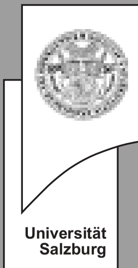


Frameworks & Design Patterns

Vorlesung Software Engineering I
Wintersemester 2003/04
Universität Salzburg

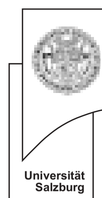
O.Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Pree

© Copyright Wolfgang Pree, All Rights Reserved

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Inhalt(I)

- Motivation
  - OO Frameworks
  - Metainformationen als Basis für die dynamische Konfiguration von Softwaresystemen
- Entwurfsmuster (= Design Patterns)
  - Überblick (Coding Patterns, Cookbooks, Framework Patterns)
  - Design Pattern Katalog (Gamma et al.)
  - Essentielle Konstruktionsprinzipien (Pree)
  - UML-F zur Beschreibung von Frameworks



© 2003, W. Pree

Inhalt(II)

- Patterns @ Work
  - Fallstudie: Wiederverwendung „kleiner“ Frameworks
- Hilfsmittel & Tips zu OOAD
  - Metriken
  - (Re-)Design-Strategien
    - Klassenfamilien, Teams, Subsysteme
    - Horizontale und vertikale (Re-)Organisation von Klassenhierarchien

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Example isn‘t another way to teach, it is the only way to teach

Albert Einstein

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Frameworks (= Product Lines = Platforms = Generic Software)

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

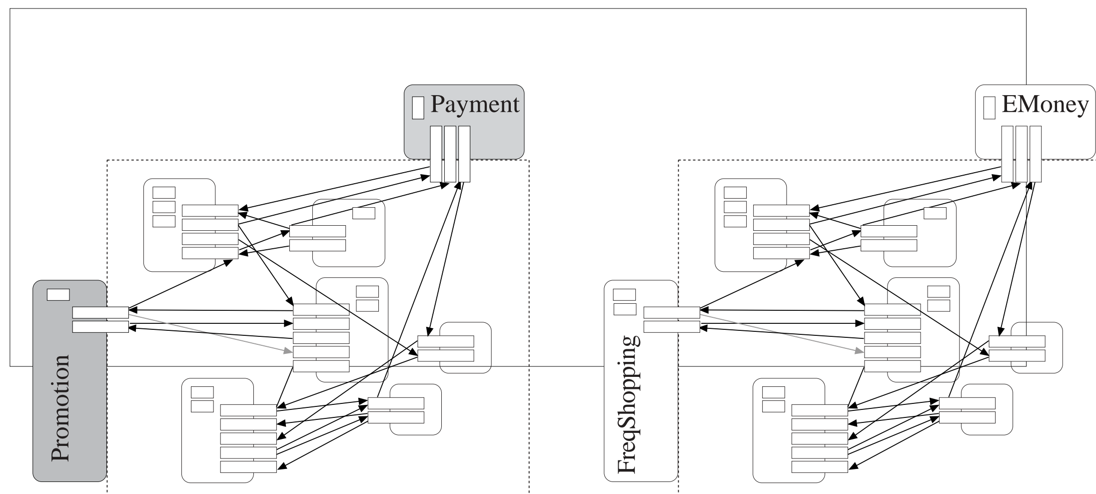
Frameworks allgemein

Beispiele für „Nicht-Software“-Frameworks:

- Küchenmaschine: durch Einstecken einer Komponente wird das vorhandene „Halbfertigfabrikat“ zum fertigen Mixer oder Fleischwolf
- neue Automodelle gleichen meist „im Kern“ (Chassis, Getriebe, Motorpalette) den Vorgängermodellen

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Konfiguration durch Einstecken von SW-Komponenten



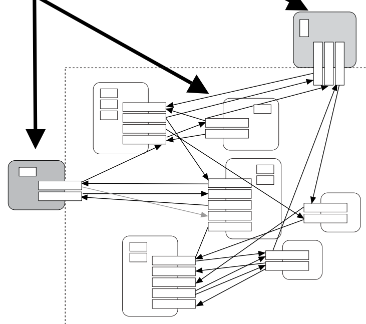
vor der Adaptierung **nach der Adaptierung**

© 2003, W. Pree

Frameworks— Definition & Dynamische Sicht

Framework:= a piece of software that is extensible through the callback style of programming

OO Frameworks: **Komponenten + Interaktion + Hot Spots** **(=Platzhalter; meist abstrakte Klassen oder Schnittstellen)**



© 2003, W. Pree

Black-Box versus White-Box Framework-Komponenten

- **Black-Box:** Wiederverwendung ohne jegliche Anpassung: **„Plug & Work“**
- **White-Box:** Anpassung durch **Unterklassenbildung** erforderlich
- **Je reifer** ein Framework ist, **umso mehr Black-Box-Komponenten** sind enthalten.

© 2003, W. Pree

Abstrakte Klassen

Abstrakte Klassen standardisieren die Schnittstelle (den „Stecker“) für Unterklassen.

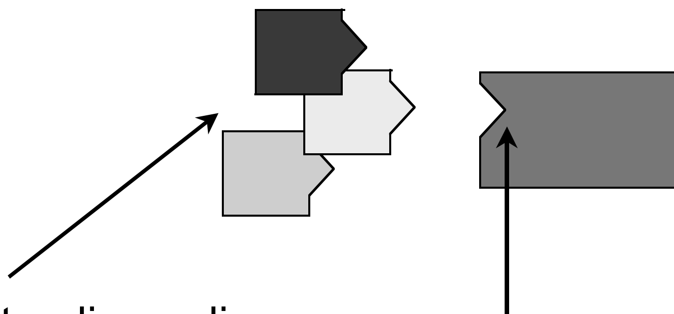
Aufgrund der dynamischen Bindung **können andere Systemteile/Komponenten nur aufgrund des Protokolls (= angebotene Methoden) von abstrakten Klassen bereits implementiert werden.**

Somit können Halbfertigfabrikate (=Frameworks) softwaretechnisch elegant entwickelt werden.

© 2003, W. Pree

Polymorphismus

Sogenannte Objekttypen sind **poly** (= viel) **morph** (= Gestalt). Anschaulich ist das mit **„Steckerkompatibilität"** vergleichbar:



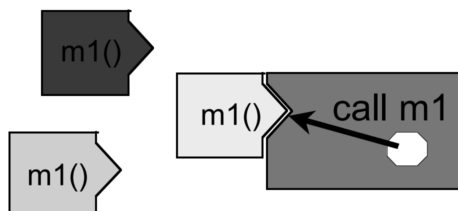
Objekte, die zu diesem Stecker kompatibel sind

„Stecker"-Standard

© 2003, W. Pree

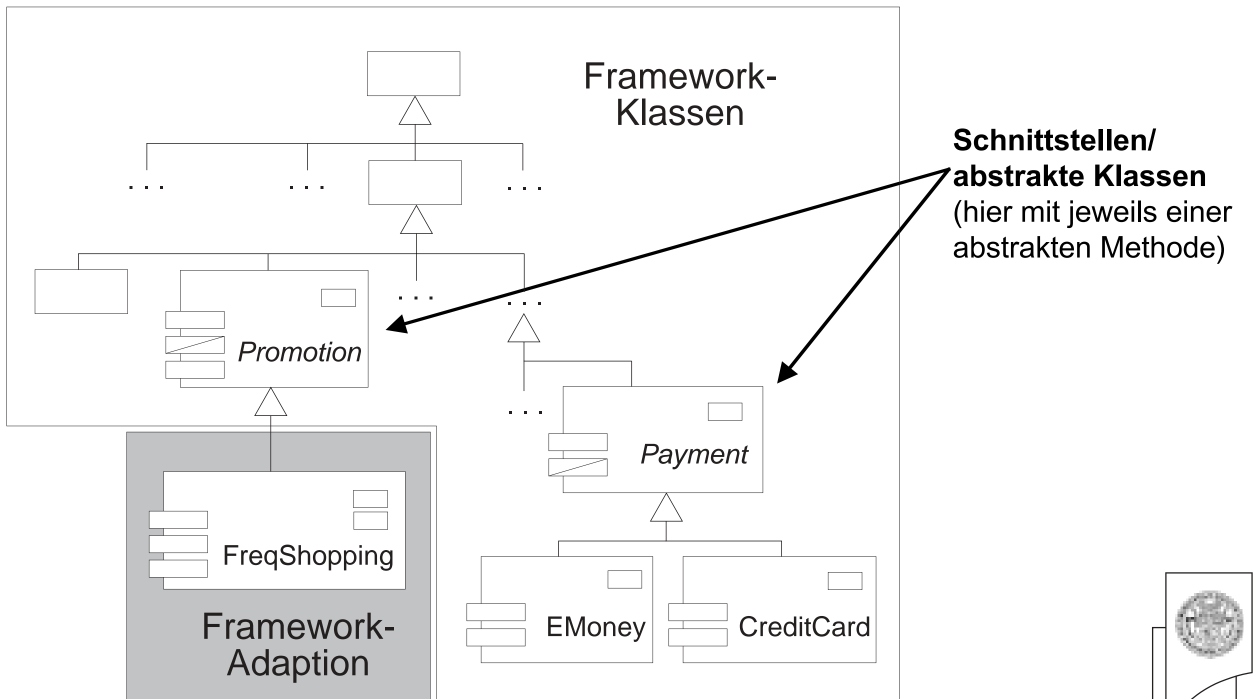
Dynamische Bindung

Dynamische Bindung heißt, daß es **vom eingesteckten Objekt abhängt, welche Methode tatsächlich ausgeführt wird.** Das gelbe Objekt implementiert m1() zB anders als das rote Objekt:



© 2003, W. Pree

Frameworks—Statische Sicht

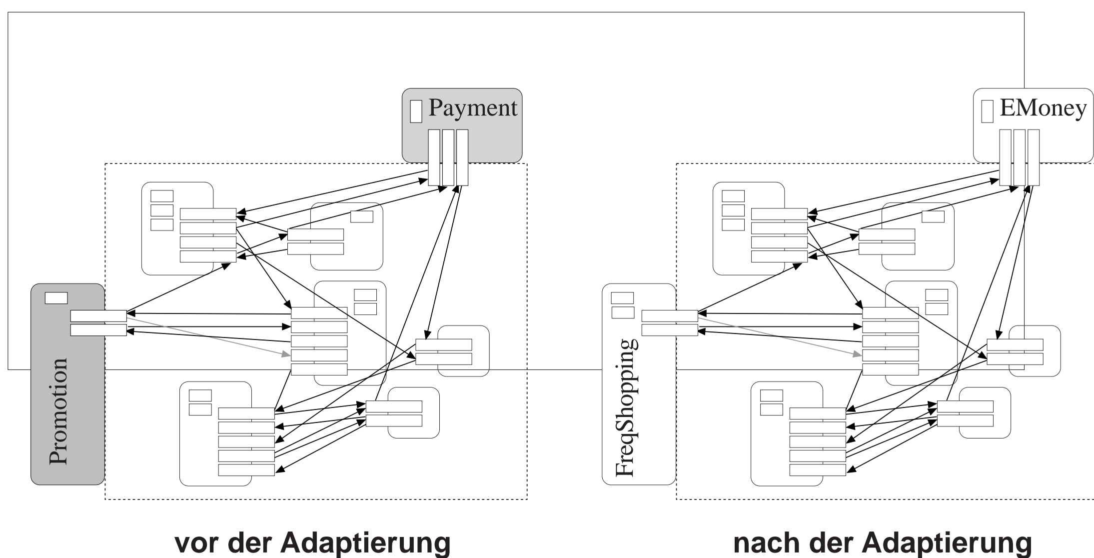


Schnittstellen/
abstrakte Klassen
(hier mit jeweils einer abstrakten Methode)

