
  
  **Without affecting other objects**

• For responsibilities that can be withdrawn

• When subclass extension is impractical
  
  **Sometimes a large number of independent extensions are possible**
  
  **Avoid combinatorial explosion**

  **Class definition may be hidden or otherwise unavailable for subclassing**
Benefits

• More flexible than static inheritance
  » Can add and remove responsibilities dynamically
  » Can handle combinatorial explosion of possibilities

• Avoids feature laden classes high up in the hierarchy
  » Pay as you go when adding responsibilities
  » Can support unforeseen features
  » Decorators are independent of the classes they decorate
  » Functionality is composed in simple pieces
Liabilities

• From object identity point of view, a decorated component is not identical
  » Decorator acts as a transparent enclosure
  » Cannot rely on object identity when using decorators

• Lots of little objects
  » Often result in systems composed of many look alike objects
  » Differ in the way they are interconnected, not in class or value of variables
  » Can be difficult to learn and debug
Why not use a collection class?

- A design using an array or linked list of the decorator class objects provides the same functionality

  » Client interface for the base object becomes more complex
    
    > More complex

  » Client becomes more specialized for the problem

    > Has to know the Decorator classes to be able to program the method operation with appropriate pre- and post-actions
Related Patterns

• Adapter changes interface to an object, while Decorator changes an object’s responsibilities

• Decorator is a degenerate Composite – only one component
  » But Decorator is not meant for object aggregation, only for added responsibility
    > Similar to the Chain of Responsibility pattern

• Strategy lets you change the internals of an object, while Decorator changes the exterior
Decorator in Java API

• Used in input classes
  » At base is an InputStream object such as System.in.
  » InputStreamReader decorates InputStream
  » BufferedReader in turn decorates InputStreamReader

inputObject =
  BufferedReader ( InputStreamReader ( System.in ) )