



# 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](http://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



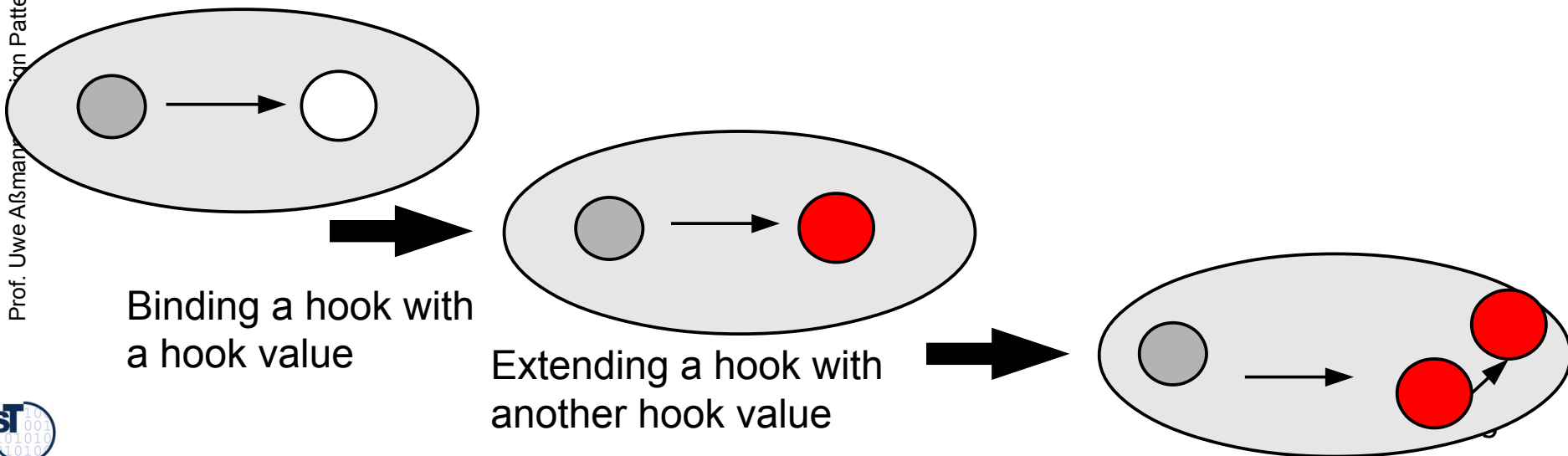
# Static and Dynamic Extensibility

---

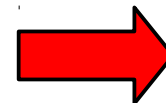
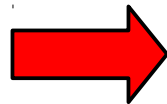
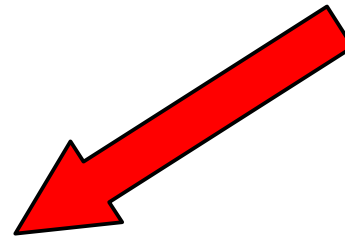
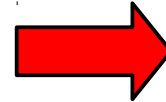
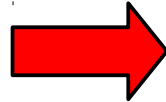
---

# 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



# Software Cocktail Mixers





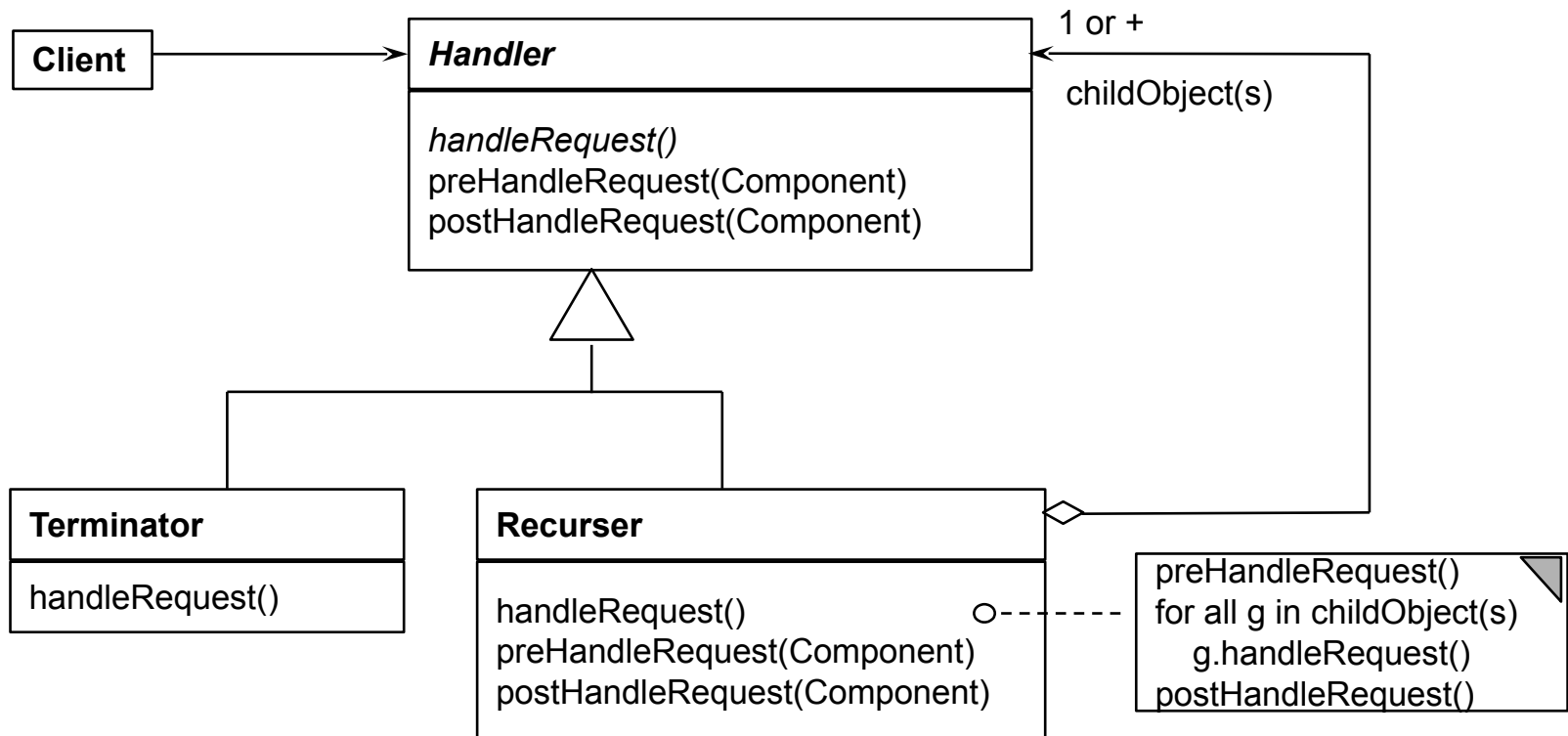
# 3.1 Object Recursion Pattern

---

---

# 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)



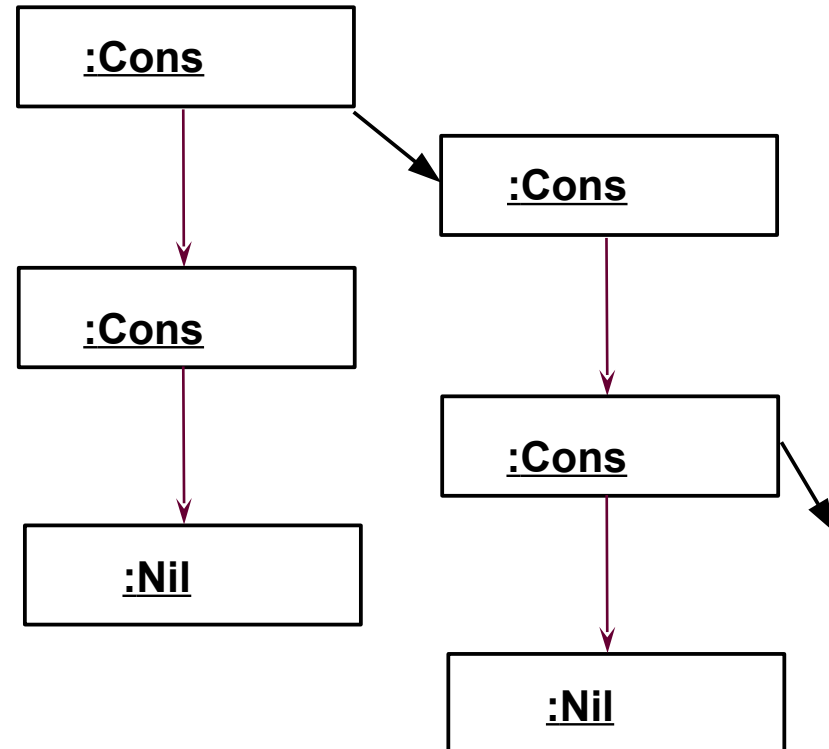
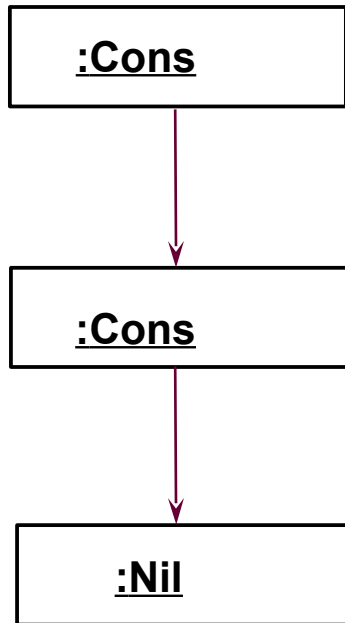


# Incentive

- ▶ 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

# Object Recursion – Runtime Structure

- ▶ 1-ObjectRecursion creates lists
- ▶ n-ObjectRecursion creates trees and graphs



