# Advertisement of PhD Positions in Twente

- 4 PhD positions in the Aselsan - University of Twente cooperation framework
  - The University of Twente (Enschede, the Netherlands) and Aselsan (Ankara,Turkey) are seeking enthusiastic and creative Ph.D. candidates of Turkish nationality, with an outstanding M.Sc. degree in Computer Science (or an equivalent qualification) and/or Electrical Engineering.
- Candidates should have thorough theoretical and practical background in software engineering methods, software architectures, programming languages and modeling techniques. Depending on the projects (see the list below) applied to by the candidate, knowledge in product line engineering, scheduling, event-driven and service-oriented architectures, formal modeling approaches and optimization techniques is favorable.
- See http://fmt.ewi.utwente.nl/projects/aselsan
- Please apply on or before 15 November 2013.

# PhD Projects

- **Productline for Optimal Schedulers (PLOS):** The project PLOS proposes a productline architecture for designing optimal schedulers for the digital receivers that takes care of application semantics in scheduling, can cope  with dynamically changing context, can deal with variations in scheduling objectives, optimizes the scheduling criteria and causes an acceptable overhead. The productline approach enables to effectively reuse the basic building elements of the scheduler asset base in different application settings.
- **Reuse of event-driven service-oriented architectures (RESA):** The project RESA aims at defining methods and techniques for enhancing reuse of event-driven service-oriented signal processing systems. To this aim, the project considers reuse with respect to new software adaptation and evolution requirements together with time performance requirements, since these two quality factors generally conflict with each other. Also,    optimization techniques will be provided for the trade-off between these quality factors. Experiments will be carried out using industrial examples.

# PhD Projects

- **Communication and verification of architecture design and its rationale (CVAR):** The project CVAR aims to define methods, techniques and tools for specifying, communicating and verifying software systems through the use of graphical notations. These notations have well-defined semantics and can be analysed through simulating the dynamics of the software models so that the software systems can be communicated easily and the possible errors can be detected conveniently before extensive programming effort is carried out. This project adopts design rationale analysis and model checking techniques.
- **Runtime verification of protocols (RTVPRO):** The RTVPRO projects develops method, techniques and tools for the specification and verification of dynamically configurable software systems (such as systems with dynamically configured protocols) through the combined use of runtime verification, runtime model-driven engineering, and model checking techniques. In addition, this project develops techniques to check the conformance of architecture models with respect to the actual execution of software that it represents.

# Software Technology Group at the University of Twente

- Prof. Dr. Mehmet Akşit, Chair Software Engineering, m.aksit@utwente.nl
  - Mrs. Jeanette Rebel-de Boer, j.a.deboer@utwente.nl
  - Özgü Özköse Erdoğan, ASELSAN, REHİS Mission Software Manager, ozkose@aselsan.com.tr.
- Excellent research environment
- Excellent carrier opportunity at Aselsan.
  - The candidates will be employed by Aselsan and will be assigned to carry on the Ph.D. program at University of Twente.
  - After succesfully completing the Ph.D. degree, they will continue with working at Aselsan.
- Team work of Research & Industry. The faculty members, Aselsan and Ph.D. candidates will cooperate to address complex industrial problems. Projects will be carried out with Aselsan located in Ankara, Turkey. Frequent visits will be made to the company to identify the relevant industrial issues and to validate the applicability of the proposed solutions.
- Turkish citizenship required
- Applicants should mail an application letter indicating the project they are applying for (see list above) with a clear motivation, a CV with a list of courses taken and projects carried out previously, an electronic copy of the MSc thesis and of any publications, and two references, to the above address.

# Chapter 4 Simple Patterns for Extensibility

Prof. Dr. U. Aßmann

Chair for Software Engineering

Fakultät Informatik

Technische Universität Dresden

Version 13-1.2, 11/16/13

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)

- 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

# Optional Literature

- 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://wwwiti.cs.uni-magdeburg.de/~rosenmue/publications/DissRosenmueller.pdf
- 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://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/ps/auto/RSAS11.pdf

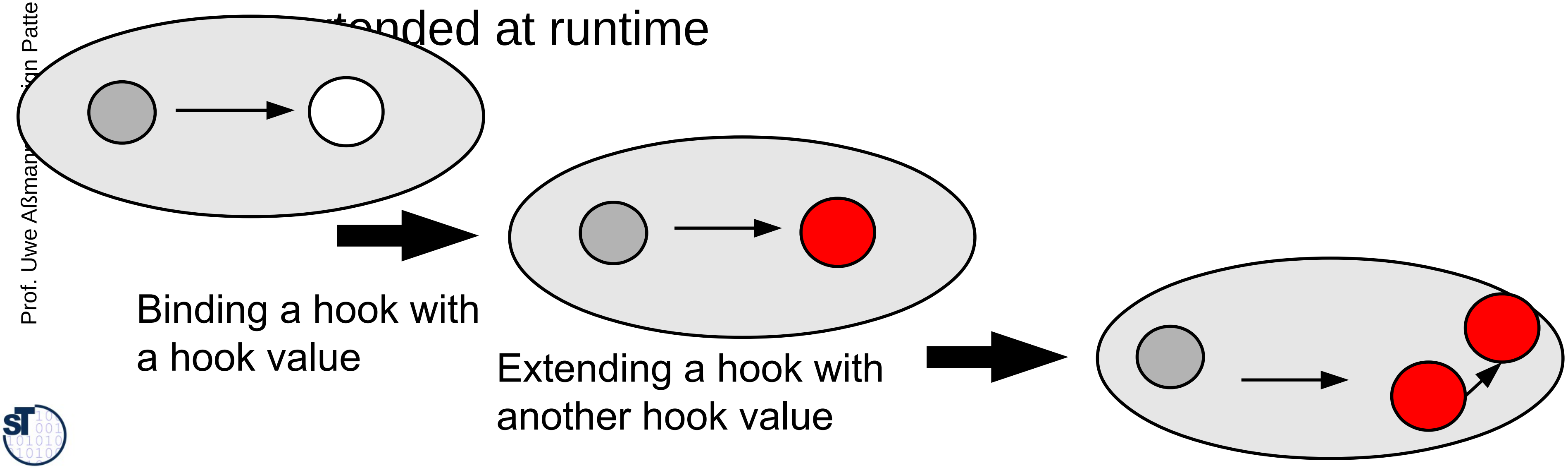
# Goal

- 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

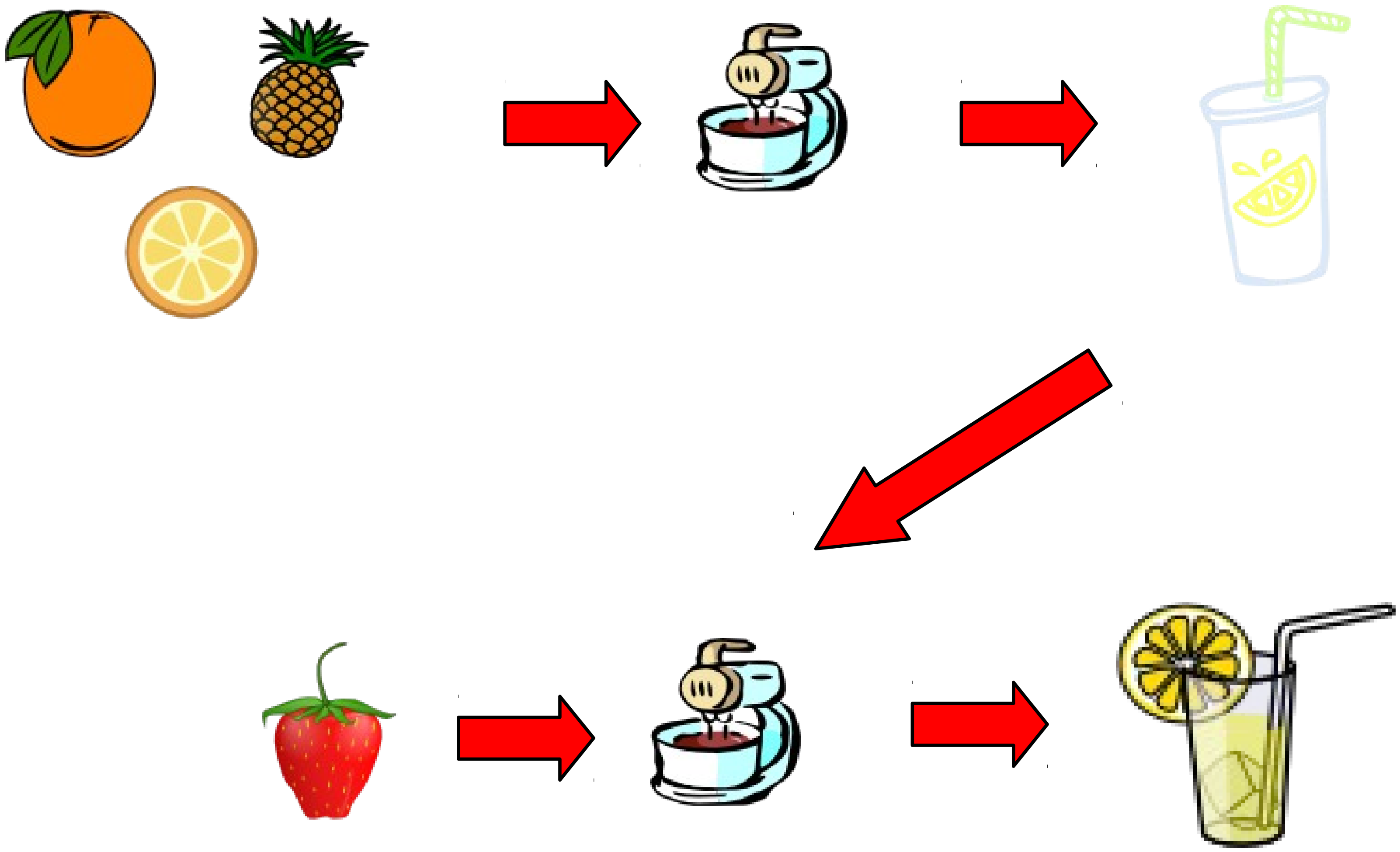
- Variability so far meant
  - Static extensibility, e.g., new subclasses
  - Often, dynamic *exchangability* (polymorphism)
  - But not dynamic extensibility
- Now, we will turn to patterns that allow for dynamic extensibility
  - Most of these patterns contain a 1:n-aggregation that is extended at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



Binding a hook with a hook value

Extending a hook with another hook value

# Software Cocktail Mixers

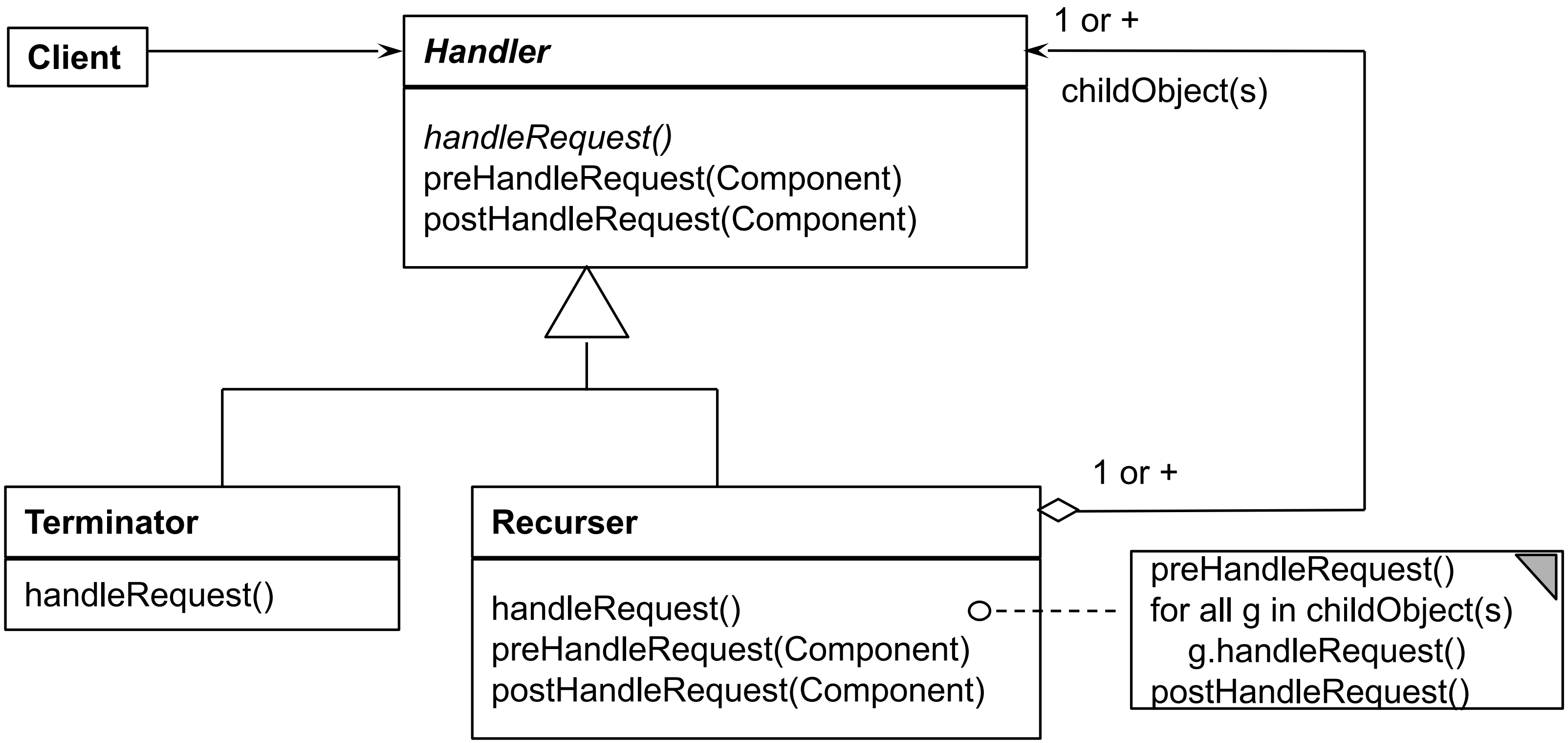


# 3.1 Recursive Extension

# 3.1.1 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 (lists, 1-Recursion) or 1:+ (trees, n-recursion), +:+ (dags or graphs, n-recursion)

