Software-Architekturen

Sommersemester 2002

Prof. Dr. Wolfgang Pree
Universität Salzburg
www.SoftwareResearch.net/SWA

Inhalt(I)

- Motivation & Wiederholung
  - OO Frameworks
  - Metainformationen (java.reflect.*) 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

Inhalt(II)

- Patterns @ Work
  - Fallstudie: Reengineering von bestehenden Systemen
  - Fallstudie: Modularisierung von OO Systemen
  - Fallstudie: Wiederverwendung „kleiner“ Komponenten

Inhalt(III)

- Architektur-Muster
  - Überblick
  - Software Architecture Analysis Method (SAAM)
- Hilfsmittel & Tips zu OOAD
  - Metriken
  - (Re-)Design-Strategien
    - Klassenfamilien, Teams, Subsysteme
    - Horizontale und vertikale Reorganisation von Klassenhierarchien

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

Albert Einstein

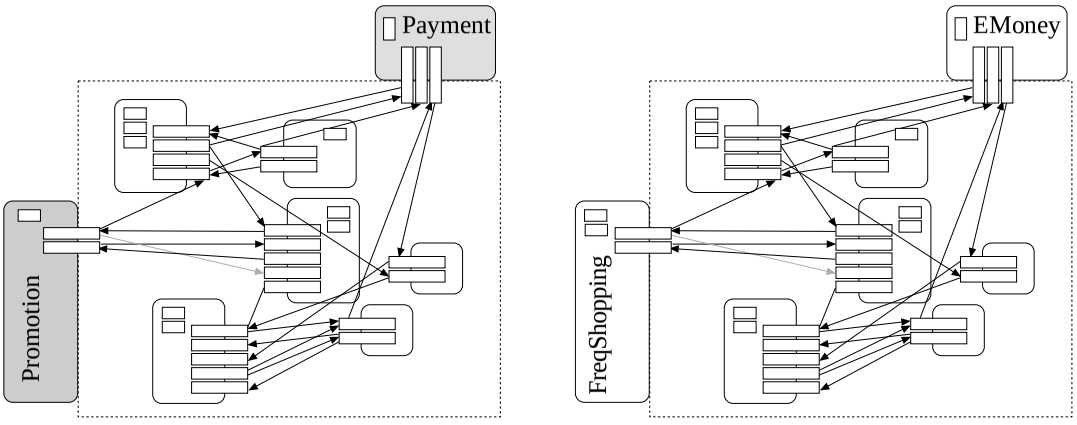
Frameworks

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

Konfiguration durch Einstecken von SW-Komponenten

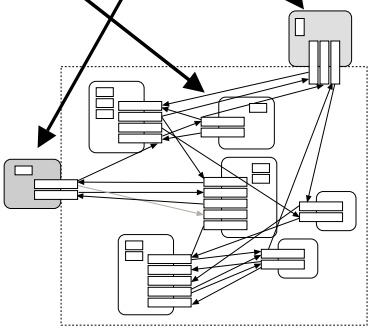


vor der Adaptierung

nach der Adaptierung

Frameworks— Dynamische Sicht

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



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.

Abstrakte Klassen

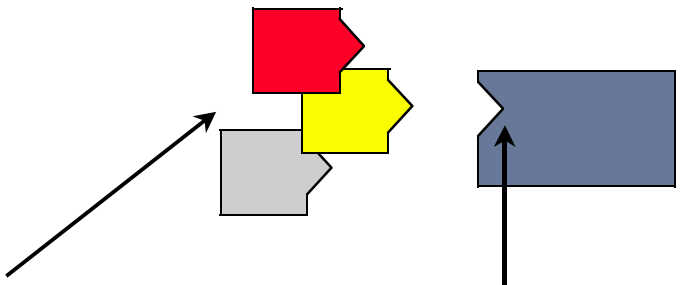
Abstrakte Klassen standardisieren die Schnittstelle (den „Stecker“) für Unterklassen.

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.

