
Einführung in die Softwaretechnologie

Wintersemester 2023 / 24

Dr. Günter Kriesel-Wünsche

Institut für Informatik III

Universität Bonn

gk@cs.uni-bonn.de

Alle Materialien auf Sciebo: <https://uni-bonn.sciebo.de/s/UnQAYYQOAfDopQ7>

Ihr Berufsbild in den Medien



a) Organisatorisches

Prüfungen

Tutorien

Voraussetzung zur Vorlesungsteilnahme

- BaPO 2011
 - ◆ Vorlesung Objektorientierte Softwareentwicklung (BA 024) bestanden
- BaPO 2019
 - ◆ Praktikum Objektorientierte Softwareentwicklung (BA 026) bestanden
- Nebenfächler
 - ◆ Was immer Ihre Prüfungsordnung bestimmt, ansonsten s.o.

- Voraussetzungen zur Prüfungszulassung
 - ◆ jeweils $\geq 50\%$ der Punkte in den theoretischen und praktischen Aufgaben
 - ◆ 3 mal die Lösung einer praktischen Aufgabe vorstellen
- Schriftliche Klausuren
 - ◆ Di, 06.2.2023, 14:30-16:00 Uhr (Einsicht am 15.2.2023, 9:30 Uhr, 1.047)
 - ◆ Di, 12.3.2023, 10:30-12:00 Uhr (Einsicht am 15.3.2023, 9:30 Uhr, 1.047)

Material

- eCampus-Kurs

https://ecampus.uni-bonn.de/goto.php?target=crs_3133453&client_id=ecampus

- ◆ Alle Organisatorischen Infos (Termine, Tutorien, ...) im dort verlinkten Wiki
- ◆ Inhaltliche Infos (s.u.) in Sciebo (ebenfalls dort verlinkt)
- ◆ Passwort: swtWS2023-24

- Sciebo

<https://uni-bonn.sciebo.de/s/UnQAYYQOAfDopQ7>

- ◆ Registrierung unter sciebo.de mit <uni-id>@uni-bonn.de
- ◆ Empfohlen: Client-Download
 - ⇒ Automatische Synchronisation (wie DropBox)
- ◆ Materialien
 - ⇒ Vorlesungsfolien
 - ⇒ Vorlesungsvideos
 - ⇒ Übungsblätter & Lösungen

Tutorien

- Warum

- ◆ Lernen was man in der Vorlesung nicht zeigen kann
- ◆ Praktische Anwendung des Vorlesungsstoffes
- ◆ Feedback zu **Ihren** Lösungen
- Prüfungsvorbereitung!

- Was

- ◆ *Wir*: Systeme vorführen
- ◆ *Sie*: Lösungen vorführen
- ◆ Miteinander sprechen ;-)

- Wann

- ◆ Veröffentlichung der Aufgaben:
 - ⇒ jeweils Freitag Abend
 - ⇒ Erstmals am 20.10.2023
- ◆ Abgabe der Lösungen:
 - ⇒ jeweils nächsten Freitag 16⁰⁰
- ◆ Tutorien beginnen übernächste Woche (ab Mo, 30.10.2023)

- Wie

- ◆ Abgabe via Git
- ◆ 1 Tutor ⇔ 5 Teilnehmer
- ◆ 1 Stunde

Anmeldung zu Tutorien

1. Online-Anmeldung

- ◆ <https://puma.cs.uni-bonn.de/>
- ◆ Max. 3 Gruppen / Zeiten wählen
- ◆ Prioritäten 1 bis 3
- ◆ Bis Mittwoch, 18.10.2023, 12:00 Uhr

Nur für Teilnehmer:innen,
die **keine**
Klausurzulassung haben

2. Zuteilungsbenachrichtigung von eurem Tutor

- ◆ per mail bis Mittwoch, den 25.10.2023
- ◆ incl. Infos mit wem Sie in einer Gruppe sind und URL ihres Git-Repos

3. Anleitung für Gruppenwechsel

- ◆ Siehe „Tutorien“-Seite im Vorlesungswiki auf eCampus
https://ecampus.uni-bonn.de/goto.php?target=crs_3133453&client_id=ecampus
- ◆ ECampus-Beitritts-PW für Teilnehmer: **swtWS2023-24**

Erneut teilnehmen? → Mail an
swt-tutoren@lists.iai.uni-bonn.de

Kommunikation

- Sie an uns:

- ◆ <https://lists.iai.uni-bonn.de/mailman/admin.cgi/swt-tutoren>
- ◆ Alles was Sie an Fragen, Hinweisen, Kommentaren haben
 - ⇒ insbesondere auch Wünsche nach Gruppenwechsel (siehe Anleitung im Wiki!)
- ◆ Bitte an die Liste, nicht direkt an mich
 - ⇒ Es sind sechs Leute, die antworten können!
 - ⇒ Sie bekommen sehr viel schneller Antwort!

- Wir an Sie:

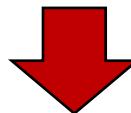
- ◆ <https://lists.iai.uni-bonn.de/mailman/admindb.cgi/swt-vorlesung>
- ◆ Termine (und kurzfristige Terminänderungen)
- ◆ Hinweise auf Materialien
- ◆ Organisatorisches
- ◆ Inhaltliches

b) Inhalts-Überblick

Was ist Softwaretechnologie / Software Engineering?

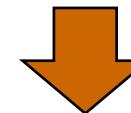
Kosten

- Softwarekosten übersteigen oft Hardwarekosten
- Softwarewartung übersteigt Entwicklungskosten → Bis zu 80% der Gesamtkosten



Qualität

- Schlechte Software schadet
 - ◆ Direkte Schäden
 - ◆ Unzufriedene Benutzer
 - ◆ ...



Kosteneffektive Entwicklung von qualitativ hochwertiger Software

Herausforderungen

Komplexität

- Viele **Teilnehmer** ▶ Komplexität der Kommunikation
- Viele **Ziele** ▶ Interessenskonflikte
- Viele **Funktionen** ▶ Komplexität des Problems
- Viele **Komponenten** ▶ Komplexität der Zusammenstellung

Keine Einzelperson kann ein ganzes System verstehen

Änderung

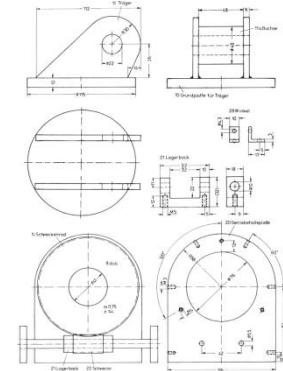
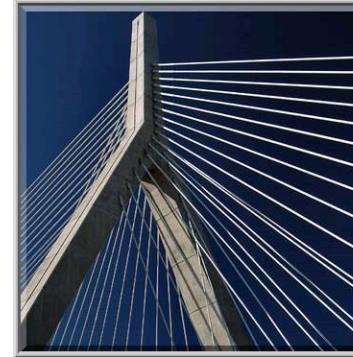
- **Welt** ▶ Gesetzgebung, Geschäftsabläufe, ...
- **Modellierter Ausschnitt** ▶ mehr Funktionalität, Systeme integrieren, ...
- **Implementierungstechnologie** ▶ neue Sprachen, Komponenten, ...
- **Team** ▶ Personen gehen, neue kommen hinzu, ...
- **Ziele** ▶ neue Geschäftsausrichtung, ...

Software ist ständiger Änderung unterworfen!

Andere Ingenieurwissenschaften

Domäne

- Klar definierte Probleme
- Ein Produkt muss gebaut werden
- Hohe Qualitätsanforderungen



Methoden

- Systematische Verfahren und ihre disziplinierte Anwendung
- Standardschnittstellen, -komponenten und -prozesse
- Wissen um verfügbare Komponenten
- Empirische Methoden zur Bewertung von Lösungen



Software Engineering ist anders

Andere Ingenieursbereiche

- Herstellung bestimmt die Endkosten
- Änderungen sind sehr teuer
 - ◆ Auswirkungen werden genau analysiert
- Änderungsanforderungen sind ziemlich selten
 - ◆ 2000 Jahre von der Euklidischen zur nicht-Euklidischen Geometrie

Software Engineering

- Herstellung ist eine einfache Duplizierung
 - ◆ Verführung, schnell zu ändern 
- Änderungen sind einfach
 - ◆ Auswirkungen werden nicht ausreichend bedacht 
- Änderungsanforderungen geschehen dauernd
 - ◆ Manchmal innerhalb von Minuten (Kunde hat nach Ihrer Präsentation seine Meinung geändert)

Software Engineering ▶ Vorlesungsthemen

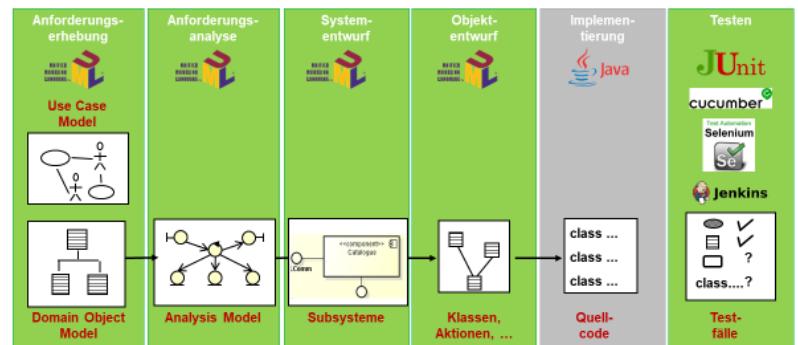
● PROZESSE

- ◆ Wie organisieren wir das ganze?



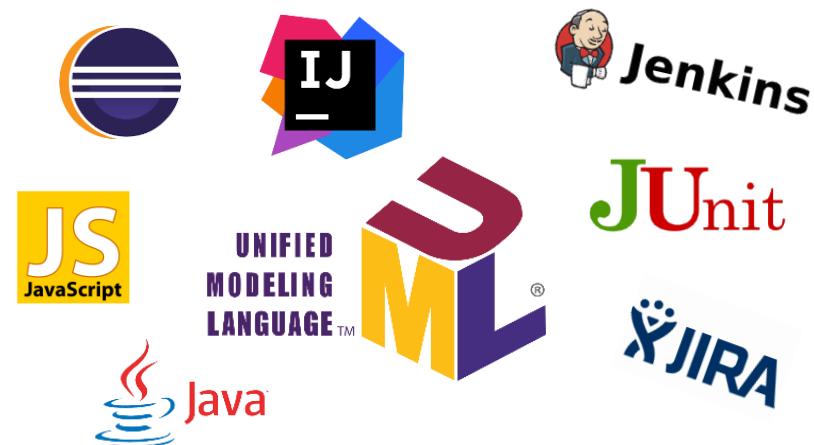
● AKTIVITÄTEN

- ◆ Was tun wir damit?



● WERKZEUGE

- ◆ Konzepte, Sprachen, Systeme



Themenbereiche und Vorlesungs-Kapitel

III. PROZESSE

12

Agile Softwareentwicklung

11

Software-Prozess-Modelle

II. AKTIVITÄTEN

10

Test

9

Objekt-Design

8

System-Design

7

Anforderungs-Analyse

6

Anforderungs-Erhebung

I. WERKZEUGE

5

Refactoring

4

Entwurfsmuster

3

Unified Modelling Language (UML)

2

OO-Modellierung

1

OO-Programmierung ++

VORWISSEN

Software Configuration Management (mit Git)

OO-Programmierung Grundlagen

Imperative-Programmierung

Ausblick

Nachfolgeveranstaltungen

Projektgruppe „Angewandte Softwaretechnologie“

- Ziel
 - ◆ Praktische Erfahrung mit Agiler Softwareentwicklung (Siehe Kapitel 12)
- Aufgaben
 - ◆ an der Schnittstelle von Software Engineering und Maschinellem Lernen
 - ◆ Co-Betreuung durch unsere Doktoranden
- Selbstorganisierte Teams
 - ◆ jede(r) ist 3 Wochen lang TeamleiterIn
 - ◆ Veranstalter sind nur Berater oder Kunden
- Maximal drei 4-er-Teams

Weiteres zum Semesterende im Vorlesungswiki (eCampus)

Kapitel 1. Objektorientierte Programmierung (OOP)

Stand: 30.10.2023

Bis Ende (incl. Anhang) überarbeitet.
Finale Version!

Welche Arten von OO-Sprachen gibt es?

Was sind die Konzepte die OO-Sprachen ausmachen?

Was sind Typen und wie beeinflussen statische Typen die Semantik der Sprache?

Statische und dynamische Semantik

Implementierungsaspekte

Umfrage: Was ist OOP?

- Was ist ein Objekt?
- Wie entstehen Objekte?
- Kann es Objekte geben, ohne dass es Klassen gibt?
- Kann es Vererbung geben, ohne dass es Klassen gibt?
- Können Variablenwerte zur Laufzeit verändert werden?
- Kann Methodencode zur Laufzeit verändert werden?
- Kann die Struktur eines Objektes zur Laufzeit verändert werden?

Kapitel-Überblick

- 1.** Prototyp-basierte OOPS – am Beispiel “JavaScript”
- 2.** Was also ist OOP?
- 3.** Klassen-basierte OOPS – am Beispiel “Java”
- 4.** Typen – am Beispiel “Java”
- 5.** Implementierung, klassenbasiert – am Beispiel “Java”
- 6.** Generische Typen – am Beispiel “Java”

1.1

Prototyp-basierte OOP – in JavaScript

Allgemeine Konzepte und ihre Ausprägung in JavaScript

Objekterzeugung → „Ex nihilo“ und Cloning

Properties / Slots → Variablen und Methoden

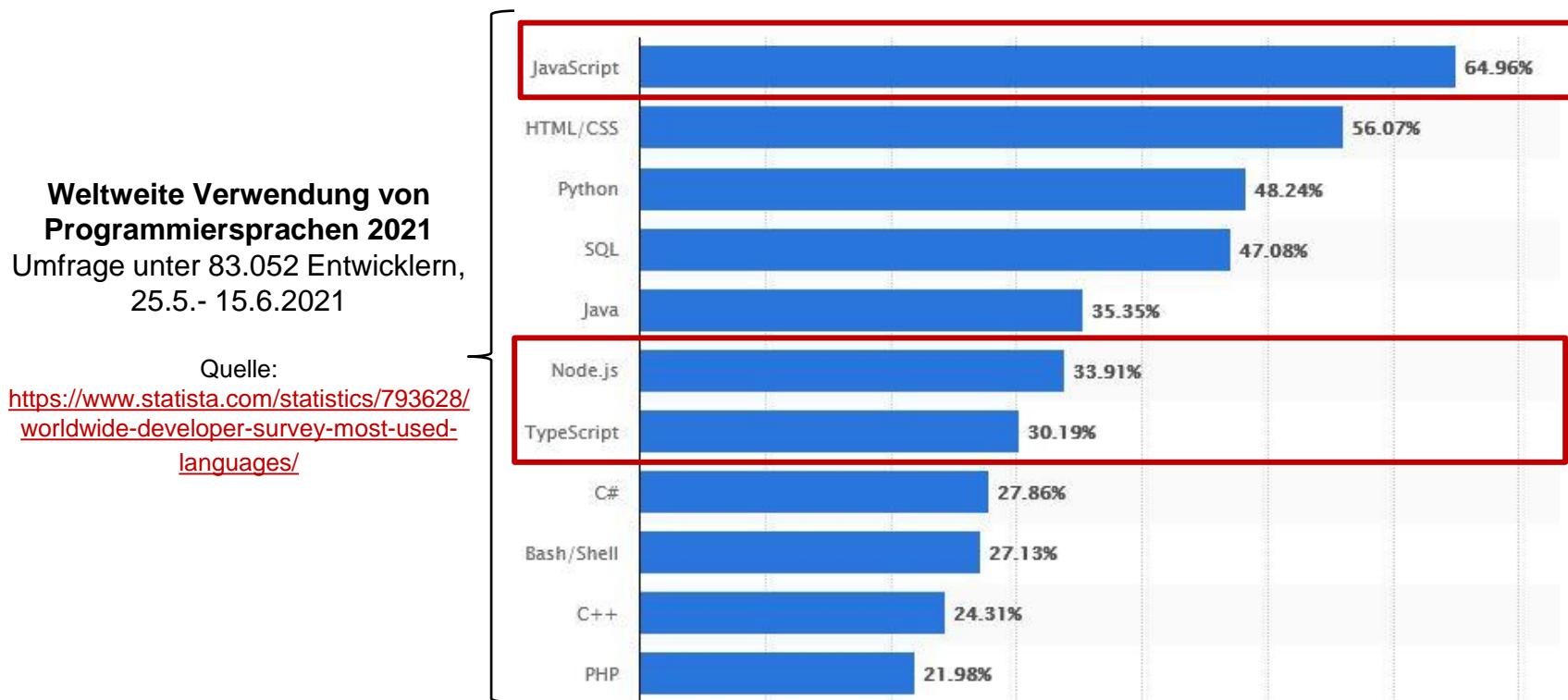
Dynamik → Modifikation zur Laufzeit

Gemeinsamkeiten → Delegation

Simulation von Klassen

JavaScript

- JavaScript ist **die** Sprache für dynamische Webseiten
 - ◆ Von allen Browser-Anbietern unterstützt und weiter standardisiert



- Verstärkt auch im Bereich Backend/Server eingesetzt (eingebettet in node.js)

JavaScript

- Designer (1997)
 - ◆ Brendan Eich (damals bei Netscape, später Mitbegründer und Chief Technology Officer von Mozilla)
 - ◆ Guy L. Steele Jr. (Co-Autor der Java Language Specification)

- Namen
 - ◆ Ursprünglich “Mocha”
 - ◆ Dann “LifeScript”
 - ◆ Dann “JavaScript”
 - ◆ Als “ECMA-Script” standardisiert



Ex-Nihilo-Objekterzeugung

- „Ex nihilo“ (lat.) = „Aus dem Nichts“
- Ein Objekt wird einfach „hingeschrieben“.
 - ◆ Die Syntax dafür wird in JavaScript als „Object Literal“ bezeichnet
- Man braucht keine Klassen und keine Instanziierung!!!

```
var emptyObject = {}; // empty object

var bond = {           // object with properties
  firstName : "James",
  lastName : "Bond",
  fullName : function () {
    return this.firstName + ' ' + this.lastName
  }
};
```

Properties

- Javascript Properties sind Paare von Namen und Werten
 - ◆ Key:Value-Notation
- Der Value-Teil kann eine Funktionsdefinition sein!
 - ◆ Properties sind also sowohl Variablen als auch Methoden

```
var emptyObject = {}; // empty object

var bond = {           // object with properties
  firstName : "James",
  lastName : "Bond",
  fullName : function () {
    return this.firstName + ' ' + this.lastName
  }
};
```

Nachrichten

- Nachricht = dynamisch gebundener Funktionsaufruf

```
var emptyObject = {}; // empty object

var bond = {           // object with properties
    firstName : "James",
    lastName : "Bond",
    fullName : function () {
        return this.firstName + ' ' + this.lastName
    }
};
```

bond.fullName(); // Liefert: "James Bond"

↑ ↑
Empfänger Nachricht

- Nachricht = dynamisch gebundener Funtionsaufruf

Was heisst "binden"?

- Allgemein
 - ◆ Abbildung eines Funktionsaufrufs (= abstrakte Operation) auf eine Methode (= ausführbarer Code)
- Dynamisches Binden
 - ◆ Methodenbestimmung zur Laufzeit
 - ◆ Objekt hat Map / Dictionary von Key : Value – Paaren
 - ⇒ Key ist der Name der Operation
 - ⇒ Value ist die Implementierung (der Code)
 - ◆ Wenn es keinen Eintrag für die aufgerufene Methode gibt, wird zur Laufzeit ein Fehler gemeldet
 - ⇒ Sofern es kein statisches Typsystem gibt (mehr dazu später)

Unterschied Property-Zugriff versus Funktionsaufruf

- Der Wert einer Property wird durch ihren Namen abgefragt
 - Auch bei Properties, die Funktionen sind.
 - Da ist der Property-Wert die Funktion!

Property-Zugriff

```
console.log(`My name is ${bond.fullName}.`);  
// --> My name is function () {  
//       return this.firstName + ' ' + this.lastName  
//     }.
```

```
console.log(`My name is ${bond.fullName()}.`);  
// --> "My name is James Bond."
```

Funktions-Aufruf

- Um eine Property deren Wert eine Funktion ist aufzurufen, müssen wir die **Argumentliste** hinter den Namen schreiben
 - Auch wenn sie leer ist

Werte sind Referenzen auf Objekte

- Variablen und Properties enthalten Referenzen auf Objekte
- Objekte sind im „Heap“ des Programms gespeichert, nicht in den Variablen.

Die Variable "bond" zeigt auf das Objekt, das durch das zweite Objektliteral erzeugt wurde.

Wir sagen allerding "das bond-Objekt", um uns einfacher darauf zu beziehen, wissend, dass das eine bewusste Vereinfachung ist.

```
var emptyObject = {}; // empty object
var bond = {
    firstName : "James",
    lastName : "Bond",
    fullName : function () {
        return this.firstName + ' ' + this.lastName
    }
};
var james = bond;
```

Ein von verschiedenen Variablen referenziertes Objekt hat verschiedene Namen. Wir sagen „james“ ist ein Alias von „bond“, also ein anderer Name des gleichen Objekts.

Objekte können wachsen

- Zur Laufzeit können neue Properties hinzugefügt werden
- ... durch Zuweisung an eine neue Property

```
james.age = 35 ; // bond hat nun auch ein Alter
```

- ... in einem Objekt das schon vorher existierte:

```
var emptyObject = {} ; // empty object
var bond = {           // object with properties
    firstName : "James",
    lastName : "Bond",
    fullName : function () {
        return this.firstName + ' ' + this.lastName
    }
};
var james = bond;
```

Nun ergibt das Leere Objekt Sinn

- Wir können ein Objekt für James Bond auch wie folgt erzeugen:
 - ◆ Erst ein leeres Objekt
 - ◆ Dann alle „properties“ schrittweise hinzufügen ...

```
var bond = {};// empty object  
  
bond.firstName = "James";  
bond["lastName"] = "Bond";} alternative  
Syntaxformen zur  
Objekt-Erweiterung  
  
bond.fullName = function () {  
    return this.firstName + ' ' + this.lastName  
};↑
```

- ◆ ... auch die Methoden!
 - Methoden sind Objekte die die Nachricht `call(obj, args)` verstehen

Delegation: Objektbasierte Vererbung

- Jedes Objekt hat ein „Elternobjekt“ von dem es erbt
 - ◆ Dies wird von einer ausgezeichneten Property referenziert
 - ◆ In JavaScript heißt sie „__proto__“
- Delegation = Nachrichten, die von einem Objekt nicht verstanden werden, werden unverändert an sein Elternobjekt weitergeleitet
 - ◆ unverändert = „this“ ist immer noch der ursprüngliche Empfänger!
- Falls das Elternobjekt die Nachricht versteht beantwortet es sie:

```
var seanConnery = {};  
  
seanConnery.__proto__ = bond;  
seanConnery.fullName(); // "James Bond"
```

„bond“ ist nun das
Elternobjekt von
„seanConnery“

"seanConnery" hat die
unbekannte Nachricht an
"bond" delegiert

Delegation aller Aufrufe (auch von Property-Zugriffen!)

- Der Property-Zugriff ist ein Methoden-Aufruf

- Daher kann er ebenfalls delegiert werden

- Der Compiler erzeugt für jede Property automatisch Zugriffsmethoden

Delegierter Property-Zugriff

Zugriffsmethoden

```
console.log(`My name is ${seanConnery.fullName}. `);  
// --> My name is function () {  
//       return this.firstName + ' ' + this.lastName  
//     }.
```

```
console.log(`My name is ${seanConnery.fullName()}. `);  
// --> "My name is James Bond."
```

Delegierter Funktionsaufruf

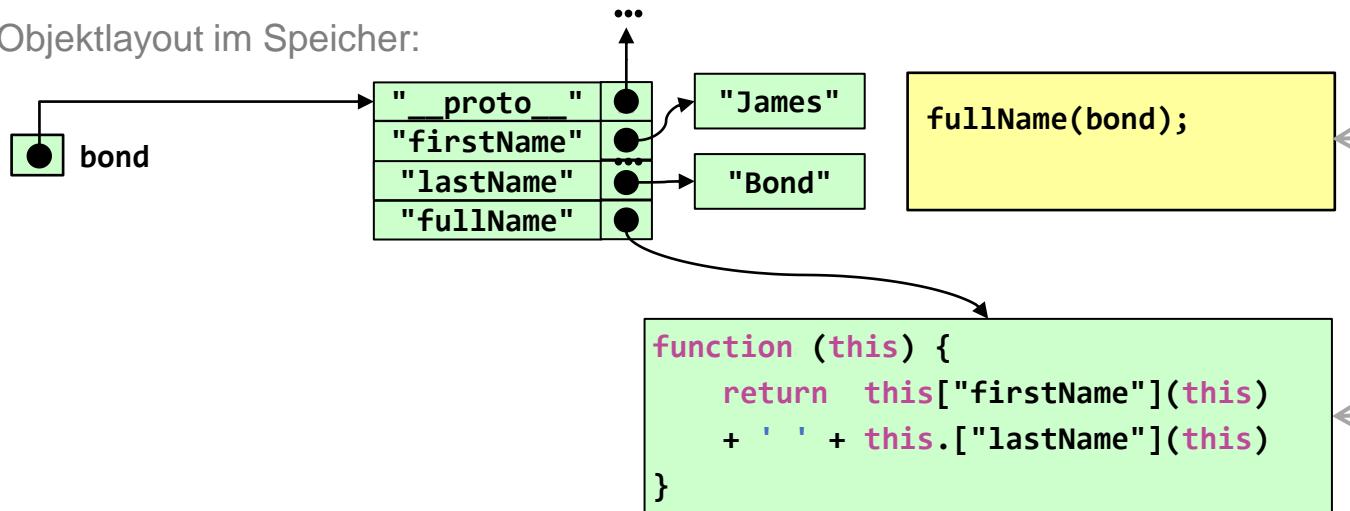
- Es werden immer alle lokal undefinierten Aufrufe delegiert, einschließlich der Aufrufe von Zugriffsmethoden!
 - Dadurch "erbt" das Kindobjekt den Zustand des Elternobjektes.

Implementierung von Objekten, Methoden, Nachrichten

```
var bond = { firstName : "James",
             lastName : "Bond",
             fullName : function () { return this.firstName
                                      + ' ' + this.lastName }
           };
```

bond.fullName(); // Nachricht an bond

Objektlayout im Speicher:



1. Übersetzung von Nachrichten als Funktionsaufrufe deren erstes Argument der Nachrichten-Empfänger ist.

2. Funktionsobjekt mit `"call(obj,args)"`-Methode. Hat „`this`“ als zusätzlichen Parameter. Property-Zugriffe sind als Nachrichten übersetzt.

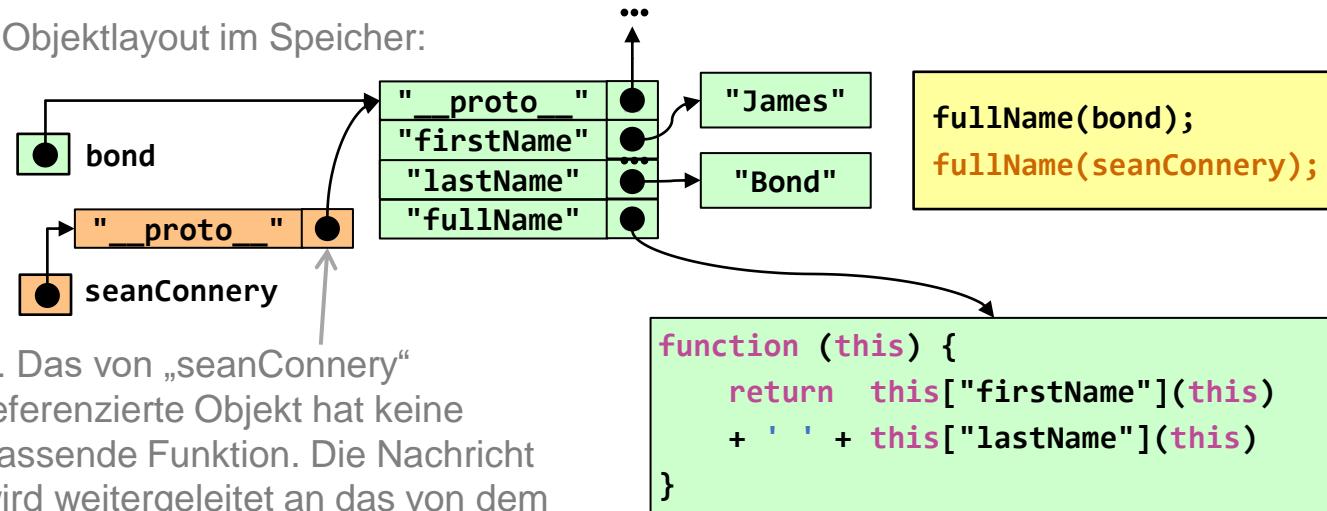
Implementierung von Delegation durch Verfolgen der "__proto__"-Referenz

```
var bond = { firstName : "James",
             lastName : "Bond",
             fullName : function () { return this.firstName
                                      + ' ' + this.lastName } };
```

```
var seanConnery = {};
seanConnery.__proto__ = bond ;
```

```
bond.fullName(); // Nachricht an bond
seanConnery.fullName(); // ... an sean
```

Objektlayout im Speicher:



2. Das von „seanConnery“ referenzierte Objekt hat keine passende Funktion. Die Nachricht wird weitergeleitet an das von dem „__proto__“-slot referenzierte Objekt. „this“ bleibt unverändert.

1. Der Empfänger wird wie üblich als „this“-Wert übergeben. Die Nachricht geht also an „seanConnery“.

3. Das Elternobjekt hat eine passende Funktion. Die wird mit den aktuellen Argumenten aufgerufen. Ab hier wiederholt sich der Vorgang für jede Nachricht im Rumpf..

Objektbasierte Vererbung ("Delegation")

- Bei der "Delegation" bleibt „this“ das Objekt, das die Nachricht ursprünglich erhalten hat.
- Eine Nachricht delegieren ist also nicht das Gleiche, wie die Nachricht an ein anderes Objekt zu senden.
 - ◆ Senden setzt "this" neu, auf das Objekt, das die Nachricht empfängt.
 - ◆ Delegieren behält den aktuellen Wert von "this" bei.
 - ◆ Achtung, Terminologie-Verwirrung: "Delegation" wird oft als "weiter-senden" der gleichen Nachricht an ein anderes Objekt" benutzt, was aber „this“ verändern würde!
- Die Essenz der Vererbung ist also der implizite "this" Parameter von Methoden! ← sowohl in objektbasierten als auch in klassenbasierten Sprachen
- Delegation ist im Prinzip dynamisch: Der Wert des „prototype“-Slots kann beliebig neu gesetzt werden.
 - ◆ Probieren Sie aus, ob das in JavaScript auch geht ;-)

Der Urknall: „Delegation“ und „SELF“

- Die erste (nicht implementierte) Idee einer prototypbasierten Sprache hat Henry Liebermann, der „Vater“ des MIT Media Lab, beschrieben in „Using prototypical objects to Implement Shared Behaviour in Object Oriented Systems“.
- David Ungar und Randall B. Smith haben in "SELF – The Power of Simplicity" das erste implementierte System beschrieben. SELF bot revolutionäre Sprachkonzepte, revolutionäre Implementierungstechniken und eine revolutionäre Benutzerschnittstelle. Es hat alles beeinflusst, was seither an Sprachentwicklung stattgefunden hat.
- Unter <https://selflanguage.org/> ist alles zu SELF zusammengetragen.
 - ◆ Schauen Sie sich das Video unten auf der Seite ("Self: The Movie") an. Es zeigt das graphische Benutzerinterface von SELF und erklärt dabei die Sprachkonzepte.
 - ◆ Das erste Video ist *nicht* empfehlenswert als Einstieg. Es ist eine Meta-Perspektive, die Zusammenfassung essentieller Einsichten für diejenigen, die schon alles sehr gut kennen.
- Unter <https://bibliography.selflanguage.org/> finden Sie eine nach Themenbereichen strukturierte Liste aller Veröffentlichungen zu SELF.
- Unter der visionären Anleitung von David Ungar und Randy Smith hat das SELF-Projekt Leute hervorgebracht, die auf eine oder andere Weise die Welt der Informatik geprägt haben. Suchen Sie mal nach Urs Hölzle(!) oder Lars Bak ;-)

1.2

Was also ist Objektorientierung (allgemein)?

Objektorientierte Programmiersprachen (OOPS)

Konzept	Prototypbasiert	Klassenbasiert
Objekt Einheiten von Daten und Code	✓	✓
Kapselung Schnittstelle begrenzt Zugriff	✓	✓
Dynamisches Binden Empfänger bestimmt das Verhalten	✓	✓
Vererbung Code verhält sich kontextspezifisch, je nach „Empfänger“ der Nachricht	dynamisch, zwischen Objekten	statisch, zwischen Klassen
Slot Code ist zuweisbarer Wert	✓	✗
Dynamische Struktur Menge der Slots ist zur Laufzeit änderbar	✓	✗
Instanziierung Objekte haben gleiche Struktur und gleiches Verhalten	✗	✓

Können Klassen prototypbasiert simuliert werden?

Was wollen wir ausdrücken?

- Instanzierung = Erzeugung von Objekten mit **gleicher Struktur** und **gleichem Verhalten**
- **Gleiche Struktur** = Gleiche Menge an „Instanzvariablen“
- **Gleiches Verhalten** = Nutzung derselben „Instanzmethoden“

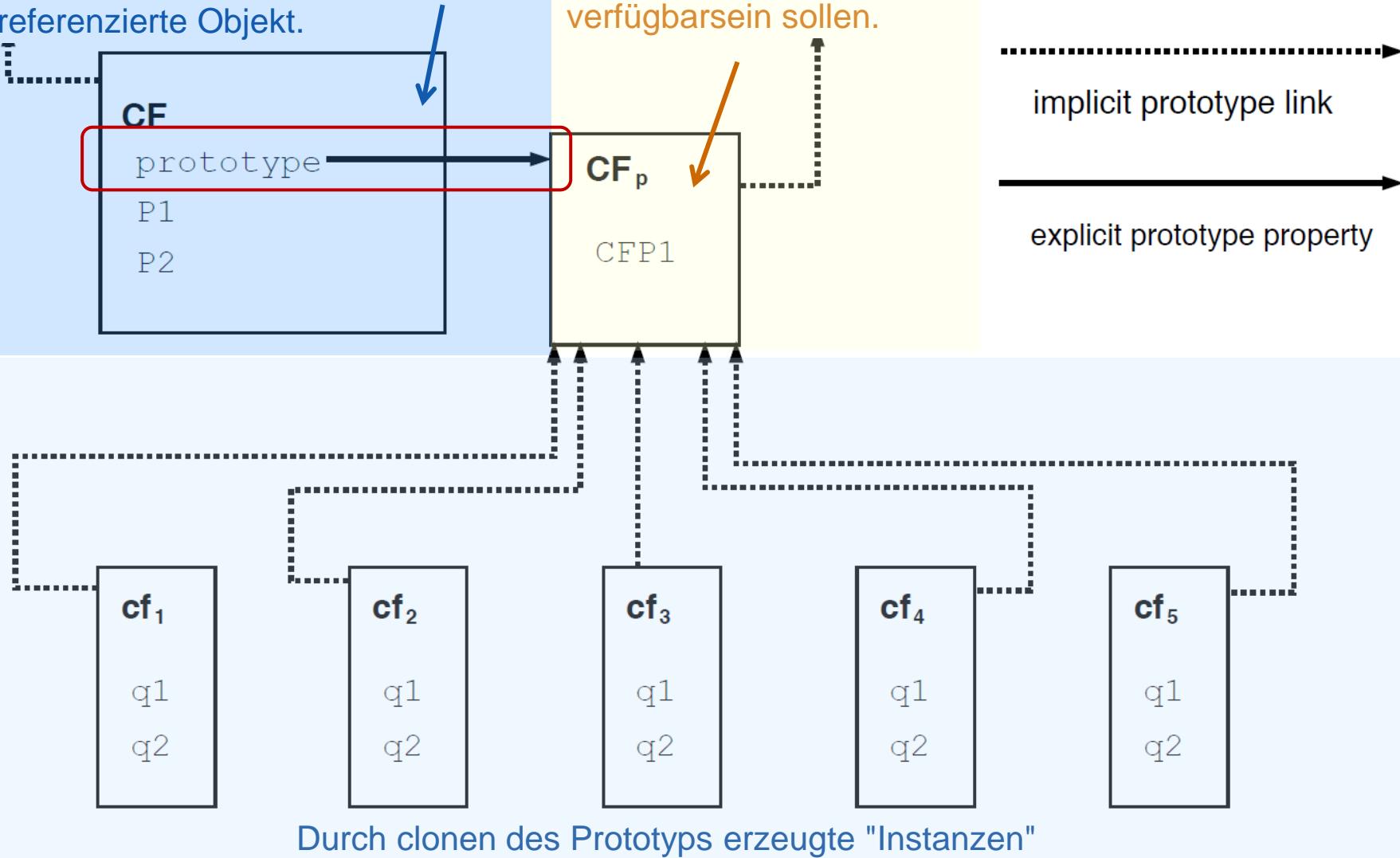
Wie machen wir es?

- **Musterobjekt** beinhaltet gewünschte Struktur („Instanzvariablen“)
- **Elternobjekt des Musterobjekts** enthält gewünschtes Verhalten (Methoden)
- **Cloning** des Musterobjektes erzeugt Objekte mit gleicher Struktur und gleichem (delegiertem) Verhalten

Simulations-idee

Der „Konstruktor“ für das Erzeugen von Objekten mit gleicher Struktur. Er Clont sich selber und setzt in den Clonen die Elternreferenz auf das von seinem "prototype" slot referenzierte Objekt.

Das gemeinsame Elternobjekt stellt die Klasse dar. Hier werden alle Slots untergebracht die allen "Instanzen" gemeinsam verfügbare sein sollen.



Siehe auch Aufgabe 1, Blatt 2

Prototypen → Klassen und Instanzen

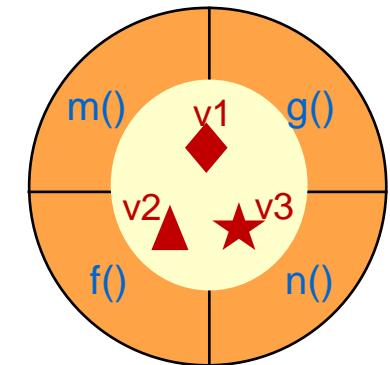
- Motivation konzeptuell
 - ◆ Prototyp-basierte OOP ist unbeschränkt flexibel, aber manchmal will man das nicht → Schutz, Sicherheit, Einheitliche Struktur von Objekten, ...
 - ⇒ Platon: Die Welt lässt sich kategorisieren.
 - ◆ Fehler in prototyp-basierten Sprachen werden erst zur Laufzeit erkannt
 - ⇒ Überholt: TypeScript ist eine statisch getypte Version von JavaScript
 - ⇒ ... die allerdings einiges von der Flexibilität zugunsten der Sicherheit aufgibt.
- Motivation implementierungstechnisch
 - ◆ Effizientere Implementierung (in grauer Vorzeit) nur möglich, wenn feste Struktur statisch bekannt war
 - ⇒ Heutzutage: Mischung von Interpreter und hoch-optimierenden „Hot-Spot“ Compilern verbindet extreme Dynamik und Effizienz
 - Anfang: „Self“ (Erste prototypbasierte Sprache bei Sun Microsystems)
 - Weite Verbreitung durch Java
 - Inzwischen sehr ausgereift

1.3 **Klassen-basierte OOP – in Java**

Rückblick und Update / Erweiterung der Konzepte aus ADIP & OOSE

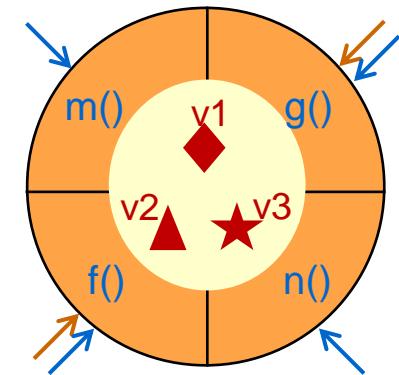
Objekte

- Objekte
 - ◆ Gekapselte, dynamisch erzeugte Einheiten von Daten und Operationen
 - Variablen → Zustand
 - Operationen → Verhalten
- Zustand eines Objektes
 - ◆ Werte der Variablen des Objektes zu einem gewissen Zeitpunkt → Zustand kann sich ändern
- Verhalten eines Objektes
 - ◆ Menge der Reaktionen auf Operationsaufrufe
 - ◆ Reaktionsmöglichkeiten: Eigene Zustandsübergänge und Aufruf von Operationen anderer Objekte



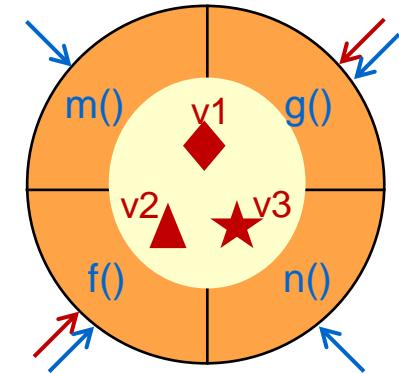
Objekte ▶ Schnittstelle

- Objekte
 - ◆ Gekapselte, dynamisch erzeugte Einheiten von Daten und Operationen
 - Variablen → Zustand
 - Operationen → Verhalten
- Schnittstelle
 - ◆ Menge für einen bestimmten Benutzerkreis aufrufbaren Operationen
 - ◆ Verschiedene Arten von „Benutzern“ können evtl. unterschiedliche Schnittstellen angeboten bekommen („public“, „package“, „protected“, ...)



Objekte ▶ Kapselung

- Objekte
 - ◆ Gekapselte, dynamisch erzeugte Einheiten von Daten und Operationen
 - Variablen → Zustand
 - Operationen → Verhalten
- Kapselung
 - ◆ Sprache stellt sicher, dass der Zustand eines Objektes nur über die in seiner Schnittstelle spezifizierten Operationen manipuliert wird
 - Bei gleichbleibendem Interface wirken sich Änderungen der Implementierung nicht auf andere Objekte aus
- Wartbarkeit und Zugriffs-Synchronisation!



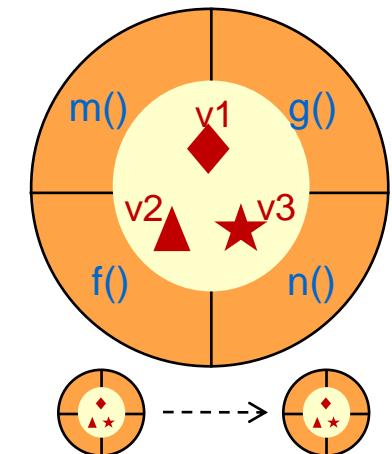
Objekte > Spezifikation und Erzeugung (1)

● Objekt-/Prototypbasierte Sprachen

- ◆ Erzeugung durch „hinschreiben“:
Die Variablen + Methoden können pro Objekt einzeln angegeben werden
 - Anschließend existiert das Objekt!
- ◆ Erzeugung durch kopieren („clonen“):
Objekte werden durch Kopieren von anderen Objekten erzeugt (und danach verändert)

● Beispiele

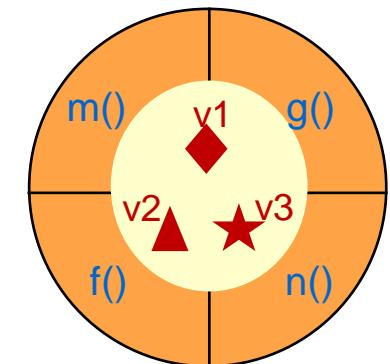
- ◆ Self – der Urvater aller prototypbasierten Sprachen
- ◆ Newton-Script – Die Sprache des Urvaters aller PDAs
- ◆ Java-Script – Die Sprache für dynamische Webseiten



Objekte ▶ Spezifikation und Erzeugung (2)

● Klassenbasierte Sprachen

- ◆ Spezifikation durch „hinschreiben“:
Die Variablen + Methoden können pro Objekt einzeln spezifiziert werden
 - Dadurch entsteht aber noch kein Objekt!
- ◆ Erzeugung durch „Instantiierung“:
Objekte werden durch Aufruf einer speziellen Operation aus der Spezifikation erzeugt
- ◆ Klasse spielt 2 Rollen: Als Modul mit Klassenvariablen und Methoden und als Schablone für Objekterzeugung durch „Instanziierung“
- ◆ Objekte werden dementsprechend als „Instanzen“ bezeichnet



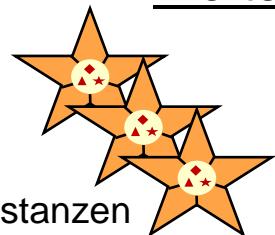
Beispiele klassenbasierter Sprachen

- Simula (1968)
 - ◆ Der Urvater aller objektorientierten Sprachen
- Smalltalk (1970)
 - ◆ Die erste „rein objektorientierte“ Sprache
 - „Rein“ heißt hier: „Alles ist ein Objekt!“ – Auch „elementare“ Datentypen (Zahlen, Strings)! – Auch Klassen!
- C++ (1979 / 1983)
 - ◆ Die erste effiziente Implementierung einer oo Sprache
- Eiffel (1986)
 - ◆ Die erste oo Sprache die Sprachkonstrukte für Modellierung mit „Design by Contract“ (DBC) anbot

Klassifikation und Instantiierung: Java

- Klasse beschreibt Menge „gleichartiger“ Objekte

- ◆ **gleiche** Schnittstelle
- ◆ **gleiche** Implementierung
- ◆ **gleiche** Variablen
- ◆ **verschiedene** Variablen-Werte



Aufruf eines anderen Konstruktors der gleichen Klasse.

```
class Bike {  
    // Instanz-Variablen:  
    Bremse vorne, hinten;  
    int gang = 18;  
  
    // Instanz-Methoden:  
    void schalten() {gang++;}  
  
    // "Konstruktor"-Methoden:  
    Bike(Bremse v, Bremse h) {  
        vorne = v; hinten = h;  
    }  
    Bike() {  
        this(..., ...)  
    }  
}
```

- Klasse ist Schablone für Objekterzeugung

```
Bike meinRad = new Bike();  
Bike deinRad = new Bike(vb, hb);
```

Klassen-Variablen und Klassen-Methoden

● Variablen & Methoden, die

- ◆ nur ein mal pro Klasse existieren
- ◆ für alle Instanzen zugreifbar sind
- ◆ durch Nachrichten an die Klasse oder an ihre Instanzen auch von außen zugreifbar sind:
 - Bike.verkaufe(10); // empfohlen
 - Bike b = new Bike();
b.verkaufe(3); // schlechter Stil

● Benutzung

- ◆ gemeinsame Eigenschaften aller Instanzen
 - z.B. klassenspezifische Konstanten
- ◆ Informationen über die Instanzen (Metainformationen)
 - z.B. Anzahl der Instanzen

```
class Bike {  
    // Instanz-Variablen:  
    ...; int gang = 18;  
  
    // Klassen-Variable:  
    static final int gänge = 21;  
  
    // Instanz-Methode:  
    void schalten() {  
        if (gang < gänge) {gang++;}  
    }  
  
    // Klassen-Variable:  
    static int nrOfBikes = 0;  
  
    // Klassen-Methode:  
    public static void verkaufe(int a)  
    {  
        if (anz < nrOfBikes)  
            nrOfBikes = nrOfBikes - anz;  
    }  
}
```

Vererbung

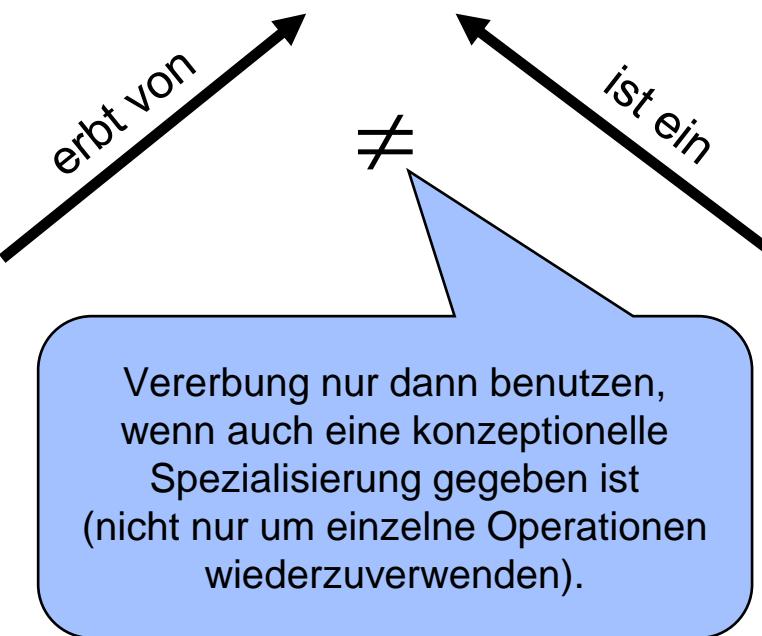
Technisch
Wiederverwendung
von Variablen und
Operationen

Mitarbeiter
Familienname Geburtstag
berechneKosten() erledigeAuftrag()

Konzeptuell
Spezialisierung bzw.
Verallgemeinerung
von Klassen

Angestellter
Abteilung Gehalt
versetze() berechneKosten()

Freiberufler
Stundensatz berechneKosten()



Vererbung in Java

● Unter-Klassen

```
class SubC extends SuperC { ... }
```

- ◆ **SubC** enthält implizit alle Methoden + Variablen aus **SuperC**, ...
 - Vererbung
- ◆ ... die nicht lokal redefiniert (reimplementiert) wurden
 - Overriding

Java erlaubt beliebig viele Ober-Interfaces ("Mehrfa^{ch}e Subtypbeziehung")

● Unter-Interfaces

```
interface SubI extends I1 ... In { ... }
```

- ◆ **SubI** enthält implizit alle Methodensignaturen aus **SubI, I1 ... In**
- ◆ Mehrfaches Auftreten der gleichen Signatur in **SubI, I1 ... In** möglich
 - kein Konflikt sondern eine einzige Methodensignatur
 - **Overloading** = verschiedenen Signaturen für gleichen Namen

Vererbung in Java

„final“ Methoden

- ◆ dürfen in Unterklassen nicht überschrieben werden
- ◆ dynamisch gebunden!!!
- ◆ jede Methode die nicht explizit „static“ oder „final“ ist, ist dynamisch gebunden („virtual“ in C++)

```
class A {  
    public void g() {...}  
}  
  
class B extends A {  
    public void f() {...}  
    final public void g() {...}  
}  
  
  
class C extends B {  
    public void f() {...}  
    [ public void g() {...} ]  
}  
                                         Compilierfehler
```

„final“ Klassen

- ◆ dürfen keine Unterklassen haben

```
final class A { ... }  
  
class B [ extends A ] { ... }  
                                         Compilierfehler
```

„final“ Variablen = Konstanten

- ◆ dürfen nicht verändert werden (s. Beispiel auf Seite 34)

Die Klasse „java.lang.Object“

hier steht implizit noch
„extends Object“

- Jede Klasse erbt von „Object“

 - ◆ unter anderem die „clone()“ Methode – shallow clone!
 - ◆ sollte in Ihrer Klasse überschrieben werden, um das cloning-Verhalten anzupassen
 - Auf die Frage, wie tief man clonen sollte kommen wir zurück wenn wir über im Kontext der UML über „Aggregation“ sprechen
- Object ist somit ein gemeinsamer Obertyp aller Objekttypen
- Das Paket „java.lang“ wird automatisch importiert
 - ◆ Sie können daher Klassen wie „Object“, „String“, etc ohne „Qualifier“ (java.lang) benutzen

1.4 Typen und Untertypen

Objektorientierte Typsysteme

- Objekte und Typen sind orthogonale Konzepte
- Dynamische Typen
 - ◆ Welche Werte haben Variablen zur Laufzeit?
 - ◆ Universell anwendbar
- Statische Typen
 - ◆ Approximation der dynamischen Typen / Werte durch den Compiler (statisch)
 - ◆ Lange nur in klassenbasierten Sprachen
 - ◆ Inzwischen auch in Prototypbasierten → TypeScript = statisch getyptes JavaScript

Typen

- Funktion
 - ◆ Typen beschränken die zulässigen Werte von Ausdrücken
- Sinn statische Typdeklaration
 - ◆ Dokumentation: **Mitarbeiter** mitarb;
 - ◆ Korrektheit: **mitarb.wiehern();** // ☹
 - ◆ Effizienz: statische Indexbestimmung
- Problem statischer Typsysteme: Starrheit
 - ◆ Nur Werte die **genau** dem deklarierten Typ entsprechen sind zulässig
- Idee: Ersetzbarkeit = auch Werte zulassen, die
 - ◆ nicht genau dem deklarierten Typ entsprechen, aber
 - ◆ in jedem Kontext eingesetzt werden können, wo der deklarierte Typ erwartet wird

Statischer versus dynamischer Typ

- Statischer Typ eines Ausdrucks
 - ◆ Der deklarierte bzw. der aus Deklarationen (ohne Datenflussanalyse) herleitbare Typ des Ausdrucks
 - ◆ Notation: $\lceil e \rfloor$ bezeichnet den statischen Typ des Ausdrucks e
- Dynamischer Typ eines Ausdrucks
 - ◆ Der Typ des Wertes des Ausdrucks zur Laufzeit
 - ◆ Notation: $\lfloor e \rfloor$ bezeichnet den dynamischen Typ des Ausdrucks e
- Beispiel

```
Link l = new TNode();    //  $\lceil \text{new TNode()} \rfloor == \text{Tnode} == \lfloor \text{new TNode()} \rfloor$ 
//  $\lceil l \rfloor == \text{Link}$ 
//  $\lfloor l \rfloor == \lceil \text{new TNode()} \rfloor == \text{TNode}$ 

TNode n = (TNode) l;    //  $\lceil l \rfloor == \text{Link}$ 
//  $\lceil (\text{TNode}) l \rfloor == \lfloor l \rfloor == \text{TNode}$ 
//  $\lceil n \rfloor == \text{TNode}$ 
//  $\lfloor n \rfloor == \lceil (\text{TNode}) l \rfloor == \text{Tnode}$ 
```

Ersetzbarkeit

B ist ein Untertyp von A

\Leftrightarrow

Instanzen von B sind immer für Instanzen von A einsetzbar

\Leftrightarrow

Instanzen von B fordern höchstens und bieten mindestens das gleiche wie Instanzen von A

Allgemeines Prinzip

\Rightarrow

Instanzen von B haben (mindestens) alle Methoden von A mit genau gleichem Namen und gleichen Parameter-Typen, sowie einem evtl. spezifischerem Ergebnis-Typ

Automatisch überprüfbares Kriterium

Eine Methode fordert korrekte Eingaben, d.h. korrekte Parameterwerte

Eine Methode bietet Ergebnisse eines bestimmten Typs

Ein Typ bietet eine Menge von Methoden mit einer bestimmten Signatur

Ersetzbarkeit ▶ Beispiel

Instanzen von des Untertyps haben
(mindestens) alle Methoden des Obertyps mit
genau gleichem Namen und Parameter-Typen,
sowie einem evtl. spezifischerem Ergebnis-Typ

```
class Point2DImpl implements Point2D {  
    private int x, y;  
    public int getX(){ return x; }  
    public int getY(){ return y; }  
    public Point2D goto(Point2D p) {  
        x = p.getX();  
        y = p.getY();  
        return this;  
    }  
}
```

```
class Point3DImpl extends Point2DImpl  
implements Point3D {  
    private int z;  
    public int getZ(){ return z; }  
    ... ? ...  
}
```

```
interface Point2D {  
    public int getX();  
    public int getY();  
    public Point2D goto(Point2D p);  
}
```

```
interface Point3D extends Point2D{  
    public int getZ();  
    public Point3D goto(Point2D p);  
}
```

Ersetzbarkeit ▶ Beispiel

```
class Point2DImpl implements Point2D {  
    private int x, y;  
    public int getX(){ return x; }  
    public int getY(){ return y; }  
    public Point2D goto(Point2D p) {  
        x = p.getX();  
        y = p.getY();  
        return this;  
    }  
}
```

```
class Point3DImpl extends Point2DImpl  
implements Point3D {  
    private int z;  
    public int getZ(){ return z; }  
    public Point3D goto(Point2D p) {  
        super(p);  
        if (p instanceof Point3D) goto( (Point3D) p);  
        return this;  
    }  
    public Point3D goto(Point3D p){  
        x = p.getX();  
        y = p.getY();  
        z = p.getZ();  
        return this;  
    }  
}
```

```
interface Point2D {  
    public int getX();  
    public int getY();  
    public Point2D goto(Point2D p);  
}
```

```
interface Point3D extends Point2D{  
    public int getZ();  
    public Point3D goto(Point2D p);  
}
```

Typ-Hierarchie (in Java): Vererbung und Implementierung

```
class Object { ... }
```

```
interface Object { ... }
```

```
class Point2DImpl implements Point2D {
    private int x, y;
    ...
}
```

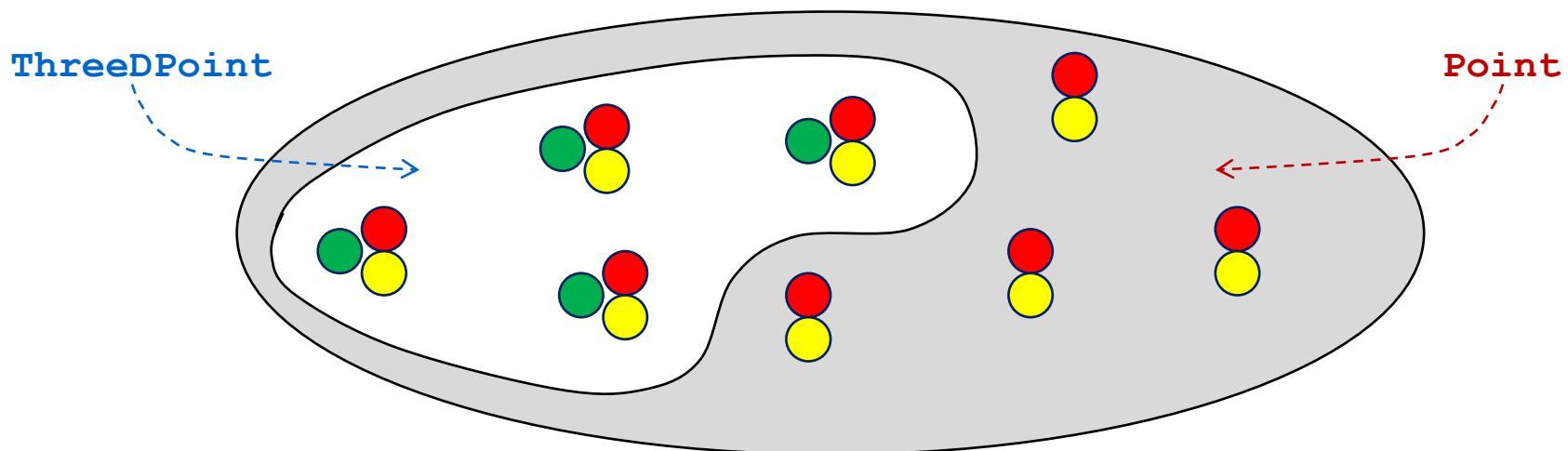
```
interface Point2D {
    public int getX();
    public int getY();
    public Point2D goto(Point2D p);
}
```

```
class Point3DImpl extends Point2DImpl
implements Point3D {
    private int z;
    ...
}
```

```
interface Point3D extends Point2D{
    public int getZ();
    public Point3D goto(Point2D p);
}
```

Ersetzbarkeit ▶ Mengensicht

Teilmengenbeziehung: Instanzen von des Untertyps sind immer auch Instanzen des Obertyps



→ Je mehr Eigenschaften gefordert werden
um so weniger Instanzen gibt es
die all die Eigenschaften haben!

→ Unterklassen definieren daher Teilmengen

1.5 **Statische und dynamische Semantik**

Der Unterschied von statischen und dynamischen Typen beeinflusst die Semantik vieler Konzepte der Sprache!!!

Statische versus dynamische Semantik

Statische Semantik

Alle vom Compiler durchgeführten Checks

- Syntaxüberprüfung
- Typ-Überprüfung
- legale Zugriffe und Aufrufe

... und Entscheidungen:

- Auf welche Variable wird hier zugegriffen? → **Verdecken?**
- Welche Methode wird hier aufgerufen? → **Binden!!**

Dies geschieht alles anhand der statischen Typen von Ausdrücken im Programmtext!

Dynamische Semantik

Was geschieht zur Laufzeit, während der Programmausführung?

- Kontrollfluss
 - ◆ Sequenz
 - ◆ Alternativen
 - ◆ Schleifen
 - ◆ Threads
- Datenfluss
 - ◆ Welche **Werte** werden durch die Auswertung der Ausdrücke berechnet?

Der Typ des Werts ist der dynamische Typ des ausgewerteten Ausdrucks

Verdecken (hiding / shadowing)

von Variablen

Die Deklaration einer Variable in einer Klasse verdeckt jede Variable **gleichen Namens** in Oberklassen

- ◆ Auch wenn sie verschiedene Typen haben!
- ◆ Klassenvariablen verdecken auch Instanzvariablen und Instanzvariablen verdecken auch Klassenvariablen

**Verdecken ist sehr schlechter Stil
(extrem verwirrender Code)**

Vermeiden!!!

von Methoden

Eine Deklaration einer **Klassenmethode** verdeckt jede **Klassenmethode gleicher Signatur** in Oberklassen

- ◆ Der Ergebnistyp ist nicht Teil der Signatur!
- ◆ Instanzmethoden werden nie verdeckt sondern von Instanzmethoden in Unterklassen „überschrieben“ („overridden“)
- ◆ Klassenmethoden verbergen oder überschreiben nicht Instanzmethoden und umgekehrt

Zugriff auf verdeckte Elemente

Prinzip

- Sei **expr** ein Ausdruck vom statischen Typ **T**
der ein Objekt **o** vom Untertyp **T_{sub}** referenziert.
- Dann sind über **expr** nur die Variablen und Operationen von **o** sichtbar die in der Schnittstelle von **T** definiert sind
 - ◆ Variablen = Instanz- und Klassenvariablen
 - ◆ Operationen = Instanz- und Klassenoperationen

Der statische Typ bestimmt die dem Compiler bekannte Schnittstelle.

Nutzung zum Zugriff auf verdeckte Elemente aus Obertypen

- Klassenvariablen und -operationen
 - ◆ Nie über ein Objekt sondern direkt über die Klasse: **MyClass.x**, **Myclass.f()**
- Instanzvariablen und Operationen
 - ◆ Vor dem Zugriff, den Ausdruck zum gewünschten Obertyp „casten“ oder einer Variablen vom gewünschten statischen Typ zuweisen:
((MyClass) expr).x; oder **MyClass temp; temp = expr; temp.x;**

Klassenmethoden sind statisch gebunden!

(Beispiel bei Rückfrage in der Vorlesung)

Deklarationen

- Sub erbt von Super
- Super hat eine static f() Methode (Klassenmethode!)
- Sub hat eine static f() Methode (Klassenmethode!)
- Super var = new Sub();

Verwirrender Code (Aufruf von Klassenmethode über eine Instanz)

- `var.f();` // Dies ist eine Aufruf der Methode aus Super, da var den statischen Typ Super hat. Aufrufe von Klassenmethoden sind NICHT dynamisch gebunden (sondern statisch, wie das Schlüsselwort „static“ es sagt ;-)). Es wird also die Methode aus Super aufgerufen, egal über welchen Typ das Objekt hat über das der Zugriff stattfindet.

Verständlicher Code (Aufruf von Klassenmethode über die Klasse)

- `Super.f();` // Hier ist sofort klar, dass das f() aus Super gemeint ist! ☺

„Welche Operation wird aufgerufen?“

- Wie lässt sich feststellen, ob die farbig hervorgehobene Zuweisung legal ist?

```
public class OverloadingTest {  
    public static void main(String args[]) {  
        int i=1;  
        float f=1.0f;  
  
        long l = myFunction(i+f);  
    }  
  
    float myFunction(float f) { ... }  
    long myFunction(long l) { ... }  
}
```

1. Ist diese Zuweisung legal???

5. Was ist die Aufrufsignatur von „+“?

4. Welche „+“-Operation wird hier aufgerufen?

3. Welcher Ergebnistyp wird hier produziert?

2. Welche dieser Funktionen wird aufgerufen?

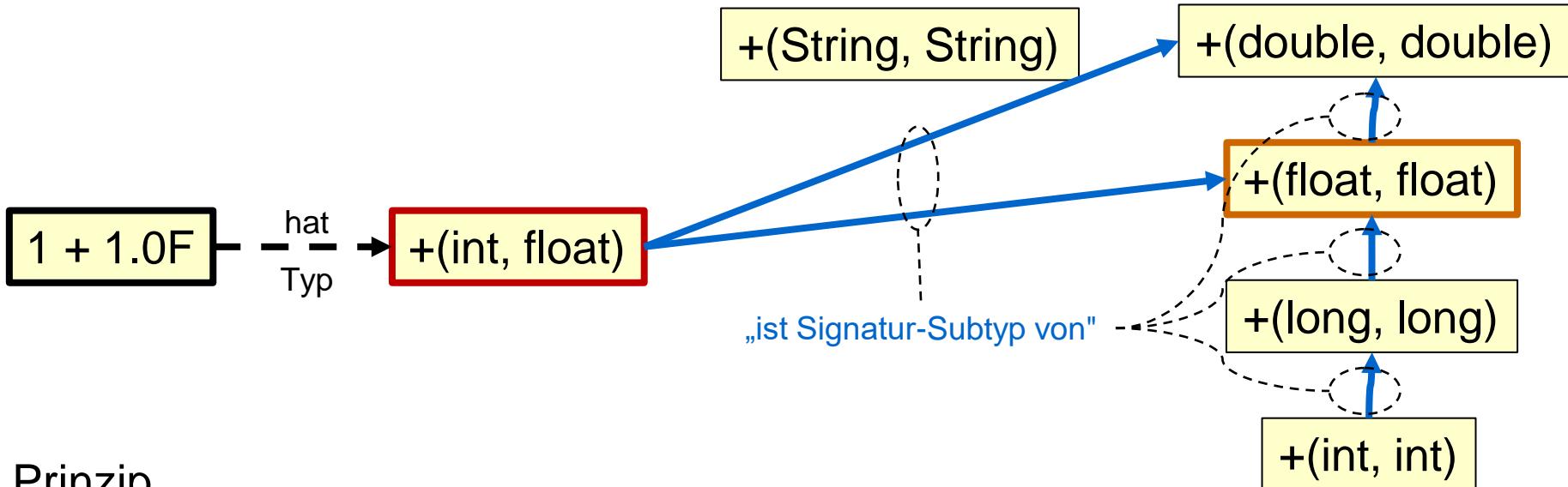
Bestimmung der aufgerufenen Methoden-Signatur

Signatur = Name und Parameter/Argumenttypen (ohne Ergebnistyp!)

Aufruf

Aufrufsignatur

Verfügbare Methodensignaturen



Prinzip

- Es wird die **spezifischste Methodensignatur** gewählt, von der die **Aufrufsignatur** ein **Subtyp** ist,
- ... immer! (Beispiel mit primitiven Typen nur der Einfachheit halber)

Bestimmung der aufgerufenen Methoden-Signatur für geht anhand statischer Typen!

Beispiel-Fortsetzung

- Wie lässt sich feststellen, ob die farbig hervorgehobene Zuweisung legal ist?

```
public class OverloadingTest {  
    public static void main(String args[]) {  
        int i=1;  
        float f=1.0f;  
  
        long l = myFunction( i+f );  
    }  
  
    float myFunction(float f) { ... }  
    long myFunction(long l) { ... }  
}
```

6. Die Aufrufsignatur von **i+f** ist **int+float**
7. Die spezifischste anwendbare Operation für **i+f** ist somit **float+float**
8. Der Ergebnistyp von **float+float** ist **float**
9. Die Aufrufsignatur von **myFunction(i+f)** ist somit **myFunction(float)**
10. Die spezifischste anwendbare Operation für **myFunction(i+f)** ist somit **myFunction(float)**
11. Der Ergebnistyp von **myFunction(float)** ist **float**.
12. Die Zuweisung **long = float** ist illegal!

Wir haben in diesem Abschnitt auf zwei wichtige Fragen der statischen Semantik zurückgeblickt:

- „Auf welche Variable wird hier zugegriffen?“
 - ◆ Auflösung von Hiding anhand der statisch verfügbaren Typinformation
- „Welche Operation wird hier aufgerufen?“
 - ◆ Bestimmung der aufgerufenen Methodensignatur anhand der statisch verfügbaren Typinformation (Folie 54 – 56)
 - ◆ Die Signatur identifiziert statisch eine Operation.
 - ◆ Sie entscheidet für Instanzoperationen noch nicht über die entsprechende Implementierung der Operation (d.h. über die ausgeführte Methode). Das geschieht erst zur Laufzeit, durch das dynamische Binden.
- Das Beispiel auf Folie 54 – 56 hat gezeigt wie die verschiedenen Aspekte der statischen Semantik zusammenspielen: (a) Bestimmung statischer Typen, (b) Überprüfung der Kompatibilität statischer Typen und (c) Entscheidung was aufgerufen (bzw. zugegriffen) wird

Zusammenfassung

Statische Semantik

1. Es kann nur auf Attribute und Methoden zugegriffen werden, deren Signatur im **statischen Typ des Empfängers** definiert ist
2. **Instanzmethoden überschreiben („overriding“), Klassenmethoden verdecken („hiding“)** Methoden mit gleicher Signatur aus Obertypen
3. **(Klassen- und Instanz-)Variablen verdecken („hiding“)** gleichbenannte Variablen aus Obertypen
4. **Auf verdeckte Variablen und Methoden** eines Obertyps kann man über Ausdrücke vom passenden statischen Typ zugreifen
5. **Statische und finale Methoden dürfen nicht überschrieben werden**

Dynamische Semantik

6. Es werden immer die dem tatsächlichen Objekt zugehörigen Varianten von **Instanzmethoden** auf diesem ausgeführt (**dynamisches Binden**)

1.6 Implementierung

Variablen-Vererbung, Hiding

Methoden-Vererbung, Hiding, Overriding, Overloading

Statisches und dynamisches Binden

□ Layout von Objekten

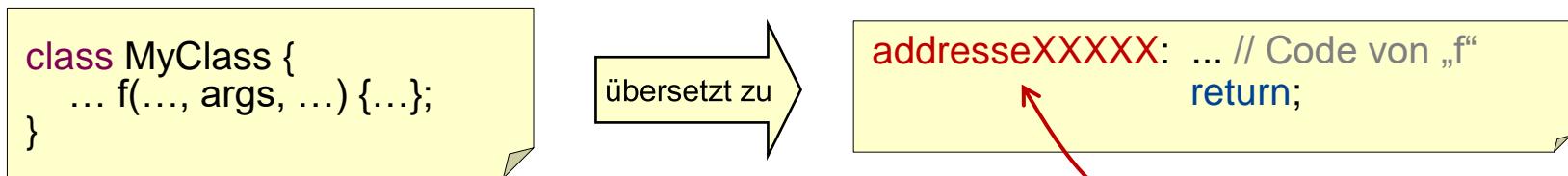
□ Layout von „vTables“

□ Codegenerierung

Prinzipien ▶ Imperativ

- Zugriffe auf statisch bekannte Speicherstellen / Code
 - ◆ ... werden wie auch in anderen imperativen Sprachen üblich kompiliert
 - ◆ ... indem ihre statisch bekannten Speicheradressen direkt im übersetzten Code eingesetzt werden

Definition



Aufruf



- Dies gilt für Zugriffe auf **Klassenvariablen**, **Klassenmethoden**, **privaten Methoden** und **super-Aufrufe**. Ferner für Aufrufe durch deren statischen Empfängertyp bekannt ist, dass eine **finale Methode** aufgerufen wird.

Prinzipien ▶ Objektorientiert

- Instanzvariablen
 - ◆ Jede Instanzvariable erhält einen eindeutigen Index im Speicherbereich des Objektes.
 - ◆ Die Indices sind die gleichen für alle Instanzen einer Klasse.
 - ◆ Geerbte Variablen haben die gleichen Indices wie in der Oberklasse.
 - ◆ Lokal definierte Variablen werden ab dem höchsten Oberklassenindex + 1 weiter nummeriert.
 - ◆ An Index 0 steht in jedem Objekt ein Verweis auf die **vtable** der Klasse.
- Instanzoperationen
 - ◆ Jede Instanzoperation erhält einen eindeutigen Index in der **vtable**
 - ◆ Geerbte und überschriebene Operationen haben die gleichen Indices wie in der Oberklasse.
 - ◆ Neue lokal definierte Operationen werden ab dem höchsten Index in der **vtable** der Oberklasse + 1 weiter nummeriert.

vtable = virtual function table (von Bjarne Stroustrup in C++ eingeführter Begriff, der allgemein übernommen wurde)

Beispiel

Index der Instanzvariable oder der Instanzoperation

```
class Super {  
    public static int x = 10;  
    public String name;  
  
    public void lerne() {...}  
    public void arbeite() {...}  
    public void arbeite(int Stunden) {...}  
}
```

```
class Sub extends Super {  
    public int gehalt; // Erweiterung  
    public String name; // Hiding  
  
    public static void y() {...}  
    public void arbeite() {...} // Overriding  
    public void gähne() {...} // Erweiterung  
    public Sub(String n, int g) {  
        name = n;  
        gehalt = g;  
    }  
}
```

Super m1,m2;
Sub a;

m1 = new Super();
a = new Sub("Eva", 2000);
m2 = a;

m1.name;
m1.lerne();
m1.arbeite();
m1.arbeite(4);

m2.lerne();
m2.arbeite();
m2.arbeite(4);
m2.gähne(); // nicht in Super

a.lerne();
a.arbeite();
a.arbeite(4);
a.gähne(); // ok in Sub
a.name;

Objekte und vTables

Übersetzte
Klassenvariablen
und -methoden

10 x

```
class Super {  
    public static int x = 10;  
    public String name;  
  
    public void lerne() {...} 0  
    public void arbeite() {...} 1  
    public void arbeite(int St 2  
    } 1
```

Übersetzte
Instanzmethoden

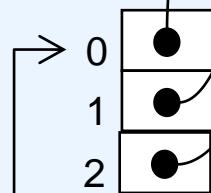
Super::lerne(this)

Super::arbeite(this)

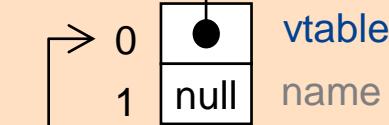
Super::arbeite(this, stunden)

Die Notation Class::member bezeichnet die
Version von member die in Class definiert ist

Virtuelle
Funktionstabelle
(vtable) der Klasse
(Referenzen auf
Instanzmethoden)



Speicherbereich
des referenzierten
Objektes
(Instanzvariablen)



Speicherbereich
einer Variablen



```
Super m1,m2;  
Sub a;  
m1 = new Super();
```

Objekte und vTables

Übersetzte Klassenvariablen und -methoden



```
class Sub extends Super {
    public int gehalt; // Erweiterung ☺
    public String name; // Hiding ☹

    public static void y() {...}
    public void arbeite() {...} // Overriding
    public void gähne() {...} // Erweiterung

    public Sub(String n, int g) {
        name = n;
        gehalt = g;
    }
}
```

2
3
-
1
3
-

Übersetzte Instanzmethoden

Super::lerne(this)

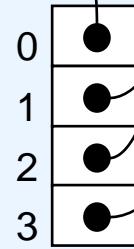
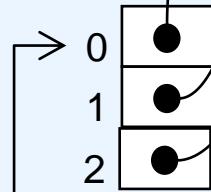
Super::arbeite(this)

Super::arbeite(this, stunden)

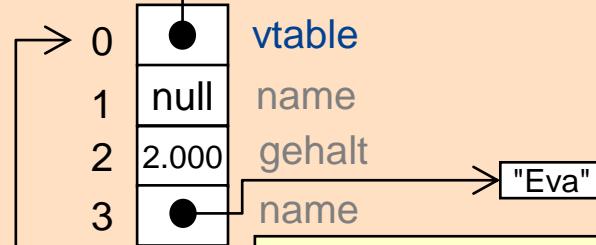
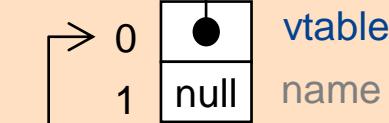
Sub::arbeite(this)

Sub::gähne(this)

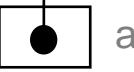
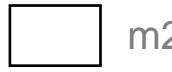
Virtuelle Funktionstabelle (vtable) der Klasse
(Referenzen auf Instanzmethoden)



Speicherbereich des referenzierten Objektes (Instanzvariablen)



Speicherbereich einer Variablen



```
Super m1,m2;
Sub a;
m1 = new Super();
a = new Sub("Eva", 2000);
```

Objekte und vTables

Übersetzte Klassenvariablen und -methoden



```
class Sub extends Super {
    public int gehalt; // Erweiterung ☺
    public String name; // Hiding ☹

    public static void y() {...}
    public void arbeite() {...} // Overriding
    public void gähne() {...} // Erweiterung

    public Sub(String n, int g) {
        name = n;
        gehalt = g;
    }
}
```

Übersetzte Instanzmethoden

Super::lerne(this)

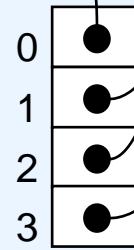
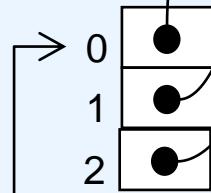
Super::arbeite(this)

Super::arbeite(this, stunden)

Sub::arbeite(this)

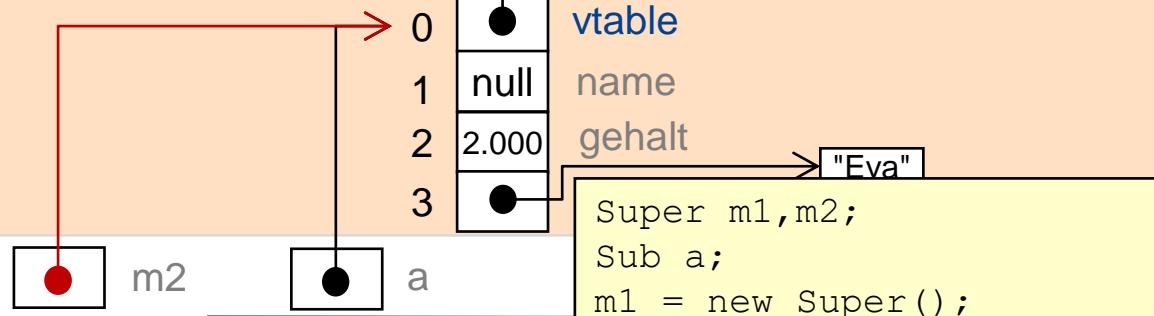
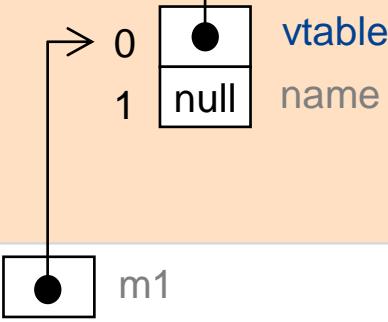
Sub::gähne(this)

Virtuelle Funktionstabelle (vtable) der Klasse
(Referenzen auf Instanzmethoden)



Speicherbereich des referenzierten Objektes (Instanzvariablen)

Speicherbereich einer Variablen



Codegenerierung für dynamische Zugriffe

- Instanzvariablen-Zugriffe sind Zugriffe auf die statisch bekannte Position im Speicher des Objektes
- Nachrichten (= dynamisches Binden) sind Aufrufe des Codes der am statisch bekannten Index in der vTable des Objekts referenziert wird

Zugriff / Aufruf im Quellcode

```
m1.name;  
m1.lerne();  
m1.arbeite();  
m1.arbeite(4);  
  
m2.lerne();  
m2.arbeite();  
m2.arbeite(4);  
m2.gähne(); // nicht in Super  
  
a.lerne();  
a.arbeite();  
a.arbeite(4);  
a.gähne(); // ok in Sub  
a.name;
```

1
0
1
2

0
1
2
⊗

0
1
2
3
3

wird übersetzt zu

Zugriff / Aufruf im übersetzten Code

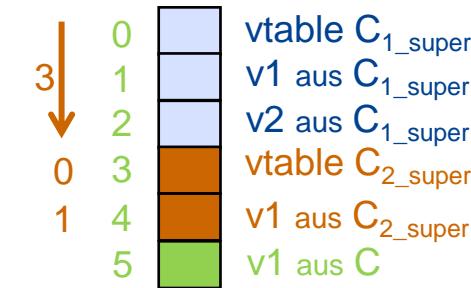
```
m1 [1];  
m1.vtable[0] (m1);  
m1.vtable[1] (m1);  
m1.vtable[2] (m1, 4);  
  
m2.vtable[0] (m2);  
m2.vtable[1] (m2);  
m2.vtable[2] (m2, 4);  
  
a.vtable[0] (a);  
a.vtable[1] (a);  
a.vtable[2] (a, 4);  
a.vtable[3] (a);  
a[3];
```

„this“ wird jeder übersetzten Instanzmethode als zusätzliches erstes Argument übergeben!!!