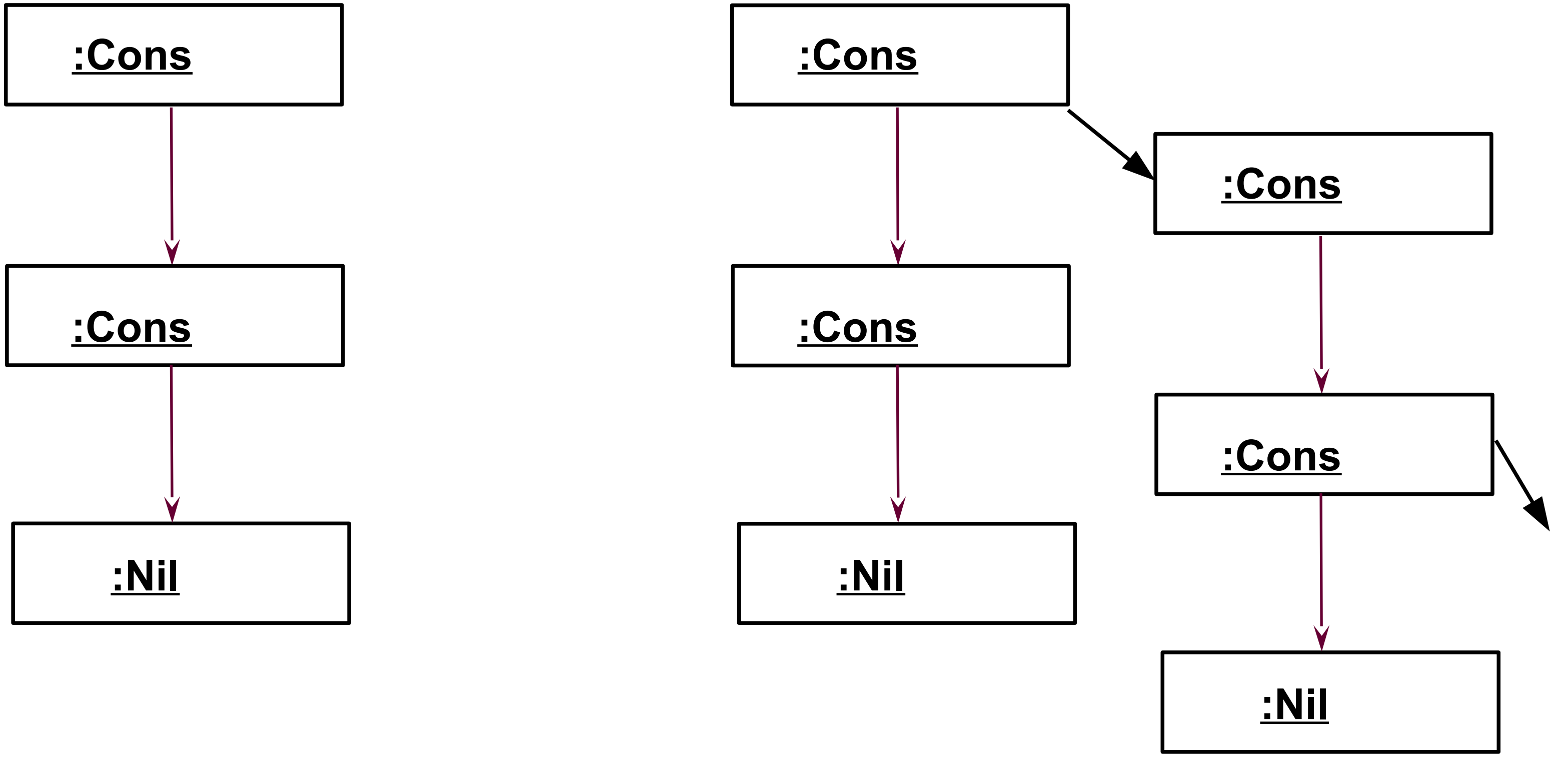


# Incentive

- 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, dags, and graphs

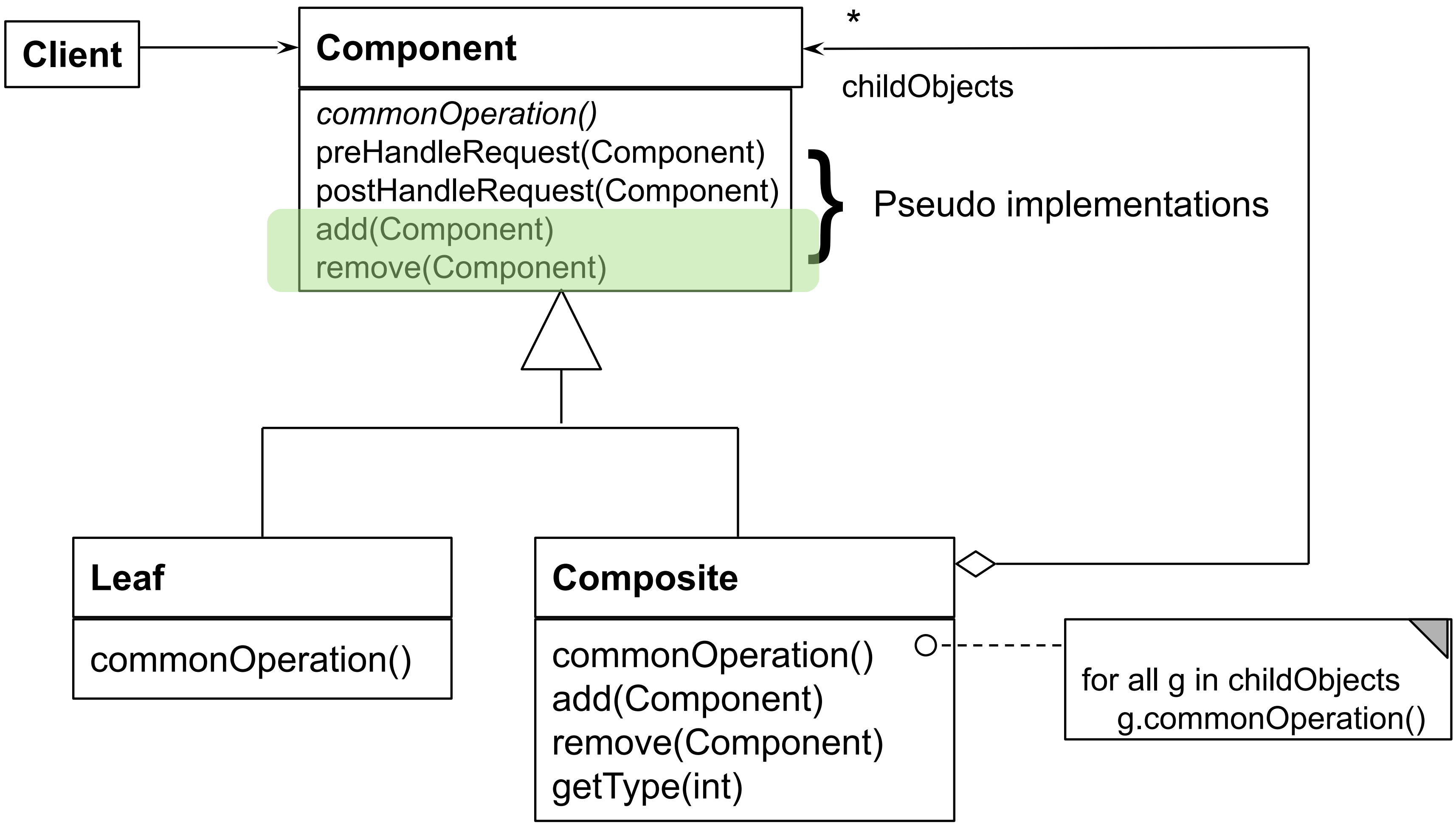


The recursion allows for building up runtime nets

# 3.1.2 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;
```

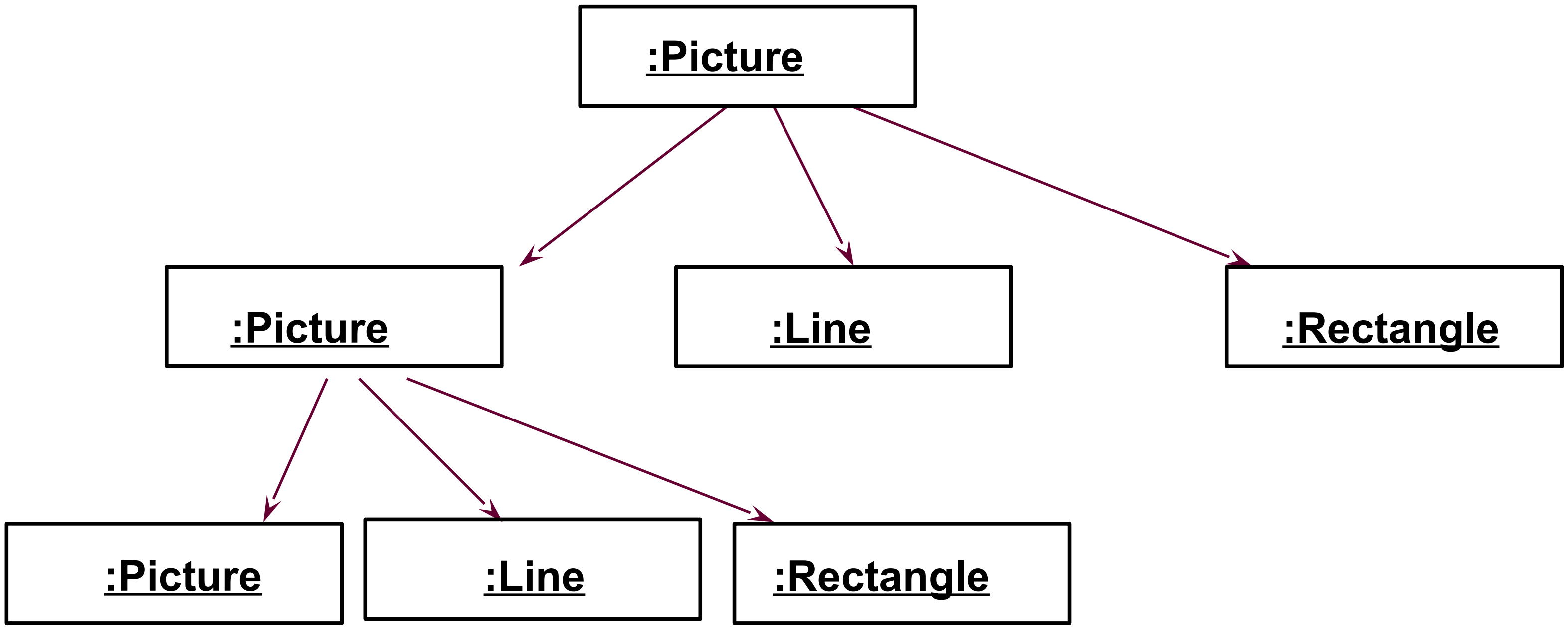
```
class Screw extends CarPart {
   int myCost = 10;
   int calculateCost() {
     return  myCost;
   }
  void addPart(CarPart c) {
      /// impossible, dont do anything
  }
}

// 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

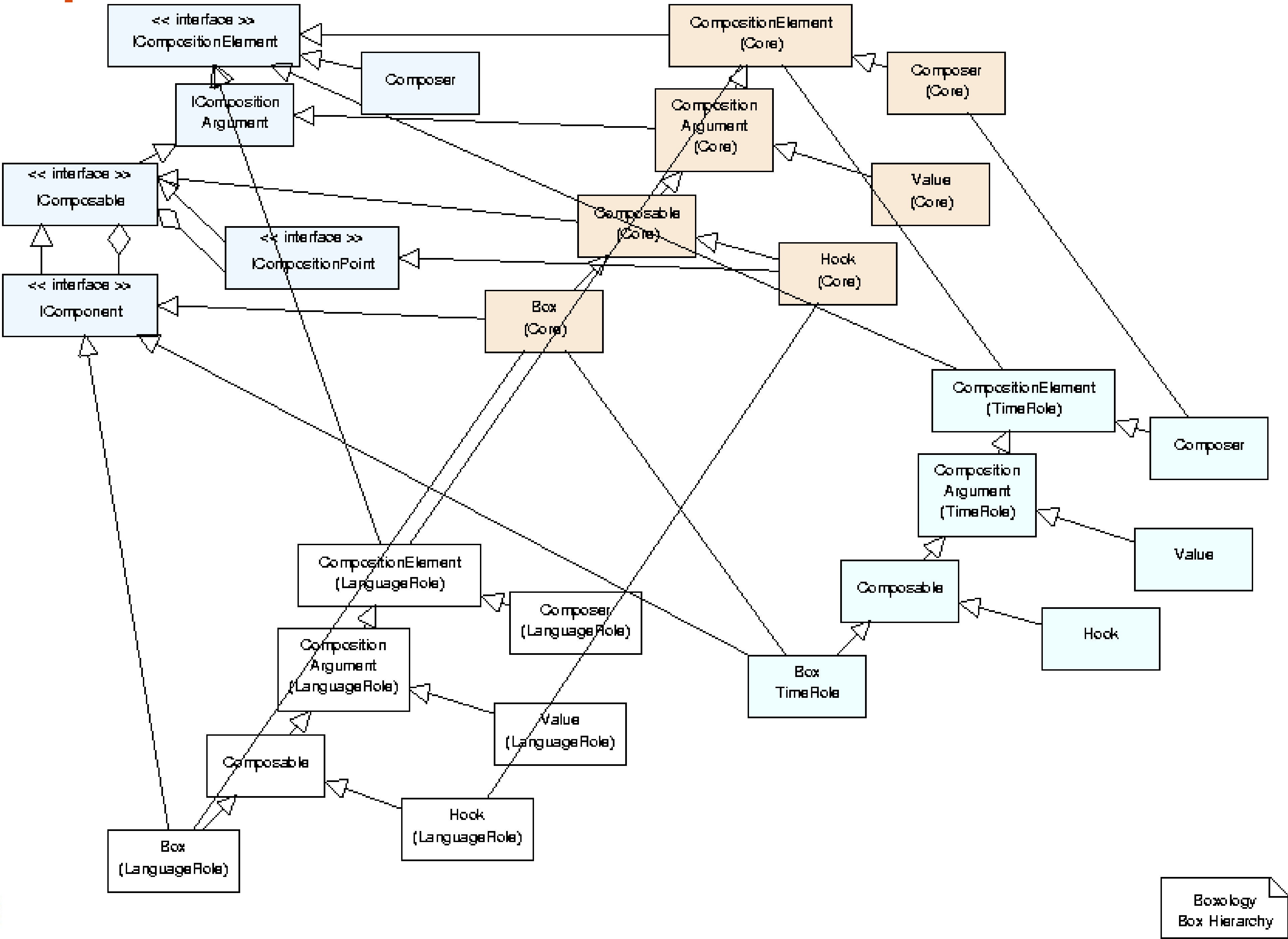
- Part/Whole hierarchies, e.g., nested graphic objects



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

# Dynamic, Recursive 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)


<< interface >>
ICompositionElement
CompositionElement
(Core)
Composer
ICompositionArgument
Composition
Argument
(Core)
Composer
(Core)
<< interface >>
IComposable
Value
(Core)
Composable
(Core)
<< interface >>
ICompositionPoint
Hook
(Core)
<< interface >>
IComponent
Box
(Core)
CompositionElement
(TimeRole)
Composer
Composition
Argument
(TimeRole)
Value
CompositionElement
(LanguageRole)
Composable
Composer
(LanguageRole)
Hook
Composition
Argument
(LanguageRole)
Box
TimeRole
Value
(LanguageRole)
Composable
Hook
(LanguageRole)
Box
(LanguageRole)
Boxology
Box Hierarchy