Polymorphismus

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

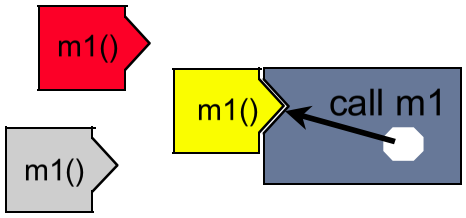


Objekte, die zu diesem Stecker kompatibel sind

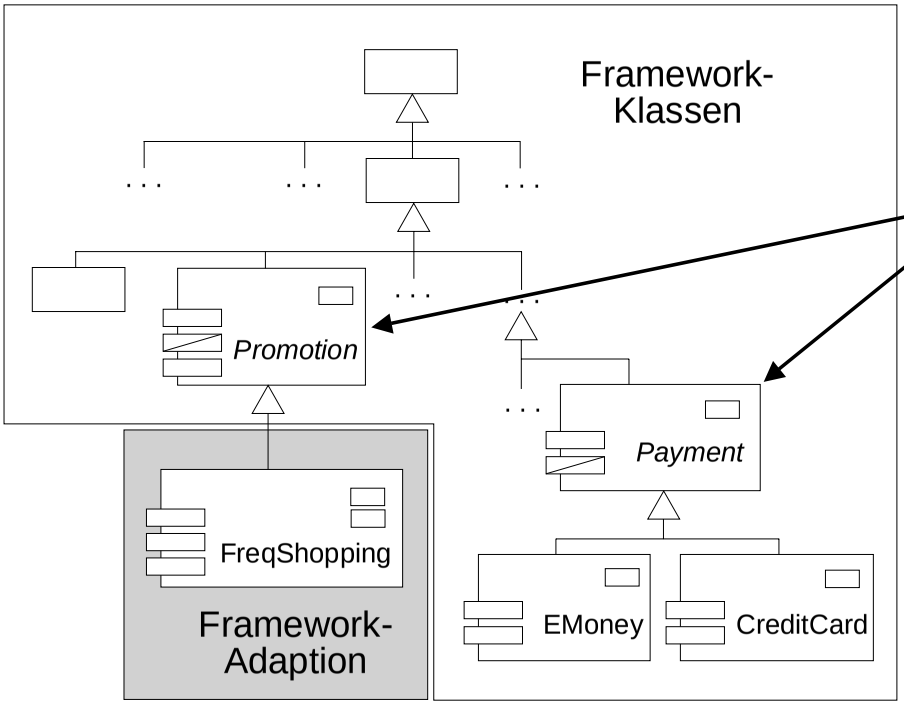
„Stecker“-Standard

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:

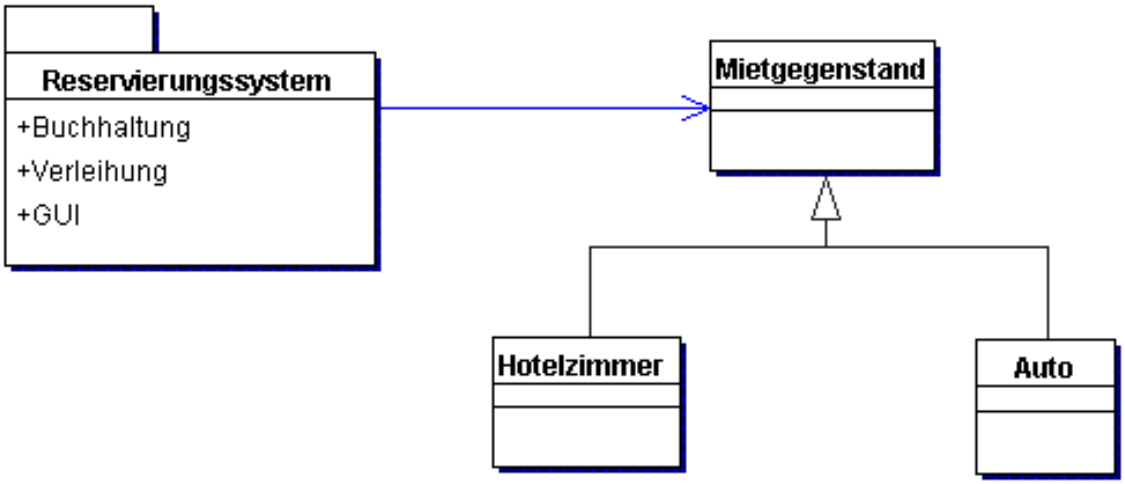


Frameworks— Statische Sicht



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

Beispiel Hotelreservierung



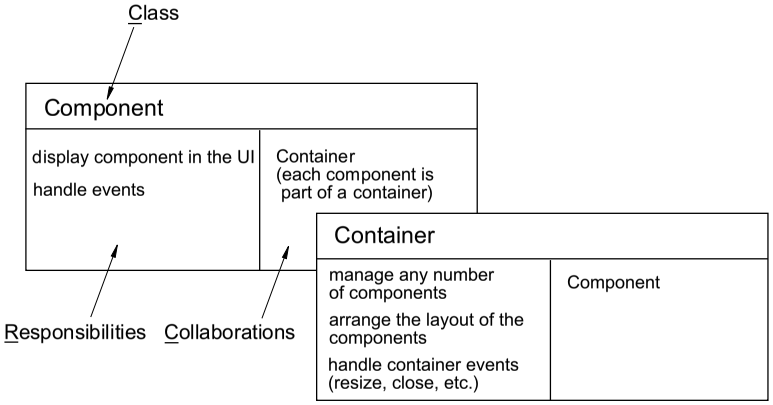
Swing-Framework?

Wie bilden die beiden abstrakten Klassen

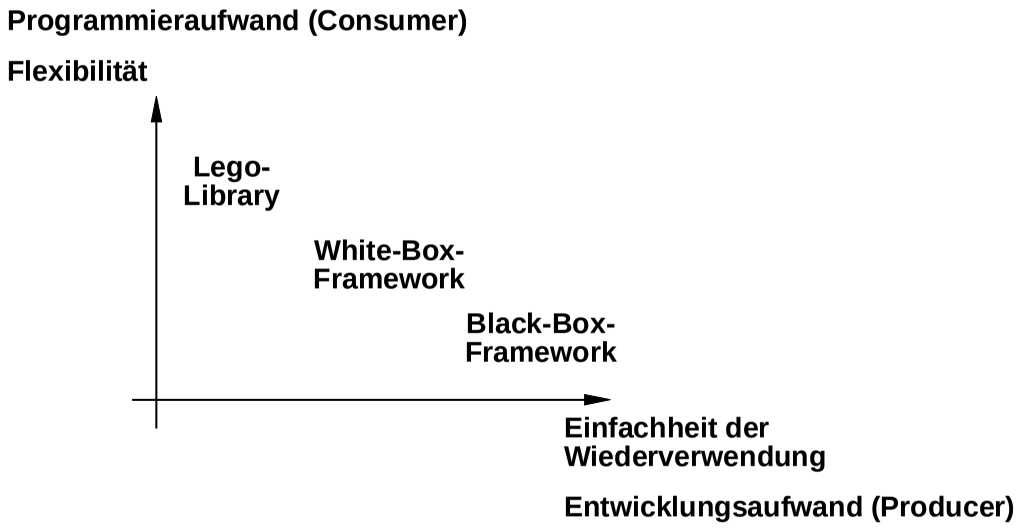
Component

Container

in Swing ein Framework?

