

Software Engineering

Teil 4: Architekturentwurf und
Objektorientiertes Design

April 2018

Dr. Christian Bartelt



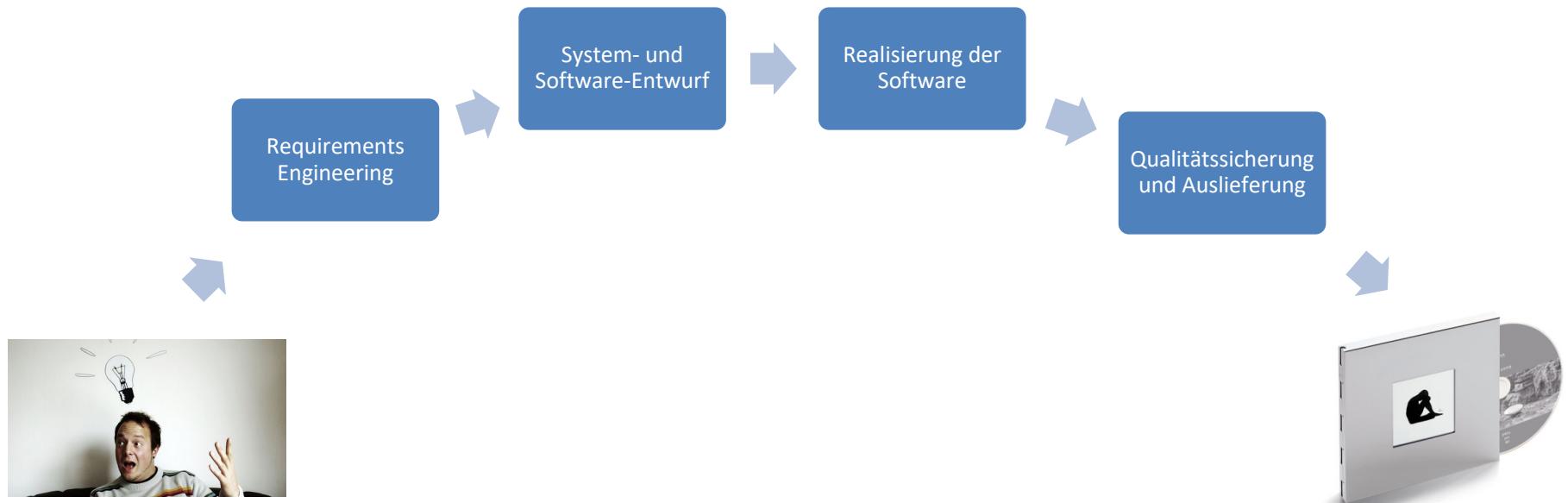
UNIVERSITATEA
BABEŞ-BOLYAI

Universität Mannheim
Institut für Enterprise Systems InES
Schloss
68131 Mannheim / Germany

UNIVERSITY OF
MANNHEIM

Zur Orientierung

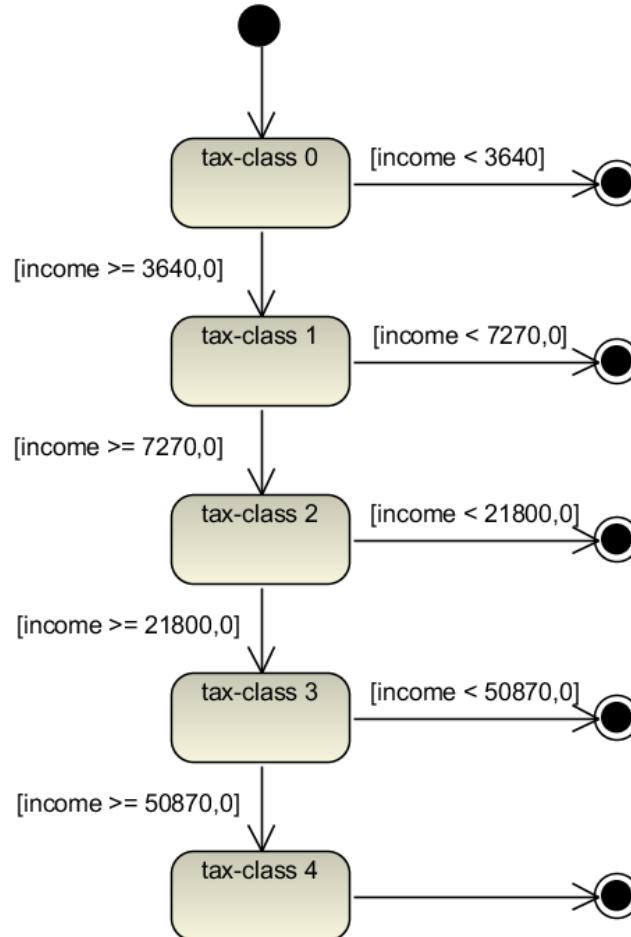
Aufgabenbereiche in der Softwareentwicklung



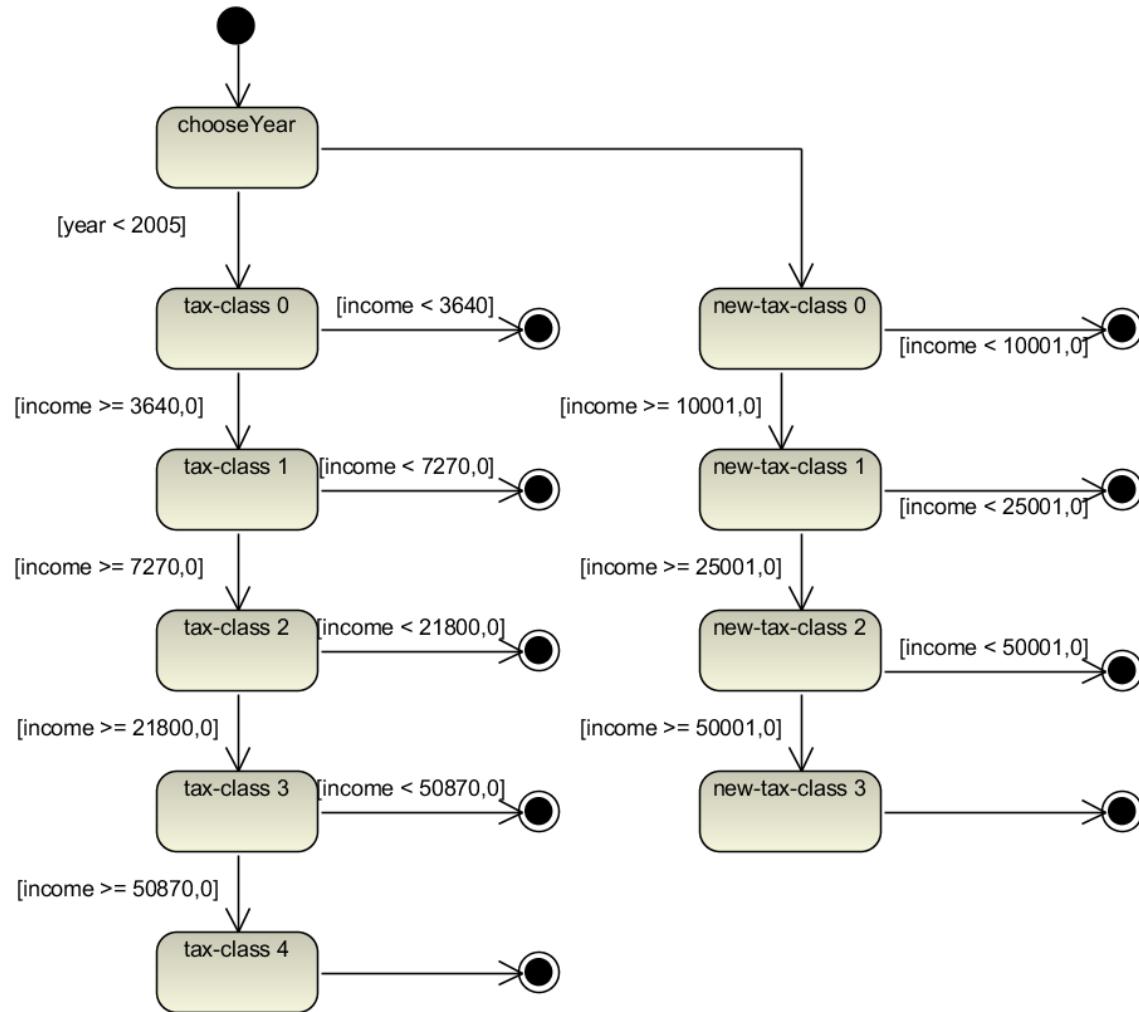
Inhalt

- Motivation und Herausforderungen
- Überblick und Konzepte
- Design Patterns
- Entwurf und Beschreibung

Der Einkommenssteuerrechner: ein einfaches Beispiel



Der Einkommenssteuerrechner: Ab 2005 anders gerechnet wg. Gesetzesänderung

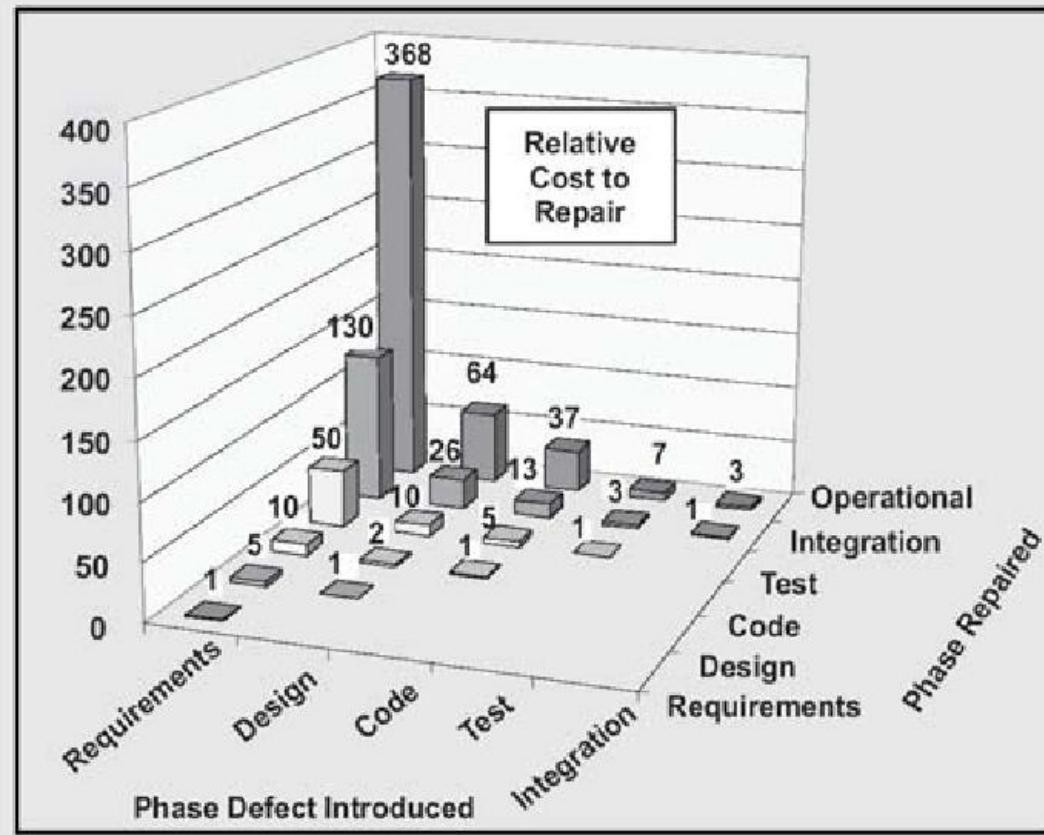


Der Einkommenssteuerrechner: jetzt auch im österreichischen Markt



Bedeutung der frühen Phasen (Wiederholung)

Figure 4: *Relative Cost of Software Fault Propagation*

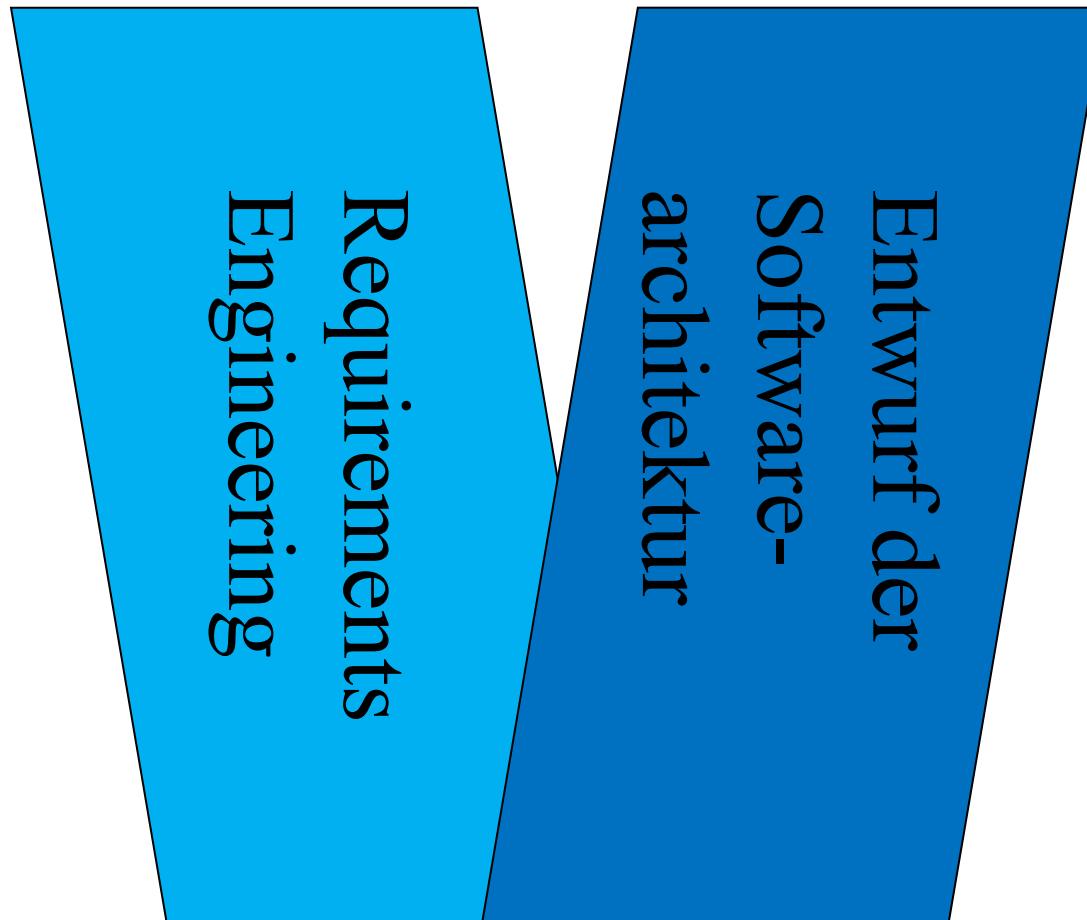


- Je später Fehler gefunden werden, um so teuer ist die Behebung!
- WHISCY Problem: Why in the hell isn't Sam coding yet?

Warum sind die frühen Phasen so entscheidend?

- Schlüsselfaktoren für erfolgreiche Softwareentwicklung
 - Requirement Engineering
 - Architekturentwurf
- Bergen hohes Risiko für Fehlentwicklung
 - Frühe Entscheidungen
 - Auswirkungen der Entscheidungen spät sichtbar
- Ansatz
 - Durchgängige Integration der Bereiche Requirements Engineering und Architekturentwurf
 - Frühzeitige und durchgängige Verfolgung, Qualitätssicherung und Kopplung von den Anforderungen und der Architektur nötig

Requirementsengineering und Architekturentwurf gehen Hand in Hand...



Inhalt

- Motivation und Herausforderungen
- Überblick und Konzepte
- Design Patterns
- Entwurf und Beschreibung

Wdh.: Objektorientierung

Was sind Objekte..?

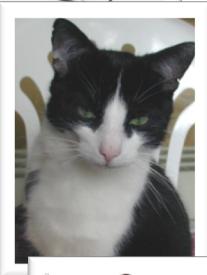
- ...sind Gegenstände von Interesse
 - einer Beobachtung
 - einer Untersuchung
 - einer Messung
- ...sind Exemplare, etwas individuelles
 - Exemplare von Dingen (Auto, Roboter)
 - Personen (Kind, Kunde, Mitarbeiter)
 - Begriffe der realen Welt (Bestellung, Reservierung) oder der Vorstellungswelt (juristische Person)
- ...was macht Objekte aus
 - Objekte können physisch oder konzeptionell sein, sie haben eine Identität
 - Objekte verfügen über Eigenschaften (Attribute) wie Größe, Name, Form usw.
 - Objekte verfügen über Beziehungen zu anderen Objekten (Links) wie Student besucht vorlesung usw.
 - Objekte weisen Vorgänge (Methoden) auf (Dinge, die sie tun können).
Beispiele: Wert festlegen, Bildschirm anzeigen, Geschwindigkeit erhöhen

Wdh.: Objektorientierung

Was sind Klassen..?



Name = Tom
Fellfarbe = grau, weiß
Gewicht = 4,5 kg



Name = Minka
Fellfarbe = schwarz, weiß
Gewicht = 4,0 kg



Name = Garfield
Fellfarbe = schwarz, orange
Gewicht = 13 kg

Alles Katzen!

- Klassen beschreiben Mengen von ähnlichen Objekten.
Ähnlich heißt:
 - Die Objekte besitzen
 - die gleichen Operationen
 - die gleichen Attribute, aber unterschiedliche Attributwerte,
 - die gleichen Beziehungsarten, aber unterschiedliche konkrete Bezüge
- Objekte, die zu einer Klasse gehören, werden auch **Instanzen** dieser Klasse genannt.
- Klassen können als **Abstraktion** von Objekten verstanden werden, d.h. es werden Details weg gelassen, Gemeinsamkeiten werden verallgemeinert.

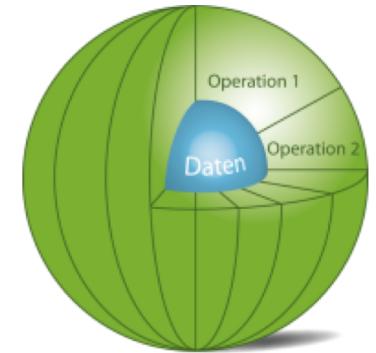
Wdh.: Objektorientierung

Wichtige Konzepte..?

Klassen und Objekte



Geheimnisprinzip



Schnittstelle und Schnittstellenimplementierung



?????????



Wichtige Konzepte der Objektorientierung (1)

Beispiel: List-Schnittstelle in Java

```
public interface List {  
    public boolean add(Object o);  
    public boolean remove(Object o);  
    public boolean contains(Object o);  
    public int indexOf(Object o);  
    public Object get(int index);  
    public Object set(int index, Object element);  
    public int size();  
    public Iterator iterator();  
  
    // Methoden, die hier nicht weiter interessieren  
}
```

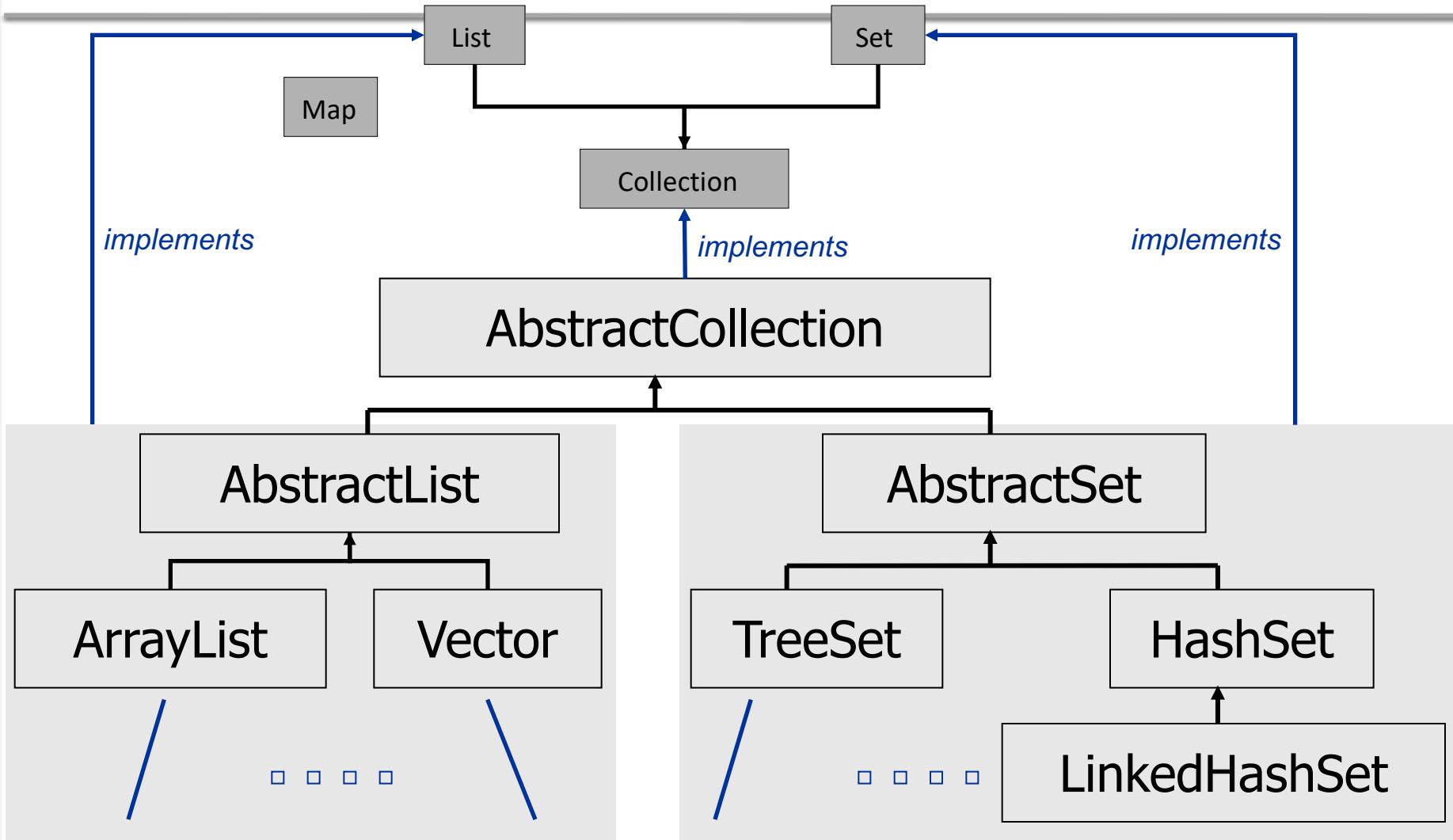
Wichtige Konzepte der Objektorientierung (2)

Beispiel: Iterator-Schnittstelle in Java

```
public interface Iterator {  
    public boolean hasNext();  
    public Object next();  
    public void remove();  
}
```

Wichtige Konzepte der Objektorientierung (3)

Beispiel: Behälter-Implementierung in Java



Wichtige Konzepte der Objektorientierung (4)

Beispiel: Eine Schnittstellen-Implementierung in Java

```
public abstract class AbstractList implements List {  
  
    public abstract int size();  
    public abstract Object get(int index);  
  
    ...  
}  
  
public class Range extends AbstractList {  
    private int start, stop;  
    public Range(int start, int stop) {  
        if (stop < start)  
            throw new IllegalArgumentException();  
        this.start = start;  
        this.stop = stop;  
    }  
    public int size() { return stop - start; }  
    public Object get(int index) {  
        if (0 <= index && index < stop - start)  
            return new Integer(start + index);  
        else  
            throw new IndexOutOfBoundsException();  
    }  
}
```

} zu realisierende Methoden

} Implementation eines Wertebereichs der List Implementation

Wichtige Konzepte der Objektorientierung (5)

Beispiel: Schnittstellen-Nutzung mit Java Klassen

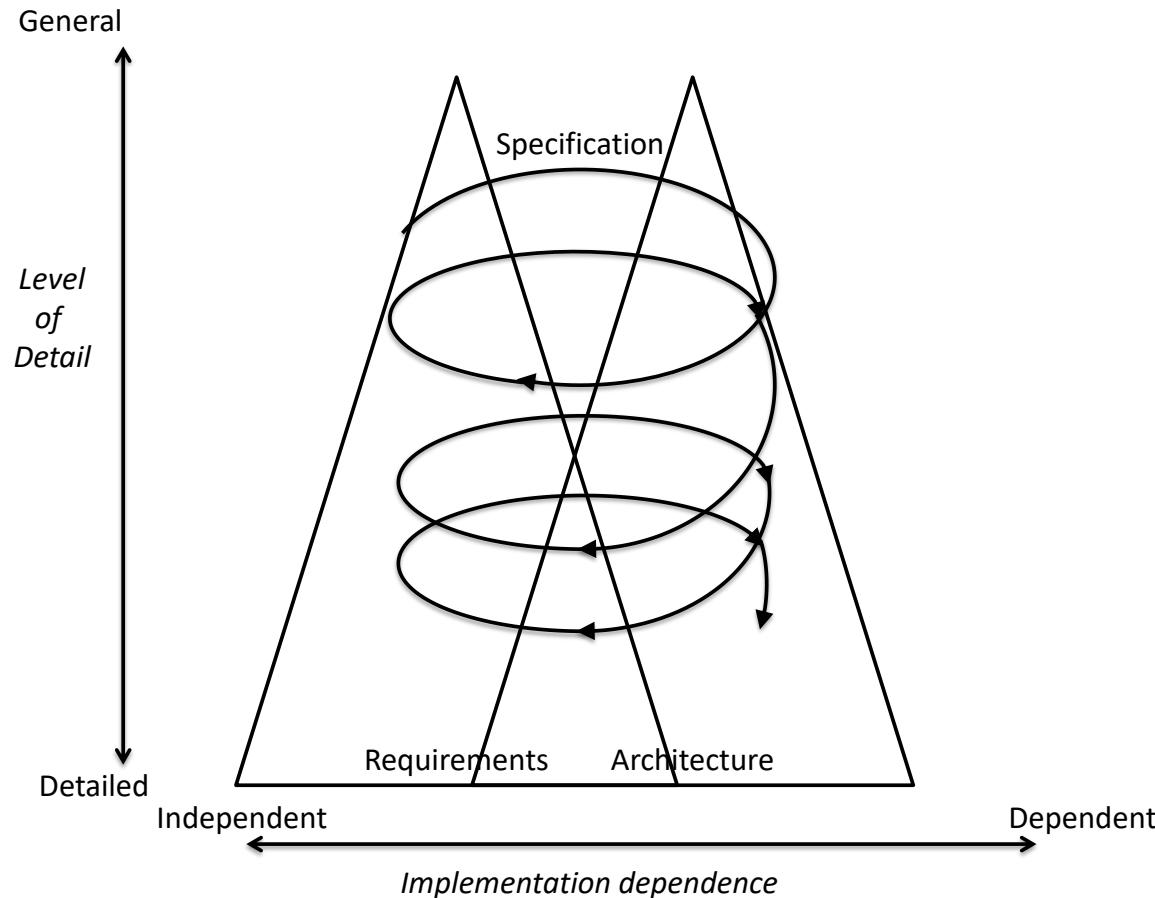
```
public class ListUser {  
  
    private List list;  
    public ListUser (List list) {  
        this.list = list;  
    }  
  
    public void foo() {  
        Iterator i = list.iterator();  
  
        // tu was mit list  
    }  
}
```

Nutzer der Schnittstelle

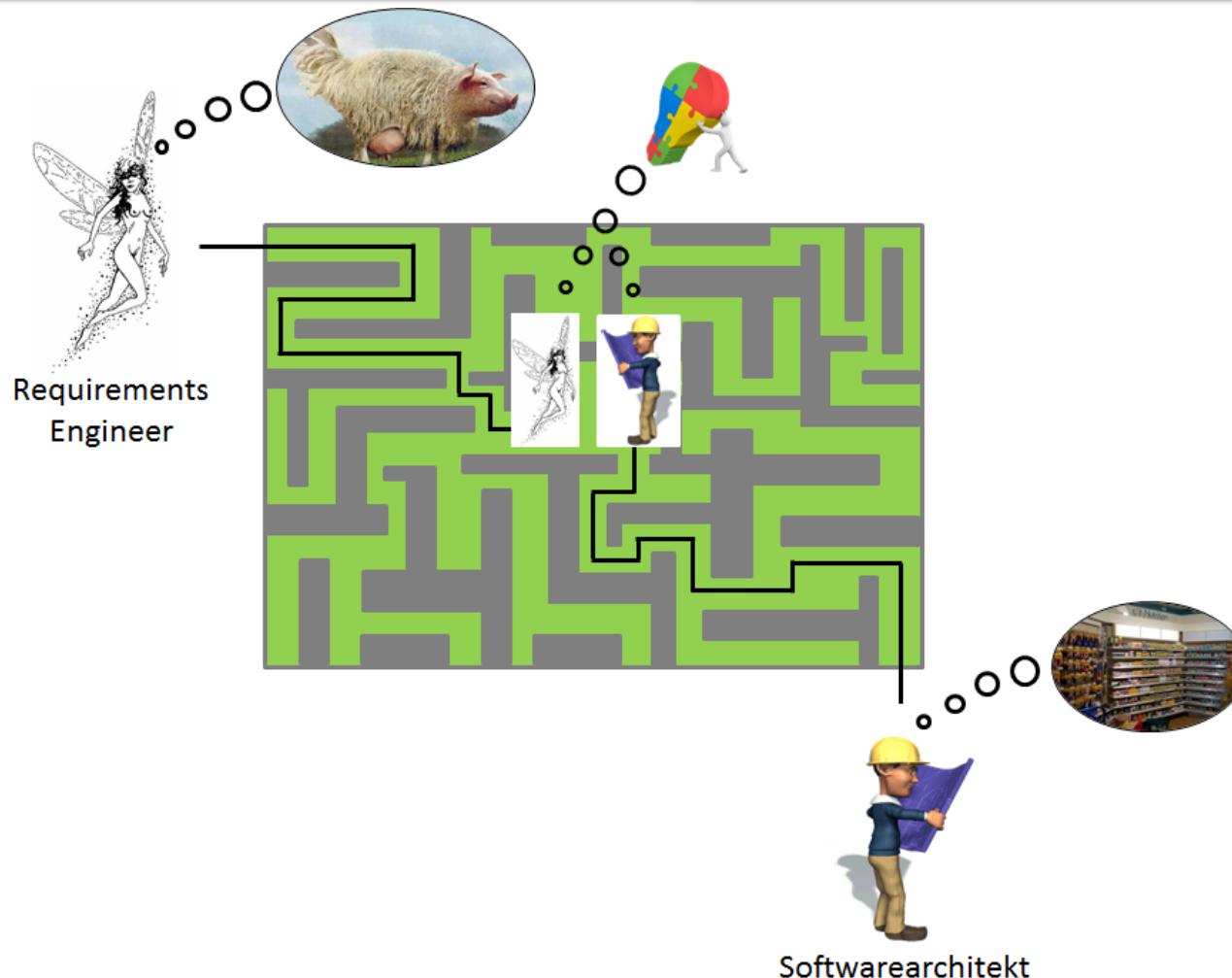
```
public static void main(String[] args) {  
    List list = new ArrayList();  
        // oder irgendeine andere Klasse,  
        // die List implementiert  
    // Liste fuellen:  
    ListUser lu = new ListUser(list);  
    lu.foo();  
}
```

- Implementierung instanziert
- Nutzer instanziert
- Verbindung herstellen

Zusammenspiel zwischen Requirements Engineering und Architekturentwurf...



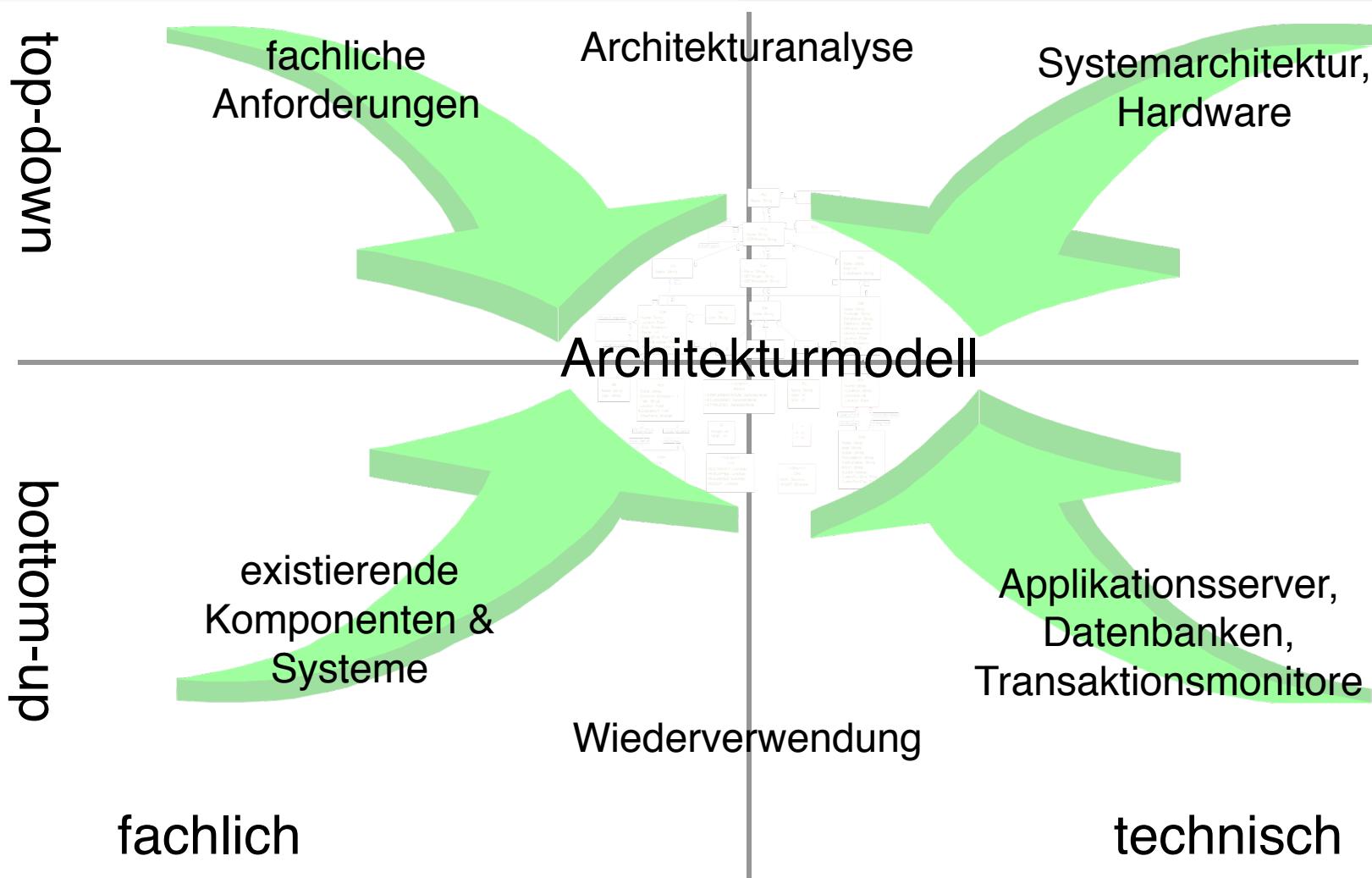
Herausforderung des Architekturentwurfs



Zielsetzung des Architekturentwurfs

- Ziel des Architekturentwurfs ist:
 - Den Bauplan des Systems zu erstellen, nach dem sich dann Erstellung, Wartung, Pflege und Weiterentwicklung ausrichten
 - Eine vollständige und prägnante Architekturbeschreibung erstellen
 - Dabei die geforderten funktionalen und nicht funktionalen Anforderungen gewährleisten
- Die Architekturbeschreibung dient als
 - Kommunikations- und Diskussionsplattform
 - Design- und Implementierungsplan

Top-down und bottom-up kombiniert mit orthogonalen Sichten: Fachlich und technisch



Anwendung der Objektorientierung garantiert kein „gutes“ Programmdesign!

- Zentrale Fragen:
 - Was macht ein gutes Entwurf aus?
 - Wie kommt man zu einem guten Entwurf?
- Zentrale Konzepte: „gutes“ = „modulares“ Design
 - hohe Kohäsion
 - Geringe Kopplung
 - Komposition und Abstraktion / Information Hiding

Kohäsion

- Beschreibt die inhaltliche Zusammengehörigkeit von Komponenten hinsichtlich ihres Aufgabereichs
- Ziel ist eine hohe Kohäsion innerhalb von Komponenten

NützlicheDinge
wurzelAus(n : real) : real
log(x : real,y : real) : real
substring(s : String, pos : int, length l) : String

Math
wurzelAus(n : real) : real
log(x : real,y : real) : real
StringBib
substring(s : String, pos : int, length l) : String

Schlechte Kohäsion!

Kopplung

- Was ist Kopplung?
 - Entsteht durch die Abhangigkeit zweier oder mehr Komponenten voneinander
 - Definiert den Grad der Abhangigkeit
 - Gibt Aufschluss daruber, wie stark nderungen an Komponenten andere beeinflussen
 - Arten von Kopplung (zwischen Komp. A und B)
 - Aufruf: A ruft B auf (bzw. einen ihrer Dienste)
 - Erzeugung (A erzeugt B)
 - Daten (gemeinsame Datenstrukturen)
 - Laufzeitumgebung bzw. Hardware (missen in gleicher VM laufen)
 - Zeit (zeitlich Bedingungen bzgl. ihrer Ausfhrung, Bsp. B muss existieren solange A existiert)
 - Ziel ist eine **geringe** Kopplung zwischen Komponenten
-

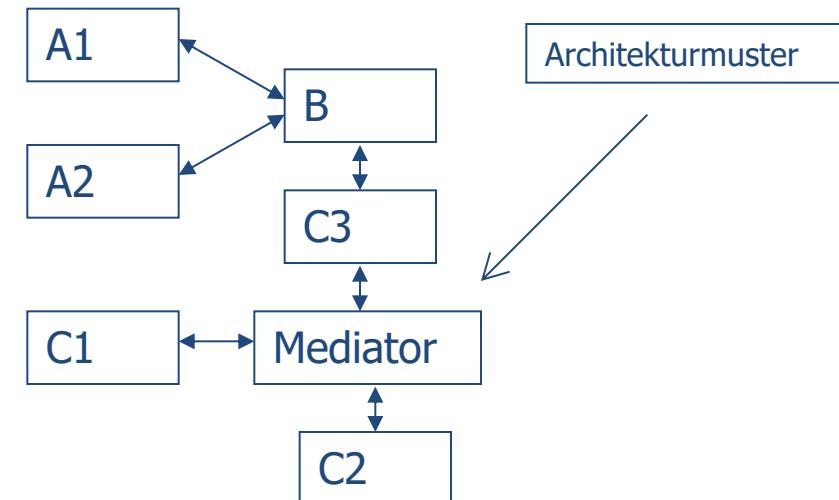
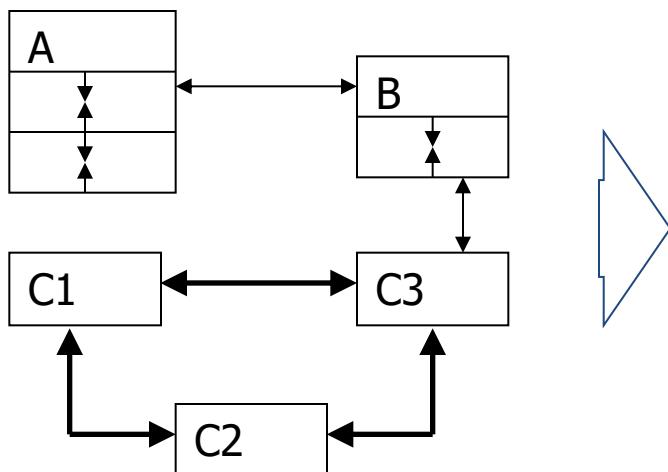
Auf der Suche nach einer „guten=modularen“ Architektur: Modularität durch geringe Kopplung und hohe Kohäsion

Kopplung \triangleq Abhängigkeit zwischen System-Elementen, z.B. durch
Interaktion, Vererbung, etc. Ziel: minimieren

\longleftrightarrow stark \longleftrightarrow schwach

Kohäsion \triangleq Zusammenhalt eines System-Elements, z.B. Kapselung von
Daten und Funktionen Ziel: maximieren

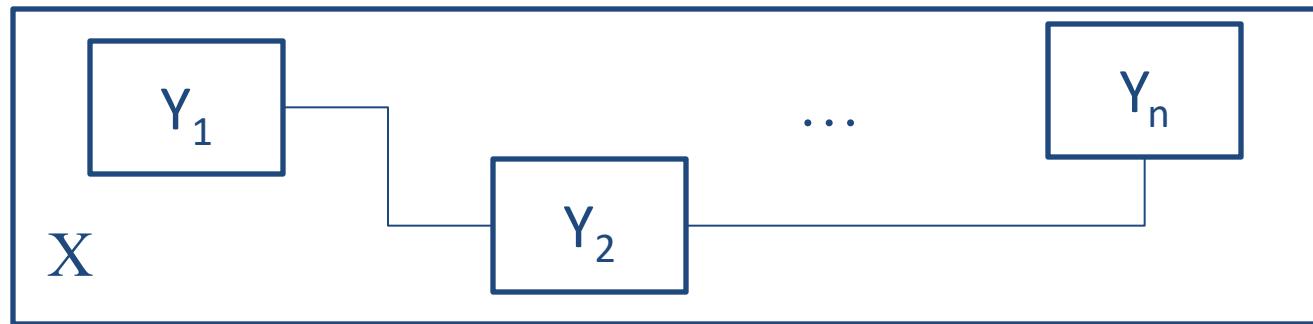
$\rightarrow\leftarrow$ stark



Kopplung und Kohäsion sind durch Metriken messbar!

Gute Architektur durch: (De-)Komposition und Information Hiding (Abstraktion)

- **Komposition:** Zusammensetzen eines Elementes X von der Art K aus Elementen Y_1, \dots, Y_n , die ebenfalls von der Art K sind.



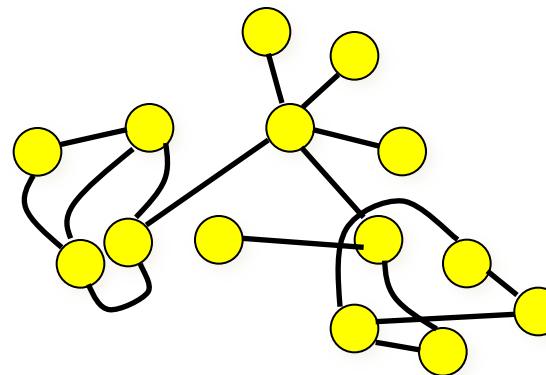
- **Abstraktion:** Verbergen der Details von Y_1, \dots, Y_n durch eine abstraktere Sicht, die durch X zur Verfügung gestellt wird. Häufig wird dies durch Schnittstellen erreicht.

Wie kommt man zu einer „guten“ = „modularen“ Architektur?

- **Kompositionales Zusammenspiel von hoher Kohäsion, passende Abstraktion und Information Hiding sowie eine geringe Kopplung**

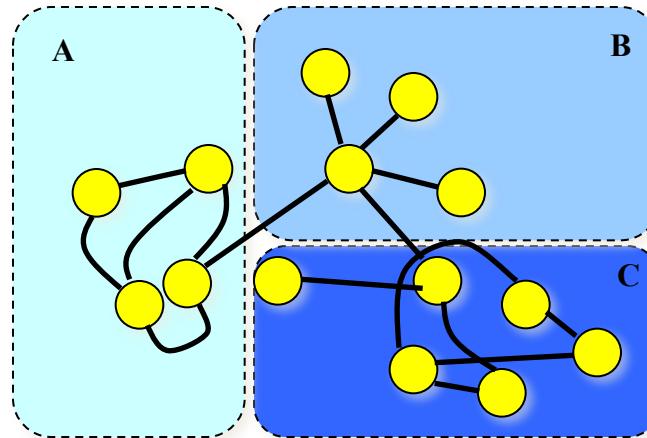
Ein Beispiel: wie kommt man zur Softwarearchitektur?

- Ein gängiger Weg, um die Komponenten einer Architektur zu ermitteln besteht in der Betrachtung der Abhängigkeiten



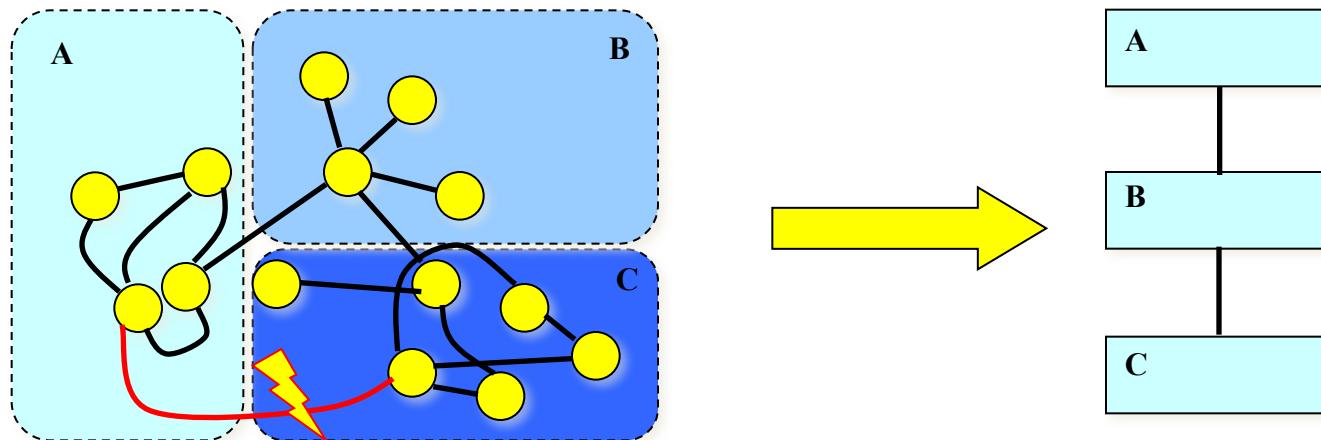
Ein Beispiel: wie kommt man zur Softwarearchitektur?

- Ein gängiger Weg, um die Komponenten einer Architektur zu ermitteln besteht in der Betrachtung von Abhängigkeiten
- Komponenten „gruppieren“ Elemente mit starker Abhängigkeit



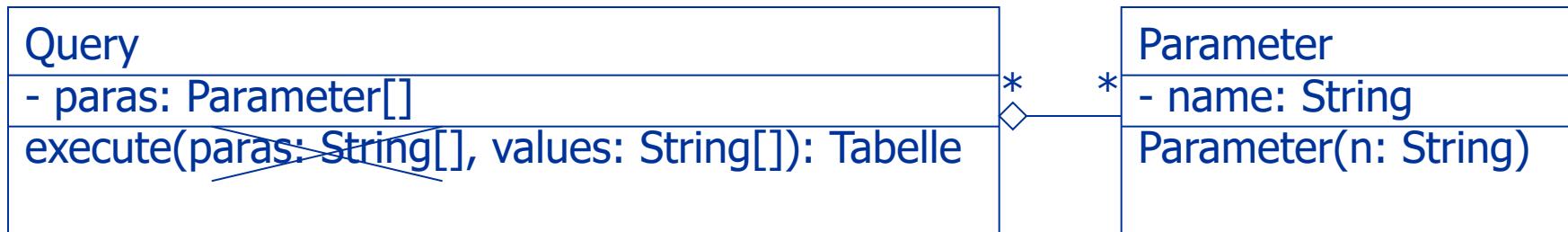
Ein Beispiel: wie kommt man zur Softwarearchitektur?