# Relations of Composite to Other Programming Domains

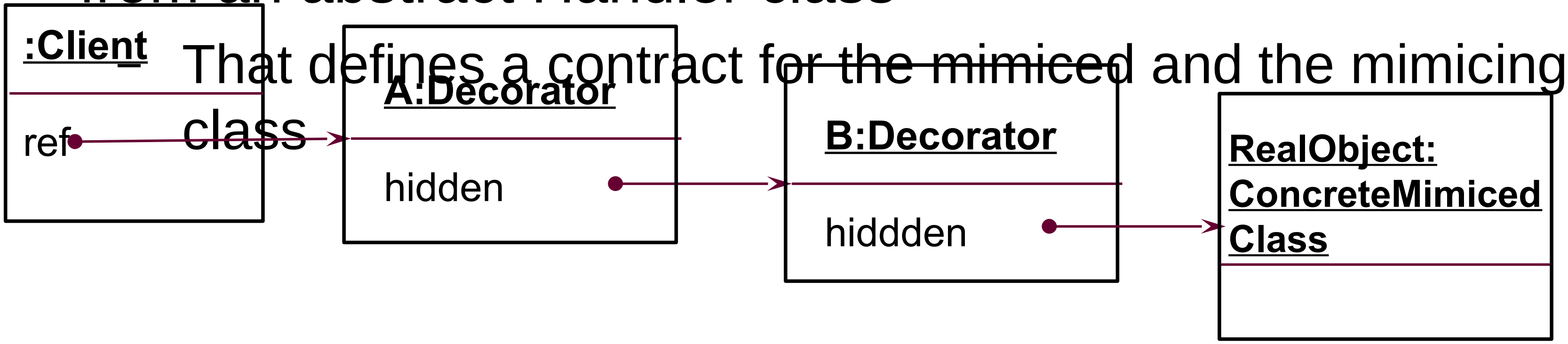
- Composite pattern is the heart of functional programming
  - Because recursion is the heart of functional programming
  - It has discovered many interesting algorithmic schemes for the Composite:
    - Functional skeletons (map, fold, partition, d&c, zip...)
    - Barbed wire (homo- and other morphisms)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

# 3.1.3 Decorator

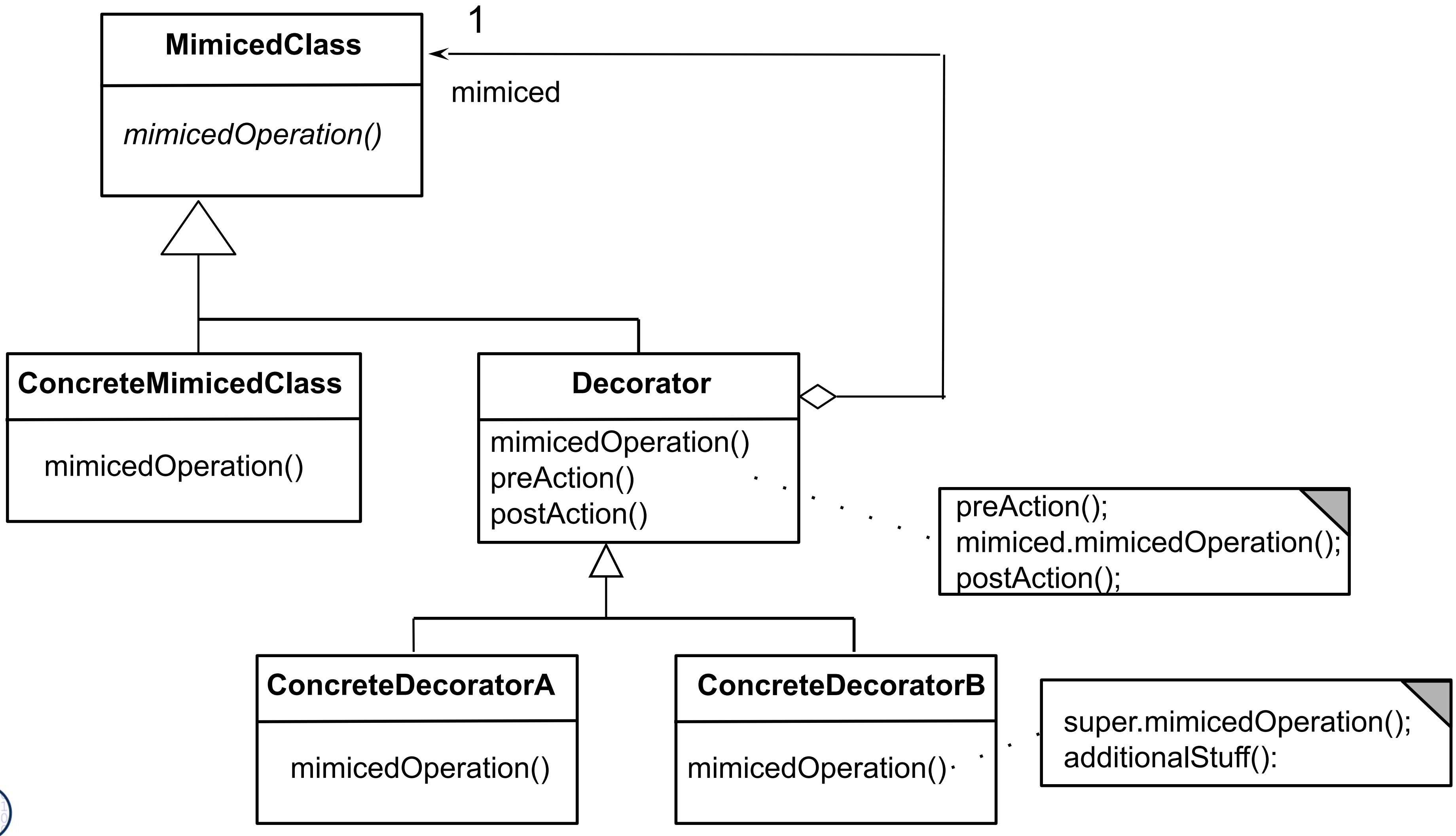
...as a Variant of ObjectRecursion and Composite

# Decorator Pattern

- A Decorator is a *skin (wrappers)* of another object
  - Core objects are in 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 mimiced and the mimicing class



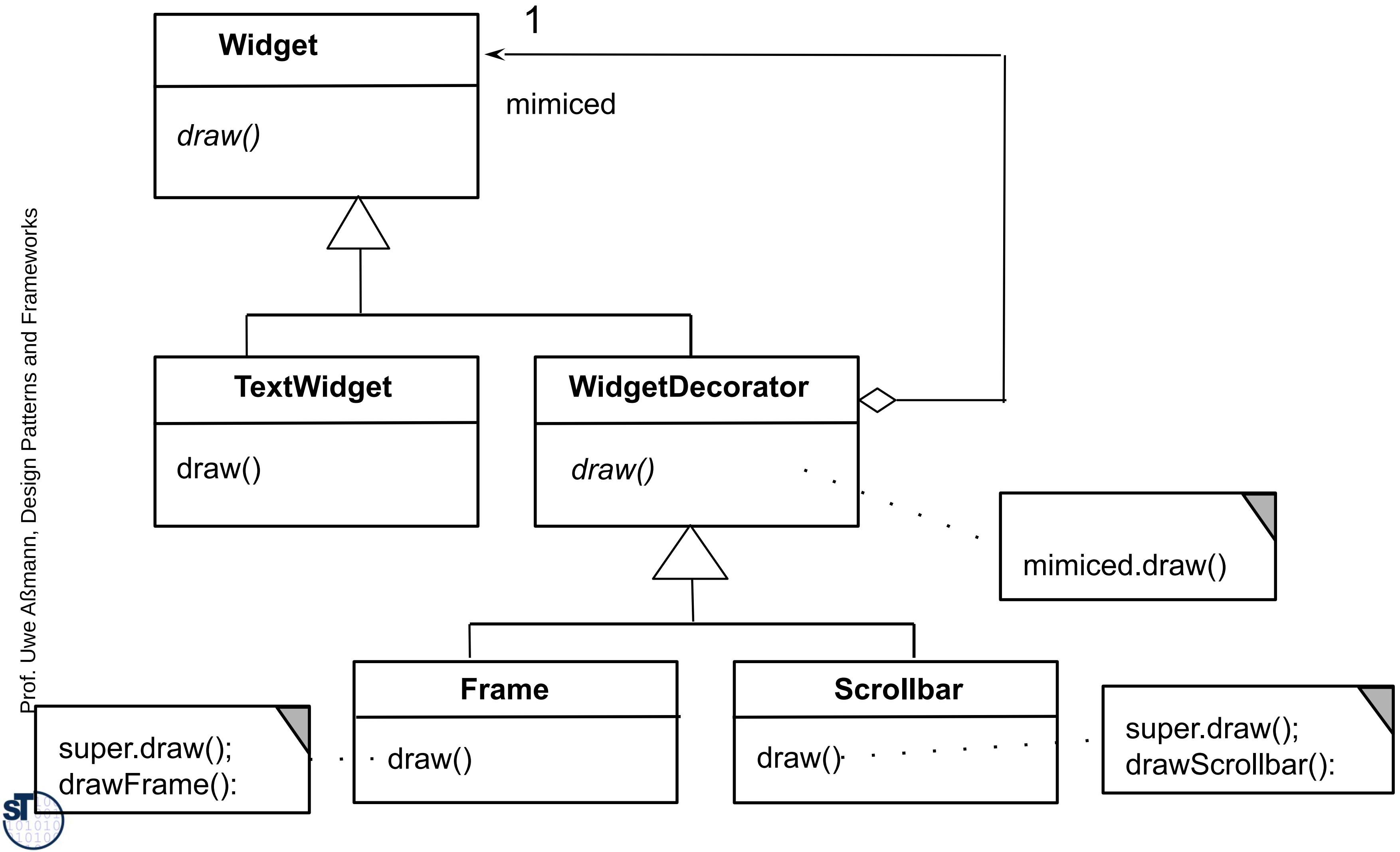
# Decorator – Structure Diagram



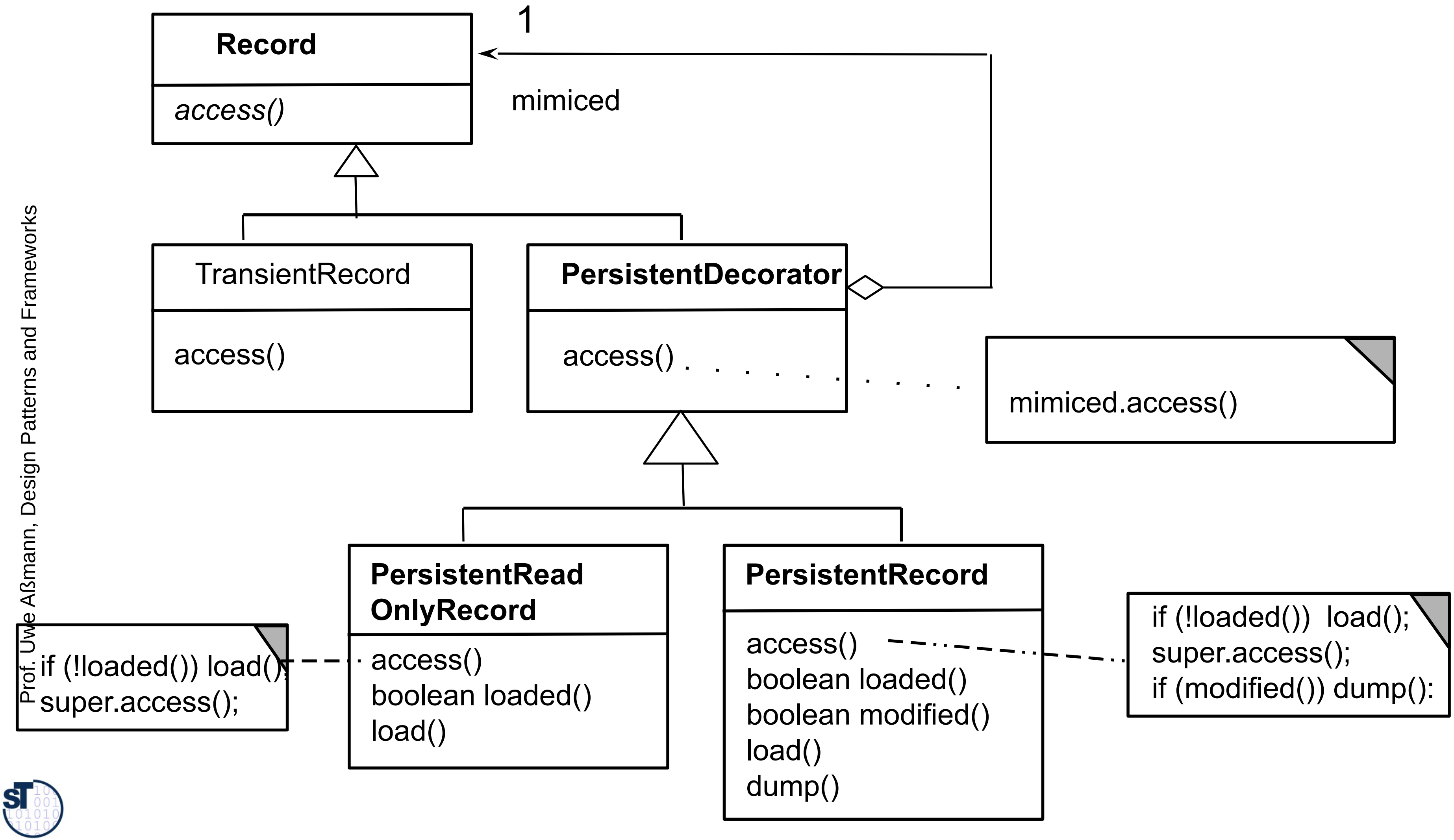
preAction();
mimiced.mimicedOperation();
postAction();

super.mimicedOperation();
additionalStuff():

