Chapter 4
Simple Patterns for Extensibility

Prof. Dr. U. Aßmann

Chair for Software Engineering

Facultät Informatik

Technische Universität Dresden

Version 11-0.1, 10/21/11

1) Object Recursion
2) Composite
3) Decorator
4) Chain of Responsibility
5) Proxy
6) *-Bridge
7) Observer

Literature (To Be Read)

► On Composite, Visitor: T. Panas. Design Patterns, A Quick Introduction. Paper in Design Pattern seminar, IDA, 2001. See home page of course.

► Gamma: Composite, Decorator, ChainOfResponsibility, Bridge, Visitor, Observer, Proxy

► J. Smith, D. Stotts. Elemental Design Patterns. A Link Between Architecture and Object Semantics. March 2002. TR02-011, Dpt. Of Computer Science, Univ. of North Carolina at Chapel Hill, www.citeseer.org

---

Goal

► Understanding extensibility patterns
  - ObjectRecursion vs TemplateMethod, Objectifier (and Strategy)
  - Decorator vs Proxy vs Composite vs ChainOfResponsibility
► Parallel class hierarchies as implementation of facets
  - Bridge
  - Visitor
  - Observer (EventBridge)
► Understand facets as non-partitioned subset hierarchies
► Layered frameworks as a means to structure large systems, based on facets

---

Static and Dynamic Extensibility

Variability vs Extensibility

- ▶ Variability so far meant
  - ▪ Static extensibility, e.g., new subclasses
  - ▪ Often, dynamic *exchangability* (polymorphism)
  - ▪ But not dynamic extensibility
- ▶ Now, we will turn to patterns that allow for dynamic extensibility
  - ▪ Most of these patterns contain a 1:n-aggregation that is extended at runtime

Binding a hook with
a hook value

Extending a hook with
another hook value
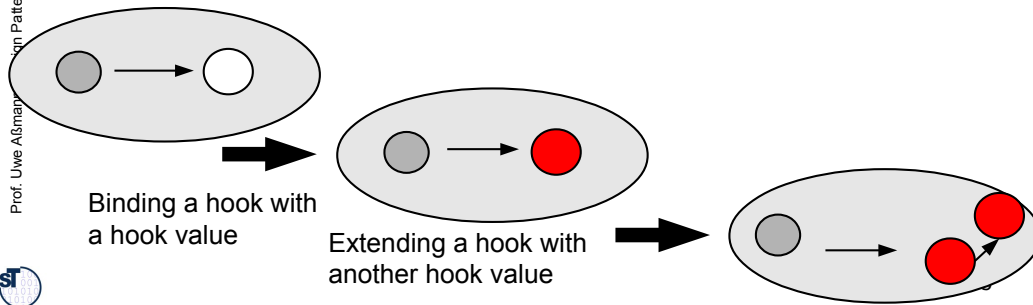
Software Cocktail Mixers

6
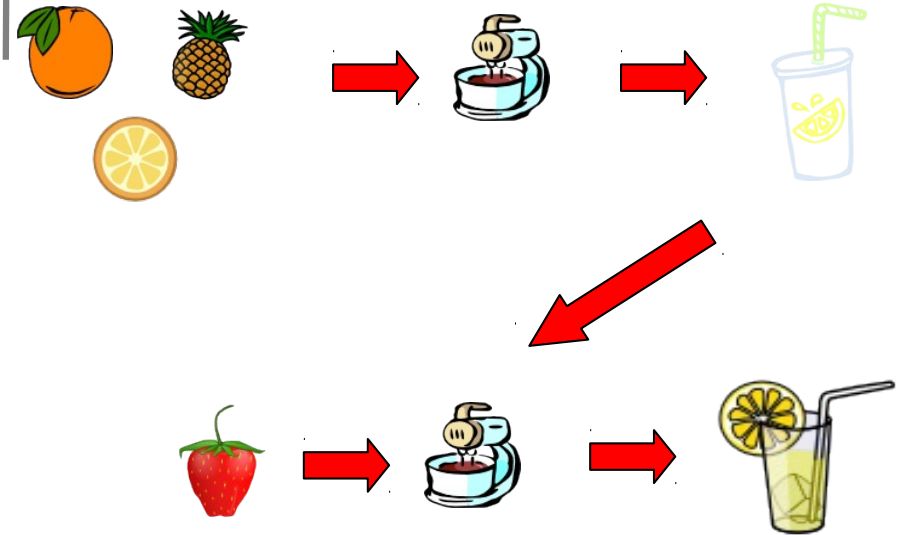
3.1 Object Recursion Pattern

7

Object Recursion

- ▶ Similar to the TemplateMethod, Objectifier and Strategy
- ▶ But now, we allow for *recursion* in the dependencies between the classes (going via inheritance and aggregation)
- ▶ The aggregation can be 1:1 or 1:n (1-Recursion, n-Recursion)

```
Client  ──────▷  Handler                          1 or +
                                                  childObject(s)
                 handleRequest()
                 preHandleRequest(Component)
                 postHandleRequest(Component)
                         △
            ┌────────────┴────────────┐
   Terminator                Recurser
   handleRequest()           handleRequest()          preHandleRequest()
                             preHandleRequest(Component)  for all g in childObject(s)
                             postHandleRequest(Component)   g.handleRequest()
                                                         postHandleRequest()
```

8
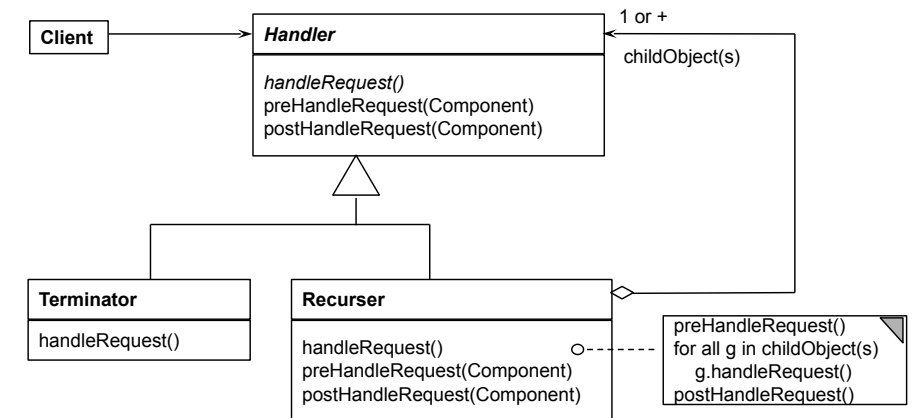
Incentive

- ▶ ObjectRecursion is a simple (sub)pattern
  - ▪ in which an abstract superclass specifies common conditions for two kinds of subclasses, the Terminator and the Recurser (a simple *contract*)
- ▶ Since both fulfil the common condition, they can be treated uniformly under one interface of the abstract superclass