- Ein gängiger Weg, um die Komponenten einer Architektur zu ermitteln besteht in der Betrachtung von Abhängigkeiten
- Komponenten „gruppieren“ Elemente mit starker Abhängigkeit
- Abhängigkeiten zwischen Komponenten werden zu Schnittstellen
- Ist die Architektur einmal festgelegt, gibt sie Richtlinien vor für die Einführung neuer Abhängigkeiten (hier: keine Verbindung zwischen den Komponenten A und C)



Schlechte Beispiele: Keine Schmalen und vollständigen Schnittstellen

Kontext: Query-Schnittstelle der Datenhaltungsschicht



Kontext: Applikationsschicht

keine Typüberprüfung



Problem: Keine echte Schnittstelle
Nutzer und Erbringer kennen weitere Details

Inhalt

- Motivation und Herausforderungen
- Überblick und Konzepte
- Design Patterns
- Entwurf und Beschreibung

Muster – Wieso, weshalb, warum?

- Klassen sind sehr kleine Einheiten, deren Wiederverwendung steigert die Produktivität kaum.
 - Datenbanken, Middleware und COTS-Komponenten ermöglichen Wiederverwendung auf der Ebene von ausführbaren, binären Komponenten.
 - Frameworks und Bibliotheken unterstützen lediglich die Wiederverwendung von Programmcode.
 - Aspect-oriented programming und Codegeneratoren unterstützen die Wiederverwendung von spezifischen Lösungsansätzen auf der Implementierungsebene.
 - Muster als Instrumentarium zur Wiederverwendung von Spezifikationen, Design und Entwurfswissen
-



Ursprung des Musteransatzes

- Der Begriff „Pattern“ als Lösungsmuster, das Lösungsansätze für ähnliche bzw. sich wiederholende Designprobleme bietet, geht zurück auf den Architekten Christopher Alexander
- Beispiel Window Place [Ale77]:
 - Jeder liebt Aussichtsplätze in Hochhäusern, auf Bergen oder am Meer
 - Wenn man in einen Raum kommt der Fenster und Sitzplätze hat, die Sitzplätze aber nicht bei den Fenster sind, dann steht man vor folgendem Problem:
 - Man möchte sich hinsetzen, um es bequem zu haben
 - Man möchte möglichst in der Nähe des Lichts sein
 - Jeder Raum, in dem man sich länger aufhält sollte mindestens einen „Window Place“ haben – Sitzplätze vor dem Fenster

„Every pattern we define must be formulated in the form of a rule which establishes a relationship between a context, a system of forces which arises in that context and a configuration, which allows these forces to resolve themselves in that context.“

Ziele von Entwurfsmustern

- Ziel von Entwurfsmustern ist:
 - Dokumentation und Wiederverwendung von Lösungen, die zuvor erfolgreich eingesetzt wurden.
- In vielen objektorientierten Systemen lassen sich wiederkehrende Muster von Klassen und kommunizierenden Objekten finden.
- Diese Muster lösen spezifische Entwurfsprobleme und machen objektorientierte Entwürfe
 - Flexibler
 - Eleganter
 - Wiederverwendbar

Definition Muster:

Ein Muster ist eine schematische Lösung für eine Klasse von Problemen das sich in einem gewissen Kontext bewährt hat.

Muster sind Lösungsbeschreibungen

- Ein Designproblem beruht darauf, dass in gewissen Situationen (**Kontext**), sich widerstrebende Kräfte (**Problem**) entgegenstehen. Die **Lösung** besteht darin, mit den in der jeweiligen Designdisziplin zur Verfügung stehenden Werkzeugen die Situation so zu gestalten, dass diese Kräfte optimal ausgeglichen werden.
 - In der klassischen Architektur ist dies die räumliche Gestaltung von Gebäuden
 - In der Softwarearchitektur ist dies die Struktur der Komponenten und deren Beziehungen
- Ein Muster beschreibt eine Lösung für ein Designproblem
 - Muster sind konstruktiv, d.h. sie liefern Instruktionen zur Lösung einer Problemsituation
 - Muster sind abstrakt , d.h. sie bieten ein breites Anwendungsspektrum mit vielen verschiedenen Anwendungsfällen
 - Muster sind stabil, d.h. sie erfassen den stabilen Kern der Lösung
- Muster sind über die Zeit hinweg gültig und weitestgehend invariant gegenüber neuen Möglichkeiten

Mustersammlungen

- Musterkataloge sind eine einfache meist kategorisierte Sammlungen ähnlicher oder komplementärer Muster.
- Eine Mustersprache bildet eine Einheit, in der die zusammenhängenden Muster miteinander kooperieren, um ein gemeinsames Problem zu lösen.
- Mustersysteme umfassen eine Menge von Mustern auf verschiedenen Abstraktionsstufen. Die Beziehungen zwischen den Mustern beschreiben, wie sich durch die Kombination der Muster Lösungen komplexerer Probleme konstruieren lassen.
- In der Regel entwickelt sich aus einem Musterkatalog eine Mustersprache und eventuell dann ein Mustersystem.

Entwurfsmuster (Design Patterns)

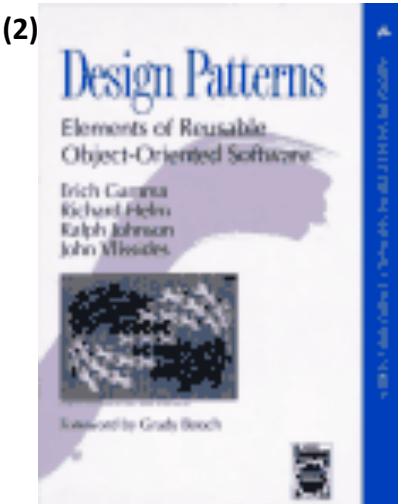
- Der Entwurf wiederverwendbarer objektorientierter Softwarebausteine ist schwer:
 - Relevante Objekte aufspüren
 - Sie zu Klassen passender Granularität abstrahieren
 - Ihre Schnittstellen sowie Vererbungshierarchien definieren
 - und die zentralen Beziehungen zwischen ihnen festlegen.
- Dabei Spagat zwischen
 - Erfüllung der vorliegenden spezifischen Anforderungen für ein konkretes Projekt.
 - Möglichst allgemeingültige Lösungen, um sie bei zukünftigen Problemen und Anforderungen wiederverwenden zu können.

Überblick - Entwurfsmuster

- Struktur
 - **Adapter^(1,2)**
 - Brücke (Bridge)^(1,2)
 - **Dekorierer (Decorator)^(1,2)**
 - **Fassade (Facade)^(1,2)**
 - Fliegengewicht (Flyweight)⁽²⁾
 - **Kompositum (Composite)^(1,2)**
 - **Proxy^(1,2)**
- Verhalten
 - **Befehl (Command)^(1,2)**
 - **Beobachter (Observer)^(1,2)**
 - **Besucher (Visitor)^(1,2)**
 - Interpreter (Interpreter)⁽²⁾
 - Iterator^(1,2)
 - Memento⁽²⁾
 - **Rolle⁽¹⁾**
 - **Schablonenmethode (Template Method)^(1,2)**
 - **Strategie (Strategy)^(1,2)**
 - Vermittler (Mediator)^(1,2)
 - **Zustand (State)^(1,2)**
 - Zuständigkeitskette (Chain of Responsibility)⁽²⁾
- Erzeugung
 - **Abstrakte Fabrik (Abstract Factory)^(1,2)**
 - **Erbauer (Builder)⁽²⁾**
 - **Fabrikmethode (Factory Method)^(1,2)**
 - Objektpool⁽¹⁾
 - Prototyp (Prototype)^(1,2)
 - **Singleton^(1,2)**

Legende:

Fett = wird in der Vorlesung behandelt



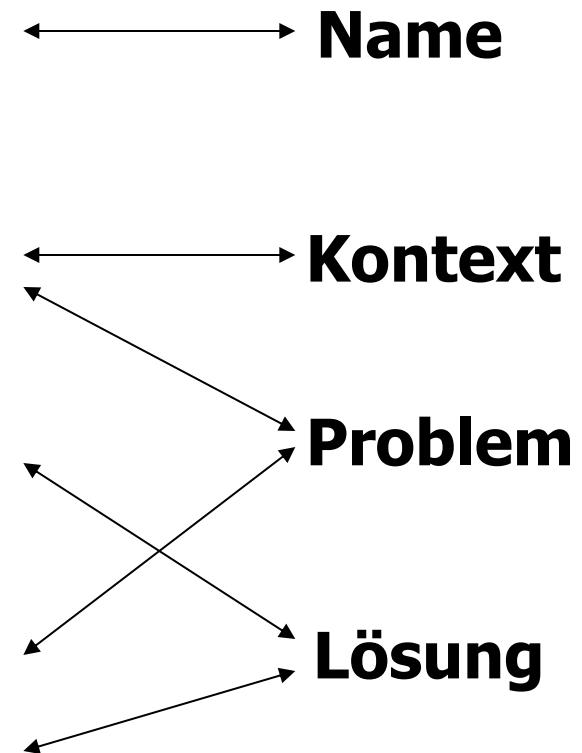
Beschreibungsschema

- Bezeichnung und Überblick
 - Name
 - Kurzbeschreibung, Zweck
 - Alternative Bezeichnungen

- Zweck
 - Motivation und Zielsetzung
 - Anwendbarkeit

- Lösung
 - Graphische Beschreibung der Struktur
 - Beteiligte Klassen
 - Interaktion

- Konsequenzen
 - Vor- und Nachteile
 - Implementierung und Beispiel-Code
 - Anwendungsbeispiele
 - Bezug zu anderen Mustern

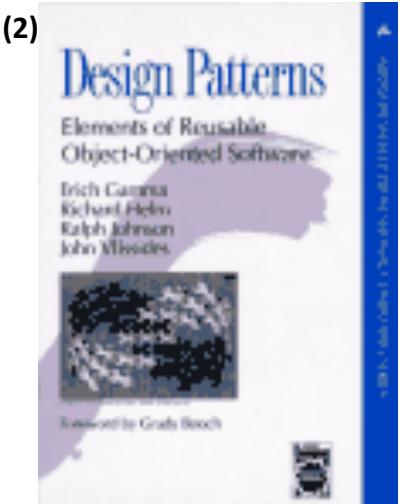


Überblick - Entwurfsmuster

- Struktur
 - **Adapter^(1,2)**
 - Brücke (Bridge)^(1,2)
 - Dekorierer (Decorator)^(1,2)
 - Fassade (Facade)^(1,2)
 - Fliegengewicht (Flyweight)⁽²⁾
 - Kompositum (Composite)^(1,2)
 - Proxy^(1,2)
- Verhalten
 - Befehl (Command)^(1,2)
 - Beobachter (Observer)^(1,2)
 - Besucher (Visitor)^(1,2)
 - Interpreter (Interpreter)⁽²⁾
 - Iterator^(1,2)
 - Memento⁽²⁾
 - Rolle⁽¹⁾
 - Schablonenmethode (Template Method)^(1,2)
 - Strategie (Strategy)^(1,2)
 - Vermittler (Mediator)^(1,2)
 - Zustand (State)^(1,2)
 - Zuständigkeitskette (Chain of Responsibility)⁽²⁾
- Erzeugung
 - Abstrakte Fabrik (Abstract Factory)^(1,2)
 - Erbauer (Builder)⁽²⁾
 - Fabrikmethode (Factory Method)^(1,2)
 - Objektpool⁽¹⁾
 - Prototyp (Prototype)^(1,2)
 - Singleton^(1,2)

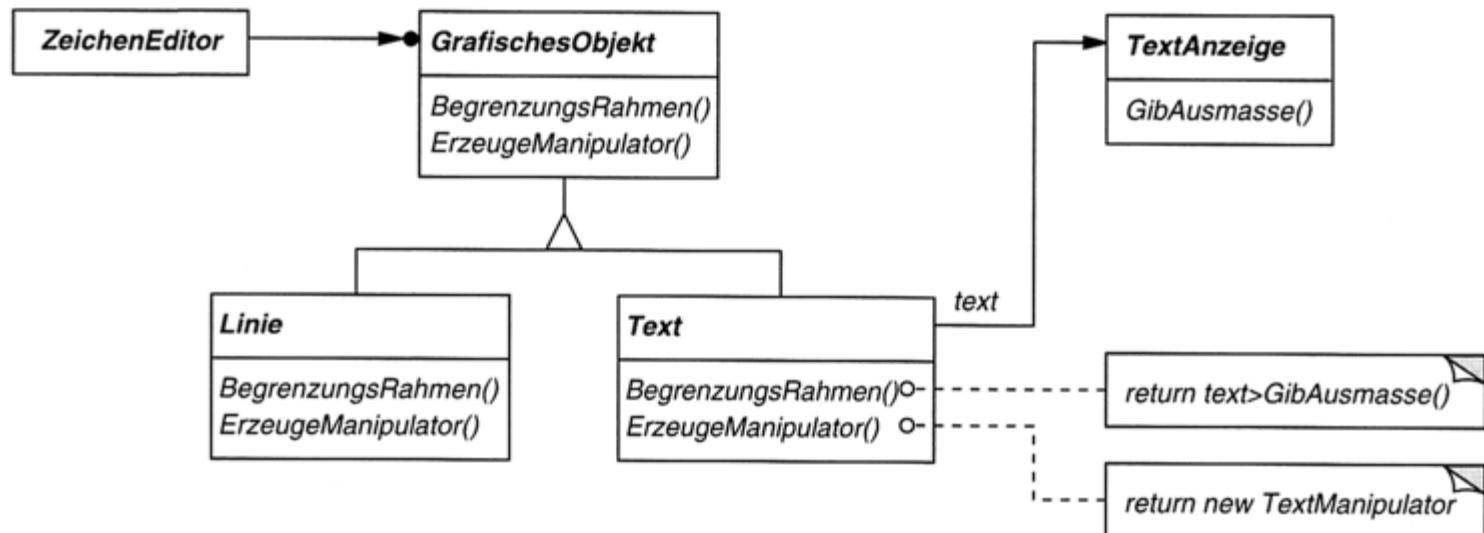
Legende:

Fett = wird in der Vorlesung behandelt



Adapter Muster (1)

- **Name:** Adapter (dt. Adapter)
- **Zweck:** Dient zur Übersetzung einer Schnittstelle in eine andere. Dadurch wird die Kommunikation von Klassen mit zueinander inkompatiblen Schnittstellen ermöglicht.
- **Motivation:**



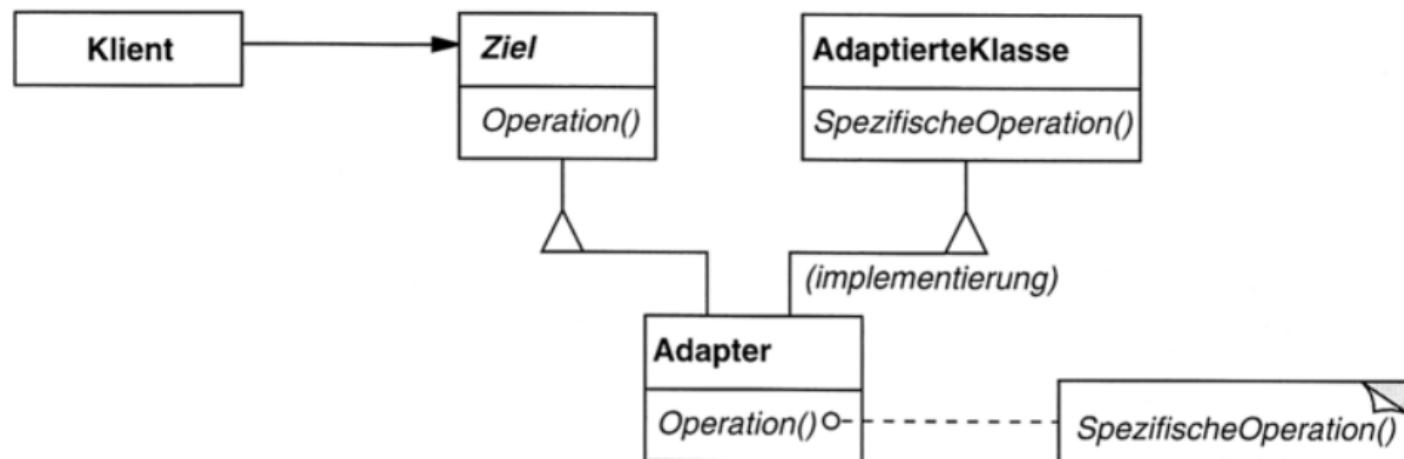
Quelle: Entwurfsmuster, Seite 172

Adapter Muster (2)

- **Anwendbarkeit:**

- Eine existierende Klasse verwenden wollen, deren Schnittstelle aber nicht der von Ihnen benötigten Schnittstelle entspricht.
- Eine wiederverwendbare Klasse erstellen wollen, die mit unabhängigen oder nicht vorhersehbaren Klassen zusammenarbeitet, das heißt, Klassen, die nicht notwendigerweise kompatible Schnittstellen besitzen.

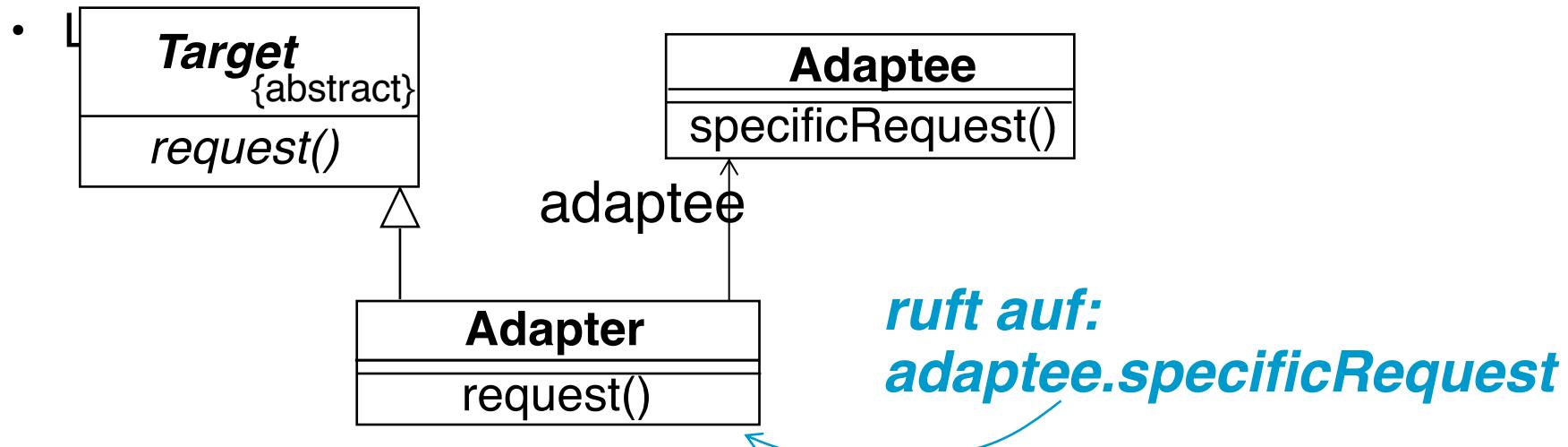
- **Struktur:**



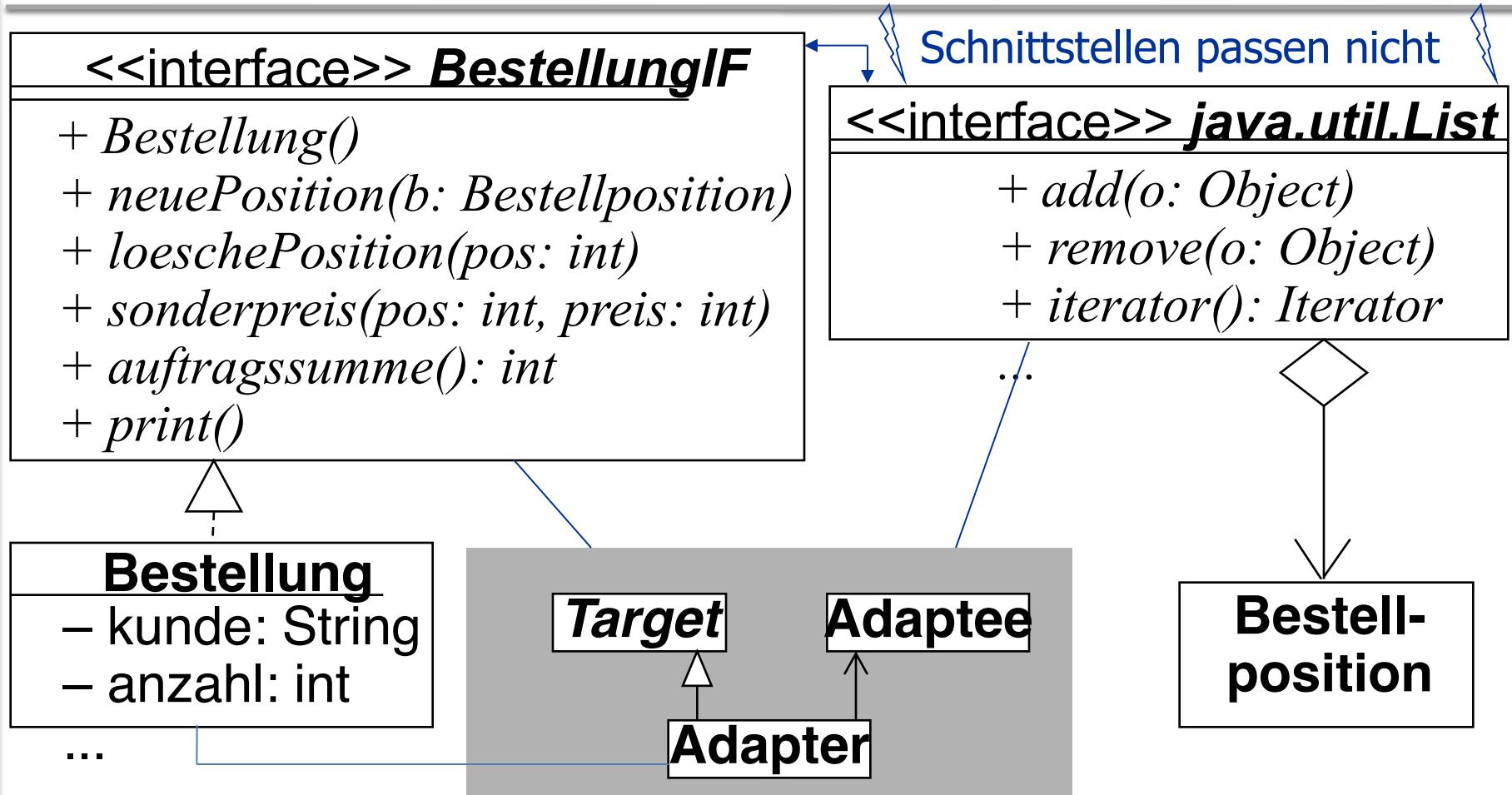
Quelle: Entwurfsmuster, Seite 174

Adapter Muster (3)

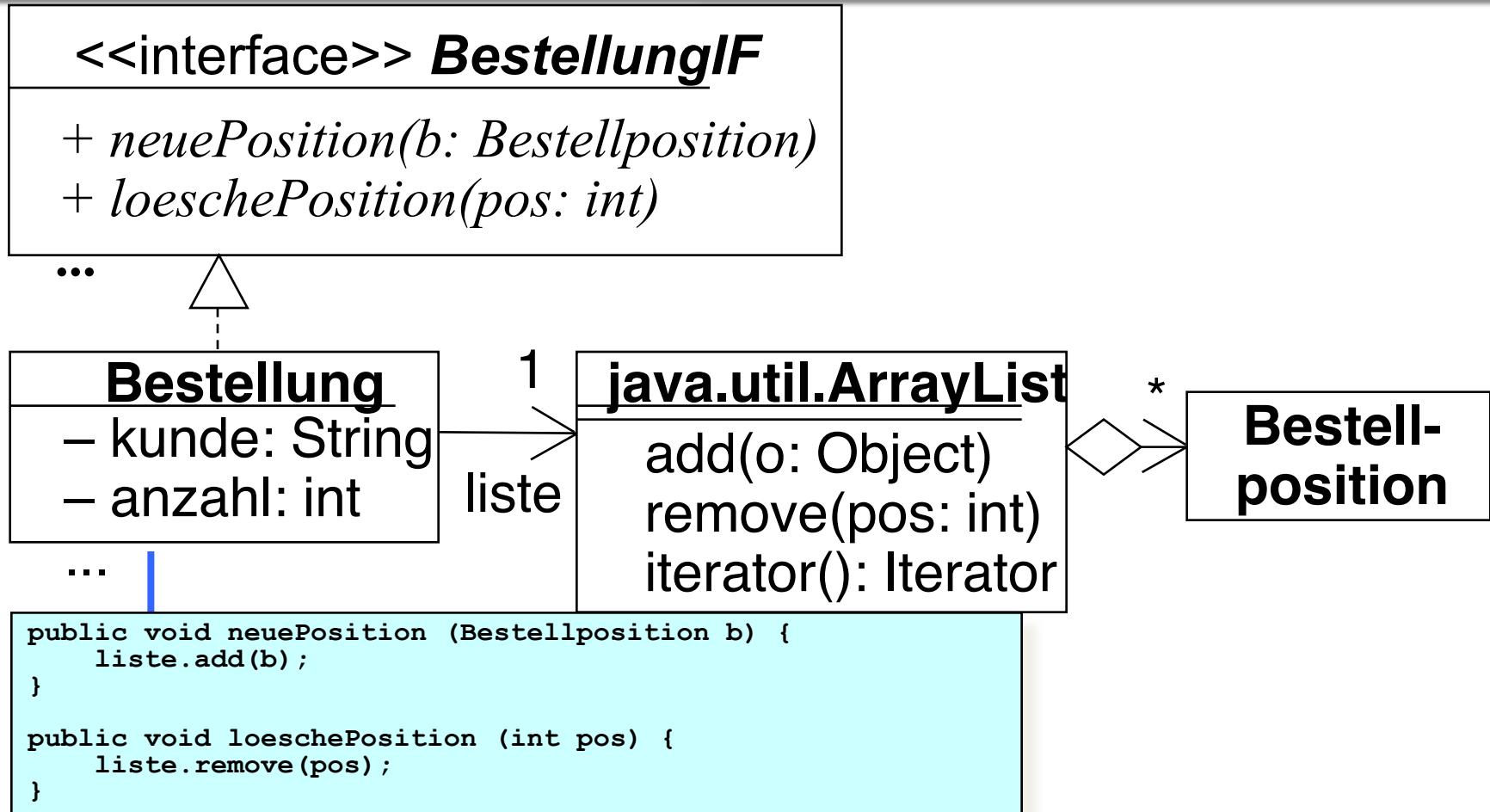
- Name: **Adapter** (auch: Wrapper)
- Problem:
 - Anpassung der Schnittstelle eines vorgegebenen Objekts (*adaptee*) auf eine gewünschte Schnittstelle (*target*)



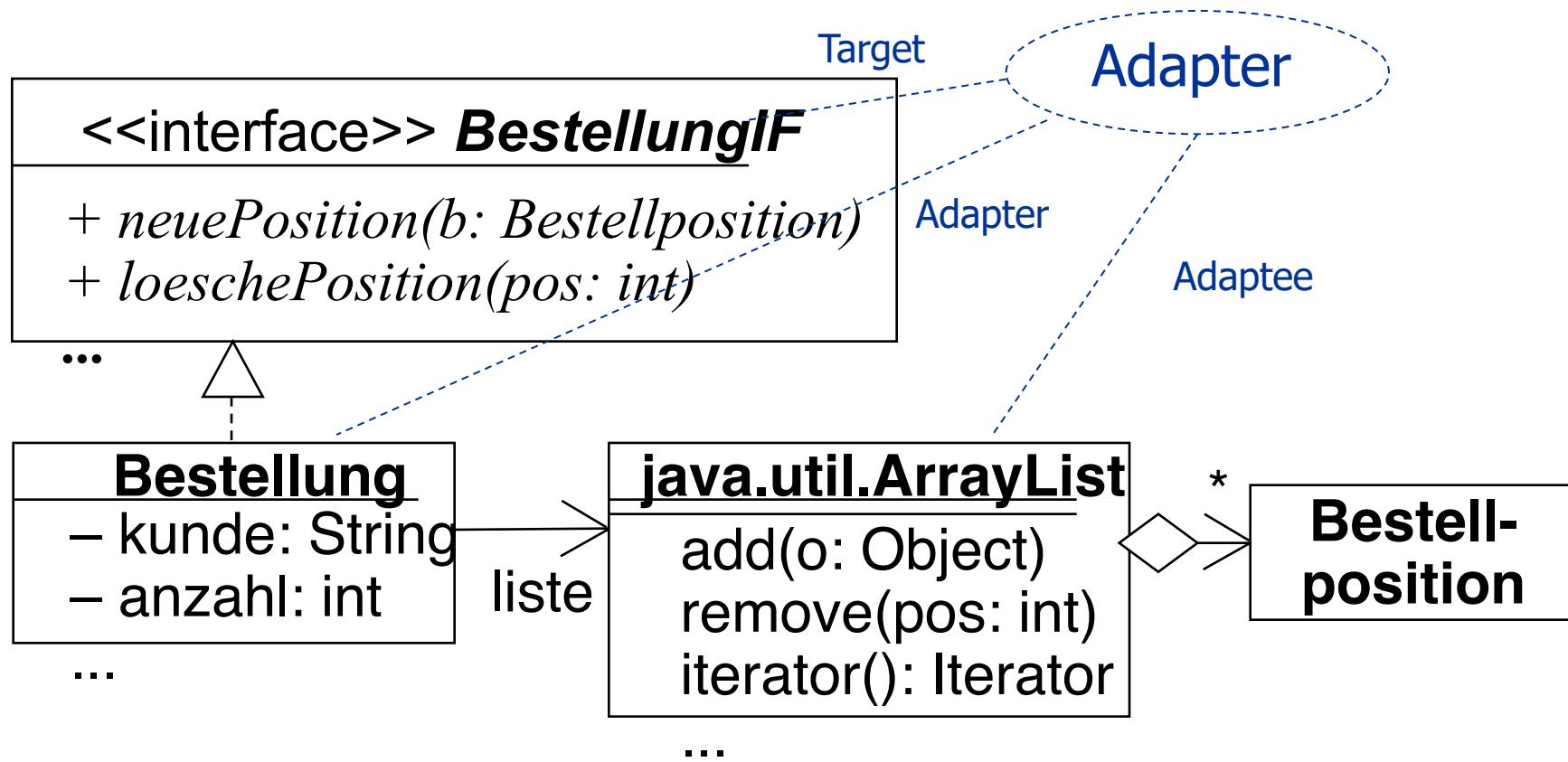
Adapter Muster (4)



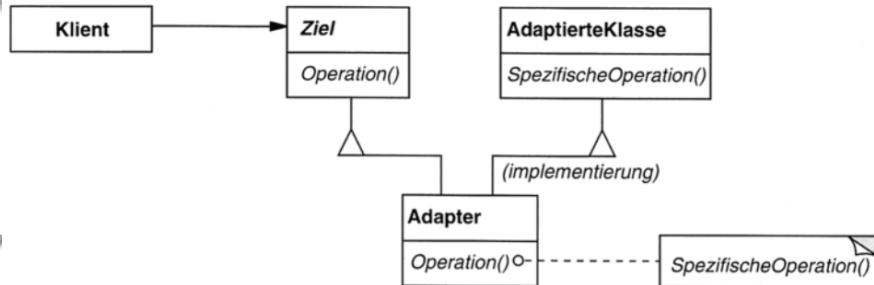
Adapter Muster (5)



Adapter Muster (6)



Adapter Muster – Programmbeispiel (1)



```
// Datei: Person.java
public class Person
{
    private String nachname;
    private String vorname;

    public Person (String nachname, String vorname)
    {
        this.nachname = nachname;
        this.vorname = vorname;
    }

    public void print()
    {
        System.out.println (vorname + " " + nachname);
    }
}
```

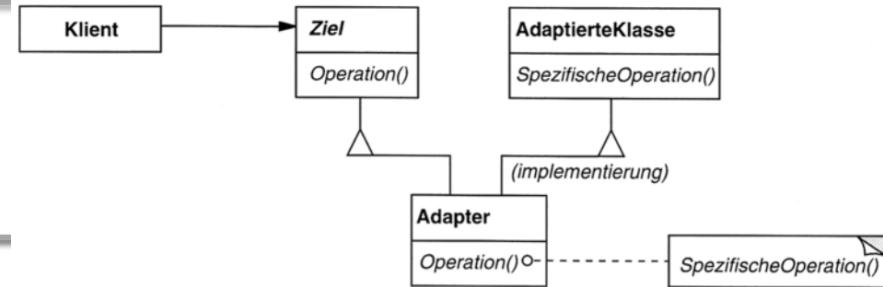
Hilfsklasse

```
// Datei: IPersonenLeser.java
import java.util.Vector;
public interface IPersonenLeser
{
    // hat als Rückgabewert einen Vektor von Personen
    public Vector<Person> lesePersonen();
}
```

Schnittstelle für alle Klassen, die Personendaten einlesen können

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 83ff

Adapter Muster – Programmbeispiel (2)



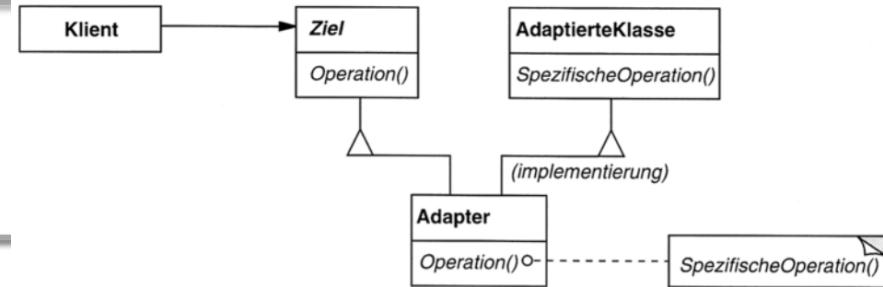
```
// Datei: CSVLeser.java
import java.io.*;
import java.util.Vector;

public class CSVLeser
{
    // hat als Rückgabewert einen Vektor vom Typ eines String-Arrays
    public Vector<String> lesePersonenDatei (String file)
    {
        Vector<String> personen = new Vector<String[]>();
        try
        {
            BufferedReader input =
                new BufferedReader (new FileReader (file));
            String strLine;
            while ((strLine = input.readLine()) != null)
            {
                String[] splitted = strLine.split (",");
                if (splitted.length >= 2)
                    personen.add (new String []
                        {splitted[0], splitted[1]}));
            }
            input.close();
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
        return personen;
    }
}
```

- Zu adaptierende Klasse
- Ließ Personendaten aus einer Datei in den Zwischenpuffer ein
- Name der Methode zum einlesen anders als in IPersonenLeser
- Gibt Array zurück, keinen Vector

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 83ff

Adapter Muster – Programmbeispiel (3)



```
// Datei: CSVLeserAdapter.java
import java.util.Vector;

public class CSVLeserAdapter implements IPersonenLeser
{
    private String file;

    public CSVLeserAdapter (String file)
    {
        this.file = file;
    }

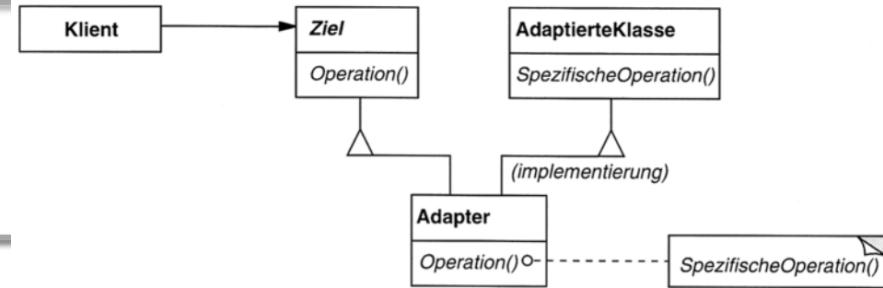
    public Vector<Person> lesePersonen()
    {
        CSVLeser leser = new CSVLeser();
        Vector<String []> gelesenePersonen =
            leser.lesePersonenDatei (file);
        Vector<Person> personenVector = new Vector<Person>();

        for (String [] person : gelesenePersonen)
            personenVector.add (new Person (person [0], person [1]));
        return personenVector;
    }
}
```

- **Adapter**
- Implementiert **IPersonenLeser**
- Delegiert das einlesen der Personen aus der CSV-Datei an ein aggregiertes Objekt der Klasse **CSV-Leser**
- Daten werden aufbereitet, dass sie der Schnittstelle genügen

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 83ff

Adapter Muster – Programmbeispiel (4)



```
// Datei: TestAdapter.java
import java.util.Vector;

public class TestAdapter
{
    public static void main (String[] args)
    {
        IPersonenLeser leser = new CSVLeserAdapter ("Personen.csv");
        Vector<Person> personen = leser.lesePersonen();

        for (Person person : personen)
            person.print();
    }
}
```

- **Klient**

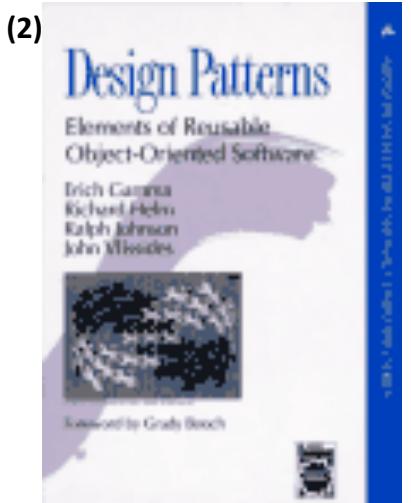
Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 83ff

Überblick - Entwurfsmuster

- Struktur
 - Adapter^(1,2)
 - Brücke (Bridge) ^(1,2)
 - Dekorierer (Decorator) ^(1,2)
 - Fassade (Facade) ^(1,2)
 - Fliegengewicht (Flyweight) ⁽²⁾
 - Kompositum (Composite) ^(1,2)
 - Proxy^(1,2)
- Verhalten
 - Befehl (Command) ^(1,2)
 - Beobachter (Observer) ^(1,2)
 - Besucher (Visitor) ^(1,2)
 - Interpreter (Interpreter) ⁽²⁾
 - Iterator^(1,2)
 - Memento⁽²⁾
 - Rolle⁽¹⁾
 - Schablonenmethode (Template Method) ^(1,2)
 - Strategie (Strategy) ^(1,2)
 - Vermittler (Mediator) ^(1,2)
 - Zustand (State) ^(1,2)
 - Zuständigkeitskette (Chain of Responsibility) ⁽²⁾
- Erzeugung
 - Abstrakte Fabrik (Abstract Factory) ^(1,2)
 - Erbauer (Builder) ⁽²⁾
 - Fabrikmethode (Factory Method) ^(1,2)
 - Objektpool⁽¹⁾
 - Prototyp (Prototype) ^(1,2)
 - Singleton^(1,2)

Legende:

Fett = wird in der Vorlesung behandelt

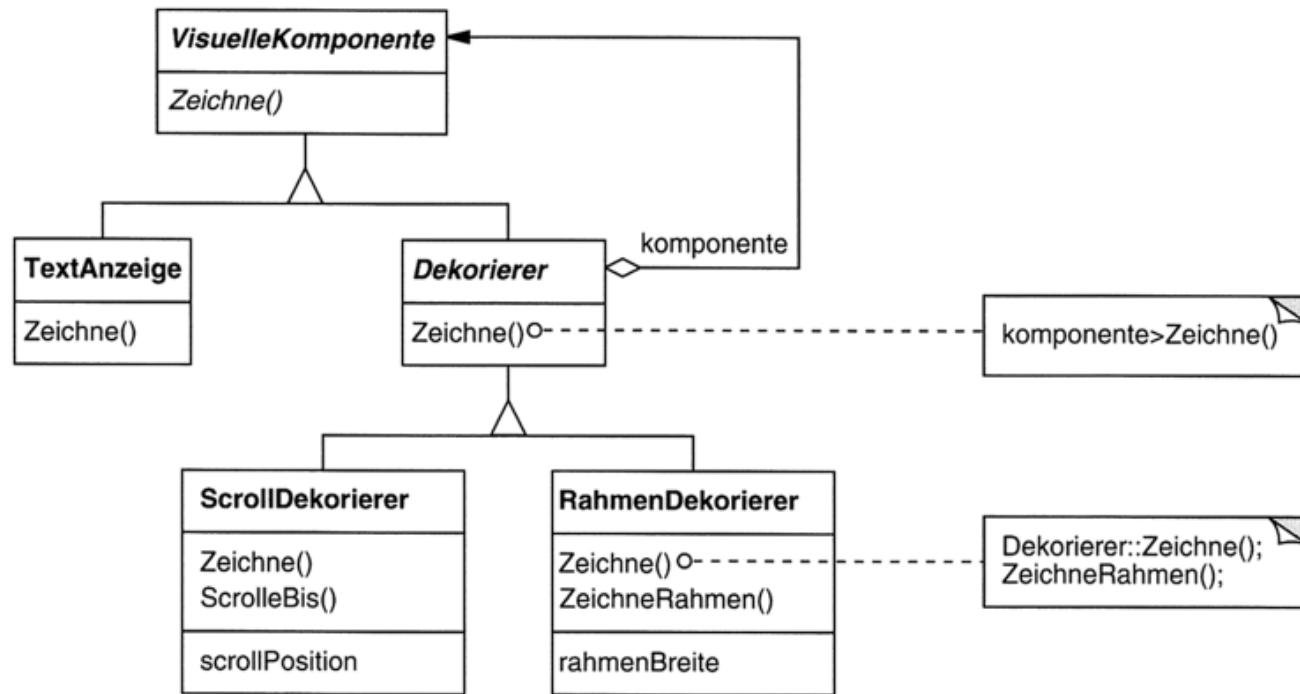


Decorator (1)

- **Name:** Decorator (dt. Dekorierer)

- **Zweck:** Erweitere ein Objekt dynamisch um Zuständigkeiten.

- **Motivation:**



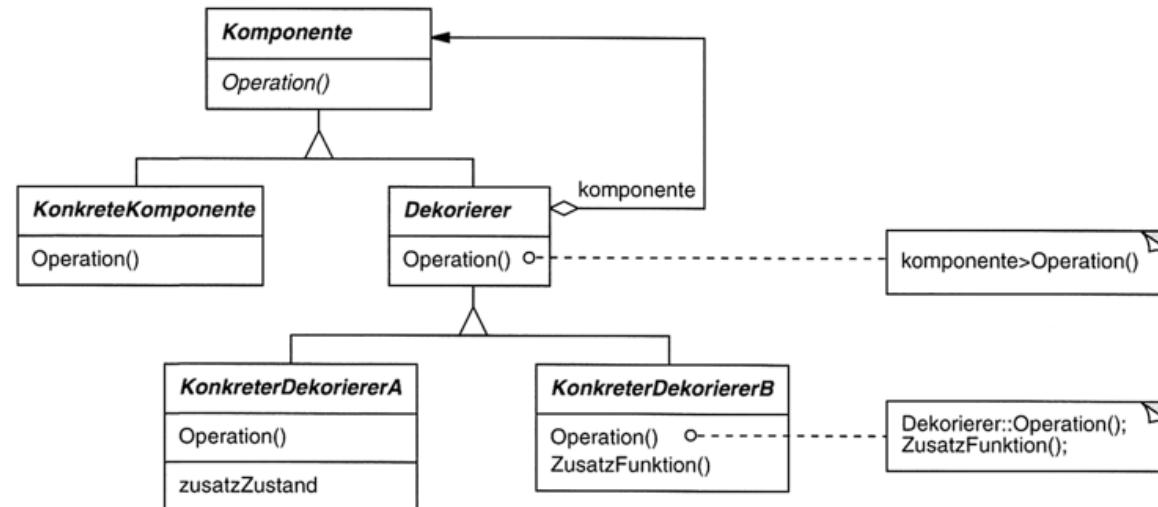
Quelle: Entwurfsmuster, Seite 201

Decorator (2)

- **Anwendbarkeit:**

- Um einzelnen Objekte zusätzliche Funktionalität dynamisch und transparent hinzuzufügen, ohne andere Objekte mit einzubeziehen.
- Um Funktionalität hinzufügen, die auch wieder entfernt werden kann.
- Wenn die Erweiterung mittels Unterklassenbildung nicht praktisch durchführbar ist.