© 2003, W. Pree

SOFTWARE RESEARCH LAB — Universität Salzburg

Konfiguration durch Einstecken von SW-Komponenten

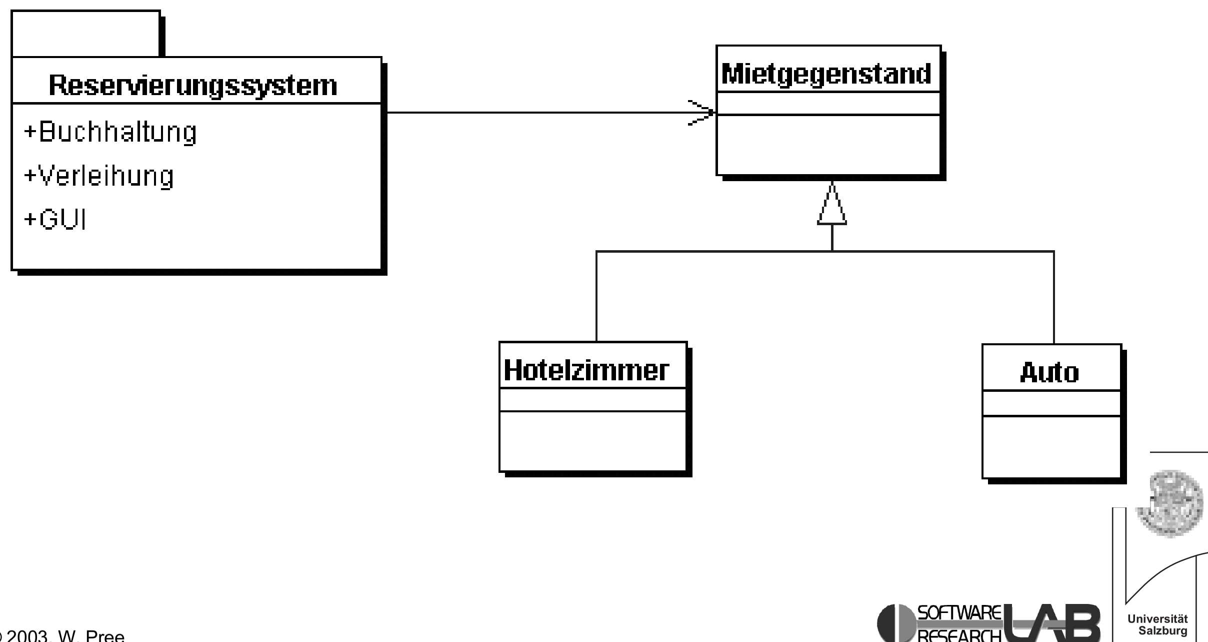


vor der Adaptierung | **nach der Adaptierung**

© 2003, W. Pree

SOFTWARE RESEARCH LAB — Universität Salzburg

weiteres Beispiel: Hotelreservierung



© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

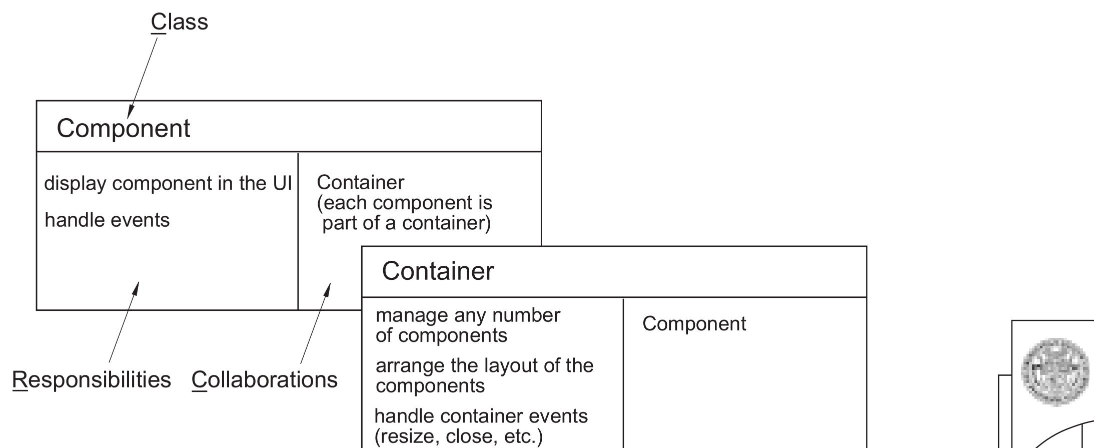
Swing-Framework?

Wie bilden die beiden abstrakten Klassen

Component

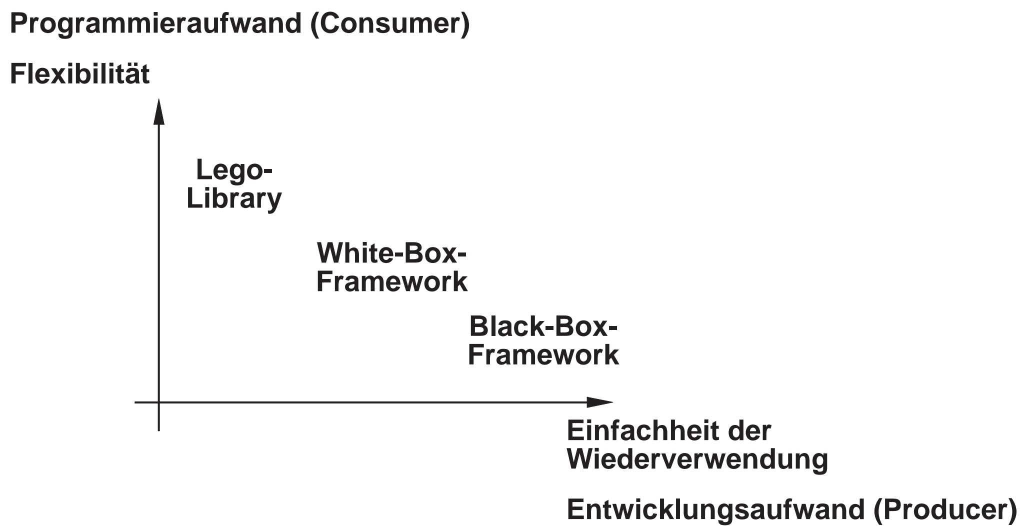
Container

in Swing ein Framework?



© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

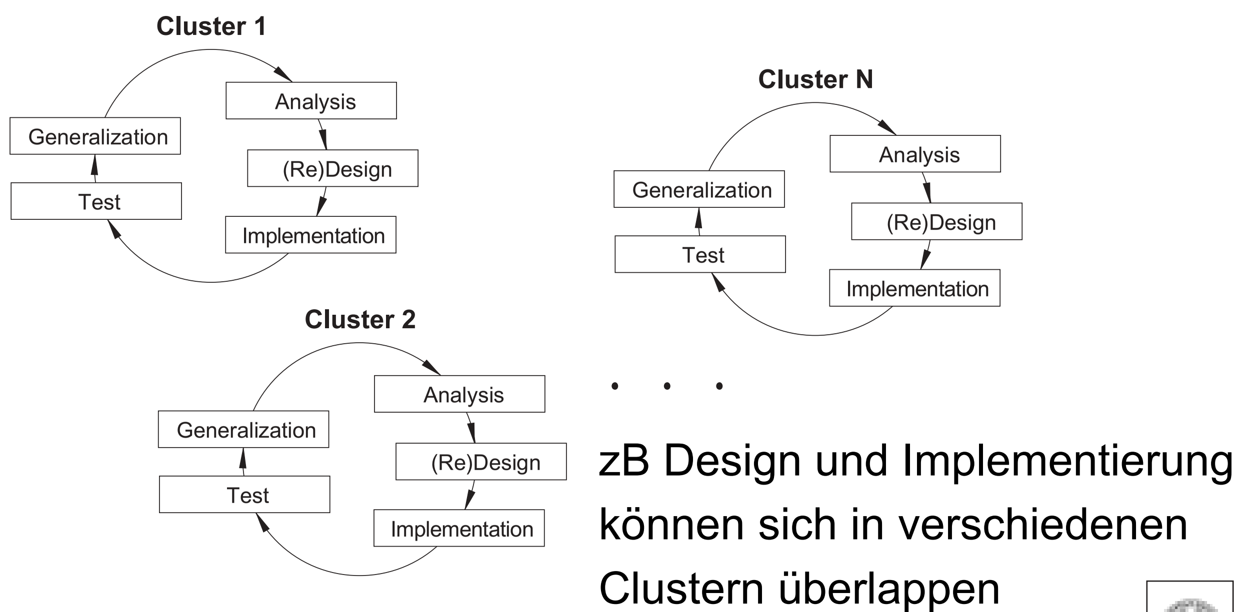
Legokasten/WB-/BB-Framework



(Darstellung adaptiert aus dem Tutorial von Erich Gamma, OOP'96)

© 2003, W. Pree

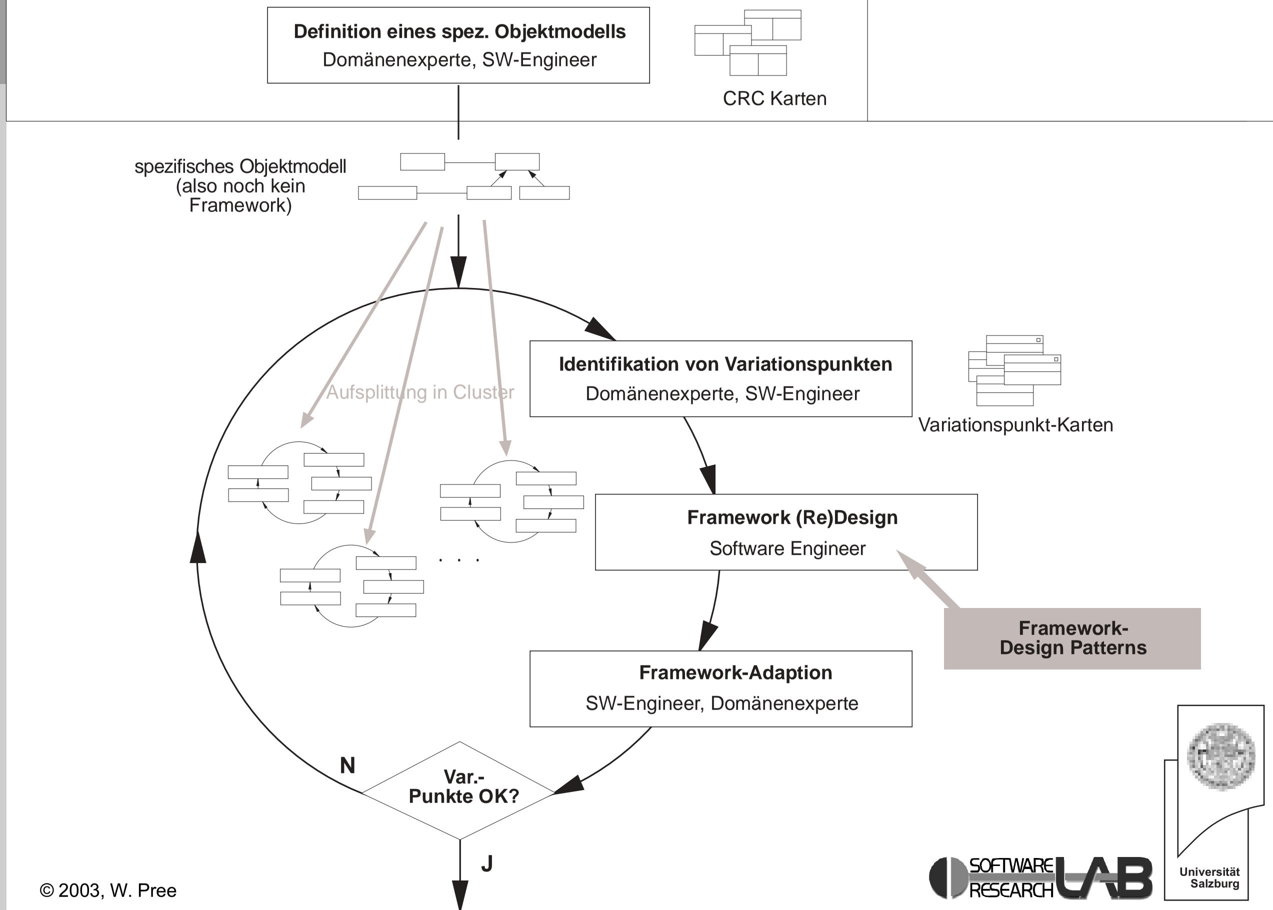
OO:
Cluster-Modell (Bertrand Meyer)



zB Design und Implementierung können sich in verschiedenen Clustern überlappen

© 2003, W. Pree

erweitertes Cluster-Modell (B. Meyer) bei Framework-Entwicklung



© 2003, W. Pree

Dynamische Konfiguration von Frameworks

- White-Box :: Black-Box Plug&Work
- Meta-Information

SOFTWARE RESEARCH LAB
Universität Salzburg

© 2003, W. Pree

Fallstudie Rounding Policy

© 2003, W. Pree
SOFTWARE RESEARCH LAB Universität Salzburg

Rounding Policy

Kontext: Wir nehmen an, es sei eine Klasse CurrencyConverter mit folgenden Eigenschaften bereits implementiert:

Der CurrencyConverter speichert intern eine Hauptwährung (zB US$) und eine Umrechnungstabelle:

1 US$=				
	Valuten (sell):	0.87	Valuten (buy):	0.92
	Devisen:	0.90	Devisen:	0.91
	Euro			

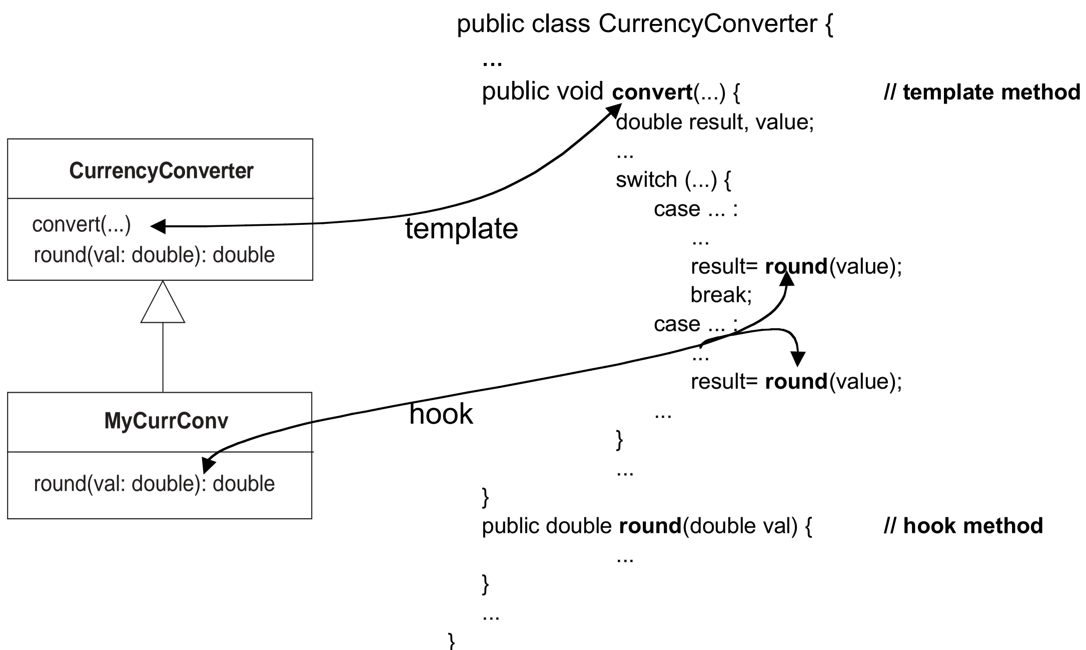
...

Flexibilitätsanforderung: Die Art, wie gerundet wird, soll flexibel einstellbar sein (auf 1, 10, 100, etc.; nur aufrunden; bestimmte Grenzwerte, etc.).

Man überlege sich zwei Entwürfe, wie diese Flexibilität erreicht werden kann.

© 2003, W. Pree
SOFTWARE RESEARCH LAB Universität Salzburg

Entwurf I (nur WB-Konfig.)

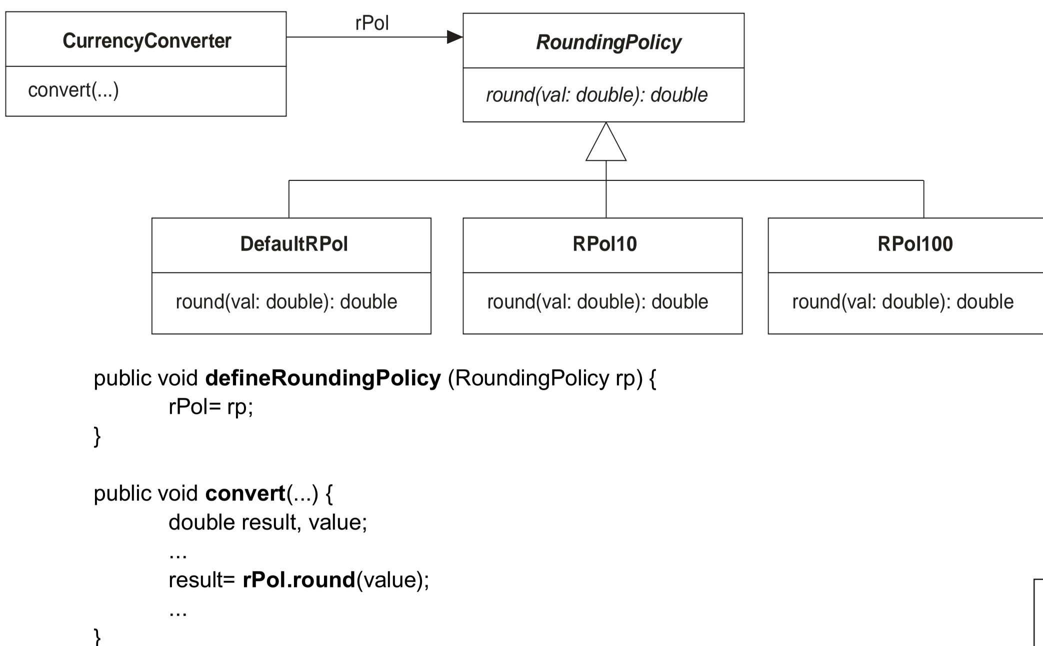



```
public class CurrencyConverter {
  ...
  public void convert(...) {              // template method
        double result, value;
        ...
        switch (...) {
            case ... :
                ...
                result= round(value);
                break;
            case ... :
                ...
                result= round(value);
            ...
        }
        ...
  }
  public double round(double val) {       // hook method
        ...
  }
  ...
}
```


© 2003, W. Pree

SOFTWARE RESEARCH LAB
Universität Salzburg

Entwurf II (BB&WB-Konfig.)



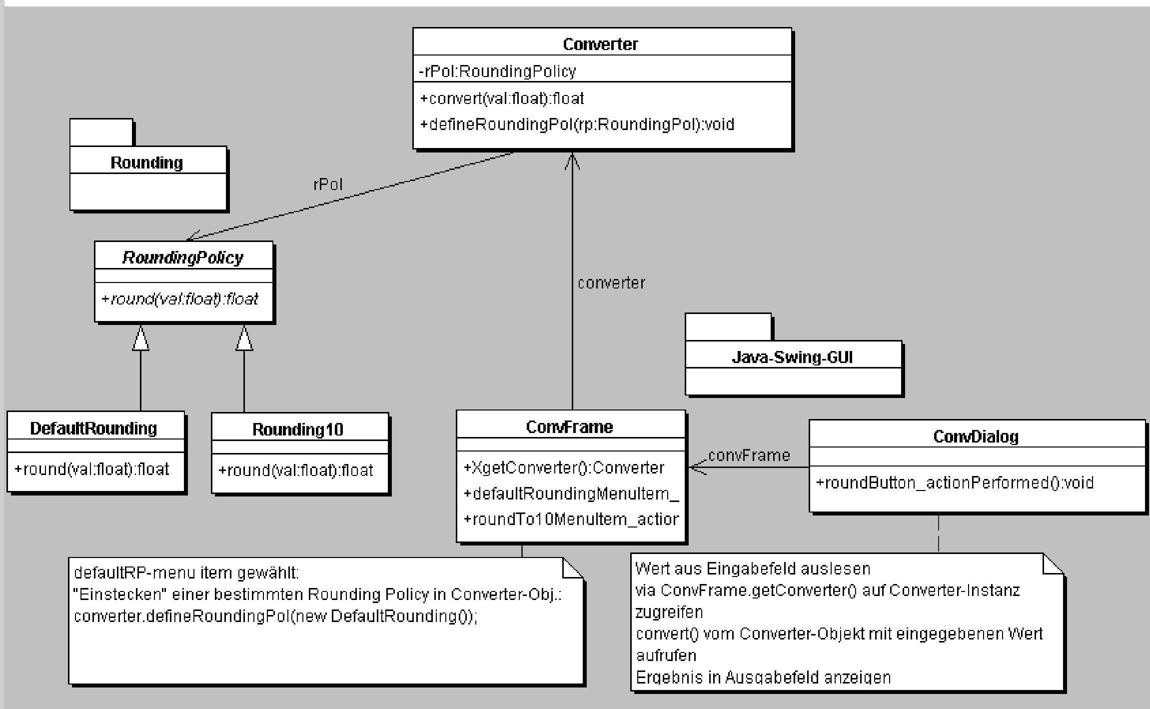

```
public void defineRoundingPolicy (RoundingPolicy rp) {
        rPol= rp;
}

public void convert(...) {
        double result, value;
        ...
        result= rPol.round(value);
        ...
}
```


© 2003, W. Pree

SOFTWARE RESEARCH LAB
Universität Salzburg

Fachliche Klassen + GUI



© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Dynamische Erweiterung

Wie kann auf Basis des Entwurfs II eine Erweiterung um neue RoundigPolicy-Klassen zur Laufzeit erfolgen?

Überlegen Sie eine mögliche Benutzerschnittstelle und eine entsprechende Dynamic-Computing-Realisierung (siehe nachfolgender Abschnitt).

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Metaklassen als Basis für Dynamic Computing

© 2003, W. Pree

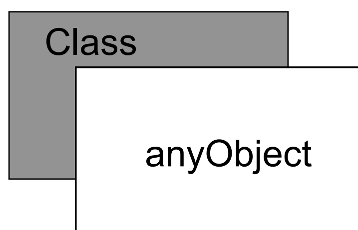
Einführung

- Metaklassen representieren Klassen in der aktuellen JVM sowie in .NET.
- Mit Metaklassen kann man Typen und Elemente einer Klasse (Methoden, ...) zur Laufzeit abfragen. Darüberhinaus kann man eine Klasse, deren Name als String vorliegt, instanzieren, eine Methode aufrufen und Arrays modifzieren (vergrößern, verkleinern).

© 2003, W. Pree

Klassenobjekte (I)

Jedes Objekt hat gleichsam ein **„Schattenobjekt“**, das eine Instanz der **Klasse Class** ist. Dieses Schattenobjekt kann Fragen über das eigentliche Objekt beantworten (zB welche Instanzvariablen es hat).