9

Object Recursion – Runtime Structure

- ▶ 1-ObjectRecursion creates lists
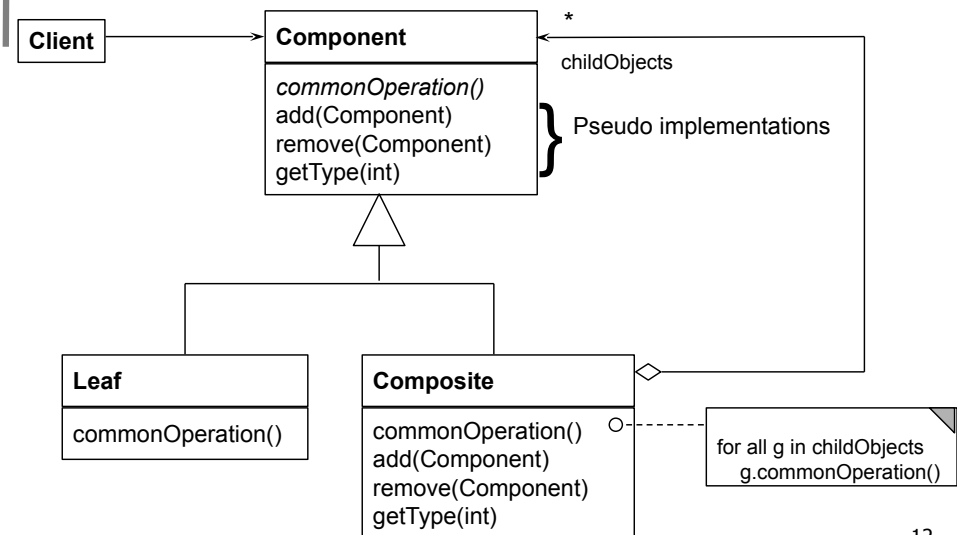- ▶ n-ObjectRecursion creates trees and graphs
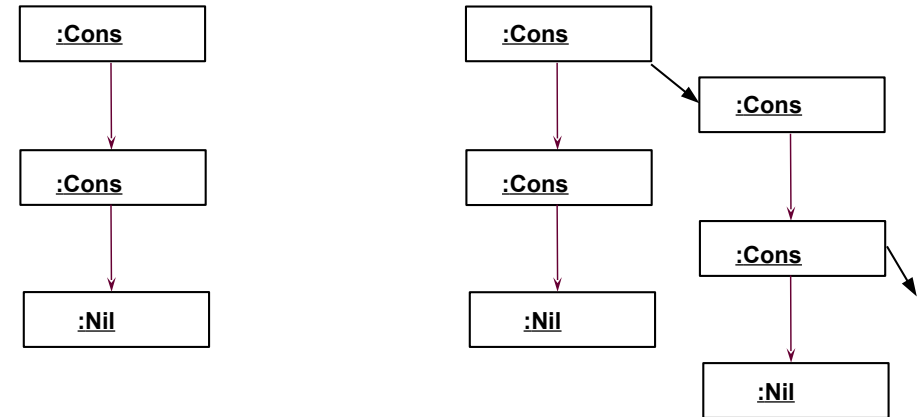
The recursion allows for building up runtime nets

10

3.2 Composite

11

Structure Composite

- ▶ Composite can be seen as instance of n-ObjectRecursion


```
Client → Component                    * childObjects
         commonOperation()
         add(Component)          } Pseudo implementations
         remove(Component)
         getType(int)

              △
       ┌──────┴──────┐
     Leaf          Composite
     commonOperation()   commonOperation()
                         add(Component)
                         remove(Component)
                         getType(int)
```

for all g in childObjects
g.commonOperation()

12

Piece Lists in Production Data

```
abstract class CarPart {
    int myCost;
    abstract int calculateCost();
}
class ComposedCarPart extends CarPart {
    int myCost = 5;
    CarPart [] children;  // here is the n-
                          recursion
    int calculateCost() {
      for (i = 0; i <= children.length; i++) {
        curCost += children[i].calculateCost();
      }
      return curCost + myCost;
    }
    void addPart(CarPart c) {
        children[children.length] = c;
    }
}
```
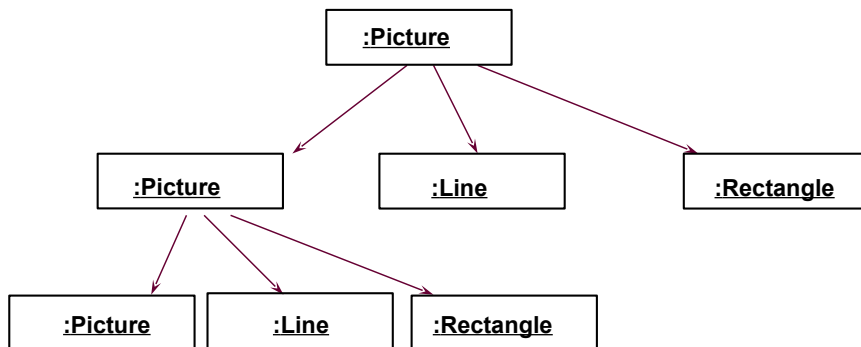
```
class Screw extends CarPart {
    int myCost = 10;
    int calculateCost() {
        return  myCost;
    }
    void addPart(CarPart c) {
        /// impossible, dont do anything
    }
}

// application
int cost = carPart.calculateCost();
```

13

Purpose

► The Composite is older as ObjectRecursion, from GOF
  ▪ ObjectRecursion is a little more abstract
► As in ObjectRecursion, an abstract superclass specifies a contract for two kinds of subclasses
  ▪ Since both fulfil the common condition, they can be treated uniformly under one interface of the abstract superclass
► Good method for building up trees and iterating over them
  ▪ The iterator need not know whether it works on a leaf or inner node. It can treat all nodes uniformly for
    · Iterator algorithms (map)
    · Folding algorithms (folding a tree with a scalar function)
► The Composite's secret is whether a leaf or inner node is worked on
► The Composite's secret is which subclass is worked on

14

Composite Run-Time Structure

► Part/Whole hierarchies, e.g., nested graphic objects

```
                          :Picture
            ┌────────────────┼──────────────┐
            ▼                ▼               ▼
        :Picture          :Line         :Rectangle
      ┌────┴────┐
      ▼         ▼          ▼
  :Picture   :Line    :Rectangle
```

common operations: draw(), move(), delete(), scale()

15

Dynamic Extensibility of Composite

► Due to the n-recursion, new children can always be added into a composite node
► Whenever you have to program an extensible part of a framework, consider Composite
► Problems:
  ▪ Pattern is hard to employ when it sits on top of a complex inheritance hierarchy
    · Then, use interfaces only or mixin-based inheritance (not available in most languages)