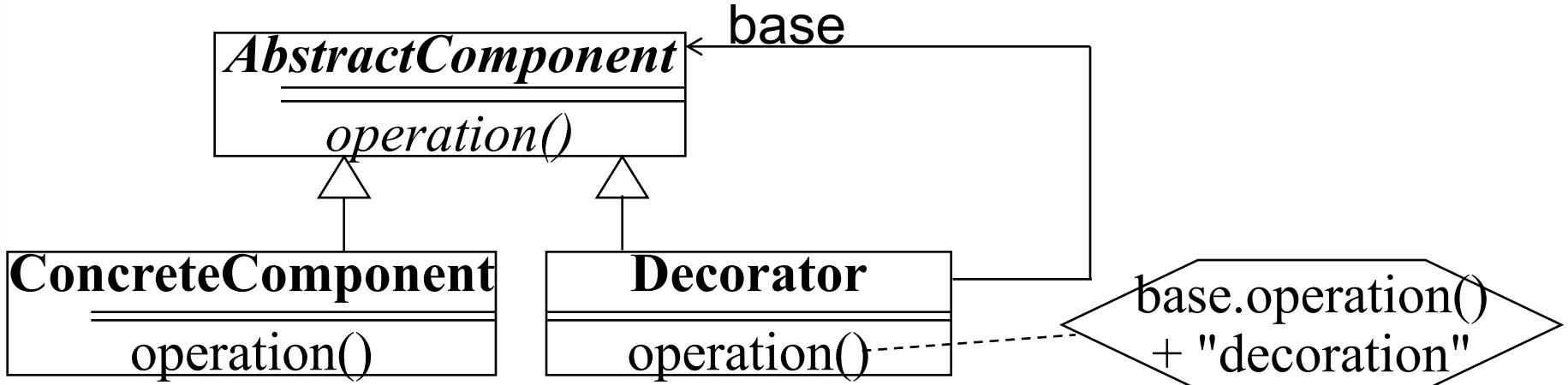
- **Struktur:**



Quelle: Entwurfsmuster, Seite 202

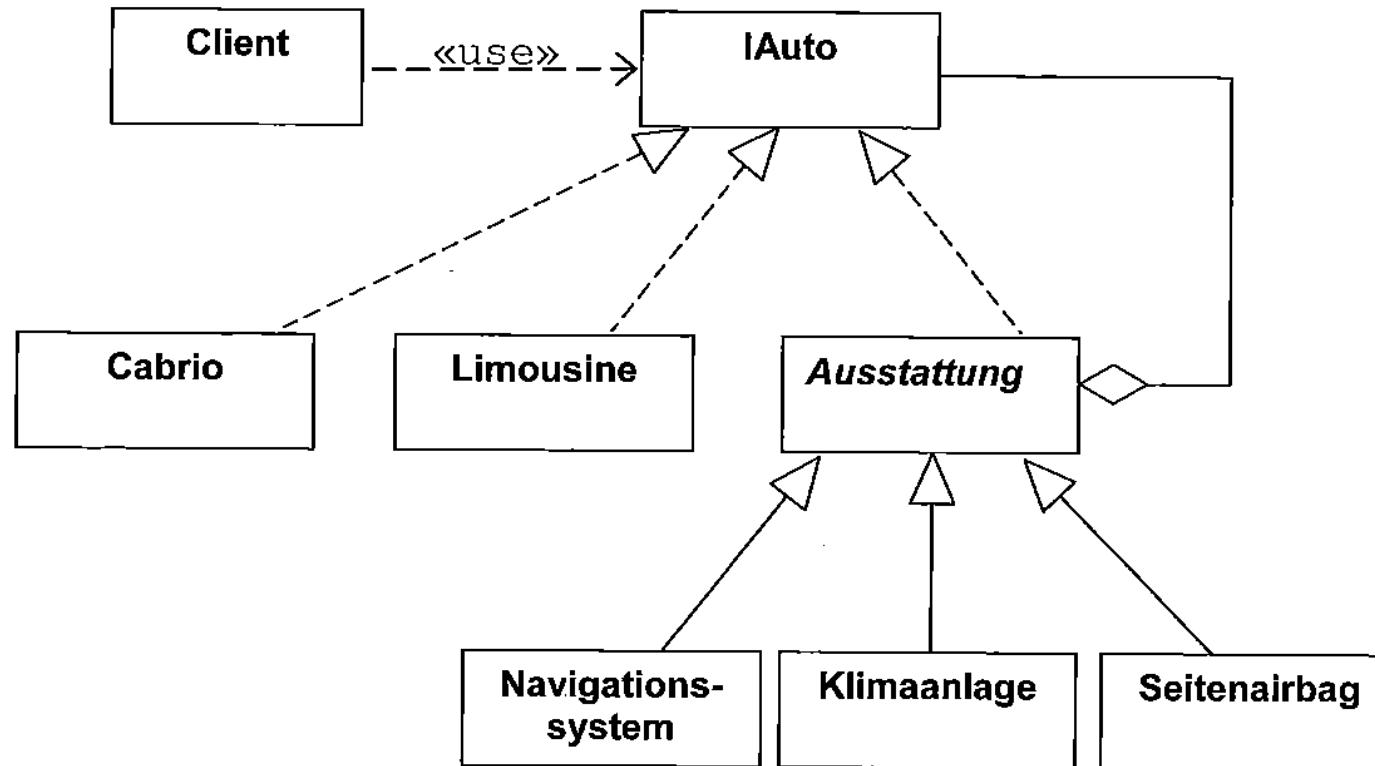
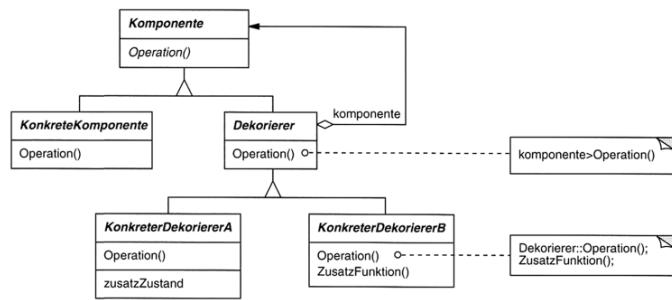
Decorator (3)

- **Problem:** Zu einer Klasse, die eine abstrakte Schnittstelle implementiert, sollen flexibel weitere Eigenschaften hinzugefügt werden, so dass eine neue Klasse entsteht, die die gleiche Schnittstelle implementiert.
- **Lösung:** Definition einer Klasse für Zwischenobjekte, die die Operationen an die Originalklasse delegieren, nachdem sie ggf. zusätzliche Funktionalität erbracht haben.



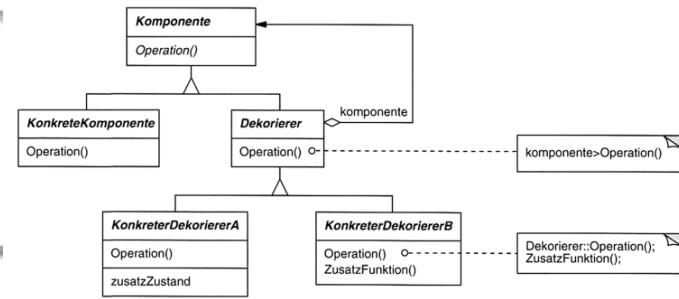
Achtung: Etwas vereinfachte Version
(ohne abstrakte Decorator-Oberklasse)

Decorator – Programmbeispiel (1)



Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 105ff

Decorator – Programmbeispiel (2)



```
// Datei: IAuto.java
public interface IAuto
{
    public int gibKosten();
    public void zeigeDetails();
}
```

```
// Datei: Limousine.java
class Limousine implements IAuto
{
    public void zeigeDetails()
    {
        System.out.print ("Limousine");
    }
    public int gibKosten()
    {
        return 35000;
    }
}
// Datei: Cabrio.java
class Cabrio implements IAuto
{
    public void zeigeDetails()
    {
        System.out.print ("Cabrio");
    }

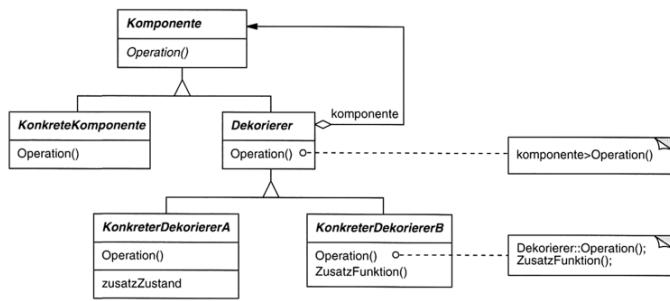
    public int gibKosten()
    {
        return 50000;
    }
}
```

- Komponente

- Konkrete Komponenten
- Methoden geben Art des Autos und Grundkosten zurück

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 105ff

Decorator – Programmbeispiel (3)



```
// Datei: Ausstattung.java
public abstract class Ausstattung implements IAuto
{
    protected IAuto auto;

    public Ausstattung (IAuto pIAuto)
    {
        auto = pIAuto;
    }
}
```

- **Abstrakter Dekorierer**

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 105ff

Decorator – Programmbeispiel (4)

```
// Datei: Klimaanlage.java
class Klimaanlage extends Ausstattung
{
    public Klimaanlage(IAuto pIAuto)
    {
        super(pIAuto);
    }

    public void zeigeDetails() // "dekoriert" die Details
    {
        auto.zeigeDetails();
        System.out.print ("", Klimaanlage");
    }

    public int gibKosten() // "dekoriert" die Kosten
    {
        return auto.gibKosten() + 1500;
    }
}

// Datei: Seitenairbags.java
class Seitenairbags extends Ausstattung
{
    public Seitenairbags(IAuto pIAuto)
    {
        super (pIAuto);
    }

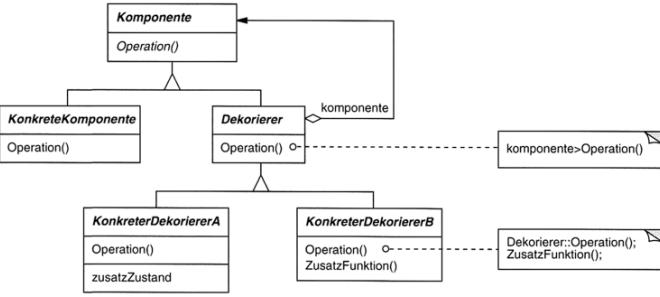
    public void zeigeDetails() // "dekoriert" die Details
    {
        auto.zeigeDetails();
        System.out.print("", Seitenairbags");
    }

    public int gibKosten() // "dekoriert" die Kosten
    {
        return auto.gibKosten() + 1000;
    }
}
```

```
// Datei: Navigationssystem.java
class Navigationssystem extends Ausstattung
{
    public Navigationssystem (IAuto pIAuto)
    {
        super (pIAuto);
    }

    public void zeigeDetails() // "dekoriert" die Details
    {
        auto.zeigeDetails();
        System.out.print ("", Navigationssystem");
    }

    public int gibKosten() // "dekoriert" die Kosten
    {
        return auto.gibKosten() + 2500;
    }
}
```



- **Konkrete Decorierer**

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 105ff

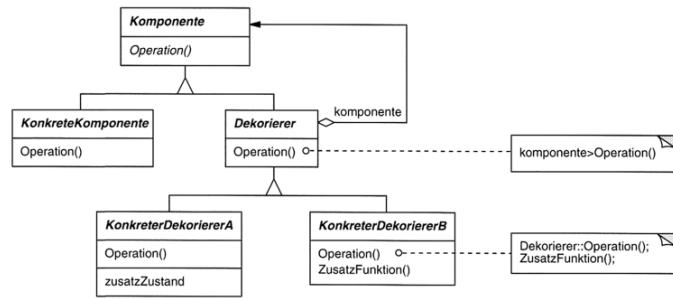
Decorator – Programmbeispiel (5)

```
// Datei: Client.java
class Client
{
    public static void main(String[] args)
    { // Auto mit Klimaanlage
        IAuto auto = new Klimaanlage (new Limousine());
        auto.zeigeDetails();
        System.out.println ("\nfuer " + auto.gibKosten() +
                           " Euro\n");

        // Dynamische Erweiterung der Limousine mit Ausstattungen
        auto = new Navigationssystem (new Seitenairbags(auto));
        auto.zeigeDetails();
        System.out.println ("\nfuer " + auto.gibKosten() +
                           " Euro\n");

        // Cabrio Variante
        auto = new Navigationssystem (new Seitenairbags(new Cabrio()));
        auto.zeigeDetails();

        System.out.println ("\nfuer " + auto.gibKosten() +
                           " Euro\n");
    }
}
```



- **Klient**
- Zeigt die Dekoration von Komponenten mit verschiedenen Ausstattungen

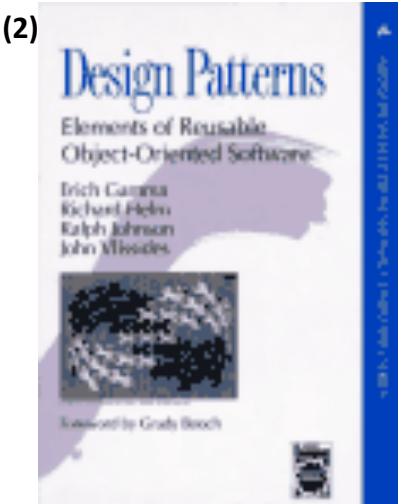
Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 105ff

Überblick - Entwurfsmuster

- Struktur
 - Adapter^(1,2)
 - Brücke (Bridge)^(1,2)
 - Dekorierer (Decorator)^(1,2)
 - Fassade (Facade)^(1,2)
 - Fliegengewicht (Flyweight)⁽²⁾
 - Kompositum (Composite)^(1,2)
 - Proxy^(1,2)
- Verhalten
 - Befehl (Command)^(1,2)
 - Beobachter (Observer)^(1,2)
 - Besucher (Visitor)^(1,2)
 - Interpreter (Interpreter)⁽²⁾
 - Iterator^(1,2)
 - Memento⁽²⁾
 - Rolle⁽¹⁾
 - Schablonenmethode (Template Method)^(1,2)
 - Strategie (Strategy)^(1,2)
 - Vermittler (Mediator)^(1,2)
 - Zustand (State)^(1,2)
 - Zuständigkeitskette (Chain of Responsibility)⁽²⁾
- Erzeugung
 - Abstrakte Fabrik (Abstract Factory)^(1,2)
 - Erbauer (Builder)⁽²⁾
 - Fabrikmethode (Factory Method)^(1,2)
 - Objektpool⁽¹⁾
 - Prototyp (Prototype)^(1,2)
 - Singleton^(1,2)

Legende:

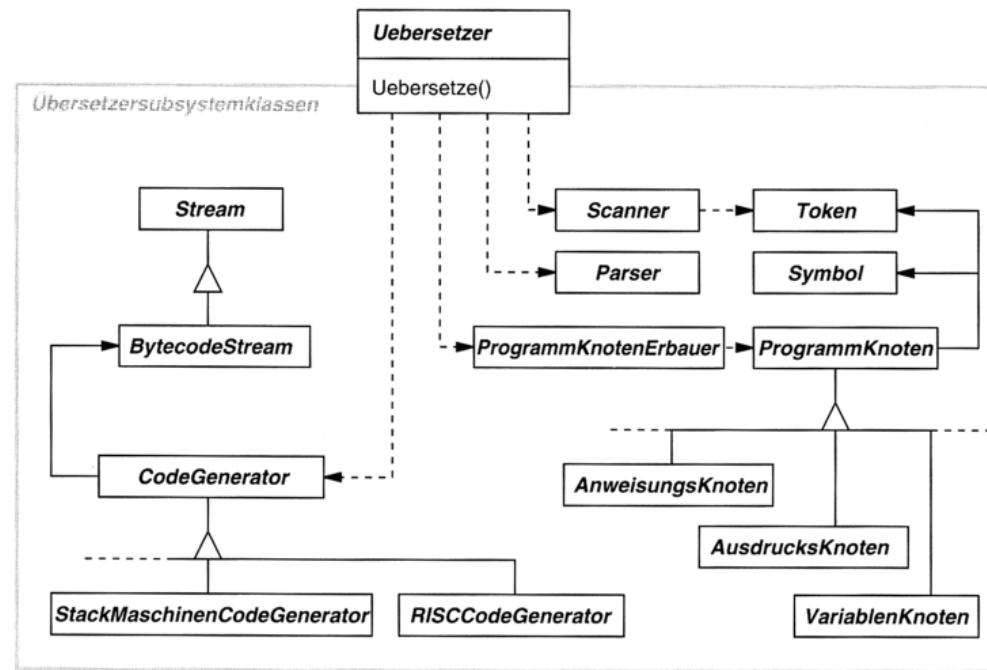
Fett = wird in der Vorlesung behandelt



Facade (1)

- **Name:** Facade (dt. Fassade)
- **Zweck:** Bietet eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems.

Motivation:

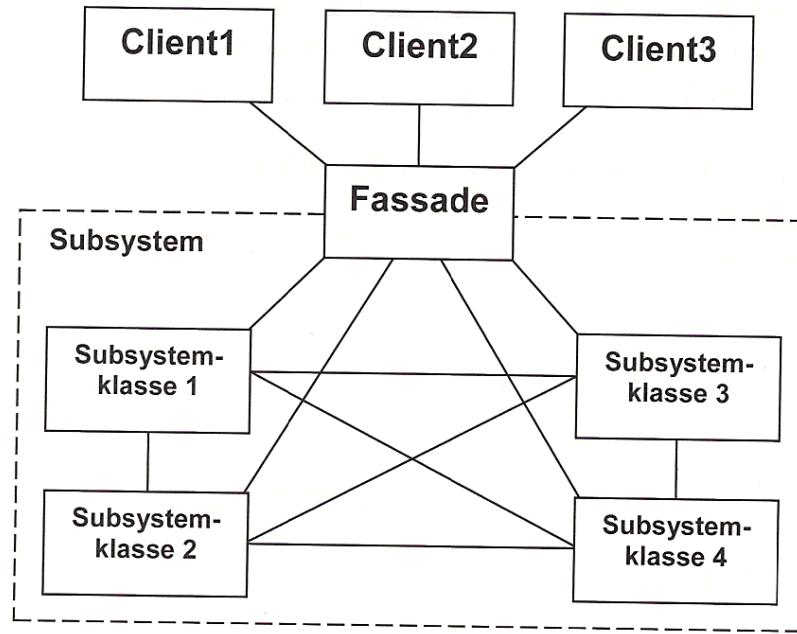


Quelle: Entwurfsmuster, Seite 213

Facade (2)

- **Anwendbarkeit:**
 - Sie eine einfach Schnittstelle zu einem komplexen System anbieten wollen.
 - Es viele Abhängigkeiten zwischen den Klienten und den Implementierungsklassen einer Abstraktion gibt.
 - Sie Ihre Subsysteme in Schichten aufteilen wollen.

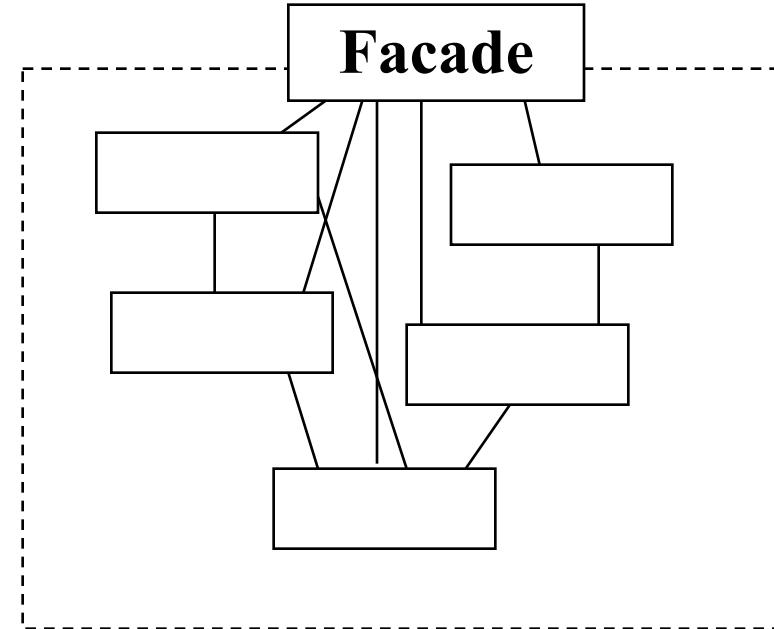
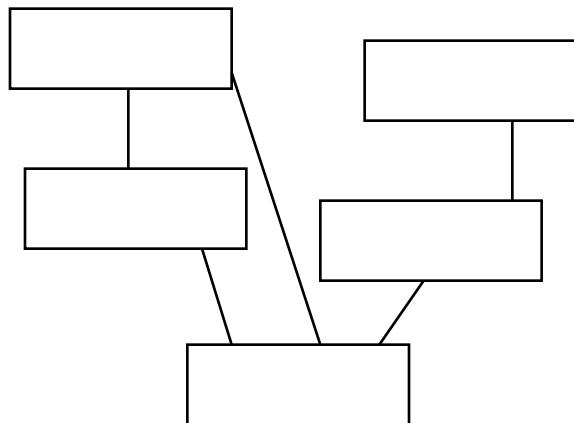
- **Struktur:**



Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 117

Facade Muster (3)

- **Problem:** Ein komplexes Subsystem bietet eine Vielzahl von Schnittstellen an, die vereinheitlicht werden sollen.
- **Lösung:** Definition einer einheitlichen abstrakten Schnittstelle und Abbildung auf die bestehenden Schnittstellen durch Delegation (wie beim Objektadapter)



Facade – Programmbeispiel (1)

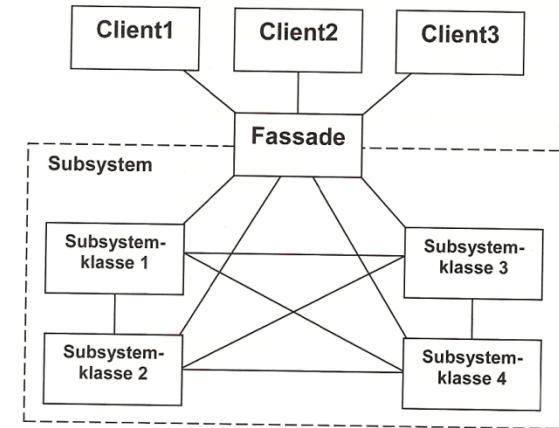
```
// Datei: FahrwerkLager.java
public class FahrwerkLager
{
    int fahrwerkAnzahl = 0;

    public void lagerFuellen (int anzahl)
    {
        fahrwerkAnzahl = fahrwerkAnzahl + anzahl;
        System.out.println ("Lager wurde um " + anzahl +
                           " Fahrwerke aufgestockt.");
    }

    public void fahrwerkEntnehmen (int anzahl)
    {
        fahrwerkAnzahl = fahrwerkAnzahl - anzahl;
        System.out.println ("Fuer die Produktion wurden " + anzahl +
                           " Fahrwerke entnommen.");
    }
}

// Datei: GetriebeLager.java
public class GetriebeLager
{
    int getriebeAnzahl = 0;
    public void lagerFuellen (int anzahl)
    {
        getriebeAnzahl = getriebeAnzahl + anzahl;
        System.out.println ("Lager wurde um " + anzahl +
                           " Getriebe aufgestockt.");
    }

    public void getriebeEntnehmen (int anzahl)
    {
        getriebeAnzahl = getriebeAnzahl - anzahl;
        System.out.println ("Fuer die Produktion wurden " + anzahl +
                           " Getriebe entnommen.");
    }
}
```



• Subsystemklassen

```
// Datei: MotorLager.java
public class MotorLager
{
    int motorenAnzahl = 0;

    public void lagerFuellen (int anzahl)
    {
        motorenAnzahl = motorenAnzahl + anzahl;
        System.out.println ("Lager wurde um " + anzahl +
                           " Motoren aufgestockt.");
    }

    public void motorEntnehmen (int anzahl)
    {
        motorenAnzahl = motorenAnzahl - anzahl;
        System.out.println ("Fuer die Produktion wurden " + anzahl +
                           " Motoren entnommen.");
    }
}
```

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 118ff

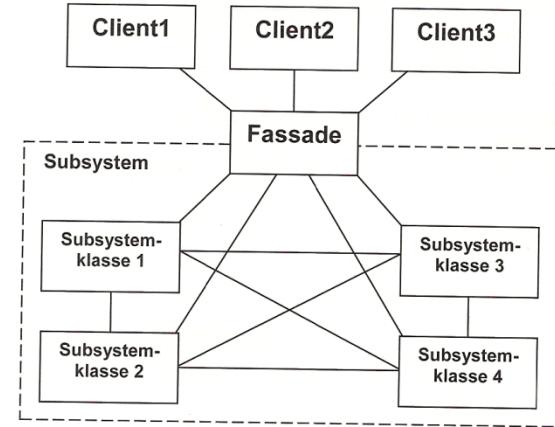
Facade – Programmbeispiel (2)

```
// Datei: LagerFassade.java
public class LagerFassade
{
    private FahrwerkLager fw;
    private GetriebeLager g;
    private MotorLager m;

    LagerFassade()
    {
        fw = new FahrwerkLager();
        g = new GetriebeLager();
        m = new MotorLager();
    }

    public void alleLagerFuellen (int anzahl)
    {
        System.out.println ("Bestaende aller Lager werden " +
                            "gefuellt.");
        fw.lagerFuellen (anzahl);
        g.lagerFuellen (anzahl);
        m.lagerFuellen (anzahl);
        System.out.println();
    }

    public void produktionsteileEntnehmen (int anzahl)
    {
        System.out.println ("Alle fuer die Produktion " +
                            "notwendigen Teile werden entnommen.");
        fw.fahrwerkEntnehmen (anzahl);
        g.getriebeEntnehmen (anzahl);
        m.motorEntnehmen (anzahl);
        System.out.println();
    }
}
```



- Fassade
- Ermöglicht gemeinsame Aufstockung und Abbuchung von Lagerteilen aus allen drei Lagern durch die Methoden
 - alleLagerFüllen()
 - produktionsteileEntnehmen

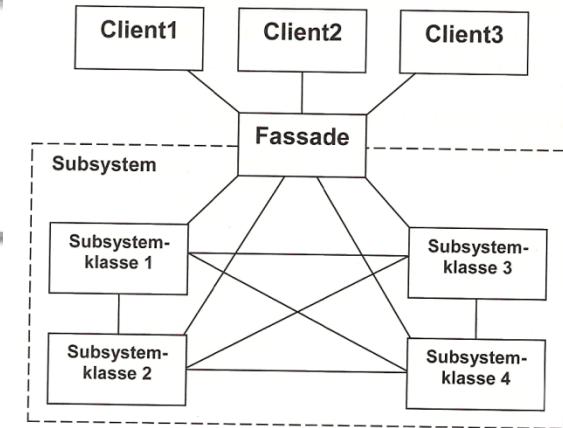
Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 118ff

Facade – Programmbeispiel (3)

```
// Datei: TestFacade.java
public class TestFacade
{
    public static void main (String[] args)
    {
        LagerFacade fassade = new LagerFacade();

        // Lager werden durch Lagerverwaltung gefuellt.
        fassade.alleLagerFuellen (10);

        // Teile werden durch die Produktion entnommen.
        fassade.produktionsteileEntnehmen (5);
    }
}
```



Klient

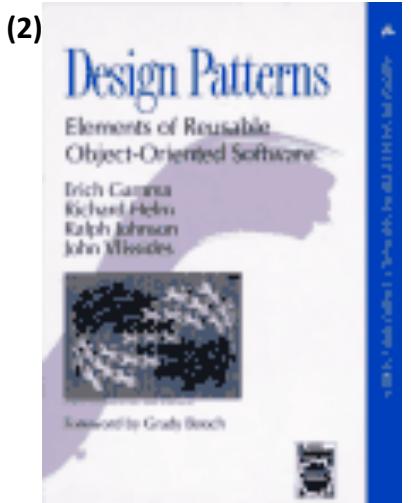
Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 118ff

Überblick - Entwurfsmuster

- Struktur
 - Adapter^(1,2)
 - Brücke (Bridge) ^(1,2)
 - Dekorierer (Decorator) ^(1,2)
 - Fassade (Facade) ^(1,2)
 - Fliegengewicht (Flyweight) ⁽²⁾
 - Kompositum (Composite) ^(1,2)
 - Proxy^(1,2)
- Verhalten
 - Befehl (Command) ^(1,2)
 - Beobachter (Observer) ^(1,2)
 - Besucher (Visitor) ^(1,2)
 - Interpreter (Interpreter) ⁽²⁾
 - Iterator^(1,2)
 - Memento⁽²⁾
 - Rolle⁽¹⁾
 - Schablonenmethode (Template Method) ^(1,2)
 - Strategie (Strategy) ^(1,2)
 - Vermittler (Mediator) ^(1,2)
 - Zustand (State) ^(1,2)
 - Zuständigkeitskette (Chain of Responsibility) ⁽²⁾
- Erzeugung
 - Abstrakte Fabrik (Abstract Factory) ^(1,2)
 - Erbauer (Builder) ⁽²⁾
 - Fabrikmethode (Factory Method) ^(1,2)
 - Objektpool⁽¹⁾
 - Prototyp (Prototype) ^(1,2)
 - Singleton^(1,2)

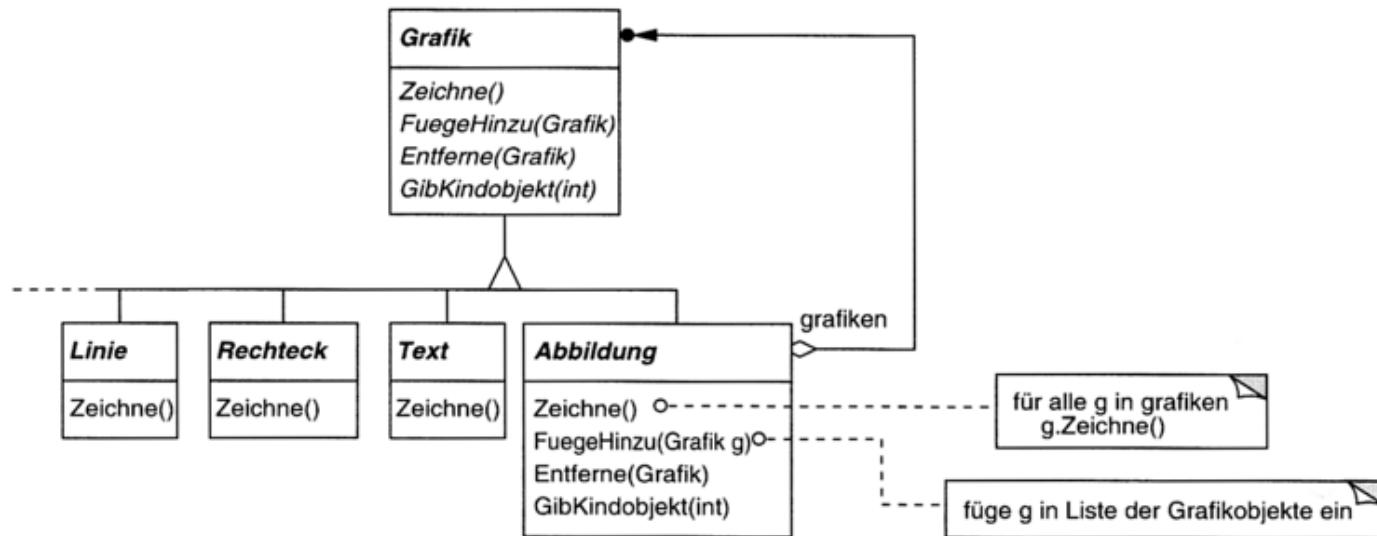
Legende:

Fett = wird in der Vorlesung behandelt



Composite (1)

- **Name:** Composite (dt. Kompositum)
- **Zweck:** Repräsentiere baumartige Hierarchien mit Hilfe von Objekten. Auf einfache und zusammengesetzte Objekte kann in der gleichen Weise zugegriffen werden.
- **Motivation:**



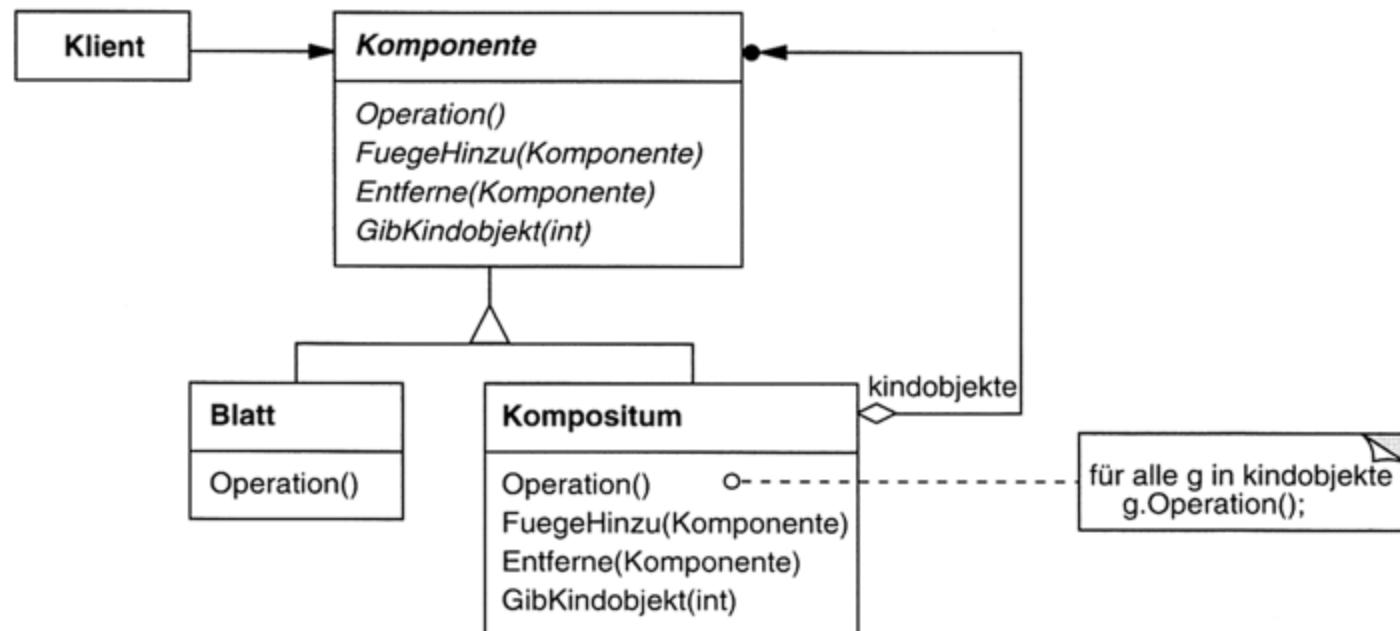
Quelle: Entwurfsmuster, Seite 240

Composite (2)

- **Anwendbarkeit:**

- Darstellung von Hierarchien (Whole-Part-Structures)
- Für Clients soll kein Unterschied in der Schnittstelle von zusammengesetzten und einfachen Objekten sein

- **Struktur:**



Quelle: Entwurfsmuster, Seite 241

Composite (3)

- **Teilnehmer:**

- **Komponente (Grafik):**
 - Definiert gemeinsame Schnittstelle von einfachen und zusammengesetzten Objekten
 - Implementiert im Einzelfall nötiges Standardverhalten für alle Klassen
 - Optional: definiert Schnittstelle für Zugriff auf den Vaterknoten
- **Blatt (Rechteck, Linie, Text):**
 - Repräsentiert einfache Objekte ohne Kinderknoten
 - Implementiert das Verhalten einfacher Objekte
- **Kompositum (Abbildung):**
 - Implementiert das Verhalten zusammengesetzter Objekte
 - Speichert (Verweise auf) Kind-Objekte
 - Implementiert Operationen für Management der Kinder (Hinzufügen, Löschen, ...)
- **Klient:**
 - Manipuliert hierarchische Strukturen über die Schnittstelle der Klasse Component

- **Interaktion:**

- Vaterknoten können Aufträge an ihre Kinder weiterleiten und davor/danach noch eigene Operationen durchführen

Composite (4)

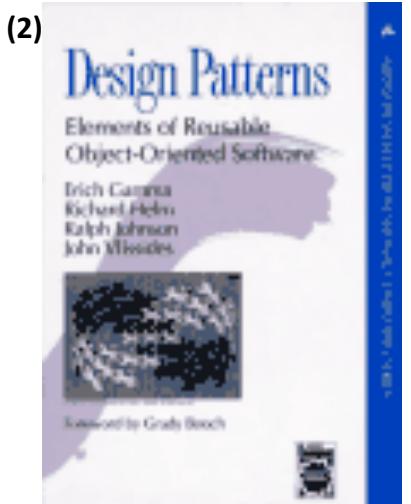
- **Vor- und Nachteile:**
 - + Einheitliche Schnittstelle für Clients erlaubt es, Clients einfach zu halten
 - + Neue Component-Klassen können leicht über Vererbung hinzugefügt werden, ohne dass die Clients geändert werden müssen
 - Über die uniforme Schnittstelle zum Einfügen beliebiger Component-Objekte können beliebige Baumstrukturen erzeugt werden; Einschränkungen müssen über Runtime-Checks durchgeführt werden
- **Implementierung:**
 - Referenzen von Kindern zu ihrem Vaterknoten
 - Sharing von Component-Objekten
 - Breite der Schnittstelle von Component
 - addChild/deleteChild-Operationen
 - Caching
 - ...

Überblick - Entwurfsmuster

- Struktur
 - Adapter^(1,2)
 - Brücke (Bridge) ^(1,2)
 - Dekorierer (Decorator) ^(1,2)
 - Fassade (Facade) ^(1,2)
 - Fliegengewicht (Flyweight) ⁽²⁾
 - Kompositum (Composite) ^(1,2)
 - Proxy^(1,2)
- Verhalten
 - Befehl (Command) ^(1,2)
 - Beobachter (Observer) ^(1,2)
 - Besucher (Visitor) ^(1,2)
 - Interpreter (Interpreter) ⁽²⁾
 - Iterator^(1,2)
 - Memento⁽²⁾
 - Rolle⁽¹⁾
 - Schablonenmethode (Template Method) ^(1,2)
 - Strategie (Strategy) ^(1,2)
 - Vermittler (Mediator) ^(1,2)
 - Zustand (State) ^(1,2)
 - Zuständigkeitskette (Chain of Responsibility) ⁽²⁾
- Erzeugung
 - Abstrakte Fabrik (Abstract Factory) ^(1,2)
 - Erbauer (Builder) ⁽²⁾
 - Fabrikmethode (Factory Method) ^(1,2)
 - Objektpool⁽¹⁾
 - Prototyp (Prototype) ^(1,2)
 - Singleton^(1,2)

Legende:

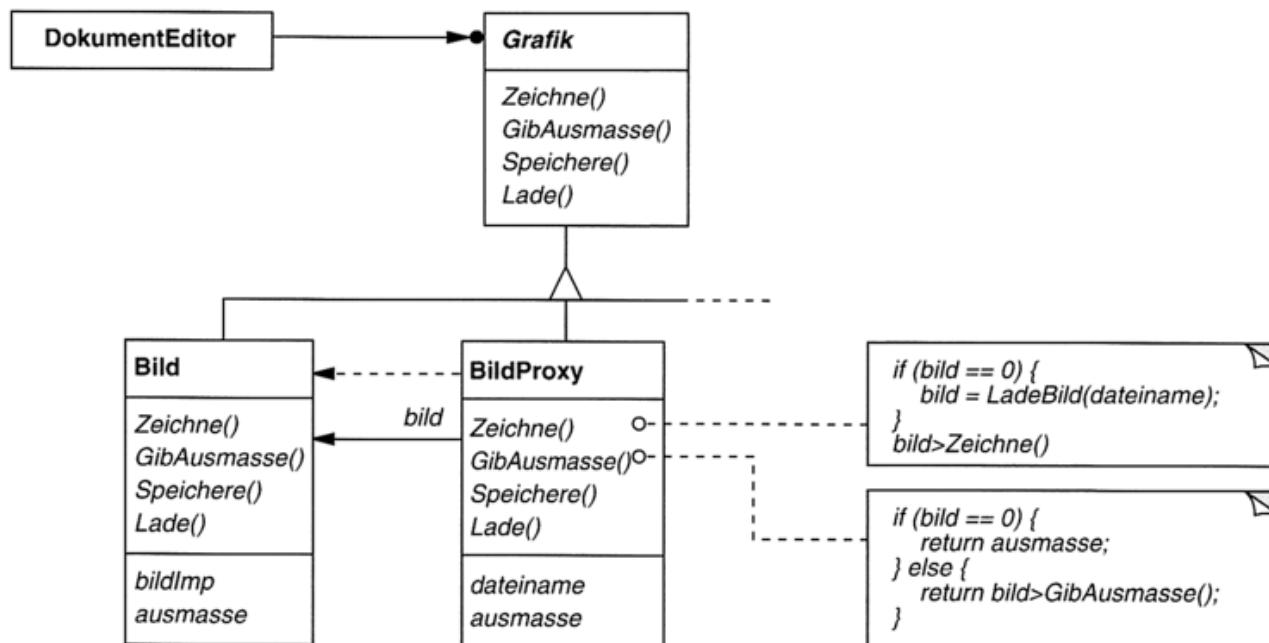
Fett = wird in der Vorlesung behandelt



Proxy (1)

- **Name:** Proxy (dt. Proxy)
- **Zweck:** Kontrolliere den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreterobjekts.

Motivation:



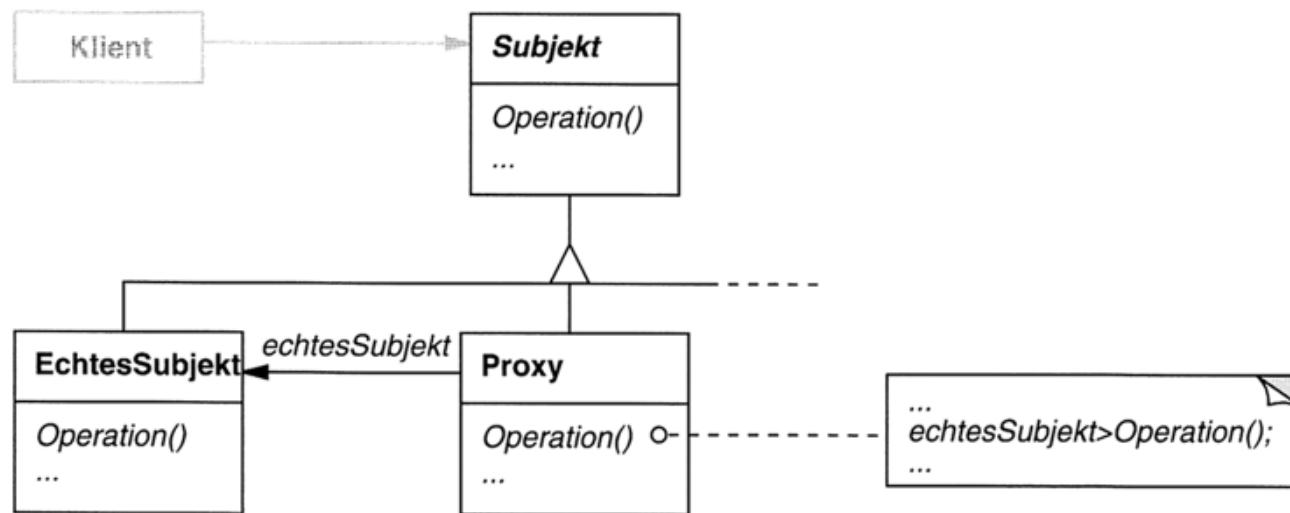
Quelle: Entwurfsmuster, Seite 255

Proxy (2)

- **Anwendbarkeit:**

- Ein **Remote-Proxy** stellt einen lokalen Stellvertreter für ein Objekt in einem anderen Adressraum dar.
- Ein **virtueller Proxy** erzeugt teure Objekte auf Verlangen.
- Ein **Schutz-Proxy** kontrolliert den Zugriff auf das Originalobjekt.

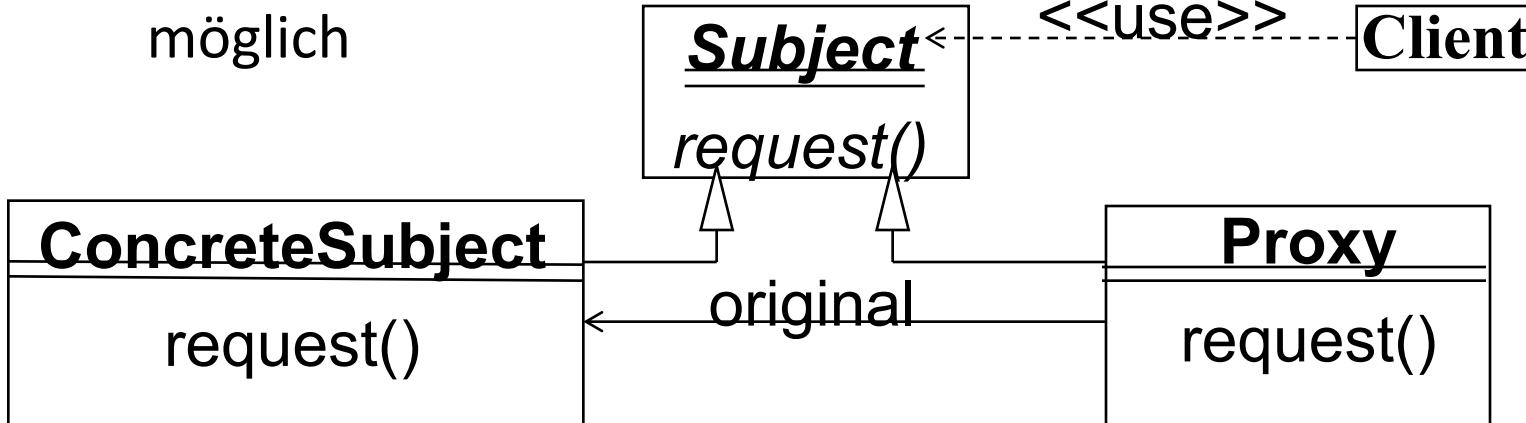
- **Struktur:**



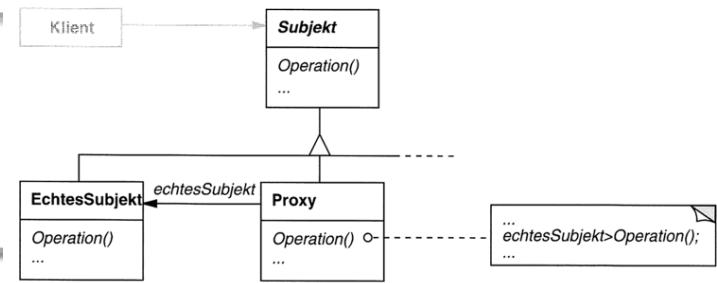
Quelle: Entwurfsmuster, Seite 257

Proxy Muster (3)

- **Problem:** Der direkte Zugriff auf ein Objekt ist problematisch.
- **Lösung:** Stellvertreter-Objekt
 - Alias erlaubt alle Operationen, die auf Originalen möglich sind
 - Weiterleitung von Operationen an das Original
 - Spezielle Interpretation von Operationen in Spezialfällen möglich



Proxy – Programmbeispiel (1)



```
// Datei: IDateizugriff.java
public interface IDateizugriff
{
    public String getName();
    public String getInhalt();
}
```

- **Subject**

```
// Datei: Dateizugriff.java
public class Dateizugriff implements IDateizugriff
{
    String name;

    public Dateizugriff (String name)
    {
        this.name = name;
        System.out.println ("Echtes Objekt instanziert.");
    }

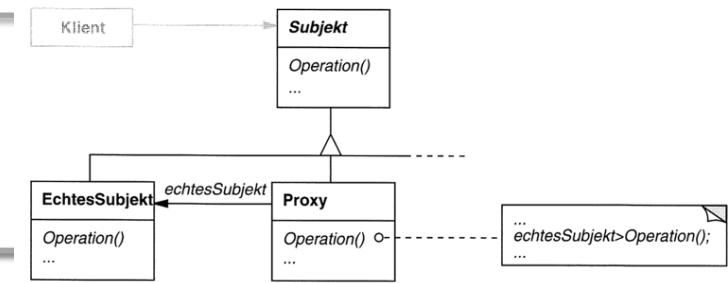
    public String getName()
    {
        return this.name;
    }

    public String getInhalt()
    {
        //Simulation einer Datei-Leseoperation
        return ("Daten von " + name);
    }
}
```

- **Echtes Subject**

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 140f

Proxy – Programmbeispiel (2)



```
// Datei: ProxyDateizugriff.java
public class ProxyDateizugriff implements IDateizugriff
{
    String name;
    IDateizugriff realeDatei;

    public ProxyDateizugriff(String name)
    {
        this.name = name;
        System.out.println("Stellvertretendes Objekt instanziert.");
    }

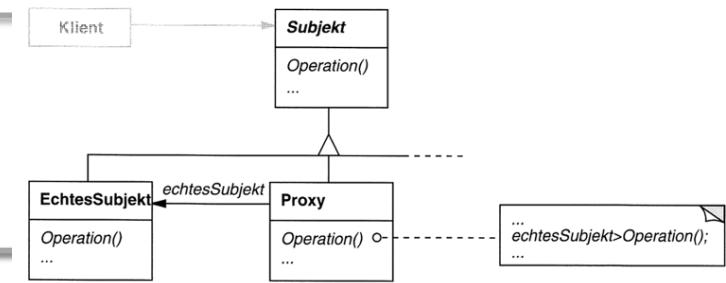
    public String getInhalt()
    {
        if(realeDatei == null)
        {
            System.out.println("Inhalt liegt lokal nicht vor.");
            // Das echte Datei-Objekt wird erzeugt.
            realeDatei = new Dateizugriff(name);
        }
        return realeDatei.getInhalt();
    }

    public String getName()
    {
        return name;
    }
}
```

- **Proxy**
- Speichert den Namen seiner zugehörigen Datei
- Solange ein Client nur die Methode **getName()** aufruft, ist es nicht nötig, die Datei zu öffnen und den Inhalt zu laden
- Erst wenn der Client **getInhalt()** aufruft, muss die Operation über den Zugriff auf das echte Objekt ausgeführt werden
- Dazu wird das Originalobjekt erzeugt, der neben dem Namen auch den Inhalt kennt

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 140f

Proxy – Programmbeispiel (3)



```
// Datei: TestProxy.java
public class TestProxy
{
    public static void main (String[] args)
    {
        // Das Proxy-Objekt wird erzeugt.
        ProxyDateizugriff pDatei =
            new ProxyDateizugriff ("TestDatei.dat");
        System.out.println();
        System.out.println("Name: " + pDatei.getName());
        System.out.println("Inhalt: " + pDatei.getInhalt());
        System.out.println();
        System.out.println("Inhalt: " + pDatei.getInhalt());
    }
}
```

Stellvertretendes Objekt instanziert.

