Software Engineering

Lesson Design Pattern
Decorator
v1.0

Uwe Gühl

Fall 2007/ 2008

Decorator

- Intent:
  - Decorators modify individual objects dynamically
  - So they offer a flexible alternative to hierarchies with subclassing to extend functionalities of objects
  - ... known as „Wrapper“ as well
  - ... is a Structural Pattern

Decorator

- Motivation
  Add to separate objects additional responsibilities – not to the entire class
- Typical example: GUI with additional properties like
  - additional border
  - additional scrolling

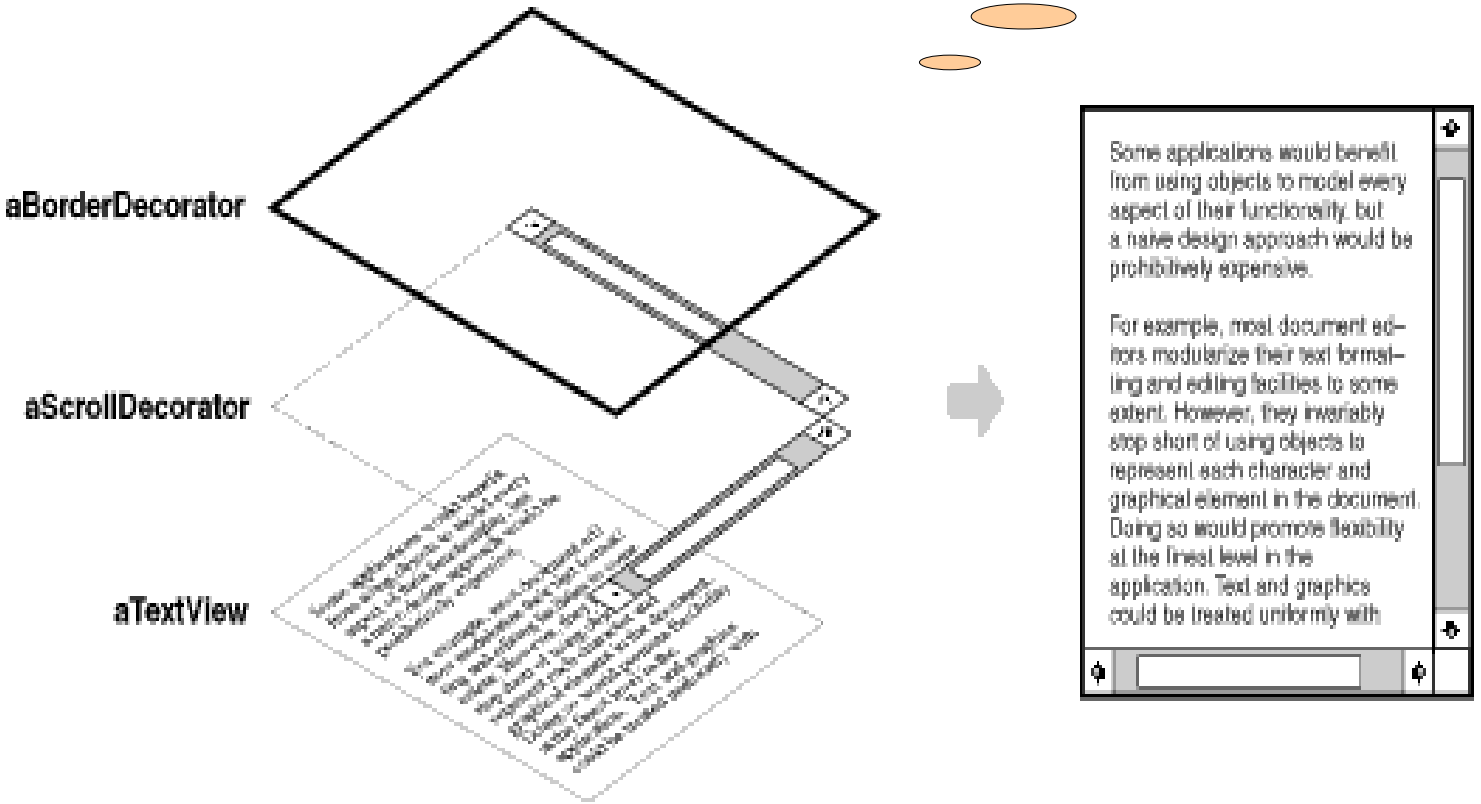
Decorator

- Ideas
  - Possible solution 1: Inheritance
    Problem: Inheriting a border means that every instance of a subclass has a border
    - Static approach not flexible
    - Client can not determine, how and when a border should be set on a component
  - Possible solution 2: Including the component in another object, that adds a border
    The including component is named ***Decorator***.

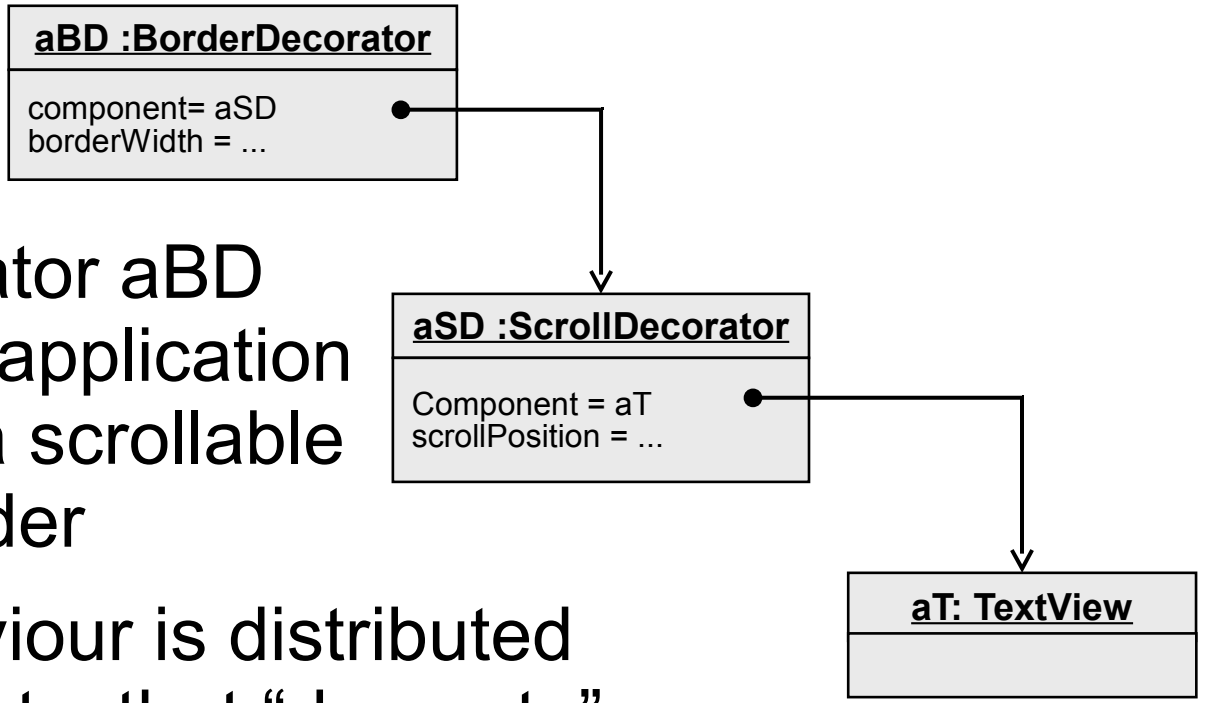
Decorator

- Example

How to realize?