Multiple Oberklassen (Eiffel, C++, ...)

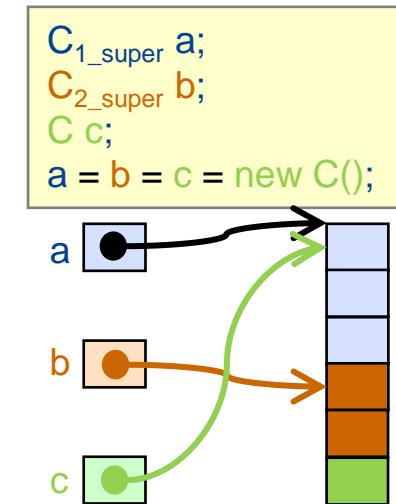
- Wenn eine Klasse **C** mehrere direkte Oberklassen hat ...

- ◆ ... ist im **Objektlayout** für jede Oberklasse je ein Abschnitt für die von der Klasse geerbten Instanzvariablen und vTable-Verweise enthalten
- ◆ Der Bereich der "eigenen" Instanzvariablen von **C** kommt nach allen Oberklassenbereichen
- ◆ Der Compiler merkt sich für jede Oberklasse C_{i_super} die entsprechende Verschiebung des Anfangsbereichs im Objektlayout (z.B. **3** für C_{2_super})



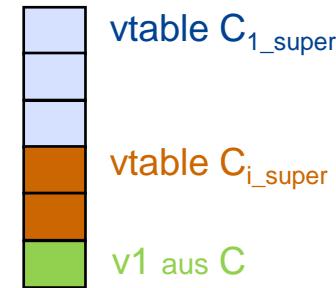
- Wenn ein Objekt des statischen Typs **C** an einen Ausdruck zugewiesen wird, dessen statischer Typ C_{i_super} ist ...

- ◆ ... wird eine entsprechend verschobene Objektreferenz übergeben (der Originalwert wird nicht verändert).
- ◆ So funktionieren die Indexzugriffe aus dem für den Obertyp C_{i_super} kompilierten Code auch auf **C**-Instanzen ☺



Multiple Obertypen

- Wenn ein Typ T mehrere direkte Obertypen hat (Klassen oder Interfaces) ...
 - ◆ ... ist durch die vorhin erläuterte Objektstruktur im Objekt-Layout je ein eigener vTable-Verweis für jede Oberklasse enthalten
 - ◆ ... dieser verweist auf einen Abschnitt der eigenen vTable, der genauso strukturiert ist wie die vTable der Oberklasse (gleiche Indices)
 - ◆ Der Bereich der „eignen“ Methoden kommt nach denen aller Oberklassen (genauso wie der der „eigenen“ Variablen)
 - ◆ Obertypen die Interfaces sind werden genauso behandelt
 - ⇒ Der einzige Unterschied ist, dass das "Objektlayout" für jeden Interface-Obertyp mangels Instanzvariablen zu einem einzigen Eintrag, dem vTable-Verweis degeneriert.



Siehe Beispiel auf nächster Folie
(Klasse mit Oberklasse und implementiertem Interface)

Objekte und vTables

Wir nehmen an f() und g() sind in I definiert und haben dort jeweils Index 0 und 1

```
class Sub2 extends Super
    implements I {
    public void arbeite() {...} // aus Super
    public void gähne() {...} // Erweiterung
    public void f() {...} // aus I
    public void g() {...} // aus I
    ...
}
```

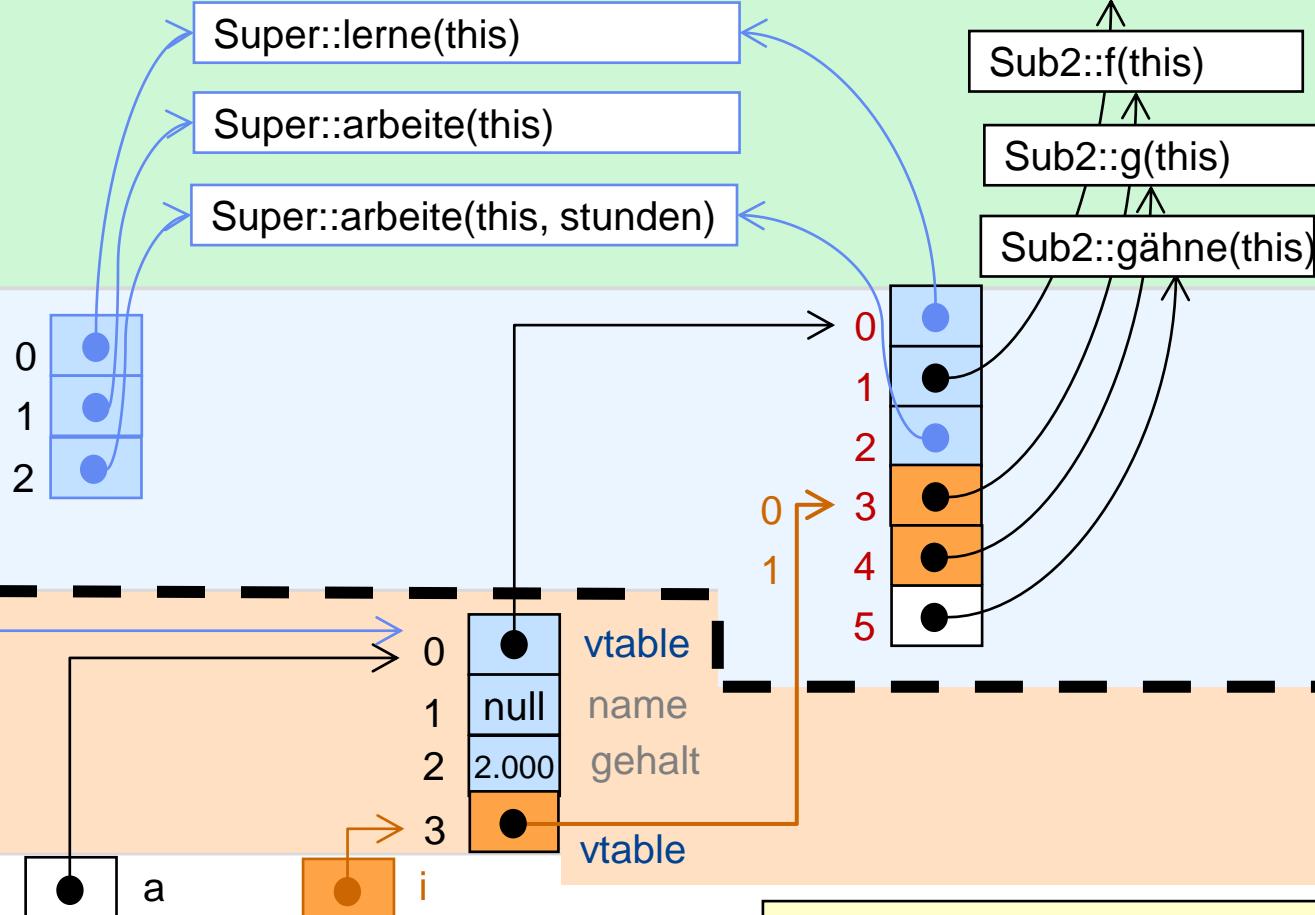
Verschiebung durch vorherige Obertypen
 1
 5
 $3=3+0$
 $4=3+1$
 Index aus I

Übersetzte Instanzmethoden

Virtuelle Funktionstabelle (vtable) der Klasse
(Referenzen auf Instanzmethoden)

Speicherbereich des referenzierten Objektes
(Instanzvariablen)

Speicherbereich einer Variablen



Zusammenfassung Implementierung (1)

- Ausführbarkeit von Oberklassencode auf Instanzen von Unterklassen
 - ◆ „Gleiche“ Objektstruktur (Gleiche Variable an gleichem Objekt-Index)
 - ◆ „Gleiche“ Struktur der vTables (Gleiche Operation an gleichem vTable-Index)
- Hiding von Instanzvariablen
 - ◆ Jede Instanzvariable hat eigenen Index im Objektlayout (= eigene Speicherzelle)
- Hiding von Klasenvariablen und Klassenmethoden
 - ◆ Feste Adresse der Speicherzelle / des Codes vom Compiler festgelegt
- Overloading
 - ◆ Jede Signatur (= Operation) hat eigenen Index in Methodentabelle
- Overriding
 - ◆ In der vTable der Unterkasse wird an der gleichen Position auf anderen Code verwiesen als in der vTable der Oberklasse
- Vererbung
 - ◆ In beiden vTables wird an gleicher Position auf den gleichen Code verwiesen

Zusammenfassung Implementierung (2)

- In diesem Abschnitt haben Sie die zur Laufzeit existierenden Strukturen kennengelernt und insbesondere wie dynamisches Binden dadurch umgesetzt wird.
- Sie sollten nun verstehen, was im Fall von
 - ◆ Hiding
 - ◆ Overloading
 - ◆ Overriding
 - ◆ Vererbungzur Laufzeit geschieht.
- Insbesondere sollte auch das Zusammenspiel von
 - ◆ **statischer Semantik** (Bestimmung von aufgerufenen **Operationen** = Signaturen = Indices in Objekten oder vTables)
 - ◆ **dynamischer Semantik** (Bestimmung der aufgerufenen **Implementierung** = Methode)klar geworden sein.

1.7 **Generische Typen**

Manuell
Sprachunterstützt
Generizität und Subtyping

Generische Datentypen

► Motivation

```
class TNode {  
    T data;  
    TNode next;  
    // ...  
}
```

Wie können wir **Tnode** für Elemente eines anderen Typs als **T** wiederverwenden (der kein Subtyp von **T** ist)?

- Erste Idee
 - ◆ Kopieren und **T** durch gewünschten Datentyp ersetzen
- Vorteil
 - ◆ Leicht möglich
- Nachteil
 - ◆ Kopien für alle benötigten Varianten von Listen / Stapel / Schlangen / ...
 - ⇒ **Nachträgliche** Modifikationen müssen **für jede Variante** nachgeführt werden
 - ⇒ Wartbarkeit des Codes wesentlich eingeschränkt
 - ◆ Kopieren und destruktive Änderungen nur vertretbar, wenn höchstens 1 bis 2 Kopien benötigt werden

```
class T2Node {  
    T2 data;  
    T2Node next;  
    // ...  
}
```

Generische Datentypen

► Manuell

```
class TNode {  
    T data;  
    TNode next;  
    // ...  
}
```

- Wir erhalten eine „generische“ Klasse wenn wir **T** durch **Object** ersetzen:

```
public class List {  
    private Node head;           // Kopf der Liste  
    // ...  
}  
public class Node {  
    Object data;                // Datenelement  
    Node next;                  // Zeiger auf Listenzelle  
    // ...  
    Object getData() {...}      // Datenelement herausgeben  
}
```

- Überall wo wir die Listenelemente als T-Instanzen nutzen wollen, müssen wir nun allerdings casts auf T einfügen:

```
List list = new ArrayList();  
list.add("str");  
String s = (String)list.get(0);      // cast erforderlich  
...  
Integer i = (Integer)list.get(0);    // cast + Laufzeit-Fehler ☹
```

Generische Datentypen: Abgrenzung

- Keine Generizität

```
public class Node {  
    MyData data;  
    Node next;  
    setData(MyData) ...  
}
```

```
Node n = new Node(); // Verwendung
```

zu spezifisch, nicht wiederverwendbar für Daten eines anderen Typs

- Manuelle „Generizität“

```
public class Node {  
    Object data;  
    Node next;  
    setData(Object) ...  
}
```

```
Node n = new Node(); // Verwendung
```

zu allgemein, mit viel manuellem Aufwand verbunden (Code müsste lauter Casts enthalten) und fehleranfällig

- Sprachgestützte Generizität: Ein Datentyp ist generisch, wenn er hinsichtlich des Typs seiner Elemente durch eine Typvariable parametrisiert ist.

```
public class Node<T> {  
    T data;  
    Node<T> next;  
    setData(T) ...  
}
```

```
Node<MyData> n = new Node<MyData>();
```

generisch,
anwendbar auf
beliebige Typen

Der Wert von T wird erst bei der Variablen Deklaration oder Objekt-Instanziierung festgelegt

Generische Typen und Parametrisierte Typen

Generischer Typ

- Ist mit einer oder mehreren **Typvariablen** parametrisiert die in spitzen Klammern hinter dem Typnamen angegeben werden:

```
public class Node<T> {  
    T data;  
    Node<T> next;  
    void setData(T) ...  
    T getData() ...  
}
```

Deklaration der Typvariablen T

Nutzung von T als Platzhalter für einen Typ

- Ein "generischer Typ" ist eine Schablone für die "Instanziierung" von unendlich vielen „parametrisierten Typen“

Parametrisierter Typ

- Entsteht durch „Typinstanziierung“

```
Node<MyData> n =  
    new Node<MyData>();
```

Typparameter wird in Typdeklarationen und bei der Objekterzeugung durch „Typargument“ ersetzt (wie bei einem Prozederaufruf)

Typargumente können beliebige Objekttypen sein (auch parametrisierte!)

```
Node<List<Integer>> n =  
    newNode<List<Integer>>();
```

Ein Typargument darf kein primitiver Typ sein (z.B. nicht „int“ statt „Integer“)

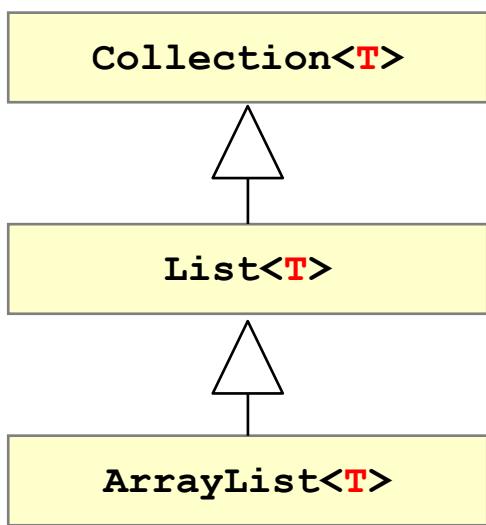
Generische Typen und Subtypbeziehung

Der schwierigste Teil des Generizitäts-Konzepts in OO Sprachen ...
... aber unerlässlich, wenn Sie es verstehen und effektiv einsetzen wollen

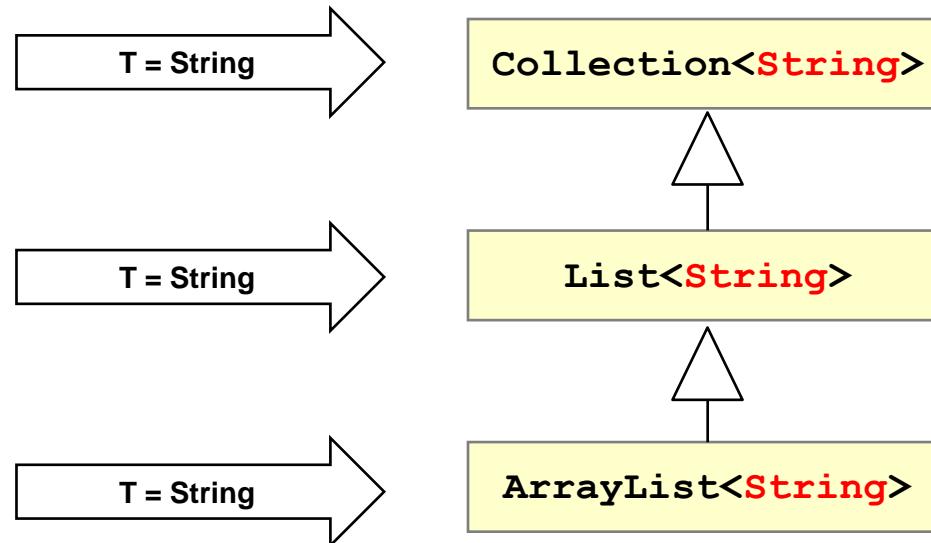
Gleiche Instanziierung von Untertypen liefert wieder Untertypen

Aus generischen Typen die voneinander Untertypen sind entstehen durch die Instanziierung eines Typparameters mit dem gleichen Typ wieder Untertypen

Generische Typ hierarchie



Parametrisierte Typ hierarchie



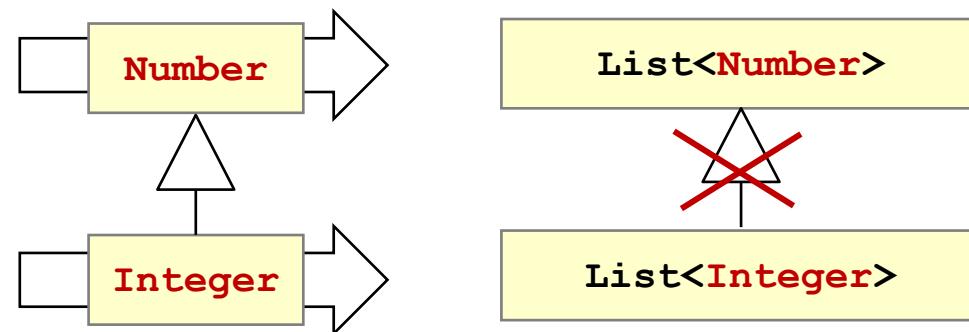
Verschiedene Instanziierung eines Typs liefert nie Untertypen

... auch dann nicht, wenn die für die gleiche Typvariable eingesetzten Typen in einer Untertypbeziehung stehen.

Generischer Typ



Parametrisierte Typen



„Given two concrete types A and B (for example, Number and Integer),
MyClass<A> has no relationship to MyClass,
regardless of whether or not A and B are related.“

→ <https://docs.oracle.com/javase/tutorial/java/generics/inheritance.html>

Verschiedene Instanziierung eines Typs liefert nie Untertypen

Seien **A** und **B** verschiedene Typen (keine Typvariablen)

Legale Typbenutzung

- Genau gleicher Typ auf rechter und linker Seite der Zuweisung
- Z.B. so ...

```
List<A> n = new List<A>();  
List<B> n = new List<B>();
```

- ... oder so:

```
List<A> n1;  
List<A> n2;  
n1 = n2;           // Legal  
n2 = n1;           // Legal
```

Illegal Benutzung

- Verschiedene Instanziierungen eines generischen Typs auf rechter und linker Seite der Zuweisung
- Z.B. so ...

```
List<A> n = new List<B>(); // ☹  
List<B> n = new List<A>(); // ☹
```

- ... oder so:

```
List<A> n1;  
List<B> n2;  
n1 = n2;           // Typfehler ☹  
n2 = n1;           // Typfehler ☹
```

Warum gibt es keine Subtypbeziehung zwischen Instanzen des gleichen Typs?

- Beispiel für Problem

```
List<String> ls = new ArrayList<String>(); // 1: OK
List<Object> lo = ls; // 2: Illegal in Java!
```

- Zeile 1 is legal (den ArrayList ist ein Subtyp von List).
- Bei Zeile 2 stellt sich die Frage: “Ist eine ‘Liste von Strings’ auch eine ‘Liste von Objekten’ ”?
- Instinktiv würde man sagen: “Na klar!”
- Wenn das wahr wäre könnten wir wie folgt weitermachen:

```
lo.add(new Object()); // 3: Objekt in von lo referenziertes Array einfügen
String s = ls.get(0); // 4: Versuch einer Stringvariablen ein (beliebiges)
                     // Objekt zuzuweisen, das kein String ist! ☹
```

- Fazit: Eine Variable vom Typ List<String> darf also nicht an eine vom Typ List<Object> zugewiesen werden ← daher keine Subtypbeziehung!

Raw Types

- Definition
 - ◆ Der Raw Type ist der Typ den man enthält, wenn man alle Typparameter löscht
 - ◆ Beispiel: Für `List<T>` ist `List` der Raw Type

- Zuweisung an RawType ist erlaubt!

```
List l = new ArrayList<String>(); // OK, mit Warnung:  
// List is a raw type. References to generic type List<E>  
// should be parameterized
```

- Warum?
 - ◆ Rückwärtskompatibilität zu nicht-generischem Code (vor Java 5)
 - ◆ RawType ist die nicht-generische Version des Typs
 - ◆ Nicht-Generische Typen sind keine RawTypes!

Zusammenfassung Generizität

- Generizität informell
 - ◆ "Code funktioniert auf Objekten verschiedenen Typs"
 - ◆ ... ohne dass sie in einer Subtypbeziehung stehen!
- Generizität technisch
 - ◆ Deklarationen von Klassen und Methoden sind mit Typvariablen parametrisiert
- Generischer Typ
 - ◆ Unbeschränkt
 - ◆ Beschränkt durch Bound(s) ← siehe Anhang
- Parametrisierter Typ
 - ◆ Instanziierung der Typparameter
 - ◆ Keine Subtypbeziehung zwischen verschiedenen Instanzen
 - ◆ Sonderfall "Wildcard," ← siehe Anhang
- Effiziente Implementierung in Java ← siehe Anhang

1.8 Inferierte und strukturelle Typen

Herleitung von Variablen-Typen aus der Struktur der enthaltenen Werte
...am Beispiel von TypeScript

Wo ist der Unterschied?

JavaScript

```
const user = {  
    name: "Hayes",  
    id: 0,  
};  
  
user.name = 123
```

TypeScript

```
user = {  
    name: "Hayes",  
    id: 0,  
};  
  
user.name = 123
```

- Laufzeitfehler!
- Compilerfehler!
- TypeScript „inferiert“ die Typen von Ausdrücken ☺ ...
- ... auch wenn das Programm keine Typ-Deklarationen enthält

Strukturelle Typinferenz

Herleitung des Typs eines Ausdrucks aus dem Programmcode.

- Inferenz aus Literalen

- ◆ Zahlen → jede Zahl ist ein eigener Typ
- ◆ Zeichenketten → ebenso
- ◆ Objektliteralen

```
user = {  
    name: "Hayes",  
    id: 0,  
};  
user.name = 123 // Compiler-Fehler
```

Typen werden (in TypeScript) aus der Struktur der Werte / Objekte hergeleitet.

- Propagierung via Zuweisungen

- ◆ Die linke Seite der Zuweisung hat den gleichen Typ wie die Rechte:

```
other = user // other hat Typ von user  
other.id = 123 // OK  
other.f(1) // Compiler-Fehler
```

- Propagierung via Funktionsaufrufen

- ◆ Jede Parameterübergabe ist auch eine Zuweisung
- ◆ Die Ergebnisrückgabe ist auch eine Zuweisung

Deklarierte Typen

- Man kann Typen auch explizit deklarieren
- Inferenz und explizite Deklaration können kombiniert werden
- Auch die Subtypbeziehung kann deklariert oder hergeleitet werden
 - ◆ point3a bietet das Interface von Point ← deklariert
 - ◆ und point3b bietet das Interface von Point ← inferiert

```
interface Point {  
    x: number;  
    y: number;  
}  
  
function logPoint(p: Point) {  
    console.log(` ${p.x}, ${p.y}`);  
}  
  
const point = { x: 12, y: 26 };  
logPoint(point); // "12, 26"
```

```
interface Point3D extends Point {  
    z: number;  
}  
  
declare const point3a : Point3D;  
logPoint(point3a); // ok, ist subtyp  
  
const point3b = { x: 12, y: 26, z: 89 };  
logPoint(point3b); // ok, ist subtyp
```

Strukturelle Typinferenz: Vor- und Nachteile

- Vorteile

- ◆ Typinformation im Programm → Sicherheit, Autocompletion, etc. ☺
- ◆ Kein Aufwand für den Programmierer
- ◆ „Normale“ JavaScript-Programme können überprüft werden

- Nachteil

- ◆ Zufällige Ersetzbarkeitsbeziehungen ohne konzeptuelle Entsprechung
- ◆ Aber: Risiko gering, dass das bei nicht-trivialen Typen / Schnittstellen auftritt

```
class Singer {  
    hit() { // greatest success song  
    }  
}  
  
class Golfer {  
    hit() { // hit the ball  
    }  
}  
  
let obj: Singer = new Golfer();  
obj.hit();
```

Typ-Vereinigung

Effekt

- Mengenvereinigung
- Durchschnitt ihrer Interfaces – man kann nur das aufrufen, was alle Elemente in ihrer Schnittstelle haben
 - ◆ Im Beispiel: `length()` gibt es für Strings und Arrays – OK!

Ein „Typ-Alias“ = Ein Name für einen strukturellen Typ

Syntax für Typ-Vereinigung

```
type LockStates = "locked" | "unlocked";  
type UngeradeUnter10 = 1 | 3 | 5 | 7 | 9;
```

```
function getLength(obj: string | string[]) {  
    return obj.length;  
}
```

Typ-Durchschnitt

Effekt

- Mengendurchschnitt
- Vereinigung der Interfaces
 - ◆ Die Durchschnittsmenge bietet das Interface beider Mengen!

Nutzen

- Gemeinsames herazufaktorisieren
 - ◆ Hier wird ErrorHandling als eigener Typ deklariert, ...
 - ◆ ... und bei Bedarf mit jedem Typ der ErrorHandling braucht kombiniert, durch den &-Operator

```
interface ArtworksData {  
    artworks: { title: string }[];  
}  
  
interface ArtistsData {  
    artists: { name: string }[];  
}  
  
// Konsistente Fehlerbehandlung für beide obigen Typen.  
// Trennung von Fehlerbehandlung und anderen „Belangen“:  
  
interface ErrorHandling {  
    success: boolean;  
    error?: { message: string };  
}  
  
// Bei Bedarf können Typen deklariert  
// werden, die beides können:  
  
type ArtworksResponse = ArtworksData & ErrorHandling;  
type ArtistsResponse = ArtistsData & ErrorHandling;  
  
const handleArtistsResponse = (response: ArtistsResponse) => {  
    if (response.error) {  
        console.error(response.error.message);  
        return;  
    }  
  
    console.log(response.artists);  
};
```

Syntax für
Typ-Durchschnitt

Zusammenfassung OOP

(Stichworte für Klausurvorbereitung)

- Prototypbasiert (= objektbasierte) Sprachen
 - ◆ Eigenständige Objekte, dynamische Objektstruktur
 - ◆ Delegation (= objektbasierte Vererbung)
- Klassenbasierte Sprachen
 - ◆ Instanziierung / Garantierte, feste Objektstruktur
 - ◆ Statische Vererbung
- Gemeinsame Prinzipien
 - ◆ Objekte, Nachrichten / dynamisches Binden, Vererbung, Objekterzeugung
- Typen und Untertypen
 - ◆ Signaturen und Schnittstellen
 - ◆ Ersetzbarkeit!!!
- Implementierung
 - ◆ Objekt- und vTable-Layout
 - ◆ Umsetzung von Vererbung, Hiding, Overriding, Overloading, dynamic Binding
- Generizität
 - ◆ Generische und parametrisierte Typen, Subtypbeziehung parametrisierter Typen

Anhang

Ab hier "nur" Erinnerungen an was Sie in OOSE / AIPro gehört haben
Folgendes wird nicht erneut geprüft (obiges schon)

Beschränkte Generizität ▶ Motivation

- Eine Typvariable **T** kann irgendeinen unbekannten Typ darstellen
- Wir wissen daher nicht, welche Operationen er enthält
- Manchmal brauchen wir das aber, z.B. um `Node<T>` folgendes hinzuzufügen:

```
public class Node<T> {  
    T data;  
  
    public MyClass(T v) {  
        data = v;  
    }  
  
    public void setData(T v) {  
        data = v;  
    }  
  
    public T getData() {  
        return data;  
    }  
}
```

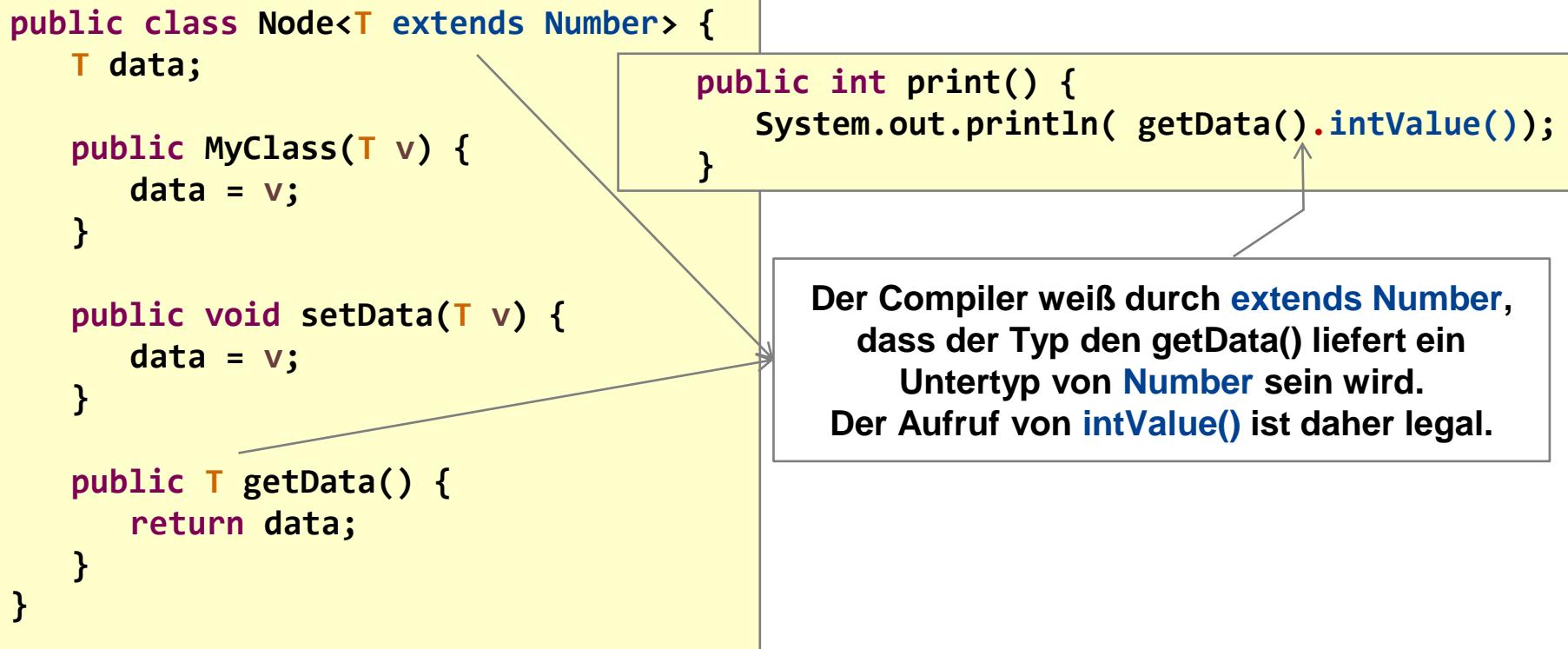
```
        public int print() {  
            System.out.println( getData().intValue());  
        }
```

Der Compiler muss wissen, dass die `getData()`-Methode ein Objekt liefert, dass eine `intValue()`-Methode hat, sonst erlaubt er uns nicht sie aufrufen.

Wir brauchen also einen Weg anzugeben, dass **T** ein Subtyp von **Number** sein muss (damit er `intValue()` enthält).

Beschränkte Generizität ▶ Definition

- **Syntax:** Einer Typvariablen wird eine obere Schranke zugeordnet
- **Semantik:** Die legalen Werte für die Typvariable werden auf Untertypen der Schranke eingeschränkt (daher der Name "Beschränkte Generizität").



Beschränkte Generizität ▶ Multiple Schranken

Jede Typvariable kann mehrere obere Schranken haben.

- Syntax: Schranken werden mit & getrennt

```
public class Example<T extends Person & Cloneable & Printable> {  
    ...  
}
```

- Statische Semantik
 - ◆ Der Typ durch den die Typvariable ersetzt wird muss ein Untertyp aller Schranken sein
 - ◆ Falls eine der Schranken eine **Klasse** ist, muss sie an erster Stelle stehen

Das Prinzip gilt gleichermaßen für generische Typen und generische Methoden.

Generische Methoden ▶ Motivation (1)

Was machen wir, wenn `print()` nicht in `Node<T>` sondern in einer anderen Klasse definiert werden soll?

```
public class Node<T> {  
    T data;  
  
    public MyClass(T v) {  
        data = v;  
    }  
  
    public void setData(T v) {  
        data = v;  
    }  
  
    public T getData() {  
        return data;  
    }  
}
```

```
public class Other {  
    ...  
    public int print(Node<Number> n) {  
        System.out.println(n.getData().intValue());  
    }  
}
```

- Folgendes wäre legal ...
- ... es würde aber nur für `Number` funktionieren, nicht für Untertypen von `Number` (z.B `Integer`)
 - ◆ ... den `Node<Integer>` ist ja kein Subtyp von `Node<Number>` (siehe Seite 8-10)

Generische Methoden ▶ Motivation (2)

Was machen wir, wenn `print()` nicht in `Node<T>` sondern in einer anderen Klasse definiert werden soll?

```
public class Node<T> {  
    T data;  
  
    public MyClass(T v) {  
        data = v;  
    }  
  
    public void setData(T v) {  
        data = v;  
    }  
  
    public T getData() {  
        return data;  
    }  
}
```

- Folgendes wäre ebenfalls legal ...

```
public class Other<E extends Number> {  
    ...  
    public int print( Node<E> n ) {  
        System.out.println(n.getData().intValue());  
    }  
}
```

- ... und würde auch für Subtypen von **Number** funktionieren.
- Wenn die Typvariable **E** jedoch nur von der `print()`-Methode genutzt wird, können wir sie auch lokal zu der Methode deklarieren □

Generische Methoden ▶ Definition

- Eine Methode ist generisch, wenn sie eigene (lokale) Typvariablen definiert.
- Die lokalen Typvariablen stehen in spitzen Klammern vor dem Ergebnistyp:

```
public class Node<T> {  
    T data;  
  
    public MyClass(T v) {  
        data = v;  
    }  
  
    public void setData(T v) {  
        data = v;  
    }  
  
    public T getData() {  
        return data;  
    }  
}
```

- Beispiel ...

```
public class Other {  
    ...  
    public <E extends Number> int print(Node<E> n) {  
        System.out.println(n.getData().intValue());  
    }  
}
```

- Dies funktioniert durch die Angabe "extends Number" auch für Subtypen von Number (wie das Beispiel auf der vorherigen Folie)

Wildcards (1)

- Wenn eine Methode eine lokale Typvariable nur ein einziges mal verwendet, kann man die Deklaration der Typvariablen weglassen und als Typangabe die "Wildcard" ? benutzen

```
public class Node<T> {  
    T data;  
  
    public MyClass(T v) {  
        data = v;  
    }  
  
    public void setData(T v) {  
        data = v;  
    }  
  
    public T getData() {  
        return data;  
    }  
}
```

- Beispiel ...

```
public class Other {  
    ...  
    public int print(Node<? extends Number> n) {  
        System.out.println(n.getData().intValue());  
    }  
}
```

- Die Schranke "extends Number" wird dabei in die Typangabe integriert

Wildcards (2)

- Die "Wildcard" **?** ist ein unbenannter (= "anonymer") Typ, keine Typvariable.
- Folgende beide Definitionen sind äquivalent

```
public <A extends Number, B extends Integer>
    int print(Node<A> n, List<B> l);
```

- ◆ In der obigen Definition wurden die potentiell unterschiedlichen konkreten Typen durch zwei unterschiedliche Typvariablen dargestellt
- ◆ Im Folgenden statt vieler einmalig verwendeter Typvariablen A, B, ... immer nur "?"

```
public int print(Node<? extends Number> n, List<? extends Integer> l);
```

- ◆ Jedes Vorkommnis von "?" kann implizit einen anderen Typ darstellen

- Wildcard legal als Typ von
 - ◆ Parameter
 - ◆ lokaler Variable
 - ◆ Feld ← anstelle einer in einer Klasse einmalig verwendeten Typvariablen

Wildcards (3)

- Die "Wildcard" ? kann nach oben oder nach unten beschränkt werden:
 - ◆ <? extends Number> = Number oder ein Untertyp von Number
 - ◆ <? super Number> = Number oder ein Obertyp von Number

Faustregel zur Verwendung von **extends** und **super**

Unterscheidung von Input-Parametern und Output-Parametern:

- Input-Parameter: <? extends UpperBound>
- Output-Parameter: <? super LowerBound>
- Beispiel: Generisches Kopieren einer Liste von Elementen eines Typs T

```
void <T> copy( List<? extends T> source, List<? super T> destination) {  
    ...  
}
```

- ◆ Parameter "source" = input
- ◆ Parameter "destination" = output

Konkrete Typinstanzierung versus Wildcards

Beispiel

Unsere Knoten-Klasse. Wir beziehen uns im Folgenden darauf.

```
public class Node<T> {
    T data;

    public MyClass(T v) {
        data = v;
    }

    public void setData(T v) {
        data = v;
    } // <- Lesezugriff

    public T getData() {
        return data;
    } // <- Schreibzugriff
}
```

Prinzip (Kurzfassung)

- **Typinstanzierung** erlaubt nur einen Typ, davon aber alle Operationen
- **Wildcards** erlauben auch Unter- oder Obertypen. Damit das typsicher ist, müssen sie alle schreibenden Operationen (die etwas anderes als **null** zuweisen) verbieten ← siehe Seite 10.
- Warum das so ist schauen wir uns auf der nächsten Folie an, anhand des `Node<T>`-Beispiels

Typinstanzierung versus Wildcards

Typinstanziierung

- Nur der **angegebene Typ** kann genutzt werden ← siehe Seite 10:

```
Node<Number> nn = new Node<Number>(1.0);
Node<Integer> ni = new Node<Integer>(1);
nn = ni; // Compiler-Fehler, kein Supertyp
```

- Der Compiler weiß, dass z.B. **nn** ein Objekt von genau dem Typ `Node<Number>` sein wird
- Daher ist der Aufruf aller Operationen dieses Typs zulässig

```
nn.getMember();
nn.setMember(100.0);
```

Wildcard

- Zusätzlich zu dem als Schranke angegebenen Typ kann ein Obertyp oder Untertyp genutzt werden:

```
Node<? extends Number> nn =
    new Node<Integer>(1);
```

- Der Compiler weiß daher nicht, was für ein Objekt **nn** referenzieren wird
- Daher erlaubt er nur Aufrufe von lesenden Operationen

```
nn.getMember();
nn.setMember(100.0); // Compiler-Fehler
```

Beschränkte Generizität wirkt ähnlich

- Beispiel: Beschränkte Typvariable T

```
<T extends Number> void print(Node<T> c) {  
  
    System.out.println(c.getMember().intValue());  
  
    c.setMember(30); // Fehler:  
    // The method setMember(T) in the type Node<T>  
    // is not applicable for the arguments (int)  
}
```

- Auch hier "Schreibverbot", aus gleichem Grund
 - ◆ Verhindern, in Objekt eines unbekannten Typs etwas kaputt zu machen!
- Keine Überraschung
 - ◆ Wildcards sind ja Sonderfall für beschränkte Typvariablen die nur ein mal vorkommen
 - ◆ Was schon im Spezialfall ein Problem ist, ist es im allgemeineren Fall auch

Implementierung generischer und parametrisierter Typen in C++ ("Templates")

Generischer Typ

```
public class Node<T> {  
    T data;  
    Node<T> next;  
    void setData(T) ...  
    T getData() ...  
}
```

- Template wird nicht übersetzt!!!
- Nur die daraus instanzierten parametrisierten Typen werden übersetzt, jeder einzeln ☹

Parametrisierter Typ

```
Node<MyData> n =  
    new Node<MyData>();
```

```
Node<List<Integer>> n =  
    new Node<List<Integer>>();
```

- "template expansion" für jede Instanzierung ☹☹☹

```
public class Node_MyData {  
    MyData data;  
    Node<MyData> next;  
    void setData(MyData) ...  
    MyData getData() ...  
}
```

```
Node_MyData n = new  
Node_MyData();
```

```
public class  
Node_List_Integer {  
    List_Integer data;  
    Node<MyData> next;  
    void setData(List_Integer)  
    ...  
    List_Integer getData() ...  
}
```

```
Node_List_Integer n  
= new  
Node_List_Integer();
```

➤ lange Compilierung, riesiger Code

Implementierung generischer und parametrisierter Typen in Java

Generischer Typ

```
public class Node<T> {  
    T data;  
    Node<T> next;  
    void setData(T) ...  
    T getData() ...  
}
```

Parametrisierter Typ

```
Node<MyData> n =  
    new Node<MyData>();
```

```
Node<List<Integer>> n =  
    new Node<List<Integer>>();
```

- "type erasure" – Typvariable **T** wird durch Typ ersetzt
 - ◆ unbeschränkt → "Object" statt **T**
 - ◆ beschränkt → Schranke statt **T**
- "bridge methods" – in Untertypen die **T** (weiter) beschränken
 - ◆ enthalten Casts auf die lokale Schranke **S** und Weiterleitung an Methode mit Parametertyp **S**

- Casts auf den instanzierten Typ
 - ◆ ... z.B. auf **MyData**
 - ◆ ... so wie wir es manuell machen würden (siehe Seite 78)
- Schnelle Compilierung
- Kleiner ByteCode

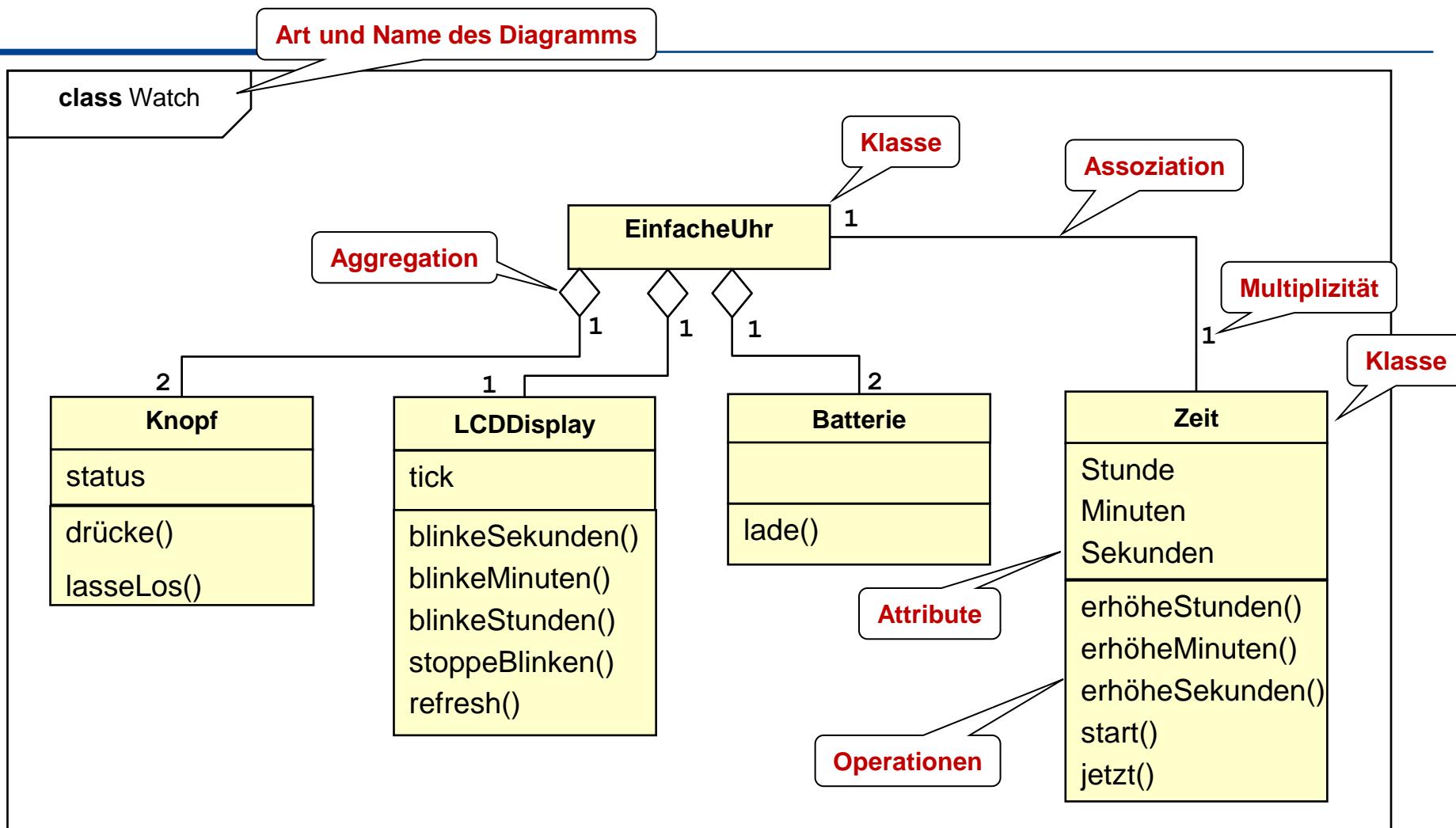
Fragen?

Kapitel 2 „Objektorientierte Modellierung“

Stand: 6.11.2023

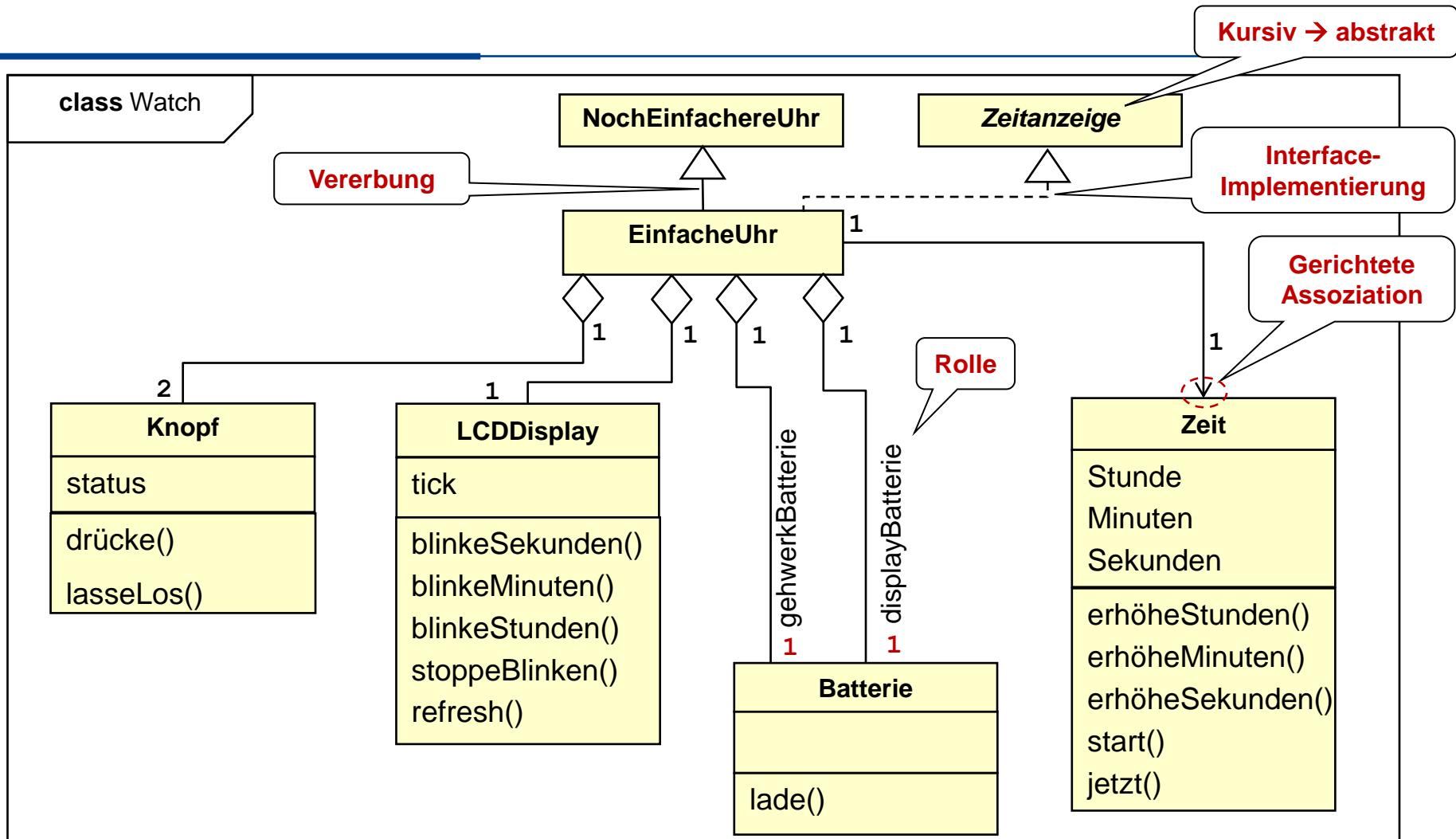
-
- 2.1 CRC-Cards
 - 2.2 Design by Contract
 - 2.3 Modellierungsprinzipien (M. Page-Jones)
 - 2.4 Abstraktheit und Stabilität
 - 2.5 Modellierungsprinzipien (R. C. Martin)

UML-Kurzübersicht ► Klassen-Diagramme



Klassendiagramme repräsentieren die Struktur eines Systems

UML-Kurzübersicht ► Klassen-Diagramme



Klassendiagramme repräsentieren die Struktur eines Systems

2.1. Schnittstellen-Identifikation: CRC-Cards

OO Modellierung: Class-Responsibility-Collaboration (CRC) Karten

- Class
 - ◆ Welche Klasse betrachten wir?
 - Responsibility
 - ◆ Beschreibt die Aufgaben der Klasse
 - Collaboration
 - ◆ Welche Klassen werden für die Aufgabe gebraucht? (evtl. auch die eigene...)
 - Nutzen
 - ◆ regt Diskussionen an
 - ◆ lenkt den Blick auf das Wesentliche
 - ◆ Hilft auf Schnittstelle statt Daten zu fokussieren
 - ◆ beugt Konzentration von zu vielen Verantwortlichkeiten an einer Stelle vor
- ⇒ “Schreib nie mehr auf, als auf eine Karte passt!”
 - eher die Klasse in zwei Klassen / Karten aufteilen!
 - 1 Karte = höchstens DIN A5 groß (halbes DIN A4 Blatt)

Bestellung	
Prüfe ob Artikel auf Lager	Artikel, Lager
Bestimme Preis	Artikel
Prüfe Zahlungseingang	Kunde, Kasse
Ausliefern	Logistik

Class (Klassen-Name)

Responsibility (Aufgaben)

Collaboration (Zusammenarbeit)

Einsatz von CRC-Cards

- Wann
 - ◆ In frühem Objektentwurf (bei DOM-Erstellung) und evtl. in Analyse
- Wozu
 - ◆ Identifikation von Klassen, Operationen und „Kollaborations“-Beziehung
 - ◆ Einzig relevante Beziehung ist „Kollaboration“ mit anderen Klassen
 - ◆ Fokus liegt auf Operationen und **was sie brauchen** um zu funktionieren
- Danach: **Woher bekomme ich was ich brauche?**
 - ◆ „Vom Aufrufer“ → Parametern und Ergebnissen
 - ◆ „Das weiss ich selbst“ → Instanzvariablen

CRC-Cards sind sehr hilfreiche für Anfänger in der OO Modellierung,
da **verhaltenszentriertes Denken gefördert** wird!

Fokus auf Schnittstellen!!!

Was kommt nach CRC-Cards?

- Ausarbeitung der Beziehungen, Kardinalitäten, ...
 - ◆ → Herkömliche Datenmodellierung (IS-Vorlesung)
- Verfeinerung des Verhaltens
 - ◆ → Design by Contract
- (Re)Strukturierung des Objektmodells
 - ◆ → Objekt-Orientierte Modellierungs-Prinzipien

Spezifikation der Schnittstellen

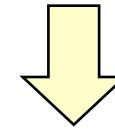
In Anforderungsanalyse: Identifikation von

- Attributen - ohne ihren Typ anzugeben
- Operationen - ohne Typangaben

Später: Hinzufügen von Typangaben

- Typ
 - ◆ Klasse
 - ◆ Interface
 - ◆ primitiver Typ
- Signatur →
 - ◆ Methodename
 - ◆ Parametertypen
- Ergebnistyp(en) sind *nicht* Teil der Signatur

Hashtable
-numElements
+put() +get() +remove() +containsKey() +size()



Hashtable
-numElements: int
+put(key: Object, entry: Object) +get(key: Object): Object +remove(key: Object) +containsKey(key: Object): boolean +size(): int

Warum reichen Signaturen nicht aus?

put(key: Object, entry:Object)

- Sie sagen nur etwas darüber, wie man eine Operation auruft.

- Sie sagen nicht, was die aufgerufene Operation macht.

HIER

- Verhaltenszusicherungen (*Contracts*)

- Sie unterscheiden nicht verschiedene Aufrufende

SYSTEMENTWURF

- Zugriffsrechte (=/= Sichtbarkeiten!)

- Sie sagen nicht, was der spezifizierte Typ selber von seiner Umgebung braucht, um die angebotenen Operationen realisieren zu können.

- Benutzte Schnittstellen (*Required Interfaces*)

- Sie sagen nicht, in welcher Reihenfolge verschiedene Operationen des gleichen Objektes aufgerufen werden müssen.

- Interaktionsspezifikation (*Behaviour Protocols*)

2.2 Schnittstellen-Spezifikation: Verhalten → "Design by Contract"

Design by Contract (DBC)

- Behauptung (Assertion)
 - ◆ Logische Aussage, die wahr sein muss
 - ◆ Macht die Annahmen explizit, unter denen ein Design funktioniert
 - ◆ Lässt sich automatisch überprüfen
- Vertrag (Contract)
 - ◆ Menge aller Assertions, die festlegen, wie zwei Partner interagieren
 - ◆ Auftraggeber = aufrufende Operation / Klasse
 - ◆ Auftragnehmer = aufgerufene Operation / benutzte Klasse
- Arten von Assertions
 - ◆ Invarianten ← legale Zustände
 - ◆ Vorbedingungen ← legales Verhalten
 - ◆ Nachbedingungen ← legales Verhalten

Design by Contract: Vorbedingungen (‘Preconditions’)

- Definition
 - ◆ Eine Precondition ist eine Voraussetzung dafür, dass eine Operation korrekt ausgeführt werden kann
- Technische Realisierung
 - ◆ Assertion die wahr sein muss, bevor eine Operation ausgeführt wird.
- Beispiel: Operation “Ziehe die Wurzel einer Zahl”
 - ◆ Signatur: `squareRoot(input int)`
 - ◆ Pre-condition: `input >= 0`
- Verantwortlichkeiten
 - ◆ Der aufgerufene Code formuliert die Vorbedingung
 - ◆ Der aufrufende Code muss die Einhaltung der Vorbedingung sicherstellen

Design by Contract: Nachbedingung (‘Postcondition’)

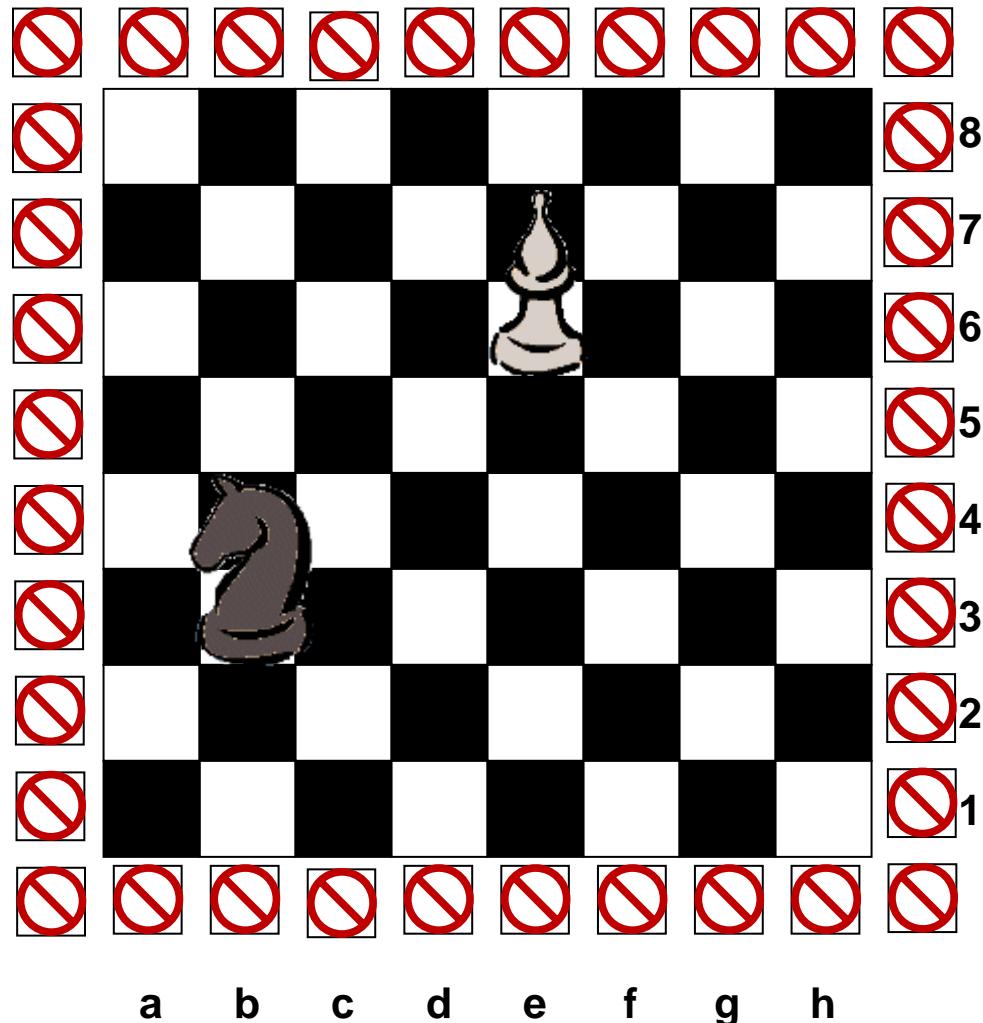
- Definition
 - ◆ Postcondition beschreibt **deklarativ** das Ergebnis eines korrekten Aufrufs
 - ◆ Sagt aus, **was** getan wird, nicht **wie** es getan wird
- Technische Realisierung
 - ◆ Assertion die wahr sein muss, *nachdem* eine Operation ausgeführt wird.
- Beispiel: Operation “Ziehe die Wurzel einer Zahl”
 - ◆ Signatur: **squareRoot(input i)**
 - ◆ Post-condition: **input = result * result**
- Verantwortlichkeiten
 - ◆ Der aufgerufene Code formuliert die Nachbedingung
 - ◆ Der aufgerufene Code garantiert die Einhaltung der Nachbedingung ... aber nur wenn die Vorbedingung wahr ist!

Design by Contract: Klasseninvariante (Class Invariant)

- Definition
 - ◆ Invariante beschreibt deklarativ legale Zustände von Instanzen einer Klasse
- Technische Realisierung
 - ◆ Assertion die für alle Instanzen einer Klasse immer wahr ist.
- Beispiel: Klasse eines Benutzerkontos
 - ◆ Invariante: Der Kontostand ist immer die Summe aller Buchungen
 - ◆ “kontostand == summe(buchungen.betrag ())”
- Verwendung
 - ◆ Invarianten werden verwendet, um Konsistenzbedingungen zwischen Attributen zu formulieren.
- Verantwortlichkeiten
 - ◆ Diese Bedingungen einzuhalten liegt in der gemeinsamen Verantwortlichkeit aller Operationen einer Klasse.

DBC spezifiziert legale Zustände

- Zustand zu einem Zeitpunkt = Die Werte aller Instanzvariablen
 - ◆ Mindestens der eigenen Var.
 - ◆ Konzeptionell ist auch der Zustand aller aggregierten Teil-Objekte eigener Zustand
- legale Zustände werden durch Klasseninvarianten definiert
 - ◆ Dame und Springer haben gleichen Zustandsraum (jedes Schachbrett-Feld)
 - ◆ Figuren dürfen Spielfeld nicht verlassen
 - ◆ Legale Schachbrett-Zustände: pro Feld max. eine Figur



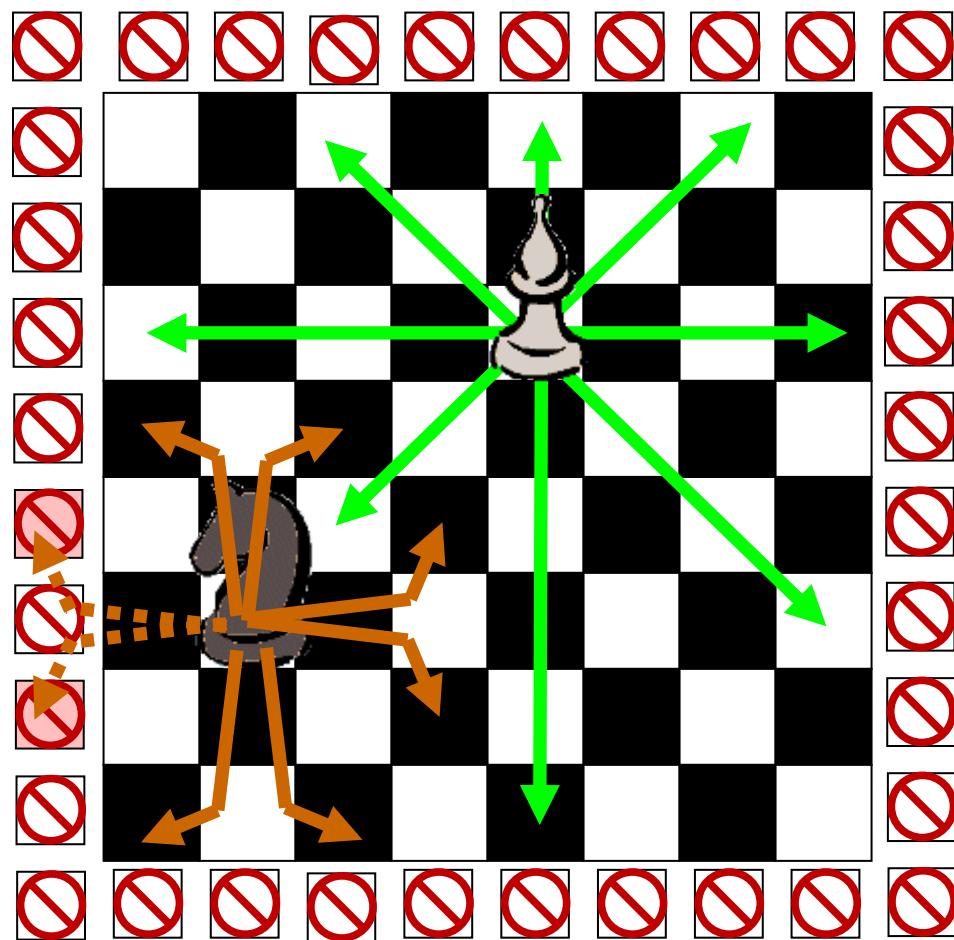
DBC spezifiziert legales Verhalten

- **Verhalten** = Zustandsübergänge und daran gekoppelte Aktionen

- ◆ Dame bewegt sich horizontal und diagonal beliebig weit
- ◆ Springer bewegt sich L-förmig

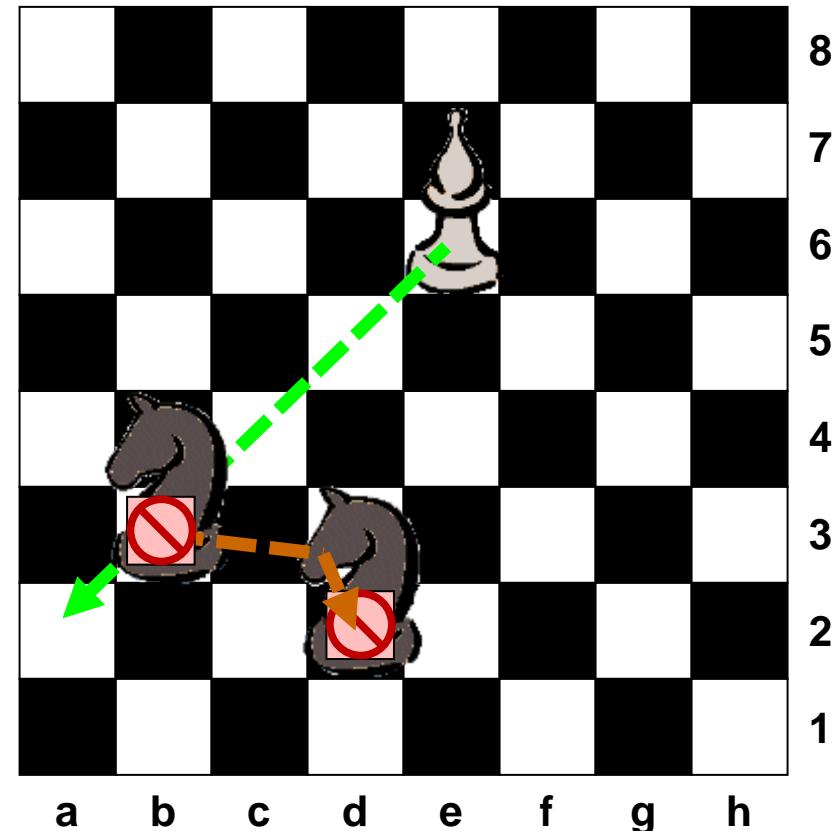
- Legales Verhalten

- ◆ Mindestens:
Zustandsübergänge die nicht in illegale Zustände führen
- ◆ Meist gibt es zusätzliche applikationsspezifische Bedingungen („Constraints“)
→ s. nächste Folie



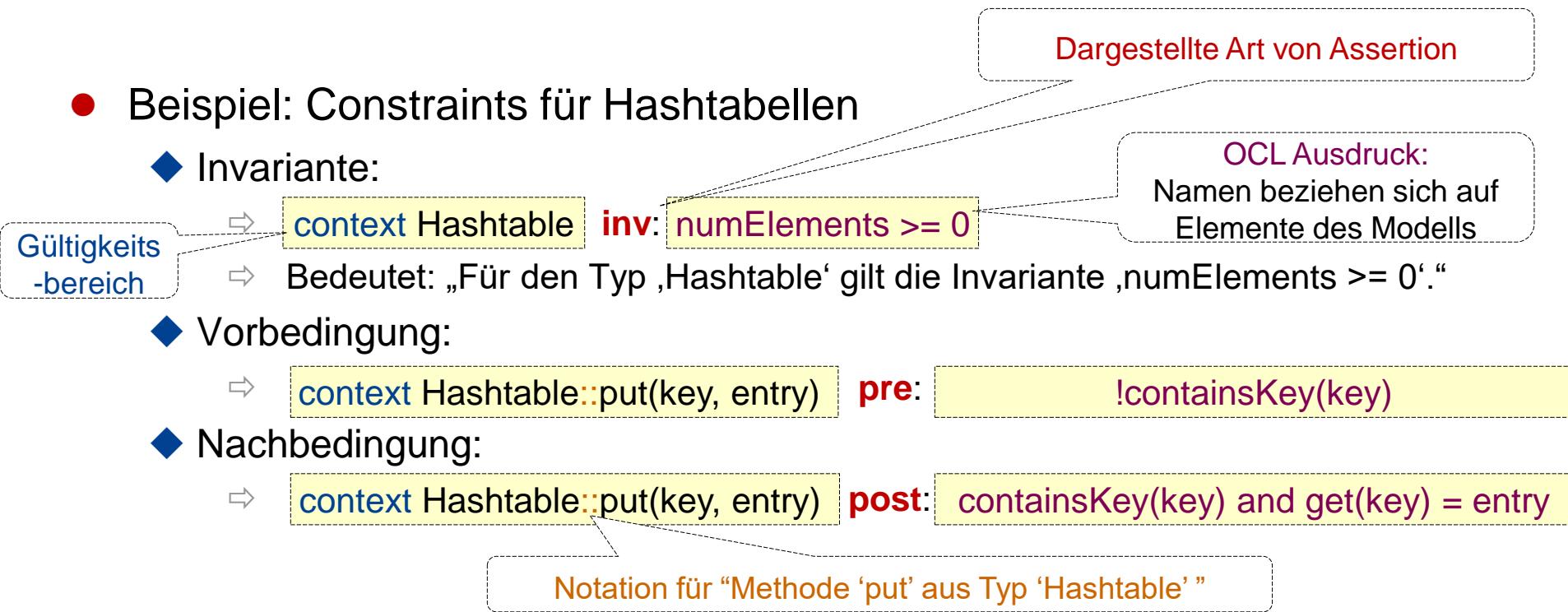
DBC spezifiziert legales Verhalten

- DBC definiert legale Zustandsübergänge durch Vor- und Nachbedingungen
- Beispiel: “Keine Figur ausser dem Springer kann über andere hinüberspringen”
 - ◆ Vorbedingung der Operation “zieheNach(Feld)": Alle Felder über die ich gehen muss sind frei
- Beispiel: „Eigene Figuren schlagen ist verboten“
 - ◆ Vorbedingung der Operation “zieheNach(Feld)": Feld nicht von eigener Figur belegt



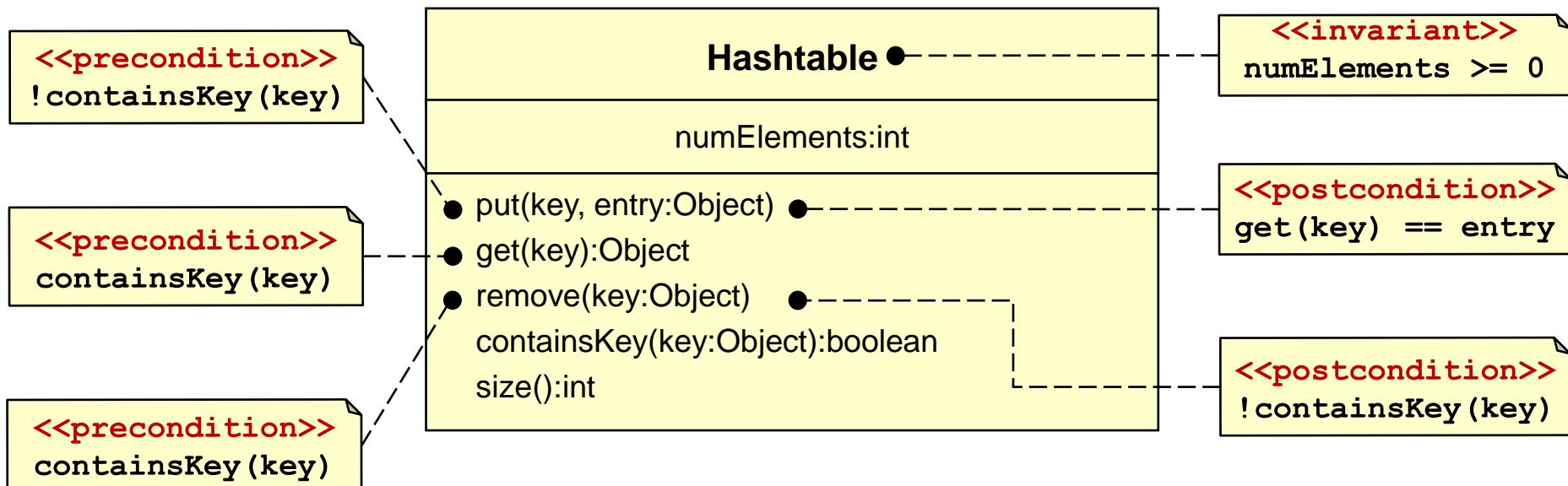
Formulierung von Kontrakten in UML: → OCL (Object Constraint Language)

- OCL erlaubt die formale Spezifikation von Bedingungen (Constraints) für Werte von Modellelementen
- Ein Constraint wird in Form eines OCL Ausdrucks angegeben, der entweder den Wert wahr oder falsch hat.
 - ◆ „Constraint“ in OCL = „Assertion“ in DBC



Formulierung von Kontrakten in UML

- Jede Assertion kann auch als Notiz dargestellt und an das jeweilige UML Element “angehängt” werden.
 - ◆ Die Art der Assertion wird als Stereotyp (in doppelten spitzen Klammern) angegeben ← was Stereotypen genau sind besprechen wir noch



Besondere Schlüsselworte in Nachbedingungen („postconditions“)

- Problem
 - ◆ In Nachbedingungen muss manchmal der Zustand vor und nach der Ausführung der Operation unterscheiden werden.
- Syntax
 - ◆ **expr@pre** = Der Wert des Ausdrucks **expr** vor Ausführung der **Operation** auf die sich die Nachbedingung bezieht.
 - ◆ **expr@post** = Der Wert des Ausdrucks **expr** nach Ausführung der **Operation** auf die sich die Nachbedingung bezieht.
- Beispiele

Wenn nichts explizit angegeben ist bezieht man sich **in Nachbedingungen auf den Nachzustand** und **in Vorbedingungen auf den Vorzustand**.

 - ◆ **context Person::birthdayHappens() post:** **age = age@pre + 1**
 - ◆ **age@pre** bezieht sich hier auf den Wert des Feldes **age** vor Begin der Ausführung der Operation **birthdayHappens()** → „Alter Wert“
 - ◆ **a.b@pre.c** – Alter Wert des Feldes b von a. Darin der neue Wert des Feldes c.
 - ◆ **a.b@pre.c@pre** – Alter Wert des Feldes b von a. Darin der alte Wert des Feldes c.

Weitere OCL-Schlüsselworte

- **result**
 - ◆ Bezug auf den Ergebniswert einer Operation (in Nachbedingungen).
- **self**
 - ◆ Bezug auf das ausführende Objekt (wie „this“ in Java).