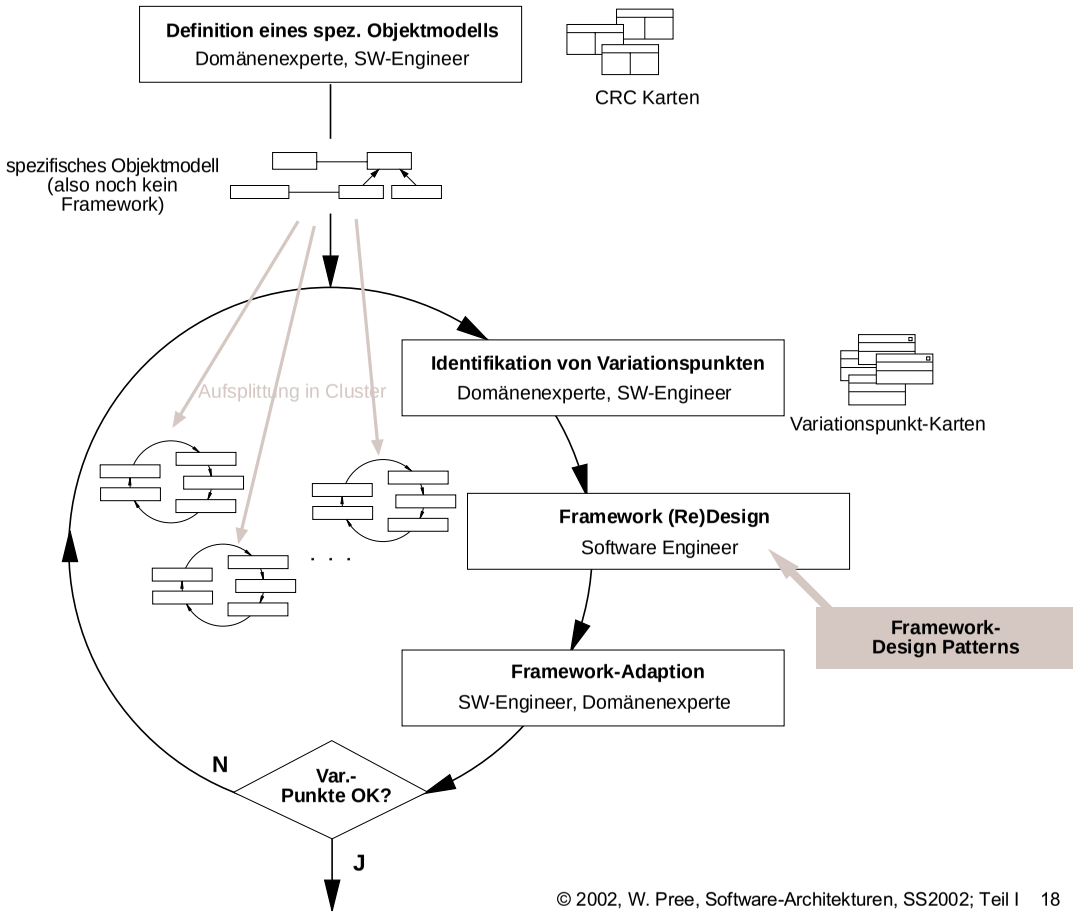


Legokasten/WB-/BB-Framework



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

erweitertes Cluster-Modell bei Framework-Entwicklung



Dynamische Konfiguration von Frameworks

- White-Box :: Black-Box Plug&Work
- Meta-Information – die java.reflect Bibliothek

Fallstudie Rounding Policy

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:

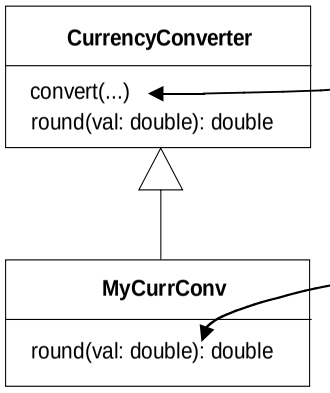
1 US$=	Valuten (sell):	0.87	Valuten (buy):	0.92
	Devisen:	0.90	Devisen:	0.91
	Euro			

...

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

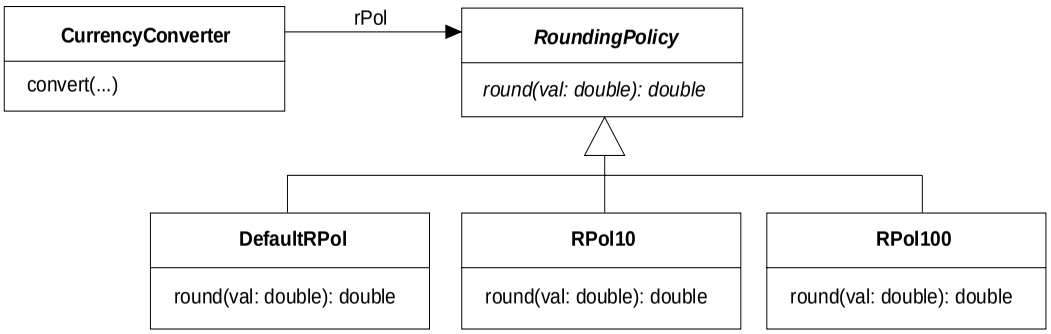
Man überlege sich zwei Entwürfe, wie diese Flexibilität erreicht werden kann.