The recursion allows for building up runtime nets

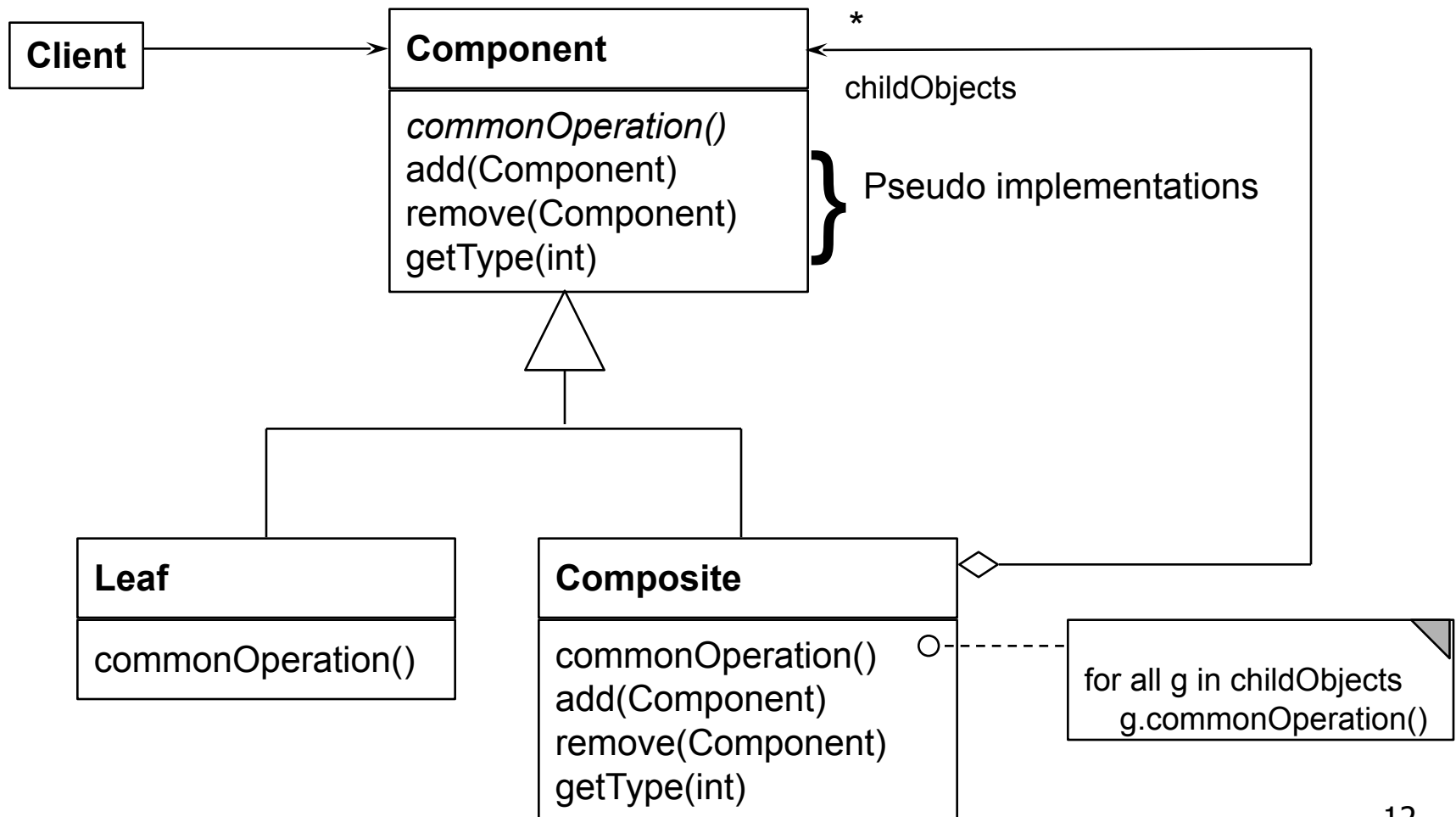


## 3.2 Composite



# Structure Composite

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



# 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
    }
}

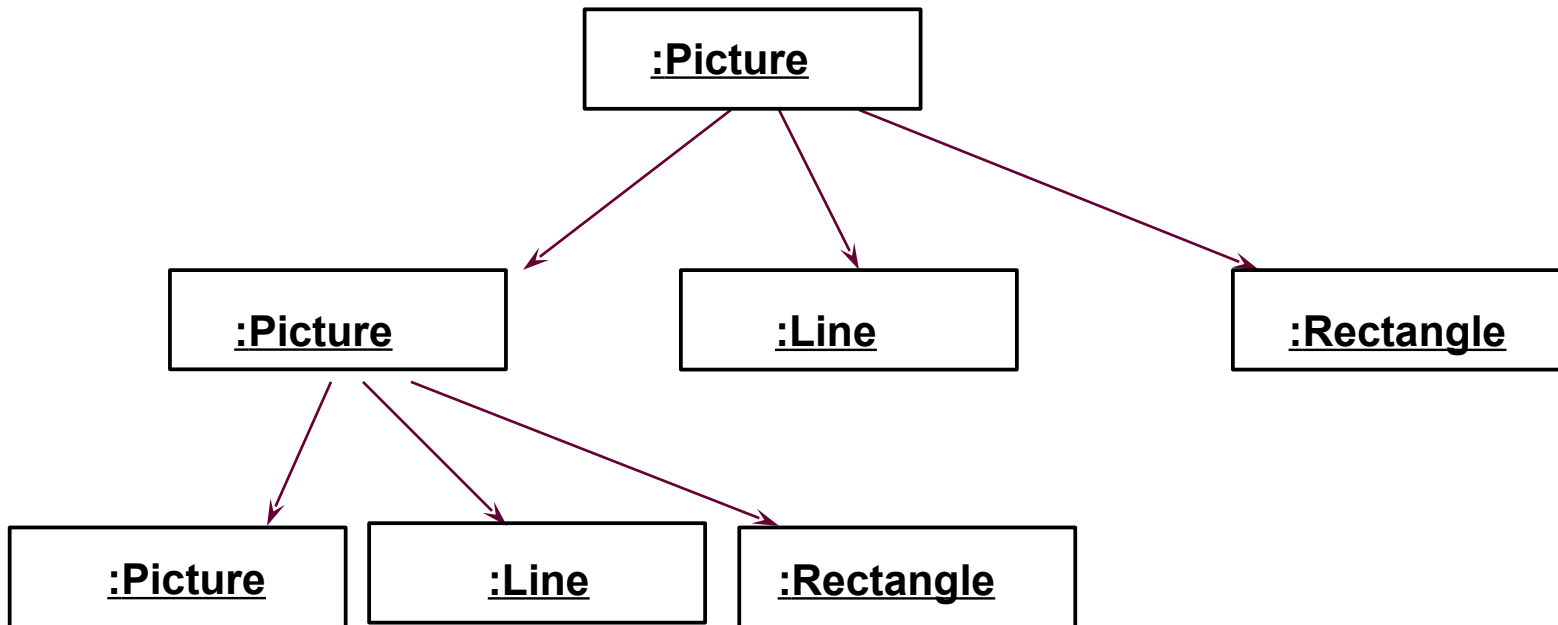
// application
int cost = carPart.calculateCost();
```

# 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

# Composite Run-Time Structure

- ▶ Part/Whole hierarchies, e.g., nested graphic objects

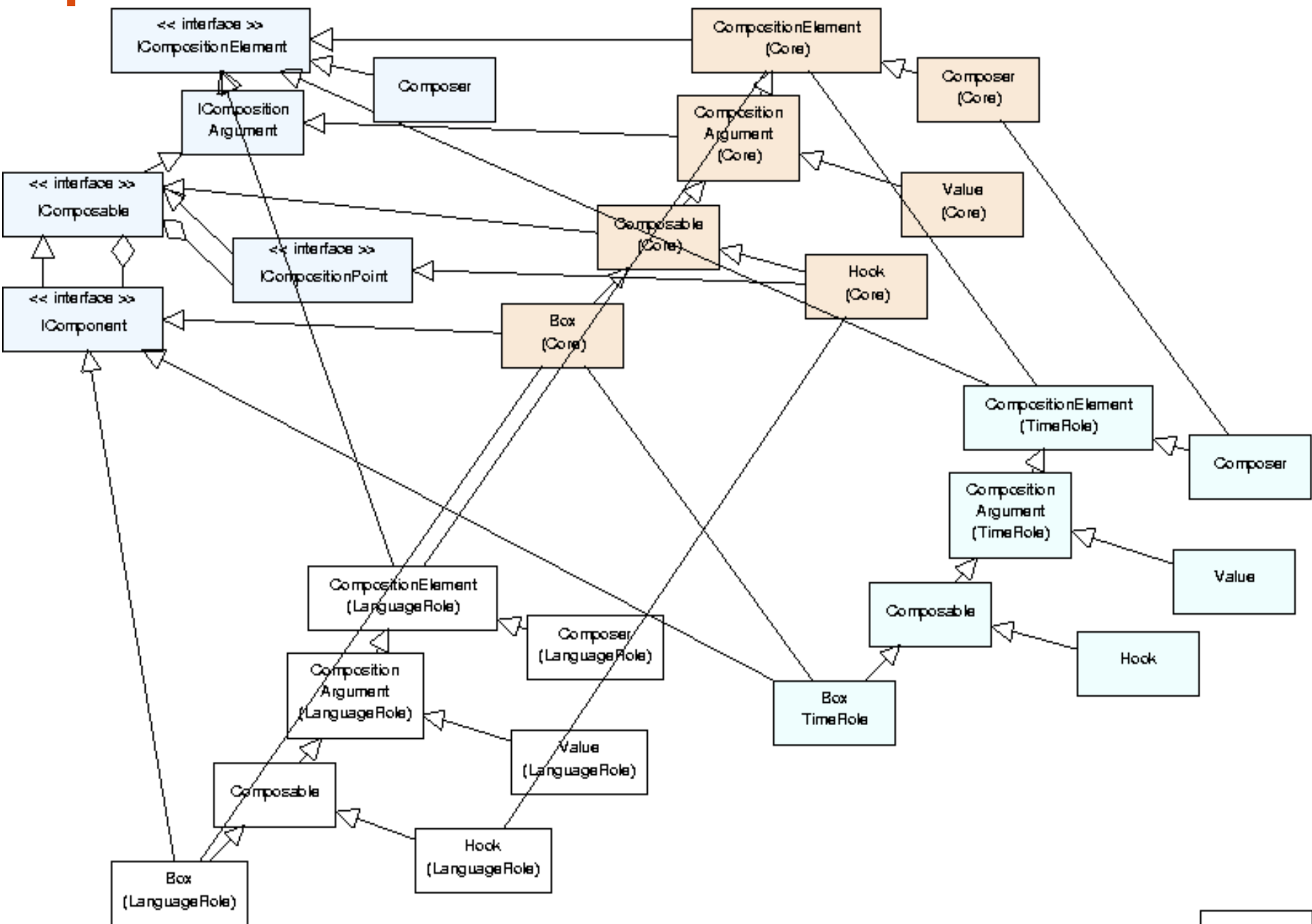


common operations: draw(), move(), delete(), scale()

# Dynamic Extensibility of Composite

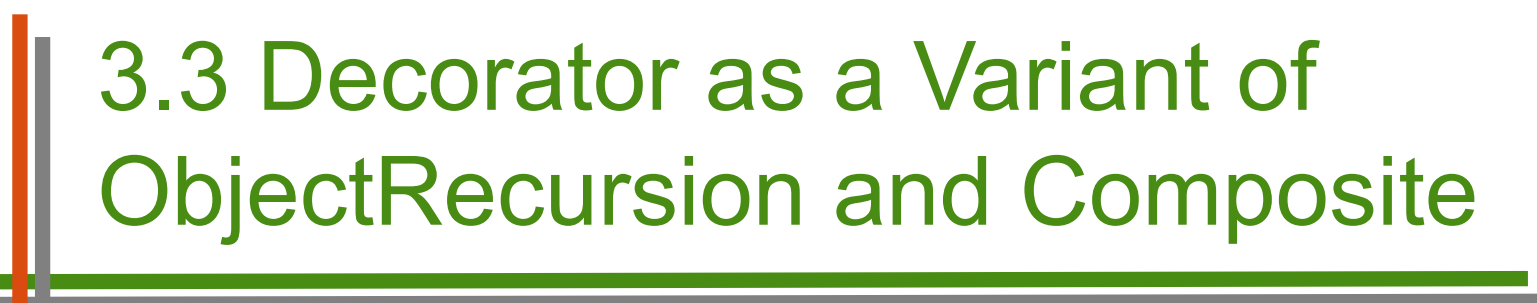
- ▶ Due to the n-recursion, new children can always be added into a composite node
- ▶ Whenever you have to program an extensible part of a framework, consider Composite
- ▶ Problems:
  - Pattern is hard to employ when it sits on top of a complex inheritance hierarchy