16

Relations of Composite to Other Programming Domains

Boxology
Box Hierarchy

► Composite pattern is the heart of functional programming
  ▪ Because recursion is the heart of functional programming
  ▪ It has discovered many interesting algorithmic schemes for the Composite:
    · Functional skeletons (map, fold, partition, d&c, zip...)
    · Barbed wire (homo- and other morphisms)
► The Composite is also the heart of attributed trees and attribute grammars
  ▪ Ordered AG are constraint systems that generate iterators and skeletons [CompilerConstruction]
► Adaptive Programming [Lieberherr] is a generalization of Composite with Iterators [Component-Based Software Engineering (CBSE)]
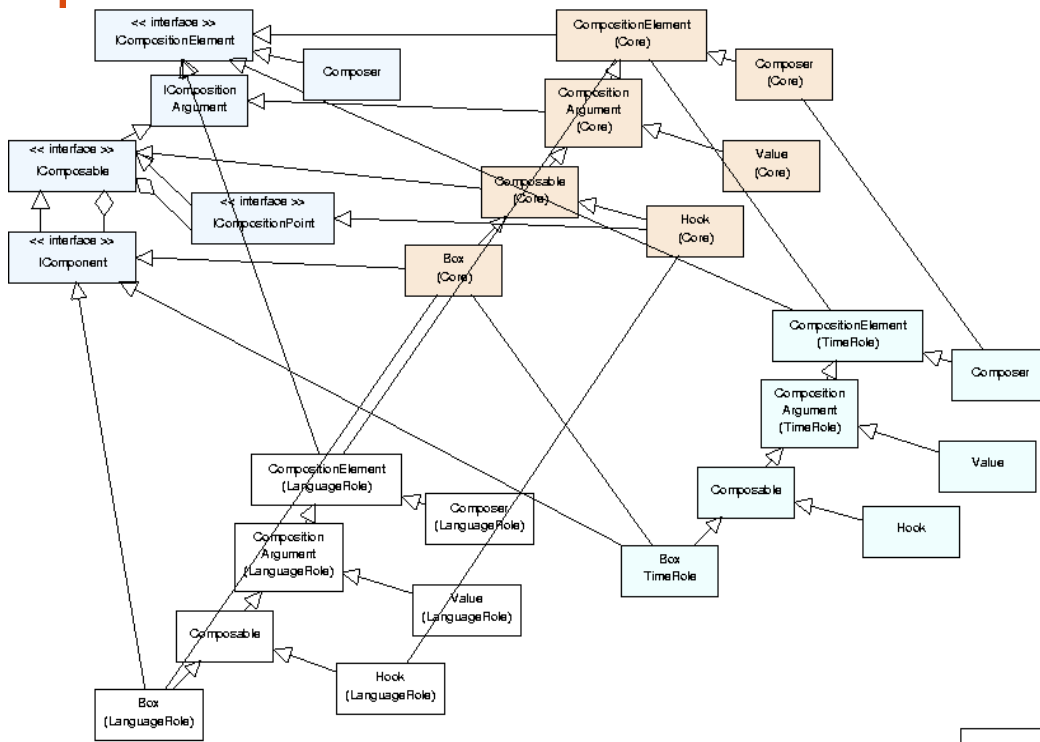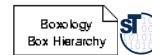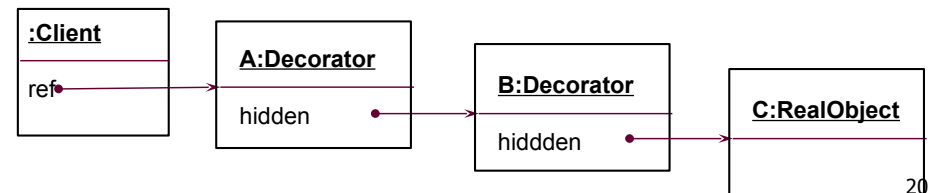
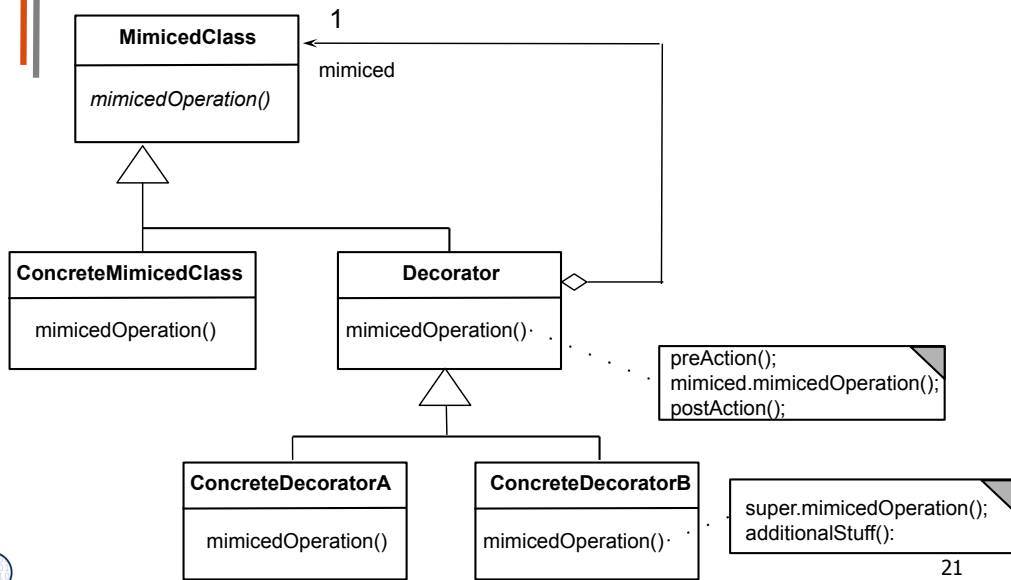# 3.3 Decorator as a Variant of ObjectRecursion and Composite

