2. Simple Patterns for Variability

Prof. Dr. U. Aßmann

Chair for Software Engineering

Faculty of Informatics

Dresden University of Technology

WS 11-0.1, 10/8/11

1) Basic Template-And-Hook Patterns
2) Faceted Objects with Bridges
3) Layered Objects
4) Dimensional Systems
5) Layered Systems

Literature (To Be Read)

- ► V. Caisin. Creational Patterns. Paper in Design Pattern seminar, IDA, 2001. Available at home page.
- ► GOF, Chapters on Creational and Structural Patterns
- ► Another good book:
- ► 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

Secondary Literature

- ► D. Karlsson: Metapatterns. Seminar Design Patterns, IDA, Linköpings universitet, 2001.
- ► W. Pree. Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1995. Unfortunately out of print.
- ► W. Zimmer. Relationships Between Design Patterns. Pattern Languages of Programming (PLOP) 1995.
- ► Uta Priss. Faceted Information Representation. Electronic Transactions in Artificial Intelligence (ETAI). 2000(4):21-33.
- ► R. Prieto-Diaz, P. Freeman. Classifying Software for Reusability. IEEE Software, Jan 1987. Prieto-Diaz has introduced facet-based classifications in software engineering. Surf also on citeseer for facets.
- ► Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. The GenVoca Model of Software-System Generation. IEEE Software, 11 (5), Sept. 1994, pp 89—94.

Goal

- ► Understanding Templates and Hooks
  - ▪ Template Method vs Template Class
  - ▪ Dimensional Class Hierarchy
- ► Understanding why Bridges implement faceted objects
- ► Understanding layered systems

2.1) Basic Template and Hook Patterns

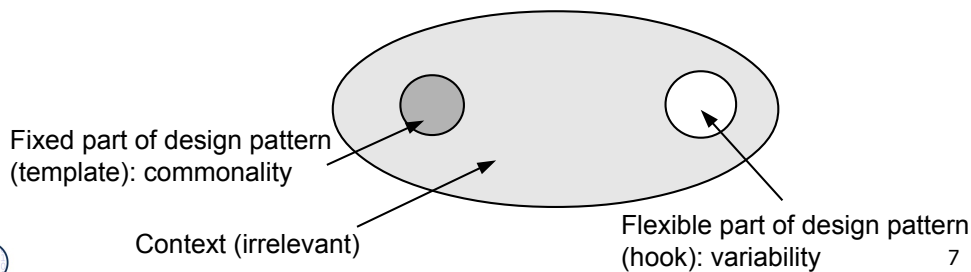# The Problem

► How to produce several products from one code base?

► Design patterns often center around

- Things that are common to several applications
  - *Commonalities* lead to *frameworks* or *product lines*
- Things that are different from application to application
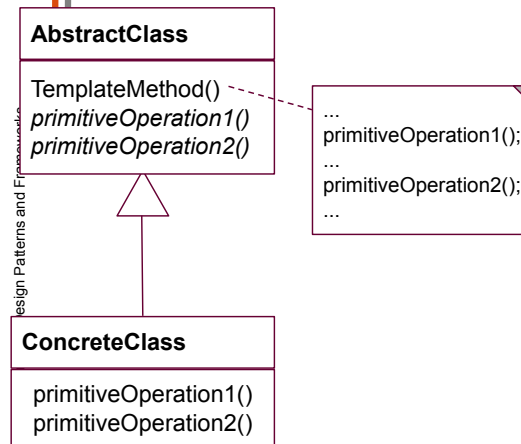 - *Variabilities* to *products* of a product line

Office Backbone

Word processor   Slide Program   Spreadsheet
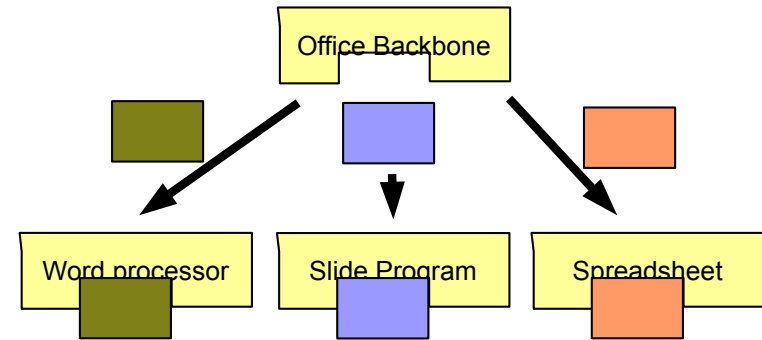
Pree's Template&Hook Conceptual Pattern

► Pree invented a (*template-and-hook T&H)* concept for the communality/variability knowledge in design patterns [Pree] [Karlsson]

► *Templates contain skeleton code*
- Common for the entire product line

► *Hooks* are placeholders for the instance-specific code
- Only for one product. Also called *slots, hotspots*

Fixed part of design pattern (template): commonality

Context (irrelevant)

Flexible part of design pattern (hook): variability

TemplateMethod Design Pattern

► Define the skeleton of an algorithm (*template method*)
- The template method is concrete

► Delegate parts to abstract *hook methods (slot methods)* that are filled by subclasses
- Requires inheritance

► Implements template and hook with the same class, but different methods

AbstractClass

TemplateMethod()
primitiveOperation1()
primitiveOperation2()

...
primitiveOperation1();
...
primitiveOperation2();
...

ConcreteClass

primitiveOperation1()
primitiveOperation2()

Ex.: Actors and Genres

- ► Binding an Actor's hook to be a ShakespeareActor or a TV actor

Actor

play()
recite()
dance()

```
...
recite();
...
dance();
...
```

ShakespeareActor

recite()
dance()

TVActor

recite()
dance()

9
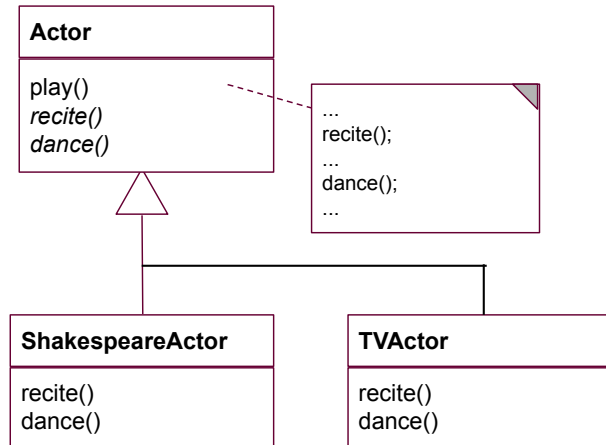
Variability with TemplateMethod

- ► Allows for varying behavior
 - Separate invariant from variant parts of an algorithm
  - TemplateMethod differs slighly from polymorphism: for a polymorphic method, one needs several subclasses
- ► Binding the hook (slot) 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

Binding the hooks with hook values (method implementations)

Template method

Hook method

10
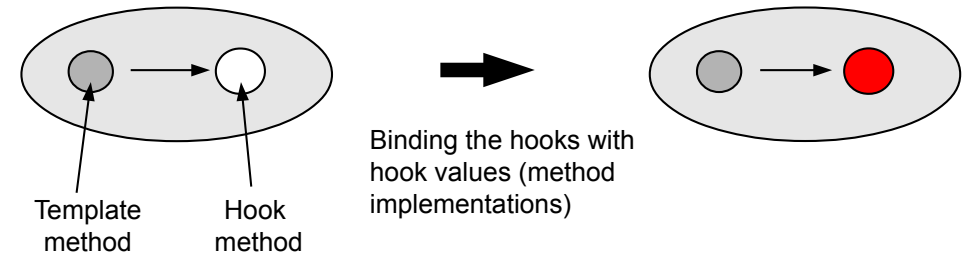
Consequences

- ► The design pattern TemplateMethod realizes the conceptual pattern T&H on the level of Methods
  - TemplateMethod – HookMethod
- ► Basis for design patterns:
  - FactoryMethod
  - TemplateClass

11

Variability vs Extension

- ► The T&H concept occurs in two basic variants
  - Binding of hooks or extension of hooks: we speak of a *slot*, if a hook can be bound only once (unextensible hook, only bindable)
  - Hooks can be extensible
  - Extension patterns are treated later

Binding a hook (slot) with a hook value

Extending a hook with another hook value

12
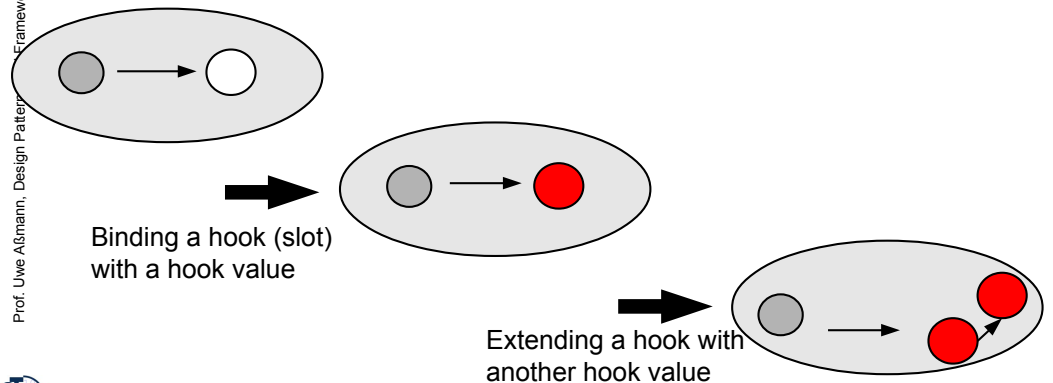
2.1.1 Template Method and Template Class

What Happens If We Reify the Hook Method?

- ▶ Methods can be reified, i.e., represented as objects
- ▶ In the TemplateMethod, the hook method can be split out of the class and put into a separate object
- ▶ Reification is done by another basic pattern, the Objectifier [Zimmer]

The Objectifier Pattern

- ▶ The pattern talks about basic polymorphism with objects (delegation)
  - ▪ Combined with an abstract class and abstract method
  - ▪ Clients call objects polymorphically

Client	
applicationMethod()	

reference → Objectifier

Objectifier	
reifiedMethod()	

reference.reifiedMethod();

ConcreteObjectifierA		ConcreteObjectifierB	
reifiedMethod()		reifiedMethod()	

Ex. Different Students

- ▶ When preparing an exam, students may use different learning styles
- ▶ Instead of a method learn(), an objectified method, a LearningStyle class, can be used

Student	
prepareExam()	

learningStyle → LearningStyle

LearningStyle	
perform()	

learningStyle.perform();

OnDemandLearning		ExhaustiveLearning	
perform()		perform()	
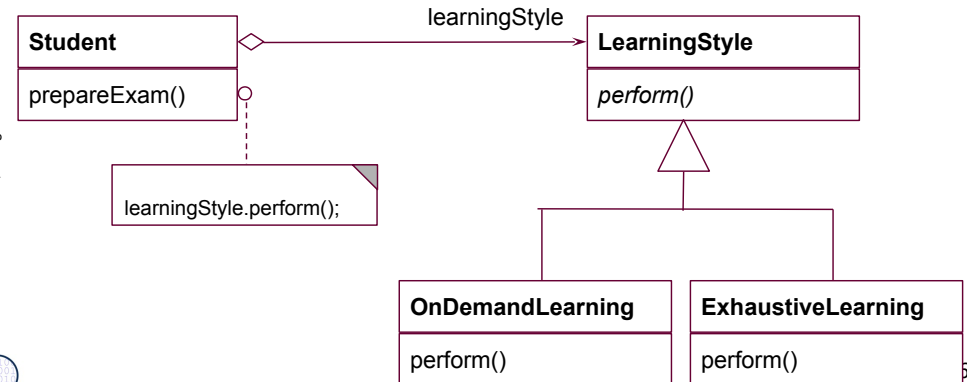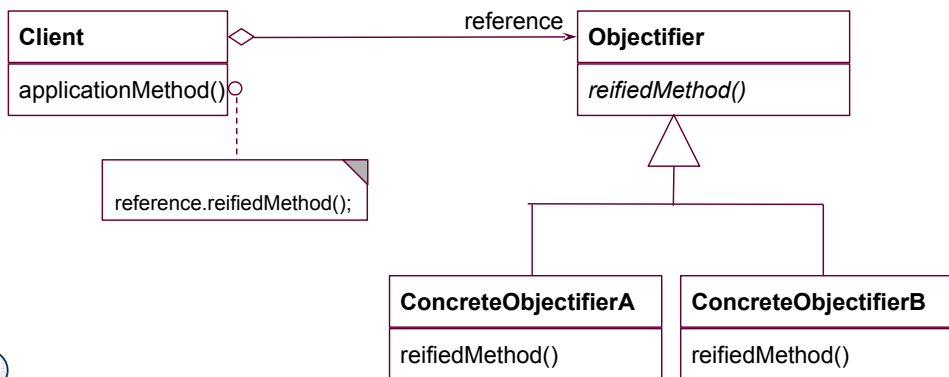
T&H on the Level of Classes