Literatur

- Gute, detaillierte Einführung auf deutsch
<https://st.inf.tu-dresden.de/files/teaching/ss09/stII09/OCL.pdf>
- OCL 2.0 Specification, Version 2.0, Date: 06/06/2005,
<http://www.omg.org/docs/ptc/05-06-06.pdf> <-- kein Tutorial sondern sehr technische Spezifikation, eher für Entwickler von UML-Tools gedacht

Design by Contract: Sprachunterstützung

- Java
 - ◆ Assertions werden ab JDK 1.4 unterstützt
 - ◆ Formulierung von postconditions und invariants ist damit möglich
- Contract4J
 - ◆ Contracts als assertions für JDK 1.5 (Java 5) formuliert
 - ◆ <http://www.contract4j.org>
- Eiffel
 - ◆ Kontrakte voll unterstützt (preconditions, postconditions und invariants)
 - ◆ [http://en.wikipedia.org/wiki/Eiffel_\(programming_language\)](http://en.wikipedia.org/wiki/Eiffel_(programming_language))
- Spec#
 - ◆ C# mit voller Kontraktunterstützung und vielen anderen Erweiterungen
 - ◆ <http://research.microsoft.com/specsharp/>
- Andere Sprachen
 - ◆ Kontrakte zumindestens in der Dokumentation explizit machen
- In allen Sprachen
 - ◆ Kontrakte als wichtiges Kriterium beim Entwurf mit beachten → Ersetzbarkeit

Subtyping / Ersetzbarkeit und Kontrakte

B ist ein Subtyp von **A**



Instanzen von **B** sind immer für Instanzen von **A** einsetzbar



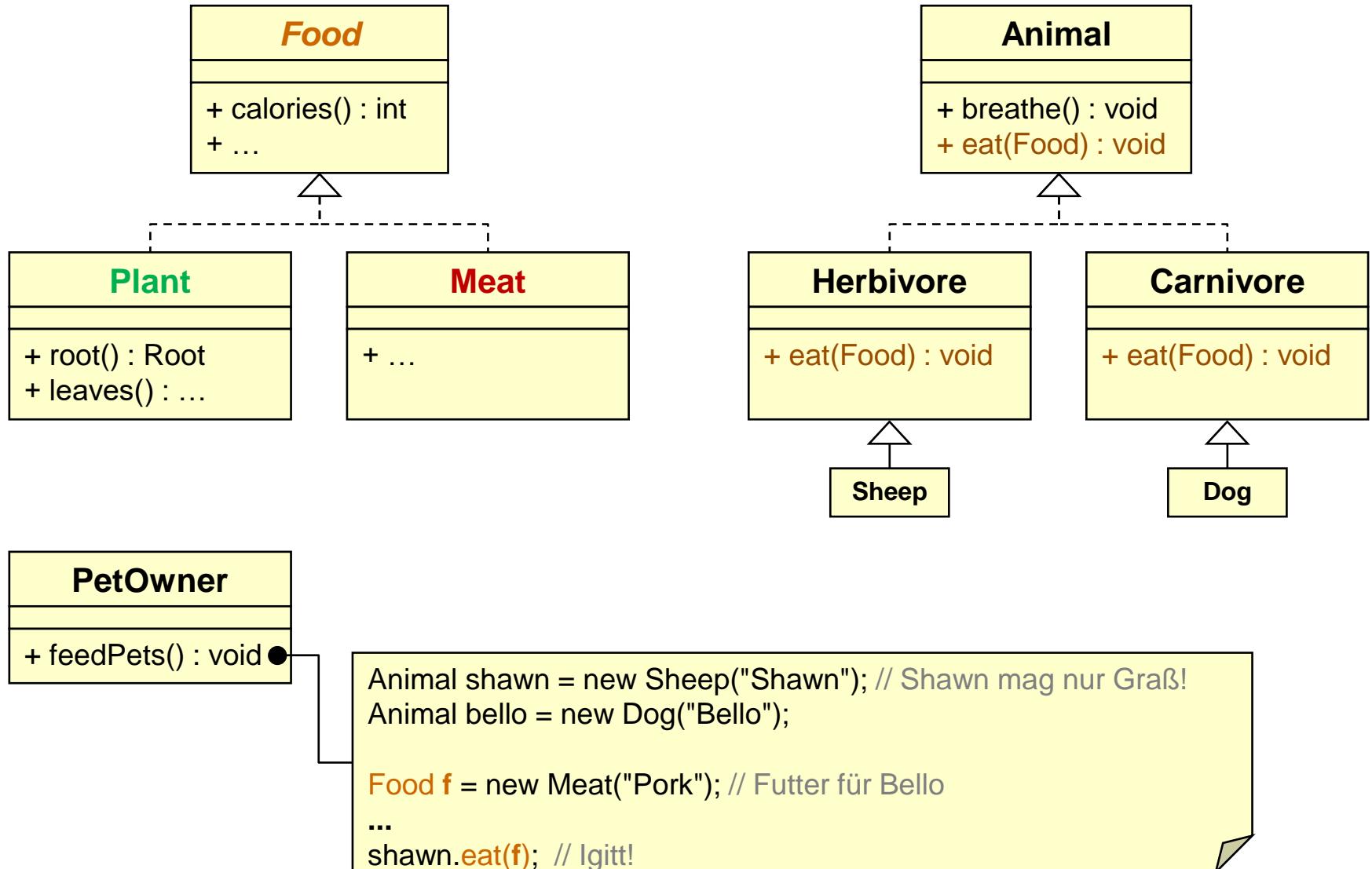
Instanzen von **B** bieten mindestens und fordern höchstens
das gleiche wie Instanzen von **A**



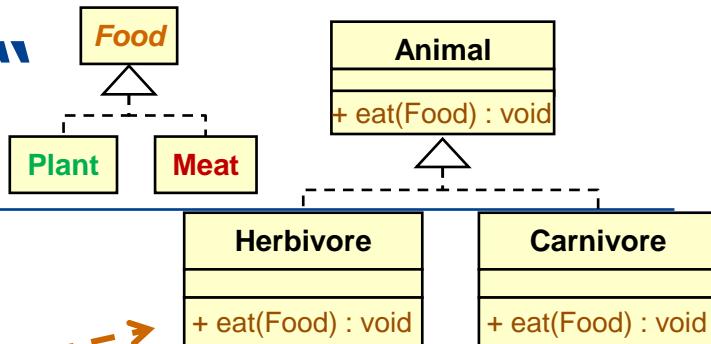
Instanzen von **B** haben mindestens alle Methoden von **A**,
und zwar mit gleichen oder stärkeren Nachbedingungen
und gleichen oder schwächeren Vorbedingungen

Was aber wäre schlimm daran, wenn das nicht gilt?
Was, wenn z.B. **B** stärkere Vorbedingungen fordert?
→ Erläuterung folgt anhand eines Beispiels

Beispiel: „Tier frisst Futter“



Beispiel: „Tier frisst Futter“

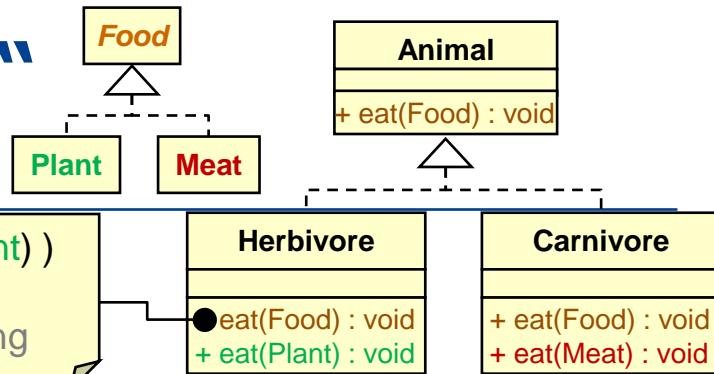


Erläuterungen zum Beispiel

- Das ist der Grund, dass es eine eigene Methode `eat(Food)` in der Klasse `Herbivore` geben muss.
- Sie überschreibt die Methode mit gleicher Signatur aus der Oberklasse, ...
 -, basiert allerdings auf der **stärkeren Annahme / Vorbedingung**, dass es nur pflanzliches Futter gibt
 - Daher sind ist der Typ `Herbivore` eine Erweiterung aber **kein Subtyp** von `Animal` (da nicht überall verwendbar wo `Animal` erwartet wird)



Beispiel: „Tier frisst Futter“



Erläuterungen (Fortsetzung)

- Das ist der Grund, dass es eine eigene Methode **eat(Food)** in der Klasse **Herbivore** geben muss.
- Sie überschreibt die Methode mit gleicher Signatur aus der Oberklasse, ...

```
if ( food instanceof(Plant) )
    eat( (Plant) food)
else // Fehlerbehandlung
```

- ..., und muss damit umgehen, dass **Herbivore** **kein Untertyp** ist:

- ◆ instanceof-Test
- ◆ bei Misserfolg Fehlerbehandlung
- ◆ für Erfolgsfall wird oft eine eigene Methode **eat(Plant)** definiert
 - ⇒ Wo statisch bekannt ist, dass ein **Plant** übergeben wird, wird direkt diese Operation vom Compiler ausgewählt
 - ⇒ Aufruf in **eat(Food)** erfordert cast des Parameters zu **Plant**



```
Animal shawn = new Sheep("Shawn"); // Shawn mag nur Graß!
Animal bello = new Dog("Bello");

Food f = new Meat("Pork"); // Futter für Bello
...
shawn.eat(f); // Igitt!
```

Was ist eine „Stärkere Vorbedingung“ oder „Schwächere Nachbedingung“?

- In statisch getypten Programmiersprachen
 - ◆ Nachbedingung = Ergebnistyp
 - ◆ Vorbedingung = Typen aller Parameter

Diese Vor- und Nachbedingungen sind durch den Compiler überprüfbar.
 - ◆ Stärkere Vorbedingung = Gleiche Operation im „Subtyp“ hat **Parameter mit speziellerem Typ** als die im „Obertyp“ ← in den meisten OO-Sprachen verboten, da das keinen korrekten Subtyp ergibt – in Java & Co sind die Typen der Parameter einer überschreibenden Methode im Subtyp die gleichen wie die im Obertyp
- Im Design-by-Contract
 - ◆ Nachbedingung = Ergebnistyp + **Assertions**
 - ◆ Vorbedingung = Parametertypen + **Assertions**

Ob dies in der Sprache ausgedrückt und von dem Compiler überprüft werden kann hängt von der jeweiligen Sprache ab (siehe Folie 2-25).
 - ◆ Stärkere Vorbedingung = Gleiche Operation im "Subtyp" hat **speziellere Parametertypen** oder **zusätzliche Assertions**

Fazit „Ersetzbarkeit und DBC“

Bei der Modellierung immer die Assertions / Constraints im Auge behalten und darauf achten

- a) dass Unterklassen Vorbedingungen nicht verstärken und Nachbedingungen nicht abschwächen
- b) dass ansonsten der Code Laufzeitfehler abfängt, die durch fehlende Ersetzbarkeit möglich sind
- c) ... und dieser Aspekt besonders getestet wird
← über automatisierte Tests sprechen wir später

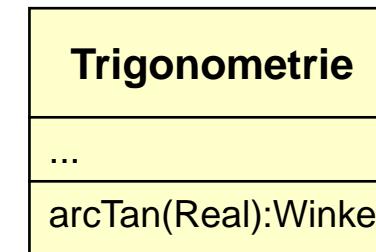
2.3 OO Modellierungs-Prinzipien

Wo modelliere ich die arcTan-Funktion?

Was halten Sie hiervon?



Und hiervon?



Prinzip: Nur Bezug auf inhärente Eigenschaften!

- **Inhärente Eigenschaft:** Eine Eigenschaft E ist für eine Klasse C inhärent, wenn C nicht ohne E definiert werden kann
- Problem
 - ◆ Bezug auf **nicht**-inhärente Eigenschaften / Klassen führt unnötige Abhängigkeiten ein
- Behandlung
 - ◆ Bezug auf nicht-inhärente Teile entfernen
 - ◆ Eventuell Teile der Klasse in andere Klassen auslagern
 - ⇒ siehe Refactorings (z.B. „Move Method“, „Move Field“, „Split Class“)

Prinzip: Externe Abhangigkeiten vermeiden!

- B ist **abhängig** von A wenn
 - ◆ Änderungen von A Änderungen von B erfordern (oder zumindest erneute Verifikation von B) um Korrektheit zu garantieren
- Prinzip
 - ◆ Abhangigkeiten zwischen Kapselungseinheiten (**Kopplung**) reduzieren
 - ◆ Abhangigkeiten innerhalb der Kapselungseinheiten (**Koharenz**) maximieren
- Nutzen
 - ◆ Wartungsfreundlichkeit
- Eine **Kapselungseinheit** kann sein
 - ◆ eine **Methode**: kapselt Algorithmus
 - ◆ eine **Klasse**: kapselt alles was zu einem Objekt gehort
 - ◆ eine **Komponente**: kapselt ein Subsystem (← siehe Kapitel 8.)

2.4 **Gibt es „akzeptable“ Abhängigkeiten?**

Stabilität
Abstraktheit

Abstraktheit eines Typs T

Definition

- $abs_T = \frac{a_T}{a_T + k_T}$
- a_T = Anzahl abstrakter Methoden in T
- k_T = Anzahl konkreter Methoden in T

Es werden nur nicht-private Methoden gezählt (da andere Typen sich nur von diesen abhängig machen können)

Beispiele

- $abs_{T1} = 4 / 4+2 = 66\%$
- $abs_{T2} = 3 / 3+0 = 100\%$

T1
+abstract1() +abstract2() +abstract3() #abstract4() +konkret1() #konkret2() - konkret3() - konkret4()

T2
+abstract1() +abstract2() ~abstract3()

Idee der Abstraktheit-Metrik: Je abstrakter ein Typ ist um so

- leichter ist er wiederverwendbar
- häufiger wird er wiederverwendet

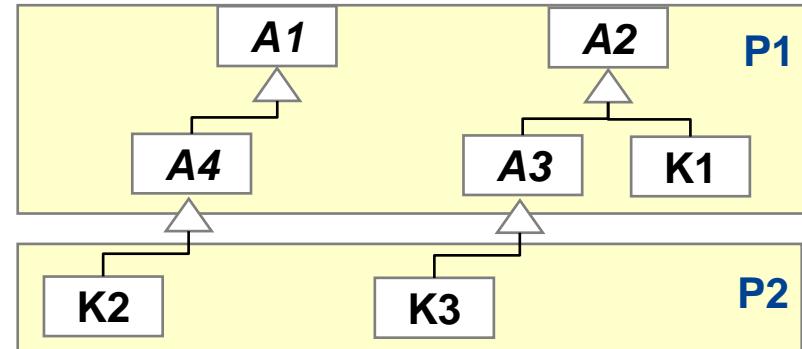
Abstraktheit eines Paketes P

Definition

- $abs_P = \frac{a_P}{a_P + k_P}$
- a_P = Anzahl abstrakter Typen in P
 - ◆ Typen T die abstrakte Methode(n) enthalten, d.h. $abs_T > 0$
- k_P = Anzahl konkreter Typen in P
 - ◆ Typen T die nur konkrete Methoden enthalten, d.h. $abs_T = 0$

Beispiele

- $abs_{P1} = 4 / 4+1 = 4/5 = 80\%$
- $abs_{P2} = 0 / 0+2 = 0/2 = 0\%$



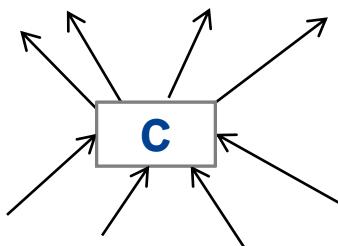
Idee der Abstraktheit-Metrik: Je abstrakter ein Paket ist um so

- leichter ist es wiederverwendbar
- häufiger wird es wiederverwendet

(In)Stabilität eines Typs oder Pakets

Abhängigkeiten

- **efferente (e)**
= Eigene Abhängigkeiten von anderen



- **afferente (a)**
= Abhängigkeiten anderer von mir

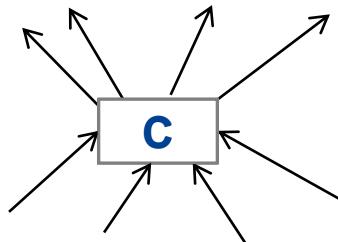
Zählen von Abhängigkeiten

- Betrachtet werden **alle statischen Abhängigkeiten**
 - ◆ Vererbungs-Beziehungen
 - ◆ Assoziationen
 - ◆ Parameter-Typen
 - ◆ Feld- und Ergebnistypen
 - ◆ Konstruktor-Aufrufe
 - ◆ Casts und instanceof-Tests
- ... aber **jede nur ein mal**:
 - ◆ **Efferent**: Jeder Typ zu dem es in **C** eine Abhängigkeit gibt wird 1 mal gezählt, nicht wie oft er vorkommt
 - ◆ **Afferent**: Jeder Typ von dem es zu **C** eine Abhängigkeit gibt wird 1 mal gezählt, nicht wie oft er sich auf **C** bezieht

(In)Stabilität eines Typs oder Pakets

Abhängigkeiten

- **efferente (e)**
= Eigene Abhängigkeiten von anderen



- **afferente (a)**
= Abhängigkeiten anderer von mir

Definitionen & Beispiel

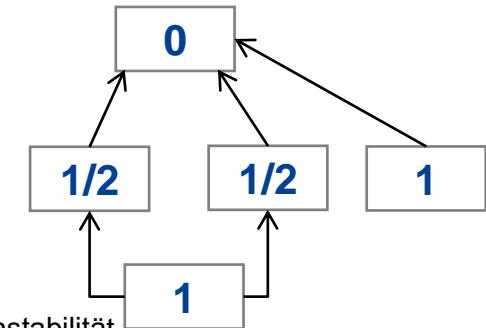
$$I = \frac{e}{e+a}$$

$$S = 1 - I$$

d.h. Stabilität = 1 – Instabilität

0 = maximal stabil

hängt von nichts ab, aber andere davon



1 = maximal instabil

nichts hängt davon ab

Idee der Instabilitäts-Metrik: Instabilität ist ein Maß dafür, wie leicht eine Softwareeinheit (Typ oder Paket) zu ändern ist. Eine mit Wert 1 (maximal instabil) ist leicht zu ändern, weil Änderungen keine Konsequenzen für Andere haben. Eine mit Wert 0 sollte auf keinen Fall geändert werden (daher maximal stabil), weil die Auswirkungen weitreichend wären.

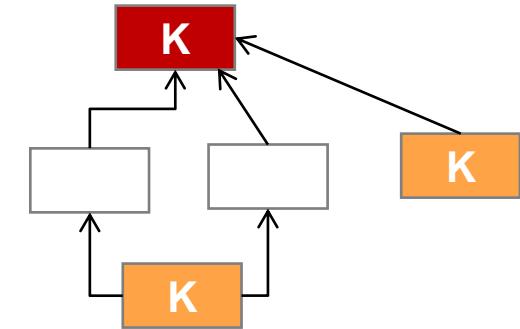
Wechselwirkung von Stabilität und Abstraktheit (1)

* Stabilität = $1 - \text{Instabilität}$

- Eine **stabile** konkrete Klasse ist unwartbar
 - ◆ Da sie **konkret** ist, werden sich viele Anforderungsänderungen in Änderungen der Klasse niederschlagen ☹
 - ◆ Da sie **stabil** ist wird jede Änderung der Klasse viele Folgeänderungen nach sich ziehen ☹
- Eine **instabile** konkrete Klasse ist schwer wiederverwendbar
 - ◆ Je konkreter, um so weniger können andere davon profitieren
 - ◆ Je konkreter, um so mehr folgeänderungsgefährdet wären die potentiellen Wiederverwender

0 = maximal stabil

hängt von nichts ab,
aber andere davon



1 = maximal instabil

nichts hängt davon ab

Wechselwirkung von Stabilität und Abstraktheit (2)

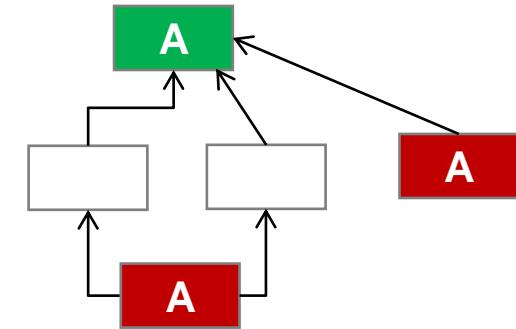
* Stabilität = $1 - \text{Instabilität}$

- Eine **stabile abstrakte Klasse** ist gut wiederverwendbar

- ◆ Da sie abstrakt ist, können viele sie wiederverwenden
- ◆ Da sie abstrakt ist, führen viele Anforderungsänderungen nicht zu Änderungen in der Klasse und damit auch zu keinen Folgeänderungen
- ◆ Folgeänderungen hätten große Auswirkungen, treten aber kaum auf

0 = maximal stabil

hängt von nichts ab,
aber andere davon



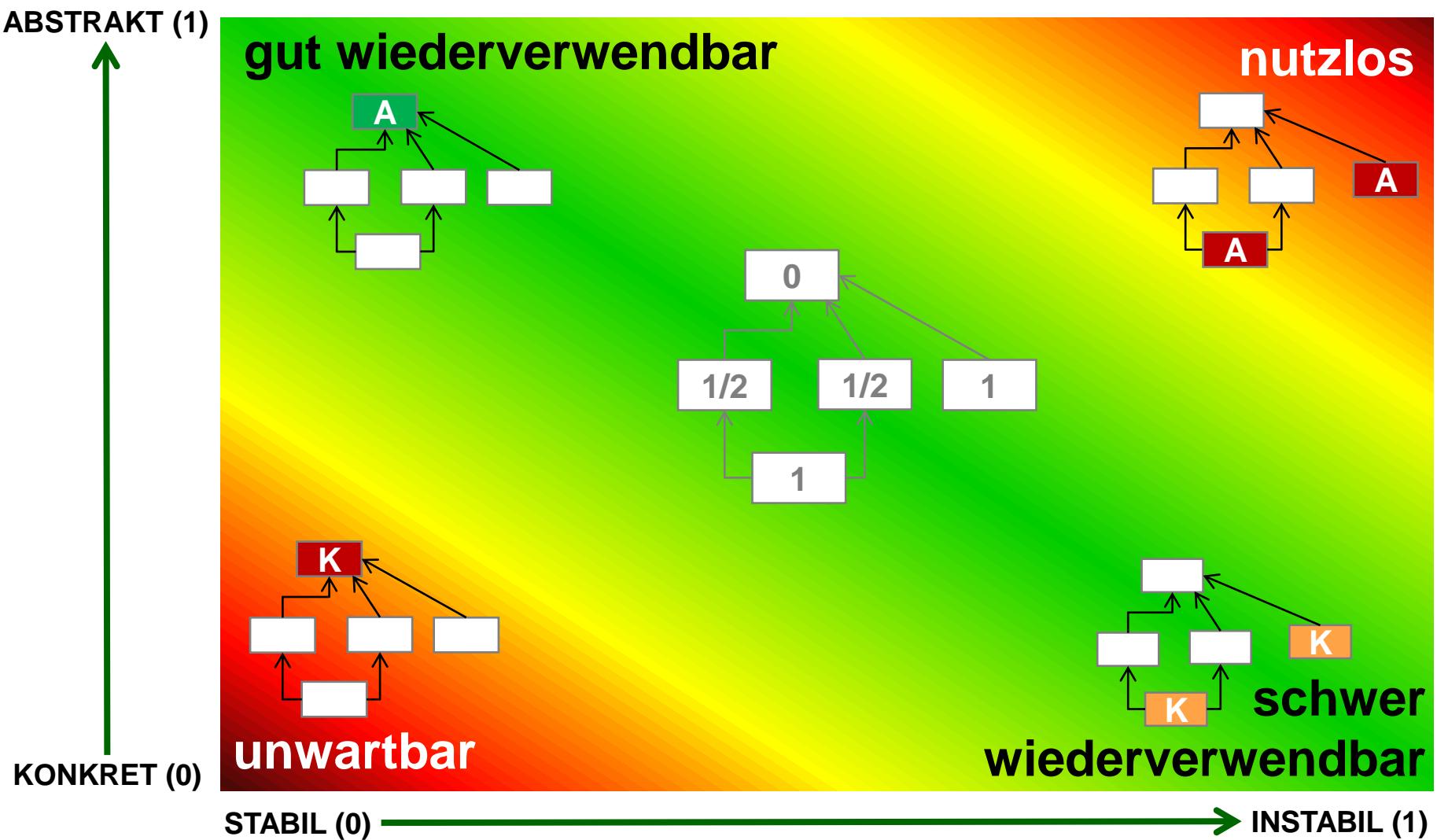
- Eine **instabile abstrakte Klasse** ist nutzlos

- ◆ Wofür die Abstraktion, wenn niemand sie nutzt?

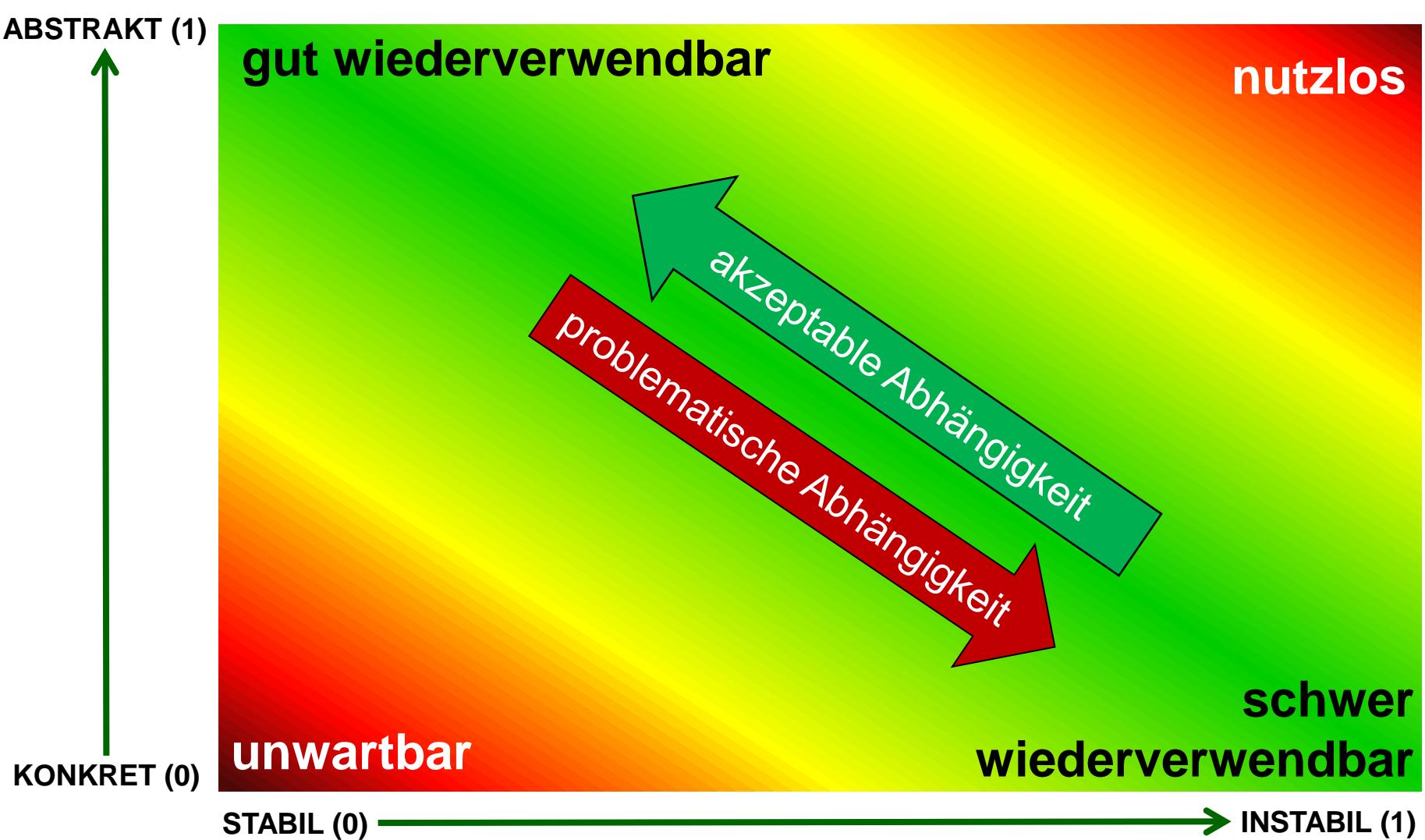
1 = maximal instabil

nichts hängt davon ab

Klassifikation nach Stabilität und Abstraktheit



Gute und schlechte Abhängigkeiten



2.5. Modellierungs-Prinzipien von Robert C. Martin

Stable Abstractions Principle (SAP)

S.

Dependency Inversion Principle (DIP)

O.

Stable Dependencies Principle (SDP)

L.

Acyclic Dependencies Principle (ADP)

I.

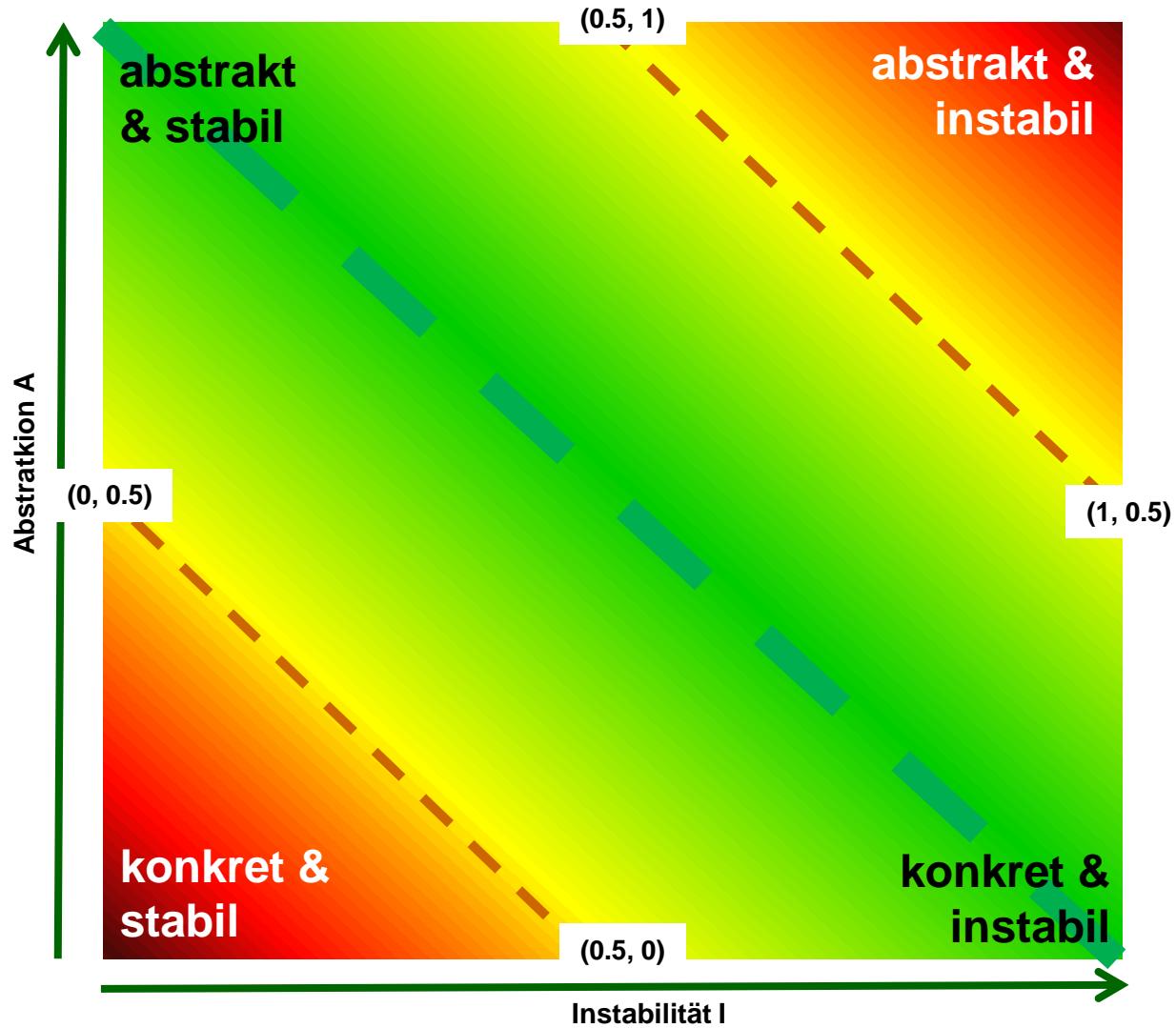
D.

Entwurfs-Prinzipien von Robert C. Martin

(1)

- Stable Abstractions Principle (**SAP**)

- Je stabiler Einheiten sind, um so abstrakter sollten sie sein
- Je instabiler Einheiten sind, um so konkreter sollten sie sein

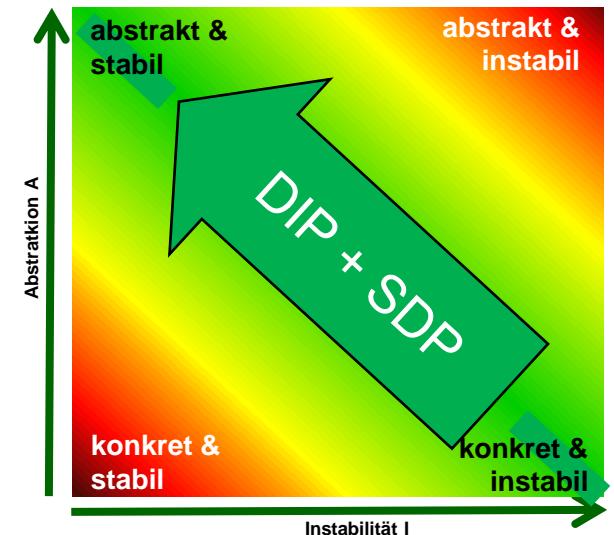


Entwurfs-Prinzipien von Robert C. Martin

(2)

Die Intuition, dass schwer Wiederverwendbares von gut Wiedwendbaren abhängig sein darf, **aber nicht umgekehrt**, wird durch die folgenden Prinzipien präzisiert:

- Dependency Inversion Principle (**DIP**)
 - ◆ Abhängigkeiten nur zu gleich abstraktem oder Abstrakterem!
- Stable Dependencies Principle (**SDP**)
 - ◆ Abhängigkeiten nur zu Stabilerem!
- Acyclic Dependencies Principle (**ADP**)
 - ◆ Der Abhängigkeitsgraph veröffentlichter Komponenten muss azyklisch sein!



Geht noch mehr?

JA! → S.O.L.I.D.

SOLID-Prinzipien von R.C. Martin (1)



- **S - Single-responsiblity Principle**

- **S - Single-responsiblity Principle**
- ◆ Jeder Typ sollte einen einzigen Grund haben geändert zu werden und daher eine einzige Aufgabe erfüllen

- **O - Open-closed Principle**

- **O - Open-closed Principle**
- ◆ Typen sollten *offen* sein für Erweiterung aber *geschlossen* für Änderung.
- ◆ Jeder Aspekt der sich ändern könnte, sollte als überschreibbare Operation gekapselt sein – somit erweiterbar, ohne geändert werden zu müssen.

- **L - Liskov Substitution Principle**

- **L - Liskov Substitution Principle**
- ◆ Ersetzbarkeitsprinzip (von Barbara Liskov)
- ◆ Erweiterungen dürfen den Kontrakt der Oberklasse nicht verletzen

SOLID-Prinzipien von R.C. Martin (2)

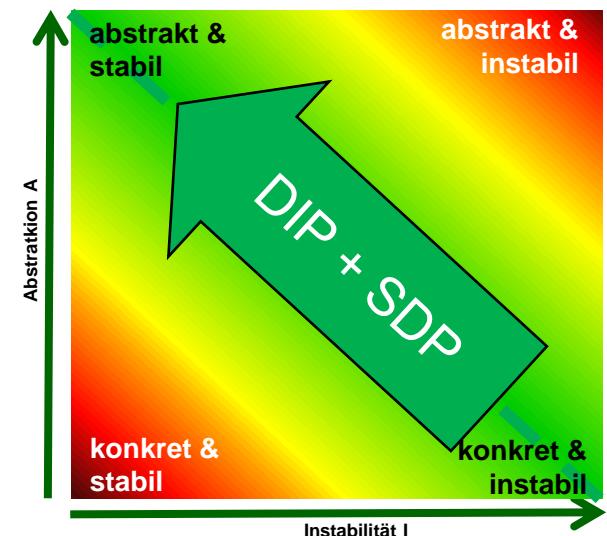


● I - Interface Segregation Principle

- ◆ Schnittstellen sollten minimal sein (nur bieten, was Klienten brauchen).
- ◆ Z.B. sollten zwei- und dreidimensionale Körper durch getrennte Interfaces beschrieben werden

● D - Dependency Inversion Principle

- ◆ Abhängigkeiten nur zu gleich abstraktem oder Abstrakterem!



Zusammenfassung & Literatur

OO Modellierung: Rückblick (1)

- CRC-Cards → Identifikation von
 - ◆ Klassen
 - ◆ Operationen
 - ◆ Kollaborationen
- Design by Contract → Verfeinerung des Verhaltens durch
 - ◆ Vorbedingungen
 - ◆ Nachbedingungen
 - ◆ Invarianten
- Ersetzbarkeit
 - ◆ auch mit Contracts (jenseits dessen, was der Compiler überprüfen kann)

OO Modellierung: Rückblick (2)

- Modellierungs-Prinzipien → Gutes Design ist gekennzeichnet durch

- ◆ Ersetzbarkeit
- ◆ Einheitliches Verhalten
- ◆ Single responsibility
- ◆ Redundanzfreiheit
- ◆ Abhängigkeit nur von inhärenten, allgemeineren und stabileren Typen
- ◆ Minimalen Schnittstellen
- ◆ Open / closed Prinzip erfüllt



Damit erreicht man
"Lokalität von Änderungen"
('locality of change').

Das ist die Voraussetzung für
Wartbarkeit.

- Lokalität von Änderungen ist ein wesentliches Qualitätsmerkmal
 - ◆ Idealerweise, jede Änderung an nur einer Stelle durchführen müssen

OO Modellierung: Literatur

- CRC-Cards
 - ◆ Konzentriert: Fowler & Scott, „UML distilled“ (2te Ausgabe), Addison Wesley
 - ◆ Original-Artikel: <http://c2.com/doc/oopsla89/paper.html>
- Design by Contract
 - ◆ Konzentriert: Fowler & Scott, „UML distilled“ (2te Ausgabe), Addison Wesley
 - ◆ Original-Buch: Bertrand Meyer, „Object-oriented Software Construction“, Prentice Hall, 1997.
 - ⇒ Ein Klassiker!
- Design by Contract (ff)
 - ◆ Bertrand Meyer: “Applying Design by Contract”. Erschienen in Zeitschrift “Computer”, Vol. 25 Issue 10, October 1992, page 40-51, IEEE Computer Society Press Los Alamitos, CA, USA, <http://dx.doi.org/0.1109/2.161279>

OO Modellierung: Literatur

- Modellierungs-Prinzipien (Grundlage des „Quiz“)
 - ◆ Page-Jones, „Fundamentals of Object-Oriented Design in UML“, Addison Wesley, 1999 →
 - ◆ Sehr empfehlenswert!
- Modellierungsprinzipien (Abhängigkeiten, SAP, DIP, ...)
 - ◆ Robert C. Martin: Design Principles and Design Patterns
http://www.objectmentor.com/resources/articles/Principles_and_Patterns.PDF
- Aufbrechen von ungünstigen Abhängigkeiten mit aspektorientierter Programmierung
 - ◆ Martin E. Nordberg: Aspect-Oriented Dependency Inversion.
OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, October 2001
http://www.cs.ubc.ca/~kdvolder/Workshops_OOPSLA2001/submissions/12-nordberg.pdf

Kapitel 3

Die Unified Modeling Language (UML)

- Stand: 20.11.2023 -

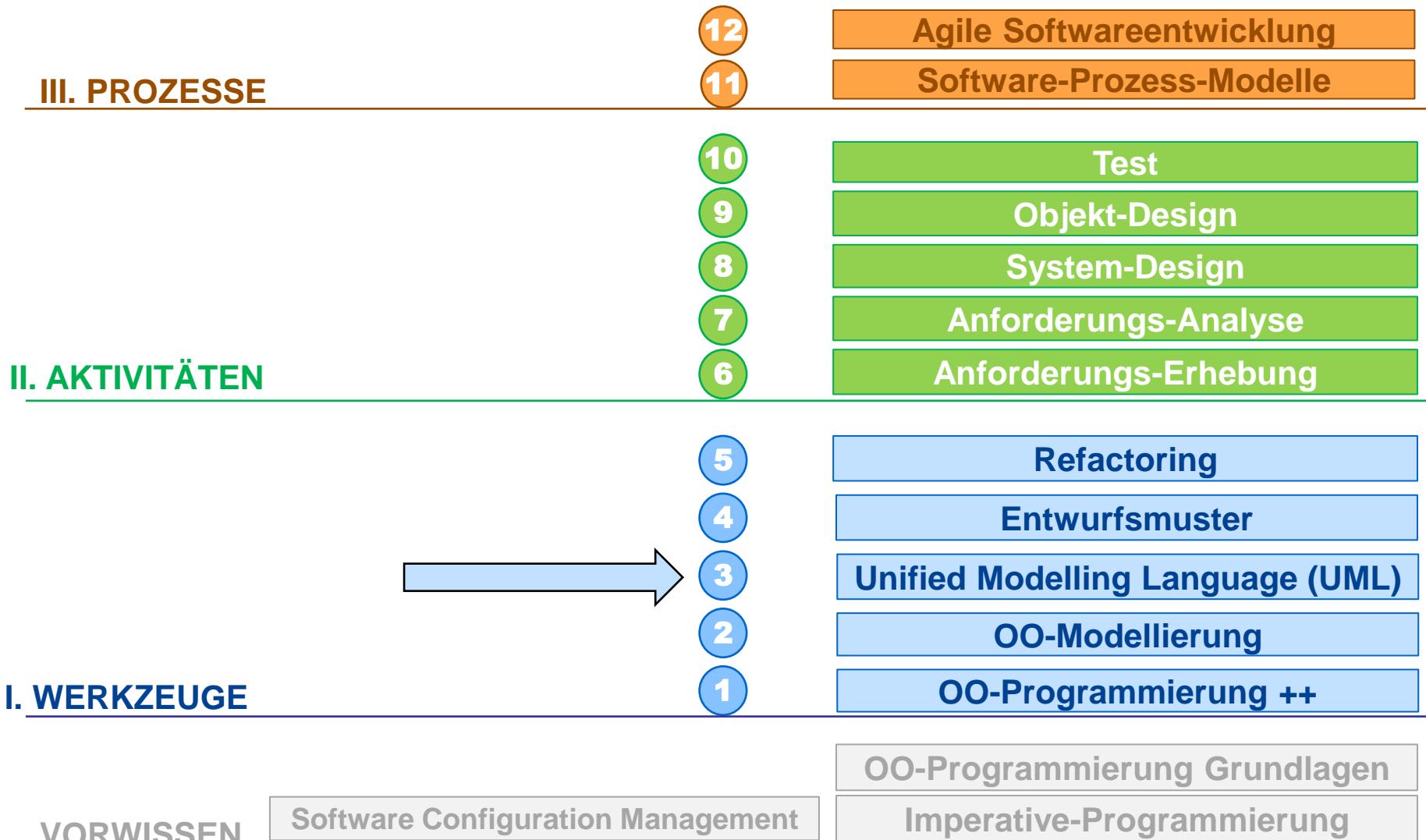
3.1 Motivation und Überblick

3.2 Strukturmodellierung

3.3 Verhaltensmodellierung

3.4 Zusammenfassung und Ausblick

Themenbereiche und Vorlesungs-Kapitel



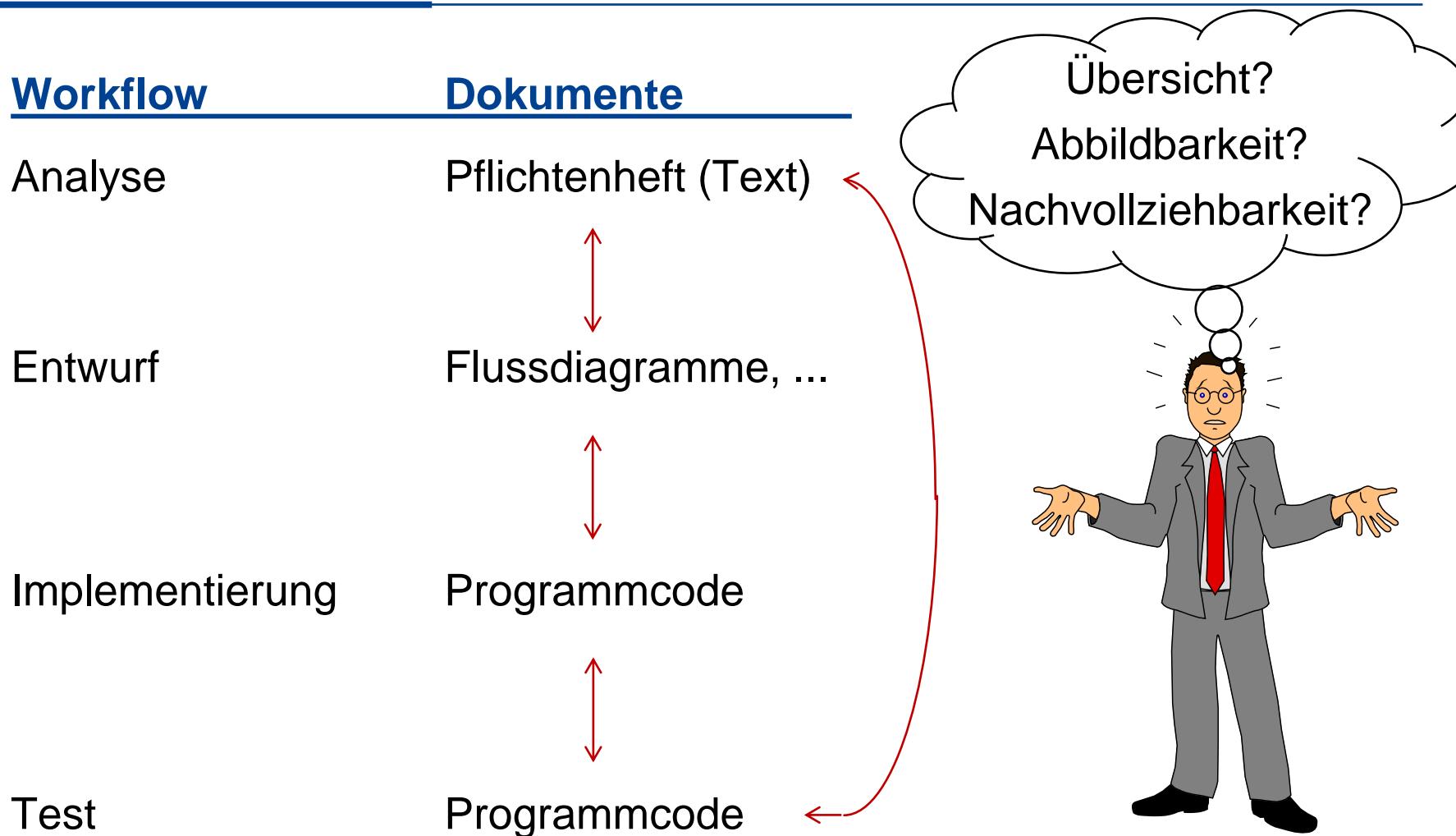
3.1 Motivation und Überblick

Ausgangspunkt
Warum Modellierung?
Warum objektorientierte Modellierung?
Warum mit der UML?
UML-Überblick

Problemstellung

- Ausgangssituation: Sie ...
 - ◆ kennen 2-3 Programmiersprachen
 - ◆ haben bisher allein oder in sehr kleinen Gruppen (<5) gearbeitet
- Neue Herausforderungen: Sie sollen ...
 - ◆ mit Kunden kommunizieren, die nichts von Informatik verstehen
 - ◆ mit Kollegen zusammenarbeiten, die andere Programmiersprachen nutzen
 - ◆ komplexe, evt. verteilte Systeme realisieren, mit einer Vielzahl von Klassen, Subsystemen, verschiedensten Technologien, ...
 - ◆ Entwürfe auf einer höheren Abstraktionsebene als Quellcode kommunizieren und dokumentieren
 - ◆ auch die Software-Verteilung, -Einführung und den Betrieb planen
- Lösungsweg: Abstraktion des zu realisierenden Systems
 - ◆ Erst **modellieren**, dann **programmieren!**

Probleme im traditionellen SW-Prozess



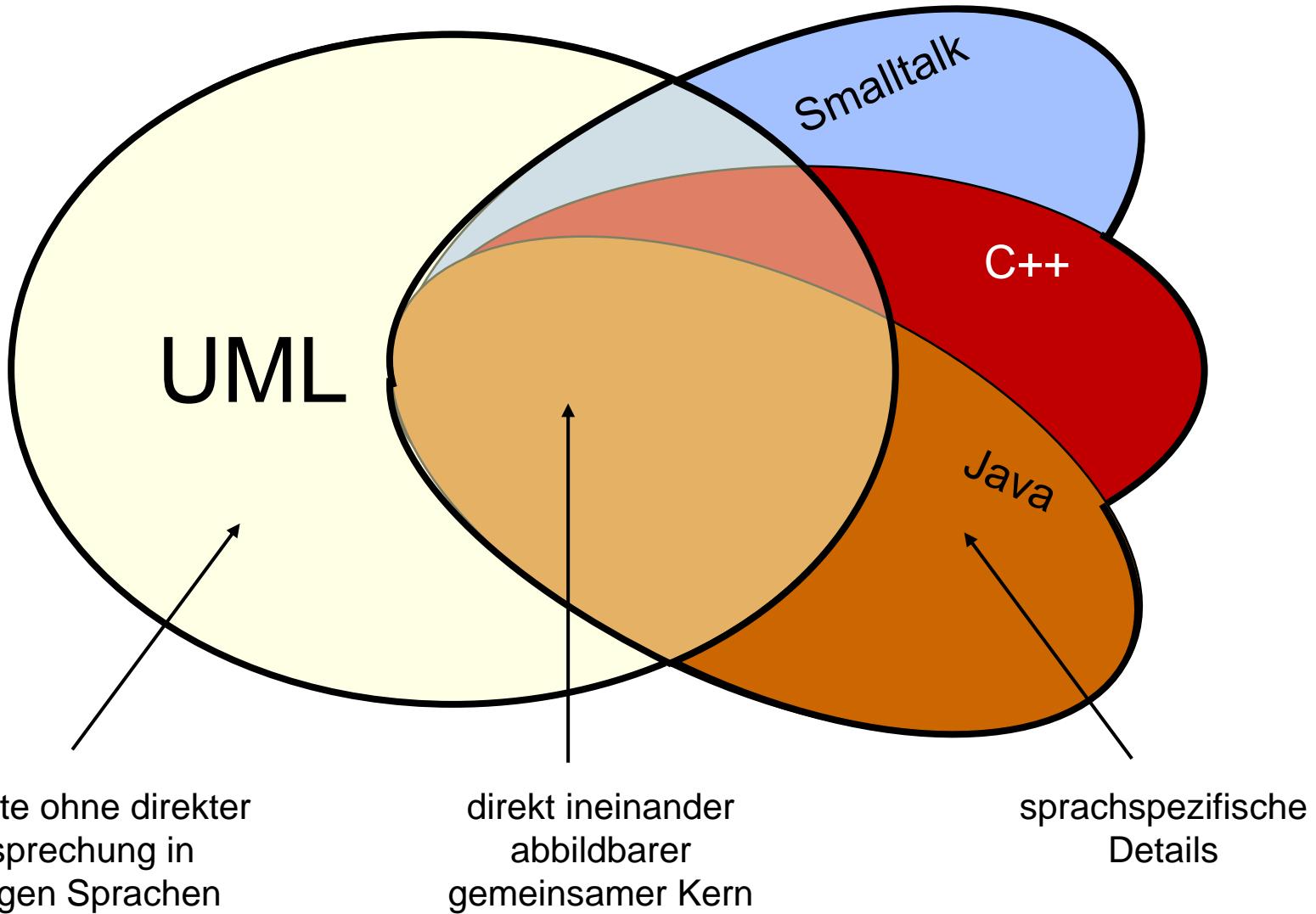
Unified Modeling Language (UML)

- Standardisierte graphische Notation für alle Aktivitäten der Softwareentwicklung
 - ◆ Nutzen für Anforderungserhebung, Entwurf, Implementierung, Einsatz („deployment“), ...
 - ◆ Besondere Unterstützung für objektorientierte Modellierung
 - ◆ Standardisiert durch die OMG (Object Management Group) → www.omg.org
- Bietet zahlreiche Diagrammtypen für verschiedene Sichten von Software
 - ◆ statische Sichten → Struktur
 - ◆ dynamische Sichten → Verhalten
- Mittlerweile breite Werkzeugunterstützung für...
 - ◆ ... die Erstellung von UML Diagrammen
 - ◆ ... Code-Generierung aus Diagrammen (**Forward Engineering**)
 - ◆ ... Diagramm-Generierung aus Code (**Reverse Engineering**)
 - ◆ ... bidirektionale Synchronisation von Diagrammen und Code (**Round-Trip Engineering**)

Nutzen der UML

- Kommunikation mit Anwendern
 - ◆ Wenig formal
 - ◆ Einfache graphische Notation
 - ◆ Konzentration auf das Wesentliche
- Kommunikation mit Kollegen
 - ◆ Austausch von Designs, ...
 - ◆ Abstraktion von Sprachdetails
 - ◆ Schutz vor übereilter Implementationssicht
- Bandbreite
 - ◆ unterstützt alltägliche und exotische Konzepte
 - ◆ manuell, rechner-unterstützt und kombiniert einsetzbar

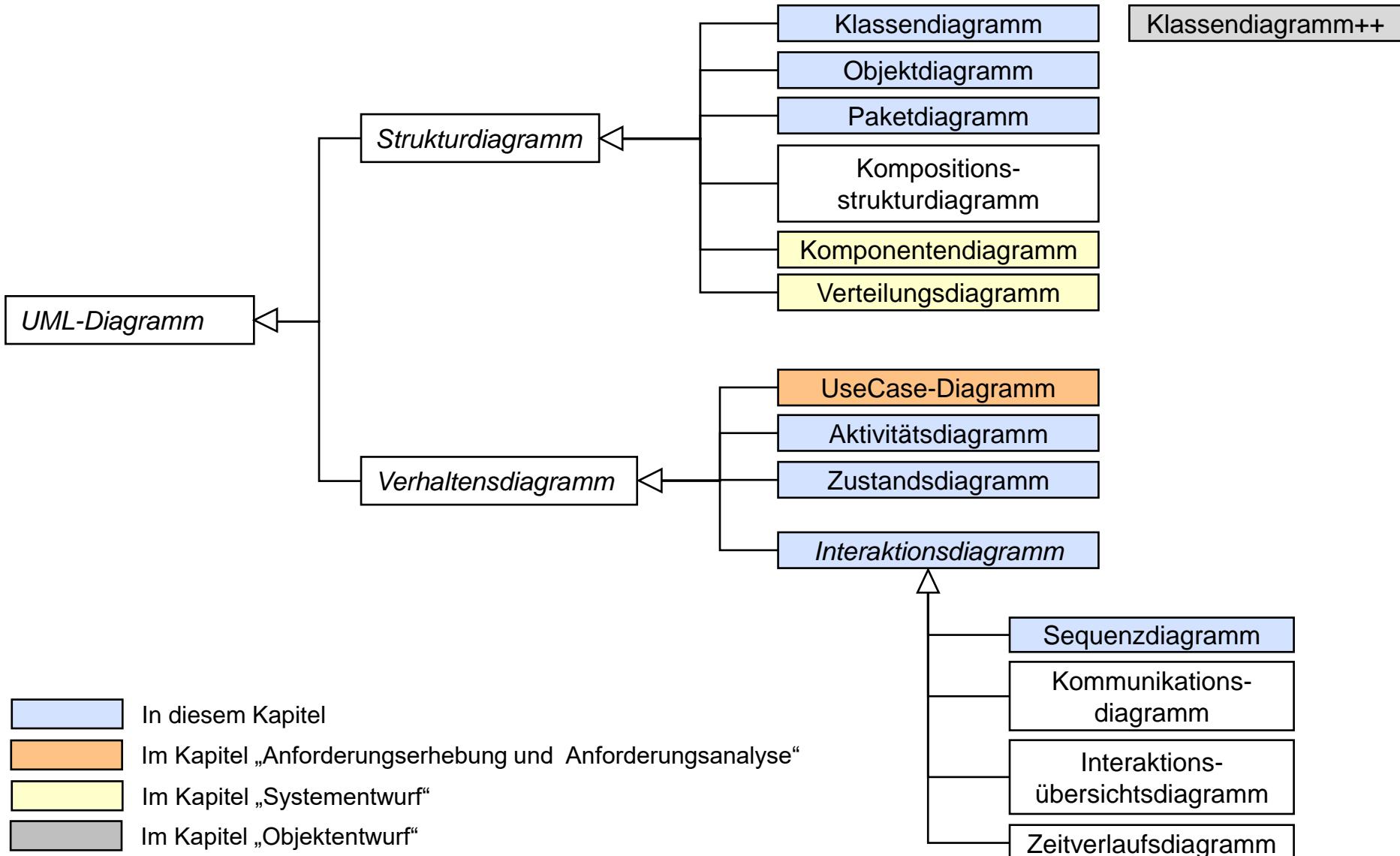
Bezug von UML zu gängigen OO Sprachen



UML: Geschichte und Literatur

- Entstanden aus der Zusammenführung dreier führender objekt-orientierten Methodologien
 - ◆ OMT (James Rumbaugh) – Klassendiagramme, Sequenzdiagramme, ...
 - ◆ OOSE (Ivar Jacobson) – Anwendungsfalldiagramme, ...
 - ◆ Booch (Grady Booch) – Software-Prozess → „Unified Process“
- Literatur: Standardwerke
 - ◆ “The Unified Modeling Language User Guide” (Booch & al 1999)
 - ◆ “The Unified Modeling Language Reference Manual” (Rumbaugh & al 1999)
 - ◆ alle im Addison Wesley / Pearson Verlag)
- Empfehlung
 - ◆ “UML Distilled” (Fowler & al. 2000, Addison Wesley) – kurz und gut!
 - ◆ „UML@Work“ (Hitz & Kappel 2005, dpunkt) – UML 2.0, deutsch, ausführlich!

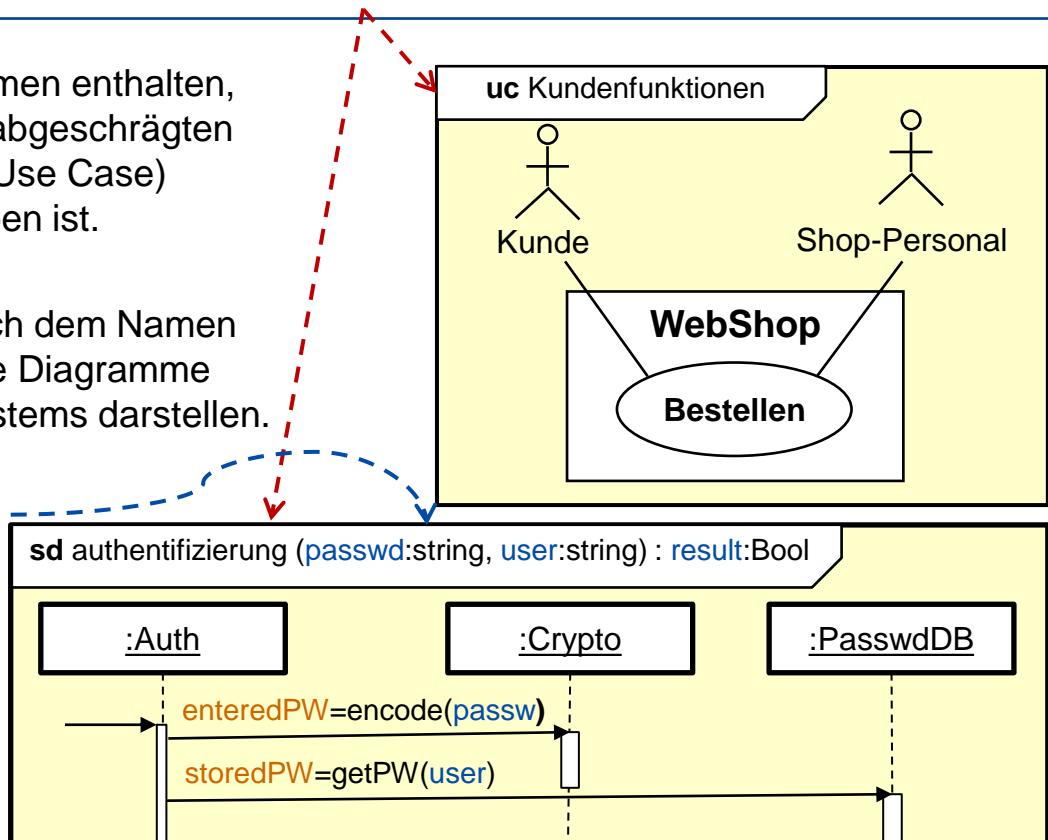
UML Diagrammtypen: Was wird wo erläutert?



UML-Diagrammtypen

Art (uc, sd, ...) und Name des Diagramms

- Jedes UML-Diagramm ist in einem Rahmen enthalten, in dessen oberen linken Ecke in einem abgeschrägten Rechteck die Diagrammart (zb „uc“ für Use Case) und der Name des Diagramms angegeben ist.
- Der Name des Diagramms ist nicht gleich dem Namen des modellierten Systems. Es kann viele Diagramme geben, die Ausschnitte des gleichen Systems darstellen.



- Im Folgenden werden manchmal die Rahmen die das Gesamtdiagramm darstellen weggelassen (aus Platzgründen und um Folien nicht mit Inhalten zu überfüllen die in dem jeweiligen Kontext nicht wichtig sind).
- Bitte denken Sie trotzdem, wenn Sie Aufgaben bearbeiten, an den Gesamtdiagramm-Rahmen mit der Angabe der Diagrammart, denn auch diese Elemente werden bewertet (in den Übungsaufgaben wie in der Klausur)!

UML-Diagrammtypen ► Übersicht

UML 2.5.1, Seite 684			
Diagrammtyp-Name		Diagrammtyp-Beschriftung	
	deutsch	englisch	lang
Struktur-Diagramme	Klassen-	Class	class
	Objekt-	Object	-
	Paket-	Package	package
	Kompositionsstruktur-	Composite structure	package
	Kompositionsobjekt-	Composite object	package
	Komponenten-	Component	component
	Verteilungs-	Deployment	deployment
Verhaltens-Diagramme	Anwendungsfall-	Use case	use case
	Aktivitäts-	Activity	activity
	Zustands-	State machine	state machine
	Interaktions-	Sequence	interaction
Diagramme	Kommunikations-	Communication	interaction
	Interaktionsübersicht-	Interaction overview	interaction
	Zeitverlaufs-	Timing	interaction



3.2 Strukturdiagramme „im Kleinen“

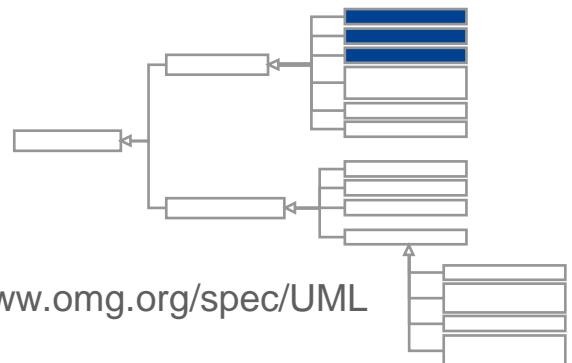
Klassendiagramme → „UML@Work“, Seite 52 - 101

Objektdiagramme → „UML@Work“, Seite 118 - 120

Paketdiagramme → „UML@Work“, Seite 120 – 134

Alles im Detail → UML Spezifikation 2.5.1 (5.12.2017) <https://www.omg.org/spec/UML>

– Kap 8-12 (Seite 63-284)



3.2.3 Objektdiagramme

Objektdiagramm ▶ Objekte

Objekte werden durch Rechtecke dargestellt:

- **Das Namensfeld** einer Instanz ist unterstrichen und besteht (in dieser Reihenfolge) aus:

- ◆ Einem Namen für die **Rolle der Instanz** in einer Interaktion (optional)
- ◆ Einem „/*I*“ gefolgt vom einem Namen für die **Rolle des Typs** in einer Komposition (optional)
- ◆ Einem **Doppelpunkt** als Trenner (zwingend)
- ◆ Dem **Typ** der Instanz (zwingend)

- **Die Attribute** können zusammen mit ihren jeweiligen Werten angegeben werden (optional)

- Methoden werden nicht angegeben

- ◆ Modellierung mit Hilfe von Klassen als Abstraktion, auch wenn wir das Modell mit einer prototyp-basierten Sprache umsetzen

rad1 : Rad

linkesVR / Vorderrad : Rad

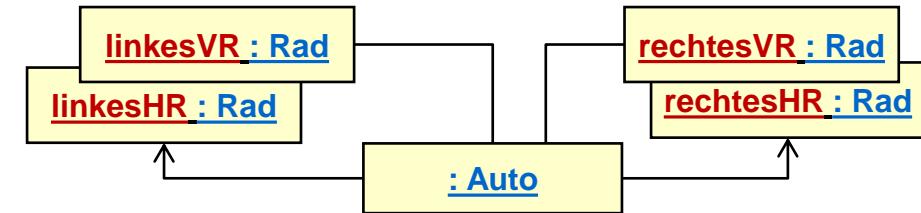
: Tarif

zone2price = { {'1', 1.80}, {'2', 2.40}, {'3', 3.60} }

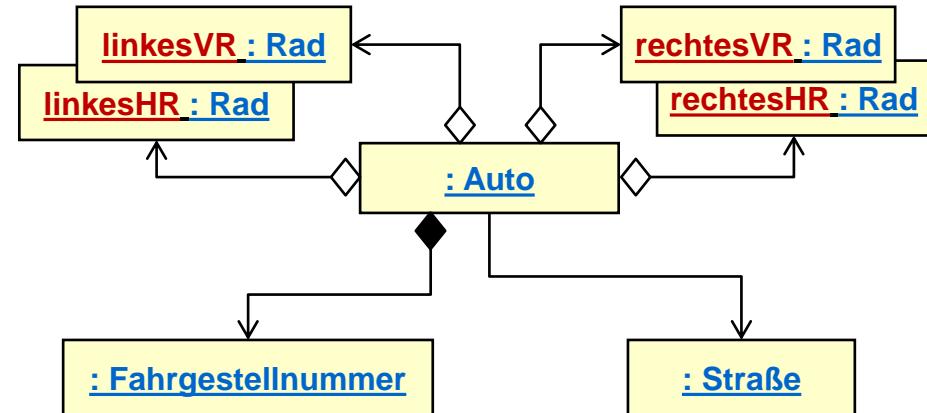
Eine anonyme Instanz von Tarif
mit Details zu den Variablenwerten

Objektdiagramm ▶ Beziehungen

- Beziehungen werden **im Objektdiagramm** durch Linien dargestellt:
 - gerichtete Linie = Referenz
 - ungerichtete Linie = unspezifiziert



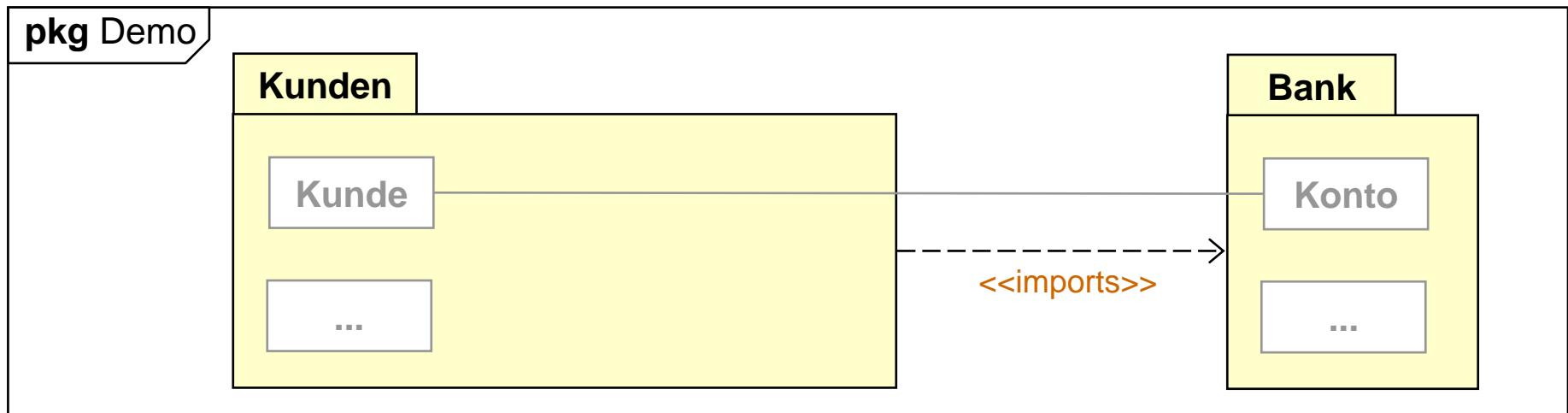
- Nicht dem Standard entsprechend aber manchmal hilfreich:
 - Aggregation
 - Komposition
- (Die Notation dafür – dunkle und helle Raute – ist im Standard nur für Klassendiagramme definiert).



3.2.2 Paketdiagramme

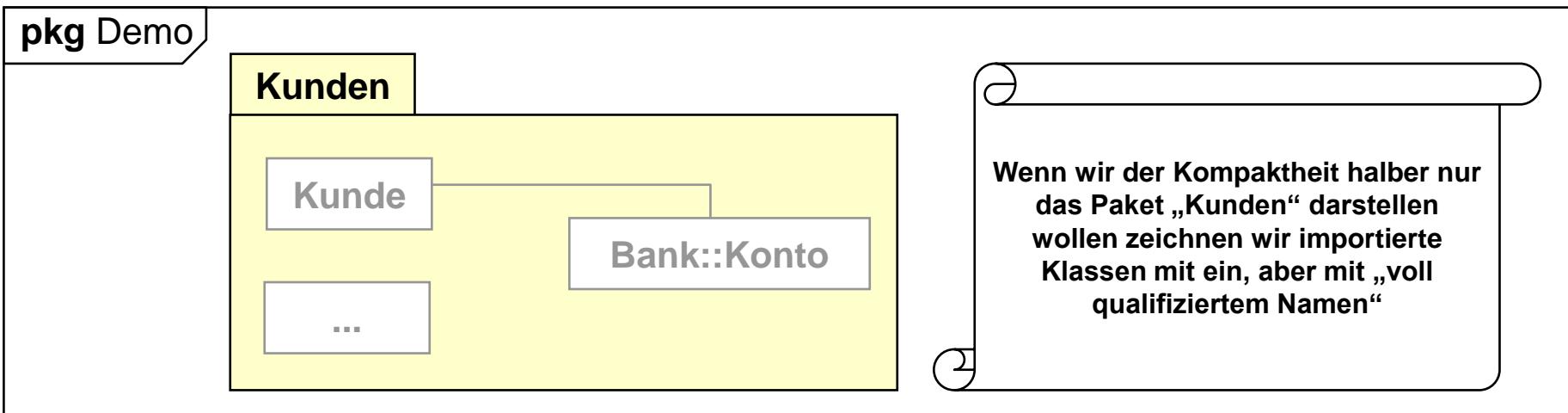
Paketdiagramme

- Bedeutung
 - ◆ Gruppierung thematisch zusammengehöriger Klassen
- Darstellung
 - ◆ „Karteikarten“, auf denen die dazugehörigen Klassen aufgemalt sind
- Import aus anderem Package
 - ◆ Gestrichelter Abhängigkeits-Pfeil mit stereotyp <<imports>>



Paketdiagramme

- Bedeutung
 - ◆ Gruppierung thematisch zusammengehöriger Klassen
- Darstellung
 - ◆ „Karteikarten“, auf denen die dazugehörigen Klassen aufgemalt sind
- Import aus anderem Package
 - ◆ Gestrichelter Abhängigkeits-Pfeil mit stereotyp <<imports>>
- Referenz auf eine Klasse in einem anderen Package
 - ◆ package::class



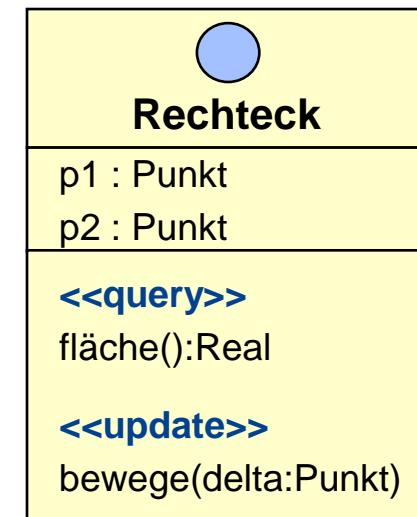
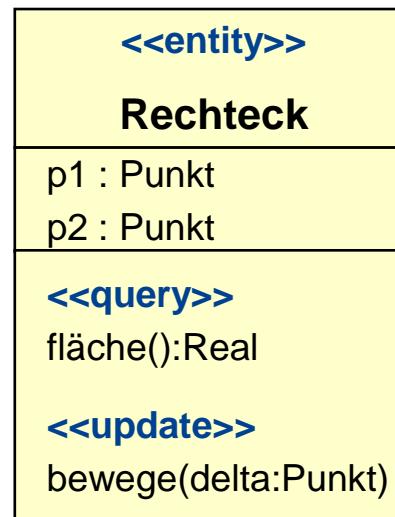
Stereotypen ▶ Spezielle semantische Kategorien

- Problem: Bisher nur Notationen für eine feste Menge von Konzepten
 - ◆ Die Welt ist aber unendlich...
- Idee: Stereotypen
 - ◆ Hinweis auf Semantik für die es keine spezifische Notation gibt
 - ◆ Ansatzpunkt für anwendungsspezifische Erweiterungen
- Notation
 - ◆ “<<stereotyp>>“ oder graphisches Symbol, z.B.
 - ⇒ <<controller>>
 - ⇒ <<imports>>
 - ◆ Was Sie zwischen << und >> schreiben bleibt Ihnen überlassen
- Anwendbarkeit
 - ◆ Allgemein (Klassen, Variablen, Operationen, Beziehungen, ...)