    - Then, use interfaces only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





# Relations of Composite to Other Programming Domains

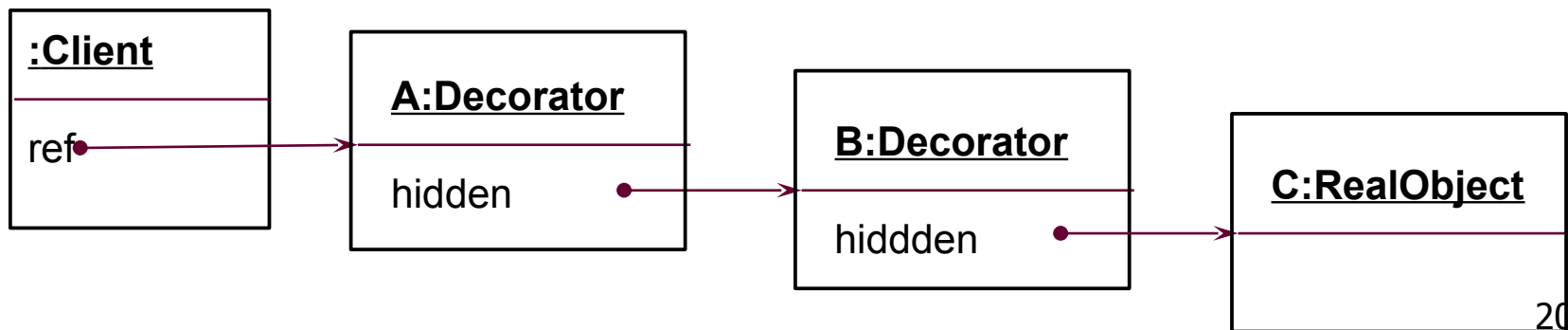
- ▶ Composite pattern is the heart of functional programming
  - Because recursion is the heart of functional programming
  - It has discovered many interesting algorithmic schemes for the Composite:
    - Functional skeletons (map, fold, partition, d&c, zip...)
    - Barbed wire (homo- and other morphisms)
- ▶ 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)]



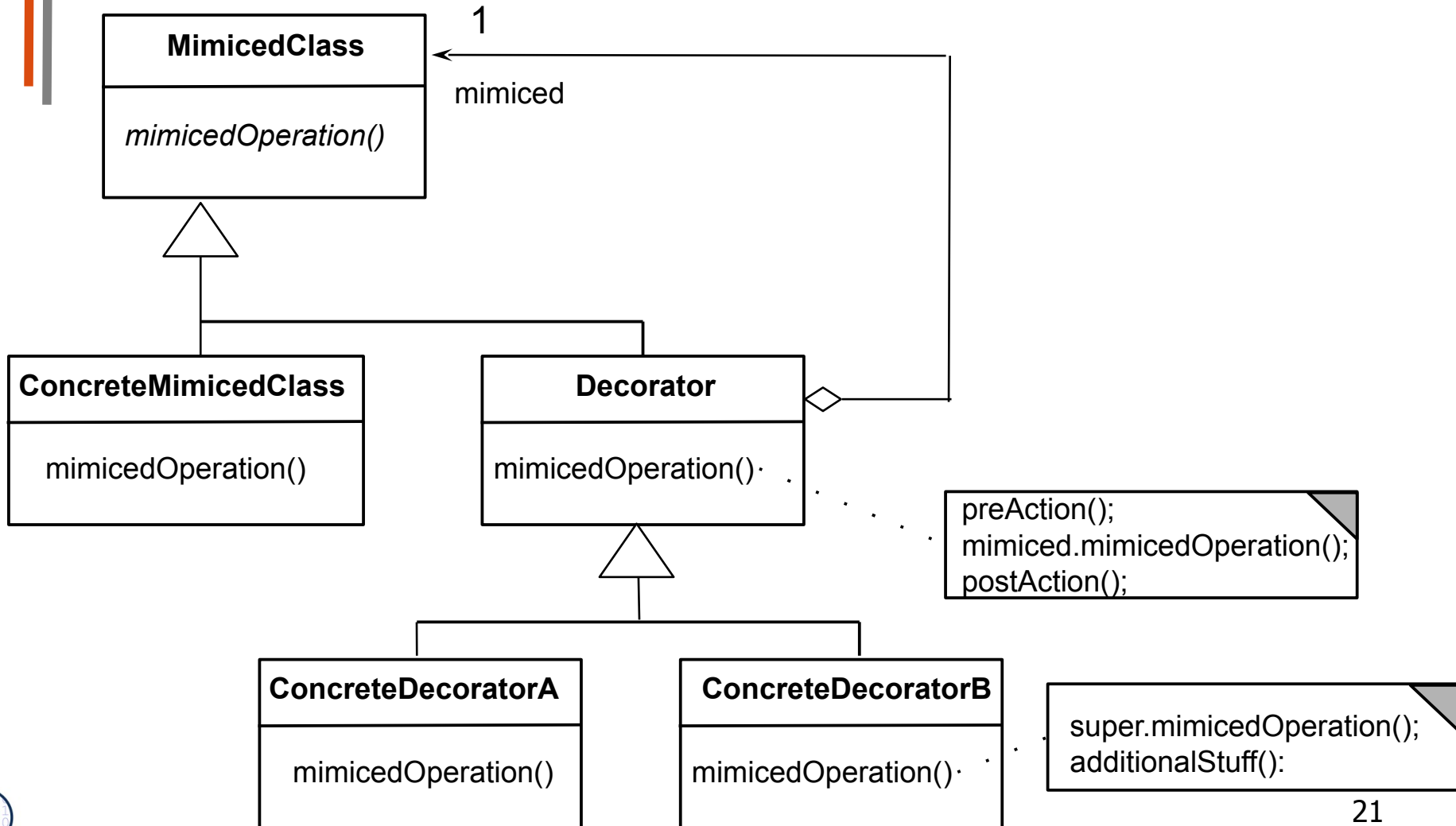
## 3.3 Decorator as a Variant of ObjectRecursion and Composite

# Decorator Pattern

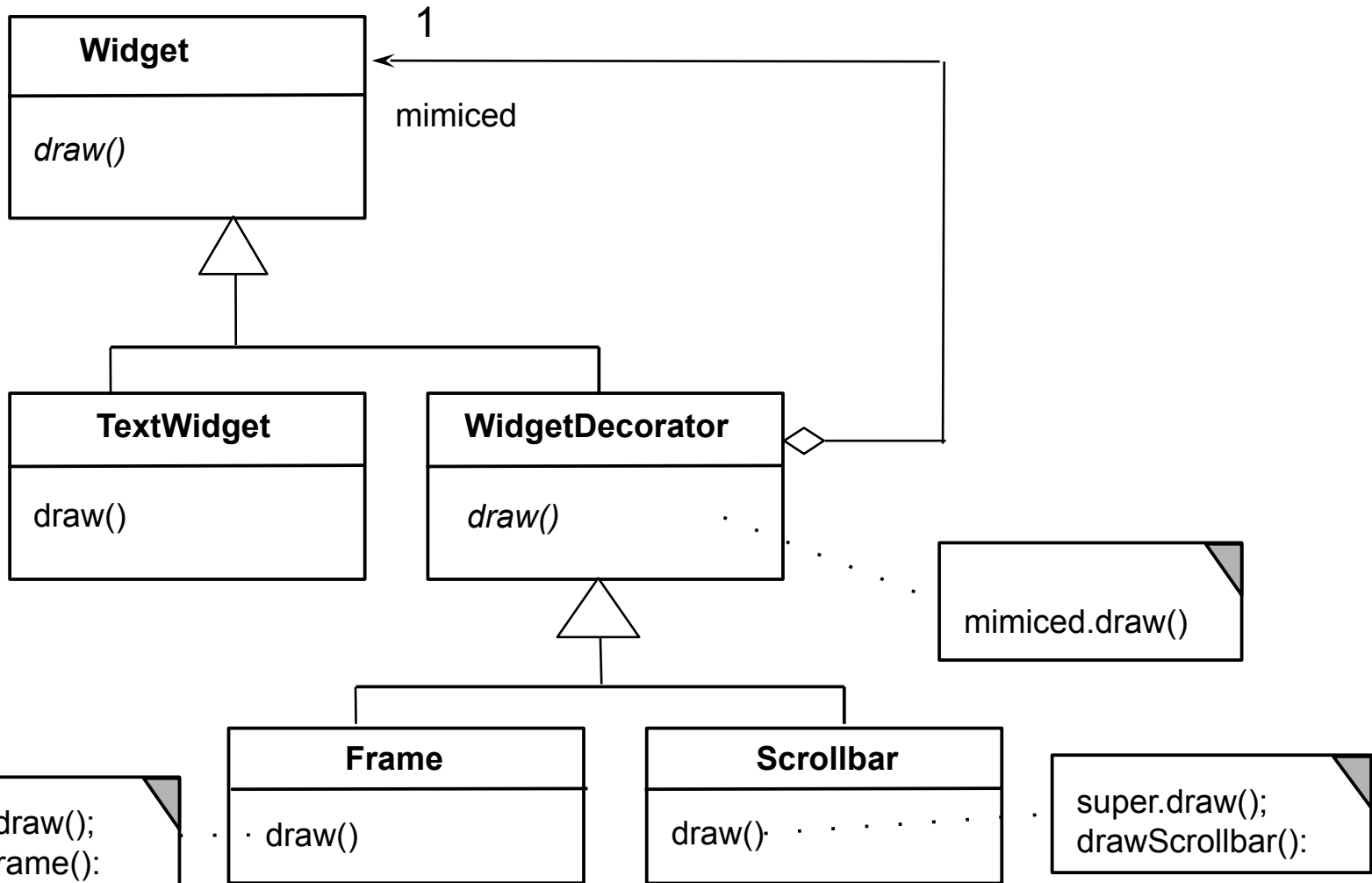
- ▶ A Decorator is a *skin* of another object
- ▶ It is a 1-ObjectRecursion (i.e., a restricted Composite):
  - A subclass of a class that contains an object of the class as child