- ▶ With the Objectifier, we can build now Template&Hook classes
  - ▪ Additional roles for some classes
    - · The template role
    - · The hook role
- ▶ Resulting patterns:
  - ▪ Template Class
  - ▪ Generic Template Class
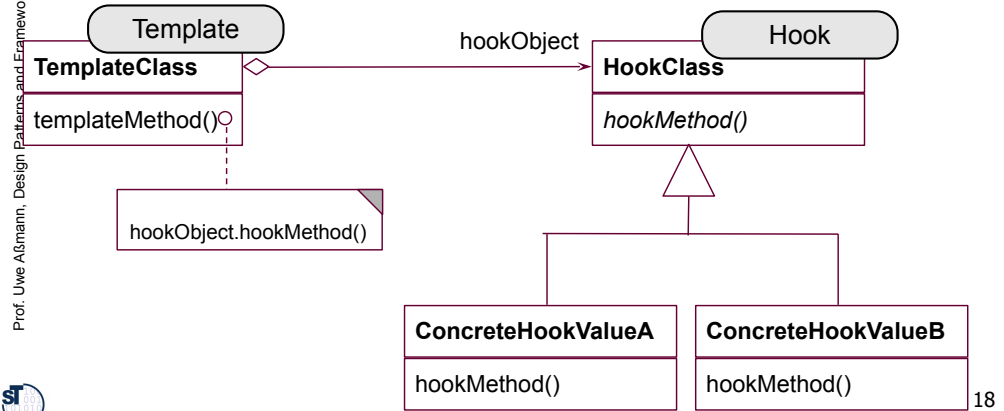 - ▪ Dimensional Class Hierarchies for variability with parallel class hierarchies
    - · Implementation of facets
    - · Bridge, Visitor

Template Class

- ▶ Is combined from TemplateMethod and Objectifier
  - ▪ We explicitly fix a template class in the Objectifier
  - ▪ The template method and the hook method are found in different classes

---

Template Class

- ▶ 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, Prototype, State, Strategy, Visitor.

Actors and Genres as Template Class

- ▶ Consistent exchange of recitation and dance behavior possible
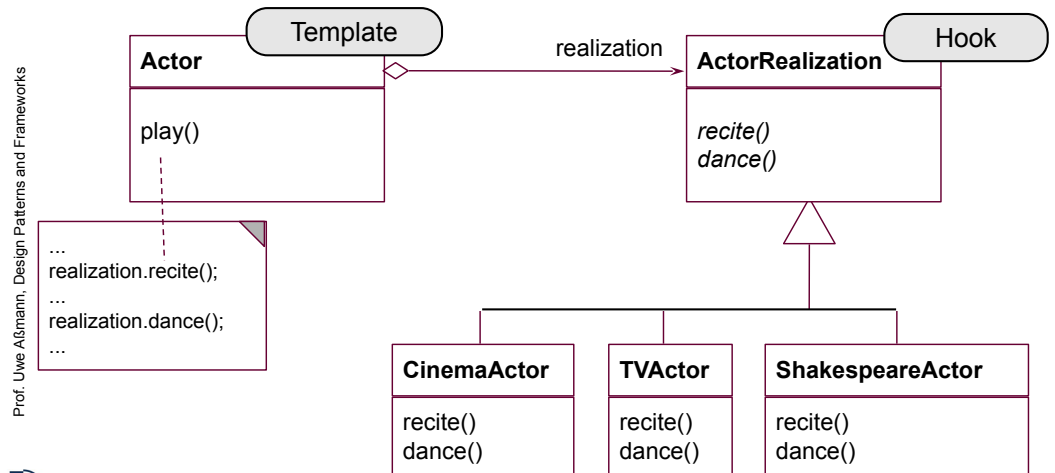
Variability with TemplateClass

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

Hook (slot) methods

Hook Class

Template Class

Template methods

Binding the hooks with hook values (method implementations)

The GOF-Pattern Strategy

▶ The GOF-Strategy hands out the roles *client and algorithm*

Client

Client
contextInterface()

strategy

Algorithm

Strategy
algorithmInterface()

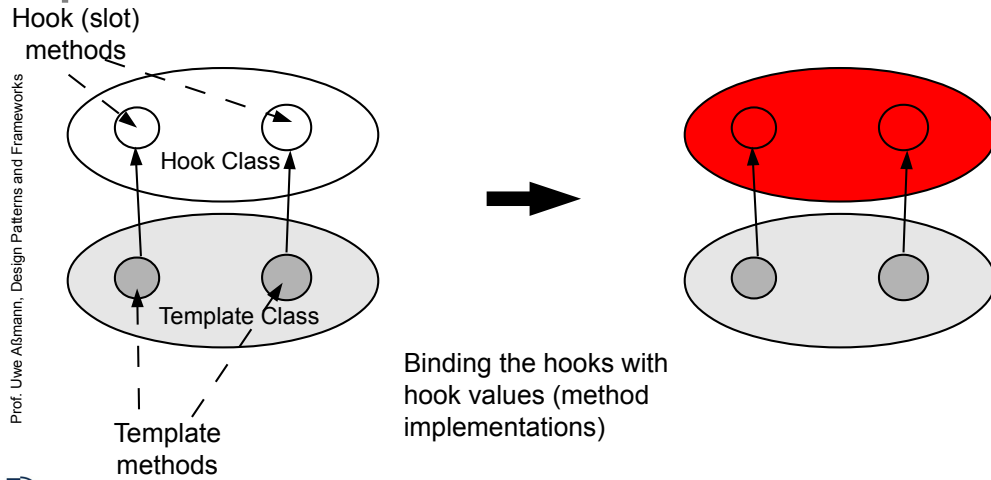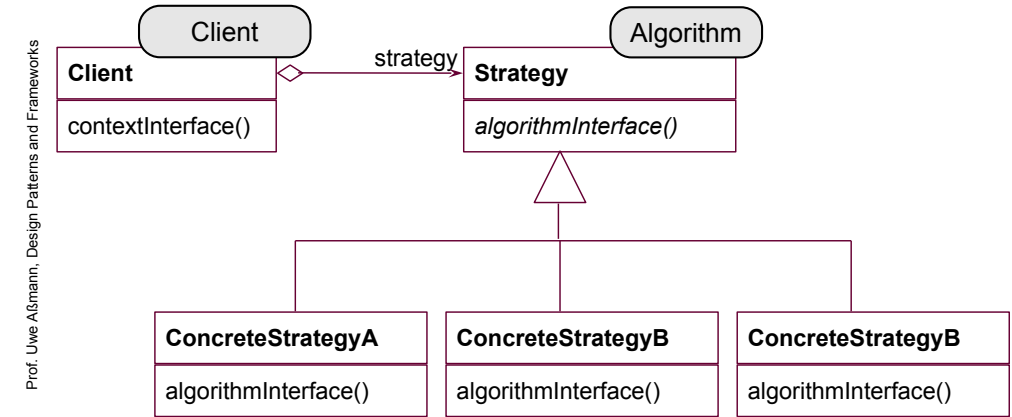| **ConcreteStrategyA** |
|---|
| algorithmInterface() |

ConcreteStrategyB
algorithmInterface()

ConcreteStrategyB
algorithmInterface()

The GOF-Strategy Is Related To TemplateClass

▶ The GOF Strategy has the same structure as the Objectifier, but has a **different incentive**
- It is not for reifying methods, but for varying methods only

▶ TemplateClass also has a different incentive
- Hence, TemplateClass hands out other roles for the classes
- The client class is considered as a template class
- The strategy class as the hook class

Design Patterns may have the same structure and/or behavior, but can have a different incentive

Example for Strategy

▶ Encapsulate formatting algorithms

Client

TextApplication
traverse() repair()

formatter

Algorithm

Formatter
format()

formatter.format()

SimpleFormatter
format()

TeXFormatter
format()

ArrayFormatter
format()
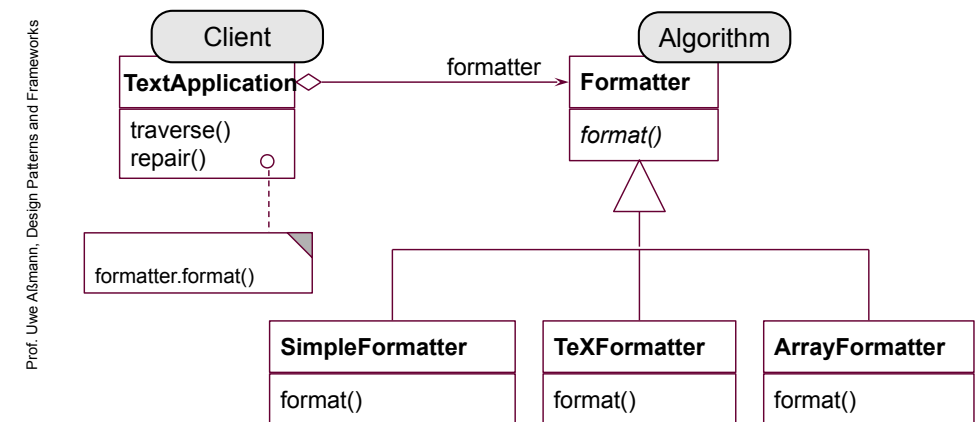
GenericTemplateClass

- ▶ In languages with generic classes (C++, Ada95, Java 1.5, C#, Sather, Cecil, Eiffel), TemplateClass can be realized with GenericTemplateClass
- ▶ The subclassing of the hook hierarchy is replaced by static generic expansion
  - ▪ Hence, more type safety, less runtime dispatch

T

TemplateClass

T hookObject;
templateMethod()○----- hookObject.hookMethod()

<<instantiates>>   <<instantiates>>
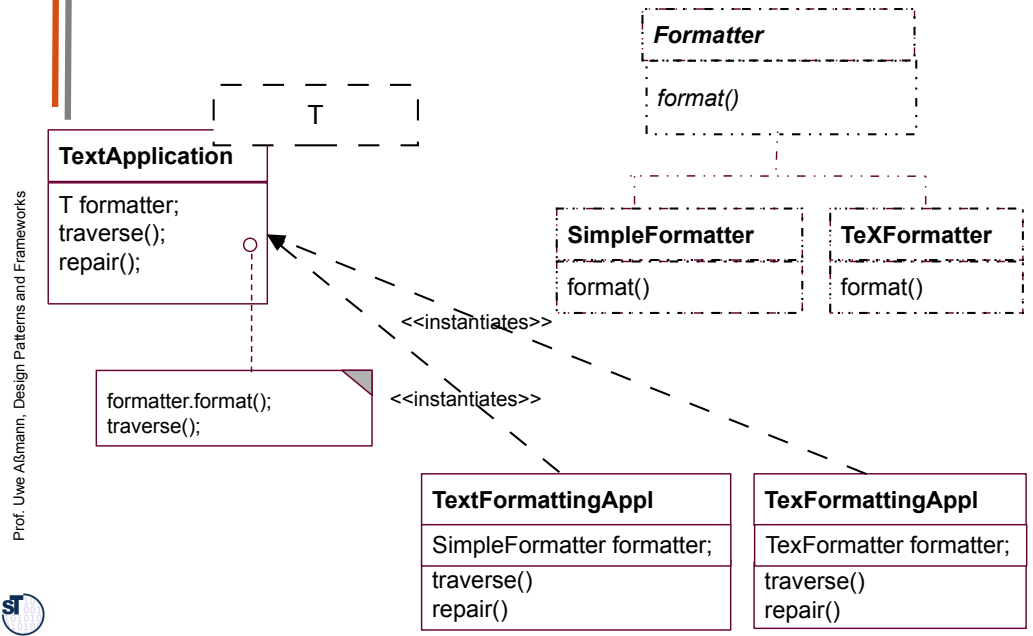
ConcreteStrategyA

