25) Entwurfsmuster (Design Patterns) - Eine Einführung

Prof. Dr. Uwe Aßmann
Lehrstuhl Softwaretechnologie
Fakultät für Informatik
TU Dresden
13-1.1, 5/27/13

Achtung: Dieser Foliensatz ist in Englisch, weil das Thema in der Englisch-sprachigen Kurs “Design Patterns and Frameworks” wiederkehrt. Mit der Bitte um Verständnis.

Introductory Papers, Recommended

- A. Tesanovic. What is a pattern? Paper in Design Pattern seminar, IDA, 2001. Available at ST
  http://www-st.inf.tu-dresden.de/Lehre/WS04-05/dpf/seminar/tesanovic-WhatIsAPattern.pdf
- Brad Appleton. Patterns and Software: Essential Concepts and terminology.
  - http://csis.pace.edu/~grossman/dcs/Patterns%20and%20Software-%20Essential%20Concepts%20and%20Terminology.pdf Compact introduction into patterns.

Other References

- [The GOF (Gang of Four) Book] E. Gamma, R. Johnson, R. Helm, J. Vlissides. Design Patterns. Addison-Wesley.
- Head First Design Patterns. Eric Freeman & Elisabeth Freeman, mit Kathy Sierra & Bert Bates.O'Reilly, 2004, ISBN 978-0-596-00712-6
  - German Translation: Entwurfsmuster von Kopf bis Fuß. Eric Freeman & Elisabeth Freeman, mit Kathy Sierra & Bert Bates. O'Reilly, 2005, ISBN 978-3-89721-421-7
- F. Buschmann. N. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-orientierte Software-Architektur. Addison-Wesley.
  - Design patterns and architectural styles. MVC, Pipes, u.v.m.
- M. Fowler. Refactoring. Addison-Wesley.
- Papers:
  - D. Riehle, H. Zülinghoven, Understanding and Using Patterns in Software Development. Theory and Practice of Object Systems 2 (1), 1996. Explains different kinds of patterns. http://citeseer.ist.psu.edu/riehle96understanding.html
  - W. Zimmer. Relationships Between Design Patterns. Pattern Languages of Programming (PLOP) 1995.

- MVC
  - http://exadel.com/tutorial/struts/5.2/guess/strutsintro.html
  - http://www.c2.com/cgi/wiki?ModelViewController
- “Quality without a name” (QWAN principle)
  - http://en.wikipedia.org/wiki/The_Timeless_Way_of_Building

Thus it will be seen that engineering is a distinctive and important profession. To some even it is the topmost of all professions. However true that may or may not be to-day, certain it is that some day it will be true, for the reason that engineers serve humanity at every practical turn. Engineers make life easier to live—easier in the living; their work is strictly constructive, sharply exact; the results positive. Not a profession outside of the engineering profession but that has its moments of wabbling and indecision—of faltering on the part of practitioners between the true and the untrue. Engineering knows no such weakness. Two and two make four. Engineers know that. Knowing it, and knowing also the unnumbered possible approach a problem with a certainty of conviction and a confidence in the powers of their working-tools nowhere permitted men outside the profession.

Charles M. Horton. Opportunities of engineering. www.gutenberg.org, eBook #24681; Harper and Brothers, 1922.

Why is the Frauenkirche Beautiful?

History: How to Write *Beautiful* Software

- Beginning of the 70s: the window and desktop metaphors (conceptual patterns) are discovered by the Smalltalk group in Xerox Parc, Palo Alto
- 1978/79: Goldberg and Reenskaug develop the MVC pattern for user Smalltalk interfaces at Xerox Parc
  - During porting Smalltalk-78 for the Eureka Software Factory project
- 1979: Alexander's Timeless Way of Building
  - Introduces the notion of a *pattern* and a *pattern language*
- 1987: W. Cunningham, K. Beck OOPSLA paper “Using Pattern Languages for Object-Oriented Programs” discovered Alexander's work for software engineers by applying 5 patterns in Smalltalk
- 1991: Erich Gamma's PhD Thesis about Design Patterns
  - Working with ET++, one of the first window frameworks of C++
  - At the same time, Vlissides works on InterViews (part of Athena)
- 1991: Pattern workshop at OOPSLA 91, organized by B. Anderson
- 1993: E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. ECOOP 97, LNCS 707, Springer
- 1995: First PLOP conference (Pattern Languages Of Programming)
- 1995: GOF book

The Most Popular Definition

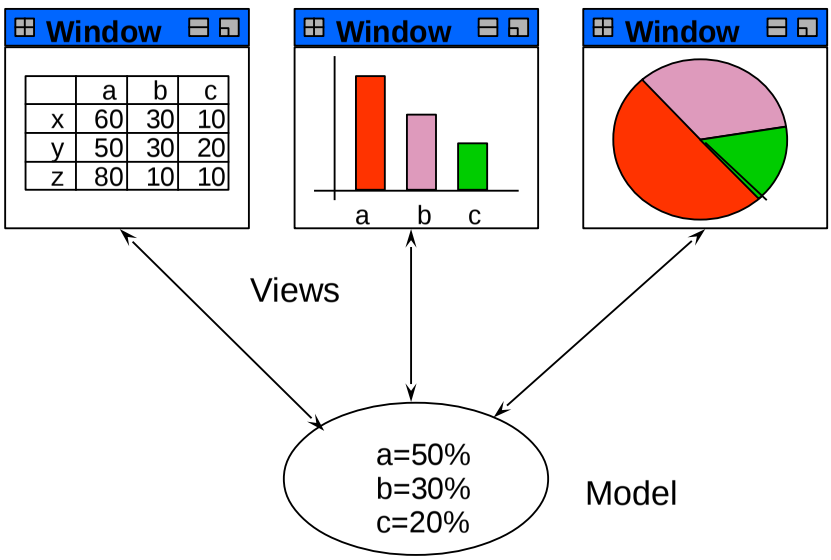
A **Design Pattern (Entwurfsmuster)** is a *solution pattern,*

- a description of a standard solution for
- a frequent design problem
- in a certain context

- Goal of a Design Pattern: Reuse of design information
  - A pattern must not be “new”!
  - A pattern writer must have a “aggressive disregard for originality”
- Such *solution patterns* are well-known in every engineering discipline
  - Mechanical engineering
  - Electrical engineering
  - Civil engineering and architecture

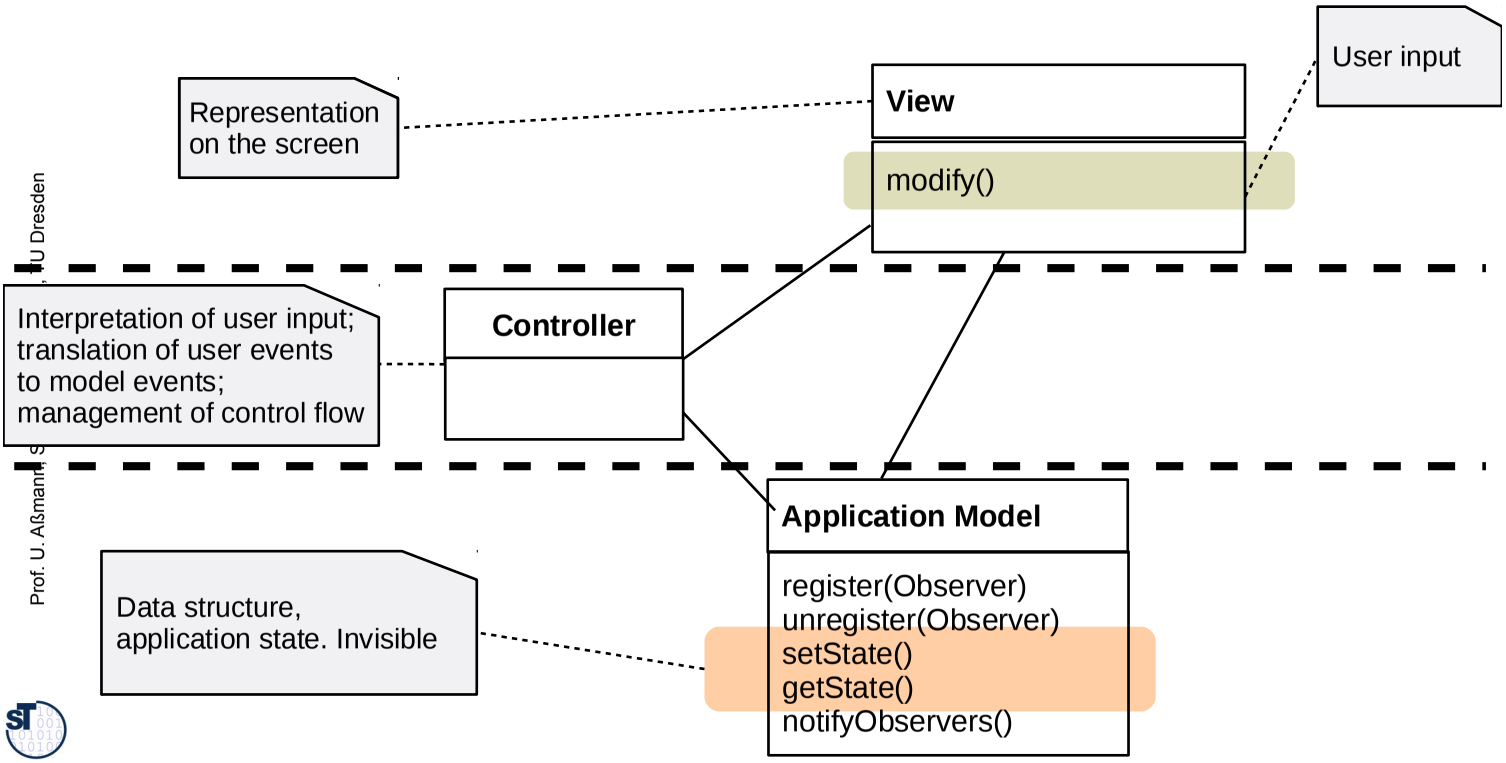
A Problem in Interactive Applications

- How do I display and edit a data structure on the screen?
  - Reaction on user inputs?
  - Maintaining several views
  - Adding and removing new views
- Solution: Model-View-Controller pattern (MVC), a set of classes to control a data structure behind a user interface
  - Developed by Goldberg/Reenskaug in Smalltalk 1978



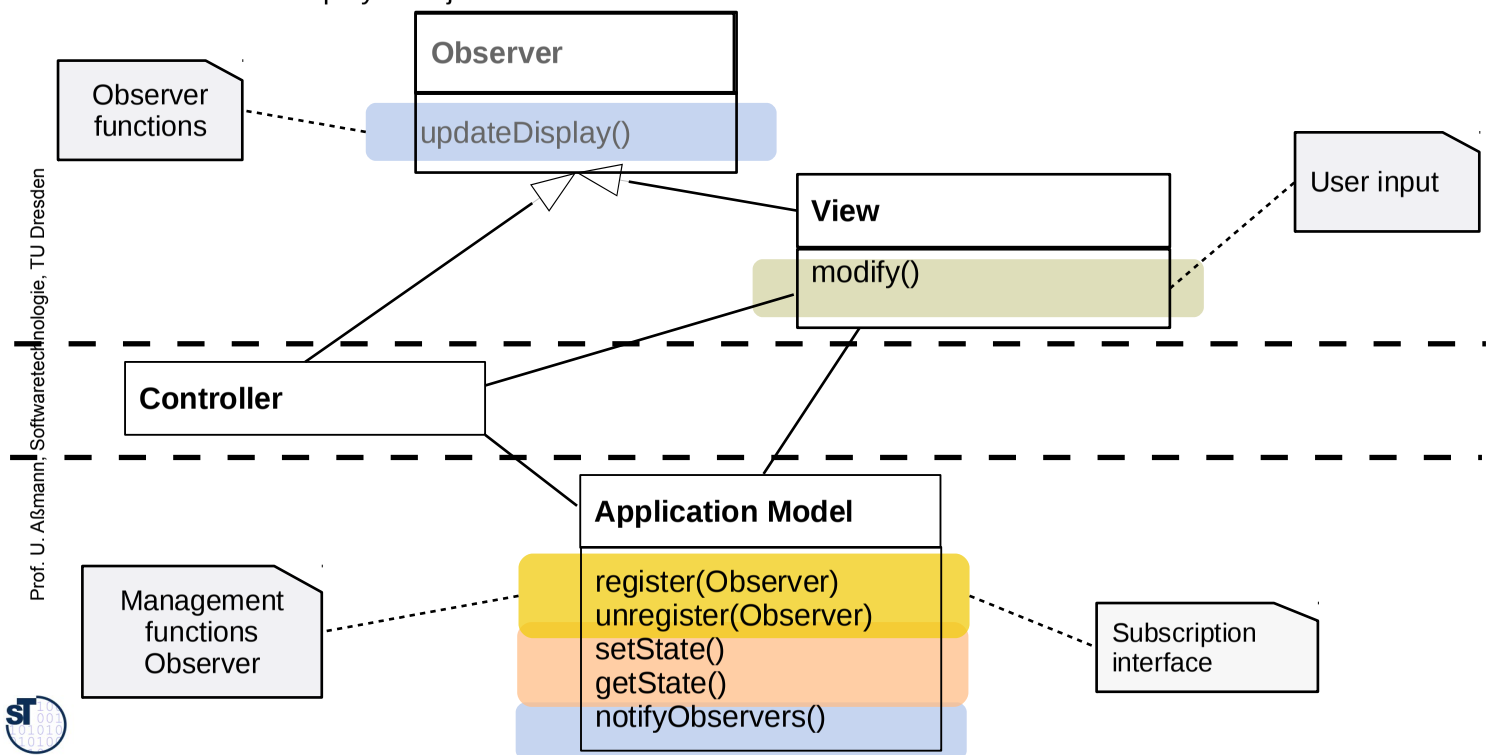
Design Pattern Model/View/Controller (MVC)

- MVC is a set of classes to control a data structure behind a user interface
- Layered structure of View, Controller and ApplicationModel



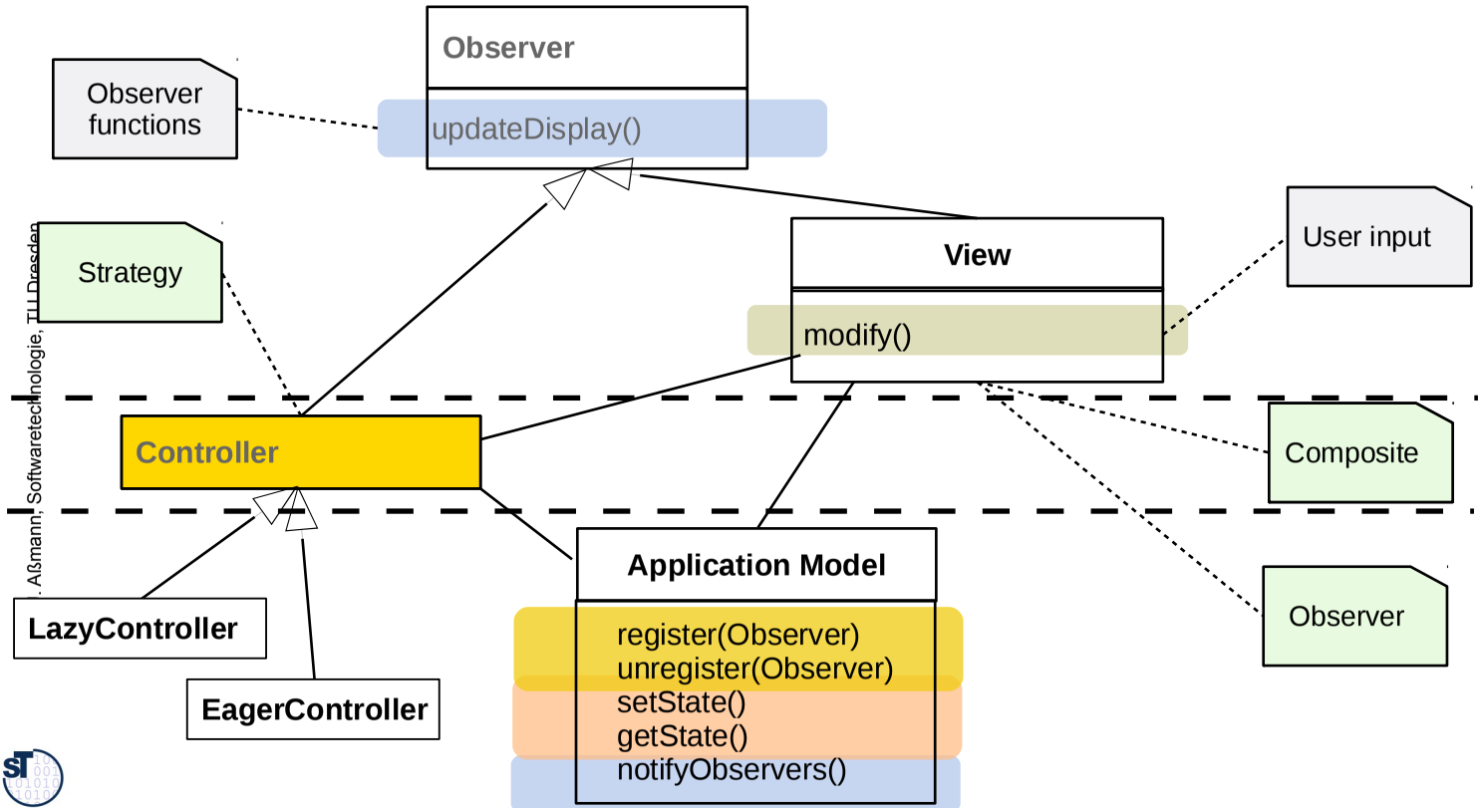
Design Pattern Model/View/Controller (MVC)

- The MVC is a complex design pattern. The layers are connected by the simpler patterns Observer, Composite, Strategy.
  - The Controller interpretes the input of the user and transmits them into actions on the model
  - Controller and View play Listener role from *Observer* (asynchronous communication)
  - Model plays Subject role



Design Pattern Model/View/Controller (MVC)

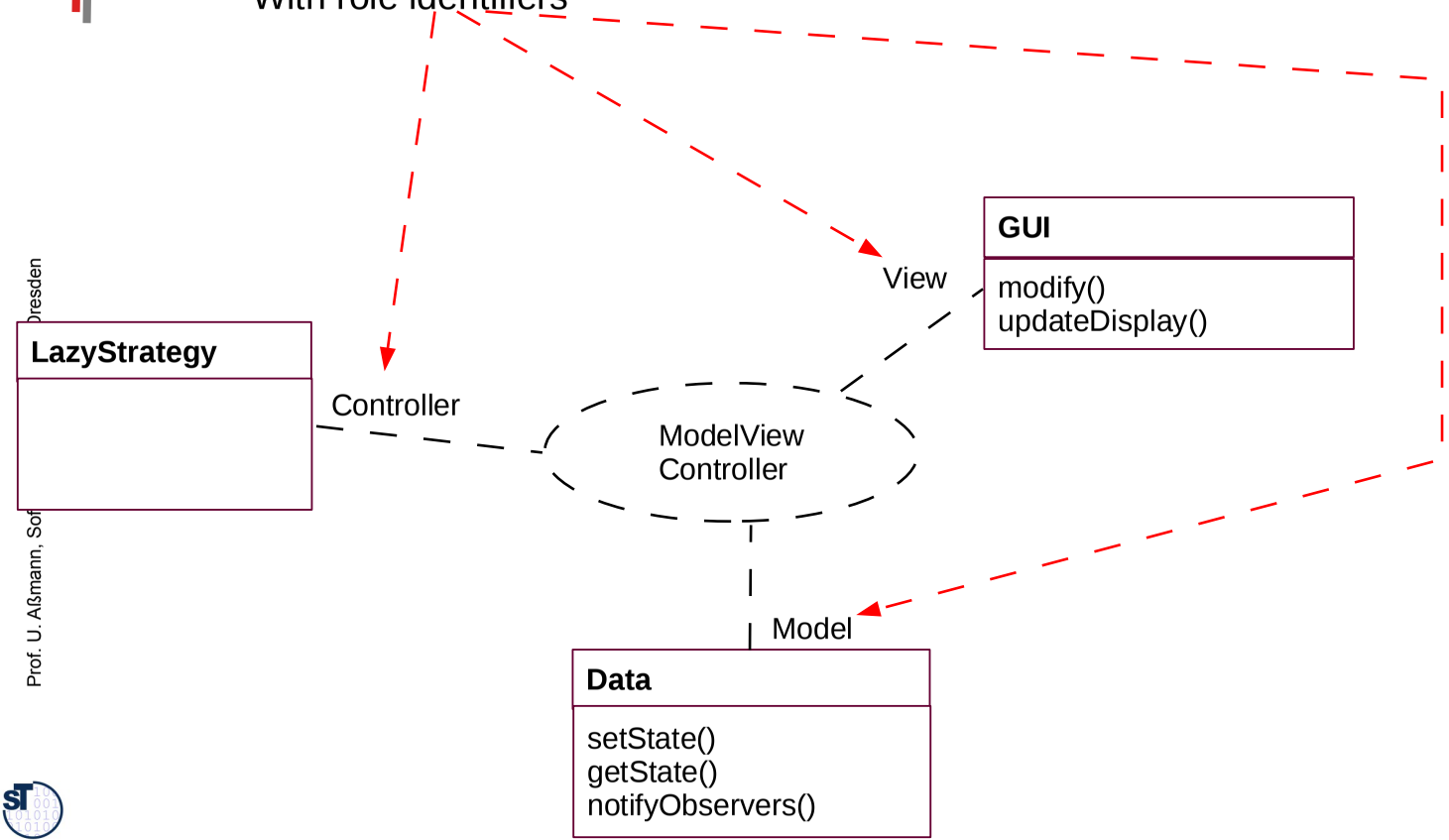
- Controller follows Strategy pattern (variation of updating the screen)
- Relation within Views by Composite (tree-formed views)



Prof. U. Aßmann, Softwaretechnologie, TU Dresden

Design Pattern Model/View/Controller (MVC)

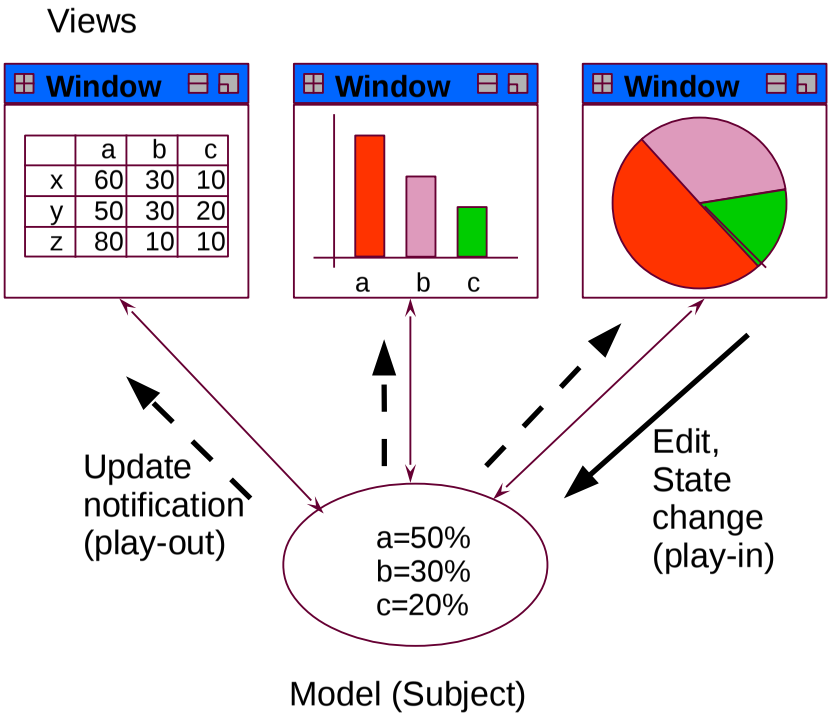
- UML has a specific notation for patterns (**collaboration classes**)
  - With role identifiers



Prof. U. Aßmann, Softwaretechnologie, TU Dresden

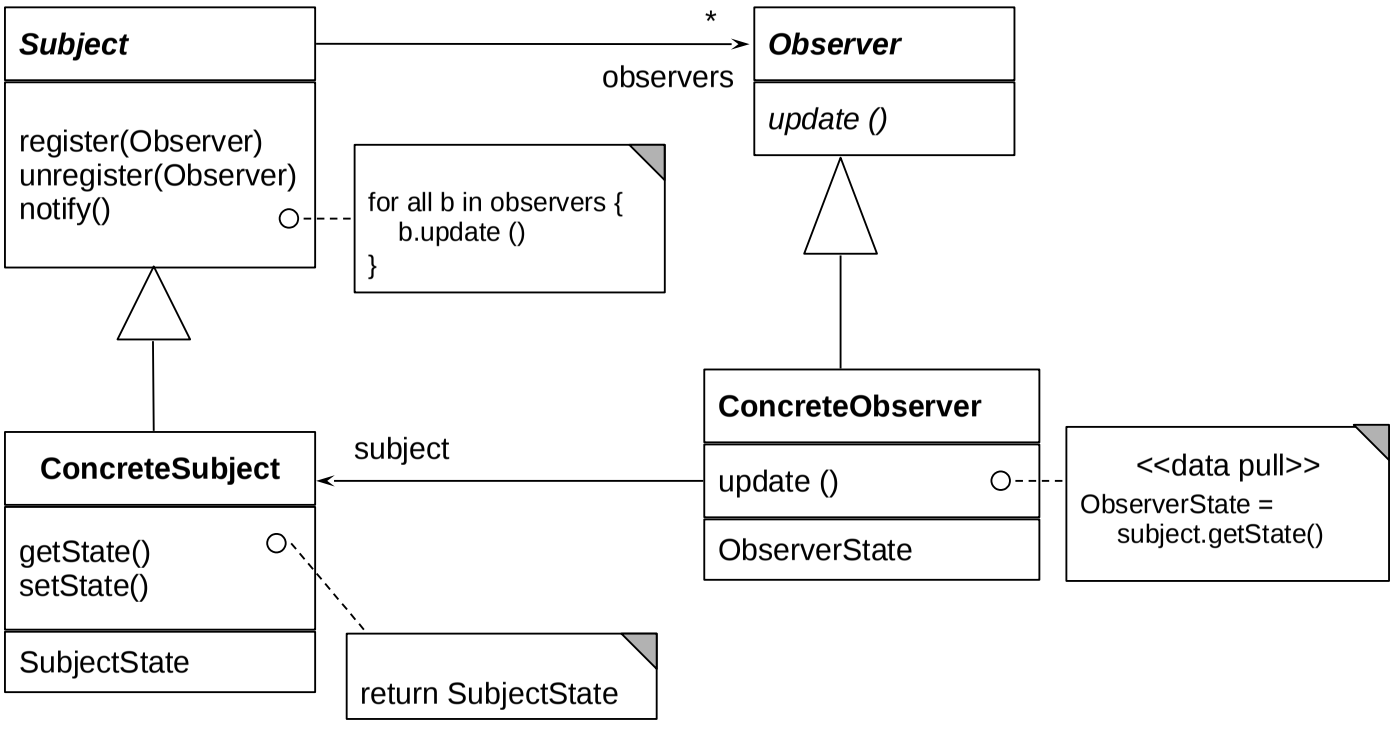
Pattern 1: Observer

- Views may register as Observer at the model (Subject)
  - They become *passive* observers of the model
  - They are notified if the model changes.
  - Then, every view updates itself by accessing the data of the model.
- Views are independent of each other
  - The model does not know how views visualize it
  - Observer decouples views strongly



