



Chapter 4

Simple Patterns for Extensibility

1

Prof. Dr. U. Aßmann

Chair for Software Engineering

Fakultät Informatik

Technische Universität Dresden

WS 16/17, Nov 7, 2016

Lecturer:

Dr. Sebastian Götz

1) Recursive Extensibility

1) Object Recursion

2) Composite

3) Decorator

4) Chain of Responsibility

2) Flat Extension

1) Proxy

2) *-Bridge

3) Observer



Literature (To Be Read)

2

- ▶ 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
<http://www.cs.unc.edu/techreports/02-011.pdf>

Optional Literature

3

- ▶ Marko Rosenmüller. Towards Flexible Feature Composition: Static and Dynamic Binding in Software Product Lines. PhD thesis, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, June 2011. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.220.8672>
- ▶ Marko Rosenmüller, Norbert Siegmund, Sven Apel, and Gunter Saake. Flexible Feature Binding in Software Product Lines. Automated Software Engineering, 18(2):163-197, June 2011. http://www.witi.cs.uni-magdeburg.de/iti_db/publikationen/ps/auto/RSAS11.pdf

Goal

4

- ▶ 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

5



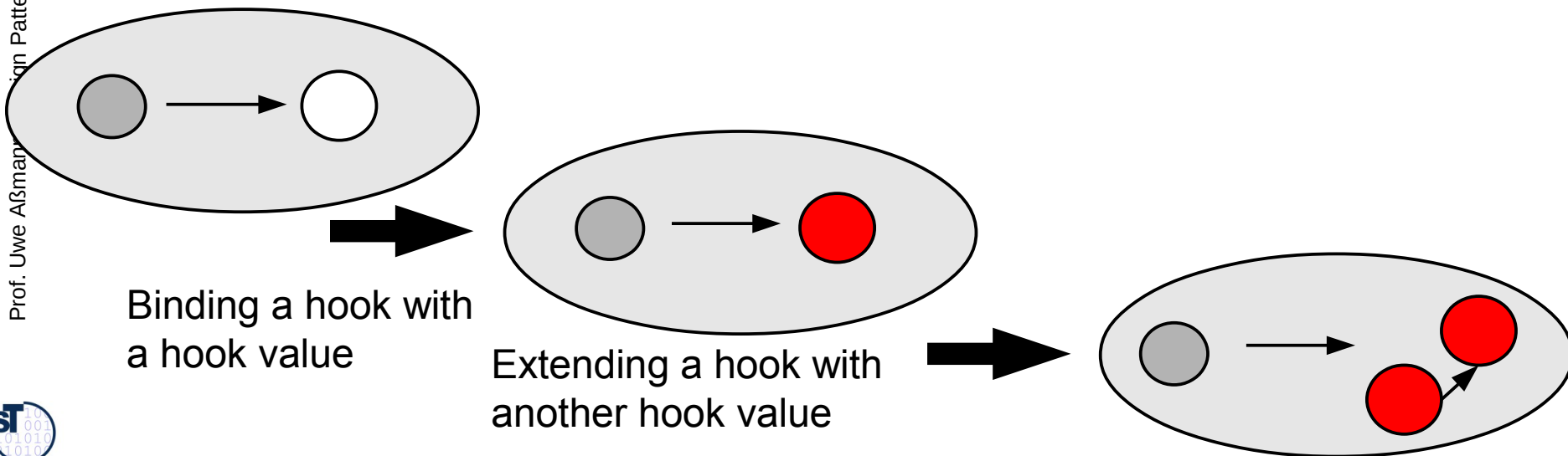
Variability vs Extensibility

6 ▶ 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





3.1 Recursive Extension



7





3.1.1 Object Recursion Pattern

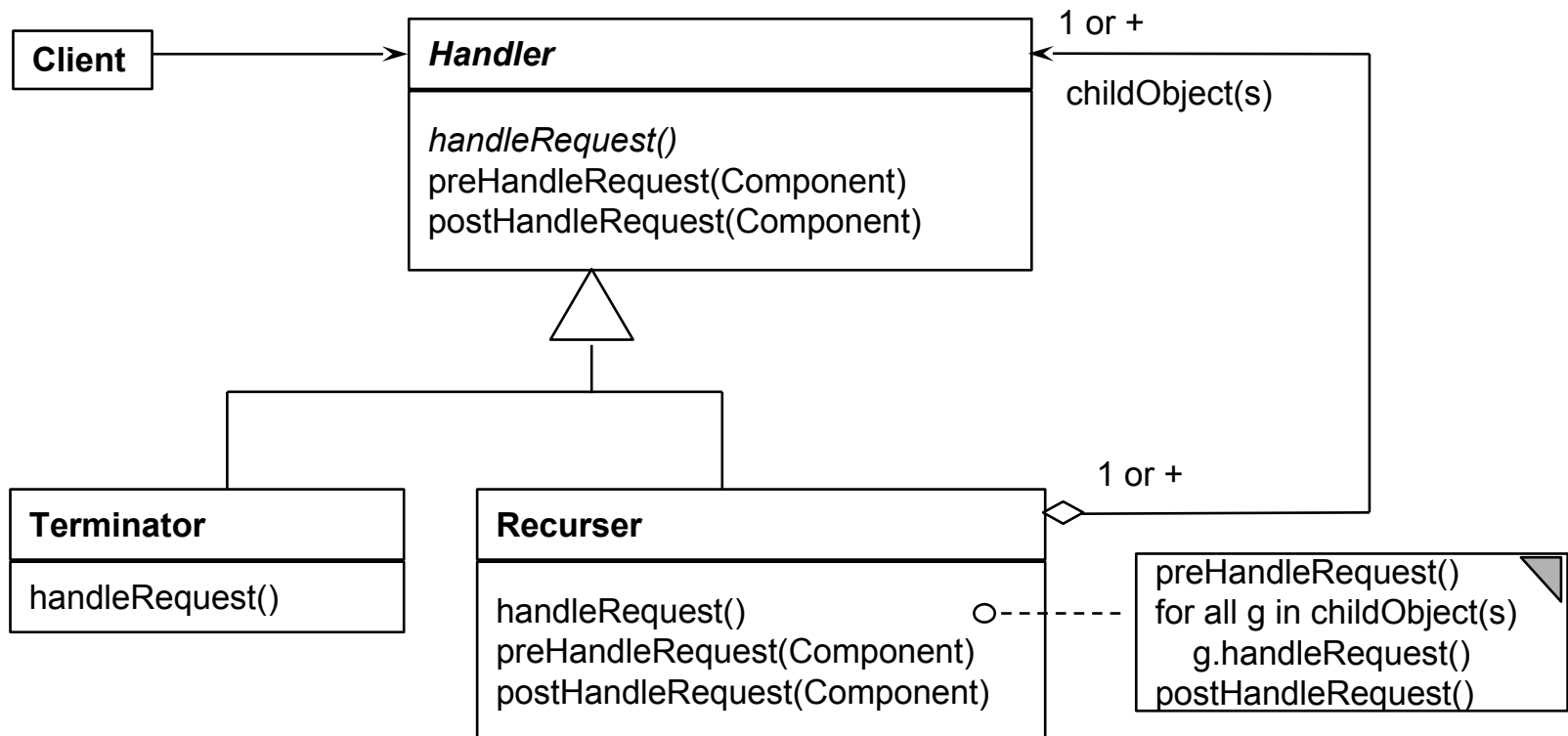
8



Object Recursion

9

- ▶ Similar to 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 (lists, 1-Recursion) or 1:* (trees, n-recursion), *:* (DAGs or graphs, n-recursion)



Incentive

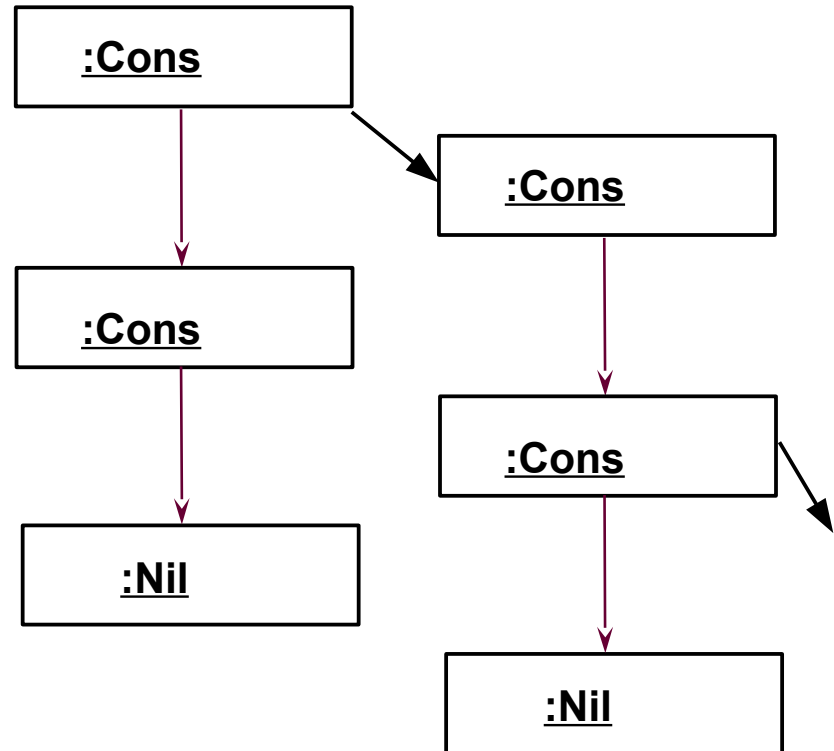
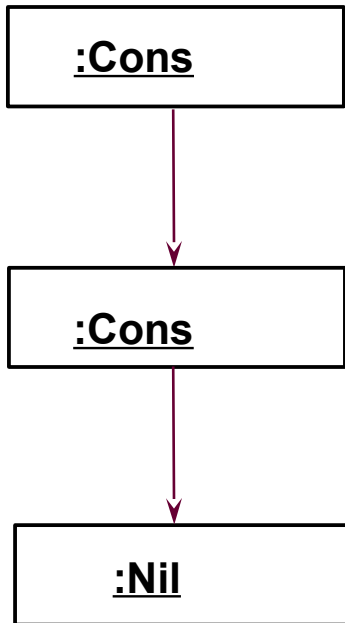
10

- ▶ 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 fulfill the common condition, they can be treated uniformly under one interface of the abstract superclass

Object Recursion – Runtime Structure

11

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



The recursion allows for building up runtime nets



3.1.2 Composite

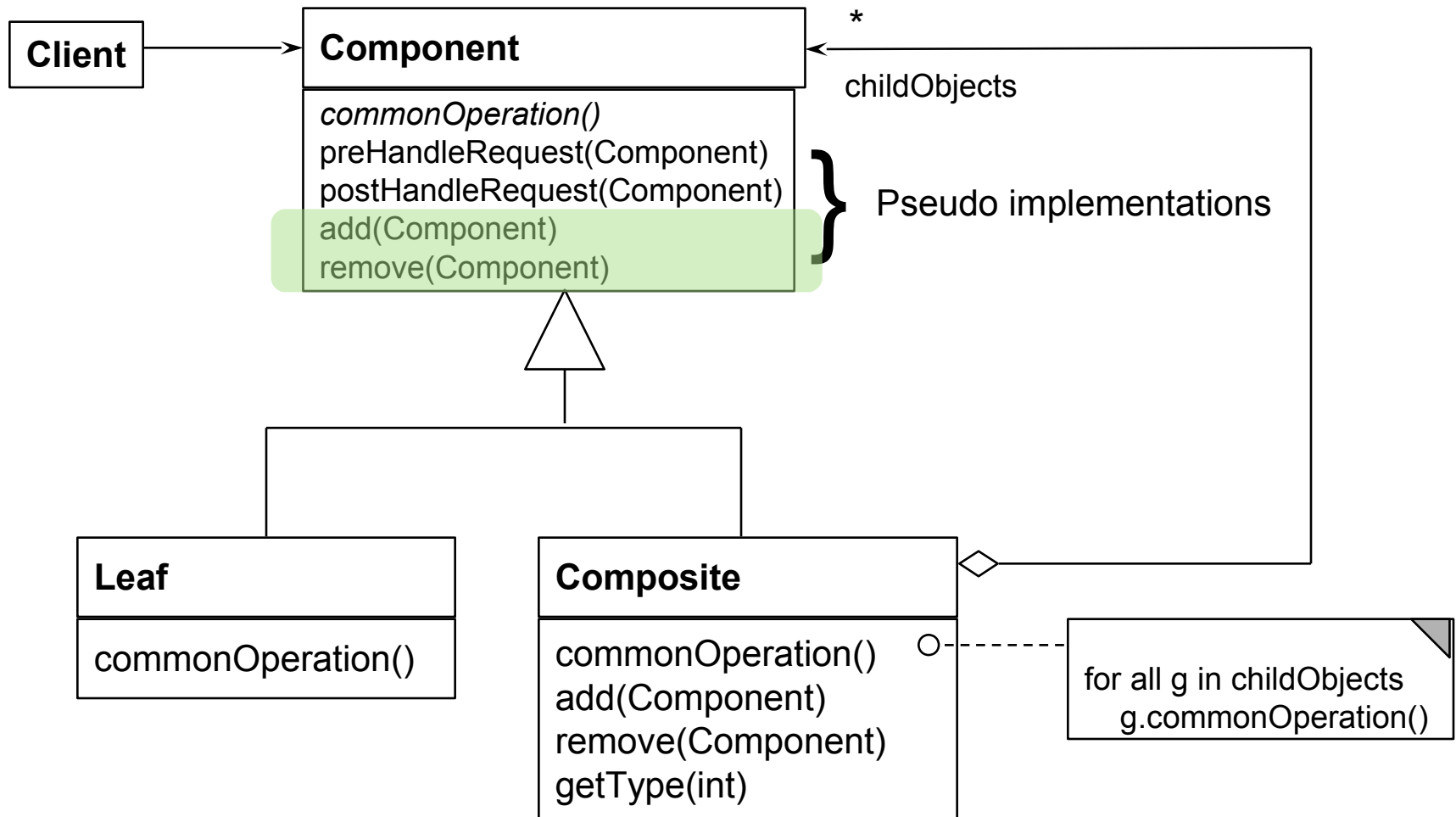
12



Structure Composite

13

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



Piece Lists in Production Data

14

```
abstract class CarPart {
    int myCost;
    abstract int calculateCost();
}
class ComposedCarPart extends CarPart {
    int myCost = 5;
    // here is the n-recursion
    CarPart [] children;
    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;
    }
}
// application
int cost = carPart.calculateCost();
```