- **Ausgabe**

Name: TestDatei.dat
Inhalt liegt lokal nicht vor.
Echtes Objekt instanziert.
Inhalt: Daten von TestDatei.dat

Inhalt: Daten von TestDatei.dat

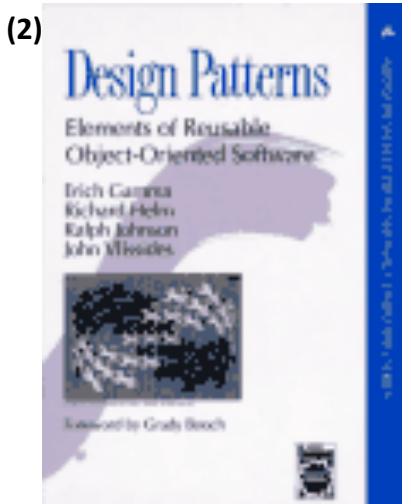
Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 140f

Überblick - Entwurfsmuster

- Struktur
 - Adapter^(1,2)
 - Brücke (Bridge) ^(1,2)
 - Dekorierer (Decorator) ^(1,2)
 - Fassade (Facade) ^(1,2)
 - Fliegengewicht (Flyweight) ⁽²⁾
 - Kompositum (Composite) ^(1,2)
 - Proxy^(1,2)
- Verhalten
 - Befehl (Command) ^(1,2)
 - Beobachter (Observer) ^(1,2)
 - Besucher (Visitor) ^(1,2)
 - Interpreter (Interpreter) ⁽²⁾
 - Iterator^(1,2)
 - Memento⁽²⁾
 - Rolle⁽¹⁾
 - Schablonenmethode (Template Method) ^(1,2)
 - Strategie (Strategy) ^(1,2)
 - Vermittler (Mediator) ^(1,2)
 - Zustand (State) ^(1,2)
 - Zuständigkeitskette (Chain of Responsibility) ⁽²⁾
- Erzeugung
 - Abstrakte Fabrik (Abstract Factory) ^(1,2)
 - Erbauer (Builder) ⁽²⁾
 - Fabrikmethode (Factory Method) ^(1,2)
 - Objektpool⁽¹⁾
 - Prototyp (Prototype) ^(1,2)
 - Singleton^(1,2)

Legende:

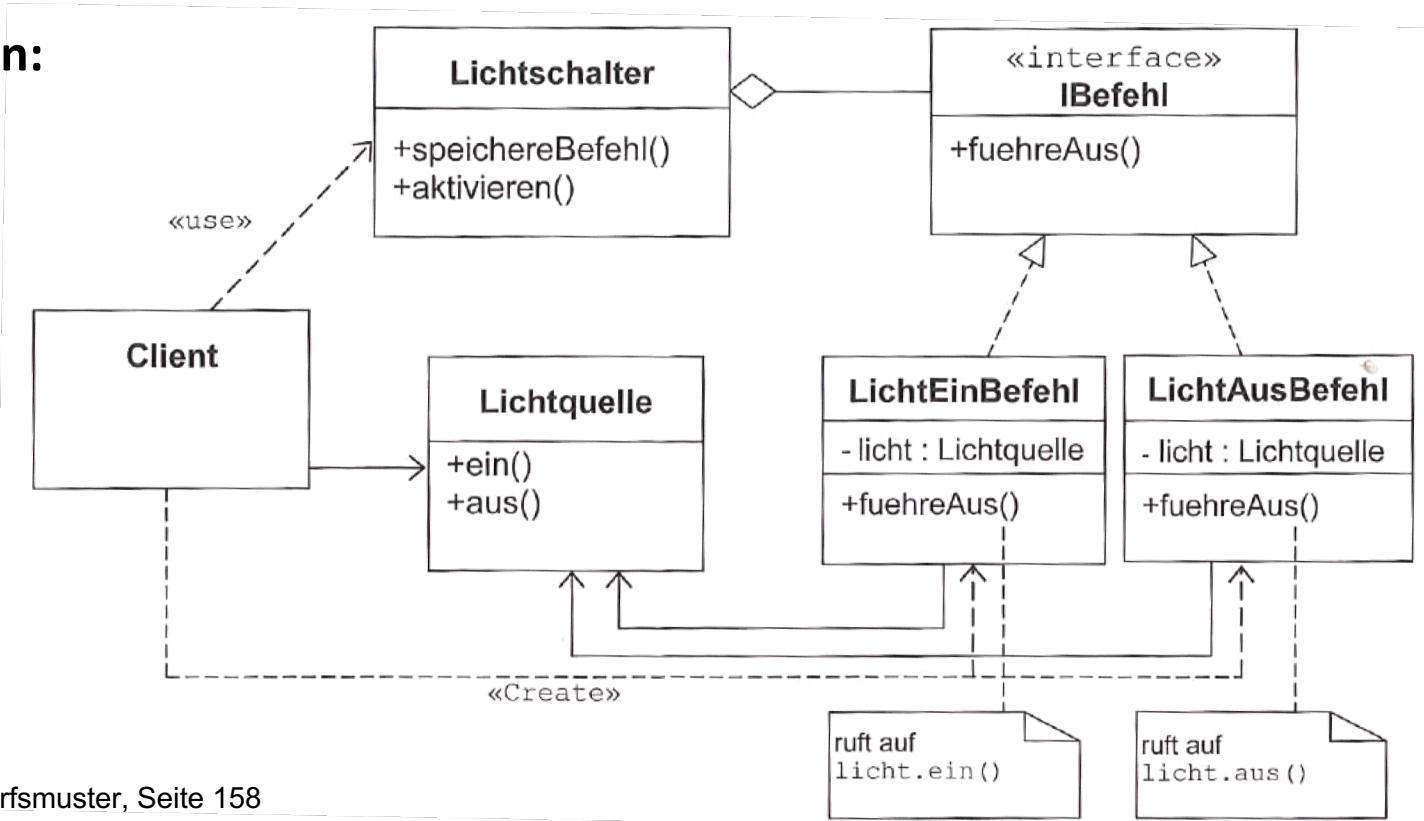
Fett = wird in der Vorlesung behandelt



Command (1)

- **Name:** Command (dt. Befehl / Kommando)
- **Zweck:** Soll einen Befehl (Aktion, Anfrage, Methodenaufruf) in einem Objekt kapseln.

Motivation:



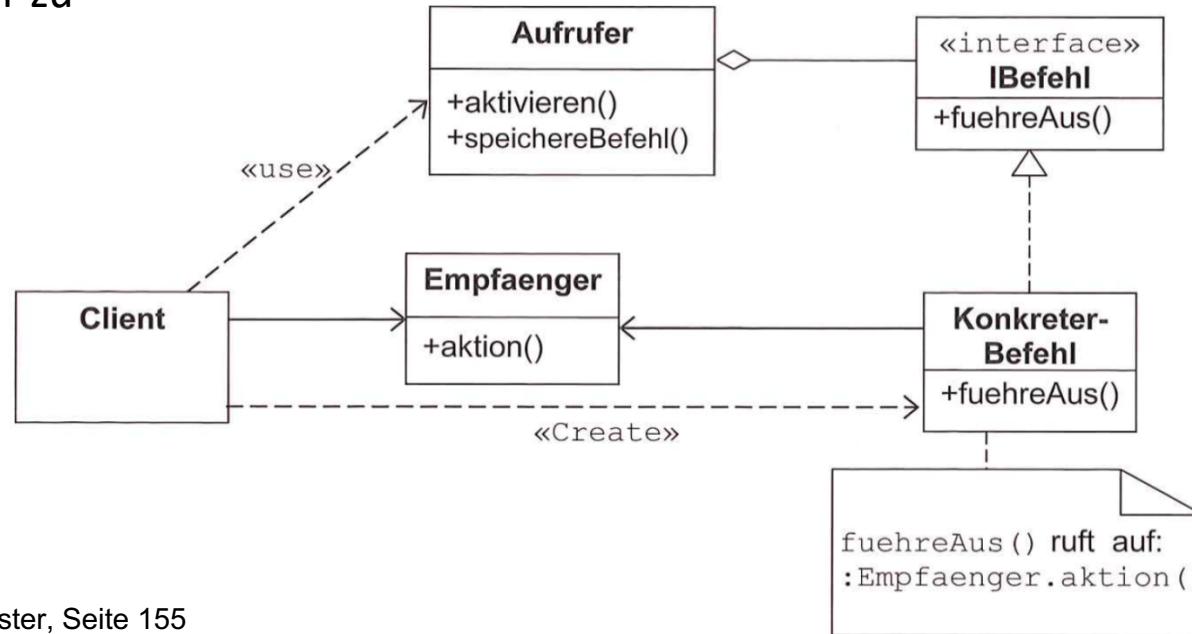
Quelle: Entwurfsmuster, Seite 158

Command (2)

- **Anwendbarkeit:**

- Der Aufrufer soll nicht von der Ausprägung eines konkreten Befehls abhängen.
- Er soll nur die Abstraktion eines Befehls kennen und somit sollen Details eines konkreten Befehls dem Aufrufer verborgen bleiben
- Ein Client bzw. eine Anwendung erstellt konkrete Befehle und weist ihnen einem Empfänger zu

- **Struktur:**



Quelle: Entwurfsmuster, Seite 155

Command (3)

- **Vor- und Nachteile:**

- + Das Erzeugen eines Befehls und seine Ausführung sind zeitlich entkoppelt
- + Asynchrones Aufrufen
- + Befehle dynamisch zur Laufzeit austauschen
- + Befehlsobjekte können wiederverwendet werden
- Verschachtelung der Aufrufe
- Implementierung in verteilten Systemen komplex

- **Implementierung:**

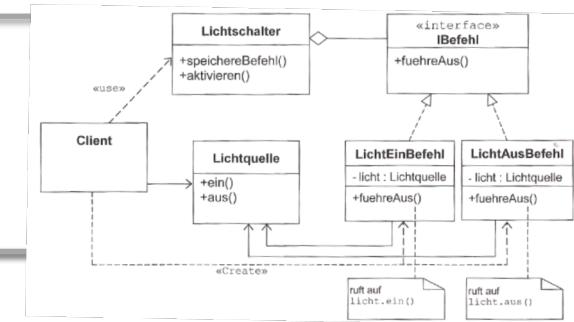
- Logging – Vereinfachte Protokollierung
- Zeitversetzte Ausführung – Einreichung der Befehle in eine Warteschlange
- Undo-/Redo-Funktionalität – Rückgängigmachen von Befehlen
- Zusammengesetzte Befehle - Makros
- Transaktionen – Rollback
- Recovery nach Systemcrash – persistente Speicherung der Befehle

Command – Programmbeispiel (1)

```
// Datei: IBefehl.java
public interface IBefehl
{
    public void fuehreAus();
}
```

```
public class LichtQuelle
{
    public void ein()
    {
        System.out.println ("Licht wurde eingeschaltet!");
    }

    public void aus()
    {
        System.out.println ("Licht wurde ausgeschaltet!");
    }
}
```



Abstrakte Schnittstelle für konkrete Befehlsklasse

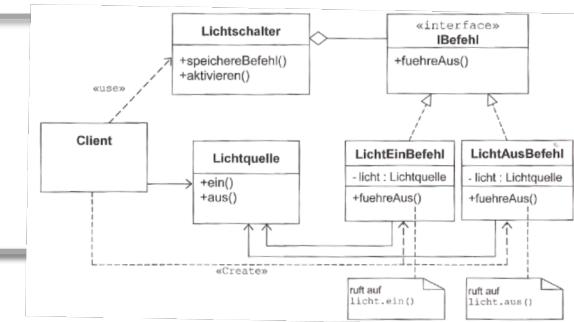
Empfängerklasse für konkrete Befehle. Methoden ein() aus(), um eine Lichtquelle zu schalten

Command – Programmbeispiel (2)

```
public class LichtSchalter
{
    private IBefehl befehl; // Referenz auf IBefehl

    public void speichereBefehl (IBefehl befehl) •
    {
        this.befehl = befehl;
    }

    public void aktivieren()
    {
        befehl.fuehreAus();
    }
}
```



Aufruferklasse

- Die Referenz zeigt dann auf den konkreten Befehl → Ausführung

```
public class LichtEinBefehl implements IBefehl
{
    private LichtQuelle licht;

    public LichtEinBefehl (LichtQuelle licht)
    {
        this.licht = licht;
    }

    public void fuehreAus() //Definition der Methode fuehreAus()
    {
        licht.ein();
    }
}
```

Konkrete Befehlsklassen

- Referenz auf Lichtquelle und Methode führe aus()
- Analog LichtAusBefehl

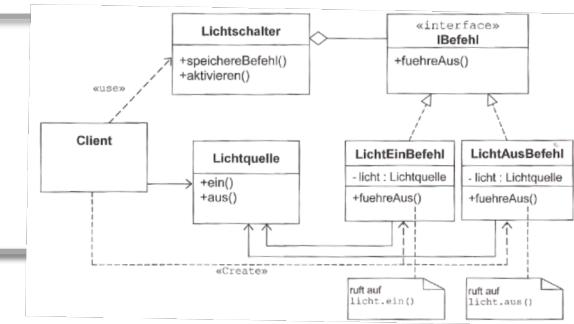
Command – Programmbeispiel (3)

```
// Datei: Client.java
public class Client
{
    public static void main (String[] args)
    {
        LichtSchalter schalter = new LichtSchalter();
        LichtQuelle licht = new LichtQuelle();

        IBefehl lichtAn = new LichtEinBefehl (licht);
        IBefehl lichtAus = new LichtAusBefehl (licht);

        schalter.speichereBefehl (lichtAn);
        schalter.aktivieren();

        schalter.speichereBefehl (lichtAus);
        schalter.aktivieren();
    }
}
```



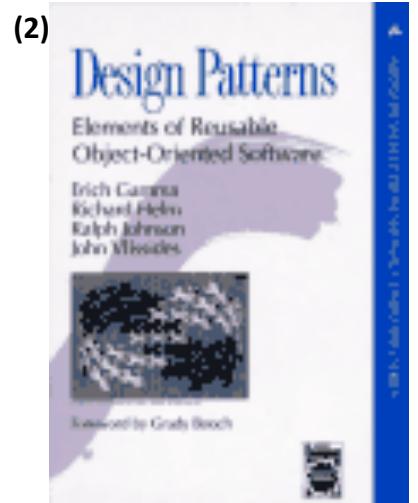
- Client erstellt konkreten Befehl und wird in einem Objekt der Klasse Lichtschalter gespeichert
- Durch aktivieren() des Schalters wird der Befehl ausgelöst

Überblick - Entwurfsmuster

- Struktur
 - Adapter^(1,2)
 - Brücke (Bridge) ^(1,2)
 - Dekorierer (Decorator) ^(1,2)
 - Fassade (Facade) ^(1,2)
 - Fliegengewicht (Flyweight) ⁽²⁾
 - Kompositum (Composite) ^(1,2)
 - Proxy^(1,2)
- Verhalten
 - Befehl (Command) ^(1,2)
 - Beobachter (Observer) ^(1,2)
 - Besucher (Visitor) ^(1,2)
 - Interpreter (Interpreter) ⁽²⁾
 - Iterator^(1,2)
 - Memento⁽²⁾
 - Rolle⁽¹⁾
 - Schablonenmethode (Template Method) ^(1,2)
 - Strategie (Strategy) ^(1,2)
 - Vermittler (Mediator) ^(1,2)
 - Zustand (State) ^(1,2)
 - Zuständigkeitskette (Chain of Responsibility) ⁽²⁾
- Erzeugung
 - Abstrakte Fabrik (Abstract Factory) ^(1,2)
 - Erbauer (Builder) ⁽²⁾
 - Fabrikmethode (Factory Method) ^(1,2)
 - Objektpool⁽¹⁾
 - Prototyp (Prototype) ^(1,2)
 - Singleton^(1,2)

Legende:

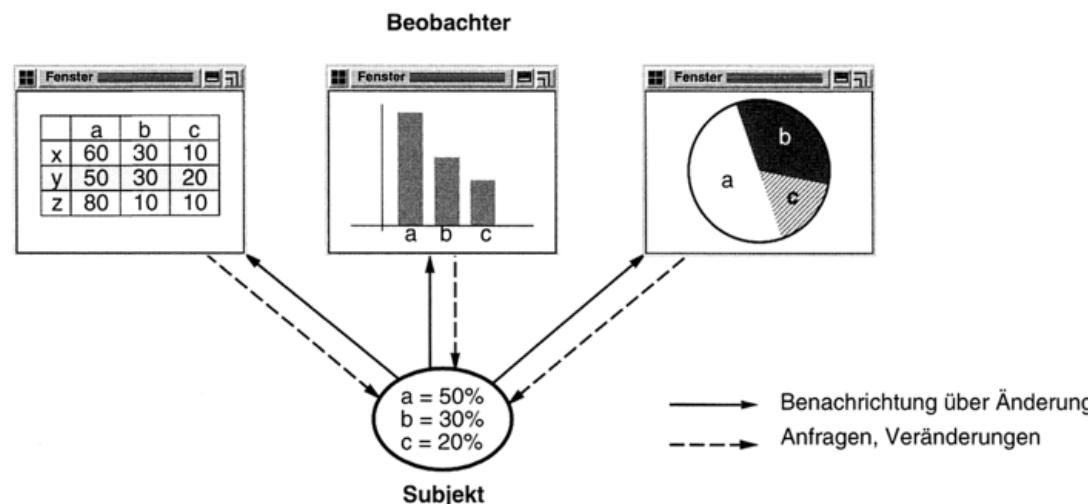
Fett = wird in der Vorlesung behandelt



Observer (1)

- **Name:** Observer (dt. Beobachter)
- **Zweck:** Definiere eine 1-zu-n-Abhangigkeit zwischen Objekten, so dass die Änderung des Zustands eines Objekts dazu fuhrt, dass alle abhangigen Objekte benachrichtigt und automatisch aktualisiert werden.

Motivation:



Quelle: Entwurfsmuster, Seite 288

Grundidee des Beobachter-Musters

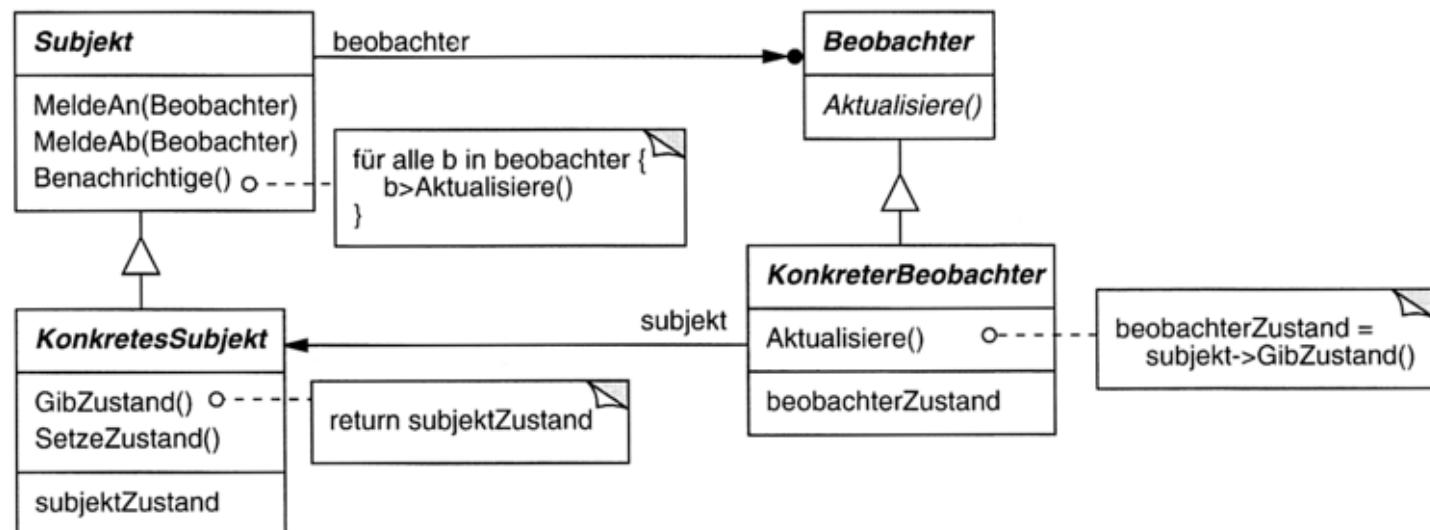
- Grundidee des Beobachter-Musters
 - Das Modell benachrichtigt die Präsentation, wenn sich etwas am Datenbestand verändert hat (sich also z.B. der Wasserstand verändert hat oder sich die Aktienkurse verändert haben).
 - Die Präsentation hat die Rolle des Beobachters (*Observer*), also desjenigen, der den Zustand eines anderen Objektes beobachtet. Das Modell hat die Rolle desjenigen, der beobachtet wird (*Observable*).
- Grundsätzlicher Ablauf beim Beobachter-Muster
 - Alle Beobachter registrieren sich beim Beobachteten.
 - Wird der Datenbestand (also das Modell) verändert, so werden alle registrierten Beobachter nacheinander durch einen Methodenaufruf beim jeweiligen Beobachter darüber benachrichtigt.
 - Der Beobachter kann sich dann über Methodenaufrufe beim Modell die aktuellen Daten besorgen.

Observer (1)

• Anwendbarkeit:

- Wenn die Änderung eines Objekts die Änderung anderer Objekte verlangt und Sie nicht wissen, wie viele Objekte geändert werden müssen.
- Wenn ein Objekt in der Lage sein sollte, andere Objekte zu benachrichtigen, ohne Annahmen darüber treffen zu dürfen, wer diese anderen Objekte sind.

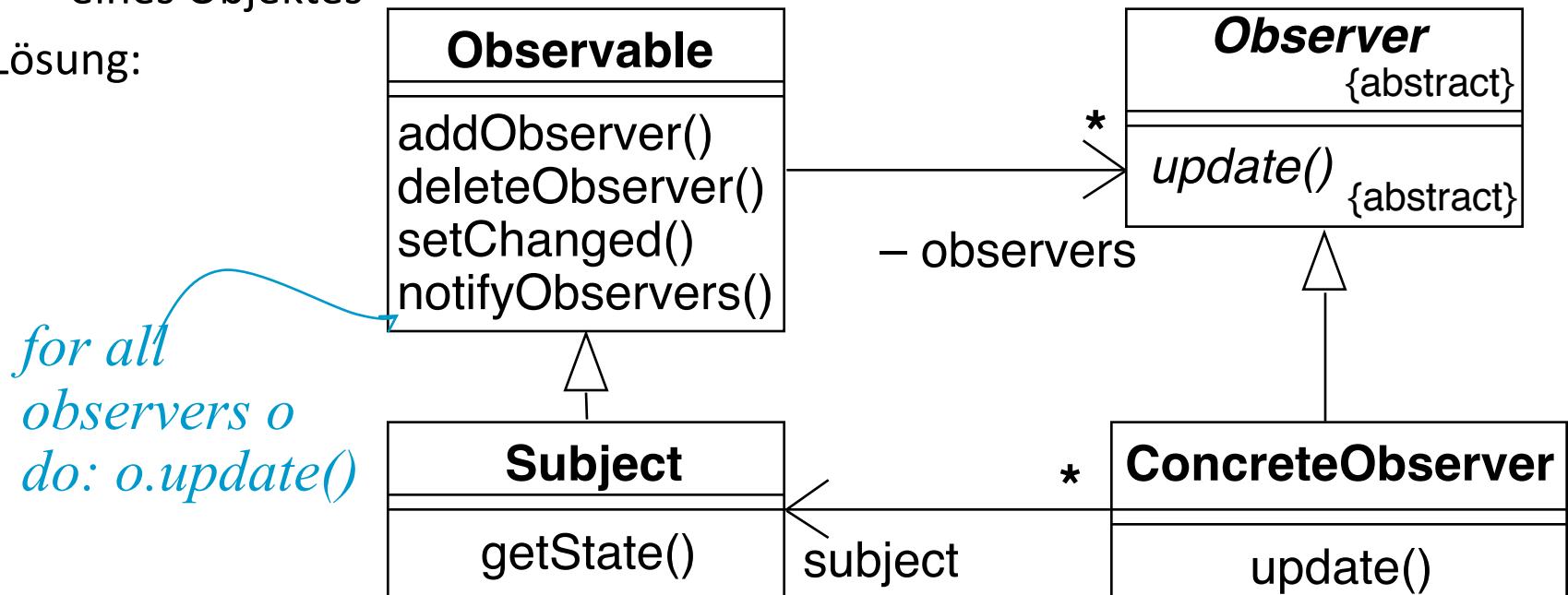
• Struktur:



Quelle: Entwurfsmuster, Seite 289

Observer (2)

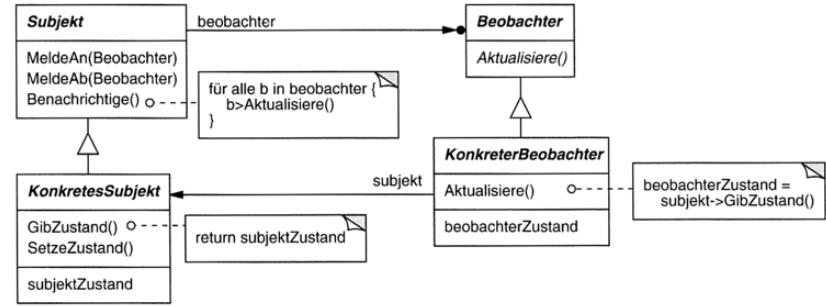
- Name: **Observer** (dt.: Beobachter)
- Problem:
 - Mehrere Objekte sind interessiert an bestimmten Zustandsänderungen eines Objektes
- Lösung:



Observer – Programmbeispiel (1)

```
// Datei: IBeobachter.java
public interface IBeobachter
{
    public void aktualisieren (IBeobachtbar b);
}
```

```
// Datei: IBeobachtbar.java
public interface IBeobachtbar
{
    public void anmelden (IBeobachter beobachter);
    public void abmelden (IBeobachter beobachter);
    public String gibZustand();
}
```



- Beobachter (Observer)

- Subject (Observable)

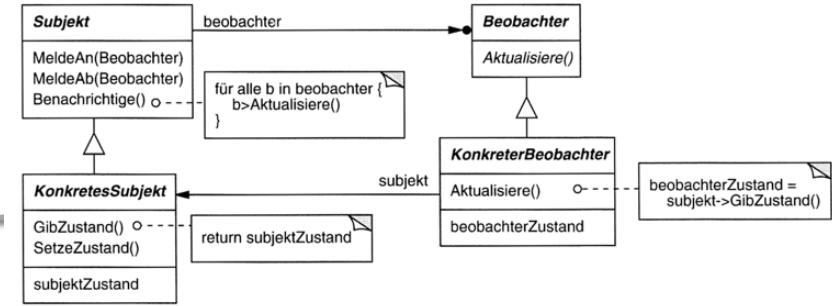
Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 169ff

Observer – Programmbeispiel (2)

```
// Datei: Abonnent.java
public class Abonnent implements IBeobachter
{
    private String name;

    public Abonnent (String name)
    {
        this.name = name;
    }

    public void aktualisieren (IBeobachtbar b)
    {
        System.out.println ("Neue Nachricht fuer " + name + ".");
        System.out.println ("Nachricht: " + b.gibZustand());
    }
}
```



- Konkreter Beobachter
- Implementierung von IBeobachter

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 169ff

Observer – Programmbeispiel (3)

```
// Datei: Newsletter.java
import java.util.Vector;

public class Newsletter implements IBeobachtbar
{
    private Vector<IBeobachter> abonnenten = new Vector<IBeobachter>();

    private String nachricht;

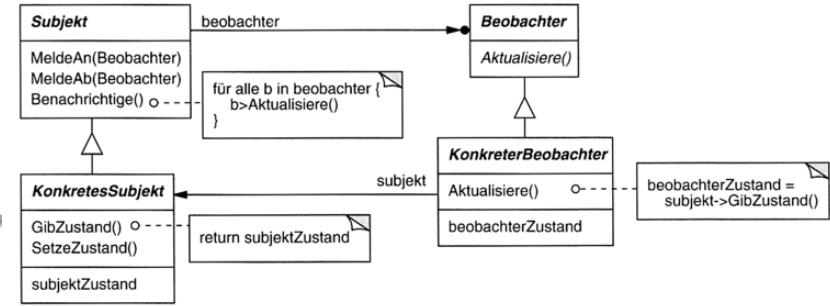
    public void aendereNachricht (String neueNachricht)
    {
        nachricht = neueNachricht;
        benachrichtigen();
    }

    public void abmelden (IBeobachter beobachter)
    {
        abonnenten.remove (beobachter);
    }

    public void anmelden (IBeobachter beobachter)
    {
        abonnenten.add (beobachter);
    }

    private void benachrichtigen()
    {
        for (IBeobachter beobachter : abonnenten)
        {
            beobachter.aktualisieren (this);
        }
    }

    public String gibZustand()
    {
        return nachricht;
    }
}
```



- Konkretes Subjekt
- Implementierung von IBeobachtbar

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 169ff

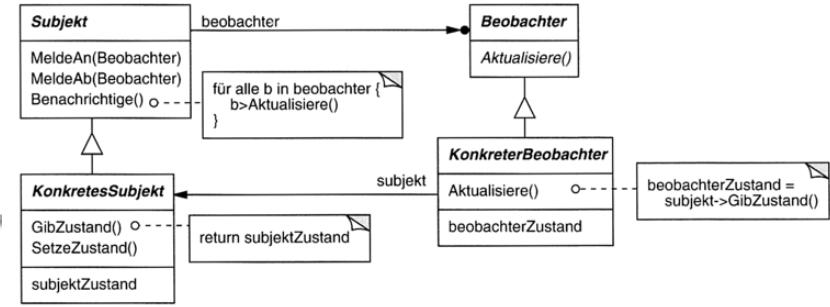
Observer – Programmbeispiel (4)

```
// Datei: TestBeobachter.java
public class TestBeobachter
{
    public static void main (String[] args)
    {
        Newsletter newsletter = new Newsletter();
        Abonnent andreas = new Abonnent ("Andreas");
        Abonnent birgit = new Abonnent ("Birgit");

        newsletter.anmelden (andreas);
        newsletter.anmelden (birgit);
        newsletter.aendereNachricht ("Neuigkeit 1");
        System.out.println();

        newsletter.abmelden (andreas);
        newsletter.aendereNachricht ("Neuigkeit 2");

        newsletter.abmelden (birgit);
        newsletter.aendereNachricht ("Neuigkeit 3");
    }
}
```



Die Ausgabe des Programms ist:

Neue Nachricht fuer Andreas.
Nachricht: Neuigkeit 1
Neue Nachricht fuer Birgit.
Nachricht: Neuigkeit 1

Neue Nachricht fuer Birgit.
Nachricht: Neuigkeit 2

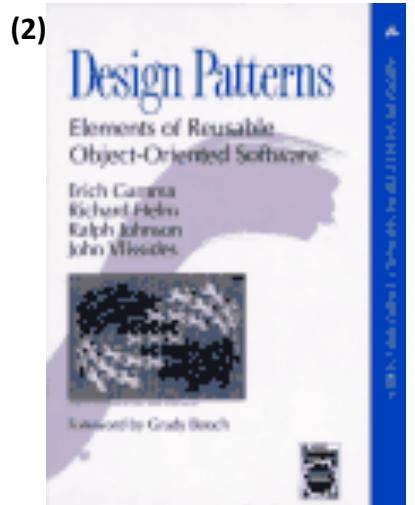
Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 169ff

Überblick - Entwurfsmuster

- Struktur
 - Adapter^(1,2)
 - Brücke (Bridge) ^(1,2)
 - Dekorierer (Decorator) ^(1,2)
 - Fassade (Facade) ^(1,2)
 - Fliegengewicht (Flyweight) ⁽²⁾
 - Kompositum (Composite) ^(1,2)
 - Proxy^(1,2)
- Verhalten
 - Befehl (Command) ^(1,2)
 - Beobachter (Observer) ^(1,2)
 - Besucher (Visitor) ^(1,2)
 - Interpreter (Interpreter) ⁽²⁾
 - Iterator^(1,2)
 - Memento⁽²⁾
 - Rolle⁽¹⁾
 - Schablonenmethode (Template Method) ^(1,2)
 - Strategie (Strategy) ^(1,2)
 - Vermittler (Mediator) ^(1,2)
 - Zustand (State) ^(1,2)
 - Zuständigkeitskette (Chain of Responsibility) ⁽²⁾
- Erzeugung
 - Abstrakte Fabrik (Abstract Factory) ^(1,2)
 - Erbauer (Builder) ⁽²⁾
 - Fabrikmethode (Factory Method) ^(1,2)
 - Objektpool⁽¹⁾
 - Prototyp (Prototype) ^(1,2)
 - Singleton^(1,2)

Legende:

Fett = wird in der Vorlesung behandelt



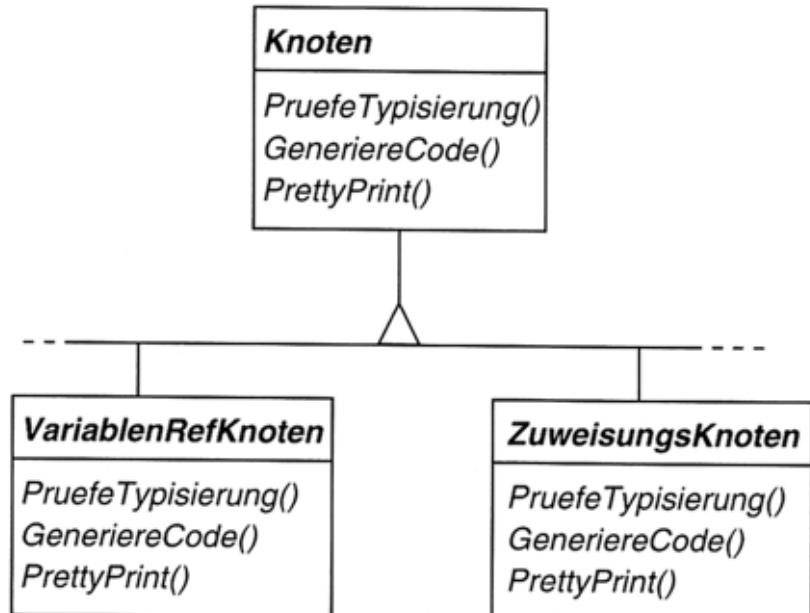
Visitor (1)

- **Name:** Visitor (dt. Besucher)
- **Zweck:** Führe auf jedem Objekt einer Struktur eine bestimmte Operation aus. Definiere neue Operationen, ohne die Klassen der Struktur zu ändern.

Motivation:

Ohne Visitor:

- Knoten implementieren alle Operationen
- beim Hinzufügen einer neuen Operation müssen alle Knoten geändert werden
- Code für gleichartige Operationen ist über viele Klassen verstreut

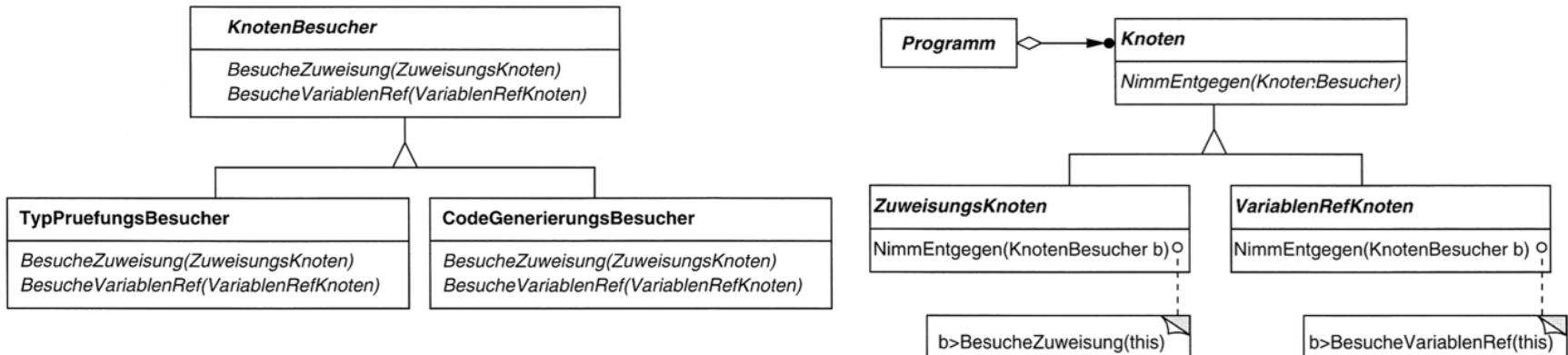


Quelle: Entwurfsmuster, Seite 302

Visitor (2)

Mit Visitor:

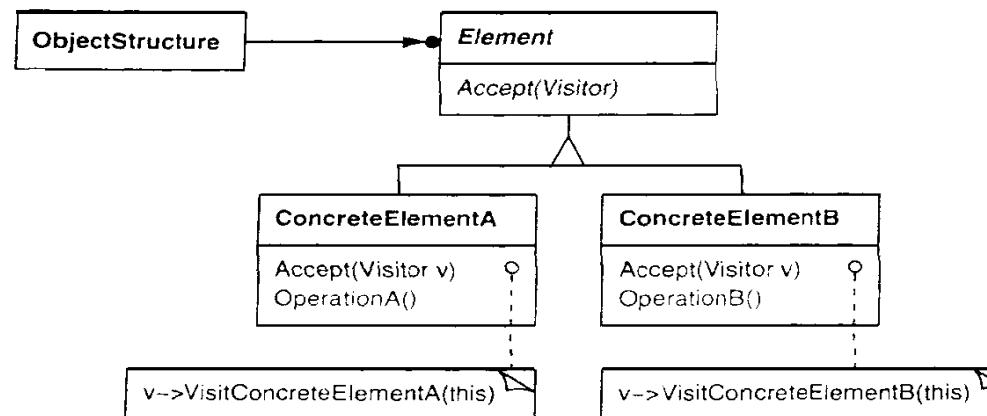
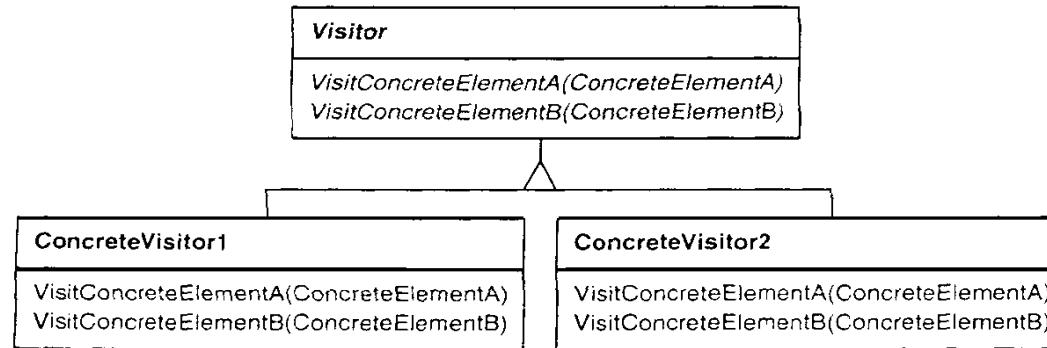
- Eine neue Visitor-Klasse für jede Operation als Unterklasse von NodeVisitor, enthält den Code für alle Typen von Knoten
- Knoten haben einen „Haken“ zum Andocken für den Visitor



Quelle: Entwurfsmuster, Seite 303

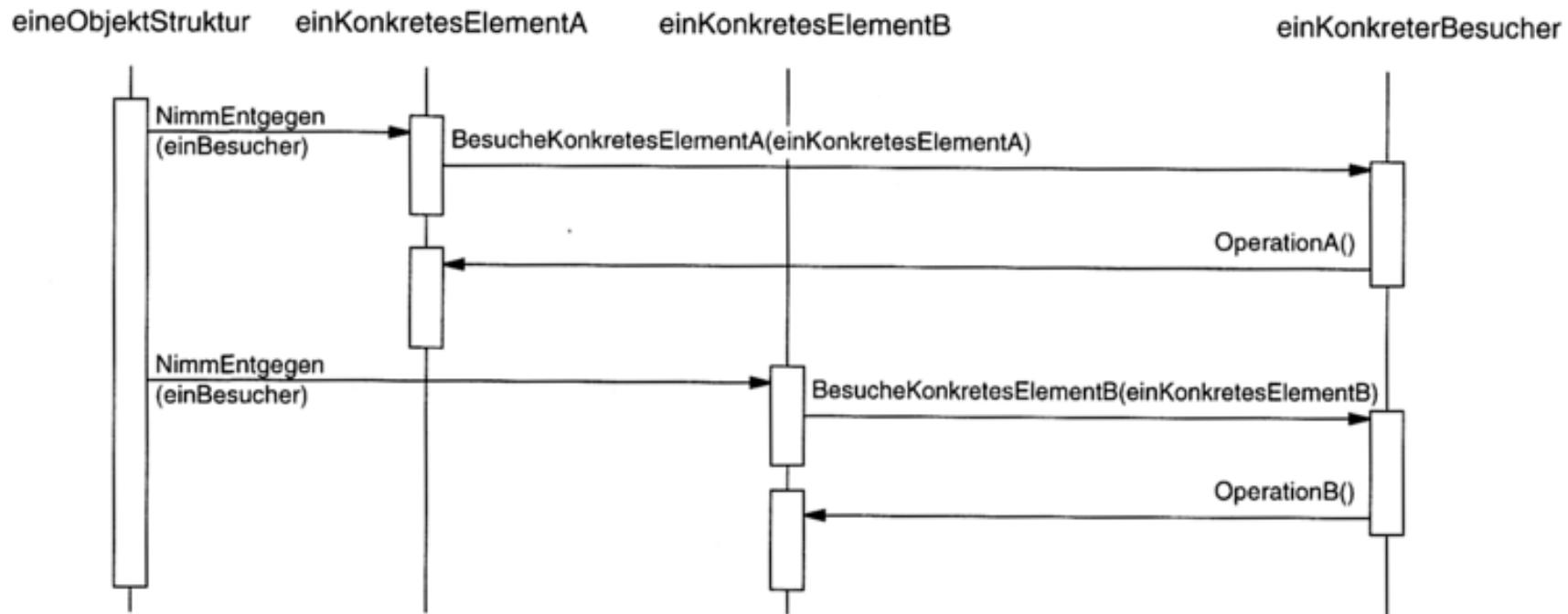
Visitor (3)

- **Struktur:**



Visitor (4)

- **Interaktion:**



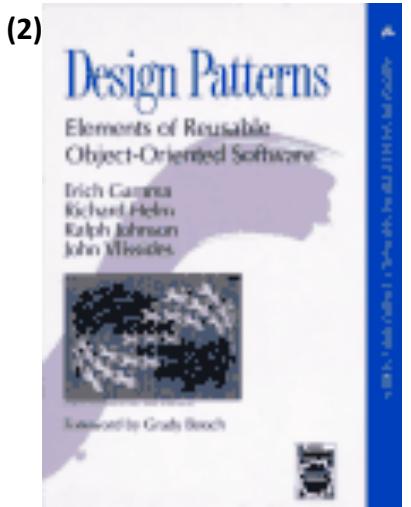
Quelle: Entwurfsmuster, Seite 306

Überblick - Entwurfsmuster

- Struktur
 - Adapter^(1,2)
 - Brücke (Bridge) ^(1,2)
 - Dekorierer (Decorator) ^(1,2)
 - Fassade (Facade) ^(1,2)
 - Fliegengewicht (Flyweight) ⁽²⁾
 - Kompositum (Composite) ^(1,2)
 - Proxy^(1,2)
- Verhalten
 - Befehl (Command) ^(1,2)
 - Beobachter (Observer) ^(1,2)
 - Besucher (Visitor) ^(1,2)
 - Interpreter (Interpreter) ⁽²⁾
 - Iterator^(1,2)
 - Memento⁽²⁾
 - Rolle⁽¹⁾
 - Schablonenmethode (Template Method) ^(1,2)
 - Strategie (Strategy) ^(1,2)
 - Vermittler (Mediator) ^(1,2)
 - Zustand (State) ^(1,2)
 - Zuständigkeitskette (Chain of Responsibility) ⁽²⁾
- Erzeugung
 - Abstrakte Fabrik (Abstract Factory) ^(1,2)
 - Erbauer (Builder) ⁽²⁾
 - Fabrikmethode (Factory Method) ^(1,2)
 - Objektpool⁽¹⁾
 - Prototyp (Prototype) ^(1,2)
 - Singleton^(1,2)

Legende:

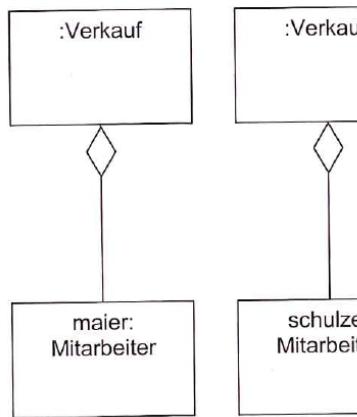
Fett = wird in der Vorlesung behandelt



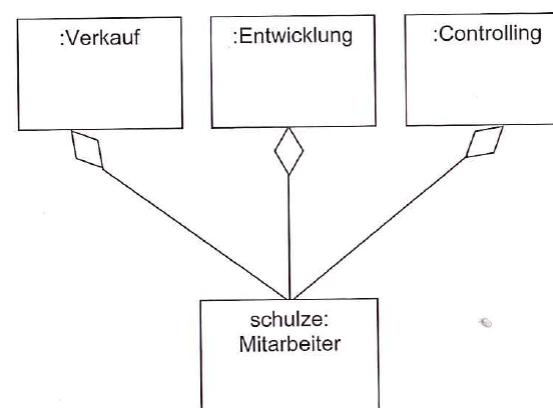
Role (1)

- **Name:** Role / Role Object Pattern (dt. Rolle)
- **Zweck:** Objekte einer Software stehen nicht alleine da. Sie wirken vielmehr mit anderen Objekten zusammen, indem sie dabei Rollen annehmen.