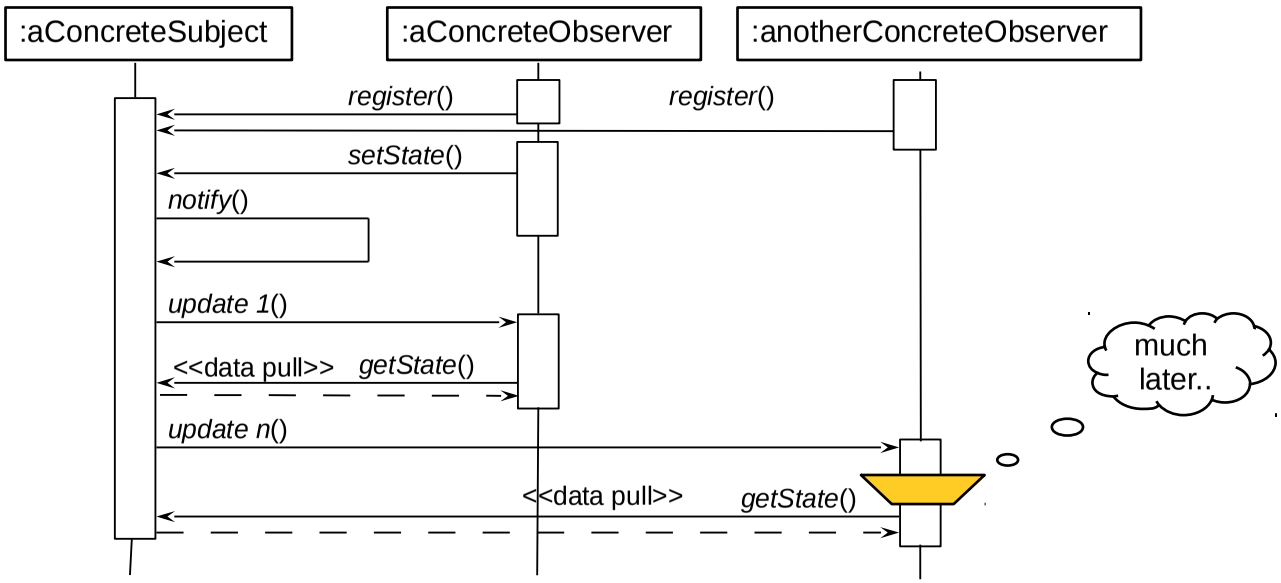
Structure Observer (pull-Variant)

- Aka Publisher/Subscriber
- Subject does not care nor know, which observers are involved: subject independent of observer



Sequence Diagram pull-Observer

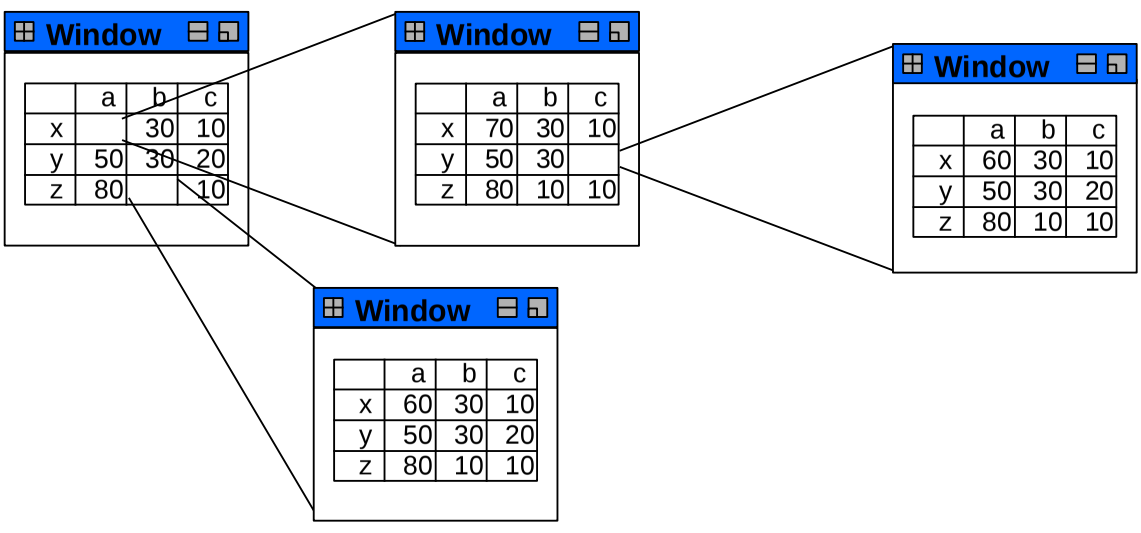
- Observer.update() does not transfer data, only announces an event
  - Anonymous communication possible
- Observer *pulls* data out itself
  - In the context of MVC, Controller or View pull data out of the application model themselves



Pattern 2: Composite (Rpt.)

Views may be *nested* (*Composite*)

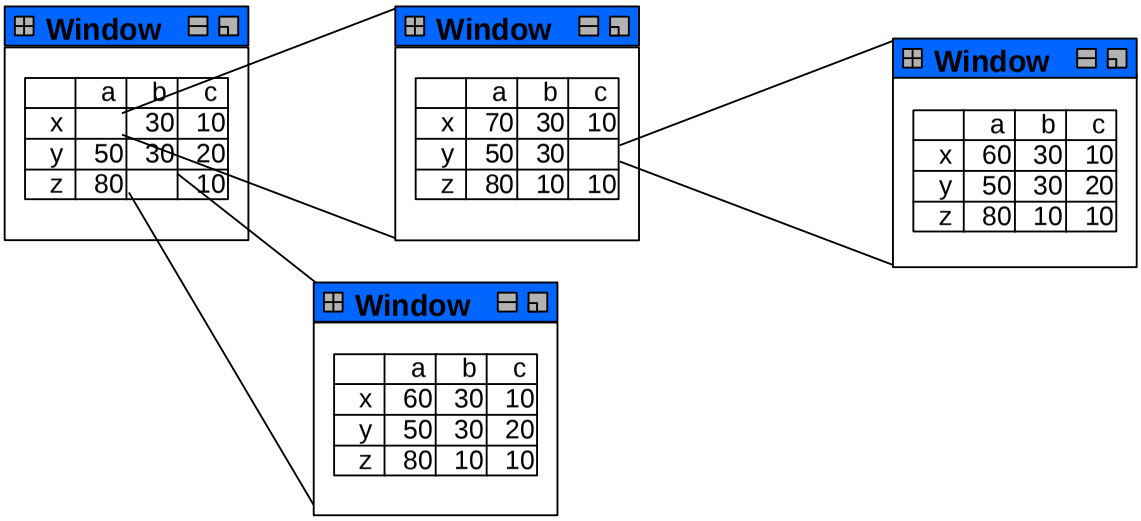
- Composite represents trees
- For a client class, Compositum unifies the access to root, inner nodes, and leaves
- In MVC, views can be organized as Composite



Pattern 2: Composite (Rpt.)

Views may be *nested* (*Composite*)

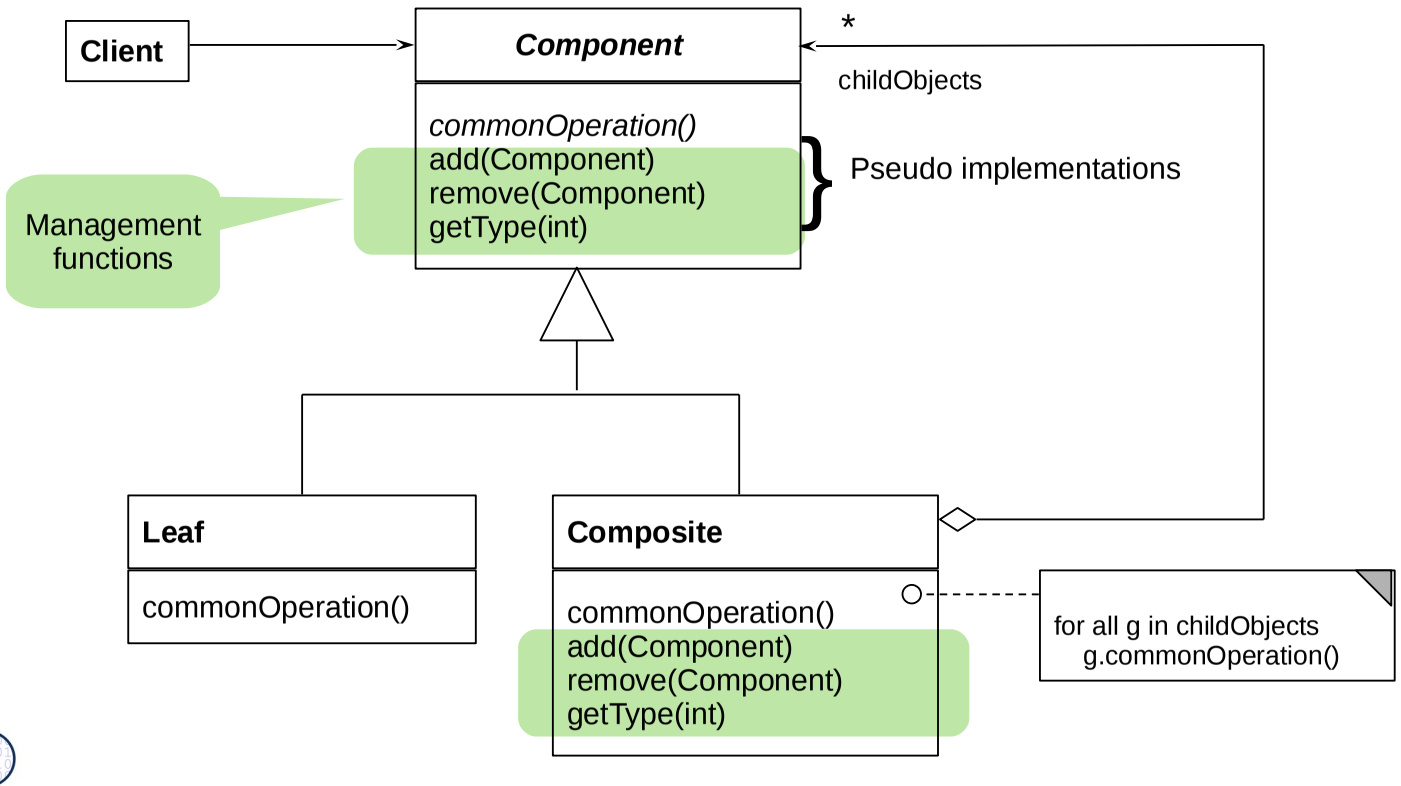
- Composite represents trees
- For a client class, Compositum unifies the access to root, inner nodes, and leaves
- In MVC, views can be organized as Composite



Prof. U. Aßmann, Softwaretechnologie, TU Dresden

Structure Composite (Rpt.)

- Composite has an recursive n-aggregation to the superclass

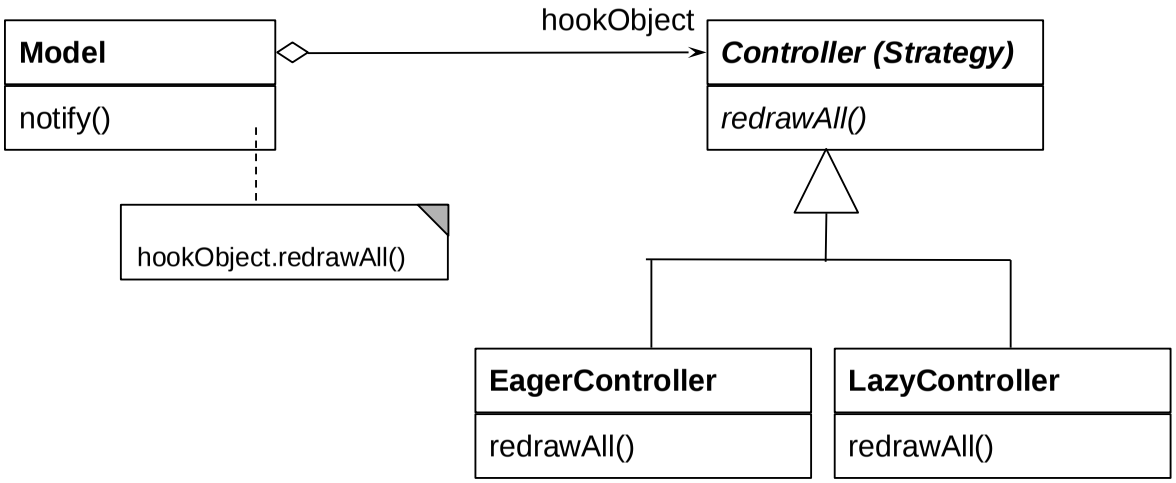


Prof. U. Aßmann, Softwaretechnologie, TU Dresden

Pattern 3: Strategy

The relation between *controller* and *view* is a *Strategy* pattern.

- There may be different control strategies
  - Lazy or eager update of views
  - Menu or keyboard input
- A view may select subclasses of *Controller,* even dynamically
- No other class changes



Different Forms of Patterns

- **Conceptual Patterns** of good system structures
  - Desktop pattern, Wastebasket pattern, Tool and Material pattern, ...
- Specific **Design Patterns** for good design structures
  - **Product Line Patterns**
  - **Architectural styles** describe course-grain styles for applications
  - **Antipatterns ("bad smells")** are defective patterns
- **Implementation Patterns** (programming patterns, idioms, workarounds) replace missing language constructs
- **Process Patterns** describe good structures in software development processes
- **Reengineering Patterns** describe good practices in reengineering
- **Organizational Patterns** describe good patterns in company structuring

A **pattern** is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts

[Riehle/Zülinghoven, Understanding and Using Patterns in Software Development]

Purposes of Design Patterns

- Improve communication in teams
  - Between clients and programmers
  - Between designers, implementers and testers
  - For designers, to understand good design concepts
- Design patterns create a glossary for software engineering (an “ontology of software design”)
  - A “software engineer” without the knowledge of patterns is a programmer
- Design patterns document abstract design concepts
  - Patterns are “mini-frameworks”
  - Documentation: in particular frameworks are documented by design patterns
  - Prevent re-invention of well-known solutions
  - Design patterns capture information in reverse engineering
  - Improve code structure and hence, code quality

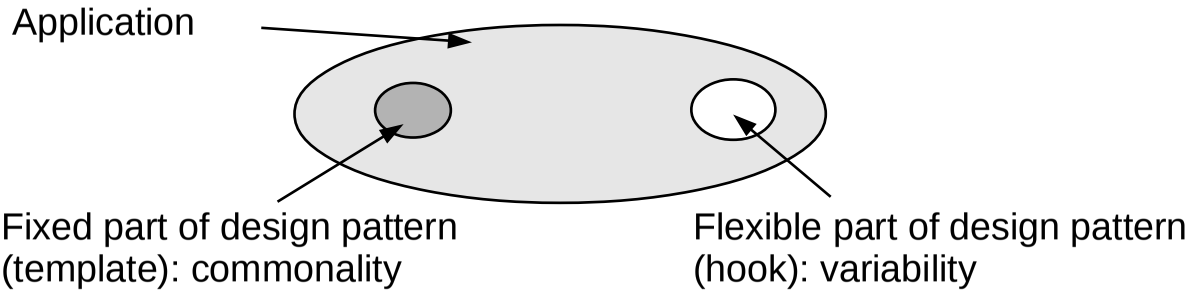
Standard Problems to Be Solved By *Product Line Patterns*

- Variability
  - Exchanging parts easily
  - Variation, variability, complex parameterization
  - Static and dynamic
  - For product lines, framework-based development
- Extensibility
  - Software must change
- Glue (adaptation overcoming architectural mismatches)
  - Coupling software that was not built for each other

25.1) Patterns for Variability

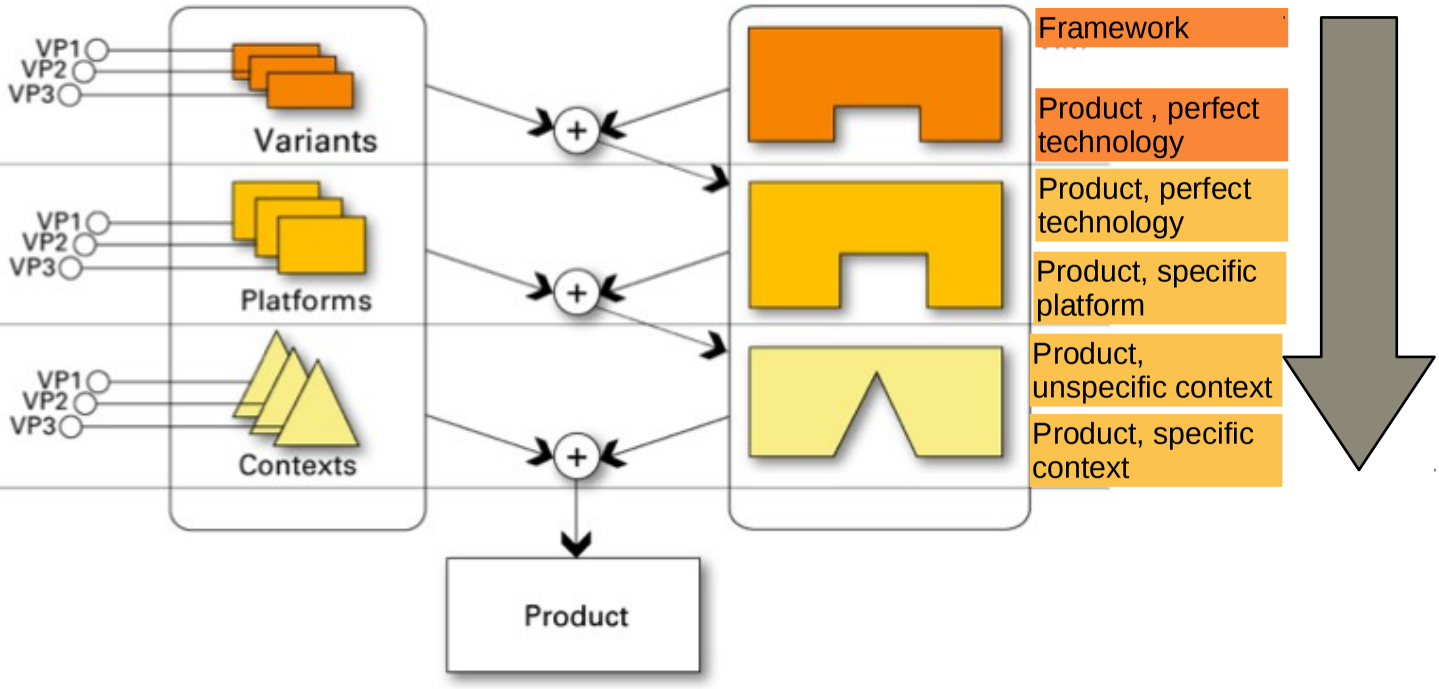
Commonalities and Variabilities

- Design patterns often center around
  - Things that are common to several applications
  - Things that are different from application to application
- Commonalities lead to *frameworks* or *product lines*
- Variabilities to *products* of a product line
- For the communality/variability knowledge, Pree invented a *template-and-hook (T&H) concept*
  - *Templates* contain skeleton code (commonality), common for the entire product line
  - *Hooks (hot spots)* are placeholders for the instance-specific code (variability)



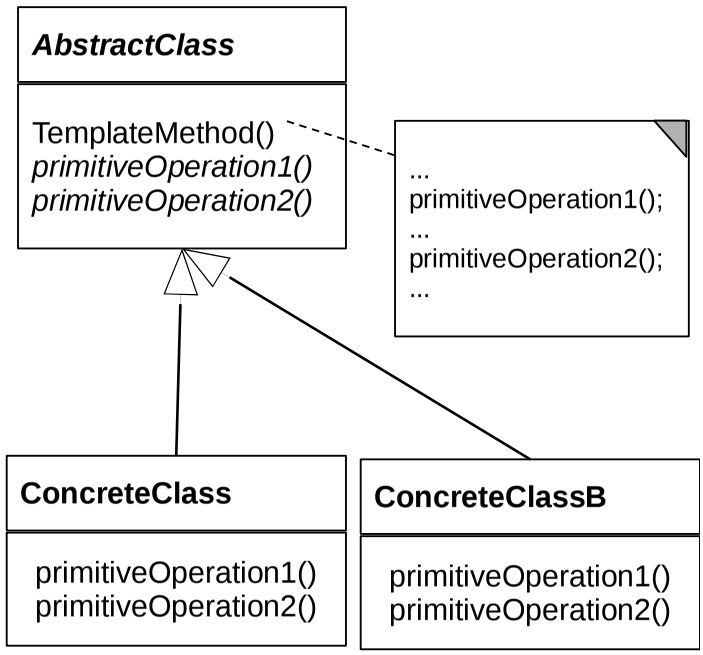
Why Do We Need Variability?

- Functional features
  - Payed vs free use
- Platforms
- Dynamic contexts (personalization, time and location)



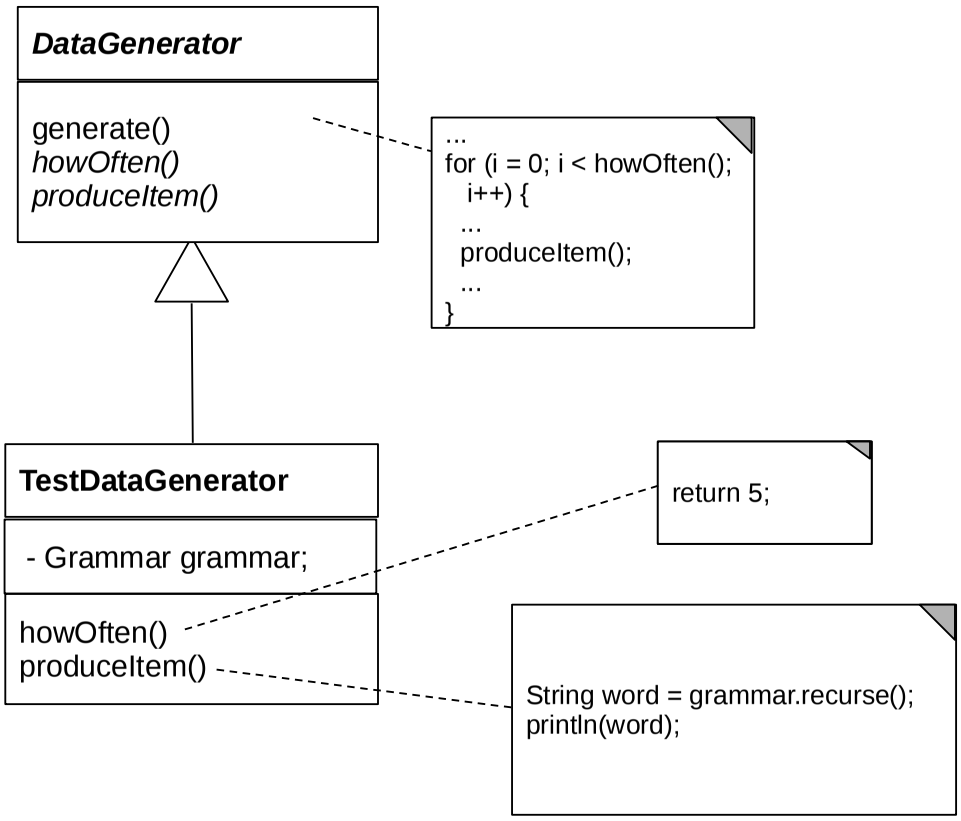
25.1.1 TemplateMethod Pattern (Wdh.)