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
      ...
  }
  ...
}
```

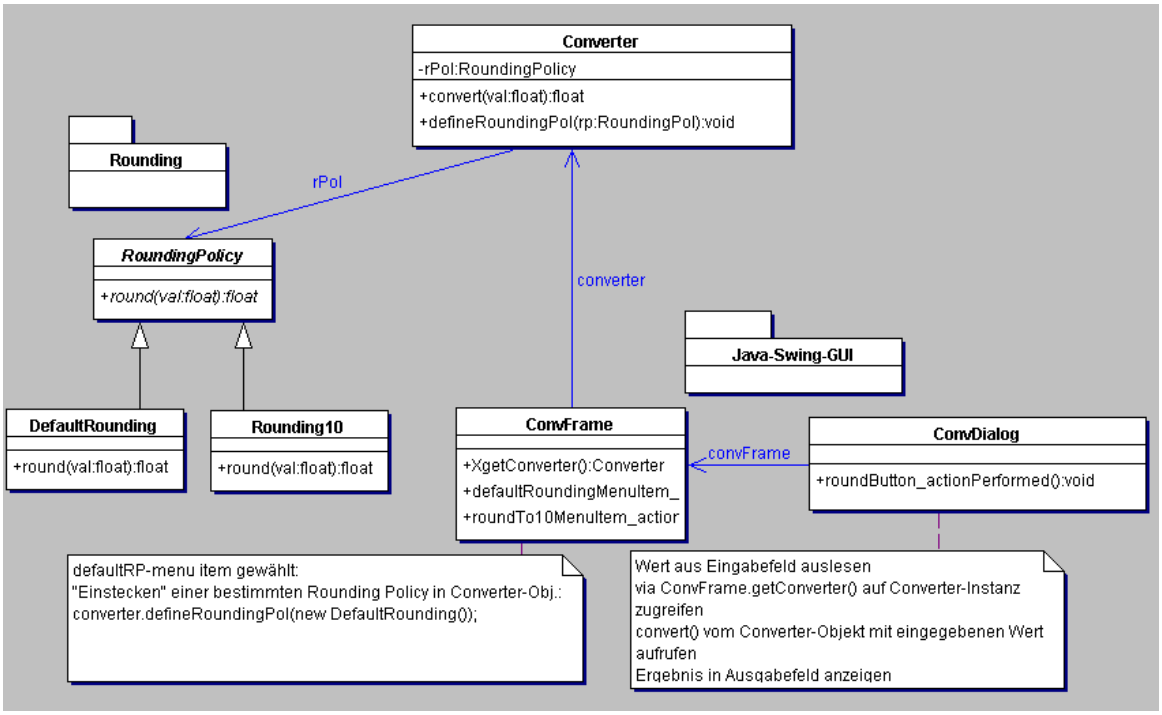
Entwurf II (BB&WB-Konfig.)




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

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


Fachliche Klassen + GUI



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).

java.reflect-Bibliothek

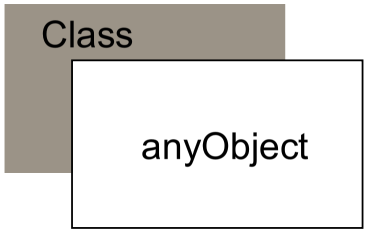
Metaklassen als Basis für Dynamic Computing

Einführung

- Metaklassen representieren Klassen in der aktuellen JVM.
- Mit Java‘s 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).

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).



Klassenobjekte (II)

Ein Klassenobjekt, also das Schattenobjekt, kann man auf verschiedene Art und Weise erhalten.

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

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

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


Metainformationen (I)

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

nicht möglich:

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

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);
}
```


Wozu *Dynamic Computing*?— Das Problem "FatWare"

Desktop-Applikationen sind Repräsentanten von ***Fatware*** (Niklaus Wirth, ETH Zürich):

- 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)

Dynamic Computing

stattdessen:

- Benutzer arbeitet mit **Basisversionen von Software**, die als Java-Applets/Applikationen via Internet/**Intranet (kann so schnell wie eine lokale Hard-Disk sein)** geladen werden.
- **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.

Links zum Java Tutorial (I)

<java-tutorial>/reflect/class/getName.html
<java-tutorial>/reflect/class/getModifiers.html
<java-tutorial>/reflect/class/getSuperclass.html
<java-tutorial>/reflect/class/getInterfaces.html
<java-tutorial>/reflect/class/isInterface.html
<java-tutorial>/reflect/class/getFields.html
<java-tutorial>/reflect/class/getConstructors.html
<java-tutorial>/reflect/class/getMethods.html

Links zum Java Tutorial (II)

<java-tutorial>/reflect/object/noarg.html
<java-tutorial>/reflect/object/arg.html
<java-tutorial>/reflect/object/get.html
<java-tutorial>/reflect/object/set.html
<java-tutorial>/reflect/object/invoke.html

Design Pattern = Entwurfsmuster

Überblick über Design Pattern Ansätze

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)

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.)

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."

