Design Patterns (cont)

Softwaretechnik

UNI FREIBURG

Albert-Ludwigs-Universität Freiburg

Matthias Keil
Institute for Computer Science
Faculty of Engineering
University of Freiburg

16. Mai 2013

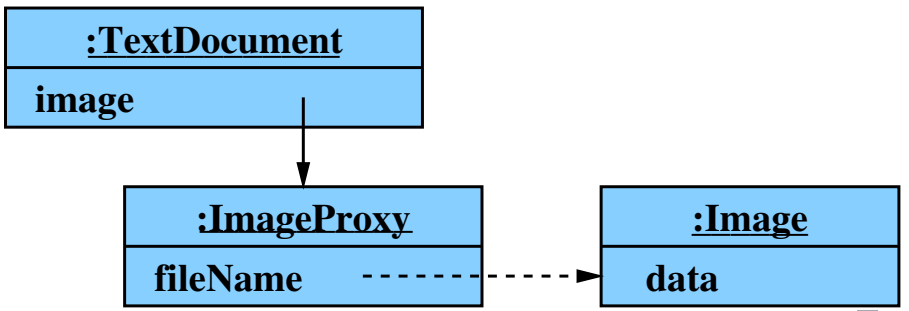
- Intent
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

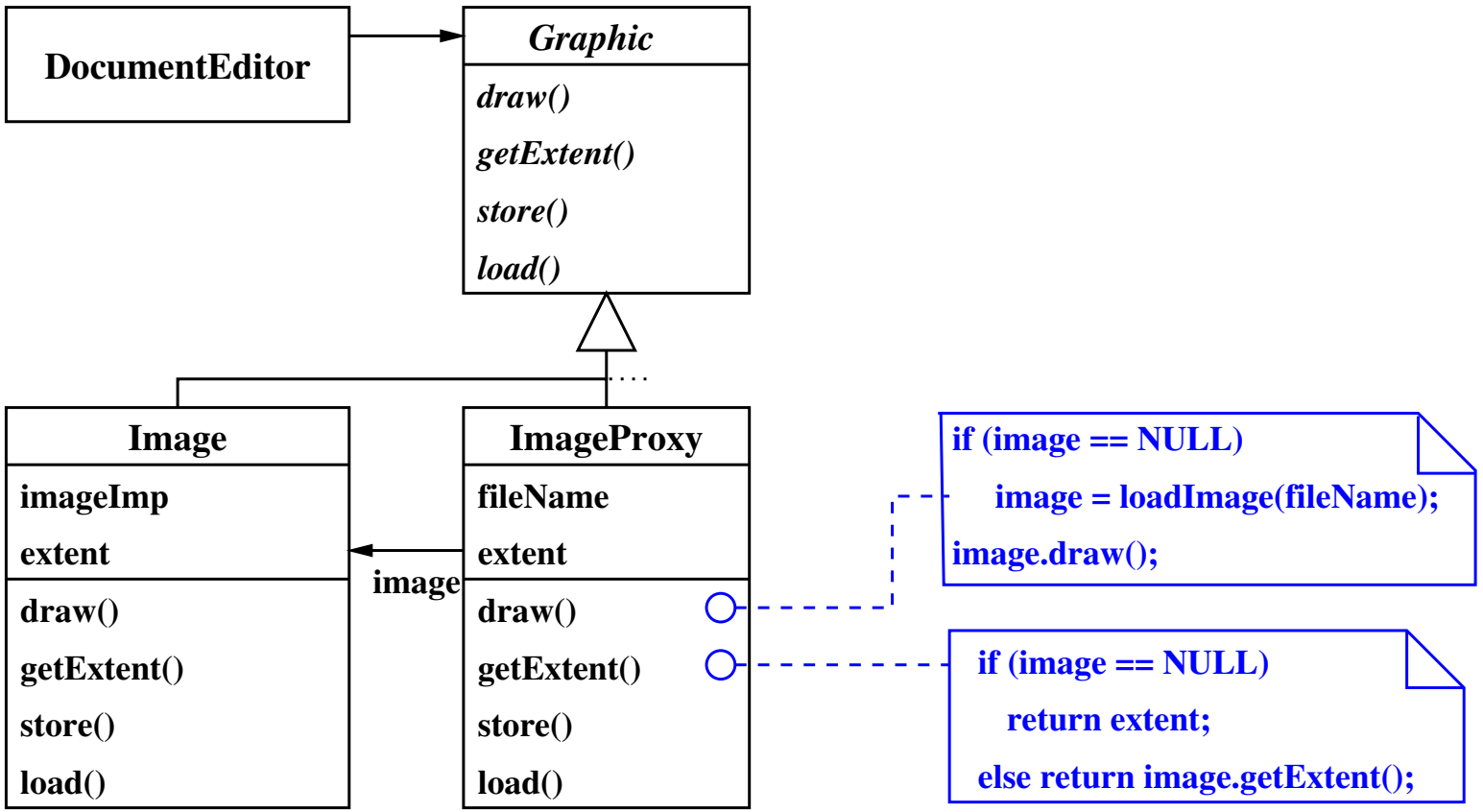
Intent

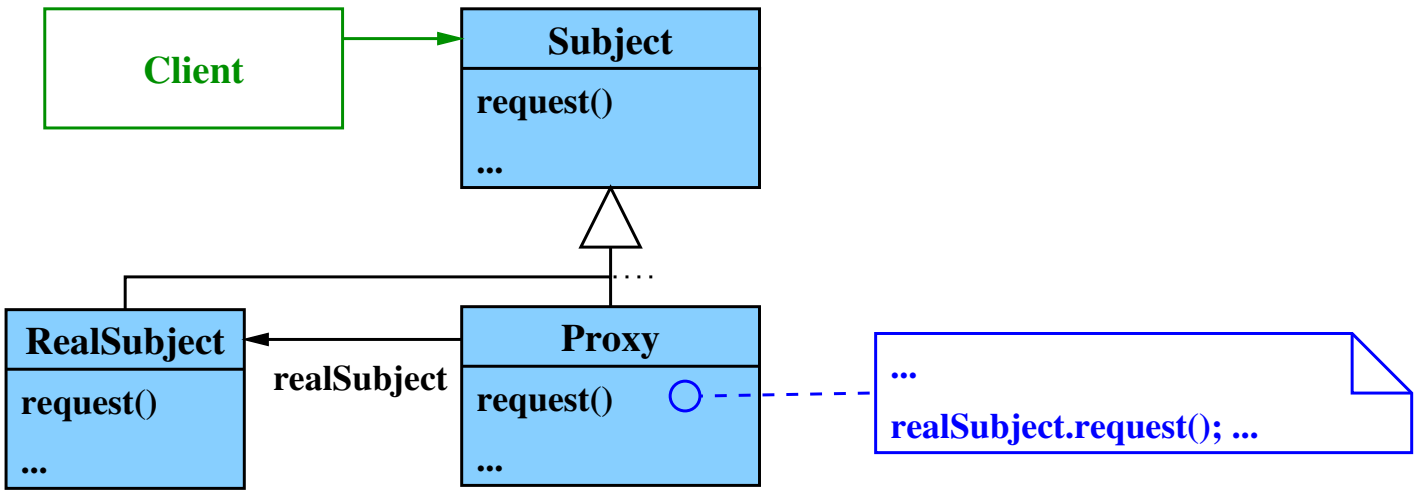
- Control access to object

Motivation

- Multi-media editor loads images, audio clips, videos etc on demand
- Represented by proxy in document
- Proxy loads the “real object” on demand




DocumentEditor
Graphic
draw()
getExtent()
store()
load()
Image
imageImp
extent
draw()
getExtent()
store()
load()
image
ImageProxy
fileName
extent
draw()
getExtent()
store()
load()
if (image == NULL)
image = loadImage(fileName);
image.draw();
if (image == NULL)
return extent;
else return image.getExtent();

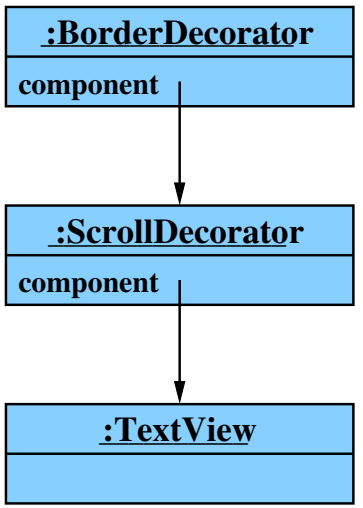

Client
Subject
request()
...
RealSubject
request()
...
realSubject
Proxy
request()
...
...
realSubject.request(); ...