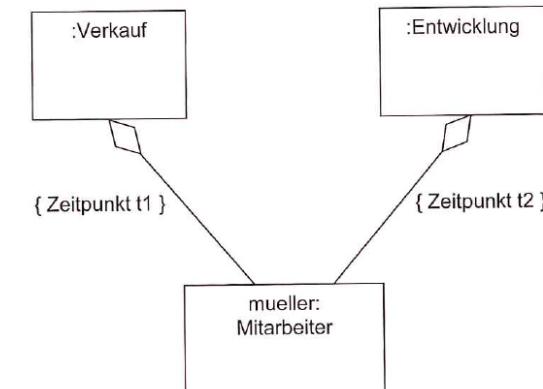
Motivation:



Mitarbeiter haben gleiche Rollen



...mehrere Rollen



Ein Mitarbeiter hat zu verschiedenen Zeitpunkten t1 und t2 jeweils eine andere Rolle

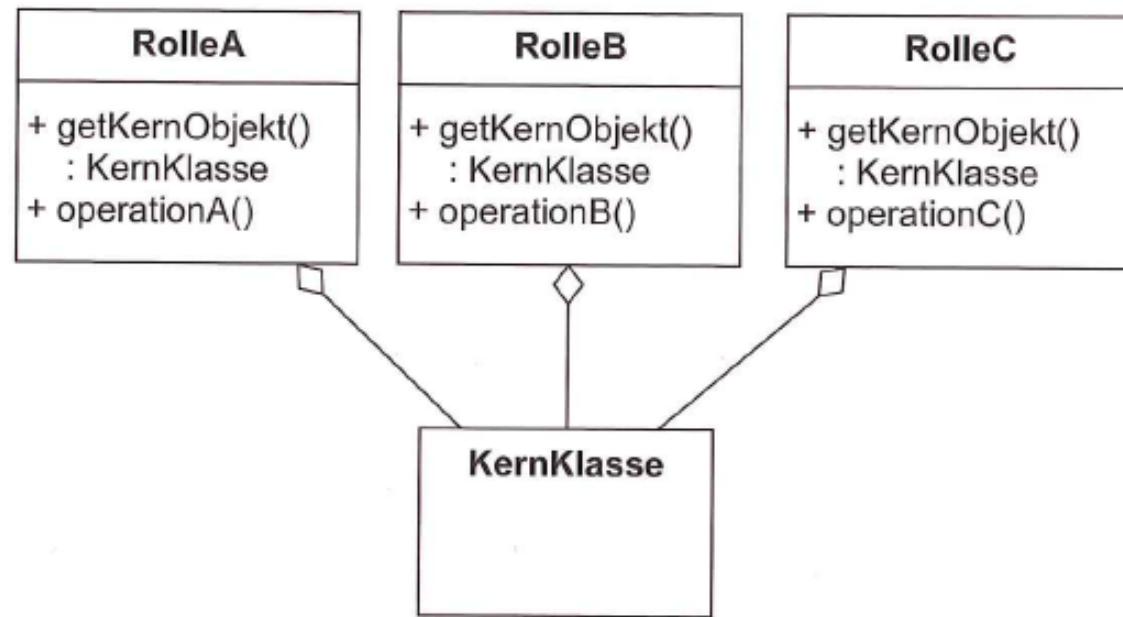
Quelle: Entwurfsmuster, Seite 206

Role (2)

- **Anwendbarkeit:**

- Ein Objekt soll dynamisch seine Rollen wechseln und mehrere Rollen gleichzeitig annehmen können.
- Mehrere Objekte können die gleiche Rolle haben.

- **Struktur: (hier: mehrere Rollen für eine Kernklasse)**



Quelle: Entwurfsmuster, Seite 207

Role (3)

- **Vor- und Nachteile:**
 - + Rollen können zur Laufzeit dynamisch angenommen und abgelegt werden
 - + Ein Objekt kann mehrere Rollen gleichzeitig spielen
 - Aufwendig festzustellen, welche Rolle ein Kernobjekt einnimmt.
 - Es werden deutlich mehr Objekte erzeugt und diese müssen auch verwaltet werden

Role – Programmbeispiel (1)

```
// Datei: Entwicklung.java
public class Entwicklung
{
    private Mitarbeiter mitarbeiter;
    private String projekt;

    public Entwicklung (String projekt)
    {
        this.projekt = projekt;
    }

    public String getAbteilungsBezeichnung()
    {
        return "Entwicklung";
    }

    public void printProjekt()
    {
        System.out.println (mitarbeiter.getName() + " "
            + "arbeitet momentan an: " + projekt);
    }

    public Mitarbeiter getMitarbeiter()
    {
        return mitarbeiter;
    }

    public void setMitarbeiter (Mitarbeiter mitarbeiter)
    {
        this.mitarbeiter = mitarbeiter;
    }
}
```

- Rollenklasse
- Analog: Verkauf

- Kernklasse

```
// Datei: Mitarbeiter.java
public class Mitarbeiter
{
    private String name;

    public Mitarbeiter (String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

Role – Programmbeispiel (2)

```
public class TestRolle
{
    public static void main (String[] args)
    {
        // Kernobjekte werden erzeugt
        Mitarbeiter maier = new Mitarbeiter ("Maier");
        Mitarbeiter schulze = new Mitarbeiter ("Schulze");

        // Rollenobjekte der Abteilung Entwicklung
        Entwicklung entwicklung1 = new Entwicklung ("Produkt 2.0");
        Entwicklung entwicklung2 = new Entwicklung (
            "Produkt Addon 1.0");

        // Zuordnung der Rollen zu den Kernobjekten
        entwicklung1.setMitarbeiter (schulze);
        entwicklung2.setMitarbeiter (maier);

        // Ausgabe der Projekte
        System.out.println ("\nAktuelle Projekte der Mitarbeiter: ");
        entwicklung1.printProjekt();
        entwicklung2.printProjekt();

        // Ein Kernobjekt spielt eine weitere Rolle
        System.out.println ("\nSchulze erhaelt "
            + "zusaetzliche Aufgaben im Verkauf:");
        Verkauf verkauf1 = new Verkauf (15000);
        verkauf1.setMitarbeiter (schulze);

        // Ausgabe der Umsatzzahlen
        System.out.println ("\nNur Schulze ist in der "
            + "Abteilung Verkauf:");
        verkauf1.printUmsatz();
    }
}
```

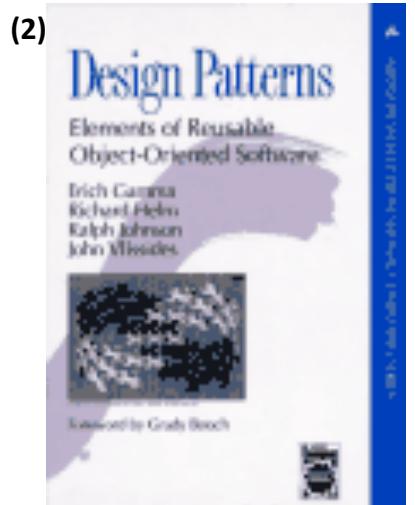
- Maier und Schulze Instanzen von Mitarbeiter
- Schulze hat zwei Rollen gleichzeitig

Überblick - Entwurfsmuster

- Struktur
 - Adapter^(1,2)
 - Brücke (Bridge) ^(1,2)
 - Dekorierer (Decorator) ^(1,2)
 - Fassade (Facade) ^(1,2)
 - Fliegengewicht (Flyweight) ⁽²⁾
 - Kompositum (Composite) ^(1,2)
 - Proxy^(1,2)
- Verhalten
 - Befehl (Command) ^(1,2)
 - Beobachter (Observer) ^(1,2)
 - Besucher (Visitor) ^(1,2)
 - Interpreter (Interpreter) ⁽²⁾
 - Iterator^(1,2)
 - Memento⁽²⁾
 - Rolle⁽¹⁾
 - Schablonenmethode (Template Method) ^(1,2)
 - Strategie (Strategy) ^(1,2)
 - Vermittler (Mediator) ^(1,2)
 - Zustand (State) ^(1,2)
 - Zuständigkeitskette (Chain of Responsibility) ⁽²⁾
- Erzeugung
 - Abstrakte Fabrik (Abstract Factory) ^(1,2)
 - Erbauer (Builder) ⁽²⁾
 - Fabrikmethode (Factory Method) ^(1,2)
 - Objektpool⁽¹⁾
 - Prototyp (Prototype) ^(1,2)
 - Singleton^(1,2)

Legende:

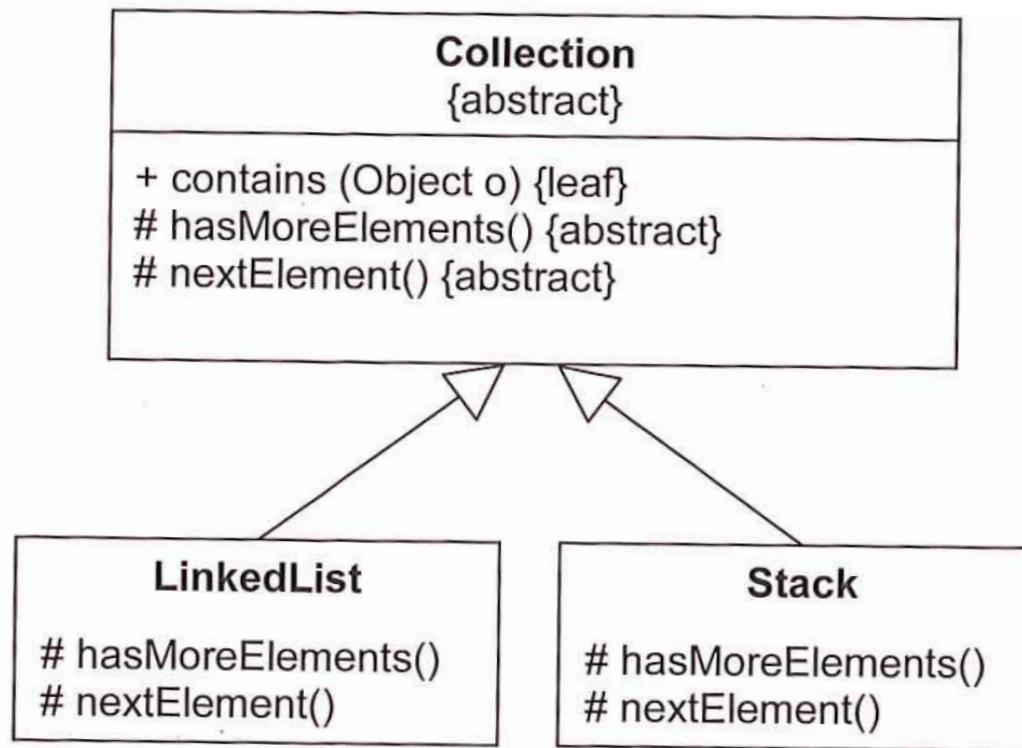
Fett = wird in der Vorlesung behandelt



Template Method (1)

- **Name:** Template Method (dt. Schablonenmethode)
- **Zweck:** Die Struktur eines Algorithmus soll in der Basisklasse festgelegt werden, Einzelheiten in einer Unterklassie

- **Motivation:**



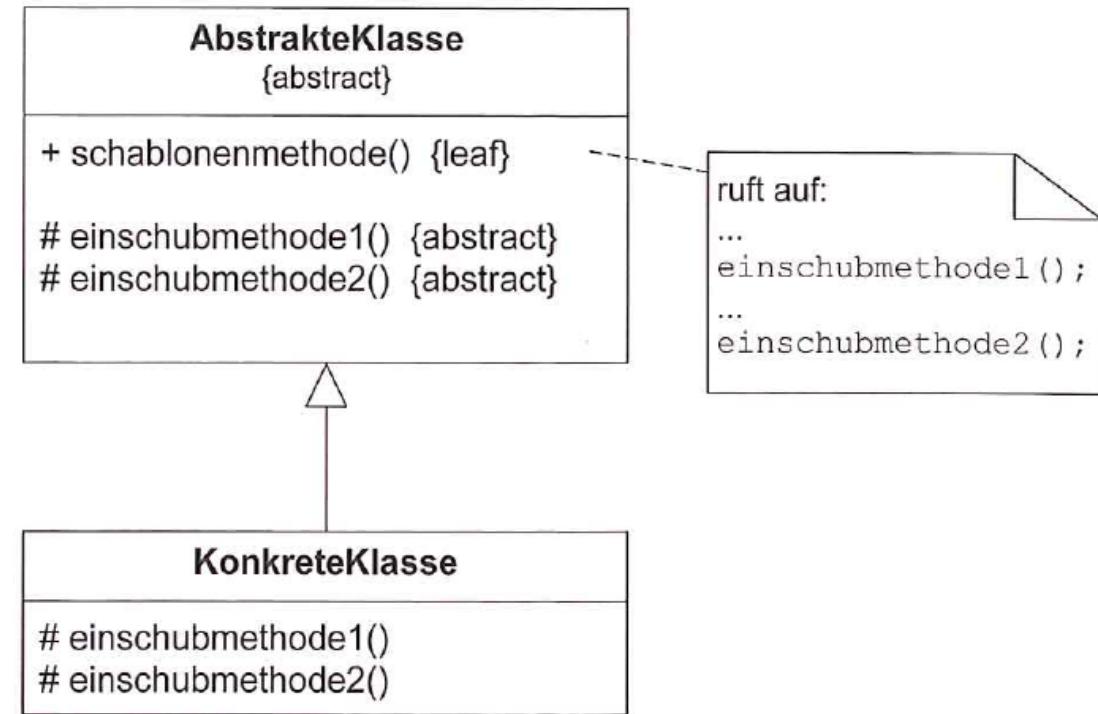
Quelle: Entwurfsmuster, Seite 150

Template Method (2)

- **Anwendbarkeit:**

- Schablonenmethode legt das Gerüst eines Algorithmus fest und benutzt dabei abstrakte Einschubmethoden
- Einschubmethoden werden in Unterklassen i.d.R. unterschiedlich implementiert

- **Struktur:**



Quelle: Entwurfsmuster, Seite 147

Template Method (3)

- **Vor- und Nachteile:**

- + Basisklasse ist nicht abhängig von ihren Unterklassen
- + Abstraktion von Verhalten
- + Wiederverwendung

Template Method – Programmbeispiel (1)

```
// Datei: Urlaubskarte.java
abstract public class Urlaubskarte // abstrakte Klasse
{
    final public void karteSchreiben() // Schablonenmethode
    {
        textSchreiben();
        zusatzSchreiben(); // abstrakte Methode wird augerufen
    }

    private void textSchreiben() // normalen Text ausgeben
    {
        System.out.println (
            "Ich bin gut an meinem Urlaubsziel angekommen.");
        System.out.println (
            "Das Essen schmeckt gut und die Gegend gefaellt mir.");
    }

    abstract protected void zusatzSchreiben(); // Einschubmethode
}
```

Template Method – Programmbeispiel (2)

```
// Datei: UrlaubskarteAnFreunde.java
public class UrlaubskarteAnFreunde extends Urlaubskarte
{
    public void zusatzSchreiben() // Eine abgeleitete Klasse, in der
                                    // die abstrakte Methode
                                    // überschrieben wird.
    {
        System.out.println ("Ich treibe viel Sport.");
    }
}

// Datei: UrlaubskarteAnFirma.java
public class UrlaubskarteAnFirma extends Urlaubskarte
{
    public void zusatzSchreiben() // Eine andere abgeleitete Klasse,
                                    // in der die abstrakte Methode
                                    // überschrieben wird.
    {
        System.out.println ("Ich freue mich wieder auf die Arbeit.");
    }
}
```

Template Method – Programmbeispiel (3)

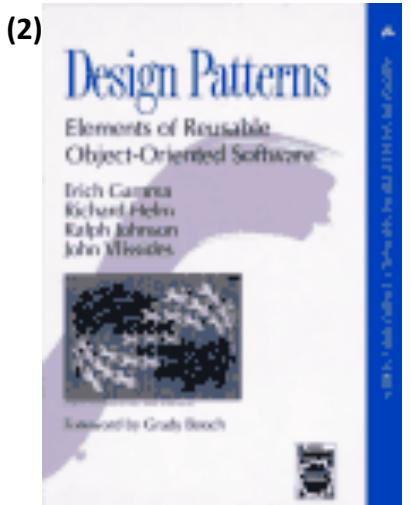
```
// Datei: Test.java // Testklasse
public class Test
{
    public static void main (String args[])
    {
        System.out.println ("Karte an die Freunde:");
        UrlaubskarteAnFreunde karteFreunde =
            new UrlaubskarteAnFreunde ();
        karteFreunde.karteSchreiben(); // dabei erfolgt der Aufruf einer
                                       // überschreibenden Methode
        System.out.println ();
        System.out.println ("Karte an die Firma:");
        UrlaubskarteAnFirma karteFirma = new UrlaubskarteAnFirma ();
        karteFirma.karteSchreiben(); // dabei erfolgt der Aufruf einer
                                       // überschreibenden Methode
    }
}
```

Überblick - Entwurfsmuster

- Struktur
 - Adapter^(1,2)
 - Brücke (Bridge) ^(1,2)
 - Dekorierer (Decorator) ^(1,2)
 - Fassade (Facade) ^(1,2)
 - Fliegengewicht (Flyweight) ⁽²⁾
 - Kompositum (Composite) ^(1,2)
 - Proxy^(1,2)
- Verhalten
 - Befehl (Command) ^(1,2)
 - Beobachter (Observer) ^(1,2)
 - Besucher (Visitor) ^(1,2)
 - Interpreter (Interpreter) ⁽²⁾
 - Iterator^(1,2)
 - Memento⁽²⁾
 - Rolle⁽¹⁾
 - Schablonenmethode (Template Method) ^(1,2)
 - Strategie (Strategy) ^(1,2)
 - Vermittler (Mediator) ^(1,2)
 - Zustand (State) ^(1,2)
 - Zuständigkeitskette (Chain of Responsibility) ⁽²⁾
- Erzeugung
 - Abstrakte Fabrik (Abstract Factory) ^(1,2)
 - Erbauer (Builder) ⁽²⁾
 - Fabrikmethode (Factory Method) ^(1,2)
 - Objektpool⁽¹⁾
 - Prototyp (Prototype) ^(1,2)
 - Singleton^(1,2)

Legende:

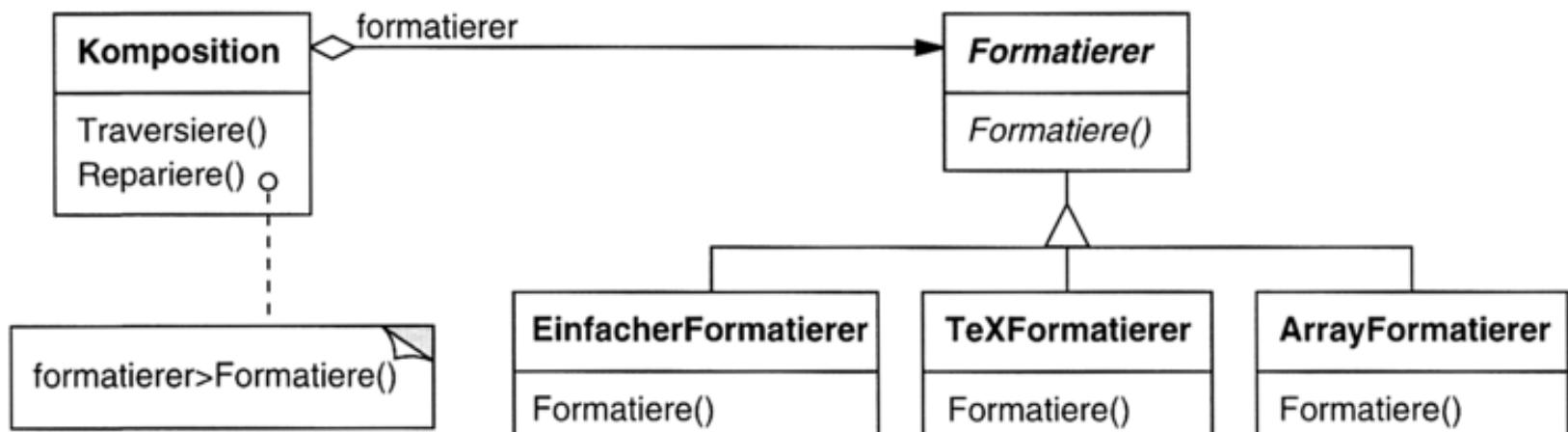
Fett = wird in der Vorlesung behandelt



Strategy (1)

- **Name:** Strategy (dt. Strategie)
- **Zweck:** Definiere eine Familie von Algorithmen, kapsele jeden einzelnen und mache sie austauschbar.

Motivation:



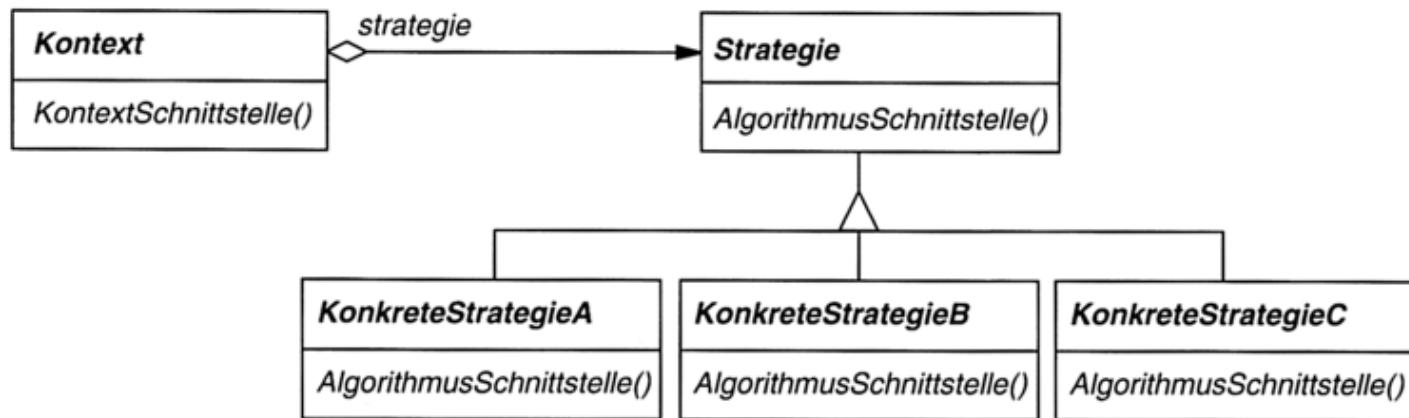
Quelle: Entwurfsmuster, Seite 374

Strategy (2)

- **Anwendbarkeit:**

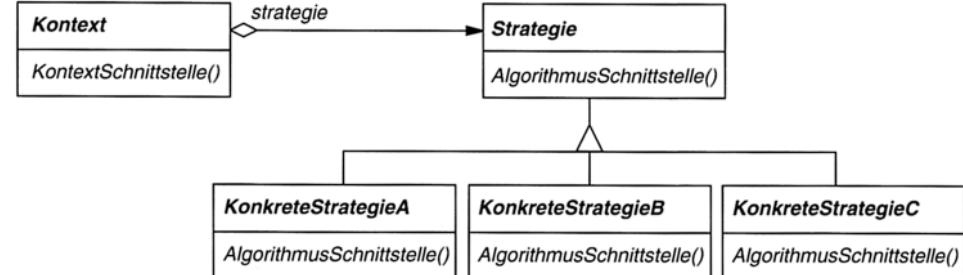
- Wenn sich viele verwandte Klassen nur in ihrem Verhalten unterscheiden.
- Wenn Sie unterschiedliche Varianten eines Algorithmus benötigen.
- Wenn ein Algorithmus Daten verwendet, die den Klienten nicht bekannt sein sollen.

- **Struktur:**



Quelle: Entwurfsmuster, Seite 375

Strategy – Programmbeispiel (1)



```
// Datei: IDatumsFormat.java
public interface IDatumsFormat
{
    public void datumAusgeben (int tag, int monat, int jahr);
}
```

```
// Datei: Datum.java
public class Datum
{
    private IDatumsFormat format = null;
    private int tag, monat, jahr = 0;

    public Datum (int tag, int monat, int jahr)
    {
        this.tag = tag;
        this.monat = monat;
        this.jahr = jahr;
    }

    public void setzeFormat (IDatumsFormat format)
    {
        this.format = format;
    }

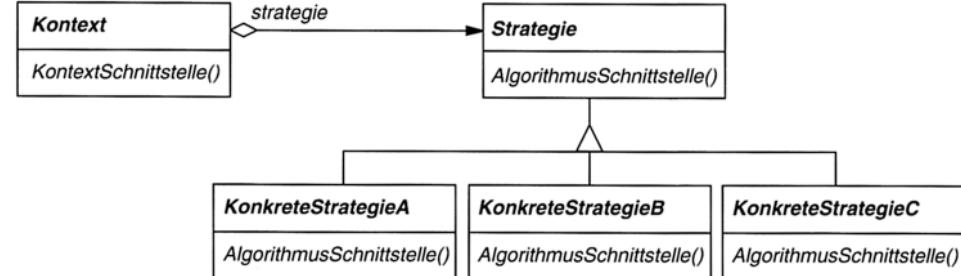
    public void ausgeben()
    {
        // ruft die Methode datumAusgeben() des Objekts auf,
        // auf das format zeigt.
        format.datumAusgeben (tag, monat, jahr);
    }
}
```

- **Strategie**

- **Kontext**

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 176ff

Strategy – Programmbeispiel (2)



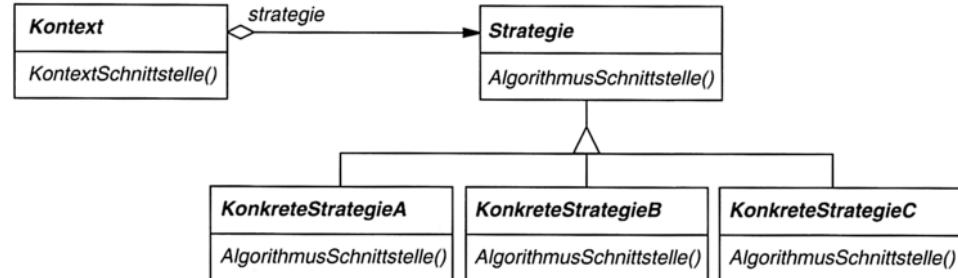
```
// Datei: EuropaeischesFormat.java
public class EuropaeischesFormat implements IDatumsFormat
{
    public void datumAusgeben (int tag, int monat, int jahr)
    {
        System.out.println ("Europaeisches Format: "
            + (tag > 9 ? tag : "0" + tag)
            + "."
            + (monat > 9 ? monat : "0" + monat)
            + "."
            + jahr);
    }
}

// Datei: AmerikanischesFormat.java
public class AmerikanischesFormat implements IDatumsFormat
{
    public void datumAusgeben (int tag, int monat, int jahr)
    {
        System.out.println ("Amerikanisches Format: "
            + (monat > 9 ? monat : "0" + monat)
            + "/"
            + (tag > 9 ? tag : "0" + tag)
            + "/"
            + jahr);
    }
}
```

- Konkrete Strategien

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 176ff

Strategy – Programmbeispiel (3)



```
// Datei: TestStrategie.java
public class TestStrategie
{
    public static void main (String[] args)
    {
        Datum datum = new Datum (21, 9, 1985);

        datum.setzeFormat (new EuropaeischesFormat ());
        datum.ausgeben ();

        datum.setzeFormat (new AmerikanischesFormat ());
        datum.ausgeben ();
    }
}
```

Hier das Protokoll des Programmablaufs:

Europaeisches Format: 21.09.1985
Amerikanisches Format: 09/21/1985

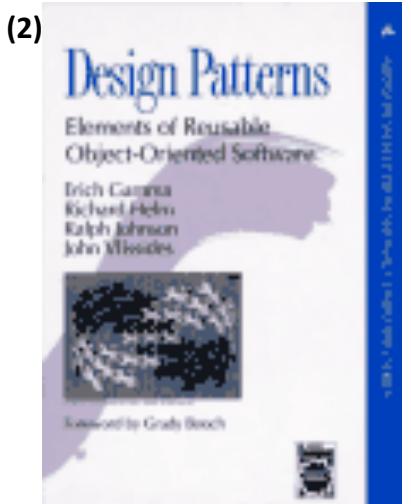
Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 176ff

Überblick - Entwurfsmuster

- Struktur
 - Adapter^(1,2)
 - Brücke (Bridge) ^(1,2)
 - Dekorierer (Decorator) ^(1,2)
 - Fassade (Facade) ^(1,2)
 - Fliegengewicht (Flyweight) ⁽²⁾
 - Kompositum (Composite) ^(1,2)
 - Proxy^(1,2)
- Verhalten
 - Befehl (Command) ^(1,2)
 - Beobachter (Observer) ^(1,2)
 - Besucher (Visitor) ^(1,2)
 - Interpreter (Interpreter) ⁽²⁾
 - Iterator^(1,2)
 - Memento⁽²⁾
 - Rolle⁽¹⁾
 - Schablonenmethode (Template Method) ^(1,2)
 - Strategie (Strategy) ^(1,2)
 - Vermittler (Mediator) ^(1,2)
 - Zustand (State) ^(1,2)
 - Zuständigkeitskette (Chain of Responsibility) ⁽²⁾
- Erzeugung
 - Abstrakte Fabrik (Abstract Factory) ^(1,2)
 - Erbauer (Builder) ⁽²⁾
 - Fabrikmethode (Factory Method) ^(1,2)
 - Objektpool⁽¹⁾
 - Prototyp (Prototype) ^(1,2)
 - Singleton^(1,2)

Legende:

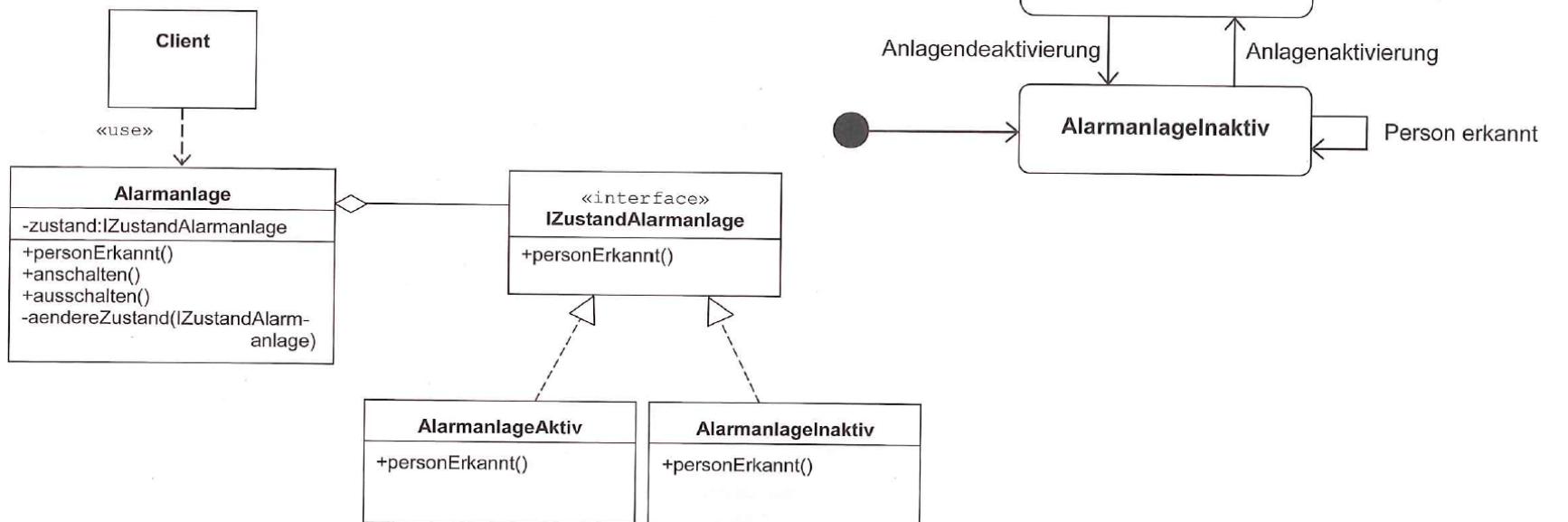
Fett = wird in der Vorlesung behandelt



State (1)

- **Name:** State (dt. Zustand)
- **Zweck:** Zustandsbehaftete Aufgaben sollen so gelöst werden, dass jeder Fall einer klassischen Bedienungsanleitung auf ein Objekt einer eigenen Klasse abgebildet wird

Motivation:



Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 197

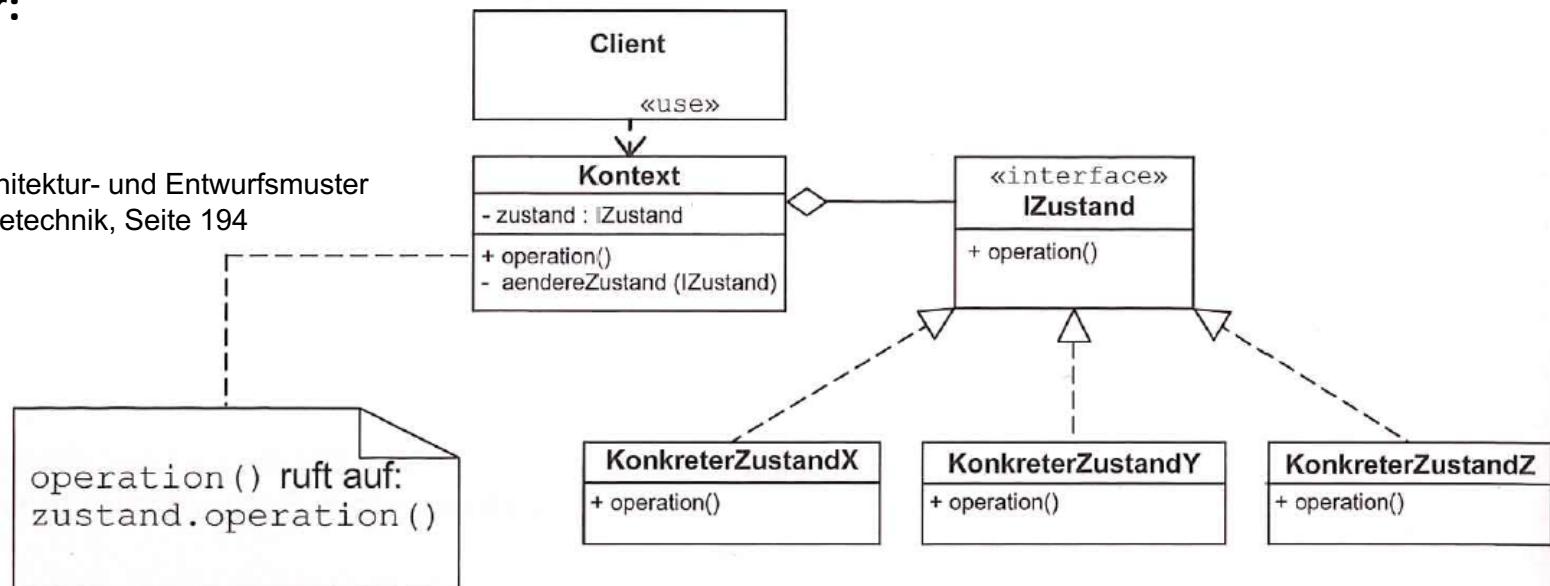
State (2)

- **Anwendbarkeit:**

- Der Kontext zur Kompilierzeit hängt nicht von den konkreten Zustandsklassen ab.
- Die einzelnen Zustände werden durch eigene Klassen gekapselt, die sich von einer Abstrakten Basisklasse oder Interface ableiten.
- Zustandsabhängiges Kontextobjekt referenziert den aktuellen Zustand und führt Zustandsänderungen durch.

- **Struktur:**

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 194



State (3)

- **Vor- und Nachteile:**
 - + Gesamtes Verhalten in einer Klasse → Übersichtlichkeit
 - + Erweiterbar
 - Hoher Implementierungsaufwand / Viele Objekte zur Laufzeit
- **Implementierung:**
 - Bedienelemente mit Zuständen einer grafischen Benutzeroberfläche
 - Zustände von parallelen Einheiten (Prozesssteuerung)

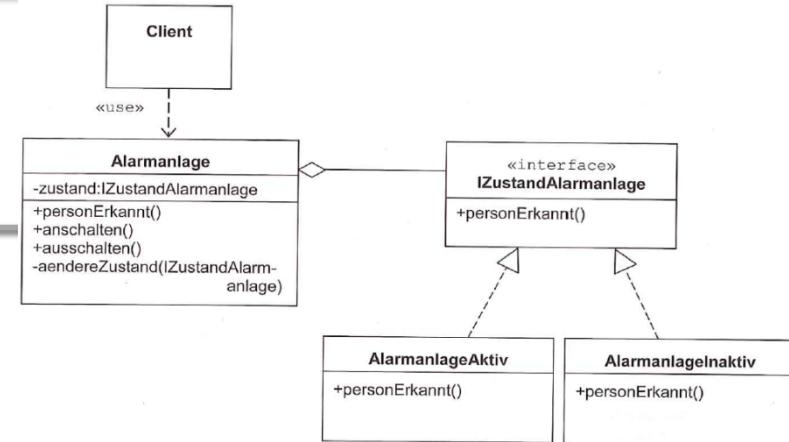
Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 200f

State – Programmbeispiel (1)

```
// Datei: IZustandAlarmanlage.java
// Schnittstelle fuer alle Zustandsklassen
public interface IZustandAlarmanlage
{
    public void personErkannt();
}
```

```
public class AlarmanlageAktiv implements IZustandAlarmanlage
{
    // Sofern eine Person erkannt wurde, ein akustisches Signal
    // ausgeben.
    public void personErkannt()
    {
        System.out.println ("RING RING");
    }
}
```

```
public class AlarmanlageInaktiv implements IZustandAlarmanlage
{
    // Sofern eine Person erkannt wurde, KEIN akustisches Signal
    // ausgeben, da dies im normalen Geschaeftsbetrieb nur stoerend
    // waere.
    public void personErkannt()
    {
        System.out.println ("Ruhig bleiben.");
    }
}
```



Methoden aller Zustandsklassen

Konkrete Zustände

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 197ff

State – Programmbeispiel (2)

```
public class Alarmanlage
{
    IZustandAlarmanlage aktiv = new AlarmanlageAktiv();
    IZustandAlarmanlage inaktiv = new AlarmanlageInaktiv();
    IZustandAlarmanlage zustand = null;

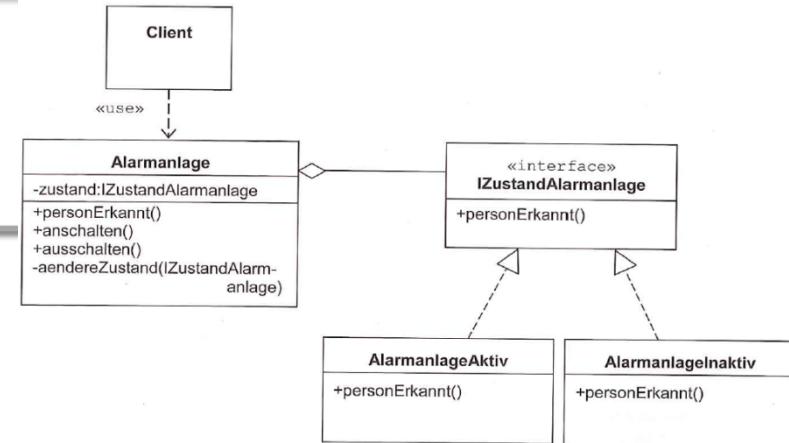
    public Alarmanlage()
    {
        zustand = inaktiv; // Startzustand
    }

    public void anschalten()
    {
        aendereZustand(aktiv);
    }

    public void ausschalten()
    {
        aendereZustand(inaktiv);
    }

    public void personErkannt()
    {
        zustand.personErkannt();
    }

    private void aendereZustand (IZustandAlarmanlage neuerZustand)
    {
        zustand = neuerZustand;
    }
}
```



Kontextklasse

Definiert, wie die Alarmanlage von außen aufgerufen werden kann

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 197ff

State – Programmbeispiel (3)

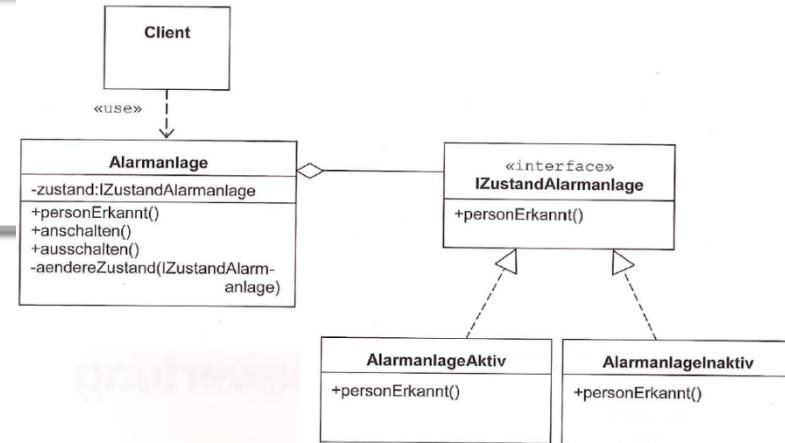
```
public class Client
{
    public static void main (String[] args)
    {
        Alarmanlage a = new Alarmanlage();

        System.out.println ("Anlage deaktivieren...");
        System.out.println ("Bei Kundentrieb stoert das.");
        a.ausschalten();

        System.out.println ("Person erkannt.");
        a.personErkannt();

        System.out.println ("Feierabend.");
        System.out.println ("Aktivierung der Alarmanlage.");
        a.anschalten();

        System.out.println ("Person erkannt.");
        a.personErkannt();
    }
}
```



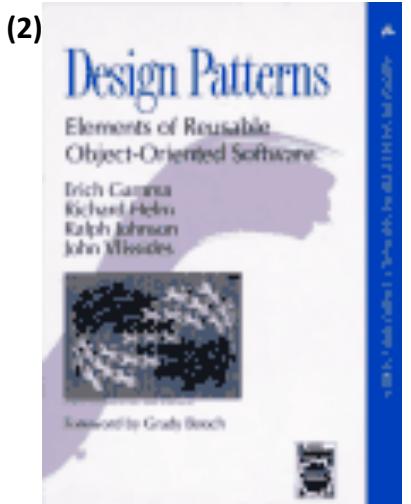
Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 197ff

Überblick - Entwurfsmuster

- Struktur
 - Adapter^(1,2)
 - Brücke (Bridge) ^(1,2)
 - Dekorierer (Decorator) ^(1,2)
 - Fassade (Facade) ^(1,2)
 - Fliegengewicht (Flyweight) ⁽²⁾
 - Kompositum (Composite) ^(1,2)
 - Proxy^(1,2)
- Verhalten
 - Befehl (Command) ^(1,2)
 - Beobachter (Observer) ^(1,2)
 - Besucher (Visitor) ^(1,2)
 - Interpreter (Interpreter) ⁽²⁾
 - Iterator^(1,2)
 - Memento⁽²⁾
 - Rolle⁽¹⁾
 - Schablonenmethode (Template Method) ^(1,2)
 - Strategie (Strategy) ^(1,2)
 - Vermittler (Mediator) ^(1,2)
 - Zustand (State) ^(1,2)
 - Zuständigkeitskette (Chain of Responsibility) ⁽²⁾
- Erzeugung
 - Abstrakte Fabrik (Abstract Factory) ^(1,2)
 - Erbauer (Builder) ⁽²⁾
 - Fabrikmethode (Factory Method) ^(1,2)
 - Objektpool⁽¹⁾
 - Prototyp (Prototype) ^(1,2)
 - Singleton^(1,2)

Legende:

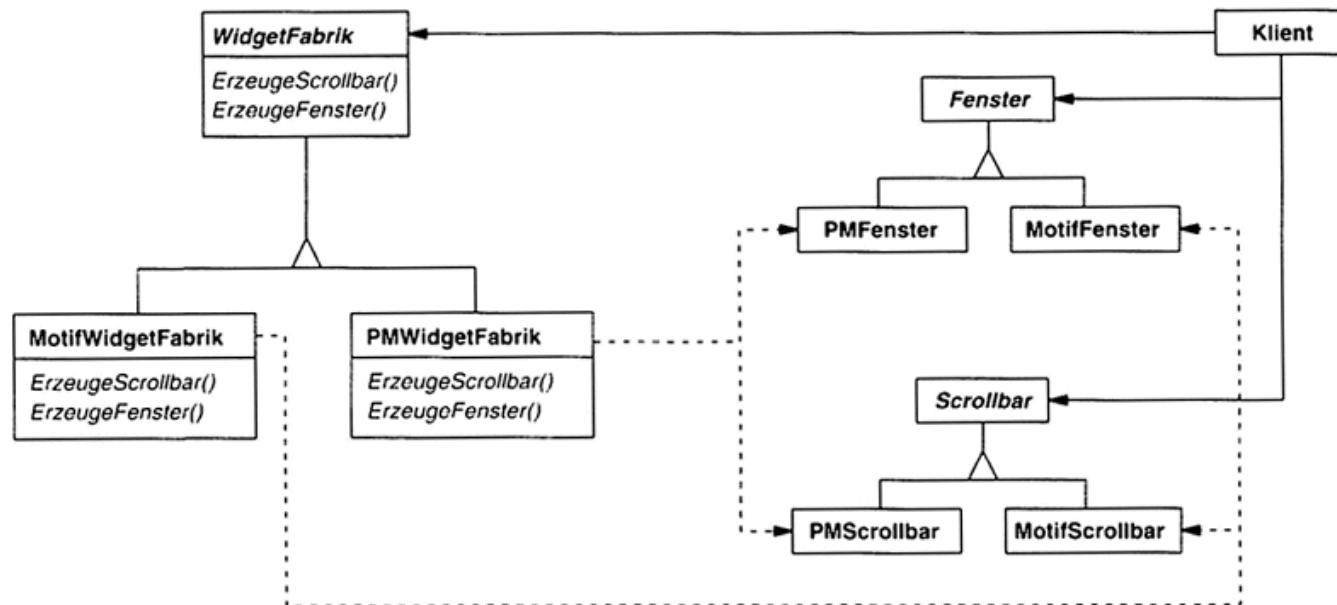
Fett = wird in der Vorlesung behandelt



Abstract Factory (1)

- **Name:** Abstract Factory (dt. Abstrakte Fabrik)
- **Zweck:** Biete eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängigen Objekte, ohne ihre konkreten Klassen zu benennen.

Motivation:



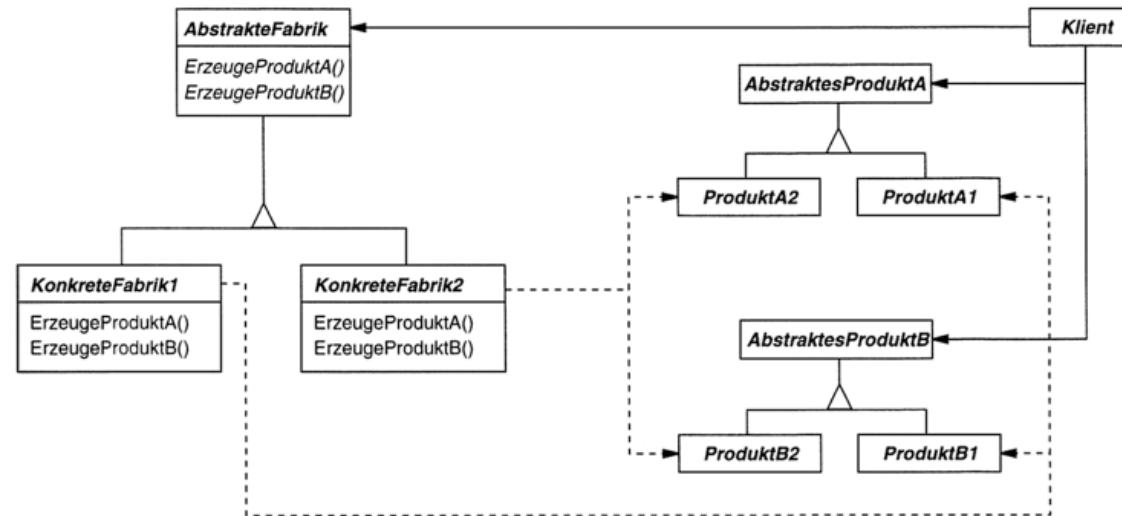
Quelle: Entwurfsmuster, Seite 107

Abstract Factory (2)

- **Anwendbarkeit:**

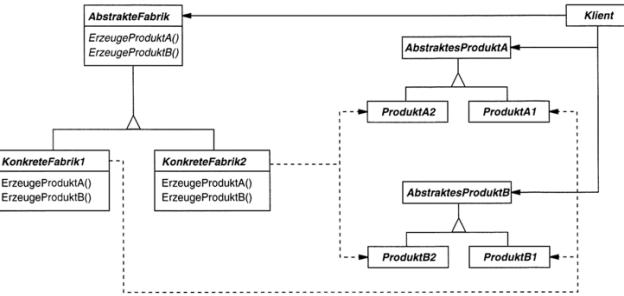
- Wenn ein System unabhängig davon sein soll wie seine Produkte erzeugt werden.
- Wenn ein System mit einer von mehreren Produktfamilien konfiguriert werden soll.
- Wenn Sie eine Klassenbibliothek von Produkten anbieten möchten, von denen Sie nur die Schnittstellen, nicht aber ihre Implementierung offenlegen möchten.

- **Struktur:**



Quelle: Entwurfsmuster, Seite 109

Abstract Factory – Programmbeispiel (1)