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

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

Framework Cookbooks (I)

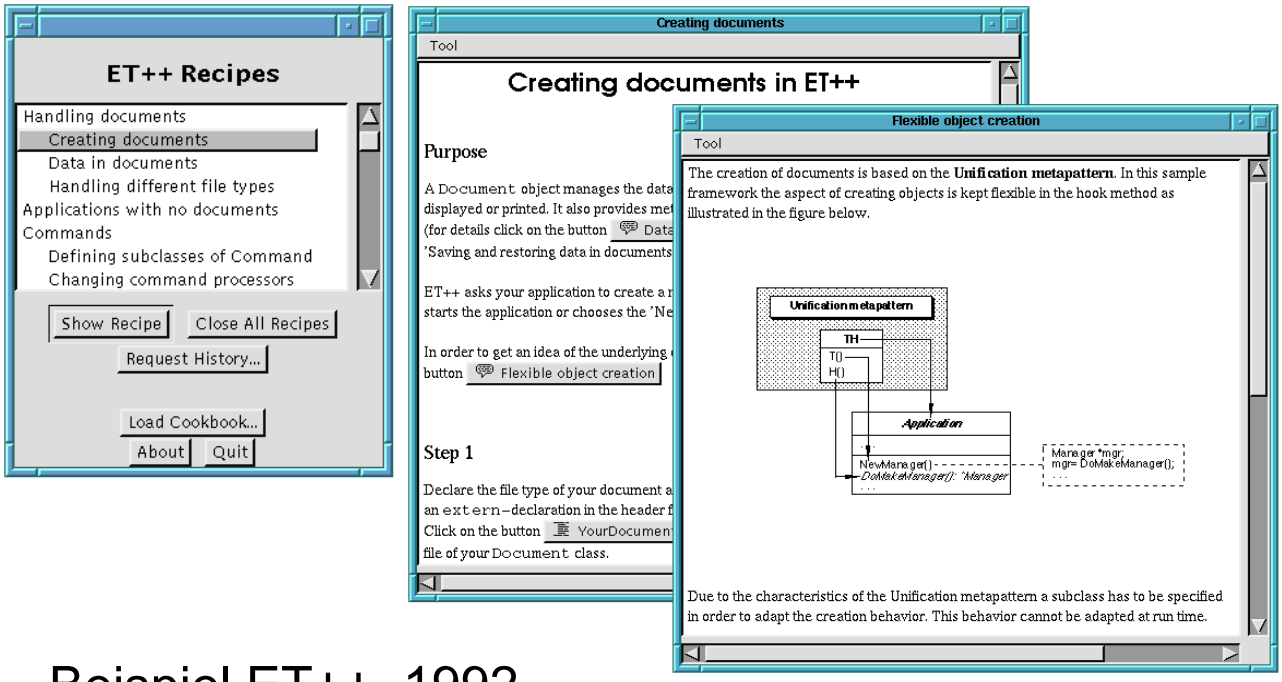
Framework Cookbooks (vgl. Java-Tutorial) bestehen aus einer **Sammlung von Rezepten.** Diese

- 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*

Framework Cookbooks (II)



Beispiel ET++, 1992

Framework Cookbooks (III)

Beispiele für Framework Cookbooks

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

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.

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

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**

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

(siehe auch Buch und CD)

Design-Pattern-Katalog

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

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:

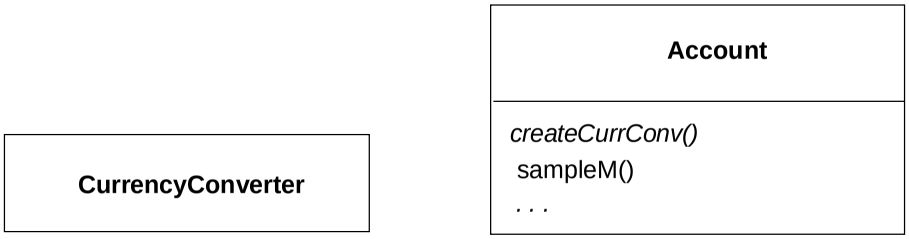
- ein Klassendiagramm mit einem Namen
- eine neue Art, ein Problem zu lösen
- eine Möglichkeit, Code wiederzuverwenden

Factory Method Pattern

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:



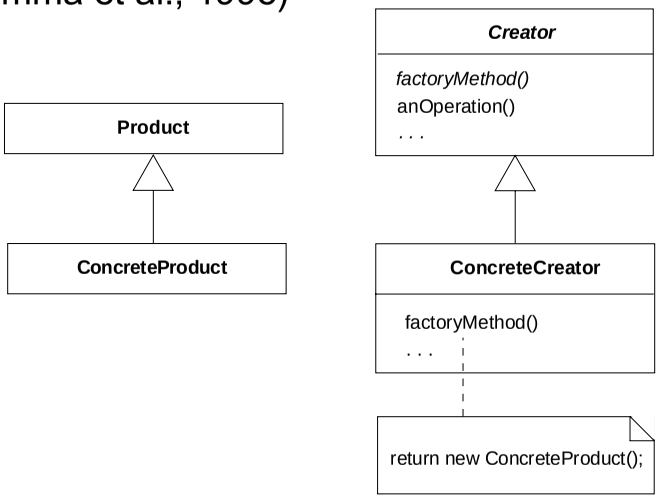
Factory Method (II)

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)

Factory Method (III)

Structure: (aus Gamma et al., 1995)



Consequences:

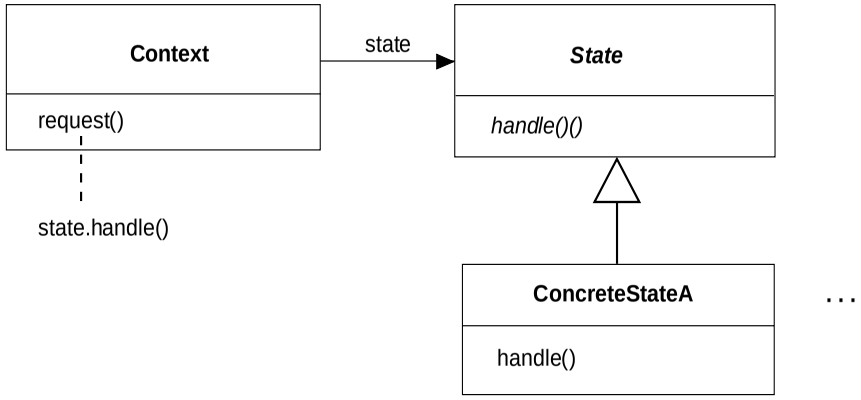
- enables framework to instantiate “abstract” classes
- requires creating subclasse to change product