HookA hookObject;

ConcreteStrategyB

HookB hookObject;

HookClass

hookMethod()

HookA

hookMethod()

HookB

hookMethod()

Generic Text Formatter

T

TextApplication

T formatter;
traverse();
repair();

○

formatter.format();
traverse();

<<instantiates>>

<<instantiates>>

Formatter

format()

SimpleFormatter

format()

TeXFormatter

format()

TextFormattingAppl

SimpleFormatter formatter;
traverse()
repair()

TexFormattingAppl

TexFormatter formatter;
traverse()
repair()

Further Work on Generic Template Parameterization

- ▶ See course CBSE
- ▶ GenVoca [Batory]
  - ▪ Generic template instantiation method for nested generics
  - ▪ Parameterization on many levels
  - ▪ Layered systems result
  - ▪ Realizable with nested C++ templates
  - ▪ See later
- ▶ Template Metaprogramming (www.boost.org)
  - ▪ Using template parameter for other purposes than hook classes

---

2.1.2 Dimensional Class Hierarchies and Bridge

Variability Pattern DimensionalClassHierarchies

Template

Hook

TemplateClass ─ hookObject ─→ *HookClass*

templateMethod()○ · · · · *hookMethod()*

hookObject.hookMethod()

MoreConcrete TemplateA
templateMethod()○

MoreConcrete TemplateB
templateMethod()○

Concrete HookClassA
hookMethod()

Concrete HookClassB
hookMethod()

// Implementation A
.. hookMethod();

// Implementation B
.. hookMethod();

29
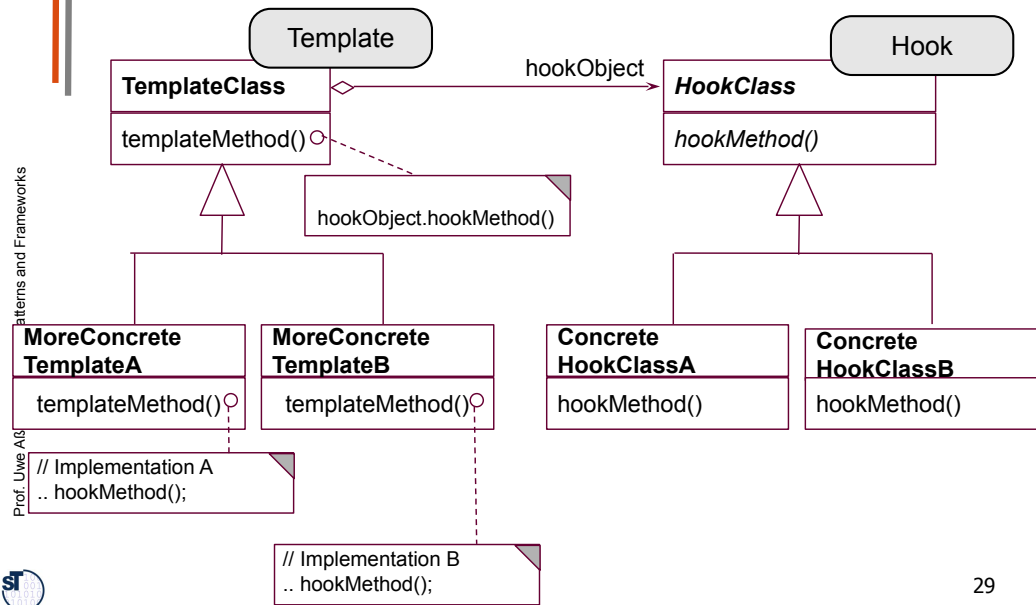
DimensionalClassHierarchies

► Vary also the template class in a class hierarchy
  - The sub-template classes can adapt the template algorithm
  - Still, the template method calls the hook methods and reuses its results
  - 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
► Upper and lower layer (dimension)
  - Template method (upper layer) calls hook methods (lower layer)
► Both hierarchies can be varied independently
  - Factoring (orthogonalization)
  - Reuse is increased
► Basis for patterns
  - Bridge, Visitor, ...
  - Basis for implementation of facets

30

Bridge Pattern

► The Bridge pattern is a variant of DimensionalClassHierarchies, with different incentive
► The left hierarchy is called *abstraction hierarchy,* the right *implementation*
  - Also *handle* vs *body*
► However, most important is the separation of two hierarchies

31

Remember Dimensional Class Hierarchy (Bridge with Template/Hook Constraint)

Template

Hook

TemplateClass ─ hookObject ─→ **HookClass**

templateMethod()○ · · · · *hookMethod()*

hookObject.hookMethod()

MoreConcrete TemplateA
templateMethod()○

MoreConcrete TemplateB
templateMethod()○

Concrete HookClassA
hookMethod()

Concrete HookClassB
hookMethod()

Implementation A
... hookObject.hookMethod();

Implementation B
... hookObject.hookMethod();

32
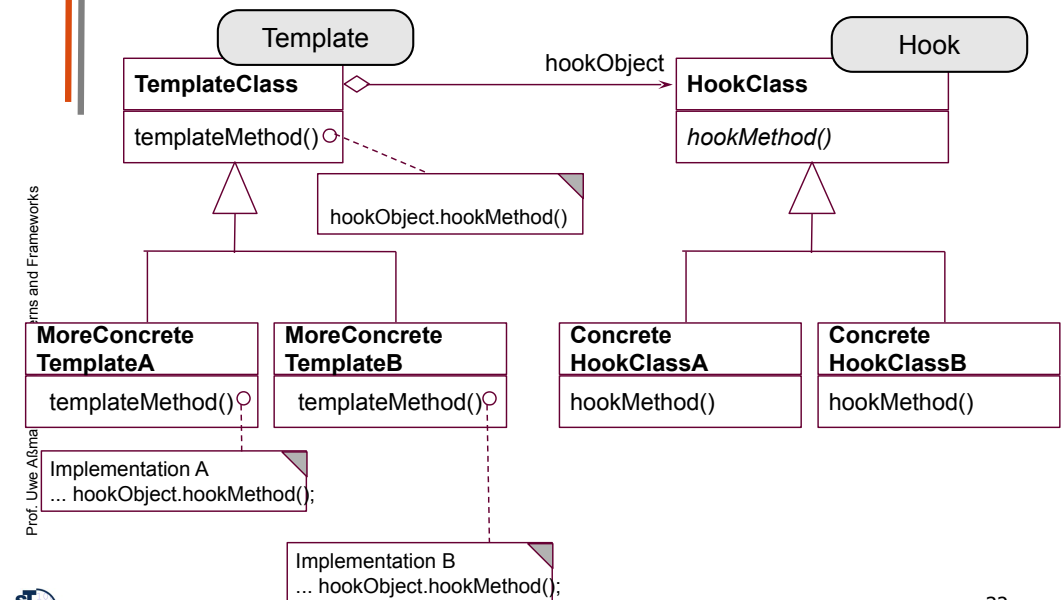
Bridge

► Different incentive (Abstraction/Implementation)

Abstraction

Abstraction

operation()

imp ──► Implementation

Implementation

operationImpl()

imp.operationImpl()

MoreConcrete AbstractionA

operation()

MoreConcrete AbstractionB

operation()

ConcreteImplA

operationImpl()

ConcreteImplB

operationImpl()

Some actions for A;
imp.operationImpl()

Some actions for B;
imp.operationImpl()
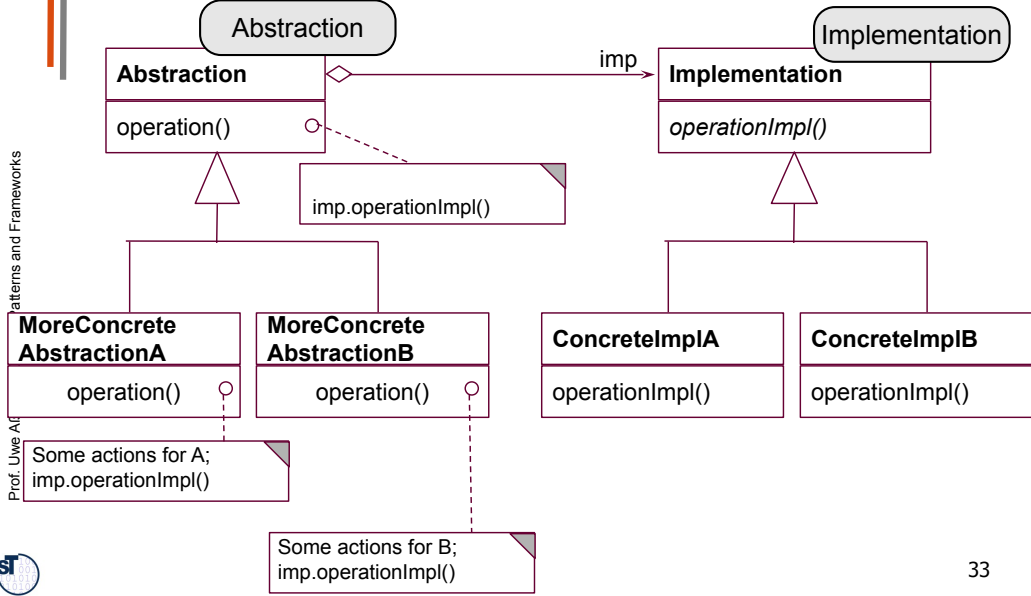
Example: DataGenerator as Bridge

Abstraction

DataGenerator

Data data;
generate()

imp ──► **GeneratorImpl**

Implementation

generateData(Data)

imp.generateData(data)

TestDataGenerator

generate()

ReportGenerator

generate()

ExhaustiveGenerator

generateData(Data)

RandomGenerator

generateData(Data)

data =
parseTestDataGrammar();
imp.generateData(data);

data =
readFromForm();
imp.generateData(data);
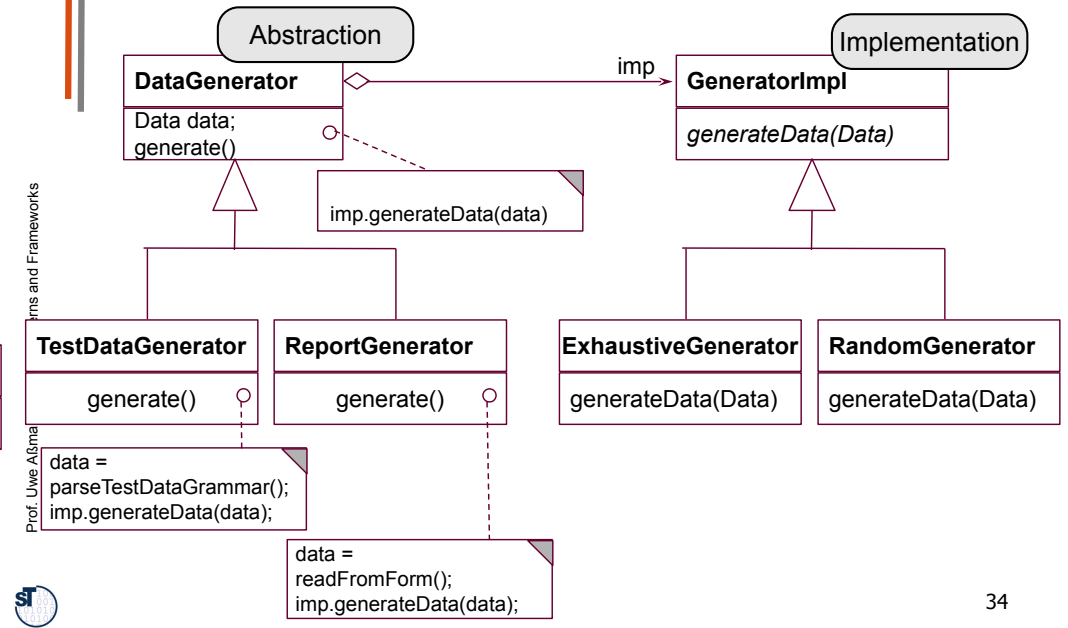
Bridge