Decorator Pattern

► A Decorator is a *skin* of another object
► It is a 1-ObjectRecursion (i.e., a restricted Composite):
  ▪ A subclass of a class that contains an object of the class as child
  ▪ However, only one composite (i.e., a delegatee)
  ▪ Combines inheritance with aggregation
► Similar to ObjectRecursion and Composite, inheritance from an abstract Handler class
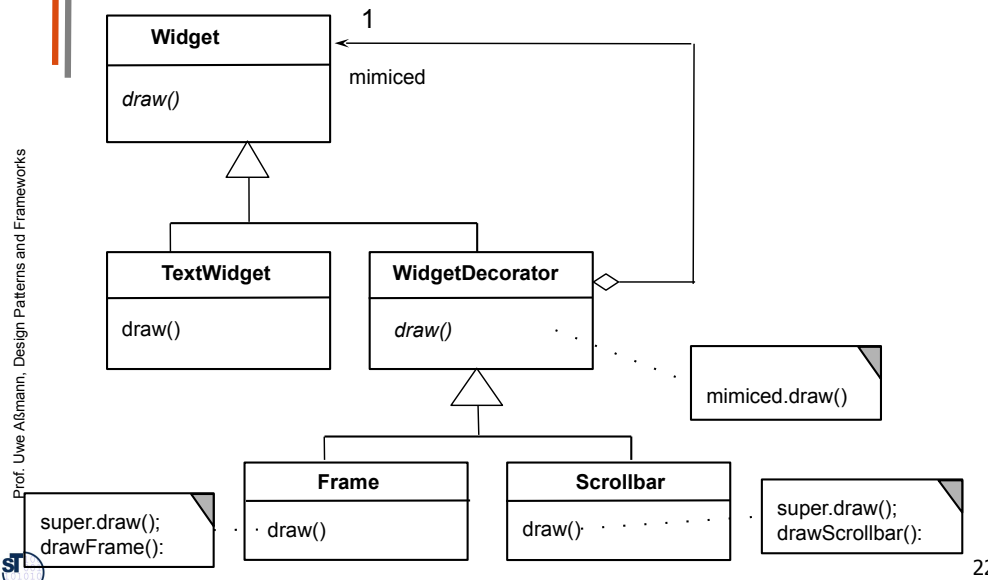 ▪ That defines a contract for the mimiced class and the mimicing class

Decorator – Structure Diagram


```
MimicedClass                              1
mimicedOperation()          mimiced
```

- ConcreteMimicedClass
  - mimicedOperation()
- Decorator
  - mimicedOperation()
    - preAction();
    - mimiced.mimicedOperation();
    - postAction();
- ConcreteDecoratorA
  - mimicedOperation()
- ConcreteDecoratorB
  - mimicedOperation()
    - super.mimicedOperation();
    - additionalStuff():

Prof. Uwe Aßmann, Design Patterns and Frameworks

21

Decorator for Widgets


```
Widget                                    1
draw()                      mimiced
```

- TextWidget
  - draw()
- WidgetDecorator
  - draw()
    - mimiced.draw()
- Frame
  - draw()
    - super.draw();
    - drawFrame():
- Scrollbar
  - draw()
    - super.draw();
    - drawScrollbar():

Prof. Uwe Aßmann, Design Patterns and Frameworks

22

Decorator for Persistent Objects


```
Record                                    1
access()                    mimiced
```

- TransientRecord
  - access()
- PersistentDecorator
  - access()
    - mimiced.access()
- PersistentReadOnlyRecord
  - access()
  - boolean loaded()
  - load()
    - if (!loaded()) load();
    - super.access();
- PersistentRecord
  - access()
  - boolean loaded()
  - boolean modified()
  - load()
  - dump()
    - if (!loaded()) load();
    - super.access();
   - if (modified()) dump():

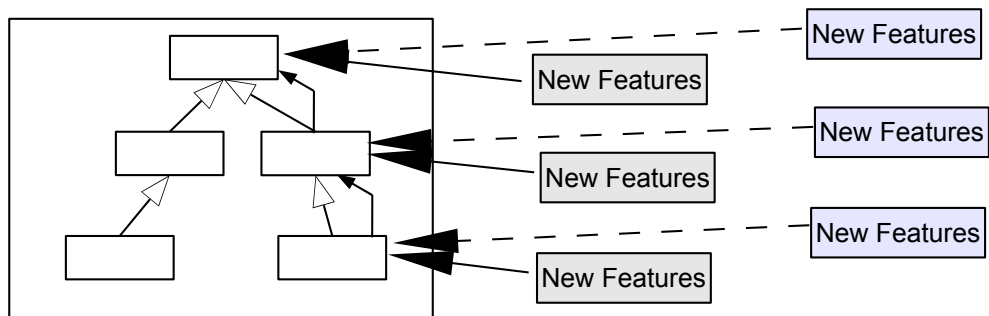Prof. Uwe Aßmann, Design Patterns and Frameworks

23
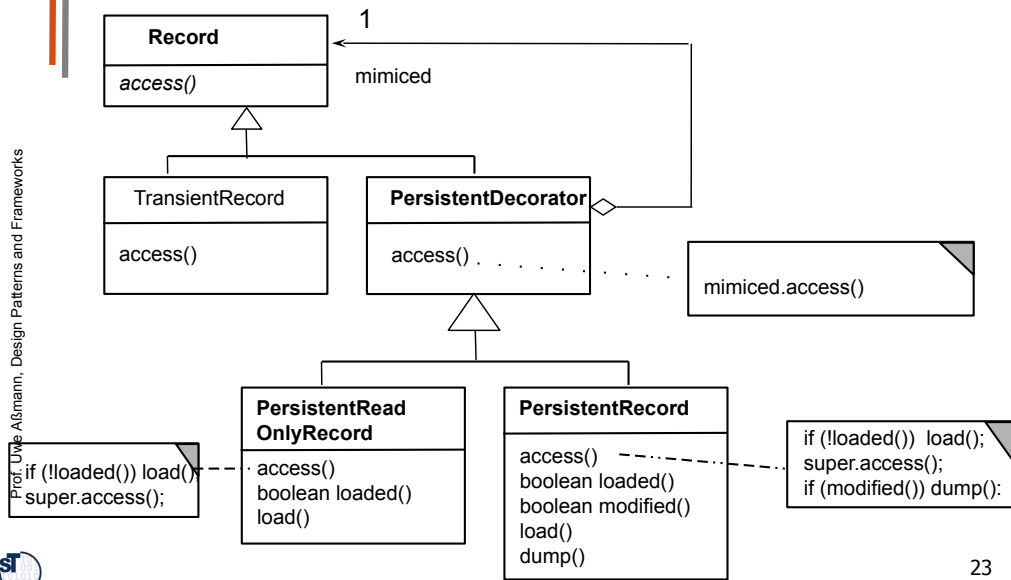
Purpose Decorator

- ► For extensible objects (i.e., decorating objects)
  - ▪ Extension of new features at runtime
  - ▪ Removal possible
- ► Instead of putting the extension into the inheritance hierarchy
  - ▪ If that would become too complex
  - ▪ If that is not possible since it is hidden in a library

Library   New Features

Library   Decorator with New Features

Prof. Uwe Aßmann, Design Patterns and Frameworks

24
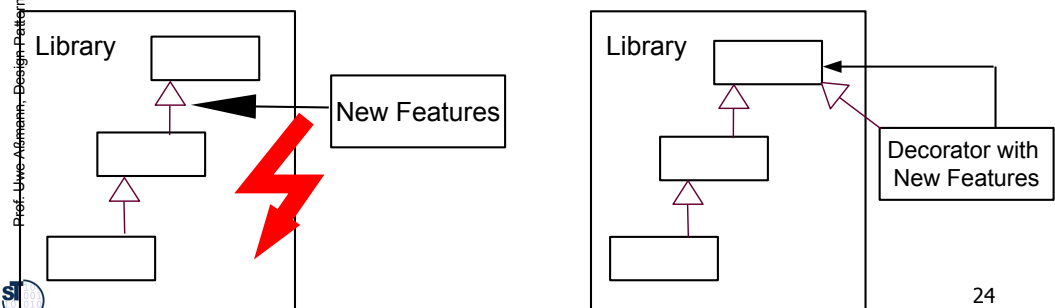
Variants of Decorators

- ▶ If only one extension is planned, the abstract superclass Decorator can be saved; a concrete decorator is sufficient
- ▶ **Decorator family:** If several decorators decorate a hierarchy, they can follow a common style and can be exchanged together