Purpose

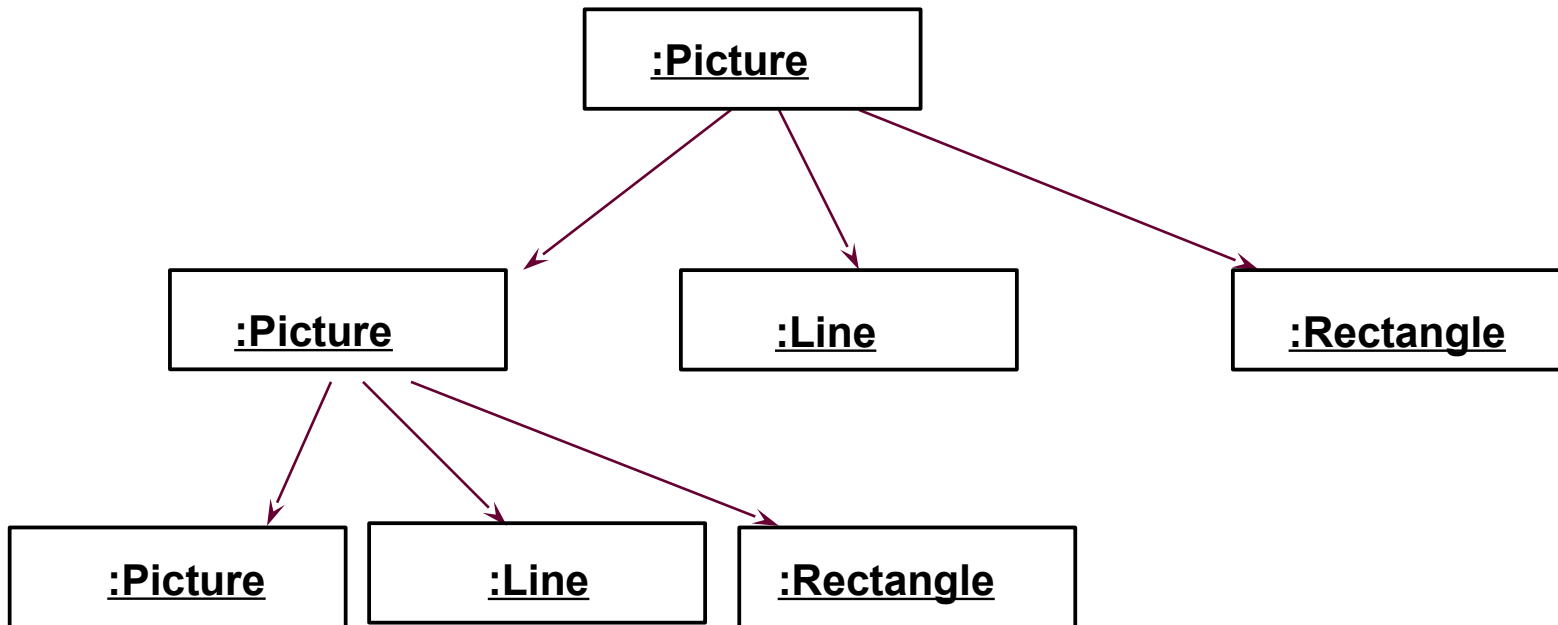
15

- ▶ 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 fulfill 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 does not need to know whether it works on a leaf or an 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

16

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



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

Dynamic, Recursive Extensibility of Composite

17

- ▶ 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

Relations of Composite to Other Programming Domains

18

- ▶ 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...)
- ▶ The Composite is also the heart of attributed trees and attribute grammars
 - Ordered AG are constraint systems that generate iterators and skeletons [CompilerConstruction]



3.1.3 Decorator

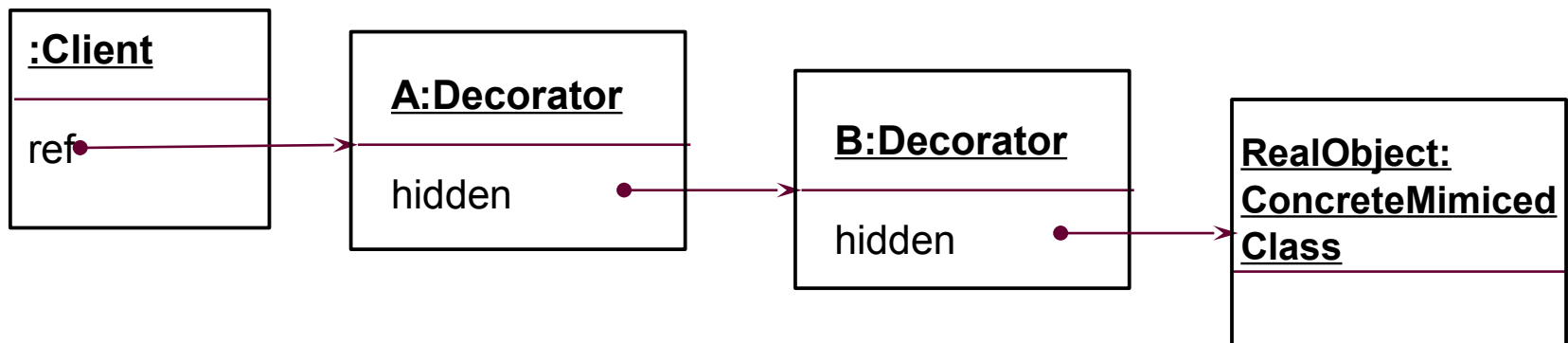
19

...as a Variant of ObjectRecursion
and Composite

Decorator Pattern

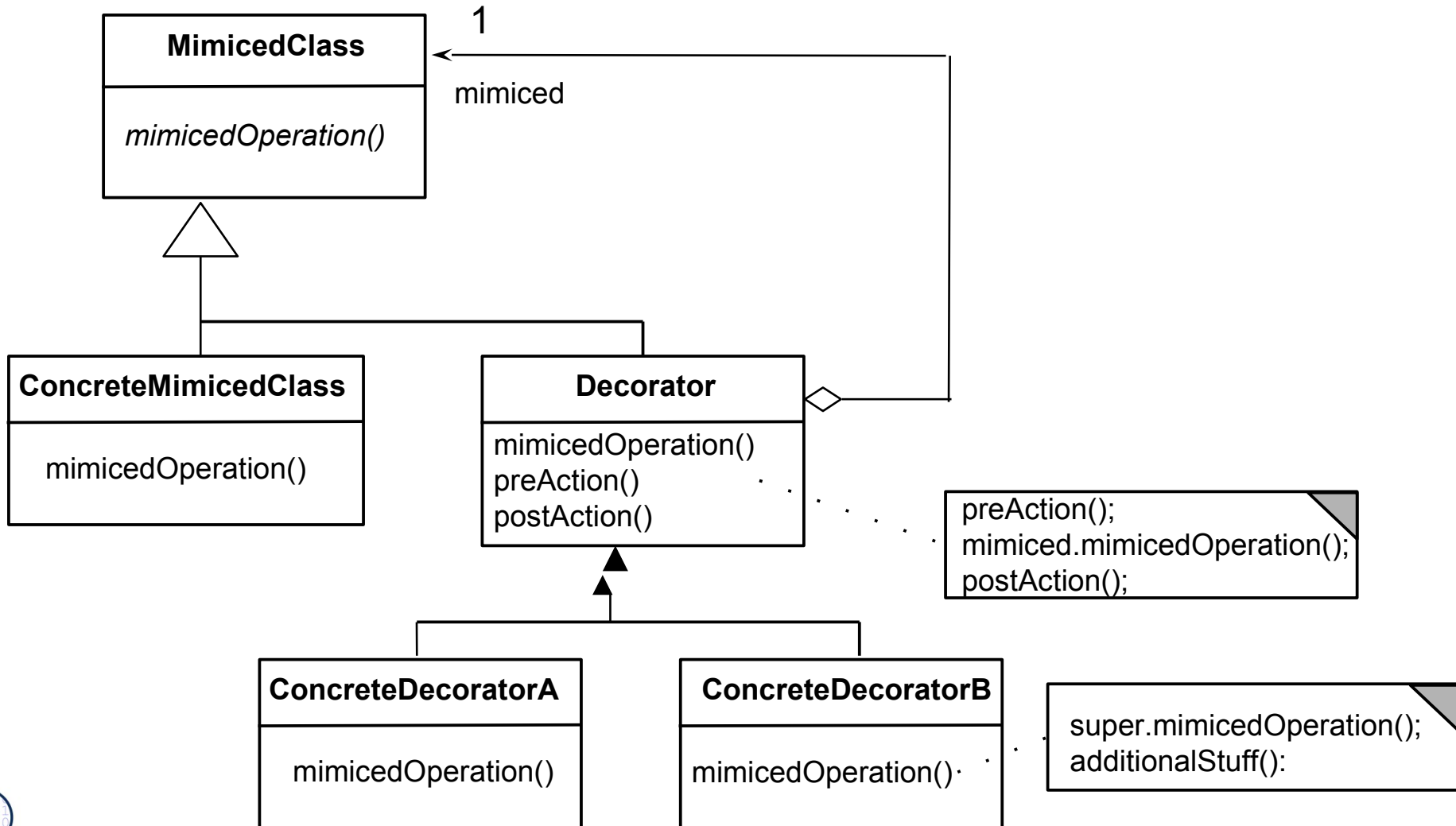
20

- ▶ A Decorator is a *skin (wrappers)* of another object
 - Core objects are at the end of a decorator chain
- ▶ 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 mimicked and the mimicing class