► Both hierarchies can be varied independently
  ▪ Factoring (orthogonalization)
  ▪ Reuse is increased
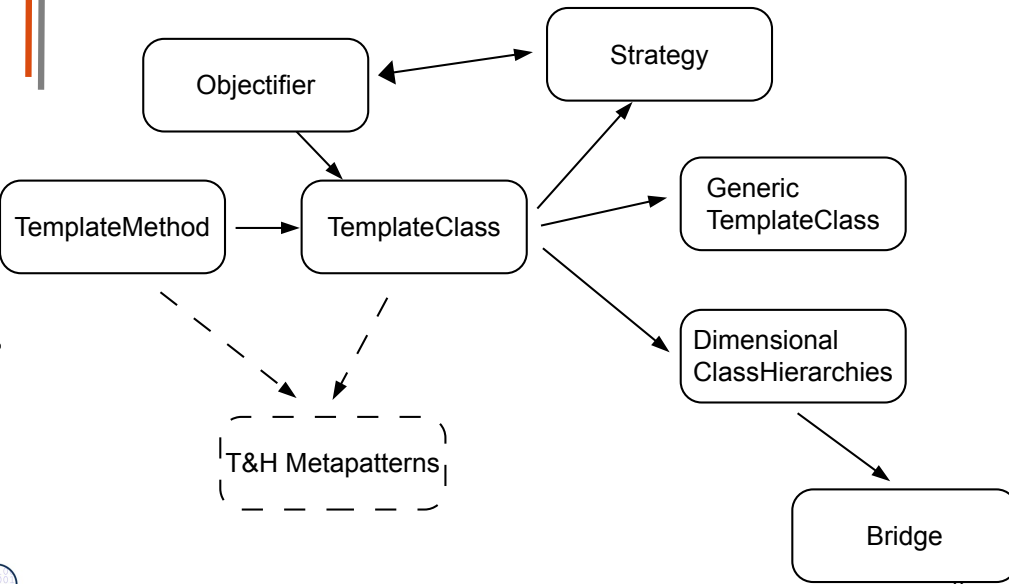► An abstraction can have several Bridges
  ▪ Bridges can be replicated
  ▪ Basis for implementation of *facets*

Multiple Bridges
DataGenerator as 2-Bridge

DataGenerator

generate()

imp ──► **GeneratorImpl**

generateData(Data)

imp.generateData(data)

TestDataGenerator

generate()

ReportGenerator

generate()

ExhaustiveGenerator

generateData(Data)

RandomGenerator

generateData(Data)

data

Data

Data cleanData()

data =
parseTestDataGrammar();
imp.generateData(data);

data =
data.cleanData();
imp.generateData(data);

StringData

TestData cleanData()

GraphData

ReportData cleanData()

Basic Variability Patterns - Overview

# Relations of Basic Patterns

- ▶ Pree book vs Gamma book
  - ▪ Pree and the GOF worked together for some time, but then they published two different books
  - ▪ Pree's focus was on templates and hooks (framework patterns)
  - ▪ GOF on arbitrary patterns in arbitrary context
- ▶ One can take any GOF pattern and make it a framework pattern by introducing the template-and-hook constraint
  - ▪ Or if you take away the template-hook constraint from a framework pattern, you get an unconstrained general pattern

Relation TemplateMethod, TemplateClass, Strategy, Observer
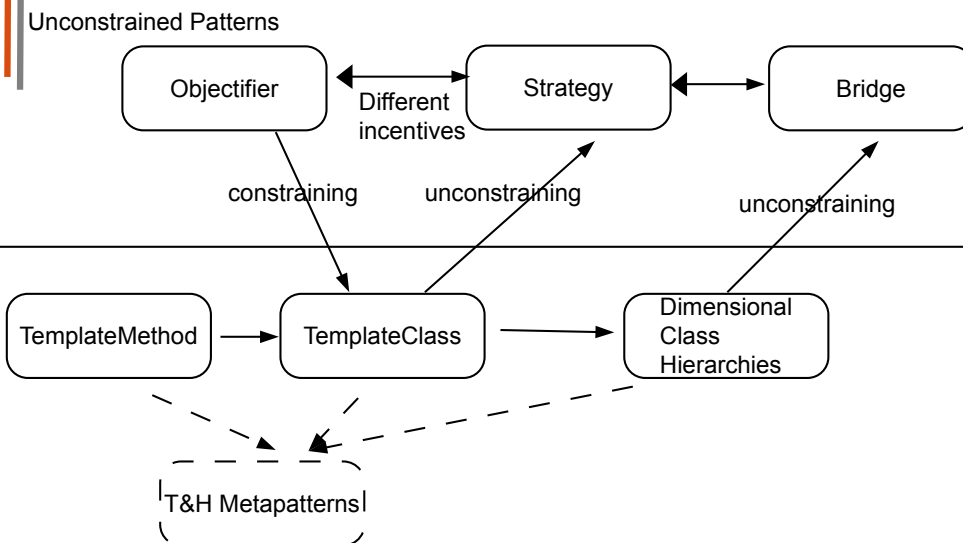
# 2.1.3 Parallel Class Hierarchies (Bridges with Constraints)

When the Dimensions cannot be Independently Varied

- ▶ Sometimes, the dimensions of a DimensionalClassHierarchies are not independent
- ▶ Instead, if it is varied on the left, also on the right must be varied
- ▶ Dimensions have equal size and structure, i.e., are isomorphic
- ▶ Typically are container classes and their elements
  - ▪ UML diagrams and their node and edge types
  - ▪ Figures and their figure elements
  - ▪ Record lists and their record types

Parallel Hierarchies with Parallelism Constraint

- ▶ Both hierarchies, collection and element, must be varied consistently
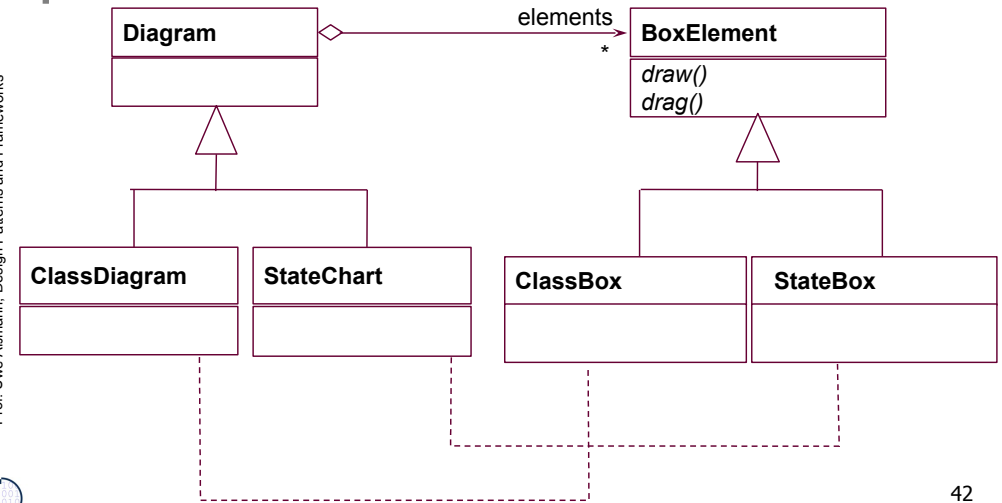
2.1.4 Visitor

Visitor - Purpose

- ▶ The Visitor is a variant of the DimensionalClassHierarchies pattern
  - ▪ Template class hierarchy models a polymorphic data structure
  - ▪ In most cases a tree
- ▶ Hook hierarchy models a polymorphic algorithm
  - ▪ Encapsulate an operation on a collection (tree) of objects as an object
  - ▪ Hook is an objectifier pattern (reified method)
- ▶ Separate tree inheritance hierarchy from command hierarchy
  - ▪ Simple extensibility of both hierarchies
  - ▪ Factoring (orthogonalization): simpler inheritance structures, otherwise multiplication of classes

Structure for Visitor

Client → ObjectStructure → Element
accept(Visitor)

First dispatch on data

b.visit...B (this)

ConcreteElementA
accept(Visitor b)
operationA()

ConcreteElementB
accept(Visitor)
operationB()

b.visitConcreteElementA (this)

Call 2nd dispatch on algorithm

Monomorphic methods on ObjectStructure

Visitor
visitConcreteElemA(ConcreteElementA)
visitConcreteElemB(ConcreteElementB)

Second dispatch

ConcreteVisitor1
visitConcreteElemA(ConcreteElementA)
visitConcreteElemB(ConcreteElementB)

ConcreteVisitor2
visitConcreteElemA(ConcreteElementA)
visitConcreteElemB(ConcreteElementB)

45

Sequence Diagram Visitor

► First dispatch on data, then on visitor

Dispatch 1

aConcreteClient   aConcreteDataObject   aConcreteVisitor

Dispatch 2

accept(aConcreteVisitor)
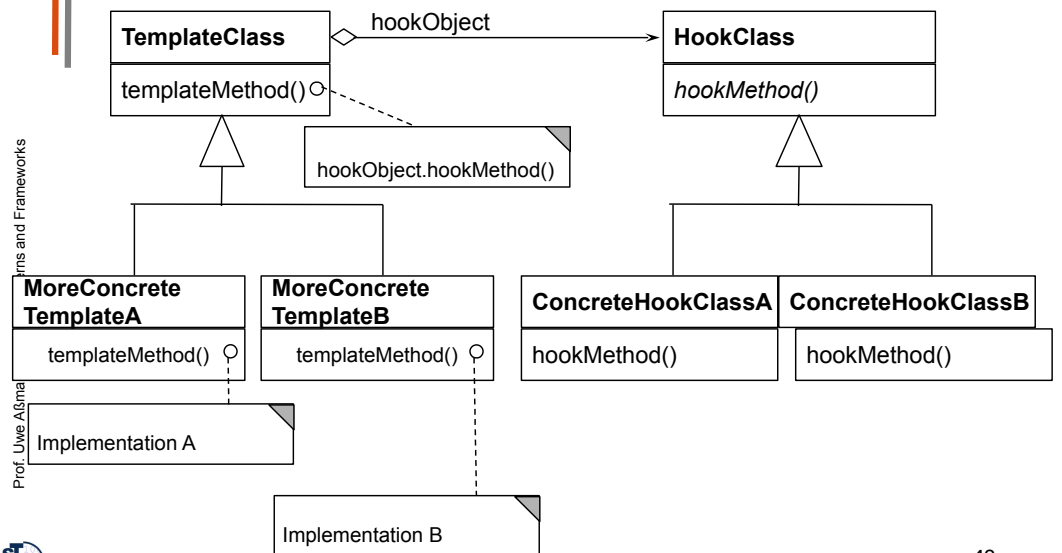
acceptDataObject (aConcreteVisitor)

46
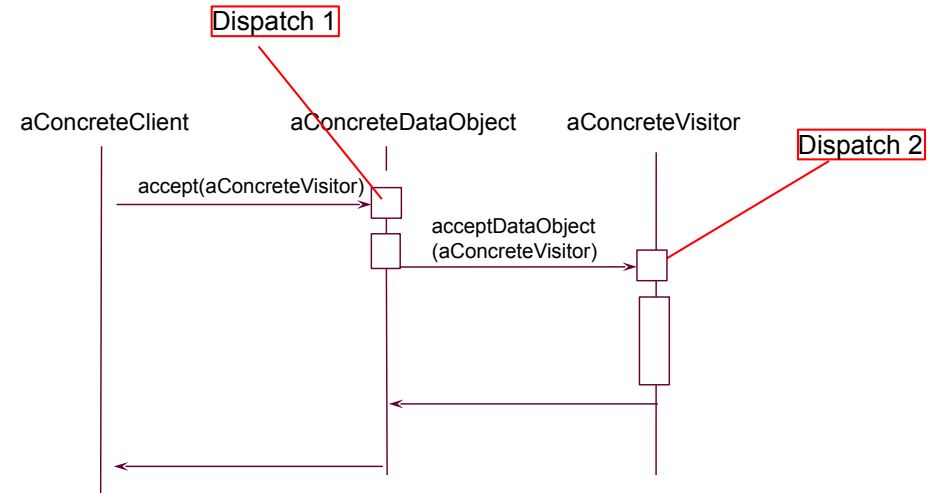
Visitor

► Implementation of a dimensional structure, in which one dimension is an algorithm
  ▪ First dispatch on dimension 1 (data structure), then on dimension 2 (algorithm)
  ▪ The dimensions, however, are not independent (no facets): the chosen implementation of the algorithm depends on the chosen implementation of the data
► Abbreviation for **multimethods**
  ▪ Dispatch/polymorphism on two arguments, not only the first (double dispatch)
  ▪ First dispatch on tree object (method *accept*), then operation (method *visit*) objects

