



## 24. Entwurfsmuster für Produktfamilien (Product Line Patterns)

Prof. Dr. Uwe Aßmann

Lehrstuhl Softwaretechnologie

Fakultät für Informatik

TU Dresden

21-1.1, 6/5/21

- 1) Patterns for Variability
- 2) Patterns for Extensibility
- 3) Patterns for Glue
- 4) Other Patterns
- 5) Patterns in AWT

Achtung: Dieser Foliensatz ist teilweise in Englisch gefasst, weil das Thema in der Englisch-sprachigen Kurs "Design Patterns and Frameworks" (WS) wiederkehrt. Mit der Bitte um Verständnis.

Softwaretechnologie (ST) © Prof. U. Aßmann



DRESDEN  
concept  
Exzellenz aus  
Wissenschaft  
und Kultur

## Obligatory Literature

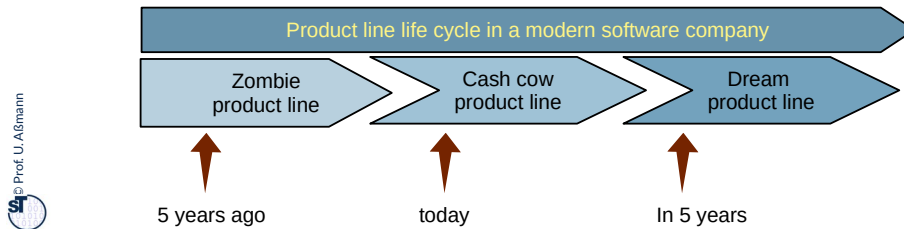
- ▶ ST für Einsteiger, Kap. Objektentwurf: Wiederverwendung von Mustern
- ▶ also: Chap. 8, Bernd Brügge, Allen H. Dutoit. Objektorientierte Softwaretechnik mit UML, Entwurfsmustern und Java. Pearson.
- ▶ James W. Cooper. Java™ Design Patterns: A Tutorial. Addison Wesley, 2000, ISBN: 0-201-48539-7
  - <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.183.2228&rep=rep1&type=pdf>
  - <http://www.informit.com/store/java-design-patterns-a-tutorial-9780201485394>  
Section Download
  - Download books at  
<http://www.freebookcentre.net/SpecialCat/Free-Design-Patterns-Books-Download.html>
- ▶ <https://refactoring.guru/design-patterns/java>
- ▶



## Standard Problems to Be Solved By *Product Line Patterns*

3 Softwaretechnologie (ST)

- ▶ Def.: A **software product line (SPL)** is a systematically engineered family of software products.
- ▶ **Product Line Patterns** are used to construct SPL, containing specific design knowledge about:
- ▶ **Variability: Exchanging parts easily**
  - Variation, variability, complex parameterization
  - Static and dynamic
  - For product lines, framework-based development
- ▶ **Extensibility: Add new features**
  - Software must change
- ▶ **Glue: Adapt to overcome architectural mismatches**
  - Coupling software that was not built for each other



Producing a product family (product line) is a successful business model for companies. Therefore, a systematic design towards product lines can be a decisive economic factor in the life of a company.

In a company, usually 2-3 product lines are active at the same time:

- The “zombie” product line is the one of the past, from which no new products are created, but old products are in use and must be maintained. Therefore, the product line also must be maintained
- The “cash cow” product line is the one of the present, from which new products are created
- The “dream” product line is designed for the future.



## 24.1) Patterns for Variability

Variability Pattern	# Run-time objects	Key feature
TemplateMethod	1	Simple variability
FactoryMethod	1	Simple variability
TemplateClass	2	Complex object
Strategy	2	Complex algorithm object
FactoryClass	3	Complex allocation of a family of objects
Bridge (DimensionalClass Hierarchy)	2	Complex object

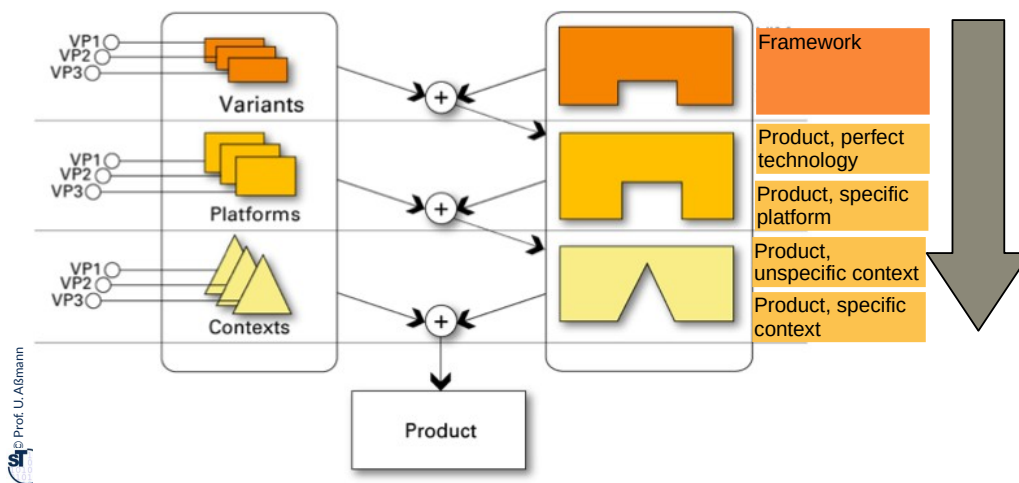
Software



## Why Do We Need Variability?

5 Softwaretechnologie (ST)

- ▶ **Functional** features, packages (payed vs free use), etc can be varied
- ▶ **Platforms** (Hardware, operating system, database, GUI package, etc.) should be varied
- ▶ **Dynamic contexts** (personalization, time and location) may vary

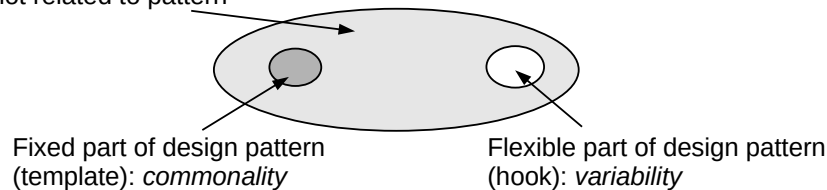


Variability takes place on different levels:

- compile time
  - functional variants
  - platform variants
- instantiation time
  - deployment variants (“configuration”)
- runtime
  - context variants

- ▶ A **variability design pattern** describes
  - Code *common* to several applications
    - Commonalities lead to *frameworks of product lines*
  - Code *different or variable* from application to application
    - Variabilities lead to *products of a product line*
- ▶ For capturing the communality/variability knowledge in variability design patterns, Pree invented the **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)

Rest of application  
not related to pattern



In a product family, common code must be extracted and reused. To this end, *commonality analysis* is conducted.

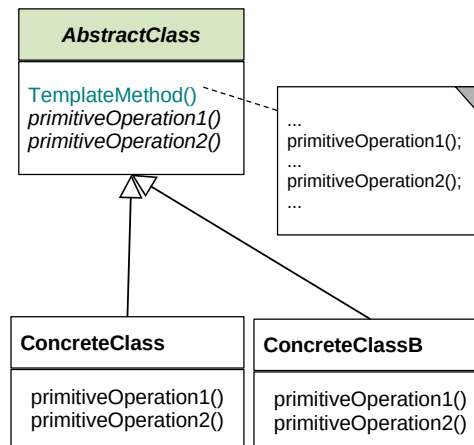
Also, variant (variable) code must be extracted and exchanged. To this end, *variability analysis* is conducted.

Then, put common code into

- superclasses and vary via specialisation and polymorphy
- core classes and the variable code into mixins; vary with mixin polymorphy

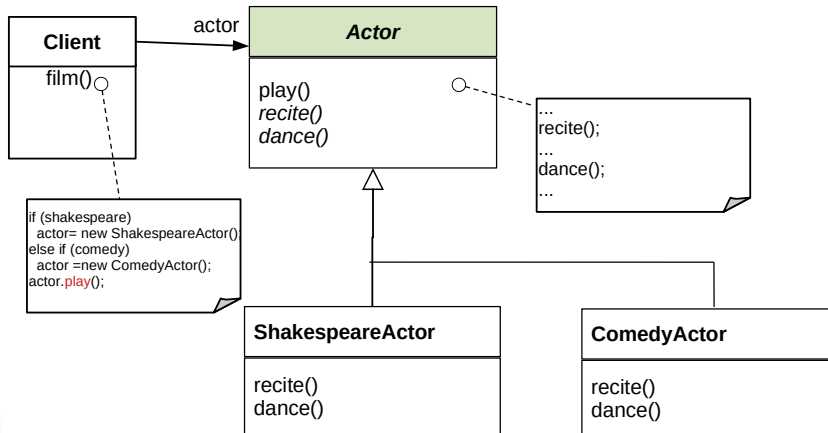
# TemplateMethod Pattern is a Variability Design Pattern (Rpt.)

- ▶ 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: TestCase in JUnit



## Actors and Genres as Template Method

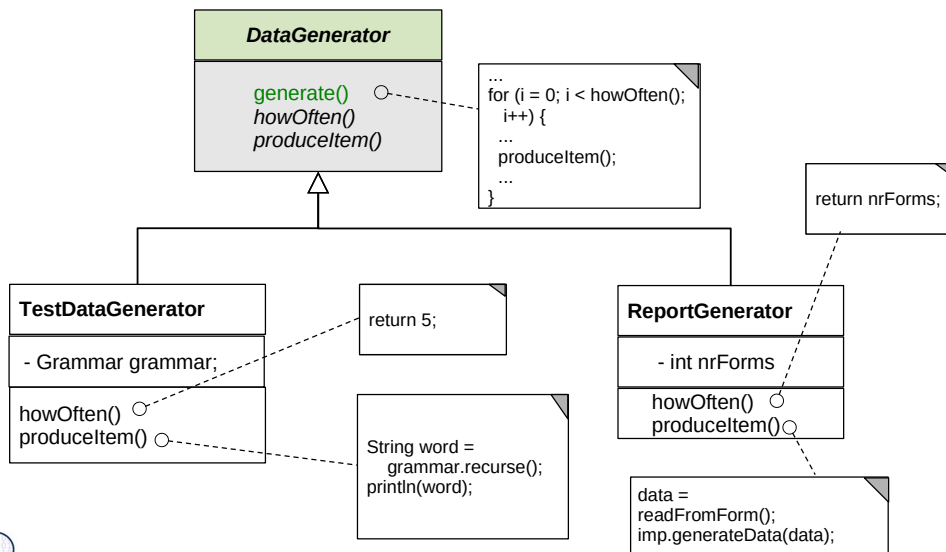
- ▶ Polymorphy in a common template method `play()`
- ▶ Binding an Actor's hook to be a ShakespeareActor or a Comedy Actor
- ▶ The behavior visible to a client will
  - be common in `play()`
  - but differ in two aspects, reciting and dancing





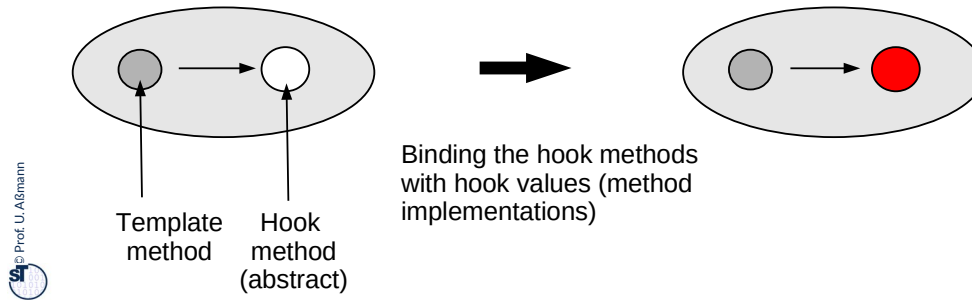
## Running Example: A Data Generator

- ▶ Parameterizing a data generator by frequency and kind of production



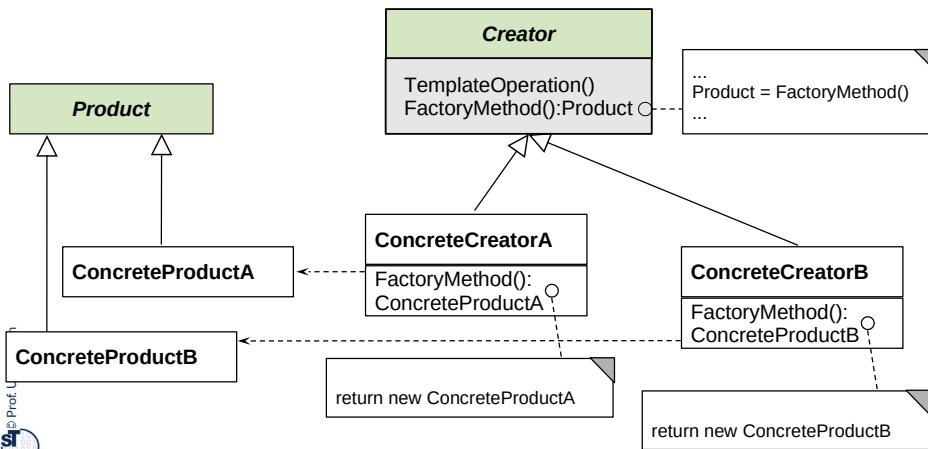
## Variability with TemplateMethod

- ▶ **Binding the hook method(s)** means to
  - Derive a concrete subclass from the abstract superclass, providing their implementation
- ▶ **Controlled variability** by only allowing for binding hook methods, but not overriding template methods