Stereotypen ▶ Beispiele

- Klasse mit Stereotypen



- Drei Äquivalente Darstellungen der Klasse PenTracker



3.2.3 Klassendiagramme

3.2.3.1 Klassen

3.2.3.2 Vererbung und Implementierung

3.2.3.3 Assoziationen

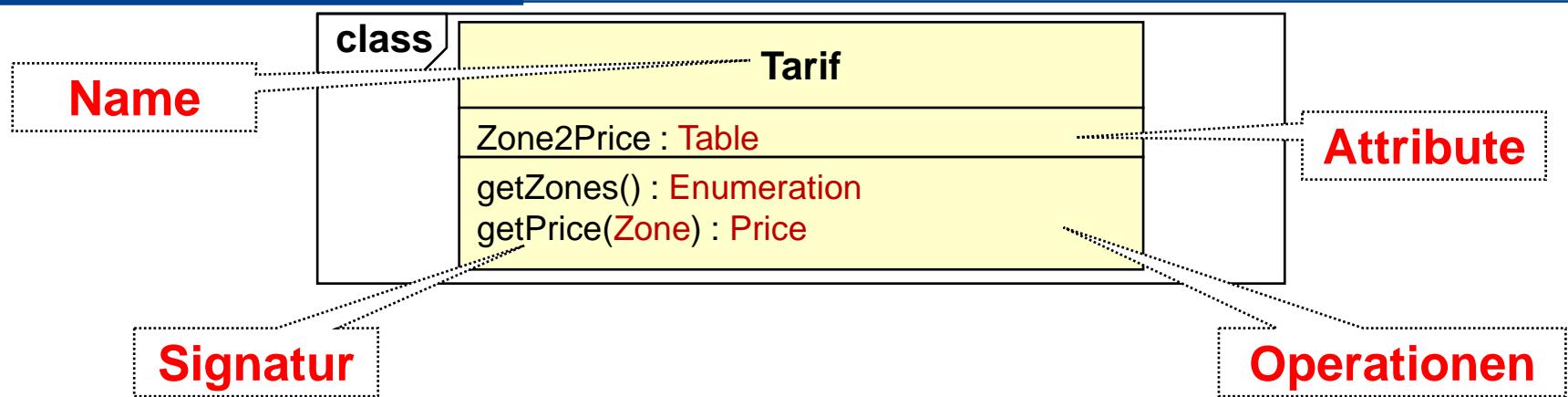
3.2.3.4 Qualifizierte Assoziationen

3.2.3.5 Aggregation und Komposition

3.2.3.6 Forward Engineering (Abbildung Klassendiagramm → Code)

3.2.3.1 Klassen

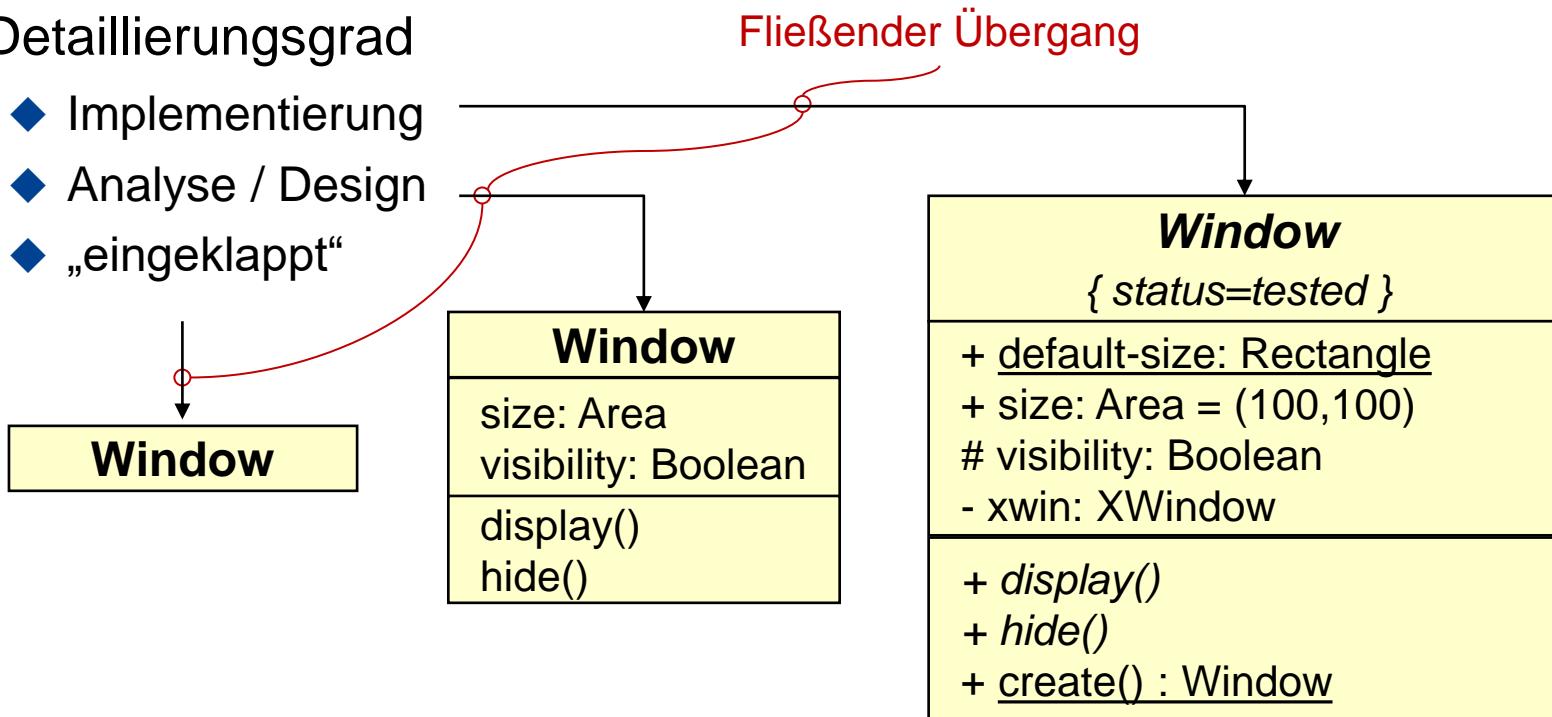
Klassendiagramm ▶ Klassen



- Eine **Klasse** repräsentiert ein Konzept.
- Eine Klasse kapselt Zustand (**Attribute**) und Verhalten (**Operationen**).
- Jedes Attribut hat einen **Typ**.
- Jede Operation hat eine **Signatur**.
- Der Klassename ist die einzige unverzichtbare Information
 - ◆ Hinzufügen weiterer Information beim Fortschreiten des Modellierungsprozesses
 - ◆ Es ist möglich, unwichtige Details in einer teilweisen Sicht des Modells zu verstecken

Klassendiagramm ▶ Verschiedene Detaillierung

- Detaillierungsgrad
 - ◆ Implementierung
 - ◆ Analyse / Design
 - ◆ „eingeklappt“



- Sichtbarkeit
(Implementierungs-Ansicht)

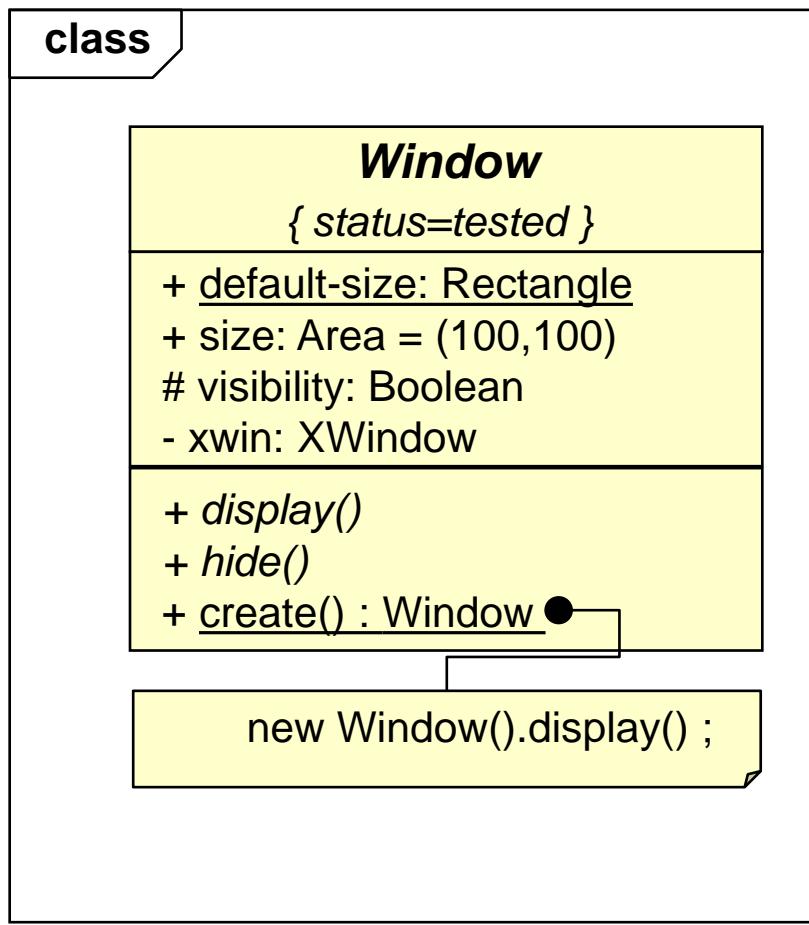
- ◆ public: +
- ◆ protected: #
- ◆ private: -
- ◆ package-private: ~ (sichtbar nur im eigenen Paket)

- Weitere Notationen

- ◆ Klassenvariable / -methode: unterstrichen
- ◆ abstrakte Klasse / Methode: *kursiv*

Klassendiagramm ➤ Umsetzung in Code

Klasse ohne Assoziationen



Java-Quellcode

```
@status{ tested = true }

abstract class Window {

    public static Rectangle default-size ;
    public Area size = new Area(100,100) ;
    protected Boolean visibility ;
    private Xwindow xwin ;

    public abstract display() {}
    public abstract hide() {}
    public static Window create() {
        Window w = new Window() ;
        w.display() ;
        return w;
    }
}
```

Abstrakte Klassen und Interfaces

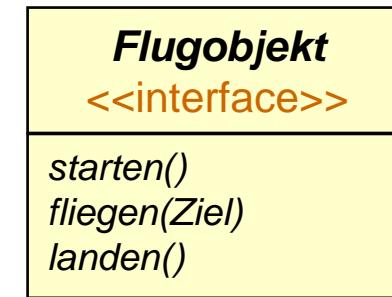
Abstrakte Klassen

- Definition
 - ◆ Implementierung unvollständig
 - ◆ Enthalten abstrakte Methoden
- Notation
 - ◆ Kursivschrift für Namen abstrakter Typen und Methoden
 - ◆ Alternativ: Constraint **{abstract}** in geschweiften Klammern
- Äquivalente Beispiele



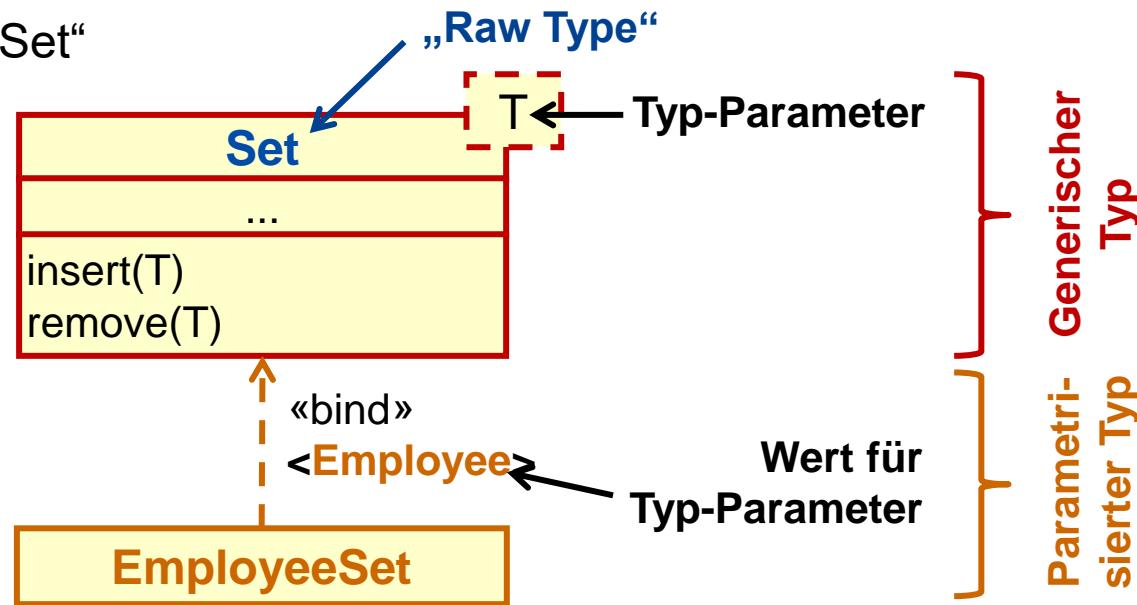
Interfaces

- Definition
 - ◆ Ganz abstrakter Typ
 - ◆ Keine Attribute und keine Methodenimplementierungen
- Notation
 - ◆ wie Abstrakte Klasse
 - ◆ evtl. zusätzlich Angabe des Stereotyps **<<interface>>**
- Beispiel



Klassendiagramm ►Generische Typen

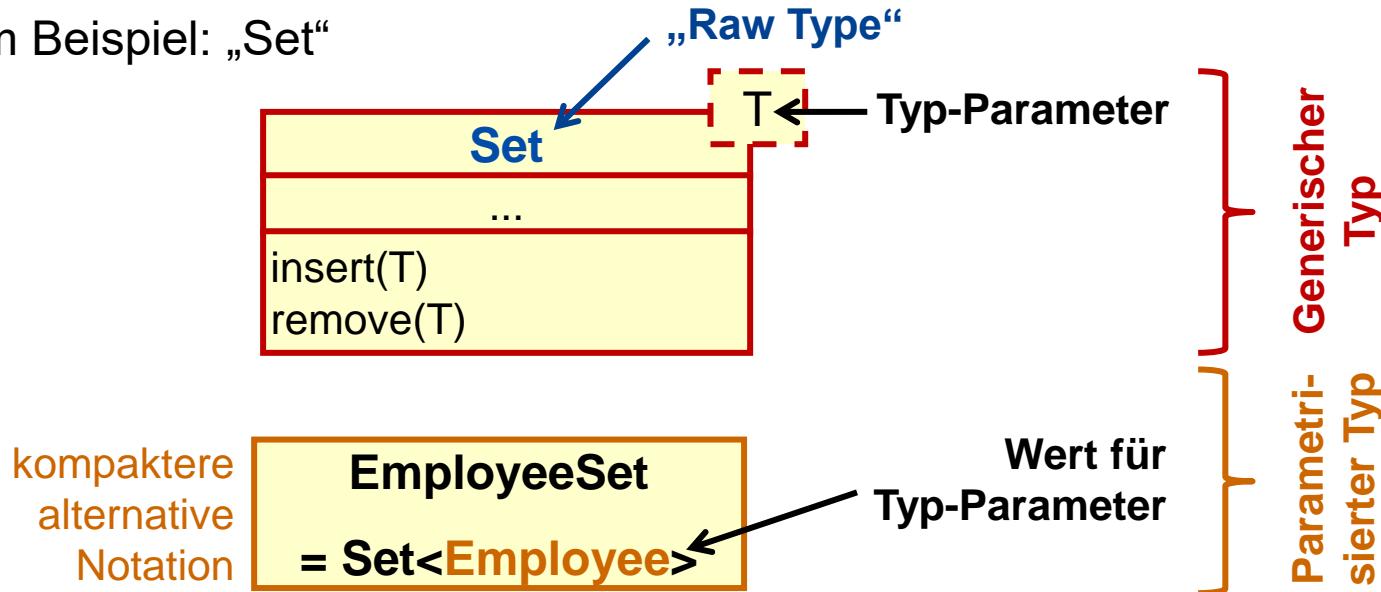
- Generischer Typ = Typ mit Typ-Parameter(n)
 - ◆ Im Beispiel: „Set<T>“, wobei T eine Variable ist, die für einen Typ steht
- „Raw Type“ = Der eigentliche Typname (ohne Parametern)
 - ◆ Im Beispiel: „Set“



- Parametrisierter Typ = Der Typ der entsteht wenn man alle Typparameter durch Typen ersetzt
 - ◆ Im Beispiel: EmployeeSet = „Set<Employee>“

Klassendiagramm ► Generische Typen

- Generischer Typ = Typ mit Typ-Parameter(n)
 - ◆ Im Beispiel: „Set<T>“, wobei T eine Variable ist, die für einen Typ steht
- „Raw Type“ = Der eigentliche Typname (ohne Parametern)
 - ◆ Im Beispiel: „Set“



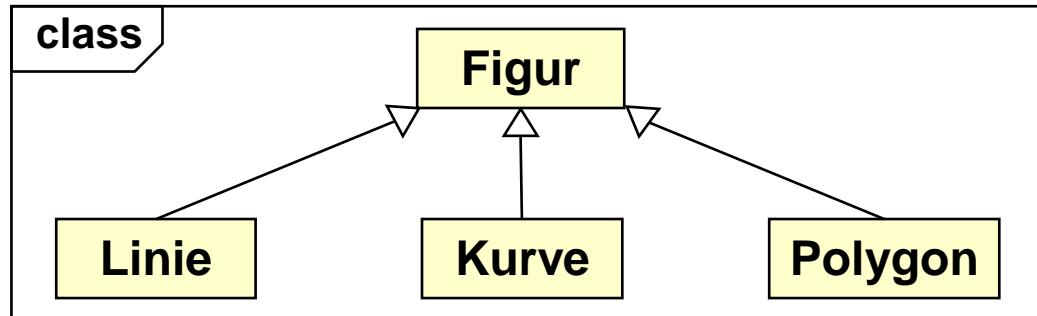
- Parametrisierter Typ = Der Typ der entsteht wenn man alle Typparameter durch Typen ersetzt
 - ◆ Im Beispiel: EmployeeSet = „Set<Employee>“

3.2.3.2 Vererbung und Implementierung

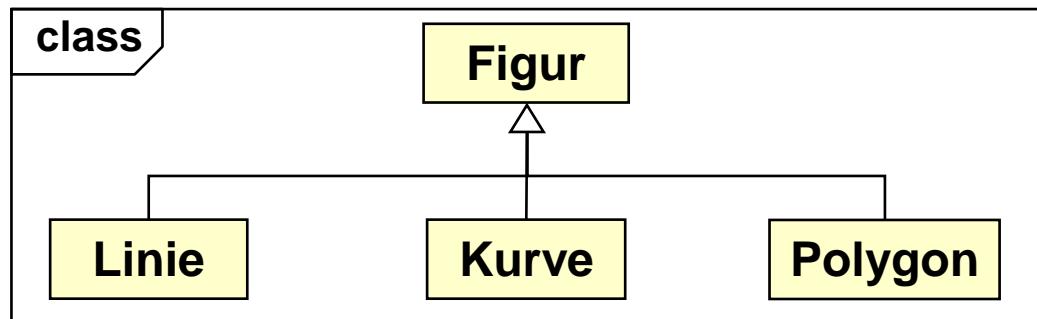
Vererbung
„Bietet“-Beziehung
„Braucht“-Beziehung

Klassendiagramm ► Vererbung

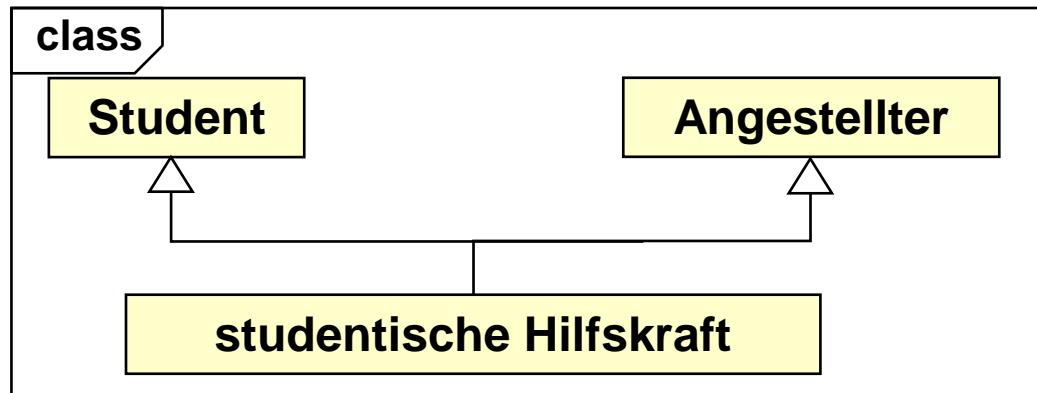
- "direkte" Darstellung



- "zusammengefasste" Darstellung

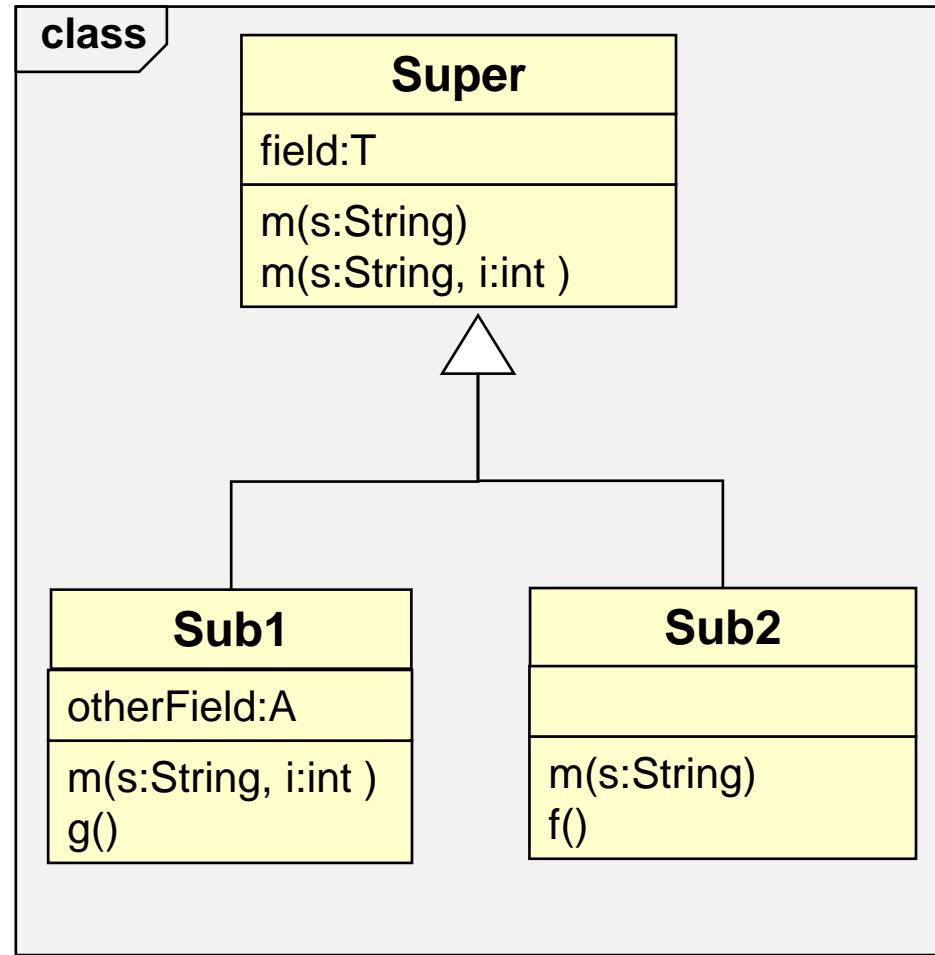


- Multiple Vererbung
 - ◆ Eine Klasse mit mehreren Oberklassen
 - ◆ Bsp: C++



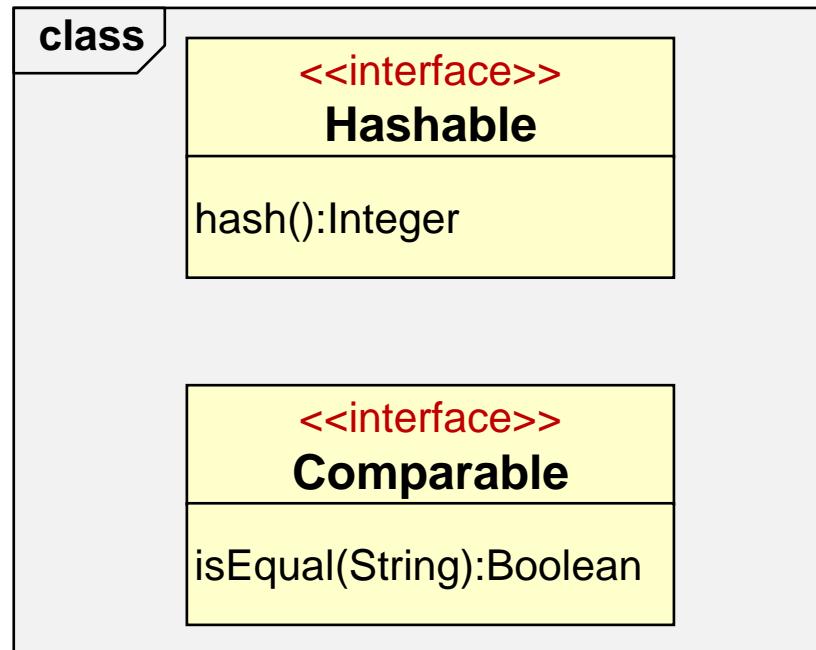
Klassendiagramm ► Vererbung

- Vererbung
 - ◆ geerbte Methoden / Attribute im Diagramm der UnterkLASSE **nicht** wiederholen
- Overriding
 - ◆ überschriebene Methoden im Diagramm der UnterkLASSE **wiederholen**
- Overloading
 - ◆ alle verschiedenen Signaturen angeben
 - ◆ auch überladene Methoden können in Untertypen überschreiben werden



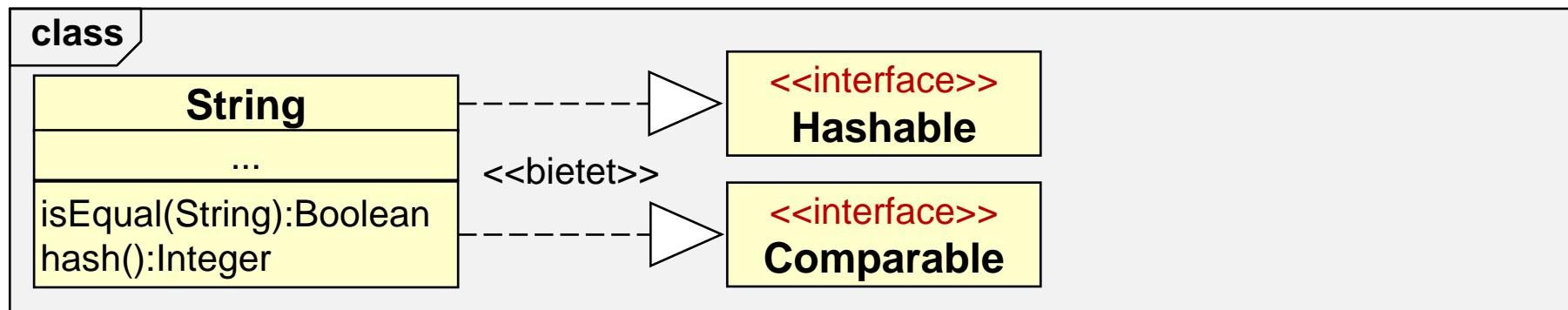
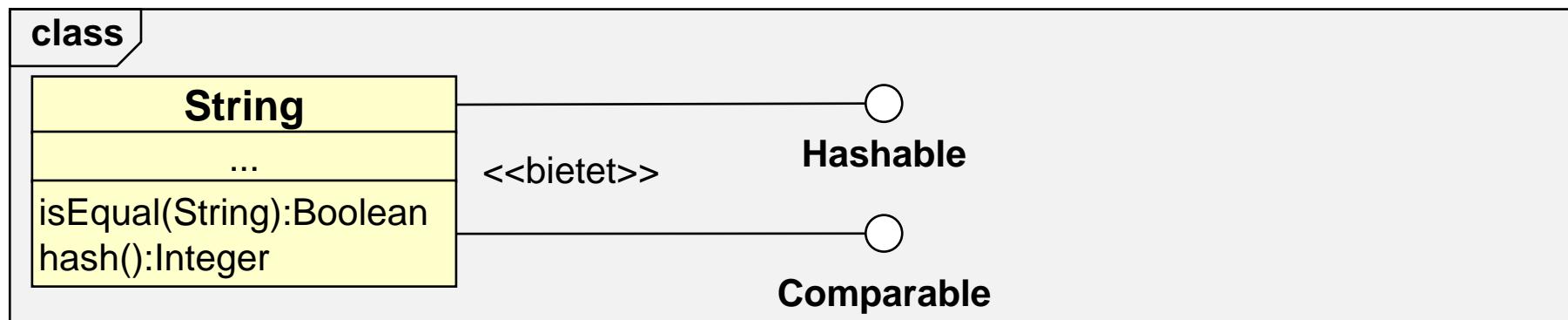
Klassendiagramm ► Interfaces

- Semantik
 - ◆ Interfaces enthalten nur Methodensignaturen und Ergebnistypen
- Notation
 - ◆ Klassensymbol mit Stereotyp **<<interface>>** im Namensfeld



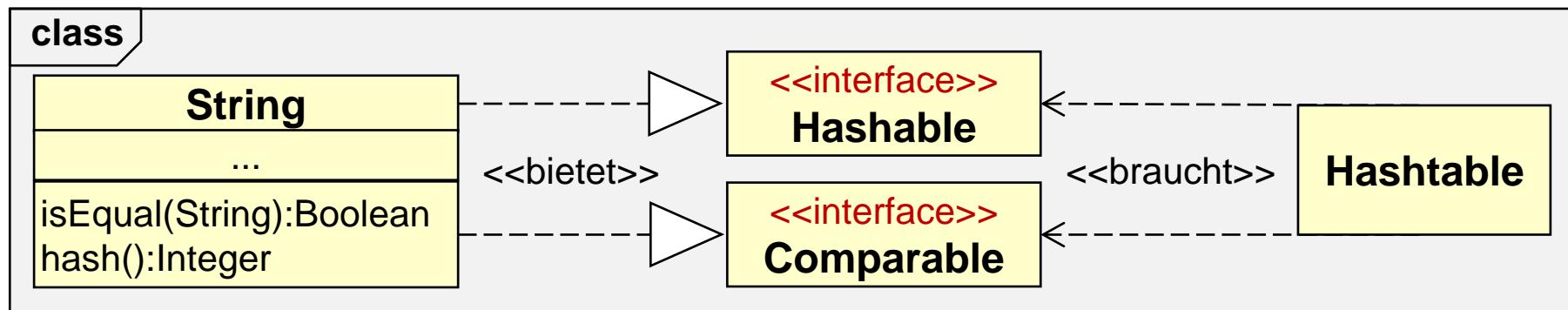
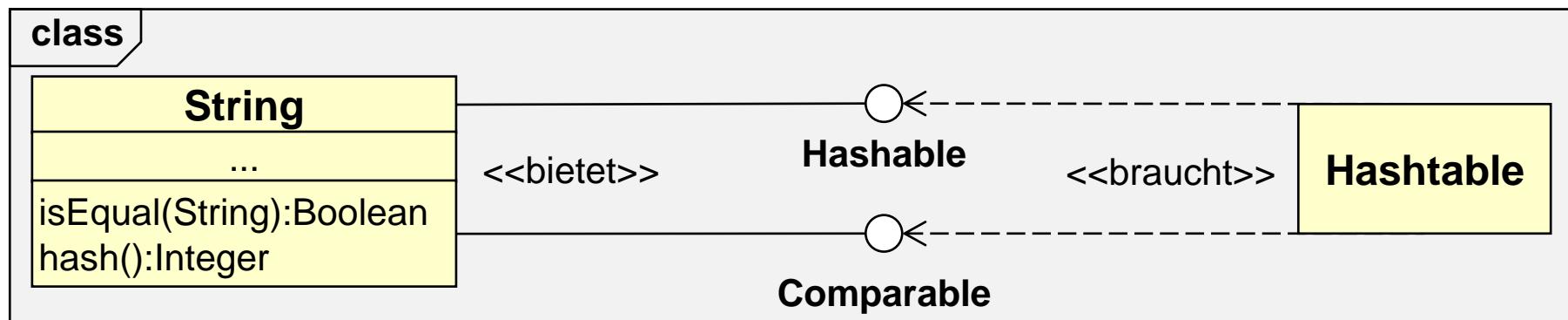
Klassendiagramm ► Implementierungs-Beziehung

- Semantik
 - ◆ „Komponente / Klasse implementiert Interface“
- Notation
 - ◆ gestrichelter Vererbungs-Pfeil oder „Lollipop“



Klassendiagramm ► „Braucht“-Beziehung

- Semantik
 - ◆ „Klasse benötigt Interface“
- Notation
 - ◆ gestrichelter spitzer Pfeil („Abhängigkeitspfeil“) + Stereotyp

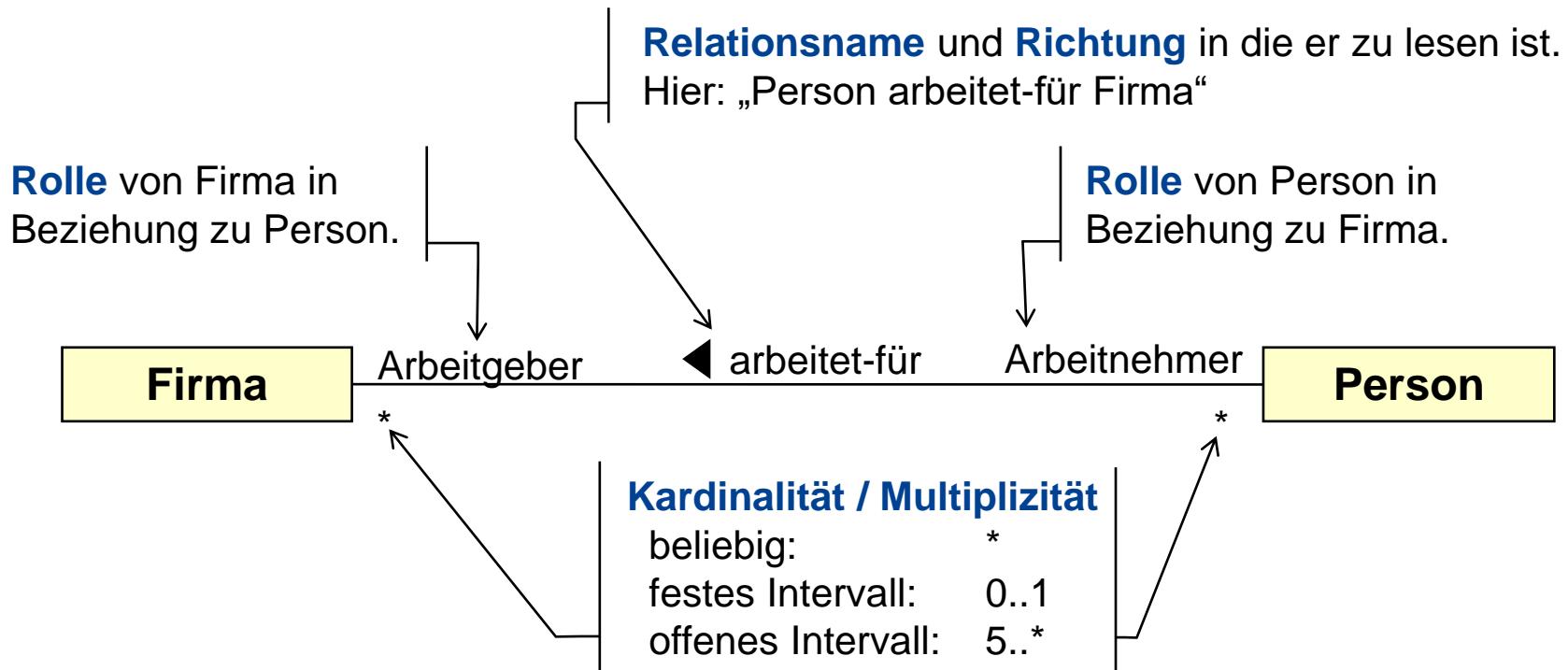


3.2.3.3 Assoziationen allgemein

Assoziationen
Rollen
Multiplizitäten

Klassendiagramm ► Assoziationen

- Assoziationen definieren Beziehungen zwischen **Instanzen** von Klassen.



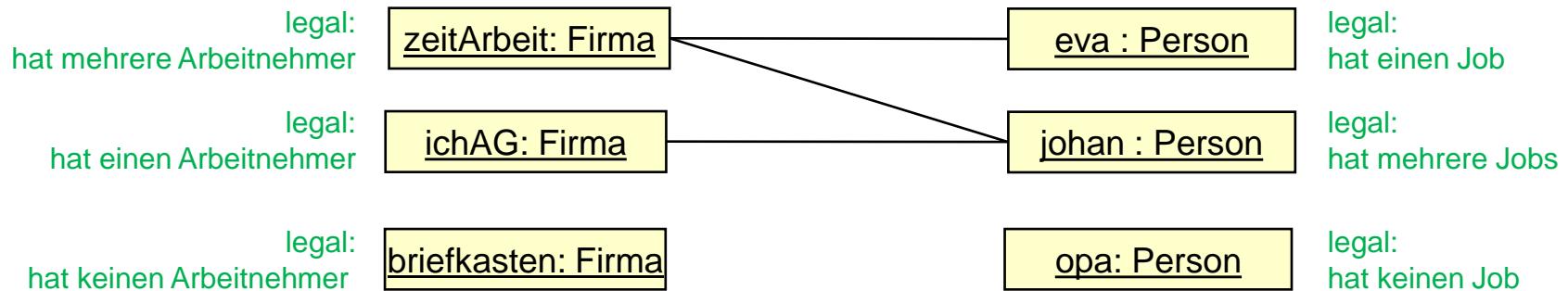
- Die Multiplizität am jeweiligen Ende einer Assoziation gibt an, mit wie vielen **Objekten** dieses Typs ein **Objekt** des gegenüberliegenden Typs in Beziehung stehen kann.

Klassendiagramm ▶ Assoziationen ▶ Multiplizität

- Assoziationen definieren Beziehungen zwischen **Instanzen** von Klassen. Die Multiplizität am jeweiligen Ende einer Assoziation gibt an, mit wie vielen **Zielobjekten** ein **Quellobjekt** in Beziehung stehen kann.
- Beispiel: **N zu M** Assoziation

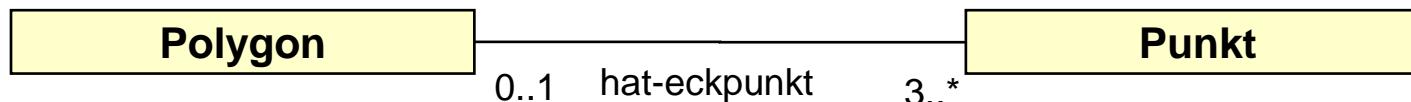


- Dazu passendes Objektdiagramm

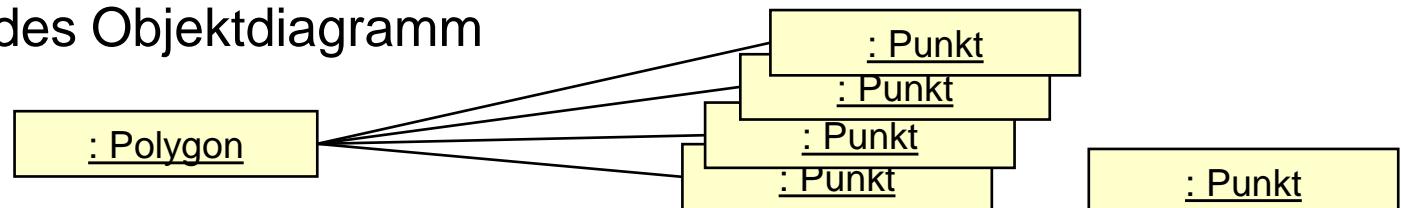


Klassendiagramm ► Assoziationen ► Multiplizität

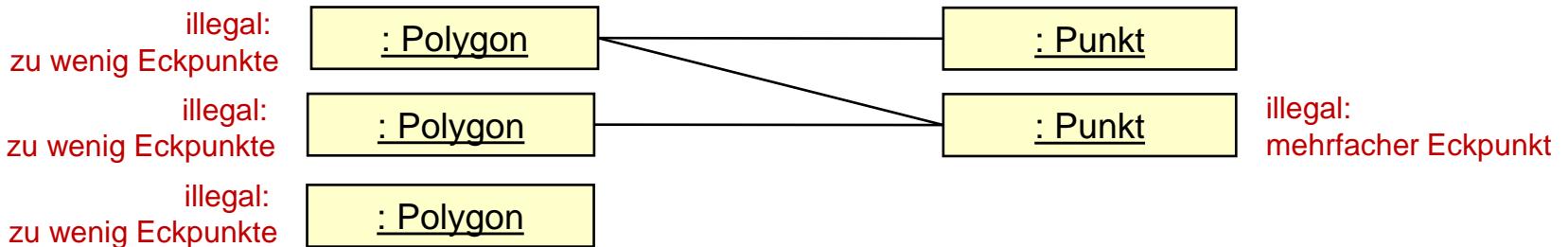
- Assoziationen definieren Beziehungen zwischen **Instanzen** von Klassen. Die Multiplizität am jeweiligen Ende einer Assoziation gibt an, mit wie vielen **Zielobjekten** ein **Quellobjekt** in Beziehung stehen kann.
- Beispiel: **1 zu N** Assoziation



- Dazu passendes Objektdiagramm



- Illegales Objektdiagramm

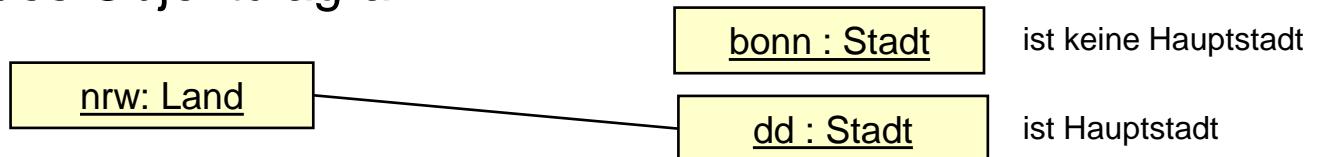


Klassendiagramm ► Assoziationen ► Multiplizität

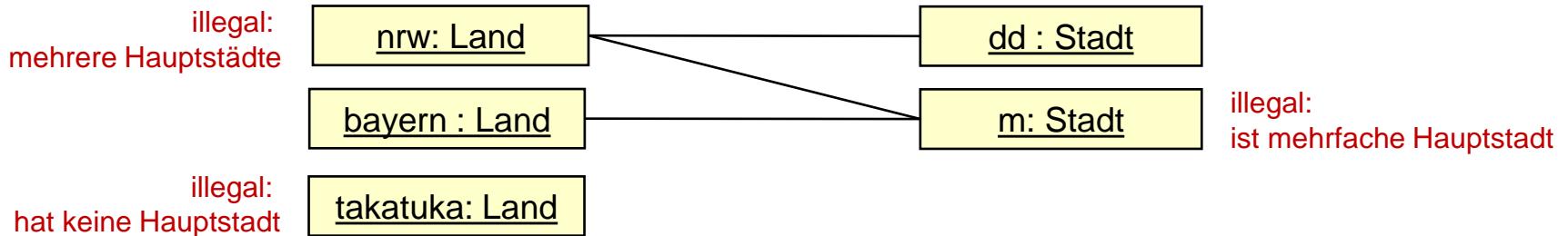
- Assoziationen definieren Beziehungen zwischen **Instanzen** von Klassen. Die Multiplizität am jeweiligen Ende einer Assoziation gibt an, mit wie vielen **Zielobjekten** ein **Quellobjekt** in Beziehung stehen kann.
- Beispiel: **1 zu 1** Assoziation



- Dazu passendes Objektdiagramm

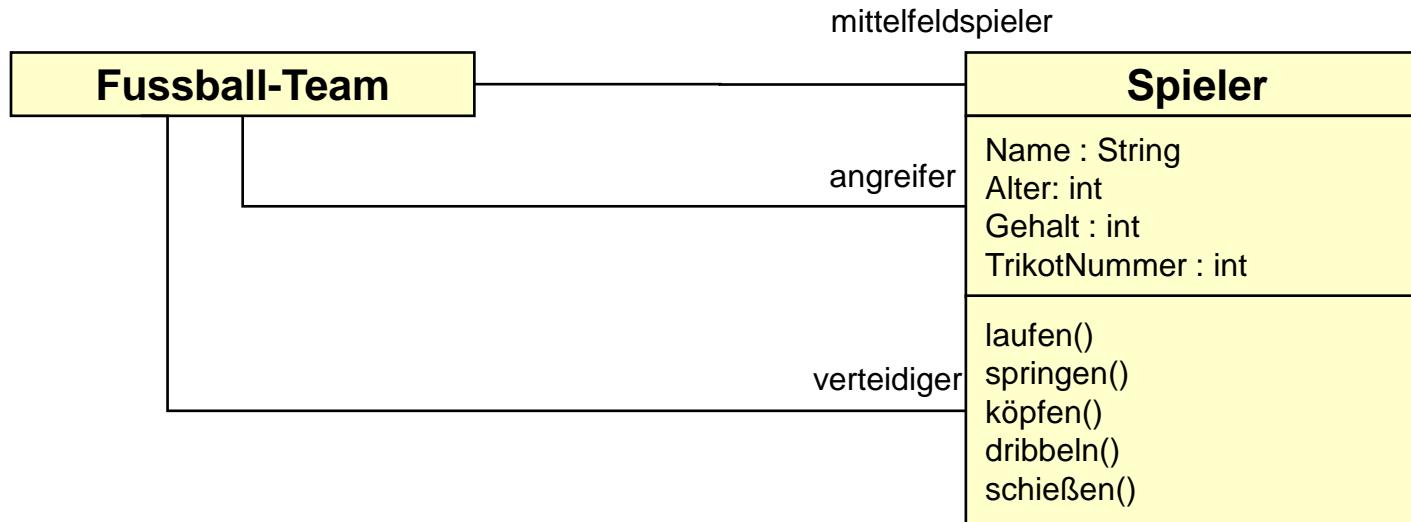


- Illegales Objektdiagramm



Klassendiagramm ► Rollen

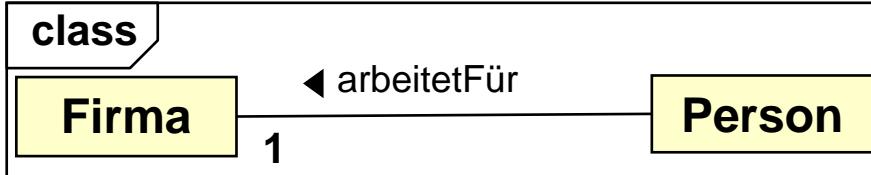
- Rollen beschreiben die Funktion eines Objektes in einer Beziehung
- Instanzen der gleichen Klasse können in verschiedenen Beziehungen verschiedene Rollen innehaben (z.B Spieler im Fussball-Team):



- Hausaufgabe: Vervollständigen Sie das Beispiel um Kardinalitäten und weitere Beziehungen / Rollen. Sprechen Sie mit Kollegen die Fußballexperten sind ;-) und überlegen Sie inwieweit verschiedene taktische Varianten („Vierer-Kette“, ...) im Diagramm ausdrückbar sind.

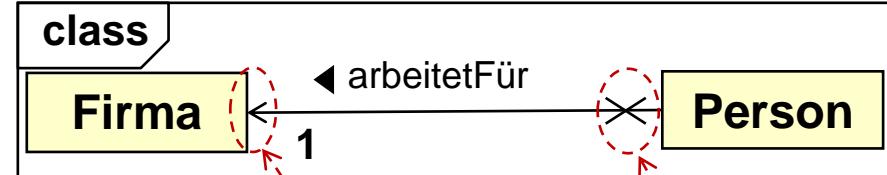
Klassendiagramm ▶ Assoziationen ▶ Navigation

Unspezifizierte Navigation



- Syntax
 - ◆ Einfaches Linien-Ende
 - ◆ ... ohne Pfeil oder Kreuz
- Semantik
 - ◆ Navigation ist noch nicht spezifiziert
 - ◆ ... aber im Prinzip noch in jede Richtungen möglich
 - ◆ Im Beispiel: Noch nicht festgelegt
 - ⇒ Kennt Person ihren Arbeitgeber?
 - ⇒ Kennt Firma ihre Angestellten?
- Nutzung
 - ◆ Konzeptuelle Sicht

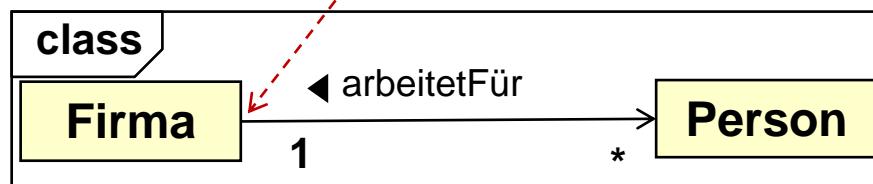
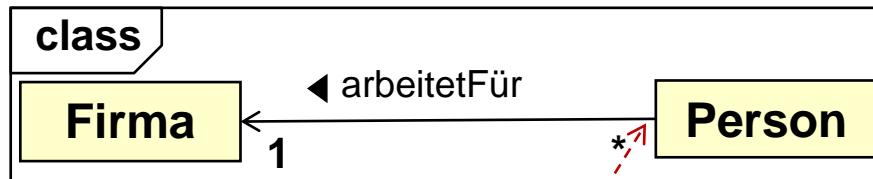
Explizite Navigationsangaben



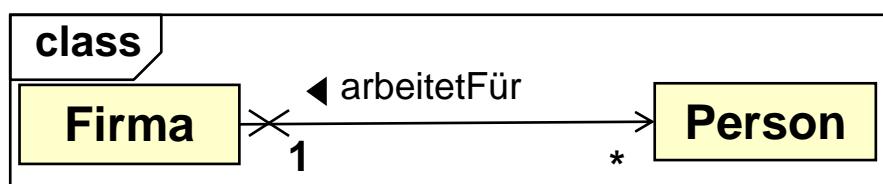
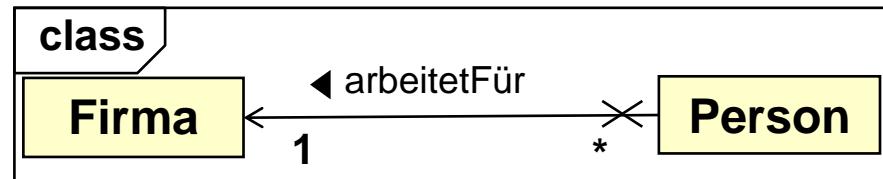
- Syntax
 - ◆ Pfeil (=Navigationsrichtung)
 - ◆ Kreuz (= nicht navigierbares Ende)
- Semantik
 - ◆ Navigation in Pfeilrichtung möglich
 - ◆ Navigation in Kreuzrichtung nicht möglich
 - ◆ Im Beispiel
 - ⇒ Person kennt ihren Arbeitgeber
 - ⇒ Firma kennt ihre Angestellten nicht
- Nutzung
 - ◆ Implementierungssicht

Klassendiagramm ▶ Assoziationen ▶ Navigation ▶ Varianten

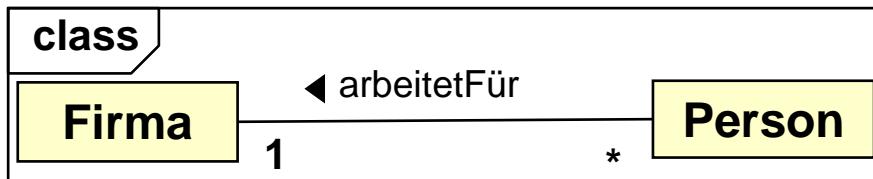
Teilweise unspezifizierte Navigation



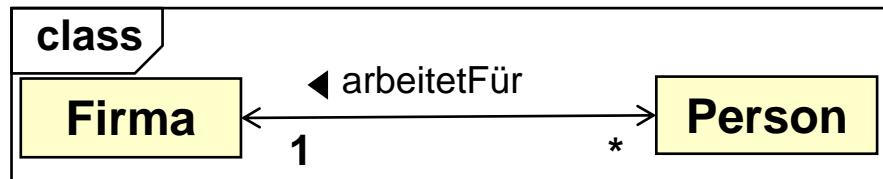
Unidirektionale Navigation



Völlig unspezifizierte Navigation



Bidirektionale Navigation



Fazit „Assoziationen“

- Alles bisher gesagte gilt für alle Arten von Assoziationen!
- ... , auch für die im Folgenden betrachteten speziellen Varianten:
 - ◆ Qualifizierte Assoziation
 - ◆ Aggregation
 - ◆ Komposition

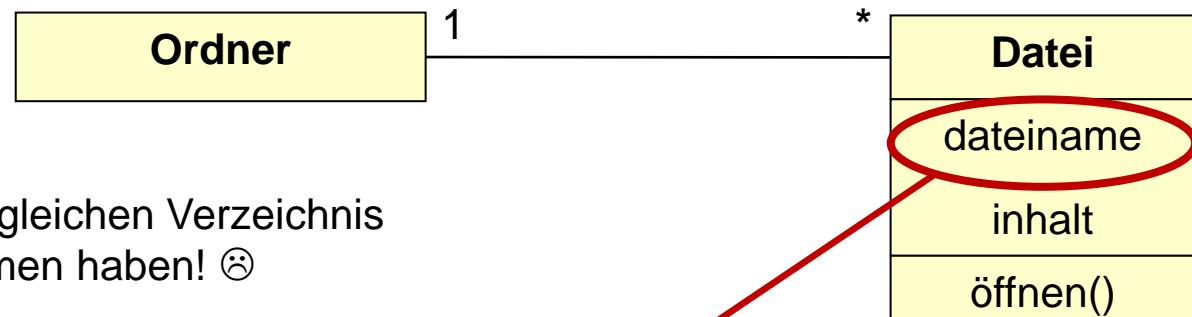
3.2.3.4 Qualifizierte Assoziationen

▶ Qualifizierte Assoziation

- Eine „Qualifikation“ (engl. „Qualifier“) präzisiert die Multiplizität einer Assoziation

Ohne Qualifikation

Ein Verzeichnis enthält
viele Dateien.

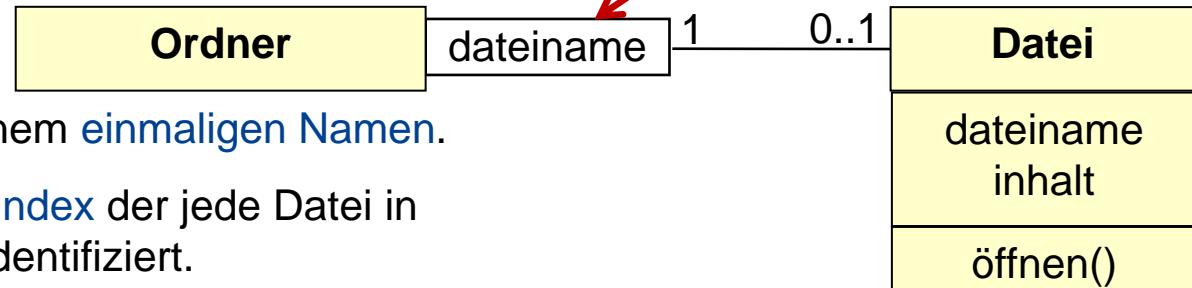


Verschiedene Dateien im gleichen Verzeichnis
können den gleichen Namen haben! ☹

Mit Qualifikation

Ein Verzeichnis enthält
viele Dateien, jede mit einem **einmaligen Namen**.

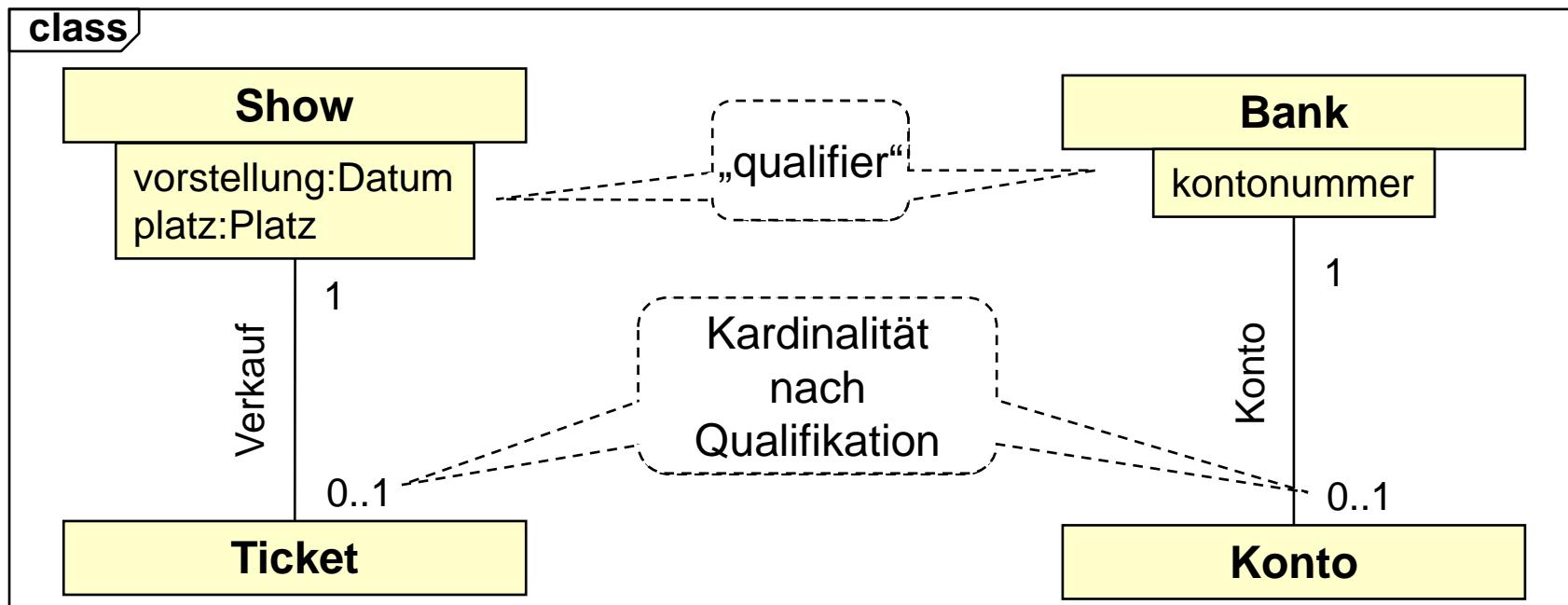
Der Dateiname dient als **Index** der jede Datei in
einem Ordner eindeutig identifiziert.



- Qualifikation modelliert **Indizierung** der Elemente am gegenüberliegenden Ende der Assoziation durch den Wert des „Qualifiers“ („Keys“)

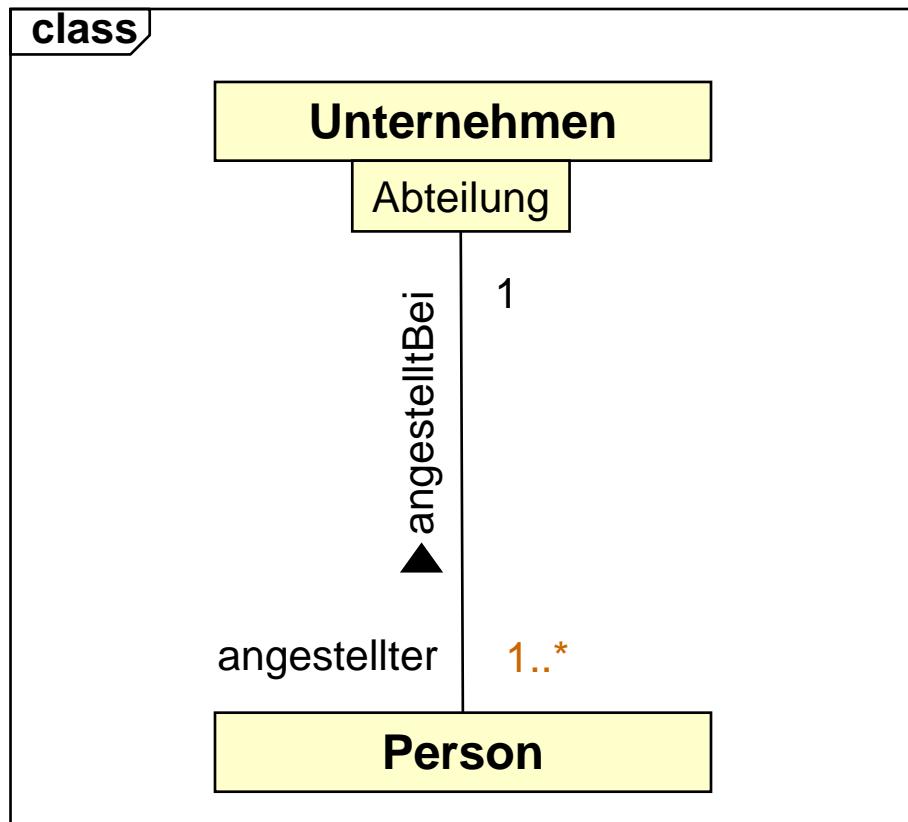
▶ Qualifizierte Assoziation

- Der Qualifier modelliert Indizierung aus Sicht der der Partnerklasse
 - Effekt: Kardinalität „am anderen Ende“ der Beziehung wird reduziert
 - ⇒ Oft nur noch 0 oder 1: Entweder gibt es kein passendes Objekt oder es ist durch den Qualifier eindeutig bestimmt
 - Beispiel: Aus Sicht der Partnerklasse „Bank“ ist die Kontonummer der Index, der eine Instanz der Klasse „Konto“ eindeutig bezeichnet.



▶ Qualifizierte Assoziation

- Qualifier können auch **partielle Indizes** sein
 - ◆ Sie bestimmen evtl. nicht ein einziges Element, sondern eine **(Teil)Menge**
 - ◆ Beispiel: In einer Abteilung können mehrere Angestellte beschäftigt sein



3.2.3.5 Aggregation und Komposition

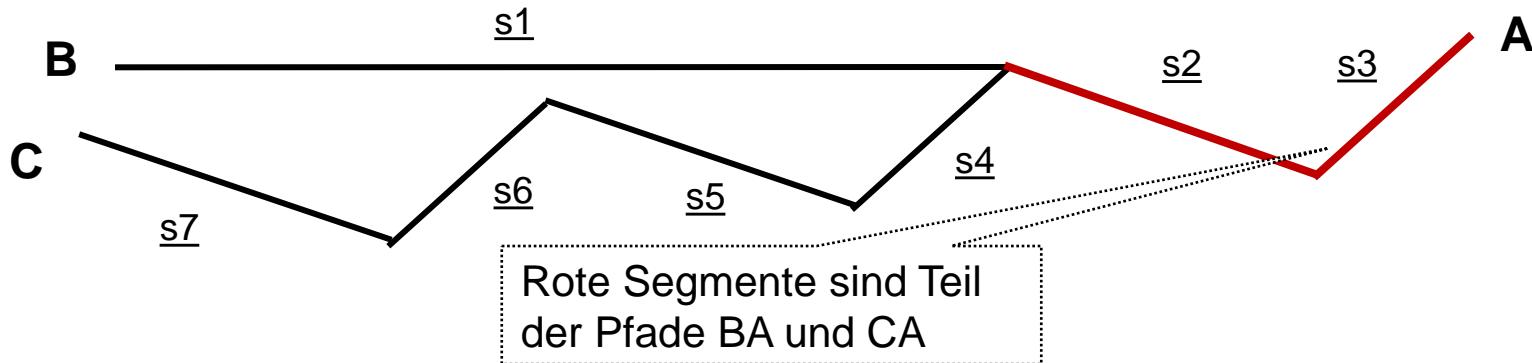
„Ist Teil von“-Beziehung
Kaskadierende Operationen
Exklusivität und Existenzabhängigkeit
Die vier Kategorien
Constraints und Abhängigkeiten

Klassendiagramm ► Aggregation

- Aggregation = „ist Teil von“-Beziehung
 - Teil darf in mehreren „Ganzen“ enthalten sein
 - Seine Lebensdauer ist nicht von der des Ganzen abhängig

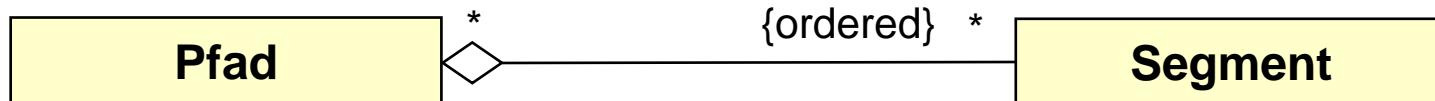


- Beispiel: Segmente können Teile des gleichen Pfades sein

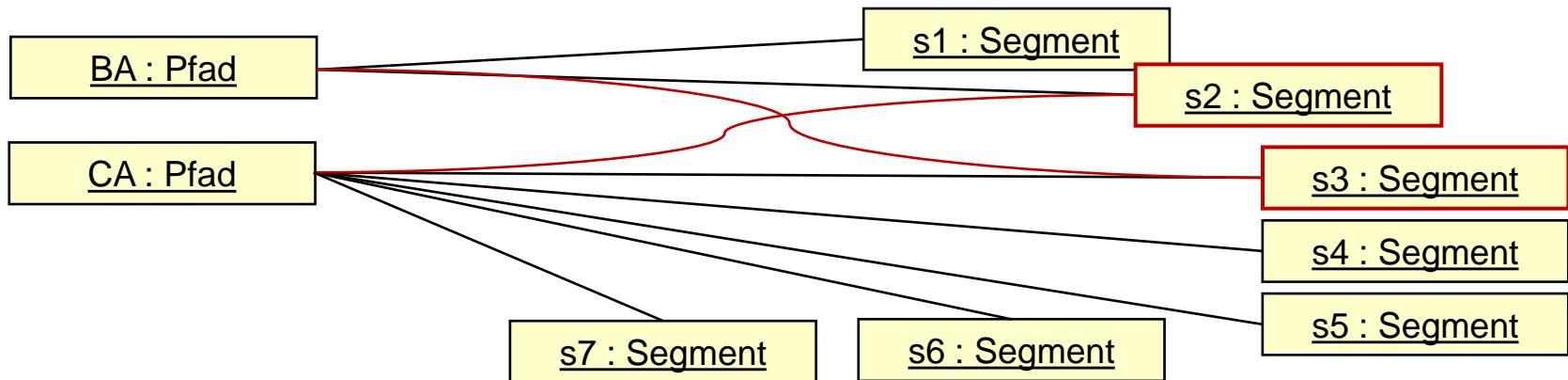


Klassendiagramm ► Aggregation

- Aggregation = „ist Teil von“-Beziehung
 - ◆ Teil darf in mehreren „Ganzen“ enthalten sein
 - ◆ Seine Lebensdauer ist nicht von der des Ganzen abhängig

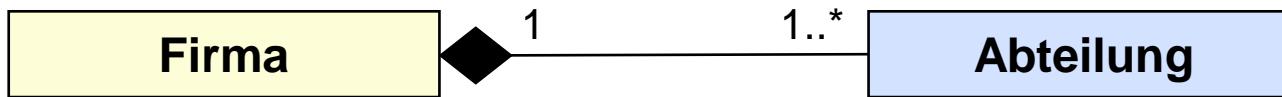


- Dazu passendes Objektdiagramm

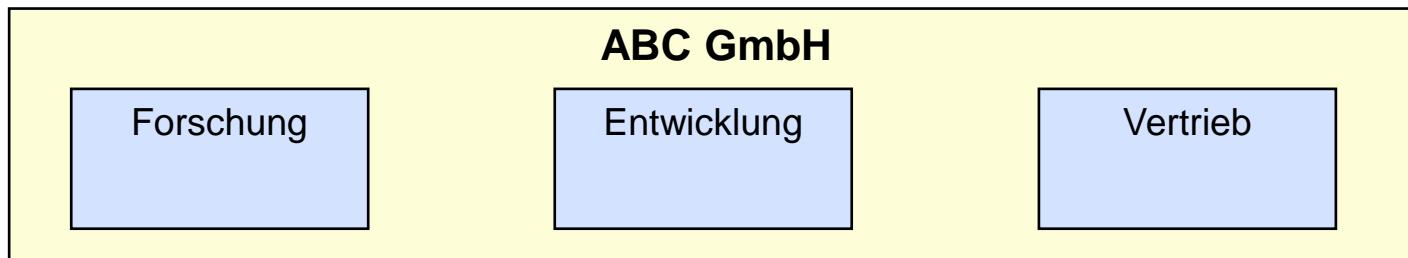


Klassendiagramm ► Komposition → Firma-Abteilung

- Komposition = „ist ausschließliches Teil von“-Beziehung
 - ◆ Teil kann nur in einem Ganzen enthalten sein
 - ◆ Lebensdauer des Teils endet mit Lebensdauer des Ganzen
 - ◆ Beispiel: Abteilung gehört zu genau einer Firma und kann ohne Firma nicht existieren

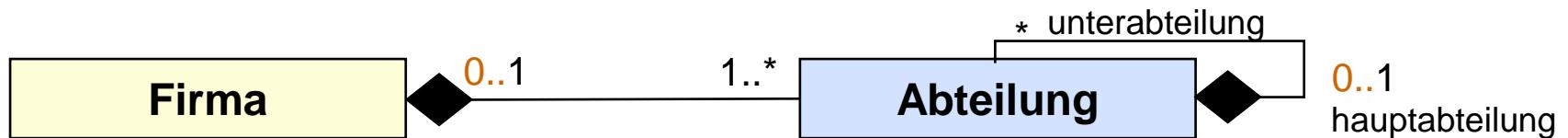


- Dazu passende Veranschaulichung der realen Welt

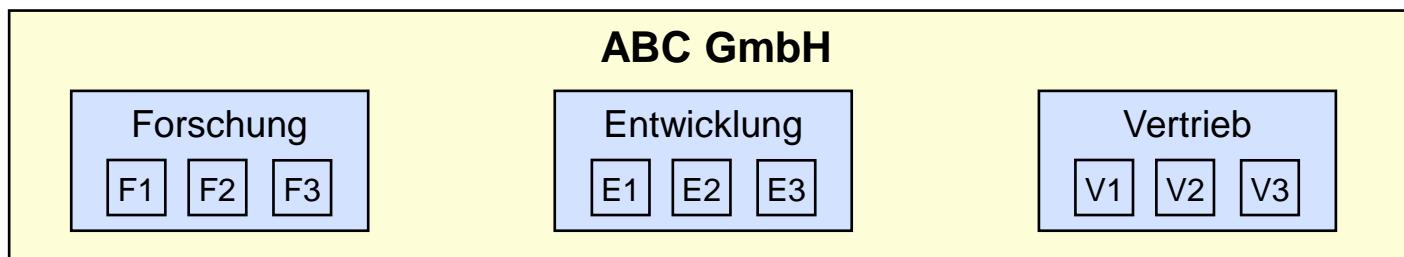


Klassendiagramm ► Komposition → Firma-Abteilung-Unterabteilung

- Komposition = „ist ausschließliches Teil von“-Beziehung
 - ◆ Teil kann nur in einem Ganzen enthalten sein
 - ◆ Lebensdauer des Teils endet mit Lebensdauer des Ganzen
 - ◆ Beispiel: Abteilung gehört zu genau einer Firma und kann ohne Firma nicht existieren

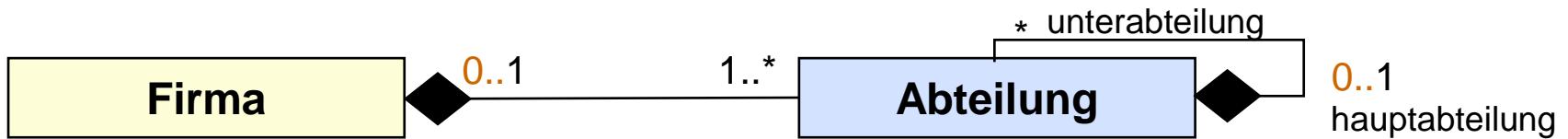


- ◆ Unterabteilung ist nicht gleichzeitig direkt in Firma enthalten
 - ⇒ Kardinalität – auch 0 erlauben (erster Schritt, reicht aber nicht)
 - ⇒ explizite Constraints { **context** Abteilung : (Firma <> null) implies (hauptabteilung = null) } { **context** Abteilung : (hauptabteilung <> null) implies (Firma = null) }

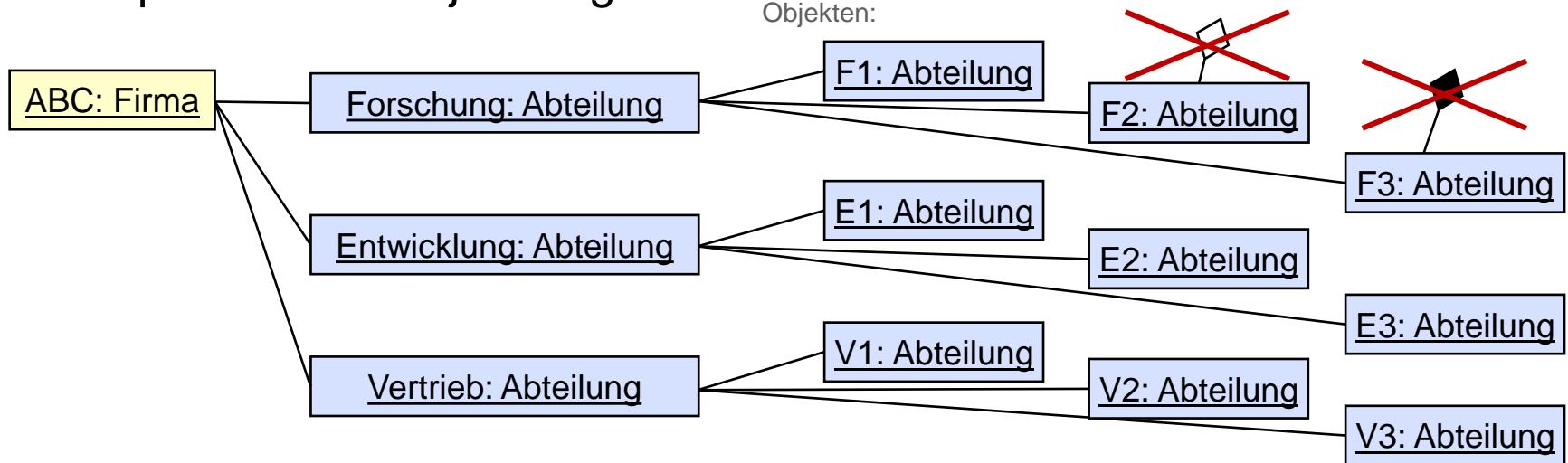


Klassendiagramm ► Komposition

- Komposition = „ist ausschließliches Teil von“-Beziehung
 - ◆ Teil kann nur in einem Ganzen enthalten sein
 - ◆ Lebensdauer des Teils endet mit Lebensdauer des Ganzen
 - ◆ Beispiel: Abteilung gehört zu genau einer Firma und kann ohne Firma nicht existieren



- Dazu passendes Objektdiagramm



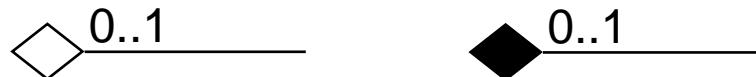
► Aggregation = „Ist Teil von“-Beziehung

- Modelliert **Einheit** eines "Ganzen" mit seinen "Teilen"
- Impliziert **meist kaskadierende Operationen**
 - ◆ Operation auf Ganzem wird auch auf allen Teilen durchgeführt
 - ◆ Beispiel: Verschiebung eines Rechtecks verschiebt alle seine Kanten
- Impliziert **manchmal Existenz-Abhängigkeit**
 - ◆ Teil existiert nicht ohne Ganzem
 - ◆ Entspricht Kaskadierung des Löschens: Teile werden mit gelöscht
- Impliziert **manchmal Exklusivität**
 - ◆ Teil ist in genau einem Ganzen enthalten
- Im Zweifelsfall die allgemeinere Notation wählen und speziellere Semantik durch **zusätzliche Bedingungen** explizit machen:
 - ◆ Kardinalitäten
 - ⇒ Exklusivität kann durch Kardinalität 1 ausgedrückt werden
 - ⇒ Manchmal auch 0..1 (→ übernächste Folie)
 - ◆ Constraints
 - ⇒ Existenzabhängigkeit als Constraint (→ überüber-nächste Folie)
- Mögliche semantische Kategorien → nächste Folie

► Die vier Kategorien

	exklusiv (nur in einem Ganzen)	nicht exklusiv (in mehreren Ganzen)
existenz-abhängig (Propagierung der Löschoperation)	<p>Komposition</p> <pre> classDiagram class Firma class Abteilung Firma "1" *-- "*" Abteilung </pre>	<p>Aggregation und Existenzabhängigkeit als Constraint (→ übernächste Seite)</p> <pre> classDiagram class Spalte class Zeile class Zelle Spalte "1" *-- "*" Zelle Zeile "1" *-- "*" Zelle </pre>
existenz-unabhängig (keine Propagierung der Löschoperation)	<p>Aggregation und Exklusivität als Kardinalität 1 oder 0..1</p> <pre> classDiagram class Auto class Reifen Auto "0..1" *-- "4" Reifen </pre>	<p>Aggregation (ohne besondere Einschränkungen)</p> <pre> classDiagram class Projekt class Mitarbeiter Projekt "*" *-- "*" Mitarbeiter </pre>

- Aggregation und auch Komposition haben oft Kardinalität 0..1 statt 1

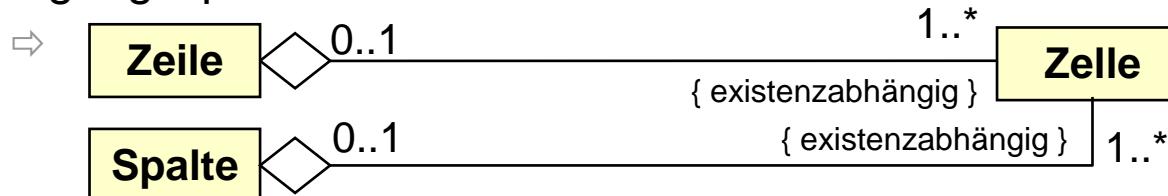


- Kardinalität 0 bedeutet hier:
 - Teile können *temporär* unabhängig existieren, insbesondere unabhängig von einem Ganzen *erzeugt* werden
 - Sehr praxisrelevanter Fall: Vieles wird „bottom-up“ produziert
- Kardinalität 1 bedeutet hier:
 - Sobald die Teile einem Ganzen zugeordnet wurden, sind sie exklusiv darin enthalten

Bedingungen („Constraints“)

- Bisher wurden nur besonders **häufige** Bedingungen modelliert, über
 - ◆ Spezialnotationen (z.B. „Komposition“) oder
 - ◆ Kardinalitäten (z.B. 1..4)
- Das reicht oft nicht aus! Es gibt daher auch eine Notation für **beliebige** Bedingungen
 - ◆ Angabe von Bedingung in geschweiften Klammern: { Bedingung }
- Alternative Formulierung von Bedingungen

- ◆ Umgangssprachlich



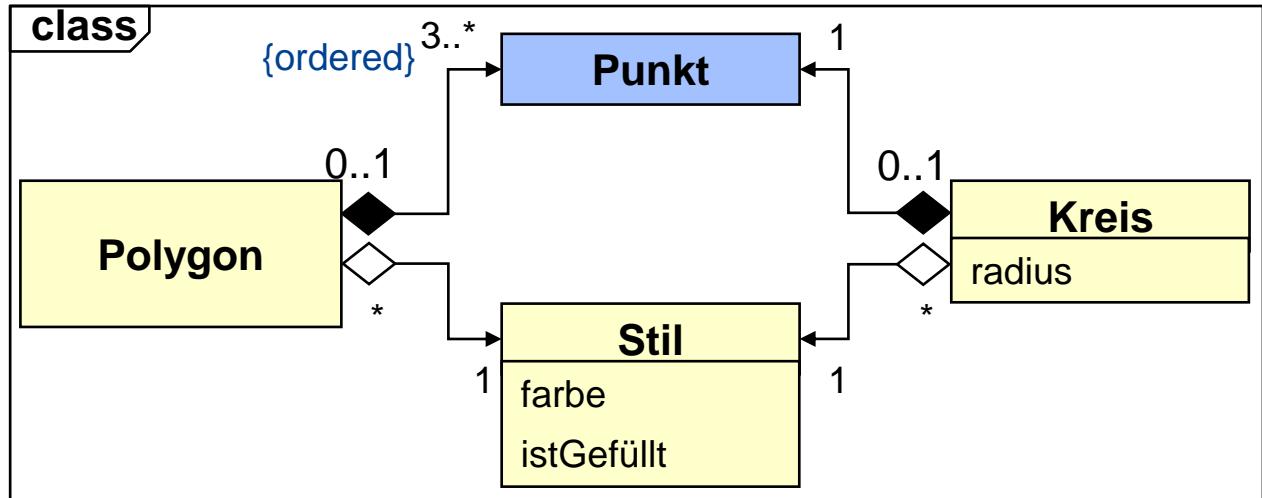
- ◆ Vordefinierte Schlüsselworte:

⇒ abstract, readOnly, query, leaf, ordered, unique, set, bag, ...

- ◆ ... oder formal mittels "Object Constraint Language"

⇒ siehe Kapitel 2. OOM, Abschnitt zu DBC

Statisches Modell: Aggregation und Komposition – Gemeinsames Beispiel

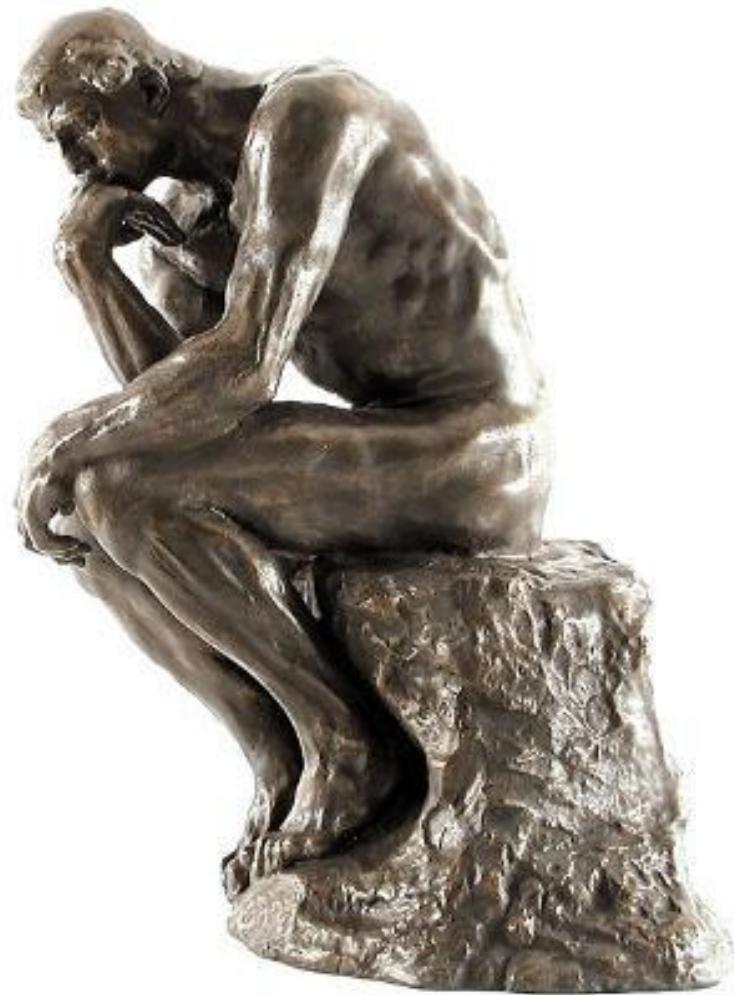
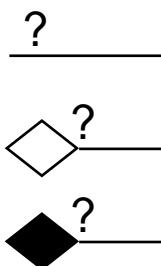


- Komposition
 - ◆ Für Beziehung Polygon → Punkt und Kreis → Punkt ...
 - ◆ ... weil man nicht möchte, dass der Mittelpunkt des Kreises auch Eckpunkt eines Polygons sein kann.
 - ◆ Dann würde nämlich ein Verschieben des Kreises das Polygon verformen!
- Aggregation
 - ◆ Für Beziehung Polygon → Stil und Kreis → Stil ...
 - ◆ ... weil man gern möchte, dass anpassen eines gemeinsamen Stils alle Formen ändert die ihn benutzen

Denksportaufgabe

- Wie würden Sie die Beziehung einer Datei zu einem Ordner modellieren?
 - ◆ Assoziation, Aggregation oder Komposition?

- Bedenken Sie:
 - ◆ Die Datei ist zu jedem Zeitpunkt exklusiver Teil eines Ordners,
 - ◆ ... kann aber in einen anderen Ordner verschoben werden
 - ◆ Wenn der Ordner gelöscht wird, wird die Datei mit gelöscht.



Klassendiagramm ▶ Assoziationen ▶ Abgeleitete Assoziationen, Rollen, Felder

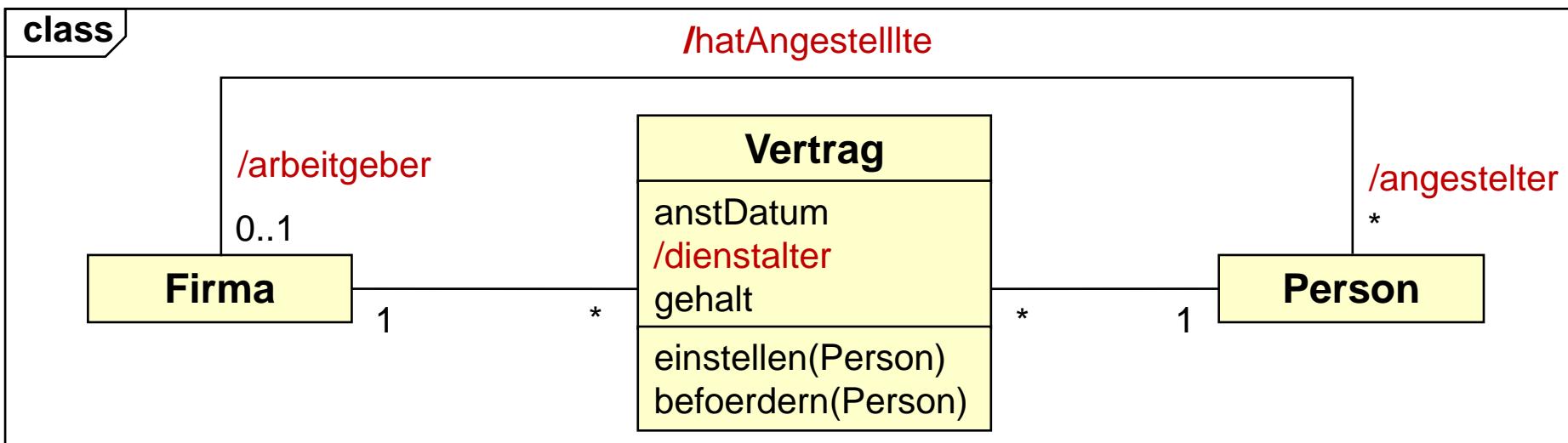
- Notation: Querstrich „/“ vor Namen von Assoziation, Rolle oder Attribut
- Bedeutung: Das Element ist nicht explizit gespeichert sondern ergibt sich aus anderen Elementen des Modells
- Beispiel: Die Beziehung **/hatAngestellte**, die Rollen **/arbeitgeber** und **/angesteller** und das Attribut **/dienstalter** sind aus dem Rest des Modells herleitbar
- Formal: Entspricht einer Klasseninvariante, z.B.

Zur Erinnerung:

(1) Alle im Modell definierten Namen dürfen im OCL-Ausdruck genutzt werden.