47

Remember: DimensionalClassHierarchies

TemplateClass
templateMethod()

hookObject

HookClass
hookMethod()

hookObject.hookMethod()

MoreConcrete TemplateA
templateMethod()

MoreConcrete TemplateB
templateMethod()

ConcreteHookClassA
hookMethod()

ConcreteHookClassB
hookMethod()

Implementation A

Implementation B

48

Visitor

Usually is missing, replaced by Visitor argument

DataStructure
templateMethod()

◇ Visitor v - - - →

Visitor
hookMethodA(DataStructure)
hookMethodB(DataStructure)

MoreConcrete DataStructureA
templateMethod(Visitor)
templateMethodA(Visitor)

v.hookMethodA(this);

MoreConcrete DataStructureB
templateMethod(Visitor)
templateMethodB(Visitor)

v.hookMethodB(this);

ConcreteVisitorA
hookMethodA(DataStructure)
hookMethodB(DataStructure)

ConcreteVisitorB
hookMethodA(DataStructure)
hookMethodB(DataStructure)

Prof. Uwe Aßmann, ...rns and Frameworks

49

Visitor As Multimethod

method(DataStructure d, Visitor v)

DataStructure
templateMethod()

◇ Visitor v - - - →

Visitor
hookMethodA(DataStructure)
hookMethodB(DataStructure)

MoreConcrete DataStructureA
templateMethod(Visitor)
templateMethodA(Visitor)

v.hookMethodA(this);

MoreConcrete DataStructureB
templateMethod(Visitor)
templateMethodB(Visitor)

v.hookMethodB(this);

ConcreteVisitorA
hookMethodA(DataStructure)
hookMethodB(DataStructure)

ConcreteVisitorB
hookMethodA(DataStructure)
hookMethodB(DataStructure)

Prof. Uwe Aß..., Patterns and Frameworks

50

Example Compiler Abstract Syntax Trees

Node
typeCheck()
generateCode()
prettyPrint()

The operations are distributed
Over the classes.
In case of extensions,
All classes must be extended.

VariableRefNode
typeCheck()
generateCode()
prettyPrint()

AssignmentNode
typeCheck()
generateCode()
prettyPrint()

Prof. Uwe Aßmann, Design Patterns and Frameworks

51
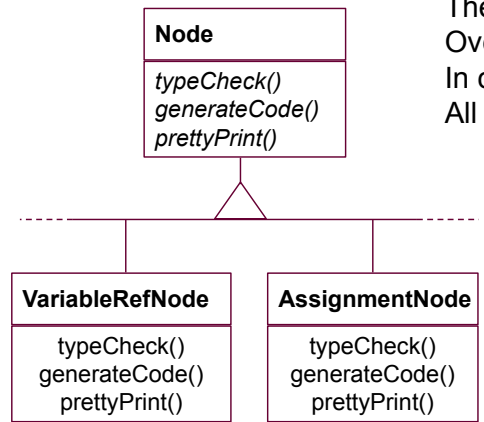
Abstract Syntax Trees with Visitors

Program

◇——●

Node
accept(NodeVisitor)

AssignmentNode
accept(NodeVisitor b)

b.visitAssignment (this)

VariableRefNode
accept(NodeVisitor)

b.visitVariableRef (this)

NodeVisitor
visitAssignment(AssignmentNode)
visitVariableRef(VariableRefNode)

TypeCheckingVisitor
visitAssignment(AssignmentNode)
visitVariableRef(VariableRefNode)

CodeGenerationVisitor
visitAssignment(AssignmentNode)
visitVariableRef(VariableRefNode)

Prof. Uwe Aßmann, Design Patterns and Frameworks

52

2) Dimensional Class Hierarchies (Bridges) as an Implementation of Facet Classifications

... in the following, we use the patterns Bridge and DimensionalClassHierarchies interchangeably

Facet Classifications

- ► A *facet* is an orthogonal dimension of a model
  - Every facet has its separate model
  - All facet classes are abstract
- ► Facets factorize inheritance hierarchies
  - Hence, facets simplify inheritance hierarchies
- ► Final, concrete classes in the (combined) model inherit from every dimension (every facet)
  - All classes in facets are *independent,* i.e., don't know of each other
  - A final class offers the union of all features

- ► "Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained --on the contrary!-- by tackling these various aspects simultaneously."
- ► E. W. Dijkstra "On the Role of Scientific Thought", EWD 447 Selected Writings on Computing: A Personal Perspective, pages 60–66, 1982.

SOC

- ► It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by "focussing one's attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.

Intelligent thinking and scientific thought

► Scientific thought comprises "intelligent thinking" as described above. A scientific discipline emerges with the --usually rather slow!-- discovery of which aspects can be meaningfully "studied in isolation for the sake of their own consistency", in other words: with the discovery of useful and helpful concepts. Scientific thought comprises in addition the conscious search for the useful and helpful concepts.
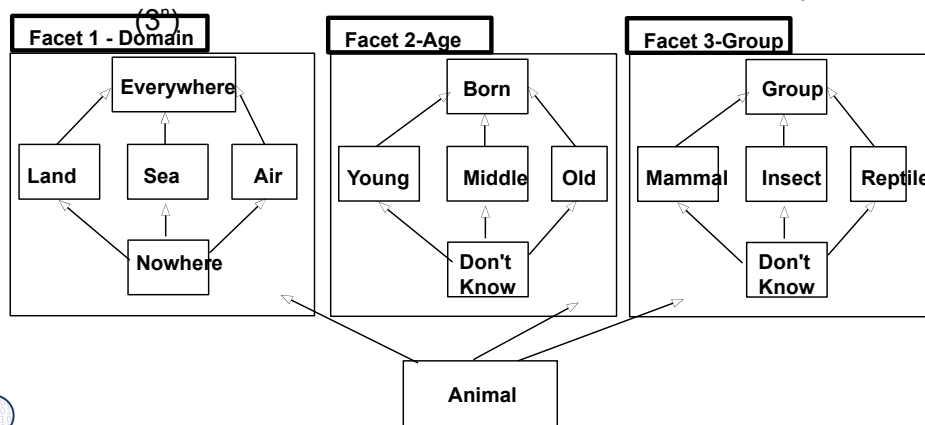
Facets of the Data Generator
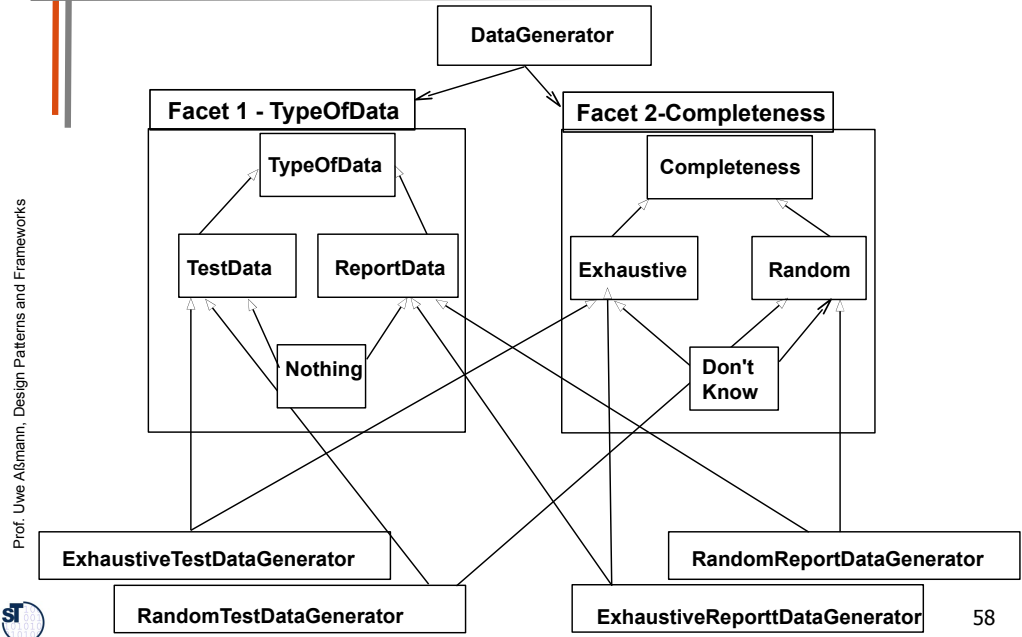
Facets in Living Beings

► The following model of Living Beings has 3 facets
  ▪ Domain (where does an animal live?); Age; Group of Animal
    ▪ All other classes are abstract
► Final, concrete classes inherit from all facets.
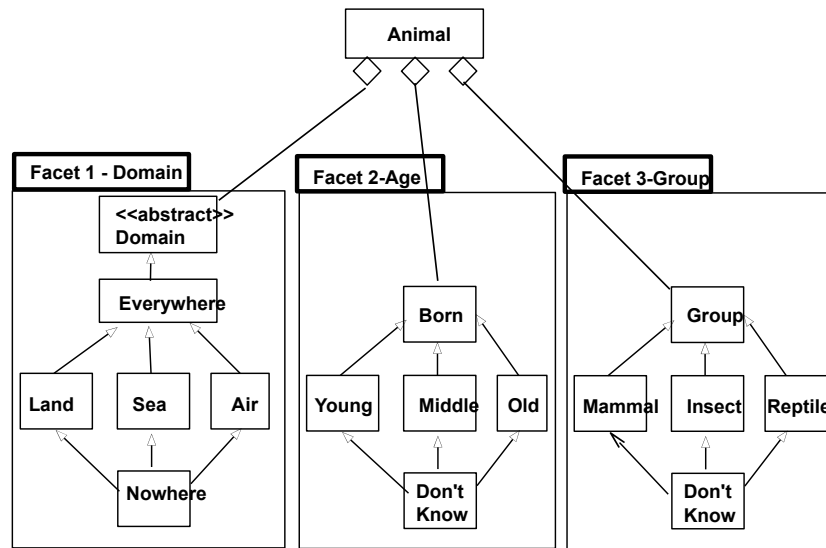► Facets Factorize Models: A full model would multiply all classes (3^n)

Facets Can Be Implemented by Multi-Bridges

► One central facet (abstraction), others are delegatees in bridges (implementation, group, nature, etc.)
► Advantage
  ▪ All facets can be varied independently
  ▪ Simple models
► Restriction: facets model only one logical object
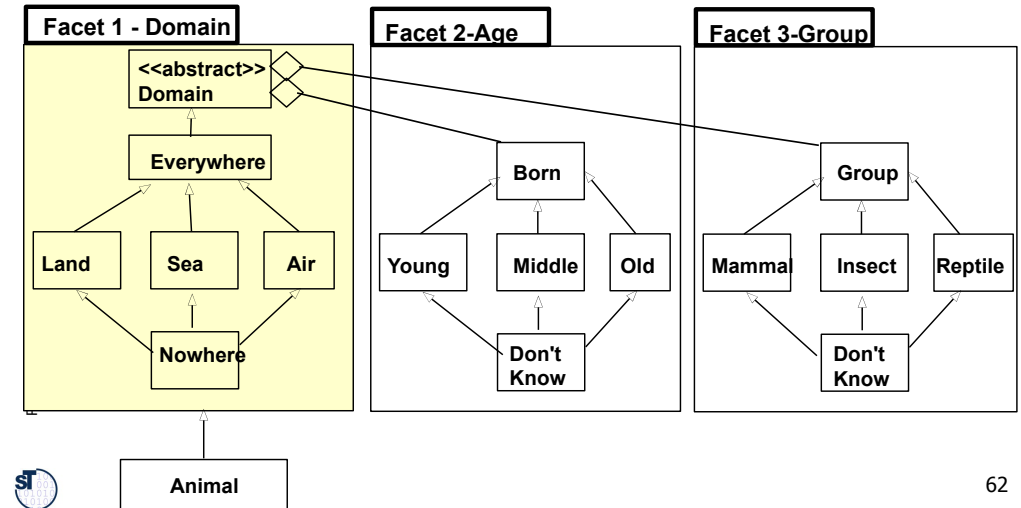 ▪ With several physical objects