  - However, only one composite (i.e., a delegatee)
  - Combines inheritance with aggregation
- ▶ Similar to ObjectRecursion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



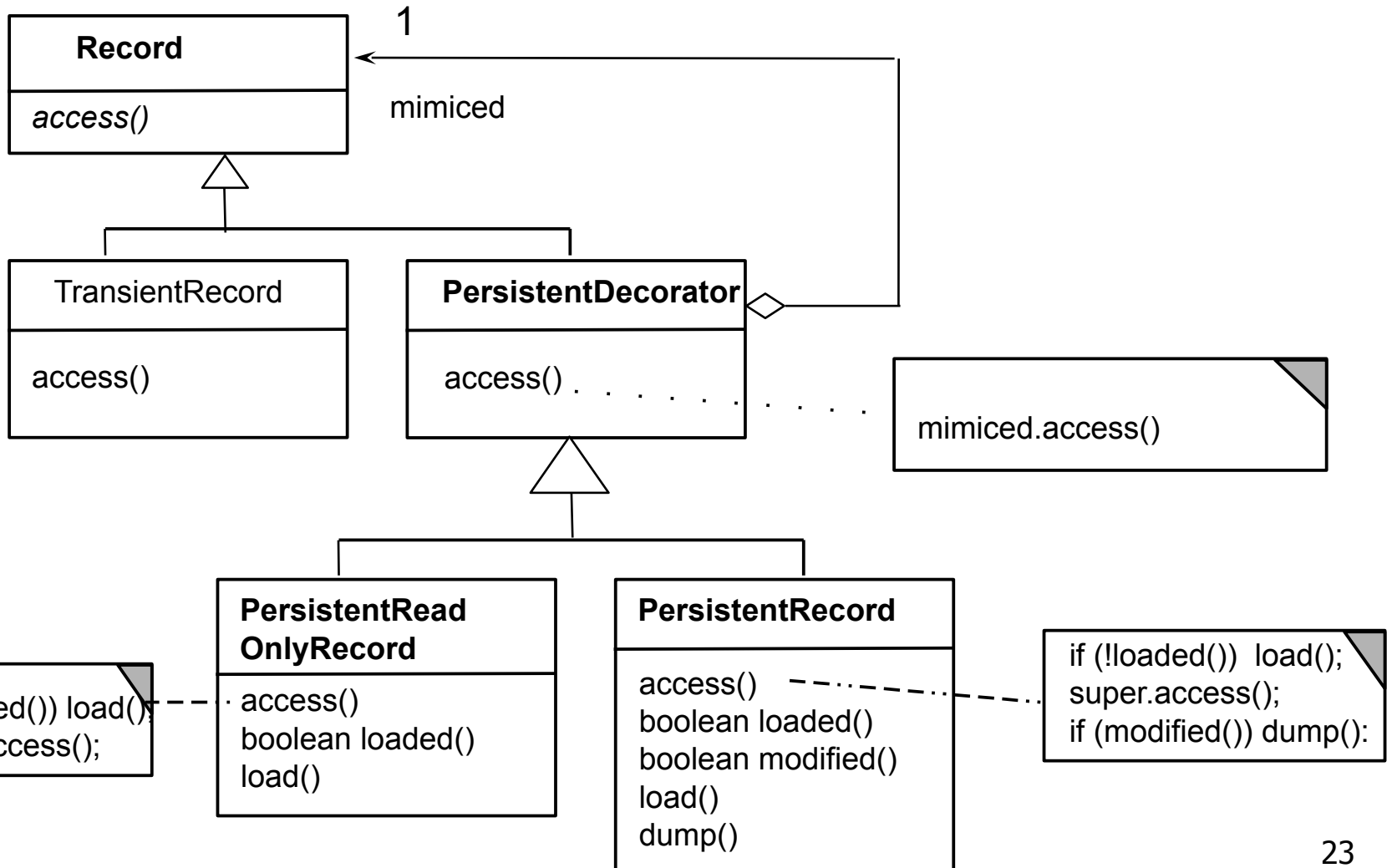
# Decorator – Structure Diagram



# Decorator for Widgets

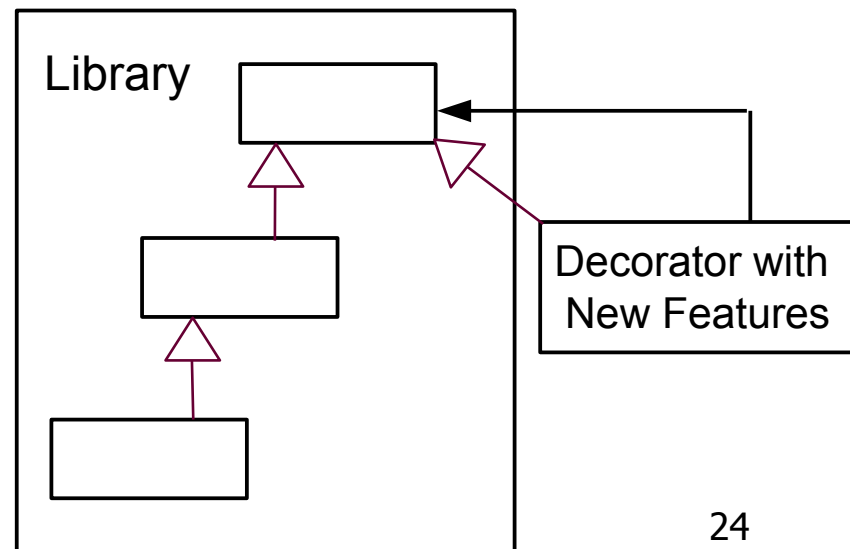
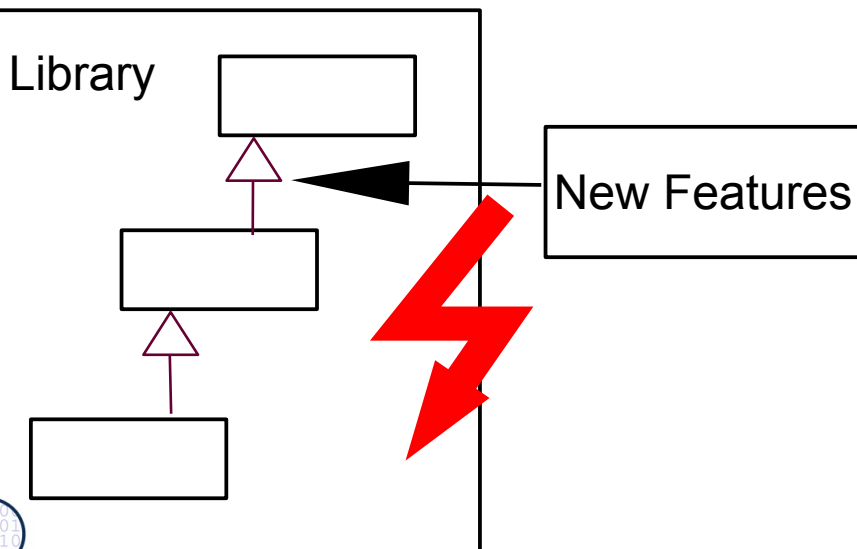


# Decorator for Persistent Objects



# Purpose Decorator

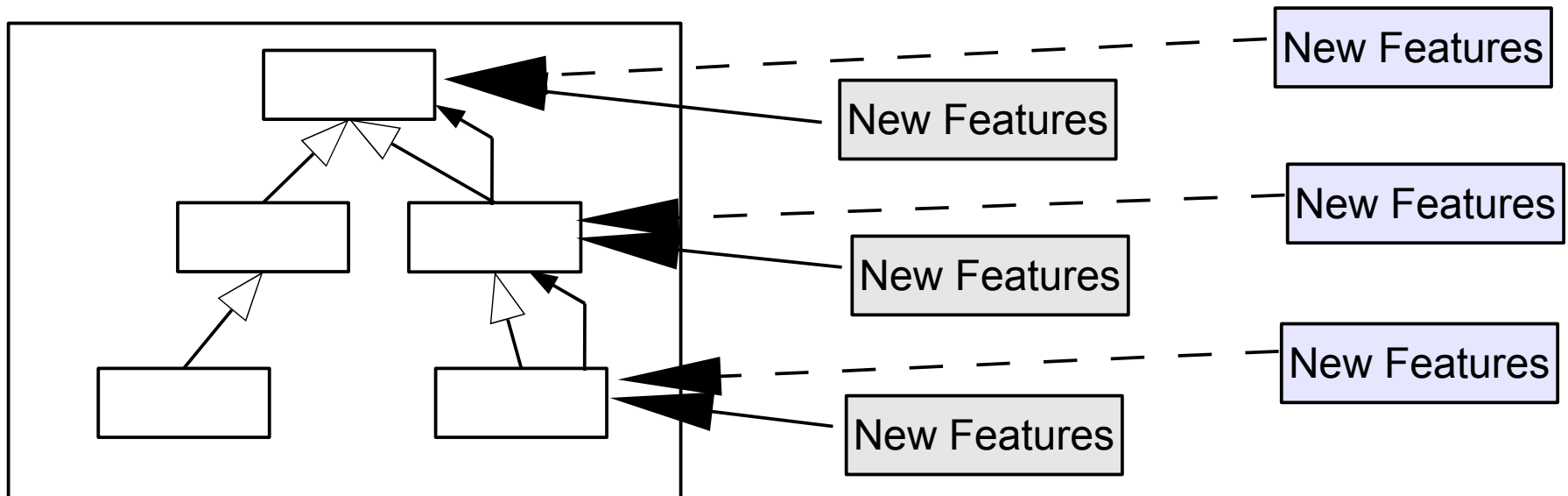
- ▶ For extensible objects (i.e., decorating objects)
  - Extension of new features at runtime
  - Removal possible
- ▶ Instead of putting the extension into the inheritance hierarchy
  - If that would become too complex
  - If that is not possible since it is hidden in a library





# 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



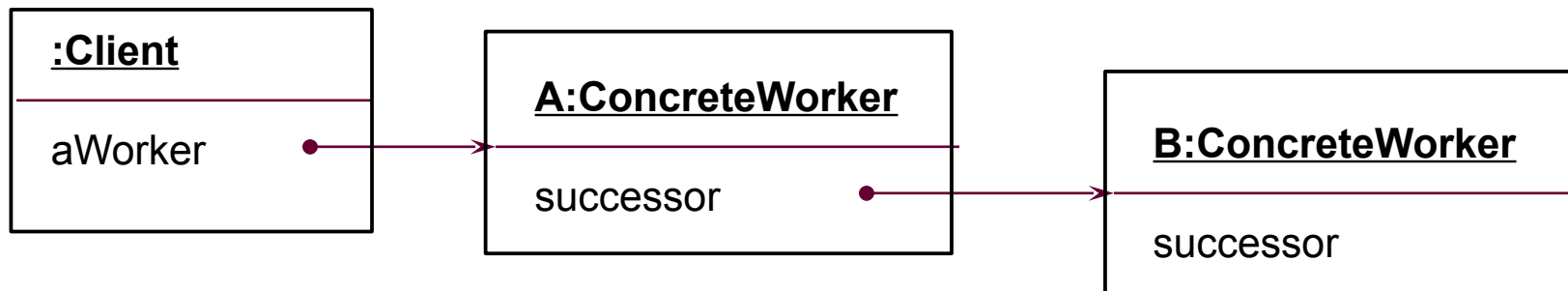


## 3.4 Chain of Responsibility

# Chain of Responsibility

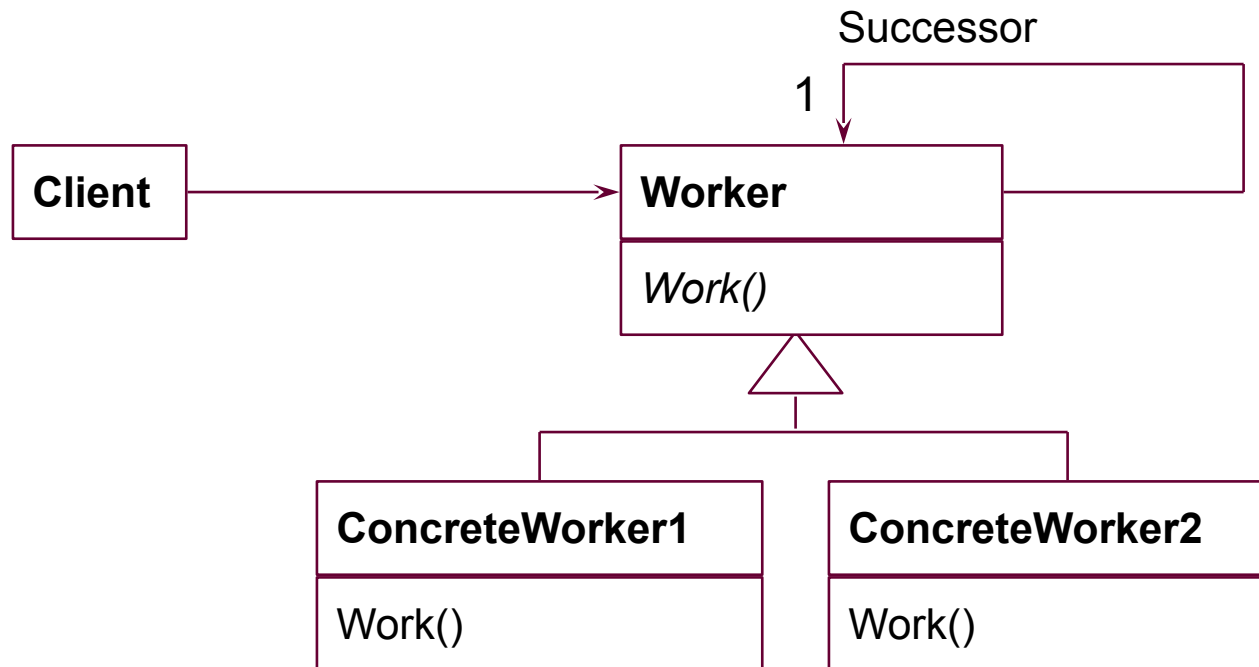