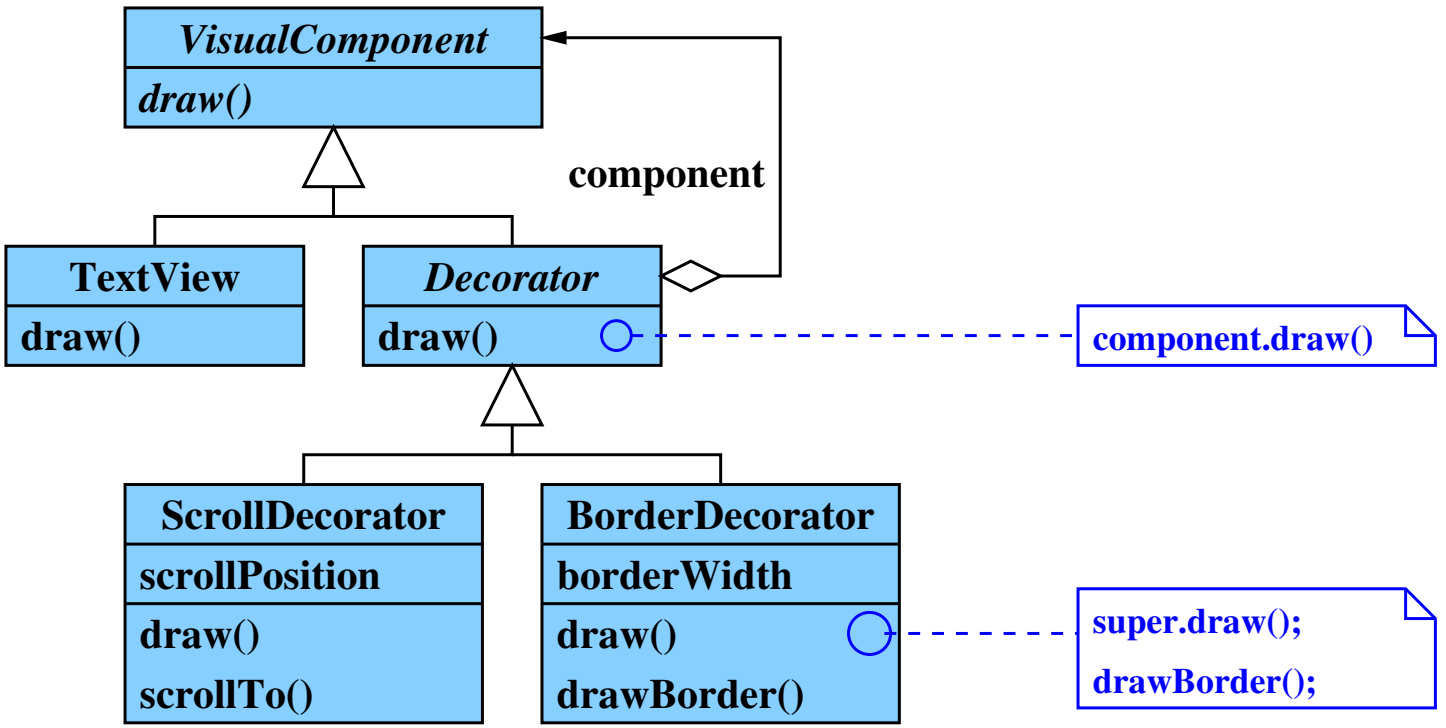
1. *Remote Proxy* Communication with object on server (CORBA)
2. *Virtual Proxy*
   - Creates expensive objects on demand
   - Delays cost of creation and initialization
3. *Protection Proxy* controls access permission to original object
4. *Smart Reference* additional operations: reference counting, locking, copy-on-write