Decorator



- Example
  - The BorderDecorator aBD behaves from the application point of view like a scrollable TextView with border
  - Actually the behaviour is distributed among three objects, that “decorate” the basic functionality
  - Typical use:

```
aT  = new TextView();
aSD = new ScrollDecorator(aT);
aBD = new BorderDecorator(aT);
aBD.draw();
```

Decorator

- Example
  - Alternative:

```
VisualComponent vC = new TextView();
vC = new ScrollDecorator(vC);
vC = new BorderDecorator(vC);
vC.draw();
```

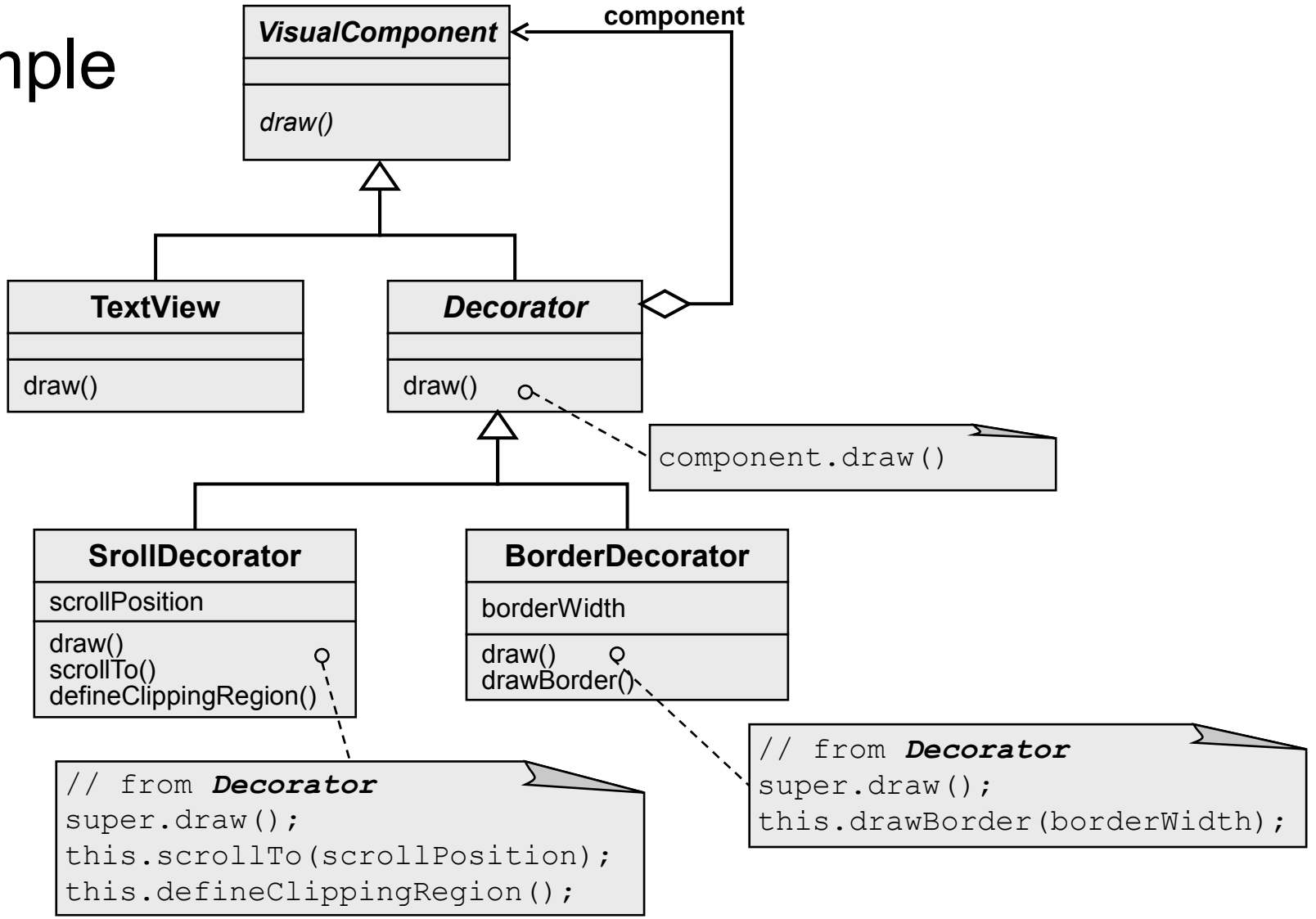
Decorator

- Consideration
  - To offer the same accessibility
    - the decorator and
    - the component

    need the same interface so that clients have transparent access
  - The decorator sends requests to the component and executes additional activities (e. g. drawing of a border)
  - Recursive use of multiple decorators allows dynamical adding of functionality