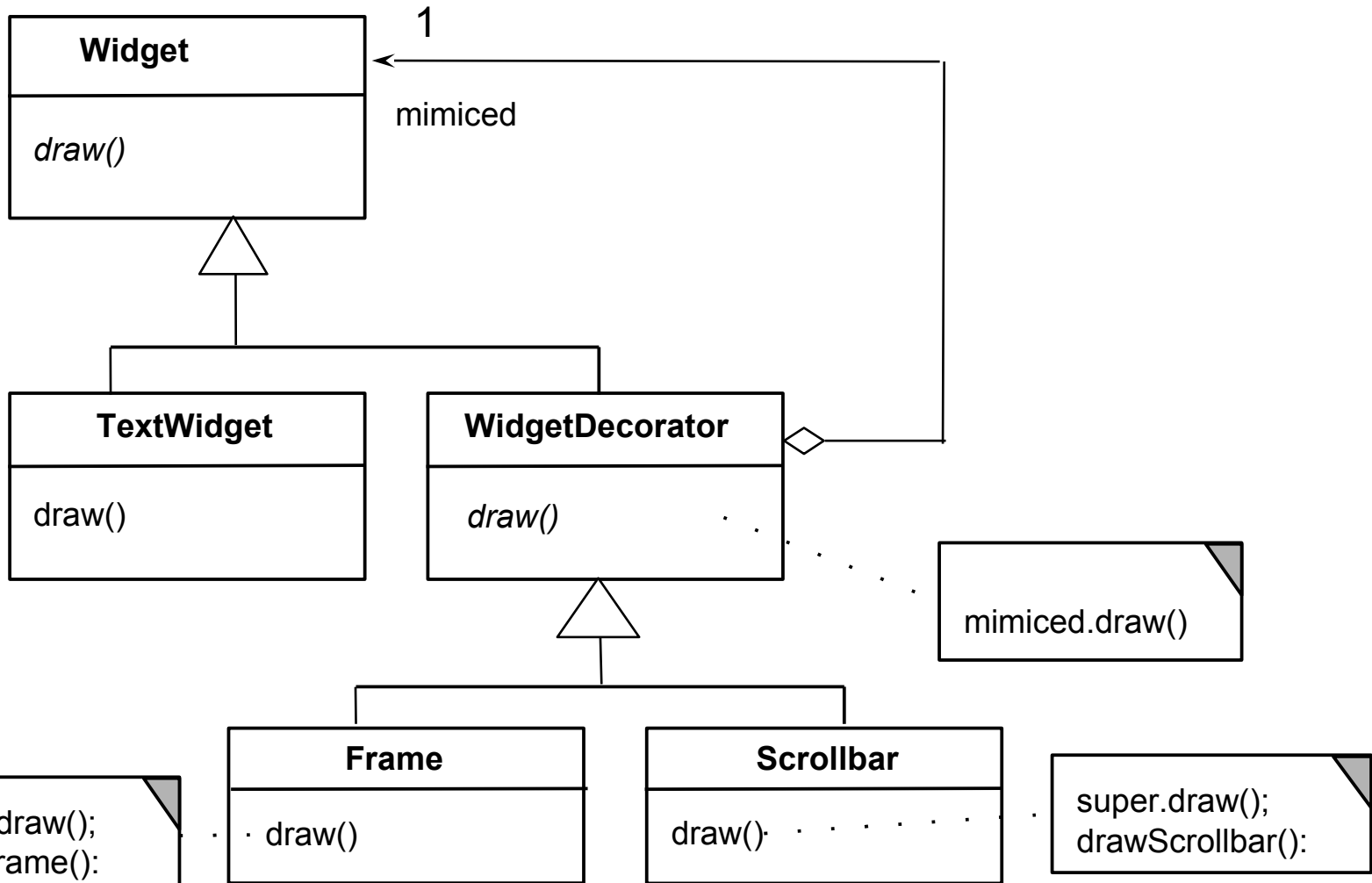
Decorator – Structure Diagram

21



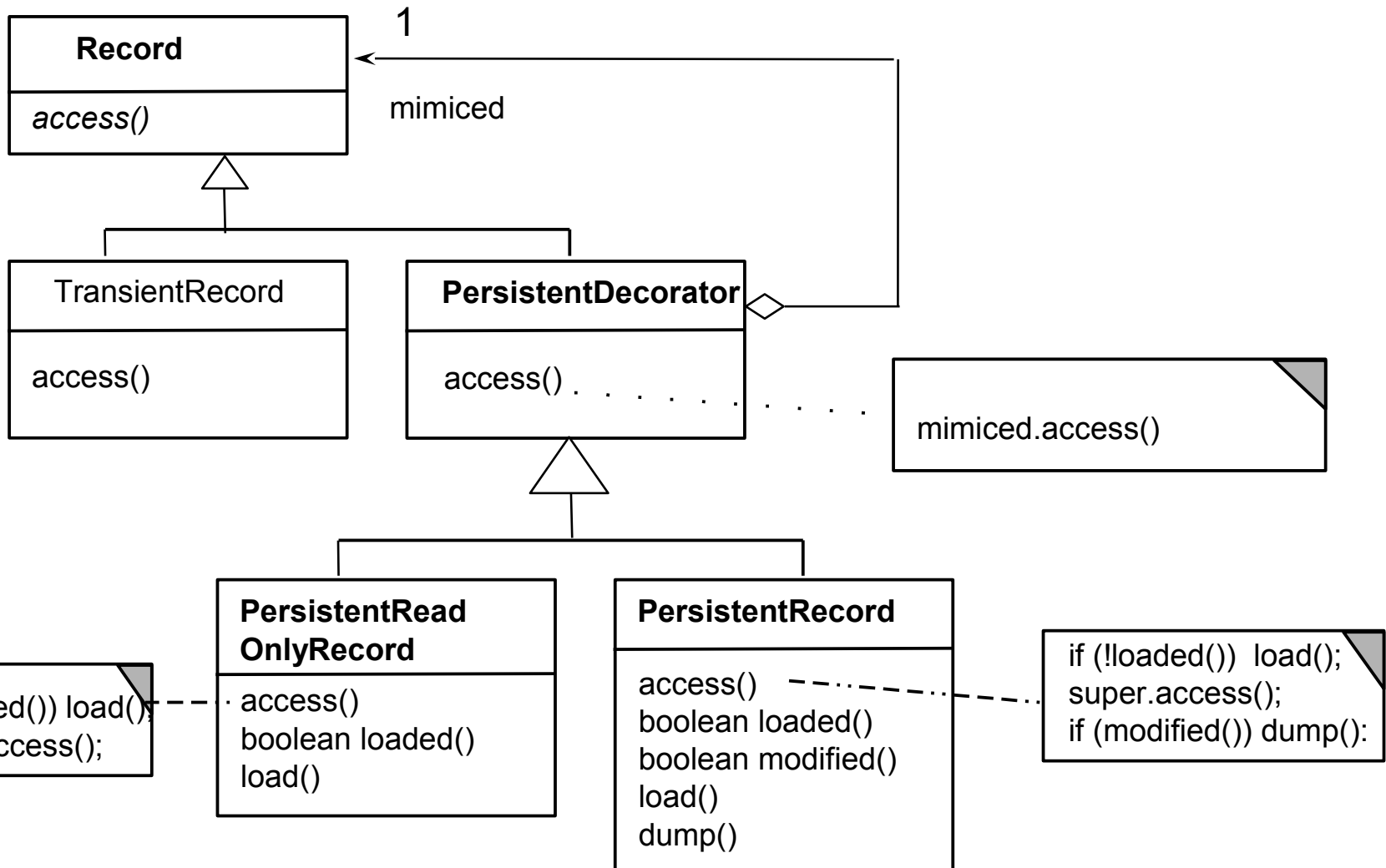
Decorator for Widgets

22



Decorator for Persistent Objects

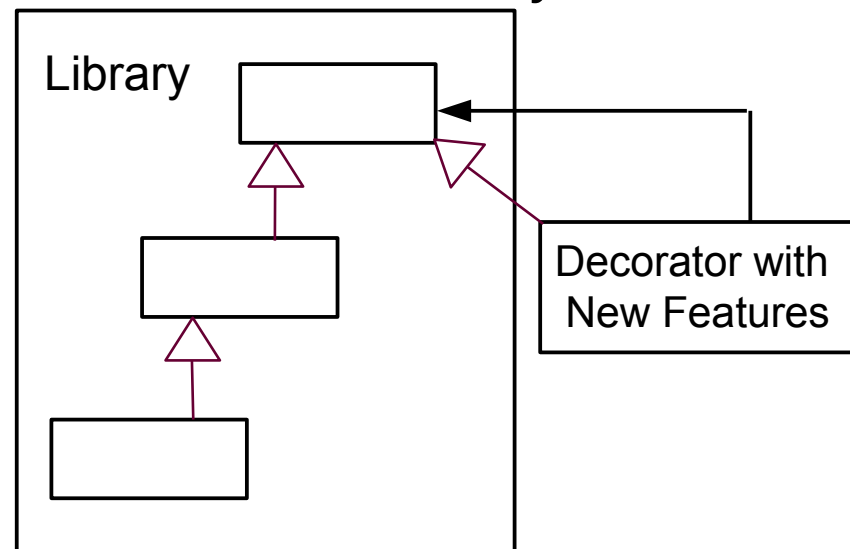
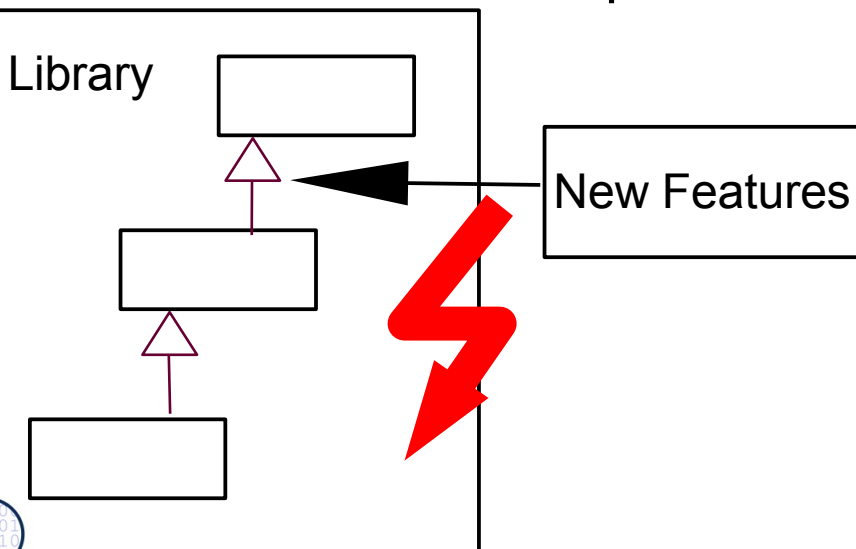
23



Purpose Decorator

24

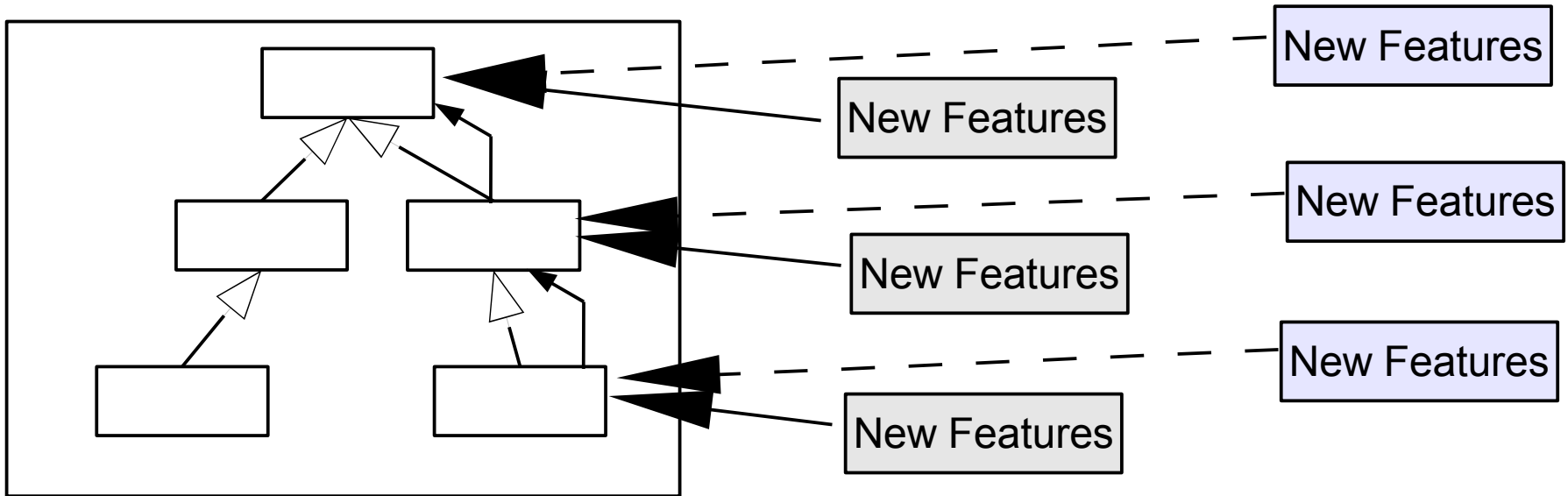
- ▶ For extensible objects (i.e., decorating objects)
 - Extension of new features at runtime
 - Removal possible
- ▶ Instead of putting the extension into the inheritance hierarchy
 - If that would become too complex
 - If that is not possible since it is hidden in a library