## 24.1.2 FactoryMethod

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





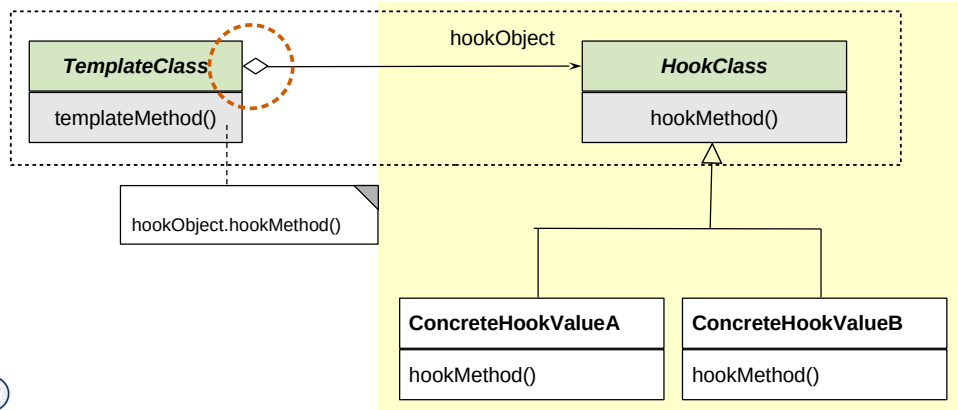
## 24.1.3 Strategy (Template Class)

- ▶ The STRATEGY pattern is a TEMPLATE CLASS pattern with the same structure, but a more specific intent



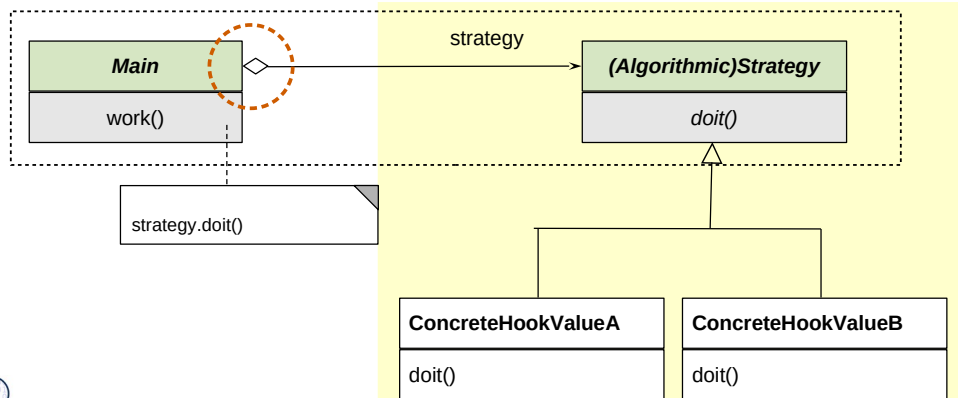
# Template Class

- ▶ 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
  - May exchange several methods (a set of methods) at the same time
- ▶ This pattern is basis of Bridge, Builder, Command, Iterator, Observer, Visitor.



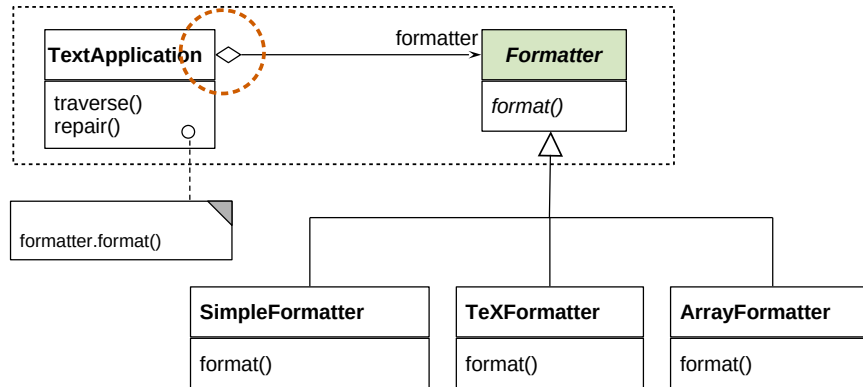
## Strategy (Specific Template Class with Algorithm Mixin)

- ▶ Similar to TemplateClass, but different intention
  - Consistent exchange of **several parts of an algorithm within a main object**, not only one method
- ▶ This pattern is basis of Bridge, Builder, Command, Iterator, Observer, Visitor.



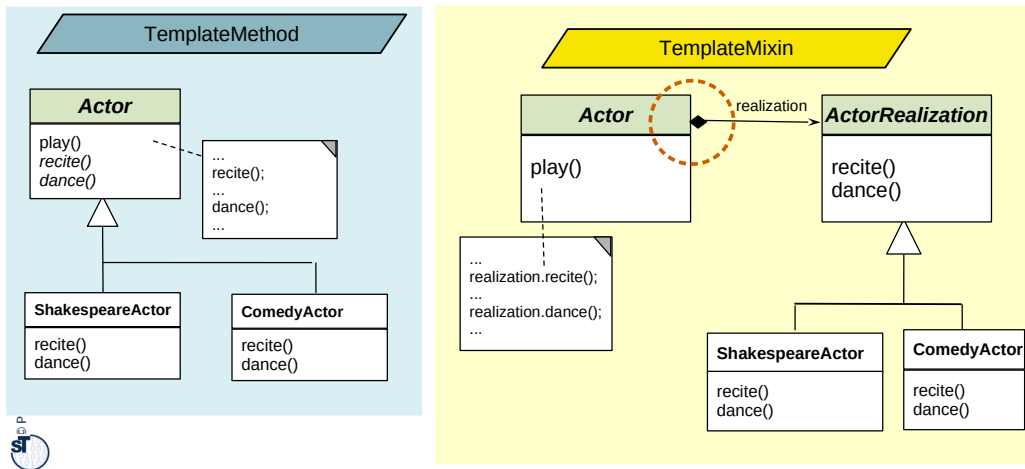
## Example for (algorithmic) Strategy

- ▶ Strategy represents an algorithm as object (but Command calls it `execute()`)
- ▶ Ex.: complex formatting algorithm
- ▶ Strategy objects are often subobjects of complex objects



## Variants of TemplateClass: Strategy, TemplateMethod, TemplateMixin

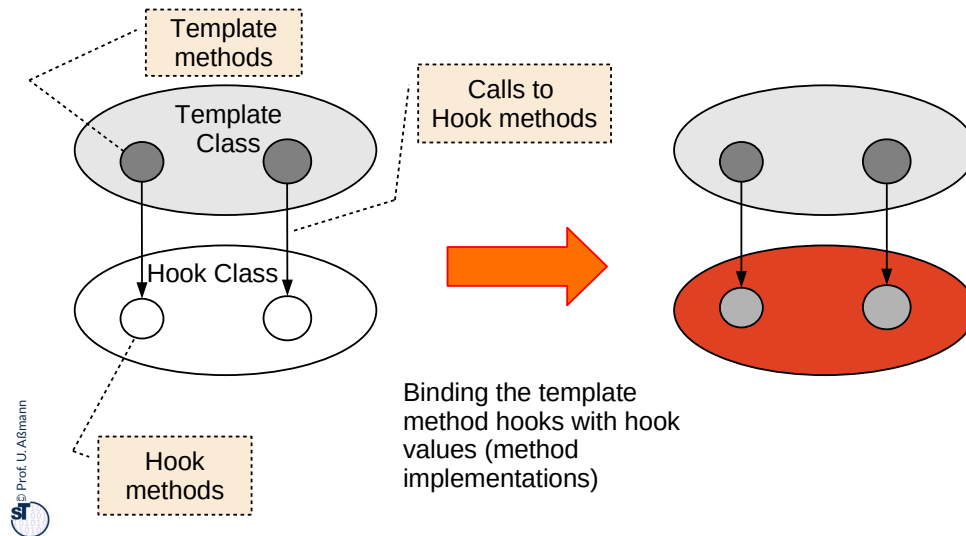
- ▶ TemplateMethod creates *one* run-time object
- ▶ TemplateClass creates *two physical objects belonging to one logical object*
- ▶ TemplateMixin is a TemplateClass with *Mixin and Composition*
- ▶ Strategy is a TemplateClass with algorithmic Hook object





## Variability with Strategy

- ▶ Binding the hook class of a Strategy means to derive a concrete subclass from the **abstract hook superclass**, providing the implementation of the hook method



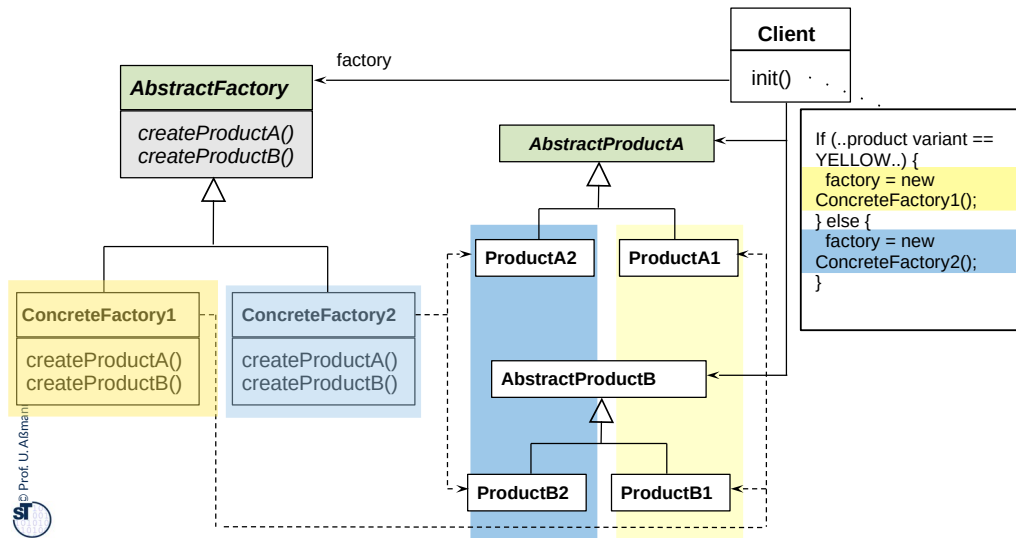


## 24.1.4. Factory Class



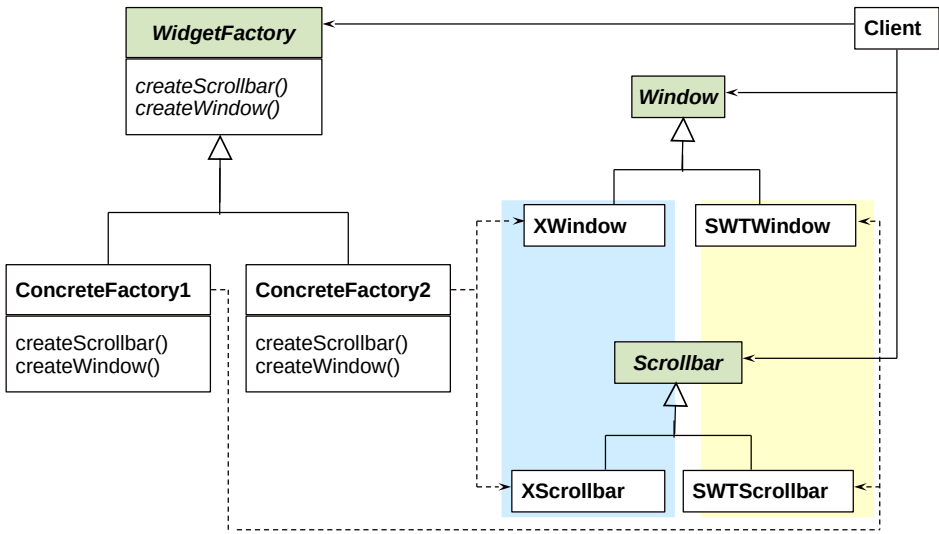
## 24.1.4 Factory Class (Abstract Factory)