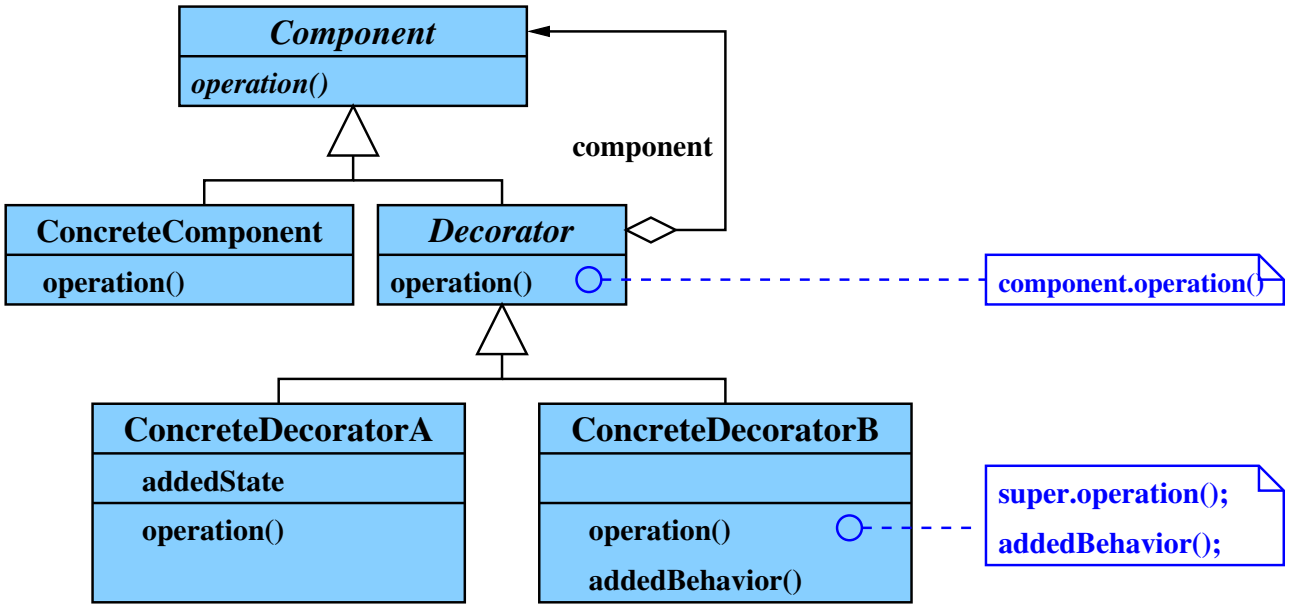
Intent

- Extend object's functionality dynamically
- More flexible than inheritance

- Graphical object can be equipped with border and/or scroll bar
- Decorator object has same interface as the decorated object
- Decorated forwards requests
- Recursive decoration




VisualComponent
draw()
component
TextView
draw()
Decorator
draw()
component.draw()
ScrollDecorator
scrollPosition
draw()
scrollTo()
BorderDecorator
borderWidth
draw()
drawBorder()
super.draw();
drawBorder();


Component
operation()
component
ConcreteComponent
operation()
Decorator
operation()
component.operation()
ConcreteDecoratorA
addedState
operation()
ConcreteDecoratorB
operation()
addedBehavior()
super.operation();
addedBehavior();

- More flexible than inheritance
- Avoids feature-laden classes high up in the hierarchy
- Decorator $\neq$ component
- Lots of little objects $\rightarrow$ hard to learn and debug