State Pattern

State (I)

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

Structure:



State (II)

Applicability:

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

Consequences:

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

Implementation:

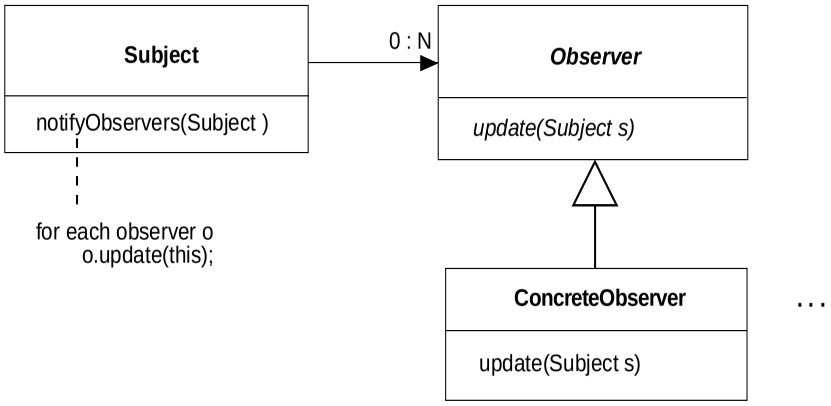
- who defines the state transitions
- creating and destroying state objects

Observer Pattern

Observer (I)

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

Structure:



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

Fallstudien:
Design Patterns auffinden und anwenden

Observer und Factory Method

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

Observer??

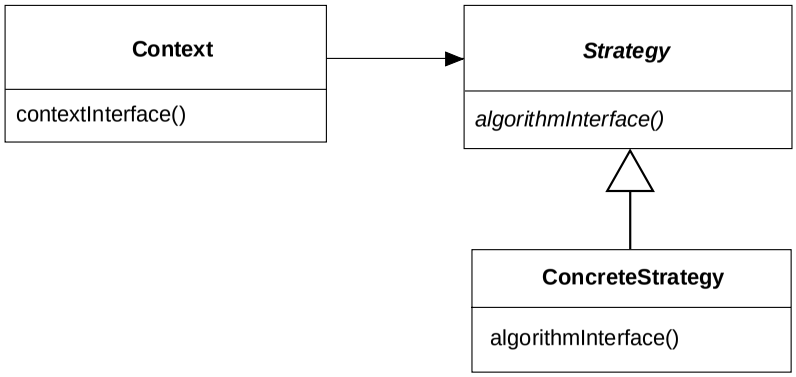
- Wie könnte das Zusammenspiel zwischen Container und Layoutmanagement beschrieben werden?

Strategy Pattern

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:



Strategy (II)

Applicability:

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

Consequences:

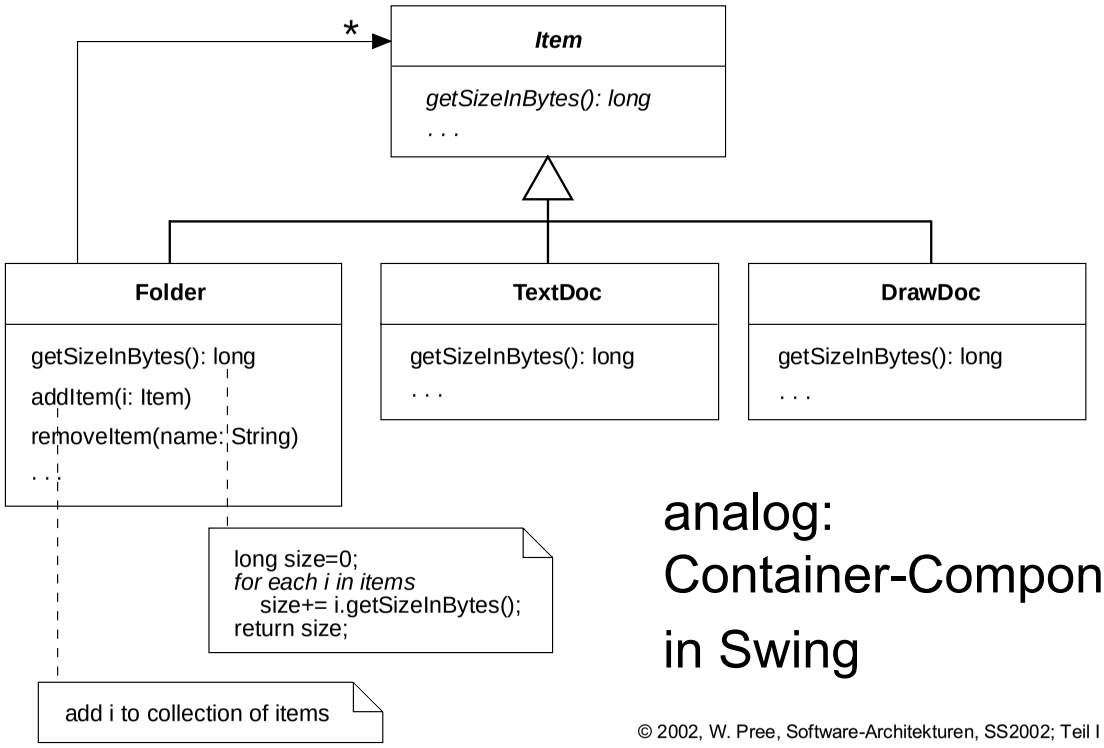
- a choice of implementations with different time and space tradeoffs
- context becomes free of of implementation details
- strategies encapsulate private data of algorithms
- increased number of objects