```
// Datei: IMutter.java
public interface IMutter
{
    public void print();
}
```

Abstrakte Produkte

```
// Datei: ISchraube.java
public interface ISchraube
{
    public void print();
}
```

```
// Datei: MutterM6.java
public class MutterM6 implements IMutter
{
    public void print()
    {
        System.out.println ("Mutter mit M6 Gewinde.");
    }
}

// Datei: MutterM10.java
public class MutterM10 implements IMutter
{
    public void print()
    {
        System.out.println ("Mutter mit M10 Gewinde.");
    }
}
```

```
// Datei: SchraubeM6.java
public class SchraubeM6 implements ISchraube
{
    public void print()
    {
        System.out.println ("Schraube mit M6 Gewinde.");
    }
}

// Datei: SchraubeM10.java
public class SchraubeM10 implements ISchraube
{
```

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 255ff

Abstract Factory – Programmbeispiel (2)

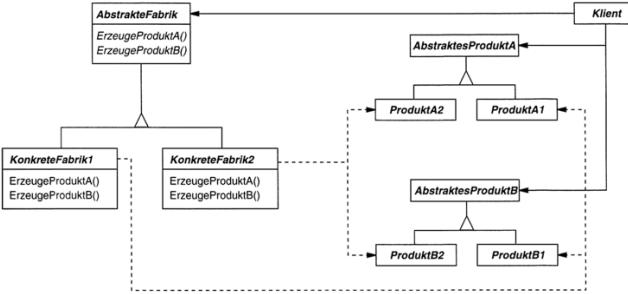
```
public interface IAbstrakteFabrik
{
    public ISchraube erzeugeSchraube();
    public IMutter erzeugeMutter();
}
```

```
// Datei: KonkreteFabrikM6.java
public class KonkreteFabrikM6 implements IAbstrakteFabrik
{
    public ISchraube erzeugeSchraube()
    {
        return new SchraubeM6();
    }

    public IMutter erzeugeMutter()
    {
        return new MutterM6();
    }
}

// Datei: KonkreteFabrikM10.java
public class KonkreteFabrikM10 implements IAbstrakteFabrik
{
    public ISchraube erzeugeSchraube()
    {
        return new SchraubeM10();
    }

    public IMutter erzeugeMutter()
    {
        return new MutterM10();
    }
}
```



Abstrakte Fabrik hat zwei Methoden.
Jede erzeugt eine Instanz eines anderen
Produktes (IMutter und ISchrauben)

Konkrete Fabriken. Erzeugen jeweils
Schrauben und Muttern mit gleichem
Gewinde

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 255ff

Abstract Factory – Programmbeispiel (3)

Prinzipiell kann eine Schachtel beliebige Kombinationen von Schrauben und Muttern in unterschiedlichen Größen enthalten.

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 255ff

```
// Datei: Schachtel.java
public class Schachtel
{
    private int anzahl;
    private int anzahlSchrauben = 0;
    private int anzahlMuttern = 0;
    private ISchraube[] schrauben;
    private IMutter[] muttern;

    Schachtel (int groesse)
    {
        anzahl = groesse;
        schrauben = new ISchraube[anzahl];
        muttern = new IMutter[anzahl];
    }

    int anzahl()
    {
        return anzahl;
    }

    public void legeSchraubeHinein(ISchraube schraube)
    {
        if (anzahlSchrauben == anzahl) // wenn schon voll
        {
            return;
        }
        else
        {
            anzahlSchrauben++;
            schrauben[anzahlSchrauben-1] = schraube;
        }
    }

    public void legeMutterHinein(IMutter mutter)
    {
        if (anzahlMuttern == anzahl) // wenn schon voll
        {
            return;
        }
        else
        {
            anzahlMuttern++;
            muttern[anzahlMuttern-1] = mutter;
        }
    }

    public void zeigeInhalt()
    {
        int i;

        for (i = 0; i < anzahlSchrauben; i++)
        {
            schrauben[i].print();
        }
        for (i = 0; i < anzahlMuttern; i++)
        {
            muttern[i].print();
        }
    }
}
```

Abstract Factory – Programmbeispiel (4)

```
public class ProduktionsMaschine
{
    private IAbstrakteFabrik fabrik = null;

    public void setFabrik (IAbstrakteFabrik fabrikRef)
    {
        this.fabrik = fabrikRef;
    }

    public void fuelleSchachtel(Schachtel schachtel)
    {
        int i;

        for (i = 0; i <= schachtel.anzahl(); i++)
        {
            schachtel.legeSchraubeHinein(fabrik.erzeugeSchraube());
            schachtel.legeMutterHinein(fabrik.erzeugeMutter());
        }
    }
}

// Datei: TestProduktion.java
public class TestProduktion
{
    static ProduktionsMaschine maschine = new ProduktionsMaschine();
    static Schachtel sch1 = new Schachtel(5);
    static Schachtel sch2 = new Schachtel(3);

    public static void main (String[] args)
    {
        maschine.setFabrik (new KonkreteFabrikM6());
        maschine.fuelleSchachtel(sch1);
        sch1.zeigeInhalt();

        System.out.println();

        maschine.setFabrik (new KonkreteFabrikM10());
        maschine.fuelleSchachtel(sch2);
        sch2.zeigeInhalt();
    }
}
```

- Referenz auf abstrakte Fabrik
- Keine Bezüge auf konkrete Produkte

- Objekt von ProduktionsMaschine wird erzeugt
- Objekt wird konkrete Fabrik zugeordnet
- Schachtel wird gefüllt
- Inhalt zeigen

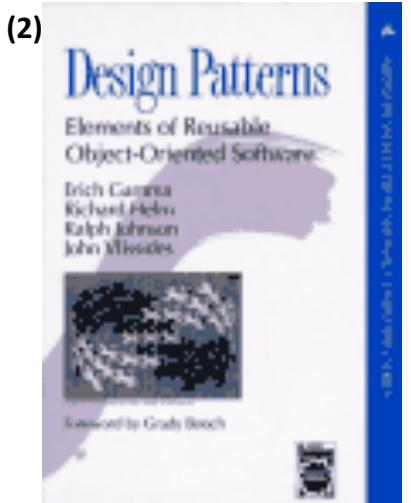
Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 255ff

Überblick - Entwurfsmuster

- Struktur
 - Adapter^(1,2)
 - Brücke (Bridge) ^(1,2)
 - Dekorierer (Decorator) ^(1,2)
 - Fassade (Facade) ^(1,2)
 - Fliegengewicht (Flyweight) ⁽²⁾
 - Kompositum (Composite) ^(1,2)
 - Proxy^(1,2)
- Verhalten
 - Befehl (Command) ^(1,2)
 - Beobachter (Observer) ^(1,2)
 - Besucher (Visitor) ^(1,2)
 - Interpreter (Interpreter) ⁽²⁾
 - Iterator^(1,2)
 - Memento⁽²⁾
 - Rolle⁽¹⁾
 - Schablonenmethode (Template Method) ^(1,2)
 - Strategie (Strategy) ^(1,2)
 - Vermittler (Mediator) ^(1,2)
 - Zustand (State) ^(1,2)
 - Zuständigkeitskette (Chain of Responsibility) ⁽²⁾
- Erzeugung
 - Abstrakte Fabrik (Abstract Factory) ^(1,2)
 - Erbauer (Builder) ⁽²⁾
 - Fabrikmethode (Factory Method) ^(1,2)
 - Objektpool⁽¹⁾
 - Prototyp (Prototype) ^(1,2)
 - Singleton^(1,2)

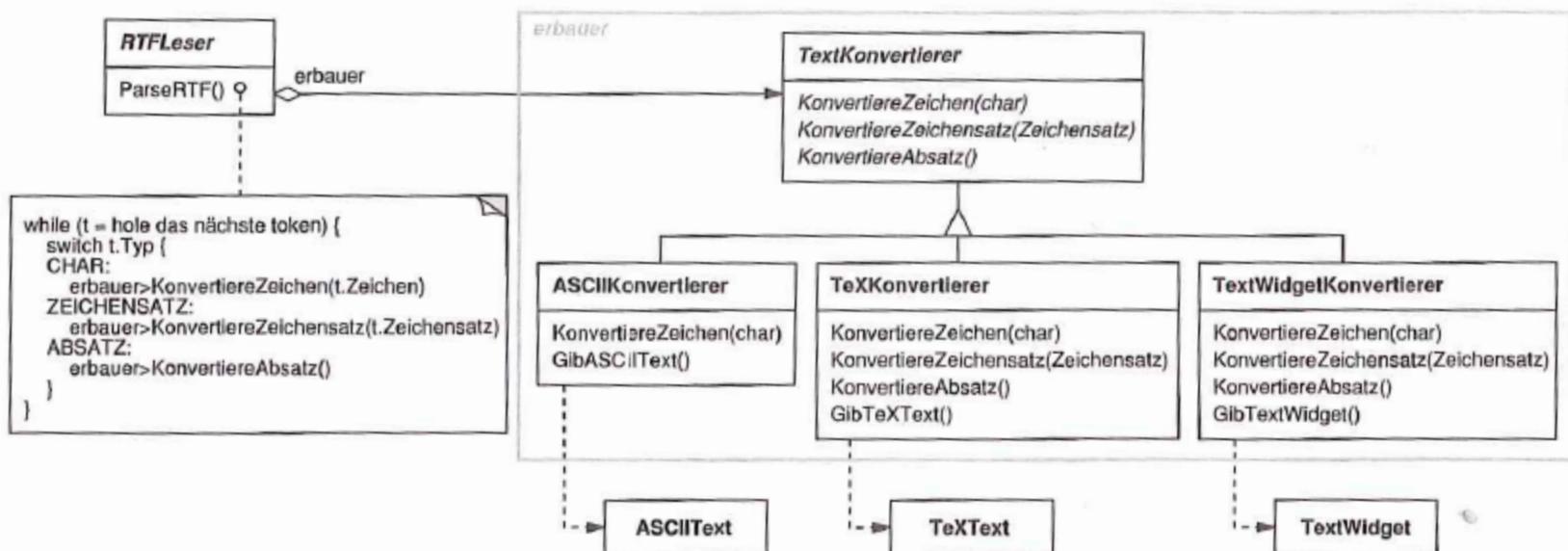
Legende:

Fett = wird in der Vorlesung behandelt



Builder (1)

- **Name:** Builder (dt. Erbauer)
- **Zweck:** Trenne die Konstruktion eines komplexen Objekts von seiner Repräsentation, so dass derselbe Konstruktionsprozess unterschiedliche Repräsentationen erzeugen kann.
- **Motivation:**



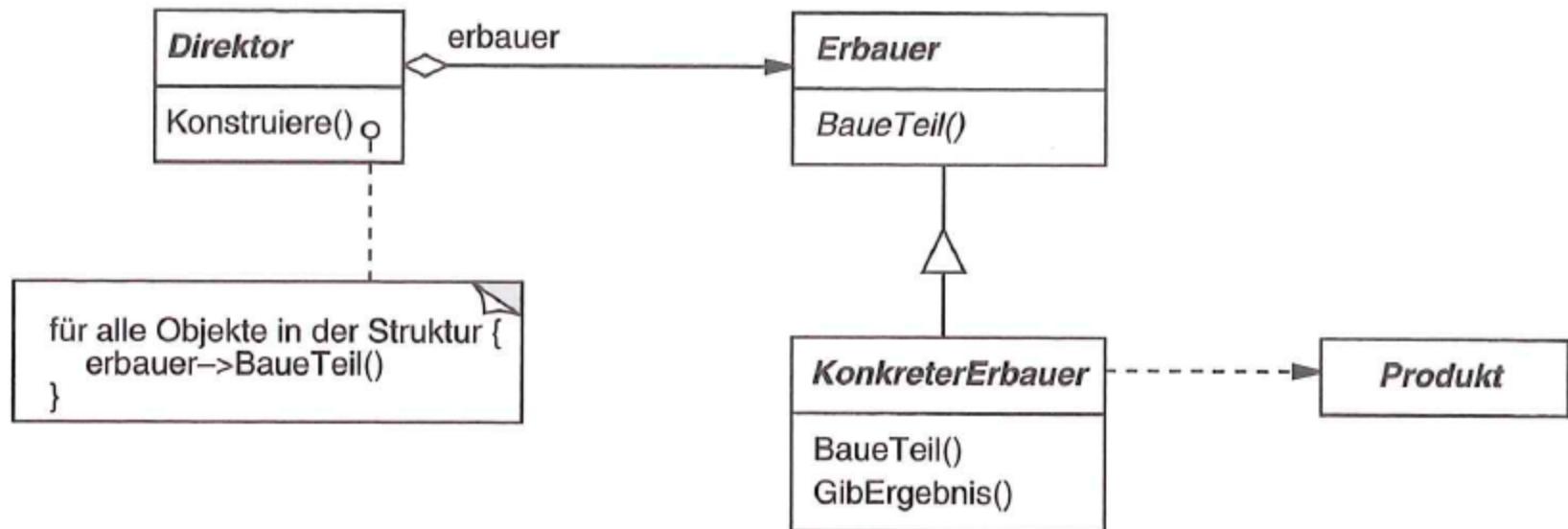
Quelle: Entwurfsmuster, Seite 120

Builder (2)

- **Anwendbarkeit:**

- Der Algorithmus zum Erzeugen eines komplexen Objekts soll unabhängig von den Teilen sein, aus denen das Objekt besteht und wie sie zusammengesetzt werden
- Der Konstruktionsprozess muss verschiedene Repräsentationen des zu konstruierenden Objekt erlauben

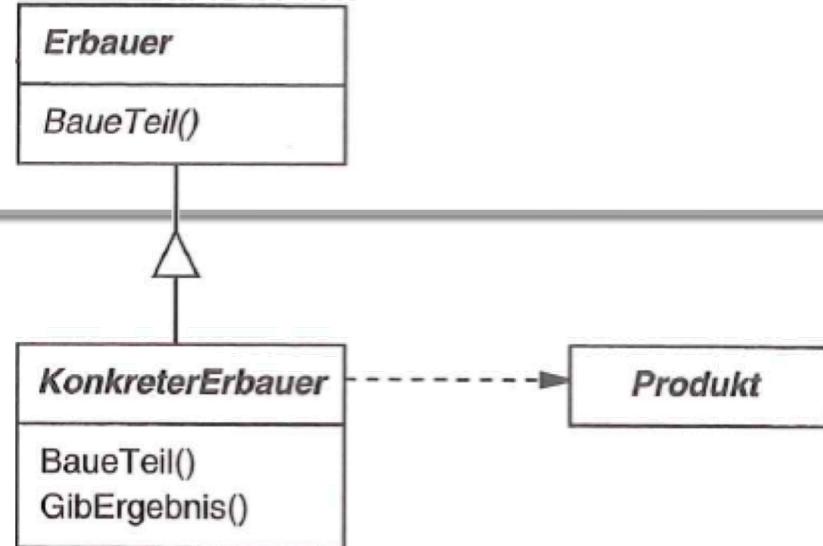
- **Struktur:**



Quelle: Entwurfsmuster, Seite 120f

Builder – Programmbeispiel (1)

Erbauer (abstrakte Schnittstelle)



```
class LabyrinthErbauer {
public:
    virtual void BaueLabyrinth() { }
    virtual void BaueRaum(int raumNr) { }
    virtual void BaueTuer(int vonRaumNr, int nachRaumNr) { }

    virtual Labyrinth* GibLabyrinth() { return 0; }

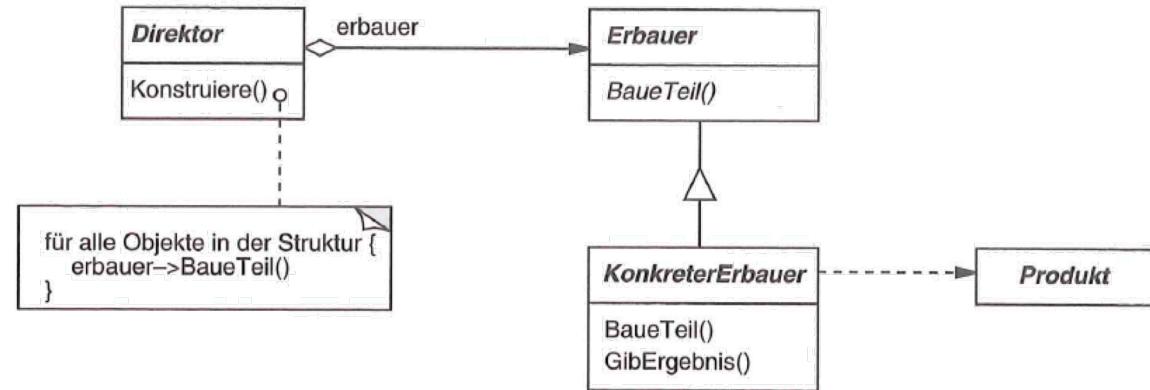
protected:
    LabyrinthErbauer();
};

};
```

Quelle des Beispiels: Entwurfsmuster, S. 121ff

Builder – Programmbeispiel (2)

Direktor (Nutzer des Erbauers)



```
Labyrinth* LabyrinthSpiel::ErzeugeLabyrinth (
    LabyrinthErbauer& erbauer)
{
    erbauer.BaueLabyrinth();

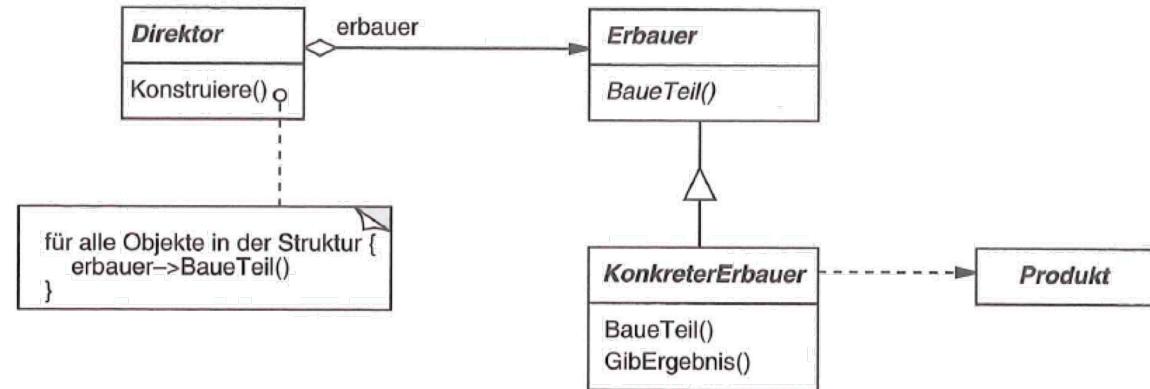
    erbauer.BaueRaum(1);
    erbauer.BaueRaum(2);
    erbauer.BaueTuer(1, 2);

    return erbauer.GibLabyrinth();
}
```

Builder – Programmbeispiel (3)

KonkreterErbauer

Hier: StandardLabyrinthErbauer



```
class StandardLabyrinthErbauer : public LabyrinthErbauer {  
public:  
    StandardLabyrinthErbauer();  
  
    virtual void BaueLabyrinth();  
    virtual void BaueRaum(int raumNr);  
    virtual void BaueTuer(int vonRaumNr, int nachRaumNr);  
  
    virtual Labyrinth* GibLabyrinth();  
  
private:  
    Richtung GemeinsameWand(Raum*, Raum*);  
    Labyrinth* _aktuellesLabyrinth;  
};
```

Builder – Programmbeispiel (4)

```
StandardLabyrinthErbauer::StandardLabyrinthErbauer() {  
    _aktuellerLabyrinth = 0;  
}  
  
void StandardLabyrinthErbauer::BaueLabyrinth() {  
    _aktuellerLabyrinth = new Labyrinth;  
}  
  
Labyrinth* StandardLabyrinthErbauer::GibLabyrinth() {  
    Labyrinth* labyrinth = _aktuellerLabyrinth;  
    return labyrinth;  
}  
  
void StandardLabyrinthErbauer::BaueRaum(int raumNr) {  
    if (! _aktuellerLabyrinth->RaumNr(raumNr)) {  
        Raum* raum = new Raum(raumNr);  
        _aktuellerLabyrinth->FuegeRaumHinzu(raum);  
  
        raum->SetzeSeite(Norden, new Wand);  
        raum->SetzeSeite(Sueden, new Wand);  
        raum->SetzeSeite(Osten, new Wand);  
        raum->SetzeSeite(Westen, new Wand);  
    }  
}
```

Konstruktor

Labyrinth bauen

Labyrinth zurückgeben

Raum bauen

Builder – Programmbeispiel (5)

```
void StandardLabyrinthErbauer::BaueTuer(int raumNr1, int raumNr2)
{
    Raum* raum1 = _aktuellesLabyrinth->RaumNr(raumNr1);
    Raum* raum2 = _aktuellesLabyrinth->RaumNr(raumNr2);
    Tuer* tuer = new Tuer(raum1, raum2);

    raum1->SetzeSeite(GemeinsameWand(raum1, raum2), tuer);
    raum2->SetzeSeite(GemeinsameWand(raum2, raum1), tuer);
}
```

Baue Tuer

```
Labyrinth* labyrinth;
LabyrinthSpiel spiel;
StandardLabyrinthErbauer erbauer;

spiel.ErzeugeLabyrinth(erbauer);
labyrinth = erbauer.GibLabyrinth();
```

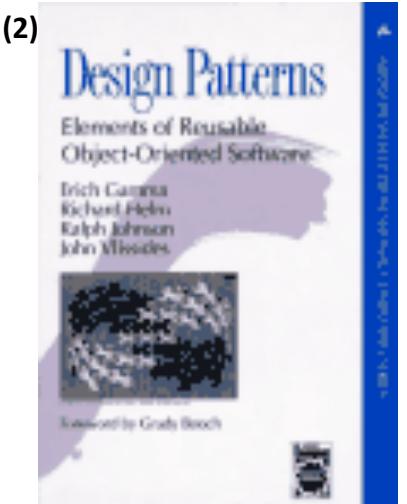
Erzeugung,
ohne konkreten
Konstruktor
kennen zu
müssen

Überblick - Entwurfsmuster

- Struktur
 - Adapter^(1,2)
 - Brücke (Bridge)^(1,2)
 - Dekorierer (Decorator)^(1,2)
 - Fassade (Facade)^(1,2)
 - Fliegengewicht (Flyweight)⁽²⁾
 - Kompositum (Composite)^(1,2)
 - Proxy^(1,2)
- Verhalten
 - Befehl (Command)^(1,2)
 - Beobachter (Observer)^(1,2)
 - Besucher (Visitor)^(1,2)
 - Interpreter (Interpreter)⁽²⁾
 - Iterator^(1,2)
 - Memento⁽²⁾
 - Rolle⁽¹⁾
 - Schablonenmethode (Template Method)^(1,2)
 - Strategie (Strategy)^(1,2)
 - Vermittler (Mediator)^(1,2)
 - Zustand (State)^(1,2)
 - Zuständigkeitskette (Chain of Responsibility)⁽²⁾
- Erzeugung
 - Abstrakte Fabrik (Abstract Factory)^(1,2)
 - Erbauer (Builder)⁽²⁾
 - **Fabrikmethode (Factory Method)**^(1,2)
 - Objektpool⁽¹⁾
 - Prototyp (Prototype)^(1,2)
 - Singleton^(1,2)

Legende:

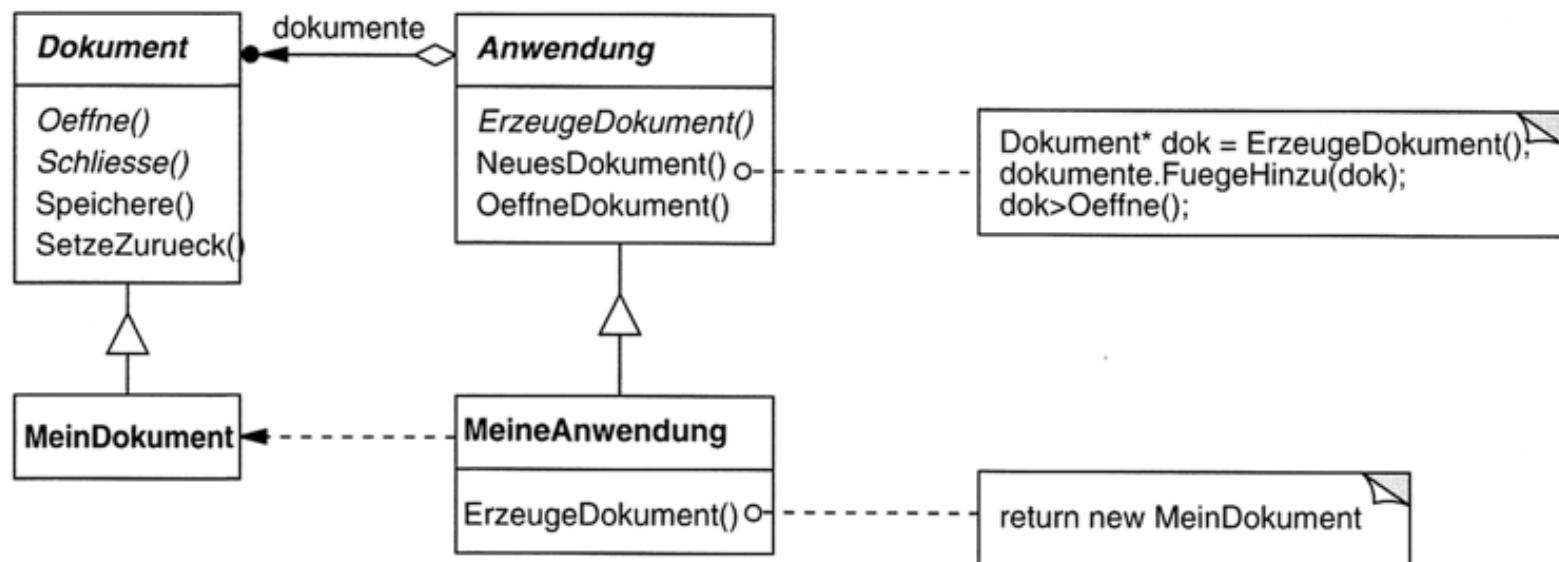
Fett = wird in der Vorlesung behandelt



Factory Method (1)

- **Name:** Factory Method (dt. Fabrikmethode)
- **Zweck:** Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist.

Motivation:



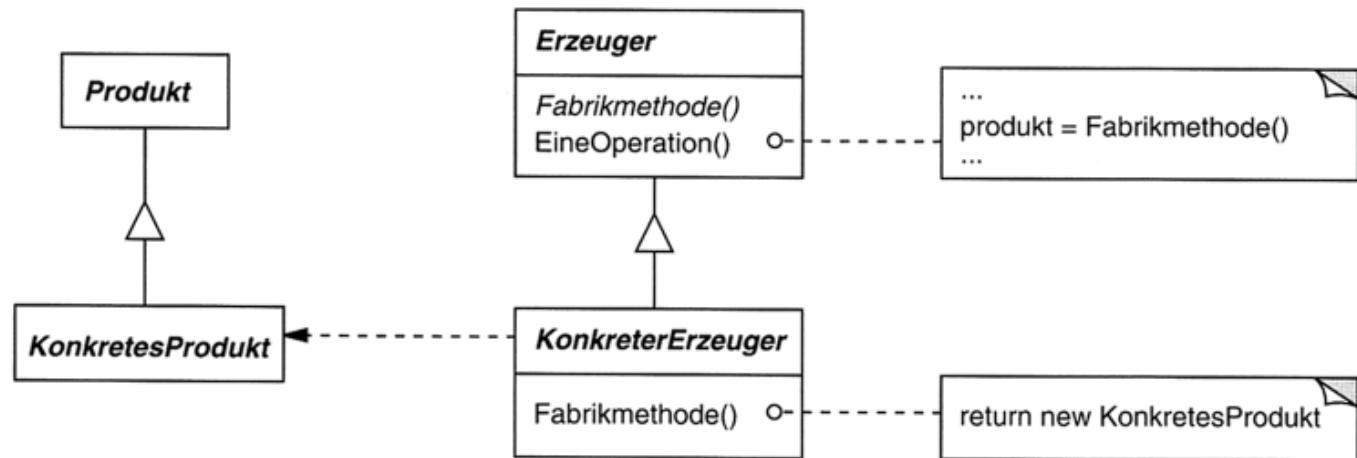
Quelle: Entwurfsmuster, Seite 132

Factory Method (2)

- **Anwendbarkeit:**

- Wenn eine Klasse die Klassen von Objekten, die sie erzeugen muss, nicht im voraus kennen kann.
- Wenn eine Klasse möchte, dass ihre Unterklassen die von ihr zu erzeugenden Objekte festlegen.

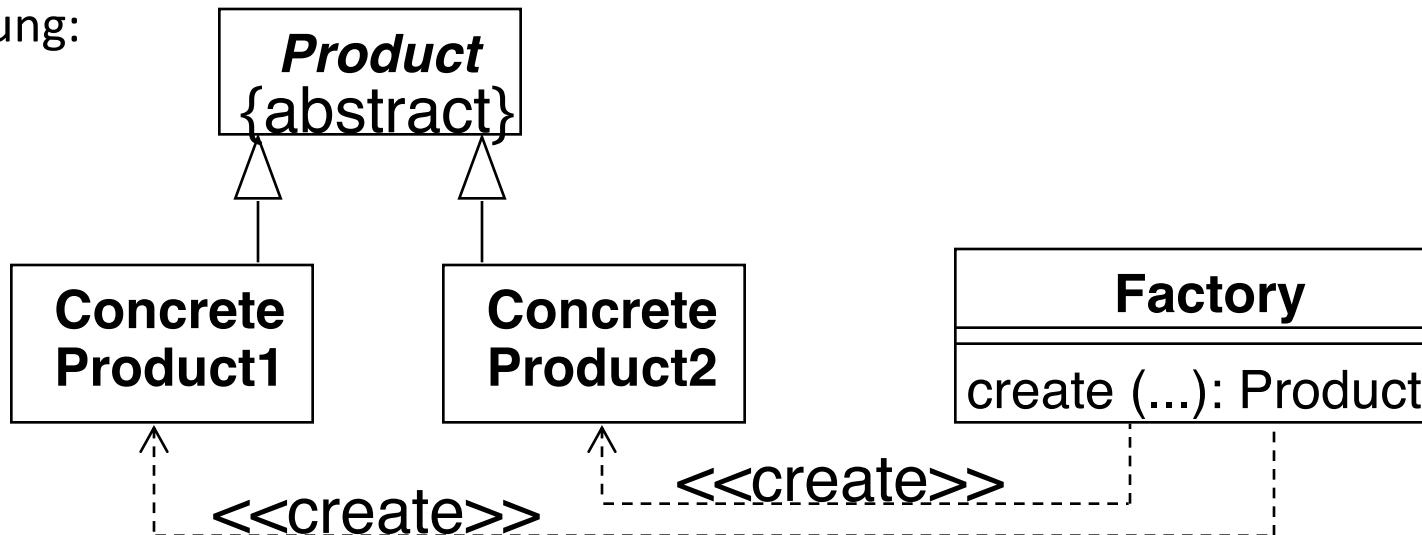
- **Struktur:**



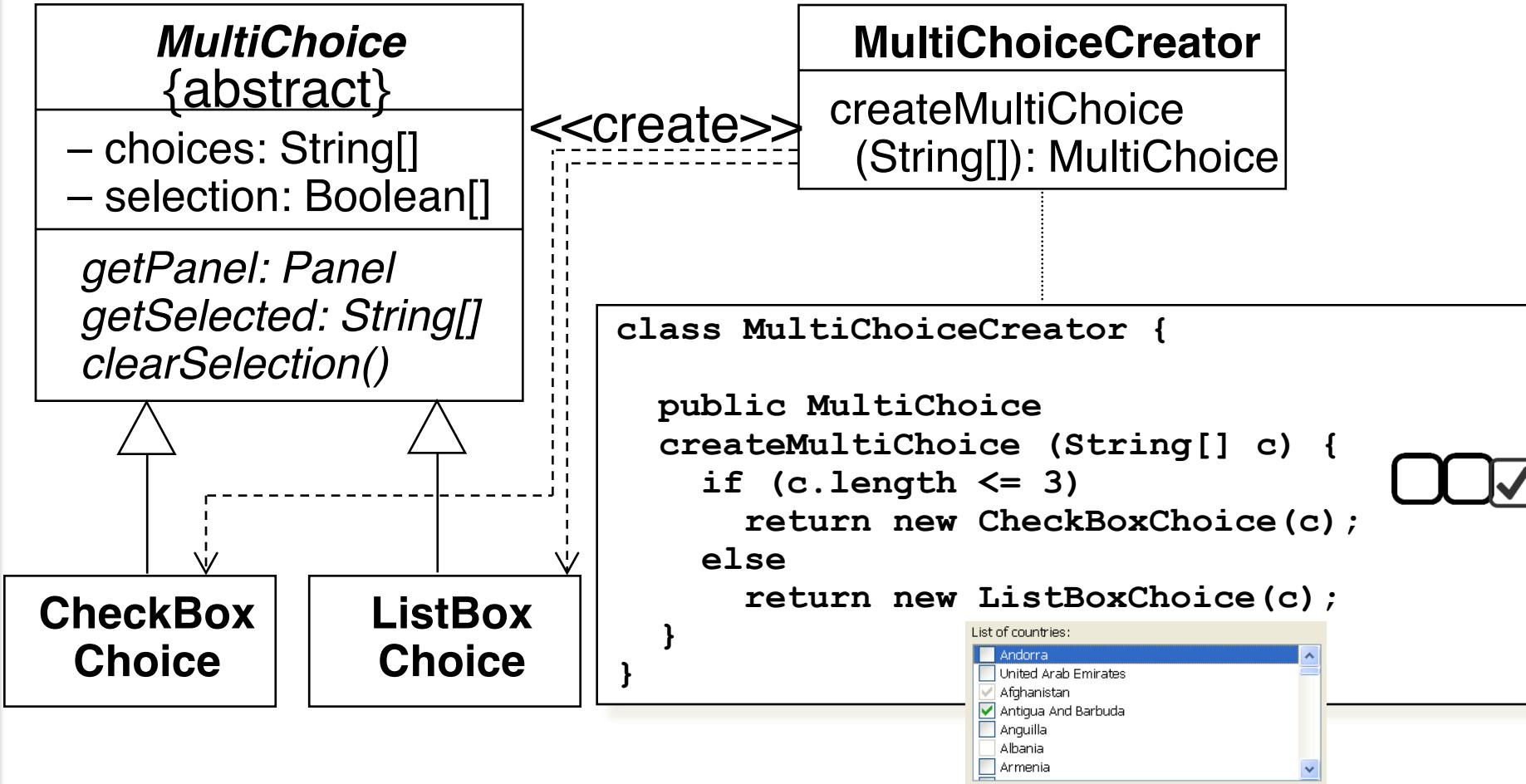
Quelle: Entwurfsmuster, Seite 133

Factory Method (3)

- Name: **Factory Method**
- Problem:
 - Bei der Erzeugung von Objekten soll zwischen Varianten gewählt werden; dies soll aber zum Zeitpunkt der Erzeugung geschehen, ohne dass der Auftraggeber der Erzeugung damit beschäftigt ist.
- Lösung:



Factory Method (4)

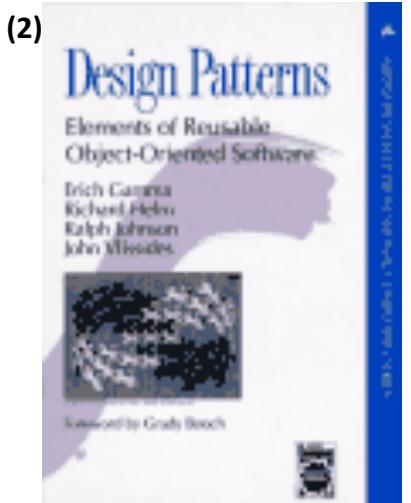


Überblick - Entwurfsmuster

- Struktur
 - Adapter^(1,2)
 - Brücke (Bridge) ^(1,2)
 - Dekorierer (Decorator) ^(1,2)
 - Fassade (Facade) ^(1,2)
 - Fliegengewicht (Flyweight) ⁽²⁾
 - Kompositum (Composite) ^(1,2)
 - Proxy^(1,2)
- Verhalten
 - Befehl (Command) ^(1,2)
 - Beobachter (Observer) ^(1,2)
 - Besucher (Visitor) ^(1,2)
 - Interpreter (Interpreter) ⁽²⁾
 - Iterator^(1,2)
 - Memento⁽²⁾
 - Rolle⁽¹⁾
 - Schablonenmethode (Template Method) ^(1,2)
 - Strategie (Strategy) ^(1,2)
 - Vermittler (Mediator) ^(1,2)
 - Zustand (State) ^(1,2)
 - Zuständigkeitskette (Chain of Responsibility) ⁽²⁾
- Erzeugung
 - Abstrakte Fabrik (Abstract Factory) ^(1,2)
 - Erbauer (Builder) ⁽²⁾
 - Fabrikmethode (Factory Method) ^(1,2)
 - Objektpool⁽¹⁾
 - Prototyp (Prototype) ^(1,2)
 - Singleton^(1,2)

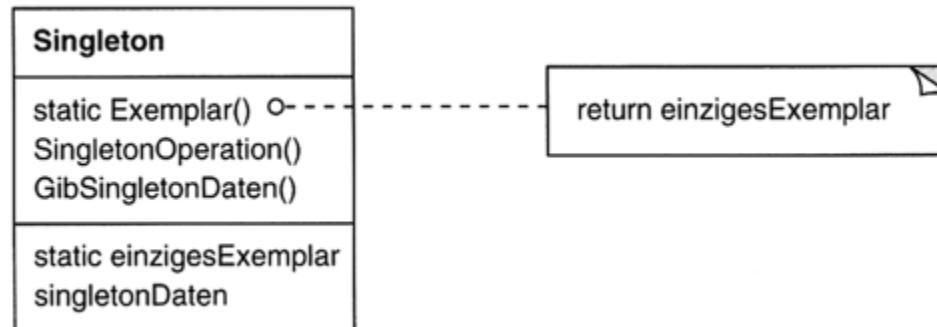
Legende:

Fett = wird in der Vorlesung behandelt



Singleton

- **Name:** Singleton (dt. Singleton)
- **Zweck:** Sichere ab, dass eine Klasse genau ein Exemplar besitzt, und stelle einen globalen Zugriffspunkt darauf bereit.
- **Anwendbarkeit:**
 - Wenn es genau ein Exemplar einer Klasse geben und es für Klienten an einem wohldefinierten Punkt zugreifbar sein muss.
 - Wenn das einzige Exemplar durch Bildung von Unterklassen erweiterbar sein soll und Klienten in der Lage sein sollen, das erweiterte Exemplar ohne Modifikation ihres Codes verwenden zu können.
- **Struktur:**



Quelle: Entwurfsmuster, Seite 158

Singleton Muster (2)

- **Problem:** Manche Klassen sind nur sinnvoll, wenn sichergestellt ist, dass immer höchstens eine Instanz der Klasse besteht (und diese bei Bedarf erzeugt wird).
- **Lösung:**
 - Modellebene: Klassen als Singleton auszeichnen
 - Programmebene: Sprachabhängig
- Beispiel (Java):

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

Singleton Muster - Programmbeispiel

```
// Datei: ToolTipManager.java
final class ToolTipManager
{
    private static ToolTipManager instance;

    private ToolTipManager()
    {
        System.out.println ("Neues Singleton erzeugt.")
    }

    public static ToolTipManager getInstance()
    {
        if (instance == null)
        {
            instance = new ToolTipManager ();
        }
        return instance;
    }