# Decorator for Widgets

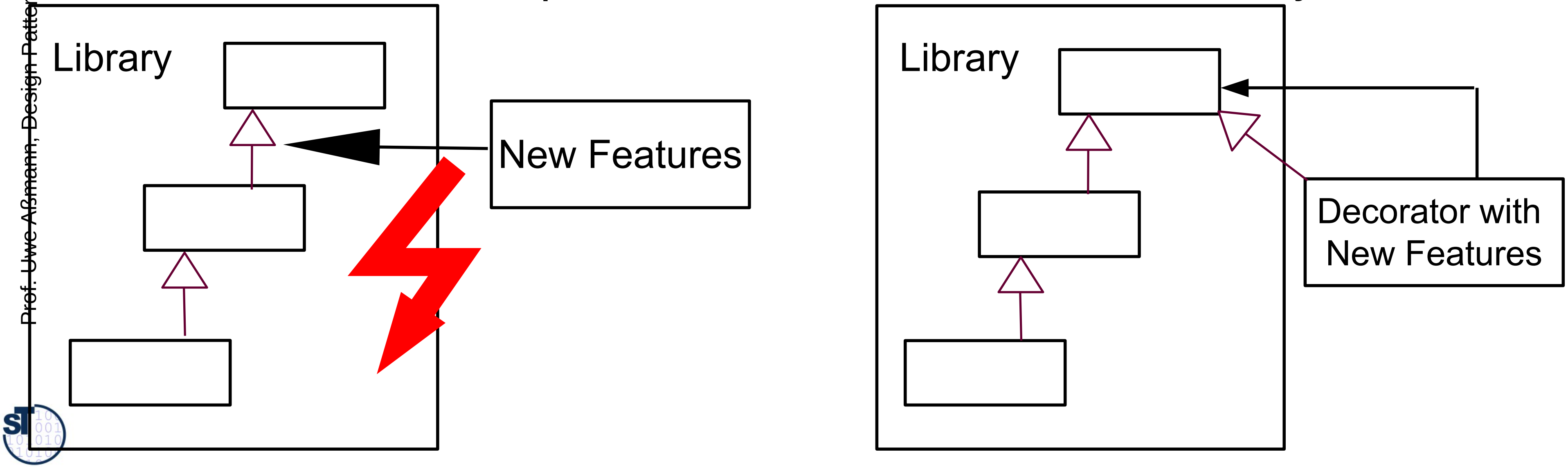


# Decorator for Persistent Objects



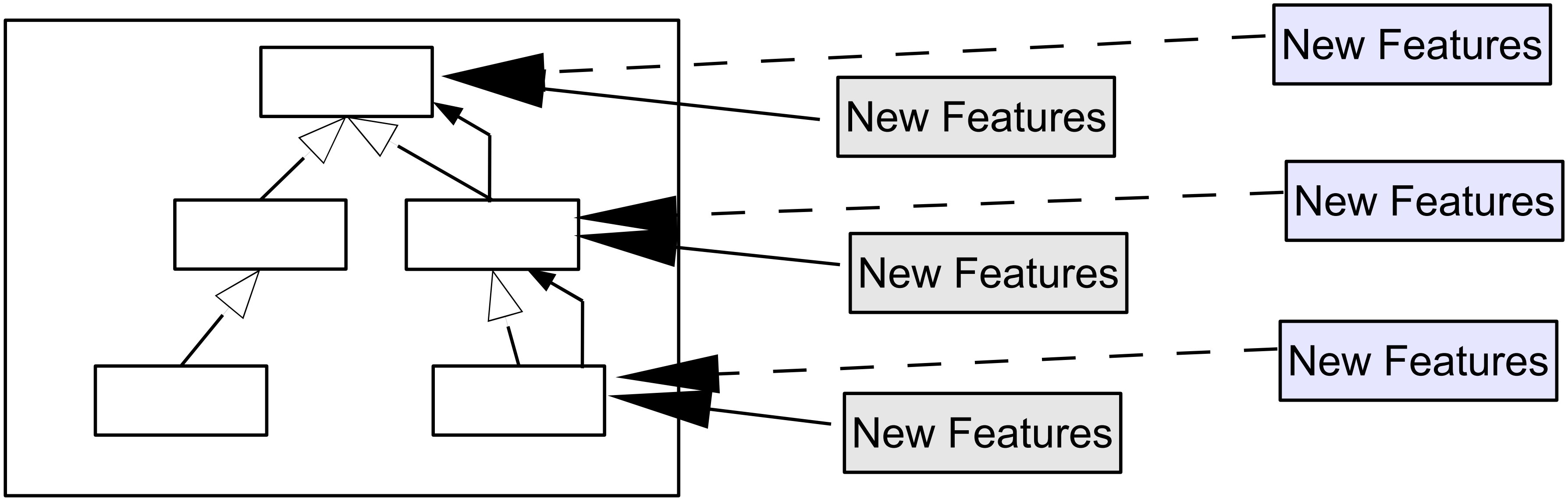
# Purpose Decorator

- 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

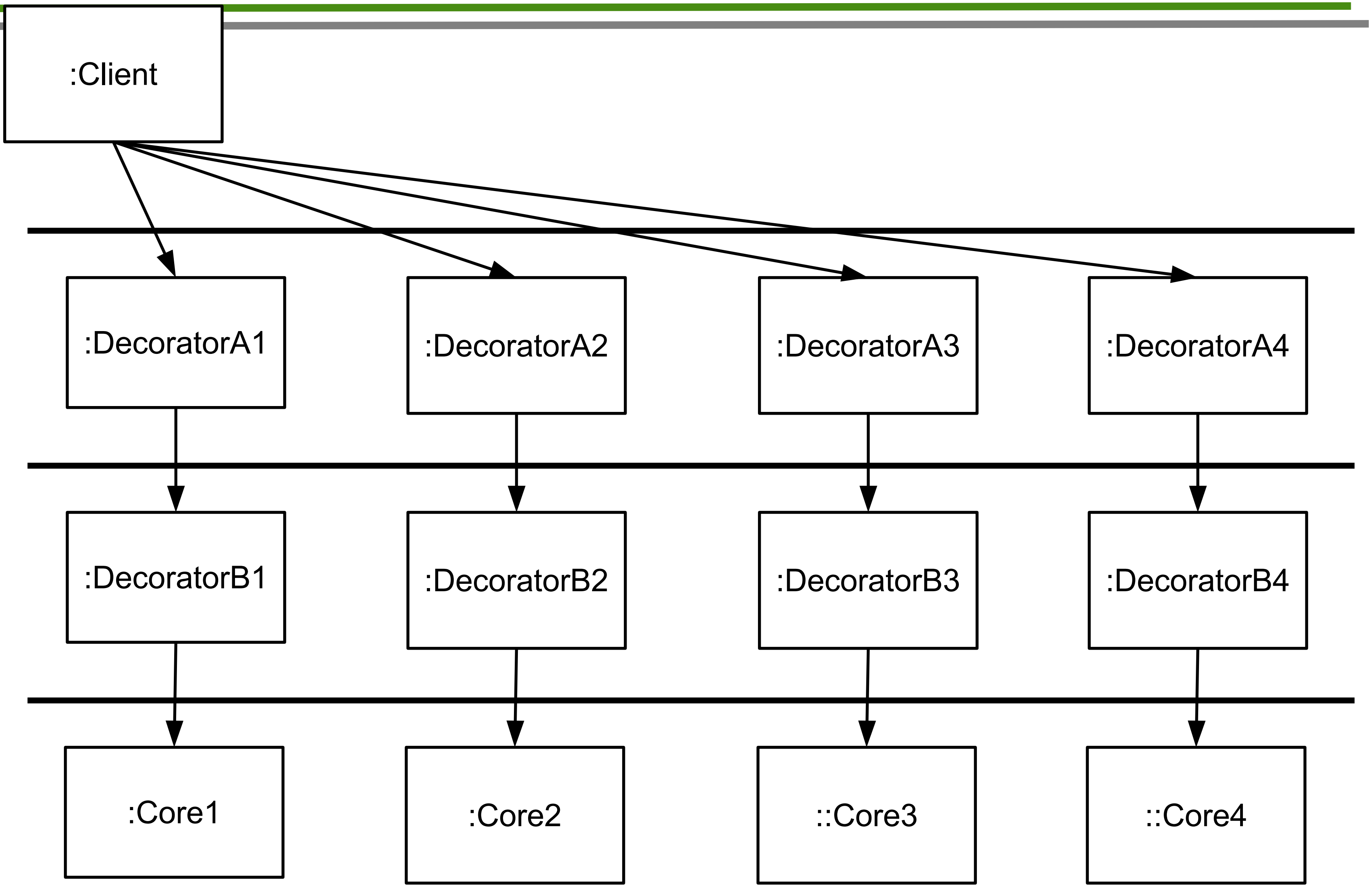


# Variants of Decorators

- 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

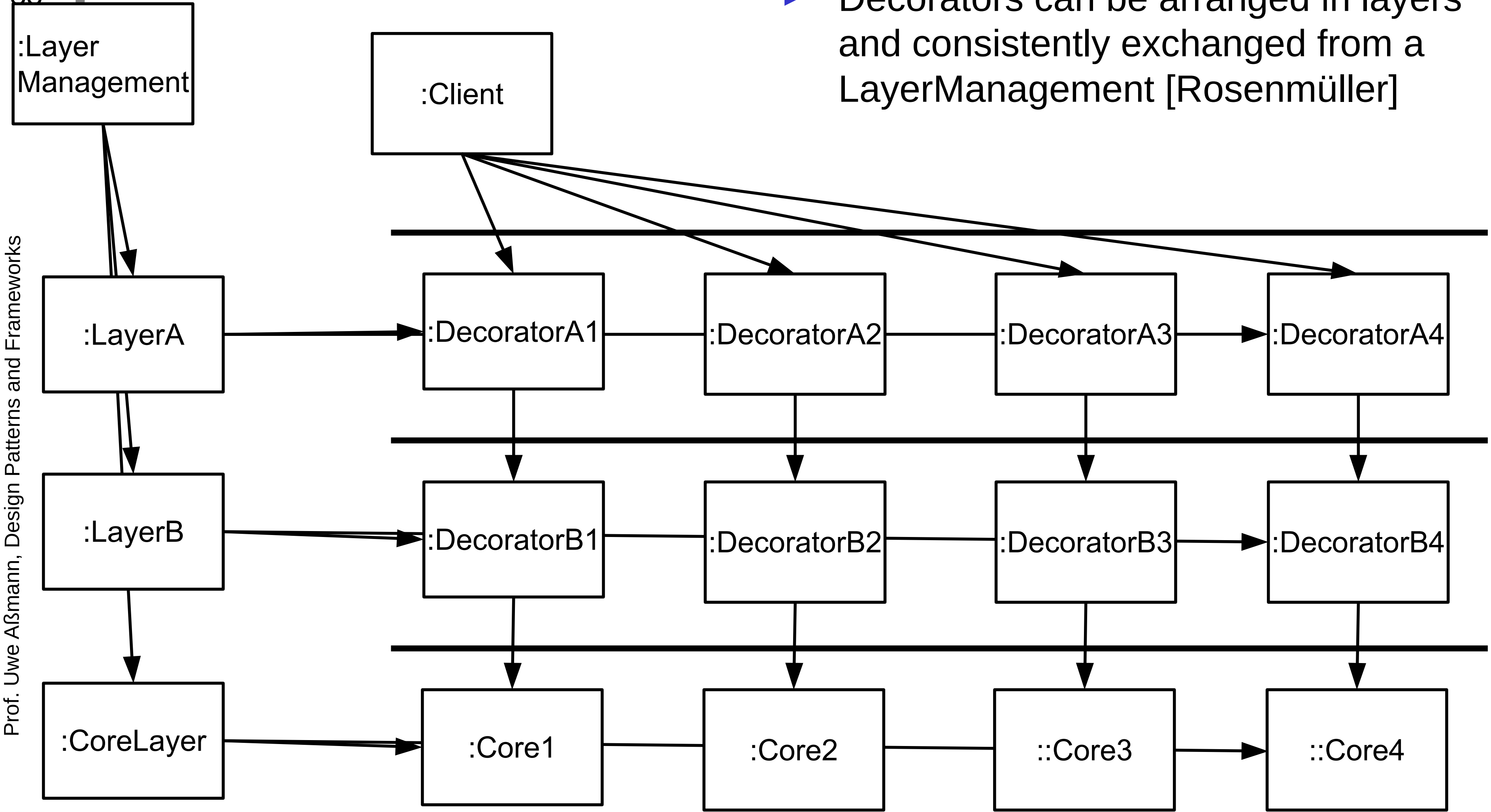


# Decorator Layers



# Decorator Layers

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

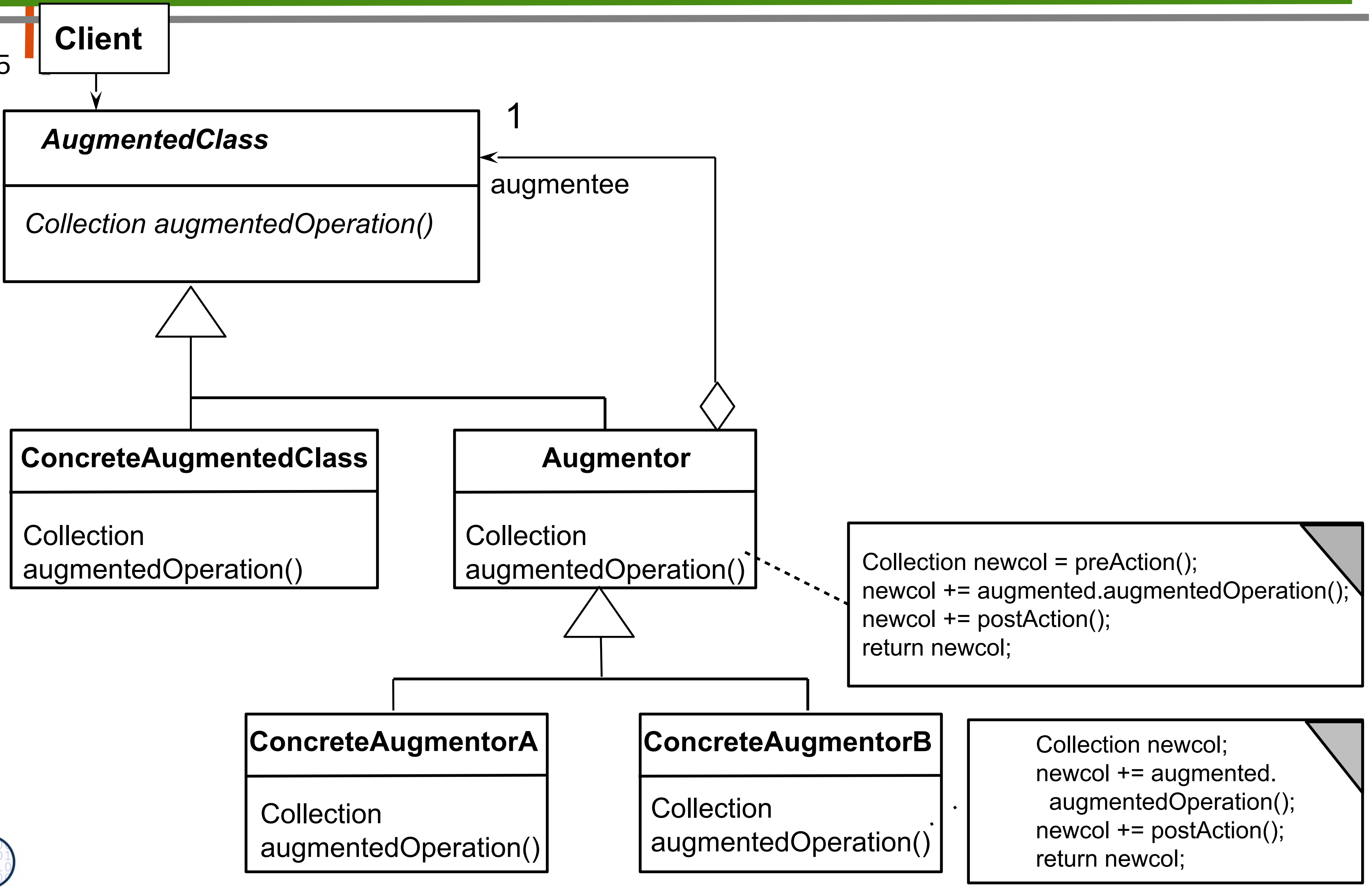


# 3.1.3.1 Augmentor

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



Client

*AugmentedClass*

*Collection augmentedOperation()*

1

augmentee

ConcreteAugmentedClass

Collection augmentedOperation()