- ▶ Delegate an action to a list of delegates
  - That attempt to solve the problem one after the other
  - Or delegate further on, down the chain
  - “daisy chain” principle

ObjectStructure:



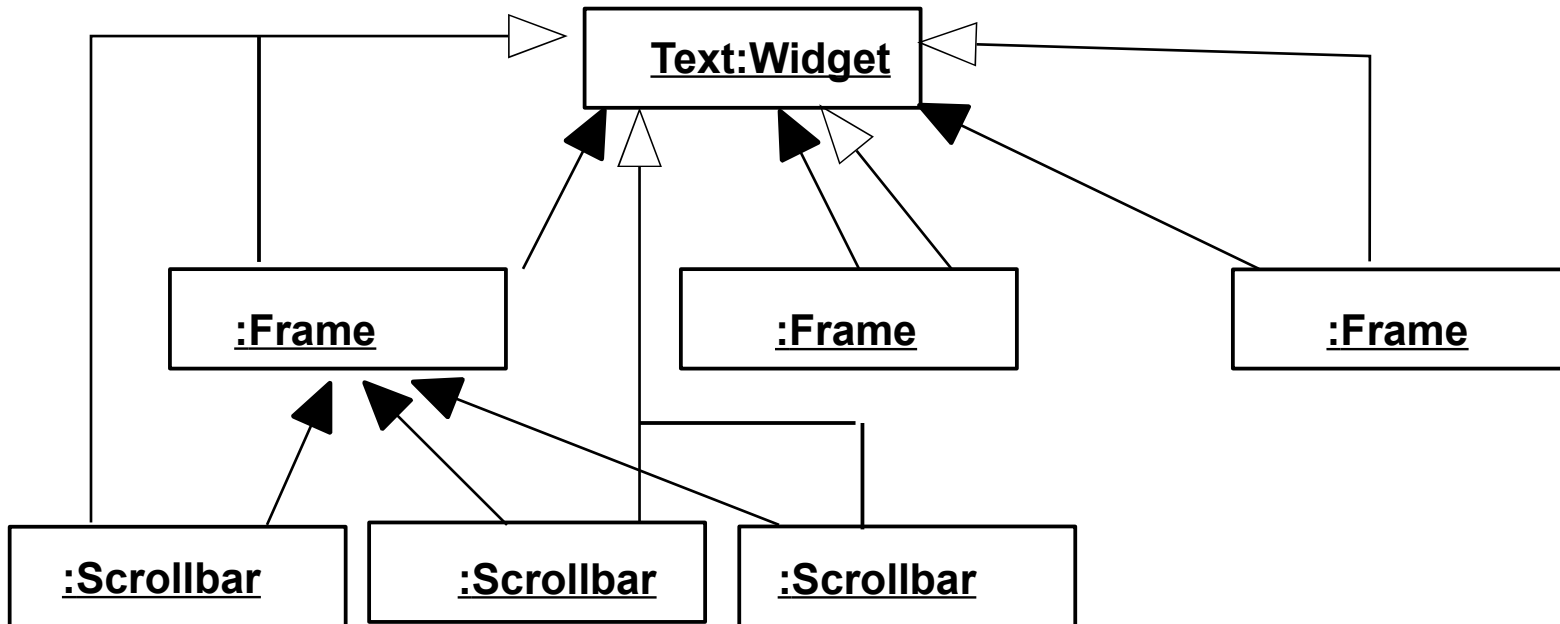
# Structure for ChainOfResponsibility

- ▶ 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)

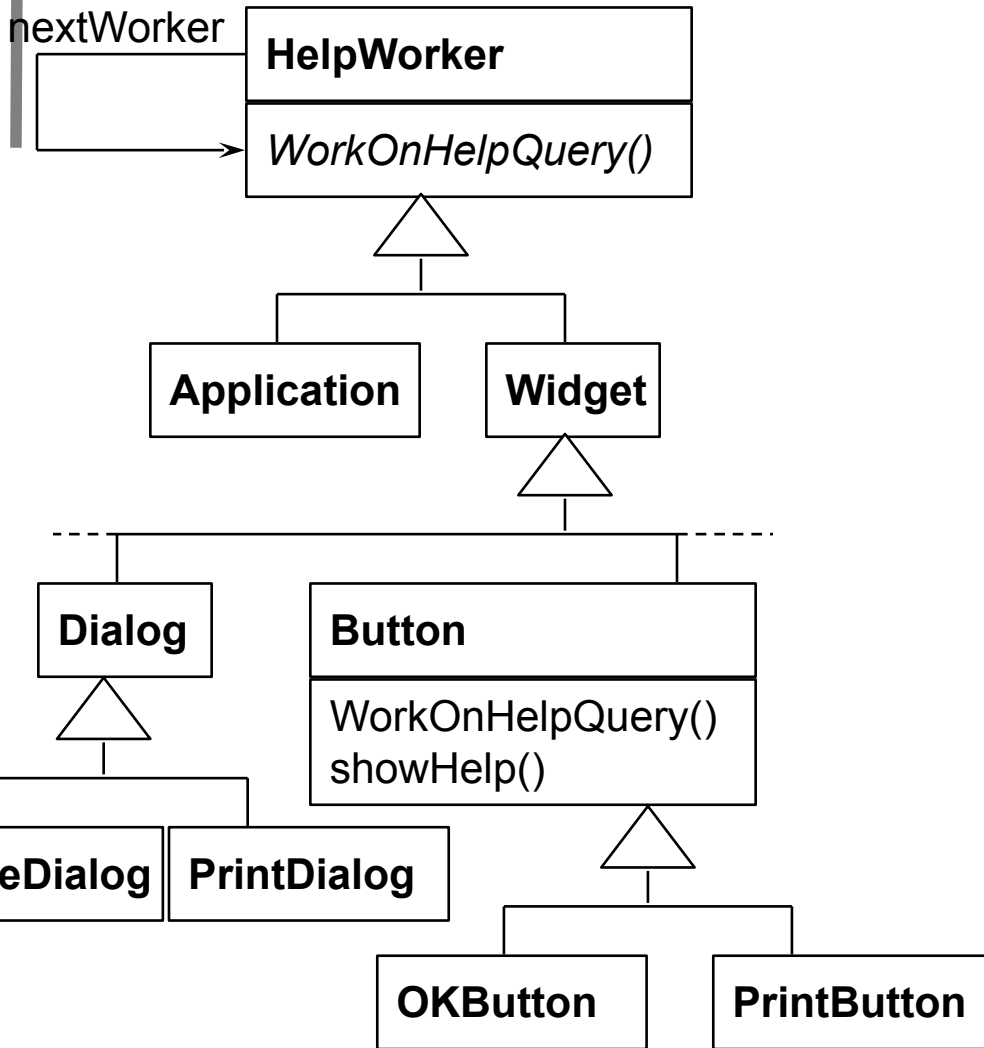


# Chains in Runtime Trees

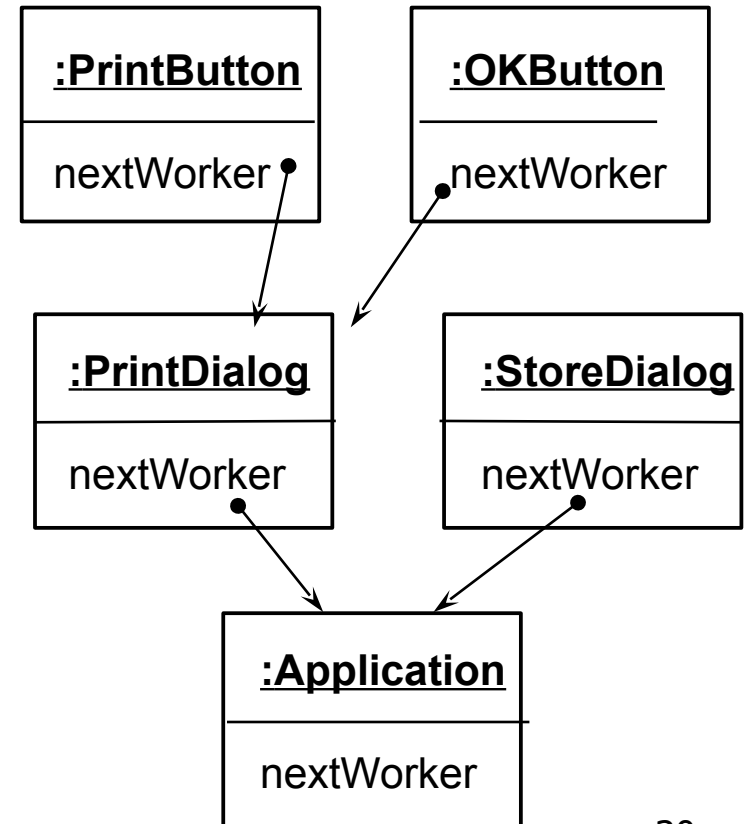
- ▶ Chains can also be parts of a tree
- ▶ Then, a chain is the path upward to the root of the tree



# Example ChainOfResponsibility Help System for a GUI



ObjectStructure:



# 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();
        }
    }
}

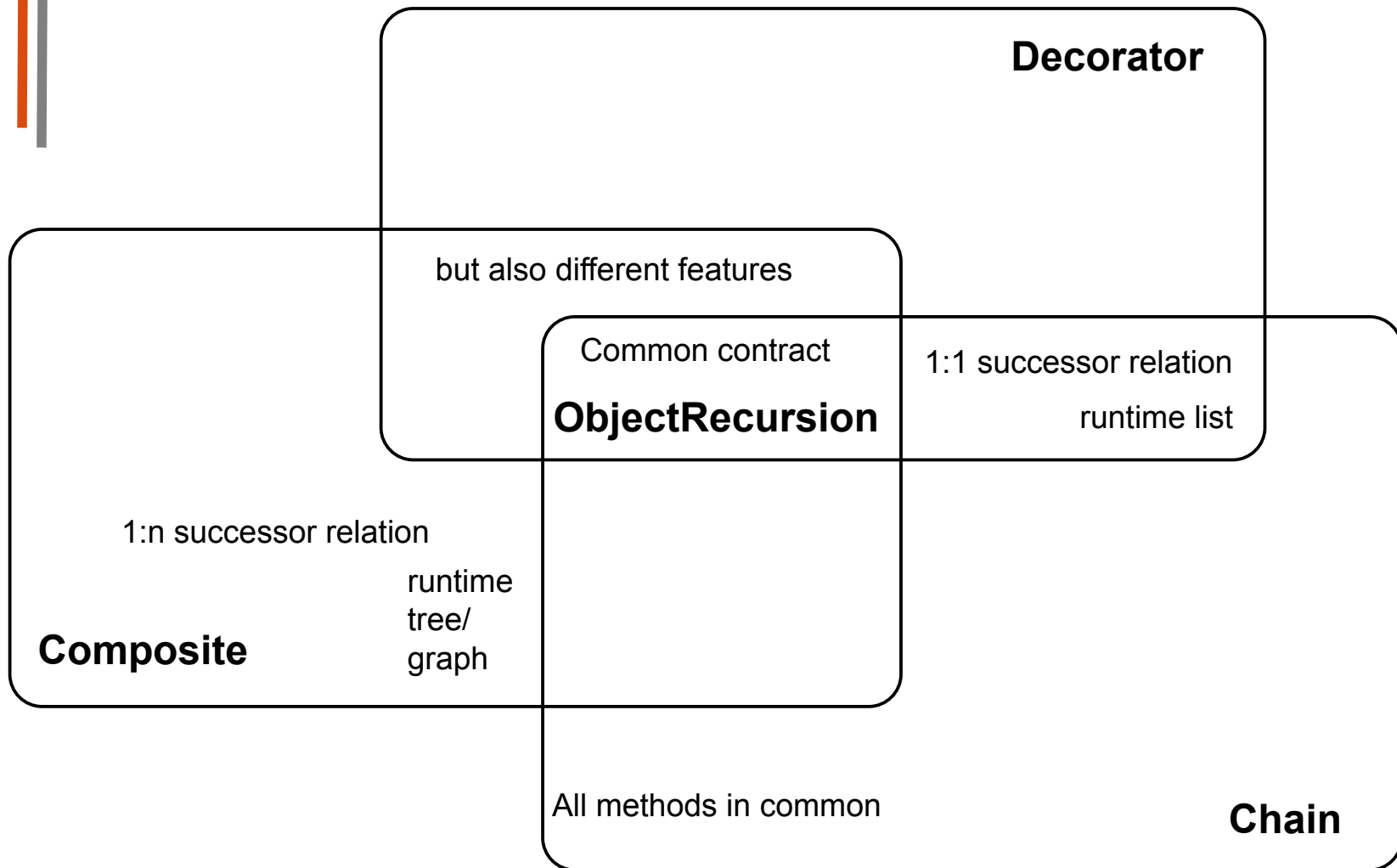
// application
button.workOnHelpQuery();
// may end in the inheritance hierarchy up in
// Widget, HelpWorker
// dynamically in application object
```