Variants of Decorators

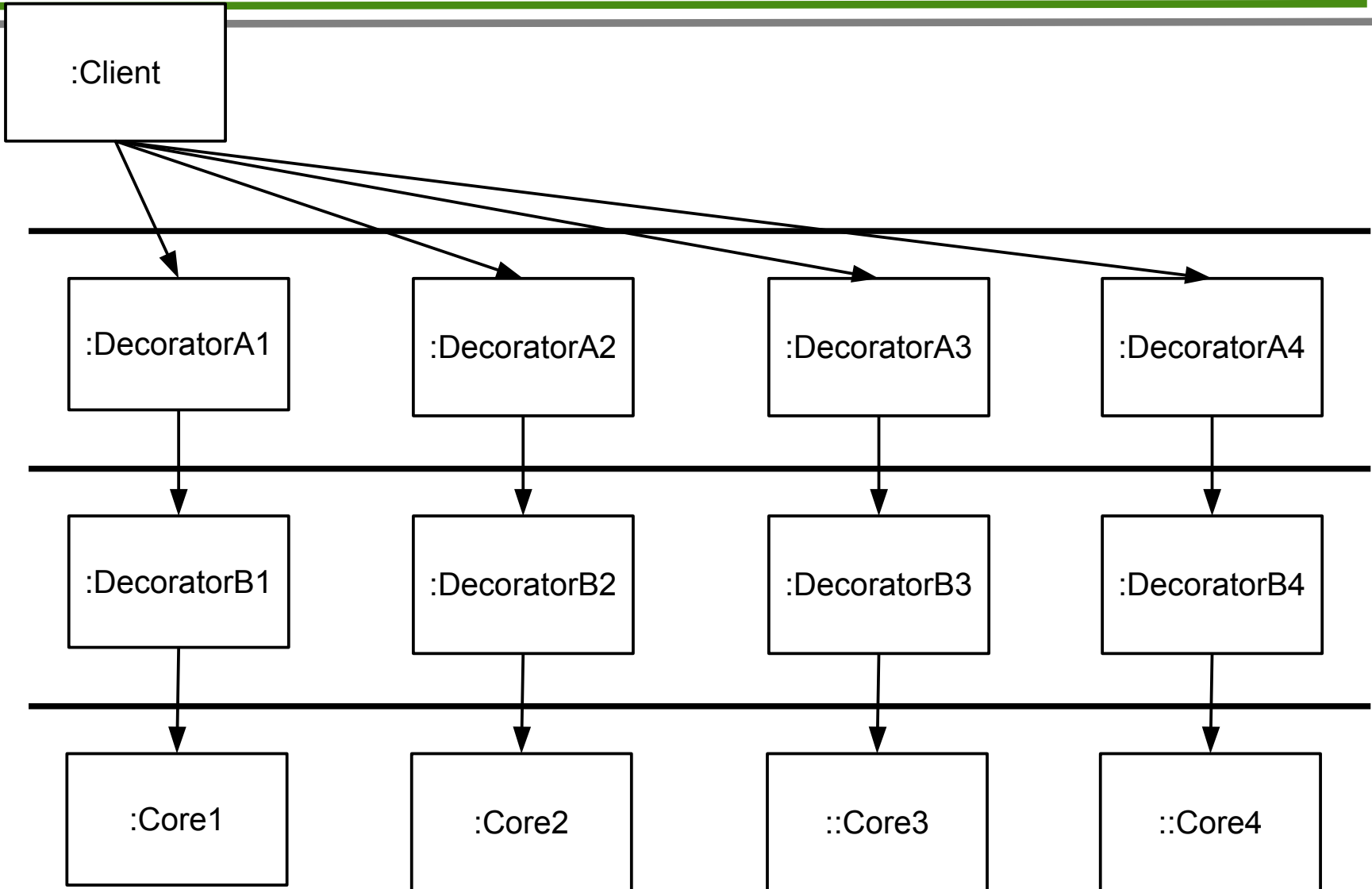
25

- ▶ 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



Decorator Layers

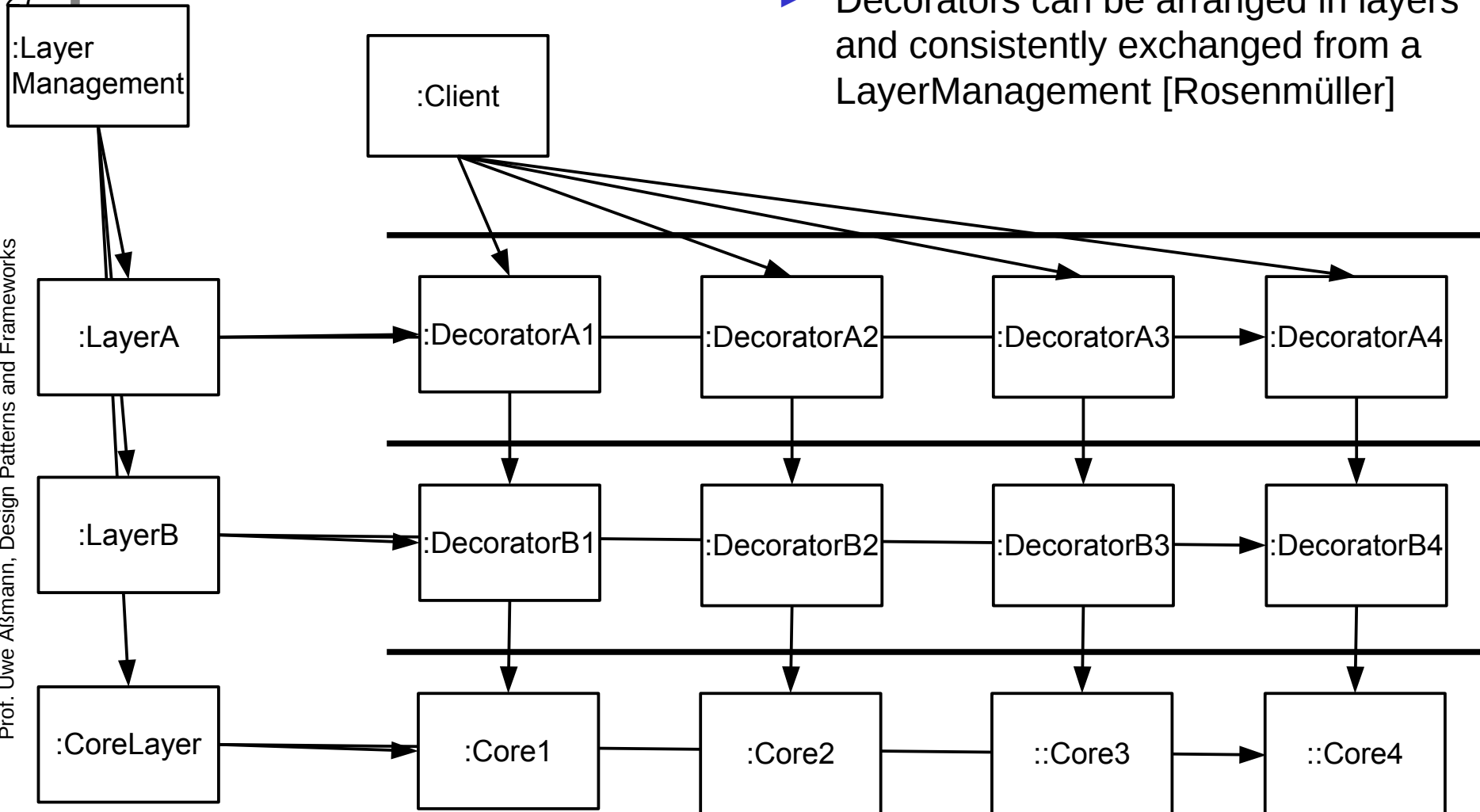
26



Decorator Layers

27

- Decorators can be arranged in layers and consistently exchanged from a LayerManagement [Rosenmüller]





3.1.3.1 Augmentor

28

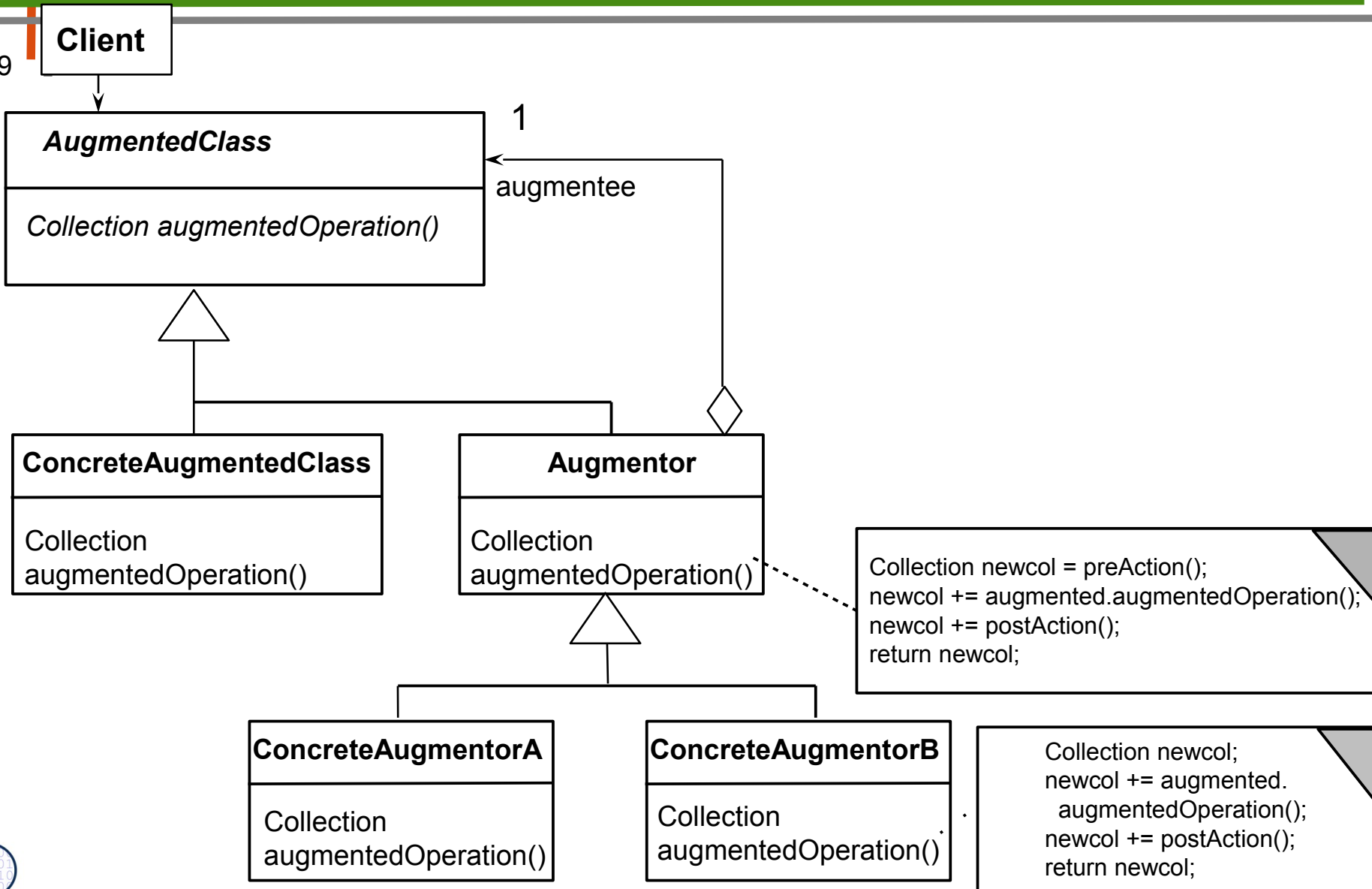
The Augmentor pattern is a Decorator enriching the behavior of the recursive method by assembling a return parameter, a Collection.

Advantage: Collecting a collection, set, or list of items from a carrier data structure can be extended from outside



Augmentor – Structure Diagram

29





3.1.4 Chain of Responsibility

30

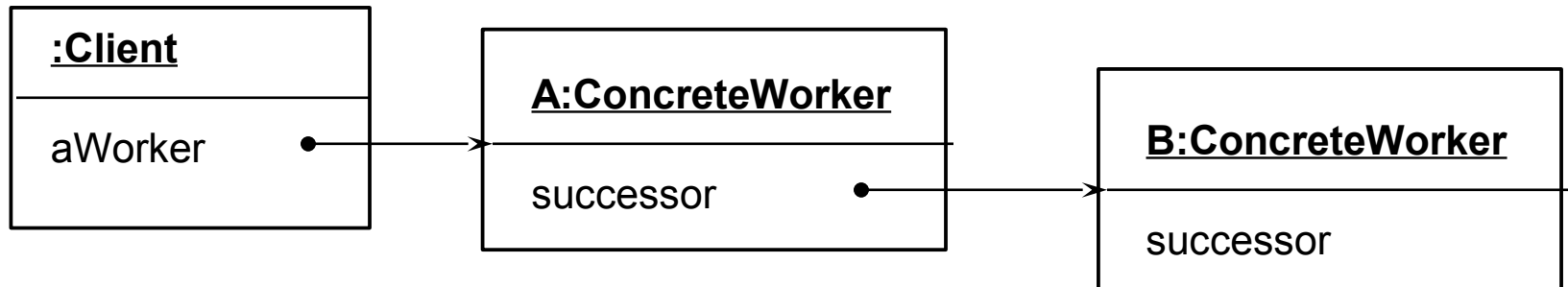


Chain of Responsibility

31

- ▶ Delegate an action to a list of delegates that attempt to solve the problem one after the other
 - They delegate further on, down the chain
 - No core object

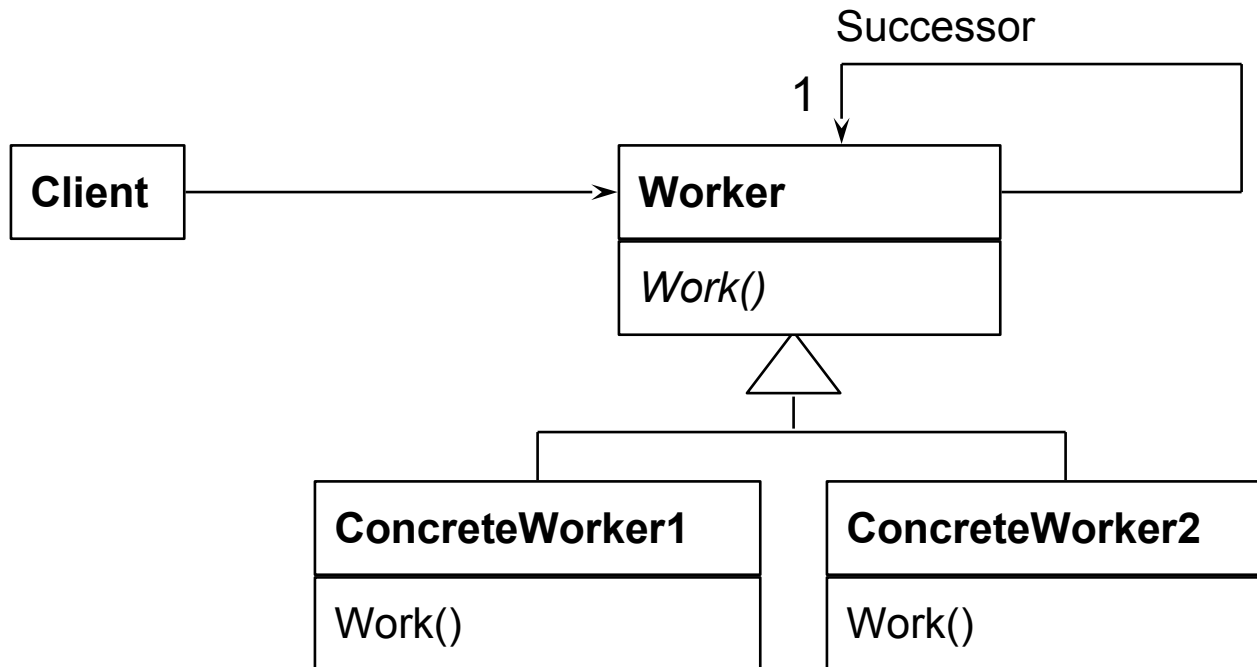
ObjectStructure:



Structure for ChainOfResponsibility

32

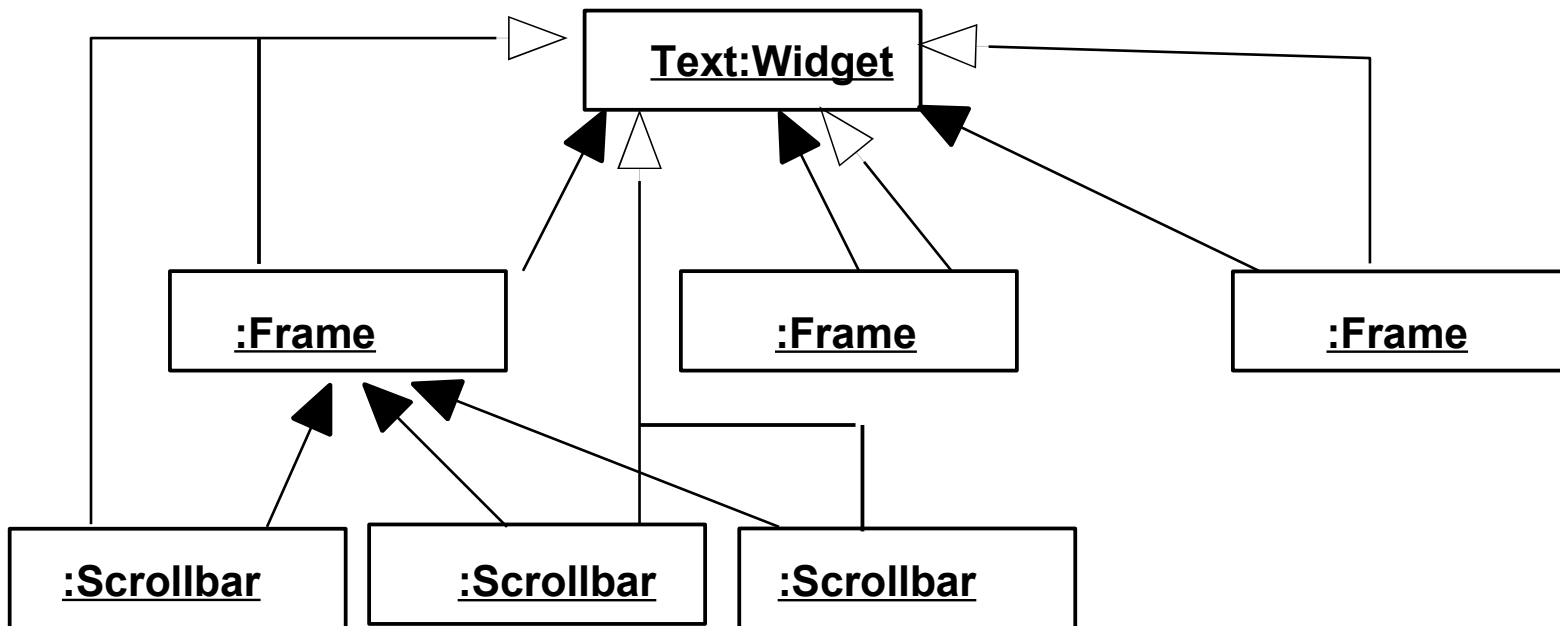
- ▶ A Chain is recursing on the abstract super class, i.e.,
 - All classes in the inheritance tree know they hide some other class (unlike the ObjectRecursion)



Chains in Runtime Trees

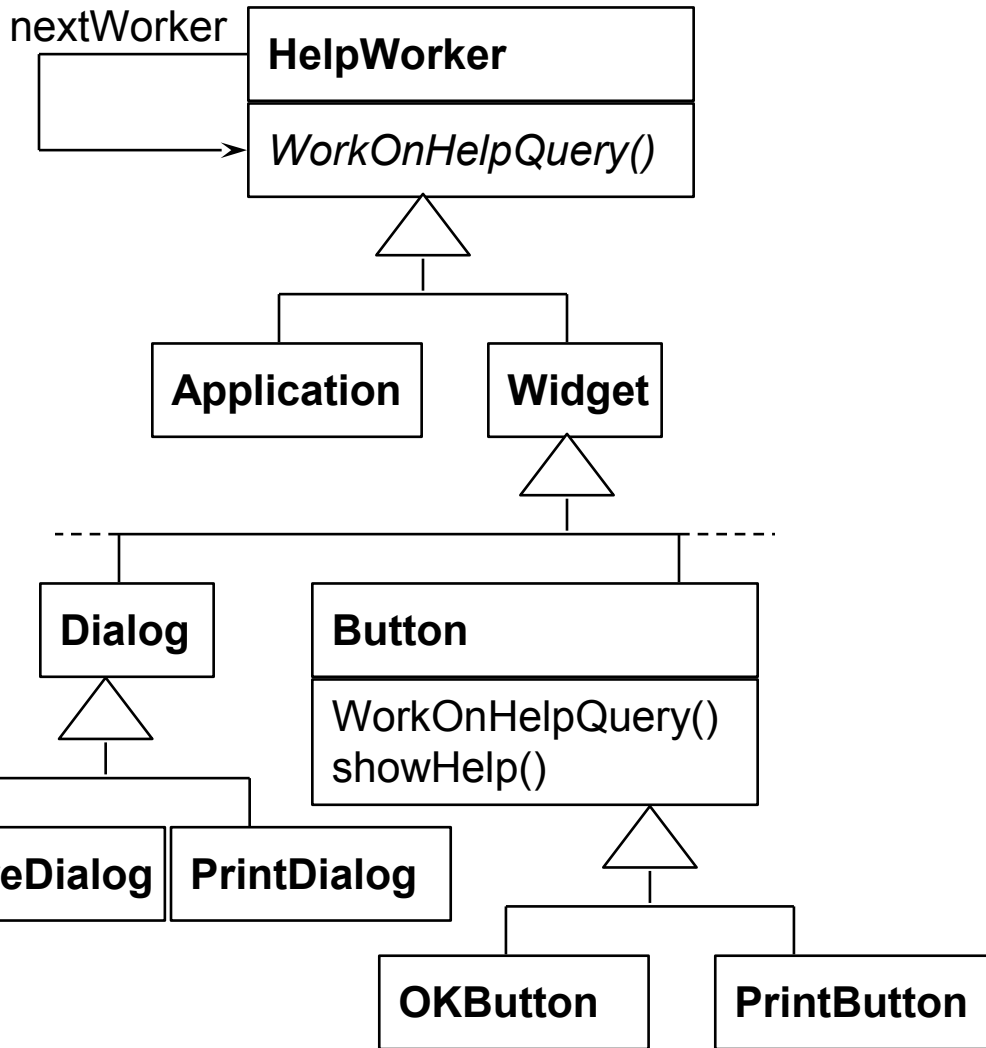
33

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

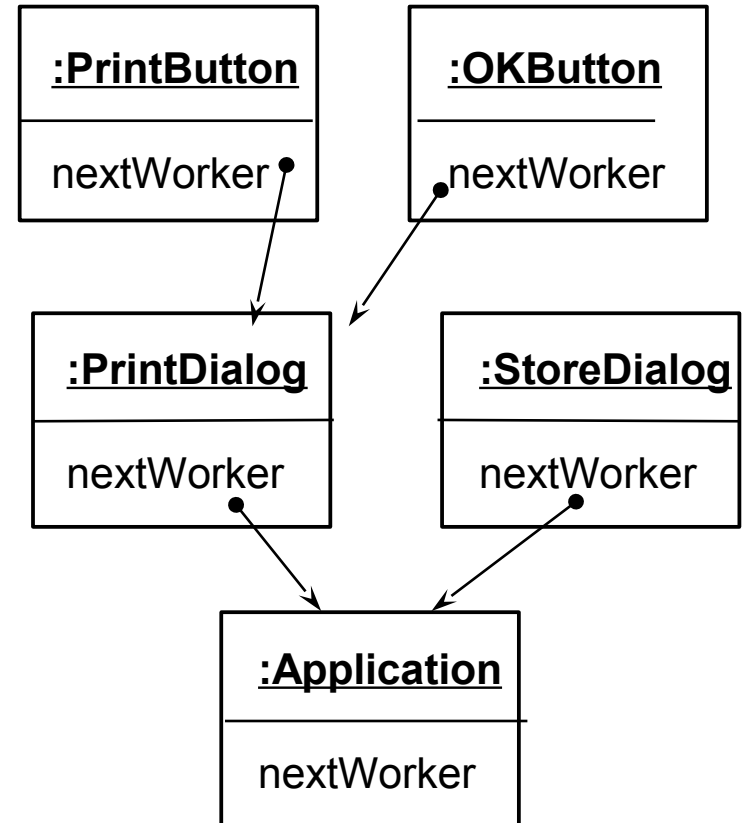


Example ChainOfResponsibility Help System for a GUI

34



ObjectStructure is a Tree of Help Functions:



Help System with Chain

35

```
abstract class HelpWorker {
    // here is the 1-recursion
    HelpWorker nextWorker;
    void workOnHelpQuery() {
        if (nextWorker != null)
            nextWorker.workOnHelpQuery();
    }
}

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
```

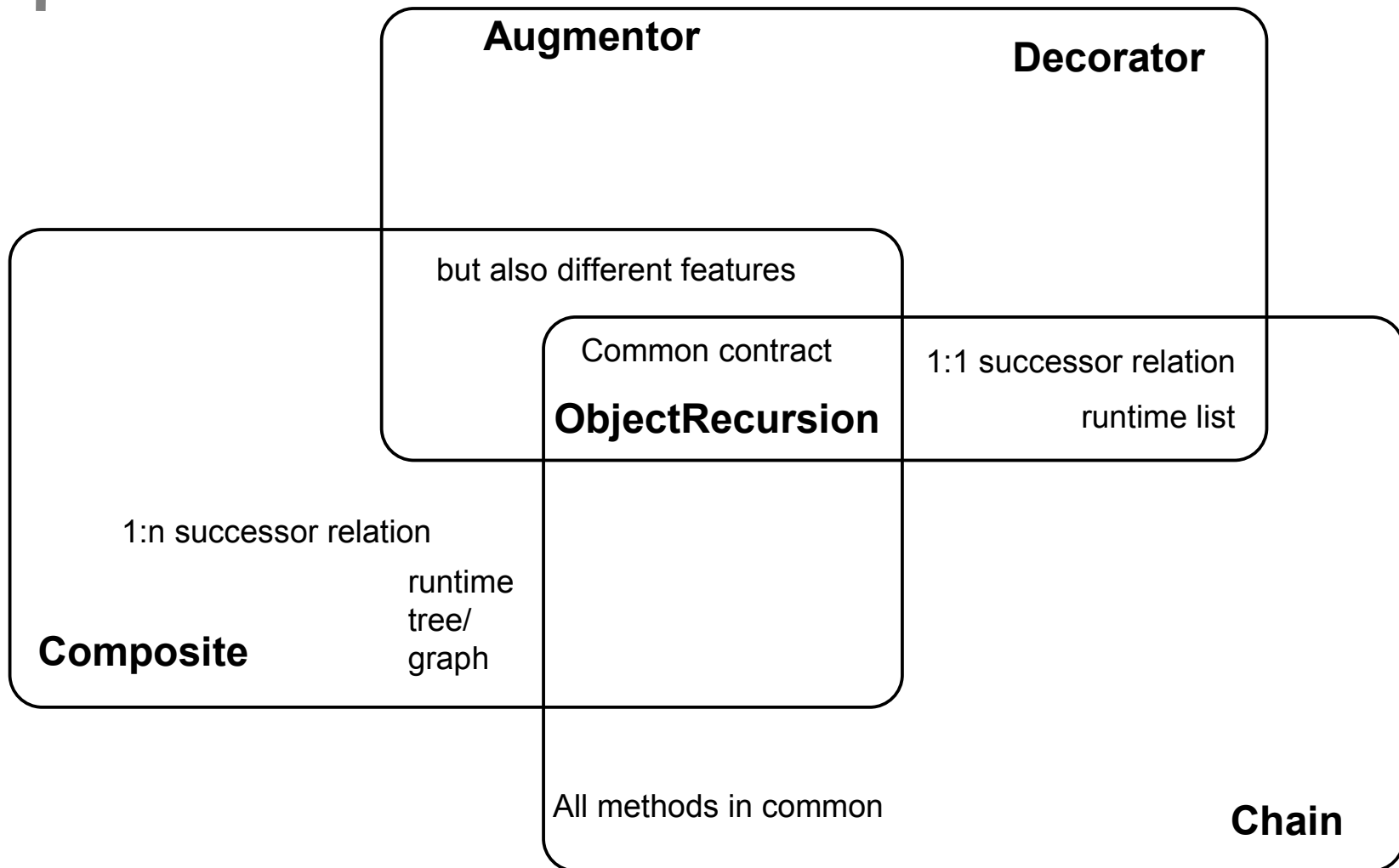
ChainOfResponsibility - Applications

36

- ▶ Realizes *Dynamic Call*:
 - If the receiver of a message is neither known at compile-time nor at allocation time (polymorphism), but only at runtime (i.e., depends on the current net of objects)
 - 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