- Define the skeleton of an algorithm (template method)
  - The template method is concrete
- Delegate parts to abstract *hook methods* that are filled by subclasses
- Implements template and hook with the same class, but different methods
- Allows for varying behavior
  - Separate invariant from variant parts of an algorithm



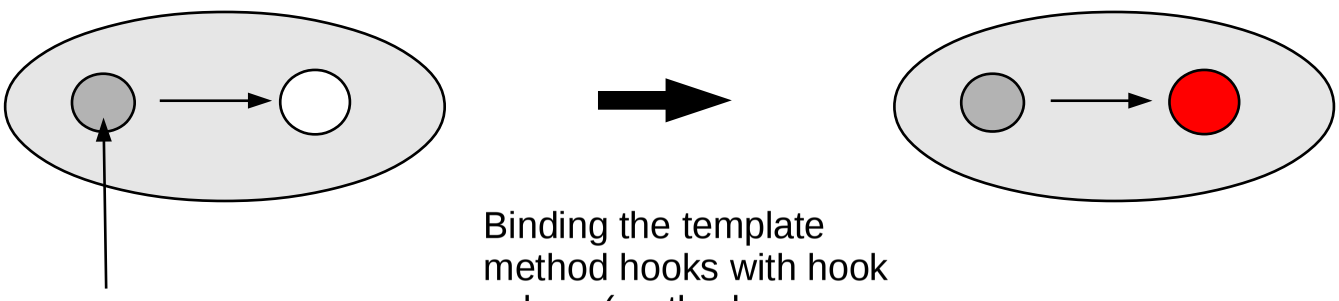
Example: A Data Generator

- Parameterizing a data generator by frequency and kind of production



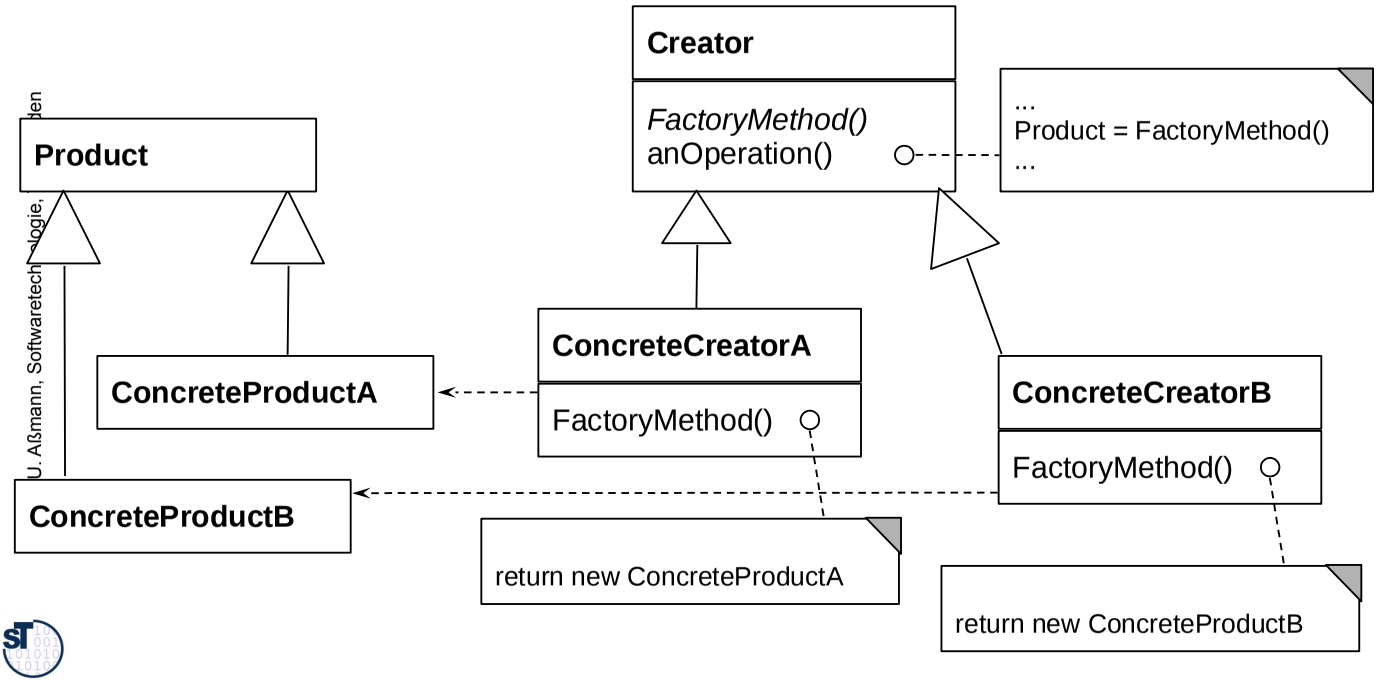
Variability with TemplateMethod

- Binding the hook means to
  - Derive a concrete subclass from the abstract superclass, providing the implementation of the hook method
- Controlled extension by only allowing for binding hook methods, but not overriding template methods



25.1.2 FactoryMethod

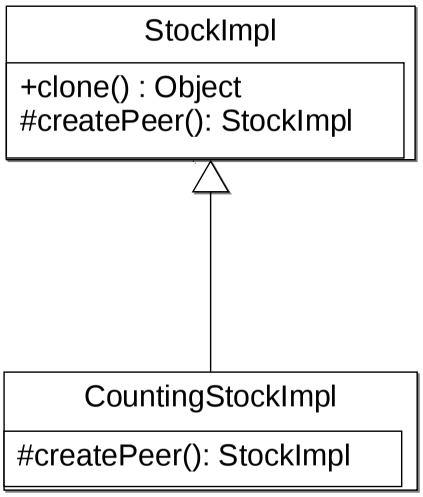
- FactoryMethod is a variant of TemplateMethod
- A FactoryMethod is a polymorphic constructor



Prof. U. Aßmann, Softwaretechnologie, TU Dresden

Factory Method im SalesPoint-Rahmenwerk

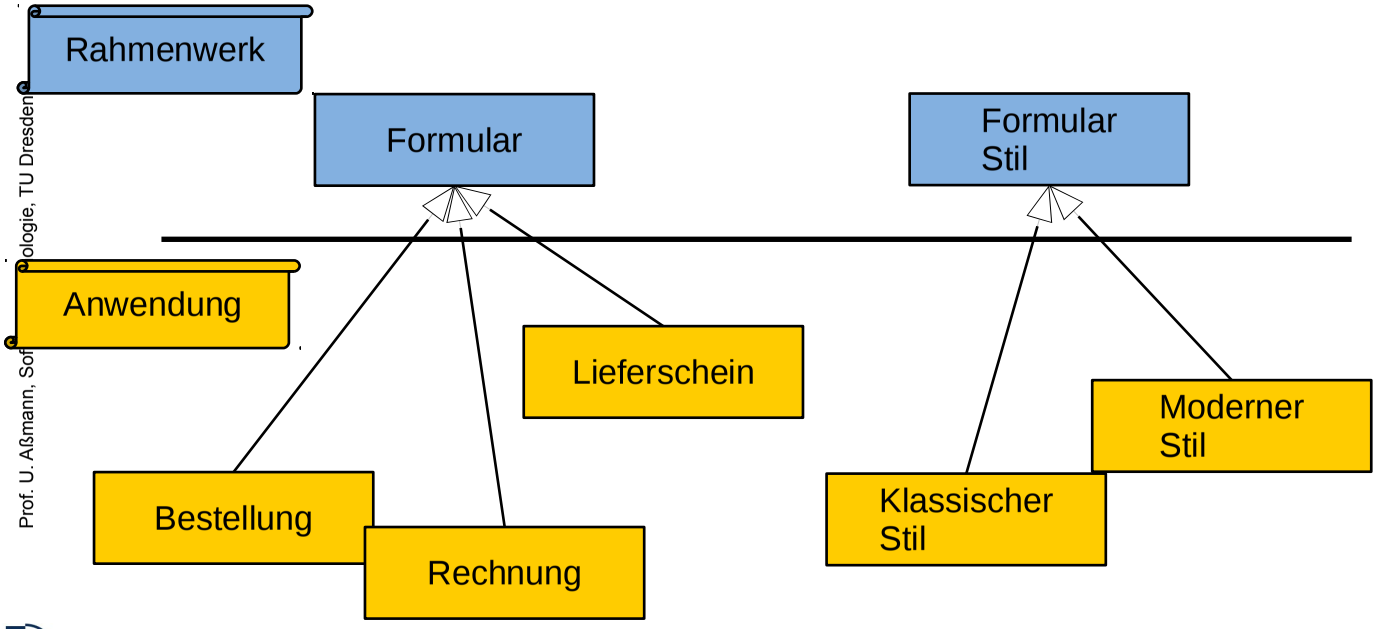
- Anwender von SalesPoint verfeinern die StockImpl-Klasse, die ein Produkt des Warenhauses im Lager repräsentiert
  - z.B. mit einem CountingStockImpl, der weiß, wieviele Produkte noch da sind



Prof. U. Aßmann, Softwaretechnologie, TU Dresden

Einsatz in Komponentenarchitekturen

- In Rahmenwerk-Architekturen wird die Fabrikmethode eingesetzt, um von oberen Schichten (Anwendungsschichten) aus die Rahmenwerkschicht zu konfigurieren:

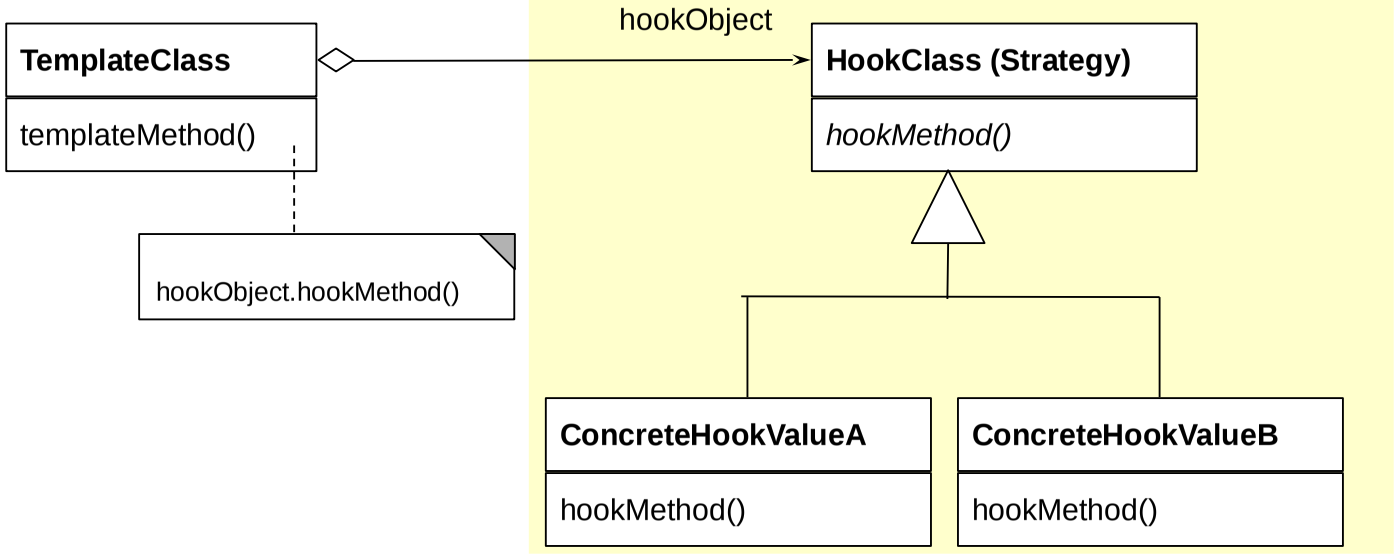


Prof. U. Aßmann, Softwaretechnologie, TU Dresden

25.1.3 Strategy (Template Class)

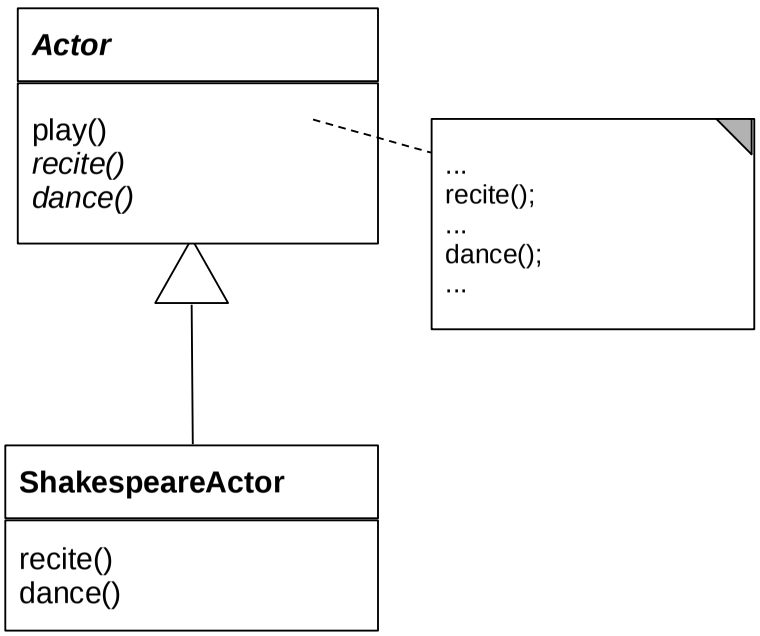
Strategy (also called Template Class)

- The template method and the hook method are found in different classes
- Similar to TemplateMethod, but
  - Hook objects and their hook methods can be exchanged at run time
  - Exchanging several methods (a set of methods) at the same time
  - Consistent exchange of several parts of an algorithm, not only one method
- This pattern is basis of Bridge, Builder, Command, Iterator, Observer, Visitor.



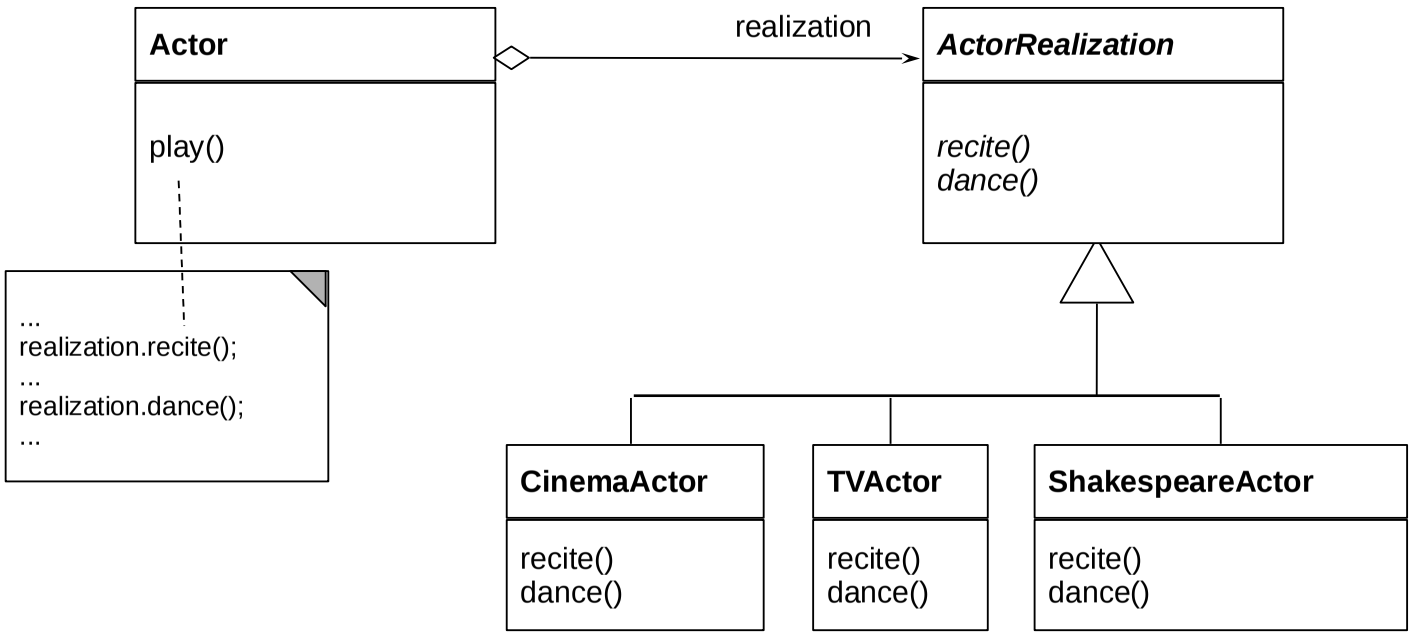
Actors and Genres as Template Method

- Binding an Actor's hook to be a ShakespeareActor



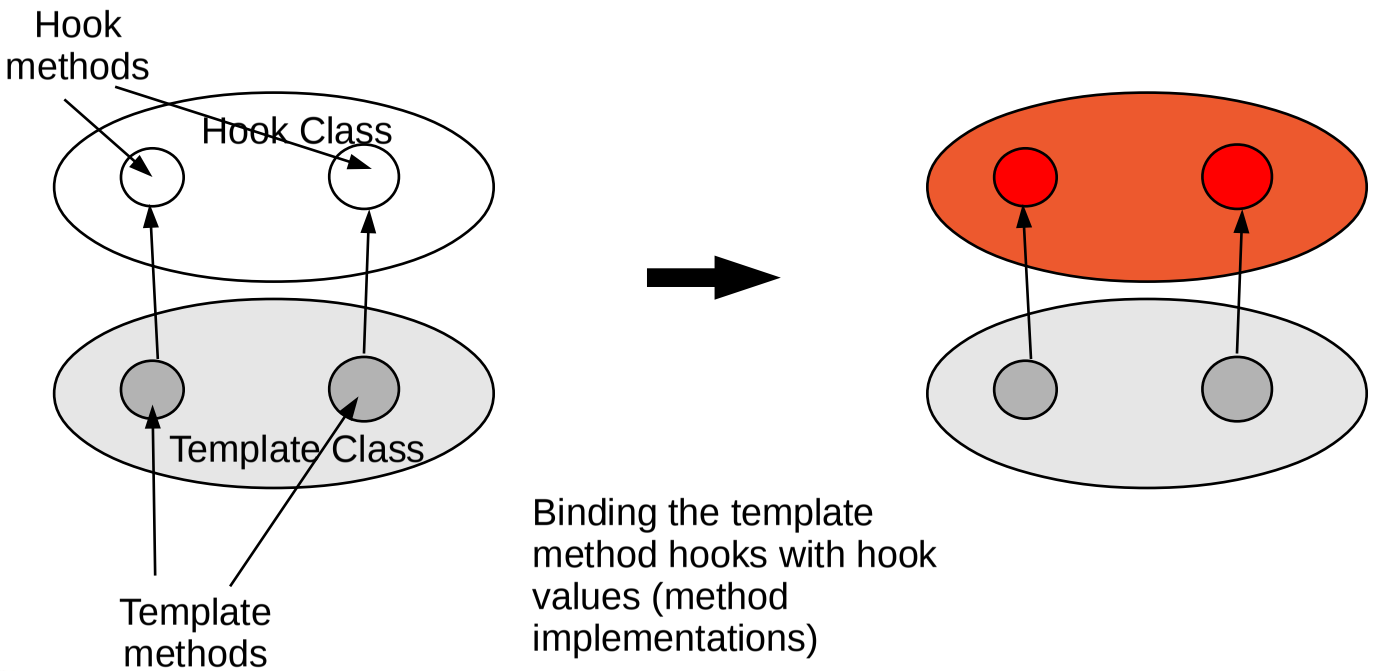
Actors and Genres as Template Class (Strategy)

- Consistent exchange of recitation and dance behavior possible



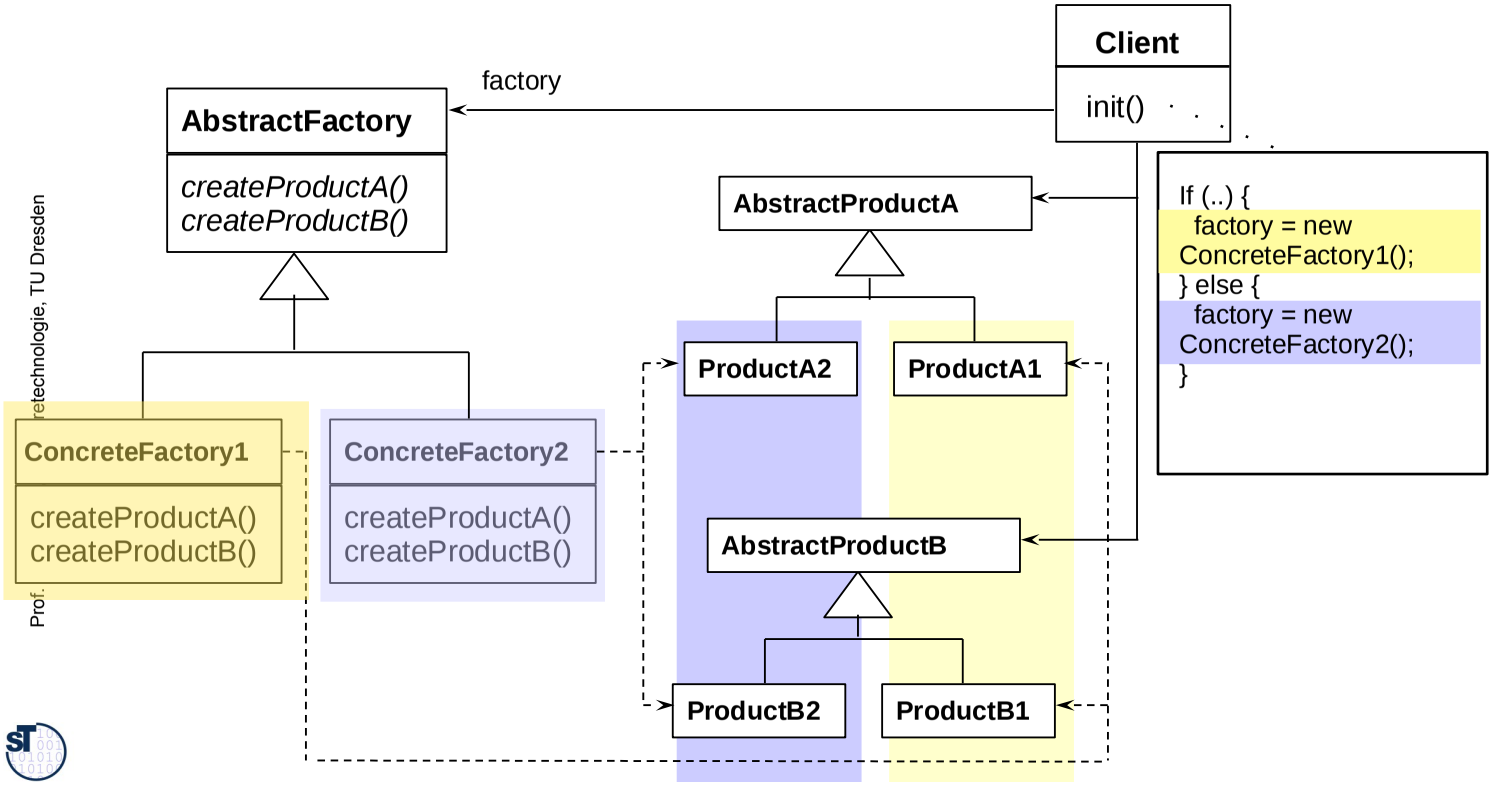
Variability with Strategy

- Binding the hook means to
  - Derive a concrete subclass from the **abstract hook superclass,** providing the implementation of the hook method



25.1.4 Factory Class (Abstract Factory)

- Allocate a family of products {A, B, ..} in different “colors” {1, 2, ..}
- Vary consistently by exchange of factory and object families