    public void operation()
    {
        // eigentliche Funktionalitaet des Singleton
        System.out.println ("operation() aufgerufen.");
    }
}
```

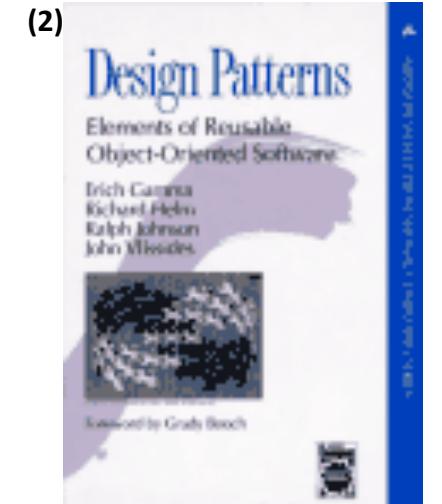
Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 264f

Überblick - Architekturmuster

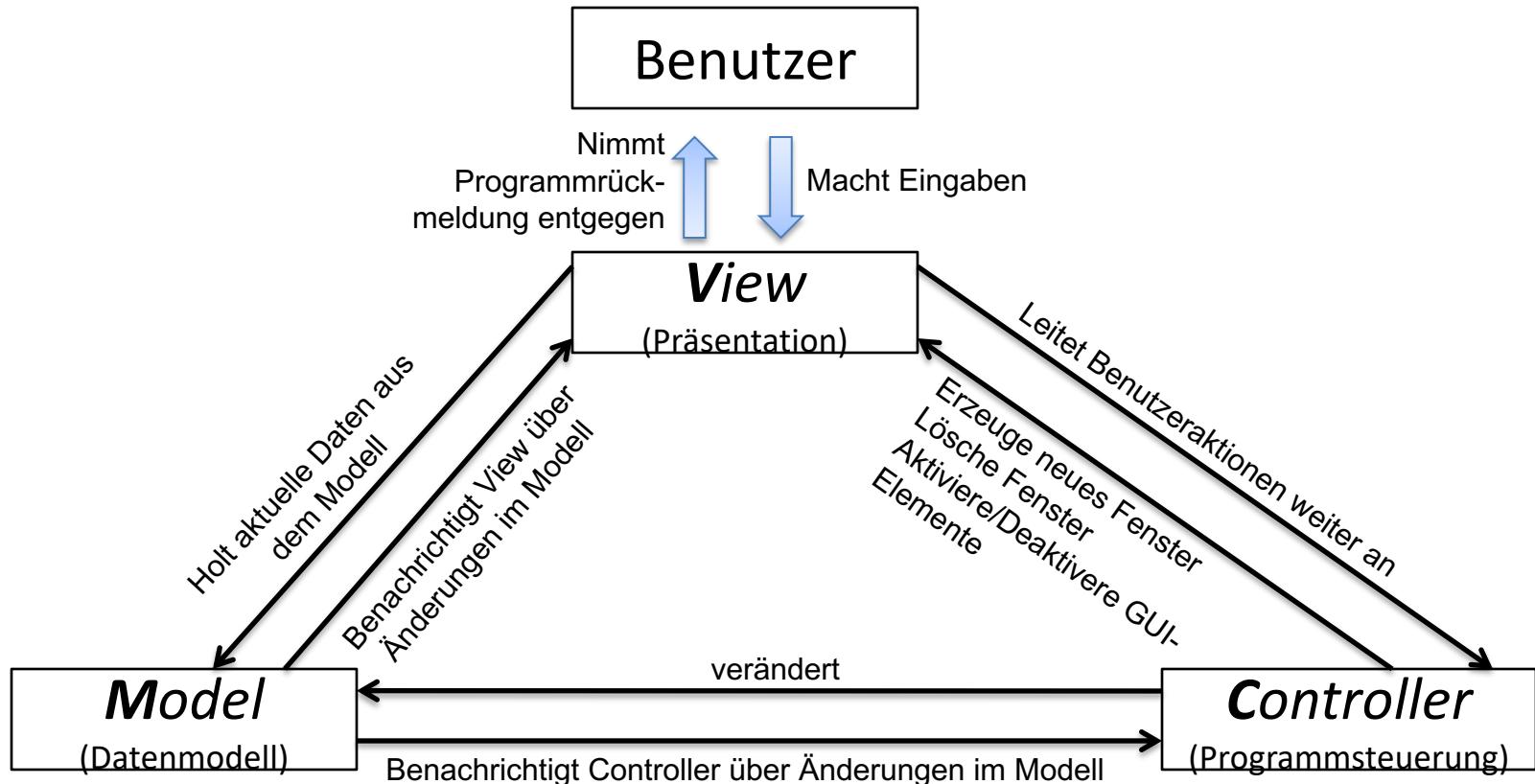
- Layers⁽¹⁾
- Pipes and Filters⁽¹⁾
- Plug-in⁽¹⁾
- Broker⁽¹⁾
- Service-Oriented Architecture⁽¹⁾
- Model-View-Controller⁽¹⁾

Legende:

Fett = wird in der Vorlesung behandelt

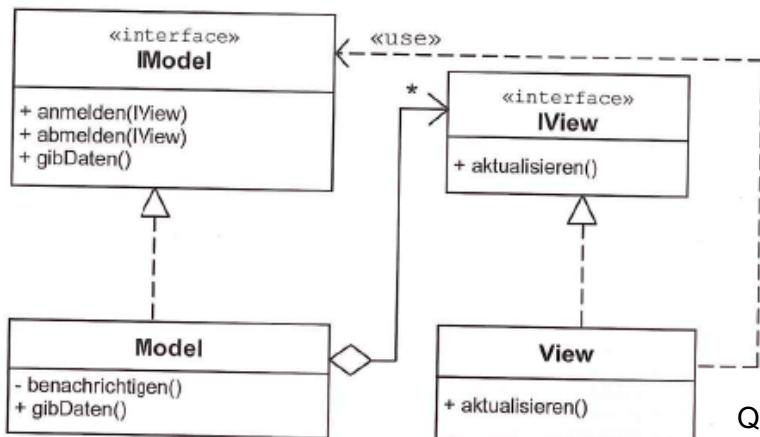


Model-View-Controller (Architekturmuster)



Model-View-Controller

- **Modell**
 - Kapselt fachliche Objekte
 - Kennt die Views und Controller, die sich angemeldet haben
 - Benachrichtigt angemeldete Komponenten über Zustandsänderungen
 - Unabhängig von konkreten Views und Controllern



- Beobachter Muster in MVC → lose Kopplung von Model zu View
- Model ist Rolle des Beobachtbaren
- View ist Rolle des Beobachters

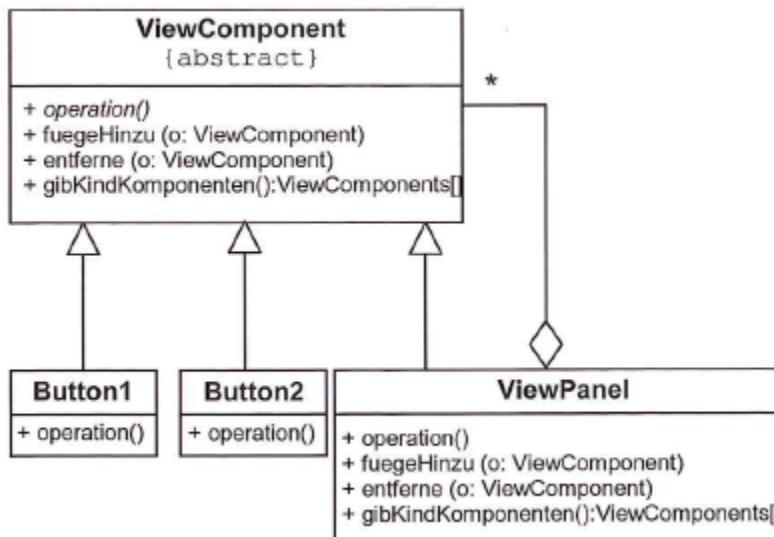
Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 385ff

Model-View-Controller

- **View**

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 385ff

- Darstellung des Modells für den Benutzer
- Kennt Modell und seinen Controller
- Keine Weiterverarbeitung der übergebenen Daten
- Realisiert einen Update-Callback-Mechanismus (Observer)

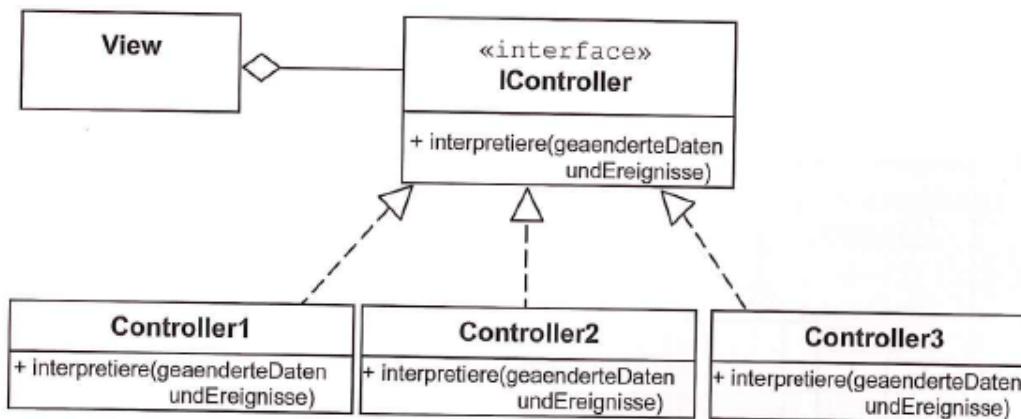


- Kompositum Muster in MVC → Zusammensetzen von grafischen Oberflächen
- Die View besteht aus Buttons und Panels
- ViewPanel → Kompositum
- Button Klassen sind Blätter

Model-View-Controller

- **Controller**

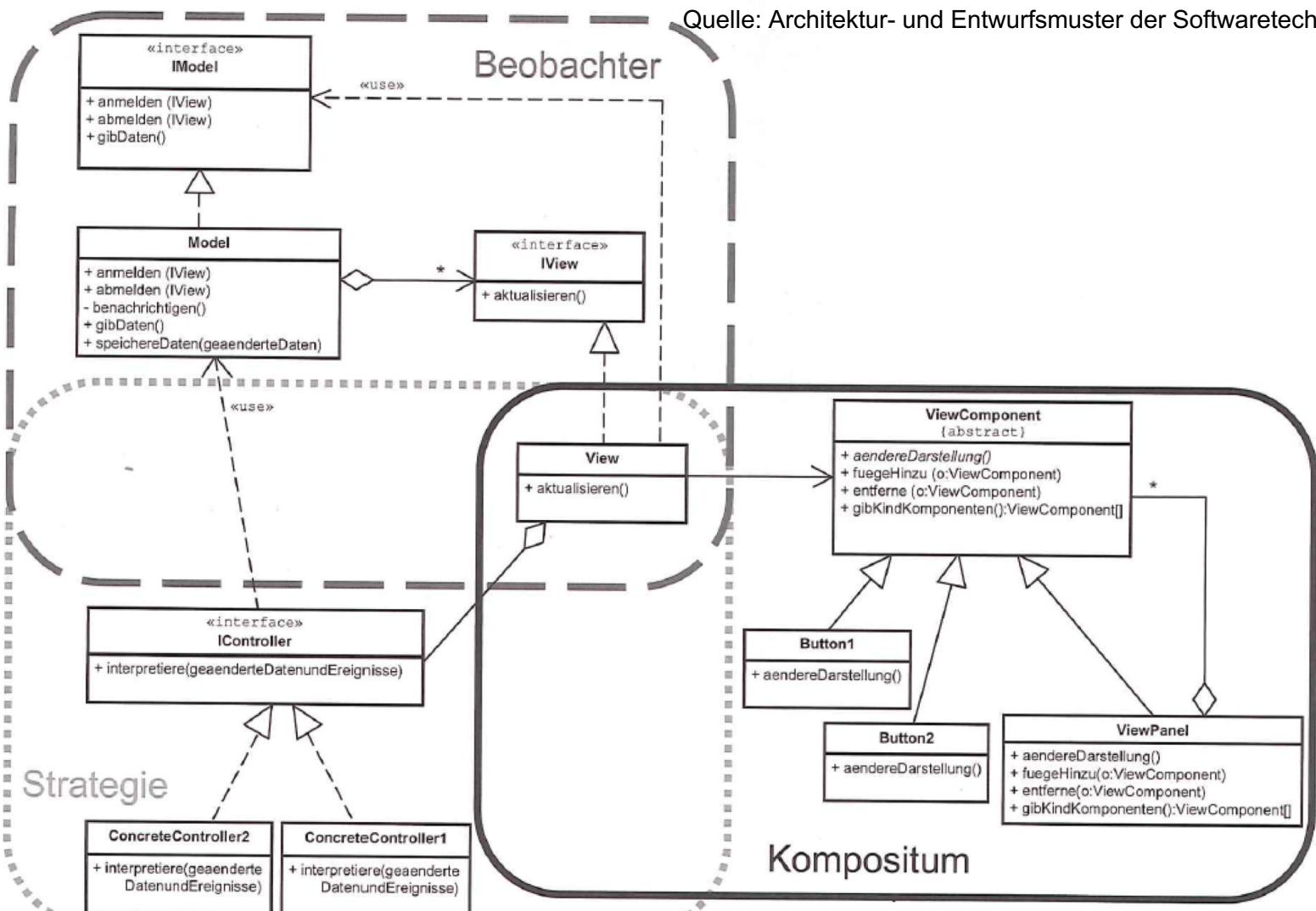
- Verarbeitet Ereignisse aus der View (od. vom Benutzer)
- Übersetzt Ereignisse in Anfragen/Befehle an Modell oder View
- Realisiert einen Update-Callback-Mechanismus
- Kann unterschiedliche Views bedienen.



- Beziehung zwischen View und Controller → Strategie M.
- Controller stellt Strategie (Verhalten) dar, dass die Nutzereingaben interpretiert
- Delegation an View

Quelle: Architektur- und Entwurfsmuster der Softwaretechnik, Seite 385ff

Zusammenspiel von Beobachter-, Strategie- und Kompositum-Muster

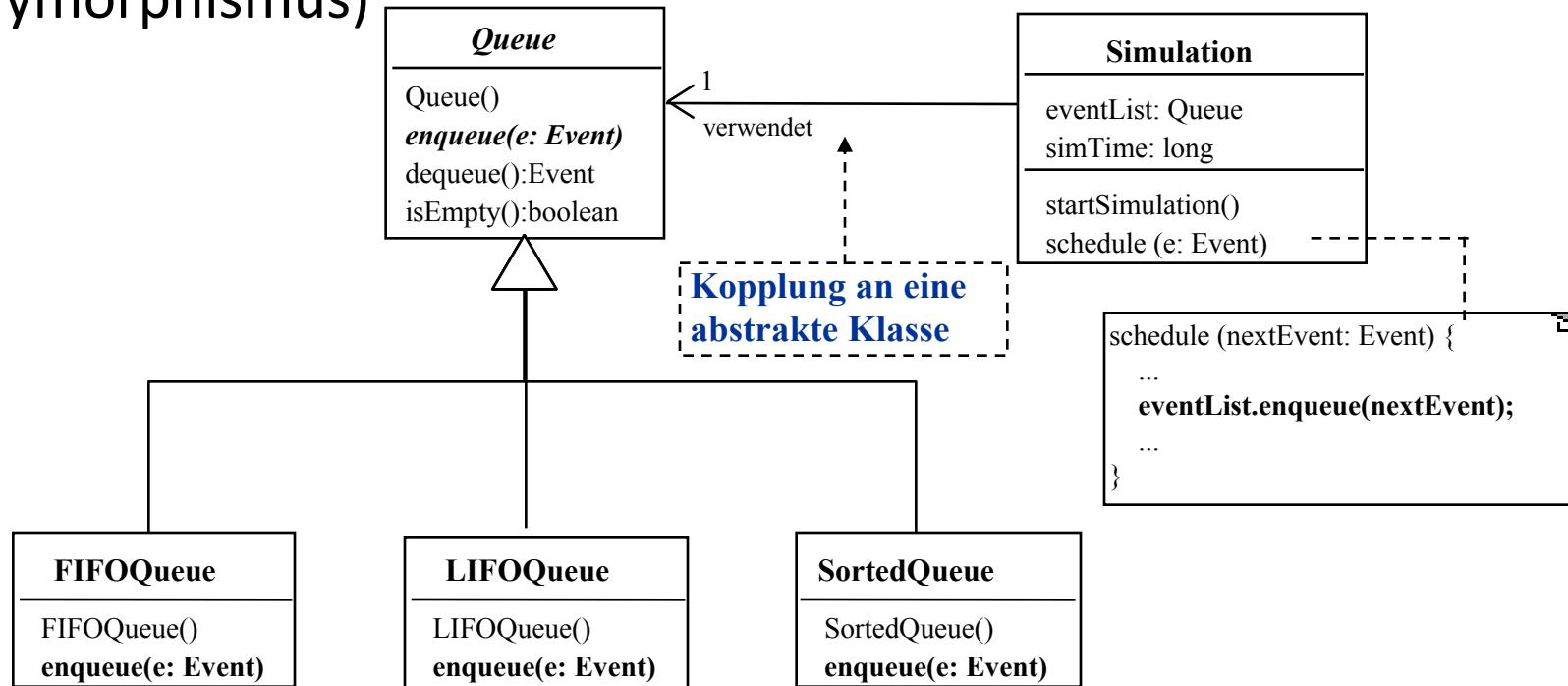


Model-View-Controller

- Vorteile
 - Erlaubt mehrere Sichten auf ein Modell
 - Automatische Synchronisation der Sichten
 - Austauschbarkeit von Darstellung und Kontrollablauf
- Nachteile
 - Relative enge Kopplung zwischen View und Controller
 - Hohe Anzahl an Aktualisierungen möglicherweise ein Performance-Problem

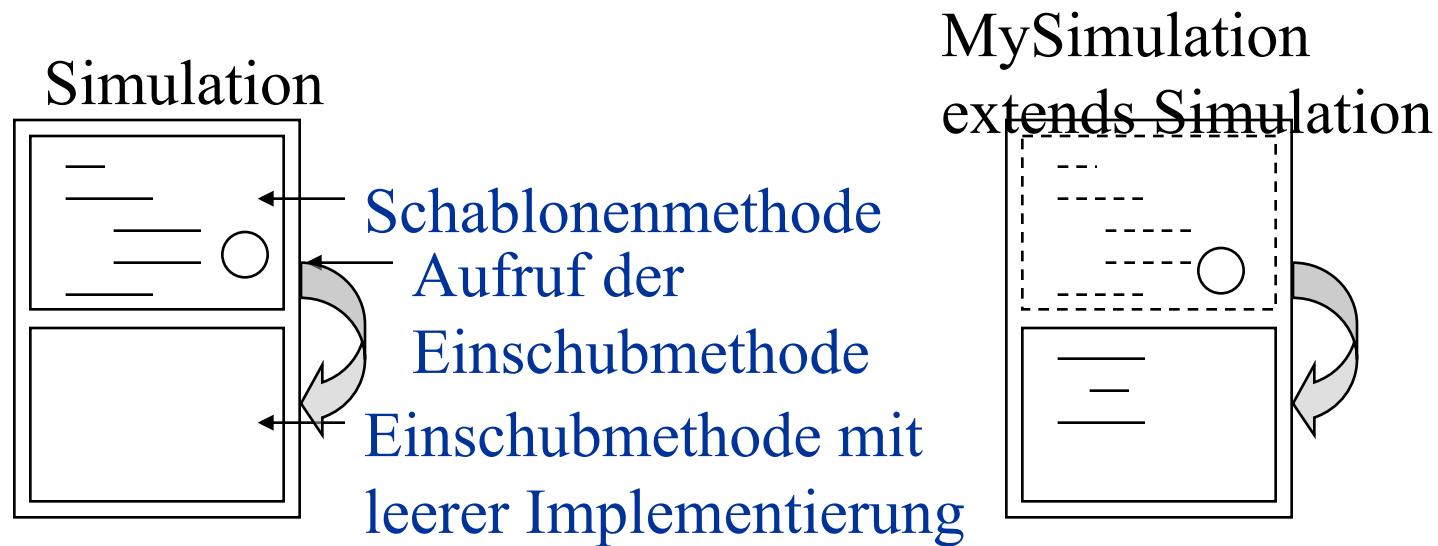
Metapattern von Pree: Abstrakte Kopplung

- abstrakte Klassen definieren eine Schnittstelle, zu der abgeleitete, implementierte Klassen kompatibel sind (Polymorphismus)



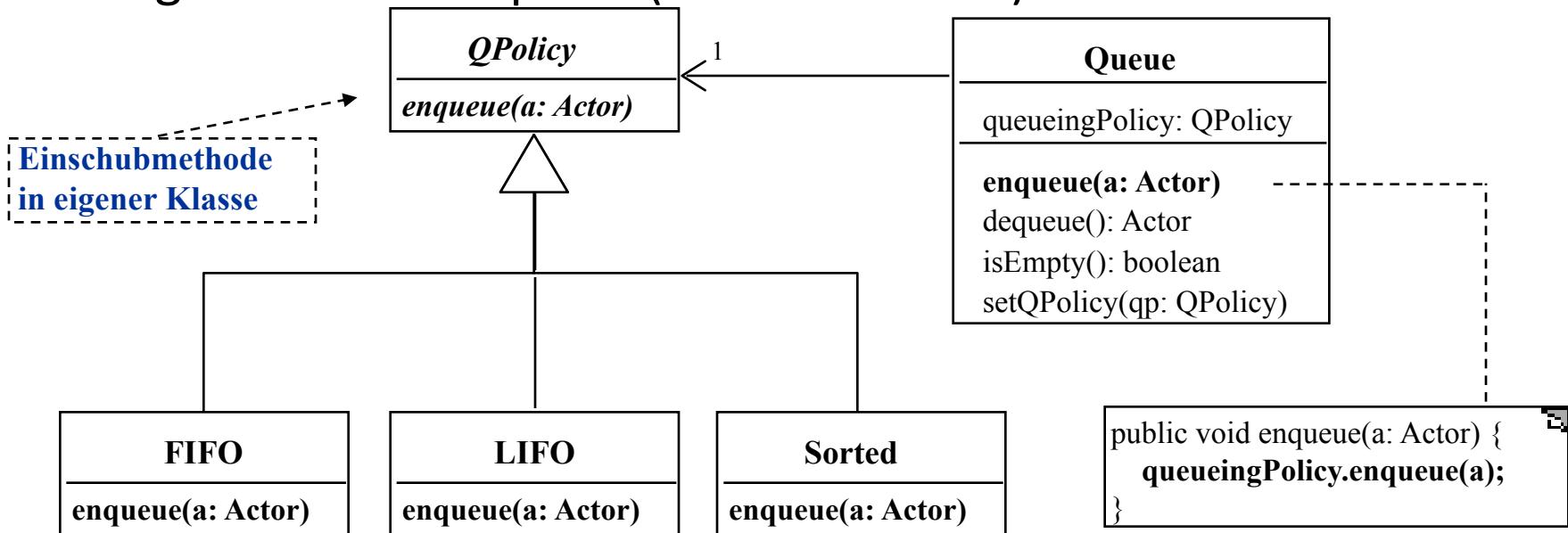
Metapattern von Pree: Einschubmethoden / Hot Spots

- Flexibilität durch Plazierung von “Einschubmethoden”-Aufrufen in “Schablonenmethode”
- Grundidee: Verhalten eines Objektes verändern, ohne den Quellcode der zugehörigen Klasse ändern zu müssen.



Metapattern von Pree: Einschubklassen

- Problem : Schablonen- und Einschubmethode in einer Klasse erlauben keine Anpassung zur Laufzeit!
- Lösung : Komposition statt Vererbung. Einschubmethode in eigener Klasse kapseln (Einschubklasse)



Literatur (1)



Design Patterns

Elements of Reusable
Object-Oriented Software

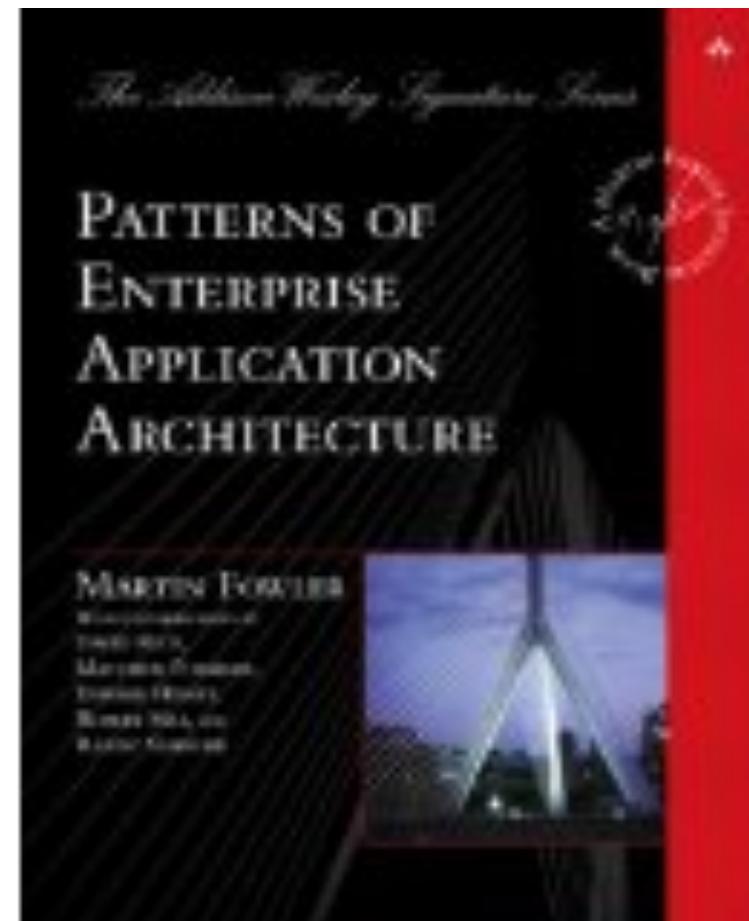
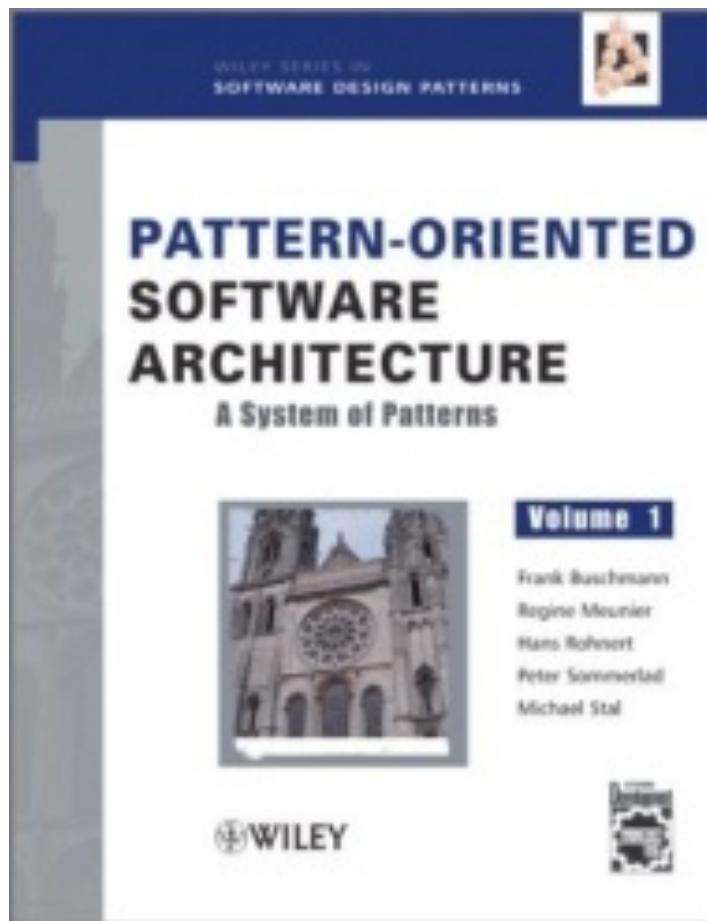
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Literatur (2)

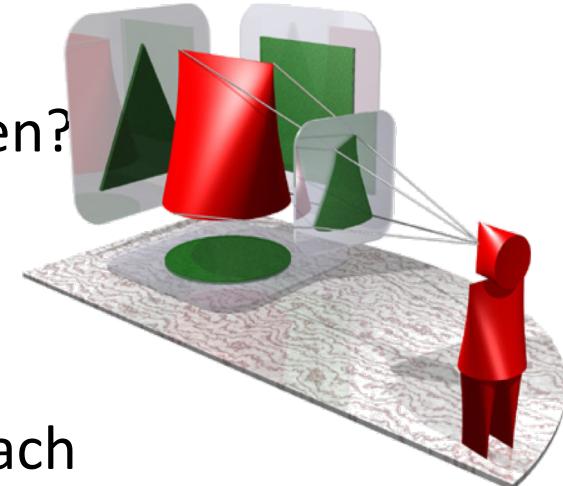


Inhalt

- Motivation und Herausforderungen
- Überblick und Konzepte
- Design Patterns
- Entwurf und Beschreibung
 - Analysemodell
 - Architekturmodell: Structural View
 - Architekturmodell: Deployment View
 - Architekturmodell: Behavioral View
 - Implementierungsmodell

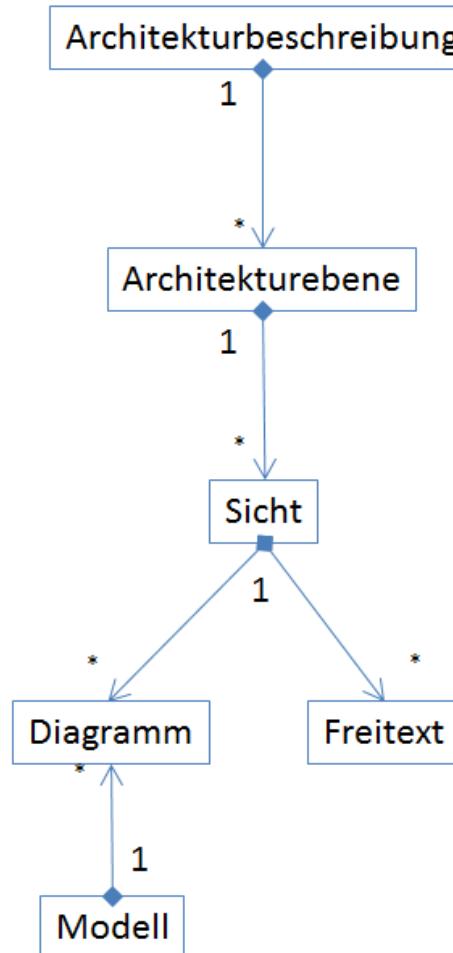
Wie kann man eine Architektur beschreiben?

- Architekturen werden mit Hilfe von (UML-)Modellen beschrieben
 - **Modelle** sind vereinfachte Abbildungen der Realität, die es erlauben Aussagen im Modell so darzustellen, dass sie von allen Beteiligten gleich interpretiert werden.
 - Generierung, Simulation, Überprüfung, etc. möglich
- Wie werden Architekturmodelle beschrieben?
 - Architekturmodelle werden mit Hilfe von **Sichten** spezifischen Sichten beschrieben.
 - Sicht der Anwendungskomponenten, Verteilungssicht, Deployment-Sicht, etc.
- Eine Architekturbeschreibung sollte so einfach wie möglich sein, aber so umfangreich wie notwendig.
- “You know you've achieved perfection in design, not when you have nothing more to add, but when you have nothing more to take away.”

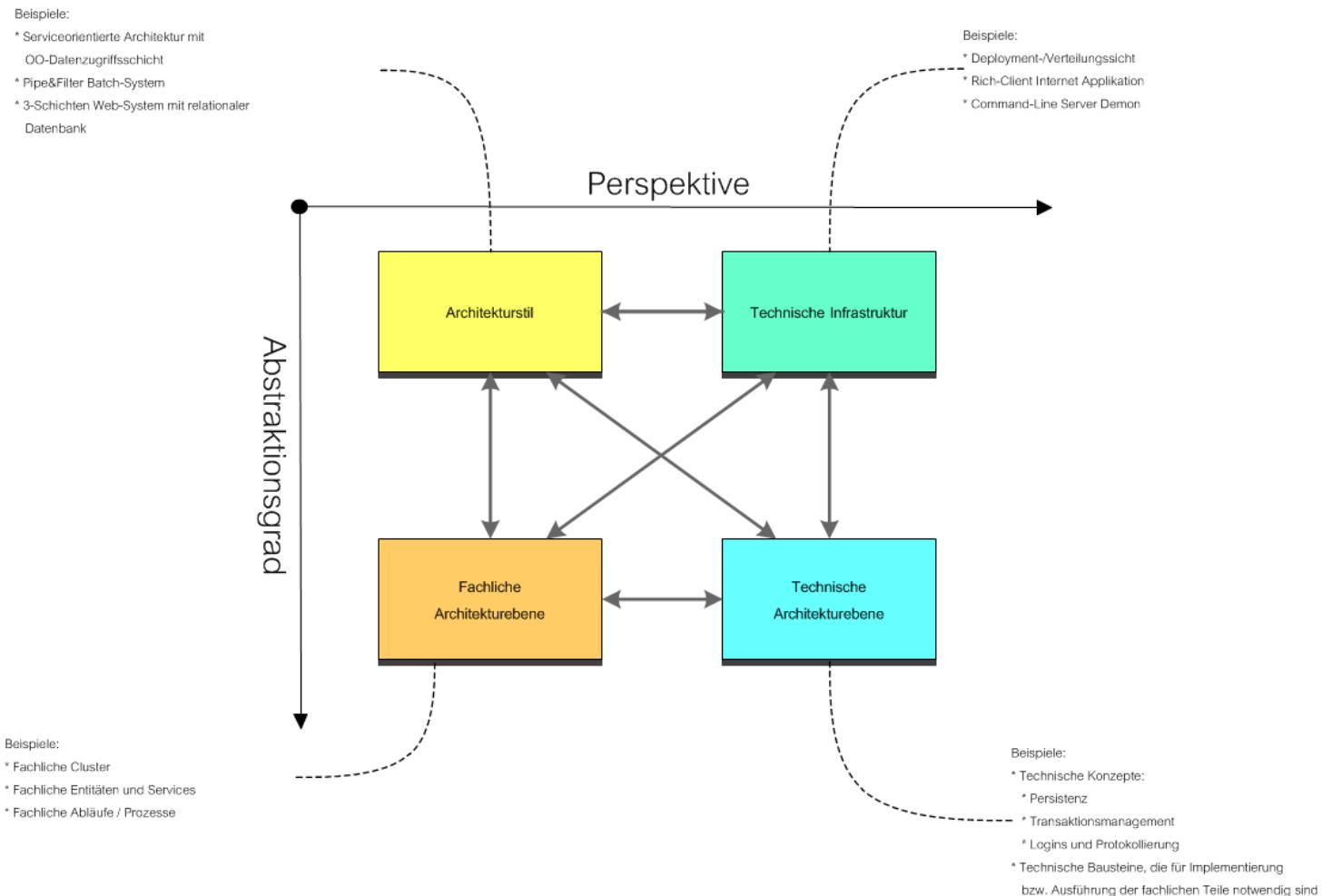


[nach Antoine de Saint-Exupery]

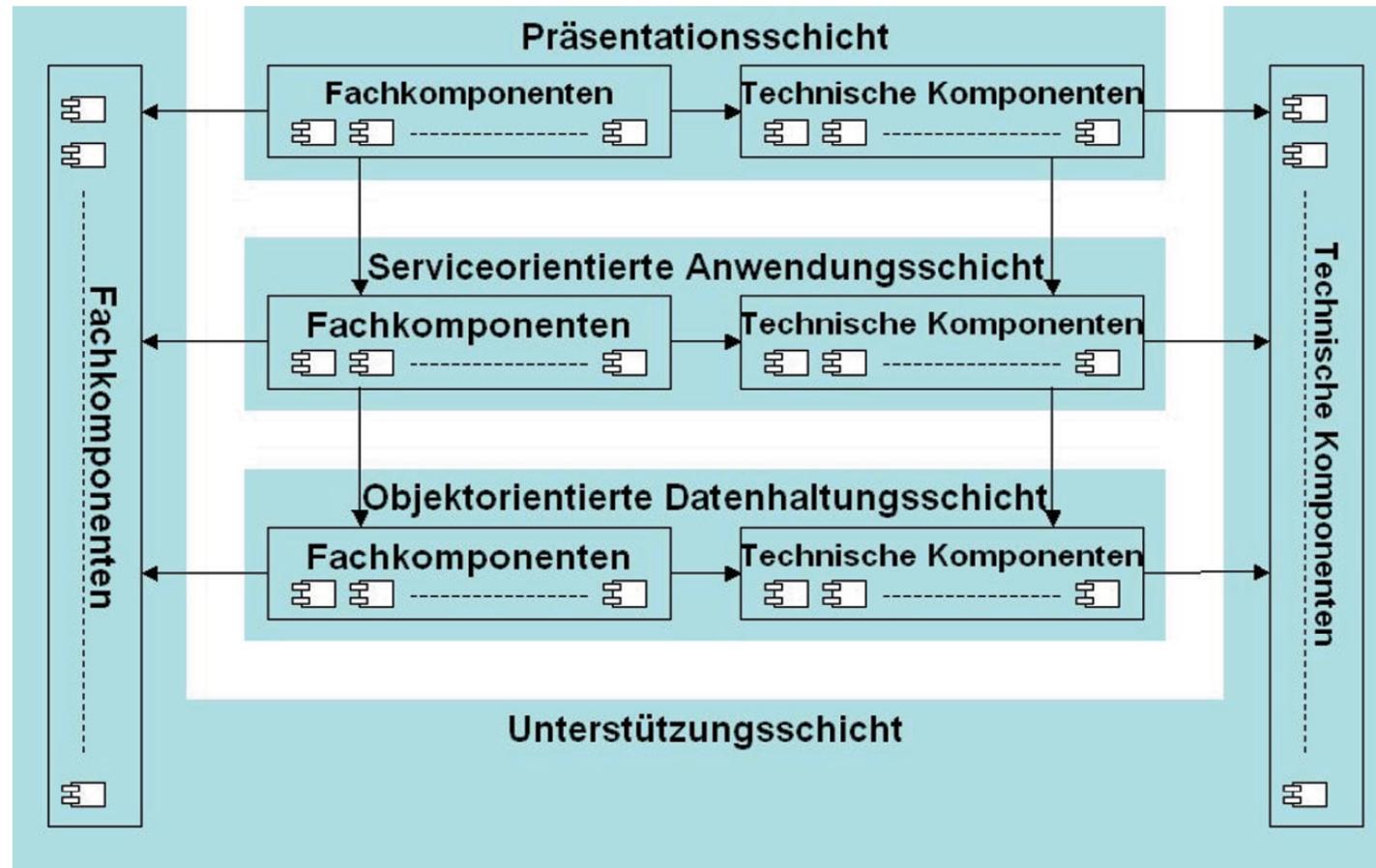
Einfaches grundlegendes Metamodell für Architekturbeschreibungen



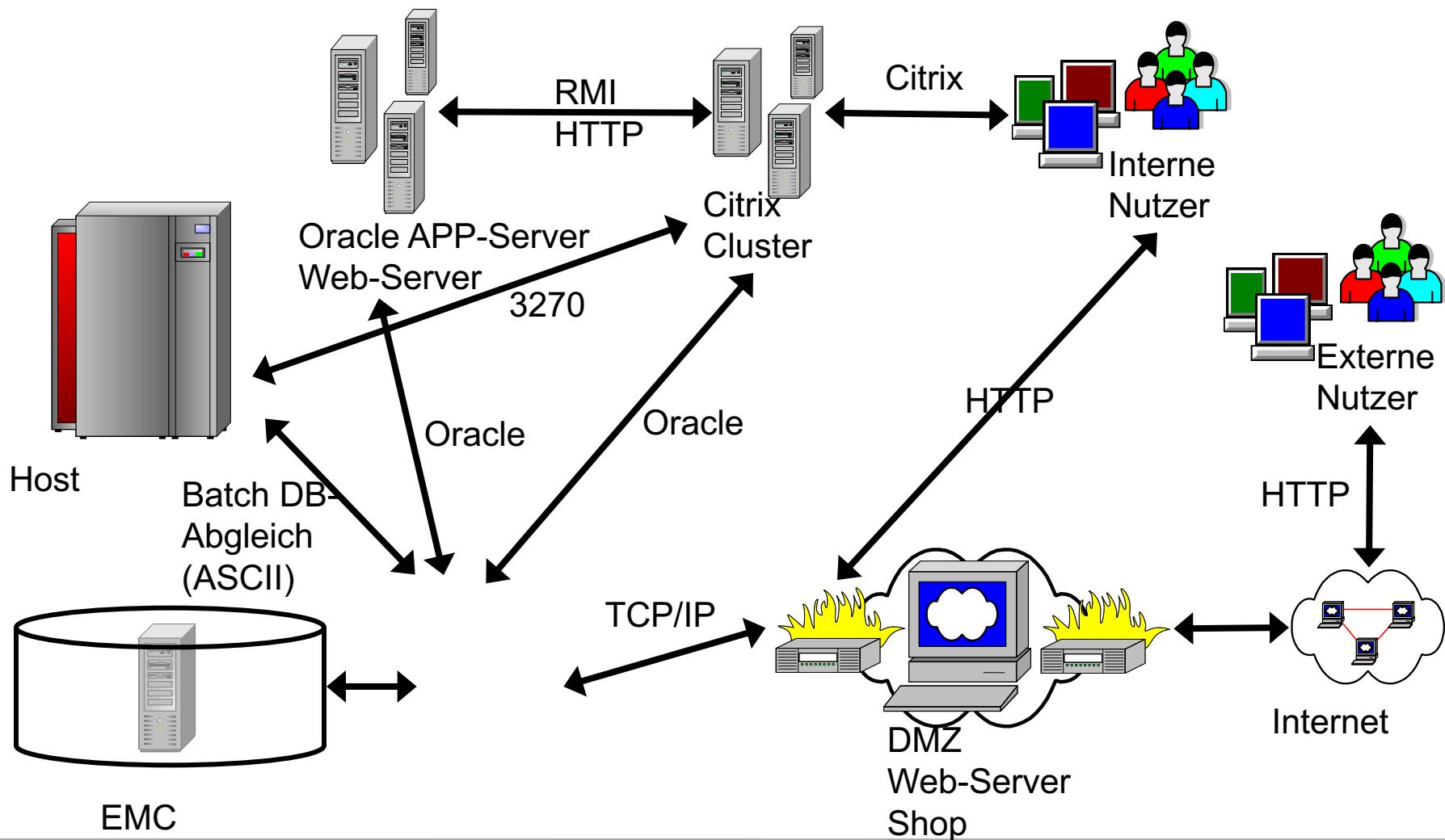
Architekturebene: Verschiedene Abstraktionsgrade und Perspektiven von Architekturbeschreibungen



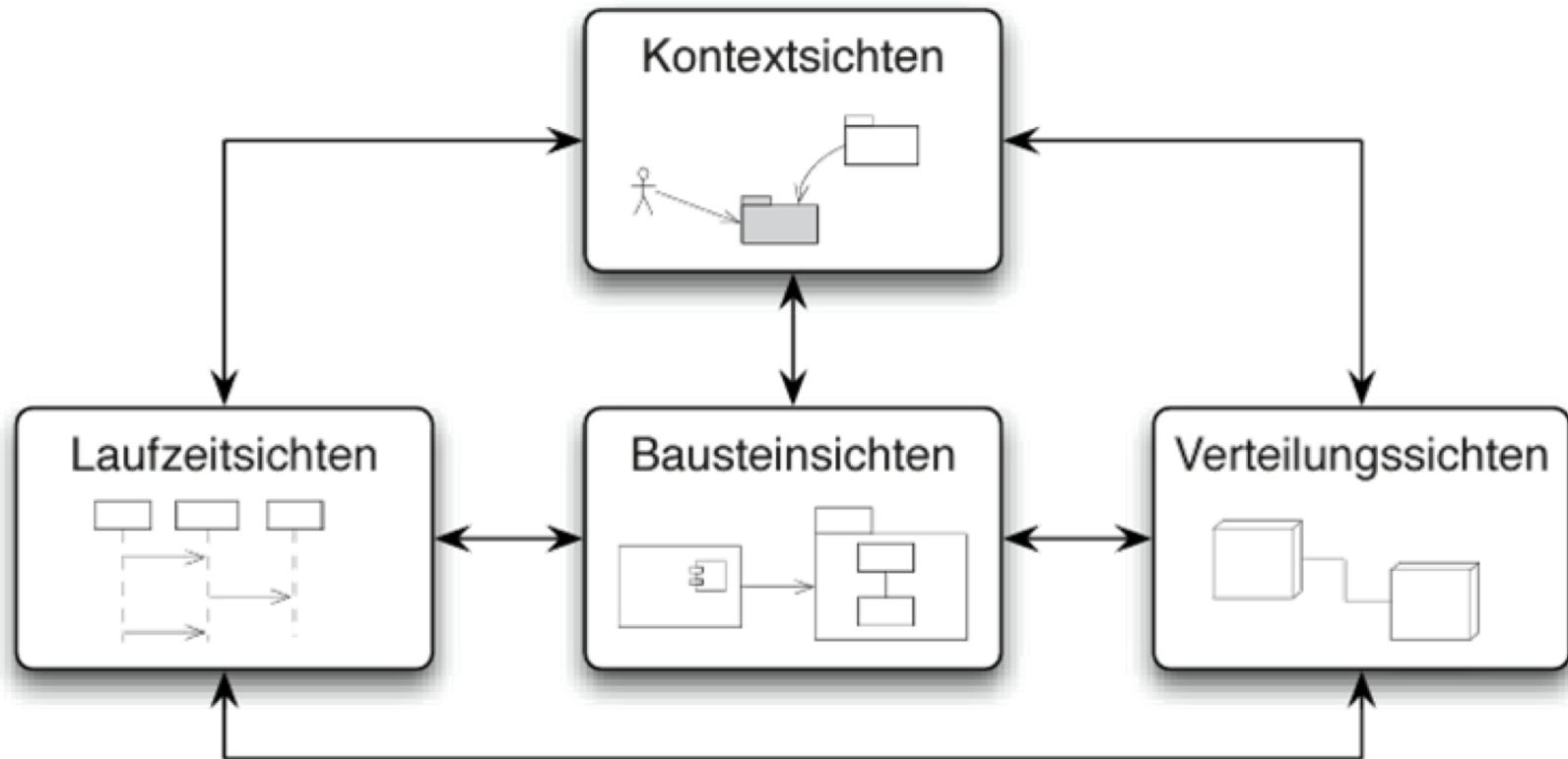
Beispiel: Architekturstil



Beispiel: Technische Infrastruktur

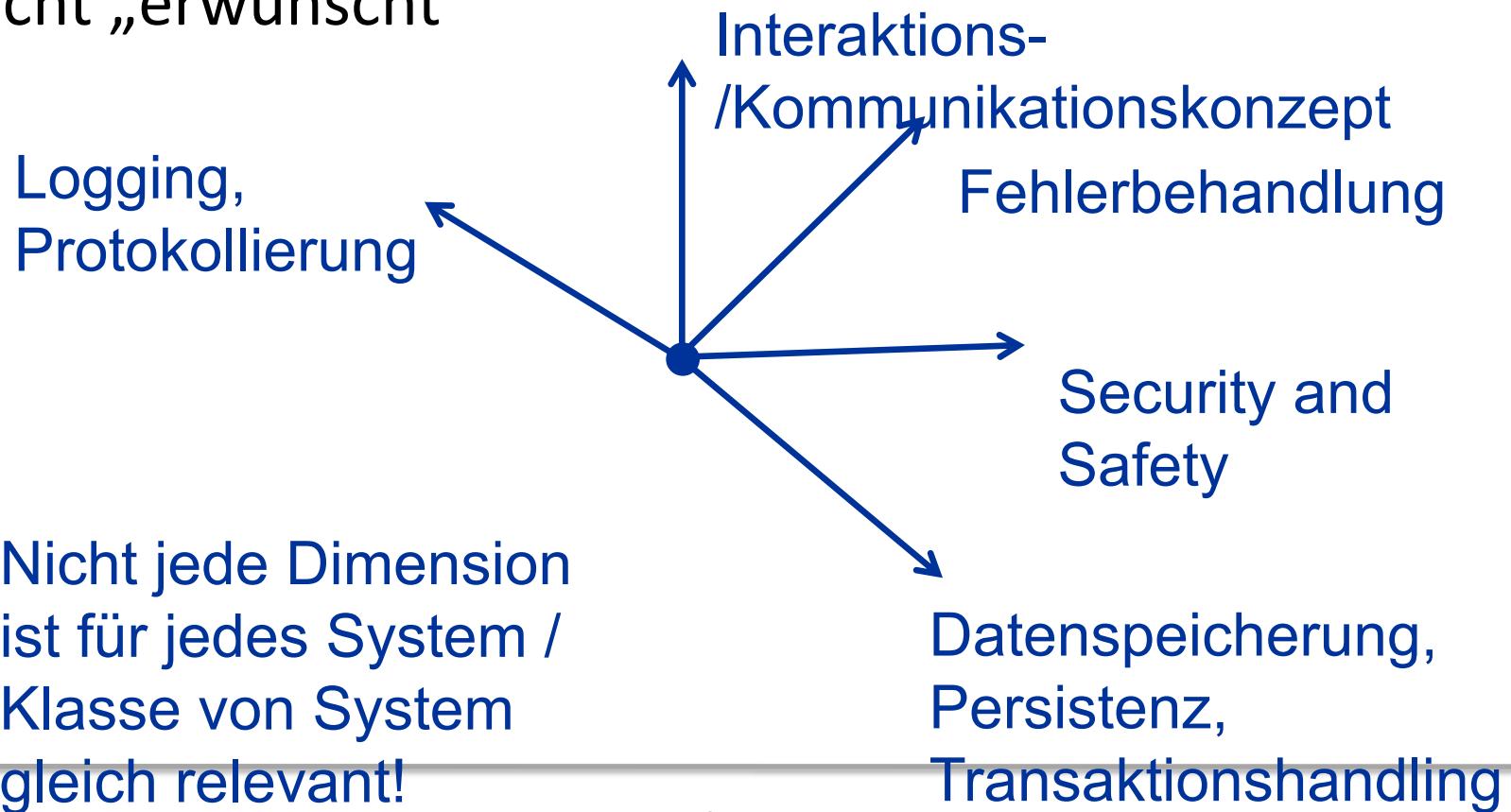


Beispiel: Fachliche Architekturebene (nach iSAQB)



Beispiel: Technische Architekturebene (1)

Querschnittliche Konzepte lassen sich meist nicht in einer der vorgestellten Sichten darstellen und/oder sind dort nicht „erwünscht“



Beispiel: Technische Architekturebene (2)

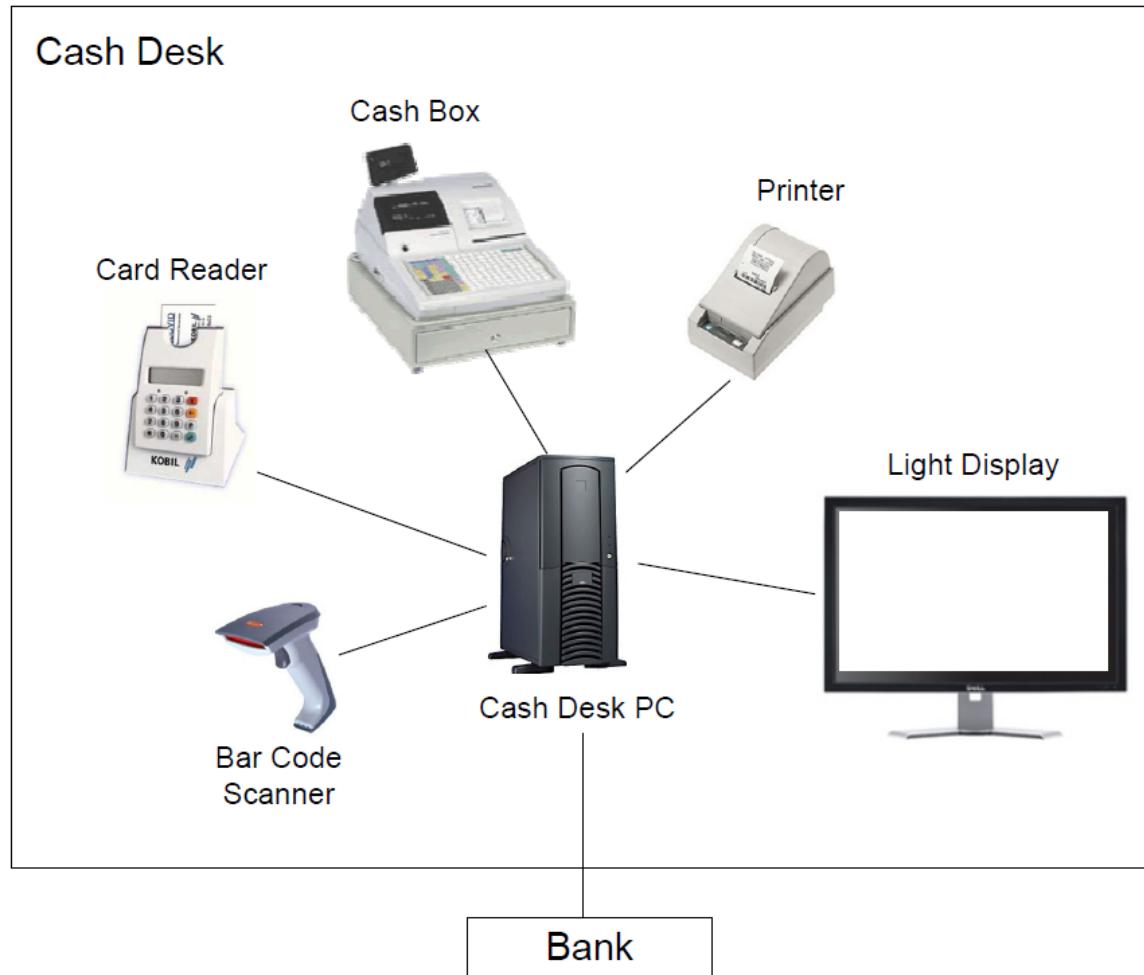