© 2003, W. Pree

Klassenobjekte (II)

Ein Klassenobjekt, also das Schattenobjekt, kann man auf verschiedene Art und Weise erhalten. (Nachfolgende Source-Code-Beispiele basieren auf java.reflect.*)

```
Class c = mystery.getClass();
```

```
TextField t = new TextField();
Class c = t.getClass();
Class s = c.getSuperclass();
```

```
Class c = Class.forName(strg);
```


© 2003, W. Pree

Metainformationen (I)

Beispiel: Eine Klasse, deren Namen nicht von vorneherein bekannt ist, muß instanziert werden.

nicht möglich:

```
String clName= "A";
Object anA= new clName;
```


© 2003, W. Pree

SOFTWARE RESEARCH LAB

Universität Salzburg

Metainformationen (II)

Lösung mittels Klasse Class:

```
String clName= "A";
Object anA;
try {
        Class aClass= Class.forName(clName);
        anA= aClass.newInstance();
} catch (Exception e) {
        System.err.print(e);
}
```


© 2003, W. Pree

SOFTWARE RESEARCH LAB

Universität Salzburg

Wozu *Dynamic Computing*?— Das Problem “FatWare”

Zahlreiche Programme sind Repräsentanten von ***Fatware*** (Niklaus Wirth, ETH Zürich), zB bei Desktop-Programmen:

- Spreadsheets mit >> 1000 Funktionen
- Office-Applikationen belegen “zig” MBs auf einer Hard-Disk und sehr viel im RAM
- HW wird langsamer schneller als Software langsamer wird
  (M. Reiser, IBM Zürich)

© 2003, W. Pree
SOFTWARE RESEARCH LAB Universität Salzburg

Dynamic Computing

stattdessen:

- Benutzer arbeitet mit **Basisversionen von Software**.
- **zusätzlich benötigte Komponenten** werden **bei Bedarf** ergänzt, indem sie **nachgeladen und dynamisch eingesteckt** werden
- Eine weitere Möglichkeit bieten **Push-Technologien zum Broadcasting von Updates**. Applikationen sind lokal bei den Clients gespeichert. Updates kommen von der Komponentensendestelle.

© 2003, W. Pree
SOFTWARE RESEARCH LAB Universität Salzburg

Überblick über Design Pattern Ansätze

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Design Patterns & Frameworks

Stimmung in der OO Community Anfang der 90er Jahre:

Frameworks sollen in den Mainstream

- Diskussion von Christopher Alexander's *Pattern Language* für Architektur
- Bruce Anderson's OOPSLA'91 Workshop *Towards an Architecture Handbook*
  - Teilnehmer: Framework-Experten
  - Begriff *Design Pattern* wird als Schlagwort kreiert. Ausgangspunkt: Erich Gamma's Beschreibung des Designs von "Mini-Frameworks" in ET++ im Rahmen seiner Dissertation (Juli 1991)

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Begriff *(Design)* Pattern

Laut Webster's Dictionary:

- *a person or thing so ideal to be worthy of imitation or copying*
- *a model, guide, plan, etc. used in making things*
- *an arrangement of form; design or decoration; as, wallpaper patterns, the pattern of a novel*
- *definite direction; as, behavior patterns*

Beispiele für Patterns im täglichen Leben:

- Anfahren eines Autos (Muster, wie Kupplung, Gaspedal und Schalthebel zu betätigen sind)
- Verkehrsregeln
- Verhaltensregeln (wie grüßt man; etc.)

© 2003, W. Pree

SOFTWARE RESEARCH LAB Universität Salzburg

Patterns für Softwareentwicklung

Allgemein akzeptierte Definition:

A design pattern is a solution of a problem in a certain context.

- *Algorithmen-Beschreibungen*
- *für OO SW: ??*

Gamma et al.:

".... frameworks are implemented in a programming language. .... In this sense **frameworks are more concrete than design patterns**. .... Mature frameworks usually reuse several design patterns."

© 2003, W. Pree

SOFTWARE RESEARCH LAB Universität Salzburg

Coding Patterns (I)

James Coplien's C++ Styles & Idioms

- beschränkt sich im wesentlichen auf C++ Tips & Tricks
- versucht die gravierendsten C++ Probleme auszumerzen (zB fehlende Garbage Collection)

Beispiel: ***Orthodox, canonical class form***

Eine Klassendefinition hat zu umfassen:

- default constructor
- assignment operator and copy constructor
- destructor

© 2003, W. Pree

Coding Patterns (II)

Zusammenfassende Beurteilung: Coding Patterns

- sind **auf der Ebene von Sprachkonstrukten** angesiedelt
- gibt es für nahezu alle Sprachen; oft implizit zB durch Programmierbeispiele oder Bibliotheken
- geben **keine Hinweise auf das anwendungsspezifische Design** von Klassen

© 2003, W. Pree

Framework Cookbooks (I)

Framework Cookbooks (vgl. Java-Tutorial, .NET-Hilfe) bestehen aus einer **Sammlung von Rezepten.** Diese

- beschreiben, **wie ein Framework an bestimmte Anforderungen angepaßt wird**
- enthalten kaum Design-Hinweise
- sind **idealerweise via Hypertextsystem** verfügbar

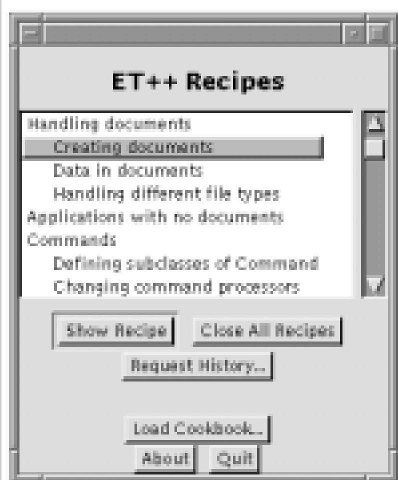
Typische Gliederung eines Rezeptes:

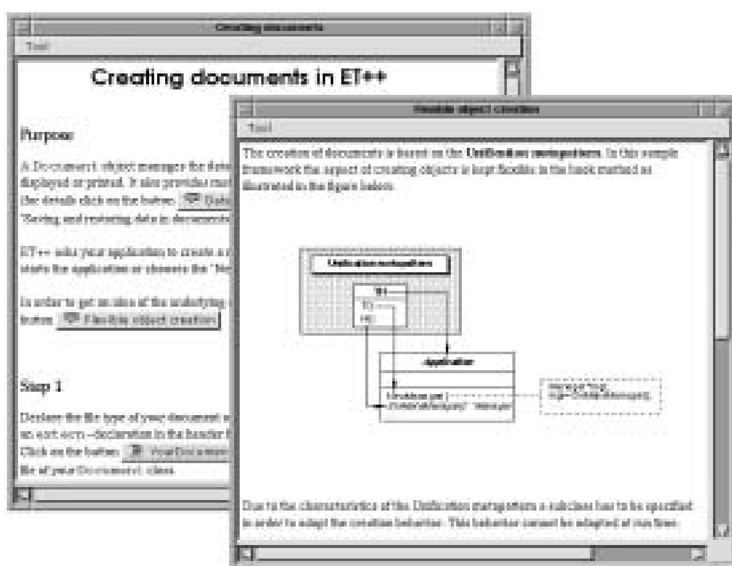
- *purpose*
- *steps how to do it*
- *source code examples*

© 2003, W. Pree

SOFTWARE RESEARCH LAB
Universität Salzburg

Framework Cookbooks (II)





Beispiel ET++, 1992

© 2003, W. Pree

SOFTWARE RESEARCH LAB
Universität Salzburg

Framework Cookbooks (III)

Beispiele für Framework Cookbooks

- **Java-Tutorial**
- **.NET-Hilfe**
- **Smalltalk Cookbooks** (VisualAge)
- MFC und .NET „Active“ Cookbooks (Anpassung des Frameworks durch Tools)

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Design-Pattern-Katalog (I)

Erich Gamma hat in seiner **Dissertation** (Univ. Zürich, 1991) Pionierarbeit geleistet, indem er **allgemein verwendbare Designprinzipien des GUI-Frameworks ET++** beschrieben hat.

Daraus ist das **Standardwerk zu Design-Patterns**, das sogenannte **Gang-Of-Four-** (GOF; Gamma, Helm, Johnson, Vlissides) **Buch** entstanden:

Design Patterns—Elements of Reusable OO Software

Addison-Wesley, 1995 (auch als CD verfügbar)

Dieses enthält **23 Katalogeinträge**; die meisten davon beschreiben das Design von weitgehend Domänen-unabhängigen Mini-Frameworks.

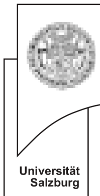
© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Design-Pattern-Katalog (II)

Grundidee: Informelle Beschreibung (Text, Klassen-/Objektdiagramme) **der jeweiligen Mini-Framework-Komponenten + deren Interaktion.**

Ein Katalogeintrag (ca. 8-12 Seiten) gliedert sich wie folgt:

- **Motivation:** Darstellung eines konkreten Szenarios, welches aufzeigt, warum ein Pattern sinnvoll ist. (informeller Text; Klassen-/Objektdiagramme)
- **Abstrakte Beschreibung** der Komponenten und deren Interaktion (informeller Text; Klassen-/Objektdiagramme)
- **Tips & Tricks** wie/wann das Pattern anzuwenden ist
- **Source-Code-Beispiele** (informeller Text; C++/Smalltalk)
- **Querreferenzen** zu anderen Katalogeinträgen



© 2003, W. Pree

Design-Pattern-Katalog (III)

Vorteile:

+ **effizientes Erlernen und Verstehen von Framework-Technologie** durch gut dokumentierte Mini-Frameworks
+ in Teams bildet sich ein **einheitliches Design-Vokabular** heraus (Factory, Composite, State, etc.)
+ wesentlich **verbesserter C++ Codierstil**

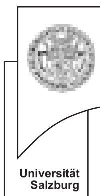
Nachteile

– Tendenz, **zu komplexe Systeme** zu entwerfen:

Flexibilität um der Flexibilität willen

Beispiel: Marketing-Support-System mit 3000 Klassen, wo 200 ausreichend wären.

– Verwirrung durch **zu viele ähnliche Katalogeinträge**



© 2003, W. Pree

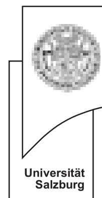
Essentielle Design Patterns (I)

Grundidee: Beschreiben die wenigen Konstruktionsprinzipien (Metapatterns), die den GoF-Katalogeinträgen zugrundeliegen.
Viele GoF-Patterns beruhen auf ein und demselben Konstruktionsprinzip und unterscheiden sich lediglich durch die Namen der Methoden bzw. Klassen.

Für jedes **Metapattern** werden angegeben:

- Grad der Flexibilität (zB zur Laufzeit änderbar oder nicht)
- typischer Aufbau der Methoden
- Möglichkeiten für Objektkompositionen

Wolfgang Pree: ***Design Patterns for OO Software Development***
Addison-Wesley, 1995

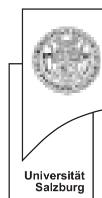


© 2003, W. Pree

(siehe auch Buch und CD)

Design-Pattern-Katalog

- Factory Method
- State, Null Object, Observer, Strategy
- Composite, Decorator



© 2003, W. Pree

Allgemeines

Katalogeinträge im GoF-Buch (Gamma, et al., 1995) ...