# 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



# Composite vs Decorator vs Chain





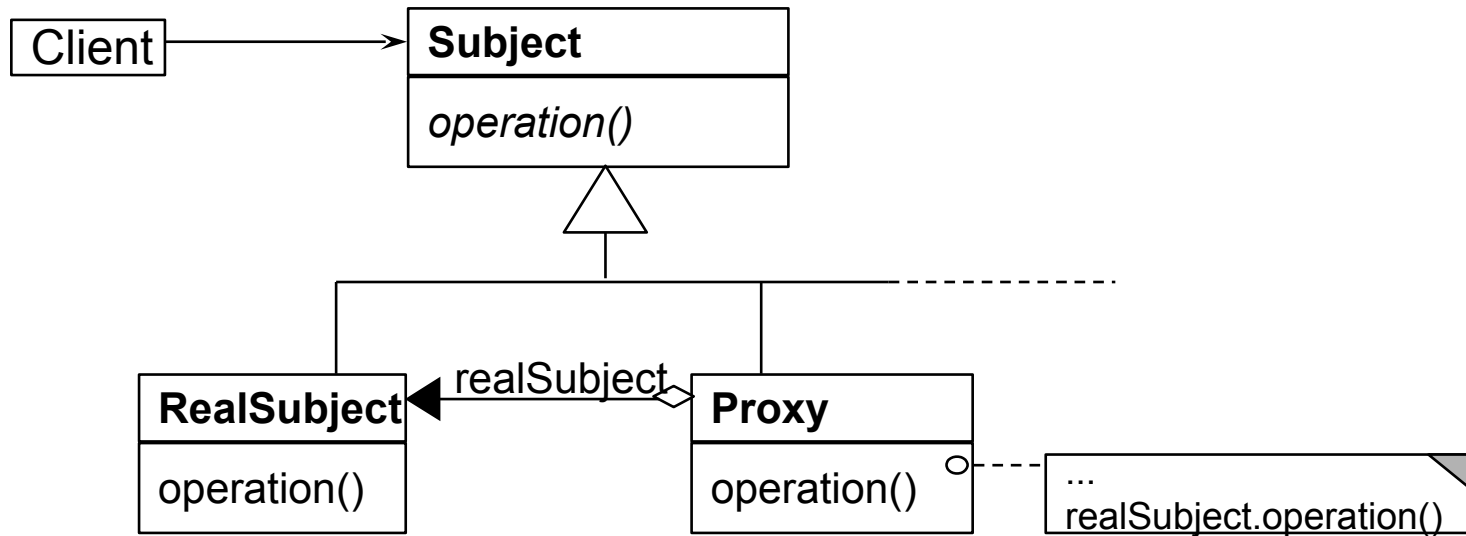
## 3.5 Proxy

---

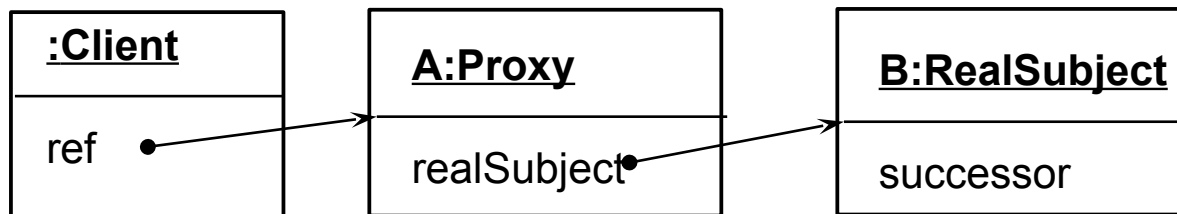
---

# 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 ObjectReursion
  - 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.

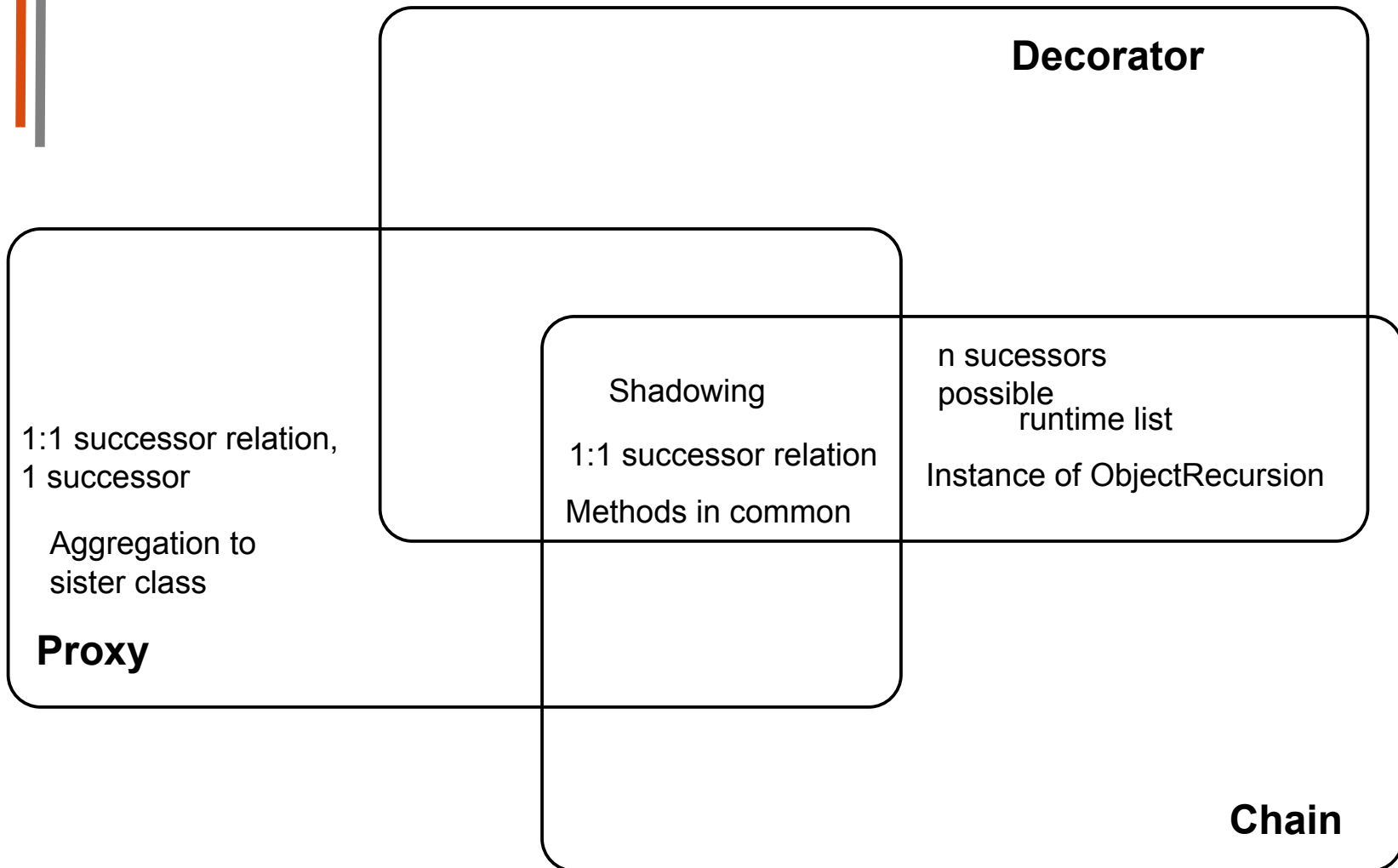
# Proxy Variants

- ▶ *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 intervene
- ▶ 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