New Features

New Features

New Features

New Features

New Features

New Features

3.4 Chain of Responsibility

Chain of Responsibility

- ▶ Delegate an action to a list of delegatees
  - ▪ That attempt to solve the problem one after the other
  - ▪ Or delegate further on, down the chain
  - ▪ "daisy chain" principle

ObjectStructure:

:Client
aWorker

A:ConcreteWorker
successor

B:ConcreteWorker
successor

Structure for ChainOfResponsibility

- ▶ A Chain is recursing on the abstract super class, i.e.,
  - ▪ All classes in the inheritance tree know they hide some other class (unlike the ObjectRecursion)
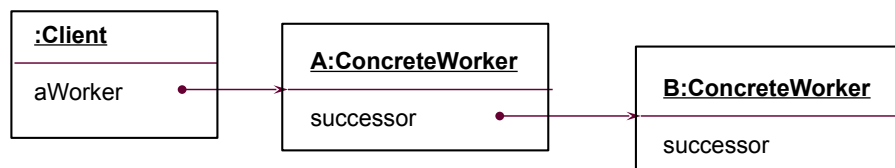
Successor

1

Client

Worker
Work()

ConcreteWorker1
Work()

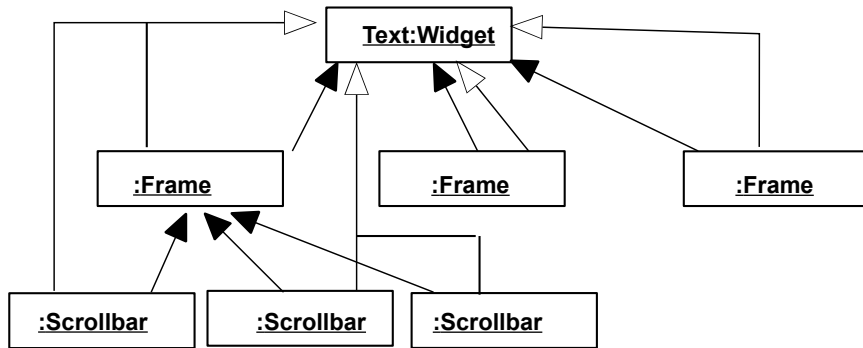| ConcreteWorker2 |
|---|
| Work() |
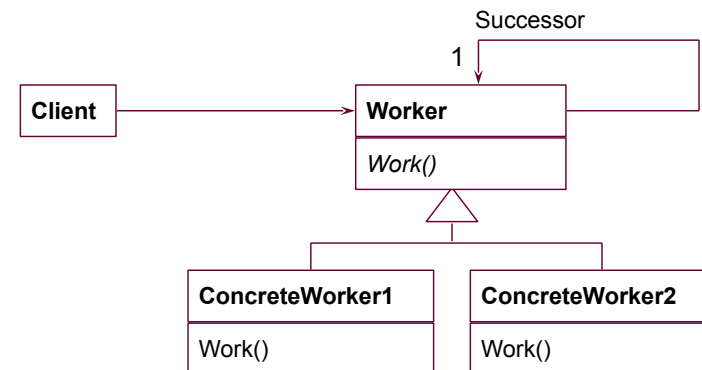
Chains in Runtime Trees

- ► Chains can also be parts of a tree
- ► Then, a chain is the path upward to the root of the tree

Text:Widget

:Frame   **:Frame**   **:Frame**

:Scrollbar   **:Scrollbar**   **:Scrollbar**

29

Example ChainOfResponsibility
Help System for a GUI

nextWorker

HelpWorker

WorkOnHelpQuery()

ObjectStructure:

Application   **Widget**

Dialog   **Button**
WorkOnHelpQuery()
showHelp()