- **erlauben, ein erfolgreiches Design**, das jemand anderer kreiert hat, **zu wiederholen**
- beschreiben eine **Lösung, die nicht offensichtlich ist**
- definieren ein **Vokabular**, um über Design zu sprechen
- sind **im informellen Stil abgefasst** (Text + OMT-Diagramme + Programm-Code) und diskutieren Vor- und Nachteile eines bestimmten Designs

Was Katalogeinträge nicht sind:

- ein Klassendiagramm mit einem Namen
- eine neue Art, ein Problem zu lösen
- eine Möglichkeit, Code wiederzuverwenden

© 2003, W. Pree

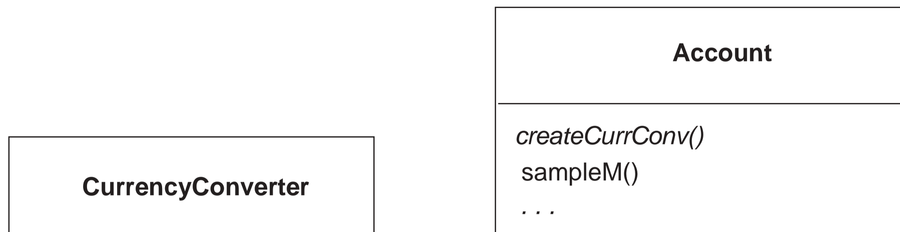
Factory Method Pattern

© 2003, W. Pree

Factory Method (I)

Intent: "Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses." (Gamma et al., 1995)

Motivation example:



© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Factory Method (II)

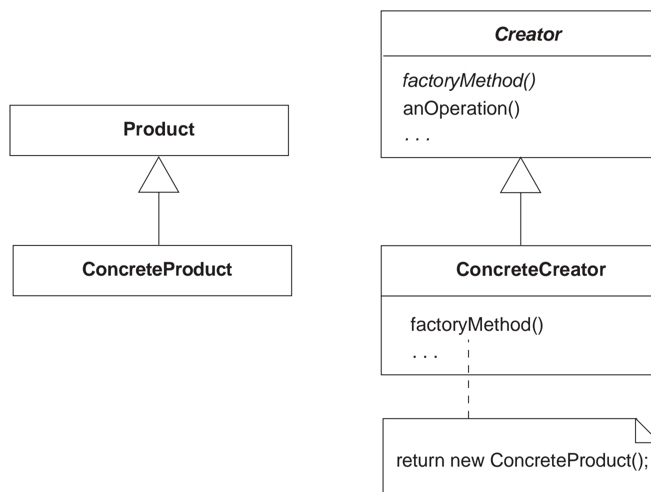
Applicability: "Use the Factory Method pattern when

- a class can't anticipate the class of objects it must create
- a class wants its subclasses to specify the objects it creates
- classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate" (Gamma et al., 1995)

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Factory Method (III)

Structure: (aus Gamma et al., 1995)



Consequences:

- enables framework to instantiate “abstract” classes
- requires creating subclasse to change product

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

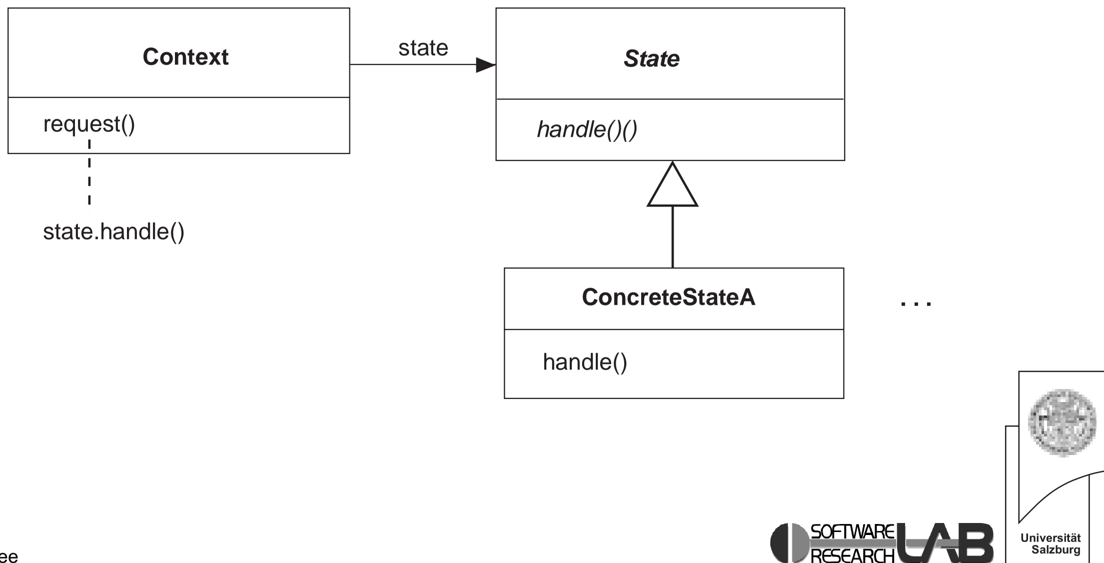
State Pattern

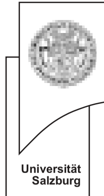
© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

State (I)

Intent: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Structure:





© 2003, W. Pree

SOFTWARE RESEARCH LAB

State (II)

Applicability:

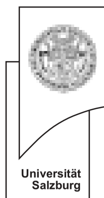
- an object's behavior depends on its state
- operations have conditional statements that depend on the objects state

Consequences:

- easy to add new states
- makes state transitions and instance variables associated with a state explicit
- increased number of classes

Implementation:

- who defines the state transitions
- creating and destroying state objects



© 2003, W. Pree

SOFTWARE RESEARCH LAB

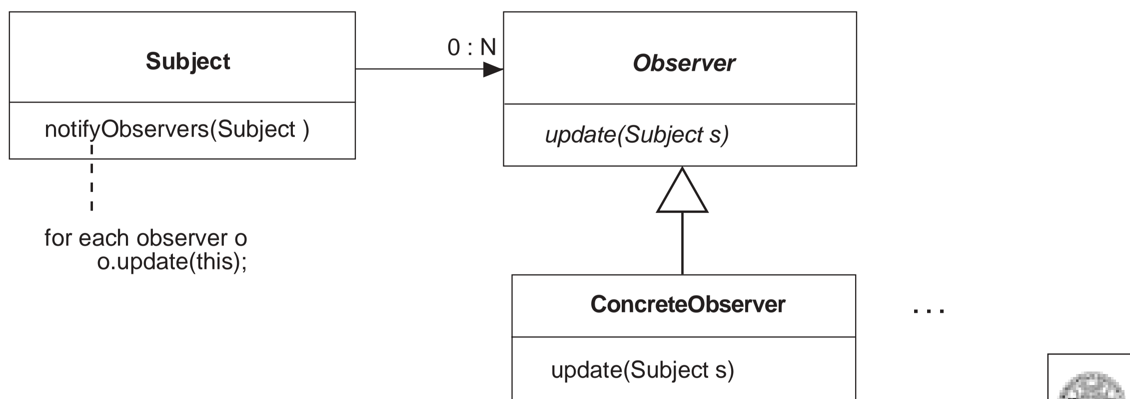
Observer
Pattern

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Observer (I)

Intent: Define a dependency between objects so that when one object changes state then all its dependents are notified

Structure:



© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Observer (II)

Applicability:

- when a change to one object requires changing others
- decouple notifier from other objects

Consequences:

- abstract coupling of subject and observer
- unexpected updates, update overhead

© 2003, W. Pree

Fallstudien:

Design Patterns auffinden und anwenden

© 2003, W. Pree

Observer und Factory Method

- Wie zeigt sich das Observer-Pattern in den .NET Frameworks?
- Wie zeigt sich das Factory Method-Pattern in Swing?
- Wie könnte das Factory Method Pattern in der Fallstudie Rounding Policy angewendet werden?

© 2003, W. Pree
SOFTWARE RESEARCH LAB Universität Salzburg

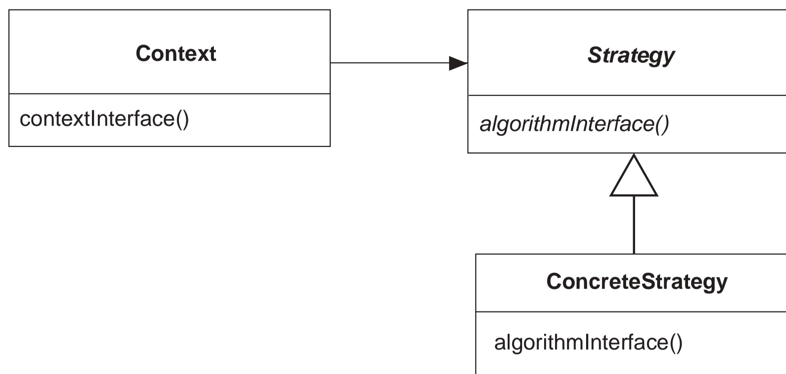
Strategy Pattern

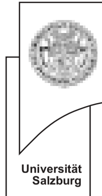
© 2003, W. Pree
SOFTWARE RESEARCH LAB Universität Salzburg

Strategy (I)

Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Structure:





© 2003, W. Pree

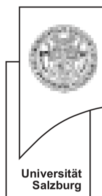
Strategy (II)

Applicability:

- when object should be configurable with algorithm to be used
- need to dynamically reconfigure

Consequences:

- a choice of implementations with different time and space tradeoffs
- context becomes free of of implementation details
- strategies encapsulate private data of algorithms
- increased number of objects



© 2003, W. Pree

Composite Pattern

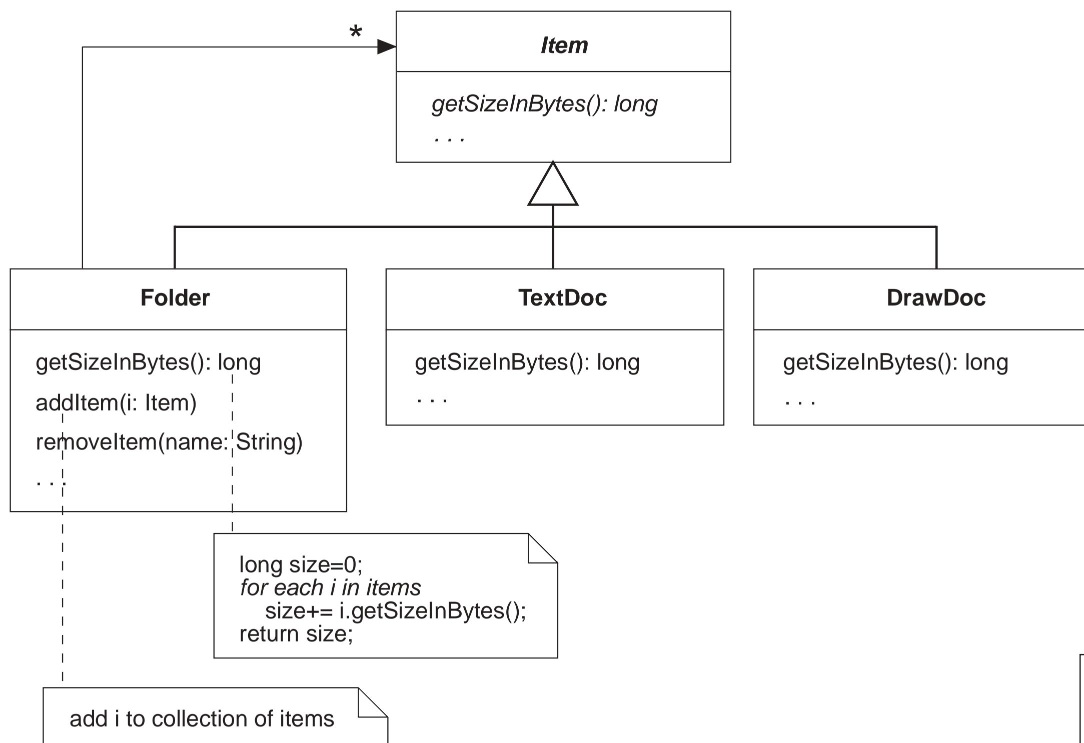
© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Composite (I)

Intent: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and composistions of objects uniformly.