## 3.6 \*-Bridge

---

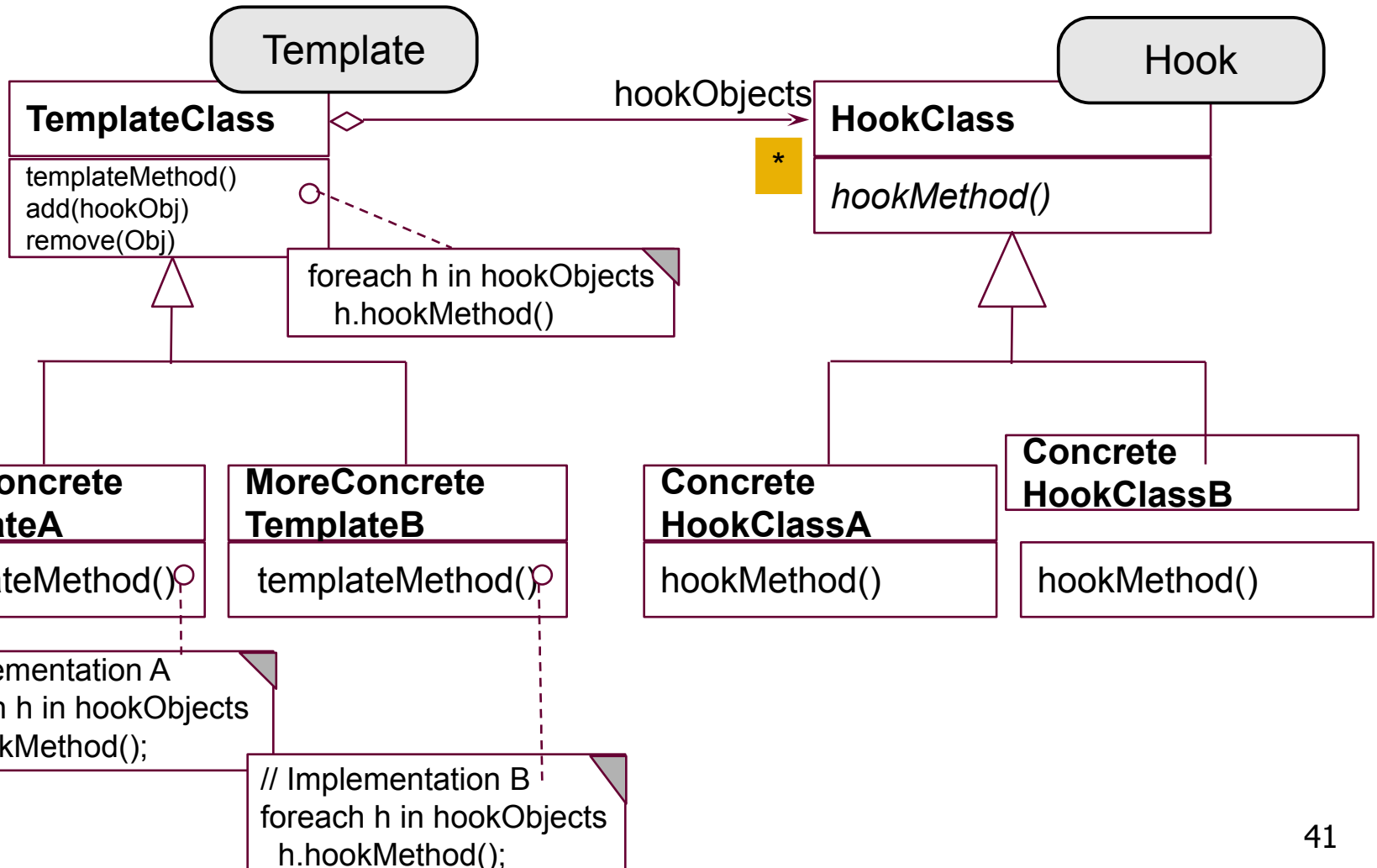
---

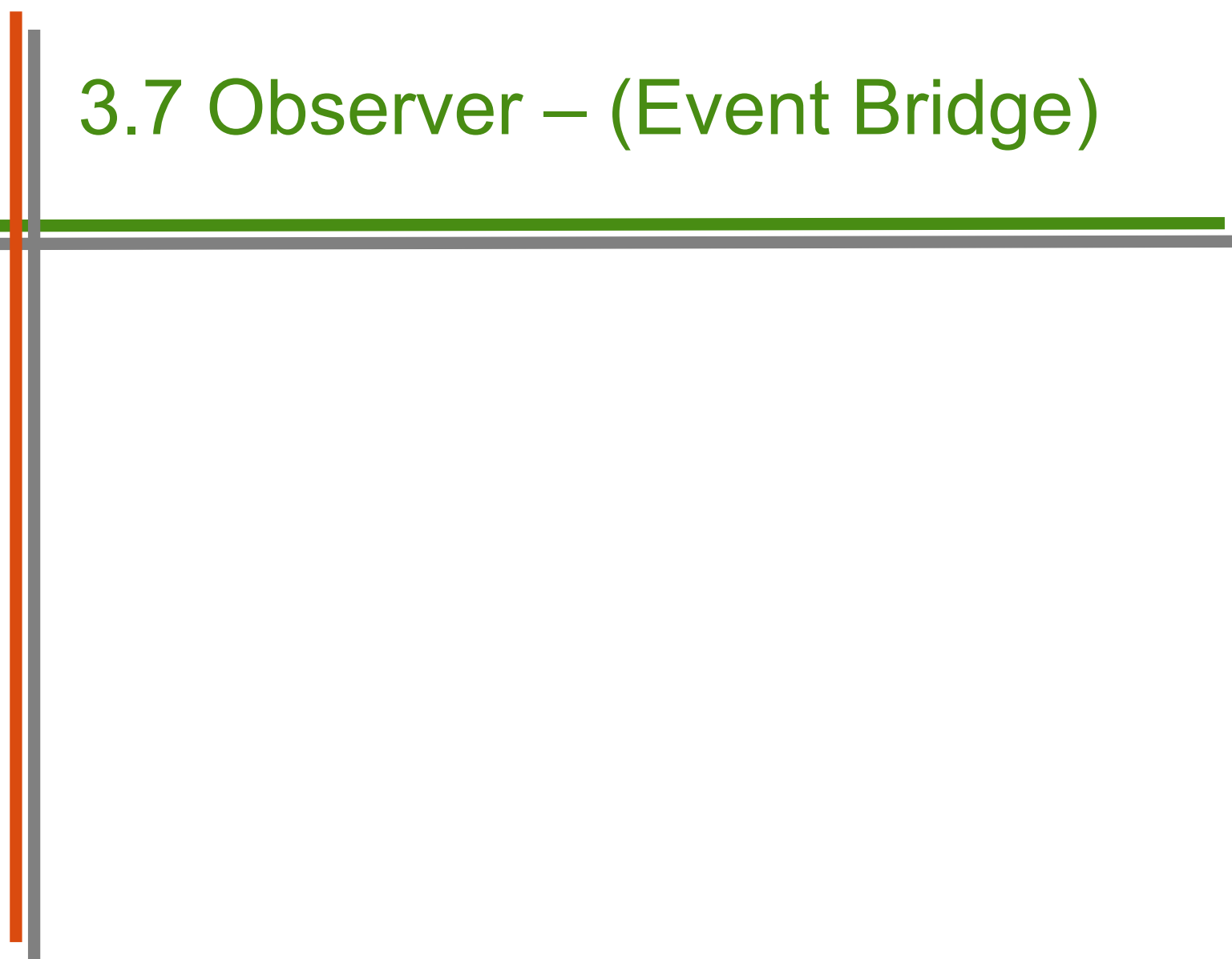


# Extensibility Pattern

## \*DimensionalClassHierarchies (\*Bridge)

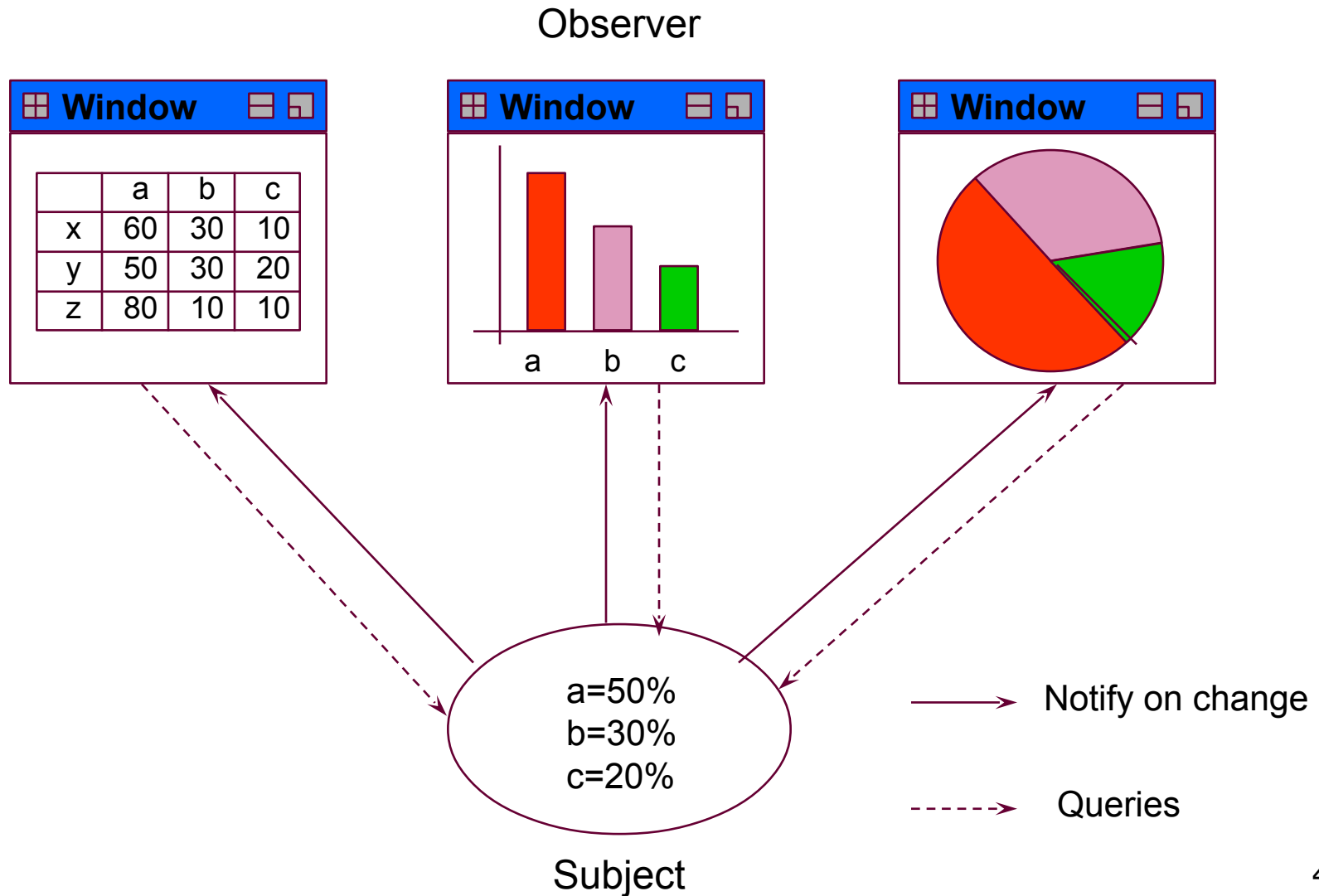
- ▶ A bridge with a collection





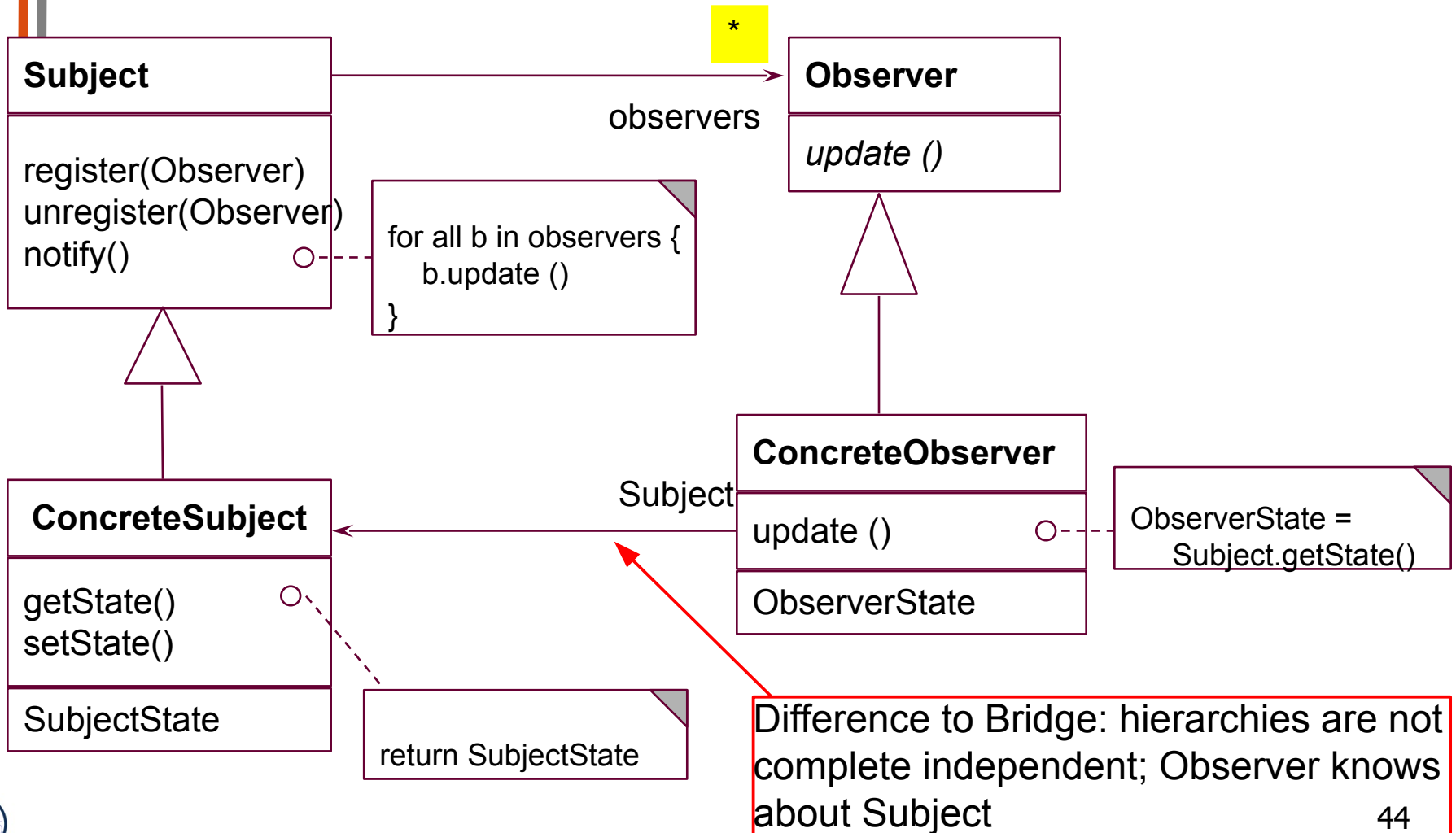
## 3.7 Observer – (Event Bridge)