StoreDialog   **PrintDialog**

OKButton   **PrintButton**

:PrintButton
nextWorker

:OKButton
nextWorker

:PrintDialog
nextWorker

:StoreDialog
nextWorker

:Application
nextWorker

30
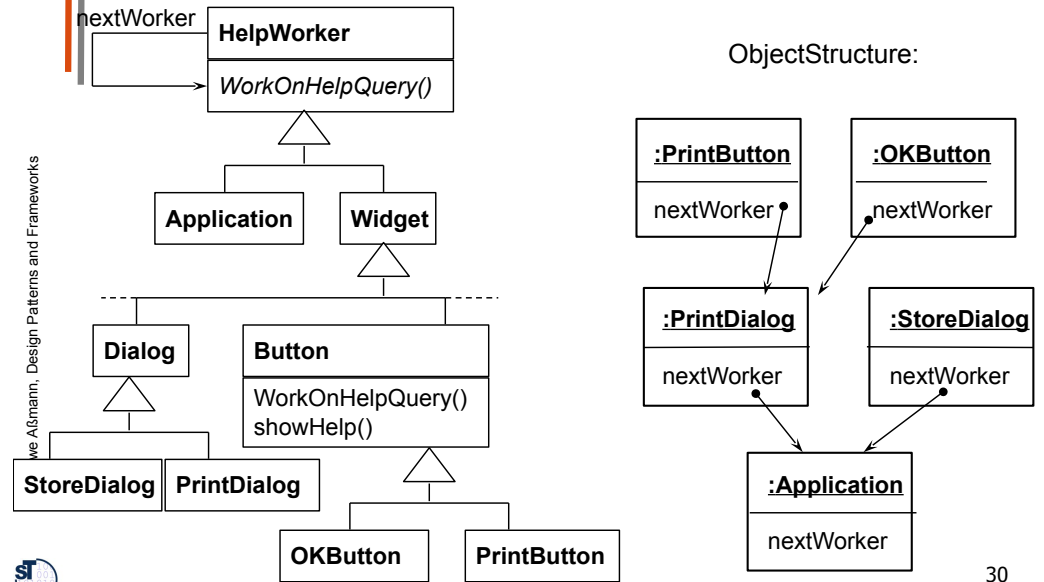
Help System with Chain

```
abstract class HelpWorker {
  HelpWorker nextWorker;  // here is the 1-
    recursion
  void workOnHelpQuery() {
  if (nextWorker)
    nextWorker.workOnHelpQuery();
  } else { /* no help available */ }
}
class Widget extends HelpWorker {
  // this class can contain fixing code
}
class Dialog extends Widget {
  void  workOnHelpQuery() {
    help(); super.workOnHelpQuery();
  }
}
class Application extends HelpWorker { ....}
```

```
class Button extends Widget {
  bool haveHelpQuery;
  void workOnHelpQuery() {
   if (haveHelpQuery) {
     help();
  } else {
     super.workOnHelpQuery();
   }
  }
}

// application
button.workOnHelpQuery();
// may end in the inheritance hierarchy up in
    Widget, HelpWorker
// dynamically in application object
```
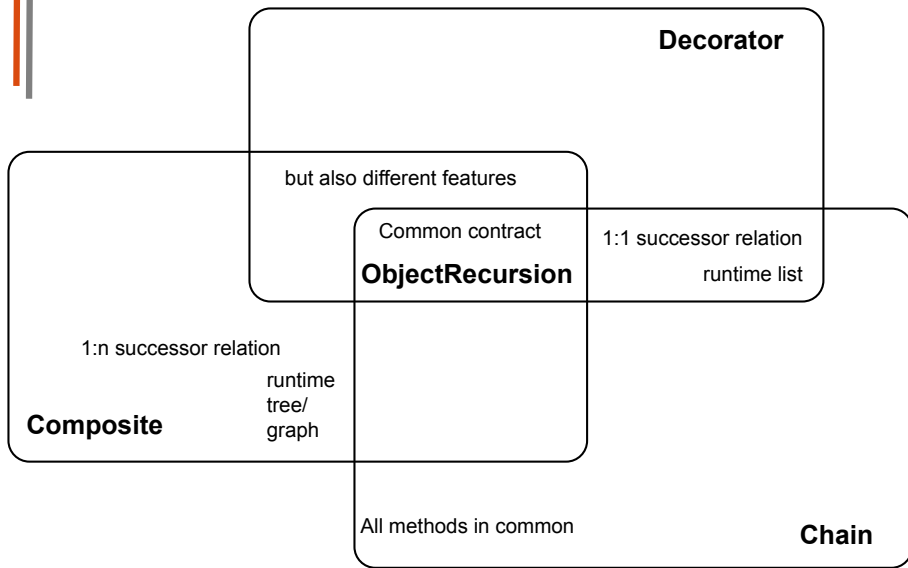
31

ChainOfResponsibility - Applications

- ► Realizes *Dynamic Call*:
  - If the receiver of a message is not known compile-time
  - Nor at allocation time (polymorphism)
  - But dynamically
  - Dynamic call is the key construct for service-oriented architectures (SOA)
- ► Dynamic extensibility: if new receivers with new behavior should be added at runtime
  - Unforeseen dynamic extensions
  - However, no mimiced object as in Decorator
- ► Anonymous communication
  - If identity of receiver is unknown or not important
  - If several receivers should work on a message

32

Composite vs Decorator vs Chain

Decorator

but also different features

Common contract
ObjectRecursion

1:1 successor relation
runtime list

1:n successor relation

runtime tree/ graph

Composite

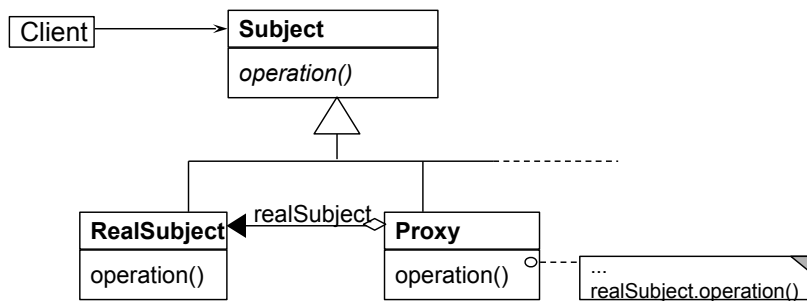All methods in common
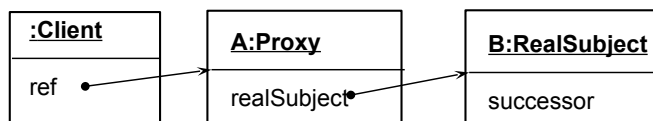
Chain

3.5 Proxy

Proxy

▶ Hide the access to a real subject by a representant

Object Structure:

Proxy

▶ The proxy object is a representant of an object
   - The Proxy is similar to Decorator, but it is not derived from ObjectRecursion
   - It has a direct pointer to the sister class, *not* to the superclass
   - It may collect all references to the represented object (shadows it). Then, it is a facade object to the represented object

▶ Consequence: chained proxies are not possible, a proxy is one-and-only

▶ Clear difference to ChainOfResponsibility
   - Decorator lies between Proxy and Chain.

Proxy Variants

- *Filter proxy (smart reference):* executes additional actions, when the object is accessed
  - Protocol proxy: counts references (reference-counting garbage collection
  - or implements a synchronization protocol (e.g., reader/writer protocols)
- *Indirection proxy (facade proxy):* assembles all references to an object to make it replaceable
- *Virtual proxy:* creates expensive objects on demand
- *Remote proxy:* representant of a remote object
- *Caching proxy:* caches values which had been loaded from the subject
  - Remote
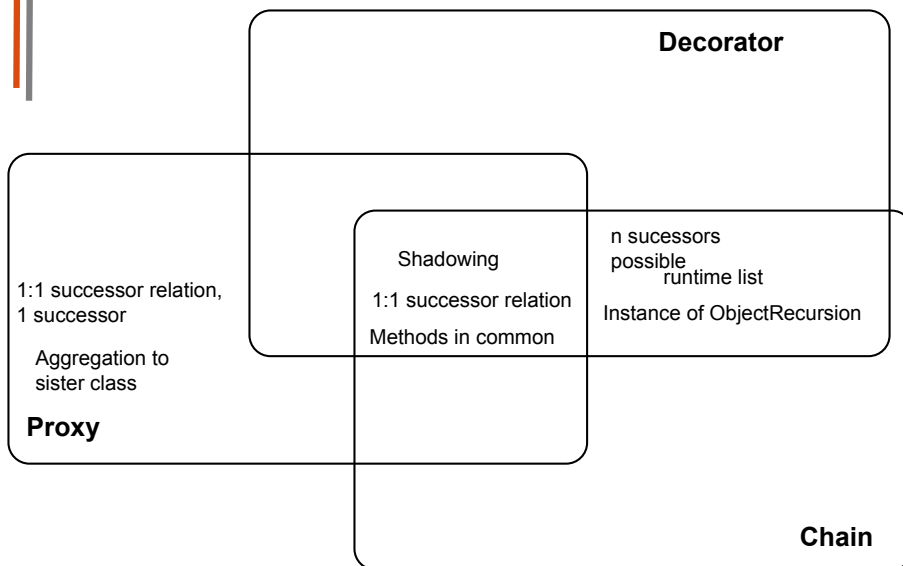 - Loading lazy on demand
- *Protection proxy*
  - Firewall