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Composite – Beispiel



© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Composite (II)

Applicability:

- need to assemble objects out of primitive objects
- represent part-whole hierarchies

Consequences:

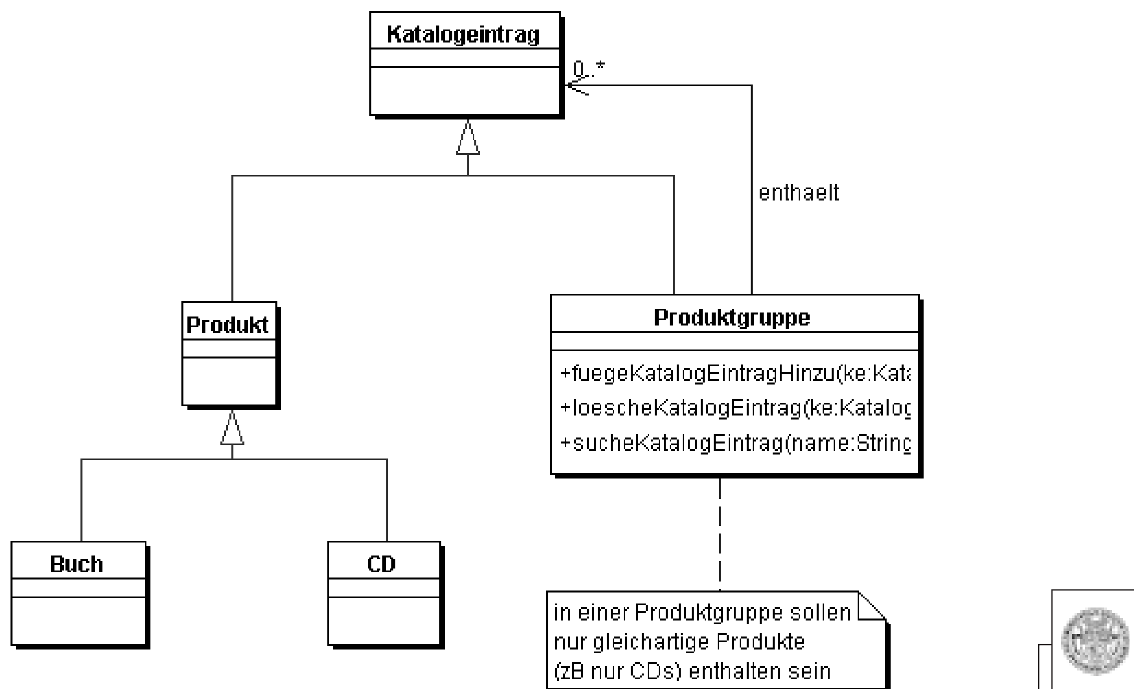
- it is easy to add new primitive objects that can be assembled into composites
- black-box reuse

Implementation:

- navigating and finding children in a composite
- back pointers to parent?

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Composite – Beispiel aus einer OO Modellierung



© 2003, W. Pree

weitere GoF-Patterns

Die verbleibenden GoF-Patterns werden im Buch/auf der CD präsentiert.

© 2003, W. Pree

Essentielle Design Patterns

- Template und Hook-Methoden
- Unification Pattern versus Separation Pattern
- „Rekursive“ Kombinationen

© 2003, W. Pree

Grundlegende Begriffe (I)

Methoden in Klassen lassen sich in folgende **zwei Kategorien** einteilen:

Template-Methoden <=> frozen spots
Hook-Methoden <=> hot spots

Template-Methoden definieren ein **komplexes Default-Verhalten**, welches **durch Hook-Methoden angepaßt** wird.

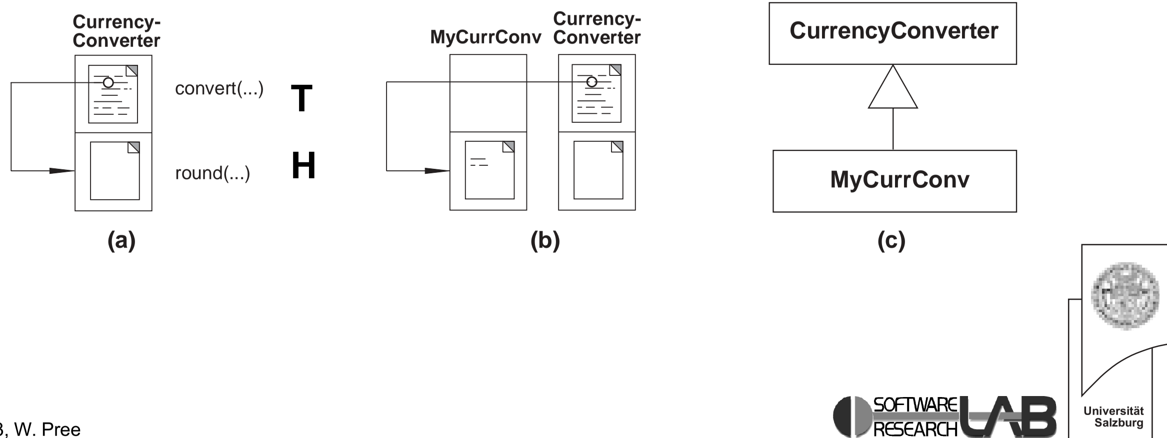
(Anm.: Template-Methode hat mit C++ Template nichts zu tun!)

© 2003, W. Pree

Grundlegende Begriffe (II)

Sind in einer Klasse sowohl Template- (T) als auch zugehörige Hook- (H) Methoden vereint, wird das **Verhalten der Template-Methoden durch Überschreiben der Hook-Methoden in Unterklassen angepaßt.**

Beispiel:



© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

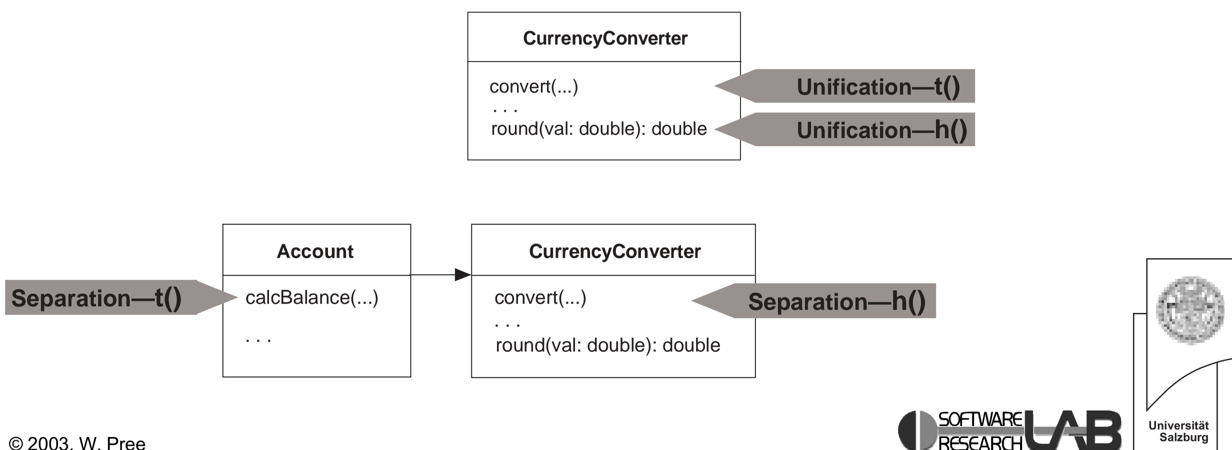
Grundlegende Begriffe (III)

Template-Klasse T := Klasse, die Template-Methode enthält

Hook-Klasse H := Klasse, die zugehörige Hook-Methode(n) enthält

Anmerkung:

Was ein T und H ist hängt vom Kontext ab:

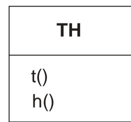


© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Essentielle Design Patterns (I)

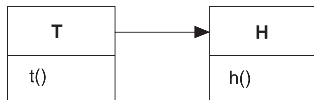
Mögliche Kombinationen von Template- und Hook-Klassen:

- Unification



  - Adaptionen nur durch **Überschreiben der Hook-Methode(n)** möglich.
  - Adaptionen benötigen daher einen Neustart der Anwendung.

- Separation

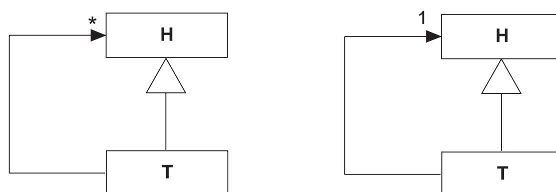


  - Das Verhalten eines T-Objekts kann durch **Komposition** verändert werden, nämlich durch **Einstecken eines spezifischen H-Objekts**.
  - **Adaptionen sind daher zur Laufzeit möglich.**

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Essentielle Design Patterns (II)

- Rekursive Kombinationen



  - **Aufbau direkter, azyklischer Graphen** wird möglich.
  - **Methodenaufrufe werden** aufgrund einer bestimmten Methodenstruktur **automatisch weitergeleitet.**

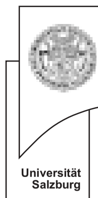
=> Die Spielwiese für Adaptionen durch Komposition wird vergrößert.

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Separation Pattern (I)

Das Separation Pattern entspricht der abstrakten Kopplung zweier Klassen. Die Template- und Hook-Klassen können auf verschiedene Arten miteinander gekoppelt sein:

- Typischerweise verfügt T über eine **Instanzvariable** vom statischen Typ H. Ein spezifisches H-Objekt wird quasi in ein T-Objekt durch Aufruf einer Methode setH(...) “eingesteckt”.
- Eine oder mehrere Methoden eines T-Objektes können über einen **Parameter** vom statischen Typ H verfügen. Somit erfolgt die abstrakte Kopplung der beiden Klassen nur temporär, nämlich während der Ausführung solcher Methoden.
- Die abstrakte Kopplung erfolgt durch eine **globale Variable** vom statischen Typ H.



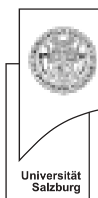
SOFTWARE RESEARCH LAB

© 2003, W. Pree

Separation Pattern (II)

Beispiele:

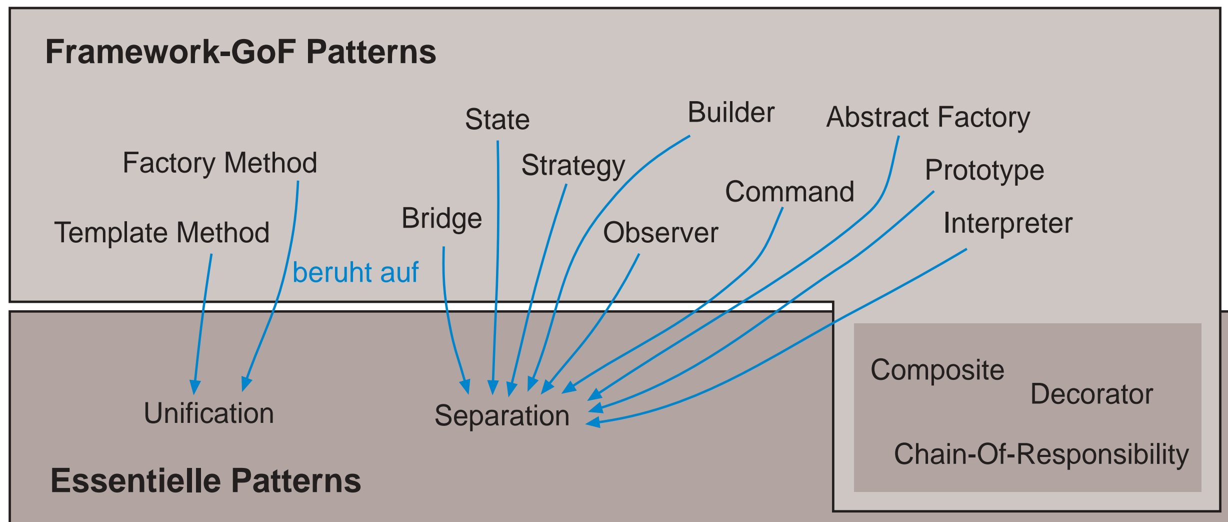
- zahlreiche GoF-Patterns
- Rounding-Policy im Currency-Converter
- Container-LayoutManager in Java-Swing