Augmentor

Collection augmentedOperation()

```
Collection newcol = preAction();
newcol += augmented.augmentedOperation();
newcol += postAction();
return newcol;
```

ConcreteAugmentorA

Collection augmentedOperation()

ConcreteAugmentorB

Collection augmentedOperation()

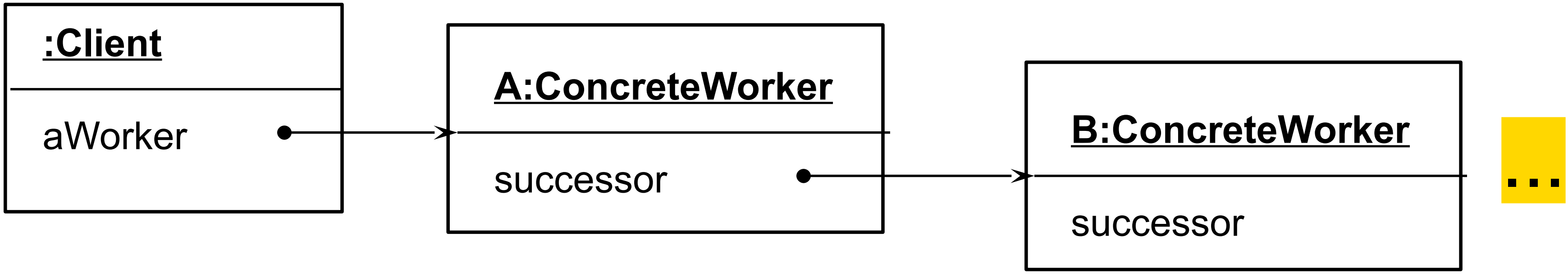
```
Collection newcol;
newcol += augmented.
  augmentedOperation();
newcol += postAction();
return newcol;
```

# 3.1.4 Chain of Responsibility

# Chain of Responsibility

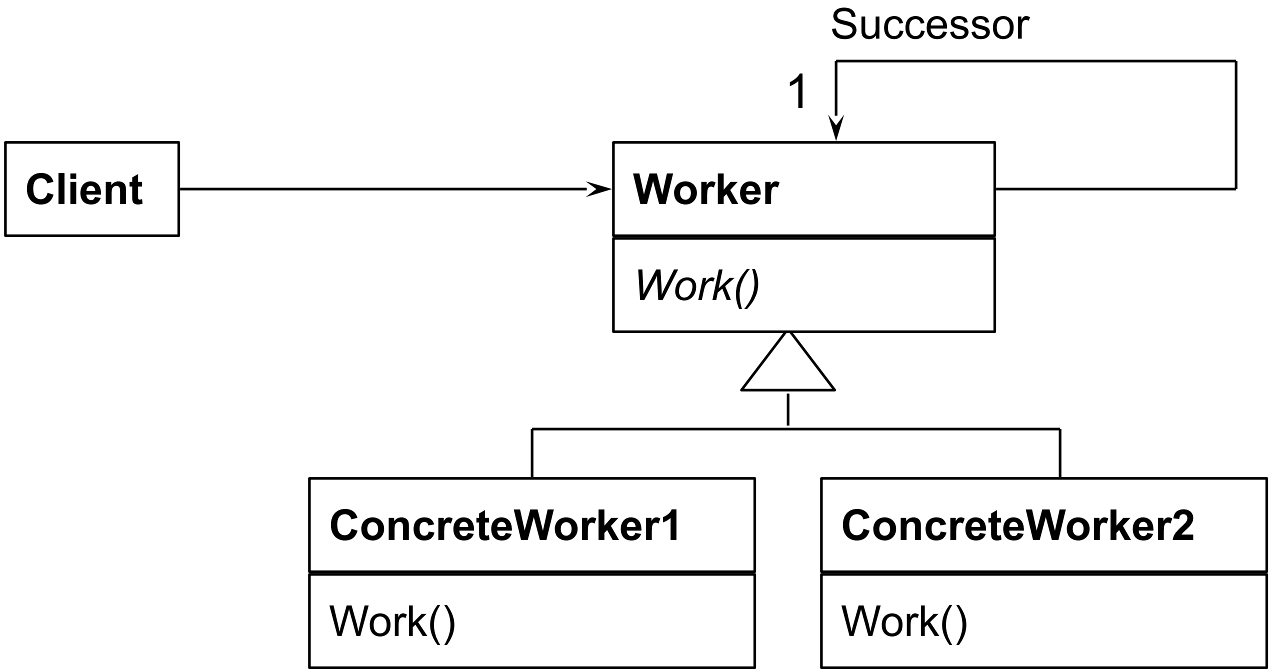
- Delegate an action to a list of delegatees that attempt to solve the problem one after the other
  - They delegate further on, down the chain (“daisy chain” principle)
  - No core object

ObjectStructure:



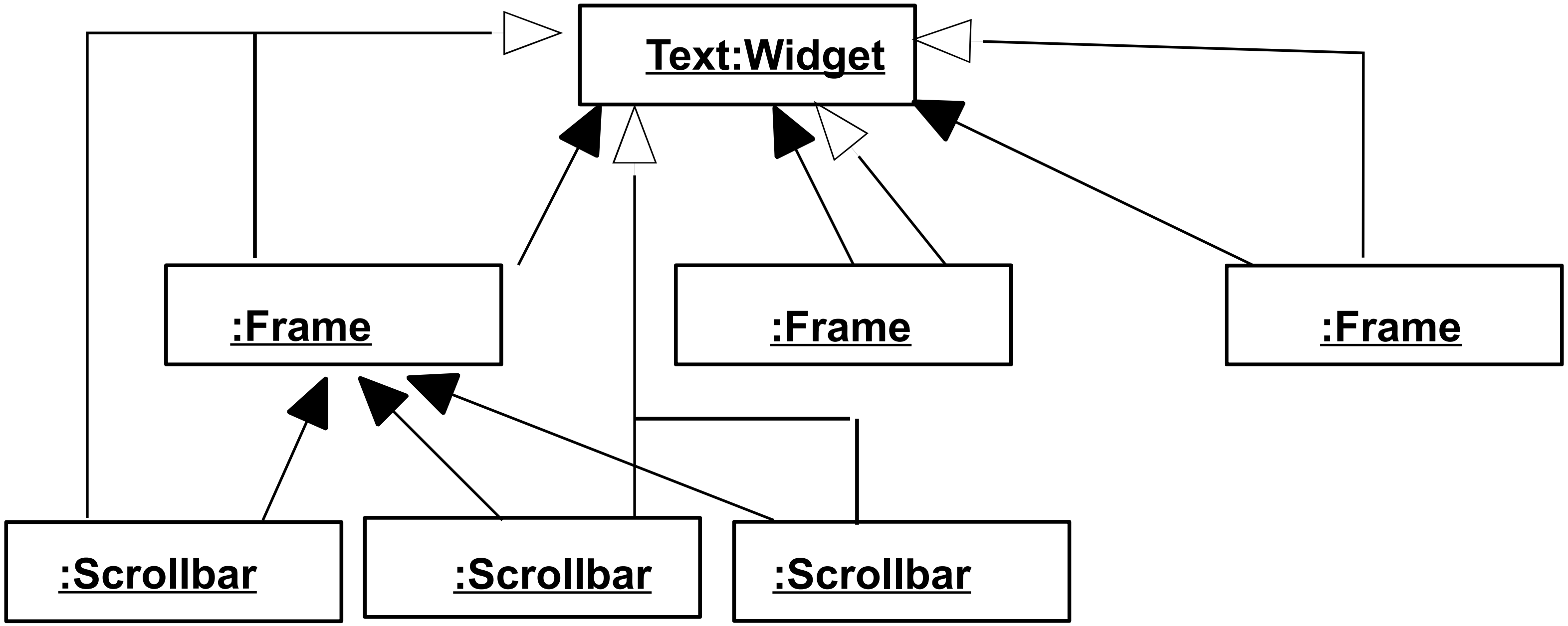
# Structure for ChainOfResponsibility

- 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)

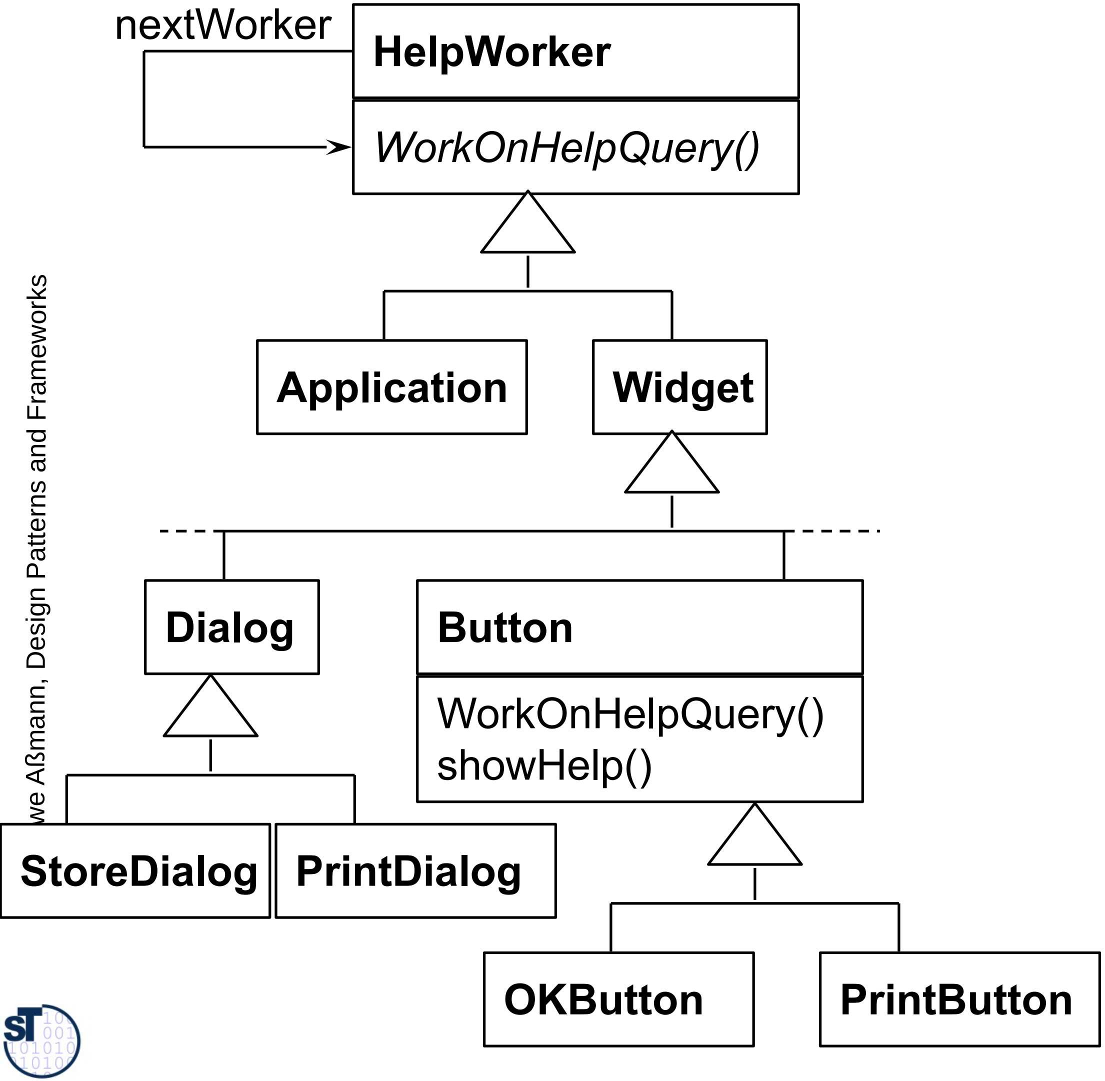


# Chains in Runtime Trees

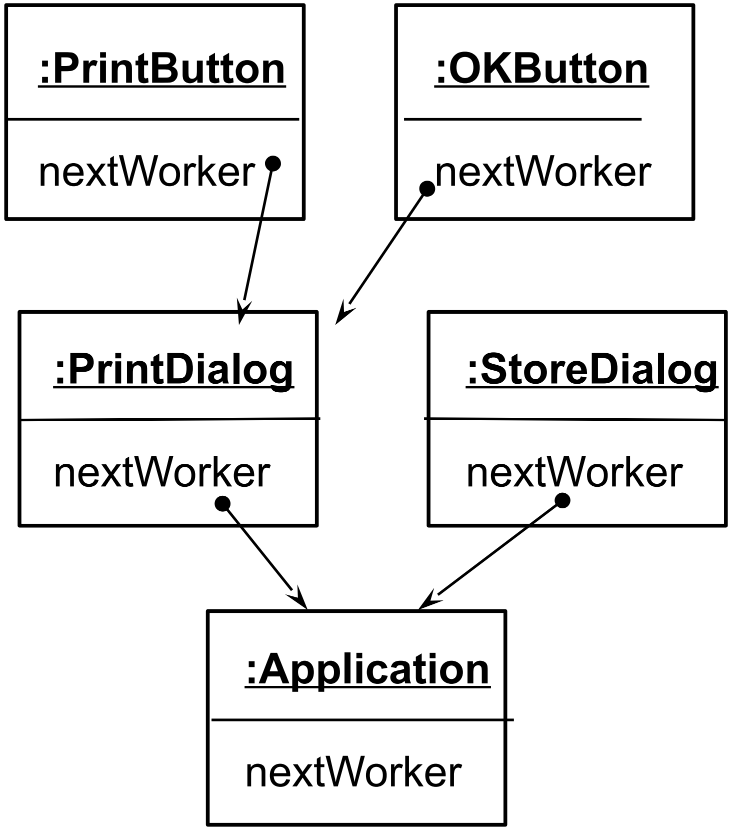
- Chains can also be parts of a tree
- Then, a chain is the path upward to the root of the tree



# Example ChainOfResponsibility Help System for a GUI



ObjectStructure is a Tree of Help Functions:



# Help System with Chain

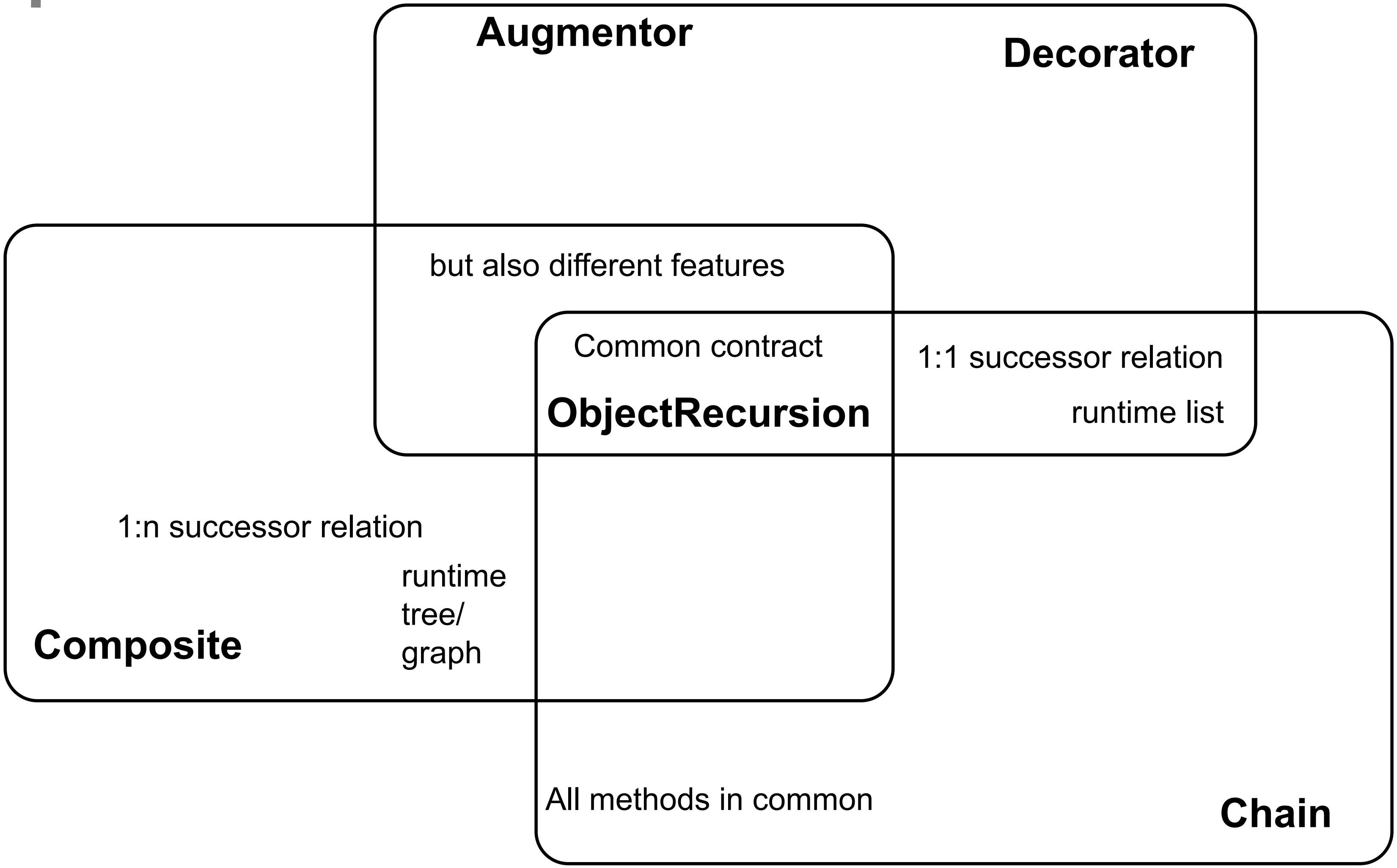
```
abstract class HelpWorker {
   HelpWorker nextWorker;  // here is the 1-
      recursion
    void workOnHelpQuery() {
    if (nextWorker)
      nextWorker.workOnHelpQuery();
    }  else { /* no help available */ }
}
class Widget extends HelpWorker {
    // this class can contain fixing code
}
class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

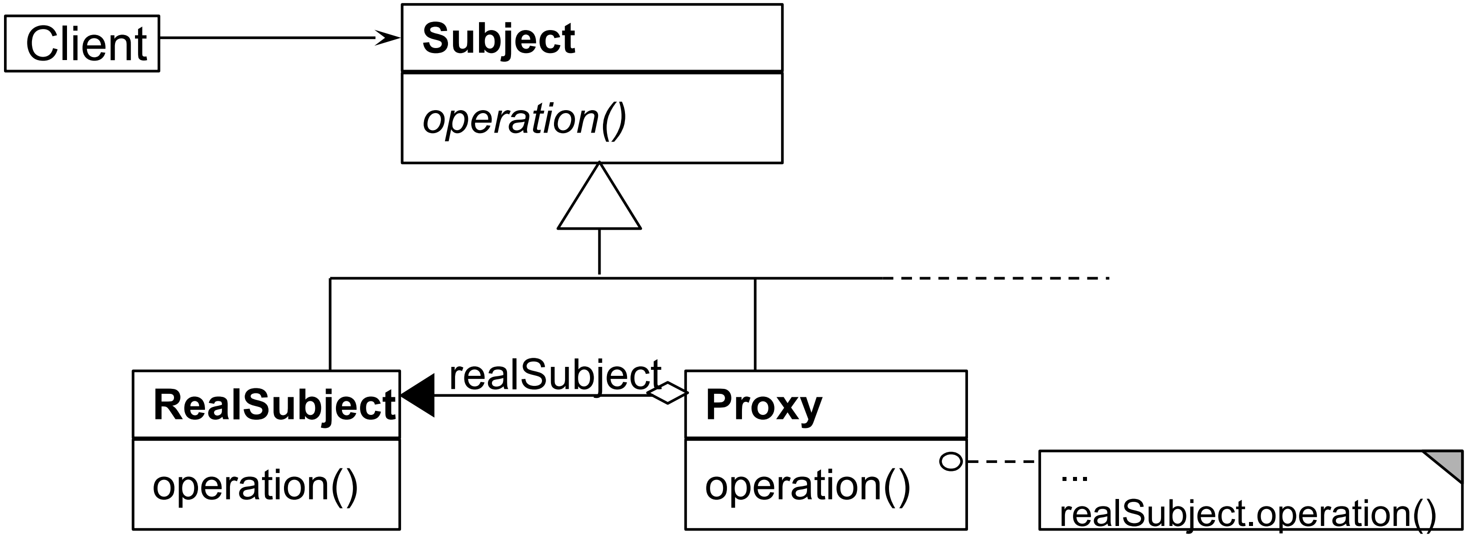
# Composite vs Decorator vs Chain



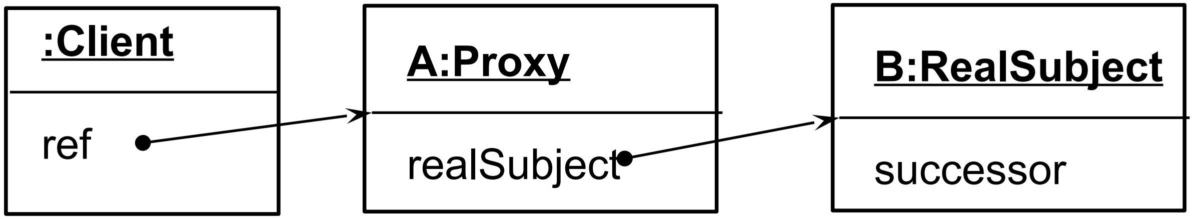
# 3.2. Flat Extensibility
# 3.2.1 Proxy

# 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 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.

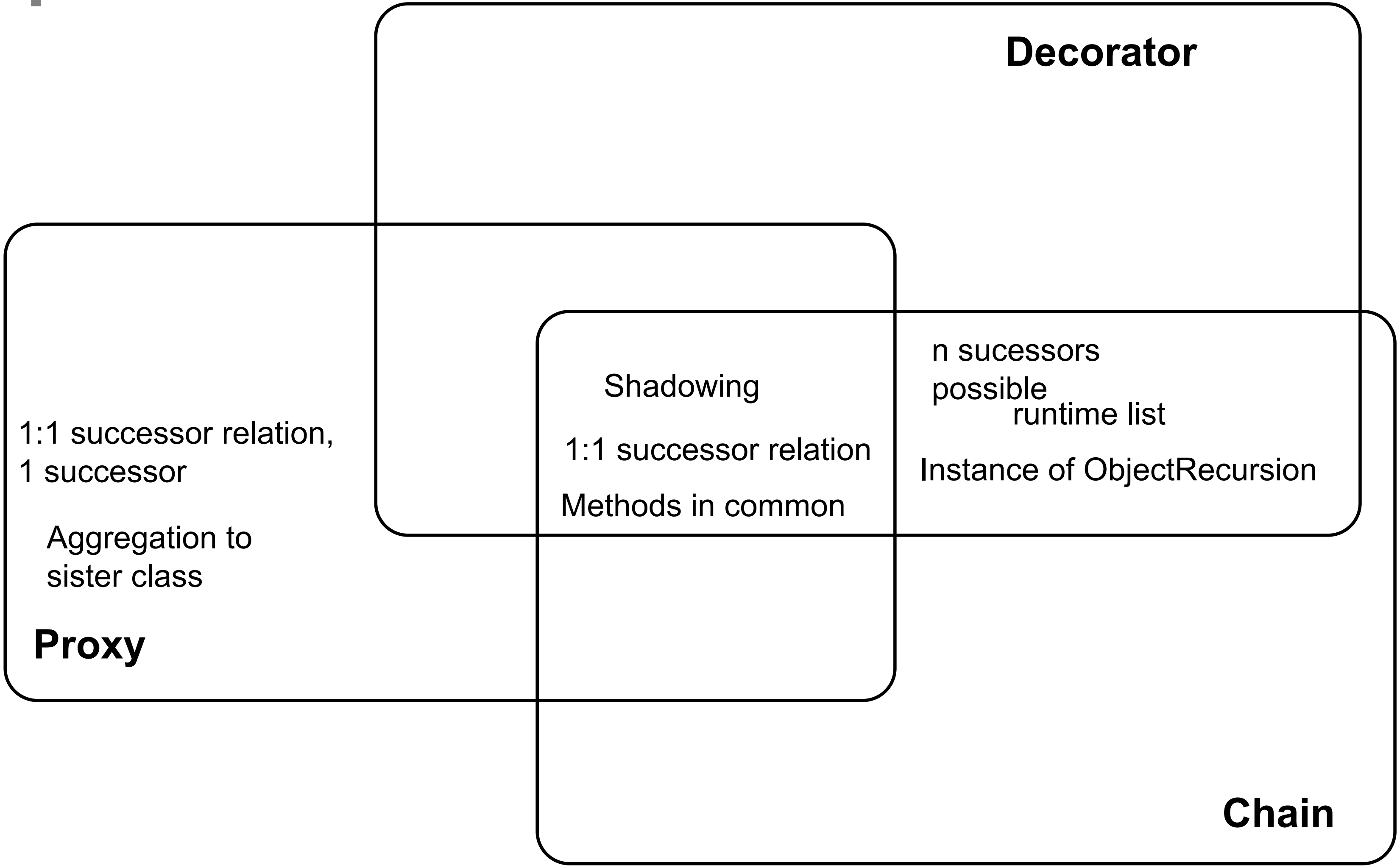
# Proxy Variants

- *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++, 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



Decorator

n sucessors possible

runtime list

Instance of ObjectRecursion

Shadowing