Proxy – Other Implementations

- Overloading of -> access operation
  - C++ and other languages allow for overloading access
  - Then, a proxy can invervene
- Built in into the language
  - There are languages that offer proxy objects
  - Modula-3 offers SmartPointers
  - Gilgul offers proxy objects

Proxy vs Decorator vs Chain
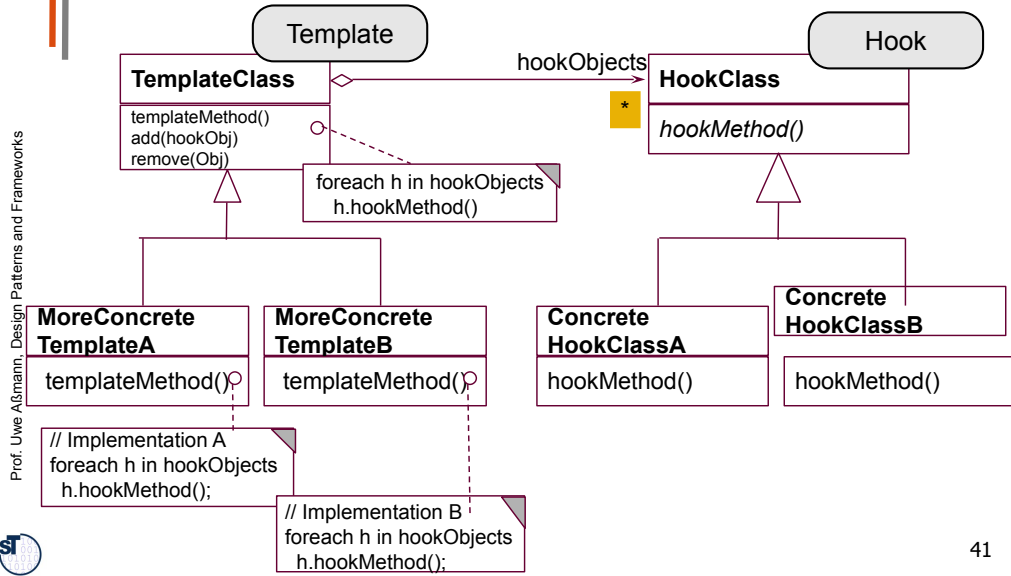
Decorator

Shadowing

1:1 successor relation

Methods in common

n sucessors possible

runtime list

Instance of ObjectRecursion

1:1 successor relation, 1 successor

Aggregation to sister class

Proxy

Chain

3.6 *-Bridge

Extensibility Pattern *DimensionalClassHierarchies (*Bridge)

▶ A bridge with a collection

Template

TemplateClass
templateMethod()
add(hookObj)
remove(Obj)

foreach h in hookObjects
  h.hookMethod()

hookObjects

Hook

HookClass
hookMethod()

*

MoreConcrete TemplateA
templateMethod()

MoreConcrete TemplateB
templateMethod()

// Implementation A
foreach h in hookObjects
  h.hookMethod();

// Implementation B
foreach h in hookObjects
  h.hookMethod();

Concrete HookClassA
hookMethod()

Concrete HookClassB
hookMethod()

41

3.7 Observer – (Event Bridge)

Design Patterns and Frameworks, © Prof. Uwe Aßmann

42

Observer (Publisher/Subscriber, Event Bridge)

Observer

	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

a=50%
b=30%
c=20%

Subject

→ Notify on change

┄→ Queries

43
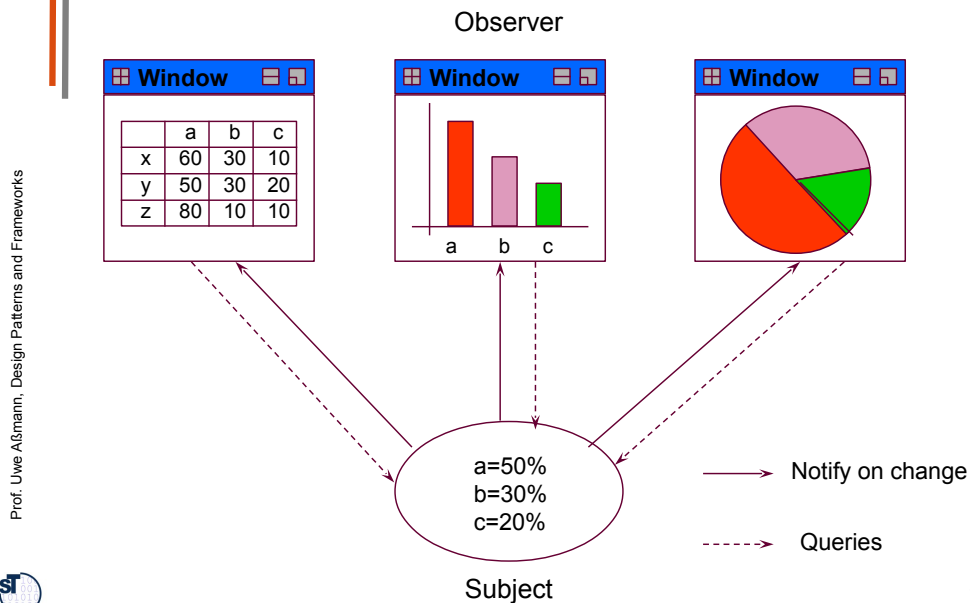
Structure Observer

▶ Extension of *-Bridge

Subject
register(Observer)
unregister(Observer)
notify()

for all b in observers {
  b.update ()
}

observers

*

Observer
update ()

ConcreteSubject
getState()
setState()

SubjectState

return SubjectState

Subject

ConcreteObserver
update ()

ObserverState

ObserverState =
Subject.getState()

Difference to Bridge: hierarchies are not complete independent; Observer knows about Subject

44

Sequence Diagram Observer

- ► Update() does not transfer data, only an event (anonymous communication possible)
- ► Observer pulls data out itself
  - ▪ Due to pull of data, subject does not care nor know, which observers are involved: subject independent of observer