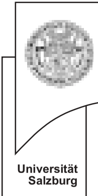


SOFTWARE RESEARCH LAB

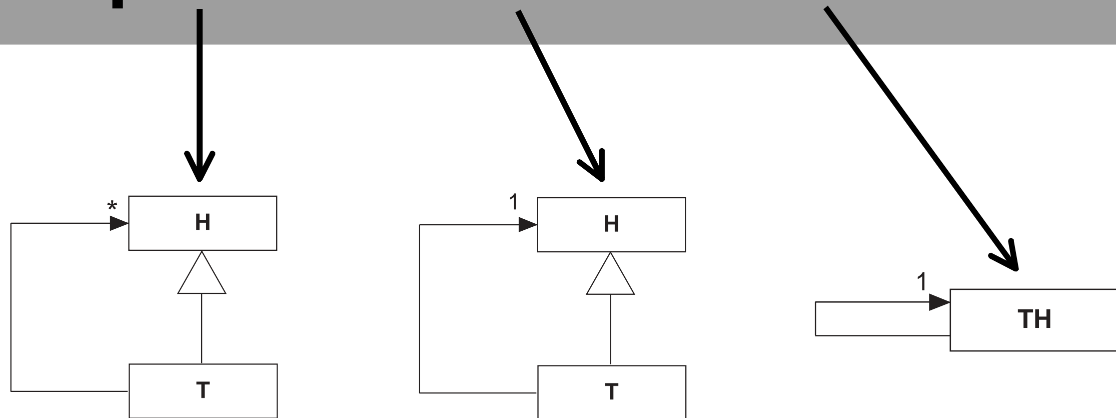
© 2003, W. Pree

Essentielle Patterns ⇔ GoF-Patterns



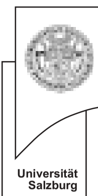


Composite–Decorator–COR

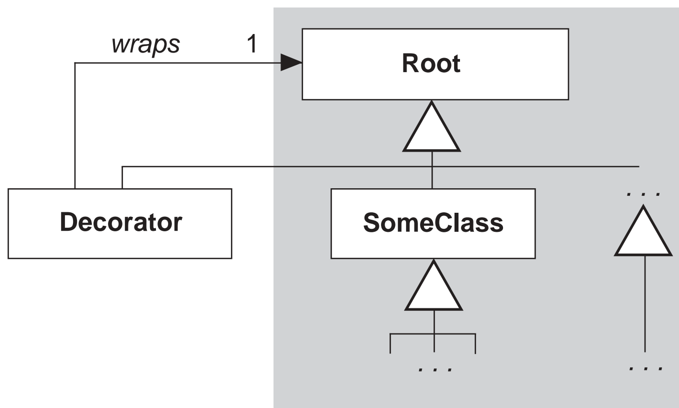


Decorator (siehe Buch/CD):

- zur Reduzierung des „Gewichts“ von abstrakten Klassen, die nahe der Wurzel der Klassenhierarchie sind
- als Alternative zu mehrfacher Vererbung

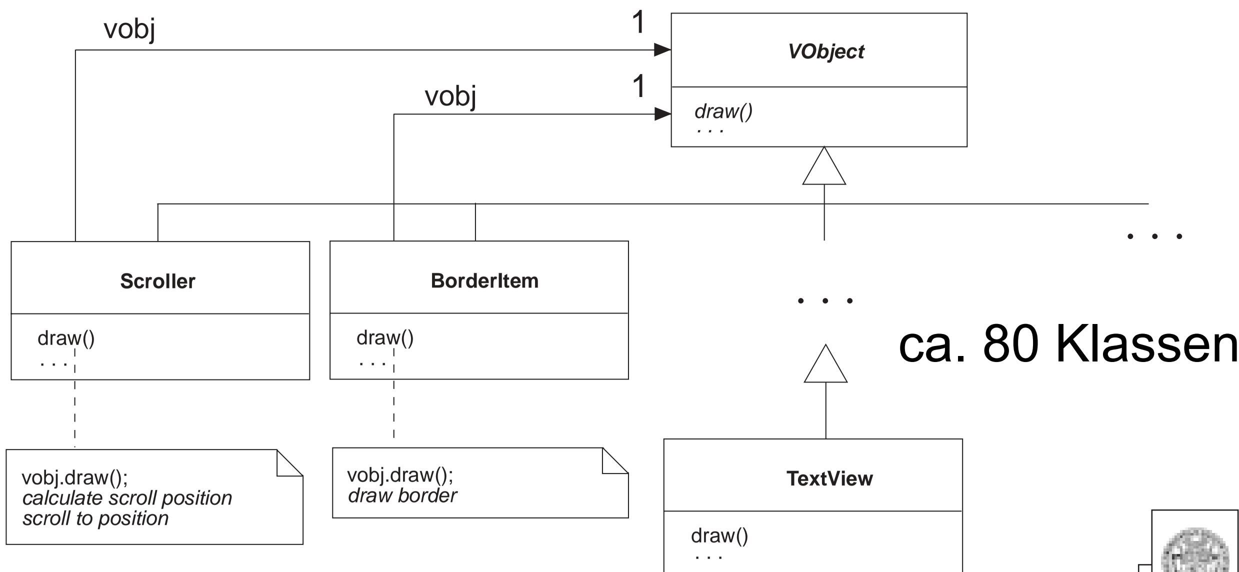


Decorator



© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Decorator–Beispiel (ET++)



© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Patterns @ Work

- kleine Frameworks (= Framelets); abstrakte Kopplung durch Namenskonventionen (Fallstudie: ListX)

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

ListX Framework

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

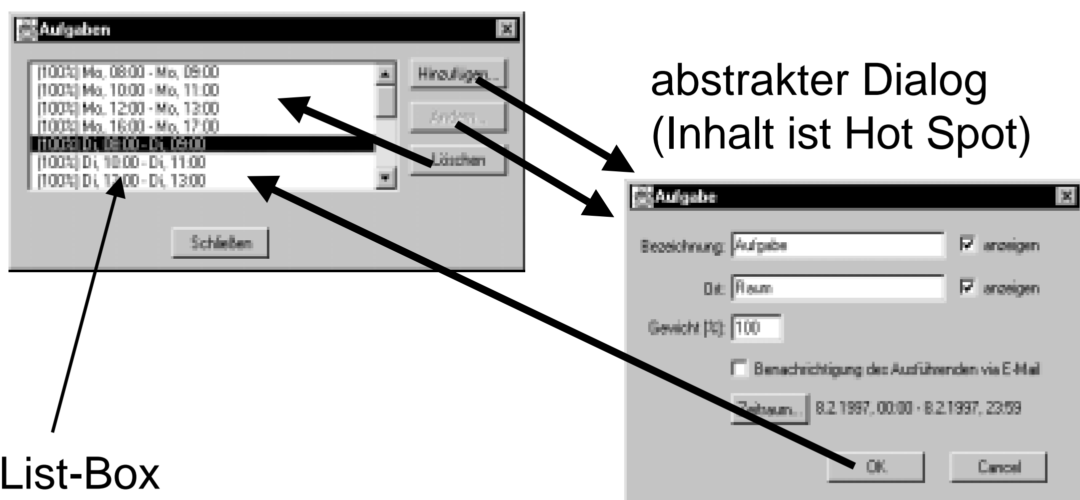
Ziele der ListX-Fallstudie

- Erkennen, daß sich Frameworks auch im kleinen Rahmen anwenden lassen und die Wiederverwendbarkeit verbessern können
- ein Framework von Grund auf entwerfen und entwickeln
- sehen, wie leicht ein gutes Framework anpassbar ist
- Metainformationen zur wesentlichen Erleichterung der Wiederverwendbarkeit einsetzen
- Verpacken eines Frameworks als Komponente

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

ListX Framework (I)

Interaktion zw. Framework-Komponenten:



abstrakter Dialog (Inhalt ist Hot Spot)

List-Box

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

ListX Framework (II)

Modellierung:

- Entwurf der Klassen und der Interaktion zwischen deren Instanzen
- Identifikation der Parameter des Frameworks

© 2003, W. Pree

ListX Framework (III)

Implementierung:

- Automatisierung der Anpassung durch Namenskonvention („Fetching“ von Daten zwischen Listeneintrag und Dialog)
- Verpackung als wiederverwendbare Komponente

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

OOAD
Richtlinien & Tips

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Metriken (I)

- Klassen:
  Anzahl der Methoden: > 5
  < 30
- Methoden:
  durschnittliche Methodenlänge:
  10 – 20 LOC

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Metriken (II)

- System:
  Anzahl der zentralen abstrakten Klassen: < 10

- Vererbungstiefe: < 15

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

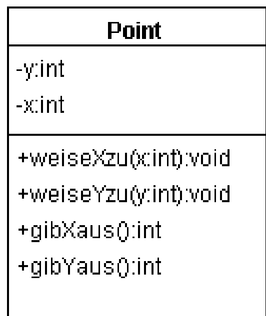
UML Tools

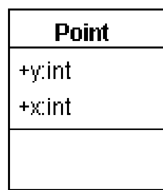
Trotz der Empfehlung der Tool-Hersteller, die Entwicklung möglichst auf UML abzustützen (OOAD; Code-Generierung), hat es sich in der Praxis bewährt, UML-Diagramme lediglich für eine überblicksmäßige Visualisierung der wichtigen Aspekte eines Systems zu benutzen.

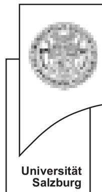
© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Information Hiding

Mittels Zugriffsrechten werden Implementierungsdetails vor dem Benutzer der Klasse versteckt und gesichert, daß Objekte immer in konsistenten Zuständen sind. Das ist bei einfachen Klassen nicht immer nötig:



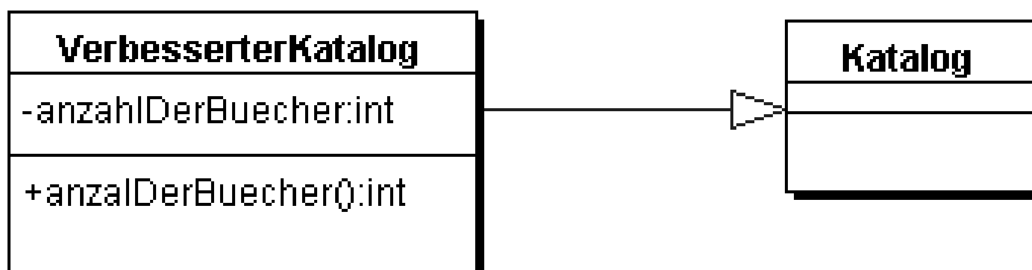


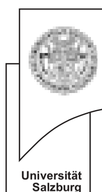


© 2003, W. Pree

Vererbung—nicht immer!

Vererbung sollte nicht verwendet werden, um Klassen zu korrigieren oder Implementierungen zu verbessern.

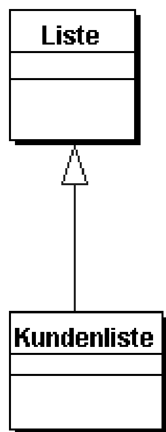


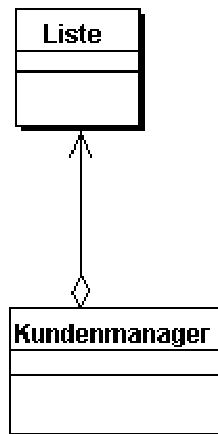


© 2003, W. Pree

Is-A versus Has-A

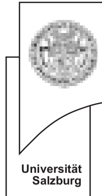
Vererbung (Is-A) sollte keinesfalls verwendet werden, um „Has-A-Beziehungen“ (Associations oder Aggregations) zu modellieren.