37





3.2. Flat Extensibility

3.2.1 Proxy

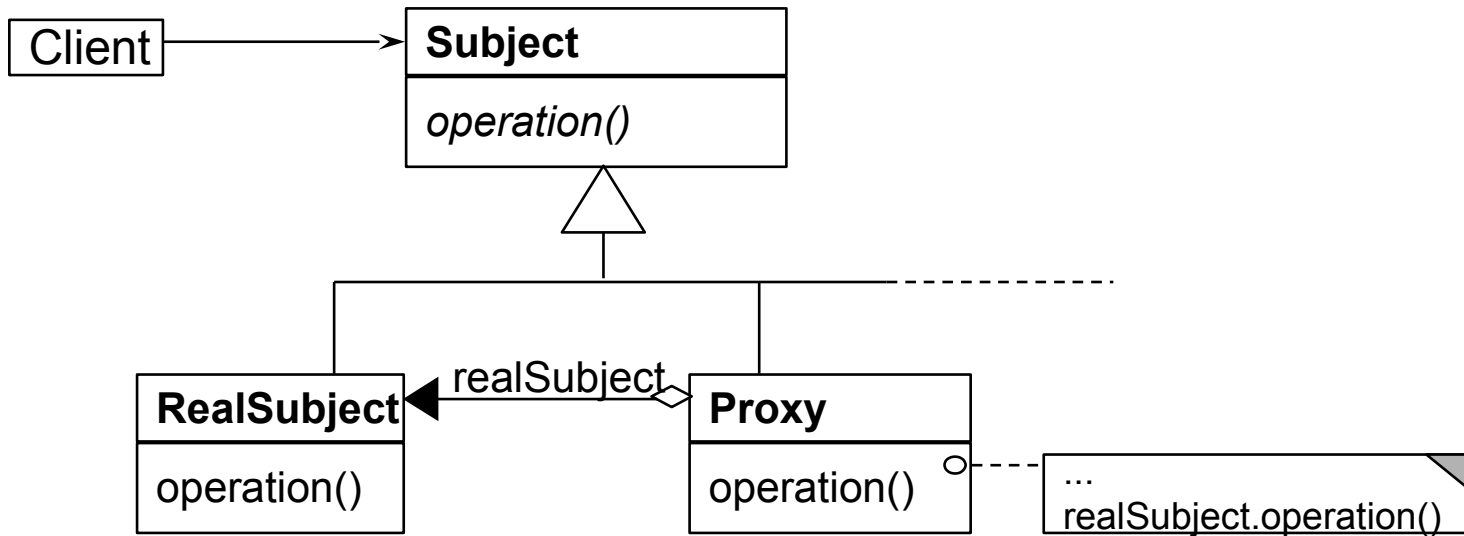
38



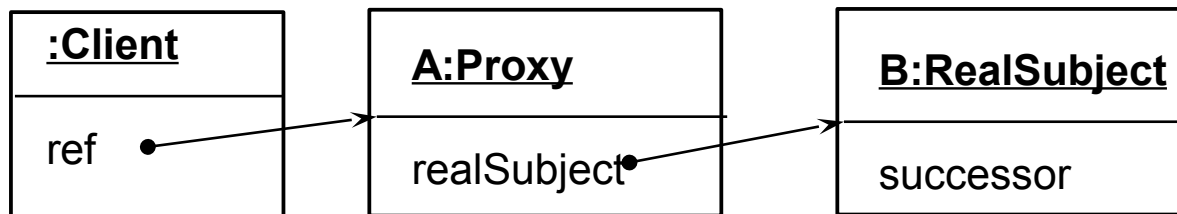
Proxy

39

- ▶ Hide the access to a real subject by a representative



Object Structure:



Proxy

40

- ▶ The proxy object is a *representative* of an object
 - The Proxy is similar to Decorator, but it is not derived from ObjectReursion
 - It extends **flat**: 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

41

- ▶ *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*: representative of a remote object
- ▶ *Caching proxy*: caches values which had been loaded from the subject
 - Remote
 - Loading lazy on demand
- ▶ *Protection proxy*
 - Firewall

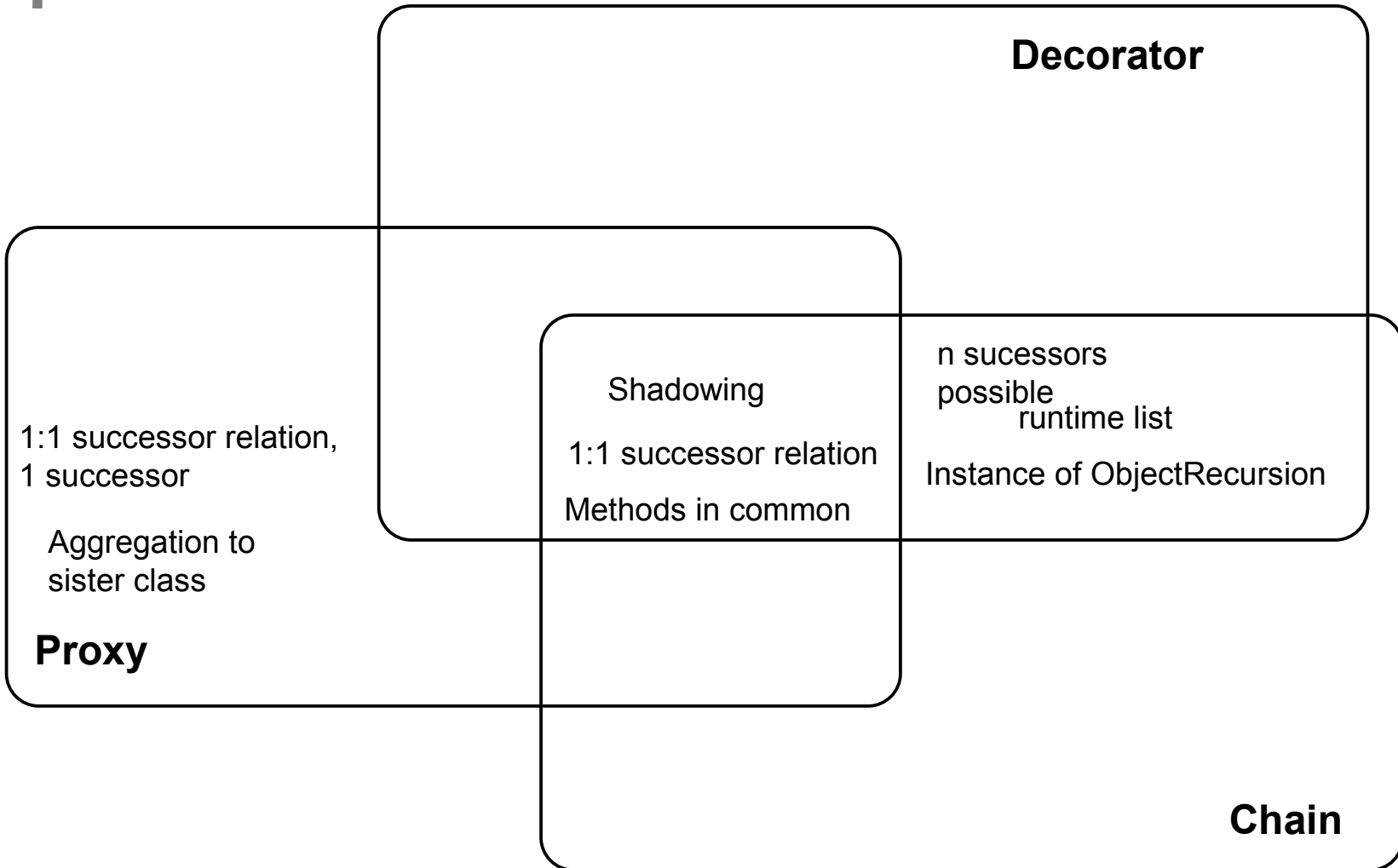
Proxy – Other Implementations

42

- ▶ Overloading of “->” access operation
 - C++, Ada and other languages allow for overloading access
 - Then, a proxy can intervene, but is invisible
- ▶ Overloading access can be 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

43





3.2.2 Star-Bridge (*-Bridge)

44

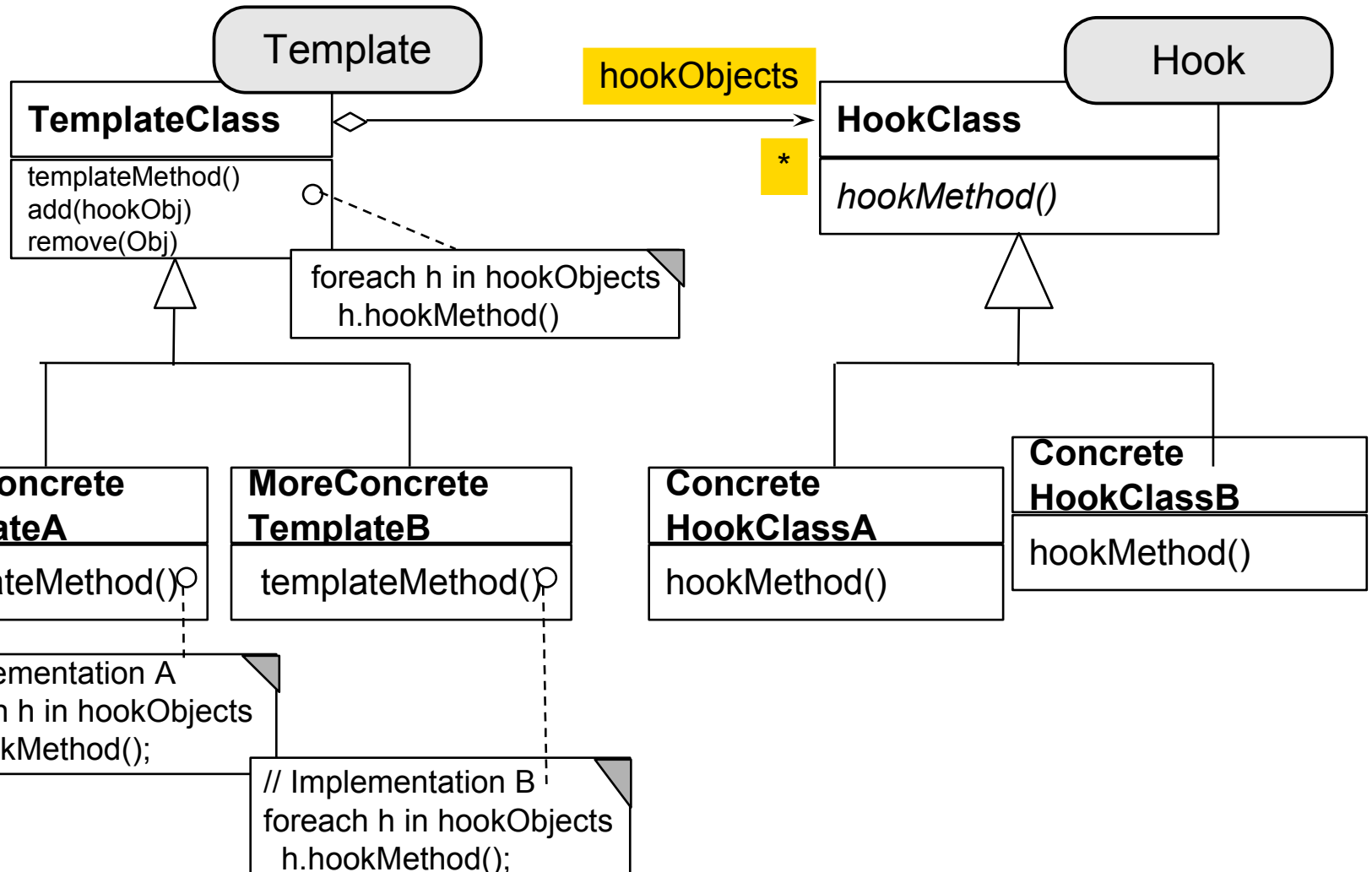


Extensibility Pattern

*DimensionalClassHierarchies (*Bridge)