Decorator

- Example

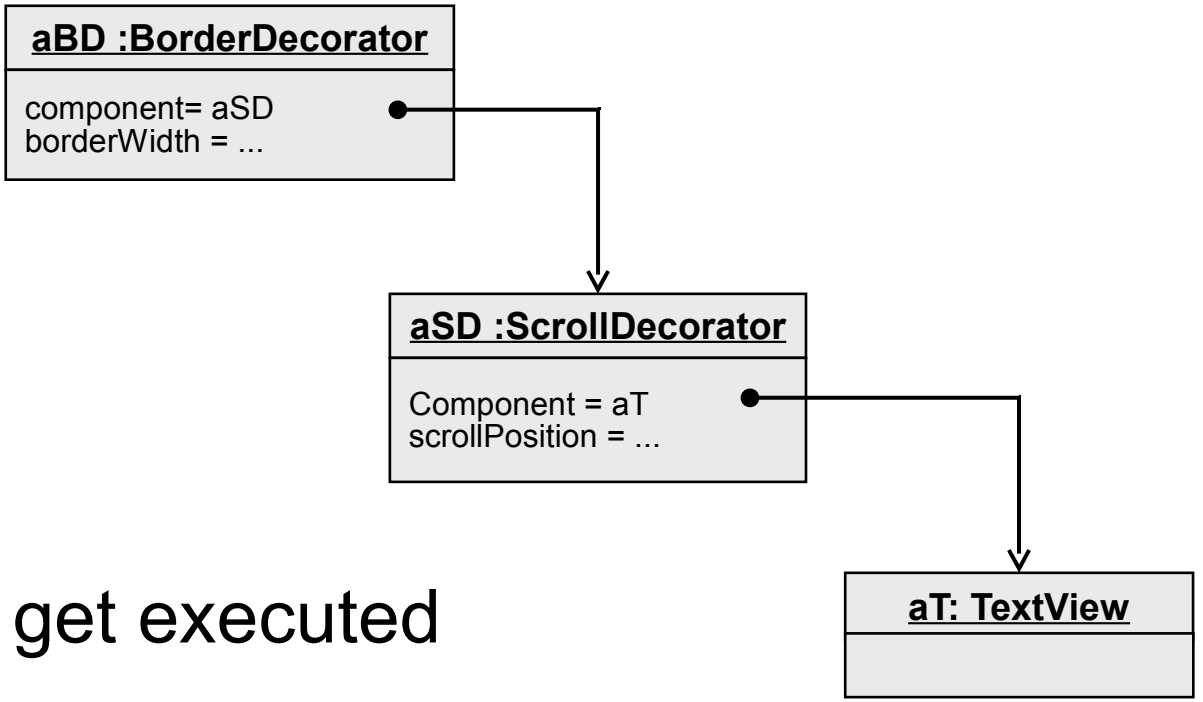


Decorator

- Example - continued
  - If called

```
aBD.draw();
```

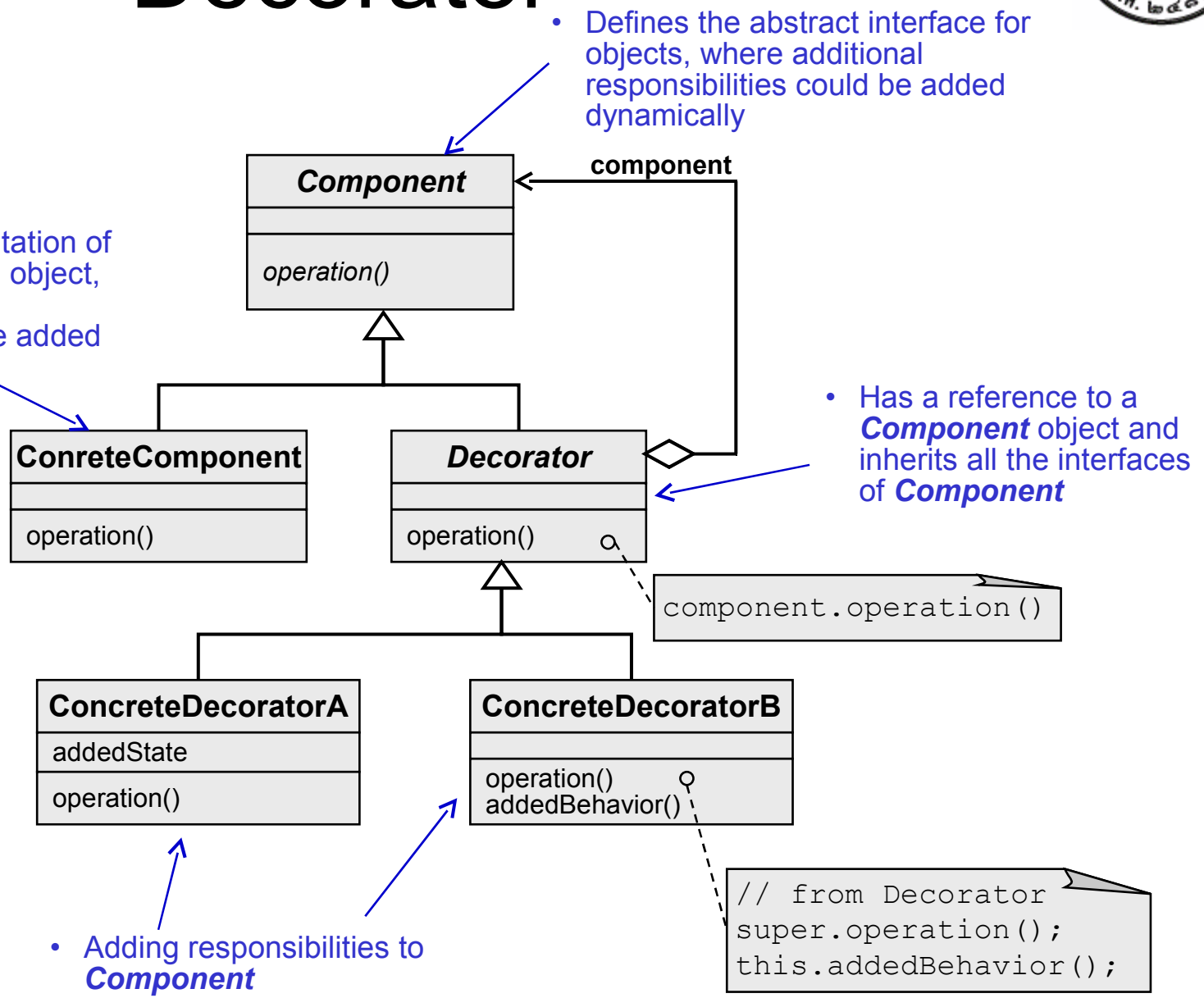
following methods get executed




```
BorderDecorator.draw();
Decorator.draw();
ScrollDecorator.draw();
Decorator.draw();
TextView.draw();
```

Decorator

- Structure



Decorator

- Collaboration
  - The ***Decorator*** forwards requests to its ***Component*** object
  - Optional additional operations could be added before or after forwarding the request

Decorator

- Applicability
  Use the Decorator Pattern
  - to add dynamically and transparent new responsibilities to specific objects
  - for responsibilities that can be withdrawn
  - when extension by subclassing is inefficient
    For example, if a big number of extensions is possible, what would produce a high numbers of subclasses to support every combination