# Observer (Publisher/Subscriber, Event Bridge)



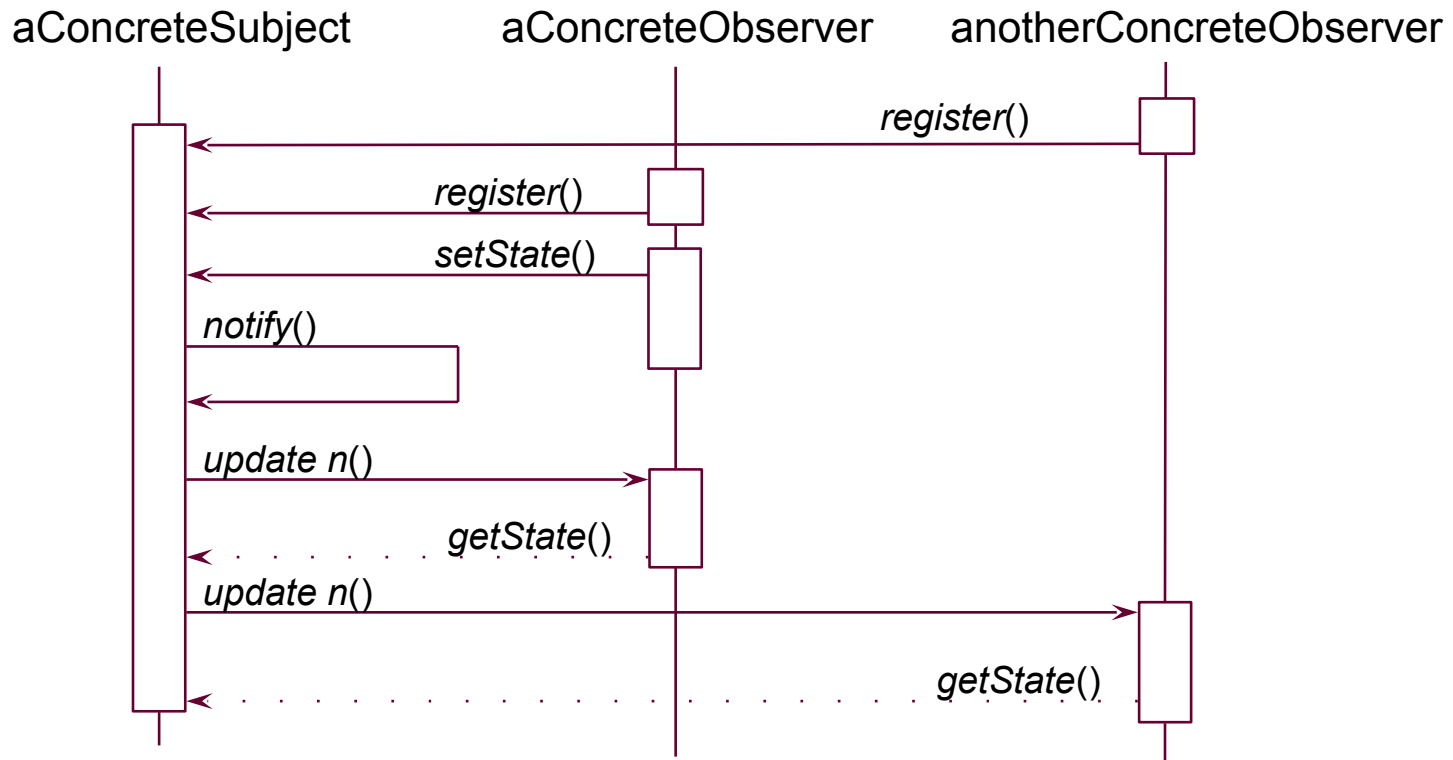
# Structure Observer

- ▶ Extension of \*-Bridge



# 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



# Observer - Applications

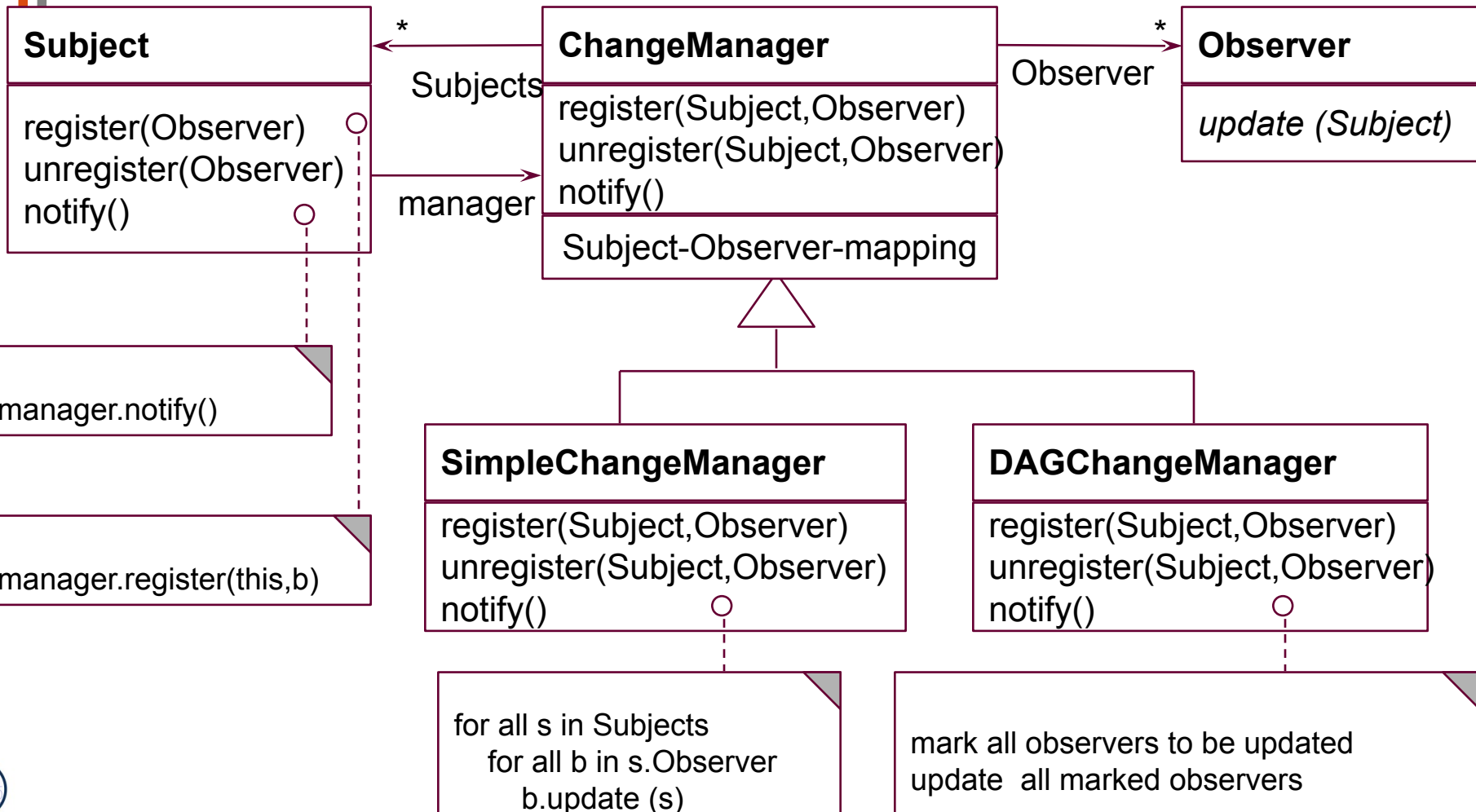
- ▶ Loose coupling in communication
  - Observers decide what happens
- ▶ Dynamic change of communication
  - Anonymous communication
  - Multi-cast and broadcast communication
  - Cascading communication if observers are chained (stacked)
- ▶ Communication of core and aspect
  - If an abstraction has 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

# 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**

# Observer with ChangeManager (Mediator)

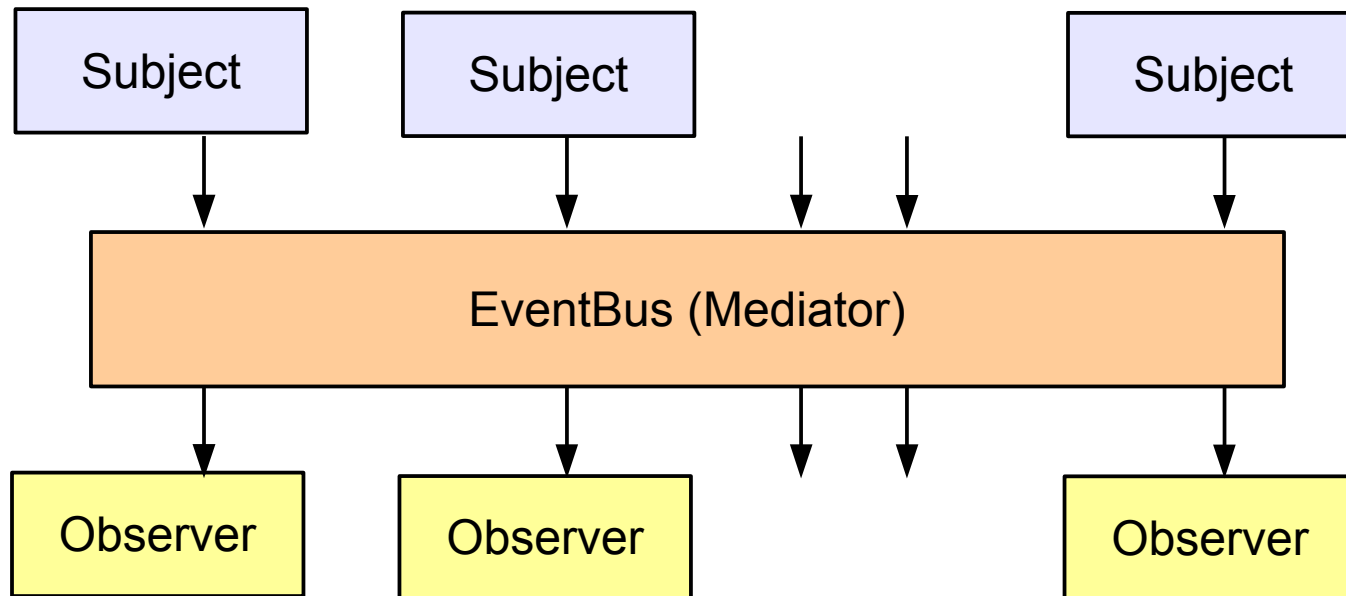
- ▶ Mediator between subjects and observer:
  - May filter events, stop cascaded propagations



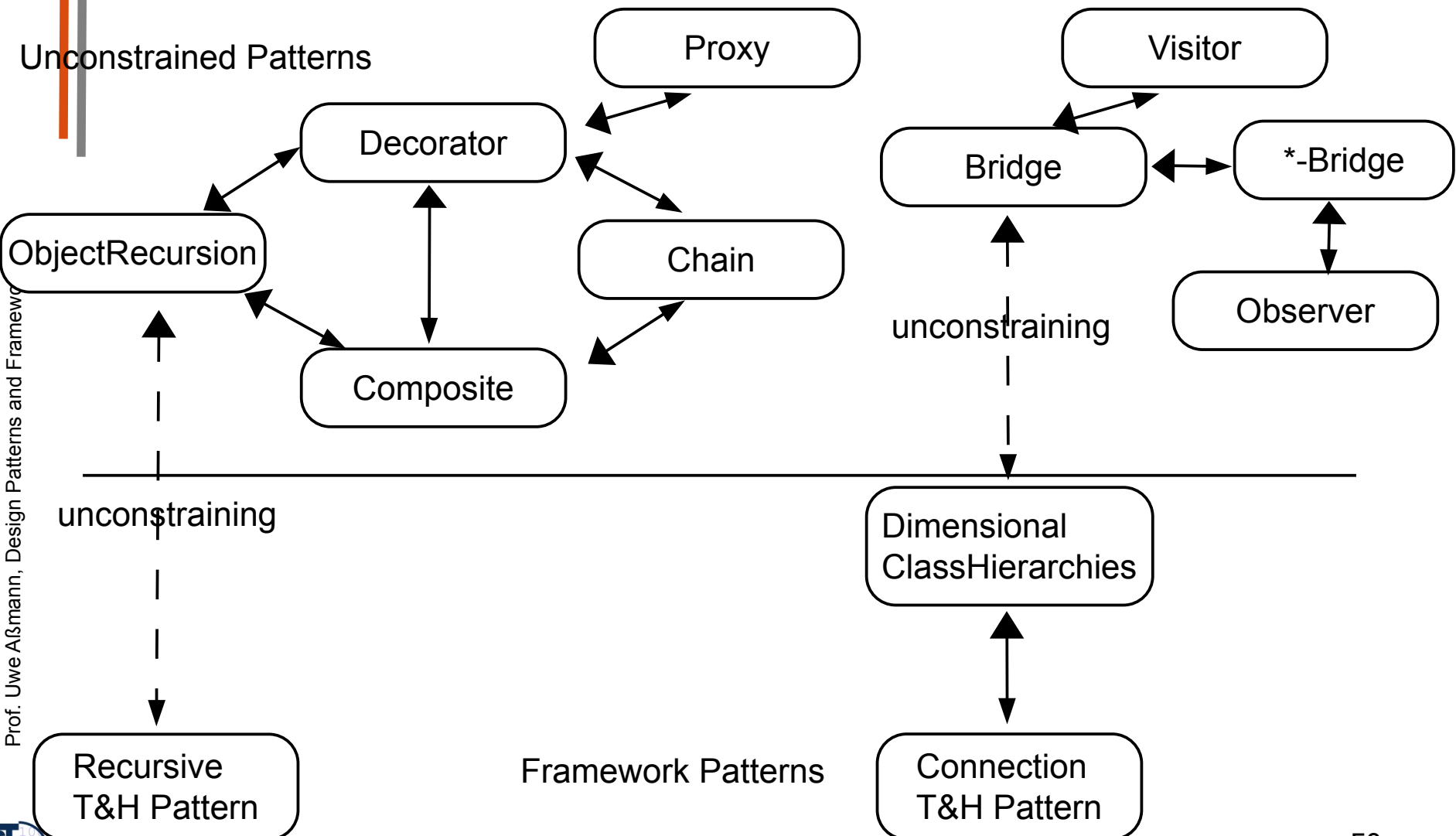


# ChangeManager is also Called Eventbus

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



# Relations Extensibility Patterns



Prof. Uwe Alßmann, Design Patterns and Frameworks



# 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

# The End