45

- ▶ A bridge with a collection





3.2.3 Observer (Event Bridge)

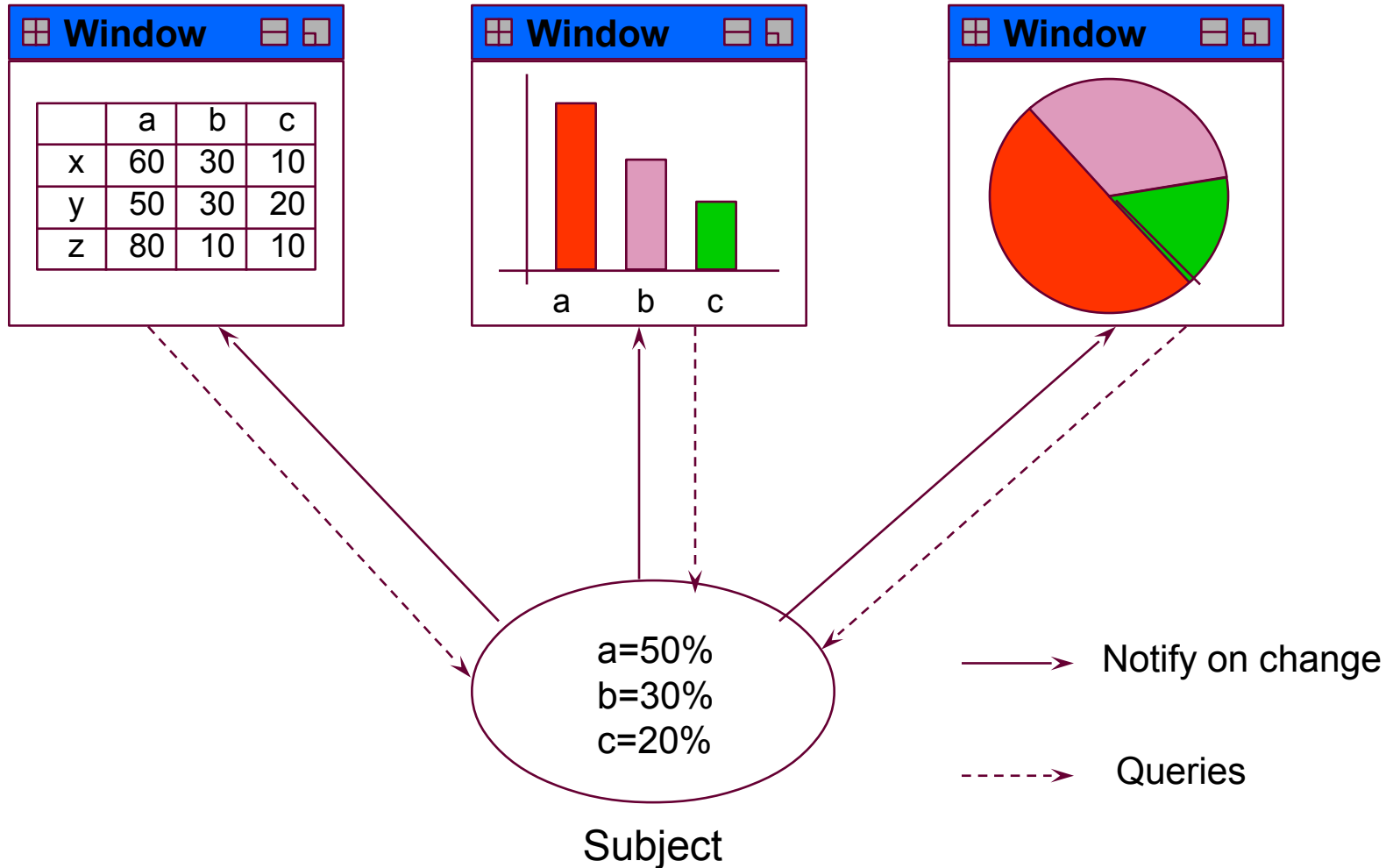
46



Observer (Publisher/Subscriber, Event Bridge)

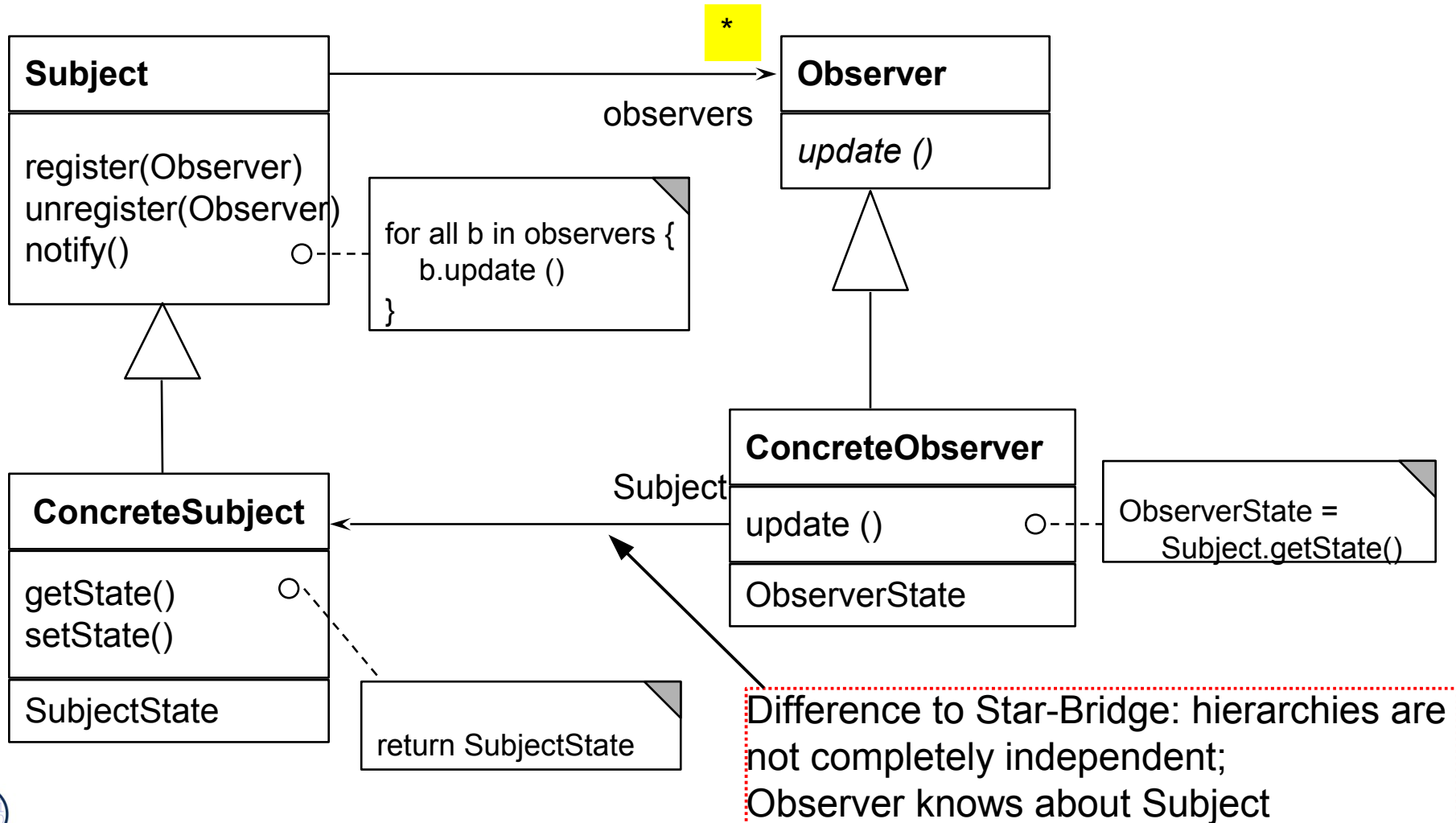
47

Observer



Structure Observer

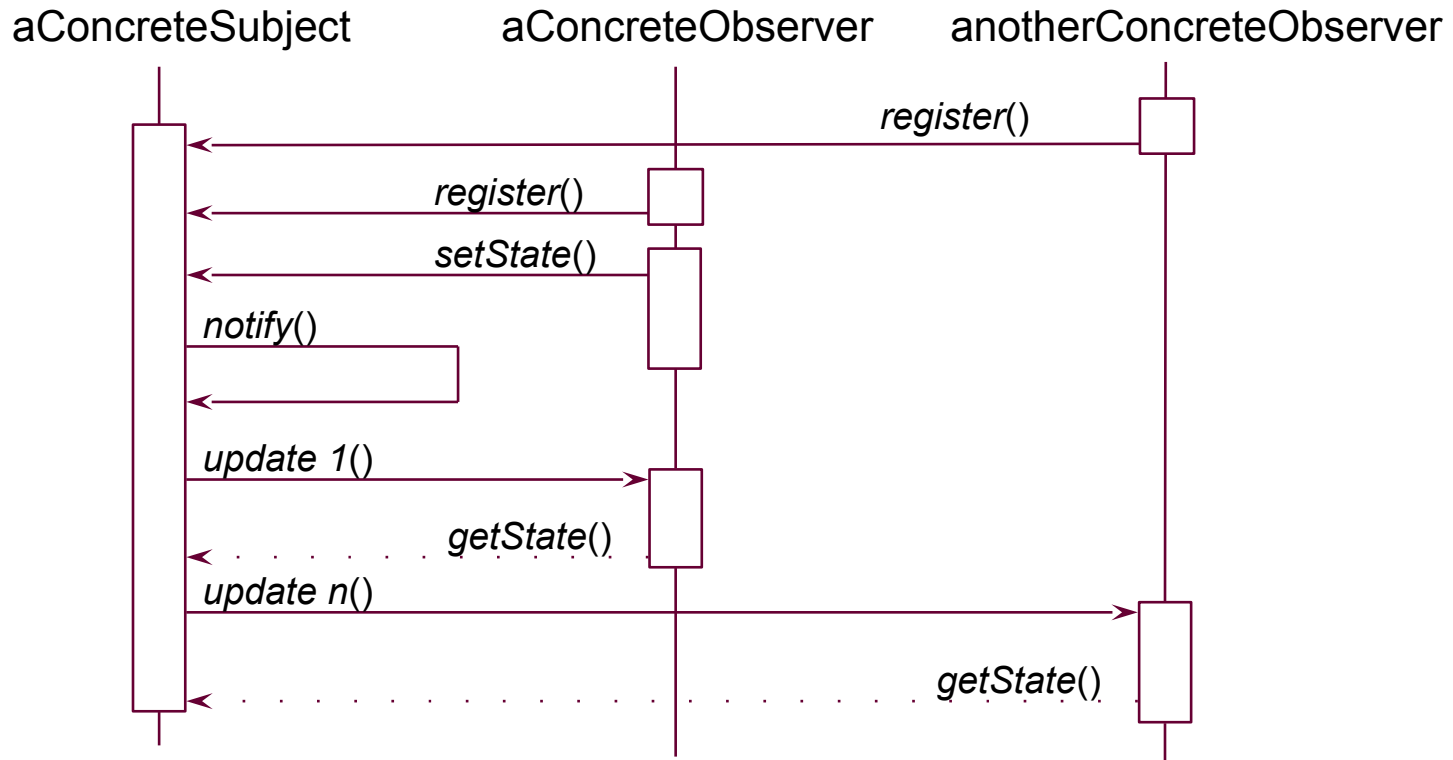
48 ▶ Extension of Star-Bridge



Sequence Diagram Observer

49

- ▶ 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 Variants

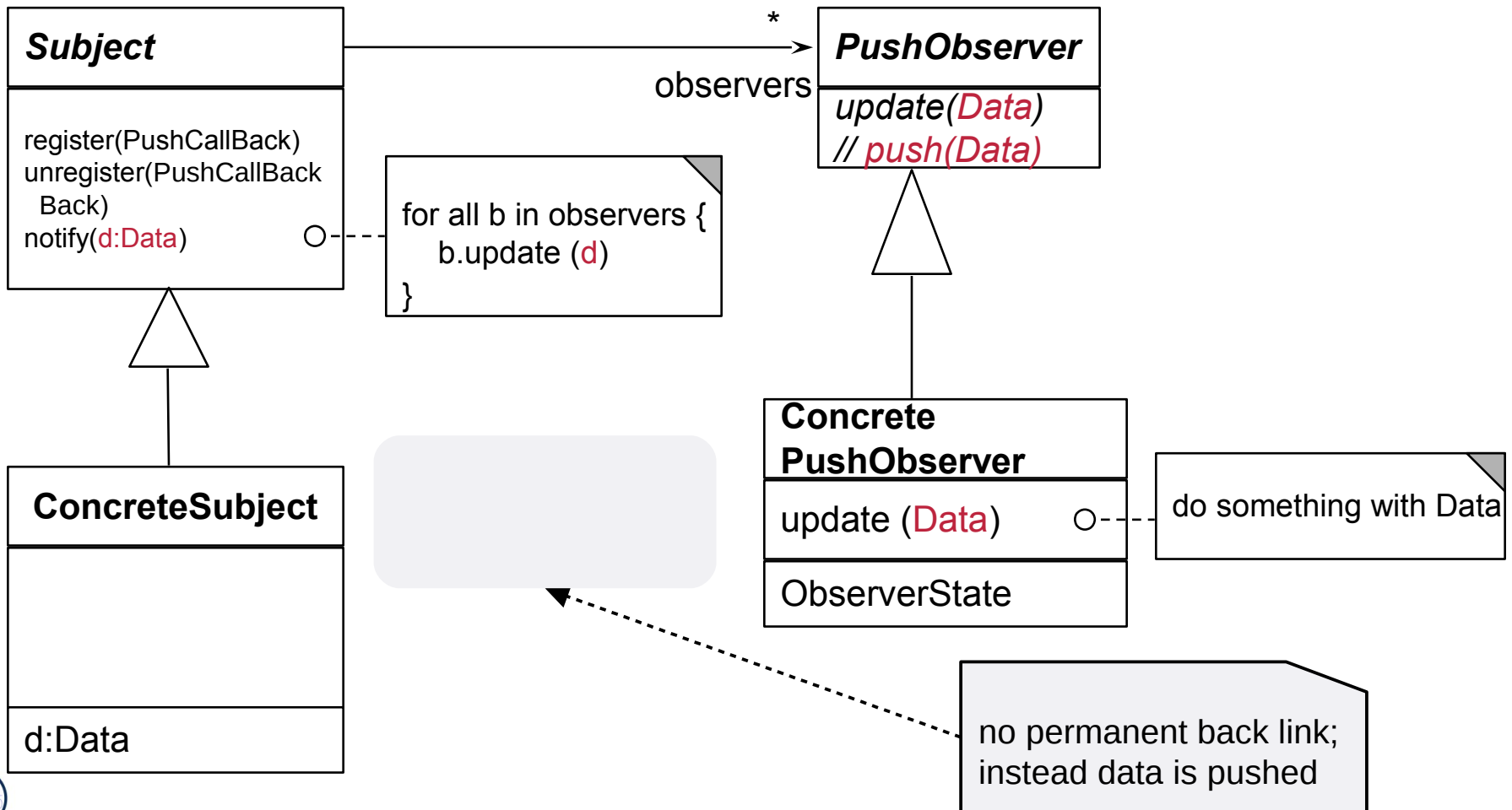
50

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

Structure Data-Pushing-Observer

51

- ▶ Subject pushes data or itself with `update (Data)`
- ▶ Pushing resembles *Sink*, if data is pushed iteratively