- Frage nach der Speicherung von Daten
- Typische Fragestellungen für den Architekten
 - Welche Ablageart?
 - Dateibasiert, Datenbanken, etc.
 - Bei Datenbanken: welche Art von Datenbank
 - Relational, objekt-orientiert, graphbasiert, Key-Value-Stores, etc.
 - Z.B. bei relationalen DB und objekt-orientierter Implementierung des Systems: technischer Zugriff?
 - Objekt-relationale Mapping-Frameworks (O/R-Mapper)
 - „Result-Set“-orientiert
 - Transaktionalität?
 - Nach welchem Prinzip werden Geschäftsprozesse in Transaktionen unterteilt

Inhalt

- Motivation und Herausforderungen
- Überblick und Konzepte
- Design Patterns
- Entwurf und Beschreibung
 - Analysemodell
 - Architekturmodell: Structural View
 - Architekturmodell: Deployment View
 - Architekturmodell: Behavioral View
 - Implementierungsmodell

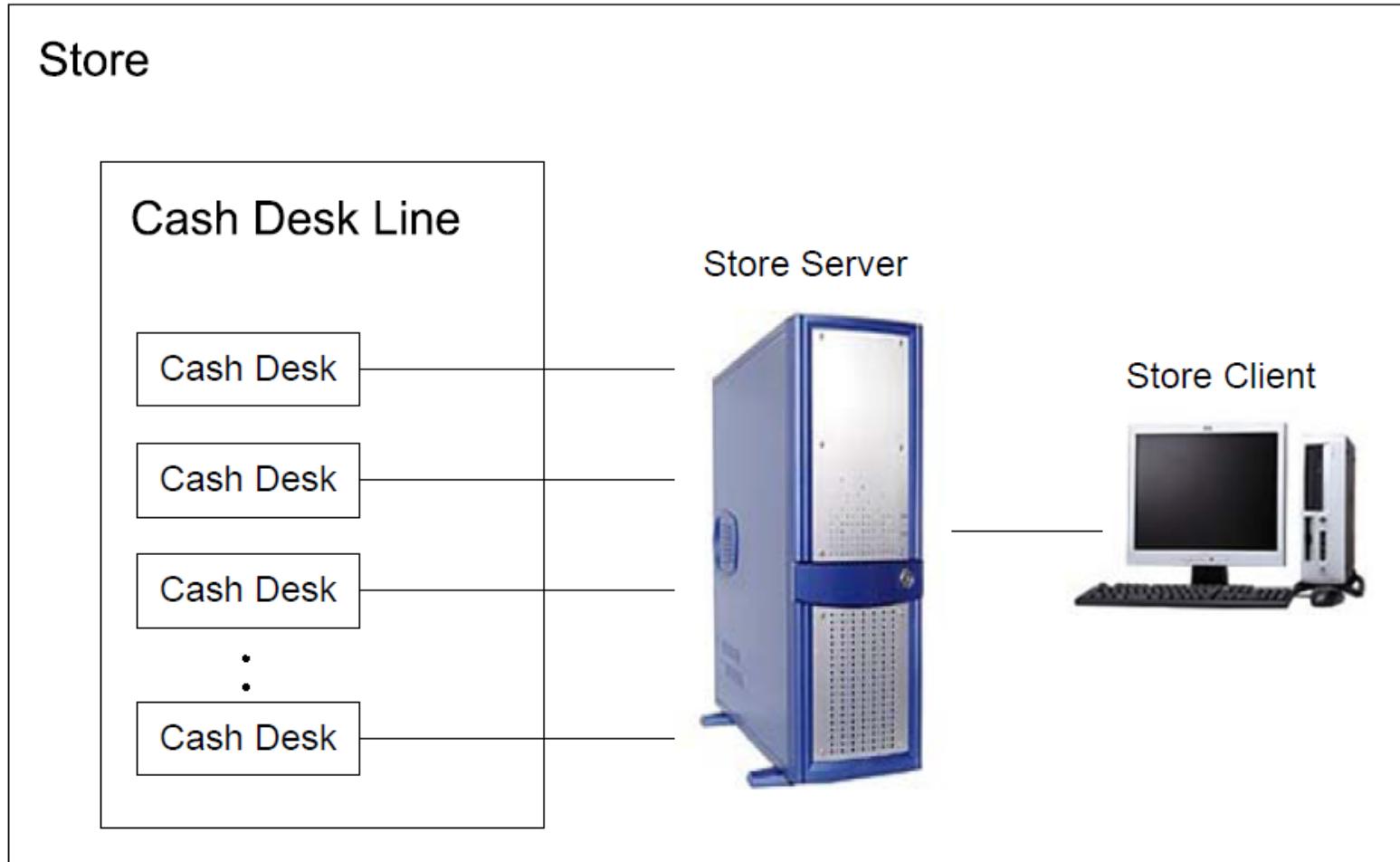
Analysemodell – System Overview (1)

= Context View



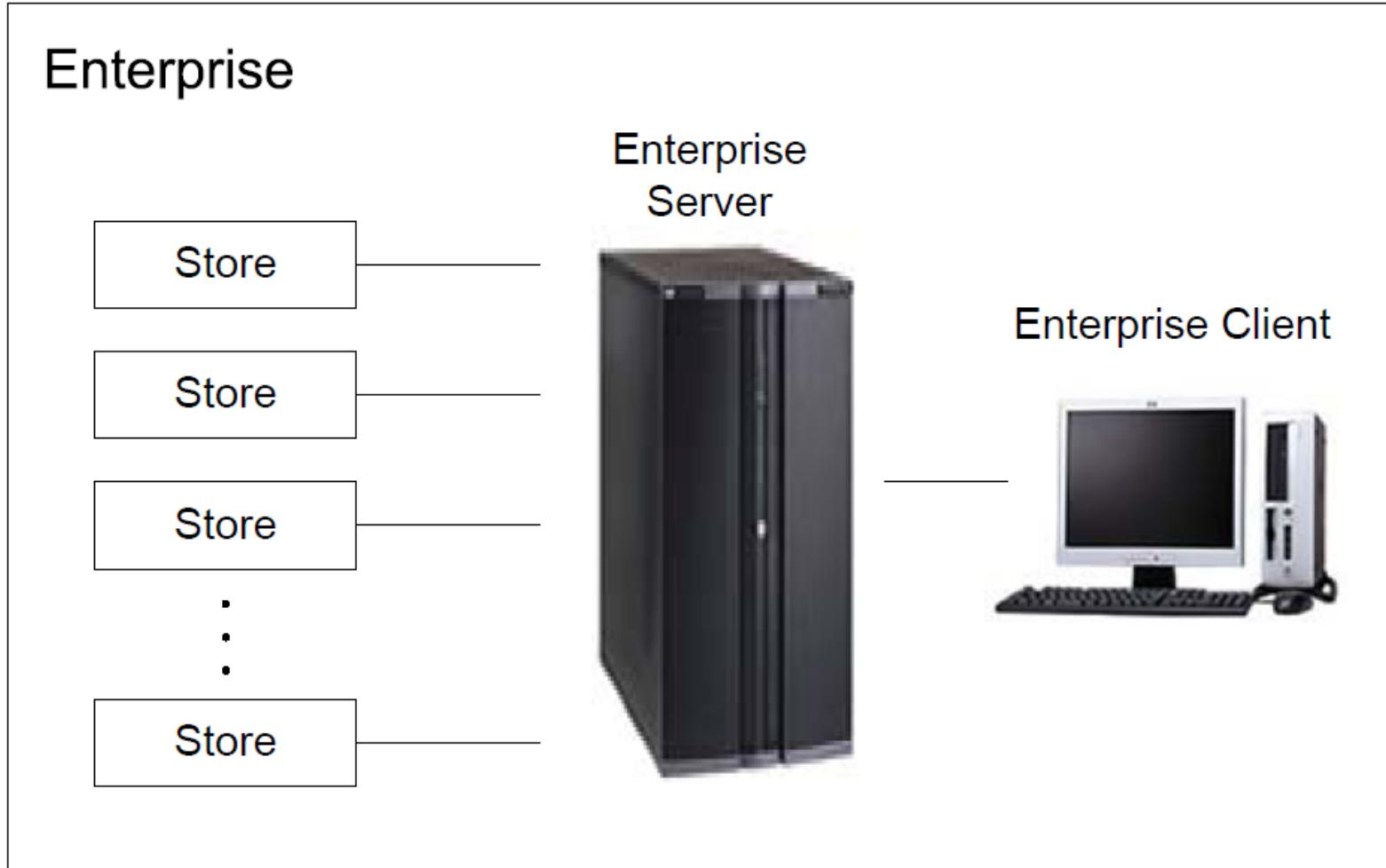
Analysemodell – System Overview (2)

= Context View

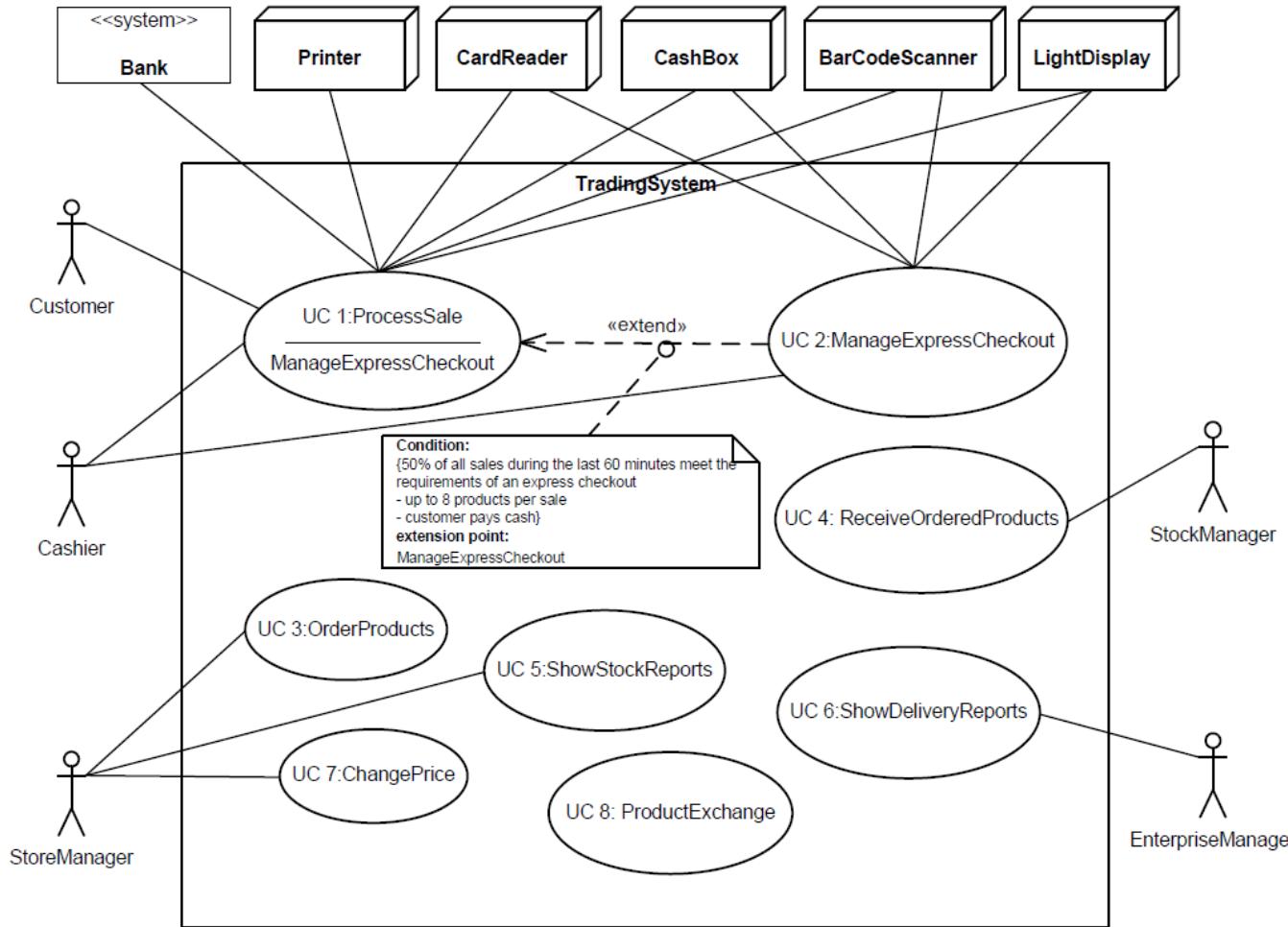


Analysemodell – System Overview (3)

= Context View



Analysemodell - Functional Requirements and Use Case Analysis (1) = Context View



Analysemodell - Functional Requirements and Use Case Analysis (2)

UC 1 - Process Sale

Brief Description At the Cash Desk the products a Customer wants to buy are detected and the payment - either by credit card or cash - is performed.

Involved Actors Customer, Cashier, Bank, Printer, Card Reader, Cash Box, Bar Code Scanner, Light Display

Precondition The Cash Desk and the Cashier are ready to start a new sale.

Trigger Coming to the Cash Desk a Costumer wants to pay his chosen product items.

Postcondition The Customer has paid, has received the bill and the sale is registered in the Inventory.

Analysemodell - Functional Requirements and Use Case Analysis (2)

Standard Process

1. The Customer arrives at the Cash Desk with goods to purchase. [arr1]
2. The Cashier starts a new sale by pressing the button *Start New Sale* at the Cash Box. [t12-1]
3. The Cashier enters the item identifier. This can be done manually by using the keyboard of the Cash Box [p13-1, t13-1] or by using the Bar Code Scanner [p13-2, t13-2].
4. Using the item identifier the System presents the corresponding product description, price, and running total. [t14-1]
The steps 3-4 are repeated until all items are registered. [n11-2]
5. Denoting the end of entering items the Cashier presses the button *Sale Finished* at the Cash Box. [t15-1]

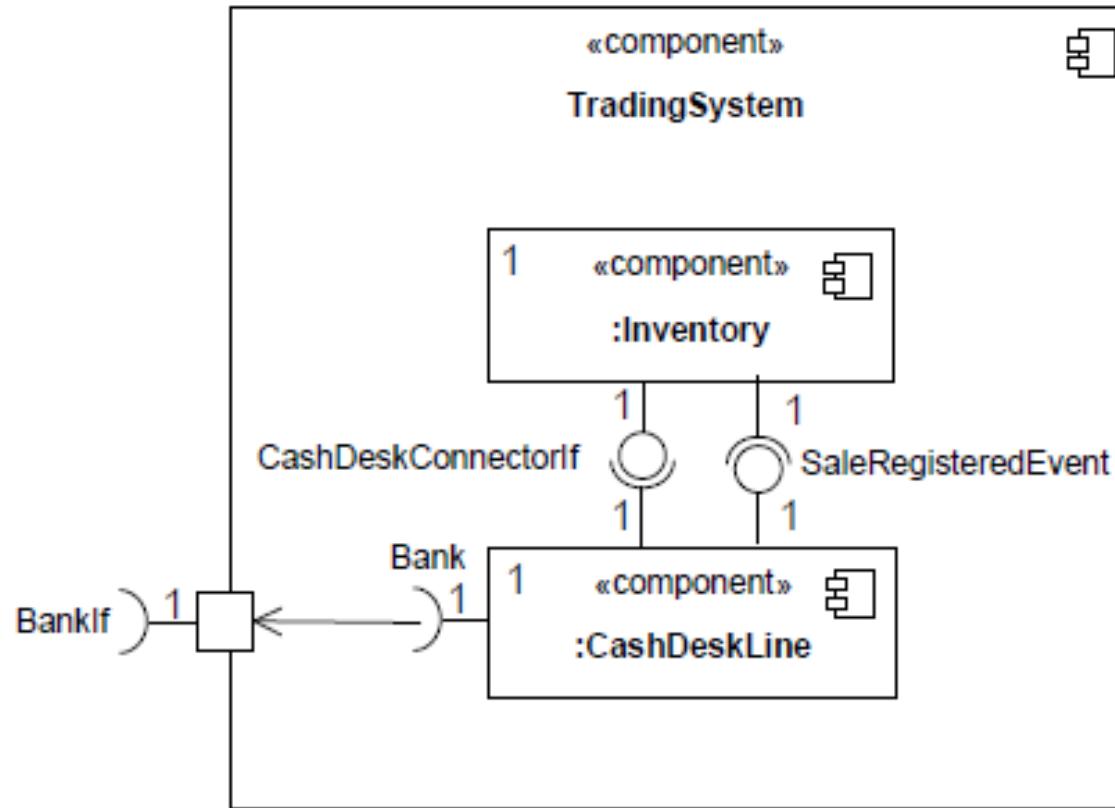
Analysemodell - Extra-Functional Properties

CoCoME Overall	
n0-1: Number of stores	200
n0-2: Cash desks per store	8
UC1 - Process Sale	
arr1: Customer arrival rate per store	PAopenLoad.PAoccurrence = ('unbounded', ('exponential', 320.0, 'hr'))
n11-1: Number of open cash desks per store	('assm', 'dist', ('histogram', 0, 0.1, 2, 0.2, 4, 0.4, 6, 0.3, 8, '#Open Cash Desks'))
n11-2: Number of goods per customer	('assm', 'dist', ('histogram', 1, 0.3, 8, 0.1, 15, 0.15, 25, 0.15, 50, 0.2, 75, 0.1, 100, '#Goods per Customer'))
t12-1: Time for pressing button "Start New Sale"	PAdemand = ('assm', 'mean', (1.0, 's'))
t13-1: Time for scanning an item	PAdemand = ('assm', 'dist', ('histogram', 0.0, 0.9, 0.3, 0.05, 1.0, 0.04, 2.0, 0.01, 5.0, 's'))
t13-2: Time for manual entry	PAdemand = ('assm', 'mean', (5.0, 's'))
t13-3: Time for signaling error and rejecting an ID	PAdemand = ('req', 'mean', (10, 'ms'))

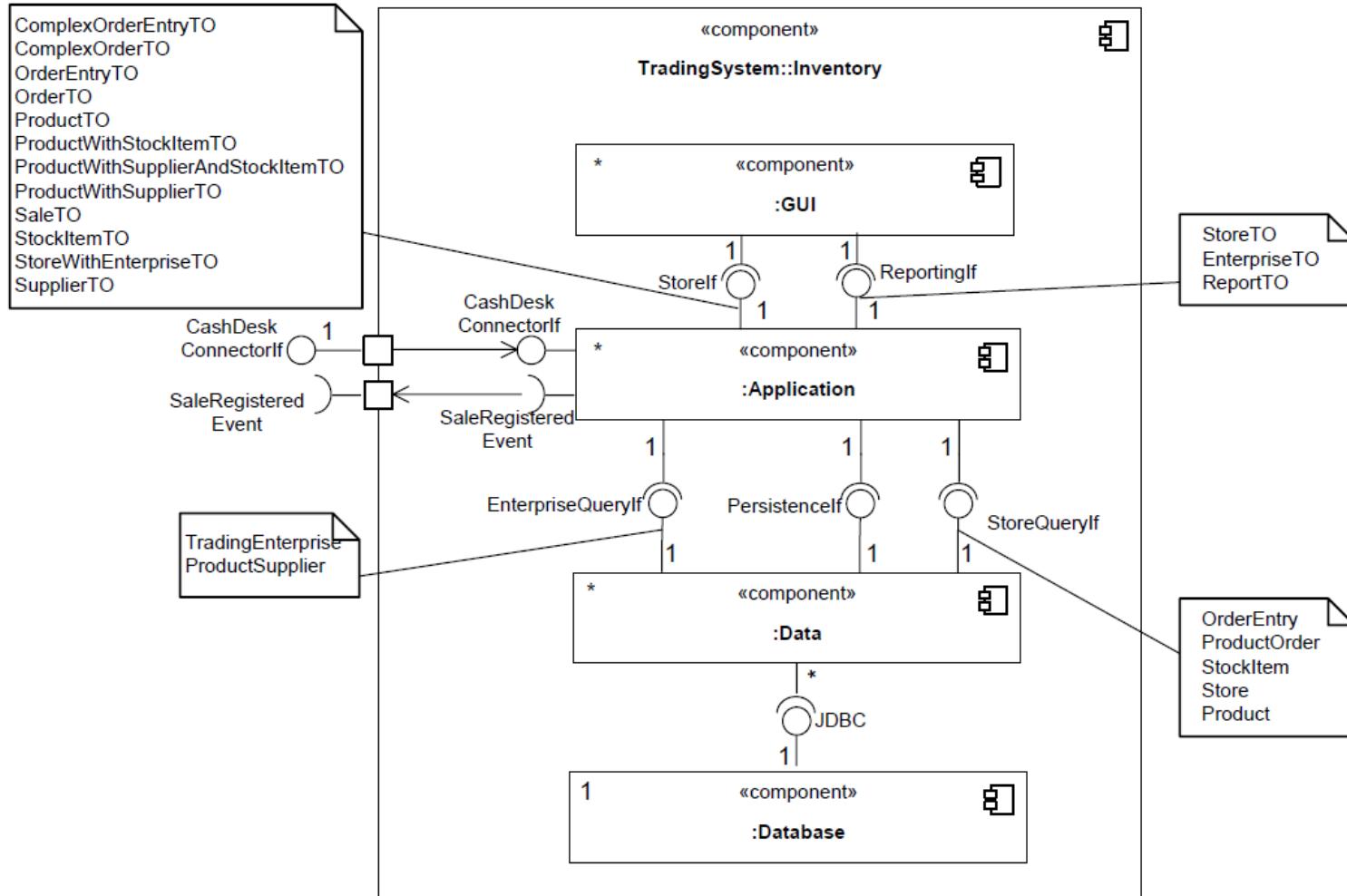
Inhalt

- Motivation und Herausforderungen
- Überblick und Konzepte
- Design Patterns
- Entwurf und Beschreibung
 - Analysemodell
 - Architekturmodell: Structural View
 - Architekturmodell: Deployment View
 - Architekturmodell: Behavioral View
 - Implementierungsmodell

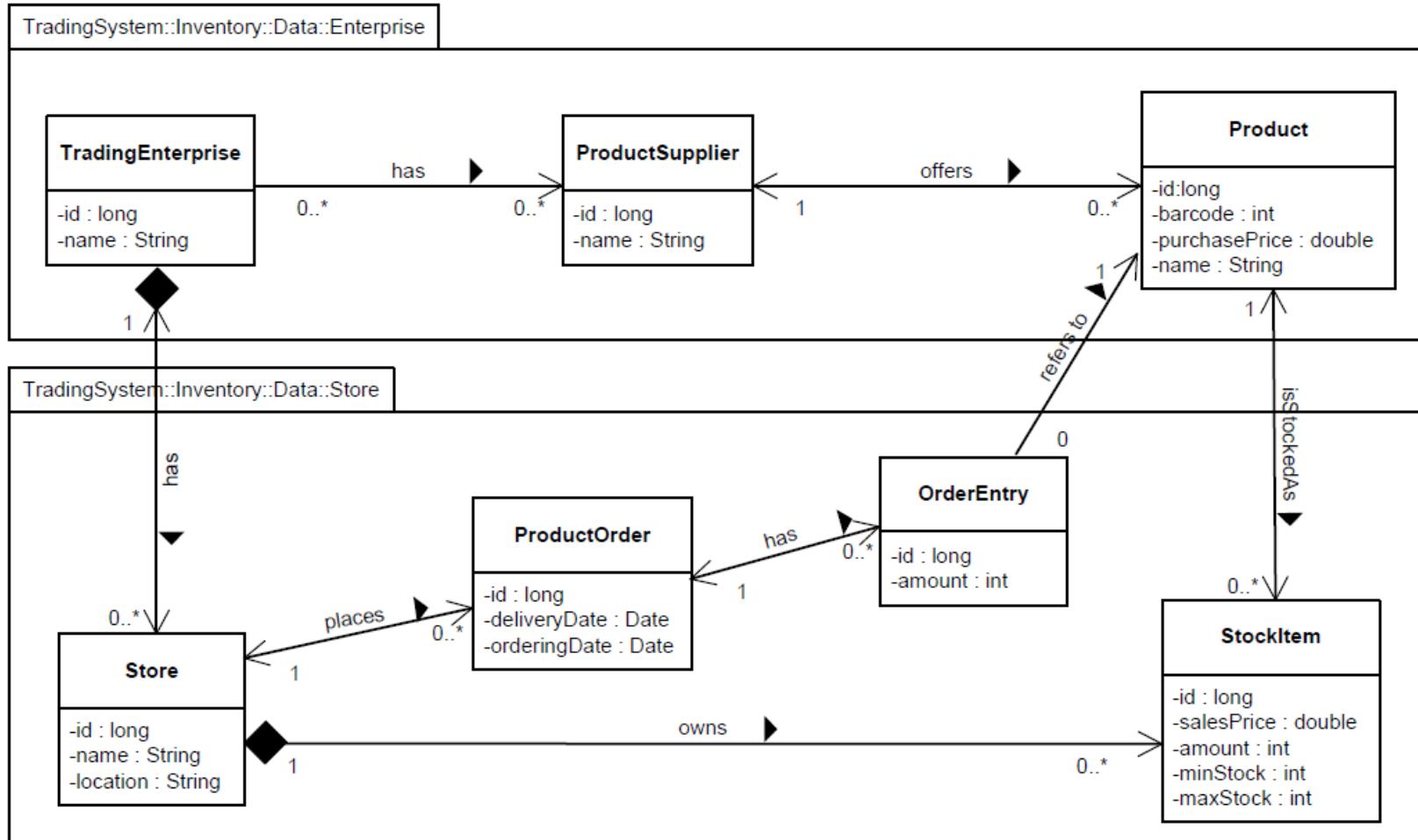
Architekturmodell – Structural View (1)



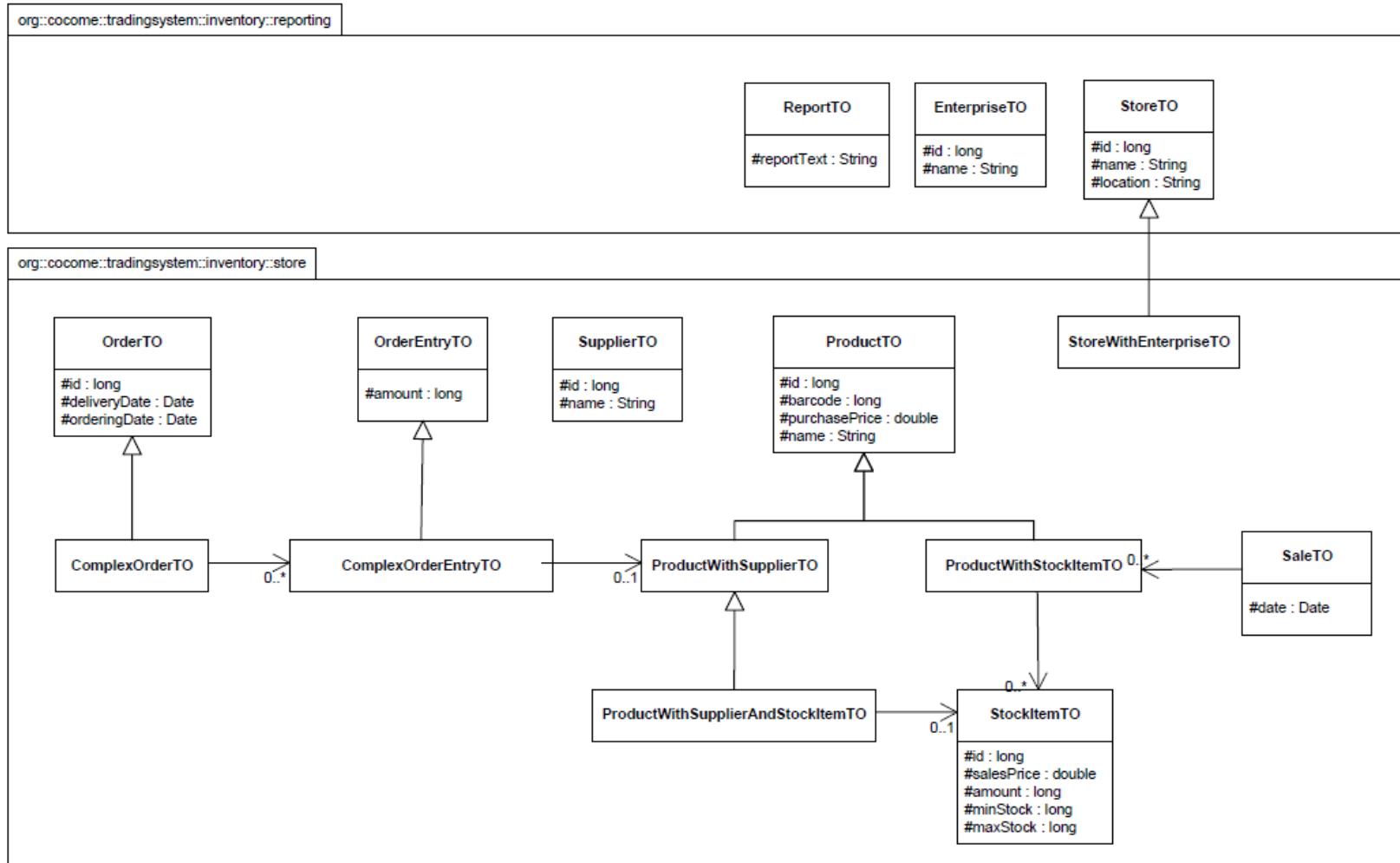
Architekturmodell – Structural View (2)



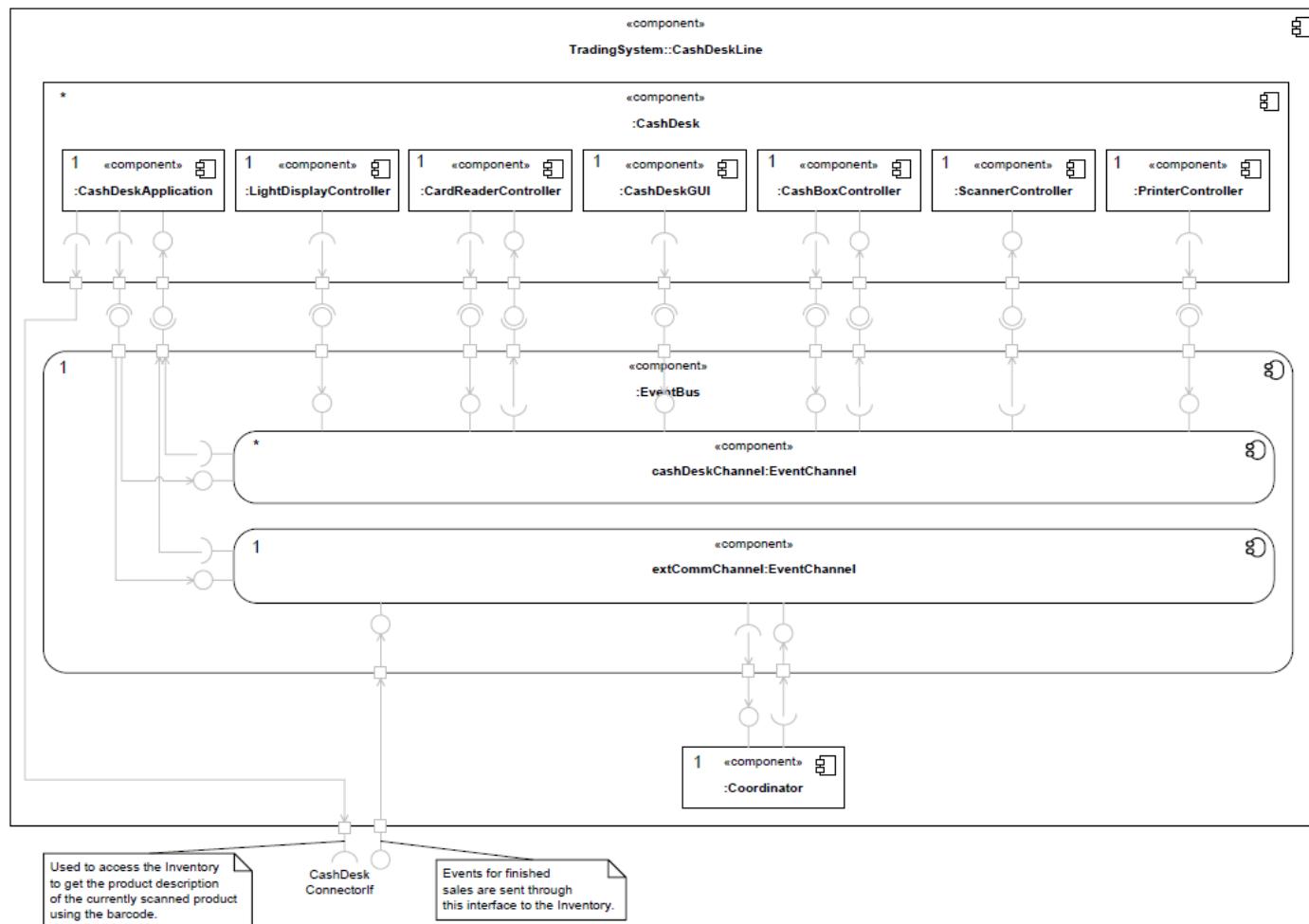
Architekturmodell – Structural View (3)



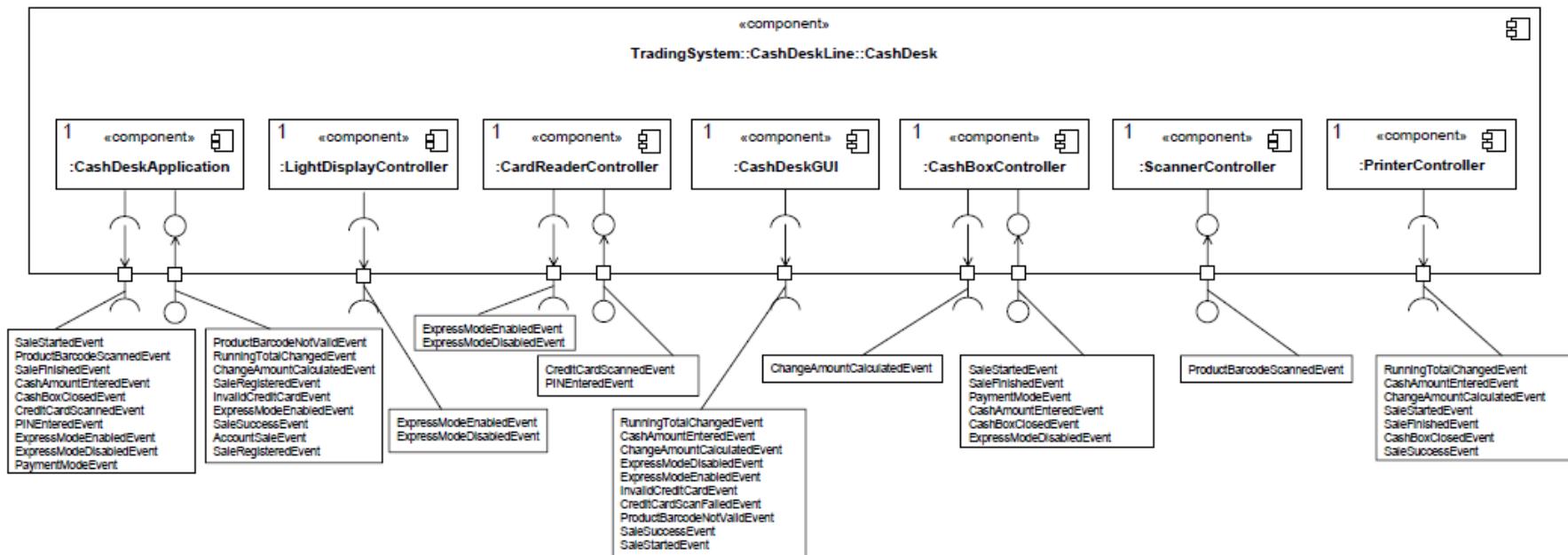
Architekturmodell – Structural View (4)



Architekturmodell – Structural View (5)



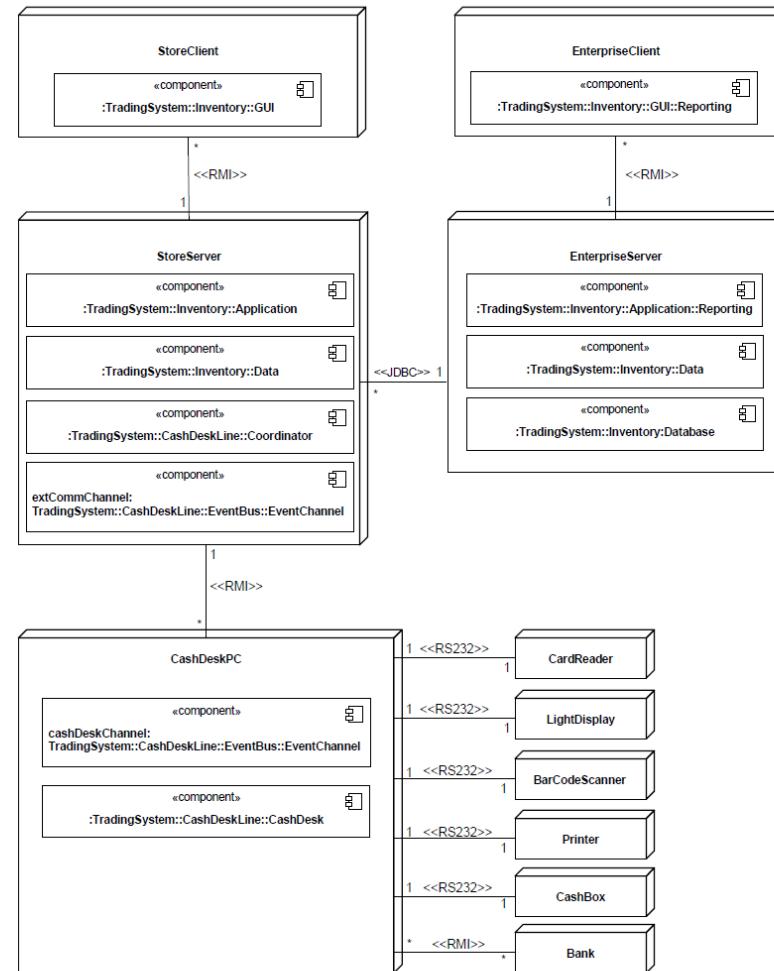
Architekturmodell – Structural View (6)



Inhalt

- Motivation und Herausforderungen
- Überblick und Konzepte
- Design Patterns
- Entwurf und Beschreibung
 - Analysemodell
 - Architekturmodell: Structural View
 - Architekturmodell: Deployment View
 - Architekturmodell: Behavioral View
 - Implementierungsmodell

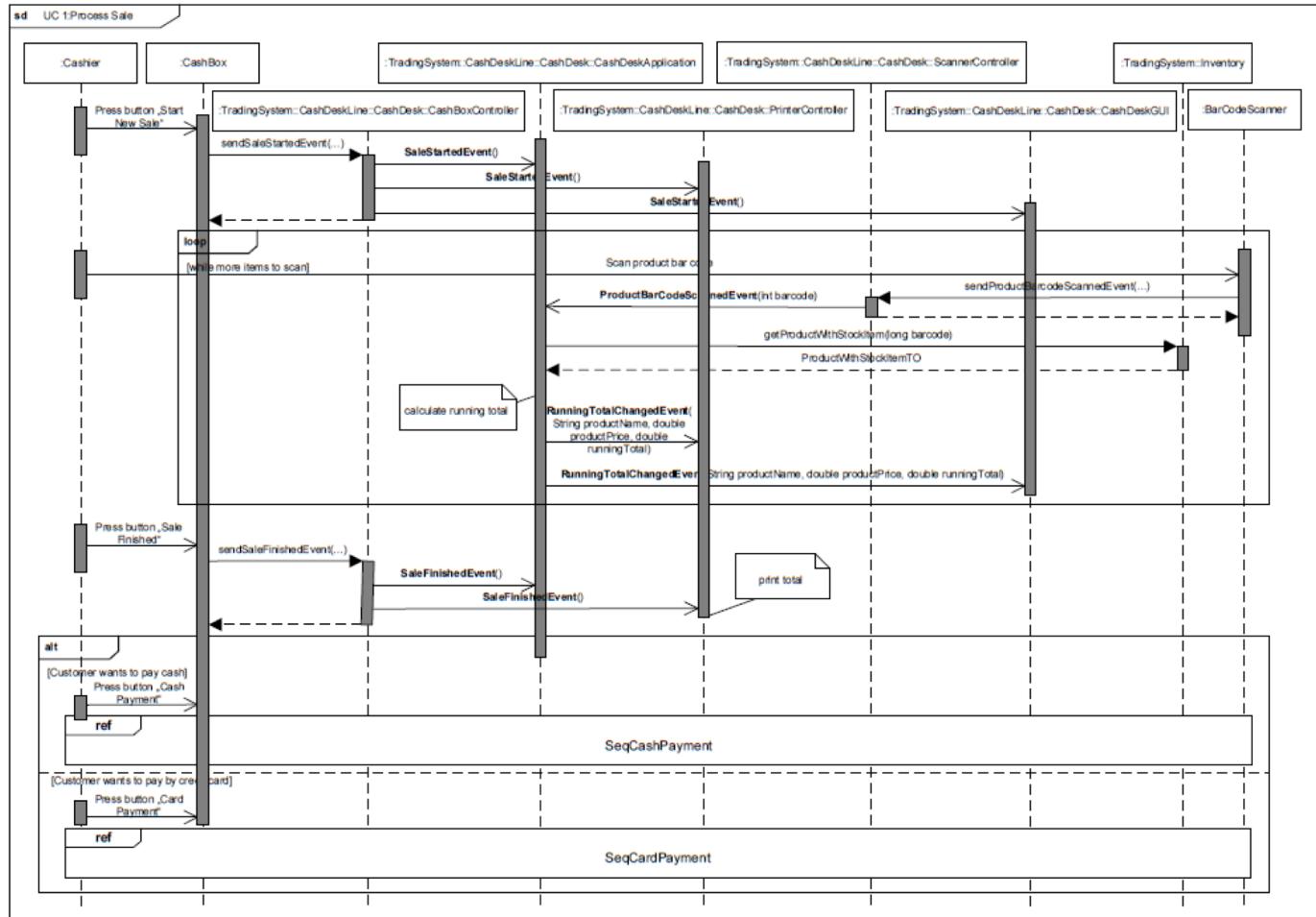
Architekturmodell – Deployment View



Inhalt

- Motivation und Herausforderungen
- Überblick und Konzepte
- Design Patterns
- Entwurf und Beschreibung
 - Analysemodell
 - Architekturmodell: Structural View
 - Architekturmodell: Deployment View
 - Architekturmodell: Behavioral View
 - Implementierungsmodell

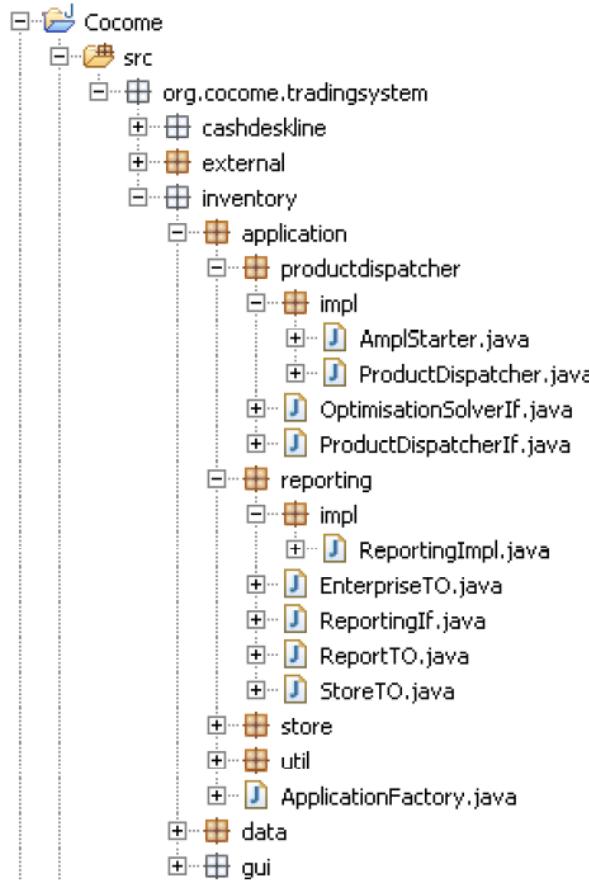
Architekturmodell – Behavioral View



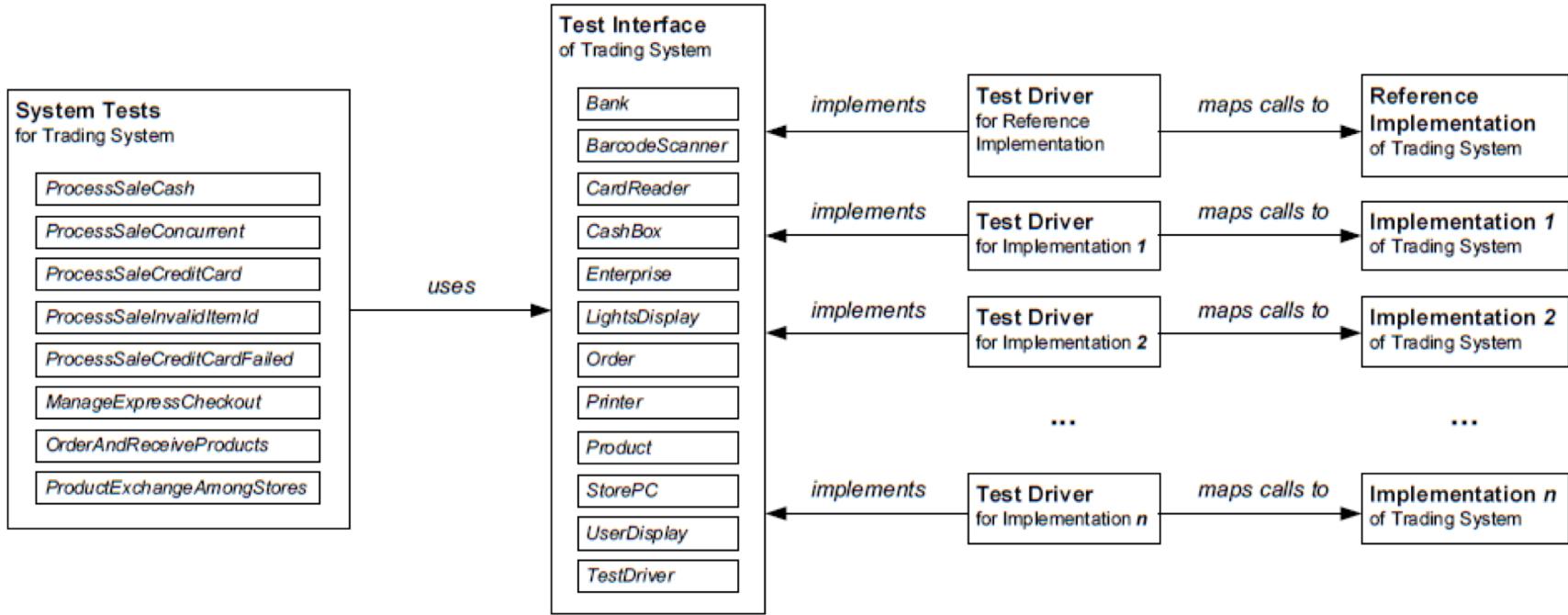
Inhalt

- Motivation und Herausforderungen
- Überblick und Konzepte
- Design Patterns
- Entwurf und Beschreibung
 - Analysemodell
 - Architekturmodell: Structural View
 - Architekturmodell: Deployment View
 - Architekturmodell: Behavioral View
 - Implementierungsmodell

Implementierungsmodell - Code-Design View



Implementierungsmodell - Test View



- Implementierungsmodell - Installations- und Start-View
 - Nur Textuelle: Siehe CoCoME Paper