(2) Wenn keine expliziten Rollen angegeben sind, sind der Typ am Ende einer Assoziation implizit auch der Name der entsprechenden Rolle. —

context Person inv:
arbeitgeber = vertrag.firma



3.2.3.6 Forward Engineering: Klassendiagramm → Code

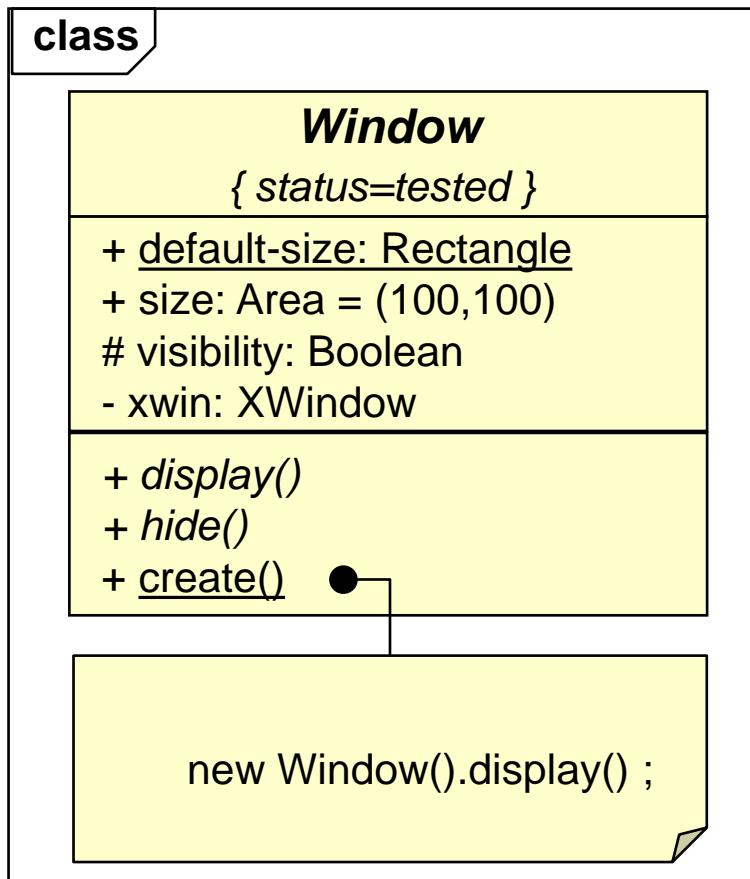
Klassendiagramm ohne Assoziationen → Code

Klassendiagramm mit Assoziationen → Klassendiagramm ohne Assoziationen

Assoziationen sind (implizite) Klassen

UML → Code

UML ohne Assoziationen



Java-Quellcode

```
@status{ tested = true }

abstract class Window {

    public static Rectangle default-size ;
    public Area size = new Area(100,100) ;
    protected Boolean visibility ;
    private Xwindow xwin ;

    public abstract display() {}
    public abstract hide() {}
    public static create() {
        Window w = new Window() ;
        w.display() ;
        return w;
    }
}
```

Umsetzung abgeleiteter Attribute und Assoziationen

- Abgeleitetes Attribut = Getter-Methode
 - ◆ Getter berechnet den aktuellen Wert
 - ◆ Kein Setter → Was hieße es, den Kontostand zu setzen ohne eine entsprechende Buchung durchzuführen???
- Implementierungsoptionen
 - ◆ Dynamische Berechnung
 - ⇒ Die erforderlichen Daten werden beim Aufruf berechnet
 - ⇒ Werte sind garantiert aktuell
 - ⇒ Evtl. hoher Berechnungsaufwand
 - ◆ Redundante Speicherung
 - ⇒ Das Berechnungsergebnis wird lokal gespeichert
 - ⇒ Weniger Berechnungsaufwand
 - ⇒ Konsistenzwahrung mit Werten aus denen die Berechnung erfolgte erfordert Implementierung eines Observer-Mechanismus (s. nächstes Kapitel)
 - Das Objekt das das abgeleitete Attribut enthält ist Observer der Objekte, die die Quelldaten enthalten

Klassendiagramm ▶ Assoziationen ▶ Implementierung

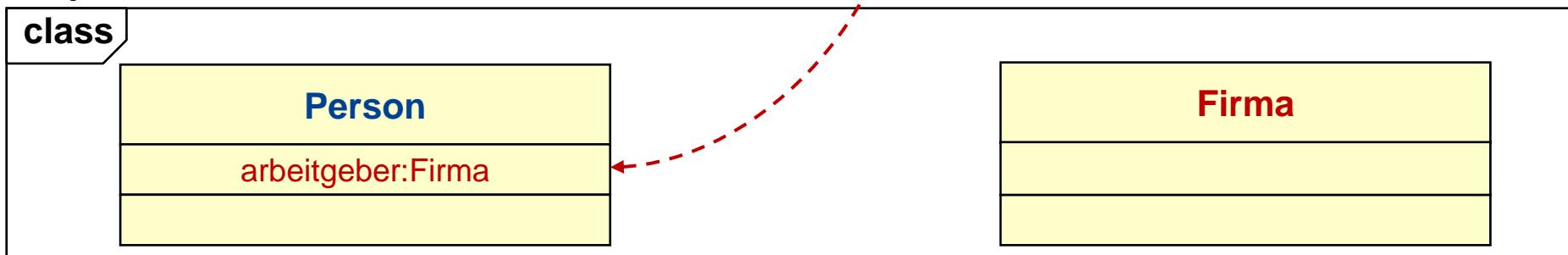
- Die Umsetzung von Assoziationen als Code wird in den nächsten Folien für alle Kombinationen folgender Fälle dargestellt:
 - ◆ Navigation
 - ⇒ Unidirektional: Implementierung als Instanzvariable
 - ⇒ Bidirektional: Implementierung als zwei Instanzvariablen oder eigene Klasse
 - ◆ Kardinalität
 - ⇒ 1: Einfache Instanzvariable
 - ⇒ *: Collection
- Dabei wird die Umsetzung nicht durch sprachspezifischen Code dargestellt, sondern als Transformation eines UML-Diagramms das eine Assoziation enthält in eines das keine enthält
- Dessen Umsetzung in Code haben Sie auf der vorvorherigen Folie gesehen.

▶ Unidirektionale Assoziation zu 1

Objektentwurfsmodell mit Assoziation



Objektentwurfsmodell ohne Assoziation



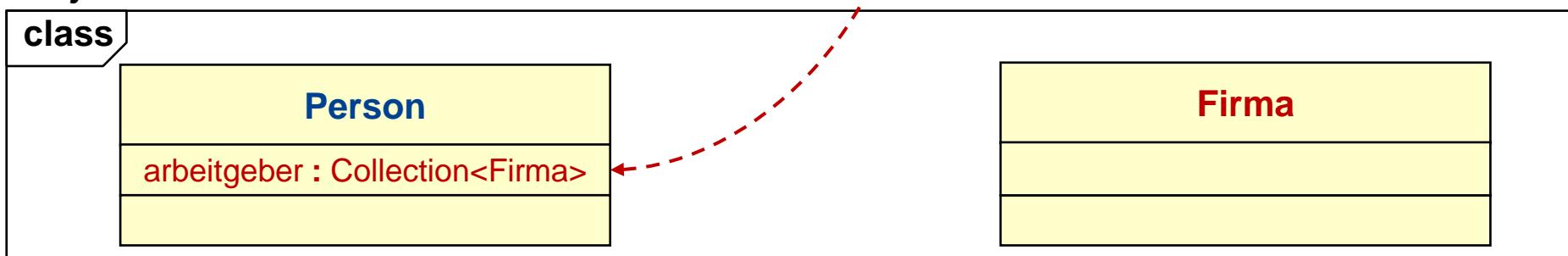
- Unidirektionale 1:1 Assoziationen sind einfach:
 - ◆ Durch **Instanzvariable** in der „referenzierenden“ Klasse implementieren
 - ⇒ Der Name der Instanzvariablen ist der Rollenname am Pfeilende
 - ⇒ Der Typ der Instanzvariablen ist die referenzierte Klasse.
 - ◆ Die „referenzierte“ Klasse hat **keine** solche Instanzvariable

▶ Unidirektionale Assoziation zu N

Objektentwurfsmodell mit Assoziation



Objektentwurfsmodell ohne Assoziation



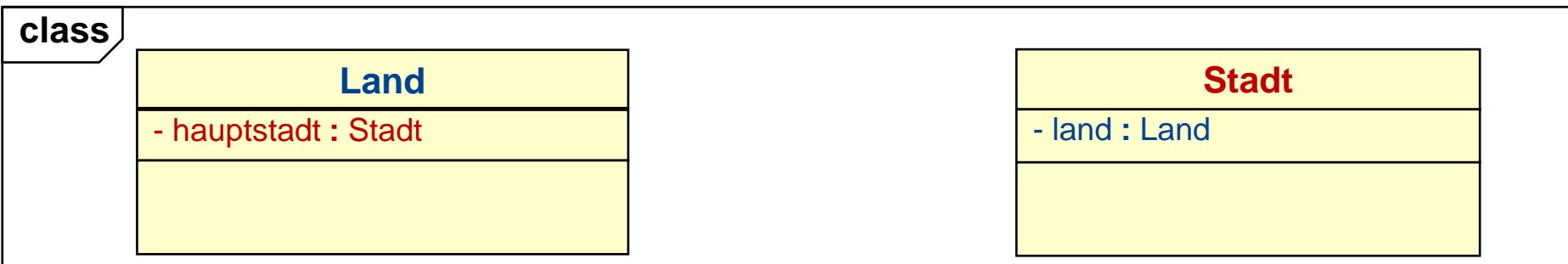
- Unidirektionale 1:N Assoziationen sind genauso einfach:
 - ◆ Durch **Instanzvariable** in der „referenzierenden“ Klasse implementieren
 - ⇒ Der Name der Instanzvariablen ist der Rollename am gegenüberliegenden Pfeilende
 - ⇒ Der Typ der Instanzvariablen ist eine **Collection** von Elementen der referenzierten Klasse.
 - ◆ Anmerkung: In Java ist *Collection* das Interface zu Klassen wie *ArrayList*, *Set*, etc.

▶ Bidirektionale Assoziation (1:1)

Objektentwurfsmodell mit Assoziation



Objektentwurfsmodell ohne Assoziation



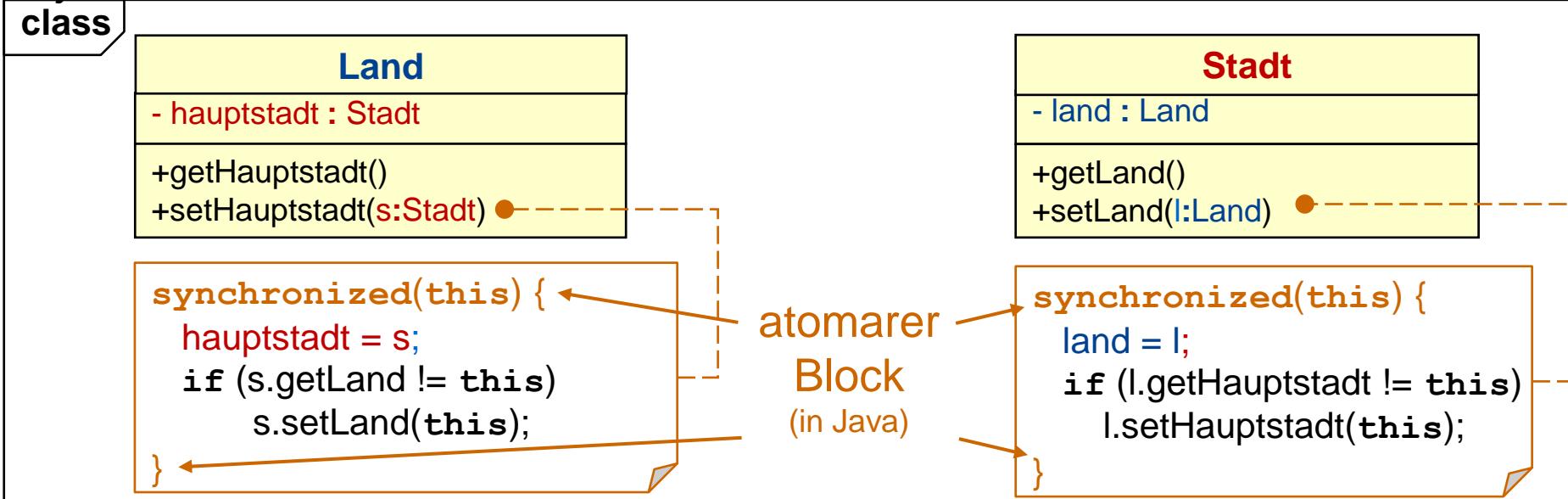
- Im Prinzip ist eine bidirektionale Assoziation das Gleiche wie zwei gegengleich zeigende Unidirektionale
- Problem: Bei Setzen einer Referenz muss der Rückverweis mit gesetzt werden
 - ◆ Man muss sicherstellen, dass beide Zuweisungen atomar geschehen
 - ◆ In Java: „synchronized“ Block oder „synchronized“ Methode (→ nächste Folie)

▶ Bidirektionale Assoziation (1:1)

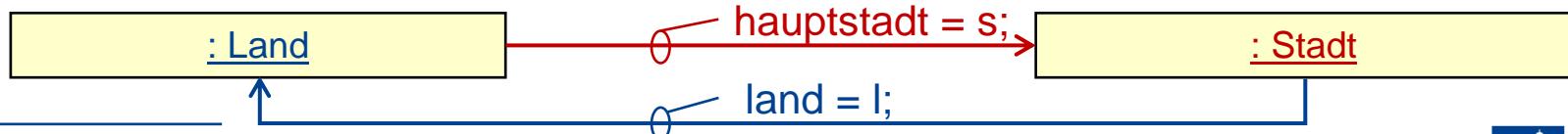
Objektentwurfsmodell mit Assoziation



Objektentwurfsmodell ohne Assoziation



Beispiel-Instanziierung



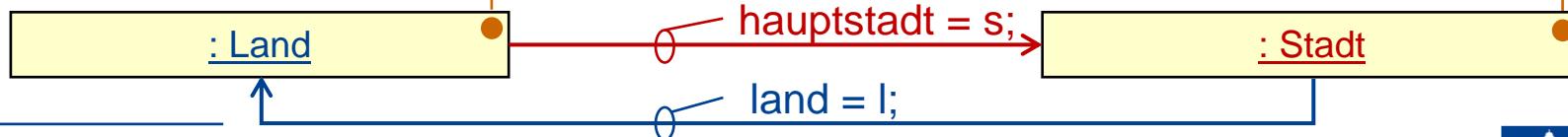
▶ Bidirektionale Assoziation (1:1)

- Achtung: Das Schema auf der vorherigen Seite ist noch nicht vollständig. Es würde ein Deadlock entstehen, wenn zwei tread versuchen, die gleiche bidirektionale Assoziation zu setzen. Dann könnte:
 - ◆ Thread 1 einen Lock auf der Land-Instanz erhalten
 - ◆ Thread 2 einen Lock auf der Stadt-Instanz erhalten
 - ◆ Anschließend würden beide beim Versuch die Rückreferenz zu setzen unendlich warten.

```
synchronized(this) {
    hauptstadt = s;
    if (s.getLand != this)
        s.setLand(this);
}
```

```
synchronized(this) {
    land = l;
    if (l.getHauptstadt != this)
        l.setHauptstadt(this);
}
```

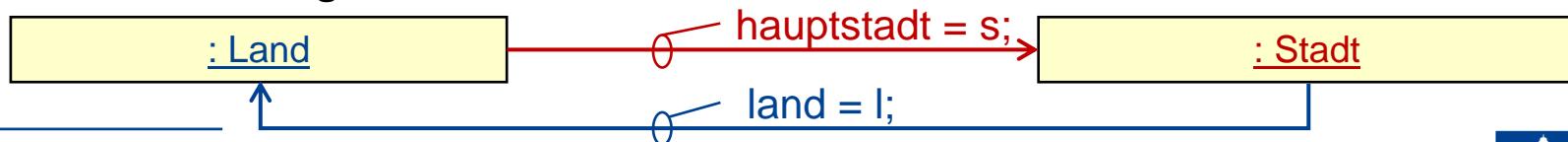
Beispiel-Instanziierung



▶ Bidirektionale Assoziation (1:1)

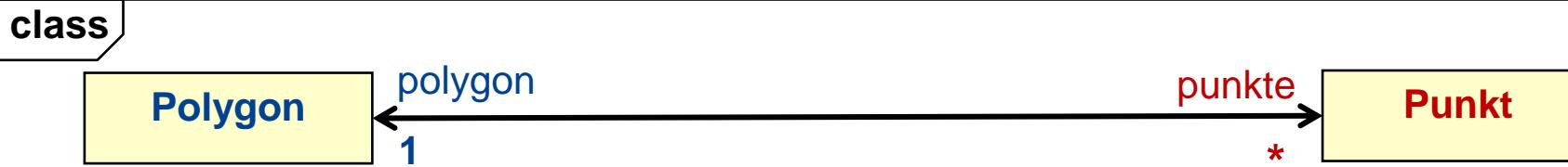
- Lösung für das Deadlockproblem: Verwendung von Lock-Objekten mit Methode `tryLock()` zum Testen, ob ein Lock möglich ist – siehe <https://docs.oracle.com/javase/tutorial/essential/concurrency/newlocks.html>
 - ◆ Selbst ausprobieren!
- Das Schema lässt sich auf 1:n und n:m Assoziationen übertragen
 - ◆ Diskutieren Sie untereinander, was dabei zu beachten ist
 - ◆ Ausprobieren!
 - ◆ Würden Sie das Schema in der Praxis anwenden?
 - ◆ Hier nicht weiter ausgeführt, da wir spätestens für n:m Assoziationen auf eine anderes Implementierungsschema ausweichen werden (ab Seite „Assoziationen sind implizite Klassen“).

Beispiel-Instanziierung

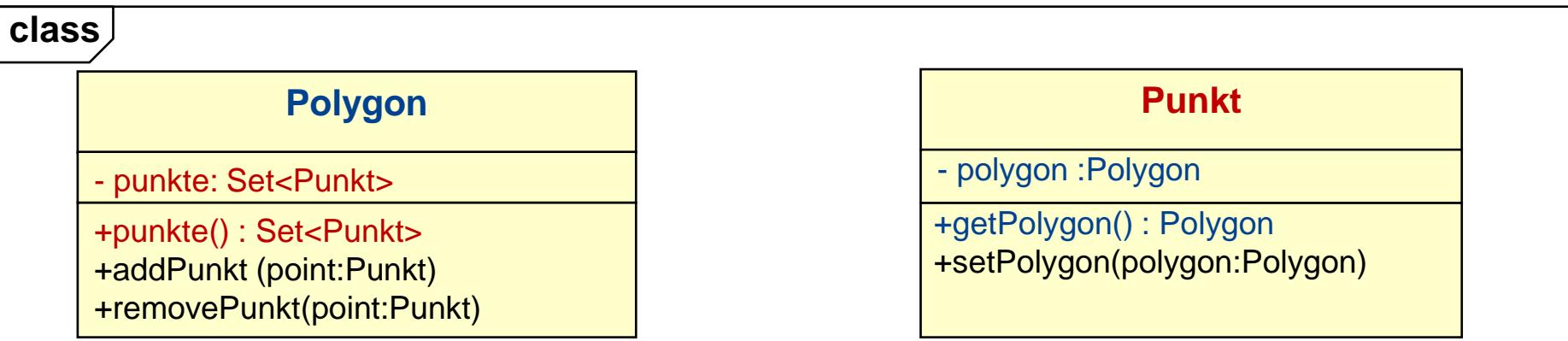


▶ Bidirektionale Assoziation (1:N)

Objektentwurfsmodell mit Assoziation



Objektentwurfsmodell ohne Assoziation



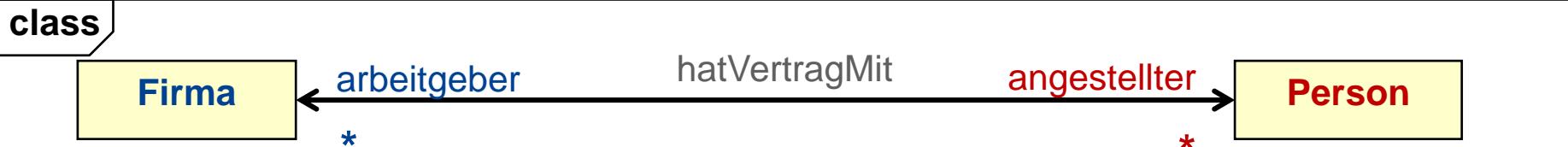
Beispiel-Instanziierung



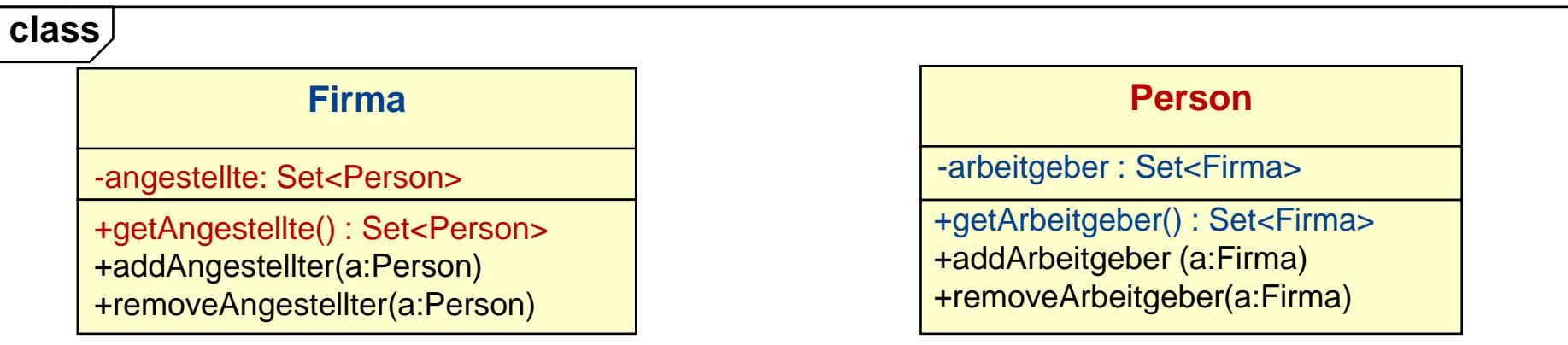
Synchronisation der Zuweisung von Werten für ‚punkte‘ und ‚polygon‘ wie bei bidirektionaler 1:1 Assoziation → analog zu 1:1-Fall

▶ Bidirektionale Assoziation (N:M naiv)

Objektentwurfsmodell vor der Transformation



Objektentwurfsmodell ohne Assoziation



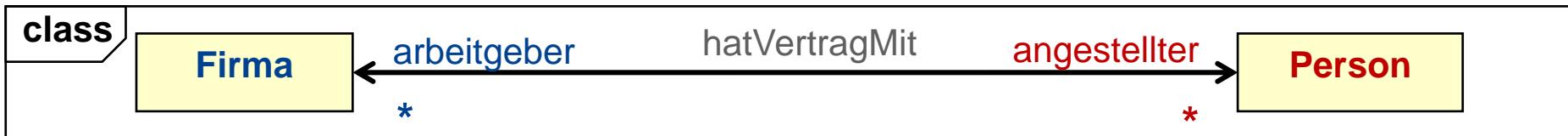
Beispiel-Instanziierung nach der Transformation



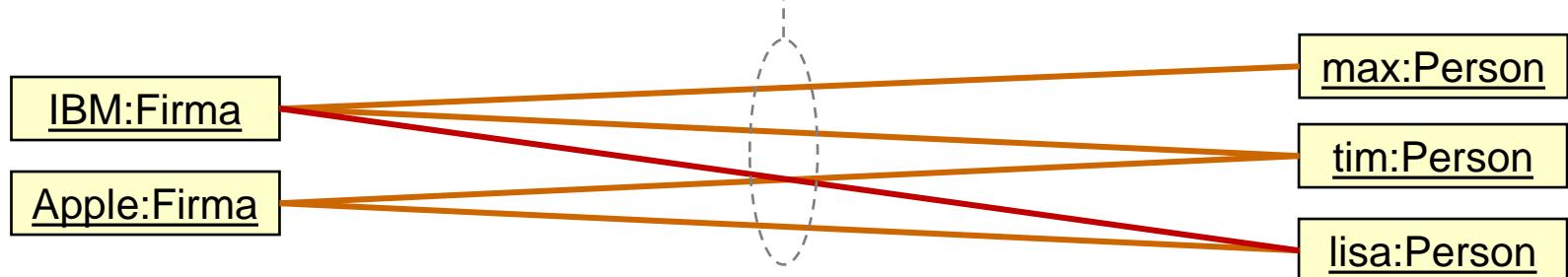
- Problem 1: Deadlockfreies setzen von Rückreferenzen aufwendig/fehleranfällig.
- Problem 2 (aller bisherigen Varianten): Die Beziehung wird fest in den Klassen verdrahtet. Das schafft zusätzliche Abhängigkeiten ☹

Assoziationen sind implizite Klassen!

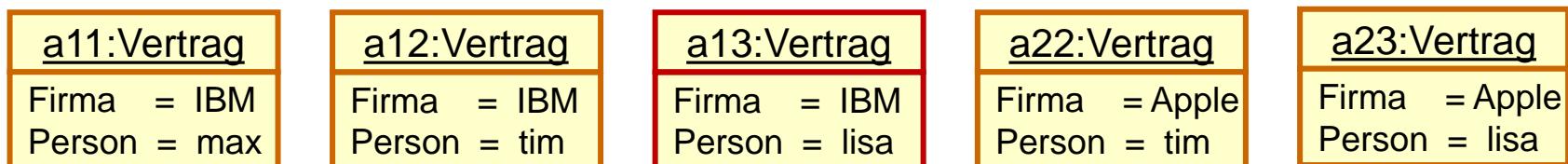
- Eine Assoziation



- Ihre Elemente ...

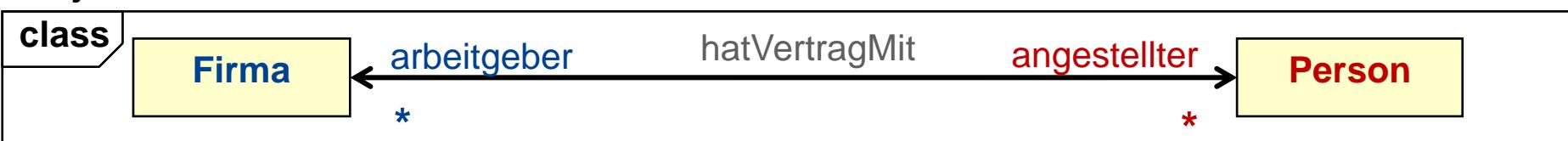


- ... können aufgefasst werden als Instanzen einer Klasse „Vertrag“

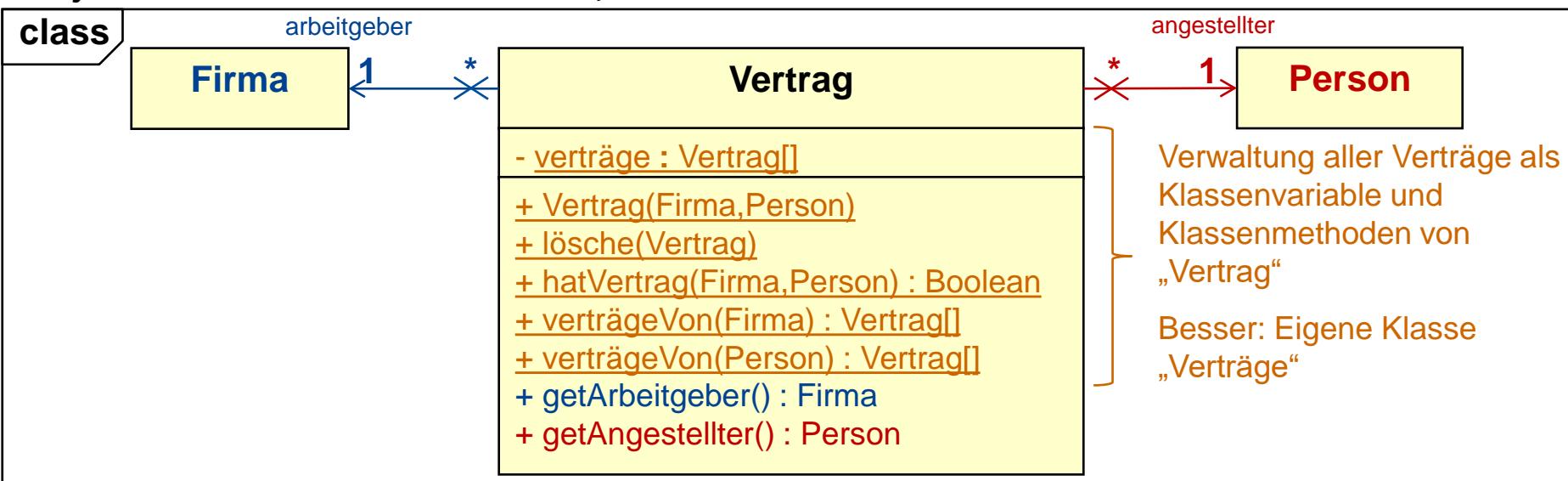


► Bidirektionale Assoziation (N:M als eigene Klasse)

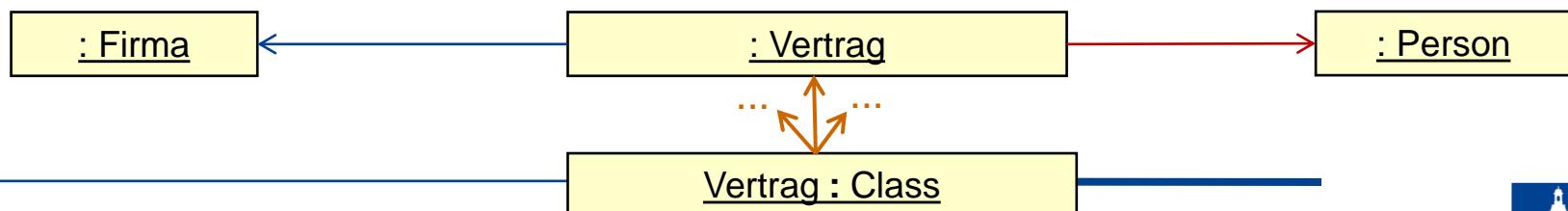
Objektentwurfsmodell mit Assoziation



Objektentwurfsmodell mit Klasse, die die Assoziation modelliert



Beispiel-Instanziierung

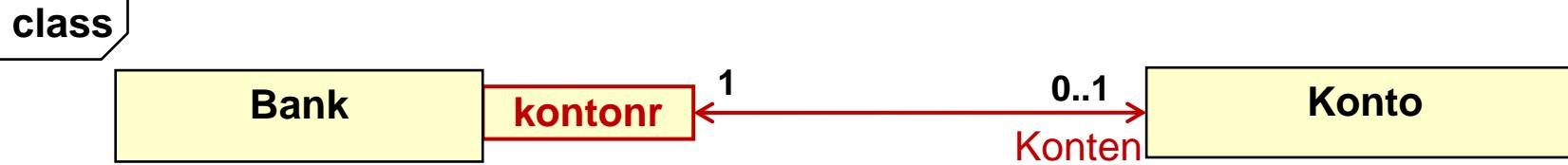


Umsetzung von Assoziationen ▶ Fazit

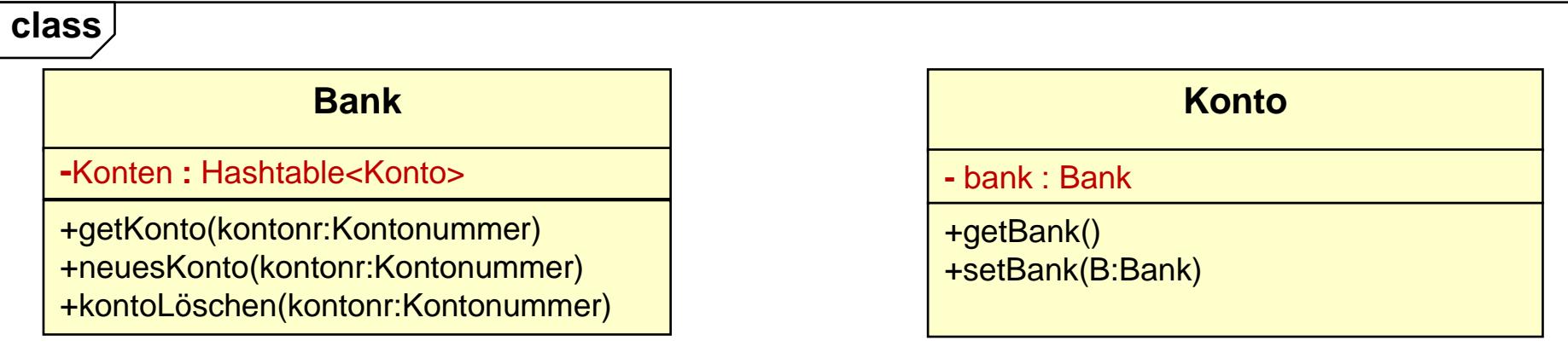
- Assoziationen als Attribute der beteiligten Klassen
 - ◆ Vorteil:
 - ⇒ Einfach bei einseitiger Navigierbarkeit
 - ⇒ Automatisierte Umsetzung von UML-Diagrammen nutzt meist diesen Ansatz
 - ◆ Nachteile:
 - ⇒ Bei beidseitiger Navigierbarkeit ist die terminierende, atomare Setzung von Rückreferenzen kompliziert
 - ⇒ Zusätzliche, zyklische Abhängigkeiten der beteiligten Klassen schaffen ein Wartbarkeitsproblem
 - ⇒ Es gibt keinen sinnvollen Ort, wo beziehungsspezifische Eigenschaften (z.B. Datum der Einstellung) oder Methoden (z.B. einstellen(), kündigen(), beurlauben(), befördern(), ...) modelliert werden könnten
- Beziehungen als Klassen
 - ◆ Keine Rückreferenzen notwendig, da alles in *einem* Objekt geschieht
 - ⇒ kein Rekursionsproblem
 - ⇒ kein Deadlockproblem
 - ◆ Modellierung spezifischer Eigenschaften und Methoden in eigener Klasse

▶ Qualifizierte Assoziation

Objektentwurfsmodell vor der Transformation



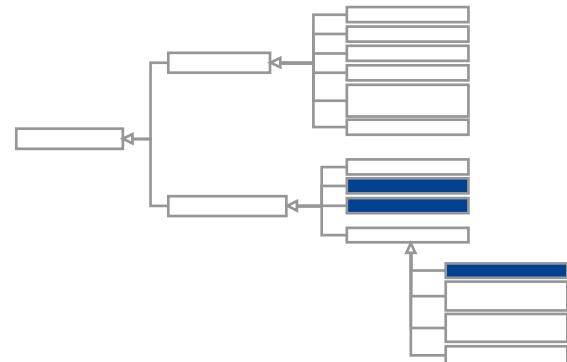
Objektentwurfsmodell nach der Transformation



- Hashtabelle implementiert Indizierung
- Der Qualifier „**kontonr**“ wird zum Schlüssel (“key”) in der Hashtabelle
- Hier bidirektionales Beispiel. Variieren je nach Navigationsrichtung.

3.3 Verhaltensmodellierung

- 3.3.1 Sequenzdiagramme
- 3.3.2 Aktivitätsdiagramme
- 3.3.3 Zustandsdiagramme



3.3.1 Sequenzdiagramme

Einführung

Kontroll- und Datenfluss

Interaktionsfragmente

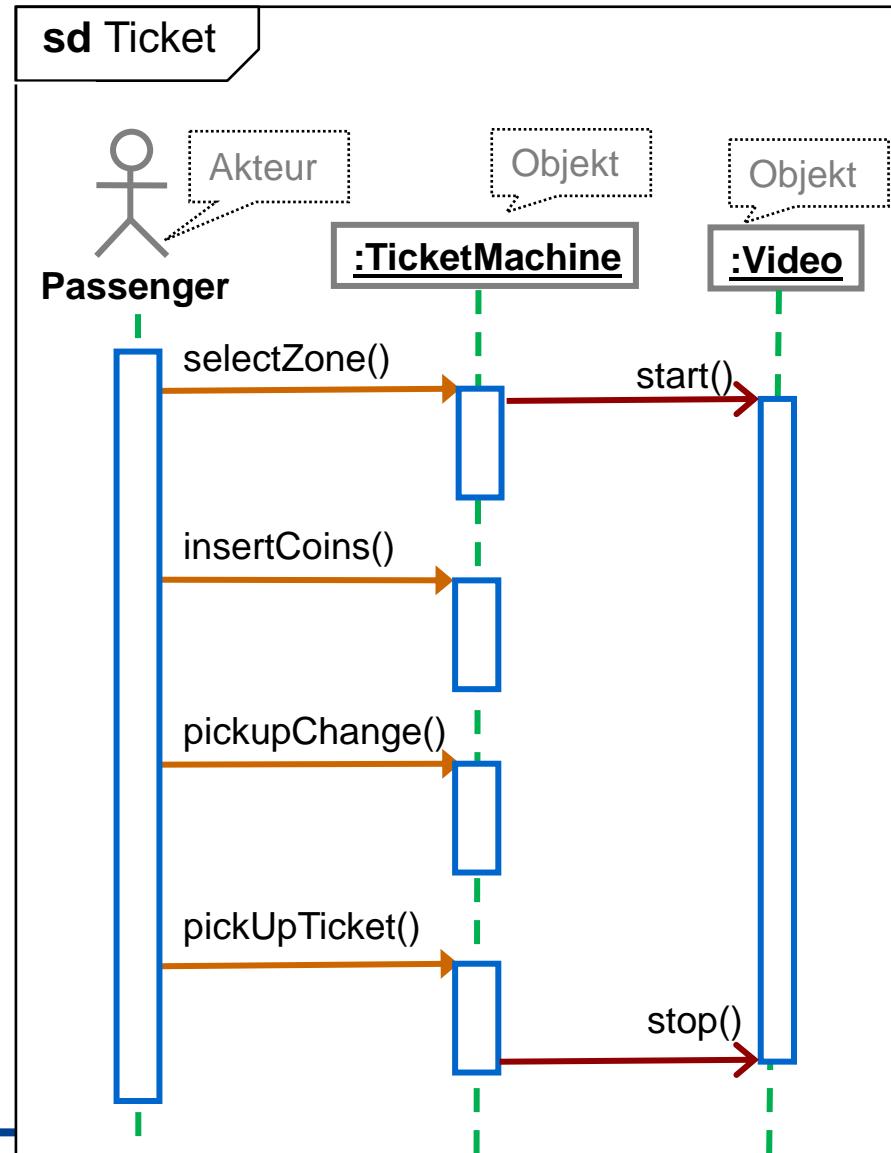
Gute Zusammenfassung → „[UML@Work](#)“, Seite 251 – 287,

Alles im Detail → UML Spezifikation 2.5.1 (5.12.2017) <https://www.omg.org/spec/UML>

– Kap 17 (Seite 565-638), insbesondere Kap 17.8

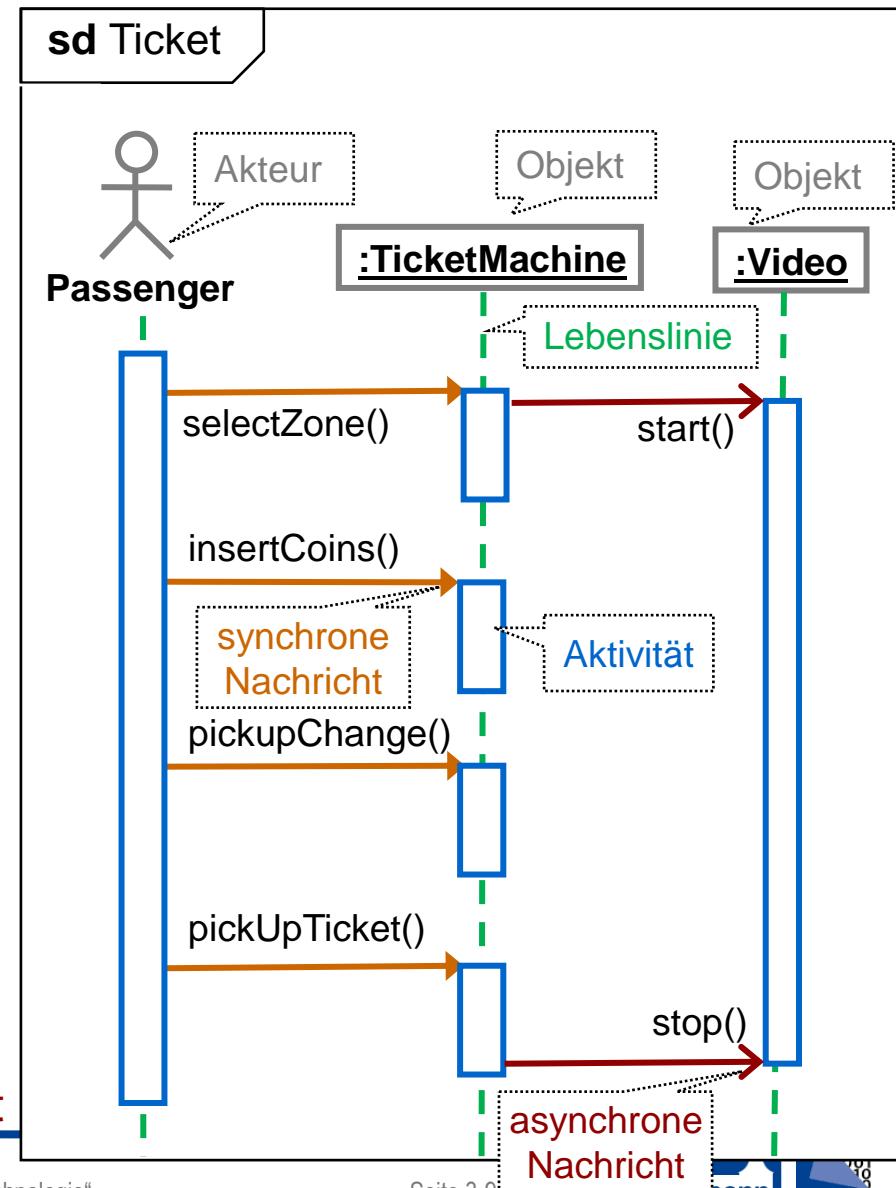
Sequenzdiagramm („sequence diagram“)

- Visualisiert Nachrichten entlang einer vertikalen Zeitachse
- Modelliert Interaktion zwischen
 - ◆ Akteuren und Objekten
 - ◆ verschiedenen Objekten
 - ◆ einem Objekt mit sich selbst
- Bietet Darstellung von
 - ◆ Nachrichten
 - ◆ Nebenläufigkeit
 - ◆ Datenfluss
 - ◆ Zeitpunkten und –dauer
 - ◆ Verzweigungen / Schleifen
 - ◆ Filterungen / Zusicherungen



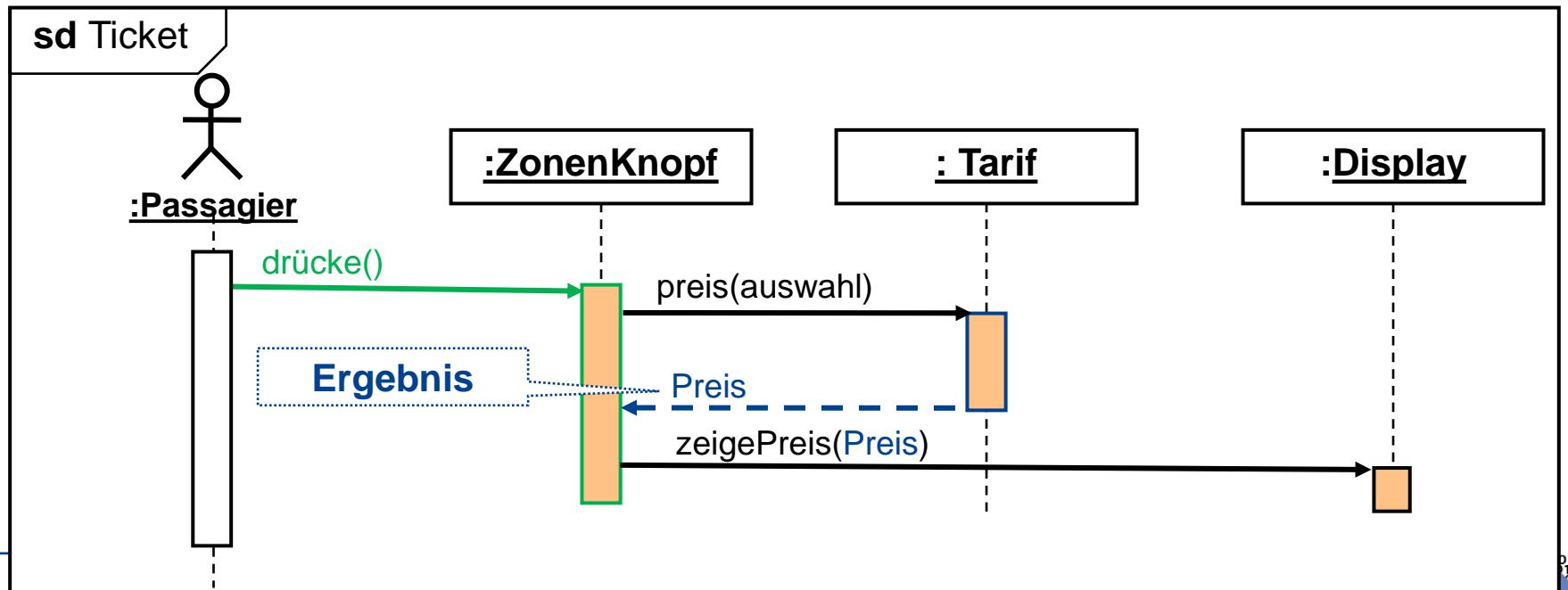
Sequenzdiagramm („sequence diagram“)

- Akteur
 - ◆ Externer „Benutzer“
 - Mensch
 - Software
 - Hardware
- Objekt
 - ◆ siehe Objektdiagramm-Notation
- Lebenslinie
 - ◆ Zeitraum in dem das Objekt existiert
 - ◆ Später: Explizite Notation für Objekt-“Geburt“ und -“Tod“
- Aktivität / Aktivierung
 - ◆ Zeitraum in dem das Objekt etwas tut
 - ◆ Ausgelöst durch **synchrone Nachricht** bei passiven Objekten
 - ◆ Thread bei aktiven Objekten – Reaktion auf **asynchrone Nachricht**



Sequenzdiagramme: Kontroll- und Datenfluss

- Jeder Pfeil beginnt an der Aktivierung, welche die Nachricht gesendet hat
- Die Aktivierung an der Pfeil endet beginnt direkt am Pfeilkopf
- Eine Aktivierung dauert (mindestens) so lange wie alle geschachtelten Aktivierungen
- Für synchrone Nachrichten erfolgen Rückgaben implizit am Ende einer Aktivierung
- Pfeile für Rückgaben werden nur benutzt, um den Datenfluss explizit zu machen

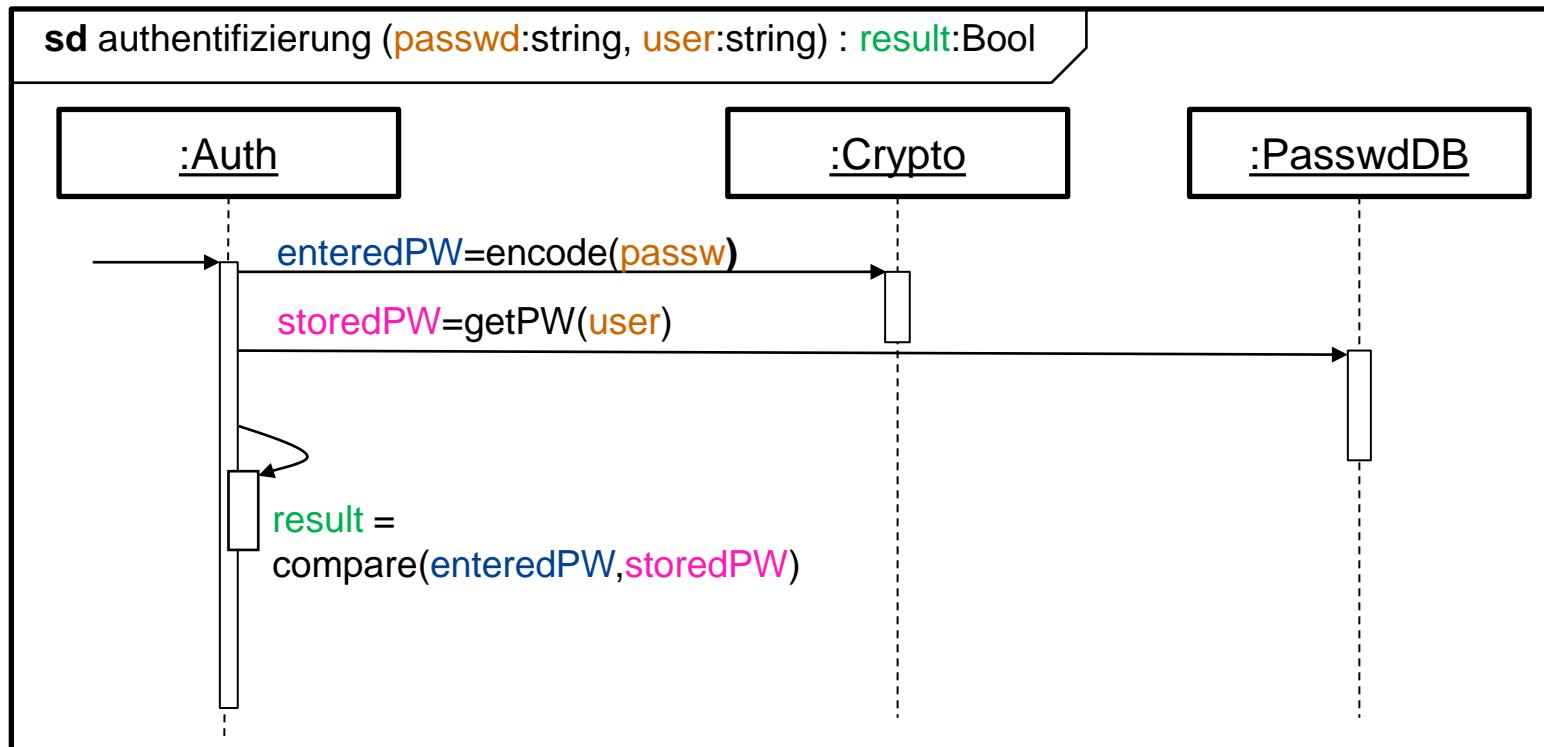


Sequenzdiagramm: Varianten von Nachrichten-Beschriftungen

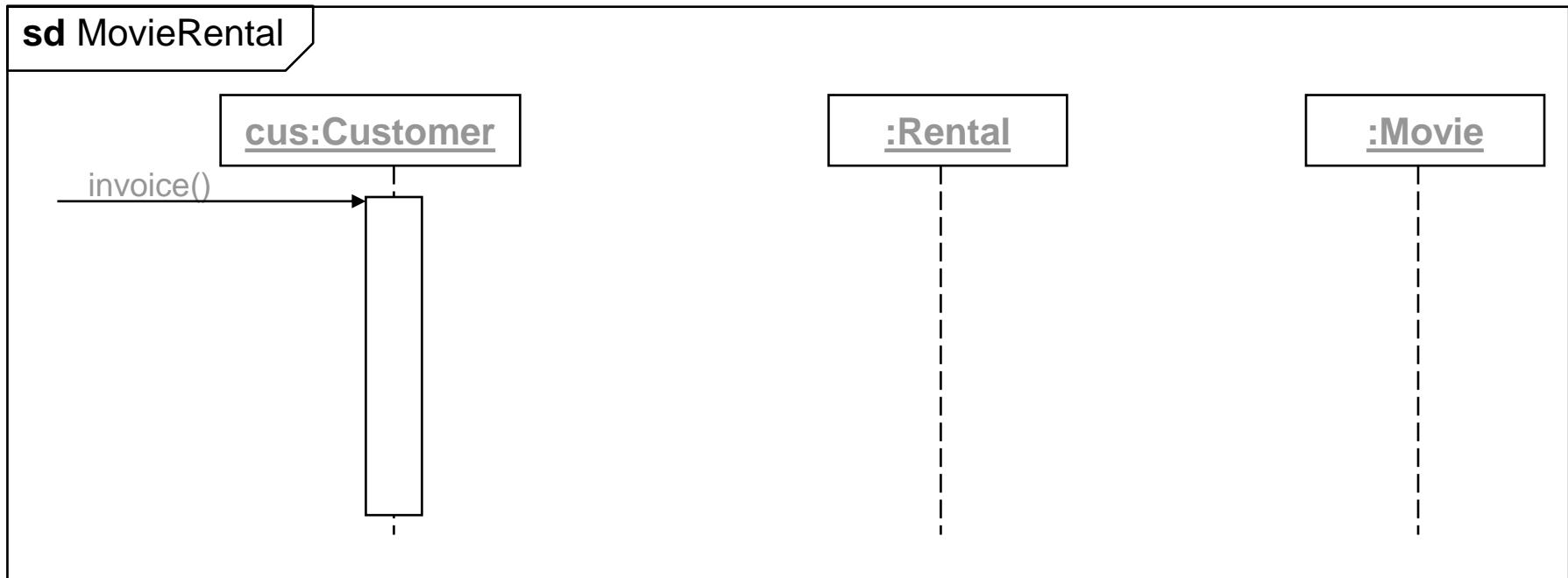
- $f(x:T1, y:T2):T$ volle Signatur
 - $f()$ Parameter ignoriert
 - $f(args)$ Parameter explizit
 - $x = f(args)$ Parameter & lokale Variable für Ergebnis
 - $x = f(args):2$ Parameter & lokale Variable für Ergebnis & Ergebniswert
 - ◆ Ergebniswert kann Konstante sein („2“)
 - ◆ ... oder der Name eines Objekts im Diagramm (z.B. „o1“ wenn es $o1:T$ im Diagramm gibt oder wenn $o1:T$ ein Parameter des Diagramms ist).
- So kann man den Datenfluss explizit modellieren

Sequenzdiagramm: Datenfluss explizit machen ► Beispiel

- Autentifizierung eines **user** anhand eines **passwd** erfolgt indem
 - das Passwort verschüsselt wird
 - das gespeicherte verschlüsselte Passwort ausgelesen wird und dann
 - verglichen wird, ob beide übereinstimmen.

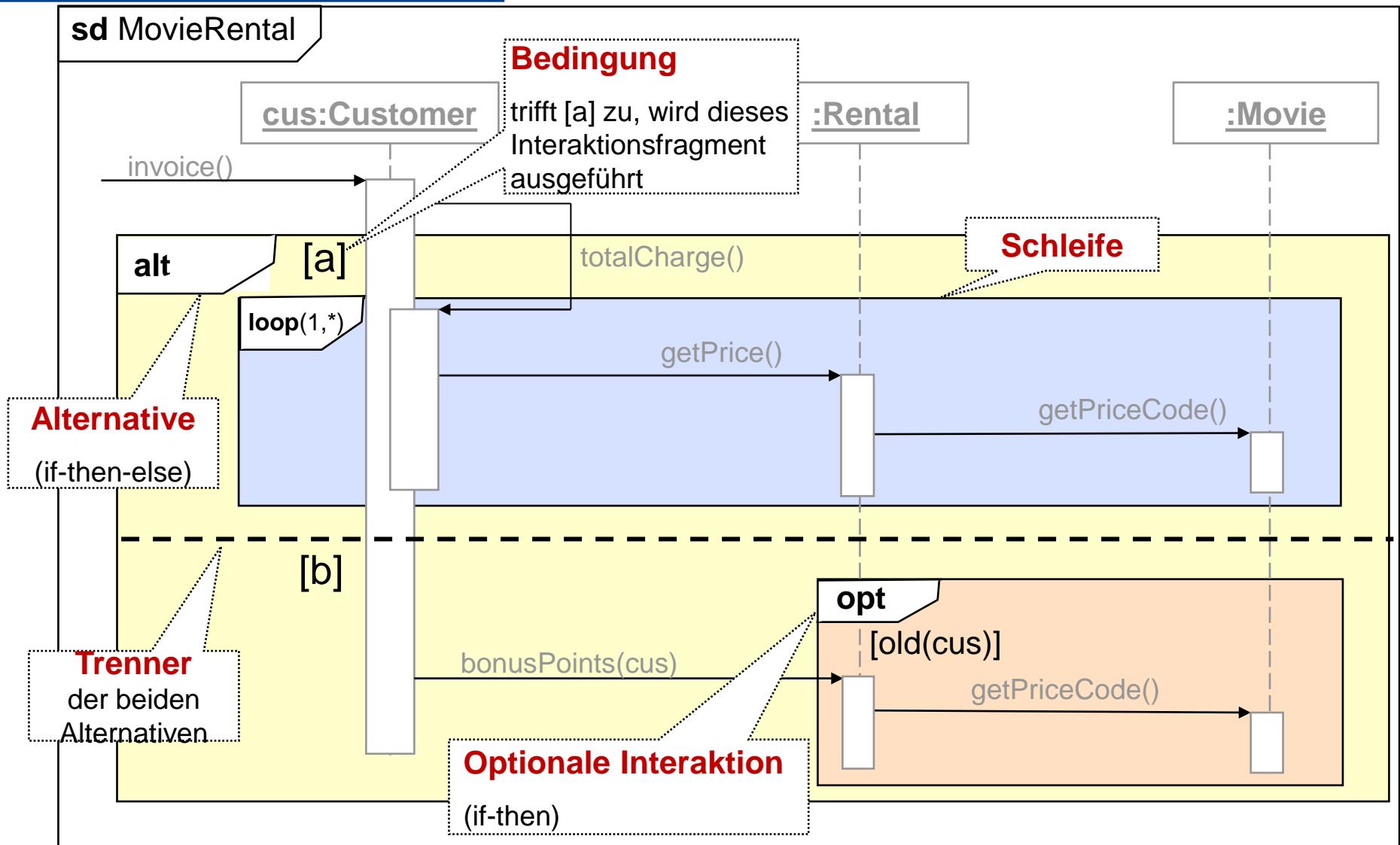


Sequenzdiagramme: Kontrollstrukturen in UML 2 → „Kombinierte Fragmente“



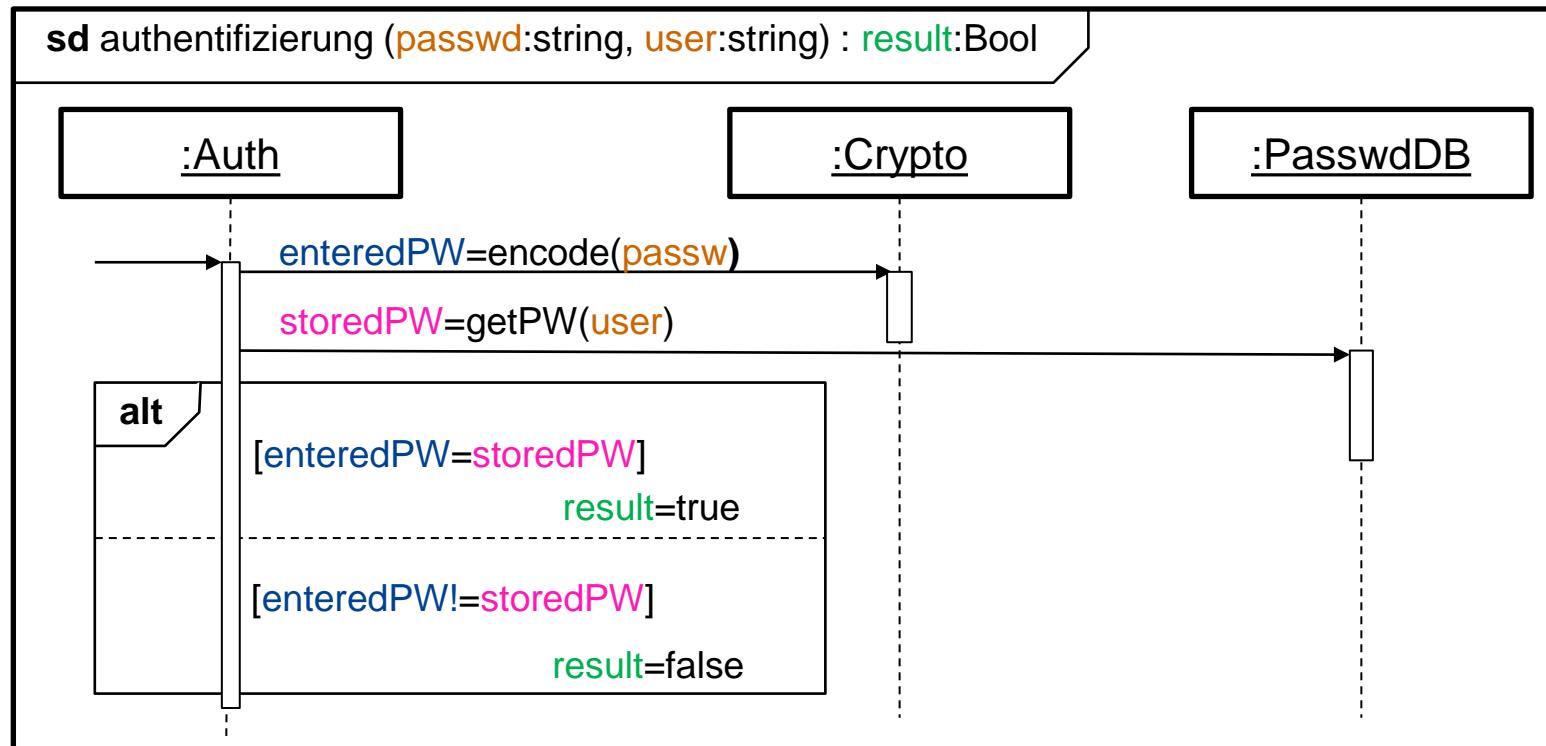
- Szenario: Ausdrucken einer Rechnung für Kunden eines Videoverleihs
 - ◆ Alternative: Falls *a* Gesamtkosten drucken, falls *b* Bonuspunkte drucken
 - ◆ Schleife: Gesamtkosten sind kosten aller Ausleihvorgänge (:Rental)
 - ◆ Optionale Aktionen: Für Altkunden Bonuspunkte anders berechnen

Sequenzdiagramme: Kontrollstrukturen in UML 2 → „Interaktionsfragmente“



Folie 94 mit „alt“-Interaktionsfragment statt compare()-Methodenaufruf

- Autentifizierung eines **user** anhand eines **passwd** erfolgt indem
 - das Passwort verschüsselt wird
 - das gespeicherte verschlüsselte Passwort ausgelesen wird und dann
 - verglichen wird, ob beide übereinstimmen.



Sequenzdiagramme: Interaktionsfragmente

- Komplexere Kontrollstrukturen werden durch Operatoren auf Interaktionsfragmenten (Ausschnitte des Sequenzdiagramms) realisiert

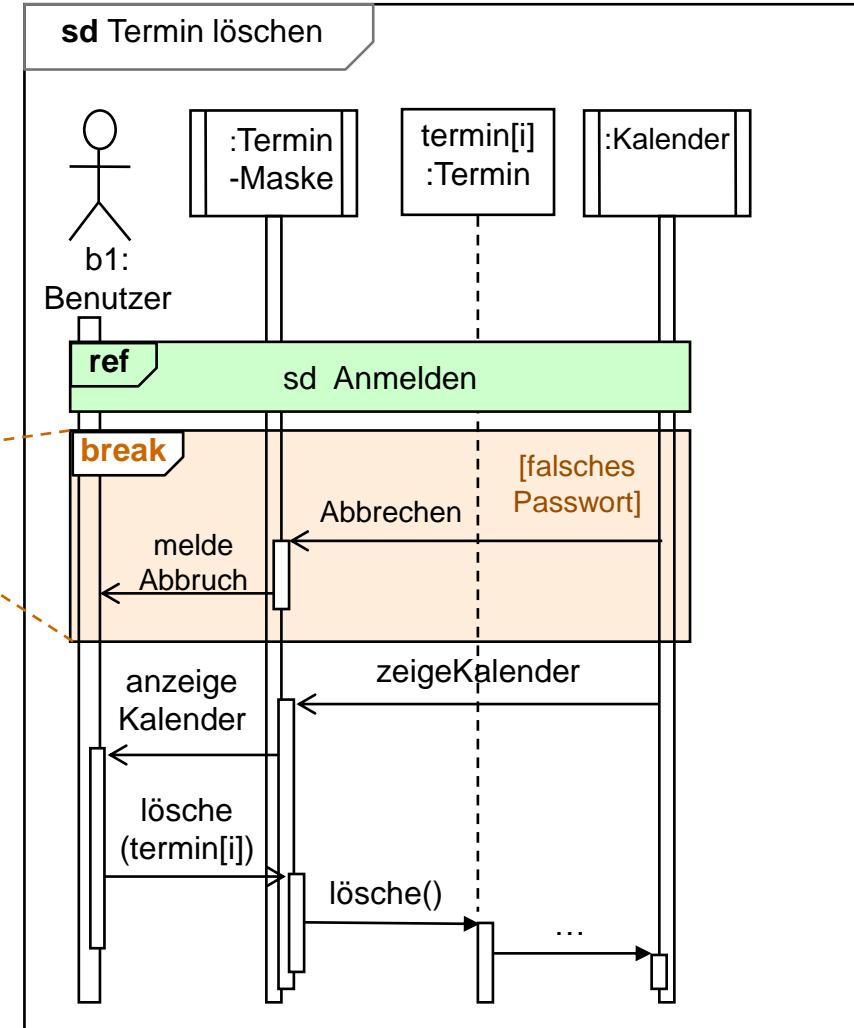
	Operator	Zweck
Verzweigungen und Schleifen	alt	Alternative Interaktionen – if-then-else
	opt	Optionale Interaktionen – if-then
	loop	Iterative Interaktionen
	break	Umgebendes Interaktionsfragment verlassen (Verallgemeinerung von "break" in Java)
Nebenläufigkeit und Ordnung	seq	Sequentielle Interaktionen mit schwacher Ordnung
	strict	Sequentielle Interaktionen mit strenger Ordnung
	par	Nebenläufige Interaktionen
	critical	Atomare Interaktionen
Filterung und Zusicherung	ignore	Irrelevante Interaktionen
	consider	Relevante Interaktionen
	assert	Zugesicherte Interaktionen
	neg	Ungültige Interaktionen

Wir beschränken uns in der Vorlesung auf die grün hervorgehobenen Operationen

Sequenzdiagramm: Kalender-Beispiel

→ Verwendung von "break"

- Gemeinsamer Kalender
 - ◆ à la Google-Kalender
- Hier: „Löschen eines Termins“
 1. Anmelden ...
 2. Abbruch falls Anmeldefehler
 3. Terminanzeige ...
 4. Selektion des Termins
 5. ... und schließlich Löschen
- Break
 - ◆ falls die Bedingung zutrifft wird das *unmittelbar* umgebende Interaktionsfragment verlassen



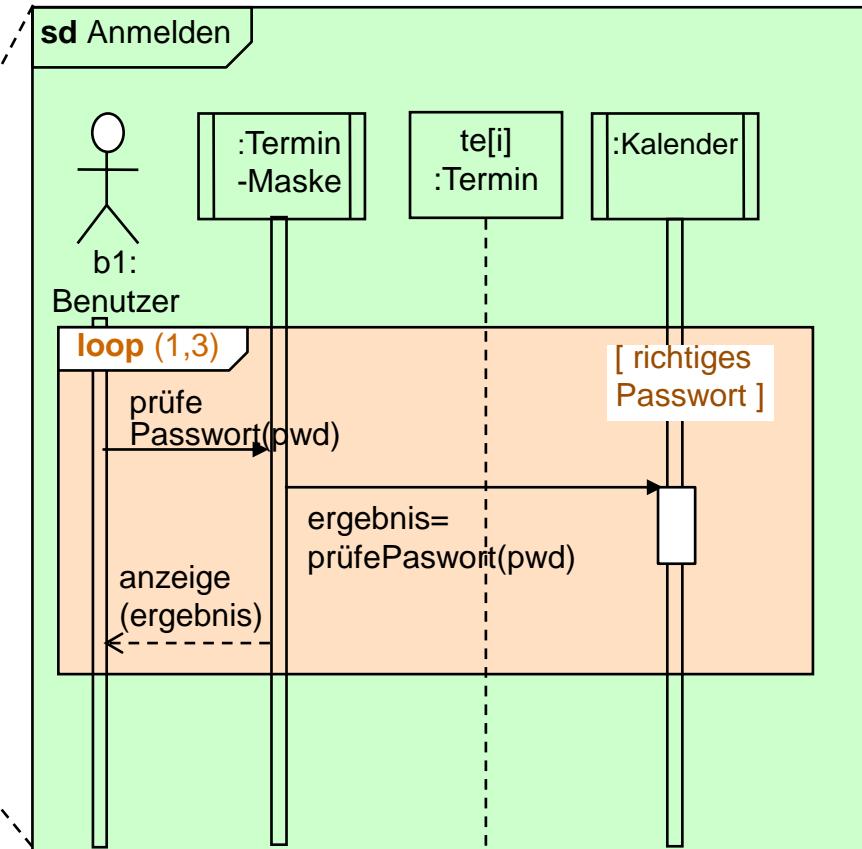
Sequenzdiagramm: Kalender-Beispiel

→ Verwendung von "loop"

- Was versteckt sich hinter der Referenz?
 - Ein beliebiges dynamisches Diagramm
 - Hier: Der Anmeldevorgang

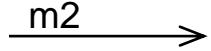
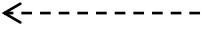
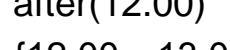


- loop(1,3) [cond]
 - Kombination von for- und do-until-Schleife
 - Wiederhole Fragment mindestens ein und maximal drei mal, aber höre auf sobald „cond“ erfüllt ist,
⇒ d.h. for (i=1..3) {...; if (cond) break;}



Sequenzdiagramme (wichtige Elemente)

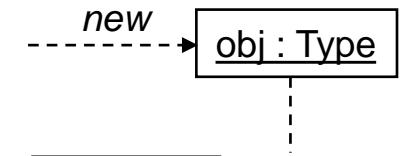
UML 2.0

- Interaktionspartner
 - ◆ passives Objekt (aktiv nur als Reaktion auf Nachrichten)
 - ◆ aktives Objekt (Prozess/Thread)
- Nachricht
 - ◆ synchron (Aufrufender wartet auf Ende der Aktivierung)
 - ◆ asynchron (Aufrufender sendet Nachricht und macht weiter)
- Antwortnachricht (bei asynchronen Nachrichten)
 - ◆ ohne Ergebnis
 - ◆ mit Rückgabewert
- Zeiteinschränkung an Nachricht
 - ◆ Absoluter Zeitpunkt
 - ◆ Relativer Zeitpunkt
 - ◆ Intervall

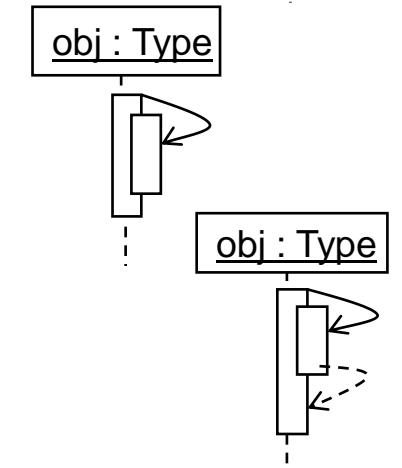
Sequenzdiagramme (wichtige Elemente)

UML 2.0

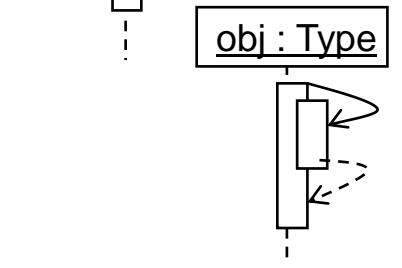
- Objekt wird durch Nachricht erzeugt



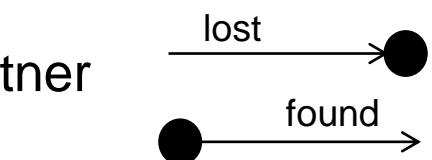
- Nachricht an sich selbst



- Rekursiver Aufruf

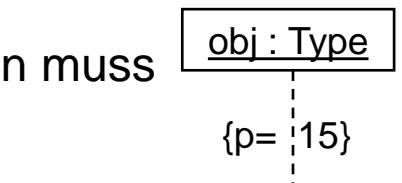


- Nachrichten ohne explizit modellierte Interaktionspartner



- Zustandsinvariante

- ◆ Bedingung, die zu einem gewissen Zeitpunkt erfüllt sein muss



Sequenzdiagramme: Fazit

- Sequenzdiagramme...
 - ◆ ...repräsentieren Verhalten bezüglich Interaktionen.
 - ◆ ...ergänzen die Klassendiagramme, welche die Struktur repräsentieren.
 - ◆ ...sind nützlich, um
 - ⇒ Verhalten präzise zu beschreiben
 - ⇒ das Verhalten eines Anwendungsfalls auf die beteiligten Objekte abzubilden
 - ⇒ beteiligte Objekte zu finden
- Der Zusatzaufwand lohnt sich auf jeden Fall bei komplexen und unklaren Abläufen
 - ◆ Aber: keine Zeit vergeuden, um Trivialitäten und Offensichtliches mit Sequenzdiagrammen zu modellieren

3.3.2 Aktivitätsdiagramme

Einführung

Tokenbasierte Semantik

Notationsübersicht

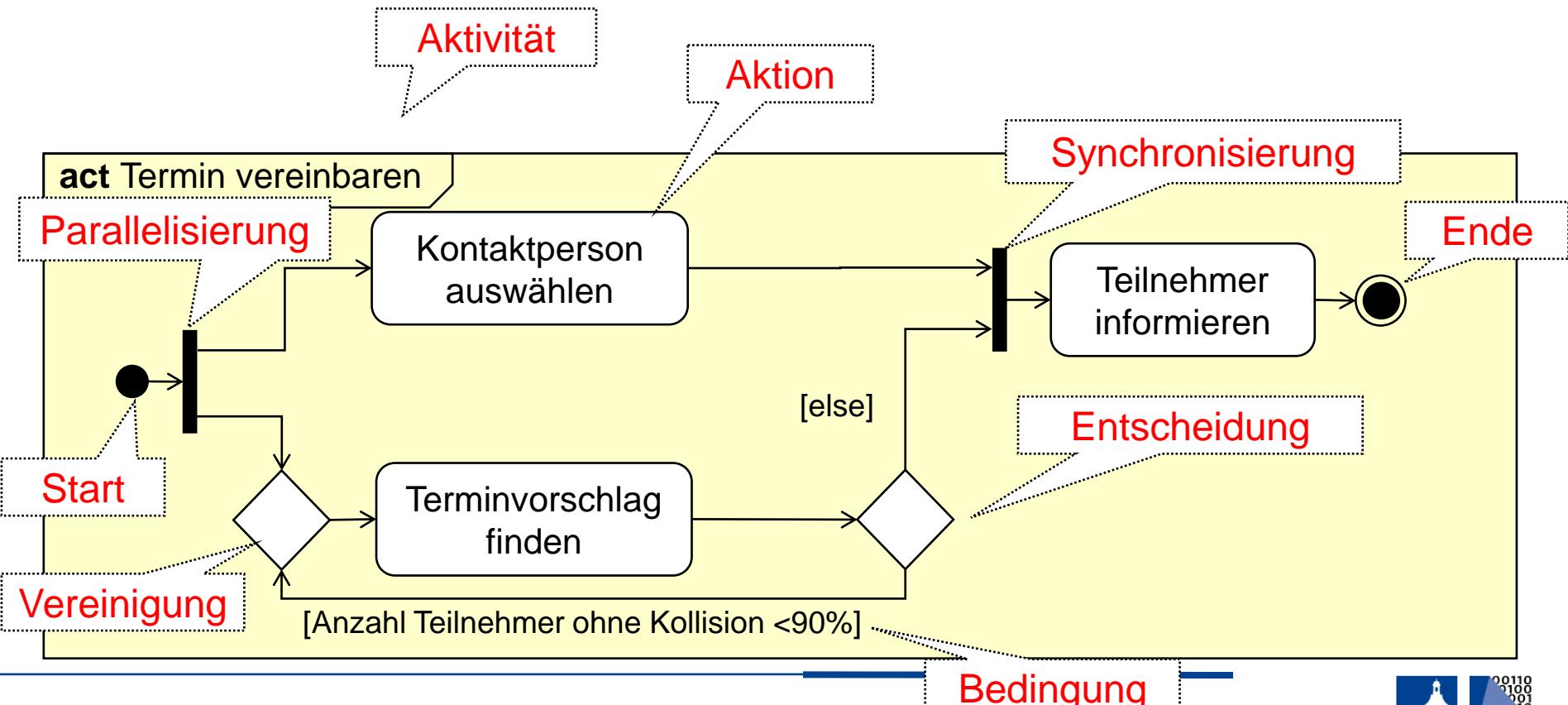
Datenfluss

Swimlanes

Alles im Detail → UML Spezifikation 2.5.1 (5.12.2017) <https://www.omg.org/spec/UML>
– Kap 15 (Seite 373 -439)

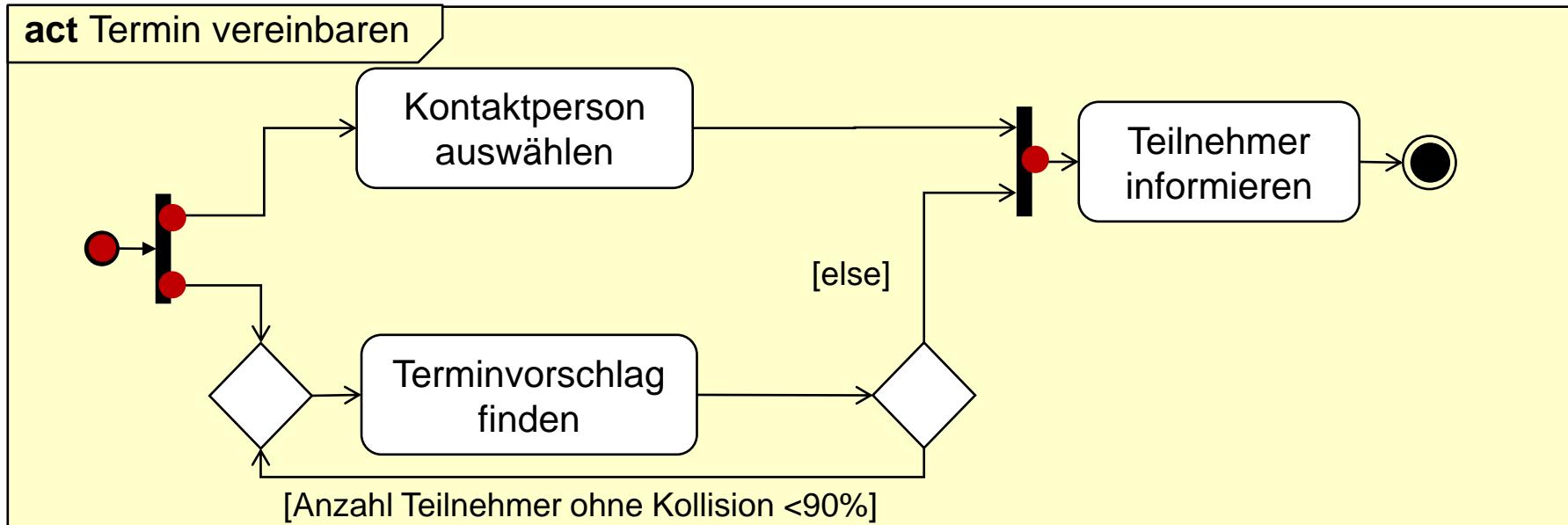
Aktivitätsdiagramme

- Eine **Aktion** ist ein elementarer Arbeitsschritt
 - ◆ atomar (d.h. Effekte werden bei Unterbrechung rückgängig gemacht)
- Eine **Aktivität** spezifiziert den Kontroll- und Datenfluss zwischen Aktionen



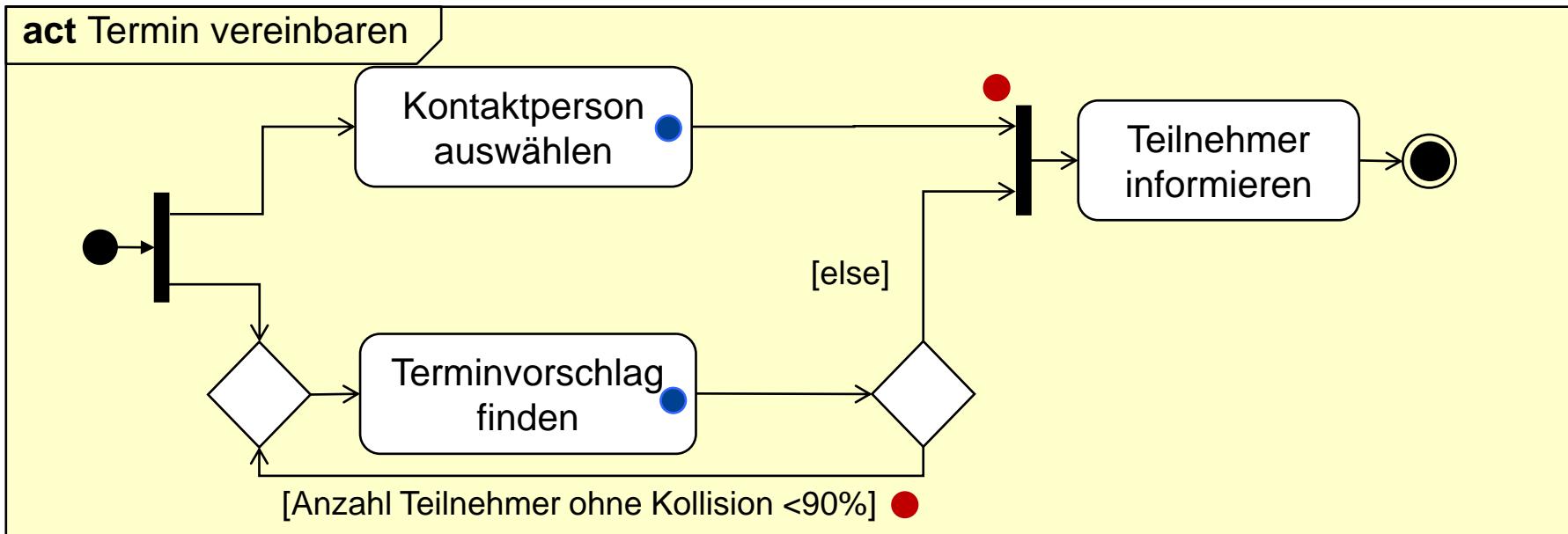
Aktivitätsdiagramme: Tokenbasierte Semantik

- Operationales Modell: Token, die Abläufen entsprechen, fließen durch das Diagramm ab dem Startknoten.
 - Parallelisierung: Auf jeder Ausgangskante fließt je ein Token weiter.
 - Synchronisation: Auf jeder Eingangskante muss ein Token ankommen.
 - Vereinigung: Jedes ankommende Token wird sofort weitergeleitet.
 - Entscheidung: Token fließt nur auf der Ausgangskante weiter, deren Bedingung wahr ist.