© 2003, W. Pree

SOFTWARE RESEARCH LAB



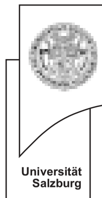
Abstrakte Klassen (I)

30% - 40% der Methoden, die in eine abstrakte Klasse herausfaktorisiert werden sollten, sind „offensichtlich“.

Mietgegenstand
-belegungsPlan:Vector
+reserviere(von:Date,bis:Date):boolean +istFrei(von:Date,bis:Date):boolean +storniere(von:Date,bis:Date):boolean *+druckeRechnungFuer(kunde:Kunde):void* +infoZuAusstattung():void

© 2003, W. Pree

SOFTWARE RESEARCH LAB



Abstrakte Klassen (II)

Wird zuwenig „herausfaktorisiert“, so wird es schwierig, Halbfertigfabrikate zu implementieren.

Wird zuviel „herausfaktorisiert“, so werden die Unterklassen zu überladen bzw. mit unnötigem Verhalten belastet.

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Interfaces versus abstrakte Klassen

Gründe für Interfaces:

- Die kritische Masse an Methoden für eine Klasse wird nicht erreicht.
- Es werden keine Instanzvariablen oder keine konkreten Methoden gebraucht.
- Unabhängigkeit von der Klassenhierarchie

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

(Re-)Design-Strategien

© 2003, W. Pree

Allgemein

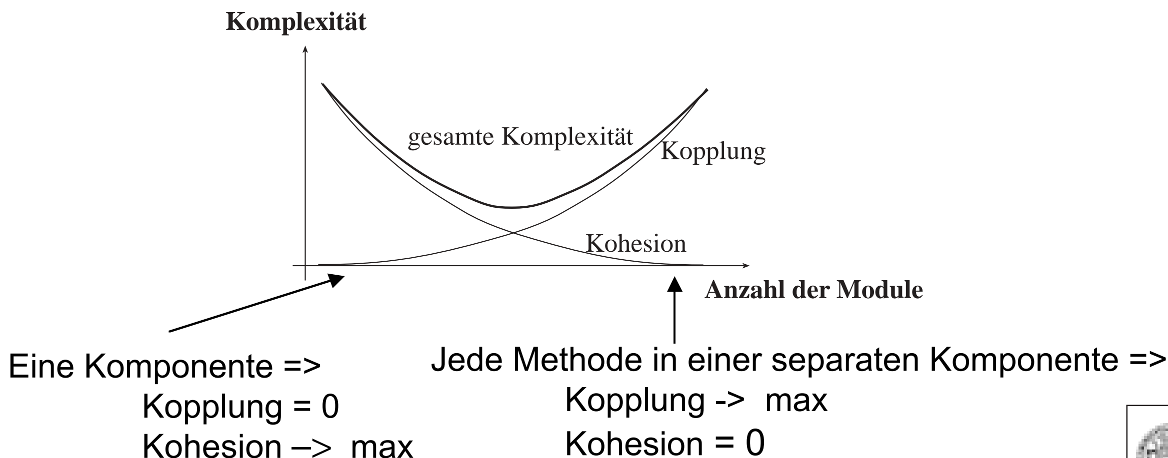
- Architektur-Zerlegung in Pakete/Module/Klassen/Komponenten/ Objekte prüfen
- Hots Spots (Flexibilität) von Frameworks prüfen

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Modularisierung/Strukturierung in der Theorie (I)

Gleichgewicht zwischen

- **Maximierung der Kohesion innerhalb einer Komponente (Modul/Klasse)**
- **Minimierung der Kopplung zwischen Komponenten**



Eine Komponente =>
Kopplung = 0
Kohesion –> max

Jede Methode in einer separaten Komponente =>
Kopplung -> max
Kohesion = 0

© 2003, W. Pree

Modularisierung/Strukturierung in der Theorie (II)

- Ausgeglichene Verteilung von „Responsibilities“ zwischen Komponenten
- Minimale Schnittstelle einer Komponente (erhöht die Kopplung innerhalb der K.)
  - enge Verbindung zw. Methoden und Instanzvariablen
  - keine redundanten Methoden
  - kleine Anzahl von Parametern
  - ausdrucksstarkes und konsistentes Namensschema
  - keine globalen Daten/Komponenten/Objekte

© 2003, W. Pree

Anwendung der Theorie auf OO Systeme

- grobkörniges Design von Klassenhierarchien
- Kopplung/Interaktion von Komponenten
- Kohesion innerhalb einer Komponente
- Evolution von Klassenhierarchien
- Hot Spots (Variation Points) von Frameworks

© 2003, W. Pree

Grobkörniges Design von Klassenhierachien (I)

Klassen können grob in **Familien, Teams und Subsysteme** unterteilt werden.

Klassenfamilien

- basieren auf abstrakten Klassen/Schnittstellen
  - Container Klassen (abstrakte Klasse Container)
  - GUI Komponenten (abstrakte Klasse Component)
- bottom-up oder top-down Entwicklung von Familien
- Wurzeln von Klassenfamilien sollten „leicht“ sein, insbesondere Datenrepräsentationen (Instanzvariable) vermeiden

© 2003, W. Pree

Grobkörniges Design von Klassenhierachien (II)

Klassenteams

- bestehen aus mehreren Klassenfamilien
  - ET++: Text team besteht aus Text, TextView undTextFormatter Familien
  - Container und Iterator Familien bilden ein Team
- werden als ganzes wiederverwendet
- sind abstrakt gekoppelt => Vermeidung von enger Kopplung

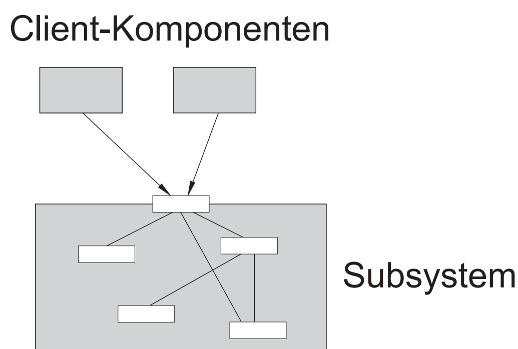
© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Grobkörniges Design von Klassenhierachien (III)

Subsysteme

Explizite Kapselung eines Klassenteams
(–> Facade Pattern)

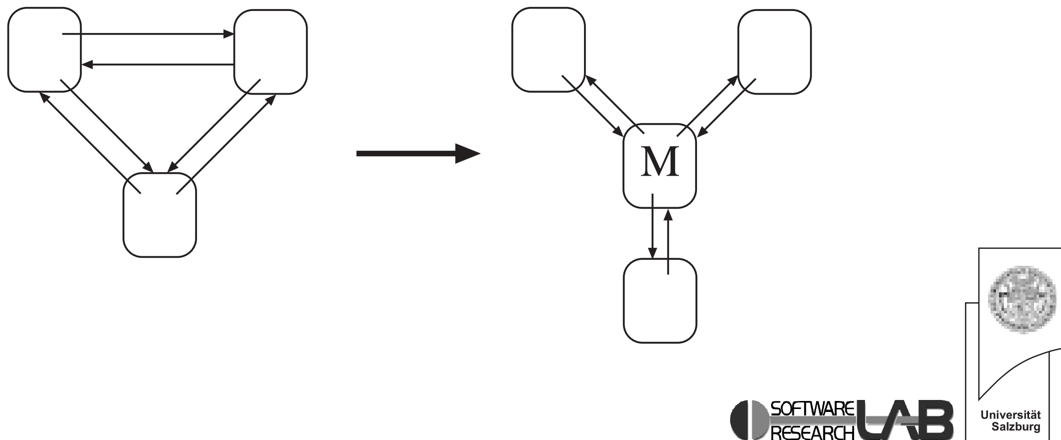
- reduziert die Kopplung zwischen Klassen eines Teams
- Definition einer zusätzlichen Abstraktion, um Wiederverwendung für Clients zu vereinfachen



© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Reduktion der Kopplung durch Mediatoren

Effekt des Mediator Patterns: Komponenten können eher unabhängig voneinander wiederverwendet werden.



© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

Erreichen von Kohesion innerhalb einer Komponente

Um die Kohesion qualitativ zu prüfen, sind folgende Fragen hilfreich

- Kann eine Komponente
  - verstanden
  - getestet

  werden, ohne zu wissen, wie sie in ein System eingebracht wird?

  siehe Fallstudie: diskrete Ereignissimulation
- Hat eine Komponente Seiteneffekte auf andere Komponenten?

© 2003, W. Pree
SOFTWARE RESEARCH LAB
Universität Salzburg

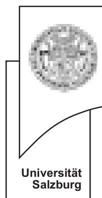
Evolution von Klassenhierarchien

Vertikale Reorganisationen

- Verschieben von gemeinsamen Eigenschaften/Verhalten hinauf in der Klassenhierarchie (-> abstrakte Klassen)
- Aufsplitten einer zu komplexen Klasse in eine Klassenfamilie
- Vermeiden des Überschreibens von zu vielen spezifischen Methoden, indem abstrakte Klassen eingeführt werden

Horizontale Reorganisationen

- Aufsplitten einer zu komplexen Klasse in ein Klassenteam/Subsystem
- Verschieben von Verhalten in „Strategie"-Klassen (Stategy/Bridge-Patterns)



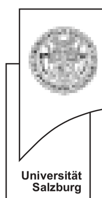
© 2003, W. Pree

Analyse von Variation Points (= Hot Spots)

Was macht die Qualität eines Frameworks aus?

Flexibilität um der Flexibilität willen—indem möglichst viele Patterns angewendet werden—führt zu unnötig komplexen Frameworks

- Flexibilität muß in der „richtigen Dosis" hinein
- Identifikation von Hot Spots stellt eine explizite Aktivität im Framework-Entwicklungsprozeß dar



© 2003, W. Pree

Literaturhinweise

© 2003, W. Pree

Literaturhinweise (I)

Gamma E., Helm R., Johnson R. and Vlissides J. (1995). Design Patterns—Elements of Reusable OO Software. Reading, MA: Addison-Wesley (auch als CD verfügbar)

Pree W. (1995). Design Patterns for Object-Oriented Software Development. Reading, Massachusetts: Addison-Wesley/ACM Press

Fontoura M., Pree W., Rumpe B. (2001) The UML-F Profile for Framework Architectures, Addison Wesley

Szyperski C. (1998) Component Software—Beyond Object-Oriented Programming, Addison-Wesley.

Fayad M., Schmidt D., Johnson R. (1999) Building Application Frameworks: Object-Oriented Foundations of Framework Design, Wiley

Fayad M., Schmidt D., Johnson R. (1999) Implementing Application Frameworks: Object-Oriented Frameworks at Work, Wiley

Fayad M., Schmidt D., Johnson R. (1999) Domain-Specific Application Frameworks: Manufacturing, Networking, Distributed Systems, and Software Development, Wiley

© 2003, W. Pree

Literaturhinweise (II)

Booch G., Jacobson I, Rumbaugh J.

- **Objektorientierte Analyse und Design. Mit praktischen Anwendungsbeispielen**
- **Das UML-Benutzerhandbuch (Unified Modeling Language User Guide)**
- **Unified Modeling Language Reference Manual,**
- **The Objectory Software Development Process,**
- **Addison-Wesley**

Österreich B. (2000) Erfolgreich mit Objektorientierung, Oldenburg Verlag

Balzert H. (2000) Objektorientierung in 7 Tagen, m. CD-ROM. Vom UML-Modell zur fertigen Web-Anwendung, Spektrum Akadem. Verlag

Gabriel R.P. (1996). Patterns of Software—Tales from the Software Community. New York: Oxford University Press

Wirth N, Gutknecht J. (1993) Project Oberon, Addison-Wesley.

© 2003, W. Pree

SOFTWARE RESEARCH LAB

Universität Salzburg