Composite Pattern

Composite (I)

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

Composite – Beispiel



analog:
Container-Component
in Swing

Composite (II)

Applicability:

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

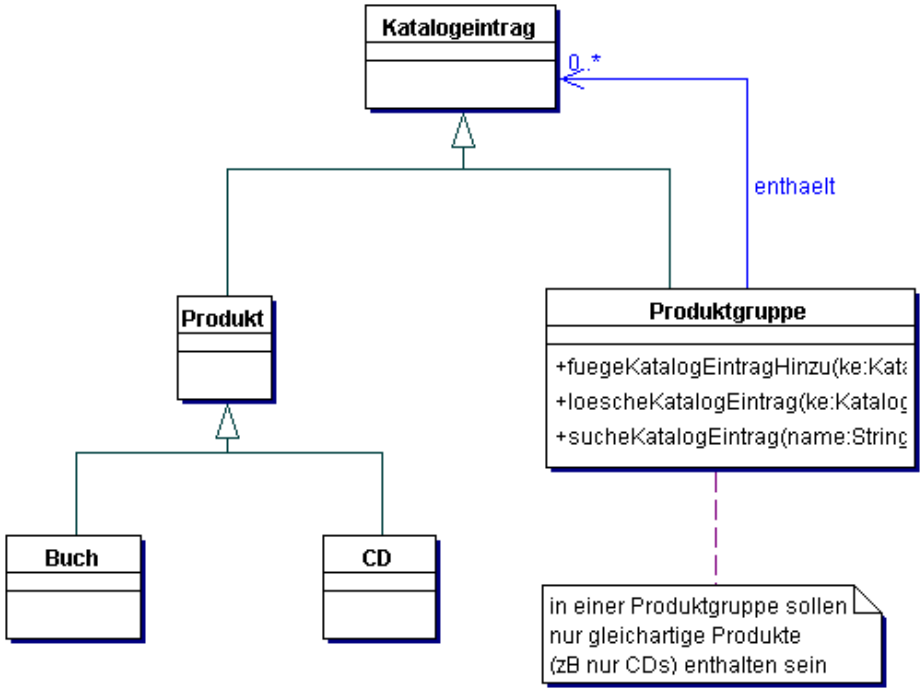
Consequences:

- 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?

Composite – Beispiel aus OO Modellierung-I



weitere GoF-Patterns

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

Essentielle Design Patterns

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

Grundlegende Begriffe (I)

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

Template-Methoden <=> frozen spots

Hook-Methoden <=> hot spots

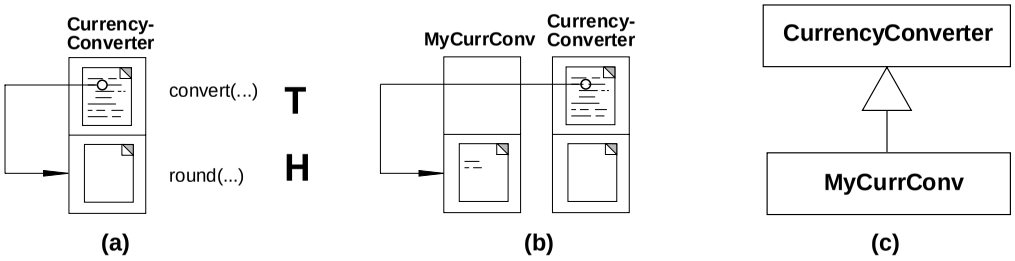
Template-Methoden definieren ein **komplexes Default-Verhalten**, welches **durch Hook-Methoden angepaßt** wird.

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

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:



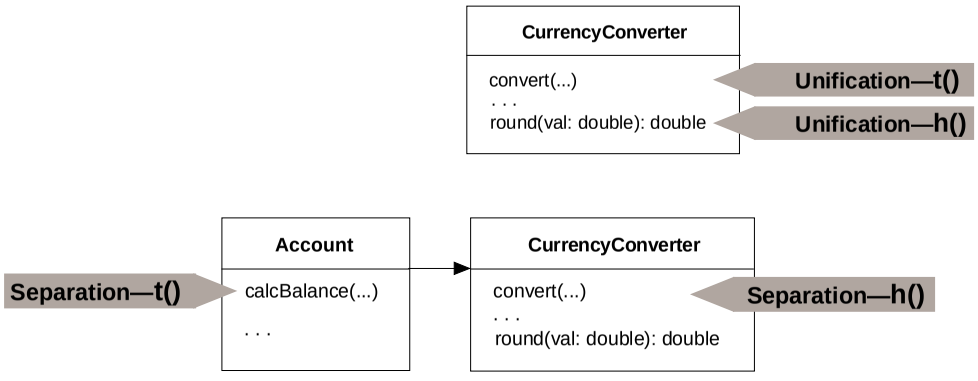
Grundlegende Begriffe (III)

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

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

Anmerkung:

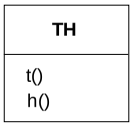
Was ein T und H ist hängt vom Kontext ab:



Essentielle Design Patterns (I)

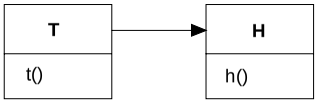
Mögliche Kombinationen von Template- und Hook-Klassen:

- Unification



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

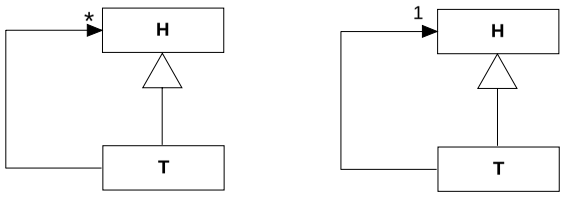
- Separation



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

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.