Aktivitätsdiagramme: Tokenbasierte Semantik

- Mehrfache gleichzeitige Abläufe (mehrfache „Threads“)
 - ◆ Ein Aktivitätsdiagramm kann **gleichzeitig mehrfach durchlaufen** werden
- Visualisierung
 - ◆ Jeder Start im Anfangszustand bringt Tokens mit neuer Farbe hervor
 - ◆ Tokens mit gleicher Farbe gehören zum gleichen anfänglichen Ablauf
- Beispiel: Der **rote Ablauf** ist schon weiter fortgeschritten als der **Blau**



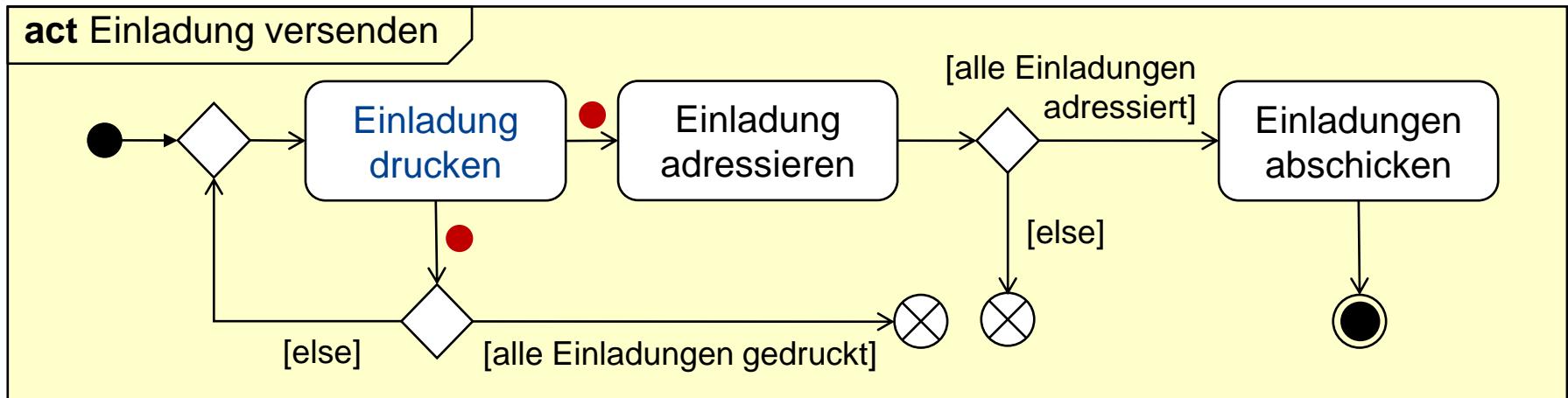
Endzustand versus Ablaufende

- Endzustand ●

- ◆ Ende aller Abläufe, die gerade das Diagramm durchlaufen
 - ⇒ Wenn ein Token den Endzustand erreicht, werden alle anderen Token der gleichen Farbe mit beendet

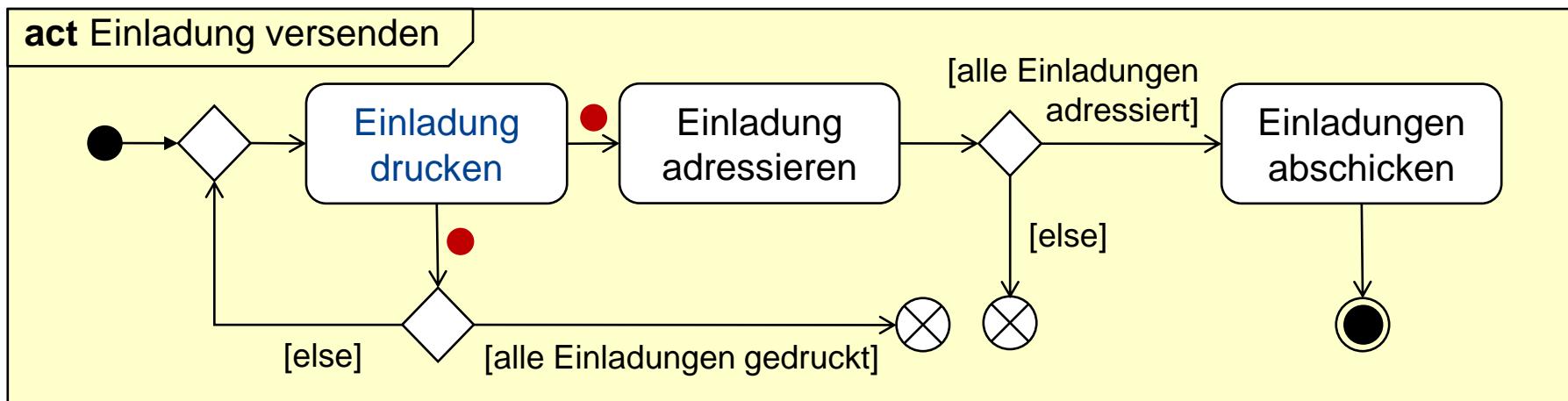
- Ablaufende ○

- ◆ Ende eines Ablaufs (andere können weiterlaufen)
 - ⇒ Nur das Token das das Ablaufende erreicht wird beendet



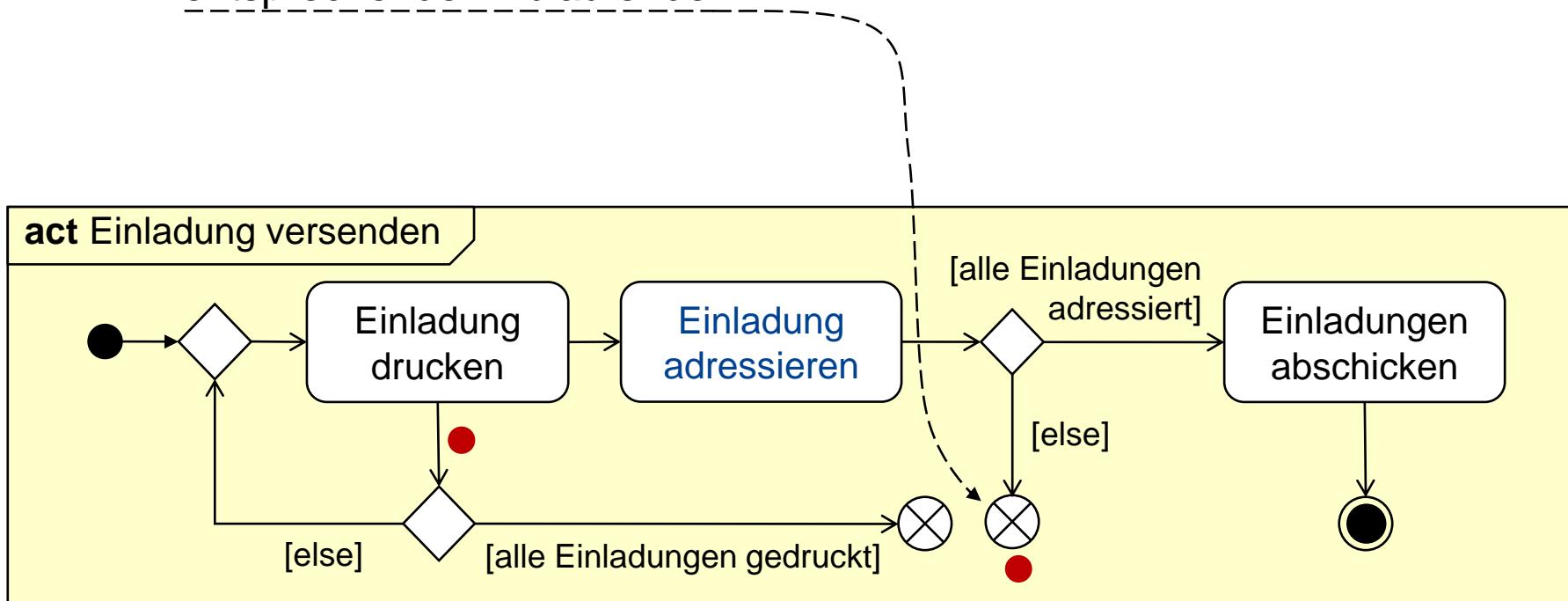
Endzustand versus Ablaufende: Beispiel

- Drucken, Adressieren und Versenden von Einladungen
 - Wenn eine **Einladung gedruckt** ist, fließen 2 Token weiter, eines auf jeder Ausgangskante (implizite Parallelisierung)
 - Die gedruckte Einladung wird daher als Nächstes adressiert und es wird gleichzeitig die nächste Einladung gedruckt



Endzustand versus Ablaufende: Beispiel

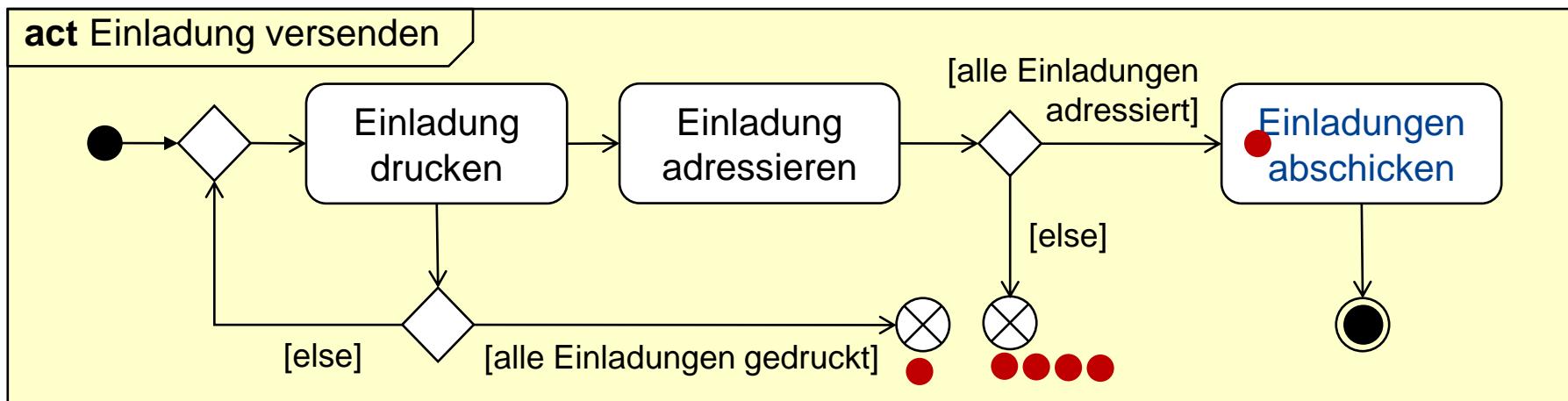
- Drucken, Adressieren und Versenden von Einladungen
 - Wenn eine **Einladung adressiert** ist und es nicht die letzte war, ist der entsprechende Ablauf beendet. Das Token verschwindet im entsprechenden Ablaufende



Endzustand versus Ablaufende: Beispiel

- Drucken, Adressieren und Versenden von Einladungen
 4. Wenn die letzte Einladung **gedruckt** ist verschwindet das „Drucken“-Token im entsprechenden Ablaufende
 5. Wenn die letzte Einladung **adressiert** ist, werden alle (gedruckten und adressierten) Einladungen gemeinsam **verschickt**

Hier die Momentaufnahme beim Verschicken von 5 Einladungen:



6. Die gesamte Aktivität ist danach beendet (Token erreicht Endzustand)

Aktivitätsdiagramme

- Ablauorientierte Notation
 - ◆ basierend auf Petri-Netzen und BPEL (Business Process Engineering Language)
- Auch für nicht objekt-orientierte Systeme
 - ◆ Aktivitäten sind unabhängig von Objekten
 - ◆ Anwendbar auch auf Geschäftsprozesse, Funktionsbibliotheken, ...
- Elementare Konzepte
 - ◆ Aktivität = benanntes Verhalten
 - ◆ Aktion = atomare Aktivität (einzelner Arbeitsschritt)
 - ⇒ „Atomar“ = kann unterbrochen werden, macht dann aber jegliche Effekte rückgängig

Aktivitätsdiagramme: Aktivitäten

- Aktivitätsdiagramm
 - ◆ Graph bestehend aus Aktivitätsknoten und Aktivitätskanten
- Aktivitätsknoten
 - ◆ Aktionsknoten: Elementare, atomare, meist vordefinierte Aktionen
 - ◆ Kontrollknoten: Alternativen, Parallelisierung, Synchronisation
 - ◆ Datenknoten: Parameter, Puffer, Datenbanken
- Aktivitätskanten
 - ◆ Kontrollflusskanten: Verbindungen von Aktions- und Kontrollknoten
 - ◆ Datenflusskanten: Verbindungen von Datenknoten

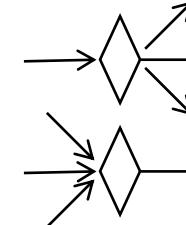
Aktivitätsdiagramme: die wichtigsten Elemente

- Aktion
- Start- und Endknoten
 - ◆ Start
 - ◆ Aktivitätsende (Ende aller Abläufe)
 - ◆ Ablaufende (Ende eines bestimmten Ablaufs)
- Alternative Abläufe
 - ◆ Entscheidungsknoten
 - ◆ Vereinigungsknoten
- Nebenläufige Abläufe
 - ◆ Parallelisierungsknoten
 - ◆ Synchronisierungsknoten

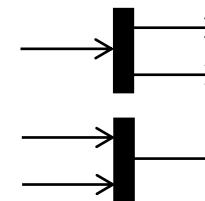
Name der Aktion



3 von 44
vordefinierten
Aktionsknoten



Kontrollknoten



Aktivitätsdiagramme: die wichtigsten Elemente

- Datenknoten

- ◆ Parameter von Aktivitäten
- ◆ Parameter von Aktionen (Pins)
- ◆ Temporäre Puffer
- ◆ Persistente Puffer



Aktivitätsparameter überlappen den Rand!

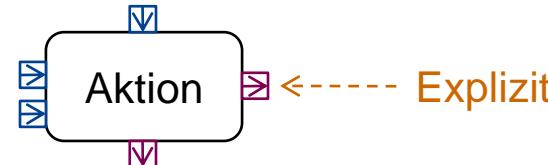


Aktionsparameter (Pins) liegen außen!



- Eingabeparameter

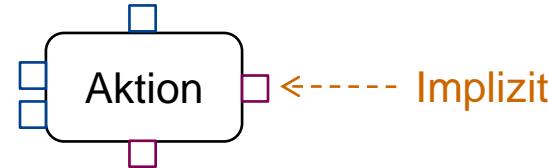
- ◆ Explizit: Pfeil im Datenknoten hin zur Aktivität / Aktion
- ◆ Implizit: Parameter links oder oben



Explizit

- Ausgabeparameter

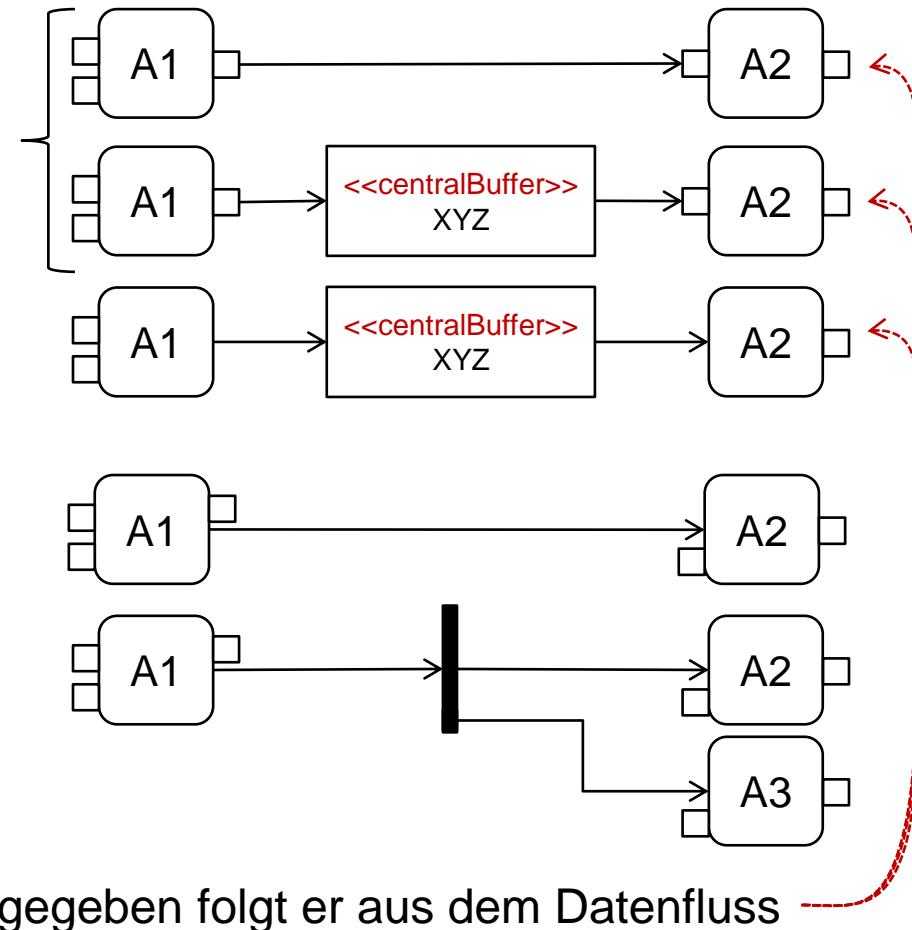
- ◆ Explizit: ... weg von der ...
- ◆ Implizit: ... rechts oder unten ...



Implizit

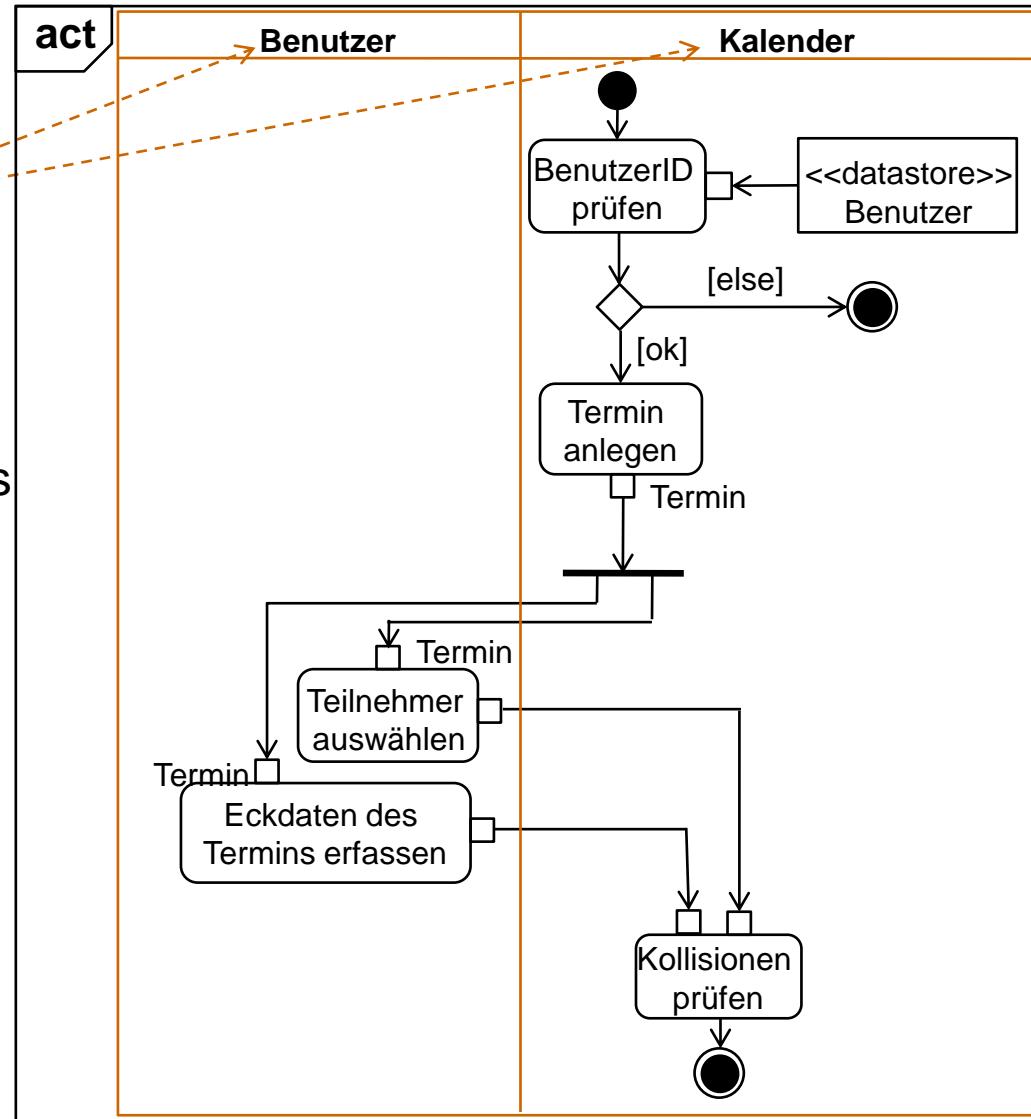
Aktivitätsdiagramme: die wichtigsten Elemente

- Datenfluss
 - ◆ Durchgezogene spitze Pfeile zwischen **Datenknoten**
 - ◆ ... oder direkt zwischen **Aktionen / Aktivitäten und Kontrollknoten**
- Kontrollfluss
 - ◆ Durchgezogene spitze Pfeile zwischen **Aktionen / Aktivitäten und Kontrollknoten**
 - ◆ Wenn Kontrollfluss nicht explizit angegeben folgt er aus dem Datenfluss



Aktivitätsdiagramm: Partitionen

- Notation
 - Benannte „Schwimmbahnen“ / „Swimlanes“
- Semantik
 - Gruppierung von Teilen eines Aktivitätsdiagramms
 - Gruppierungskriterium ist beliebig
 - Im Beispiel: Zugehörigkeit der Aktionen zu gewissen Klassen



3.3.3 Zustandsdiagramme

Einführung
Unterautomaten
Entry- und Exit
Innere Transitionen
History States
Event-Condition-Action Semantik von Transitionen
Notationsüberblick

Alles im Detail → UML Spezifikation 2.5.1 (5.12.2017) <https://www.omg.org/spec/UML>
– Kap 14 (Seite 305 - 371)

Beispiel: „Bestellungsbearbeitung“

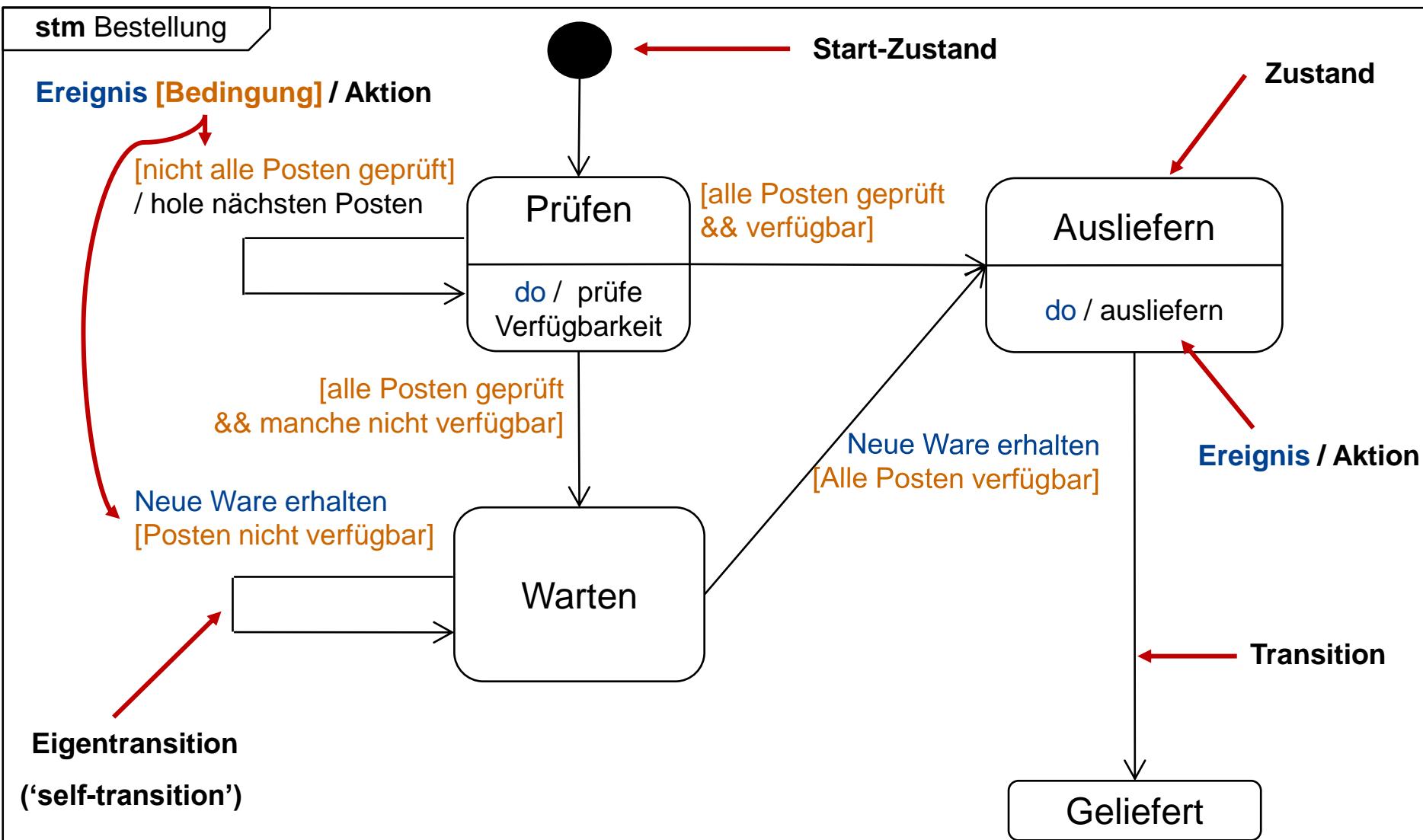
Szenario

- Eine Bestellung wird abgearbeitet, in dem erst zu jedem bestellten Artikel überprüft wird ob er vorrätig ist.
- Falls das für jeden Artikel zutrifft, wird die Bestellung ausgeliefert.
- Ansonsten wird auf die Nachlieferung der fehlenden Ware gewartet.
- Immer wenn Ware eintrifft, wird geprüft, ob die Lieferung die für die wartende Bestellung fehlenden Artikel enthält.
- Wenn alle Artikel nachgeliefert wurden wird die Bestellung ausgeliefert.

Modellierung

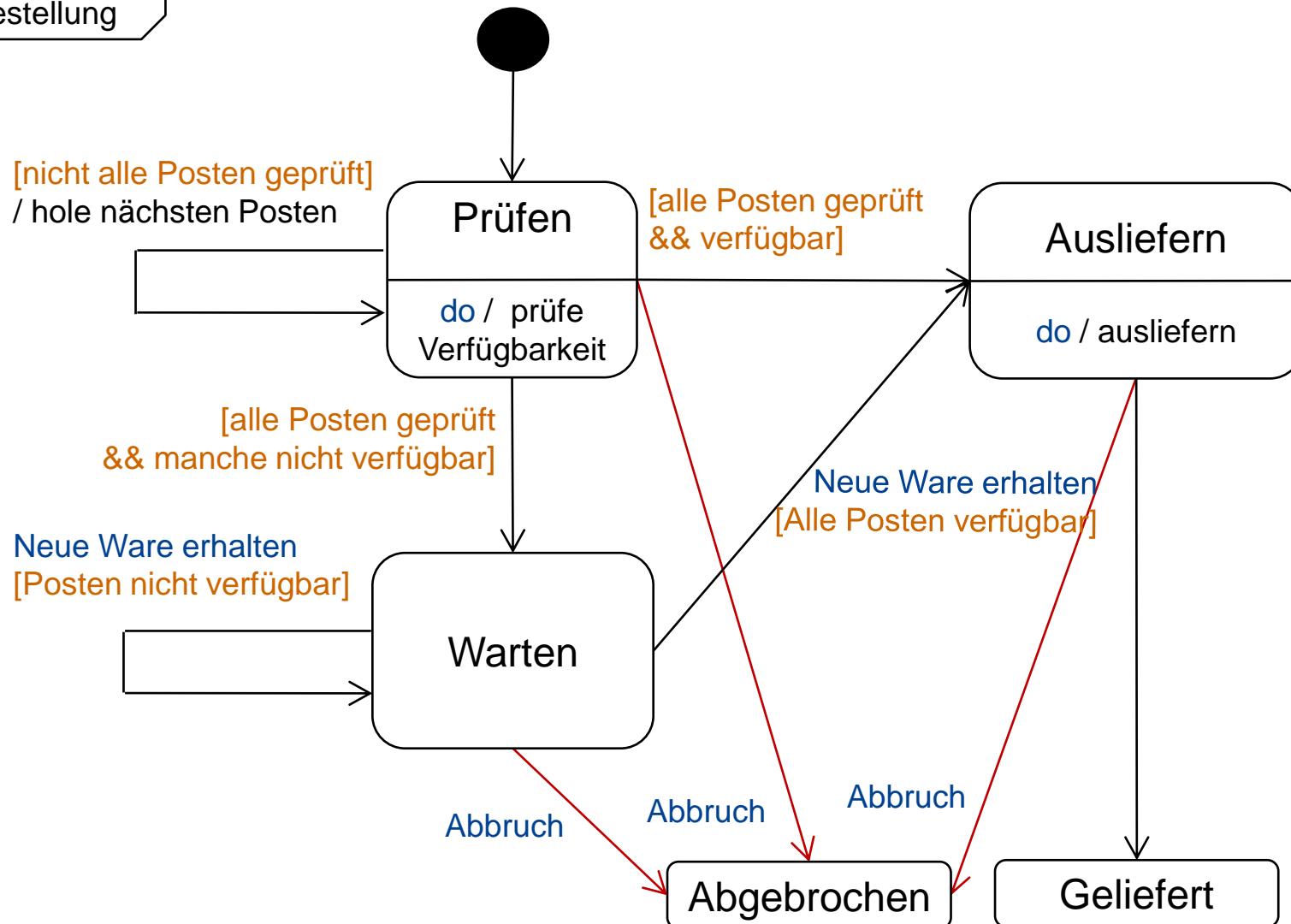
- Wir können das obige Verhalten als Aktivitätsdiagramm modellieren.
- Manchmal ist es aber hilfreicher, auf die Zustände des Systems zu schauen und die stattfindenden Zustandsübergänge → Modellierung als „Zustandsautomat“

Zustandsdiagramm: Beispiel „Bestellungsbearbeitung“

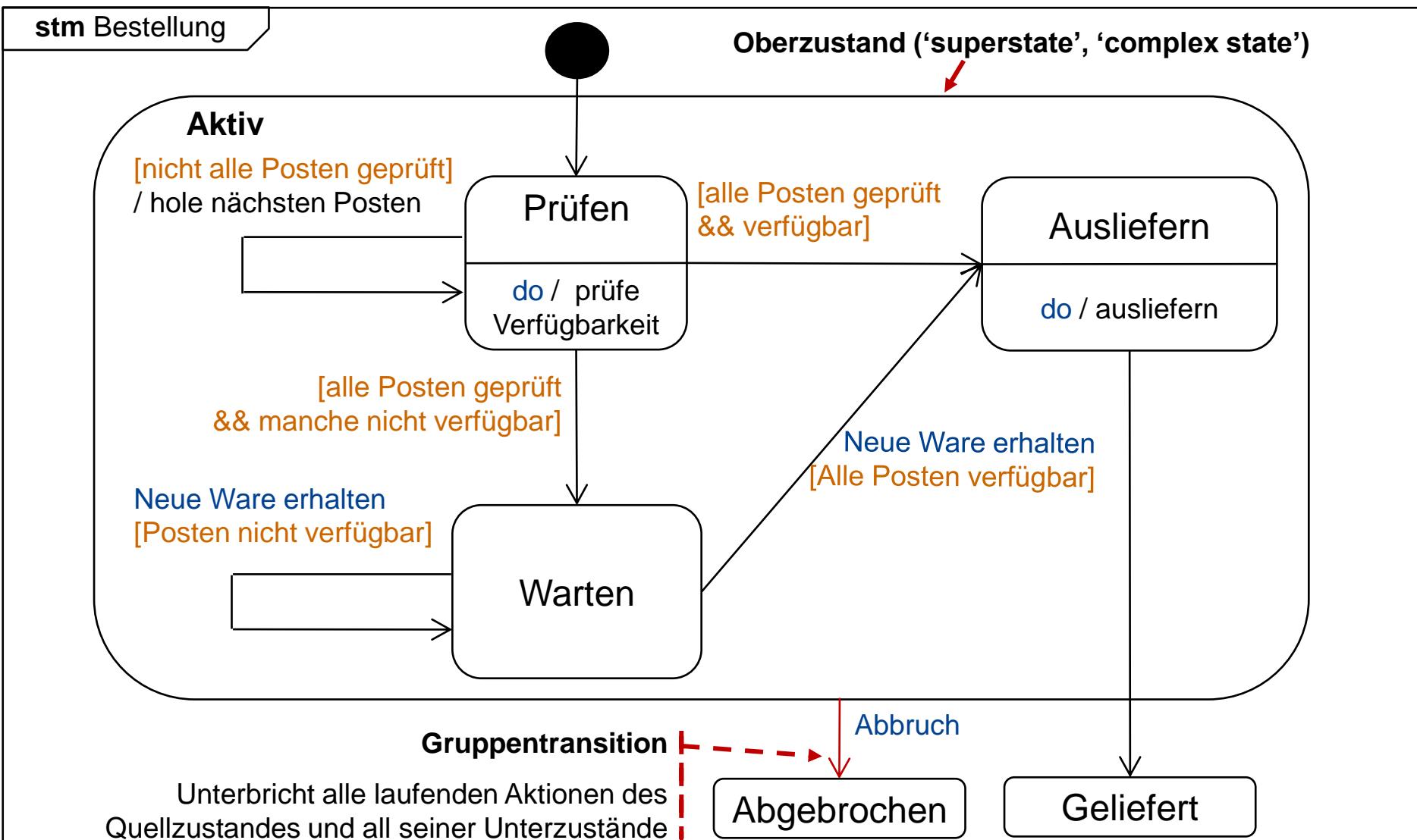


Zustandsdiagramm: Beispiel „Bestellungsbearbeitung“ mit Abbruch

stm Bestellung

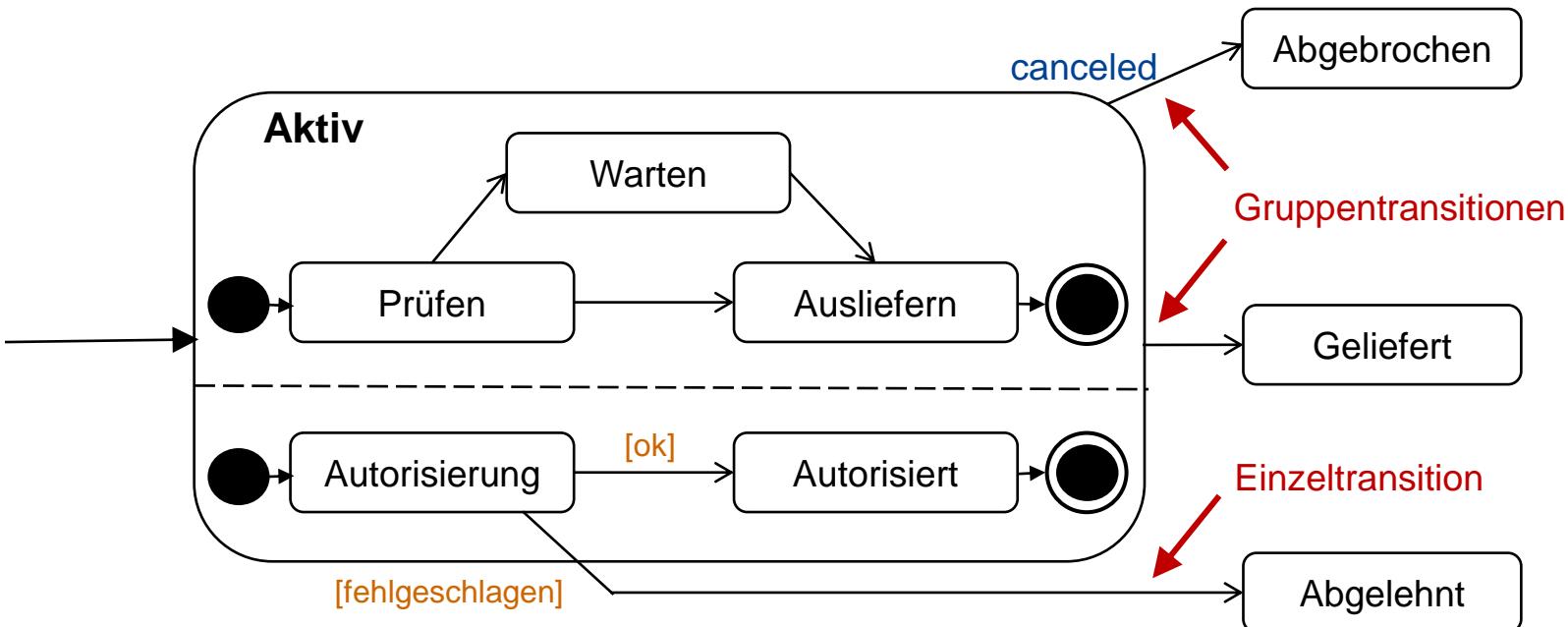


Zustandsdiagramm – Geschachtelte „Bestellungsbearbeitung“ mit Abbruch



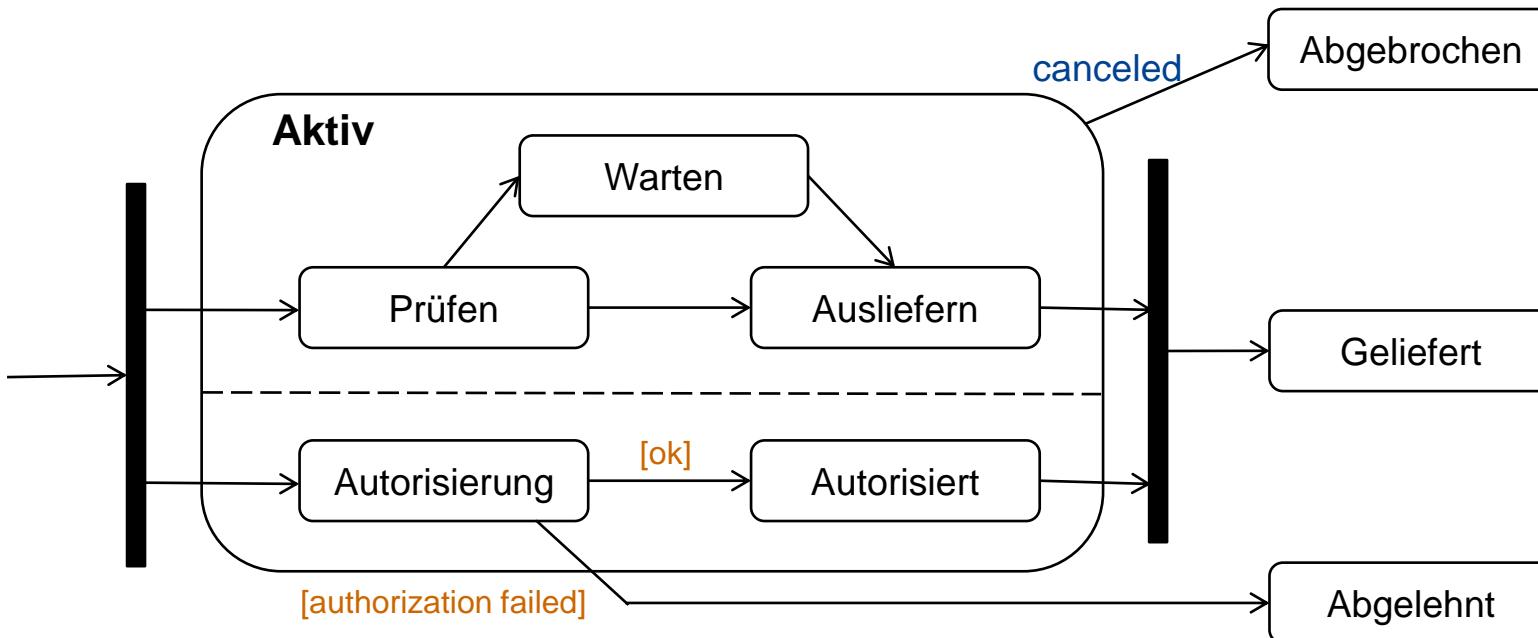
Nebenläufige Zustands-Diagramme: Mit impliziten Übergängen auf ● und von ●

- Implizite Parallelisierung
 - ◆ Implizite Transition von Außen in jeden der parallelen Startzustände
- Implizite Synchronisation
 - ◆ Implizite Transition von den parallelen Endzuständen zum nächsten äußeren Zustand
 - ◆ ... erst wenn alle parallelen Endzustände erreicht sind.



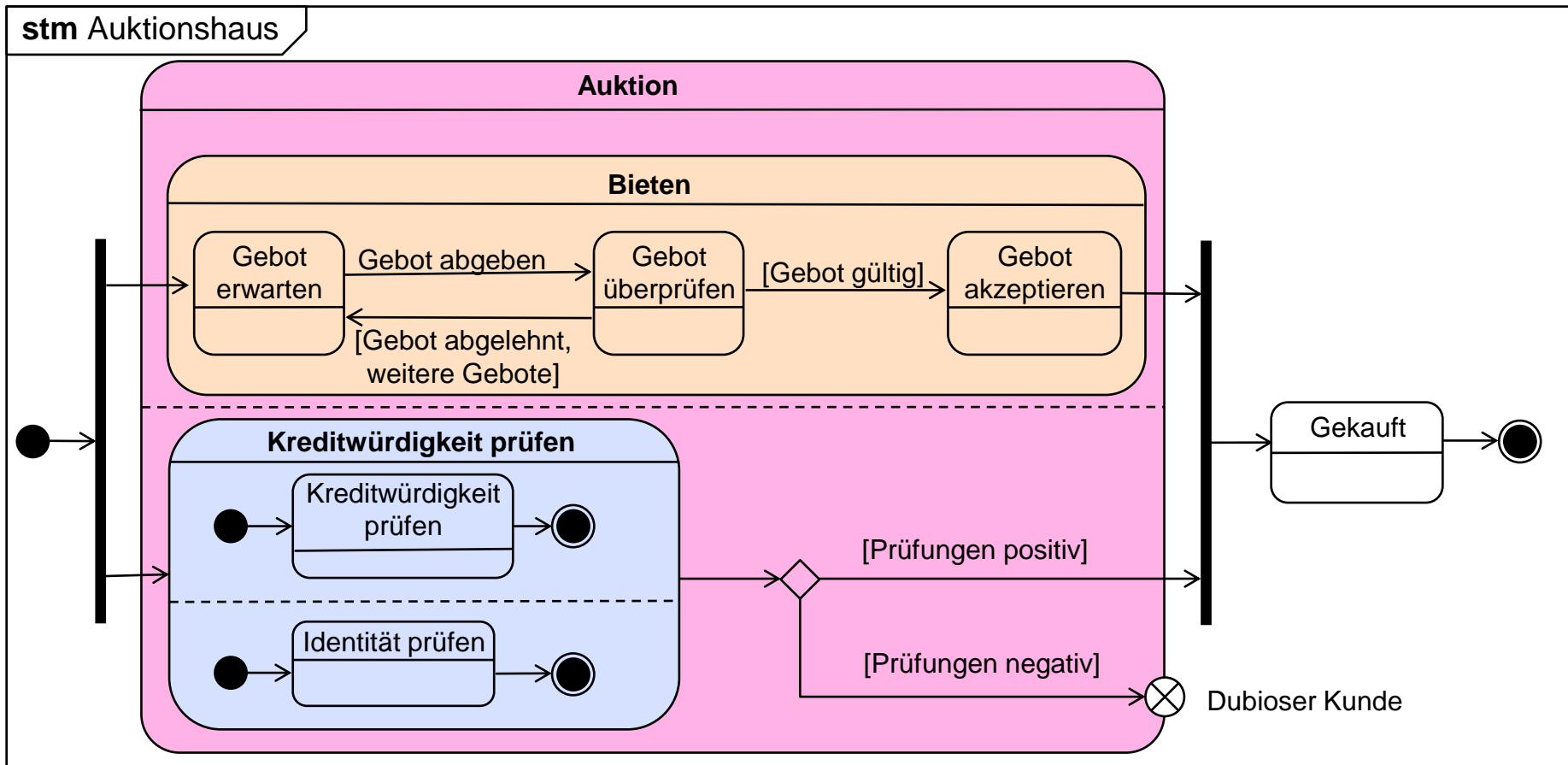
Nebenläufige Zustands-Diagramme: Mit expliziter Parallelisierung / Synchron.

- Explizite Parallelisierung: Quellzustände außerhalb, Zielzustände in verschiedenen parallelen Unterbereichen
 - ◆ Zielzustände beliebig (müssen nicht Startzustände sein)
- Explizite Synchronisation: Zielzustände außerhalb, Quellzustände in verschiedenen parallelen Unterbereichen
 - ◆ Quellzustände beliebig (müssen nicht Endzustände sein)

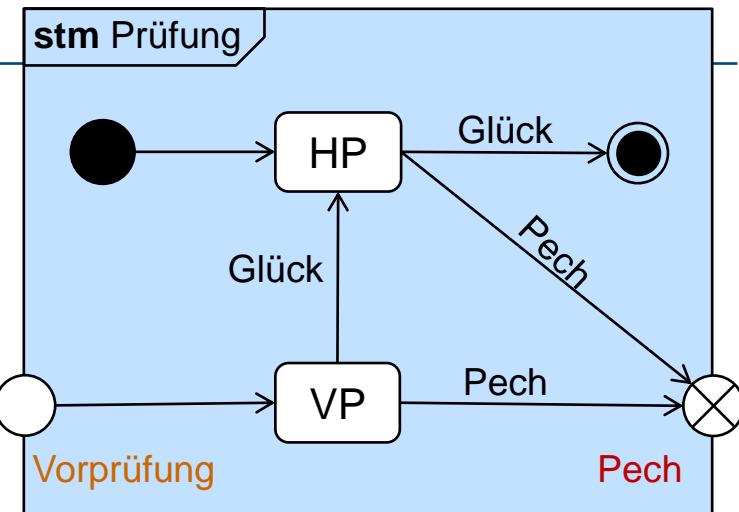
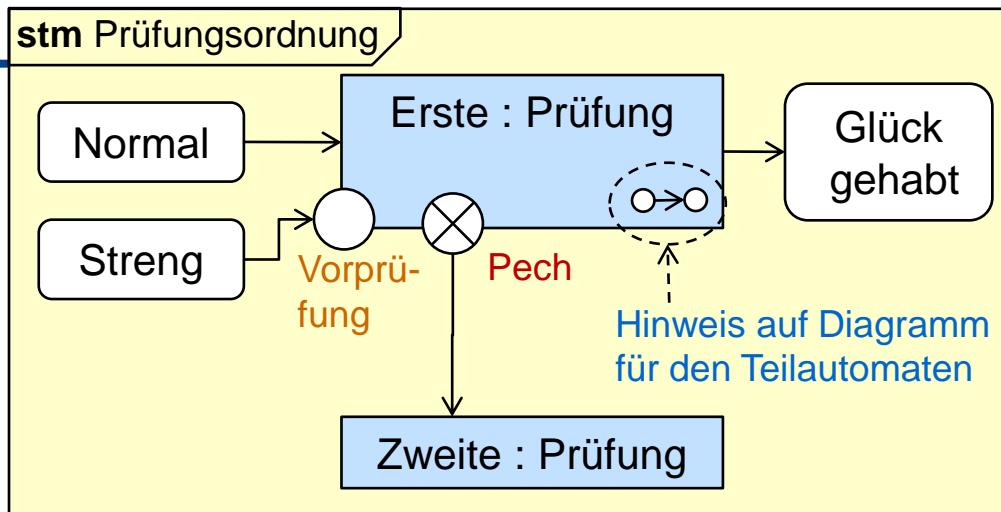


Unterautomaten: Beispiel „Auktion“

- Aus dem Aktivitätsdiagramm bekannte Kontrollflussnotation:
Entscheidung und Vereinigung



Unterautomaten: Ein- und Austiegspunkte



● Anfangszustand

- ◆ Hierhin findet eine implizite Fortsetzung statt, über die eingehenden Transitionen des umgebenden Zustands
⇒ Bsp: Normal → Erste:Prüfung

○ Einstiegspunkt

- ◆ Hierhin finden nur Transitionen über explizite Eingangskanten des Punktes statt

⇒ Bsp: Streng → Vorprüfung

● Endzustand

- ◆ Von hier findet ein implizite Fortsetzung statt, über die ausgehenden Transitionen des umgebenden Zustands die keine explizite Eventangabe haben

○ Ausstiegspunkt

- ◆ Von hier finden nur Transitionen über explizite Ausgangskanten des Punktes statt

⇒ Bsp: Pech → Pech gehabt

Transition = Zustandsübergang mit Event-Condition-Action-Semantik

- Notation
 - ◆ Spitzer Pfeil mit Beschriftung Ereignis [Bedingung] / Aktion 
- Semantik
 - ◆ Wenn das Ereignis eintritt und die Bedingung wahr ist, wird die Transition und die Aktion ausgeführt → "die Transition feuert"
 - ◆ Die Transition unterbricht evtl. noch laufende Aktionen des Quellzustandes
 - ◆ Sind mehrere Transitionen „feuerbereit“ wird nichtdeterministisch genau eine ausgewählt.
- Ereignis
 - ◆ Keines angegeben: Implizit das Ende der Aktivitäten des Quellzustands.
 - ◆ Signal(Param): Empfangenes Signal, evtl. mit Parametern.
 - ◆ Call(Param): Empfangene Nachricht, evtl. mit Parametern.
 - ◆ when(Cond): Wahr werden einer laufend überprüften Bedingung.
 - ◆ after(TimeInterval): Ablauf einer Zeitspanne (z.B. 1 Sek) seit Eintritt in den Zustand von dem die Kante ausgeht.
 - ◆ Sonstiges: Auch recht informelle "Ereignisse" zulässig.

Transition: Feinheiten der ECA-Semantik

- $\text{when}(\text{Lager leer}) / \text{Aktion}$
 - ◆ Sobald das Lager leer ist passiert Aktion



- Bestellung eingegangen $[\text{Lager leer}] / \text{Aktion}$
 - ◆ Nur wenn eine Bestellung eingeht und dann das Lager leer ist passiert Aktion



- $[\text{Lager leer}] / \text{Aktion}$
 - ◆ Wenn bei Ende der Aktivität des Quellzustands das Lager leer ist passiert Aktion



Zustand: Ein Zustand ist ein Viertupel



Name

- ◆ Mehrere unbenannte Zustände gelten als verschieden

Aktivitäten

- ◆ Was geschieht beim Eintritt in den Zustand (**entry**),
 - ✓ ... während des Zustands (**do**) und
 - ◆ ... beim Verlassen des Zustands (**exit**)



Innere Struktur

- ◆ Geschachtelte Unterzustände und die dazugehörigen Transitionen

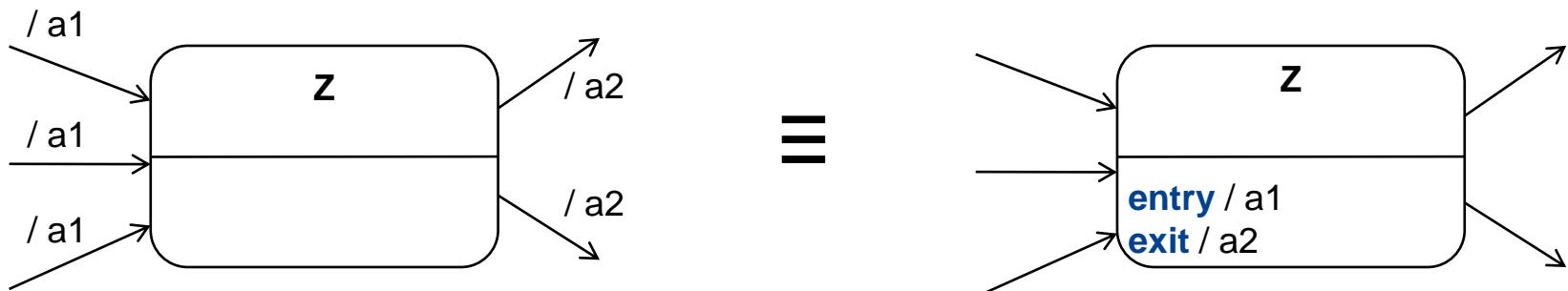
Innere Transitionen

- ◆ Interne Zustandsübergänge, die nicht die entry- und exit-Aktivitäten auslösen

Jede Kategorie ist optional aber mindestens eine muss angegeben sein.
Innere Struktur kann ersetzt werden durch Verweis auf Teilautomat.
Siehe o>o auf vorheriger Folie.

Entry- und Exit-Aktivitäten

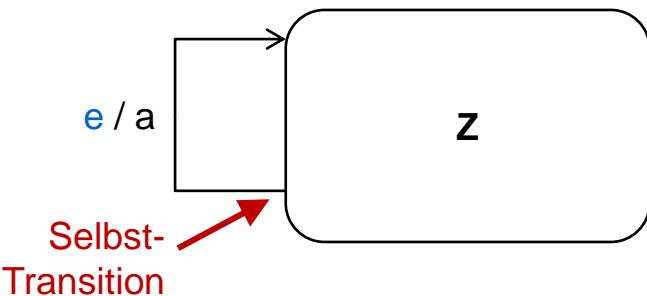
- Entry-Aktivität
 - ◆ Kurzschreibweise für Aktivität die auf **jeder** eingehenden Kante geschieht
 - Exit-Aktivität
 - ◆ Kurzschreibweise für Aktivität die auf **jeder** ausgehenden Kante geschieht
- Kapselung und Wartbarkeit
- ◆ Entry und Exit im Zustand statt auf jeder ein- und ausgehenden Kante



Innere Transitionen

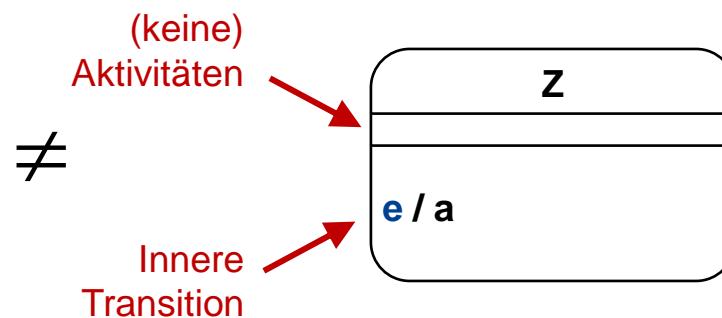
Selbst-Transition

- Externer Übergang in den selben Zustand
- Löst wie jede externe Transition alle Aktivitäten des Zustands aus.



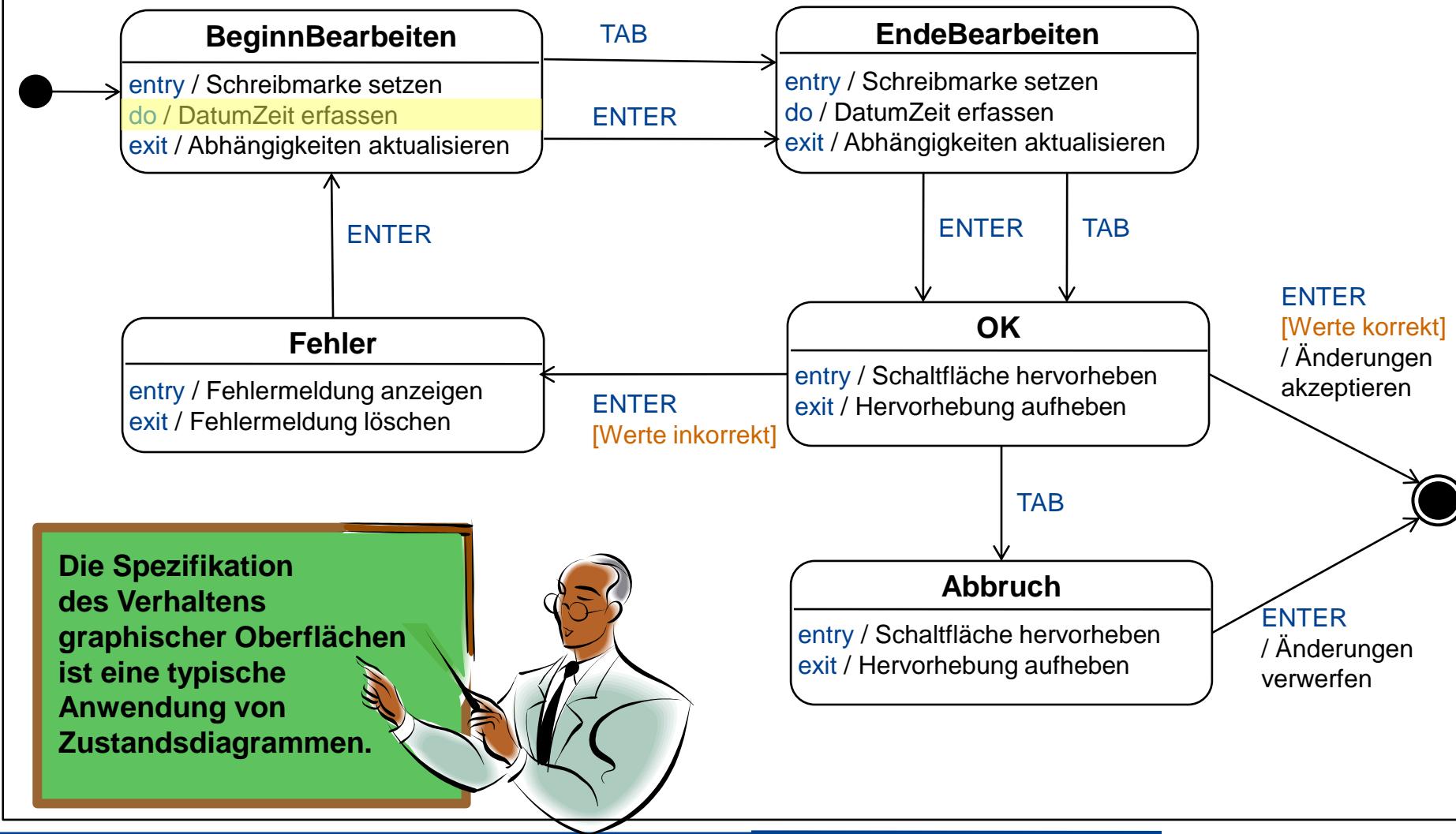
Innere Transition

- Interner Übergang in den selben Zustand
- Löst keine entry- und exit-Aktivitäten aus!

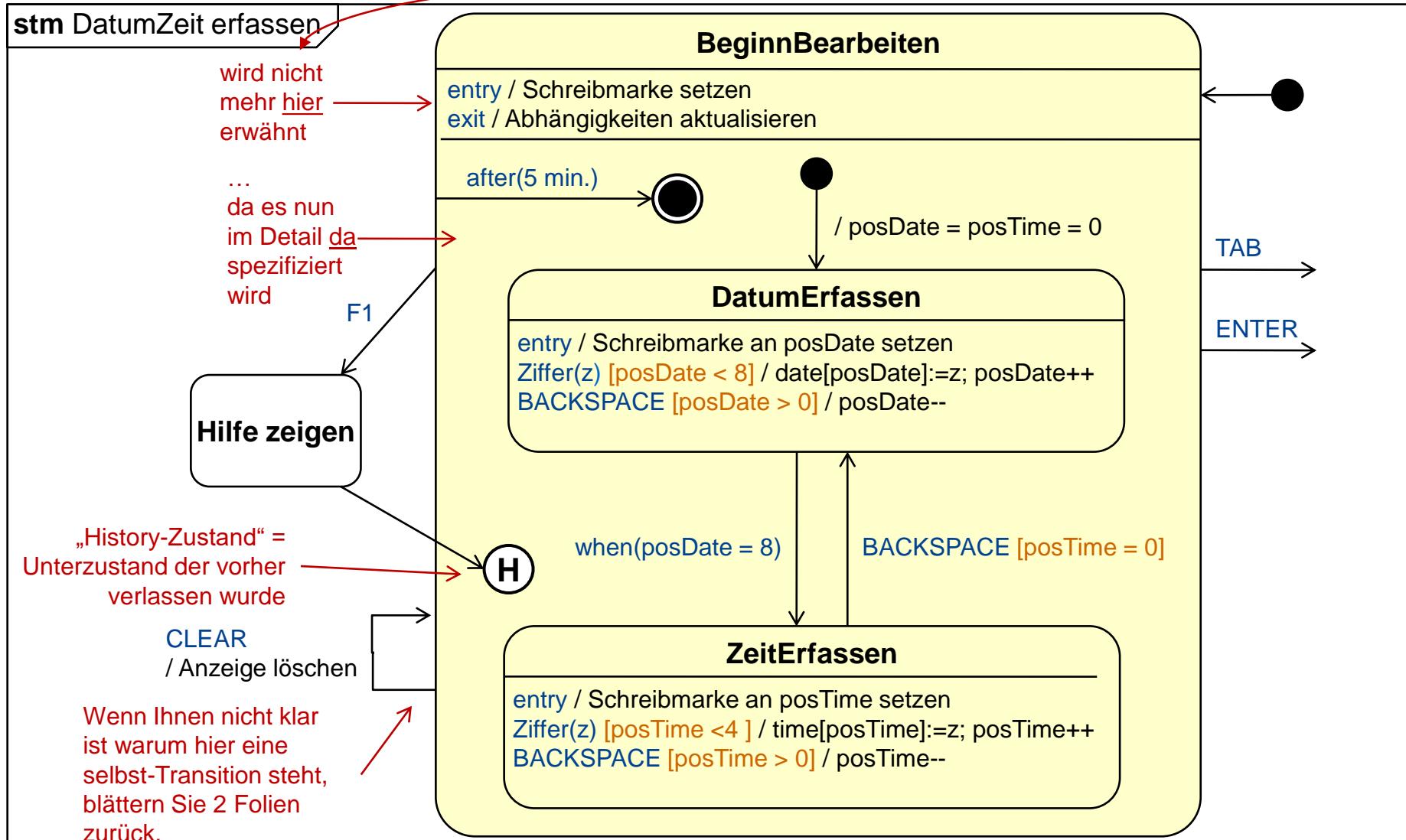


Zustandsautomat mit entry- und exit-Aktivitäten: „Termineingabeformular“

stm Termineingabeformular

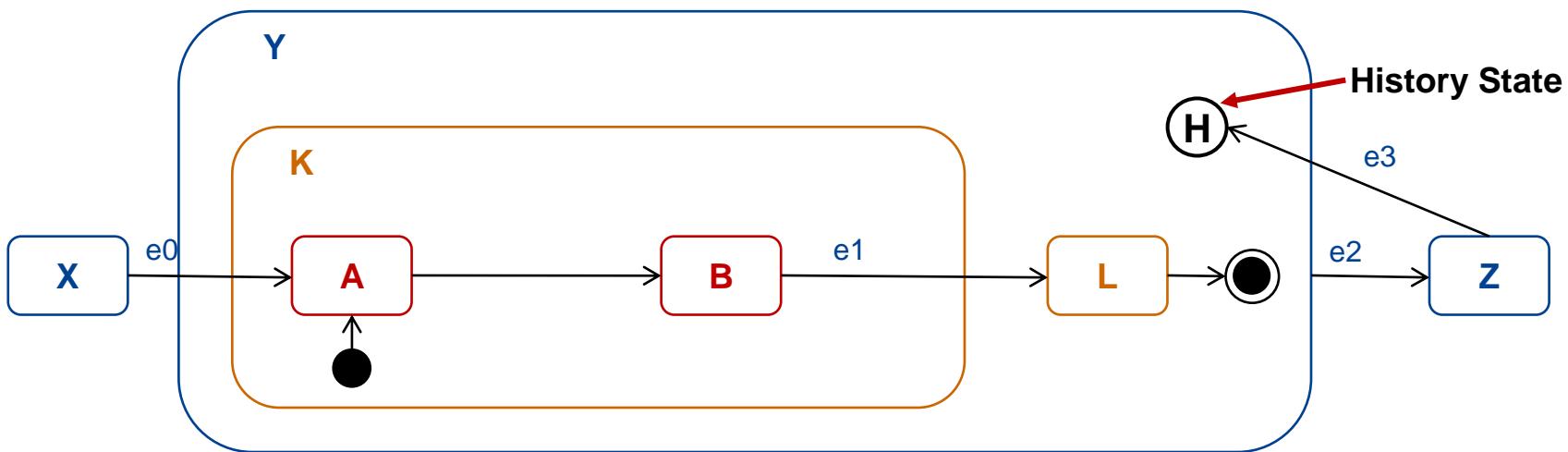


Expansion von „do / DatumZeit erfassen“



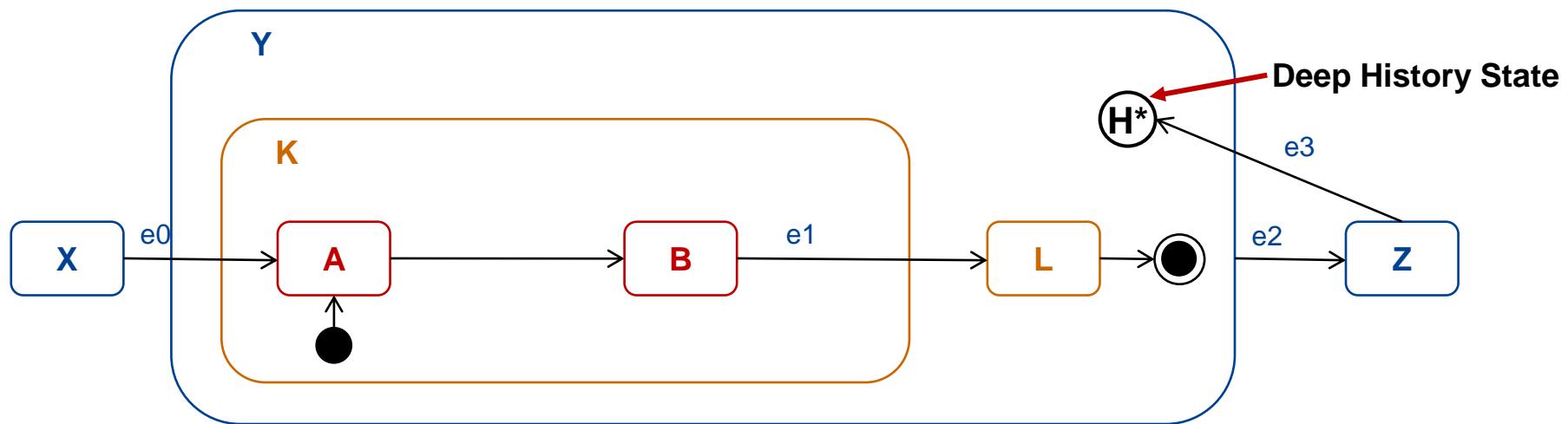
Unmittelbar vorheriger Zustand („History State“)

- **H** „merkt“ sich die Unterbrechungsgeschichte der **unmittelbaren** Teilzustände
 - ◆ Dient dazu in den **unmittelbar enthaltenen** unterbrochenen Teilzustand zurückzukehren.
 - ◆ Im folgenden Beispiel würde ‚e3‘ zu L oder K zurückkehren.
 - ◆ Falls es zu K zurückkehrt, würde es wieder im Startzustand A anfangen
 - ◆ **Merke:** **H** merkt sich nur Teilzustände auf der **nächsten Schachtelungsebene!**



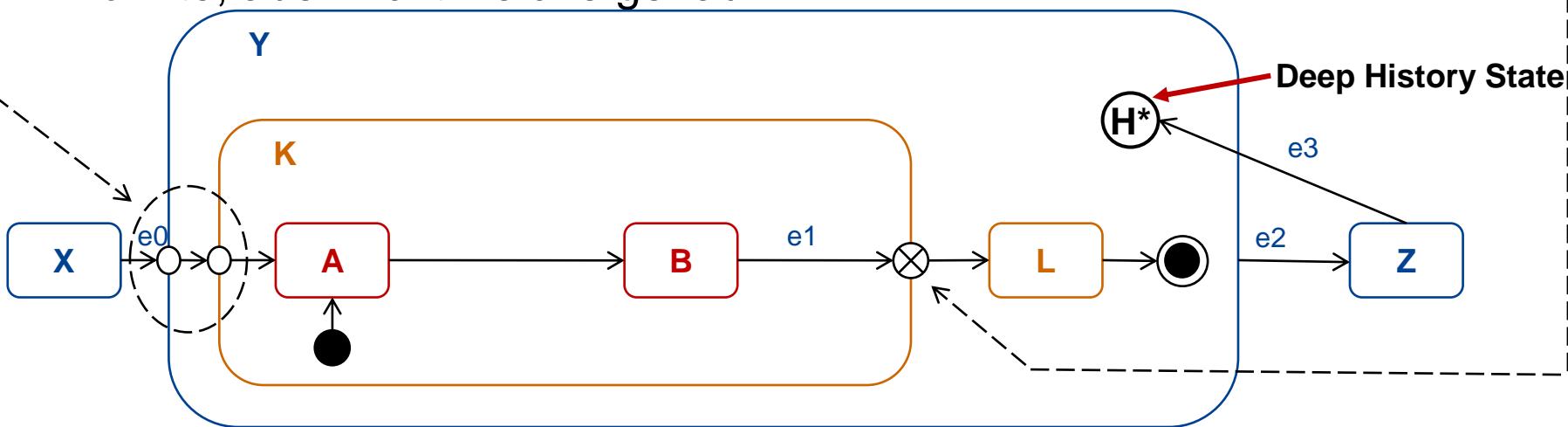
Beliebig tief geschachtelter vorheriger Zustand („Deep History State“)

- H^* „merkt“ sich die Unterbrechungsgeschichte **beliebig tief geschachtelter** Teilzustände
 - ◆ Um im vorherigen Beispiel gegebenenfalls auch nach B zurückkehren zu können würde man H^* verwenden.
 - ◆ Referenzen auf geschachtelte Unterzustände mit „Doppelpunkt-Notation“: z.B.: „Y:K:B“



Nochmal Ein- und Ausstiegspunkte

- Das Diagramm auf der vorherigen Seite hätte man auch mit **expliziten Einstiegs- und Ausstiegspunkten** modellieren können
- Vorteil: Zustände werden besser gegeneinander gekapselt
 - Äußere Zustände müssen nicht die Interna ihrer inneren Zustände kennen.
 - Z.B. muss „X“ nicht wissen, dass „Y“, „K“ und „K“ wiederum „A“ enthält.
- Die Verwendung des „Deep History State“ widerspricht der Kapselung nicht, da sie nur besagt, dass es geschachtelte Zustände geben könnte, aber nicht welche genau.



Zustandsdiagramme: Notationsüberblick

- Zustand
- Start- und Endknoten

Name des Zustands



◆ Startzustand

- ⇒ Es kann nur einen geben
- ⇒ Keine eingehenden, nur ausgehende Transitionen
- ⇒ Objekt kann in diesem Zustand nicht verweilen



◆ Endzustand

- ⇒ Es kann mehrere geben
- ⇒ Nur eingehende, keine ausgehenden Transitionen
- ⇒ Objekt muss in diesem Zustand verweilen
- ⇒ Kann Destruktion des Objekts bedeuten, muss aber nicht !



◆ Terminierungsknoten

- ⇒ Objekt, dessen Verhalten modelliert wird, hört auf zu existieren
- ⇒ Nicht identisch mit Endzustandsknoten

Zustandsdiagramme: Notationsüberblick

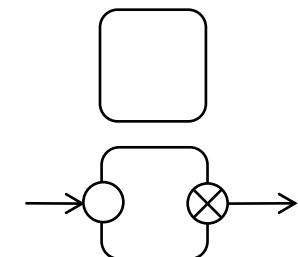
- Verbindungspunkt

- ◆ Einstiegspunkt
- ◆ Ausstiegspunkt



- Zustandsautomat

- ◆ Allgemein (Automat oder Subautomat)
- ◆ Subautomat mit Ein- uns Ausstiegspunkt



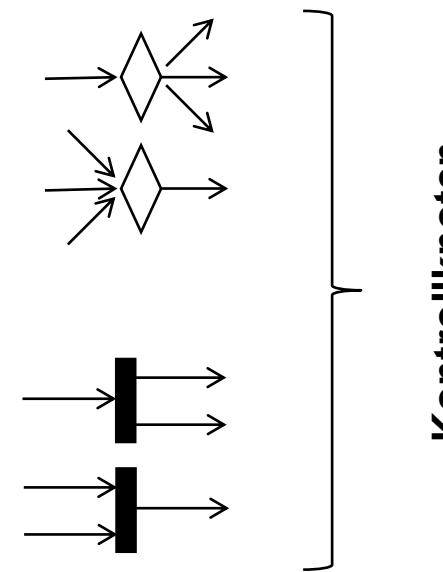
- History

- ◆ History-Zustand
- ◆ Tiefer History-Zustand



Zustandsdiagramme: Notationsüberblick

- Alternative Abläufe
 - ◆ Entscheidungsknoten
 - ◆ Vereinigungsknoten
- Nebenläufige Abläufe
 - ◆ Parallelisierungsknoten
 - ◆ Synchronisierungsknoten



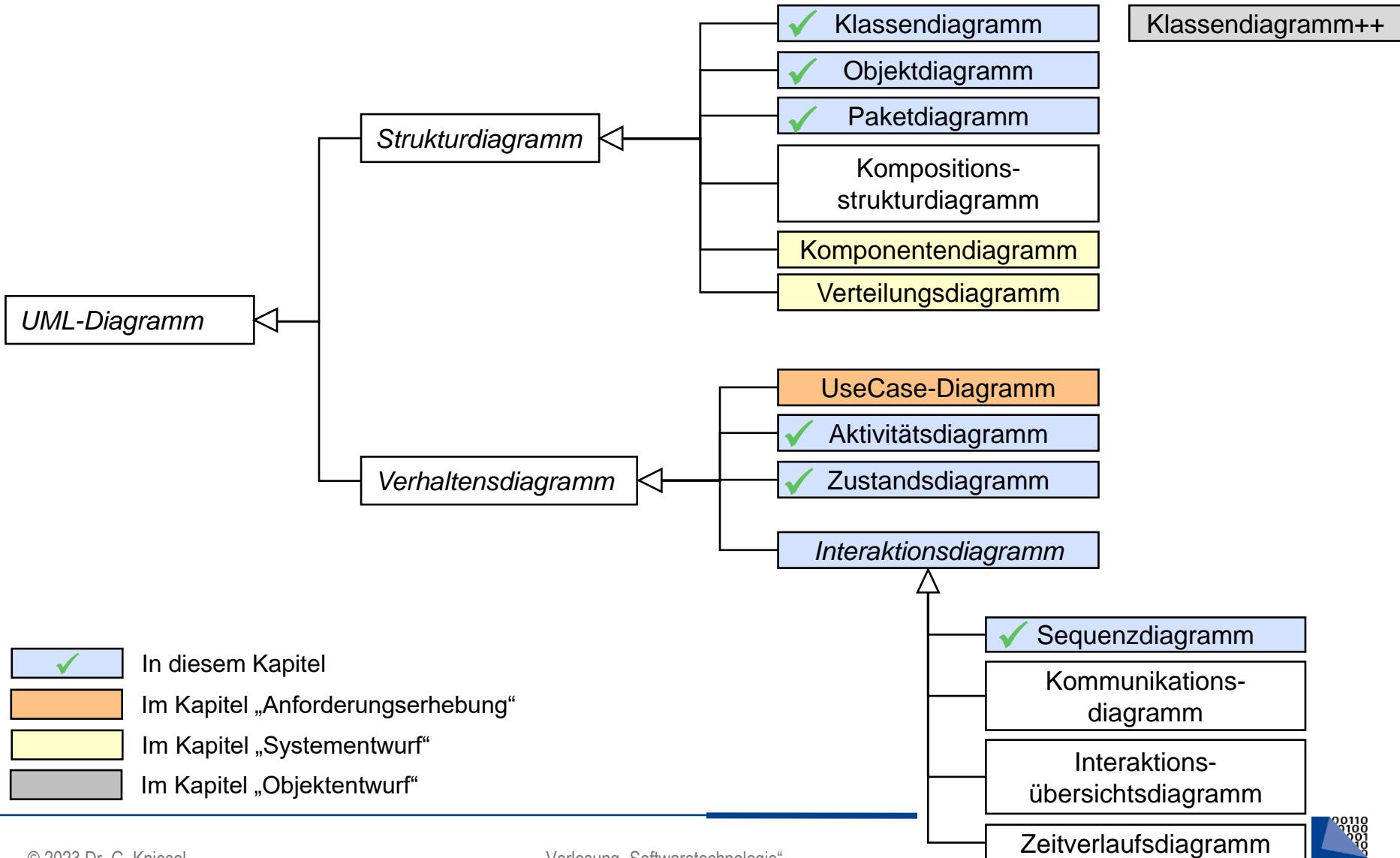
3.4 Zusammenfassung und Ausblick

Kurzrückblick der besprochenen Notationen

Kurzübersicht der ausstehenden Notationen

Überblick der Kapitel mit Inhalten aus der UML

UML Diagrammtypen: Was wird wo erläutert?