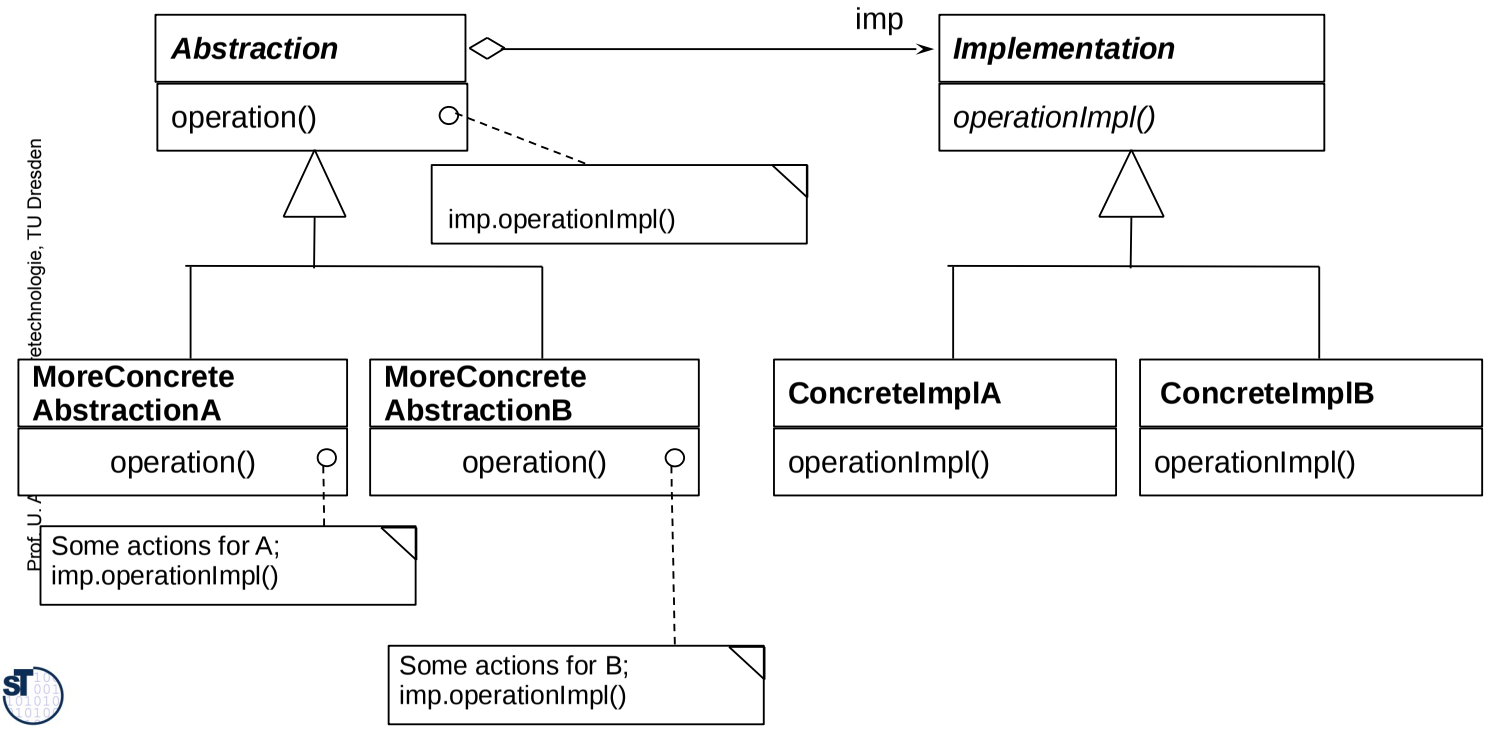


Prof. Softwaretechnologie, TU Dresden

25.1.5 Bridge (Dimensional Class Hierarchies)

Bridge, GOF-Version

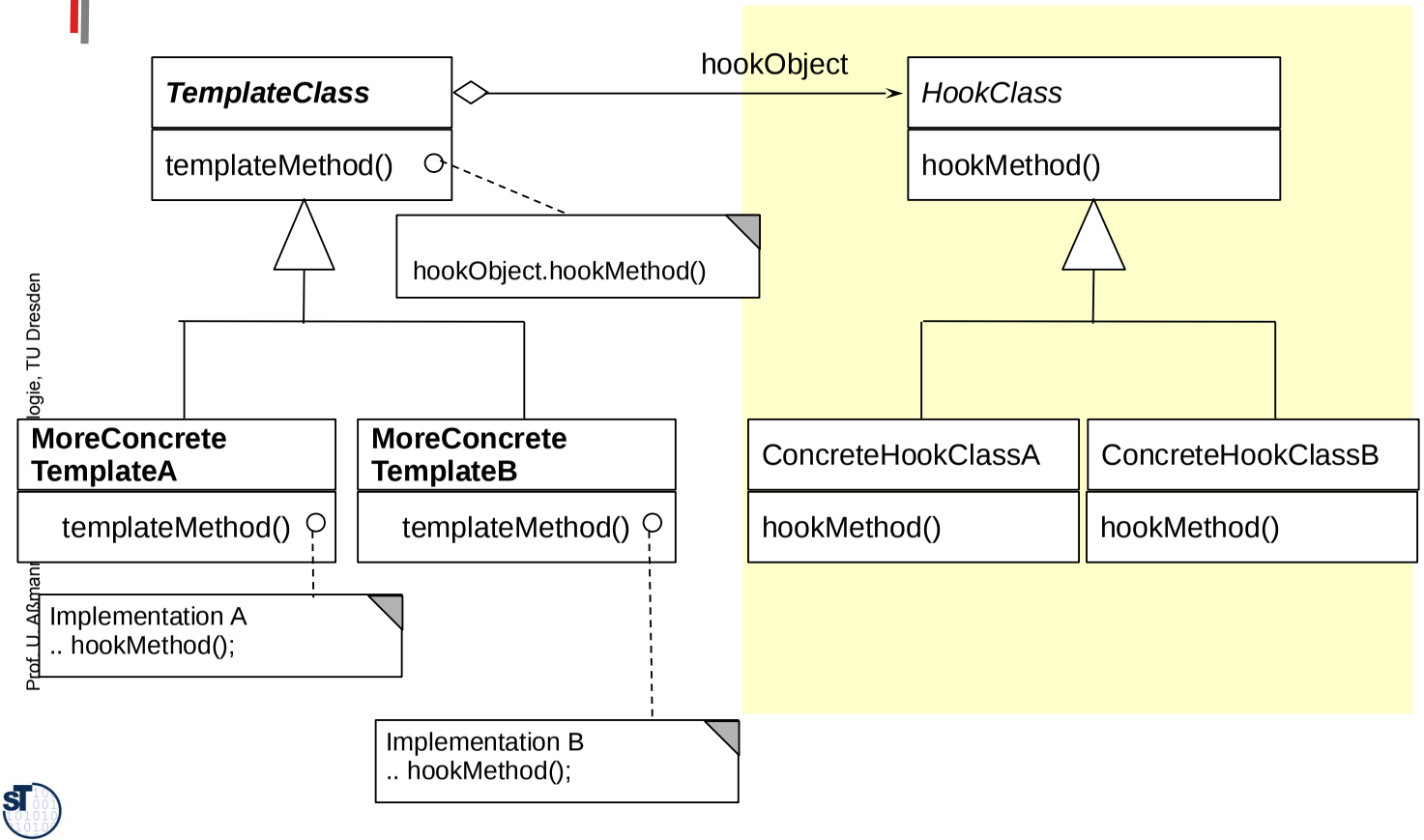
- A **Bridge** represents a complex objects with two layers
- The left hierarchy (upper layer) is called *abstraction hierarchy,* the right hierarchy (lower layer) is called *implementation*



etechnologie, TU Dresden
Prof. U. A

Bridge as Dimensional Class Hierarchies

- Bridge is an extension of TemplateClass

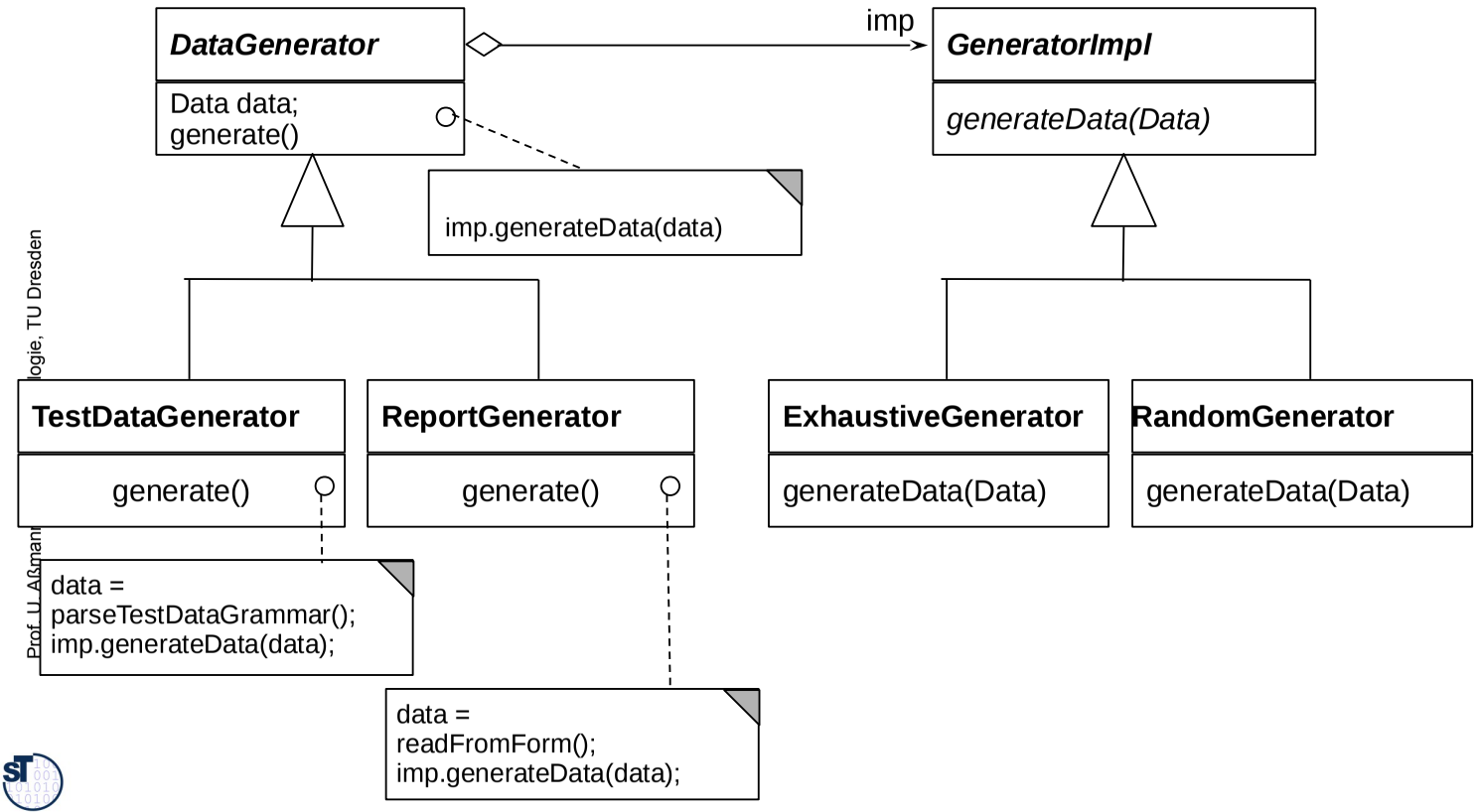


logie, TU Dresden
Prof. U. Aßmann

Bridge (Dimensional Class Hierarchies)

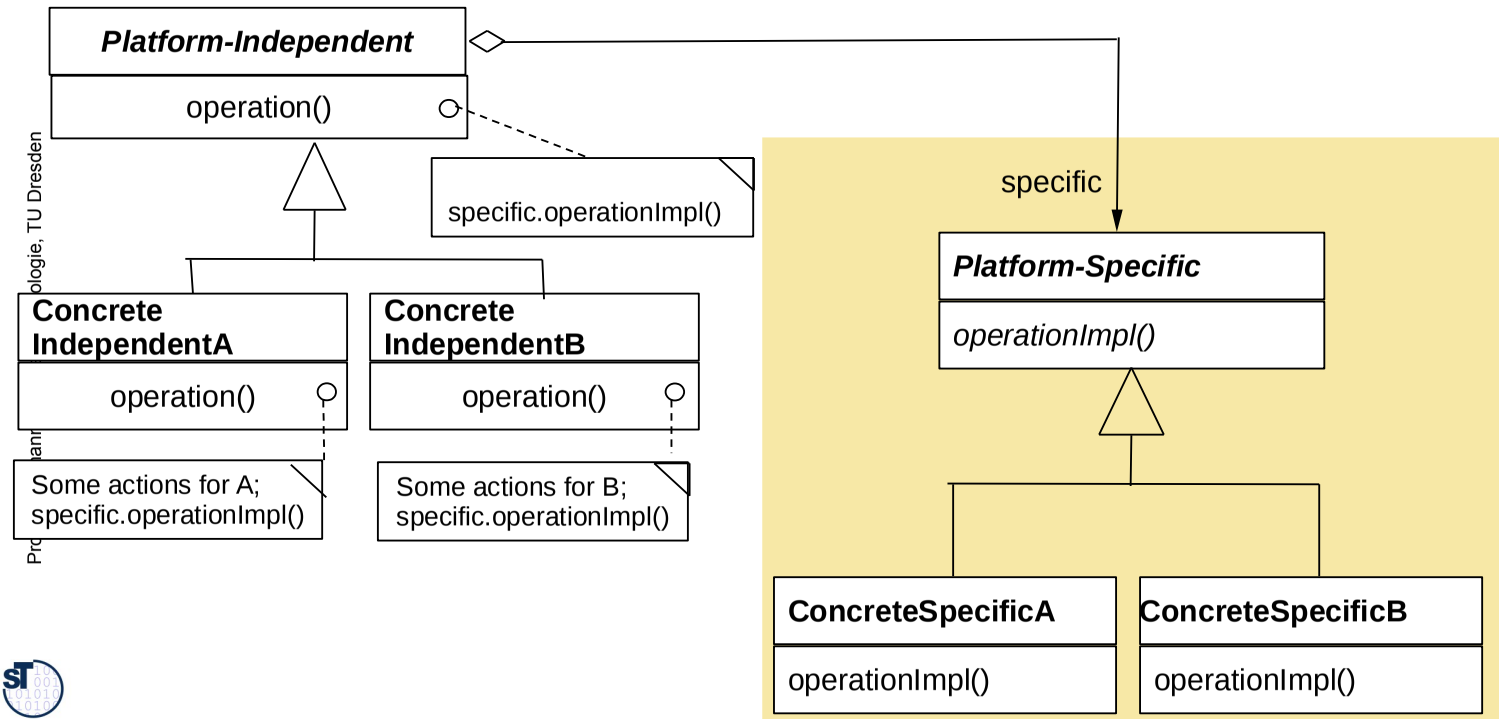
- Vary also the template class in a class hierarchy
  - The sub-template classes can adapt the template algorithm
  - Important: the sub-template classes must fulfil the *contract* of the superclass
    - Although the implementation can be changed, the interface and visible behavior must be the same
- Both hierarchies can be varied independently
  - Factoring (orthogonalization)
  - Reuse is increased
- Basis for patterns
  - Observer, Visitor

Ex. Complex Object *DataGenerator* as Bridge



Use of Bridge for Platform-Independent Coding

- Bridge can be used to implement an object with *platform-independent (left/upper hierarchy)* and platform-specific part (lower/right hierarchy)
- For every type of platform, there must be one Bridge

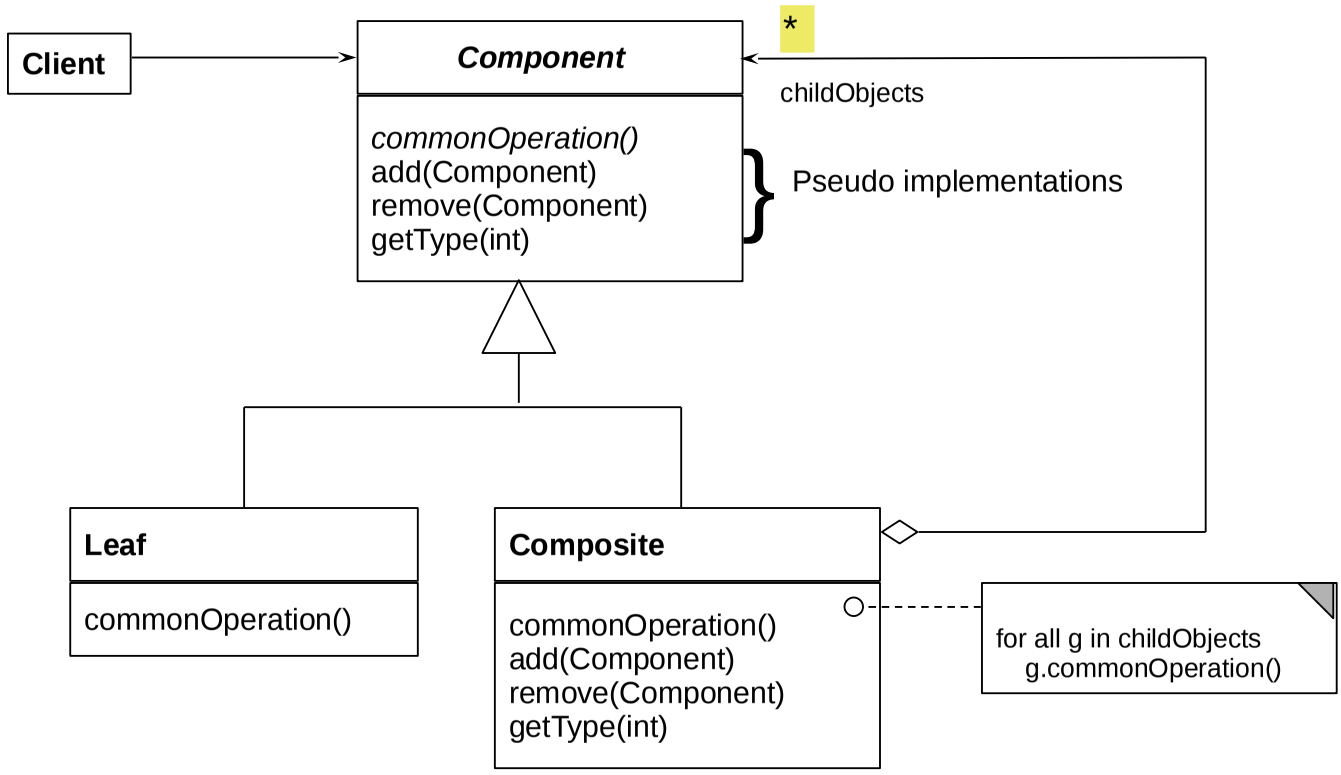


25.2) Patterns for Extensibility

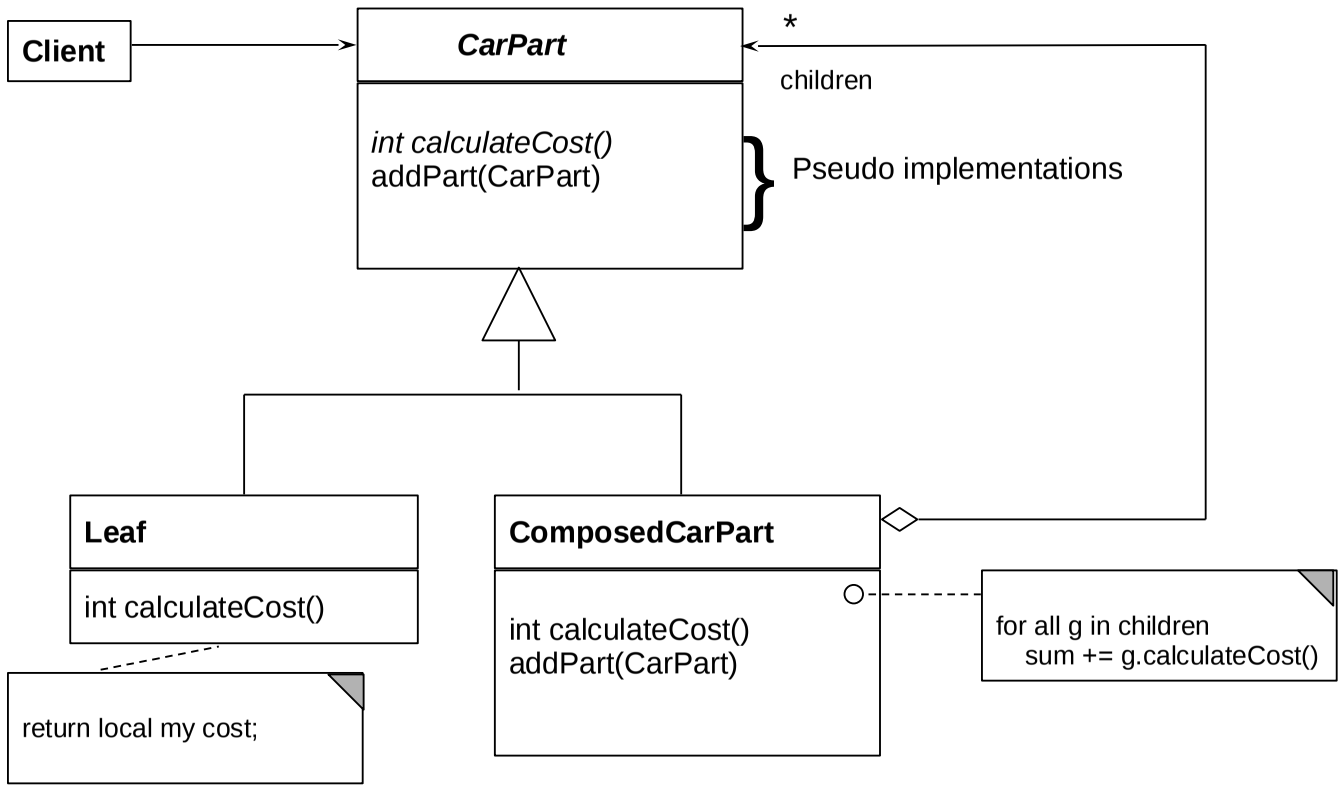
Extensibility patterns describe how to build plug-ins (complements, extensions) to frameworks

25.2.1 Structure Composite (Wdh.)

- Composite has an recursive n-aggregation to the superclass



Example: PieceLists in Cars (Wdh.)



Piece Lists in Production Data (Wdh.)

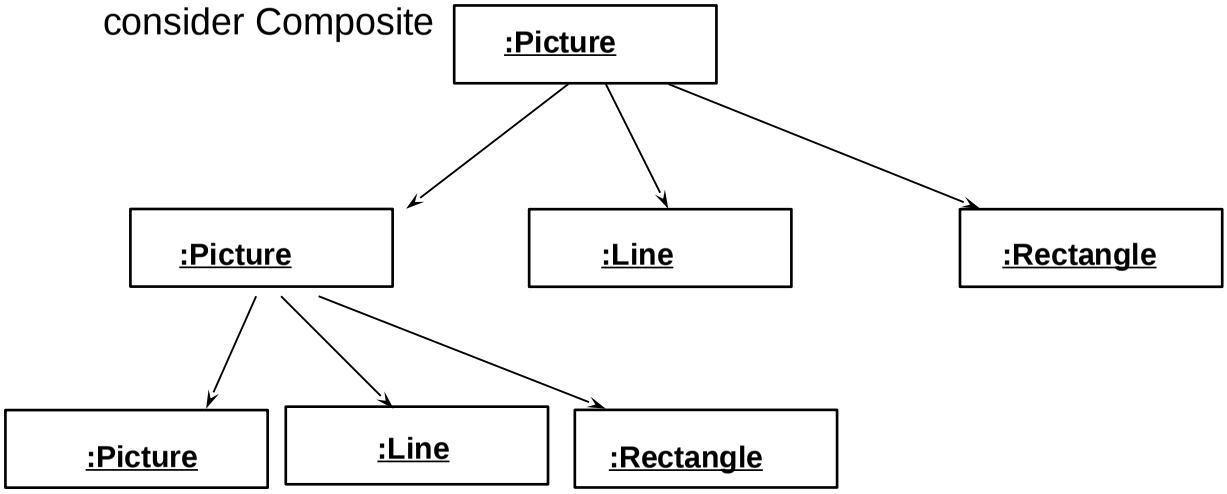

```
abstract class CarPart {
   int myCost;
   abstract int calculateCost();
}
class ComposedCarPart extends CarPart {
   int myCost = 5;
   CarPart [] children;  // here is the n-recursion
   int calculateCost() {
     for (i = 0; i <= children.length; i++) {
       curCost += children[i].calculateCost();
     }
     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();
```

Iterator algorithms (map)
Folding algorithm (folding a tree with a scalar function)

Composite (Wdh.)

- Part/Whole hierarchies, e.g., nested graphic objects
  - Attention: class diagram cannot convey the constraint that the objects form a tree!
- Dynamic Extensibility of Composite
  - Due to the n-recursion, new children can always be added dynamically into a composite node
  - Whenever you have to program an extensible part of a framework, consider Composite

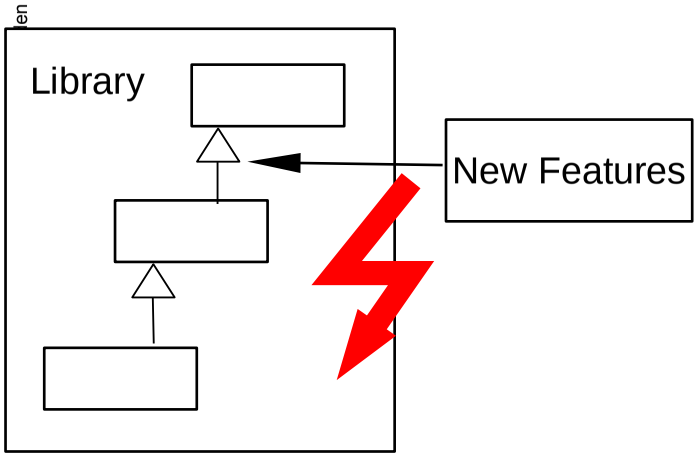


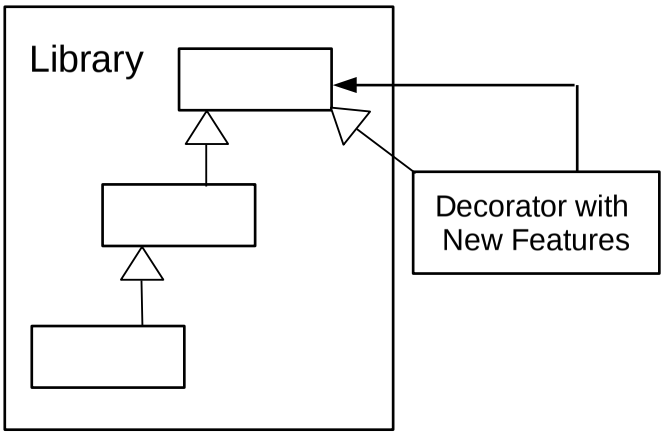
common operations: draw(), move(), delete(), scale()

25.2.2. Decorator

Problem

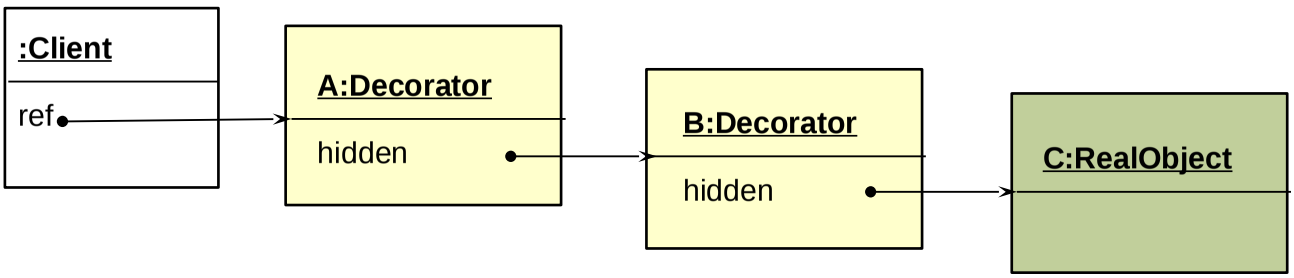
- How to extend an inheritance hierarchy of a library that was bought in binary form?
- How to avoid that an inheritance hierarchy becomes too deep?