- ▶ Allocate a family of products {Ai, Bi, ..} in different “flavors” or “colors” {1, 2, ..}
- ▶ Vary consistently by exchange of factory and object families



# Example for Factory Class

- Consistently varying a family of widgets



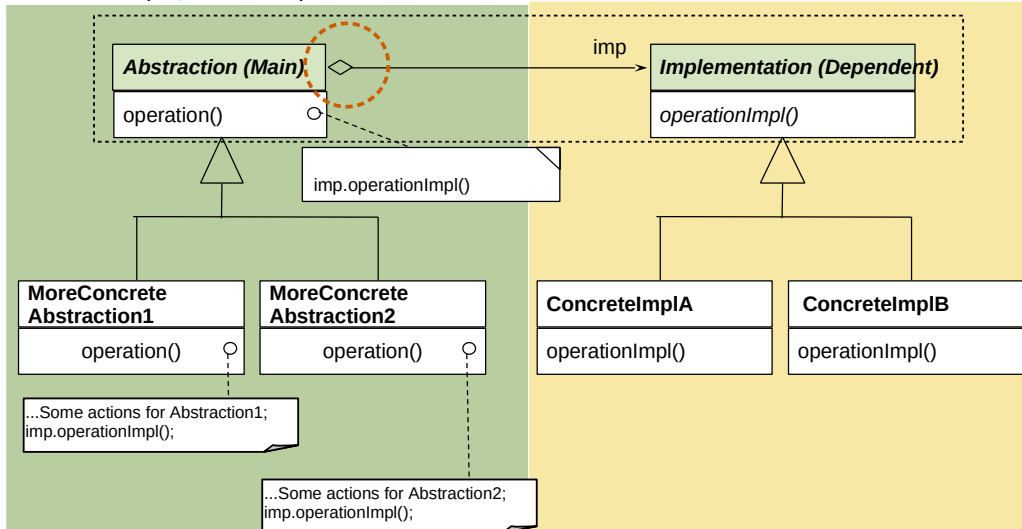


## 24.1.5 Bridge (Dimensional Class Hierarchies)



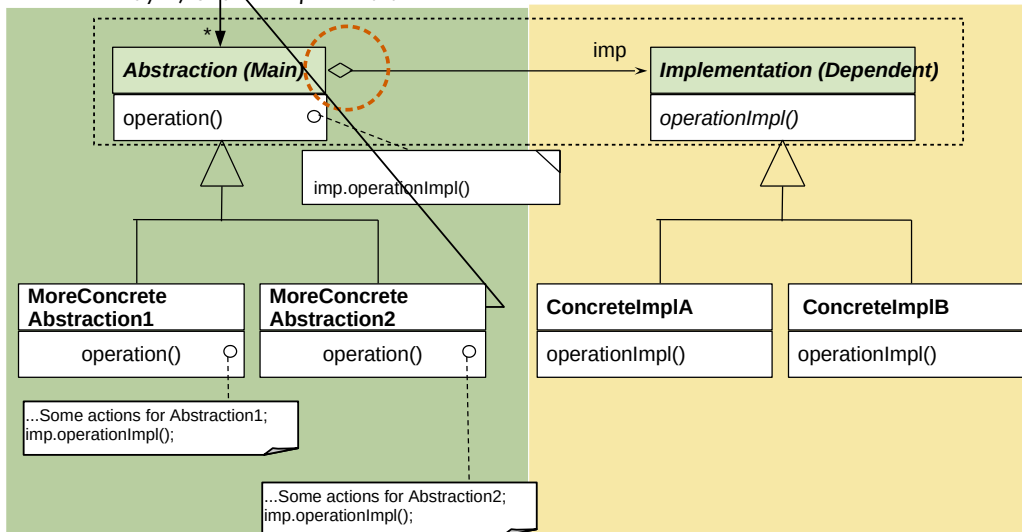
## Bridge for Complex Objects ( GOF-Version)

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

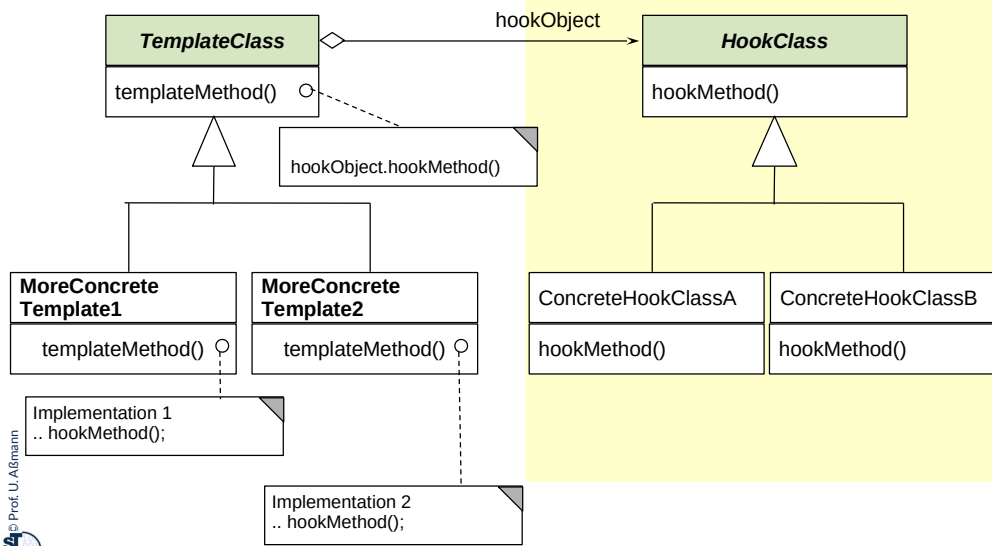


## Bridge for Complex Objects ( GOF-Version)

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



- DimensionalClassHierarchies is an extension of TemplateClass

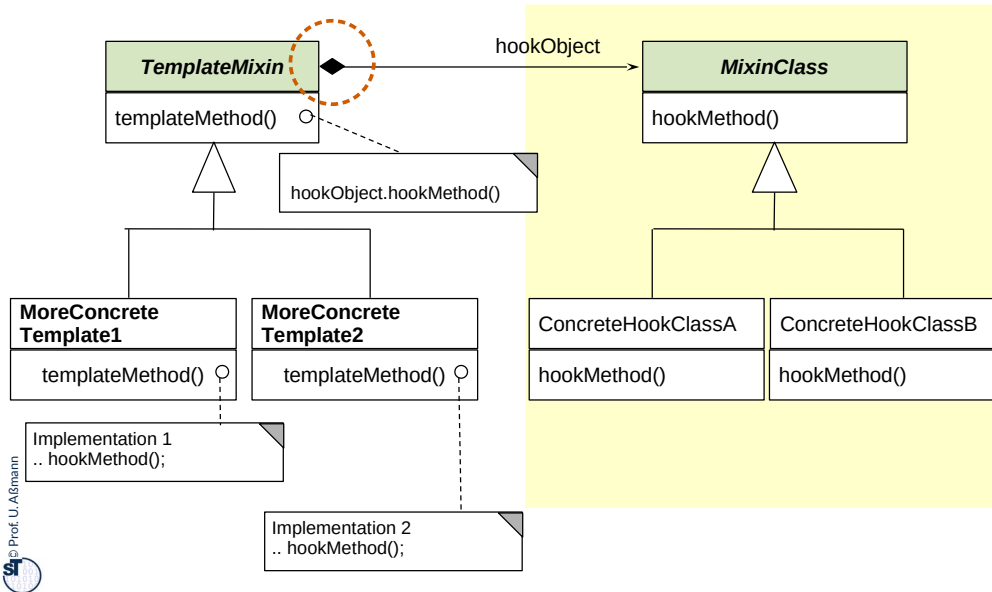


- A Bridge varies also the template class in a class hierarchy
  - The sub-template classes can adapt the template algorithm
  - Important: the sub-template classes must fulfil the contract of the superclass
    - Although the implementation can be changed, the interface and visible behavior must be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



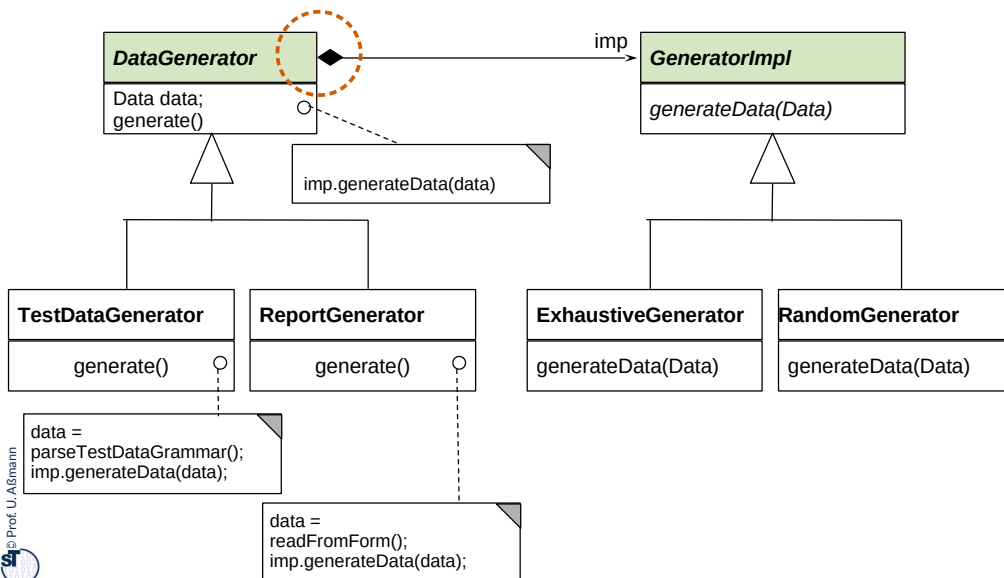
## TemplateMixin as Dimensional Mixin Variation

- ▶ Bridge is an extension of TemplateClass



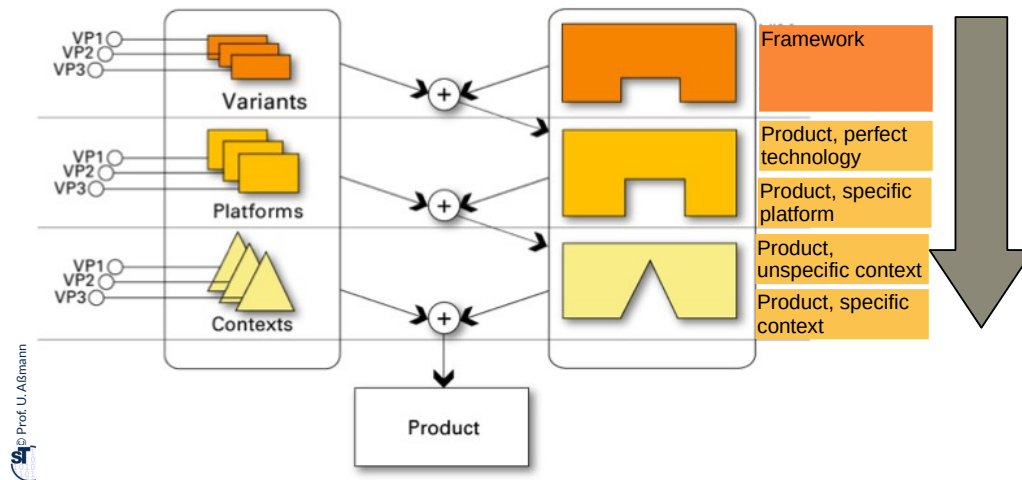
- A Bridge varies also the template class in a class hierarchy
  - The sub-template classes can adapt the template algorithm
  - Important: the sub-template classes must fulfil the contract of the superclass
    - Although the implementation can be changed, the interface and visible behavior must be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

## Ex. Complex Object *DataGenerator* as Bridge



## Rpt.: Why Do We Need Variability?

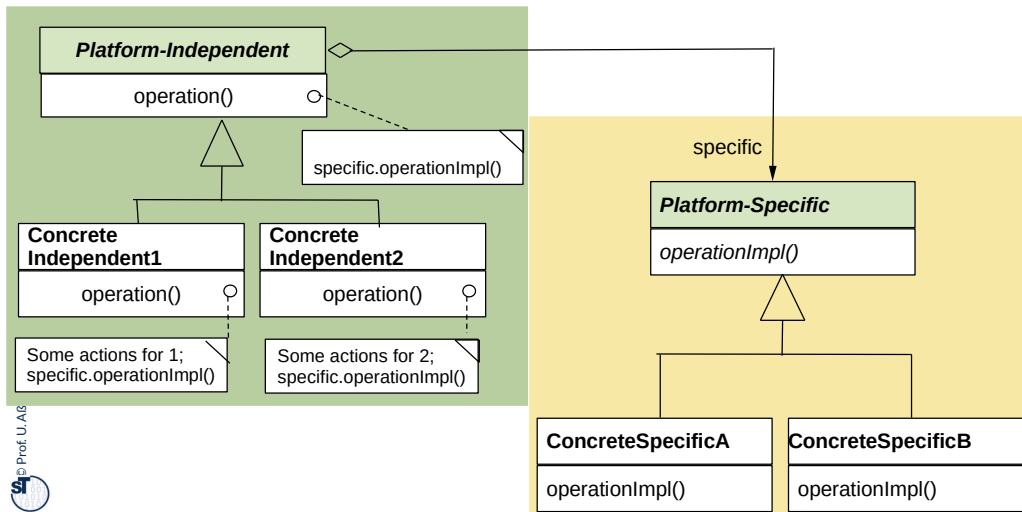
- ▶ Functional features, packages (payed vs free use), etc
- ▶ Platforms (Hardware, operating system, database, GUI package, etc.)
- ▶ Dynamic contexts (personalization, time and location)



## Use of Bridge Patterns for Separation of Platform-Independent from Platform-Dependent Code

28 Softwaretechnologie (ST)

- ▶ 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



Bridges can hide inside platform-specific code (in the hook object).