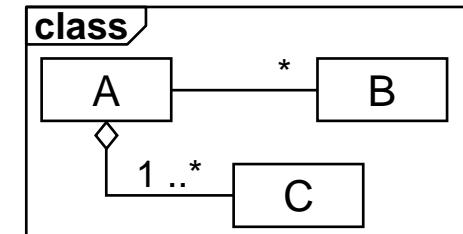


UML Kurzübersicht: Strukturdiagramme



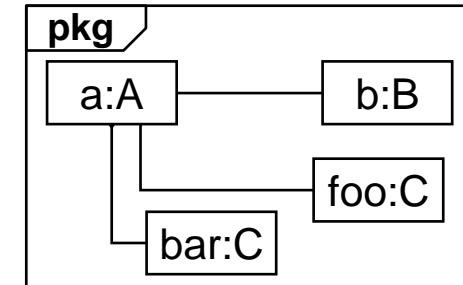
Klassendiagramm / *Class diagram*

- ◆ Beschreibt die statische Struktur des Systems:
Typen, Attribute, Operationen und Assoziationen.



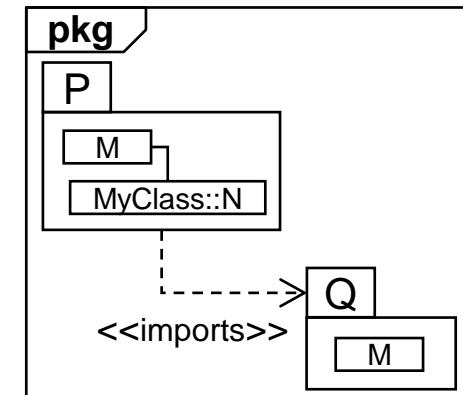
Objektdiagramm / *Object diagram*

- ◆ Beschreibt Objekte und deren Beziehung



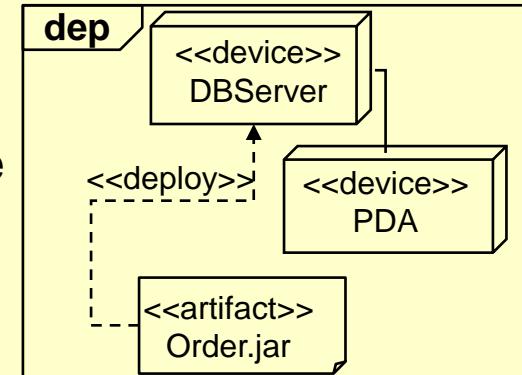
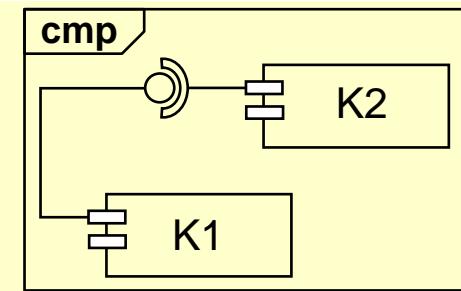
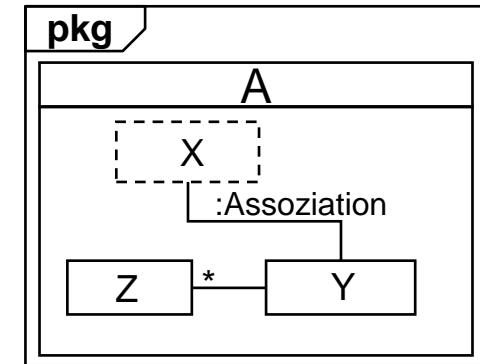
Paket Diagramm / *Package diagram*

- ◆ Beschreibt die Komposition von Paketen
- ◆ Kann Klassendiagramme enthalten



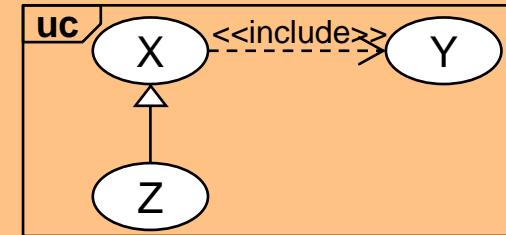
UML Kurzübersicht: Strukturdiagramme 2

- Kompositionsstrukturdiagramm / *Composite structure diagram*
 - ◆ Hierarchische Dekomposition verschiedener Systembestandteile (UML Modell-Elemente)
 - ◆ Darstellung der internen Struktur von Klassen, Komponenten, Knoten und Kollaborationen
- Komponentendiagramm / *Component diagram*
 - ◆ Zeigt die angebotenen und genutzten Schnittstellen von Komponenten
- Verteilungsdiagramm / *Deployment diagram*
 - ◆ Zeigt die Verteilung von Komponenten auf Hardware und Software (Laufzeitumgebungen)



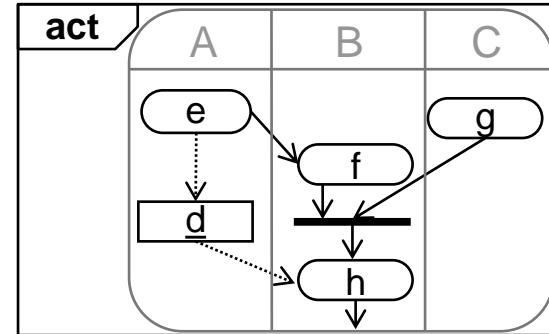
UML Kurzübersicht: Verhaltensdiagramme

- Anwendungsfalldiagramm / *Use case diagram*
 - ◆ Beschreibt das funktionelle Verhalten des Systems aus Sicht des Benutzers.



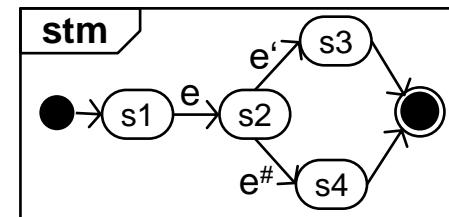
- ✓ Aktivitätsdiagramm / *Activity diagram*

- ◆ Modelliert das dynamische Verhalten eines Systems, insbesondere den Workflow, z.B. durch ein Flussdiagramm



- ✓ Zustandsdiagramm / *Statemachine diagram*

- ◆ Beschreiben das dynamische Verhalten eines individuellen Objektes als endlichen Automat



- ✓ Interaktionsdiagramm / *Interaction diagram*
- ◆ Beschreibt das Zusammenspiel verschiedener Objekte

Vier verschiedene Typen
→ s. nächste Folie

UML Kurzübersicht: Verhaltensdiagramme: Interaktion



Sequenzdiagramm / *Sequence diagram*

- ◆ Beschreibt das dynamische Verhalten zwischen Akteuren

- Kommunikationsdiagramm / *Communication diagram*

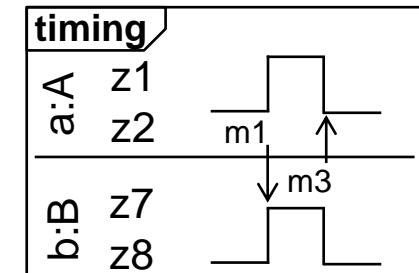
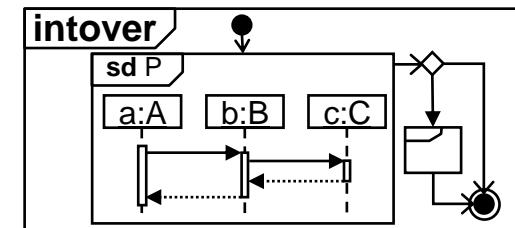
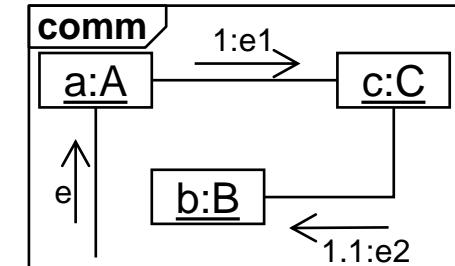
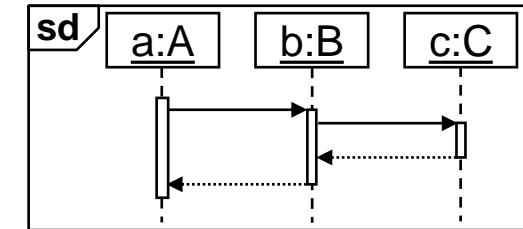
- ◆ Beschreibt das dynamische Verhalten zwischen Akteuren

- Interaktionsübersichtdiagramm / *Interaction overview diagram*

- ◆ Zeigt Interaktionsdiagramme im Kontrollfluss

- Zeitverlaufsdiagramm / *Timing diagram*

- ◆ Darstellung von Zustandsänderungen von Interaktionspartnern
- ◆ Sinnvoll bei der Modellierung von zeitkritischem Verhalten, z.B. Echtzeitsysteme



Kapitel 4

Entwurfsmuster (“Design Patterns”)

Stand: 22.11.2023  Update bis Ende Proxy (Folie 18)

Themenbereiche und Vorlesungs-Kapitel

III. PROZESSE

12

Agile Softwareentwicklung

11

Software-Prozess-Modelle

II. AKTIVITÄTEN

10

Test

9

Objekt-Design

8

System-Design

7

Anforderungs-Analyse

6

Anforderungs-Erhebung

I. WERKZEUGE

5

Refactoring

4

Entwurfsmuster

3

Unified Modelling Language (UML)

2

OO-Modellierung

1

OO-Programmierung ++

VORWISSEN

Software Configuration Management (mit Git)

OO-Programmierung Grundlagen

Imperative-Programmierung

Entwurfsmuster (Design Patterns)

Grundidee

- Dokumentation von bewährten Lösungen wiederkehrender Probleme.
- Sprache, um über Probleme und ihre Lösungen zu sprechen.
- Katalogisierungsschema um erfolgreiche Lösungen aufzuzeichnen.

Definitionsvorschlag

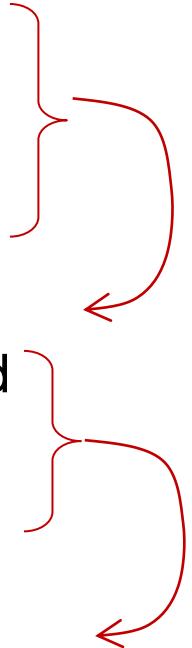
"Each pattern is a three-part rule that expresses a relation between

- a **context**,
- a **system of forces** that occur repeatedly in that context, and
- a **software configuration** that allows these forces to resolve themselves."

Richard Gabriel, on the [Patterns Home Page](#)

Pattern-Beschreibung

- Name(n) des Patterns
 - Problem, das vom Pattern gelöst wird
 - Kontext, in dem das Pattern angewendet werden kann
 - Anforderungen, die das Pattern beeinflussen
-
- Lösung. Beschreibung, wie das gewünschte Ergebnis erzielt wird
 - ◆ Varianten der Lösung
 - ◆ Beispiele der Anwendung der Lösung
-
- Konsequenzen aus der Anwendung des Patterns
-
- Bezug zu anderen Patterns
 - Bekannte Verwendungen des Patterns



Bestandteile eines Patterns: Kontext, Problem, Randbedingungen

- **Problem**
 - ◆ Beschreibung des Problems oder der Absicht des Patterns
- **Kontext (Context) / Anwendbarkeit (Applicability)**
 - ◆ Die Vorbedingungen unter denen das Pattern benötigt wird.
- **Anforderungen (Forces)**
 - ◆ Die relevanten Anforderungen und Einschränkungen, die berücksichtigt werden müssen.
 - ◆ Wie diese miteinander interagieren und im Konflikt stehen.
 - ◆ Die daraus entstehenden Kosten.
 - ◆ Typischerweise durch ein motivierendes Szenario illustriert.

Bestandteile eines Patterns: Lösung

Jede
Lösungsvariante

Die Lösung (**Software Configuration**) wird beschrieben durch:

- **Rollen**
 - ◆ Funktionen die Programmelemente im Rahmen des Patterns erfüllen.
 - ◆ Interfaces, Klassen, Methoden, Felder / Assoziationen
 - ◆ Methodenaufrufe und Feldzugriffe
 - ◆ Sie werden bei der Implementierung auf konkrete Programmelemente abgebildet (‘Player’)
- **Statische + dynamische Beziehungen**
 - ◆ Klassendiagramm, dynamische Diagramme, Text
 - ◆ Meistens ist das Verständnis des dynamischen Verhaltens entscheidend
 - ⇒ Denken in Objekten (Instanzen) statt Klassen (Typen)!
- **Teilnehmer – Participants**
 - ◆ Rollen auf Typebene (Klassen und Interfaces)
- **Beispiele der Anwendung**



Das Singleton Pattern

„Es kann nur einen geben...“

-- Juan Sánchez Villa-Lobos Ramírez

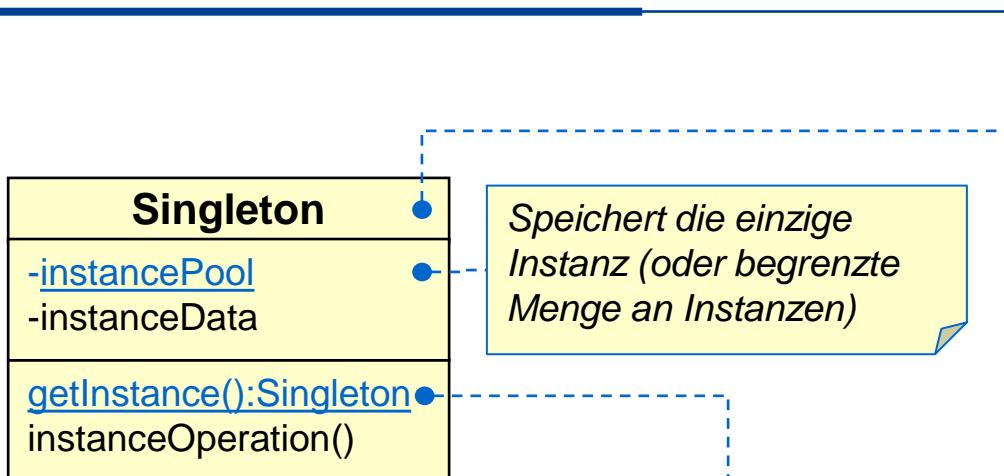


Singleton: Motivation

Beschränkung der Anzahl von Exemplaren zu einer Klasse

- Nur ein einziges Exemplar (→ „Singleton“) – häufigster Fall
 - ◆ Motivation: Zentrale Kontrolle → Z.B. über Objekterzeugung
- Feste Menge von Exemplaren größer 1 (→ „Multiton“)
 - ◆ Motivation: Platz
 - ⇒ begrenzte Ressourcen (z.B. auf mobilen Geräten)
 - ◆ Motivation: Zeit
 - ⇒ Teure Objekterzeugung durch „Object Pool“ vermeiden
 - ⇒ z.B. 1000 Instanzen eines Microservice in einer Queue vorhalten. Bei Anfrage erstes Service-Objekt aus Queue den Auftrag zuweisen. Danach wird das Service-Objekt zurück in die Queue gestellt und kann sofort wieder verwendet werden.

Singleton: Struktur + Implementierung



Besitzt nur private Konstruktoren:

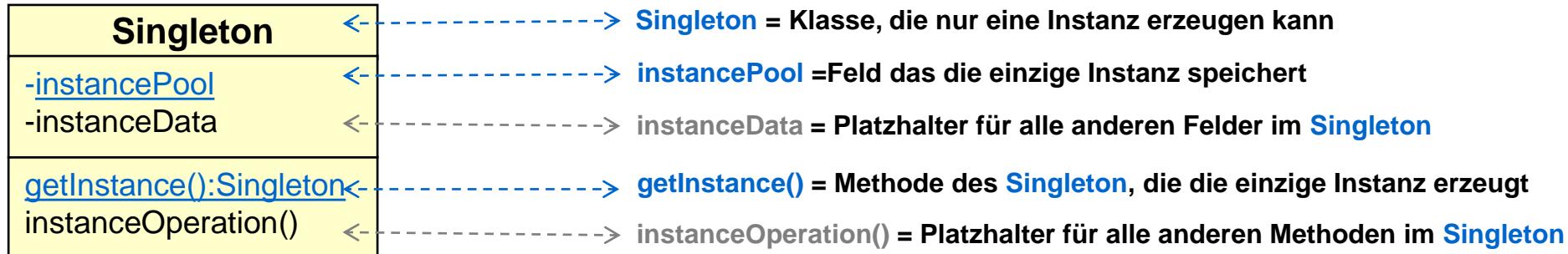
```
private Singleton() {  
    this.Singleton(arg1, ..., argN);  
}  
private Singleton(par1, ..., parN) {  
    ...  
}
```

Liefert eine Instanz (typischerweise immer die Selbe):

```
public static Singleton getInstance()  
{  
    if (instancePool == null)  
        instancePool = new Singleton();  
    return instancePool;  
}
```

- Nur private Konstruktoren
 - ◆ so wird verhindert, dass Clients beliebig viele Instanzen erzeugen können
 - ◆ Implementierung in Java: Unbedingt auch einen **privaten Konstruktor mit leerer Argumentliste** implementieren! → Sonst erzeugt der Compiler einen **öffentlichen Konstruktor mit leerer Argumentliste**.
- `instancePool` als Registry für alle Instanzen
 - ◆ Bei Multiton: typischerweise als Queue oder Stack realisiert (→ möglichst einfacher lookup-Mechanismus um schnell eine Instanz auszuwählen)

Patterns abstrahieren Rollen im Design



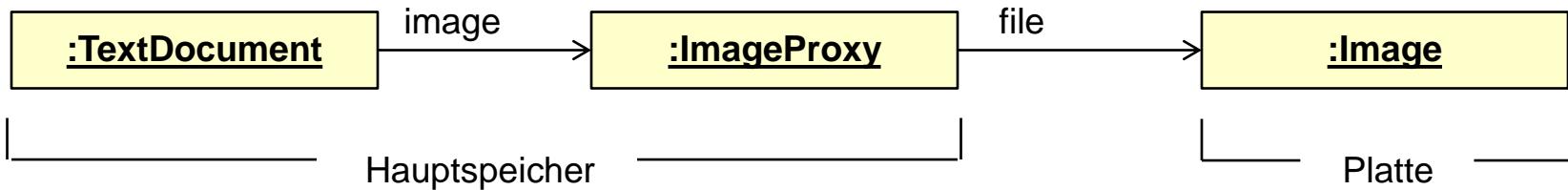
Anwendung von Patterns = Abbildung der Rollen auf gegebenes Design

- Vorhandenes Design verstehen
 - ◆ „Die Klasse **Iconkit** erfüllt alle Rollen des **Singleton**.“
→ „Aha, das ist ein Singleton!“
- Vorhandenes Design verbessern
 - ◆ „Die Klasse **MyClass** soll nur eine Instanz erzeugen
→ Aha, ich muss die schon vorhandenen **Singleton**-Rollen identifizieren und die fehlenden implementieren“

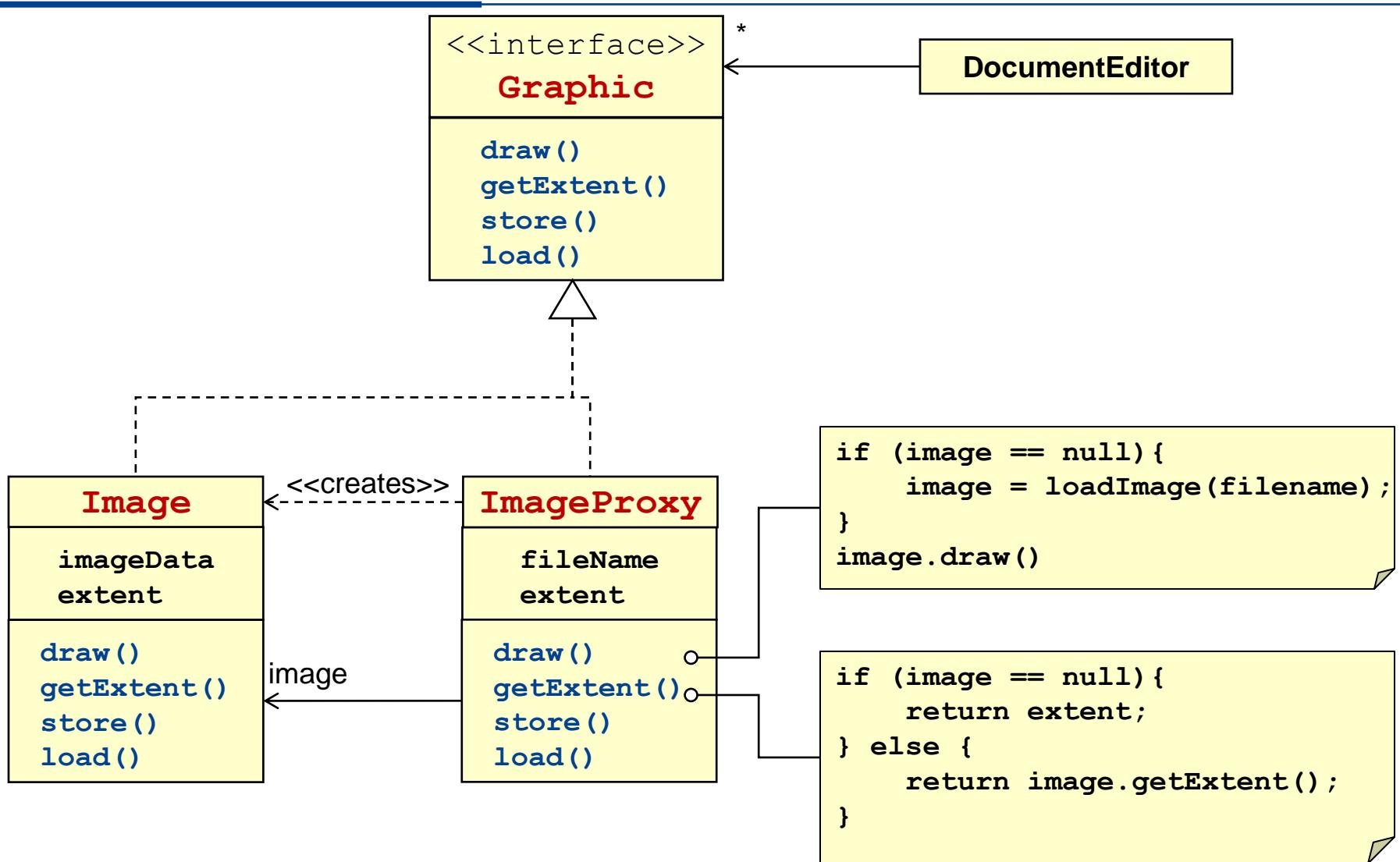
Das Proxy Pattern

Proxy Pattern (auch: Surrogate, Smart Reference)

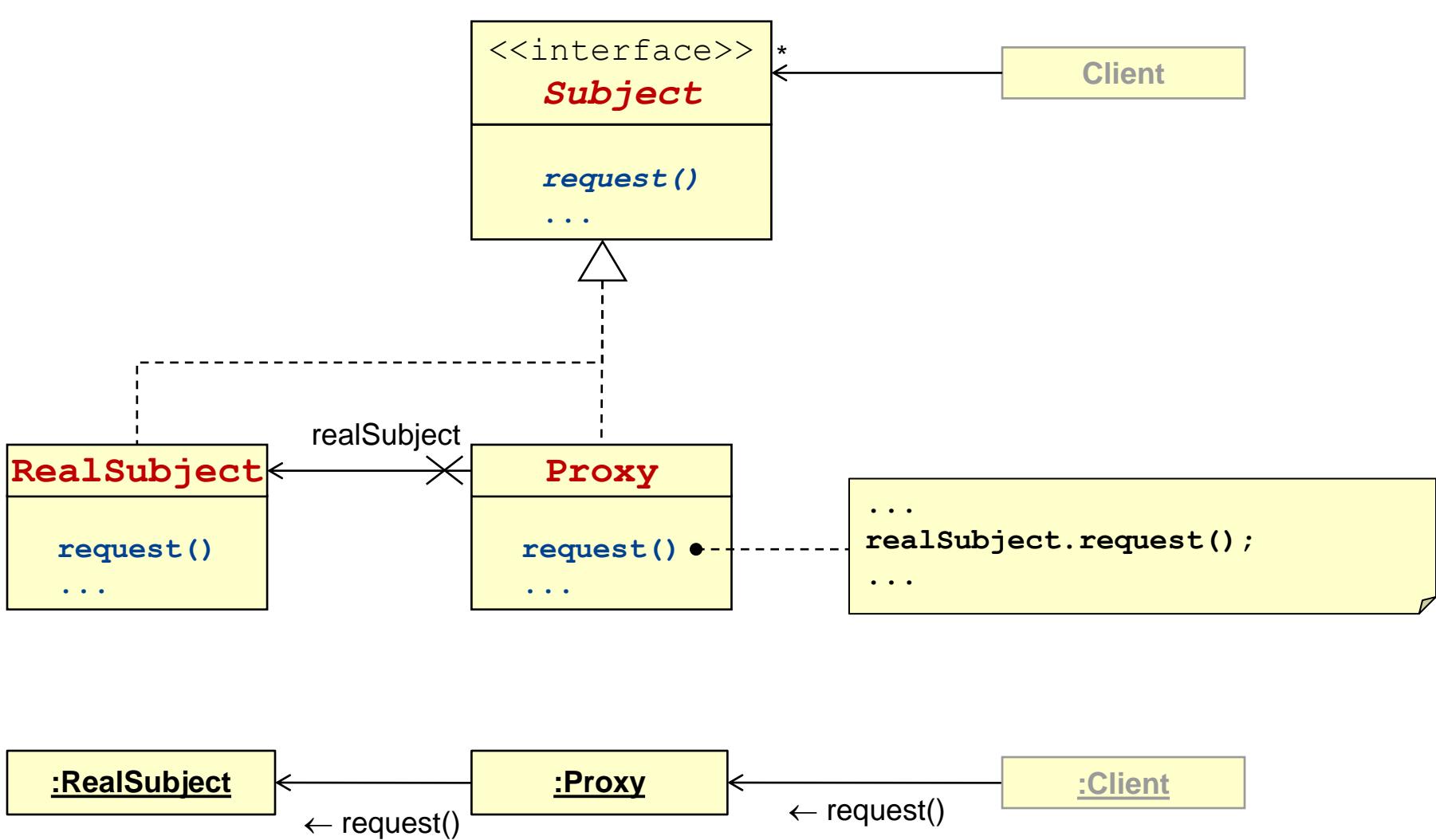
- Absicht
 - ◆ Stellvertreter für ein anderes Objekt
 - ◆ bietet Kontrolle über Objekt-Erzeugung und -Zugriff
- Motivation
 - ◆ kostspielige Objekt-Erzeugung verzögern (zB: große Bilder)
 - ◆ verzögerte Objekterzeugung soll Programmstruktur nicht global verändern
- Idee
 - ◆ Bild-Stellvertreter (Proxy) verwenden
 - ◆ Bild-Stellvertreter verhält sich aus Client-Sicht wie Bild
 - ◆ Bild-Stellvertreter erzeugt Bild bei Bedarf



Proxy Pattern: Beispiel

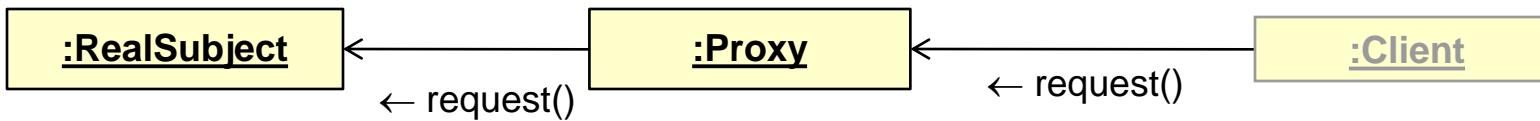


Proxy Pattern: Schema



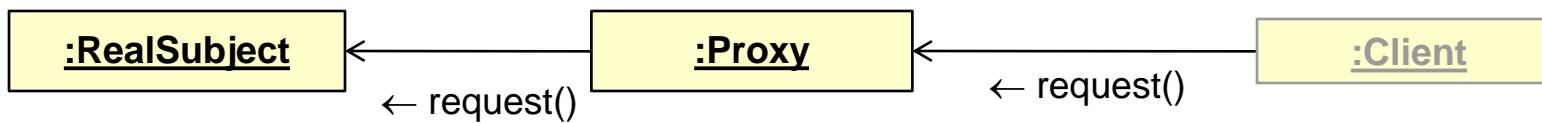
Proxy Pattern: Verantwortlichkeiten

- Subject
 - ◆ definiert das gemeinsame Interface
- RealSubject
 - ◆ das Objekt das der Proxy vertritt
 - ◆ beinhaltet die eigentliche Anwendungs-Funktionalität
- Proxy
 - ◆ bietet gleiches Interface wie "Subject"
 - ◆ besitz die einzige Referenz auf eine "RealSubject"-Instanz
 - ◆ kontrolliert alle Aktionen auf der "RealSubject"-Instanz und erweitert teilweise ihre Funktionalität (zusätzliche Checks, Buchhaltung, Aktionen)
- Zusammenspiel
 - ◆ selektives Forwarding



Proxy Pattern: Anwendbarkeit

- Virtueller Proxy
 - ◆ verzögerte Erzeugung "teurer" Objekte bei Bedarf
 - ◆ Beispiel: große Bilder in Dokument, persistente Objekte
- Remote Proxy
 - ◆ Zugriff auf Objekt auf einem anderen Rechner → Bsp: CORBA, RMI
 - ◆ Objekt-Migration (über verschiedene Rechner hinweg) → „mobile Agenten“
- Concurrency Proxy
 - ◆ Verwendung eines RealSubject der nicht „threadsafe“ ist in einer „multi-threaded“ Umgebung
 - ◆ Pufferung von Anfragen aus verschiedenen Threads in einer über locks gesicherten Queue
 - ◆ locking der RealSubject-Referenz und einzelne Weiterleitung der Anfragen



Proxy Pattern: Implementierung der Weiterleitung

Wege, um nicht jede Weiterleitungsmethode manuell zu implementieren:

- **C++:** Operator-Overloading
 - ◆ Proxy redefiniert Dereferenzierungs-Operator: `*anImage`
 - ◆ Proxy redefiniert Member-Accesss-Operator: `anImage->extent()`
- **Smalltalk:** Reflektion
 - ◆ Proxy redefiniert Methode "doesNotUnderstand: aMessage"
- **Java:** Reflektion
 - ◆ Dynamic Proxy implementiert „InvocationHandler“ Interface
- **Kotlin:** Schlüsselwort
 - ◆ `class Proxy(ri: ReallImage) : ReallImage by ri {... }`
 - ◆ Für alle in Proxy nicht definierten Methoden aus ReallImage werden vom Compiler in Proxy Methoden generiert, die an das Feld `ri` vom Typ `ReallImage` weiterleiten

Das Observer Pattern

Grundlage von Repository- und MVC-Architektur

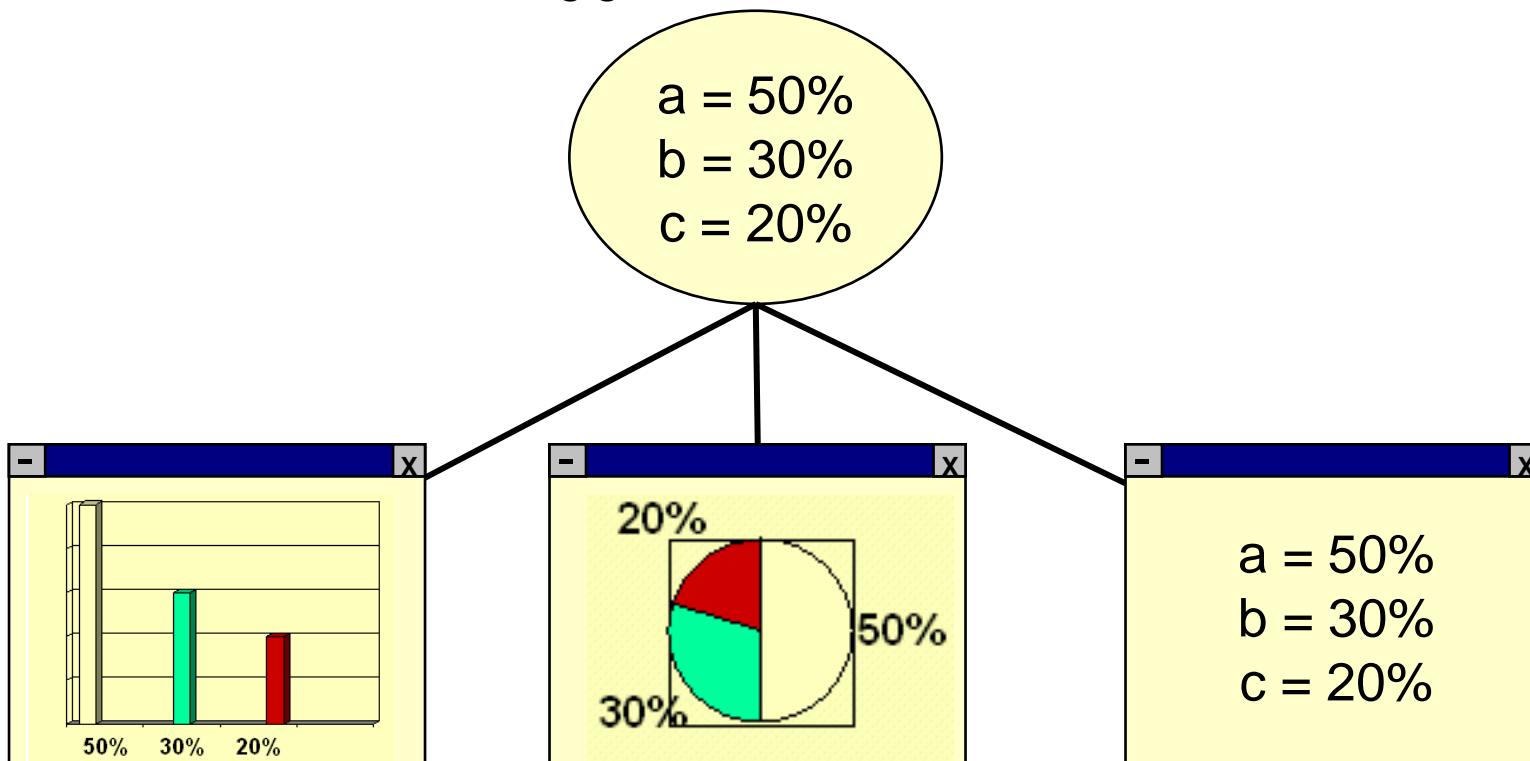
Das Observer Pattern: Einführung

- Absicht
 - ◆ Stellt eine 1-zu-n Beziehung zwischen Objekten her
 - ◆ Wenn das eine Objekt seinen Zustand ändert, werden die davon abhängigen Objekte benachrichtigt und entsprechend aktualisiert
- Andere Namen
 - ◆ "Dependents", "Publish-Subscribe", "Listener"
- Motivation
 - ◆ Verschiedene Objekte sollen zueinander konsistent gehalten werden
 - ◆ Andererseits sollen sie dennoch nicht eng miteinander gekoppelt sein.
(bessere Wiederverwendbarkeit)
- Diese Ziele stehen in einem gewissen Konflikt zueinander.
Man spricht von „*conflicting forces*“, gegenläufig wirkenden Kräften.

Das Observer Pattern: Beispiel

- Trennung von Daten und Darstellung

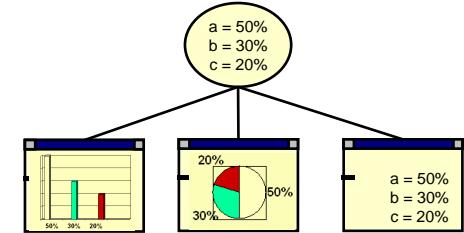
- ◆ Wenn am Modell Änderungen vorgenommen werden, werden alle Sichten des Modells aktualisiert
- ◆ Sichten sind aber unabhängig voneinander.
- ◆ Das Modell ist unabhängig von den Sichten.



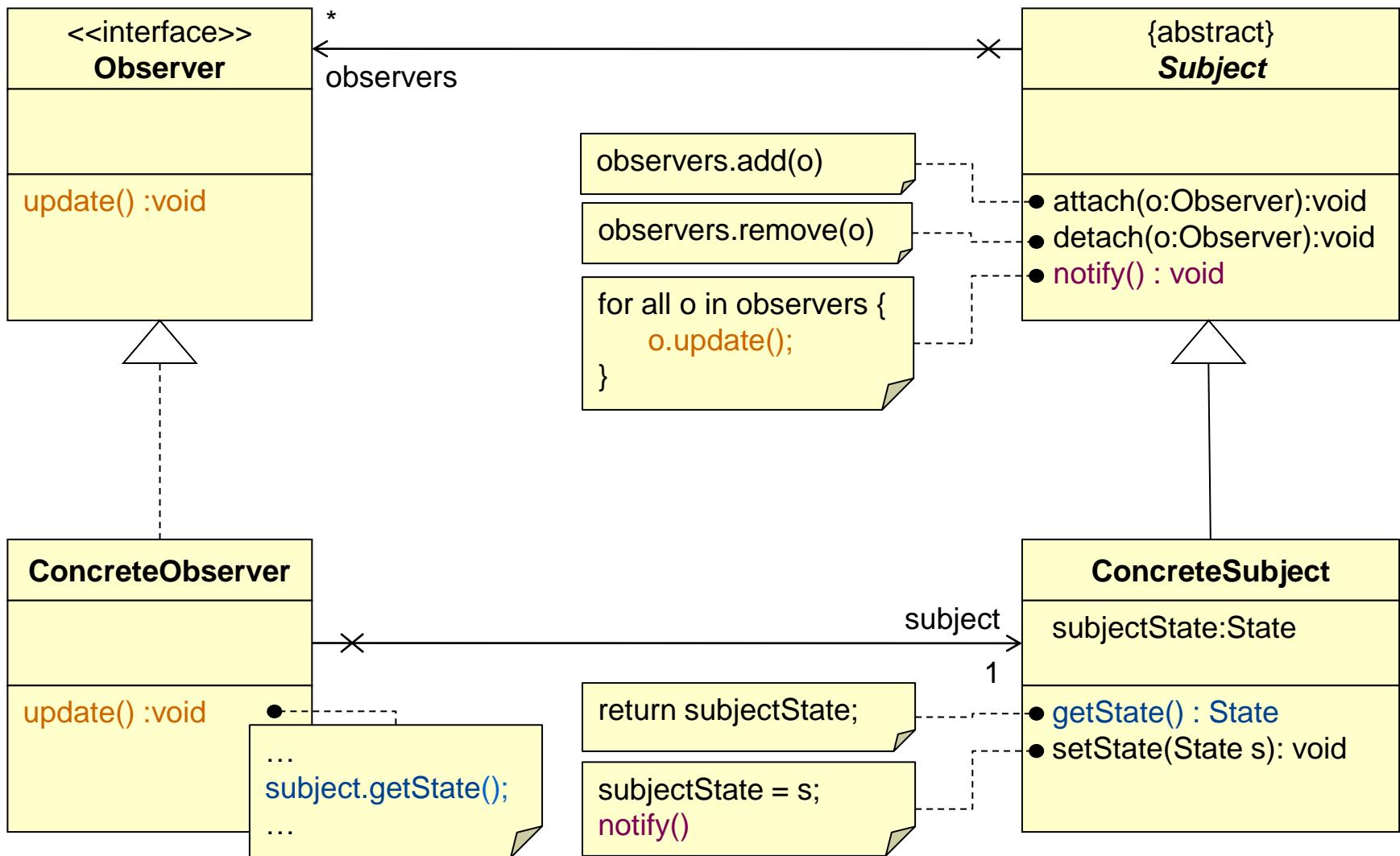
Das Observer Pattern: Anwendbarkeit

Das Pattern ist in folgenden Kontexten anwendbar:

- Abhängigkeiten
 - ◆ Ein Aspekt einer Abstraktion ist abhängig von einem anderen Aspekt.
 - ◆ Aufteilung dieser Aspekte in verschiedene Objekte erhöht Variationsmöglichkeit und Wiederverwendbarkeit.
- Folgeänderungen
 - ◆ Änderungen an einem Objekt erfordert Änderungen an anderen Objekten.
 - ◆ Es ist nicht bekannt, wie viele Objekte geändert werden müssen.
- Lose Kopplung
 - ◆ Objekte sollen andere Objekte benachrichtigen können, ohne Annahmen über die Beschaffenheit dieser Objekte machen zu müssen.



Das Observer Pattern: Struktur (N:1, Pull-Modell)



Rollenaufteilung / Verantwortlichkeiten (Pull Modell)

- Observer („Beobachter“) -- auch: Subscriber, Listener
 - ◆ `update()` -- auch: `handleEvent`
 - ⇒ Reaktion auf Zustandsänderung des Subjects
- Subject („Subjekt“) -- auch: Publisher
 - ◆ `attach(Observer o)` -- auch: `register`, `addListener`
 - ⇒ Observer registrieren
 - ◆ `detach(Observer o)` -- auch: `unregister`, `removeListener`
 - ⇒ registrierte Observer entfernen
 - ◆ `notify()`
 - ⇒ update-Methoden aller registrierten Observer aufrufen
 - ◆ `setState(...)`
 - ⇒ zustandsändernde Operation(en)
 - ⇒ für internen Gebrauch und beliebige Clients
 - ◆ `getState()`
 - ⇒ Abfrage des aktuellen Zustands
 - ⇒ damit Observer feststellen können was sich wie geändert hat

Das Observer Pattern: Konsequenzen

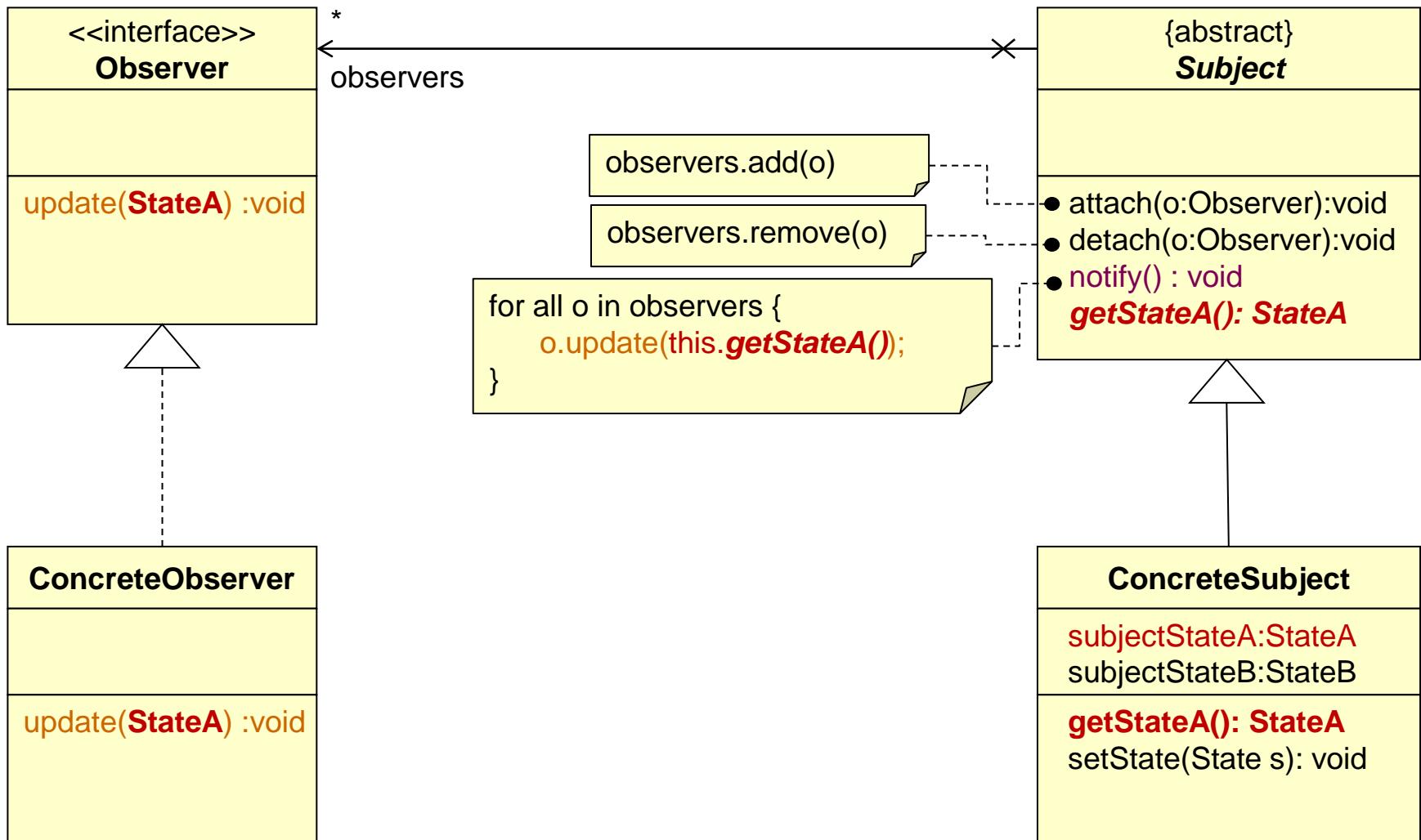
- Unabhängigkeit
 - ◆ Konkrete Beobachter können **hinzugefügt** werden, ohne konkrete Subjekte oder andere konkrete Beobachter zu ändern.
 - ◆ Konkrete Subjekte können unabhängig voneinander und von konkreten
 - ⇒ **variiert** werden
 - ⇒ **wiederverwendet** werden.
- „Broadcast“-Nachrichten
 - ◆ Subjekt benachrichtigt alle angemeldeten Beobachter
 - ◆ Beobachter entscheiden, ob sie Nachrichten behandeln oder ignorieren
- Unerwartete Aktualisierungen
 - ◆ Kleine Zustandsänderungen des Subjekts können komplexe Folgen haben.
 - ◆ Auch uninteressante Zwischenzustände können unnötige Aktualisierungen auslösen.

Das Observer Patterns: Implementierung

Wie werden die Informationen über eine Änderung weitergegeben:
„push“ versus „pull“

- **Pull:** Subjekt über gibt in „update()“ keinerlei Informationen, aber die Beobachter müssen sich die Informationen vom Subjekt holen
 - ◆ + Geringere Kopplung zwischen Subjekt und Beobachter.
 - ◆ – Berechnungen werden häufiger durchgeführt.
- **Push:** Subjekt über gibt in Parametern von „update()“ detaillierte Informationen über Änderungen.
 - ◆ + Rückaufrufe werden seltener durchgeführt.
 - ◆ – Beobachter sind weniger wiederverwendbar (Abhängig von den Parameter typen)
- Zwischenformen sind möglich

Das Observer Pattern: Struktur (N:1, Push-Modell)

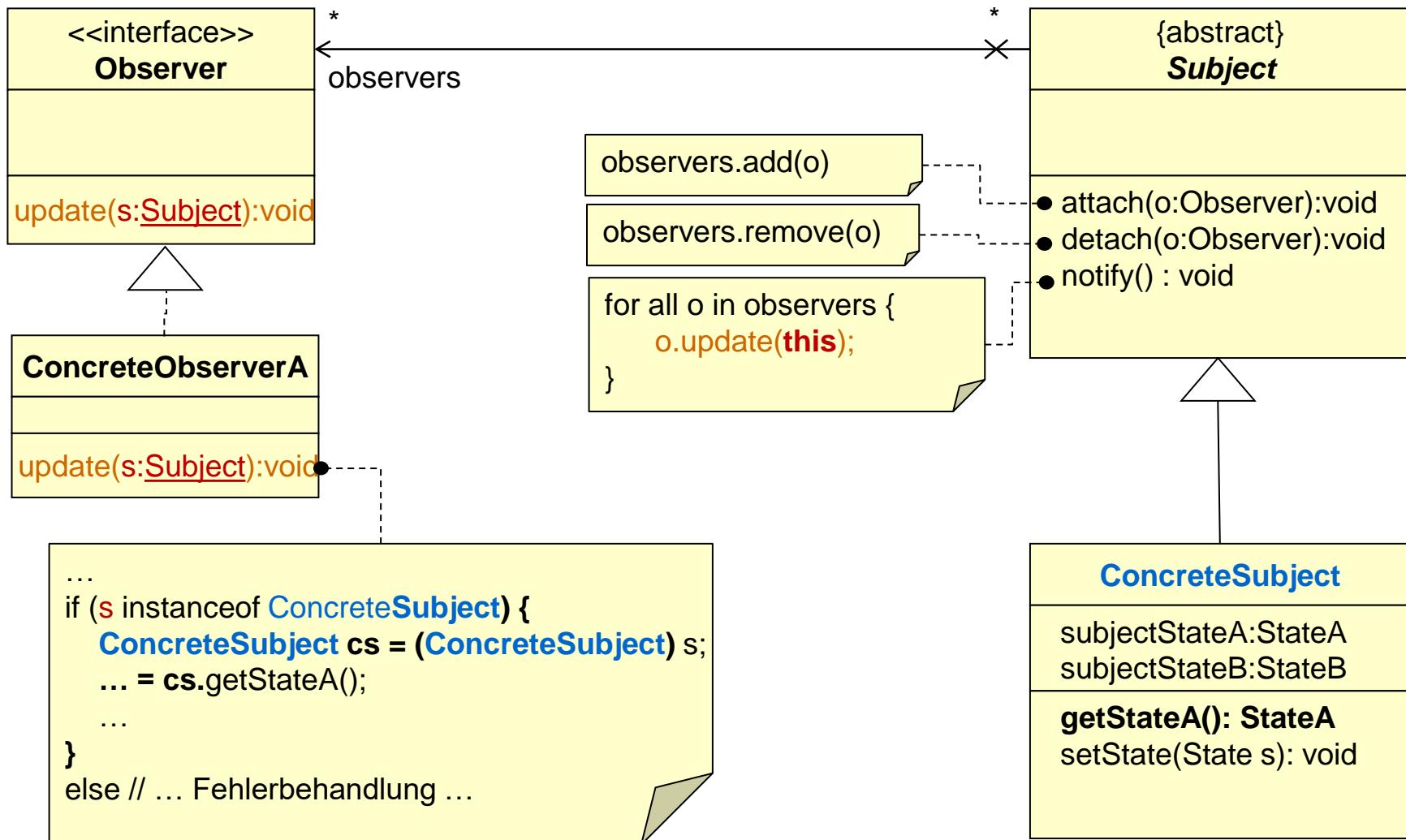


Das Observer Pattern: Implementierung

- Speicherung der Beziehung zwischen Subjekt und Beobachter
 - ◆ Instanzvariable im Subjekt oder ...
 - ◆ globale (Hash-)Tabelle
- Beobachtung mehrerer Subjekte
 - ◆ Beispiel Tabellenkalkulation: Jede Zelle muss evtl. viele Andere beobachten um sich bei deren Änderung neu zu berechnen.
 - ◆ Realisierung: Die „update()“-Methode muss einen Parameter haben, der das Subjekt angibt, das sich gerade geändert hat.
 - ◆ Querbezug zu Pull-Modell: Der Parameter kann für pull-Rückfragen an das Modell genutzt werden (keine feste Speicherung / Assoziation nötig)
 - ◆ Problem: Welchen Typ hat der Parameter der update-Methode?
 - ⇒ „Object“: Zu unspezifisch, damit kann man nicht viel anfangen
 - ⇒ „Subject“: Zu unspezifisch, damit kann man nicht viel anfangen
 - ⇒ „Concrete Subject“: Zu spezifisch für Observer anderer Concrete Subjects
 - ⇒ Häufiger Kompromiss in der Praxis: Statischer Parametertyp „**Subject**“ und Cast auf „**ConcreteSubject**“ im Rumpf der Methode aus „**ConcreteObserver**“



Das Observer Pattern: Struktur-Variante (Verbindung von Push und Pull)

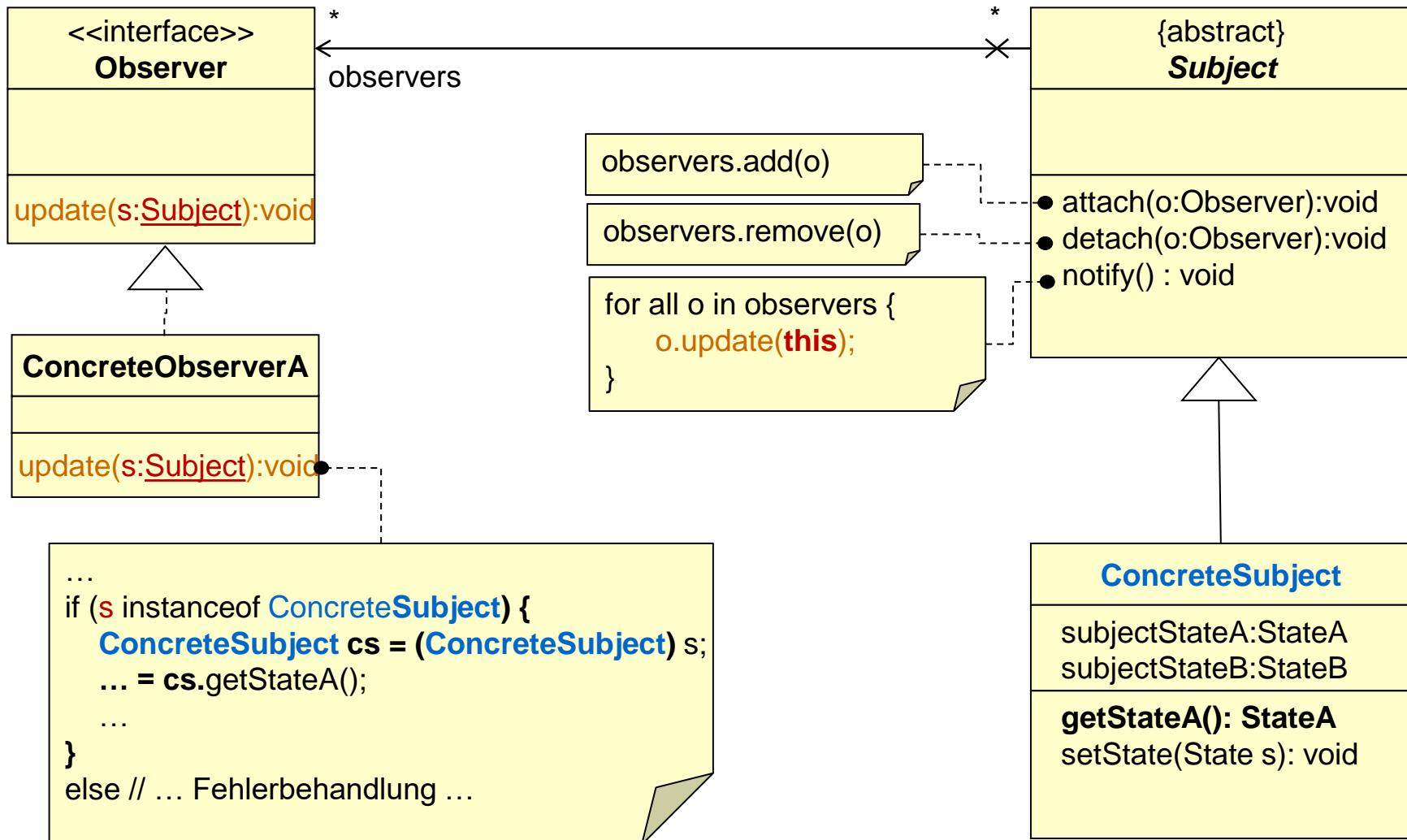


Das Observer Pattern: Implementierung

Vermeidung irrelevanter Notifikationen durch Differenzierung von Ereignissen (Events)

- Bisher: Notifikation bei *jeder* Änderung
- Alternative: Beobachter-Registrierung und Notifikation nur für spezielle Ereignisse
 - ◆ Realisierung: Differenzierung von `attach()`, `detach()`, `update()` und `notify()` in jeweils ereignisspezifische Varianten
 - ◆ Vorteile:
 - ⇒ Notifikation nur für relevante Ereignisse → höhere Effizienz
 - ⇒ Weniger „Rückfragen“ pro Ereignis → höhere Effizienz
 - ◆ Nachteil: Mehr Programmieraufwand, wenn man sich für viele Ereignistypen interessiert
 - ⇒ Aber: Werkzeugunterstützung möglich

Das Observer Pattern: Struktur-Variante (Verbindung von Push und Pull mit Events)



Das Observer Patterns: Implementierung

- Wer ruft notify() auf?
 - ◆ a) "setState()"-Methode des Subjekts:
 - ⇒ + Klienten können Aufruf von "notify()" nicht vergessen.
 - ⇒ – Aufeinanderfolgende Aufrufe von "setState()" führen zu evtl. überflüssigen Aktualisierungen.
 - ◆ b) Klienten:
 - ⇒ + Mehrere Änderungen können akkumuliert werden.
 - ⇒ – Klienten vergessen möglicherweise Aufruf von "notify()".
- ChangeManager
 - ◆ Verwaltet Beziehungen zwischen Subjekt und Beobachter.
(Speicherung in Subjekt und Beobachter kann entfallen.)
 - ◆ Definiert die Aktualisierungsstrategie
 - ◆ Benachrichtigt alle Beobachter. Verzögerte Benachrichtigung möglich
 - ⇒ Insbesondere wenn mehrere Subjekte verändert werden müssen, bevor die Aktualisierungen der Beobachter Sinn macht

Das Observer Pattern: Implementierung

- Konsistenz-Problem

- ◆ Zustand eines Subjekts muss vor Aufruf von "notify()" konsistent sein.
- ◆ Vorsicht bei Vererbung bei Aufruf jeglicher geerbter Methoden die möglicherweise „notify()“ -aufrufen :

```
public class MySubject extends SubjectSuperclass {  
    public void doSomething() {  
        super.doSomething(); // ruft "notify()" auf  
        this.modifyMyState(); // zu spät!  
    }  
}
```

- Lösung

- ◆ Dokumentation von „notify()“-Aufrufen erforderlich (Schnittstelle!)
- ◆ Besser: In Oberklasse „Template-Method Pattern“ anwenden um sicherzustellen, dass „notify()“-Aufrufe immer am Schluss einer Methode stattfinden → s. nächste Folie.

Das Observer Pattern: Implementierung

Verwendung des „Template Method Pattern“

```
public class SubjectSuperclass {  
    ...  
    final public void doSomething {  
        this.doItReally; // Aufruf der Hook-Methode an der richtigen Stelle  
        this.notify(); // notify() immer am Schluß  
    }  
    protected void doItReally() { // tut was in Oberklasse zu tun ist  
    }  
}
```

Template Method

← Hook Method (= Default-Implementierung die in Unterklassen überschrieben werden soll).

```
public class MySubject extends SubjectSuperclass {  
  
    protected void doItReally {  
        super.doItReally();  
        ... // Hier steht was in der Unterklasse zusätzlich zu tun ist  
    }  
}
```

Template Method Pattern

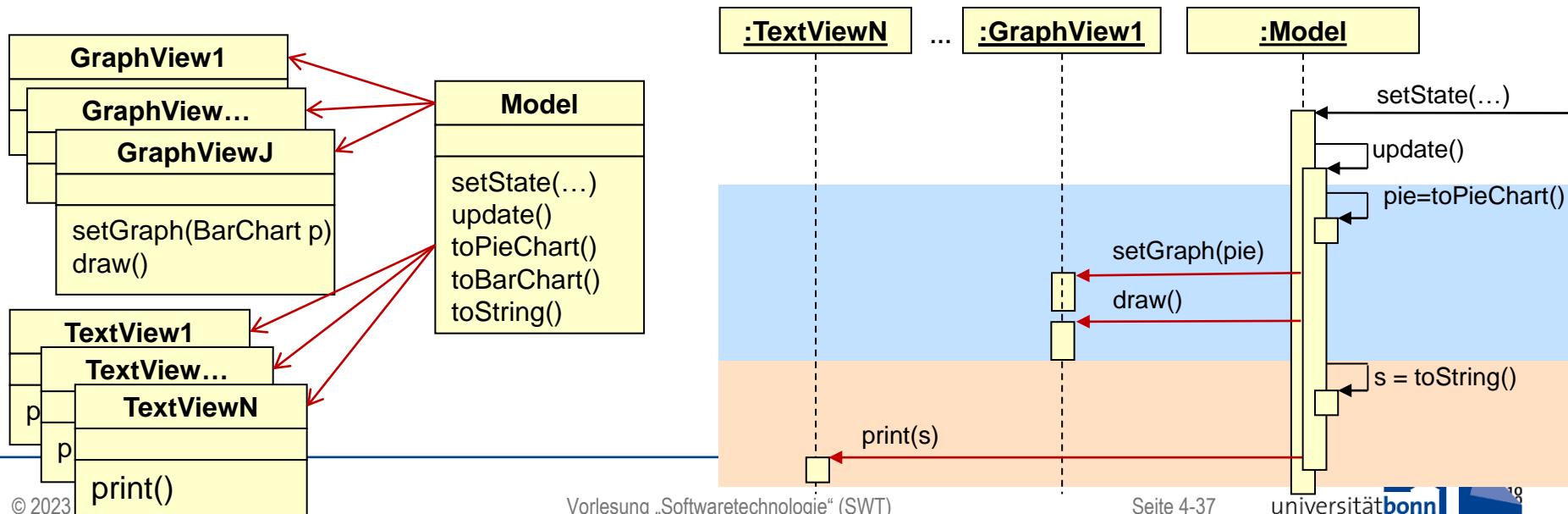
- Anwendbarkeit
 - ◆ Feste Reihenfolge von Aktionen muss garantiert werden
 - ◆ ... aber genaue Ausgestaltung der Aktionen soll anpassbar sein
- Beispiel
 - ◆ s. vorherige Folie
- Schema
 - ◆ Feste Reihenfolge → festgelegt durch nicht überschreibbare Methode („final“ Schlüsselwort in Java)
 - ⇒ Gelb unterlegte Methode auf vorheriger Seite
 - ◆ Anpassbare Aktionen → überschreibbare Methode die in der Template Method aufgerufen wird
 - ⇒ Blau unterlegte Methoden auf vorheriger Seite

Konsequenzen von Observer für Abhängigkeiten

Abhängigkeiten mit und ohne Observer

Ohne Observer: Abhängigkeit View ← Model

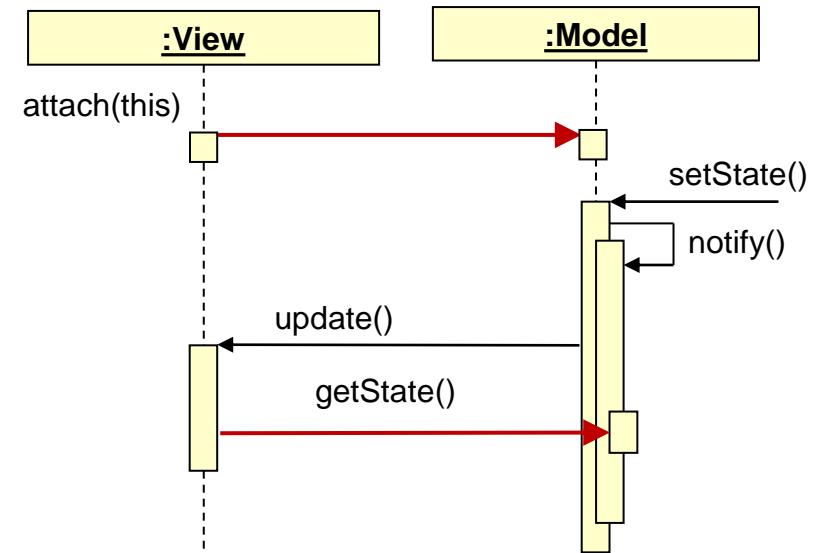
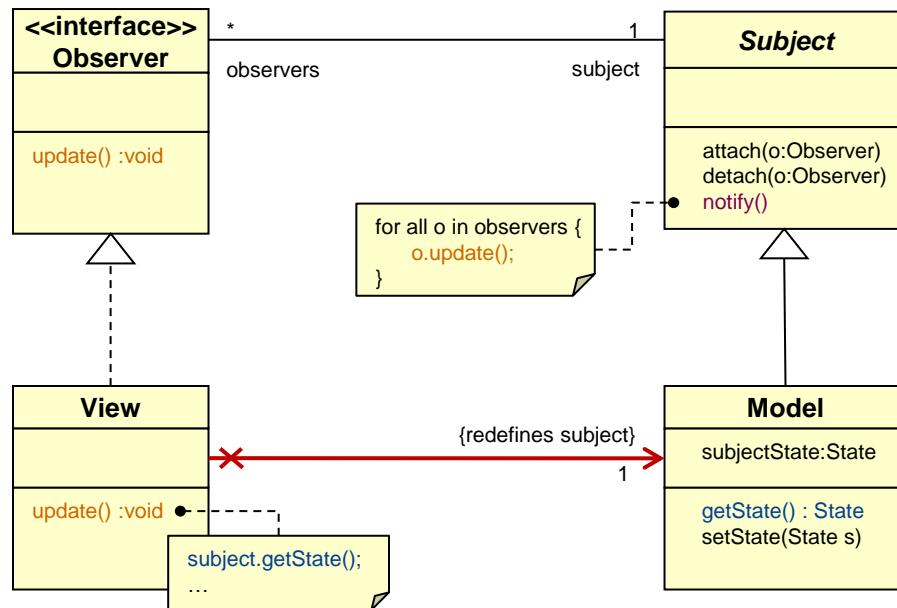
- Ein konkretes Model kennt das Interface eines jeden konkreten View und steuert was genau jeder der Views nach einem update tun soll:
 - ◆ myGraphView1.setGraph(this.toPieChart()); myGraphView1.draw();
 - ◆ myGraphView2.setGraph(this.toBarChart()); myGraphView2.draw();
 - ◆ ...
 - ◆ myTextViewN.print(myState.toString());



Abhängigkeiten mit und ohne Observer

Mit Observer: Abhängigkeit View → Model

- Ein konkretes Model kennt nur das abstrakte Observer-Interface
- Ein konkreter View kennt die zustandsabfragenden Methoden des konkreten Models
 - ◆ `model.getState1();`
 - ◆ `model.getState2();`



Nettoeffekt: Abhängigkeits- und Kontrollumkehrung

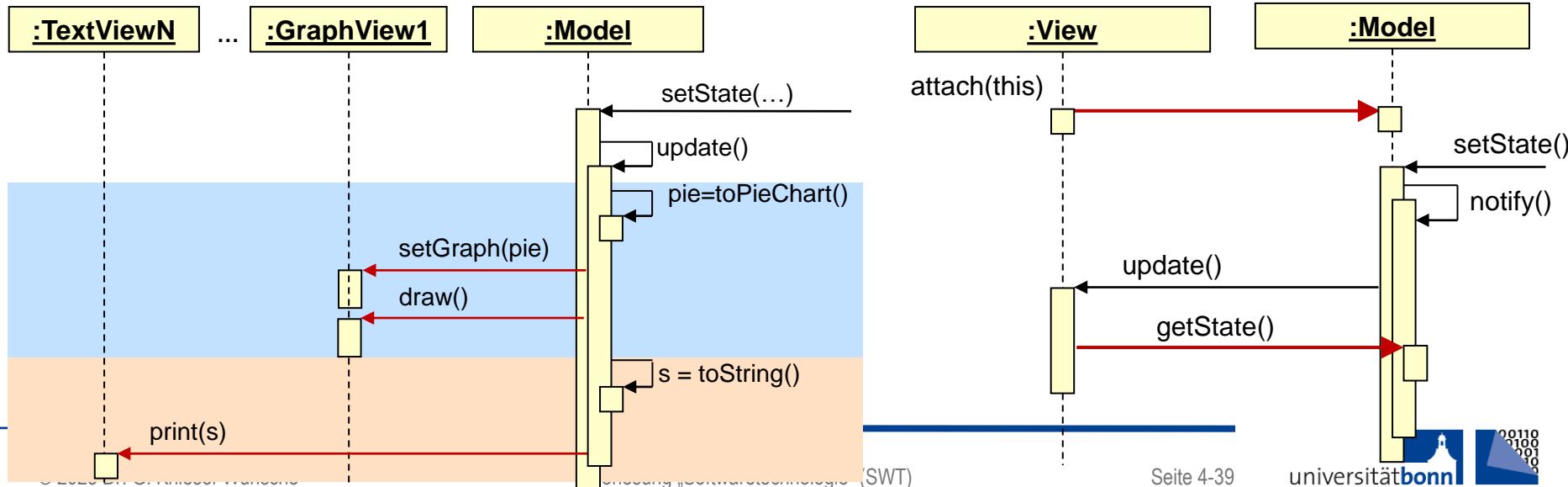
Abhängigkeitsumkehrung „Dependency Inversion“)

- Ohne Observer
 - ◆ Abhängigkeit Model → Views
- Mit Observer
 - ◆ Abhängigkeit Model ← Views

Kontrollumkehrung „Inversion of Control“)

- Ohne Observer
 - ◆ Model steuert alle updates
- Mit Observer
 - ◆ Jeder View steuert sein update

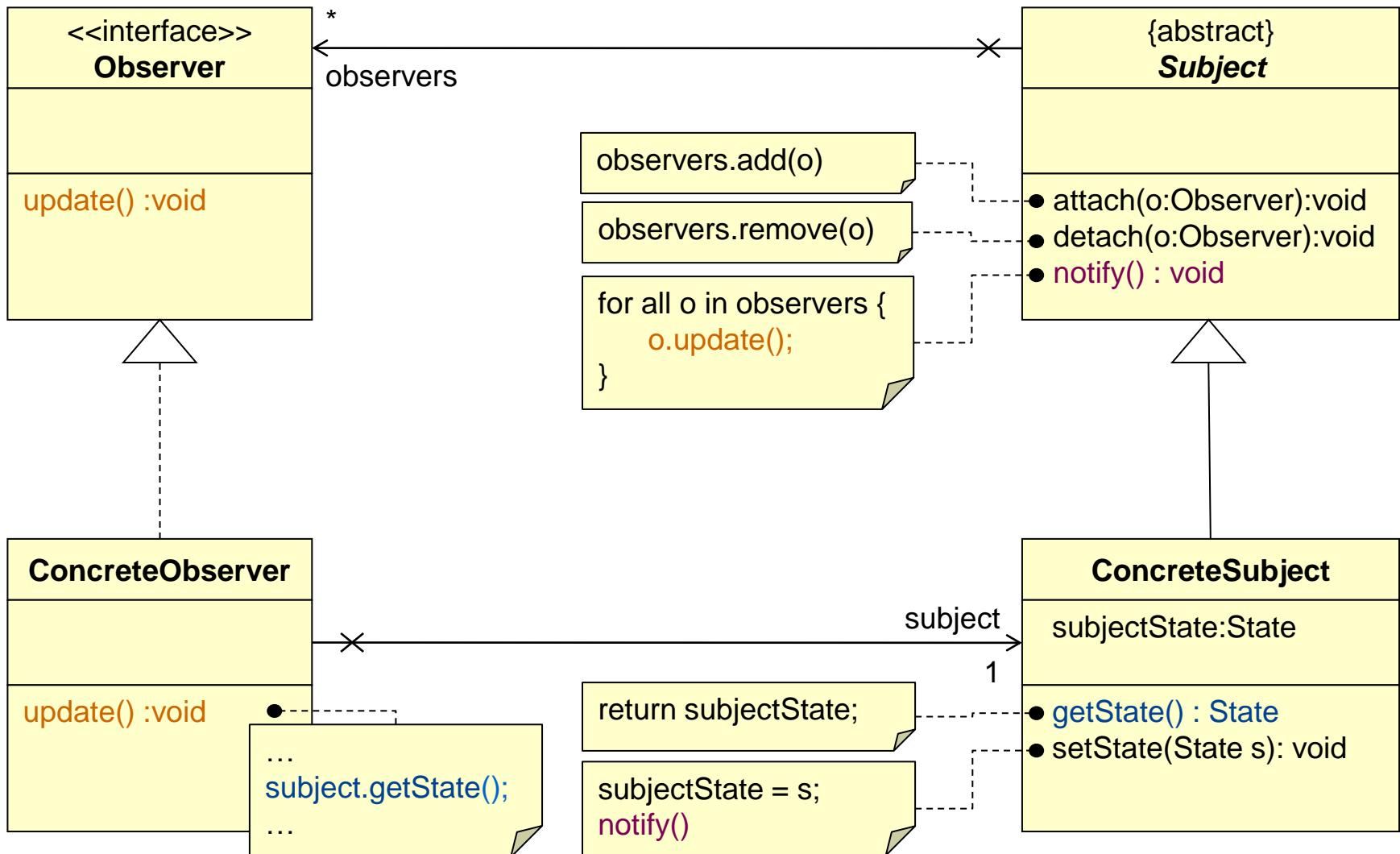
Vergleich Ablauf ohne / mit Observer



Anwendung von Patterns

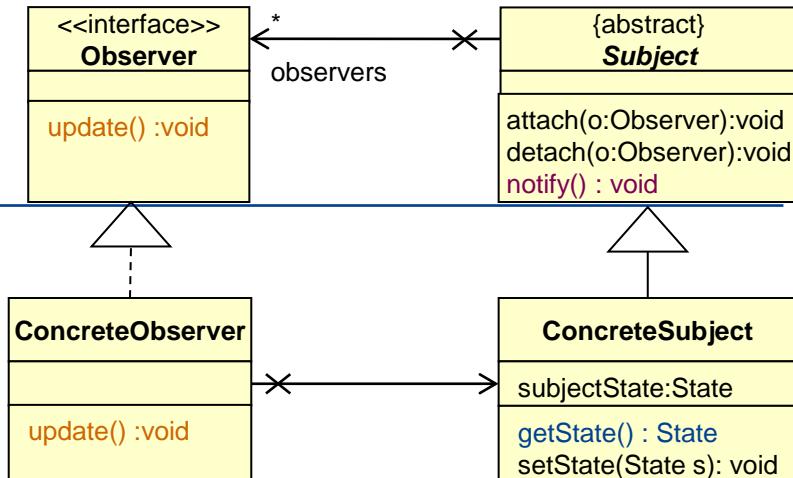
Hier eine typische Anwendung von Patterns zum Verbessern eines vorhandenen Designs
am Beispiel Observer

Das Observer Pattern: Struktur (N:1, Pull-Modell)

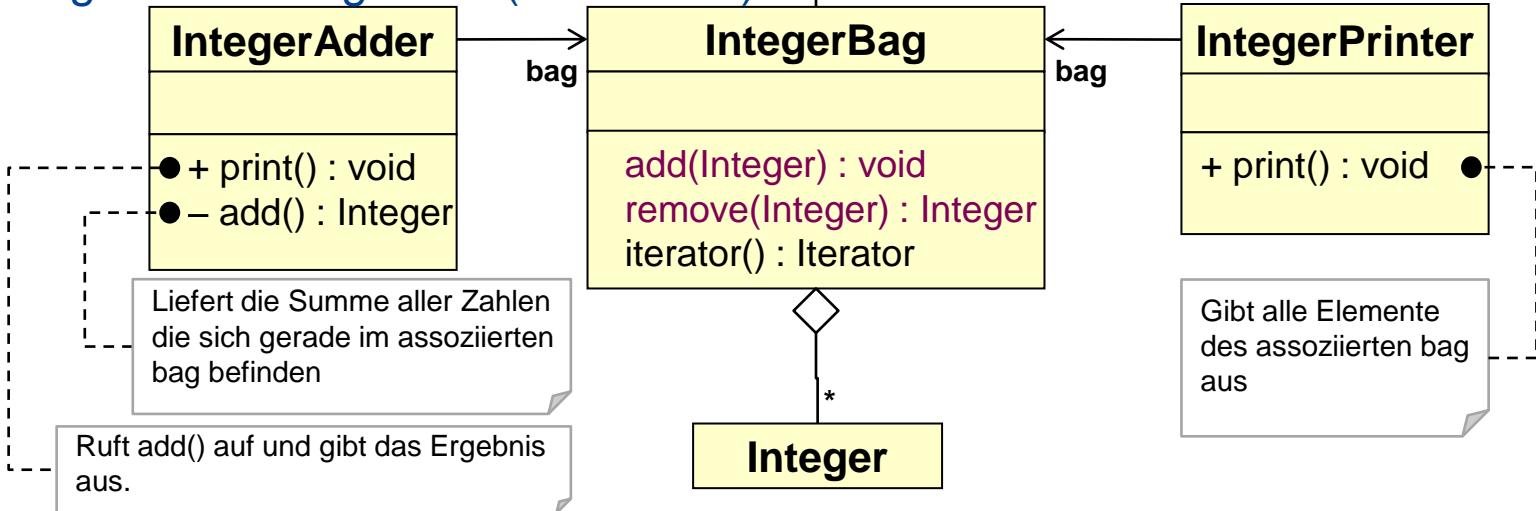


Patterns anwenden

- Anforderung: Jedes mal wenn sich der Inhalt eines IntegerBag ändert
 - ◆ gibt der assoziierte IntegerPrinter den aktuellen Inhalt aus
 - ◆ gibt der assoziierte IntegerAdder die aktuelle Summe aus
 - ◆ ... ohne dass der IntegerBag eine Abhängigkeit zu IntegerAdder oder IntegerPrinter hat



- Gegebenes Programm (Ausschnitt):