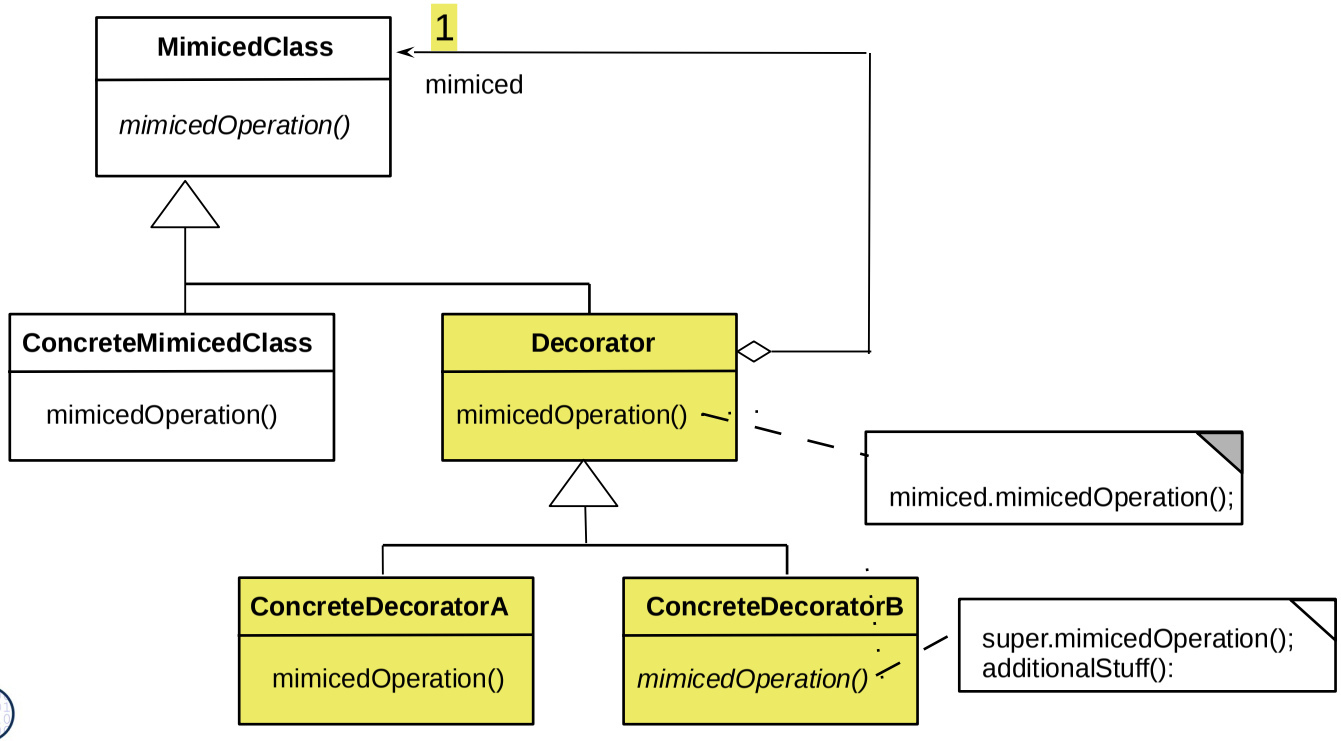
Decorator Pattern

- A Decorator object is a *skin* of another object
- The Decorator class *mimics* a class
-

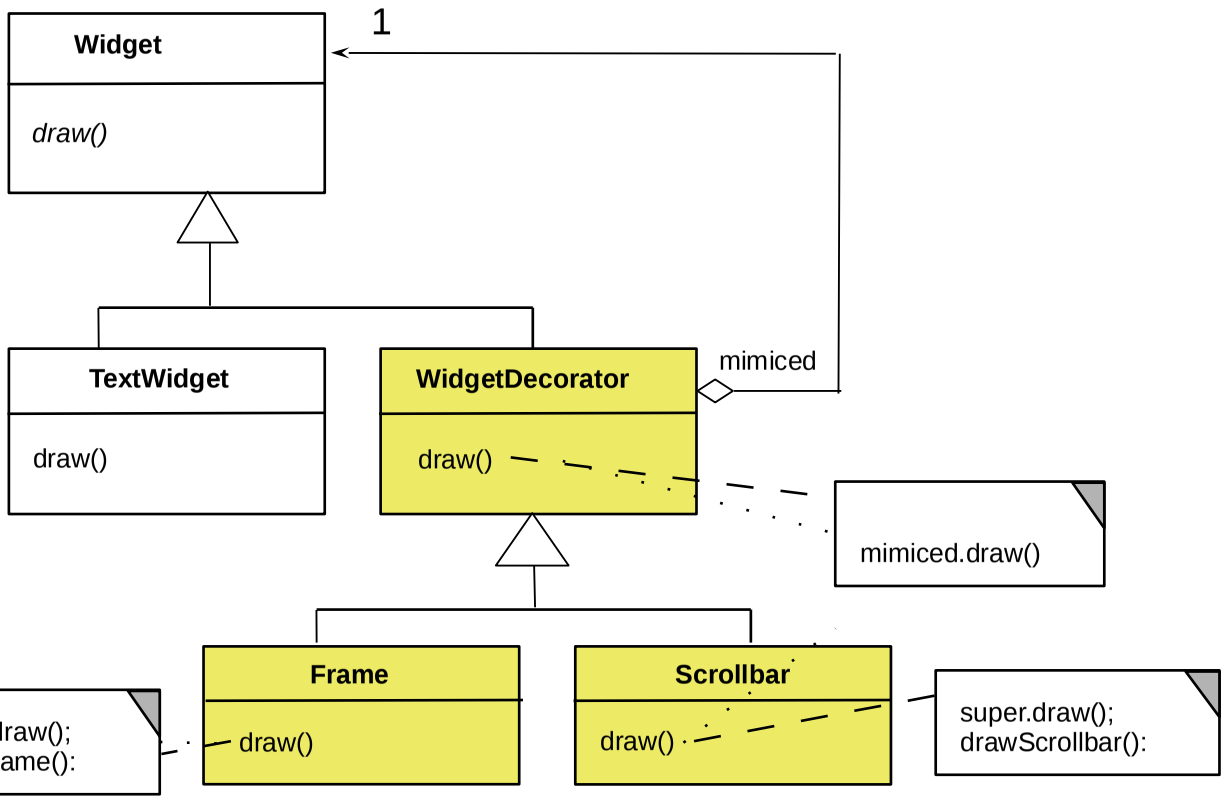


Decorator – Structure Diagram

- It is a restricted Composite with a 1-aggregation to the superclass
  - 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



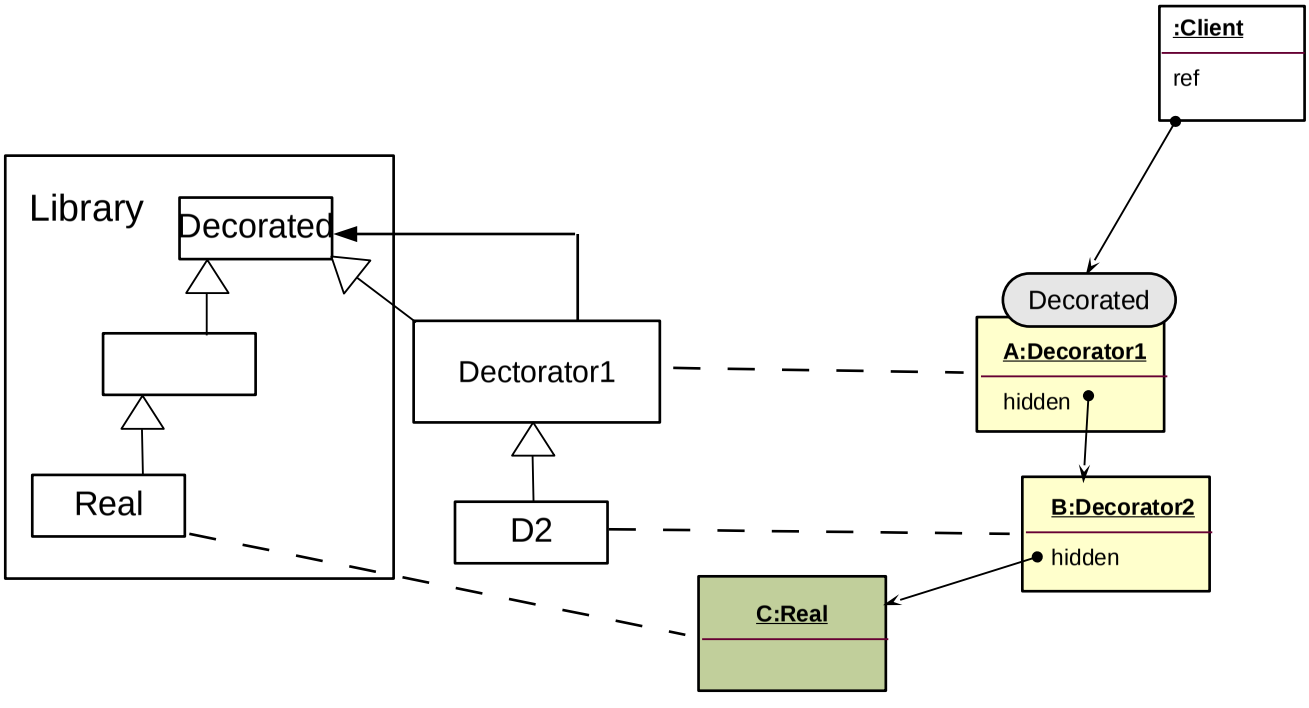
Ex.: Decorator for Widgets



Prof. U. Aßmann, Softwaretechnologie, TU Dresden

Purpose Decorator

- For dynamically extensible objects (i.e., decoratable objects)
  - Addition to the decorator chain or removal possible
- For big objects



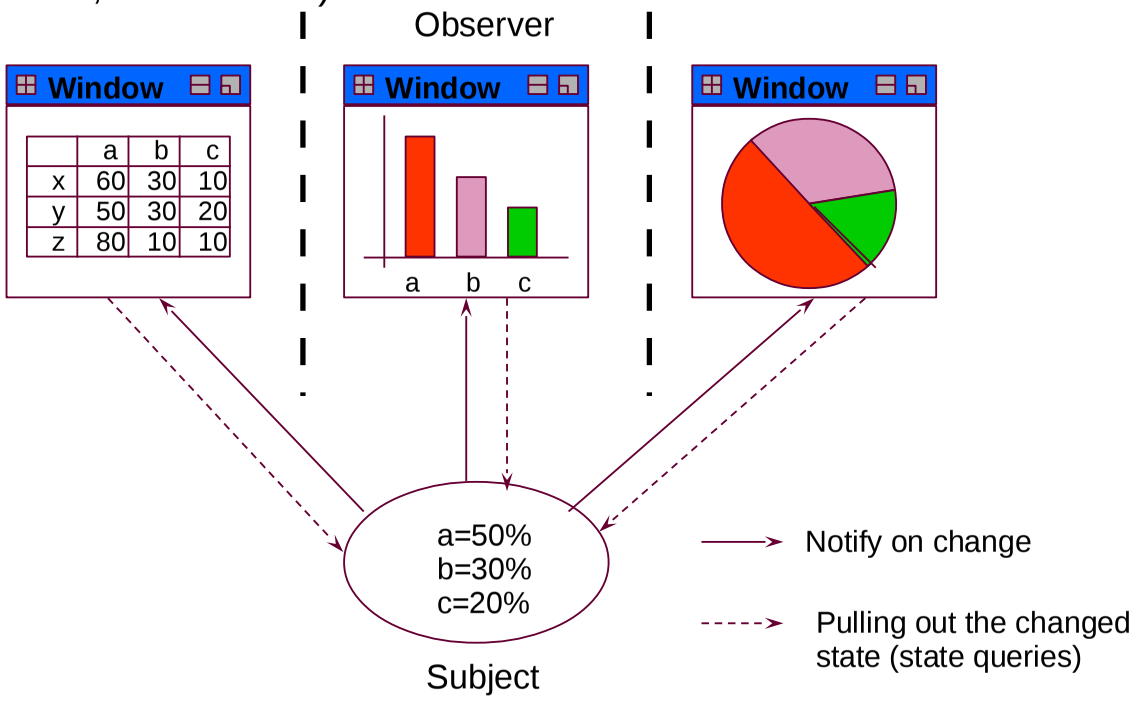
Prof. U. Aßmann, Softwaretechnologie, TU Dresden

25.2.3 Different Kinds of Publish/Subscribe Patterns – (Event Bridge)

- Publish/Subscribe patterns are for dynamic, event-based communication in synchronous or asynchronous scenarios
- Subscribe functions build up dynamic communication nets
- Callback
- Observer
- EventBus

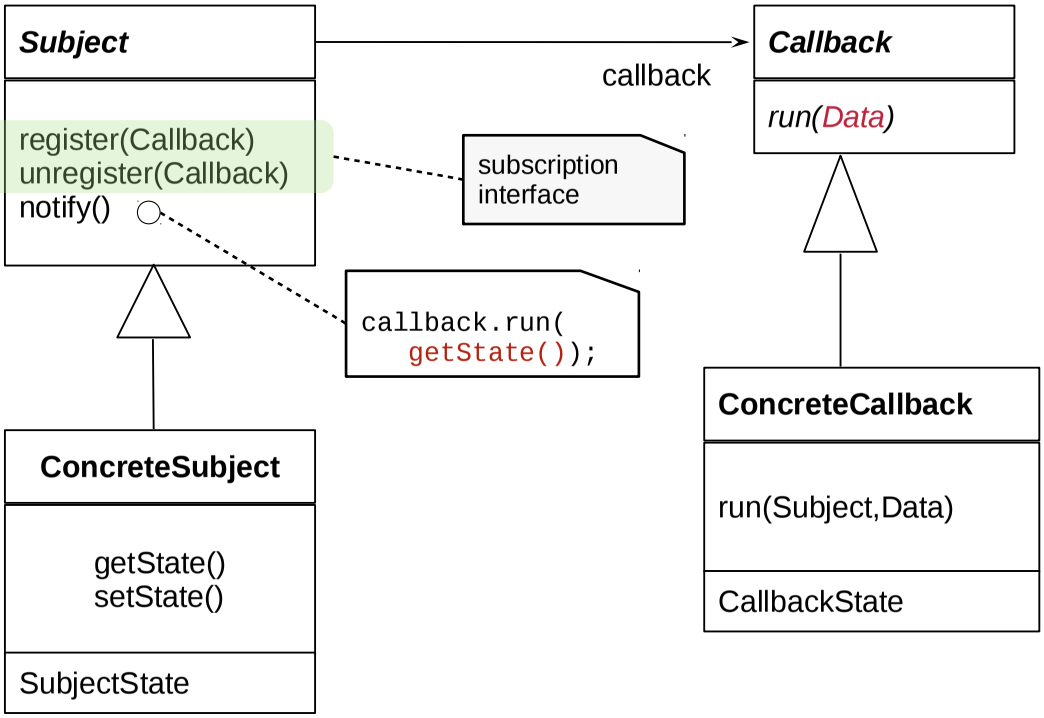
Publish/Subscribe Patterns

- Distinguish: Subscription of Observers tp Subjects // Notification of event // Source of event (subject) // Data to be transfered // Relation of Subject and Observer
- Therefore, Observer exists in several variants (push, pull, CallBack, EventBus, ChannelBus)



25.2.3.1 Publish/Subscribe with 1 Observer: Callback

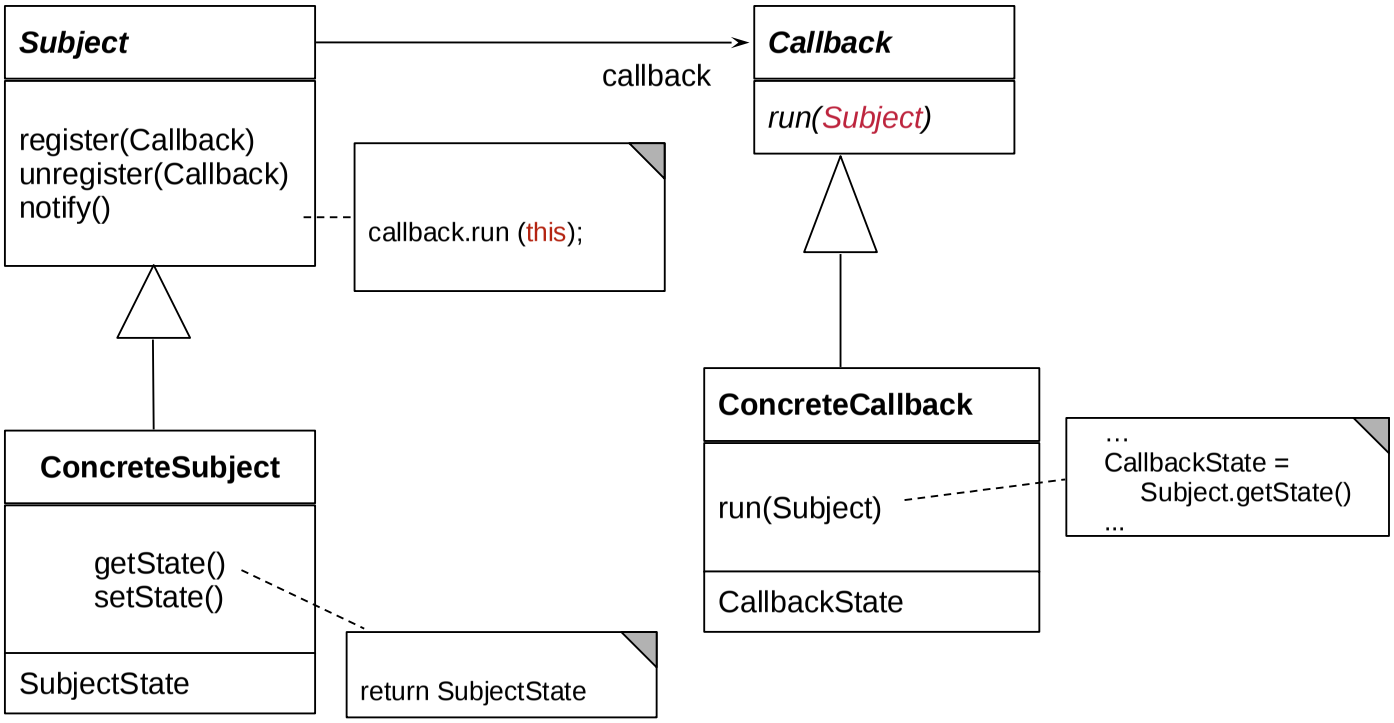
- **Callbacks** have only one observer
- A (**push-)Callback** is data-pushing



Prof. U. Aßmann, Softwaretechnologie, TU Dresden

Structure pull-Callback

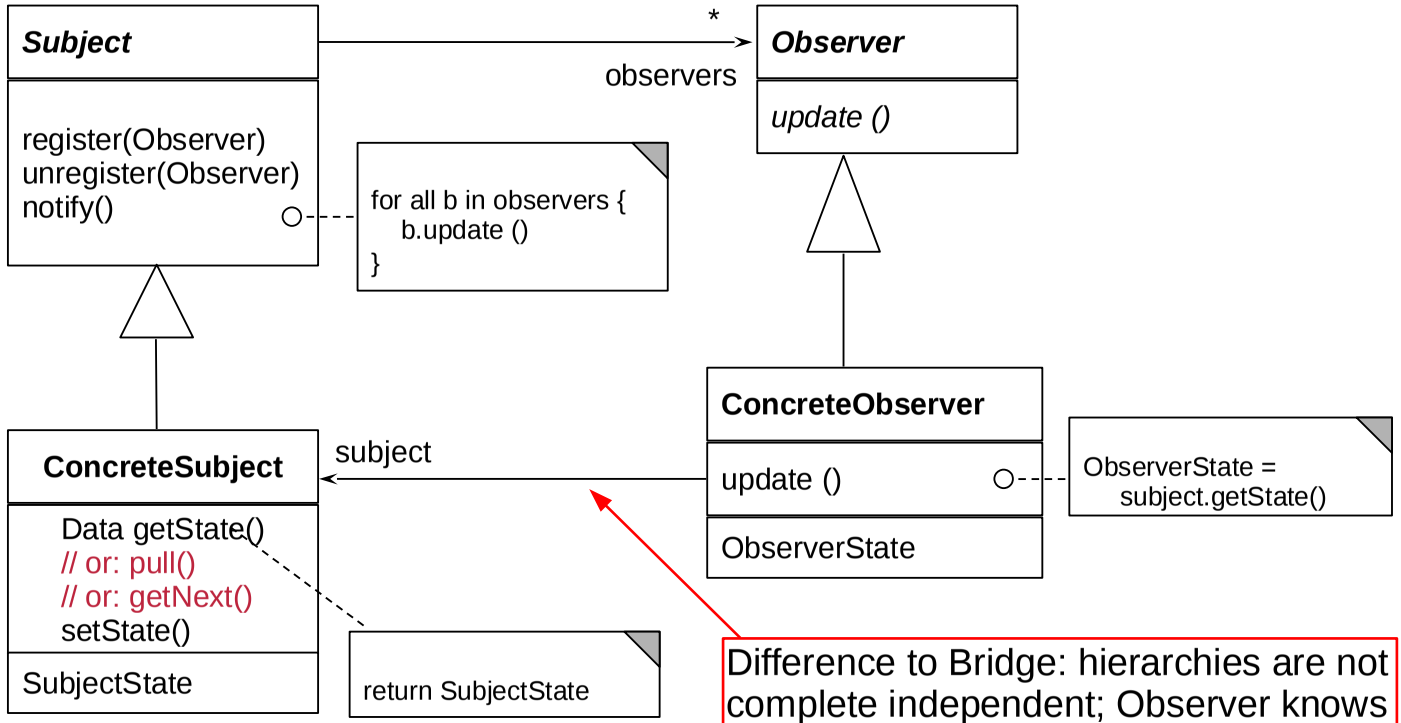
- A **pull-Callback** is Subject-pushing and data-pulling



Prof. U. Aßmann, Softwaretechnologie, TU Dresden

25.2.3.2 Pull-Observer (Publisher/Subscriber, Event Bridge) (Rpt.)

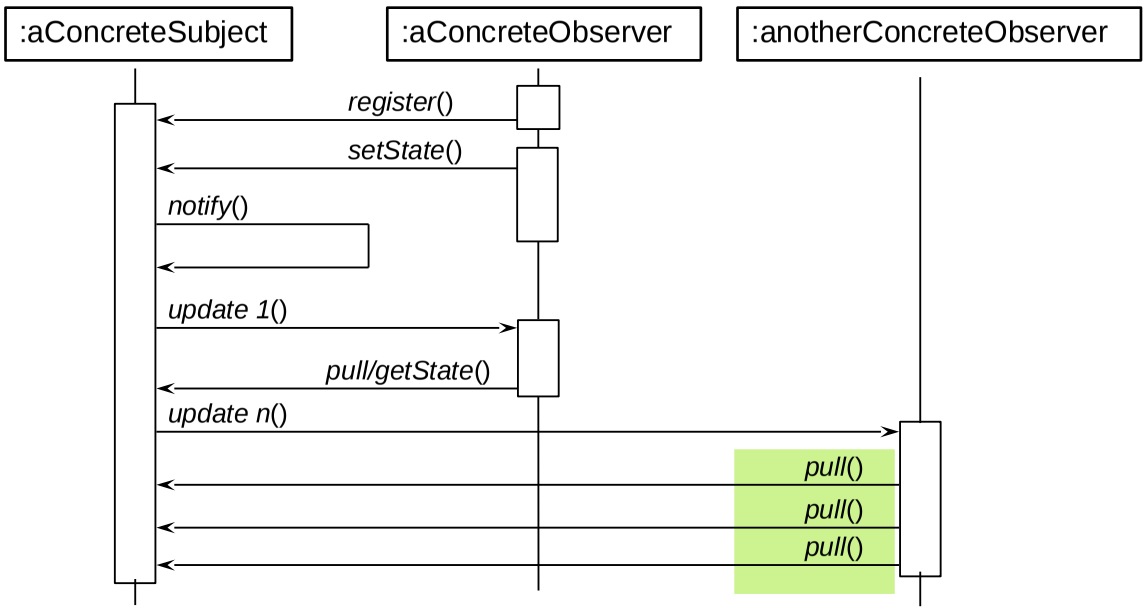
- The pull-Observer does not push anything, but pulls data later out with getState() or getNext() (same as in Iterator)
- Pulling resembles *Iterator (Stream),* if data is pulled repeatedly



Prof. U. Aßmann, Softwaretechnologie, TU Dresden

Sequence Diagram pull-Observer

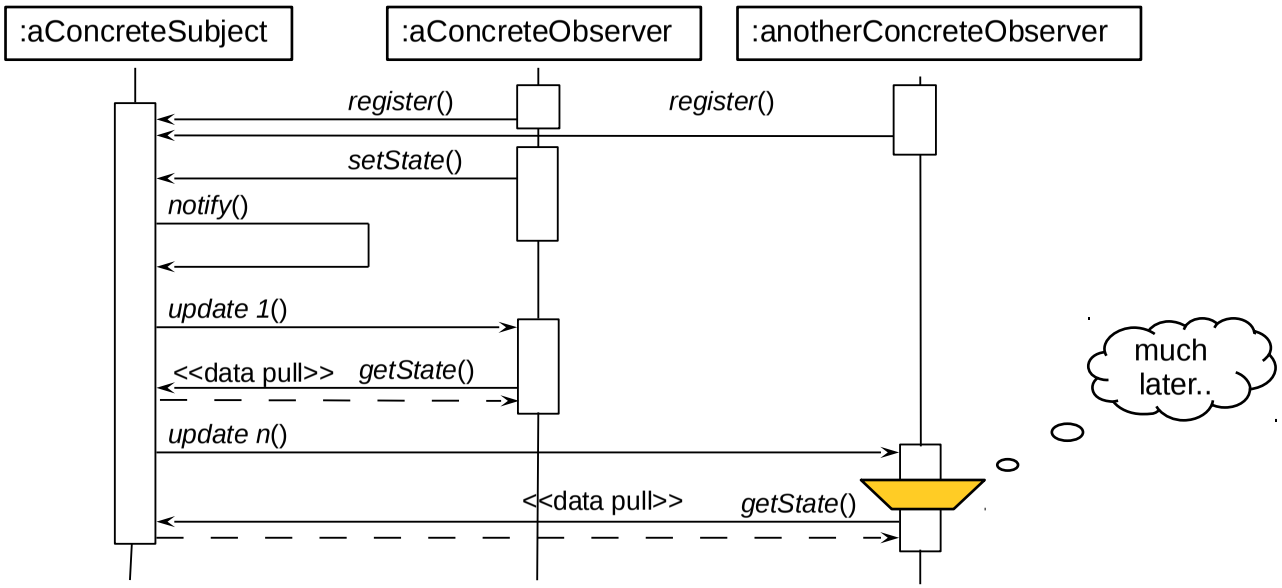
- Update() does not transfer data, only an event (anonymous communication possible)
  - Observer pulls data out itself with getState()
  - Lazy processing (on-demand processing)
- pull-Observer uses Iterator, if data is pulled iteratively