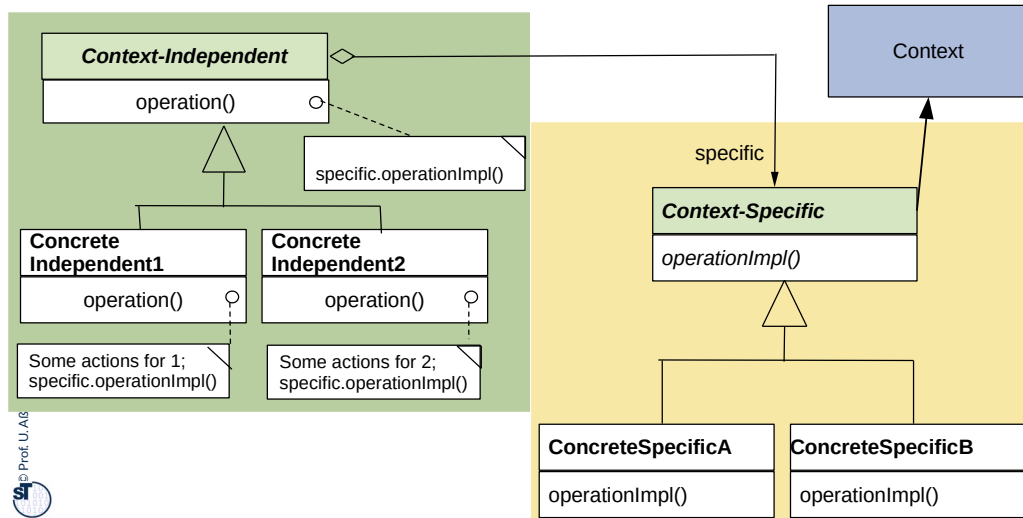
Whenever the software should be ported to a new platform, a new variant of the hook class must be programmed. Splitting the platform-independent from the platform-specific code enables the developer to reuse the platform-independent code for many platforms. If all objects in a program are programmed with "platform bridges", the software is very easy to port to new platforms.

Bridge is one of the most important patterns for software product lines.

## Use of Bridge for Separation of Context-Independent from Context-Dependent Code

29 Softwaretechnologie (ST)

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



Bridges can hide inside context-specific code that is *varied dynamically* (in the hook object).

Whenever the software should work differently in a new dynamic context, the hook object can be varied dynamically. Bridge is one of the most important patterns for *dynamic* software product lines.



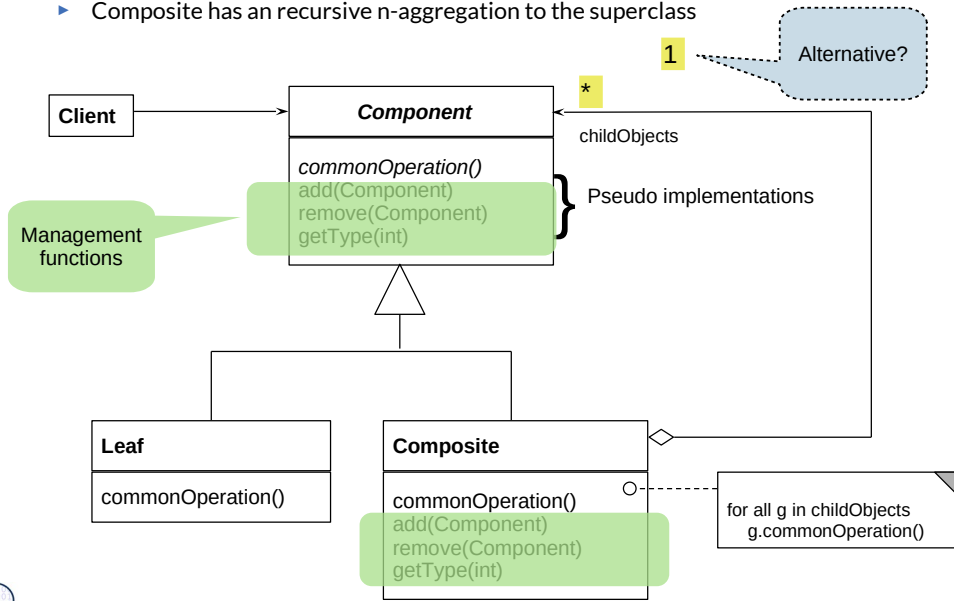
## 24.2) Patterns for Extensibility

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

Extensibility Pattern	# Run-time objects	Key feature
Composite	*	Whole/Part hierarchy
Decorator	*	List of skins
Callback	2	Dynamic call
Observer	1+*	Dynamic multi-call
Visitor	2	Extensible algorithms on a data structure
EventBus, Channel	*	Complex dynamic communication infrastructure (Appendix)

## 24.2.1 Structure Composite (Rpt.)

- ▶ Composite has an recursive n-aggregation to the superclass





## 24.2.2. Decorator

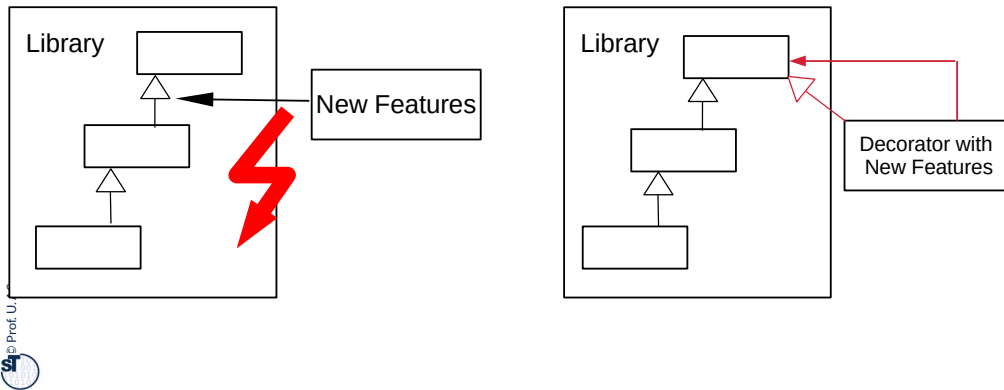
- ▶ The “sibling” of Composite





## Problem

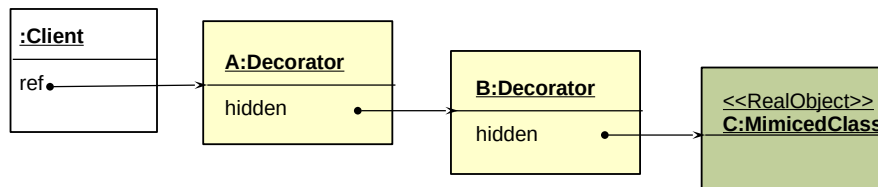
- ▶ How to extend an inheritance hierarchy of a library that was bought in binary form?
- ▶ How to avoid that an inheritance hierarchy becomes too deep?



## Snapshot of Decorator Pattern

34 Softwaretechnologie (ST)

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



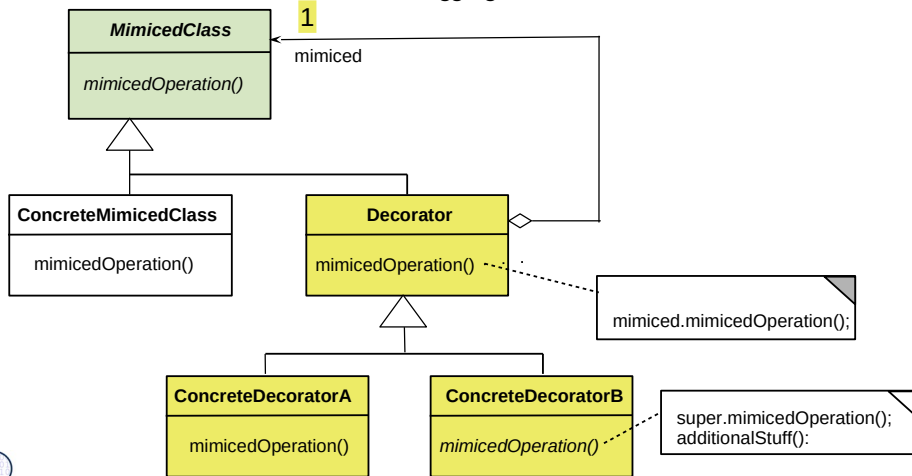
While traversing from the client to the Real Object, one passes all skin objects. Before recursing, an *prologue code (climbing down code)* is executed. After recursing, an *epilogue code (climbing up code)* is executed.

Prologue and epilogue code *wrap* the next object in the line.

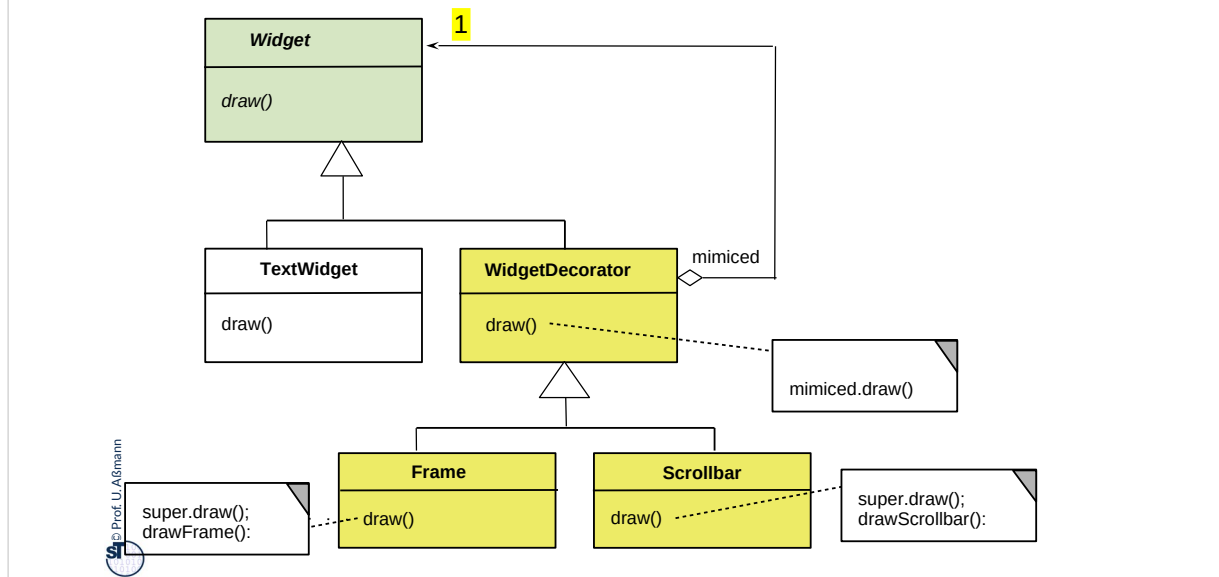
All skin objects belong logically to the real object, though being physically different.