Multi-Bridge with Core Facet

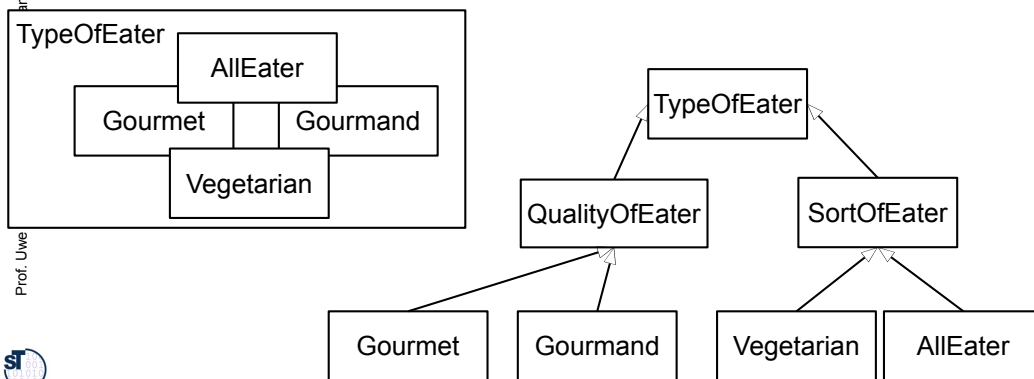► Animal as core facet, all others are *hook classes*

Animal

Facet 1 - Domain
<> Domain
Everywhere
Land · Sea · Air
Nowhere

Facet 2-Age
Born
Young · Middle · Old
Don't Know

Facet 3-Group
Group
Mammal · Insect · Reptile
Don't Know

Multiple Bridge Without Core

► Select a primary facet, relate others by bridges (n-Bridge)
► Problem: primary facet *knows* the others

Facet 1 - Domain
<> Domain
Everywhere
Land · Sea · Air
Nowhere

Facet 2-Age
Born
Young · Middle · Old
Don't Know

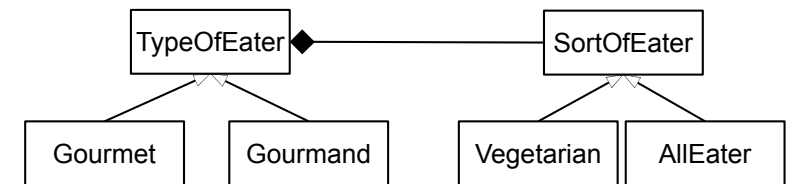Facet 3-Group
Group
Mammal · Insect · Reptile
Don't Know

Animal

How Can I Recognize Facets in Modelling?

► If a class has several different partitions, this indicates a facet model
► A model is *not* a facet model, if some class exists, whose heirs *do not partition* the class (non-partitioned inheritance)
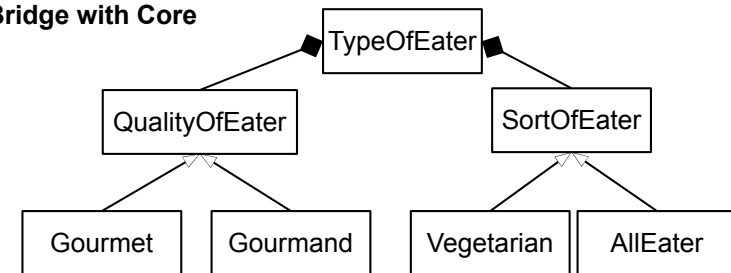

TypeOfEater
AllEater
Gourmet · Gourmand
Vegetarian

TypeOfEater
QualityOfEater → Gourmet · Gourmand
SortOfEater → Vegetarian · AllEater

Resolve with DimensionalClassHierarchies (Bridge)

► Simple Bridge

TypeOfEater ◆—— SortOfEater
Gourmet · Gourmand
Vegetarian · AllEater

Double Bridge with Core

QualityOfEater ◆— TypeOfEater —◆ SortOfEater
Gourmet · Gourmand
Vegetarian · AllEater

Comparison Facet Bridges with Facet Inheritance

► Advantages:
  - Dynamic variations possible
  - Fewer classes
► Disadvantages:
  - No type check on product classes
  - No control over which combinations are created (illegal ones or undefined ones)
  - Object schizophrenia
  - Memory consumption with allocations
  - Speed
  - --> not for embedded systems!

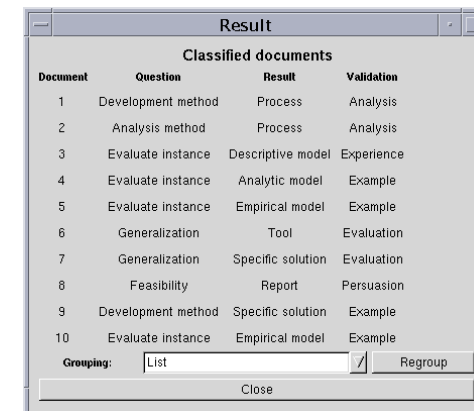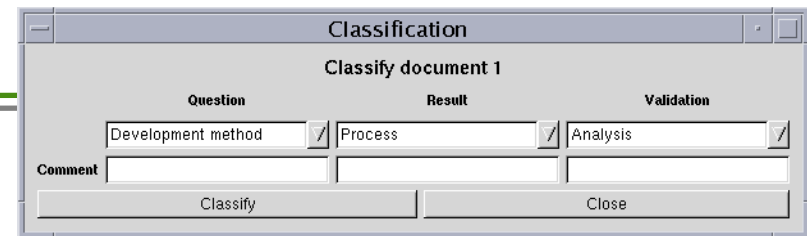## Example: Classification of Research Papers after Shaw

► How to classify a research paper?
► When is it bad, when is it good?
► Mary Shaw proposed a facet-based classification with the facets
  - Research question
  - Result
  - Evaluation

Classification of Research Papers

► 5+7+5 facet classes→175 product classes (types of research papers)

Question	Result	Validation
Development method	Process	Analysis
Analysis method	Descriptive model	Experience
Evaluate instance	Analytic model	Example
Generalization	Empirical model	Evaluation
Feasibility	Tool	
	Specific solution	Persuasion
	Report	

Classification

Classify document 1

Question	Result	Validation
Development method	Process	Analysis

Comment: [ ]

Classify | Close

Result

Classified documents

Document	Question	Result	Validation
1	Development method	Process	Analysis
2	Analysis method	Process	Analysis
3	Evaluate instance	Descriptive model	Experience
4	Evaluate instance	Analytic model	Example
5	Evaluate instance	Empirical model	Example
6	Generalization	Tool	Evaluation
7	Generalization	Specific solution	Evaluation
8	Feasibility	Report	Persuasion
9	Development method	Specific solution	Example
10	Evaluate instance	Empirical model	Example

Grouping: [List] Regroup

Close

Result

Classified documents

Question\Validation	Analysis	Experience	Example	Evaluation	Persuasion
Development method	1		9		
Analysis method	2				
Evaluate instance		3	4 5 10		
Generalization				6 7	
Feasibility					8

Grouping: Question/Validation ▽ Regroup

Close

When to Use Facet-based Models

► When the model consists of independent dimensions

► When the model is very complicated

► Realizations:

- Use multiple inheritance, when good type checking is advantageous (e.g., in frameworks)
- Use Bridge if language does not support multiple inheritance

---

Several Facet Groups are Possible

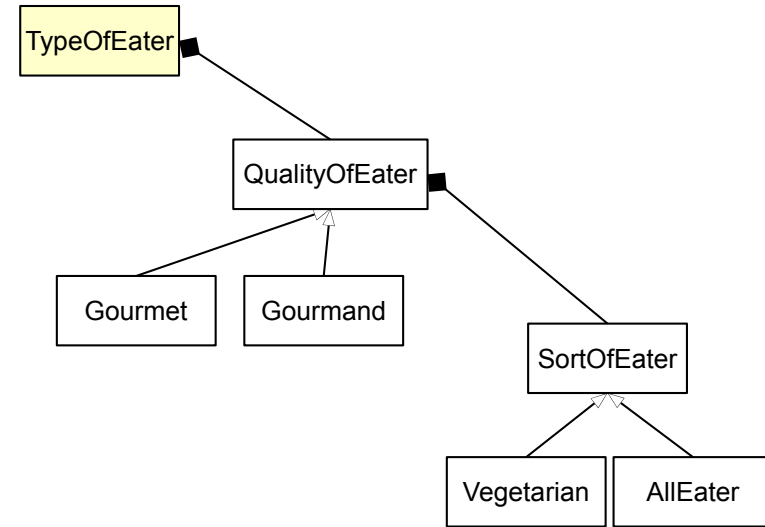---
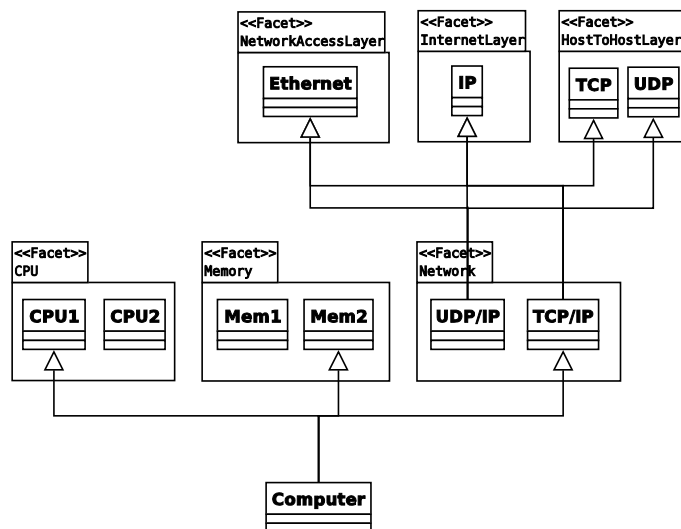
2.3) Layered Objects

Be Aware

- ▶ If you meet a Bridge, you may have a facet classification
- ▶ Only question: are the dimensions independent?
- ▶ Sometimes, dependencies exist, e.g., one dimension calls another
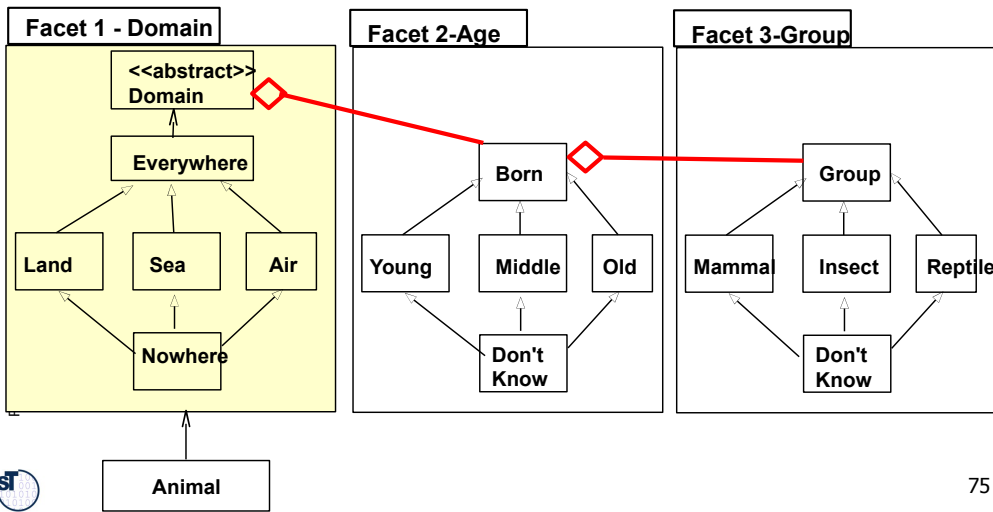 - ▪ This requires an interface (contract) between the dimensions