Prof. U. Aßmann, Softwaretechnologie, TU Dresden

Sequence Diagram pull-Observer

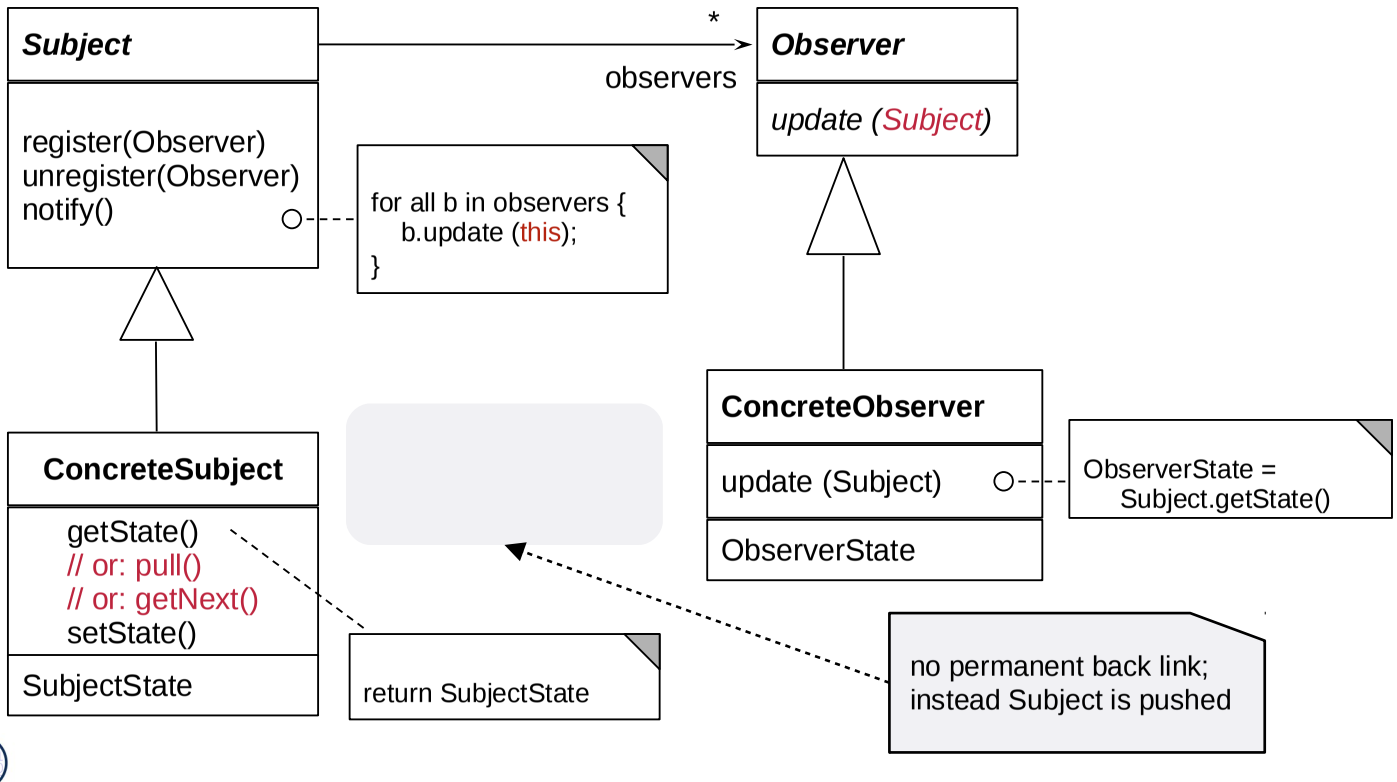
- Update() does not transfer data, only an event (anonymous communication possible)
  - Observer pulls data out itself with getState()
  - Lazy processing (on-demand processing)
- pull-Observer uses Iterator, if data is pulled iteratively



Prof. U. Aßmann, Softwaretechnologie, TU Dresden

25.2.3.3 Structure Subject-Pushing pull-Observer

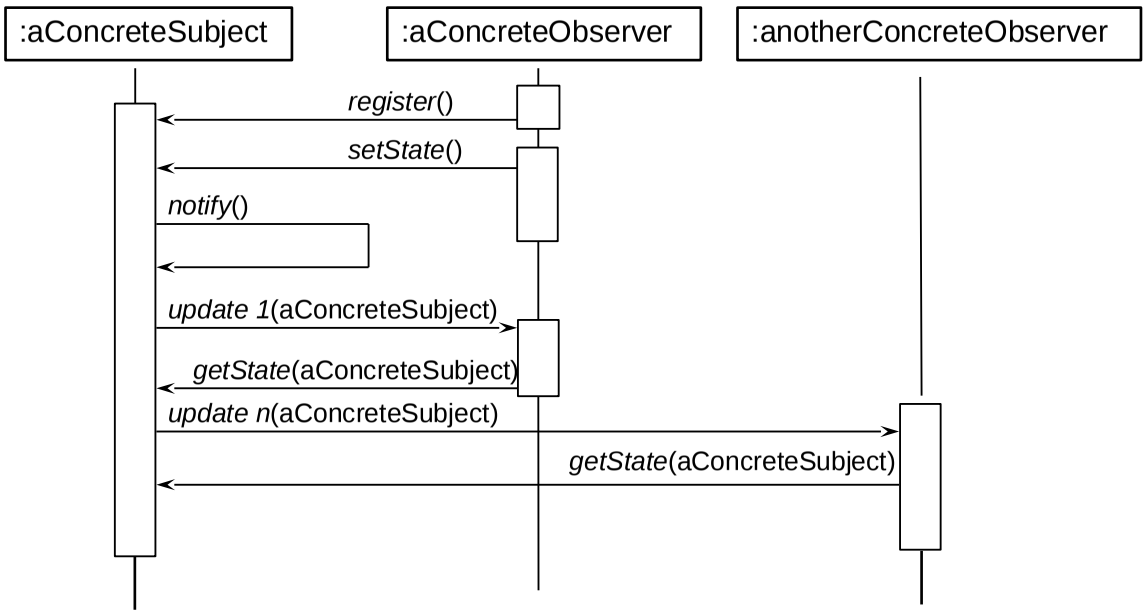
- A **Subject-pushing Observer** is a even simpler variant of the pull-Observer, which gets the subject as argument of `update()`



Prof. U. Aßmann, Softwaretechnologie, TU Dresden

Sequence Diagram Callback

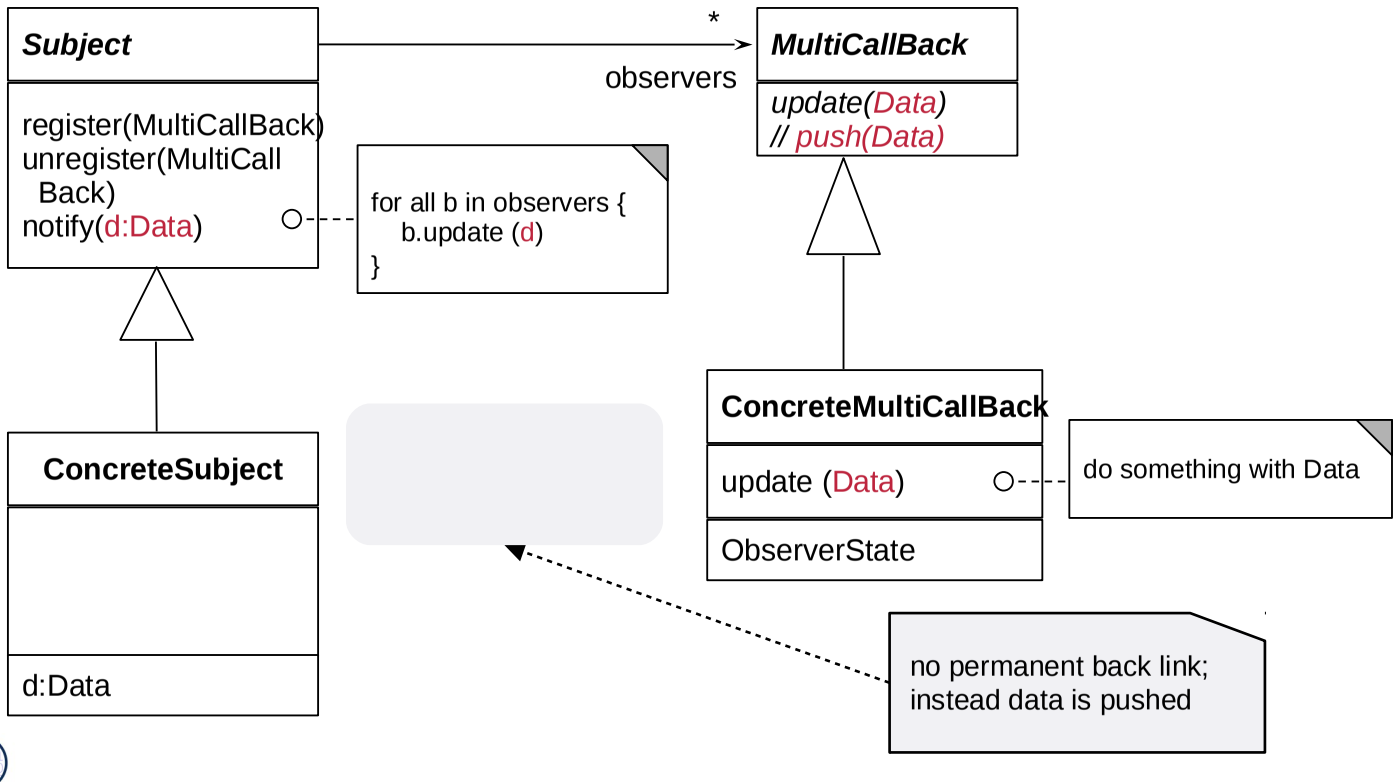
- Update() transfer Subject to Observer
  - Observer pulls data out of given Subject itself with getState(subject)



Prof. U. Aßmann, Softwaretechnologie, TU Dresden

25.2.3.4 Structure Data-Pushing-Observer

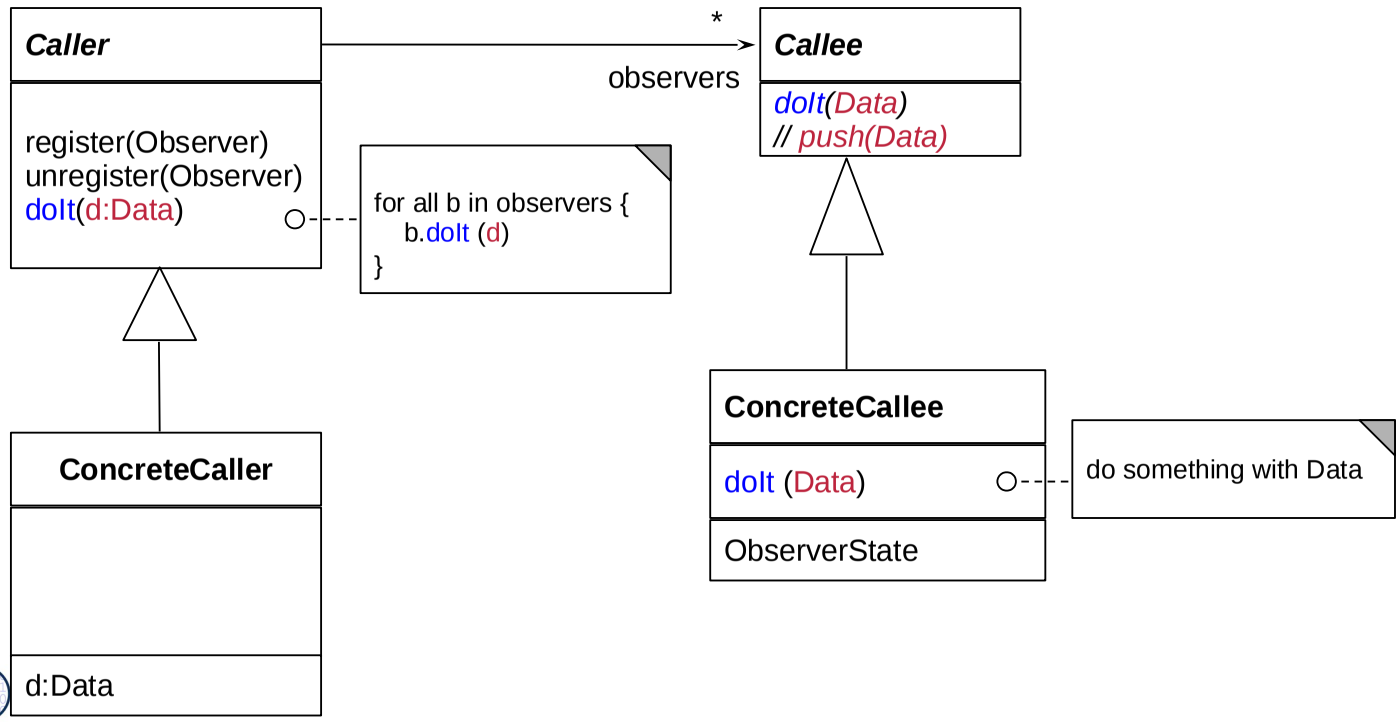
- Subject pushes data with `update(Data)`
- Pushing resembles *Sink*, if data is pushed iteratively



Prof. U. Aßmann, Softwaretechnologie, TU Dresden

25.2.3.5 Structure Multi-Call

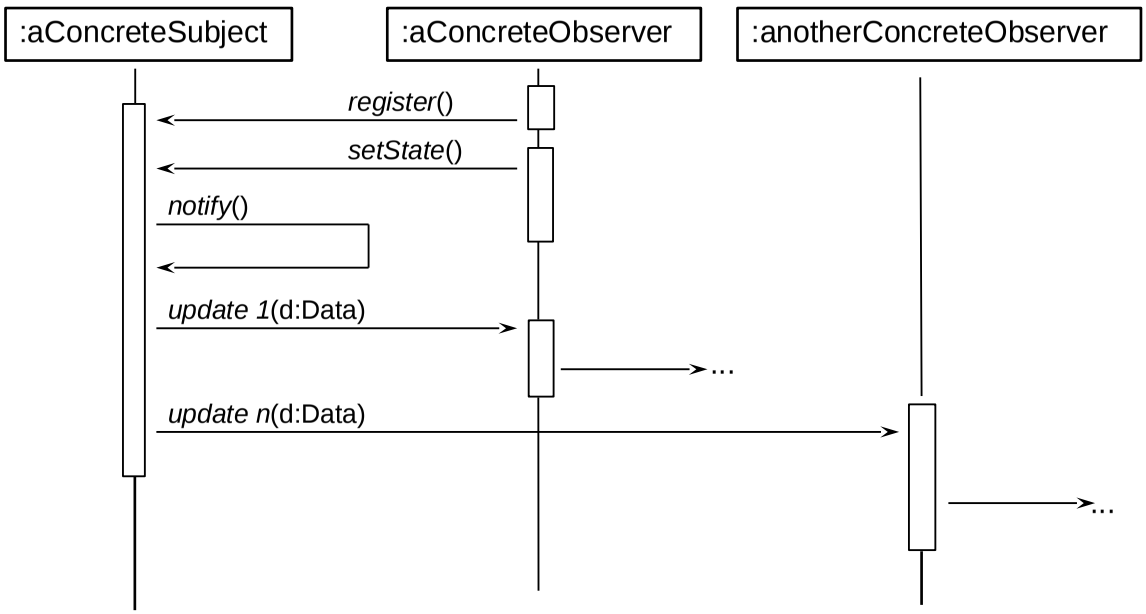
- If the methods in the Subject and the Observer are called the same, we speak of a **multi-call (extensible call)**
- At first, this looks like a normal call, but it can be extended from outside by registering new Callees



Prof. U. Aßmann, Softwaretechnologie, TU Dresden

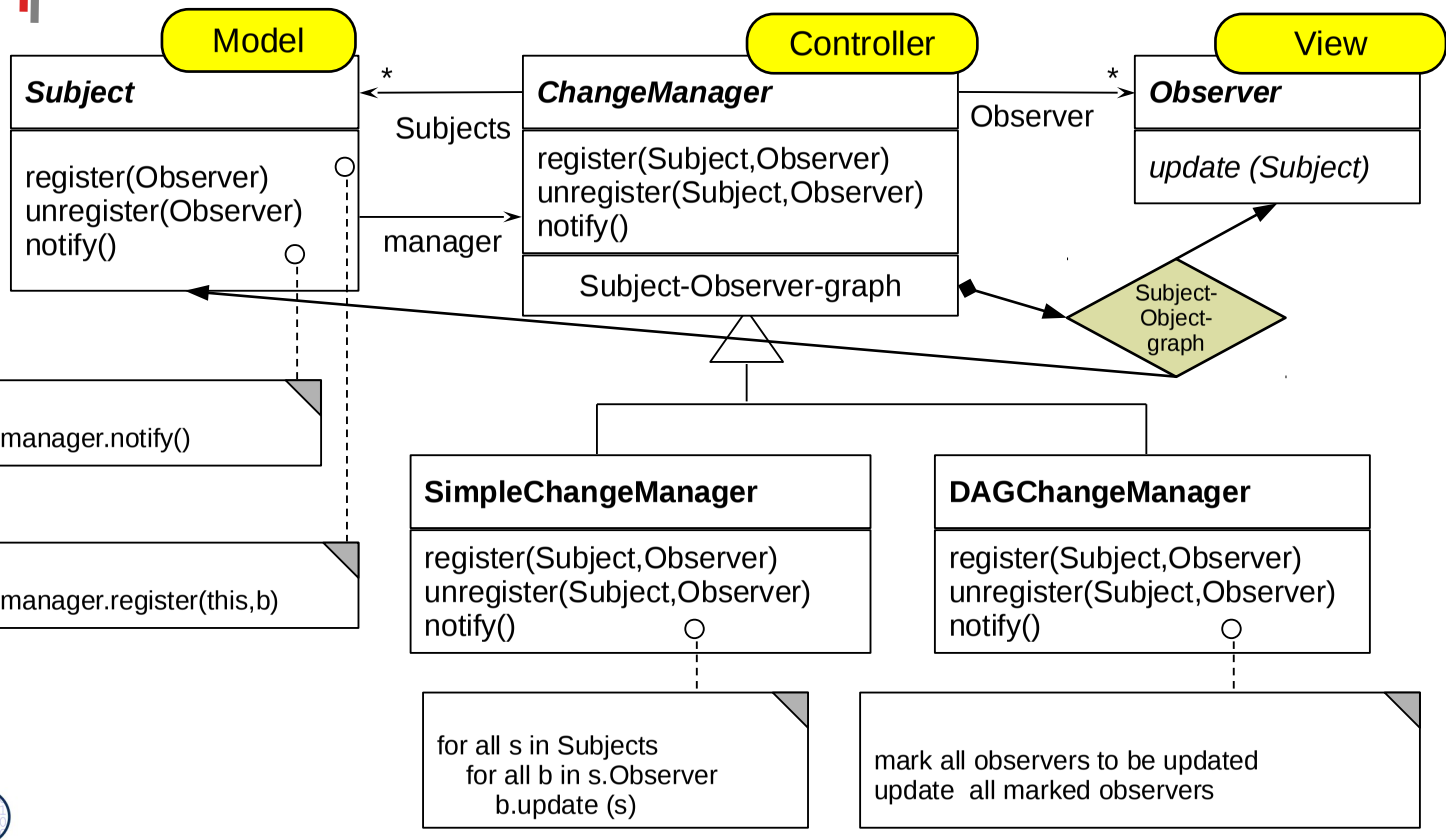
Sequence Diagram data-push-Observer

- Update() transfers Data to Observer (push)



Prof. U. Aßmann, Softwaretechnologie, TU Dresden

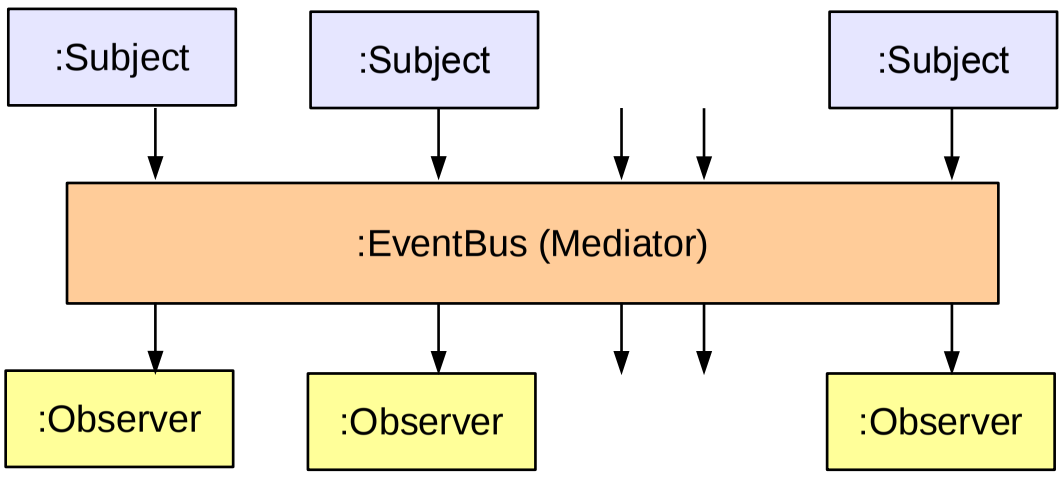
25.2.3.6 Observer with ChangeManager (EventBus)



Prof. U. Aßmann, Softwaretechnologie, TU Dresden

Observer with ChangeManager is also Called Event-Bus

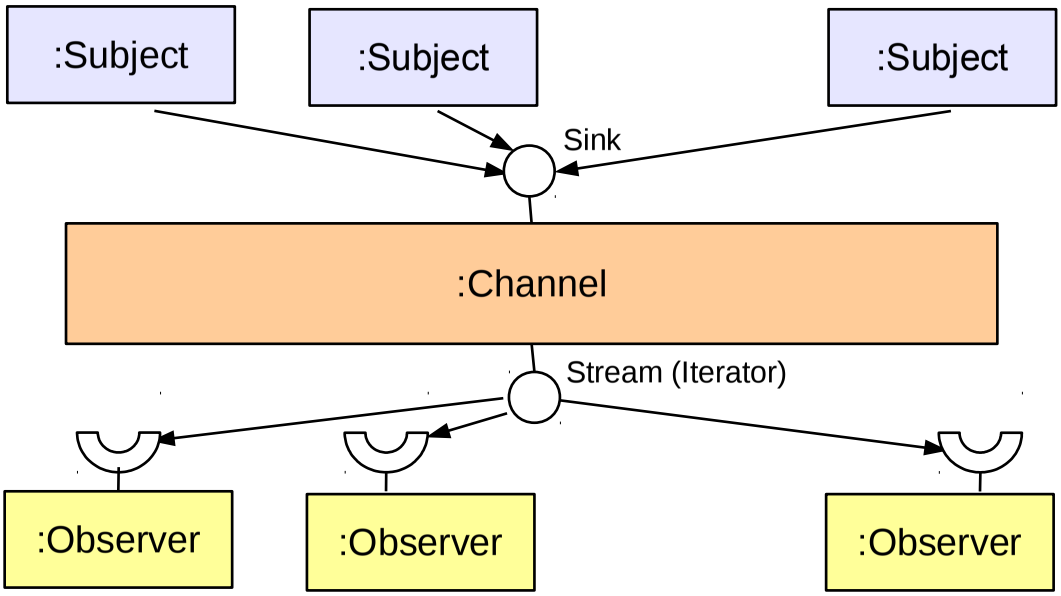
- Basis of many interactive application frameworks (Xwindows, Java AWT, Java InfoBus, ...)
- Loose coupling in communication
  - Observers decide what happens
- Dynamic extension of communication
  - Anonymous communication
  - Multi-cast and broadcast communication



Prof. U. Aßmann, Softwaretechnologie, TU Dresden

25.2.3.7 A Variant of EventBus is the Channel

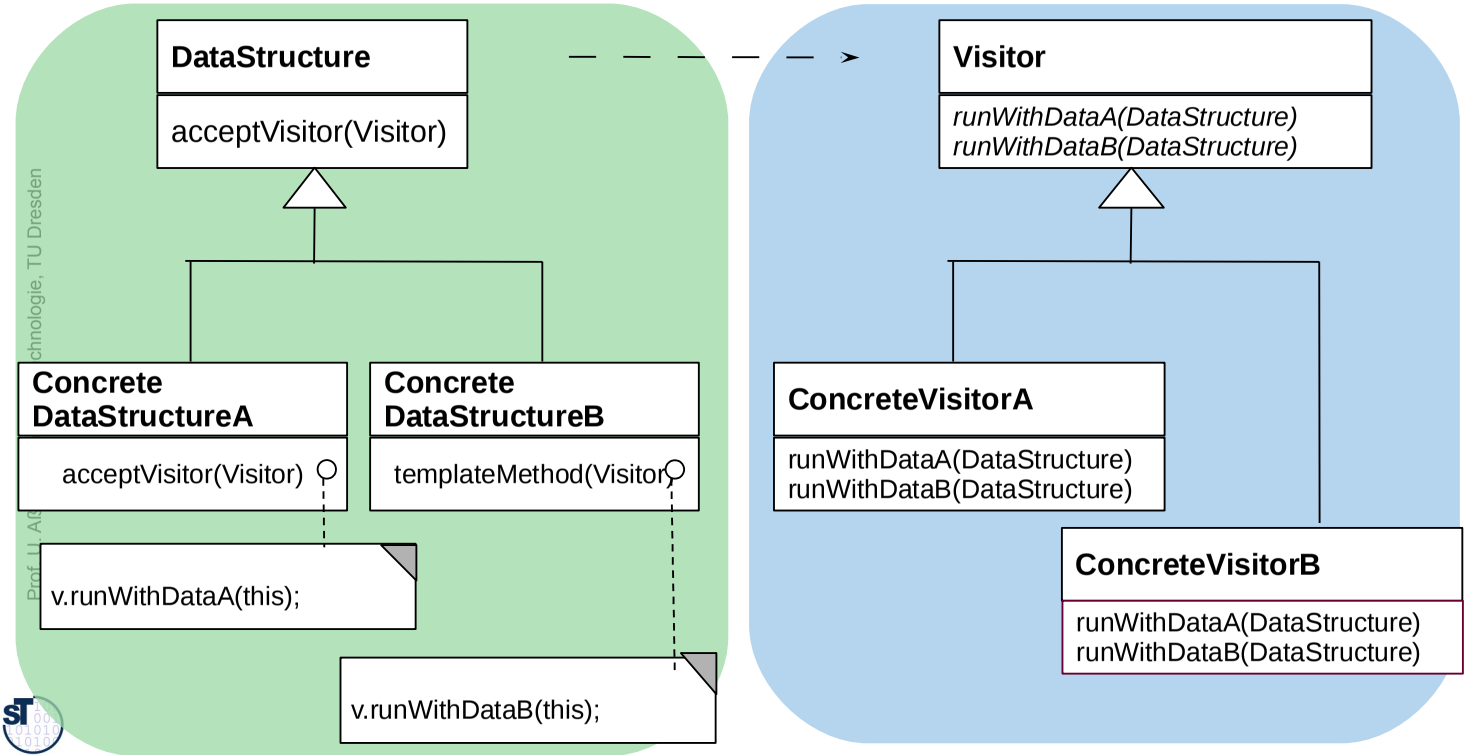
- push-Subjects and pull-Observers can be connected by Channel, to emphasize the continuous pushing and pulling
- Then Subjects write the Sink of the Channel and Observers pull the Stream of the Channel
  - Channel is a buffer