1:1 successor relation

Methods in common

1:1 successor relation,
1 successor

Aggregation to
sister class

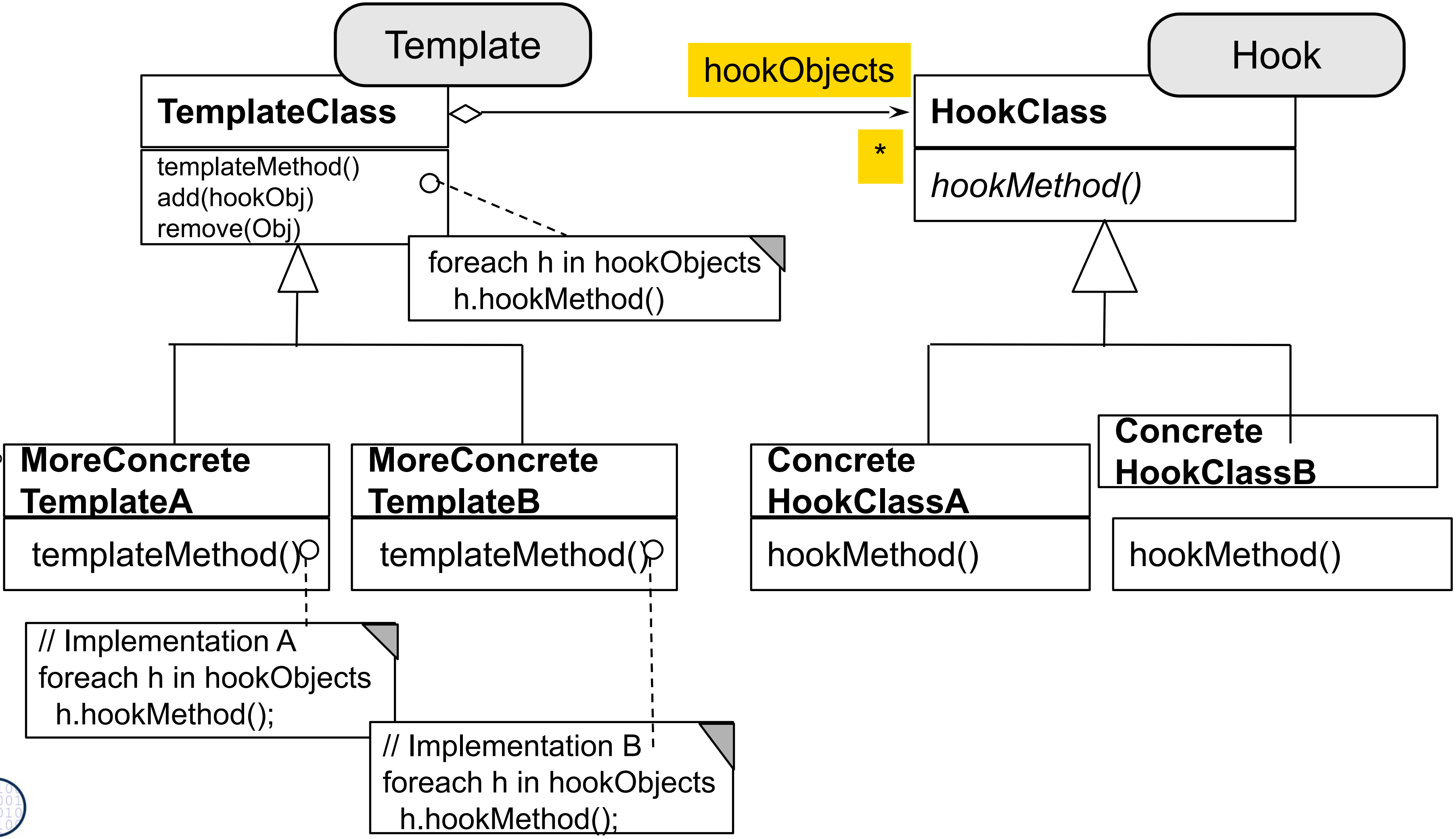
Proxy

Chain

# 3.2.2 Star-Bridge (*-Bridge)

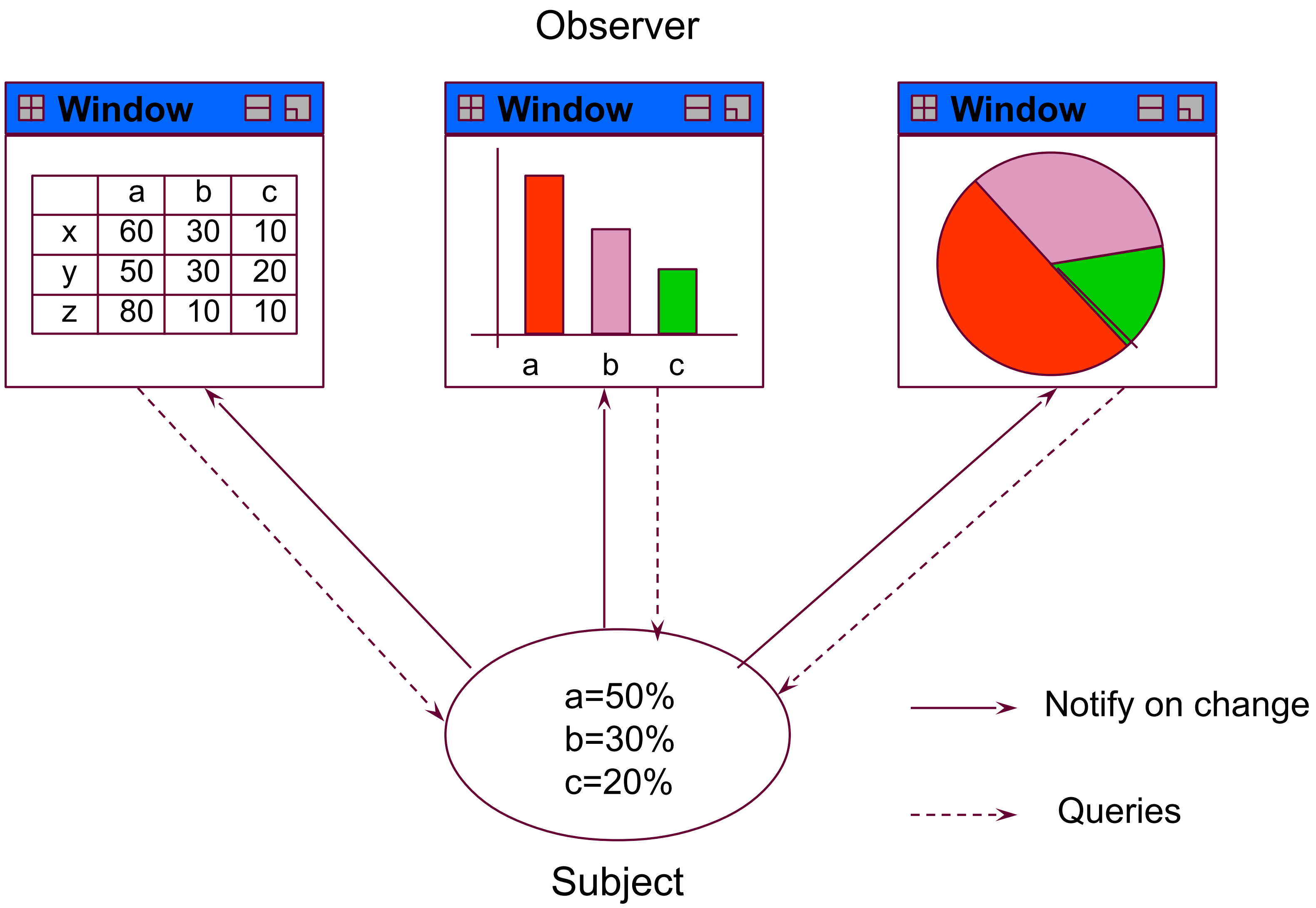
# Extensibility Pattern *DimensionalClassHierarchies (*Bridge)

- A bridge with a collection



# 3.2.3 Observer – (Event Bridge)

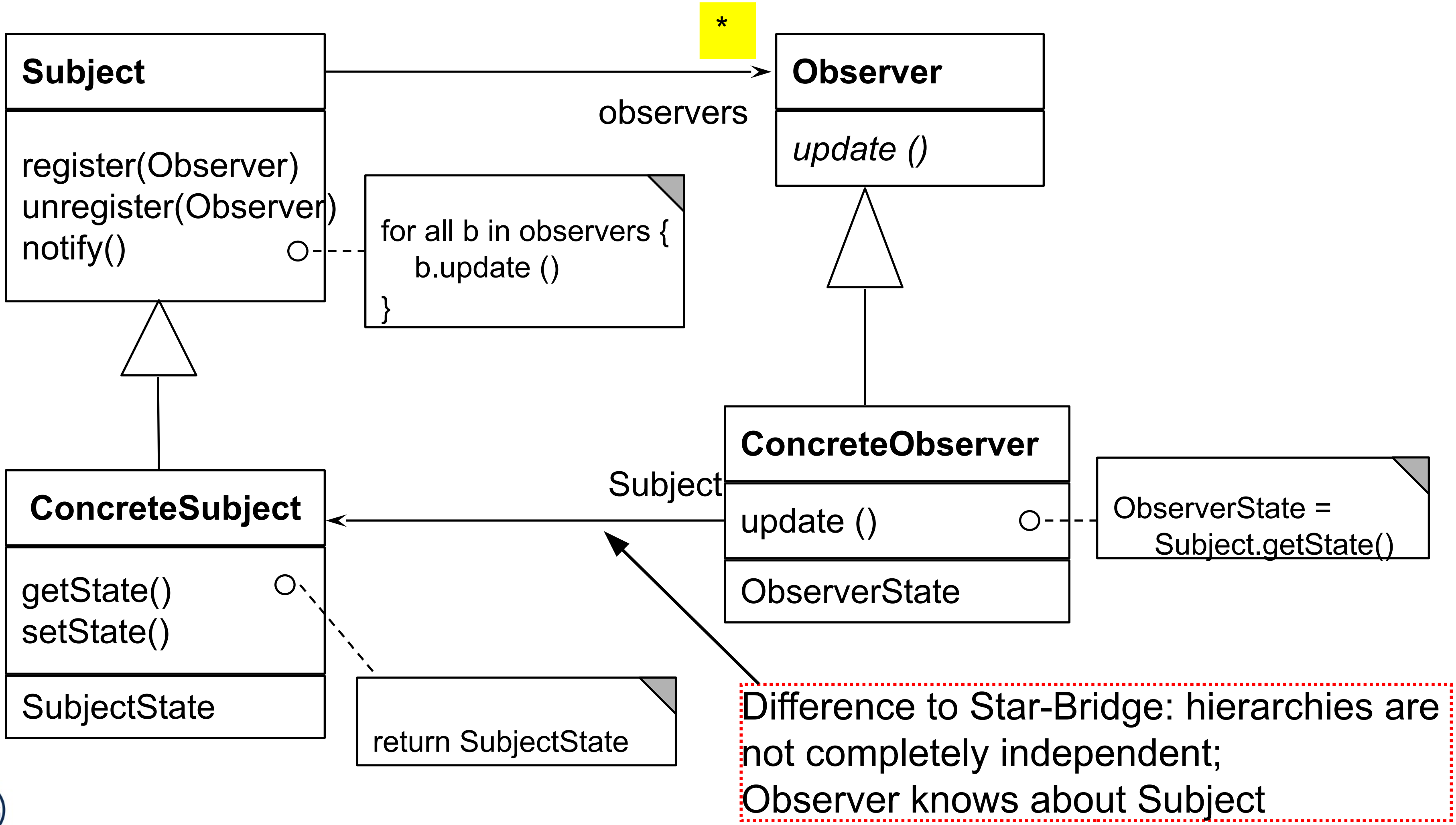
# Observer (Publisher/Subscriber, Event Bridge)



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

# Structure Observer

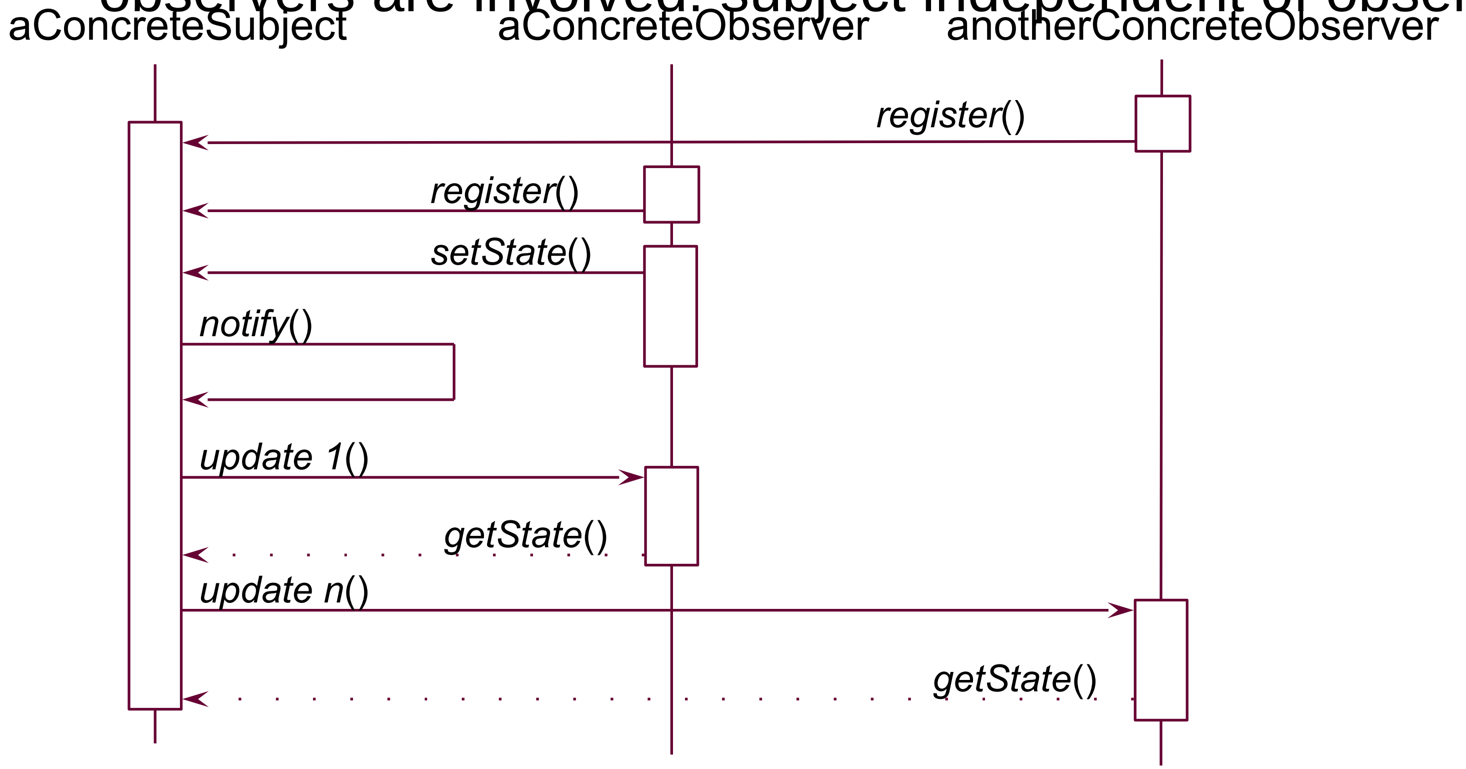
- Extension of Star-Bridge



Difference to Star-Bridge: hierarchies are not completely independent; Observer knows about Subject

# 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

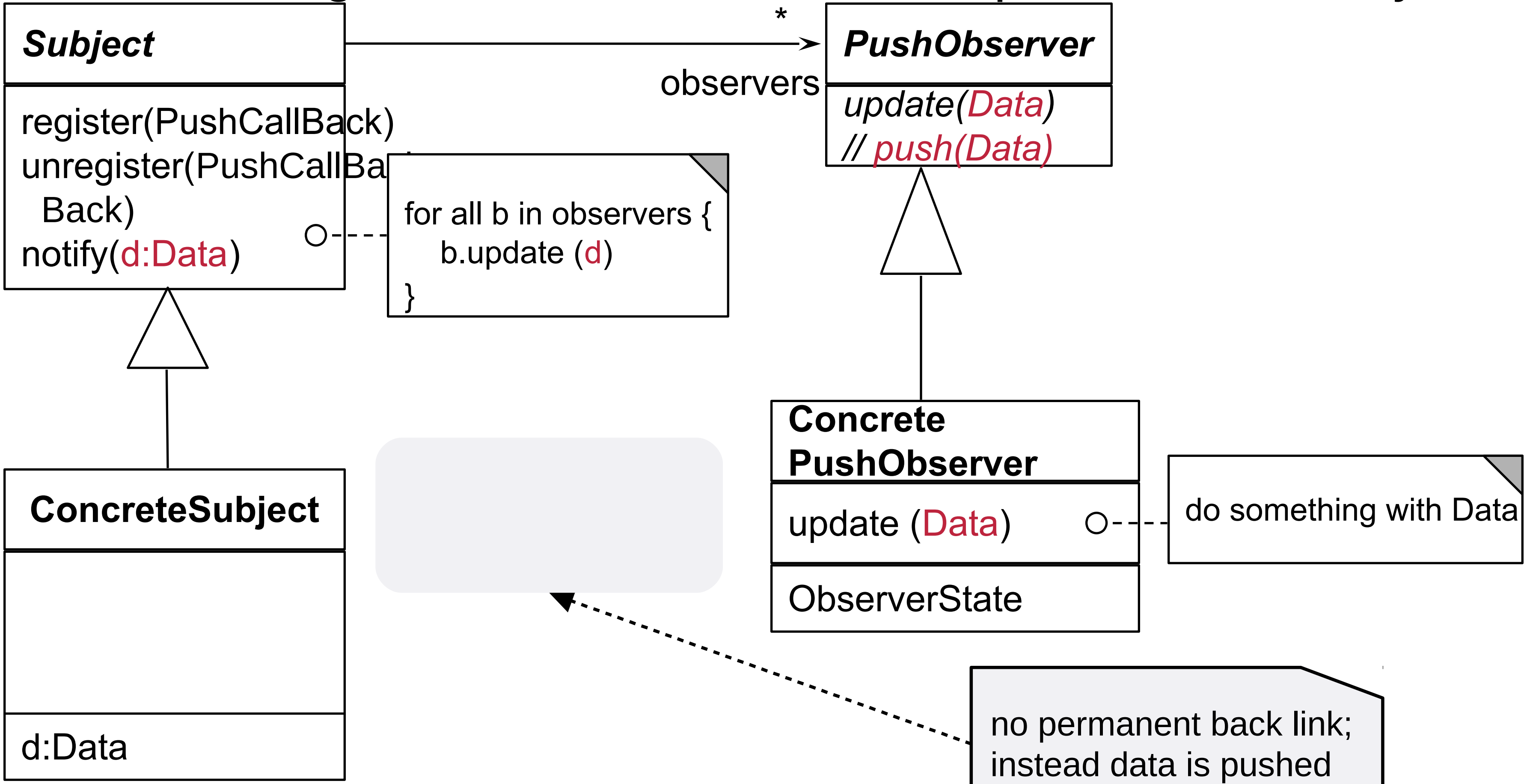


# Observer Variants

- **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**

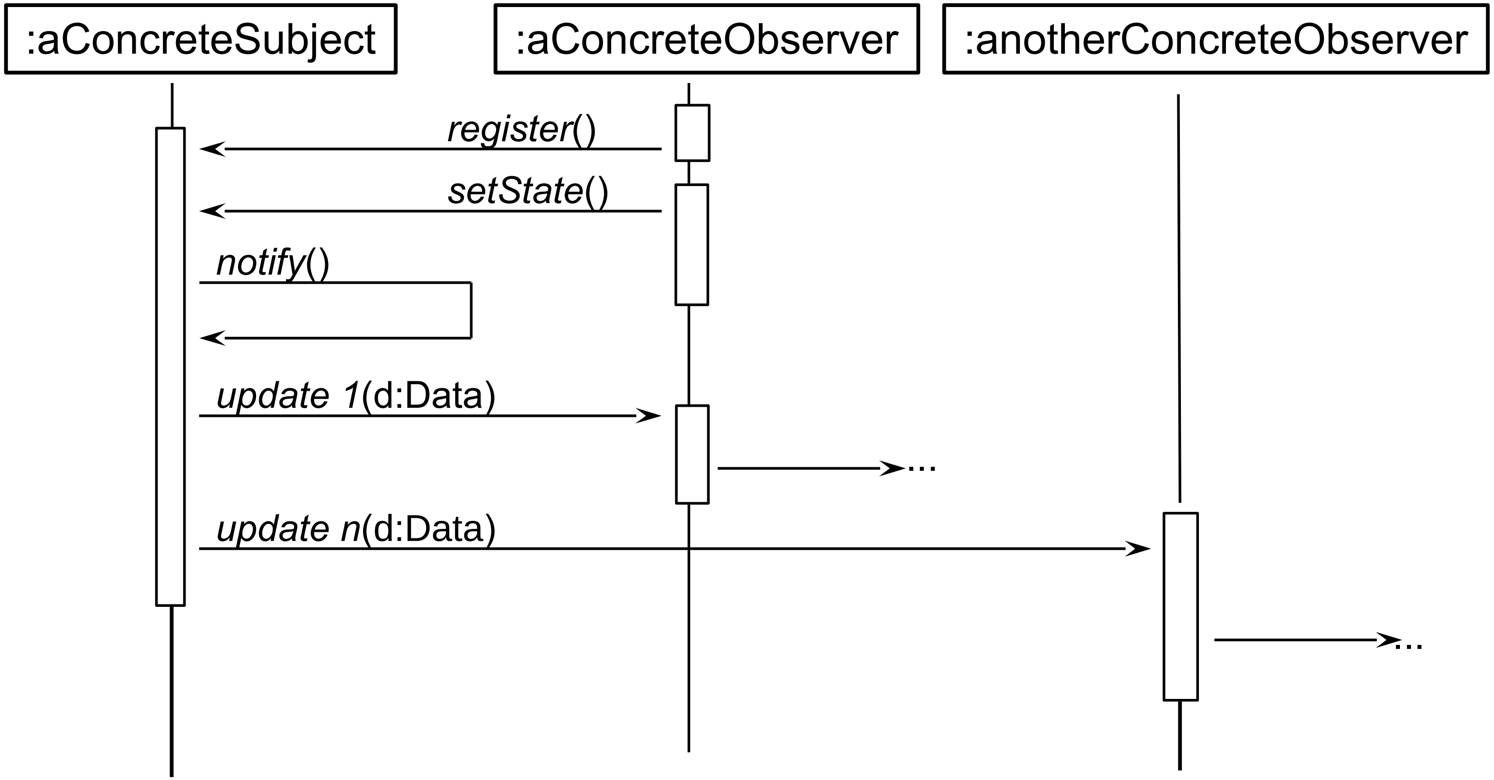