## 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



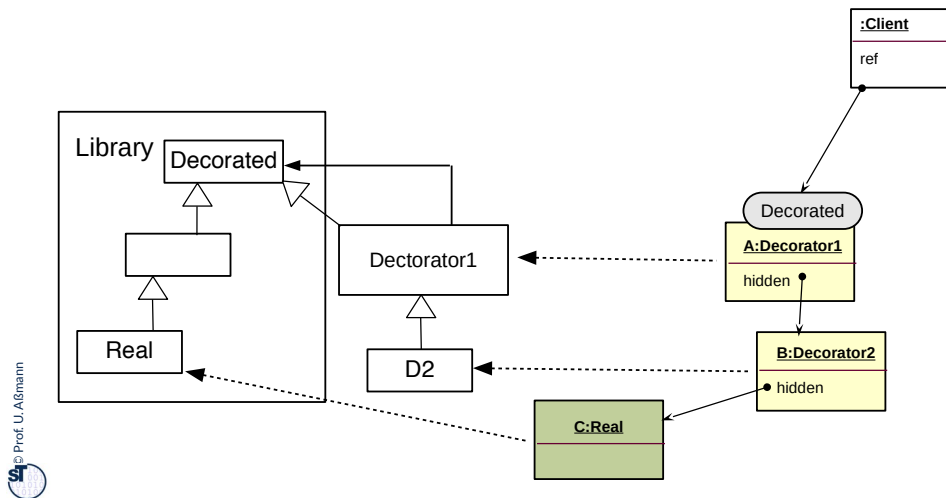
## Ex.: Decorator for Widgets



Decorator is frequently used in the implementation of widget hierarchies in GUI.

## Purpose Decorator

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



A combined class-object diagram shows how decorator class hierarchies determine the form of the skin object lists.



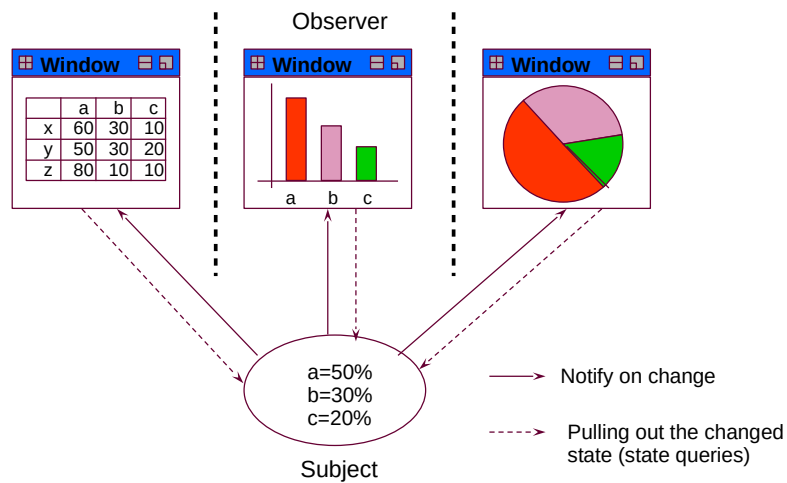
## 24.2.3 Different Kinds of Publish/Subscribe Patterns – (Event Bridge)

- ▶ 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 to Subjects // Notification of event // Source of event (subject) // Data to be transferred // Relation of Subject and Observer
- ▶ Therefore, Observer exists in several variants (push, pull, Callback, EventBus, ChannelBus)



Publish/Subscribe Patterns describe dynamic communications. For modern applications, they are indispensable.

## Overview

40 Softwaretechnologie (ST)

Variant of Pattern "Observer"		
Push		Data is flowing with the call to "update"
	Callback	1 observer
	Observer	n observer
Pull		Data is pulled on demand
	Callback	1 observer
	Observer	n observer

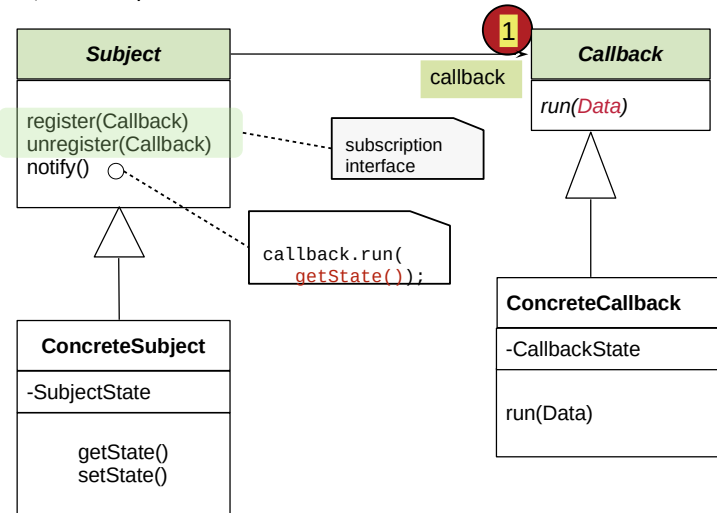
- ▶ A **callback** is a variant of the observer pattern with **one** observer
- ▶ With the "pull" variants, the Observer pulls the new state from the Subject
- ▶ With the "push" variants, the Subject pushes the state to the Observer

pull-Observer, the "normal" Observer from the GOF book, combines the registration of several observers with the pull of the changed state of the Subject. Therefore, it is the most complicated pattern of the 4 Publish/Subscribe Patterns discussed here.



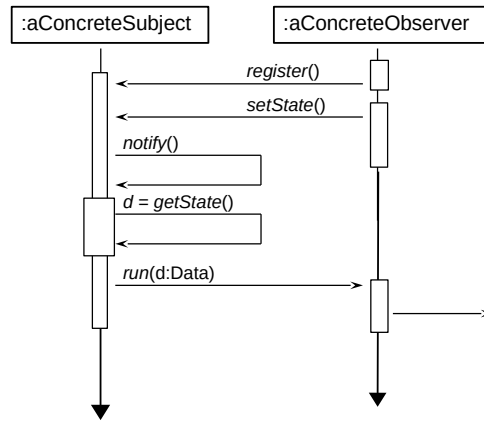
### 24.2.3.1 Publish/Subscribe with 1 Observer: Callback

- ▶ **Callbacks** have only one observer, which is not known statically, but registered dynamically, at run time
- ▶ A (**push-**)**Callback** pushes its data with the call to run



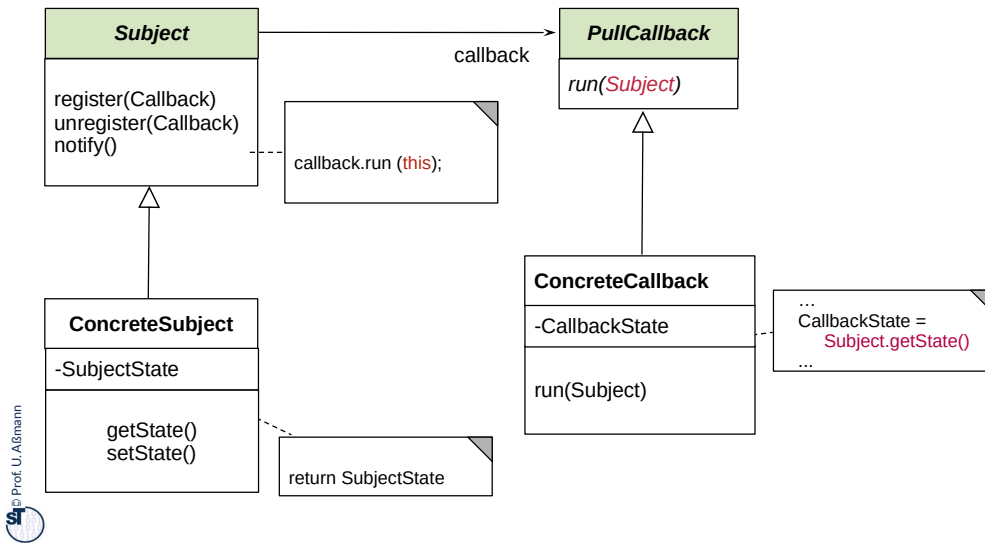
## Sequence Diagram push-Callback

- ▶ `run()` directly transfers Data to Observer (push)



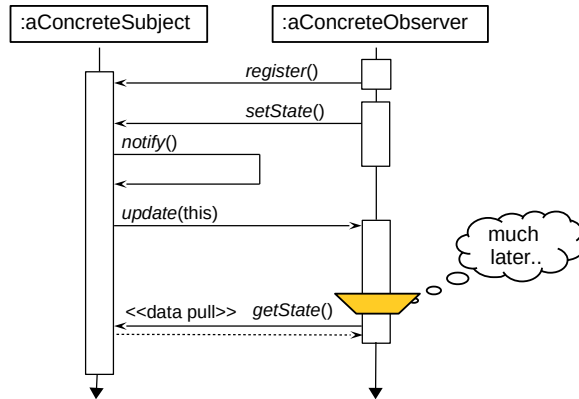
## 24.2.3.2 Structure pull-Callback (Delayed Data Transfer)

- ▶ A pull-Callback pushes the Subject to the Callback to later pull the data
- ▶ Responsibility for pull lies with the Callback; Subject is passed as argument



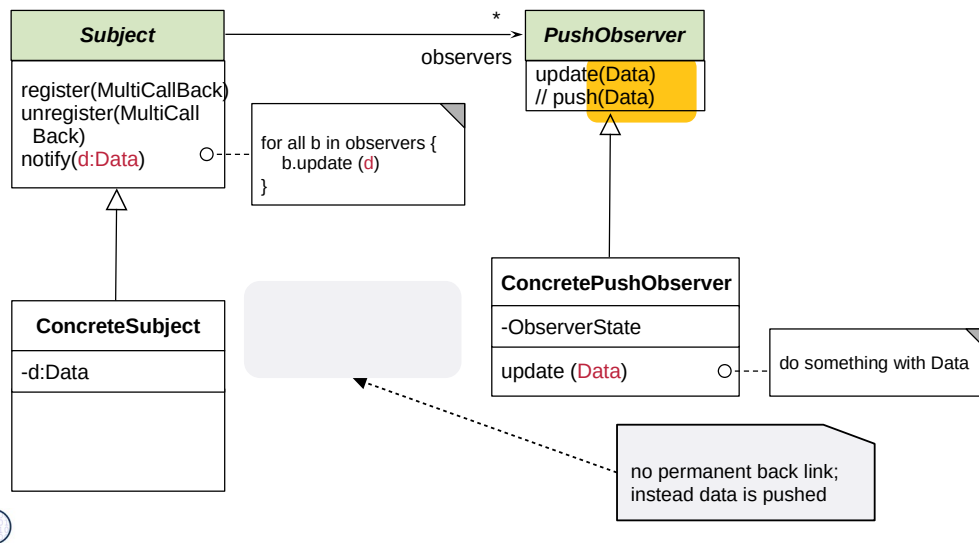
## Sequence Diagram pull-Callback

- ▶ Update() does not transfer data, only an event (anonymous communication possible)
  - Observer pulls data out itself with getState()
  - Lazy processing (on-demand processing)



### 24.2.3.3 Structure push-Observer (Immediate Data Transfer)

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

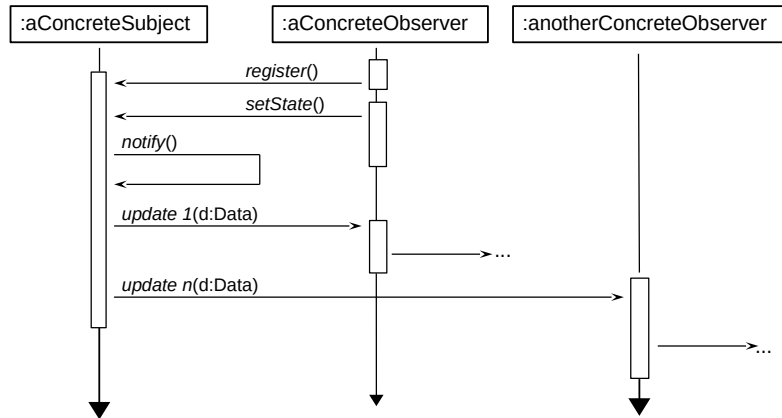


If the amount of data that is to be transferred from the subject to the observers is small, pushing the data with the update is no problem.

However, when the data is huge, its transport might be done in vain. Then, it is better to let the observer decide when to pull the data. This leads to the pull-Observer.