Decorator

- Consequences

+ More flexibility than static inheritance
  - ***Decorator***s make it possible to add and remove at runtime responsibilities
  - With inheritance new classes would be needed for each additional responsibility (e. g. BorderedScrollableTextView, BorderedTextView, ...)
    - ⇨ more classes
    - ⇨ more complexity
    - ⇨ mixing of responsibilities
  - Properties could be used more often, for example a double border for a widget with 2 BorderDecorators

Decorator

- Consequences

+ Avoids feature laden classes high up in the hierarchy
  - Instead of using an „all-in-one device suitable for every purpose“ class a simple class is sufficient with the idea to add functionality incrementally with ***Decorator*** objects if needed
  - Adding functionality with the combination of simple pieces
  - Easy independent definitions of new ***Decorator***s

Decorator

- Consequences
  - A ***Decorator*** and its ***Component*** are not identical!
    - A ***Decorator*** acts like a transparent wrapper
    - Concerning the object identity: A decorated ***Component*** is not identical to the ***Component*** itself
      → Attention with referencing

Decorator

- Consequences
  - Many small objects
    - Result of using the ***Decorator*** is often that many little similar looking objects hang around in your system
    - The objects differ only in the way they are interconnected, not in their class and not in the values of their attributes
    - Hard to learn and debug – but easy to customize if you understand the context

Decorator

- Implementation
  - Interface conformance
    - The interfaces of ***Decorator*** and ***Component*** classes must be similar
    - **ConcreteDecorator** have to inherit from a common class (at least in C++)
    - A Component must not know anything about their Decorators, that's why a reference from a Component to a Decorator makes no sense

Decorator

- Implementation
  - Omitting the abstract ***Decorator***
    - It's possible to do without the abstract ***Decorator*** class, if only one responsibility should be added – e. g. if a class hierarchy already exists
    - The responsibility of the ***Decorator*** to forward requests to ***Component***s could be merged to **ConcreteDecorator**

Decorator

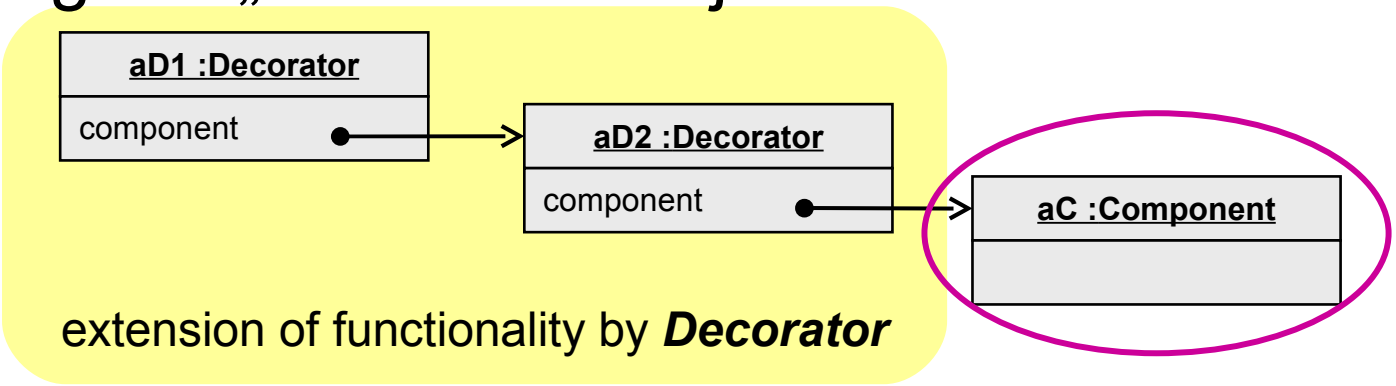
- Implementation
  - Keep ***Component*** classes lightweight
    - Goal: Simple interface of the ***Component***
      - All classes in this pattern inherit from the ***Component***
      - Risk: Subclasses contain functionality they do not need!
    - Job of the ***Component***: Definition of the interface not storage of data!
      → Storage of data should be done in subclasses