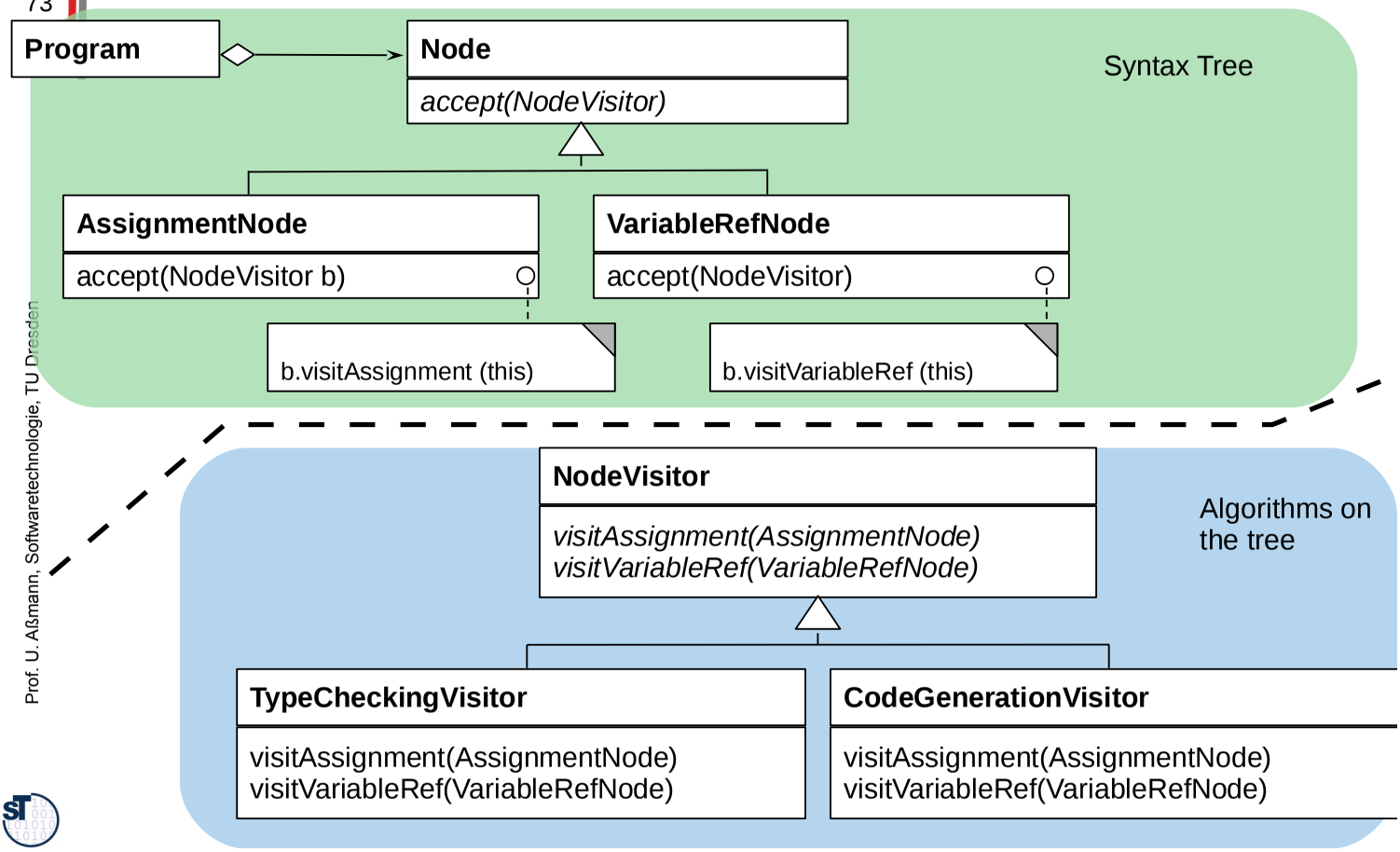
Prof. U. Aßmann, Softwaretechnologie, TU Dresden

25.2.4. Visitor

- Implementation of a 2-dimensional structure
  - First dispatch on dimension 1 (data structure), then on dimension 2 (algorithm)
- The Visitor has a lot of Callback methods

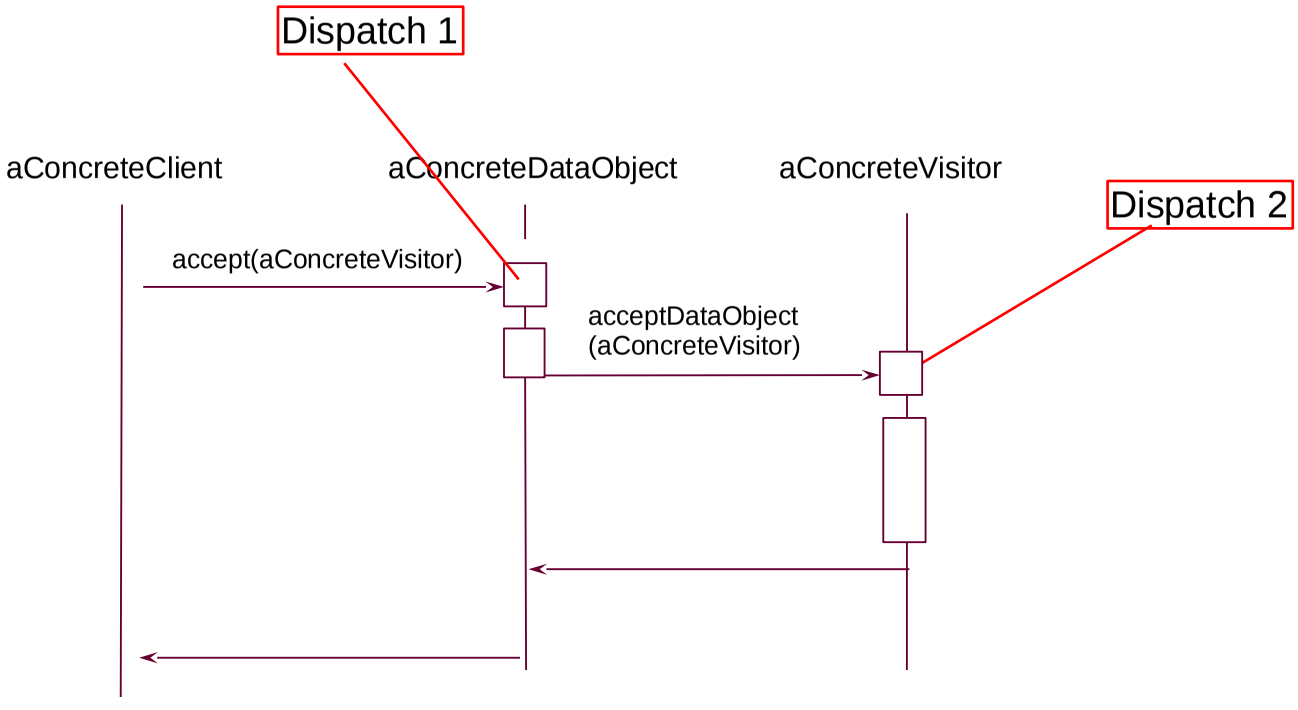


Working on Syntax Trees of Programs with Visitors



Sequence Diagram Visitor

- First dispatch on data, then on visitor



25.3) Patterns for Glue - Bridging Architectural Mismatch

25.3.1 Strukturmuster Singleton (dt.: Einzelinstanz)

- Problem:
  - Speicherung von globalem Zustand einer Anwendung
  - Sicherstellung, daß von einer Klasse genau ein Objekt besteht

Singleton
– theInstance: Singleton
getInstance(): Singleton

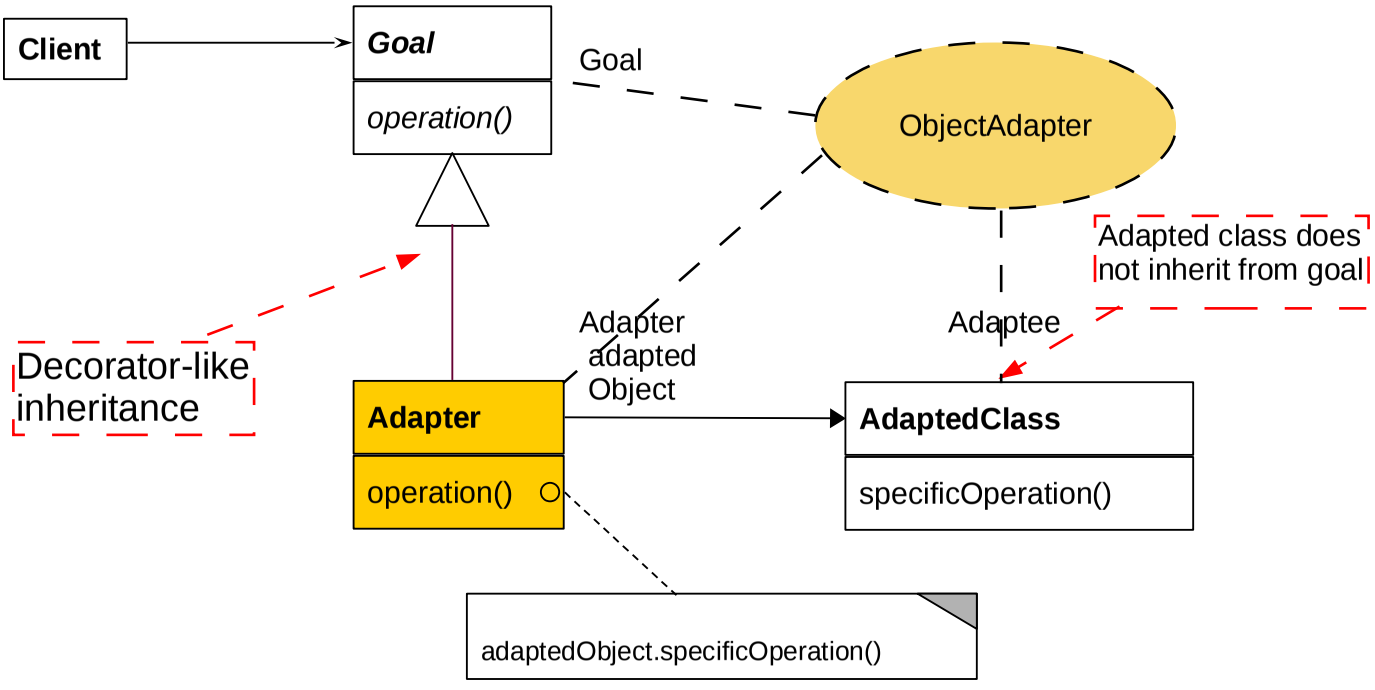
Expliziter Konstruktor wird (für andere Klassen) unbenutzbar gemacht!

```
class Singleton {

  private static Singleton theInstance;

  private Singleton () {}

  public static Singleton getInstance() {
    if (theInstance == null)
      theInstance = new Singleton();
    return theInstance;
  }
}
```

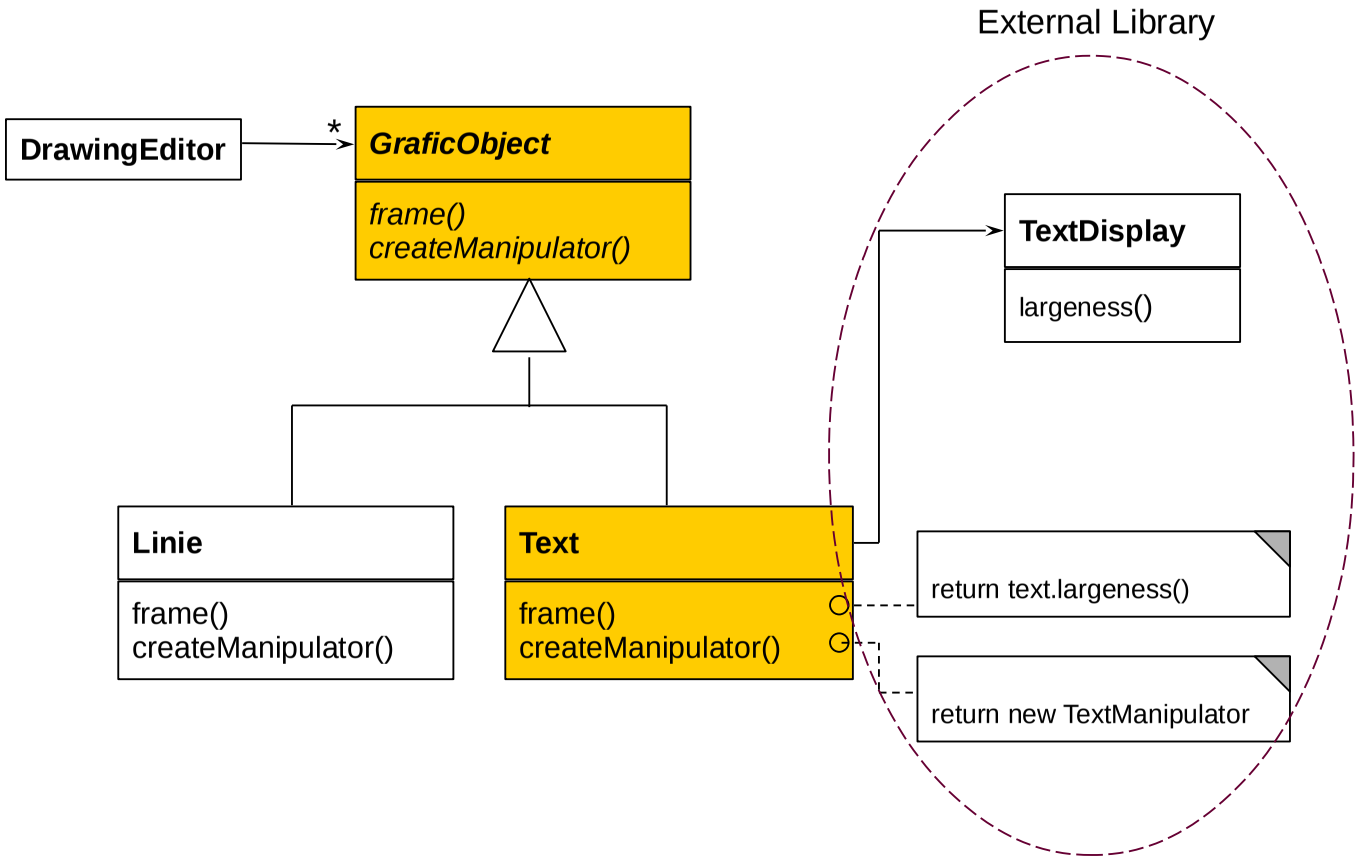

25.3.2 Adapter

Object Adapter

- An object adapter is a kind of a proxy
  - That maps one interface, protocol, or data format to another

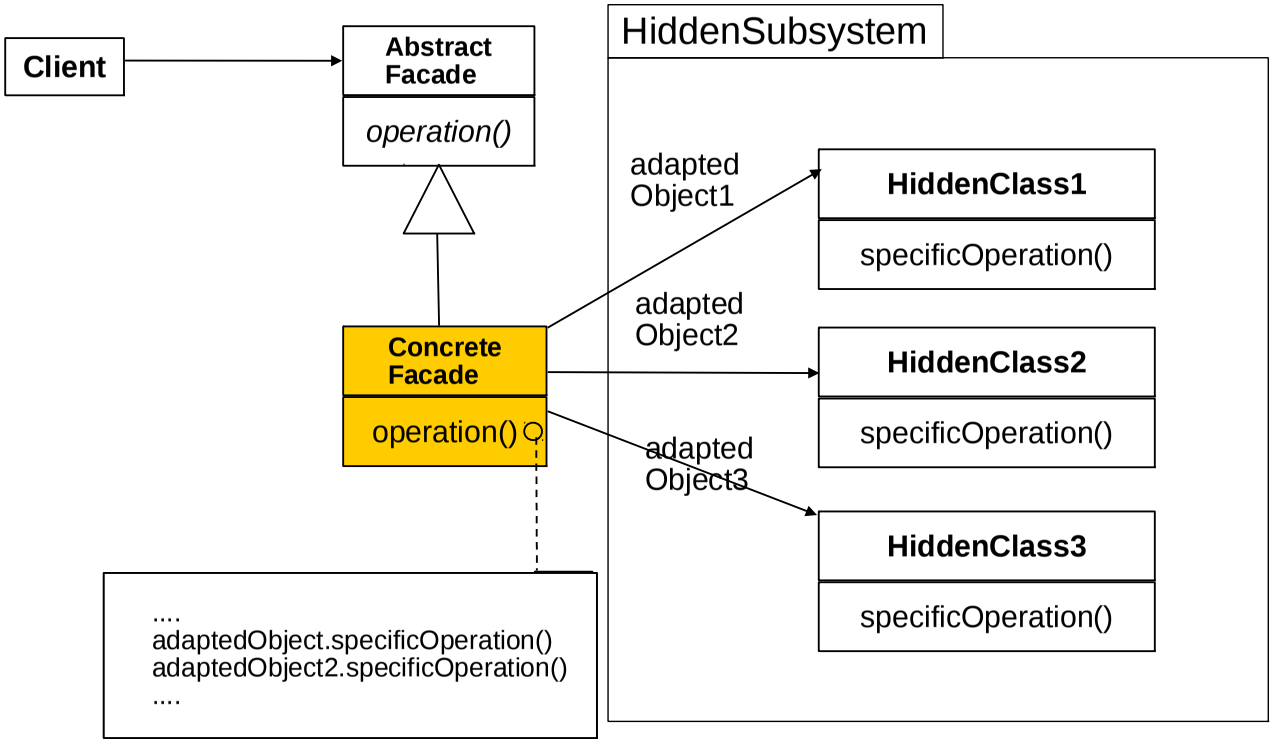


Example: Use of an External Class Library For Texts



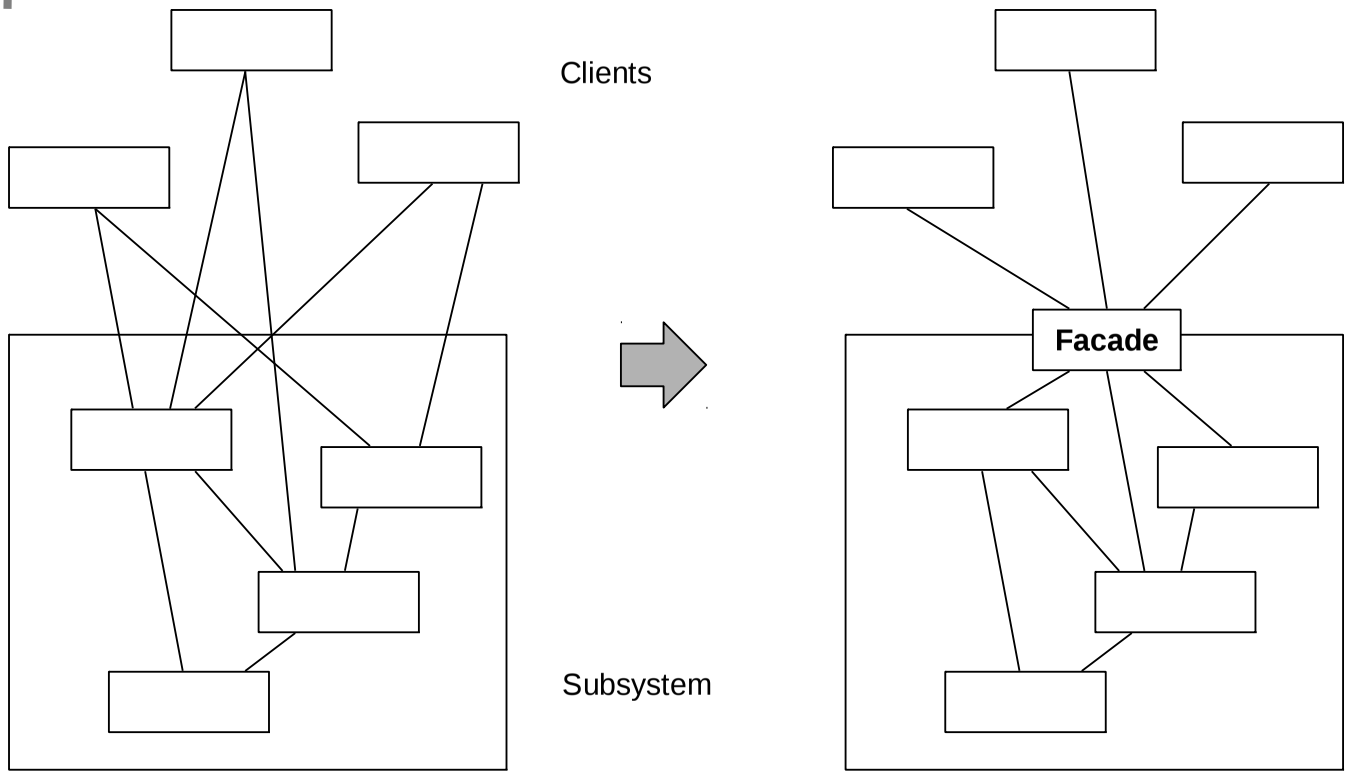
25.3.3 Facade Hides a Subsystem

- A **facade** is an object adapter hiding a complete set of objects (subsystem)
  - The facade has to map its own interface to the interfaces of the hidden objects



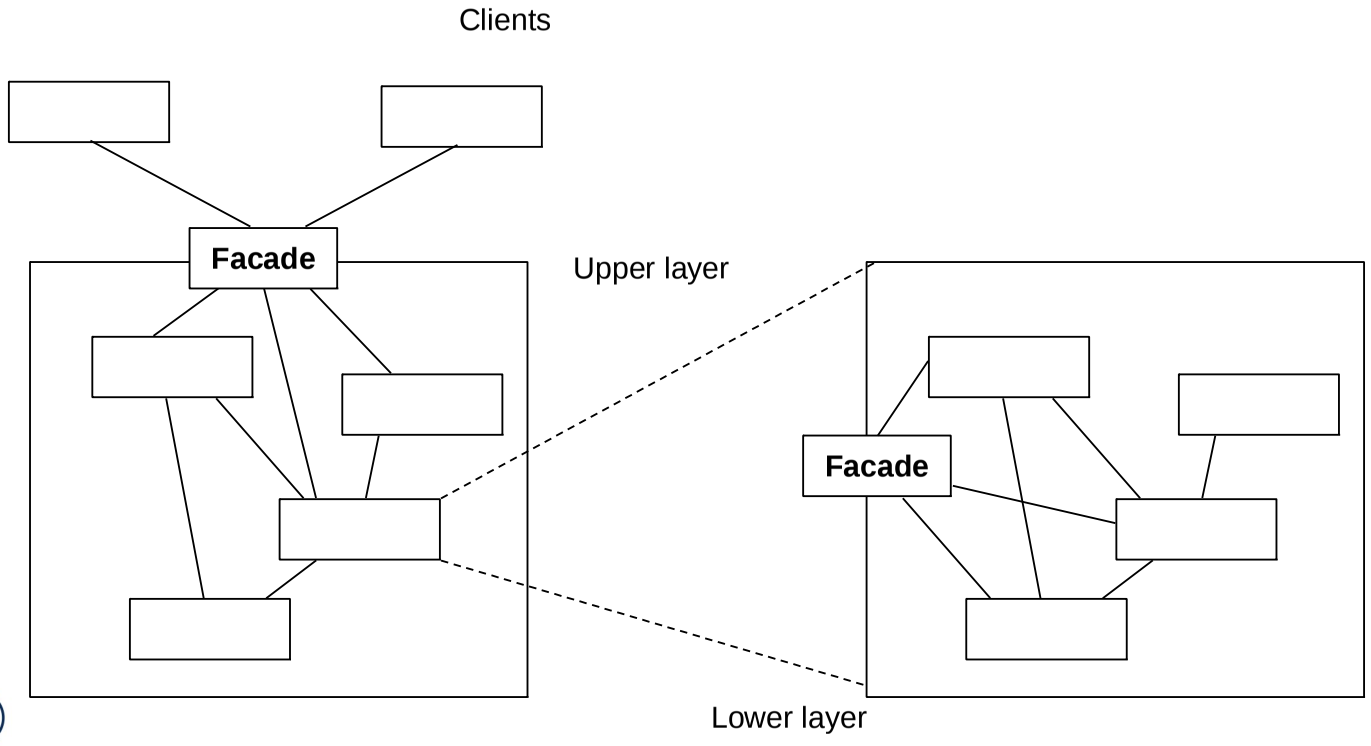
Refactoring Towards a Facade

- After a while, components are too much intermingled
- Facades serve for clear layered structure



