Decorator Pattern

CSCI-4448 - Boese

University of Colorado **Boulder**

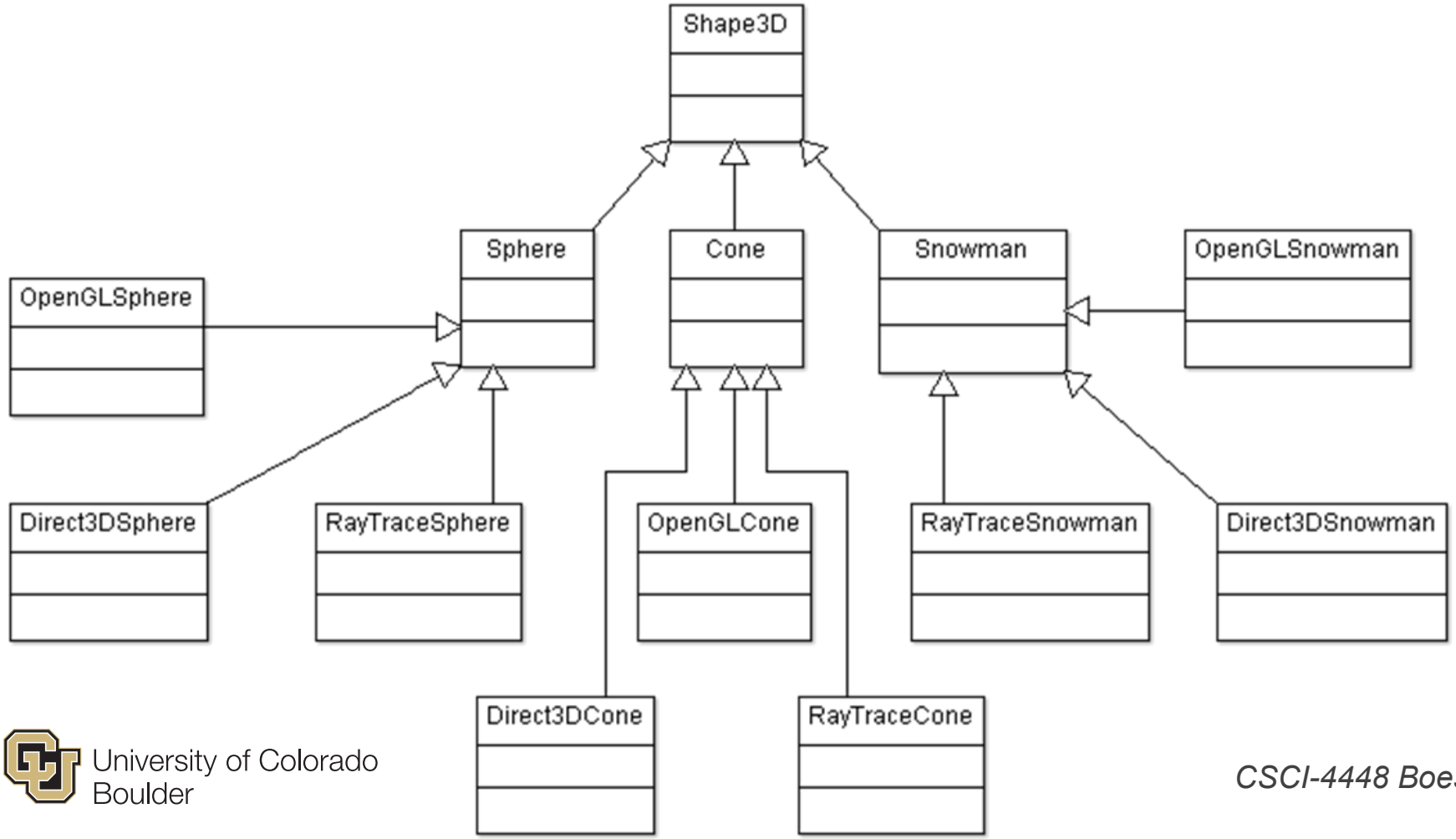
Objectives

- Problem
- Definition
- Why
- How
- Design Considerations

Problem

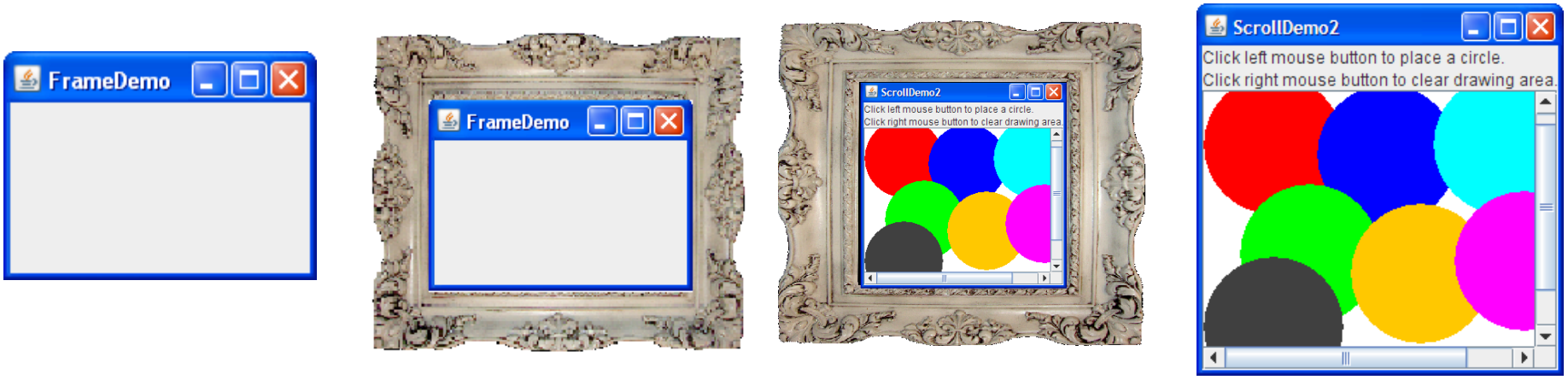
Why

- As we've seen before, subclassing can lead to classes with low cohesion, and *very* large class hierarchies.



Why

- Sometimes, subclassing makes sense
  - One type of object may be a *specialization* of another
  - Still can lead to large hierarchies
    - *Window, ScrollWindow, FancyBorderWindow, ColorFilterWindow, FancyBorderScrollWindow, ScrollColorFilterWindow, FancyBorderColorFilterWindow, FancyBorderScrollColorFilter…*



Why

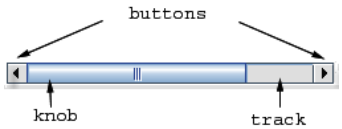
- In this case, the core window remains unchanged, but we're *adding* specific details

Why

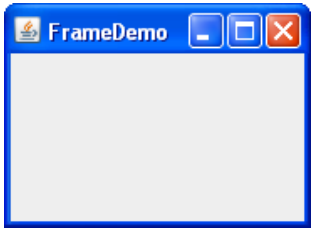
- Rather than creating several subclasses, it'd be preferable to have some common interface, and be able to compose what we want, where every piece follows the same interface

myWindow.draw() →

fancyFrame.draw() →



scrollBar.draw() →



basicFrame.draw()

Definition

Definition

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

-Gang of Four

Definition

- Pattern Name "**Decorator**"
  - Extended behavior or functionality build upon some existing substructure
- Intent
  - Dynamically add behavior to an object
    - *Inheritance adds behavior statically – once the particular subclass is instantiated, the behavior is fixed*