Decorator

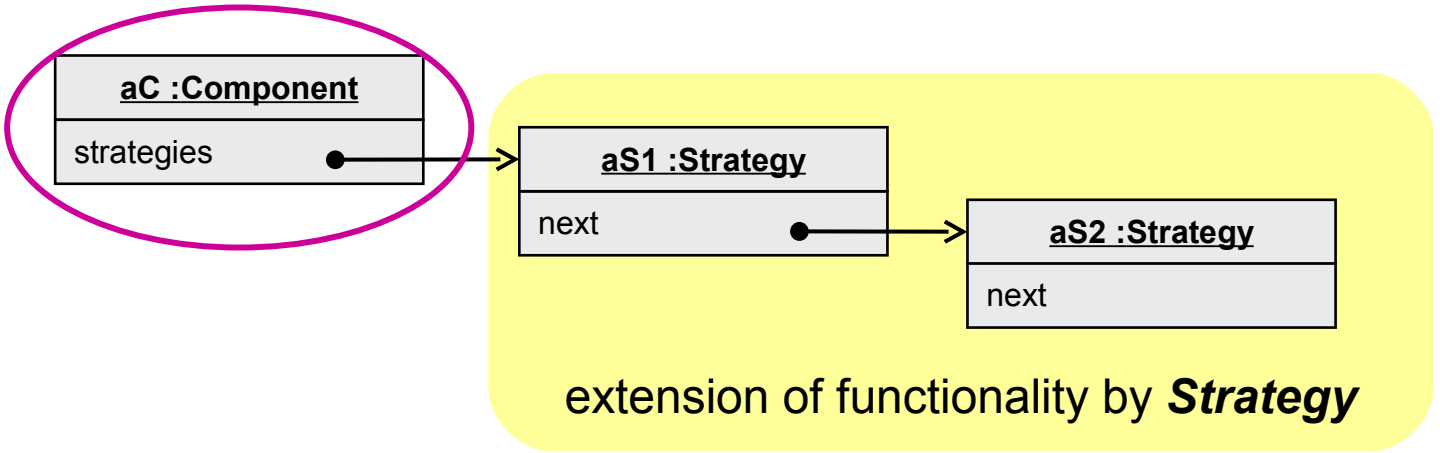
- Implementation
  - Changing the „skin“ of an object or changing the „guts“ of an object?
    - ***Decorator*** is an additional skin around an object that should change its behaviour
    - Alternative: Change of the inner parts (e. g. with the Strategy Pattern) – recommended, if the ***Component*** class tends to get too big and complex
    - Decide: When to use Decorator - when to use Strategy

Decorator

- Implementation
  - Changing the „skin“ of an object?



  - ... or changing the „guts“ of an object?