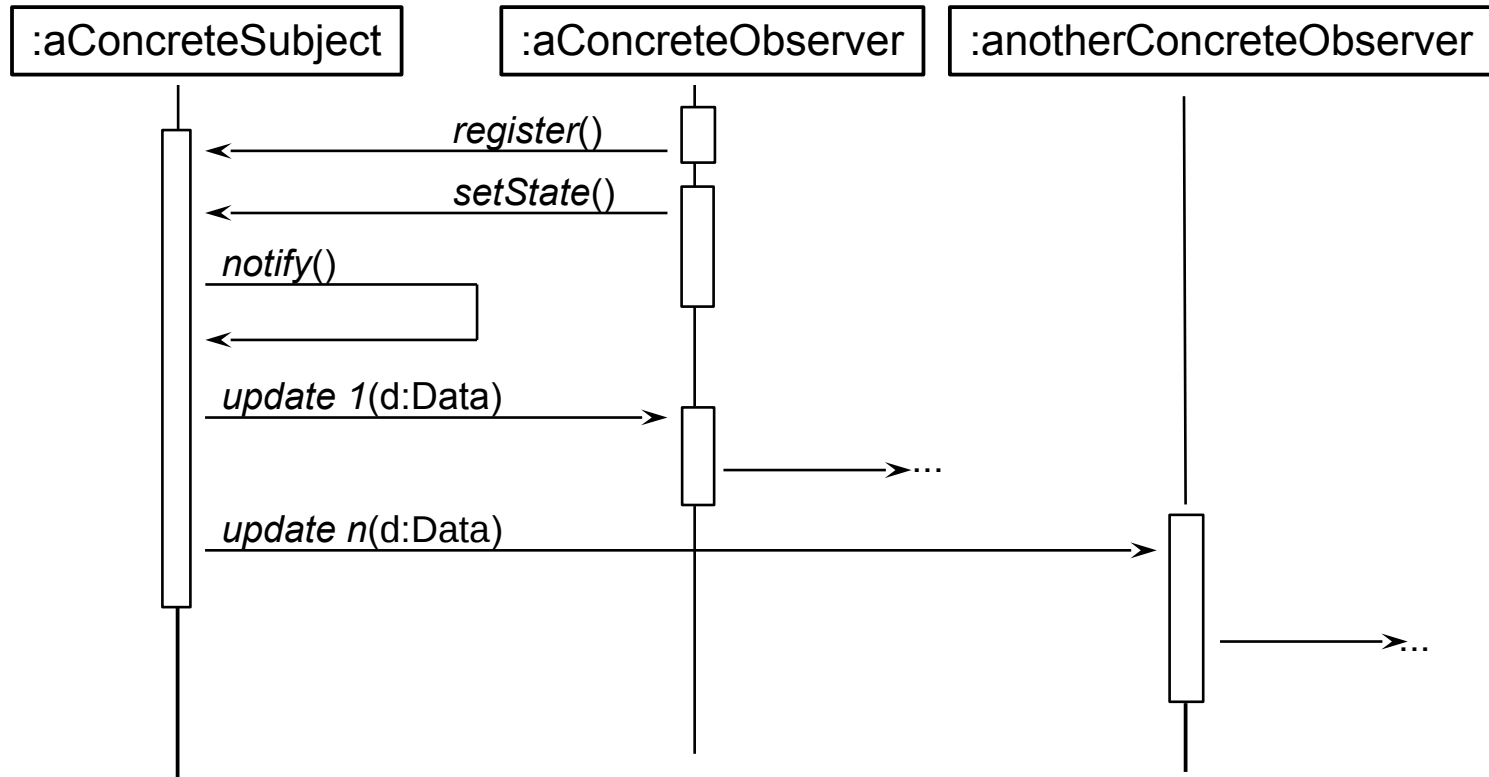


Sequence Diagram

Data-Push-Observer

52

- ▶ Update() transfers Data to Observer (push)



Observer - Applications

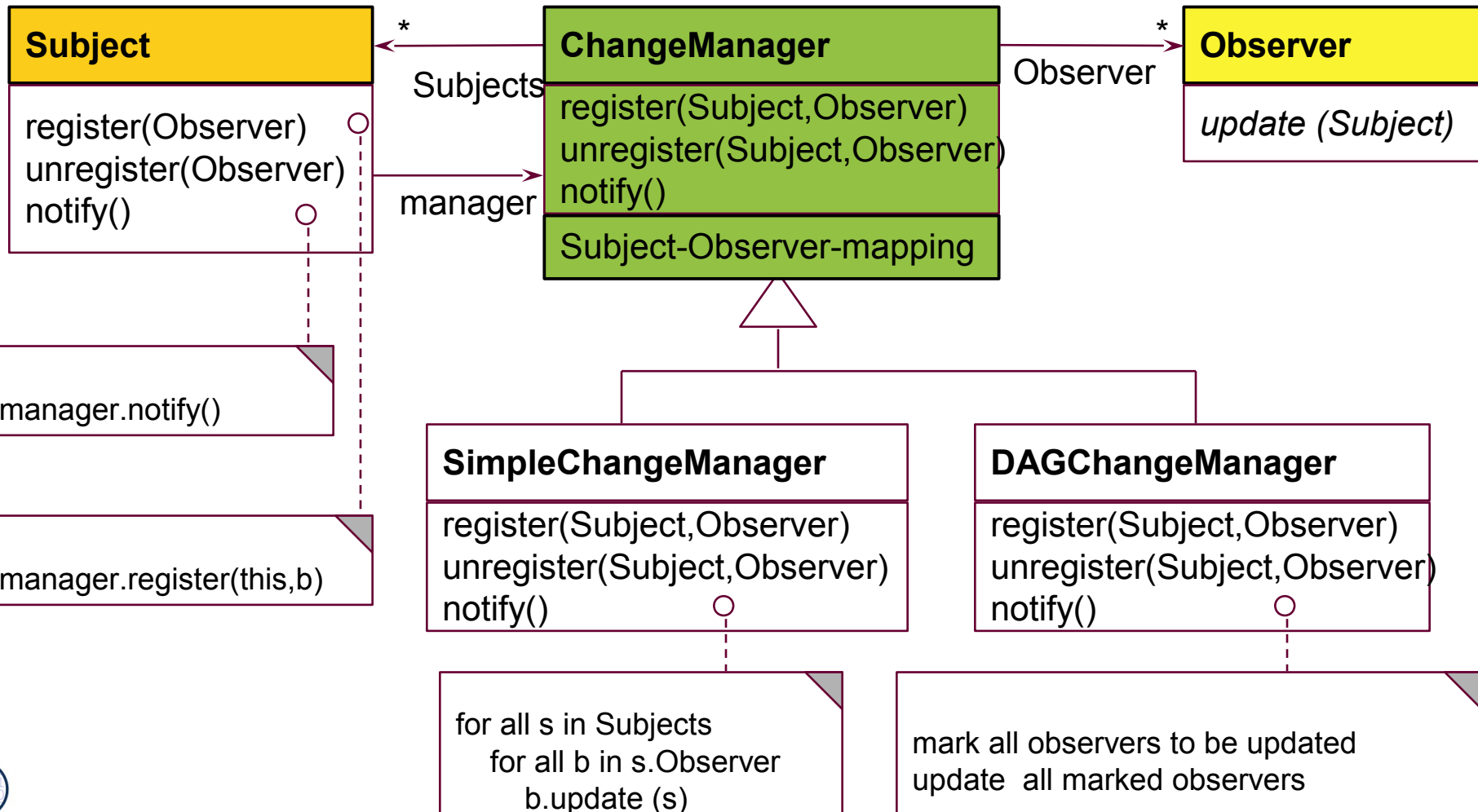
53

- ▶ 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 observing aspect
 - Observers are a simple way to implement aspect-orientation by hand
 - 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

Observer with ChangeManager (Mediator)

54

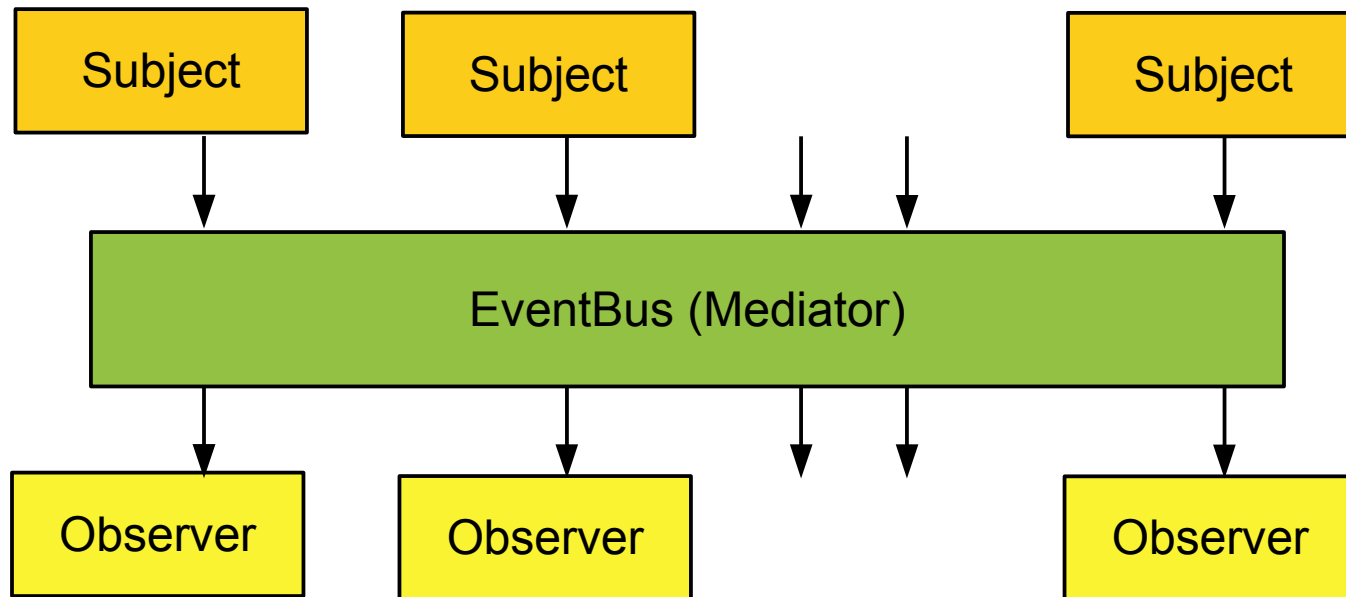
- ▶ Mediator between subjects and observer:
 - May filter events, stop cascaded propagations



ChangeManager is also Called Eventbus

55

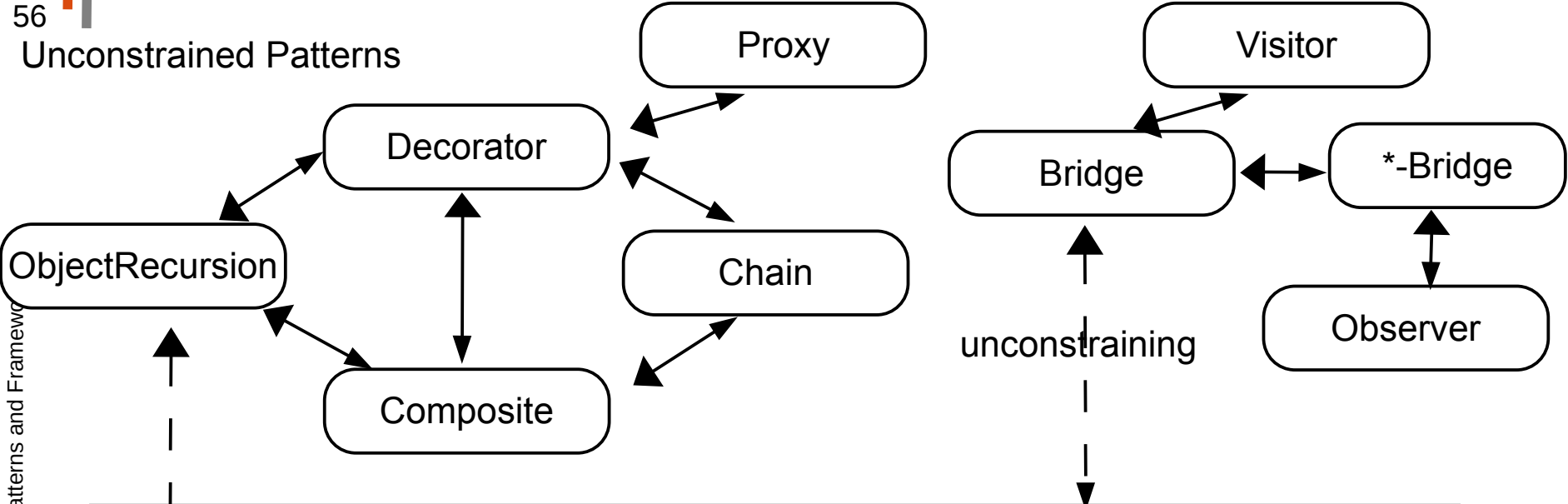
- ▶ Basis of many interactive application frameworks (Xwindows, Java AWT, Java InfoBus,)



Relations Extensibility Patterns

56

Unconstrained Patterns



unconstraining

unconstraining

Recursive T&H Pattern

Framework Patterns obeying T&H role model

Dimensional Class Hierarchies

Connection T&H Pattern



Summary

57

- ▶ 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