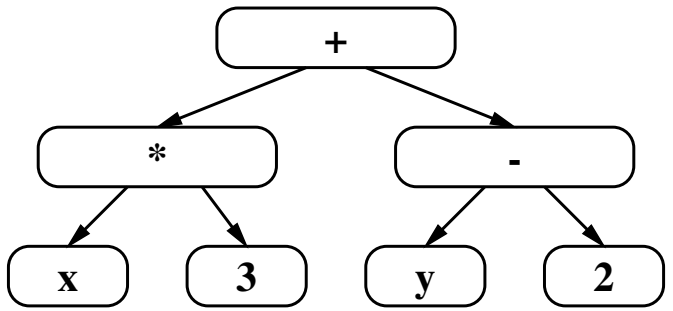
Applicability

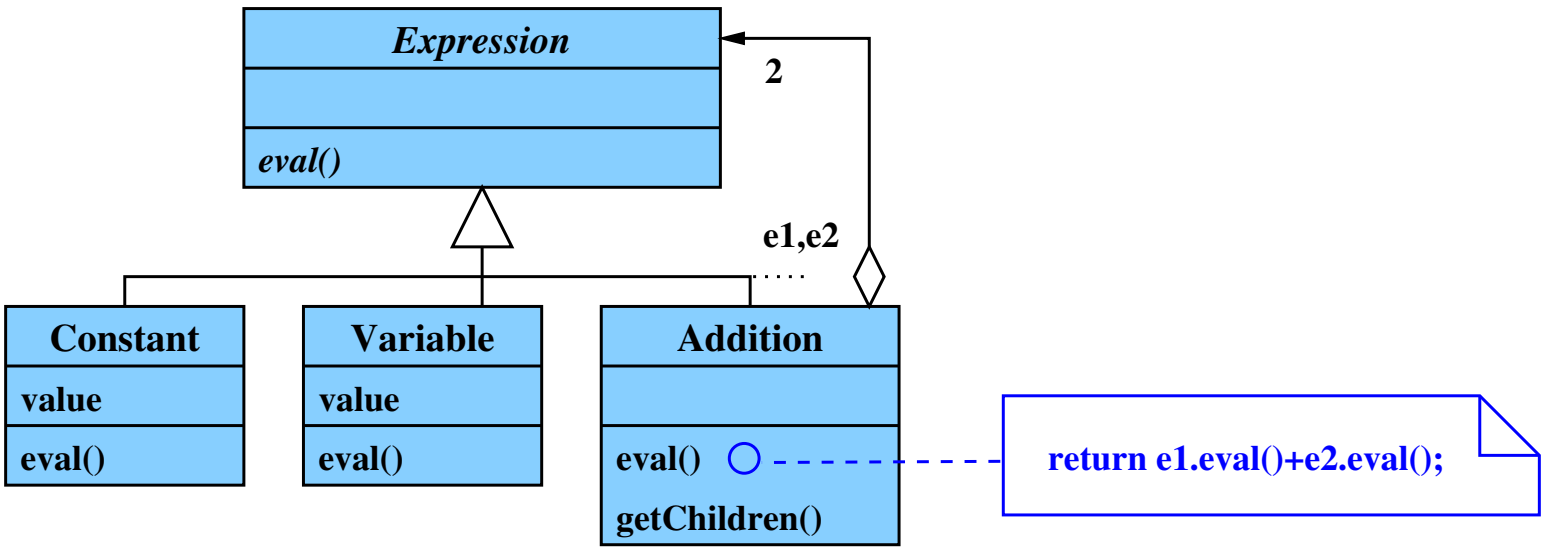
- Dynamically add responsibilities to individual objects
- For withdrawable responsibilities
- When extension by inheritance is impractical