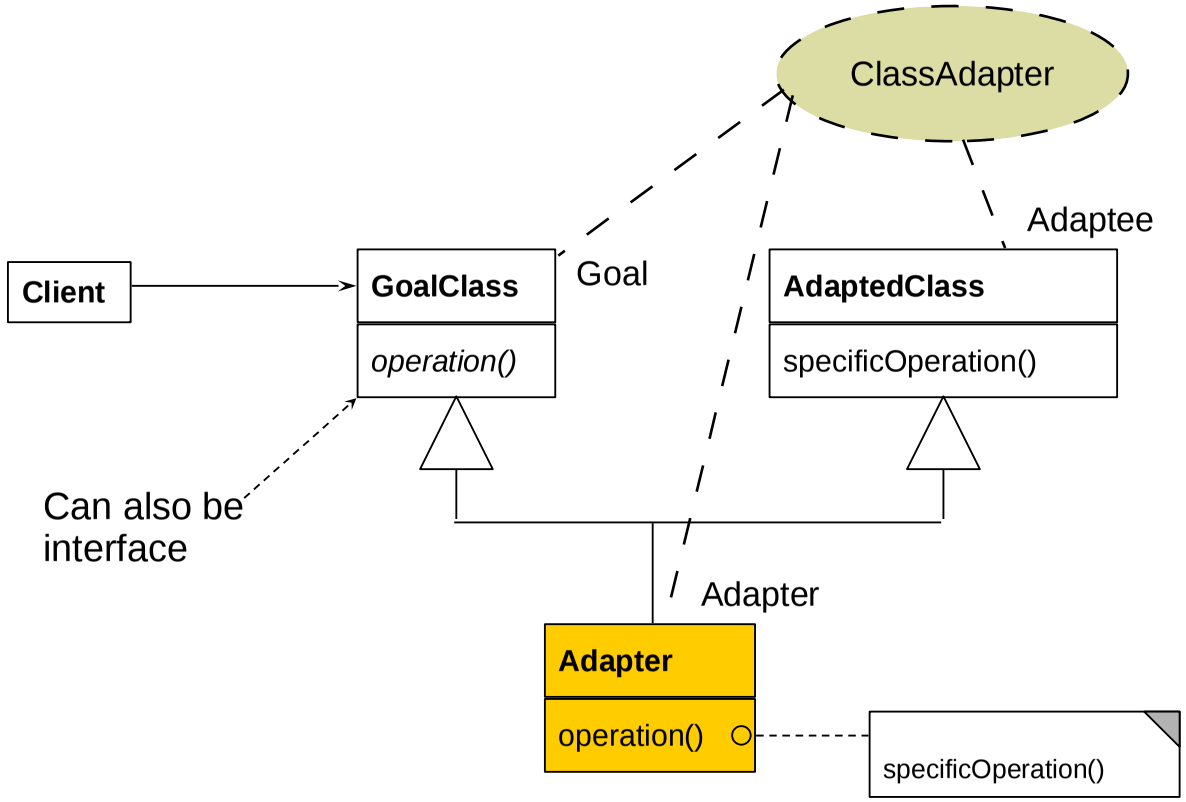
Facade and Layers

- If classes of the subsystem are again facades, **layers** result
  - Layers need nested facades



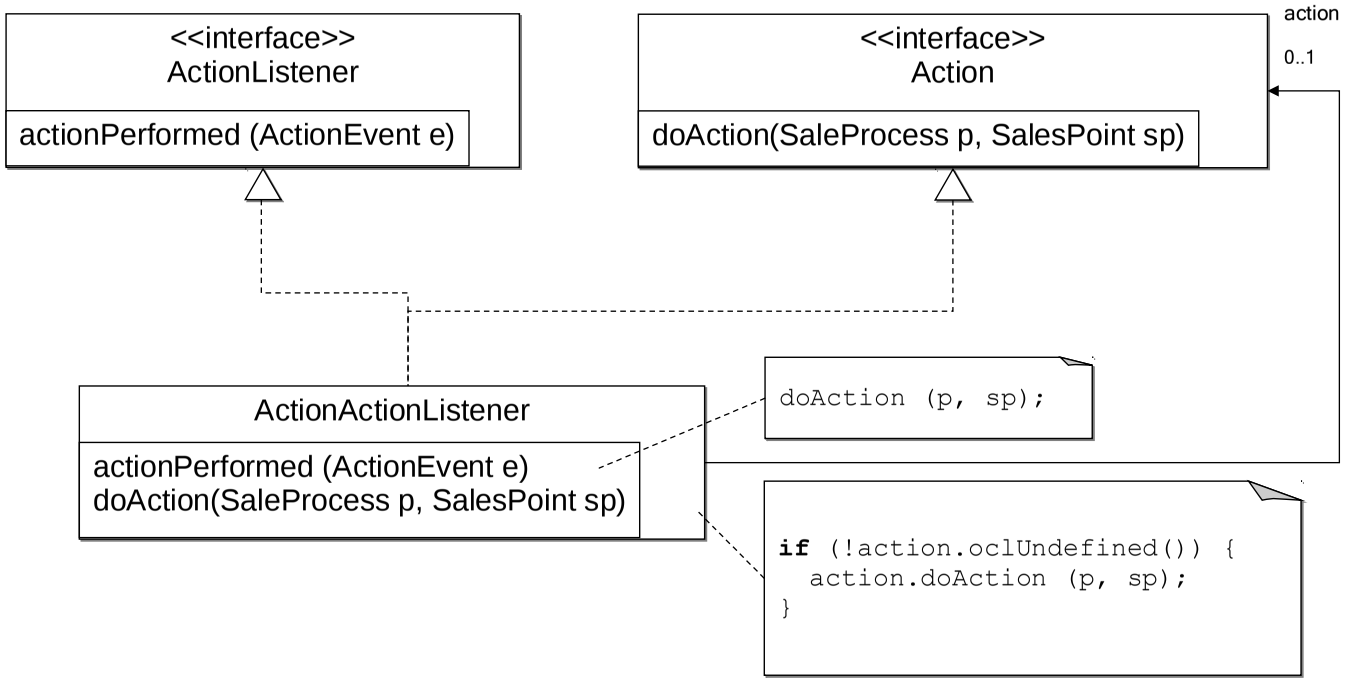
25.3.4 Class Adapter

- Instead of delegation, class adapters use multiple inheritance



Adapter for Observer in SalesPoint Framework

- In the SalesPoint framework (project course), a ClassAdapter is used to embed an Action class in an Listener of Observer Pattern



25.4 Other Patterns

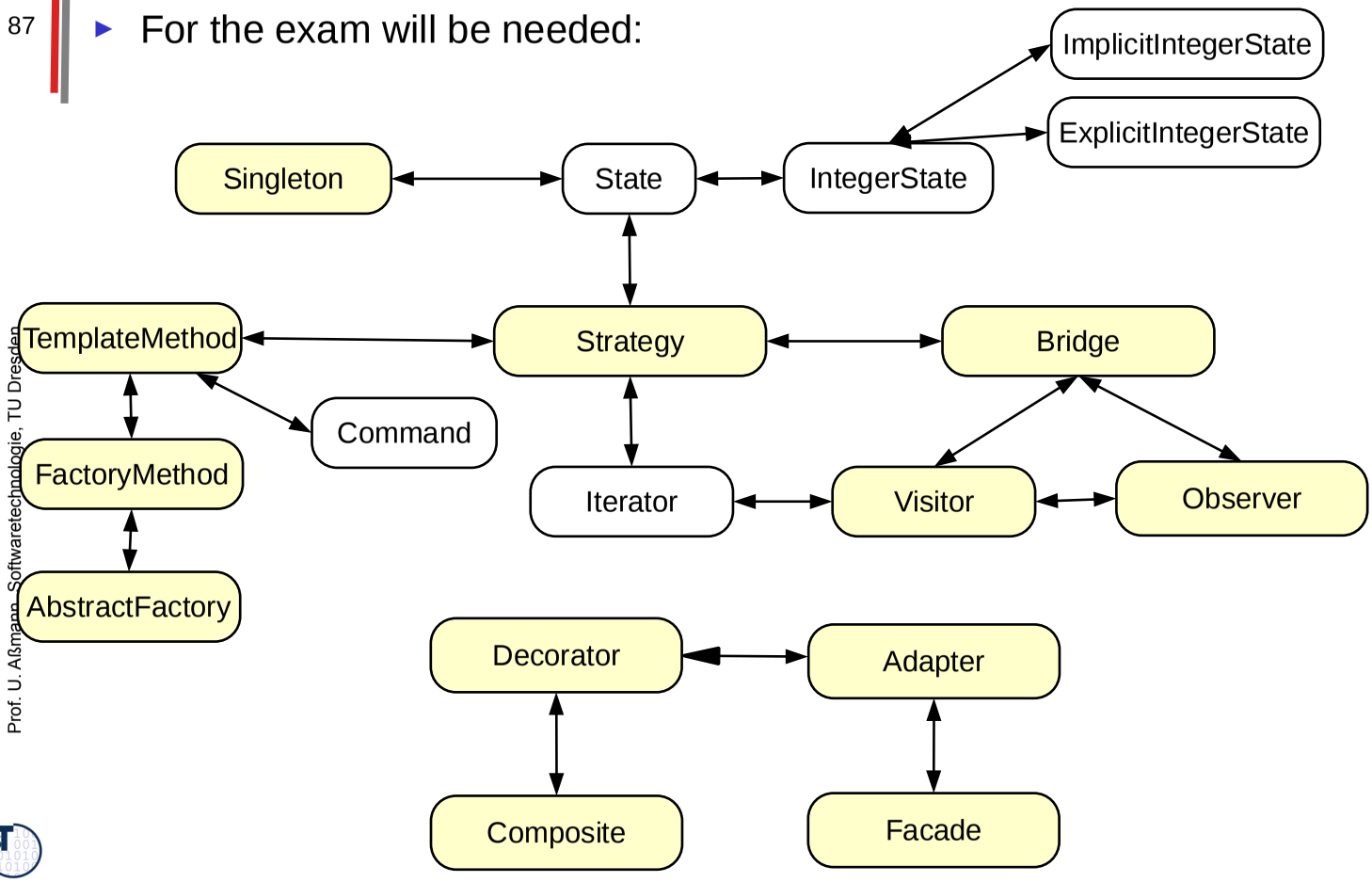
What is discussed elsewhere...

- Iterator
- Composite
- TemplateMethod
- Command

- Chapter “Analysis”:
  - State (Zustand), IntegerState, Explicit/ImplicitIntegerState
- Chapter “Architecture”:
  - Facade (Fassade)
  - Layers (Schichten)
  - 4-tier architecture (4-Schichtenarchitektur, BCED)
  - 4-tier abstract machines (4-Schichtenarchitektur mit abstrakten Maschinen)

Relations between Design Patterns

- For the exam will be needed:



Other Important GOF Patterns

Variability Patterns

- Visitor: Separate a data structure inheritance hierarchy from an algorithm hierarchy, to be able to vary both of them independently
- AbstractFactory: Allocation of objects in consistent families, for frameworks which maintain lots of objects
- Builder: Allocation of objects in families, adhering to a construction protocol
- Command: Represent an action as an object so that it can be undone, stored, redone

Extensibility Patterns

- Proxy: Representant of an object
- ChainOfResponsibility: A chain of workers that process a message

Others

- Memento: Maintain a state of an application as an object
- Flyweight: Factor out common attributes into heavy weight objects and flyweight objects
- Iterator: iterate over a collection

25.5 Design Patterns in a Larger Library

Design Pattern in the AWT

- AWT (Abstract Window Toolkit) is part of the Java class library
  - Uniform window library for many platforms (portable)
- Employed patterns
  - Observer (for widget super class java.awt.Window)
  - Compositum (widgets are hierarchic)
  - Strategy: The generic composita must be coupled with different layout algorithms
  - Singleton: Global state of the library
  - Bridge: Widgets such as Button abstract from look and provide behavior
    - Drawing is done by a GUI-dependent drawing engine (pattern bridge)
  - Abstract Factory: Allocation of widgets in a platform independent way

What Have We Learned?

- Design Patterns grasp good, well-known solutions for standard problems
- Variability patterns allow for variation of applications
  - They rely on the template/hook principle
- Extensibility patterns for extension
  - They rely on recursion
  - An aggregation to the superclass
  - This allows for constructing runtime nets: lists, sets, and graphs
  - And hence, for dynamic extension
- Architectural Glue patterns map non-fitting classes and objects to each other

Why is the Frauenkirche Beautiful?

- ..because she contains a lot of patterns from the baroque pattern language...

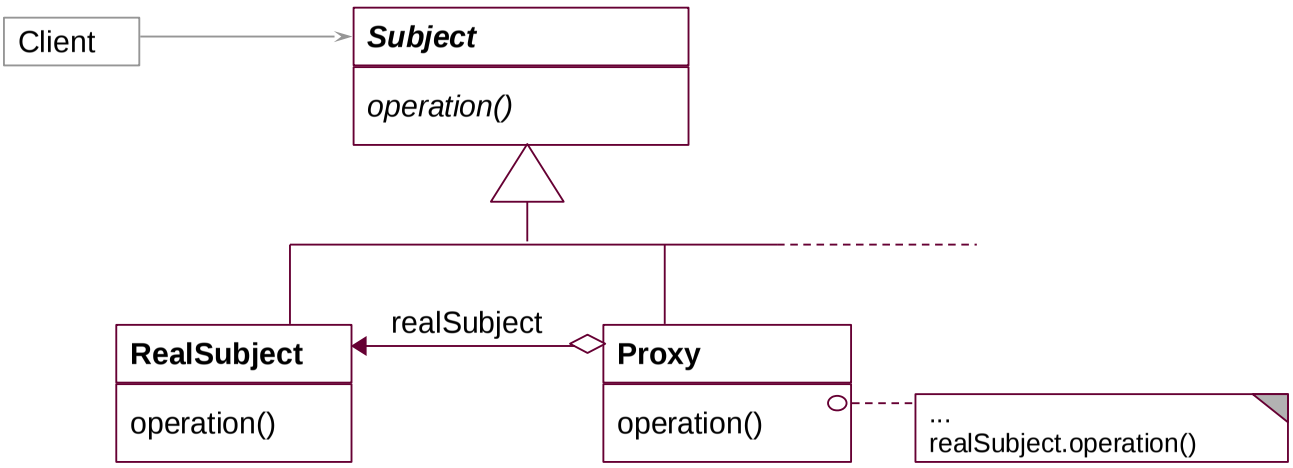
The End

- Design patterns and frameworks, WS, contains more material.
- © Uwe Aßmann, Heinrich Hussmann, Walter F. Tichy, Universität Karlsruhe, Germany, used by permission

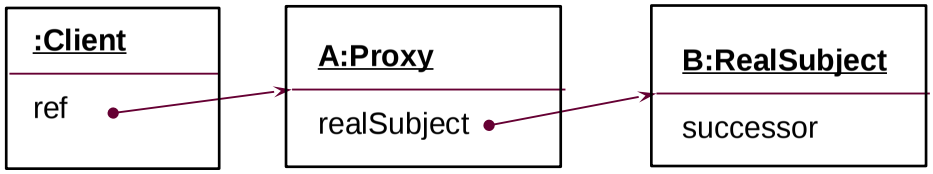
Proxy

Proxy

- Hide the access to a real subject by a representant



Object Structure:



Proxy

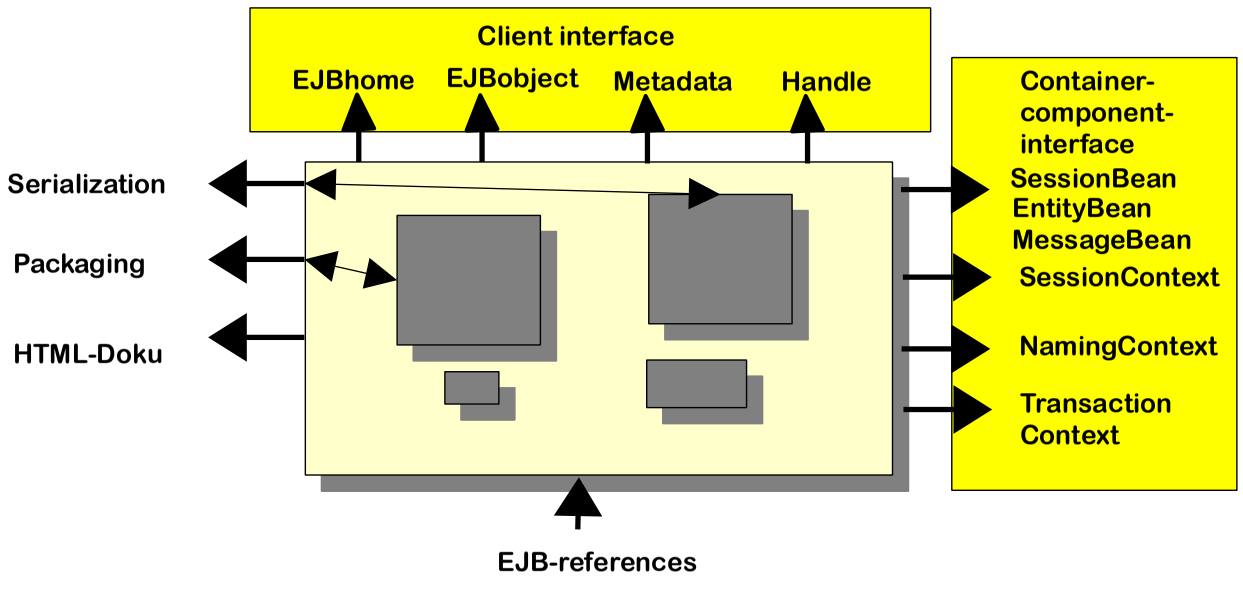
- The proxy object is a representant of an object
  - The Proxy is similar to Decorator, but it is not derived from ObjectRecursion
  - It has a direct pointer to the sister class, *not* to the superclass
  - It may collect all references to the represented object (shadows it). Then, it is a facade object to the represented object
- Consequence: chained proxies are not possible, a proxy is one-and-only
- It could be said that Decorator lies between Proxy and Chain.

Proxy Variants

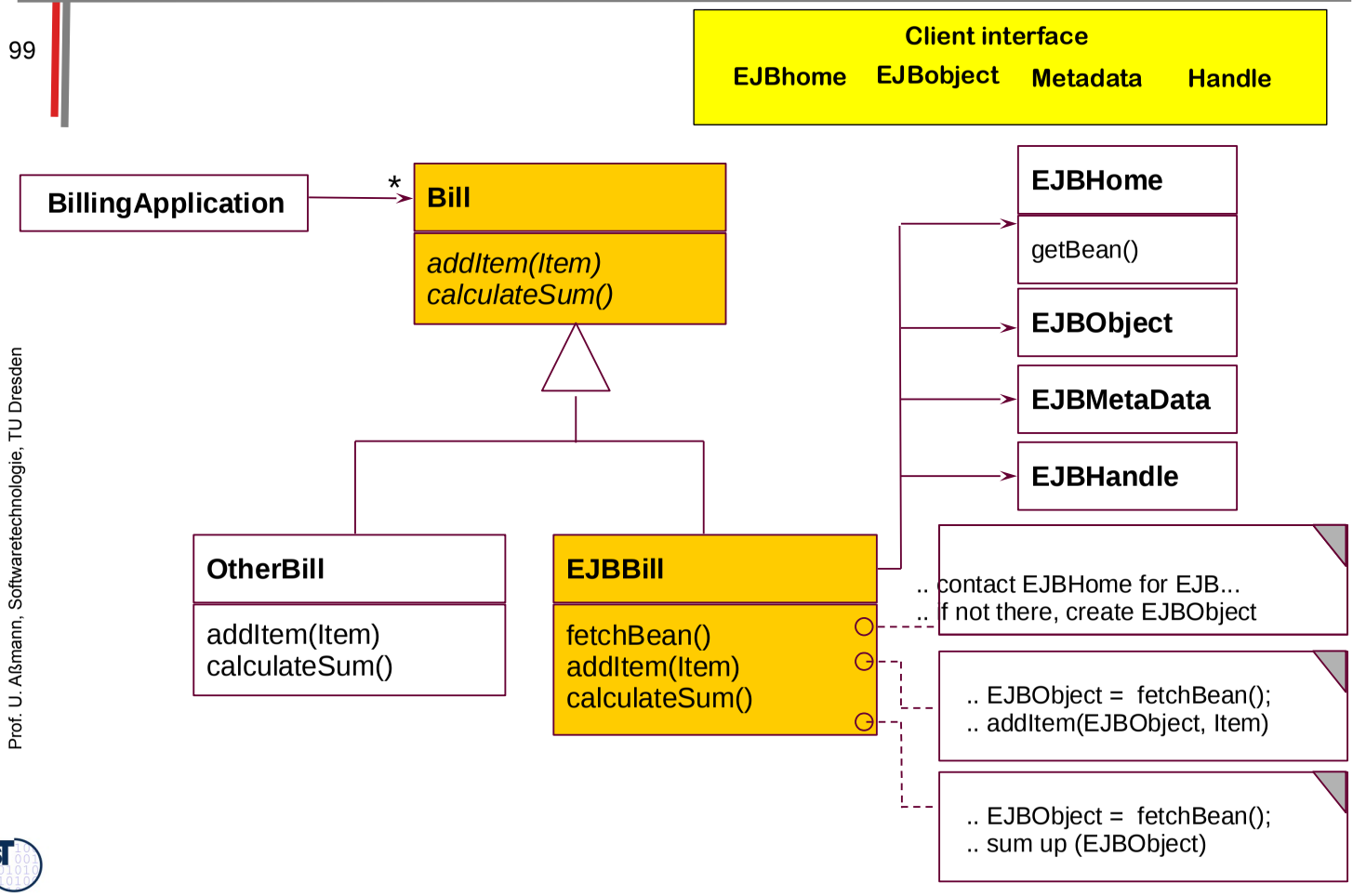
- **Filter proxy** (smart reference):
  - executes additional actions, when the object is accessed
- **Protocol proxy**:
  - Counts references (reference-counting garbage collection
  - Or implements a synchronization protocol (e.g., reader/writer protocols)
- **Indirection proxy** (facade proxy):
  - 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
  - Caching of remote objects for on-demand loading
- **Protection proxy**
  - Firewall proxy

Adapters for COTS

- Adapters are often used to adapt components-off-the-shelf (COTS) to applications
- For instance, an EJB-adapter allows for reuse of an Enterprise Java Bean in an application



EJB Adapter

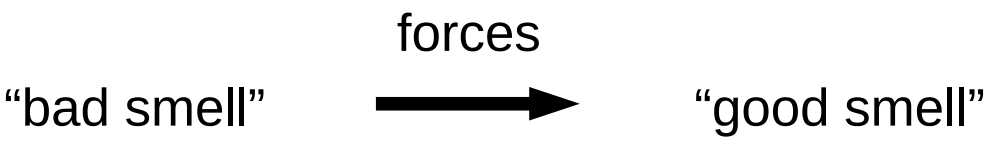


Prof. U. Aßmann, Softwaretechnologie, TU Dresden

Appendix

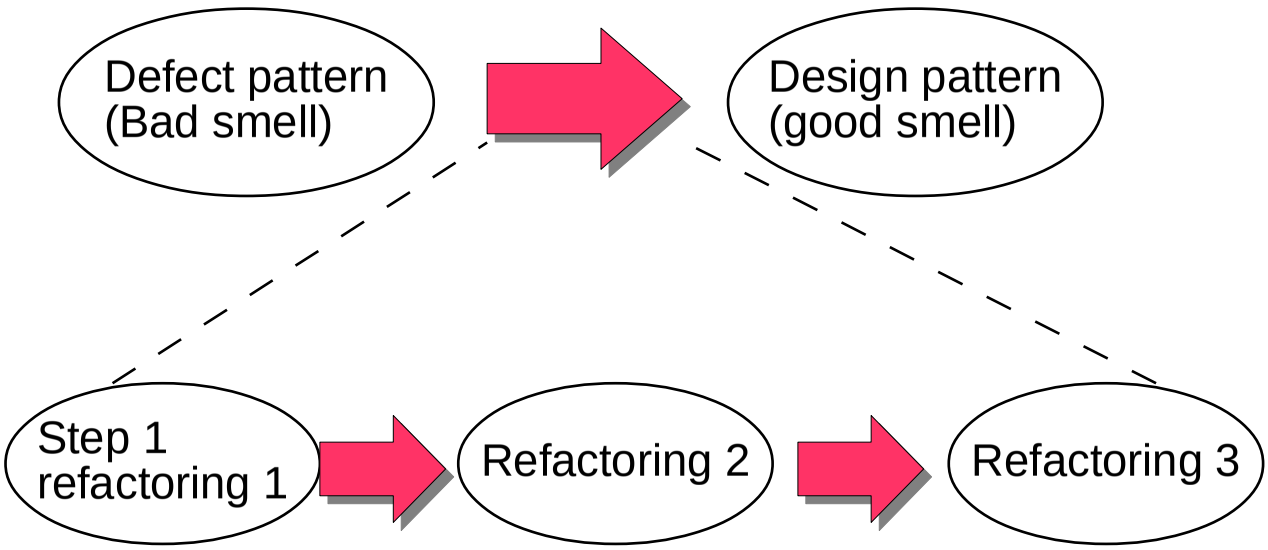
What Does a Design Pattern Contain?

- A part with a “bad smell”
  - A structure with a bad smell
  - A query that proved a bad smell
  - A graph parse that recognized a bad smell
- A part with a “good smell” (standard solution)
  - A structure with a good smell
  - A query that proves a good smell
  - A graph parse that proves a good smell
- A part with “forces”
  - The context, rationale, and pragmatics
  - The needs and constraints



Refactorings Transform Antipatterns (Defect Patterns, Bad Smells) Into Design Patterns

- Software can contain bad structure
- A DP can be a goal of a *refactoring,* transforming a bad smell into a good smell



Structure for Design Pattern Description (GOF Form)

- Name (incl. Synonyms) (also known as)
- Motivation (purpose)
  - also “bad smells” to be avoided
- Employment
- Solution (the “good smell”)
  - Structure (Classes, abstract classes, relations): UML class or object diagram
  - Participants: textual details of classes
  - Interactions: interaction diagrams (MSC, statecharts, collaboration diagrams)
  - Consequences: advantages and disadvantages (pragmatics)
  - Implementation: variants of the design pattern
  - Code examples
- Known Uses
- Related Patterns