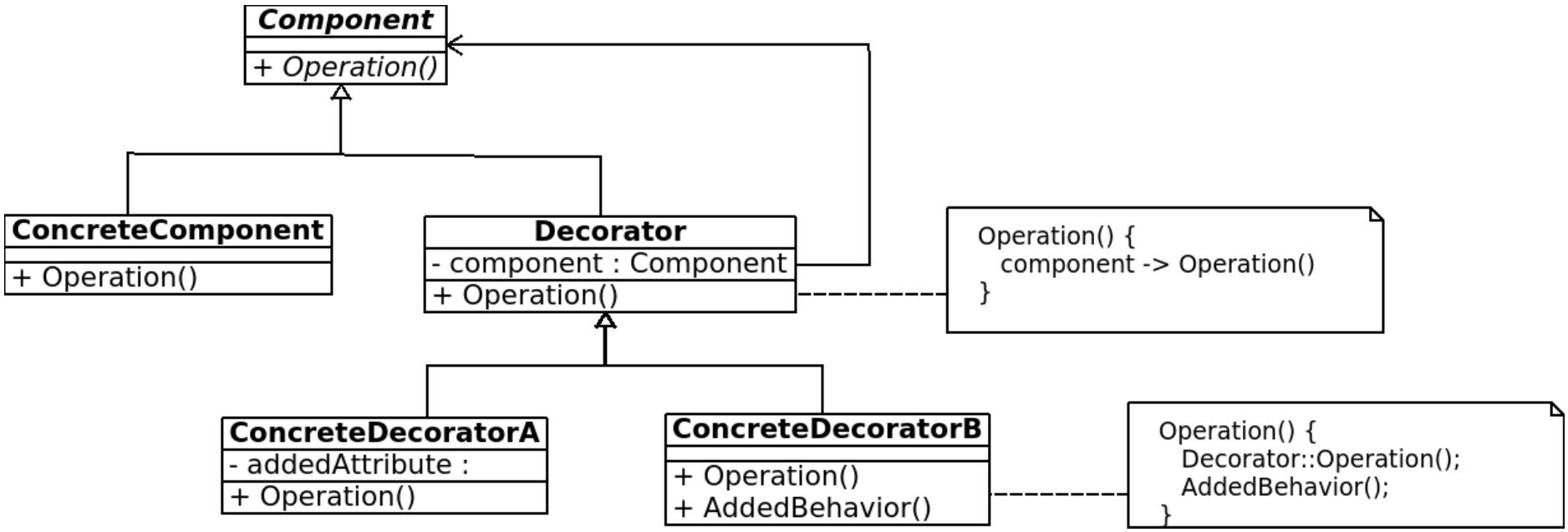
How

Decorator Pattern - Participants

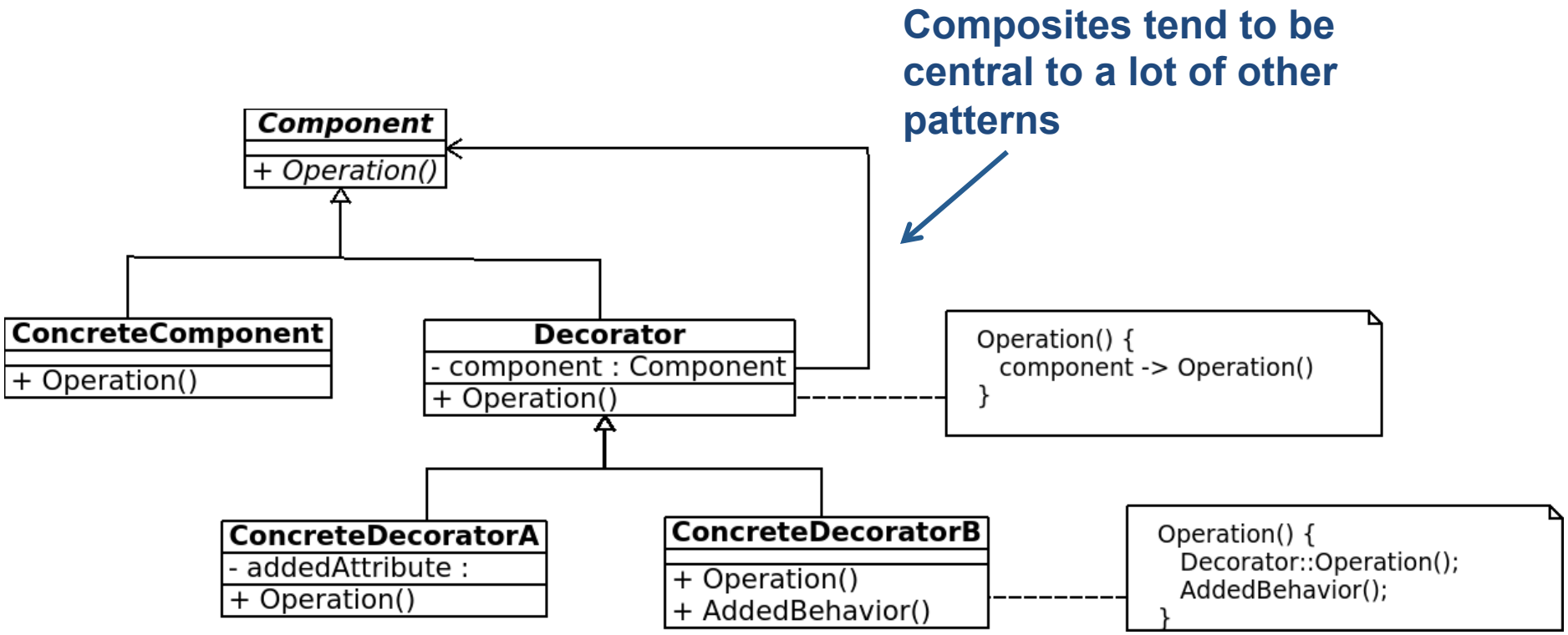
Participants

- **Component**
  - Defines the interface for objects that can have responsibilities added dynamically
- **ConcreteComponent**
  - Defines an object to which additional responsibilities can be attached
- **Decorator**
  - Maintains a reference to a Component object and defined an interface that conforms to Component's interface
- **ConcreteDecorator**
  - Adds responsibilities to the component

Decorator Pattern - Structure



Decorator Pattern - Structure



Decorator Pattern –Example

Window

+ draw()

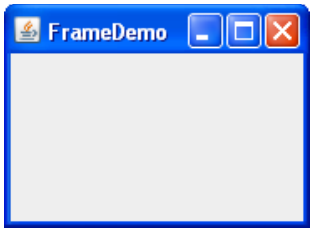
PlainWindow

+ draw()

Decorator

- baseWindow : Window

+ draw()