73

Layered Objects with Chain-Bridge
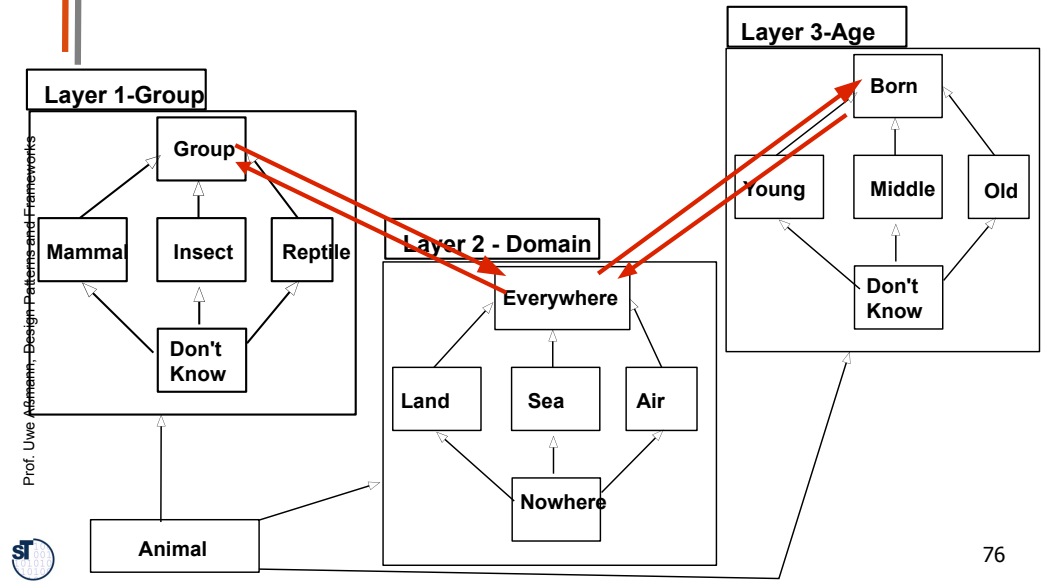
- ▶ Chain Bridge with Core

TypeOfEater

QualityOfEater

Gourmet
Gourmand

SortOfEater

Vegetarian
AllEater

74

Chain-Bridge for Layered Object Implementation

- ▶ Select a primary facet, relate others by chain-Bridges
- ▶ Here without core

Facet 1 - Domain
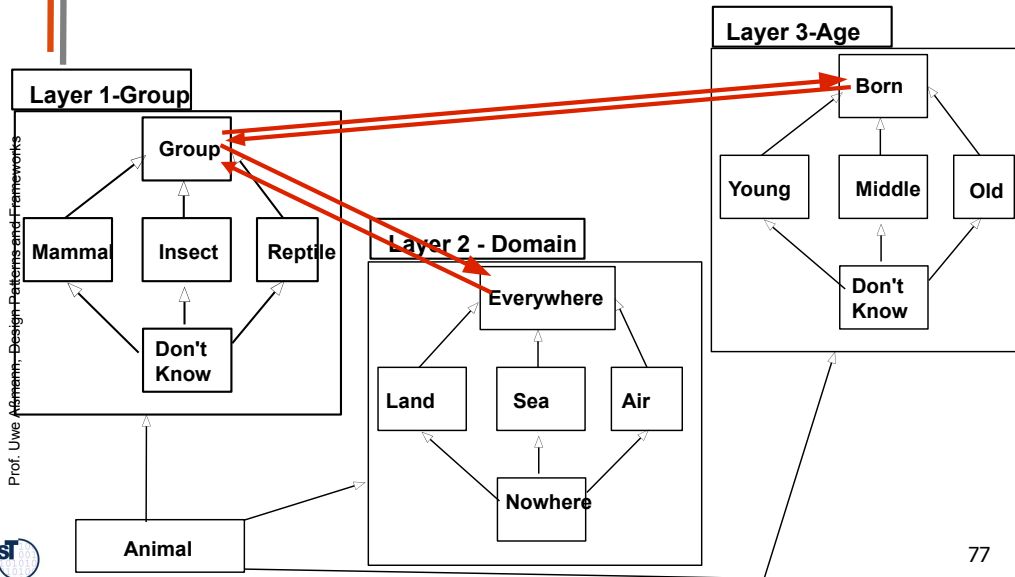
<> Domain

Everywhere

Land   Sea   Air

Nowhere

Animal

Facet 2-Age

Born

Young   Middle   Old

Don't Know

Facet 3-Group

Group

Mammal   Insect   Reptile

Don't Know

75

Layered Objects

- ▶ Upper layers depend on information of lower layers

Layer 1-Group

Group

Mammal   Insect   Reptile

Don't Know

Layer 2 - Domain

Everywhere

Land   Sea   Air

Nowhere

Animal

Layer 3-Age

Born

Young   Middle   Old

Don't Know

76

Compare to Facets

► Dimensions do not depend on information of others

Layer 3-Age
Born
Young | Middle | Old
Don't Know

Layer 1-Group
Group
Mamma | Insect | Reptile
Don't Know

Layer 2 - Domain
Everywhere
Land | Sea | Air
Nowhere
Animal

77

DataGenerator as 2-Chain-Bridge

GeneratorImpl
generateData()

Control Flow

data

DataGenerator
generate()

imp

imp.generateData()

ExhaustiveGenerator
generateData()

RandomGenerator
generateData()

Data data = data.cleanData();

Control Flow | Data Flow

Data
Data cleanData()

TestDataGenerator
generate()

ReportGenerator
generate()

imp.generateData();

TestData
TestData cleanData()

ReportData
ReportData cleanData()

Compare to DataGenerator as 2-Bridge (Facets)

GeneratorImpl
generateData(Data)

DataGenerator
generate()

imp

data

Control Flow

imp.generateData(data)

ExhaustiveGenerator
generateData(Data)

RandomGenerator
generateData(Data)

TestDataGenerator
generate()

ReportGenerator
generate()

data = data.cleanData(); imp.generateData(data);

Data
Data cleanData()

Control Flow Data Flow

data = data.cleanData(); imp.generateData(data);

TestData
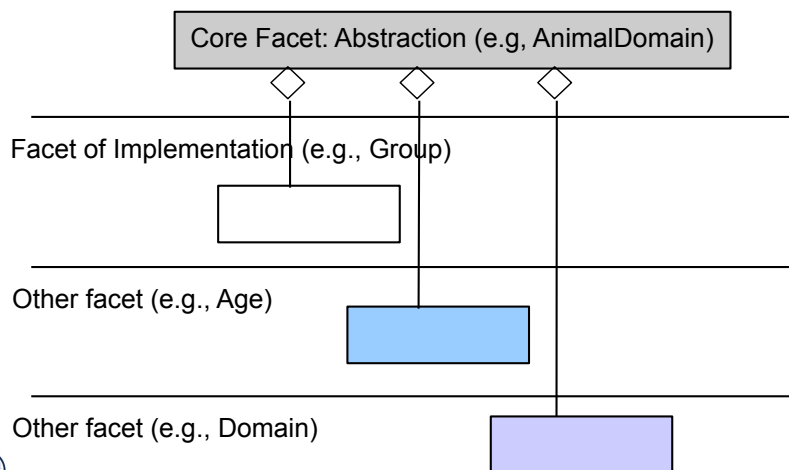TestData cleanData()

ReportData
ReportData cleanData()

Layered Object Spaces

► A layered object space is similar to a facet space
► However, layers exchange information in a *directed way*
  ▪ Upper layers *call* lower layers, which *deliver* information to upper layers
  ▪ This requires that the abstract topmost classes in a layer provide abstract methods that can be called from other layers
  ▪ The dependencies are *directed and acyclic* (form a DAG)
► Still, all classes in a layer can be exchanged freely for another
► Layered object spaces are much broader applicable than facet spaces

80
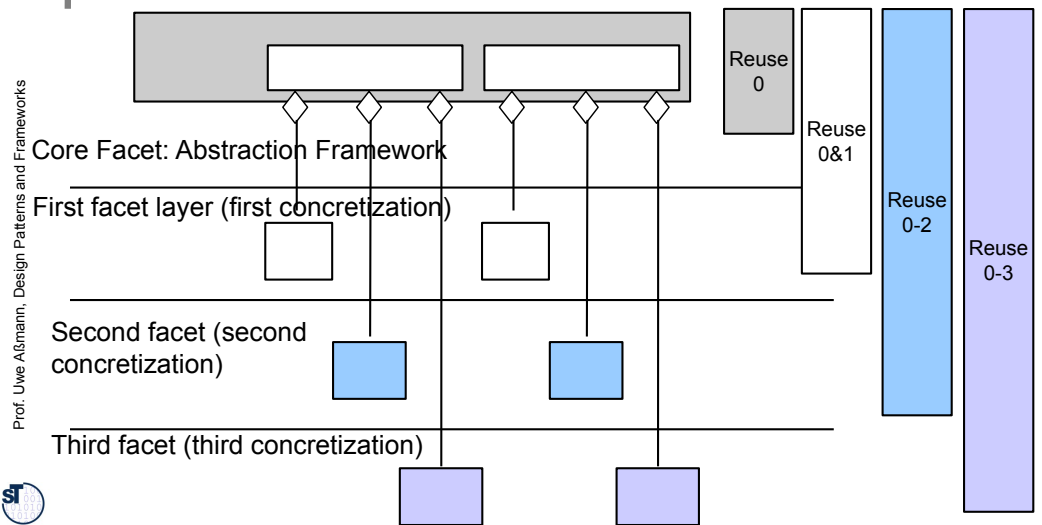
2.4) Facet-Bridge Frameworks and Dimensional Systems

Multiple Bridges for Facet-Based Systems

► So far, we looked at implementations of faceted or layered objects, i.e., models of complex objects

► Facet classifications and layered objects can be generalized to facet-based or dimensional *frameworks* and *systems*

Facet-Bridge Dimensional Systems

► Bridge patterns can be divided upon different dimensions

► Here: a triple Bridge with core and 3 dimensions, all independent

Core Facet: Abstraction (e.g, AnimalDomain)

Facet of Implementation (e.g., Group)

Other facet (e.g., Age)

Other facet (e.g., Domain)

Facet-Bridge Frameworks for Facet-Based Systems

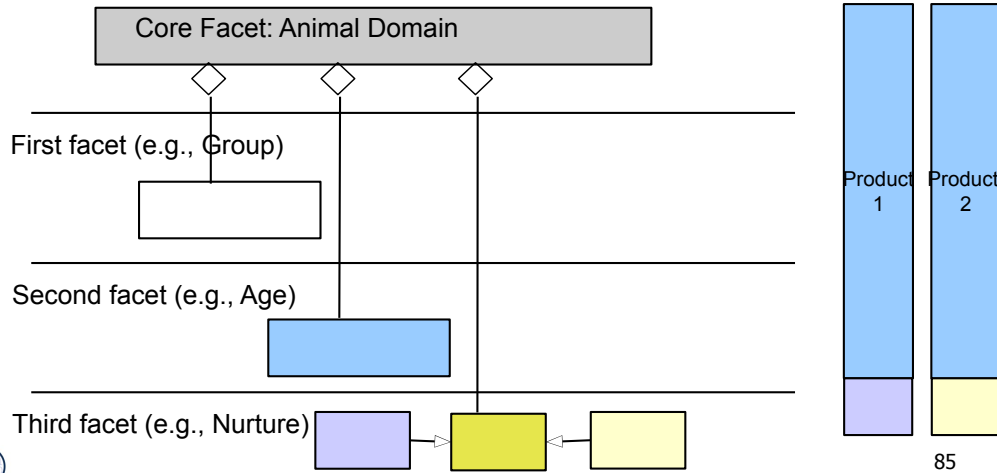► If one or several layers are fixed, and the rest is variable, *facet frameworks* result

