Decorator Pattern – Structural

• Intent
  » Attach additional responsibilities to an object dynamically.
  » Provide a flexible alternative to subclassing for extending functionality
Decorator – Motivation

• Motivation – Applicability
  
  » 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
• Compose a border decorator with a scroll decorator for text view.
Decorator – Example Diagram

VISUAL_COMPONENT *

draw *

TEXT_VIEW +
draw

DECORATOR *

component : VISUAL_COMPONENT

SCROLL_DECORATOR +
draw
scroll_position
scroll_to

BORDER_DECORATOR +
draw
border_width
draw_border
Decorator – General Structure

COMPONENT *
  method *

CONCRETE_COMPONENT +
  method

DECORATOR *
  component : COMPONENT

CONCRETE_DECORATOR_A +
  method
  other_feature

CONCRETE_DECORATOR_B +
  method
  another_feature
Decorator – Implementation

```plaintext
class COMPONENT feature method deferred end

class CONCRETE_COMP feature method is... end

class DECORATOR feature
    component : COMPONENT
end

class CONCRETE_DECORATOR feature
    method is
        pre_actions
        component.method
        post_actions
end
```

- Recursively do method for next in chain
Decorator – 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• For responsibilities that can be withdrawn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**
Decorator – 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
• 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
Decorator – Related Patterns

• Adapter changes interface to an object, while Decorator changes an objects responsibilities

• Decorator is a degenerate Composite (only one component)

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