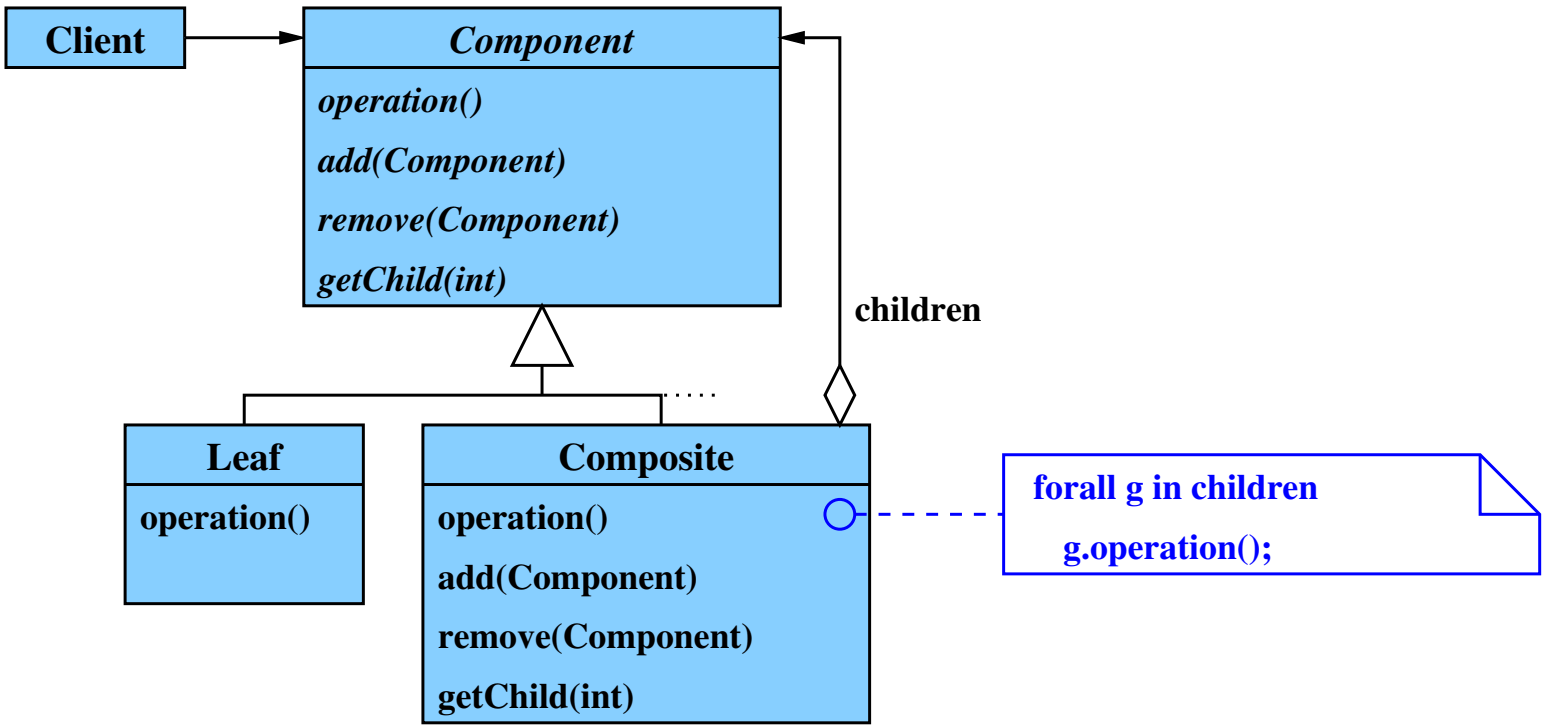
Intent

- Recursive object structures
- Uniform treatment of leaf components and containers

- arithmetic expression consists of subexpressions evaluation follows tree structure






Client
Component
operation()
add(Component)
remove(Component)
getChild(int)
children
Leaf
operation()
Composite
operation()
add(Component)
remove(Component)
getChild(int)
forall g in children
g.operation();

- Uniform client code
- Easy to add new composite classes as well as leaf classes

Applicability

- Recursive object structures

Related Patterns

- Decorator

- similar underlying concepts:
  - class-based $\rightarrow$ inheritance
  - object-based $\rightarrow$ object composition
- different goals

Adapter vs. Bridge vs. Facade

- all: flexibility through indirection
- diffeirences

Adapter:	reconciling differences between existing interfaces
Facade:	bundling of interfaces
Bridge:	interface with multiple, dynamically exchangeable implementations

Composite vs. Decorator

- both: recursive composition to organize open-ended number of objects
- Decorator adds responsibilities without subclassing
- Composite enables uniform processing of object graphs
- complementary $\rightarrow$ often used in concert

Decorator vs. Proxy

- both: indirection and forwarding
- Proxiy:
  - controls access to particular object
  - not recursive
- Decoratior:
  - stepwise addition of responsibilities
  - recursive