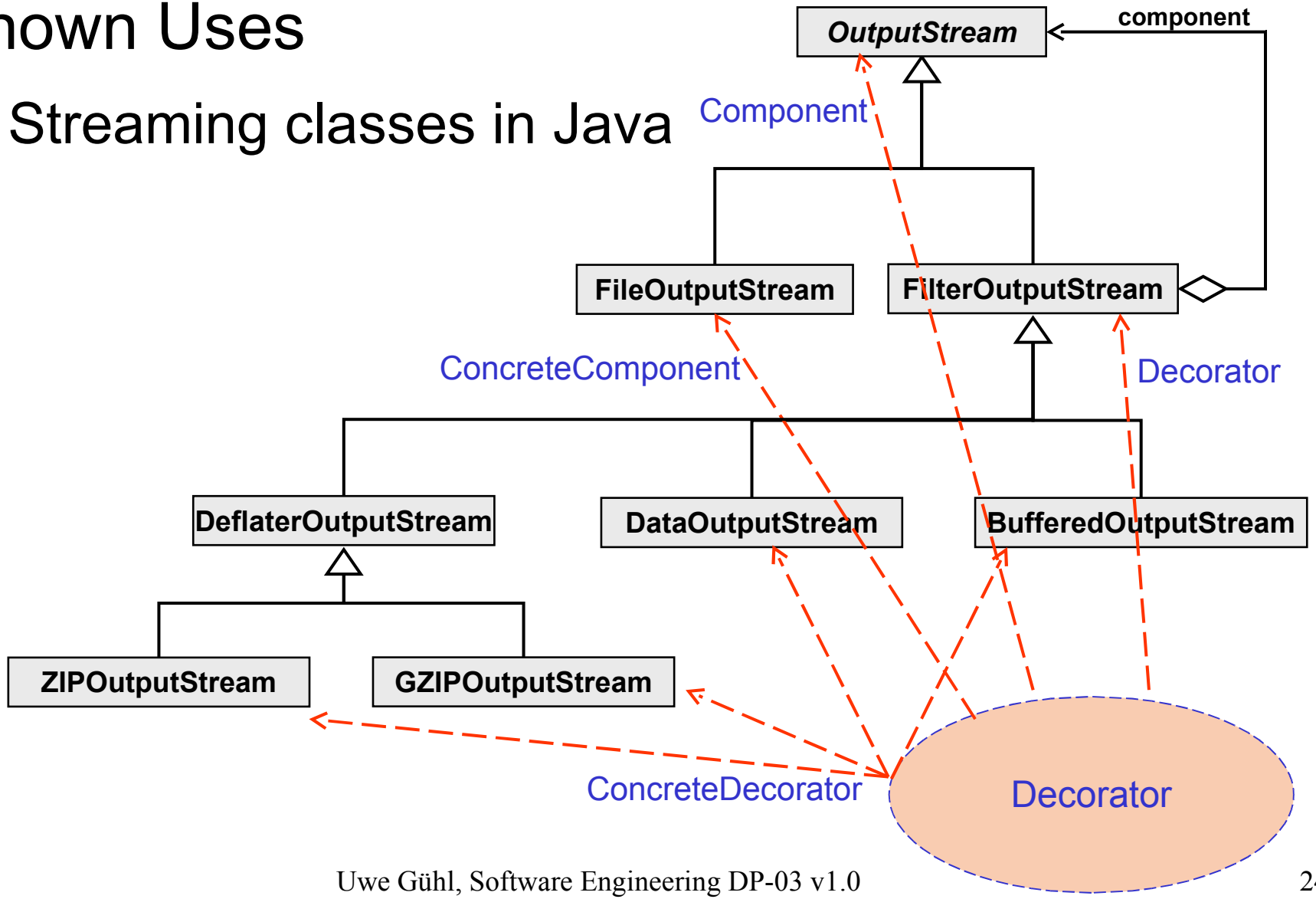


Decorator

- Known Uses (some examples out of [GHJ+95])
  - Many object oriented GUI toolkits use decorators to establish graphical utilities, especially graphical borders for widgets
    - Interviews
    - ET++
    - ObjectWorks\Smalltalk class library
    - HotDraw: DecoratorFigure
  - Non graphical examples
    - ET++ Streaming-Classes
    - Filter architecture in Struts Web application framework

Decorator

- Known Uses
  - Streaming classes in Java



Decorator

- Known Uses
  - More applications
    - Debugging Glyph from InterViews
      Debugging information before and after sending requests to components
    - PassivityWrapper from VisualWorks \ Smalltalk
      Possibility to control user activities to a component – either it is allowed or disabled

Decorator

- Related Patterns
  - Adapter
    - A decorator changes responsibilities of an object, where an adapter will give an object a new interface
  - Composite
    - A decorator is something like a degenerated composite with only one component
    - But a decorator is not intended for object aggregation
  - Strategy
    - A decorator changes the outer part of an object
    - A strategy changes the inner part of an object