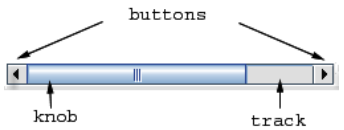
FancyBorder

+ draw()

ScrollBar

+ draw()

+ scroll()



Decorator Pattern –Example w/ Inheritance

Component

```
public abstract class Window
{
  public abstract void draw();
}
```

ConcreteComponent

```
public class PlainWindow extends Window
{
  public void draw() { … };
}
```

Decorator

```
public abstract class Decorator extends Window {
  private Window baseWindow;
  public Decorator(Window base) { baseWindow = base; }
  public void draw() { baseWindow.draw(); }
}
```

ConcreteDecorator

```
public class ScrollBar extends Decorator
{
  public ScrollBar(Window base)
    { super(base); }
  public void draw() {
    super.draw();
    // Extra drawing
  }
  public void scroll() {
    // added behavior
  }
}
```

ConcreteDecorator

```
public class FancyBorder extends
    Decorator
{
  public FancyBorder(Window base)
    {  super(base); }
  public void draw() {
    super.draw();
    // Extra drawing
  }
}
```

Decorator Example

- Can create a base instance of some plain window

```
Window myWindow = new PlainWindow();
```

- Then, add some decoration

```
Window oldWindow = myWindow;
Window myWindow = new FancyBorder(oldWindow);
```

- Modify the window to use another decoration

```
Window myOtherWindow = new ScrollBar(myWindow);
```

Design Considerations

Consequences

- More flexible than static inheritance
  - *Add responsibilities to an object by wrapping it in a decorator, vs. creating a new class for each added responsibility*
- Avoid feature-laden classes
  - *Create objects consisting of only the decorations you need, not objects with several inherited features, only some of which you use*
- Decorators and components are not identical
  - *A decorator is a different object, not a modification of the same object (e.g., as through delegation). Hence, don't rely on object identity with decorators*
- Lots of little moving parts
  - *Very flexible system, but hard to learn and debug*

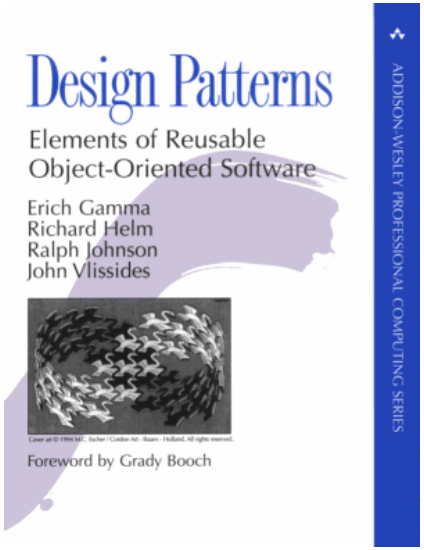
Implementation Considerations

- Interface conformance
  - *All Decorator and Component classes must share the same interface, though decorators can add to this interface*
- Omitting Decorator abstract class
  - *If only adding one responsibility, Decorator and ConcreteDecorator can be merged. Let ConcreteDecorator call Operation() on its component*
- Keep Component lightweight
  - *Decorator inherits from Component. If Component stores lots of data, then every decorator also stores this data, resulting in lots of memory being used*
  - *Component should just define a lightweight interface, not many implementation details*

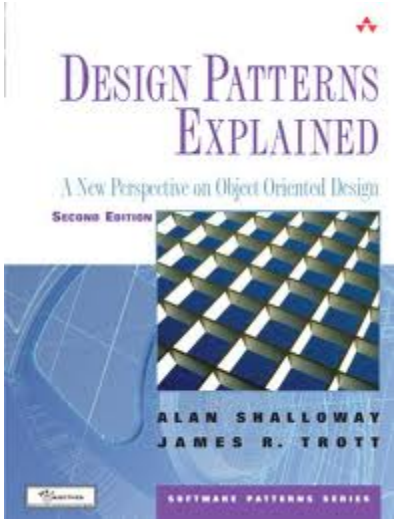
Pattern Comparison

	Decorator	Composite	Proxy
Recursive Composition	**YES**, but intent is to provide a means of adding responsibility in a recursive manner	**YES**, intent is to provide a recursive representation to a composed object where parts can be treated in the same manner as a whole	**NO,** proxy is usually a single object used to provide indirect access to another object
Provides a level of indirection to an object	**YES**, the intent is to *add* behavior to the base object	**NO**, the composed object can be accessed directly	**YES,** the intent is to act as a *stand-in* to the base object
Common interface	**YES,** common interface allows for behavior to be attached or detached dynamically	**YES,** common interface allows for clients to interact with any part of the composed object in a common way	**YES**, common interface allows proxy to be used as base object, but *intercept* and handle messages

Further Reading



- **Design Patterns**
  pp. 175 - 184



- **Design Patterns Explained**
  Chapter 17
  pp. 297-310