Core Facet: Abstraction Framework

First facet layer (first concretization)

Second facet (second concretization)

Third facet (third concretization)

Reuse 0

Reuse 0&1

Reuse 0-2

Reuse 0-3

Facet-Bridge Frameworks for Facet-Based Systems

OldLandMammalVegetarian

- ▶ Products can be instantiated on every level
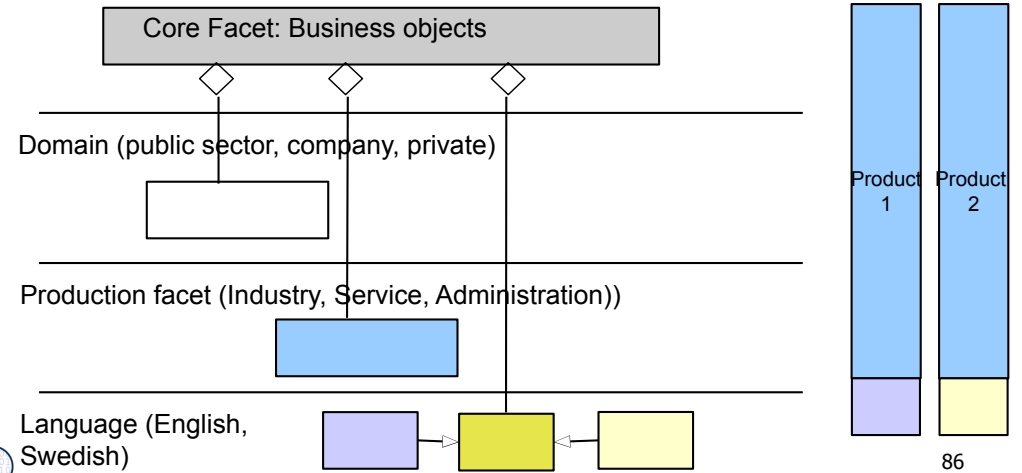- ▶ Every dimension provides additional reuse

OldLandMammalMeateater

Core Facet: Animal Domain

First facet (e.g., Group)

Second facet (e.g., Age)

Third facet (e.g., Nurture)

Product 1

Product 2

85

Facet-Bridge Frameworks for Business Software

BillForCompanyOfIndustry in English

- ▶ SAP, Peoplesoft, Intentia, IFS

BillForCompanyOfIndustry in Swedish

Core Facet: Business objects

Domain (public sector, company, private)

Production facet (Industry, Service, Administration))

Language (English, Swedish)
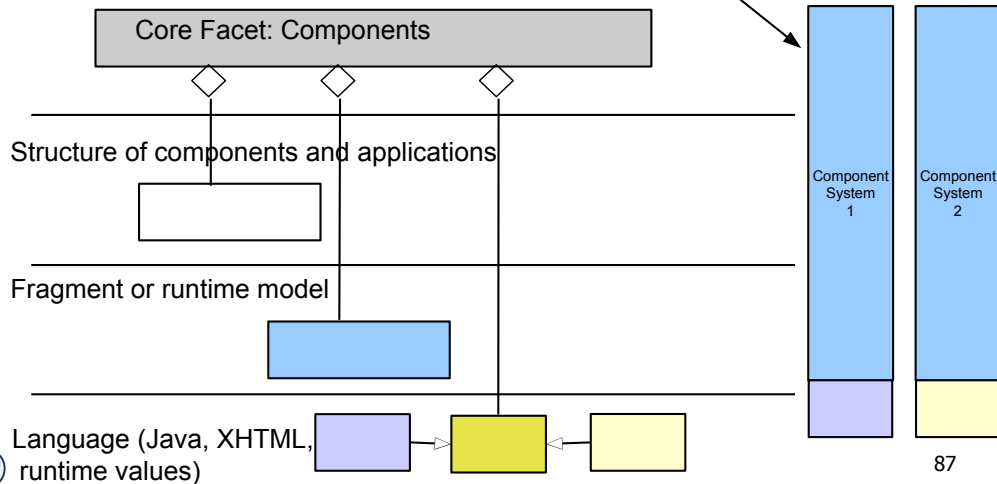
Product 1

Product 2

86

Facet-Bridge Frameworks for Component Composition

JavaFragmentComponentModel

- ▶ COMPOST (www.the-compost-system.org) is a framework for constructing component systems (a framework for frameworks)

RuntimeValueComponentModel
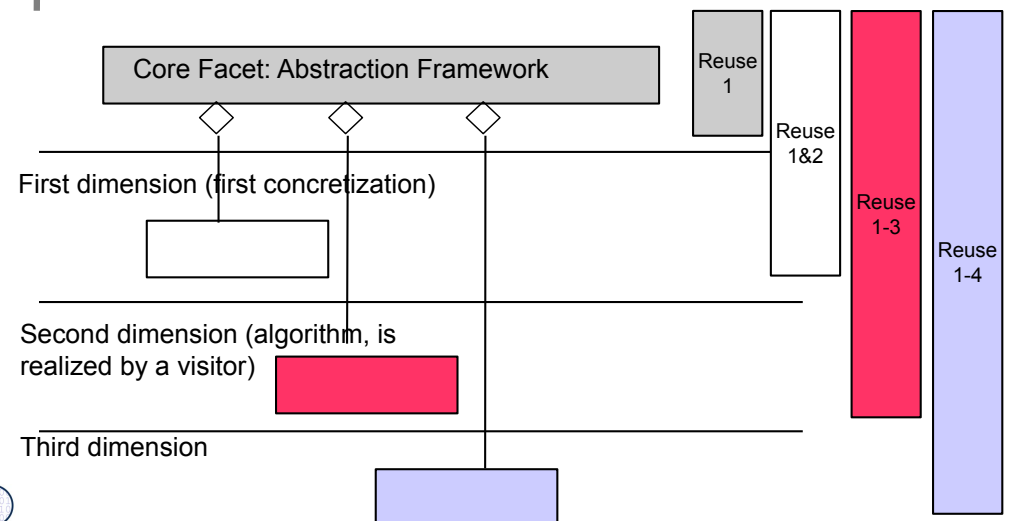
Core Facet: Components

Structure of components and applications

Fragment or runtime model

Language (Java, XHTML, runtime values)

Component System 1

Component System 2

87

Bridge and Visitor Frameworks for Dimensional Systems

- ▶ Some layers of a dimensional frameworks can be visitors

Core Facet: Abstraction Framework

First dimension (first concretization)

Second dimension (algorithm, is realized by a visitor)

Third dimension

Reuse 1

Reuse 1&2

Reuse 1-3

Reuse 1-4

Facet-based Design and Frameworks

▶ Best practice: whenever you have a huge class hierarchy, that is not completely based on partitioning
  ▪ Find out the facets
  ▪ Factor the inheritance hierarchy into the facets
  ▪ Choose a core facet
  ▪ And implement the facets with a facet framework with Bridges.
  ▪ For an n-dimensional facet problem you need at least n-1 Bridges
▶ If the "facets" are not independent, introduce layers
  ▪ And implement them with Chain-Bridges

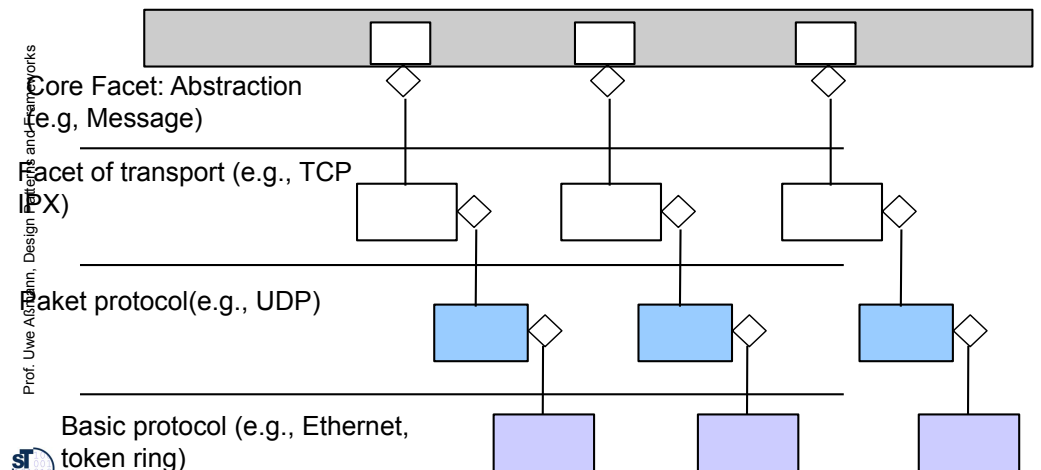# 2.5) Layered Frameworks and Systems with Chain Bridges

Layered Frameworks for Layered Systems

▶ Whenever a system is a layered architecture (stack architecture), a *layered (object) framework* can be used
  ▪ And Chain-Bridges can implement them if the layers are independent of each other (*layered chain-bridge framework*)
  ▪ The layering is an abstraction layering: more detailed things appear as lower layer
▶ Modelling criterion: every class must contribute to every layer of a layered object system
  ▪ Classes *crosscut the layers*
  ▪ In general, layered system do not meet this criterion
▶ Different products can be configured easily by varying the dimensions of the bridge

Network Stacks as Layered Bridge System

▶ ISO/OSI has 7 layers (leads to a 7-Bridge)
▶ Every layer knows the next underlying
▶ All partial objects call partial objects in lower layers

Core Facet: Abstraction (e.g, Message)

Facet of transport (e.g., TCP IPX)

Paket protocol(e.g., UDP)
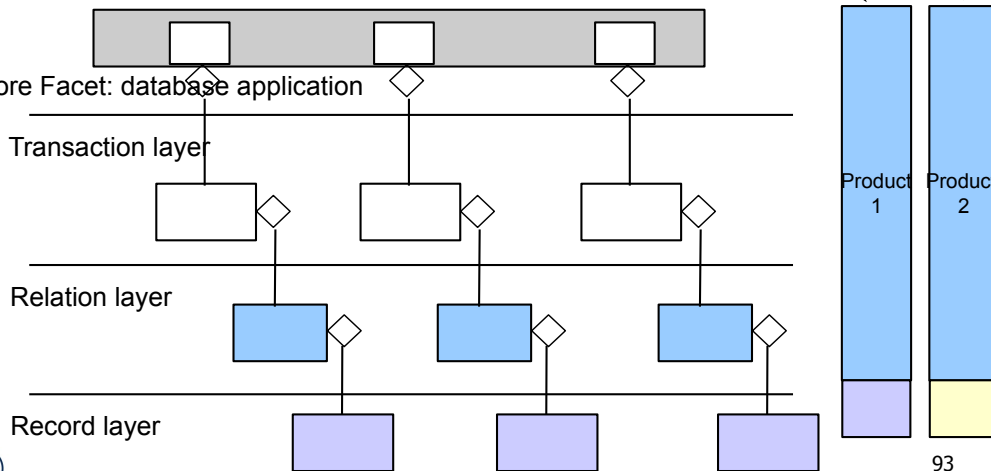
Basic protocol (e.g., Ethernet, token ring)

Databases and Layered Bridge Frameworks

► An object-oriented database, which should be integrated into an application, should be a layered bridge framework

DatabaseAppl A

Database Appl B

Core Facet: database application

Transaction layer

Relation layer

Record layer

Product 1

Product 2

The Role of Layered Frameworks

► Layered frameworks are a very important structuring method for large systems that must be parameterized, varied and extended

► On every layer, reuse is possible
  ▪ Enourmous variability

► Every layer corresponds to an *aspect* of the application
  ▪ All layers form *stacked aspects*

► A large system must be *reducible* or *layered*
  ▪ Hence, layered frameworks provide a wonderful, very general methods for product lines of very large products
  ▪ And additionally, for *extensible* systems

The Role of Layered Frameworks

► At the moment, there are three competing implementation technologies for them:
  ▪ Aspect-oriented weaving
  ▪ View-based weaving (hyperslice programming) [see Component-Based Software Engineering, summer semester)
  ▪ Hand programming
    · Chain-Bridges
    · Role Object Pattern (see later)

► To me, it looks like layered frameworks are one of *the most important* software engineering concepts of the future

---

The End