# Structure Data-Pushing-Observer

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



# Sequence Diagram Data-Push-Observer

- Update() transfers Data to Observer (push)

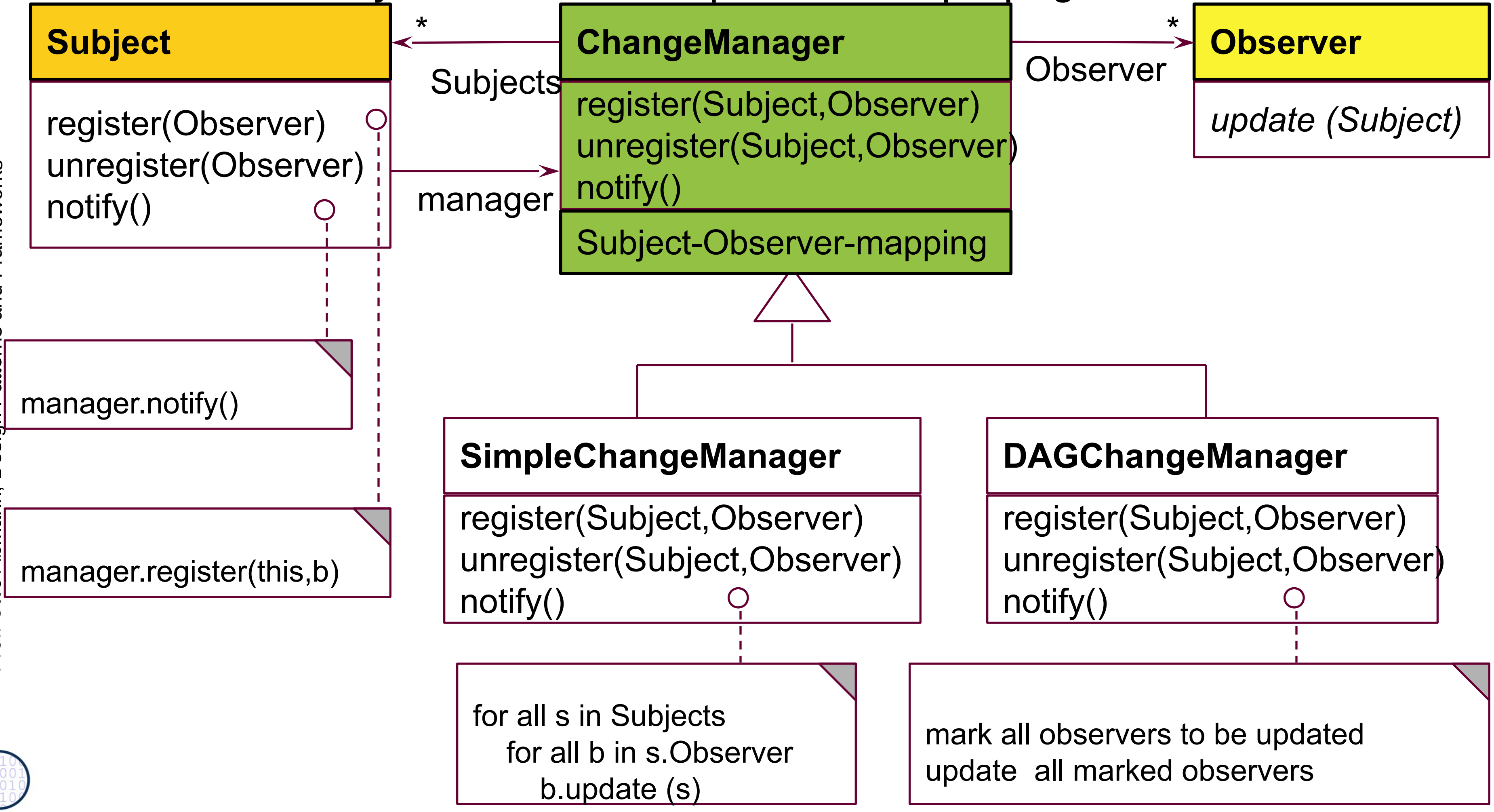


# Observer - Applications

- 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 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

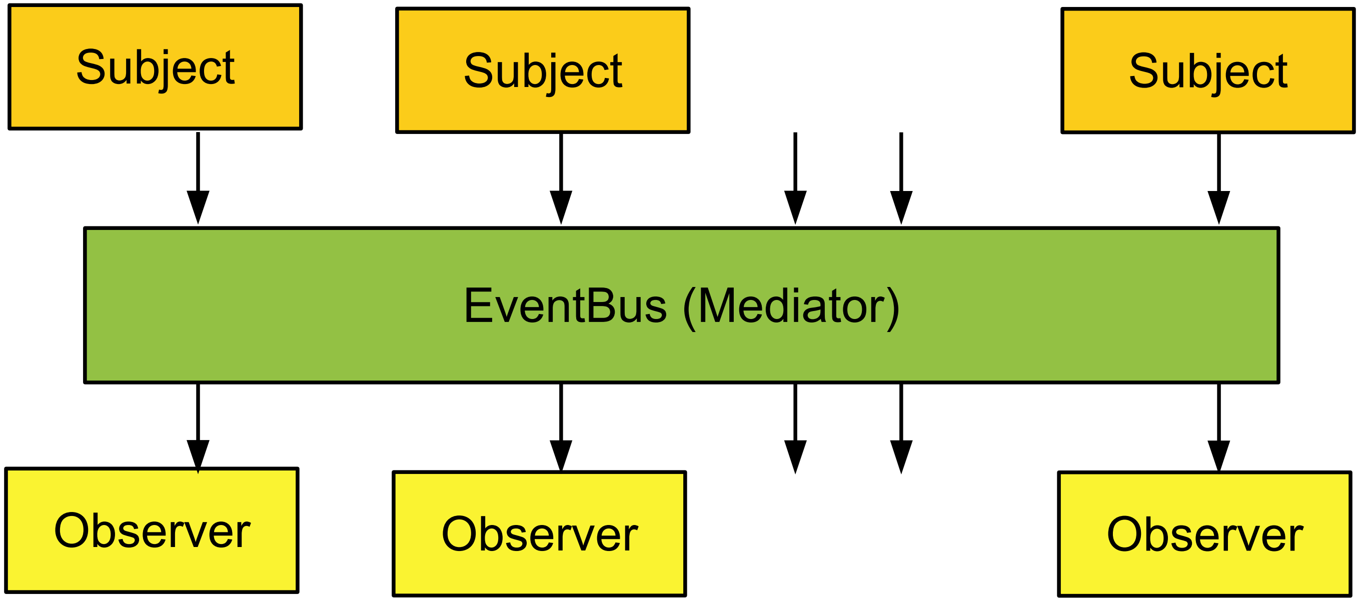
# Observer with ChangeManager (Mediator)

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

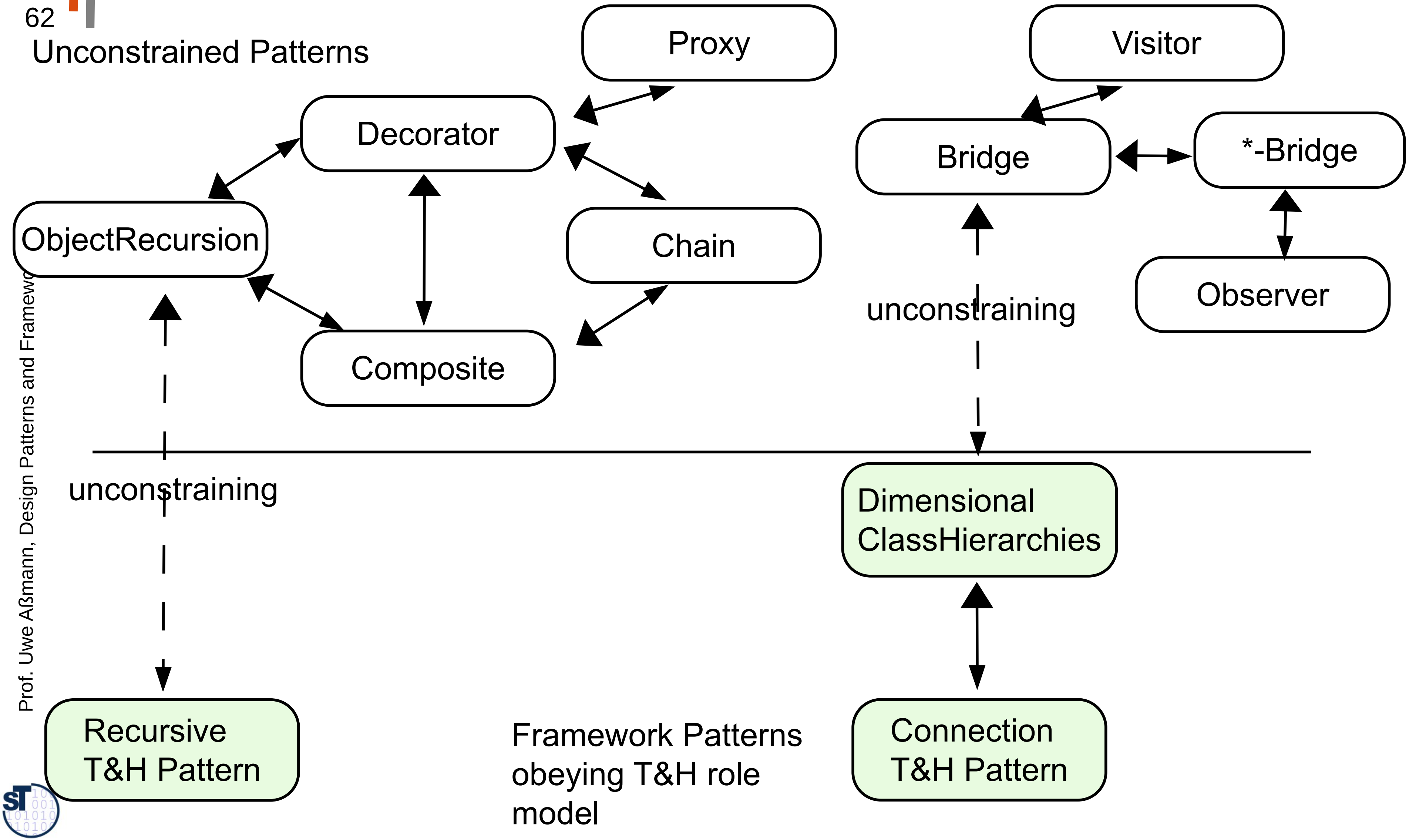


# ChangeManager is also Called Eventbus

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



# Relations Extensibility Patterns



Unconstrained Patterns

Proxy

Visitor

Decorator

Bridge

*-Bridge

ObjectRecursion

Chain

Observer

Composite

unconstraining

unconstraining

Dimensional ClassHierarchies

Recursive T&H Pattern

Framework Patterns obeying T&H role model

Connection T&H Pattern

# 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