Separation Pattern (I)

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

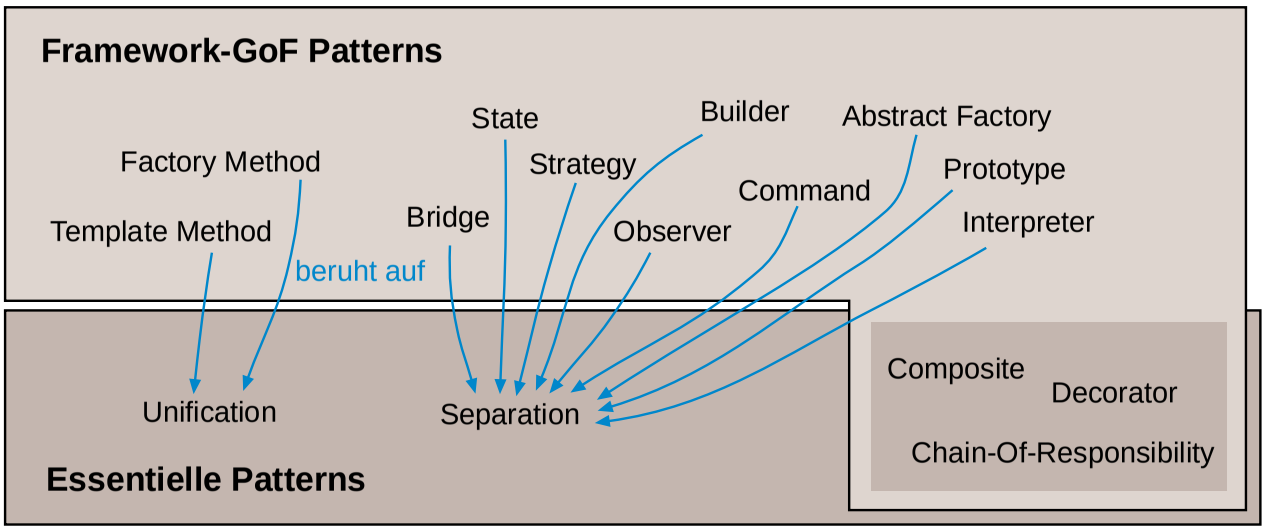
- Typischerweise verfügt T über eine **Instanzvariable** vom statischen Typ H. Ein spezifisches H-Objekt wird quasi in ein T-Objekt durch Aufruf einer Methode setH(...) “eingesteckt”.
- Eine oder mehrere Methoden eines T-Objektes können über einen **Parameter** vom statischen Typ H verfügen. Somit erfolgt 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.

Separation Pattern (II)

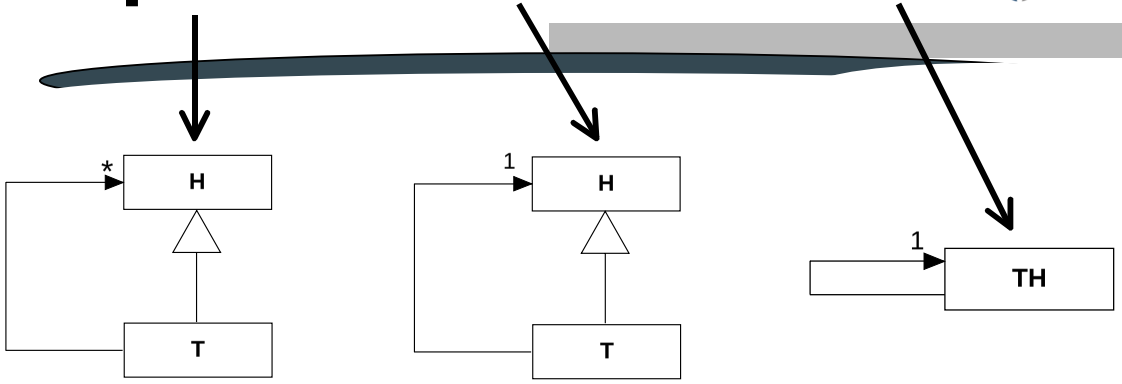
Beispiele:

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

Essentielle Patterns ⇔ GoF-Patterns



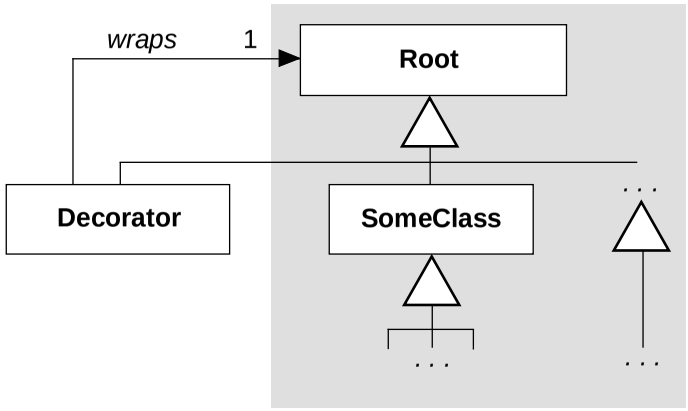
Composite–Decorator–COR



Decorator (siehe Buch/CD):

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

Decorator



Decorator–Beispiel (ET++)