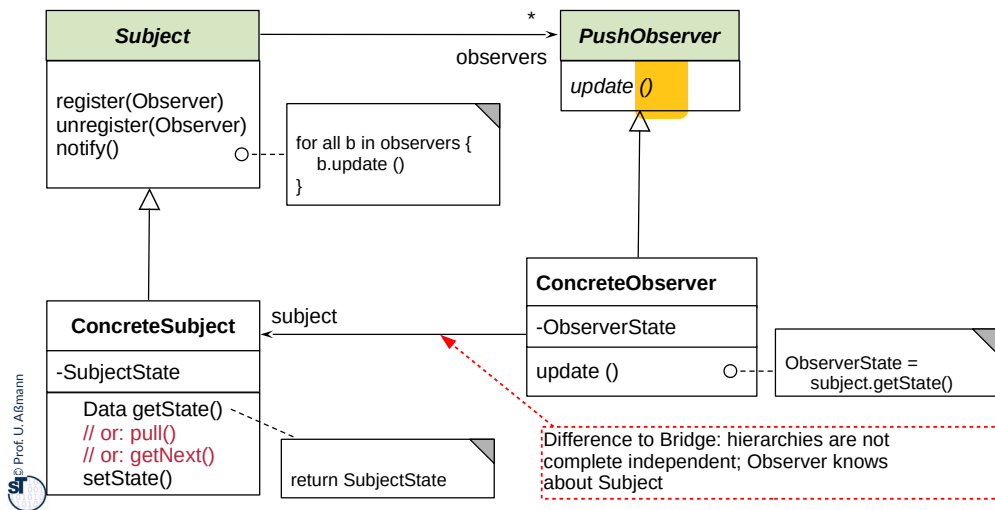
## Sequence Diagram push-Observer

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



### 24.2.3.4 Pull-Observer (Delayed Data Transfer, The Gamma Variant, 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

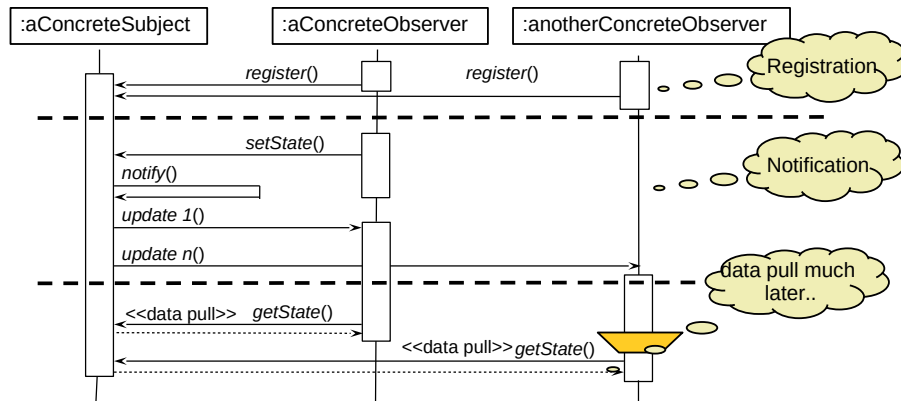


The Observer from GOF can also be called EventBridge, because it resembles a Bridge on which update events are transferred.

If the observer pulls data several times, the pattern becomes very similar to Iterator.

## Sequence Diagram pull-Observer

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







## 24.2.4. Visitor (VisitingAlgorithm)

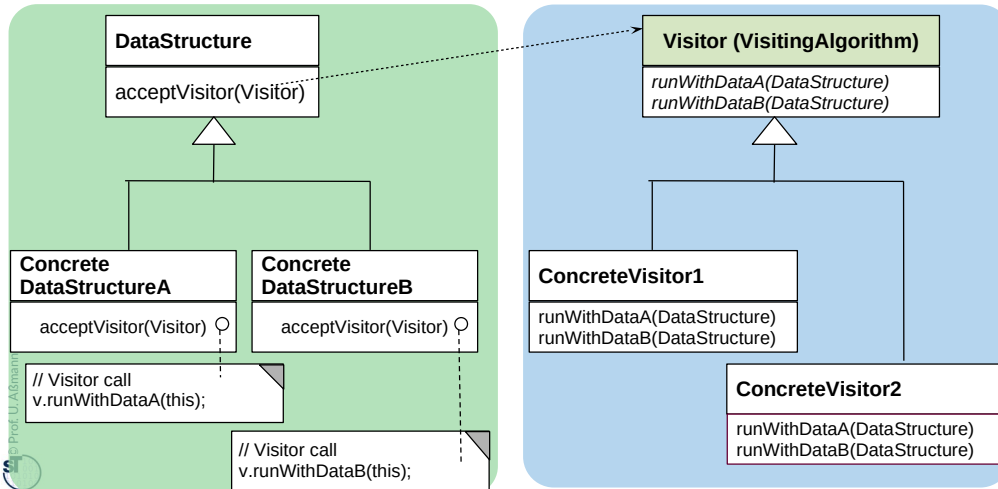
Visitor provides an extensible family of algorithms on a data structure  
Powerful pattern for modeling Materials and their Commands



## Visitor (VisitingAlgorithm)

50 Softwaretechnologie (ST)

- ▶ Implementation of *complex object* with a 2-dimensional structure
  - First dispatch on dimension 1 (data structure), then on dimension 2 (algorithm)
  - The Visitor has a lot of Command methods
- ▶ Beauty: visiting algorithms can be added without touching the DataStructure



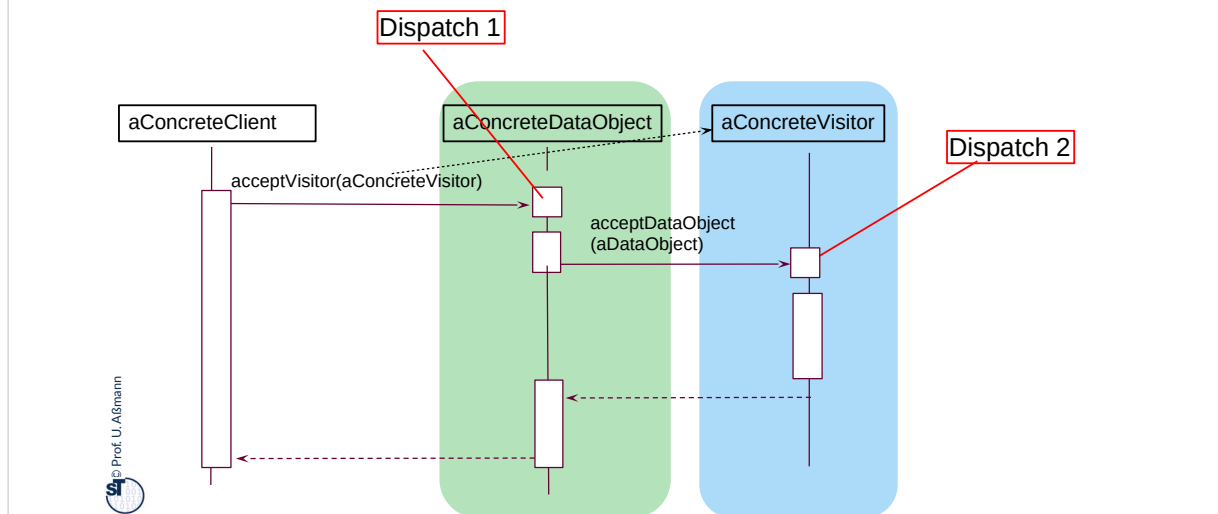
Visitor is a somewhat obscure pattern. The usual principle of object-orientation tells us to *encapsulate algorithms with data*. The only reason to split off the visiting algorithms from the data is to be able to extend the set of algorithms over time, or if this extension cannot be foreseen.

For instance, if a new customer wants a new feature of a product, a Visitor hierarchy can be extended very easily, without retesting the DataStructure hierarchy nor the old visiting algorithms, because they all stay the same. Therefore, Visitor is an evolution pattern.

## Sequence Diagram Visitor

51 Softwaretechnologie (ST)

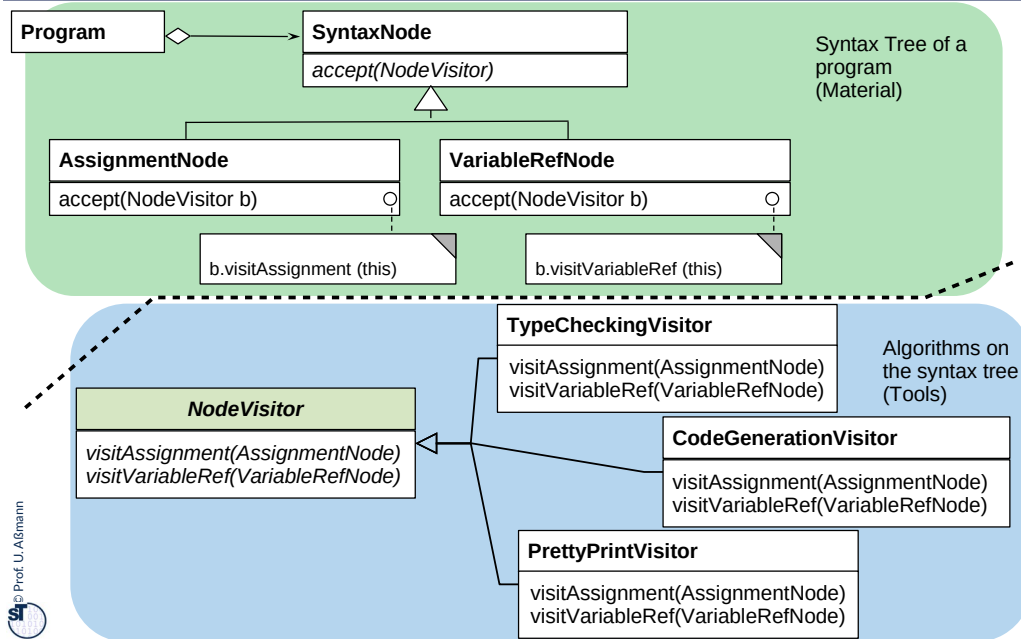
- ▶ First dispatch on data, then on visiting algorithm



“Double dispatch” is offered in some programming languages as a language concept (“multi-methods”). In such a programming language, the Visitor pattern is built in.

## Intermediate Data of a Compiler: Working on Syntax Trees of Programs with Visitors

52 Softwaretechnologie (ST)



Syntax Trees are a central data structure in compilers, refactoring tools, integrated development environments like Eclipse. Syntax trees represent the programs you write. Software tools work on syntax trees.

Therefore, Visitor appears in all tools for software development.

Usually, also all forms of documents (text documents, slides, spreadsheets, pdf documents, etc.) can be represented by syntax trees. Thus, syntax trees are also a central data structure in Word-like editors, Powerpoint-like slide editors, Spreadsheet tools, Acrobat-readers....

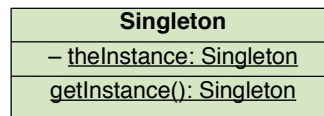


## 24.3) Patterns for Glue - Bridging Architectural Mismatch

Glue Pattern	# Run-time objects	Key feature
Singleton	1	Only one object per class
Adapter	2	Adapting interfaces and protocols that do not fit
Facade	1+*	Hiding a subsystem
Class Adapter	1	Integrating the adapter into the adaptee
Proxy (Appendix)	2	1-decorator

## 24.3.1 Singleton (dt.: Einzelinstanz)

- ▶ Problem: Store the global state of an application
  - Ensure that only one object exists of a class



The usual constructor is invisible

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



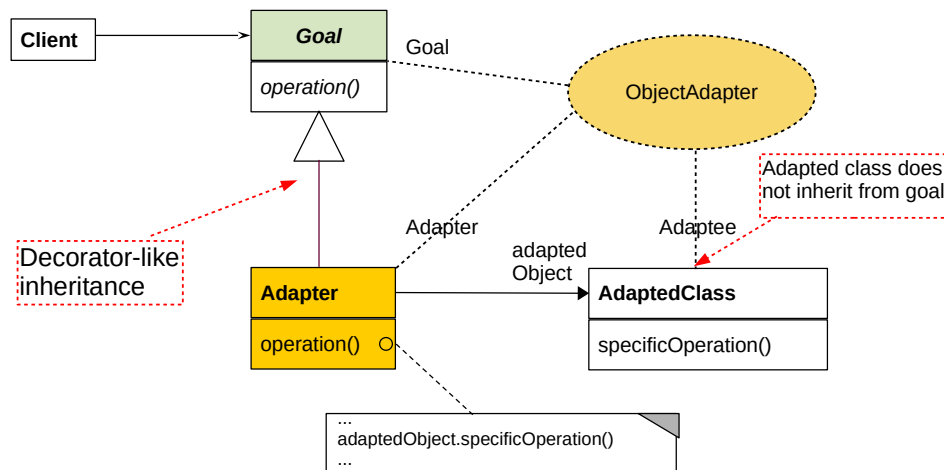


## 24.3.2 Adapter



## Object Adapter

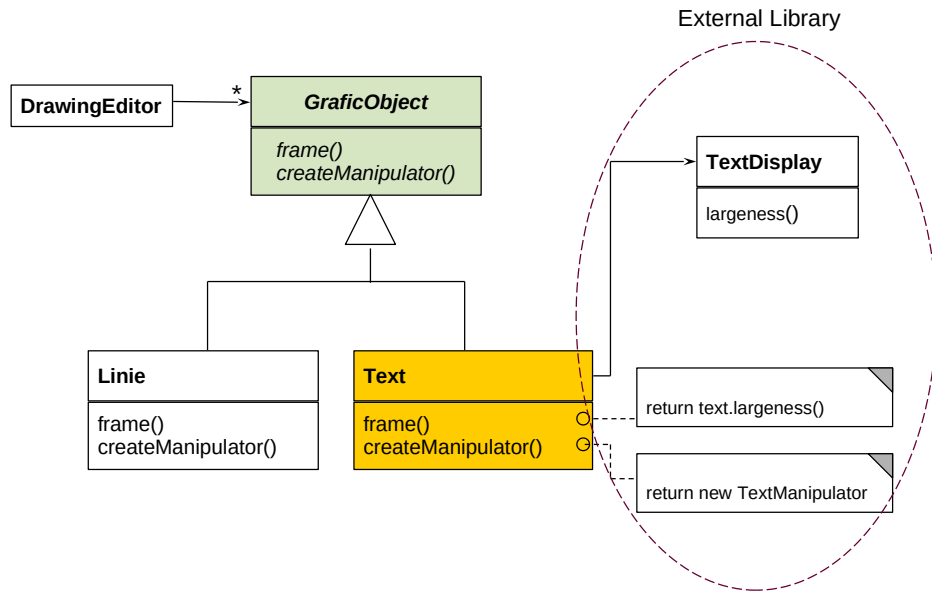
- ▶ An **object adapter** is a kind of a proxy mapping one interface, protocol, or data format to another



Adapter is one of the most useful patterns when an old system should be coupled to new classes. Adapter adapt interfaces, data formats and protocols.