45
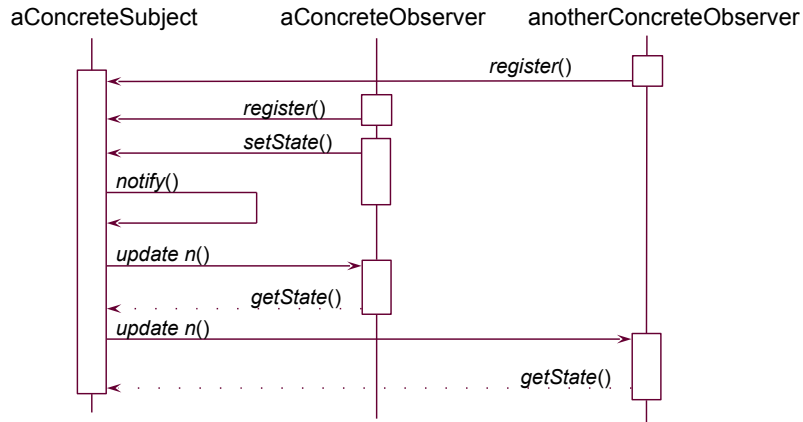
Observer - Applications

- ► Loose coupling in communication
  - ▪ Observers decide what happens
- ► Dynamic change of communication
  - ▪ Anonymous communication
  - ▪ Multi-cast and broadcast communication
  - ▪ Cascading communication if observers are chained (stacked)
- ► Communication of core and aspect
  - ▪ If an abstraction has two aspects and one of them depends on the other, the observer can implement the aspect that listens and reacts on the core
  - ▪ Observers are a simple way to implement aspect-orientation by hand

46

Observer Variants

- ► **Multiple subjects:**
  - ▪ If there is more than one subject, send Subject as Parameter of notify(Subject s).
- ► **Push model:** subject sends data in notify()
  - · The default is the pull model: observer fetches data itself
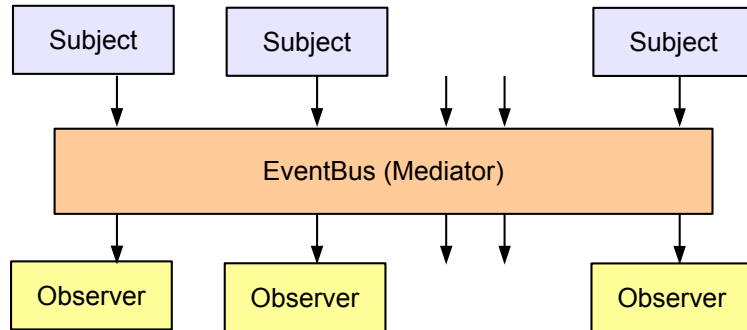- ► **Change manager**

47

Observer with ChangeManager (Mediator)

- ► Mediator between subjects and observer:
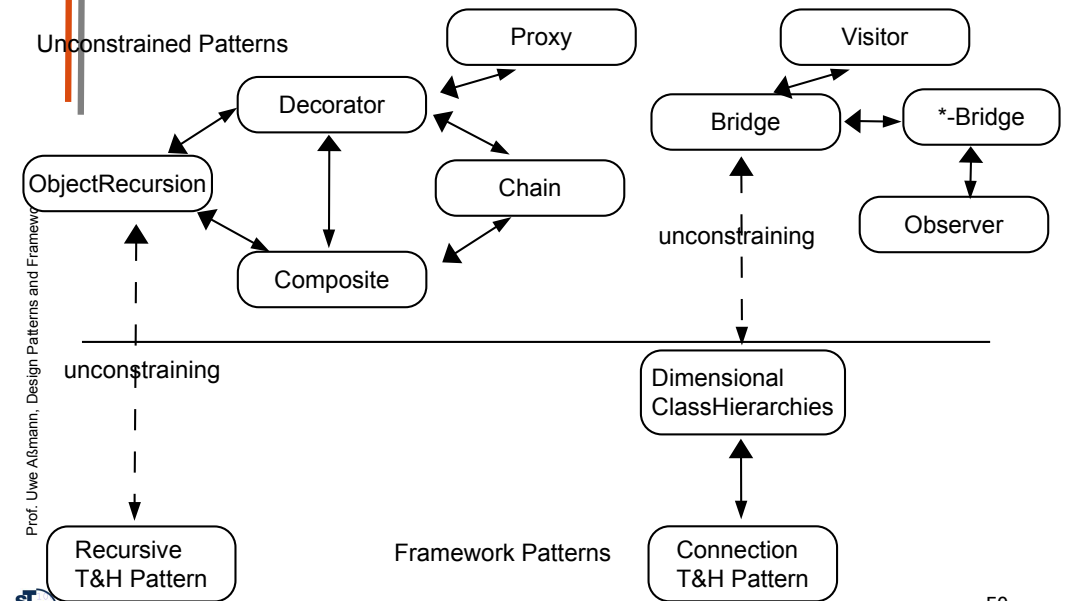 - · May filter events, stop cascaded propagations
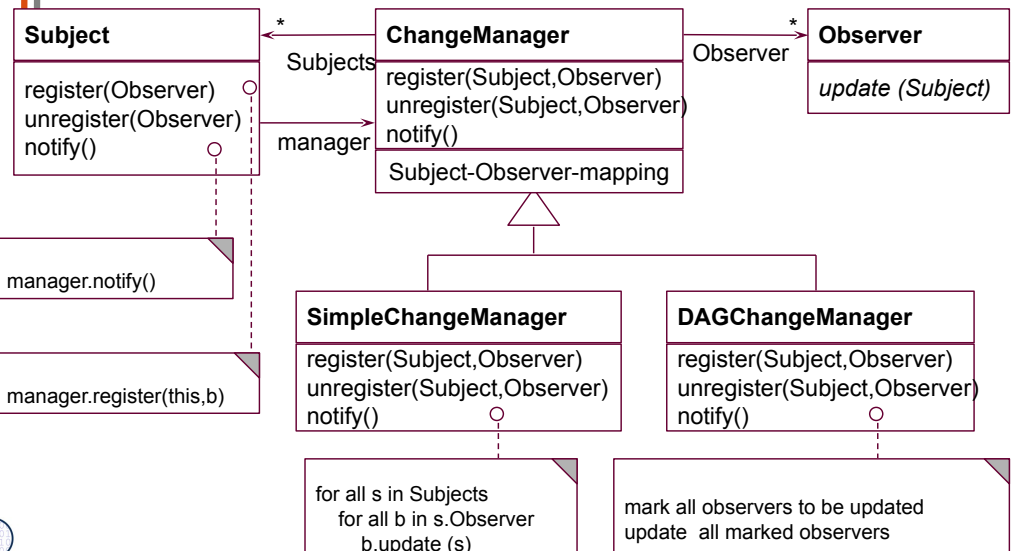
ChangeManager is also Called Eventbus

▶ Basis of many interactive application frameworks (Xwindows, Java AWT, Java InfoBus, ...)

49

Relations Extensibility Patterns

Unconstrained Patterns

50

Summary

▶ Most often, extensibility patterns rely on ObjectRecursion
  - An aggregation to the superclass
▶ This allows for constructing runtime nets: lists, sets, and graphs
  - And hence, for dynamic extension
  - The common superclass ensures a common contract of all objects in the runtime net
▶ Layered systems can be implemented with dimensional class hierarchies (Bridges)
▶ Layered frameworks are product families for systems with layered architectures

51

The End

52