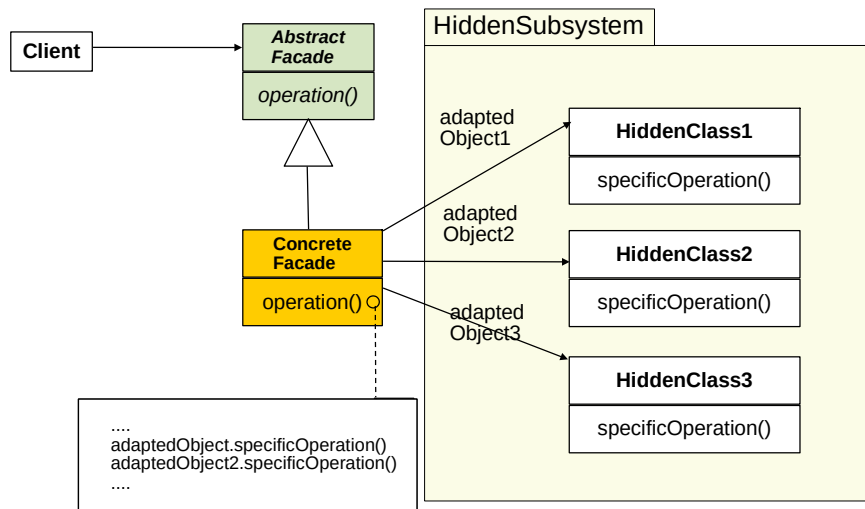


## Example: Use of an External Class Library For Texts



### 24.3.3 Facade Hides a Subsystem

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

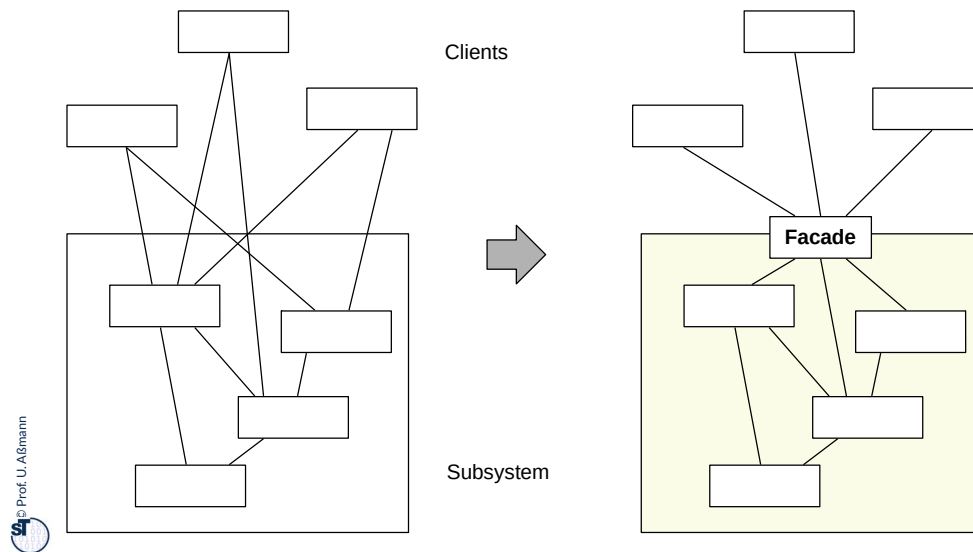


Facades are very much linked with layers, an architectural concept (architectural style).

## Refactoring a Legacy System Towards a Facade

59 Softwaretechnologie (ST)

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



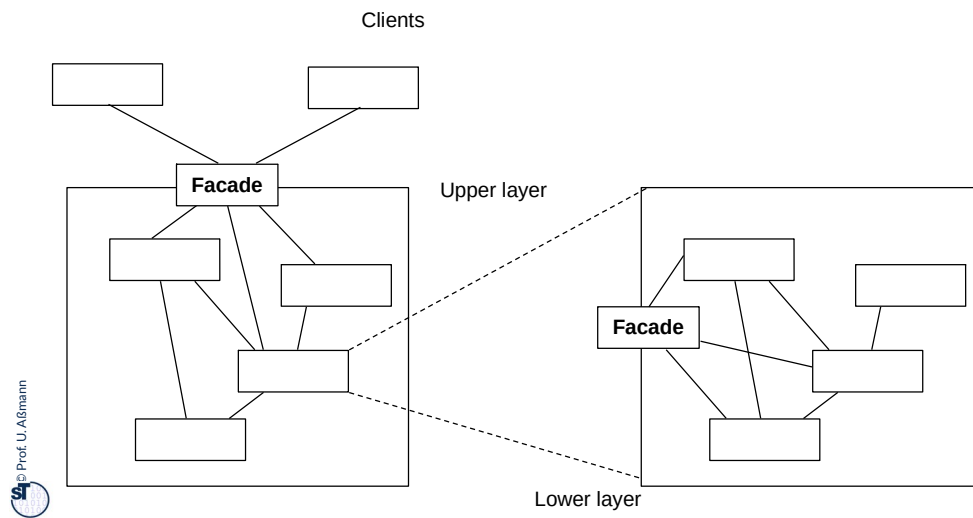
One of the most important restructuring operations in old systems is to introduce modules, subsystems, or packages by encapsulating them by a facade object.

Then, the subsystem can be exchanged behind the facade, and no client will be disturbed.

In the situation of the left side, this is not possible, because too many explicit dependencies exist from the subsystem to the clients.

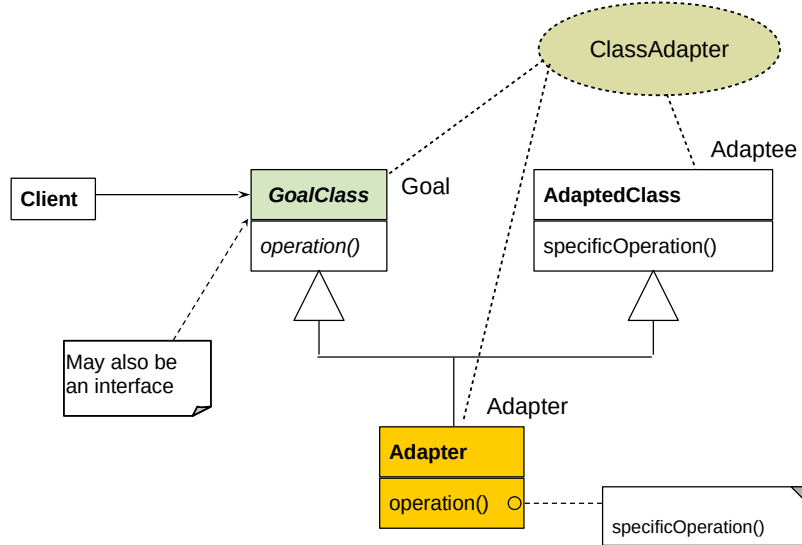
# The Layer Pattern

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



## 24.3.4 Class Adapter

- ▶ Instead of delegation, class adapters use multiple inheritance





## 24.4 Other Patterns



## What is discussed elsewhere...

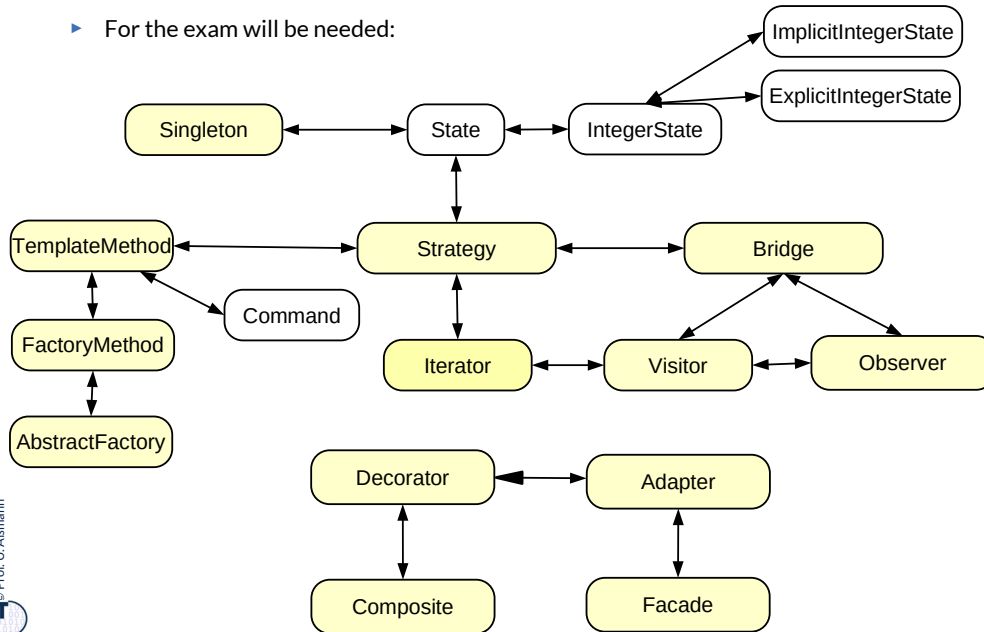
- ▶ Iterator, Sink, and Channel
- ▶ Composite
- ▶ TemplateMethod, FactoryMethod
- ▶ Command

### Part III:

- ▶ 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:



The yellow marked patterns will be important for the exam.

Try to find out a “beauty feature” for every pattern: Why does the pattern describe a beautiful solution for a standard ugly problem?



### 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





## 24.5 Design Patterns in a Larger Library



- ▶ AWT/Swing is the GUI part of the Java class library
  - Uniform window library for many platforms (portable)
- ▶ Employed patterns
  - Pull-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

## Why is the Frauenkirche Beautiful?

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



## 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

## The End

- ▶ Course "Design patterns and frameworks", WS, contains more material.
- ▶ © Pictures originally covered by the teaching license from the CD "Design Patterns" of AWL; Uwe Aßmann, Heinrich Hussmann, Walter F. Tichy, Universität Karlsruhe, Germany, used by permission



## Appendix





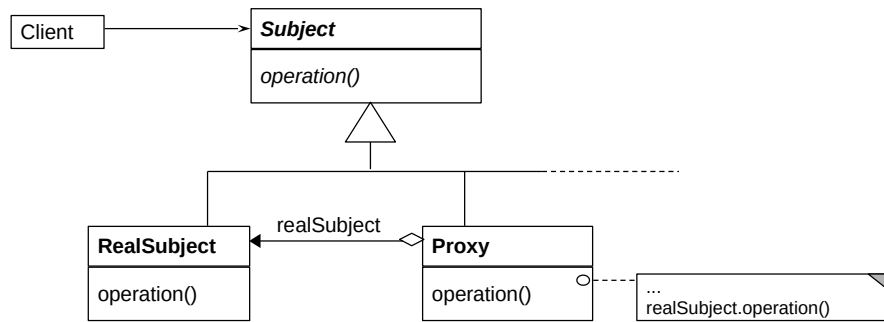
## 24.A.1 Proxy



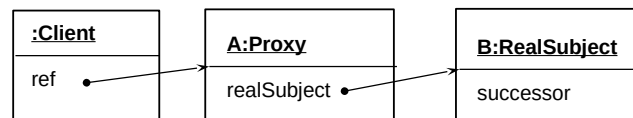


# Proxy

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



Object Structure:



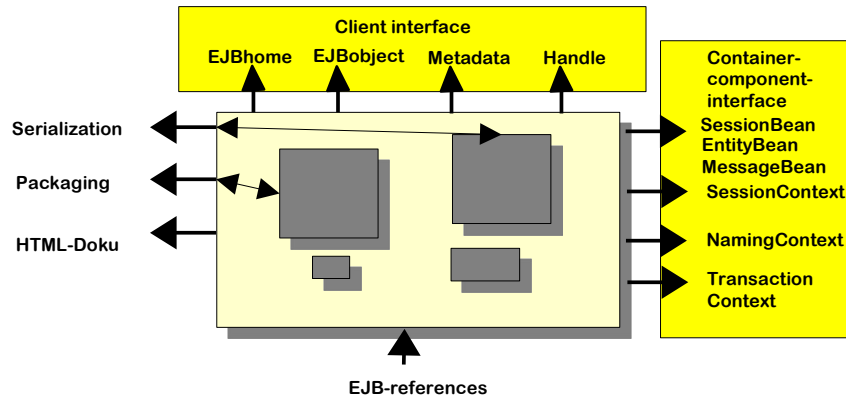
- ▶ The proxy object is a representant of an object
  - The Proxy is similar to Decorator, but it is not derived from ObjectReursion
  - It has a direct pointer to the sister class, *not* to the superclass
  - It may collect all references to the represented object (shadows it). Then, it is a facade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

## Proxy Variants

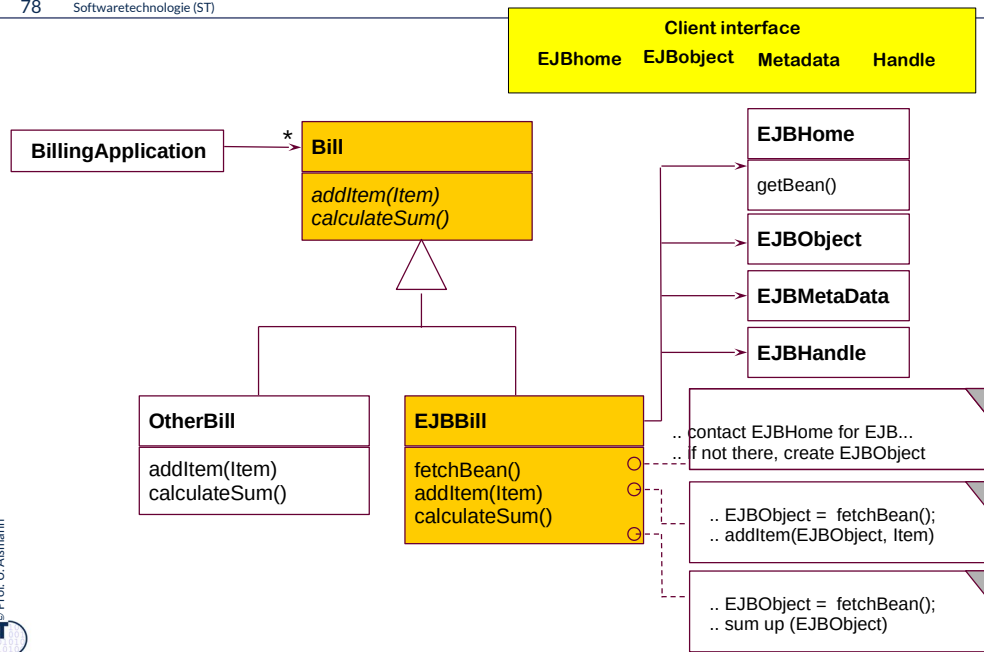
- ▶ **Filter proxy** (smart reference):
  - executes additional actions, when the object is accessed
- ▶ **Protocol proxy**:
  - Counts references (reference-counting garbage collection)
  - Or implements a synchronization protocol (e.g., reader/writer protocols)
- ▶ **Indirection proxy** (facade proxy):
  - 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 and Facades for COTS

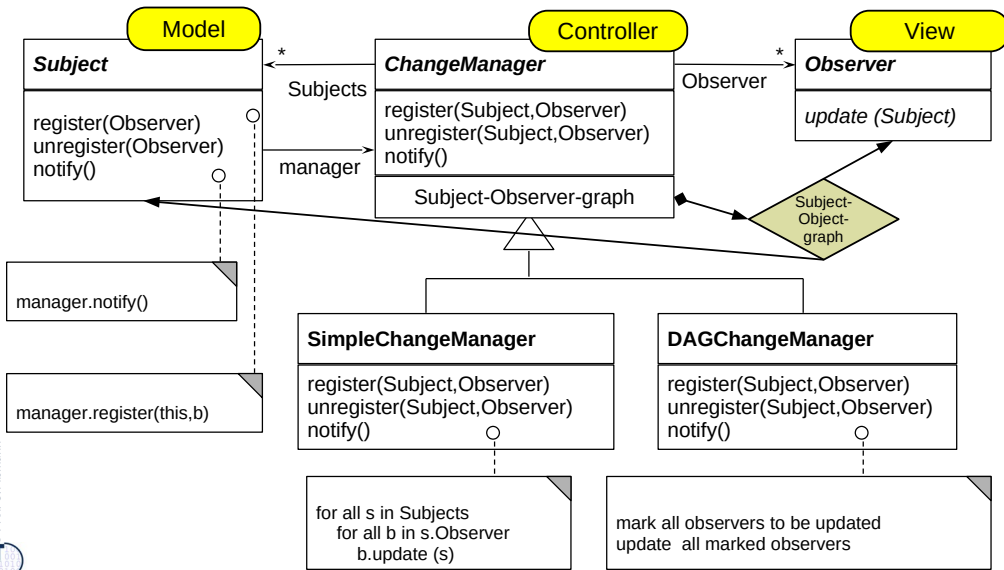
- ▶ Adapters and Facades are often used to adapt components-off-the-shelf (COTS) to applications
- ▶ For instance, an EJB-adapter allows for reuse of an Enterprise Java Bean in an application
- ▶ -> course Component-Based Software Engineering (SoSe)



# EJB Adapter

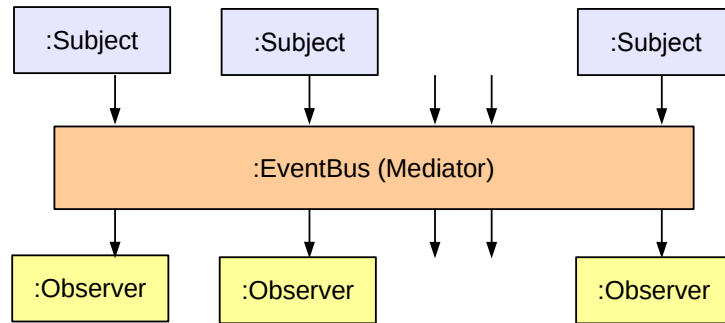


## 24.A.2 Observer with ChangeManager (EventBus)



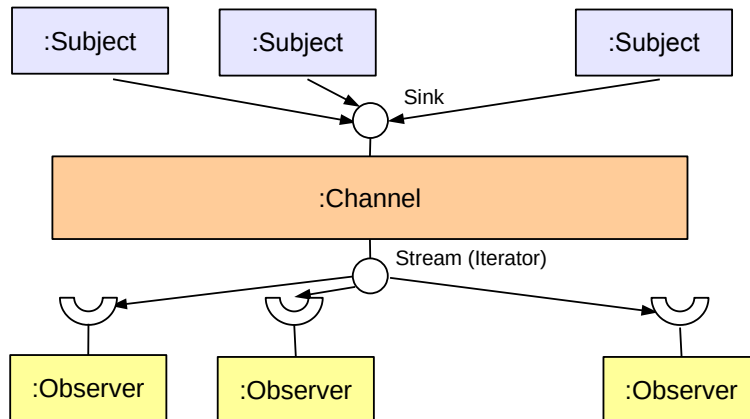
## Observer with ChangeManager is also Called Event-Bus

- ▶ 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



## A Variant of EventBus is the n:m-Channel

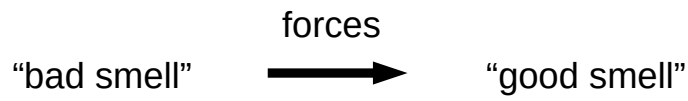
- ▶ push-Subjects and pull-Observers can be connected by Channel, to emphasize the continuous pushing and pulling
- ▶ Then Subjects write the Sink of the Channel and Observers pull the Stream of the Channel
  - Channel has a buffer





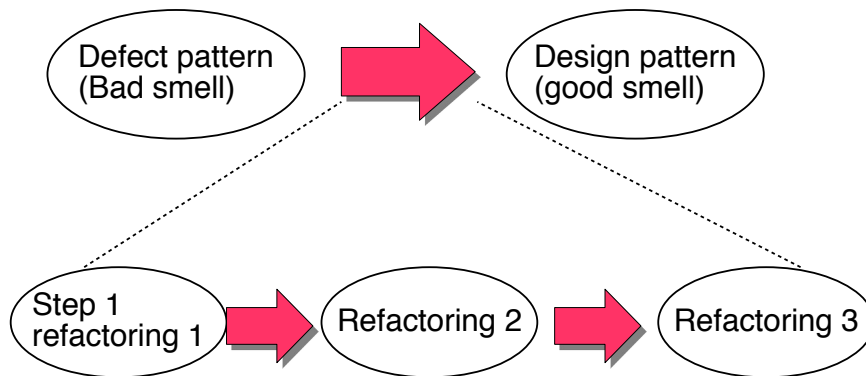
## What Does a Design Pattern Contain?

- ▶ 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

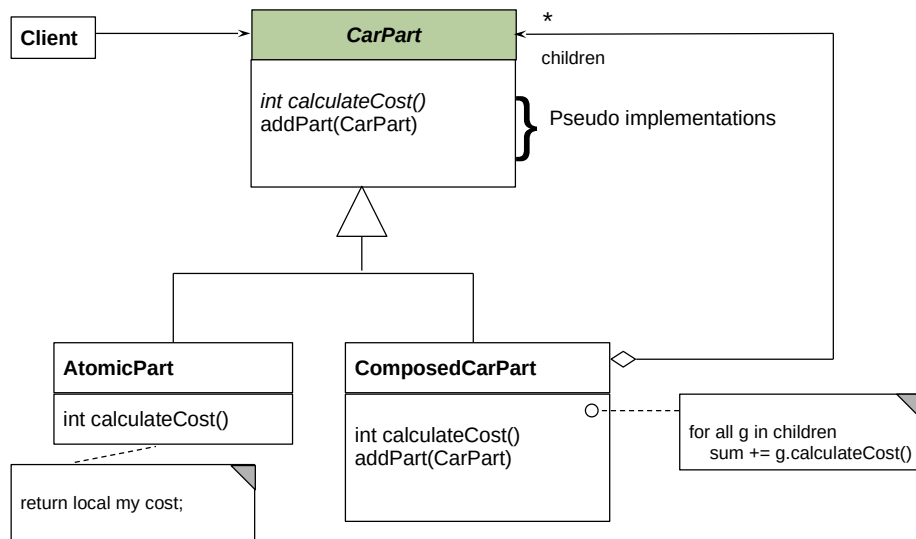


## Structure for Design Pattern Description (GOF Form)

- ▶ 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

## A.2 Example for Composite: PieceLists in Cars

- ▶ Big technical objects can have thousands of parts (piecelists, or part lists)



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

class Screw extends AtomicCarPart {
    int myCost = 10;
}

class SteeringWheel extends AtomicCarPart {
    int myCost = 200;
}

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

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

Piece lists (Stücklisten) are usually hierarchic, i.e., should be called “piece trees” or “Stückbäume”.

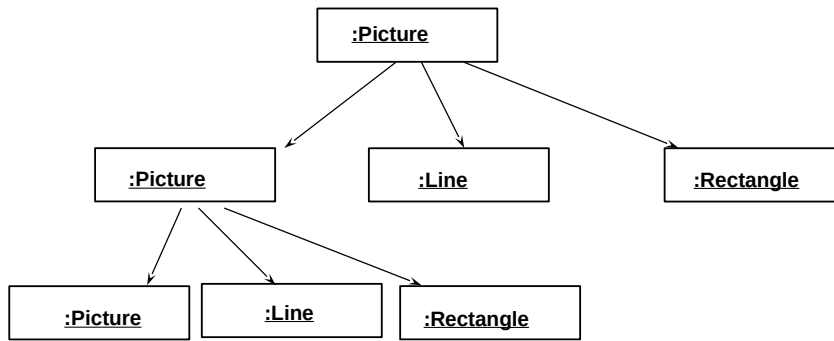
Technical objects can have thousands of parts, and Composite can manage all parts under a common interface of the “Component” abstract class.

Examples:

- Any product produced by a factory
- Any business object in a business software (ERP software): Bestellungen, Formulare, Rechnungen, Kataloge, ...

## Composite for Part/Whole Hierarchies (Structured Piece Lists)

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