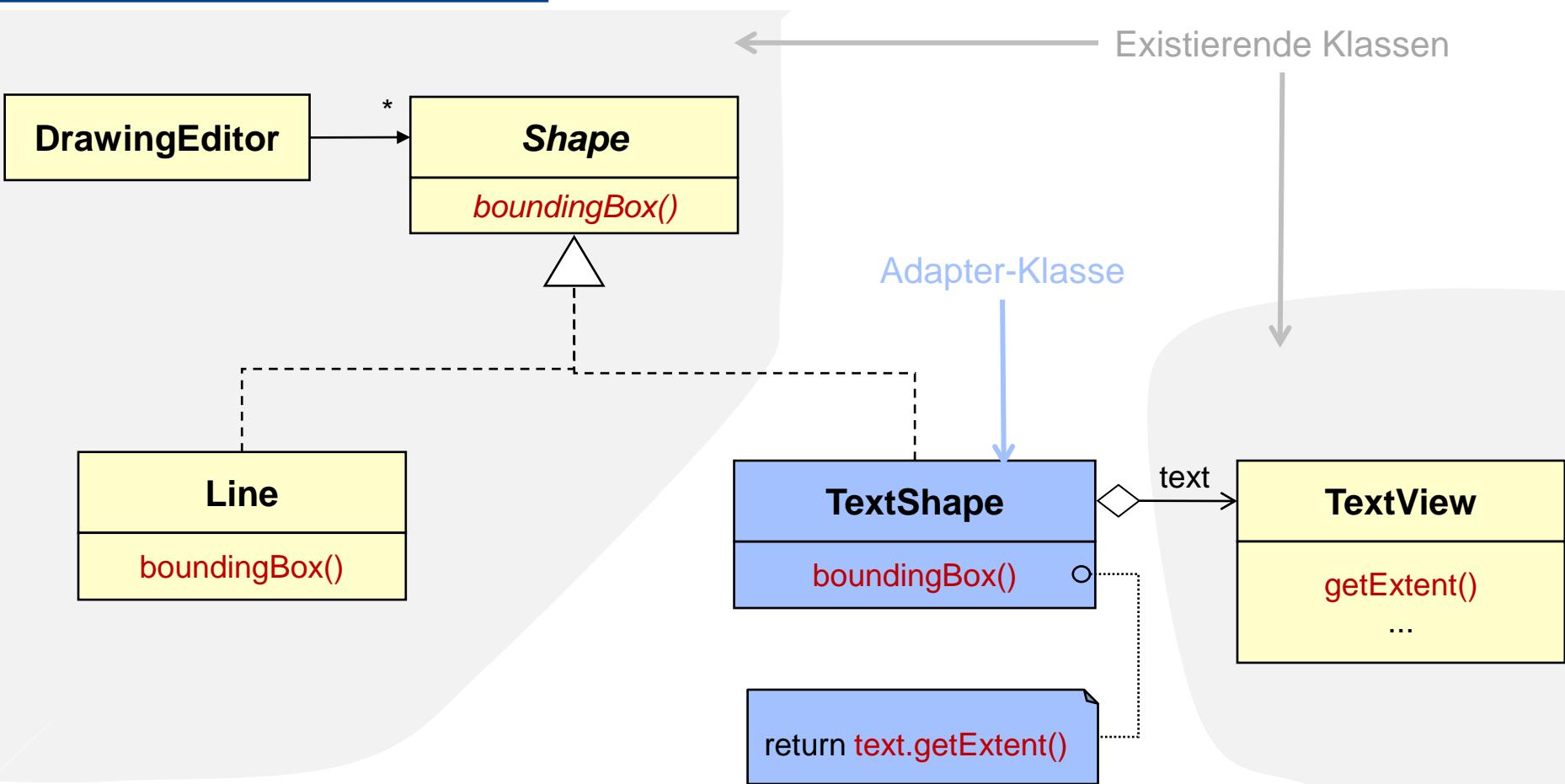


Das Adapter Pattern

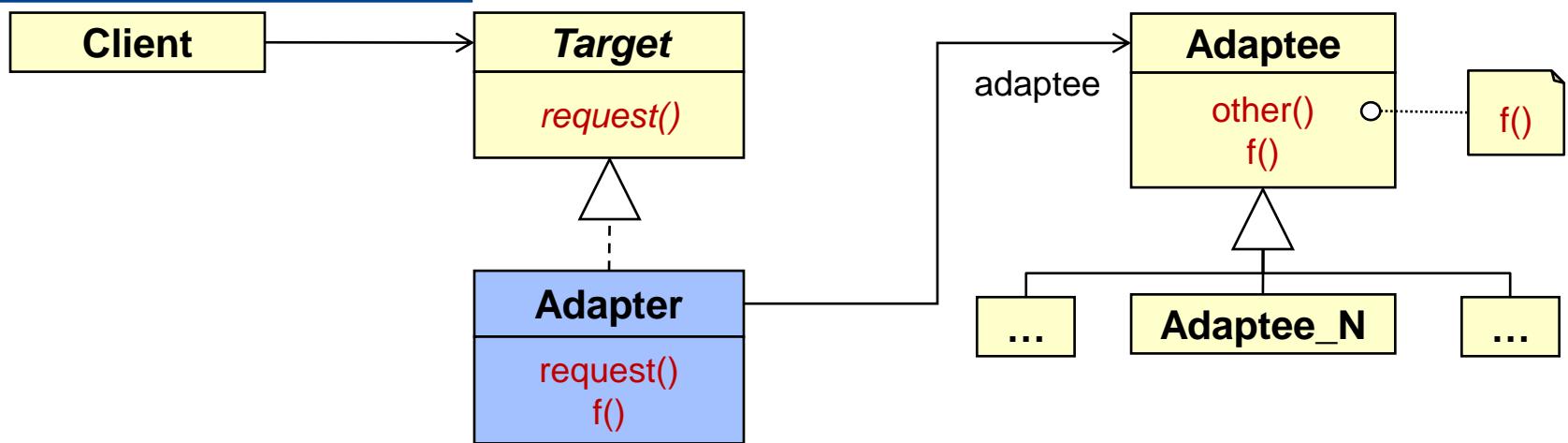
Adapter Pattern (auch: Wrapper)

- Absicht
 - ◆ Schnittstelle existierender Klasse an Bedarf existierender Clients anpassen
- Motivation
 - ◆ Graphik-Editor bearbeitet Shapes
 - ⇒ Linien, Rechtecke, ...
 - ⇒ durch "BoundingBox" beschrieben
 - ◆ Textelemente sollen auch möglich sein
 - ⇒ Klasse TextView vorhanden
 - ⇒ bietet keine "BoundingBox"-Operation
 - ◆ Integration ohne
 - ⇒ Änderung der gesamten Shape-Hierarchie und ihrer Clients
 - ⇒ Änderung der TextView-Klasse
- Idee
 - ◆ Adapter-Klasse stellt Shape-Interface zur Verfügung
 - ◆ ... implementiert Shape-Interface anhand der verfügbaren TextView-Methoden

Adapter Pattern: Beispiel



Adapter Pattern (Objekt-Adapter): Schema

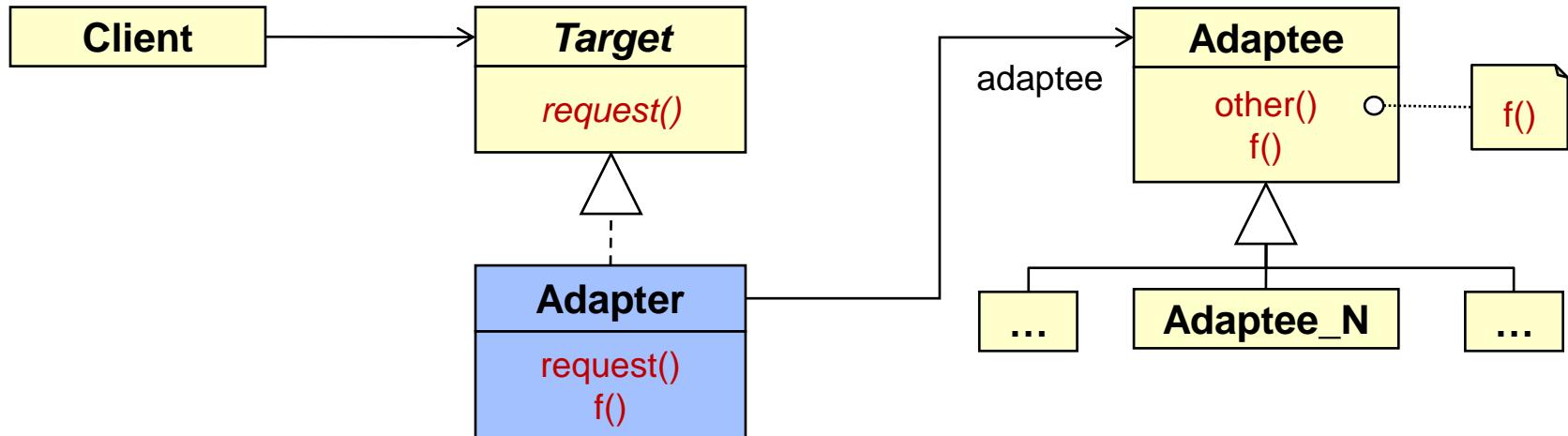


Adaptiert eine ganze Klassenhierarchie

- Beliebige Adapter-Subtypen können mit beliebigen Adaptee-Subtypen kombiniert werden
- Kombination wird erst zur Laufzeit, auf Objektebene, festgelegt

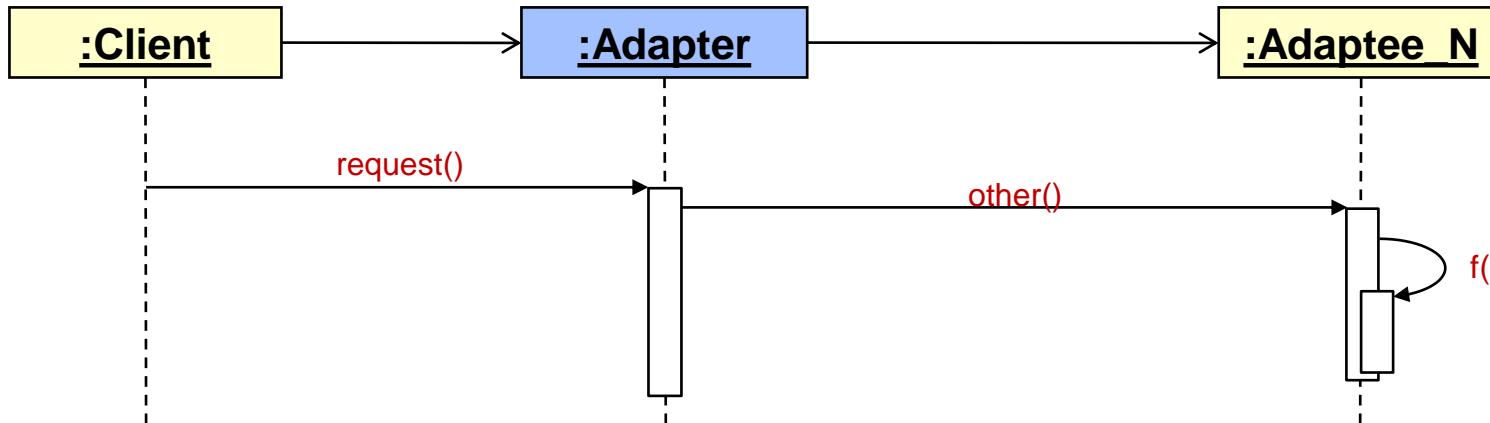


Adapter Pattern (Objekt-Adapter): Konsequenzen



Overriding nicht möglich

- Die **f()**-Definition aus Adapter wird nicht anstelle der **f()**-Definition aus Adaptee für den **f()**-Aufruf aus der Methode **other()** verwendet



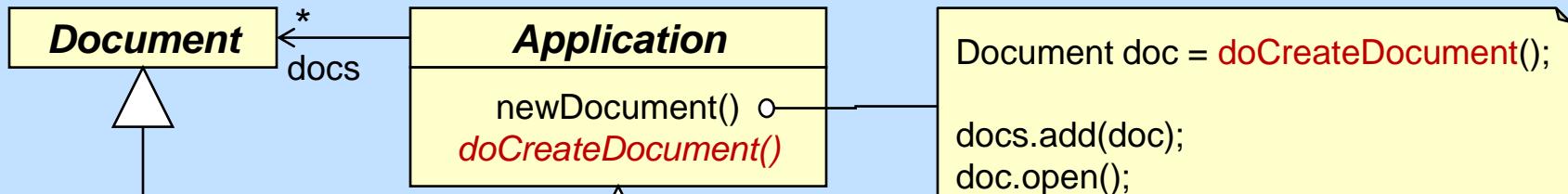
Das Factory Method Pattern

Factory Method

- Ziel
 - ◆ Entscheidung über konkreter Klasse neuer Objekte verzögern
- Motivation
 - ◆ Framework mit vordefinierten Klassen "Document" und "Application"
 - ◆ Template-Methode, für das Öffnen eines Dokuments:
 - ⇒ 1. Check ob Dokument-Art bekannt
 - ⇒ 2. Erzeugung einer Document-Instanz !?!
 - ◆ Erzeugung von Instanzen noch nicht bekannter Klassen!
- Idee
 - ◆ aktuelle Klasse entscheidet wann Objekte erzeugt werden
 - ⇒ Aufruf einer abstrakten Methode
 - ◆ Subklasse entscheidet was für Objekte erzeugt werden
 - ⇒ implementiert abstrakte Methode passend

Factory Method: Beispiel

Framework



MyDocument

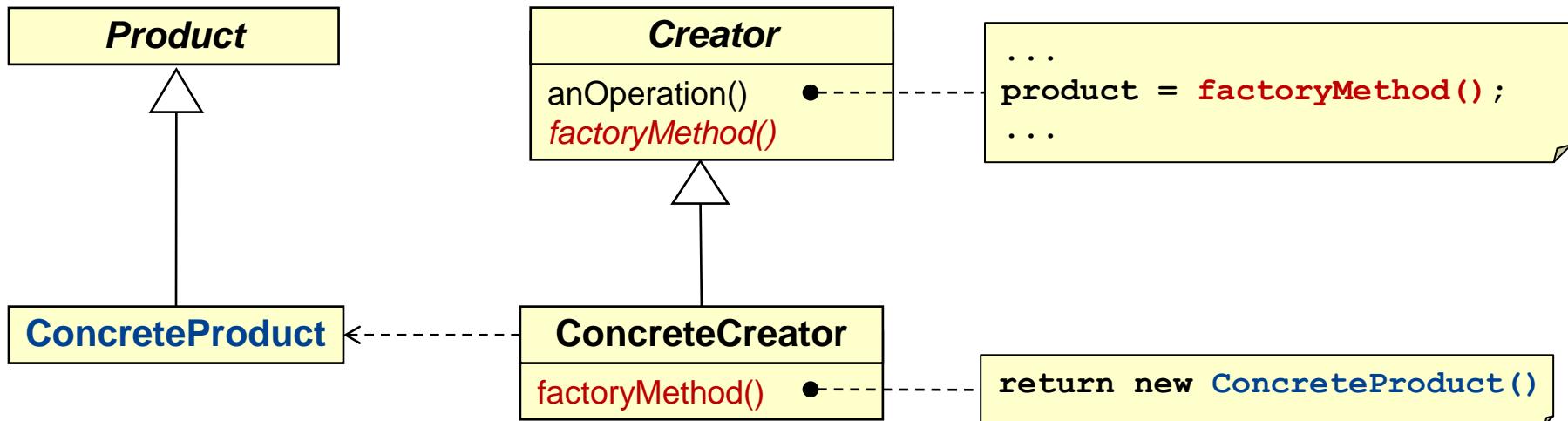
MyApplication

doCreateDocument()

return new **MyDocument()**

Applikation

Factory Method: Schema und Rollen

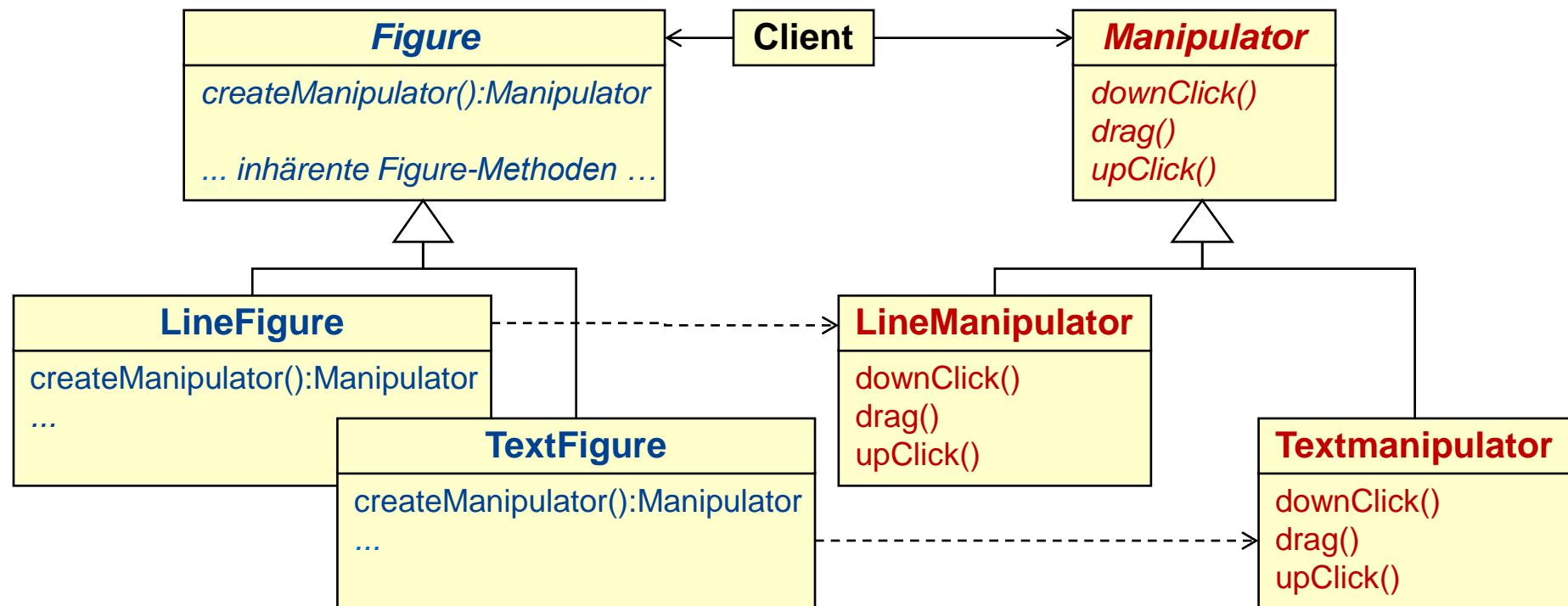


- Factory Method
 - ◆ kann abstrakt sein (s. Diagramm) oder Default-Implementierung haben
- Creator
 - ◆ Klasse im Framework, die die Factory Method (abstrakt oder als „hook“) definiert
- Concrete Creator
 - ◆ Klasse in der Applikation, die die Factory Method konkret (re)definiert

Factory Method: Anwendbarkeit

- Klasse neuer Objekte unbekannt
 - ◆ Eine Klasse (Creator) muss Objekte erzeugen, deren Art sie nicht vorhersehen kann
- Verzögerte Entscheidung über Art der erzeugten Objekte:
 - ◆ erst in Subklasse (ConcreteCreator)
 - ◆ erst beim Aufruf, durch Parametrisierung (siehe Folie zu Implementierungsvarianten)
- Mehrere Hilfsklassen
 - ◆ Wissen über konkrete Hilfsklassen an einer Stelle lokalisieren
 - ◆ Beispiel: Parallelle Hierarchien (nächste Folie)

Factory Method: Anwendbarkeit



- Verbindung paralleler Klassenhierarchien
 - ◆ Manipulators und Figures sind sauber getrennt („Separation of Concerns“)
 - ◆ Factory Method lokalisiert das Wissen über zusammengehörige Klassen
 - ◆ Restlicher Code der Figure-Hierarchie und Client-Code ist völlig unabhängig von *spezifischen* Manipulators (nur vom Manipulator-Interface)

Factory Method: Implementierung

1. Allgemein

- Fester "Produkt-Typ"

- ◆ Basisvariante: jede UnterkLASSE erzeugt festgelegten Produkt-Typ
- ◆ Vorteil: Inkrementelle Erweiterbarkeit
- ◆ Nachteil: Für jede neue Produkt-Art muss eine neue UnterkLASSE erzeugt werden

```
class Creator {  
    Product create(){  
        return new MyProduct();  
    }  
}
```

- Codierter "Produkt-Typ"

- ◆ Parameter codiert den "Produkt-Typ"
- ◆ Fallunterscheidung anhand Code
- ◆ Vorteil: nur eine Methode und Klasse nötig (keine Unterklassen)
- ◆ Nachteil: Änderung der Methode, für jeden neuen Produkttyp (keine inkrementelle Erweiterbarkeit)

```
class Creator {  
    Product create(int id) {  
        if (id==1) return new MyProduct1();  
        if (id==2) return new MyProduct2();  
        ...  
    }  
}
```

Frage: Geht auch beides?

Eine Methode/Klasse die für neue Produkttypen nicht geändert werden muss?

Factory Method: Implementierung

2. Sprachspezifisch

Antwort: Ja, aber nur in „reflektiven“ Sprachen.

- Instanz der Klasse „Class“ als "Produkt-Typ"

- ◆ „Produkt-Typ“ ist ein (Klassen-) Objekt, kein primitiver Datentyp
- ◆ Generische Instanz-Erzeugung durch Nachricht an das Klassenobjekt
- ◆ Generische Lösung aber
- ◆ ... kein statischer Typ-Check
- ◆ ... nur in reflektiven Sprachen (Smalltalk, Java, ...)

```
class Creator {  
    Object create(Class prodType) {  
        return prodType.newInstance();  
    }  
}
```

Reflektiver Aufruf des parameterlosen Default-Konstruktors in Java

Zur Erinnerung

Eine Sprache ist reflektiv, wenn die eigenen Sprachkonzepte mit den Mitteln der Sprache dargestellt werden. Z. B. sind in Java Klassen, Methoden und Felder als Objekte dargestellt, die zur Laufzeit Nachrichten empfangen können. Im obigen Beispiel übergeben wir als Parameter ein Objekt das eine Klasse darstellt und schicken ihm die Nachricht newInstance() um eine Instanz dieser Klasse zu erzeugen.

Factory Method: Konsequenzen

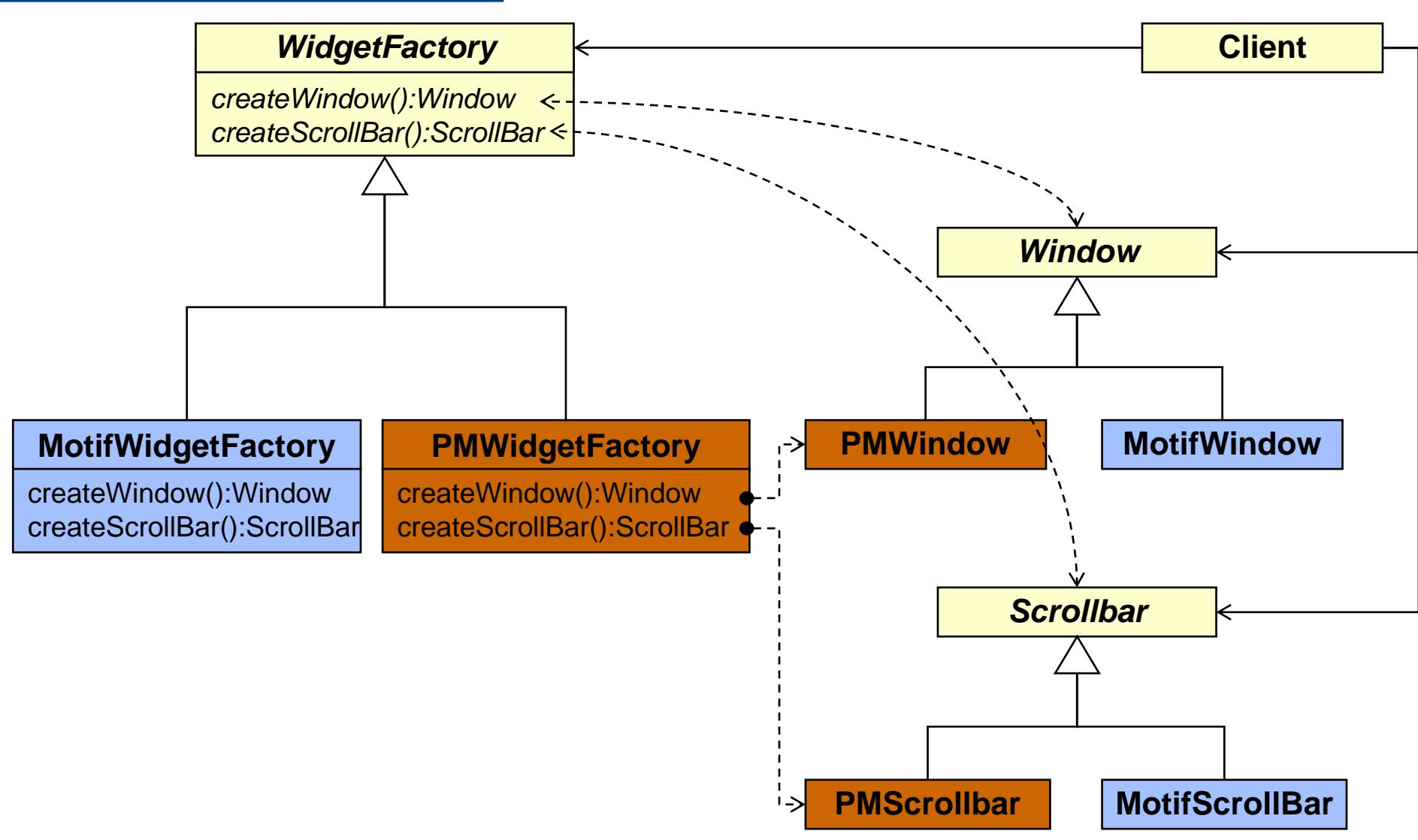
- Code wird abstrakter / wiederverwendbarer
 - ◆ keine festen Abhängigkeiten von spezifischen Klassen
- Verbindung paralleler Klassenhierarchien
 - ◆ Factory Method lokalisiert das Wissen über Zusammengehörigkeiten
- Evtl. künstliche Subklassen nur zwecks Objekterzeugung
 - ◆ Um in der Unterkasse die Factory-Methode überschreiben zu können

Das Abstract Factory Pattern

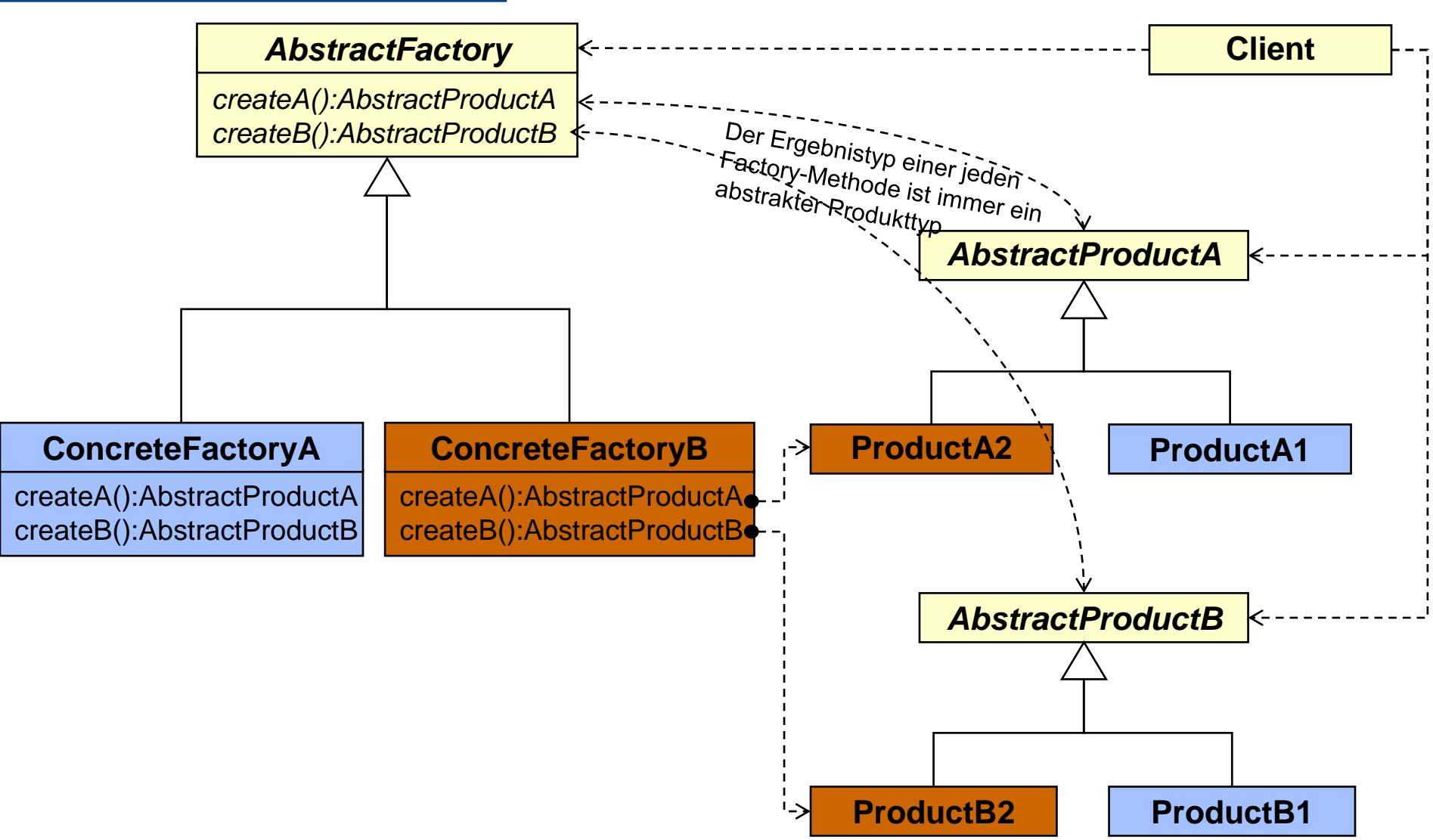
Abstract Factory

- Ziel
 - ◆ zusammengehörige Objekte verwandter Typen erzeugen
 - ◆ ... ohne deren Klassenzugehörigkeit fest zu codieren
- Motivation
 - ◆ GUI-Toolkit für mehrere Plattformen
 - ◆ Anwendungsklassen nicht von plattformspezifischen Widgets abhängig machen
 - ◆ Trotzdem soll die Anwendung
 - ⇒ alle Widgets konsistent zu einem "look and feel" erzeugen können
 - ⇒ "look and feel" umschalten können

Abstract Factory: Beispiel



Abstract Factory: Schema



Abstract Factory: Implementierung

- ConcreteFactories sind Singletons
- Produkt-Erzeugung via Factory-Methods
 - ◆ entsprechende Implementierungsvarianten (siehe Factory Method)
 - ⇒ fester Produkt-Typ (eine Methode für jedes Produkt)
 - ⇒ Codierter "Produkt-Typ" (mit Code parametrisierte Methode für alle Produkte)
 - ⇒ Klassen-Objekt als "Produkt-Typ" (mit Klassenobjekt parametrisierte Methode)

Abstract Factory: Implementierung

- Produktfamilie = Subklasse
 - ◆ Vorteil
 - ⇒ Konsistenz der Produkte
 - ◆ Nachteil
 - ⇒ neue Produktfamilie erfordert neue Subklasse, auch bei geringen Unterschieden
- Produktfamilie = von Clients konfigurierte Datenstruktur (Array, Map, ...)
 - ◆ Idee
 - ⇒ Fester Index / Symbolische Konstante oder Key pro Produkt-Typ
 - ⇒ Objekterzeugung durch Cloning des entsprechenden Eintrags
 - ◆ Vorteil
 - ⇒ keine Subklassenbildung erforderlich
 - ⇒ Verhalten dynamisch änderbar (neues Objekt an Index i im Array eintragen)
 - ◆ Nachteil
 - ⇒ Verantwortlichkeit an Clients abgegeben
 - ⇒ konsistente Produktfamilien können nicht mehr garantiert werden

Abstract Factory: Konsequenzen

- Konsistente Benutzung von Produktfamilien
 - ◆ Keine Windows-Scrollbar in MacOS-Fenster
- Abschirmung von Implementierungs-Klassen
 - ◆ Namen von Implementierungsklassen erscheinen nicht im Code von Clients
 - ◆ Clients benutzen nur abstraktes Interface
- Leichte Austauschbarkeit von Produktfamilien
 - ◆ Name einer ConcreteFactory erscheint nur ein mal im Code
 - ⇒ bei ihrer Instantiierung
 - ◆ Leicht austauschbar gegen andere ConcreteFactory
 - ◆ Beispiel: Dynamisches Ändern des look-and-feel
 - ⇒ andere ConcreteFactory instantiiieren
 - ⇒ alle GUI-Objekte neu erzeugen
- Schwierige Erweiterung um neue Produktarten
 - ◆ Schnittstelle der AbstractFactory legt Produktarten fest
 - ◆ Neue Produktart → Änderung der gesamten Factory-Hierarchie

Abstract Factory: Anwendbarkeit

- System soll unabhängig sein von
 - ◆ Objekt-Erzeugung
 - ◆ Objekt-Komposition
 - ◆ Objekt-Darstellung
- System soll konfigurierbar sein
 - ◆ Auswahl aus verschiedenen Produkt-Familien
- Konsistente Produktfamilien
 - ◆ Nur Objekte der gleichen Familie "passen zusammen"
- Library mit Produktfamilie anbieten
 - ◆ Clients sollen nur Schnittstelle kennen
 - ◆ ... nicht die genauen Teilprodukt-Arten / -Implementierungen

Das Command Pattern

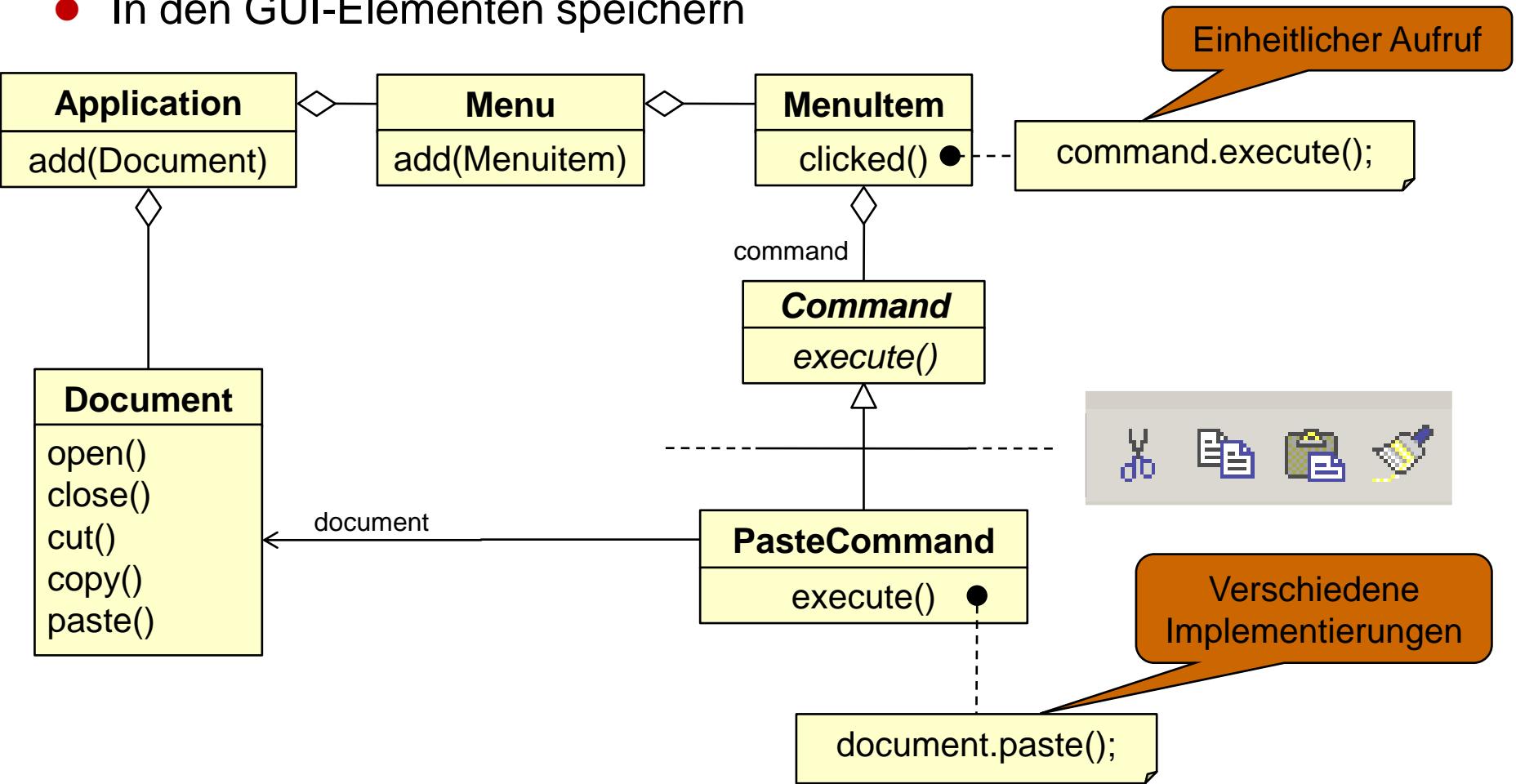
Das Command Pattern: Motivation

- Kontext
 - ◆ GUI-Toolkits mit Buttons, Menüs, etc.
- Forces
 - ◆ Wiederverwendung
 - ⇒ Über verschiedene GUI-Elemente ansprechbare Operationen nur ein mal programmieren
 - ◆ Dynamik
 - ⇒ dynamische Änderung der Aktionen von Menu-Einträgen
 - ⇒ kontext-sensitive Aktionen
 - ◆ Vielfalt trotz Einheitlichkeit
 - ⇒ Einheitlicher Code in allen GUI-Elementen
 - ⇒ .. trotzdem verschiedene Effekte

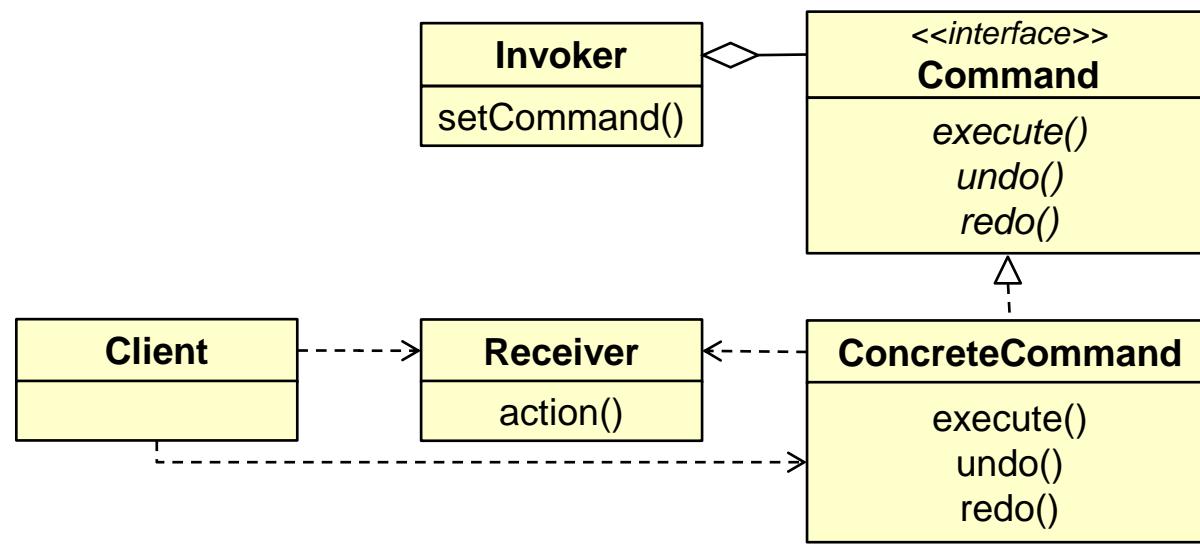


Das Command Pattern: Idee

- Operationen als Objekte mit Methode execute() darstellen
- In den GUI-Elementen speichern



Command Pattern: Schema und Rollen



- ◀ **Command Interface**
 - ◆ legt fest was Commands tun können
 - ◆ Mindestens Execute, optional auch Undo, Redo
- ◀ **ConcreteCommand**
 - ◆ Implementiert Interface
 - ◆ „Controllers“ sind oft „ConcreteCommands“

- **Execute-Methode**
 - ◆ Ausführen von Kommandos
- **Undo-Methode**
 - ◆ Rückgängig machen von Kommandos
- **Redo-Methode**
 - ◆ Rückgängig gemachtes Kommando wieder ausführen

Das Command Pattern: Konsequenzen

- Entkopplung
 - ◆ von Aufruf einer Operation und Spezifikation einer Operation.
- Kommandos als Objekte
 - ◆ Command-Objekte können wie andere Objekte auch manipuliert und erweitert werden.
- Komposition
 - ◆ Folgen von Command-Objekte können zu weiteren Command-Objekten zusammengefasst werden → Makros!
- Erweiterbarkeit
 - ◆ Neue Command-Objekte können leicht hinzugefügt werden, da keine Klassen geändert werden müssen.

Das Command Pattern: Anwendbarkeit

- Operationen als Parameter
- Variable Aktionen
 - ◆ referenziertes Command-Objekt austauschen
- Zeitliche Trennung
 - ◆ Befehl zur Ausführung, Protokollierung, Ausführung
- Speicherung und Protokollierung von Operationen
 - ◆ History-Liste
 - ◆ Serialisierung
- "Undo" von Operationen
 - ◆ Command-Objekte enthalten neben execute() auch unexecute()
 - ◆ werden in einer History-Liste gespeichert
- Recover-Funktion nach Systemabstürzen
 - ◆ History-Liste wird gespeichert
 - ◆ Zustand eines Systems ab letzter Speicherung wiederstellbar
- Unterstützung von Transaktionen
 - ◆ Transaktionen können sowohl einfache ("primitive"), als auch komplexe Command-Objekte sein.

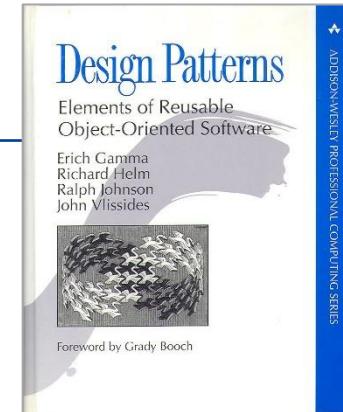
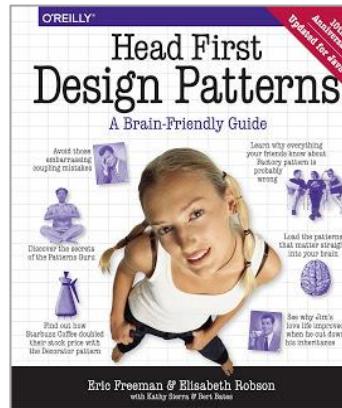
Implementierung des Command Patterns

- Unterschiedliche Grade von Intelligenz
 - ◆ Command-Objekte können "nur" Verbindung zwischen Sender und Empfänger sein, oder aber alles selbstständig erledigen.
- Unterstützung von "undo"
 - ◆ Semantik
 - ⇒ Undo (unexecute()) und redo (execute()) müssen den exakt gegenteiligen Effekt haben!
 - ◆ Problem: In den wenigsten Fällen gibt es exact inverse Operationen
 - ⇒ Die Mathematik ist da eine Ausnahme...
 - ◆ Daher Zusatzinfos erforderlich
 - ⇒ Damit ein Command-Objekt "undo" unterstützen kann, müssen evtl. zusätzliche Informationen gespeichert werden.
 - ⇒ Typischerweise: Kopien des alten Zustands der Objekte die wiederhergestellt werden sollen.
 - ◆ Tiefes Klonen
 - ⇒ Es reicht nicht eine Referenz auf die Objekte zu setzen!
 - ⇒ Man muss das Objekt "tief" klonen, um sicherzustellen dass sein Zustand nicht verändert wird.
 - ◆ History-Liste
 - ⇒ Für mehrere Levels von undo/redo

Literatur

“Gang of Four”

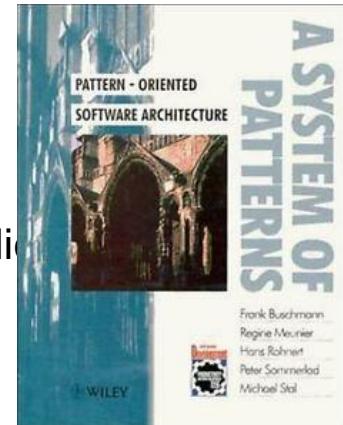
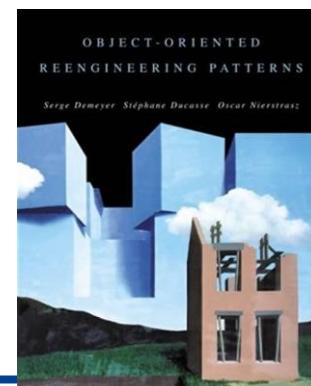
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:
“[Design Patterns – Elements of Reusable Object-Oriented Software](#)”
Addison-Wesley, 1995



- Eric Freeman, Elisabeth Robson, Bert Bates,
Kathy Sierra: “[Head First – Design Patterns](#)”
O'Reilly, 2014



- Frank Buschmann, Regine Meunier, Hans Rohnbert, Peter Sommerlad, Michael Stal:
“[Pattern-Oriented Software Architecture – A System of Patterns](#)”
John Wiley & Sons, 1996



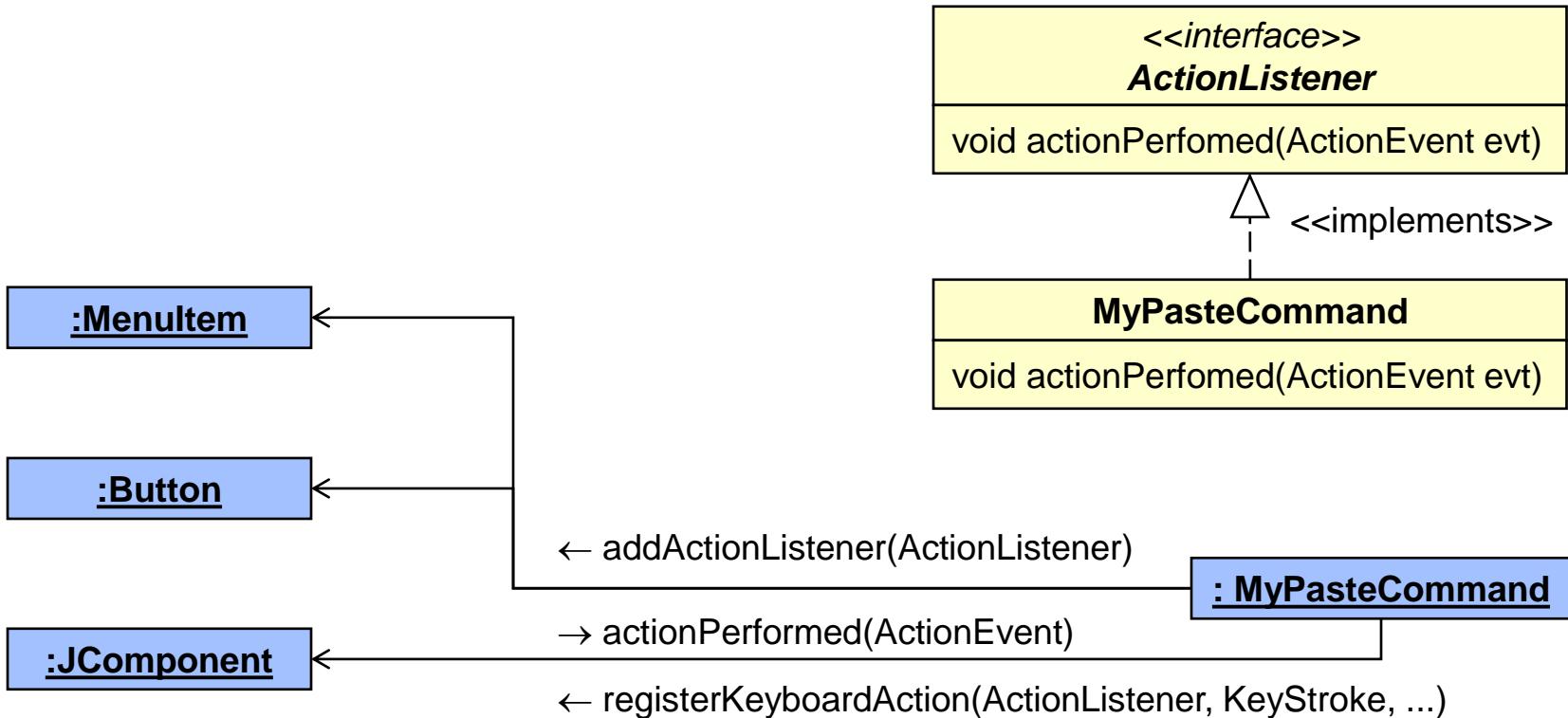
- Serge Demeyer, Stephan Ducasse, Oscar Nierstrasz:
“[Object Oriented Reengineering Patterns](#)”,
Morgan Kaufman, 2002

Anhang

Wichtige Patterns die wir aus Zeitgründen nicht behandeln
Nicht prüfungs- aber sehr praxisrelevant

Verbindung von Command und Observer im JDK

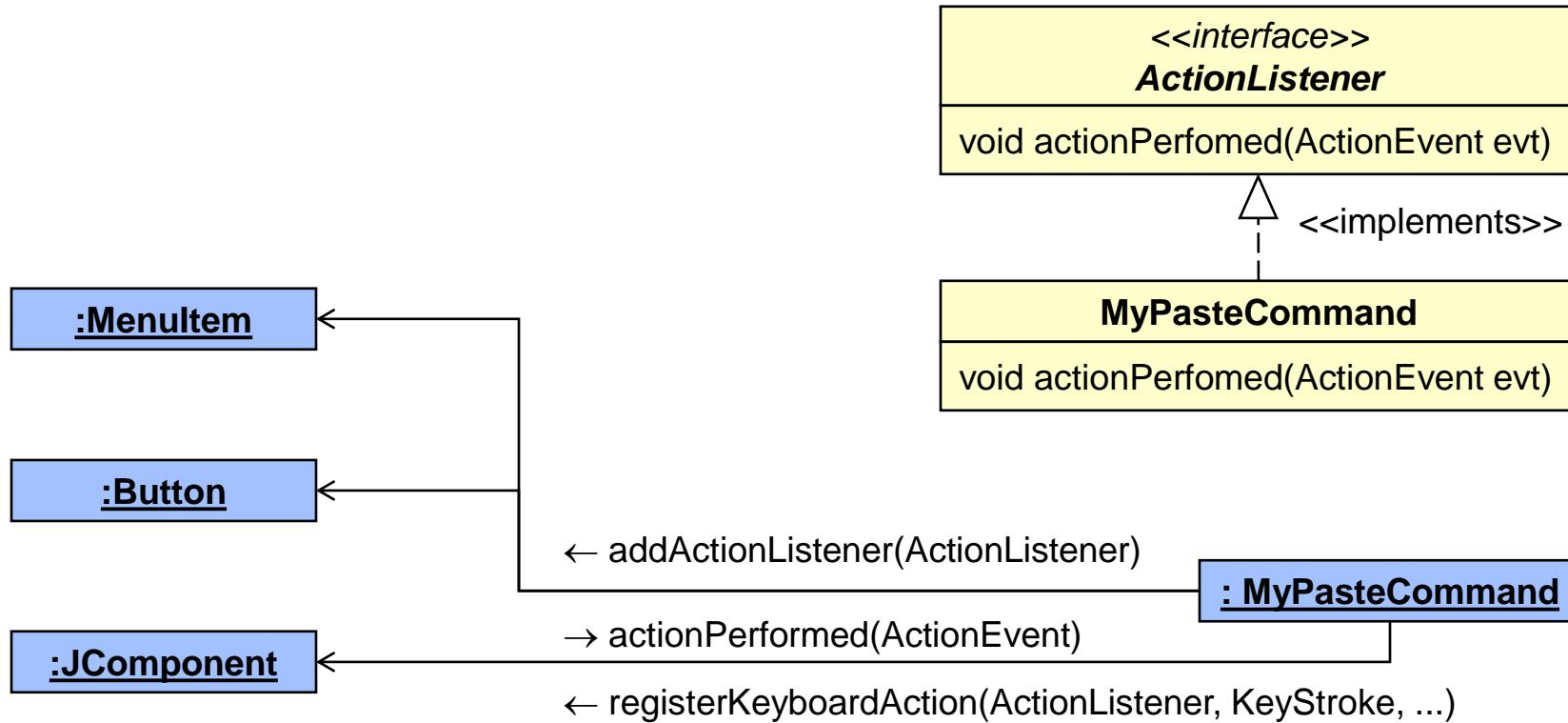
Verbindung von Observer und Command in Java



- Observer (= `ActionListener`)
 - ◆ Buttons, Menue-Einträge und Tasten generieren "ActionEvents"
 - ◆ Interface "ActionListener" ist vordefiniert
 - ◆ "ActionListener" implementieren
 - ◆ ... und Instanzen davon bei Buttons, MenuItems, etc registrieren

- Command & Observer
 - ◆ gleiche Methode (z.B. `actionPerformed`) spielt die Rolle der `run()` Methode eines Commands und die Rolle der `update()` Methode eines Observers
 - ◆ implementierende Klasse (z.B. `MyPasteCommand`) ist gleichzeitig ein Command und ein Observer

Verbindung von Observer und Command in Java verbindet auch Push und Pull



● Push Observer

- ◆ Subject „pusht“ eine „Event“-Instanz
- ◆ Der „Event“-Typ kann selbst definiert werden
- ◆ In dieser Instanz können somit weitere Informationen mit „gepusht“ werden (als Werte von Instanzvariablen)

● Pull Observer

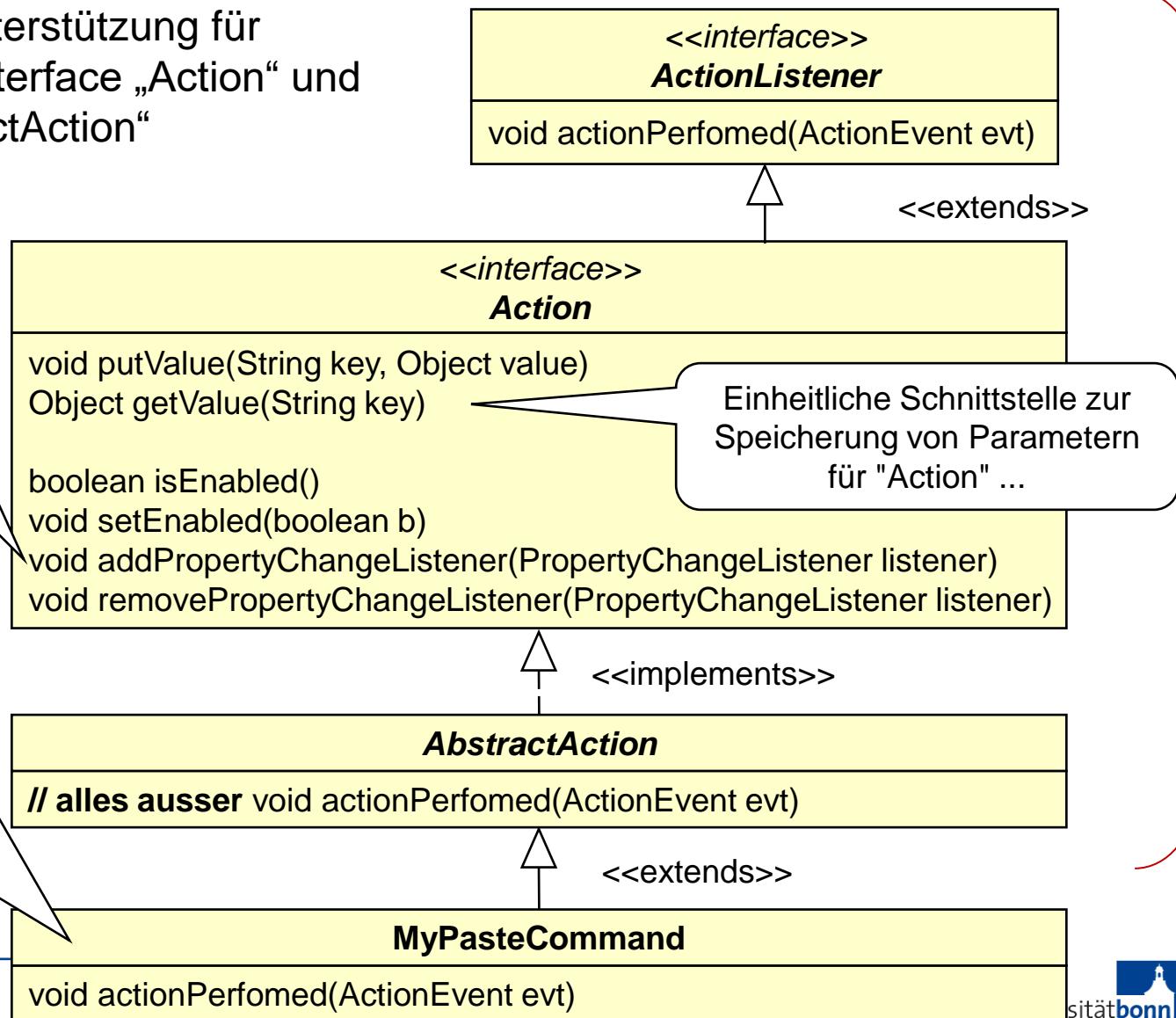
- ◆ Die als Parameter übergebene „Event“-Instanz enthält immer eine Rückreferenz auf die Quelle des Events
- ◆ Somit ist es möglich von dort weitere Informationen anzufragen („pull“)

Verbindung von Observer und Command in Java (2)

- Zusätzliche Unterstützung für „Command“: Interface „Action“ und Klasse „AbstractAction“

Aktivierung / Deaktivierung von Menüitems denen diese "Action" zugeordnet ist

... Parameter können allerdings auch direkt als Instanz-Variablen realisiert werden.



Beispiel: Änderung der Hintergrundfarbe (1)

```
class ColorAction extends AbstractAction
{ public ColorAction(..., Color c, Component comp)
  { ...
    color = c;
    target = comp;
  }

  public void actionPerformed(ActionEvent evt)
  { target.setBackground(color);
    target.repaint();
  }

  private Component target;
  private Color color;
}
```

```
class ActionButton extends JButton
{ public ActionButton(Action a)
  { ...
    addActionListener(a);
  }
}
```

- ColorAction
 - ◆ Ändern der Hintergrundfarbe einer GUI-Komponente
- ActionButton
 - ◆ Buttons die sofort bei Erzeugung mit einer Action verknüpft werden

Beispiel: Änderung der Hintergrundfarbe (2)

- Nutzung der ColorAction
 - ◆ Erzeugung einer Instanz
 - ◆ Registrierung

```
class SeparateGUIFrame extends JFrame
{ public SeparateGUIFrame()
{   ...
    JPanel panel = new JPanel();

    Action blueAction = new ColorAction("Blue", . . ., . . ., panel);

    panel.add(new ActionButton(blueAction));

    panel.registerKeyboardAction(blueAction, . . .);

    JMenu menu = new JMenu("Color");
    menu.add(blueAction);
    ...

}
}
```

Fazit

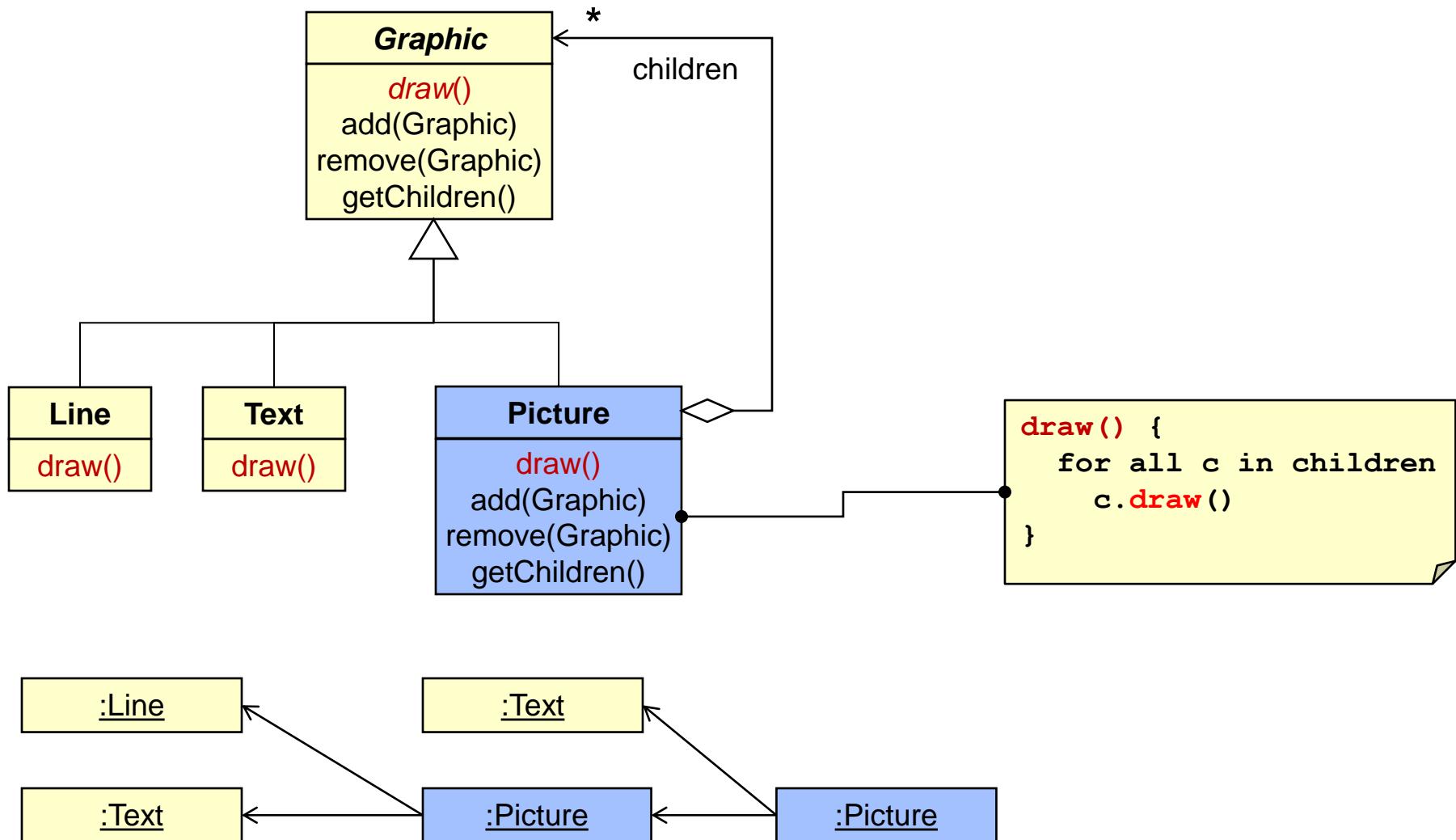
- ✓ Wiederverwendung
 - ◆ gleiche **Action** für Menu, Button, Key
- ✓ Elegante Verbindung von Push- und Pull-Observer und Command
 - ◆ in Commands sind **ActionListener** von Buttons, Menus, etc.
 - ⇒ Einheitlicher Aufruf via `actionPerformed(ActionEvent evt)`
 - ◆ Buttons und Menus sind **PropertyChangeListener** von Commands
 - ⇒ Aktivierung / Deaktivierung
 - ◆ Push- durch Daten im übergebenen Event-Objekt
 - ◆ Pull möglich durch Rückreferenz auf das Event-Auslösende Objekt im Event-Objekt
- Dynamik
 - ◆ Wie ändert man die aktuelle Aktion?
 - ◆ ... konsistent für alle betroffenen Menultems, Buttons, etc.???
 - Strategy Pattern! (→ Kapitel „Objektentwurf“)

Das Composite Pattern

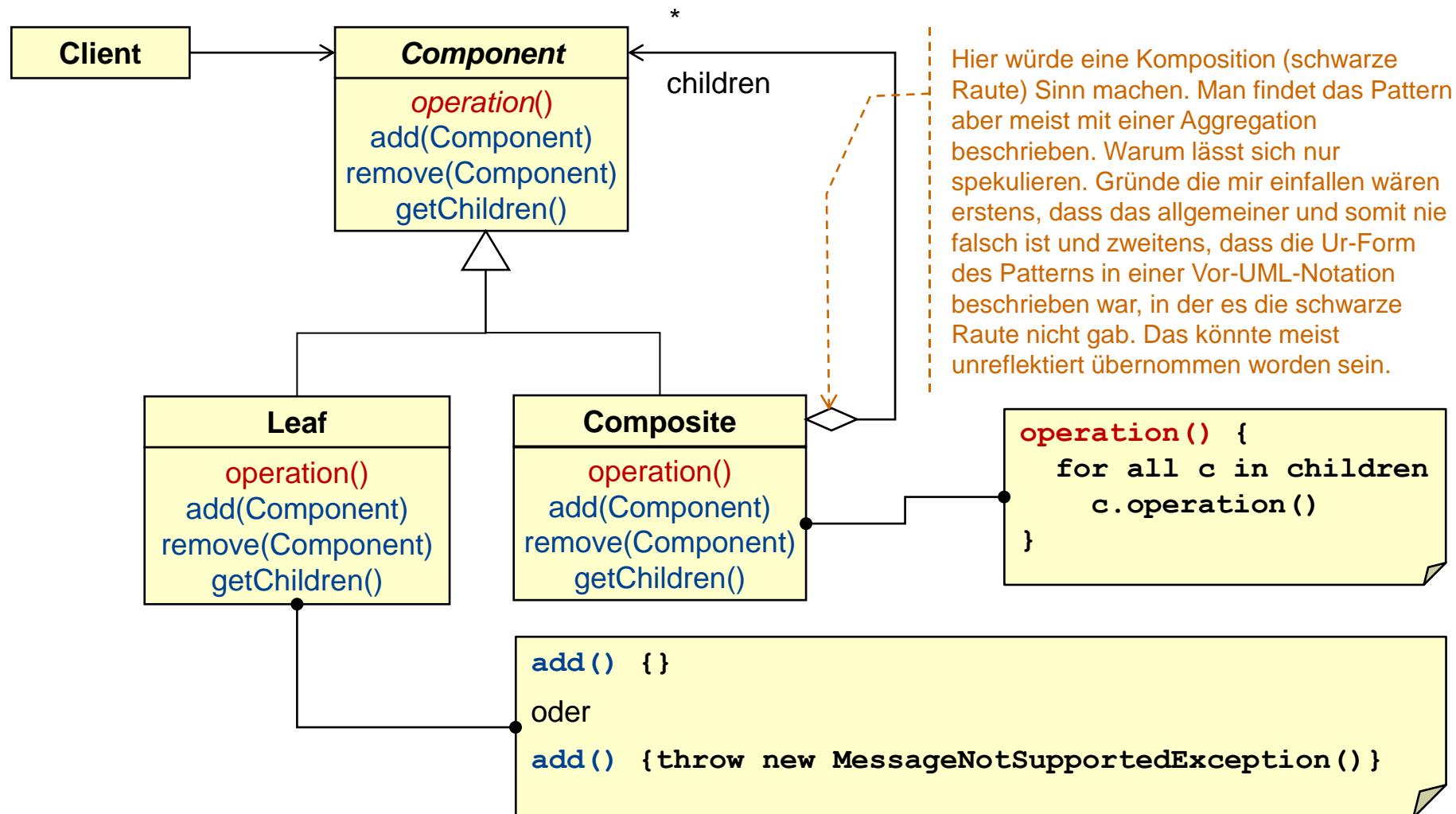
Composite Pattern

- Ziel
 - ◆ rekursive Aggregations-Strukturen darstellen ("ist Teil von")
 - ◆ Aggregat und Teile einheitlich behandeln
- Motivation
 - ◆ Gruppierung von Graphiken

Composite Pattern: Beispiel



Composite Pattern: Struktur



Composite Pattern: Verantwortlichkeiten

- Component (Graphic)

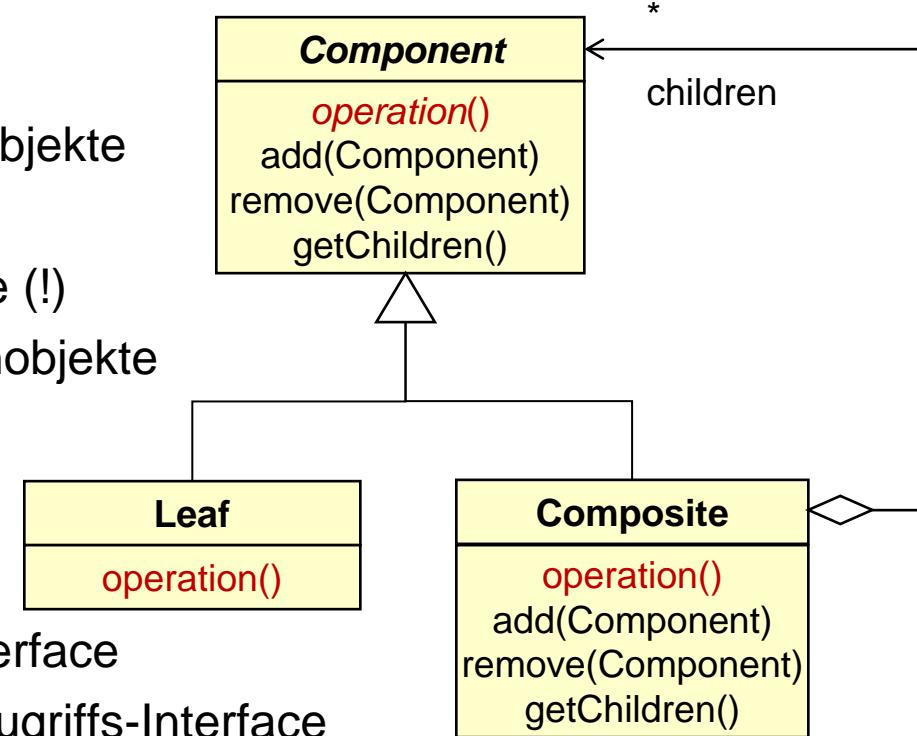
- ◆ gemeinsames Interface aller Teilobjekte
- ◆ default-Verhalten
- ◆ Interface für Zugriff auf Teilobjekte (!)
- ◆ evtl. Interface für Zugriff auf Elternobjekte

- Leaf (Rectangle, Line, Text)

- ◆ "primitive" Teil-Objekte
- ◆ implementieren gemeinsames Interface
- ◆ leere Implementierungen für Teilzugriffs-Interface

- Composite (Picture)

- ◆ speichert Teilobjekte
- ◆ implementiert gemeinsames Interface durch forwarding
- ◆ implementiert Teilzugriffs-Interface



Composite Pattern: Implementierung

- Wenn Composites wissen sollen wovon sie Teil sind

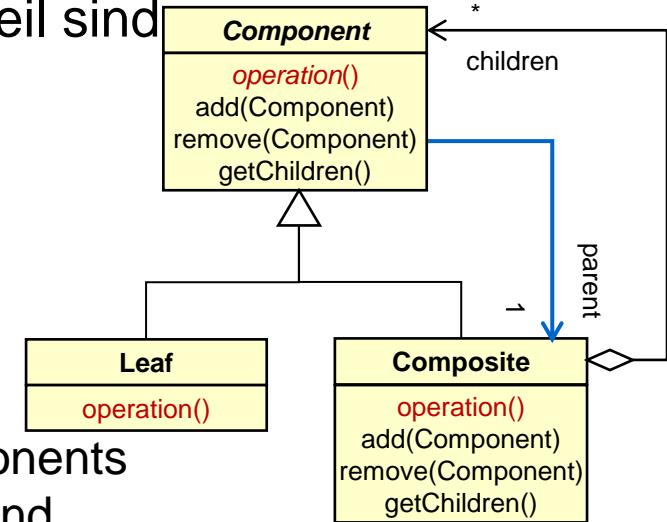
- ◆ Explizite Referenzen auf Composite in Component Klasse

- Wenn mehrere Composites gemeinsam nutzen sollen

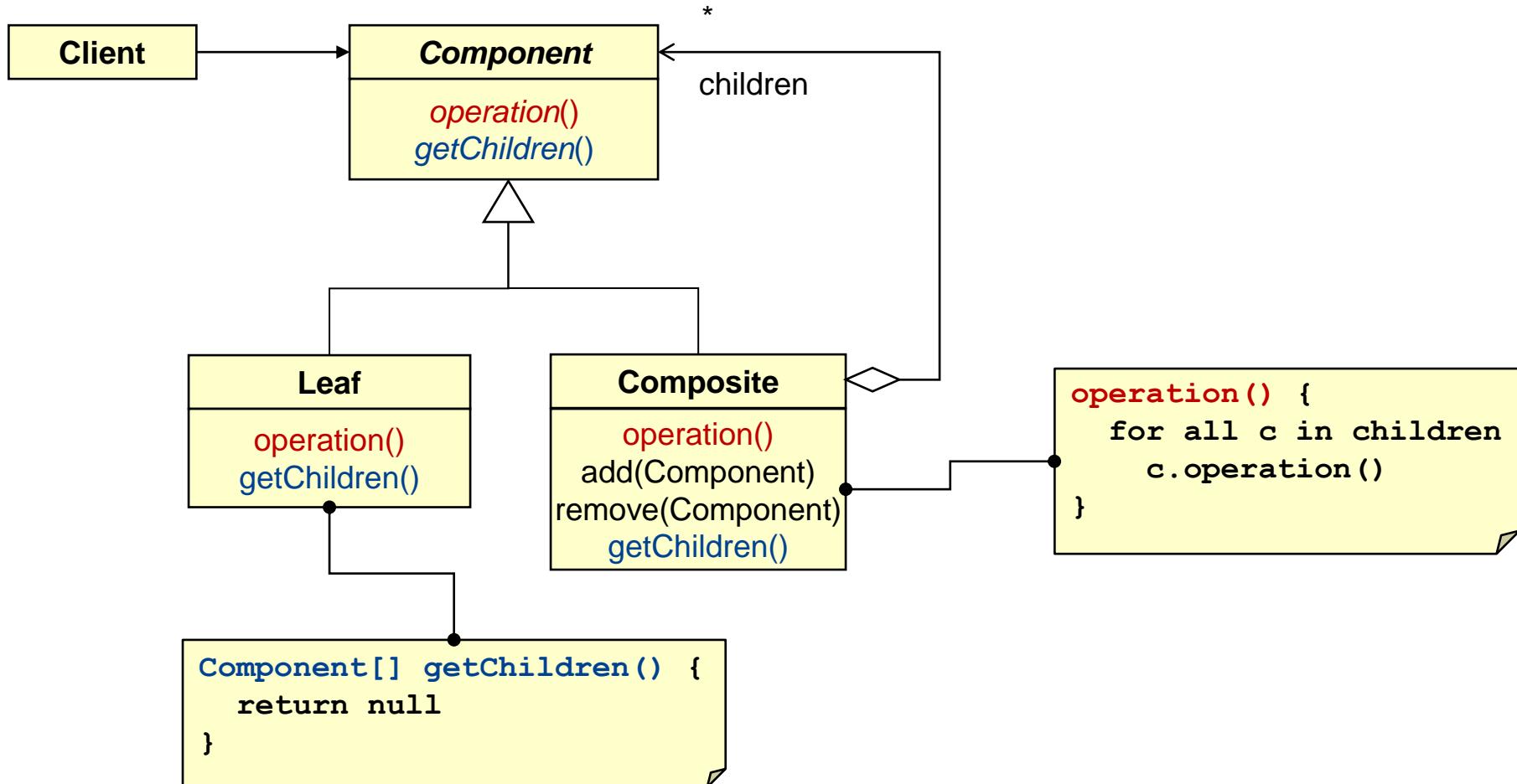
- ◆ Schließt explizite Referenz der Components auf Composite aus oder erfordert, dass Components wissen, dass sie Teile mehrerer Composites sind

- Component Interface

- ◆ "Sauberes Design": Verwaltungs-Operationen (add, remove) in Composite, da sie für Leafs nicht anwendbar sind.
 - ◆ Wunsch nach einheitlicher Behandlung aller Graphic-Objekte durch Clients
→ Verwaltungs-Operationen in Component mit default-Implementierung die nichts tut
→ Leaf-Klassen sind damit zufrieden, Composites müssen die Operationen passend implementieren.



Composite Pattern: Alternative Struktur (add / remove nicht in „Component“)



Composite Pattern: Konsequenzen

- Einheitliche Behandlung
 - ◆ Teile
 - ◆ Ganzes
- Einfache Clients
 - ◆ Dynamisches Binden statt Fallunterscheidungen
- Leichte Erweiterbarkeit
 - ◆ neue Leaf-Klassen
 - ◆ neue Composite-Klassen
 - ◆ ohne Client-Änderung
- Evtl. zu allgemein
 - ◆ Einschränkung der Komponenten-Typen schwer
 - ◆ „run-time type checks“ (instanceof)

Das Bridge Pattern

Bridge Pattern (auch: Handle / Body)

- Absicht

- ◆ Schnittstelle und Implementierung trennen
- ◆ ... unabhängig variieren

- Motivation

- ◆ portable "Window"-Abstraktion in GUI-Toolkit
- ◆ mehrere Variations-Dimensionen

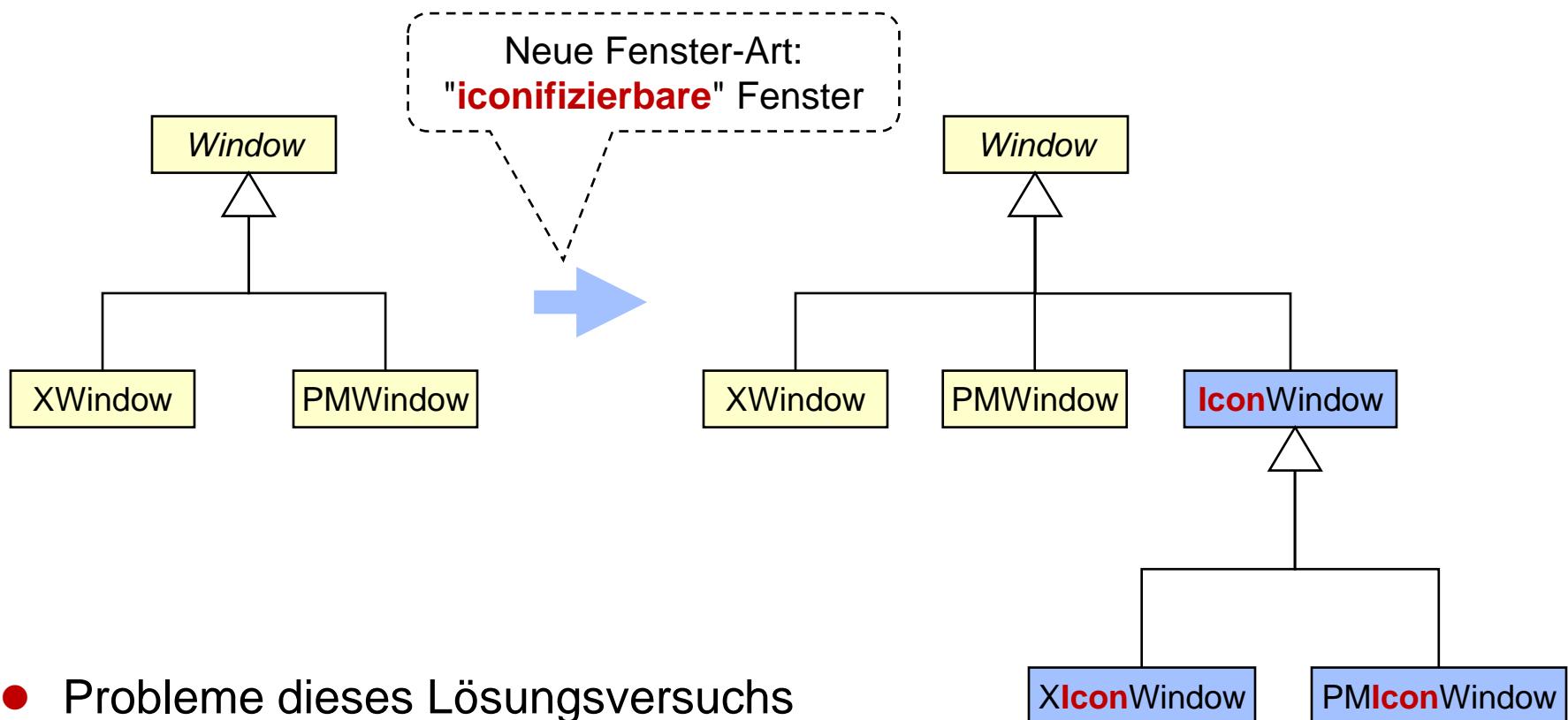
- ⇒ Fenster-Arten:

- normal / als Icon,
 - schließbar / nicht schließbar,
 - ...

- ⇒ Implementierungen:

- X-Windows,
 - IBM Presentation Manager,
 - MacOS,
 - Windows XYZ,
 - ...

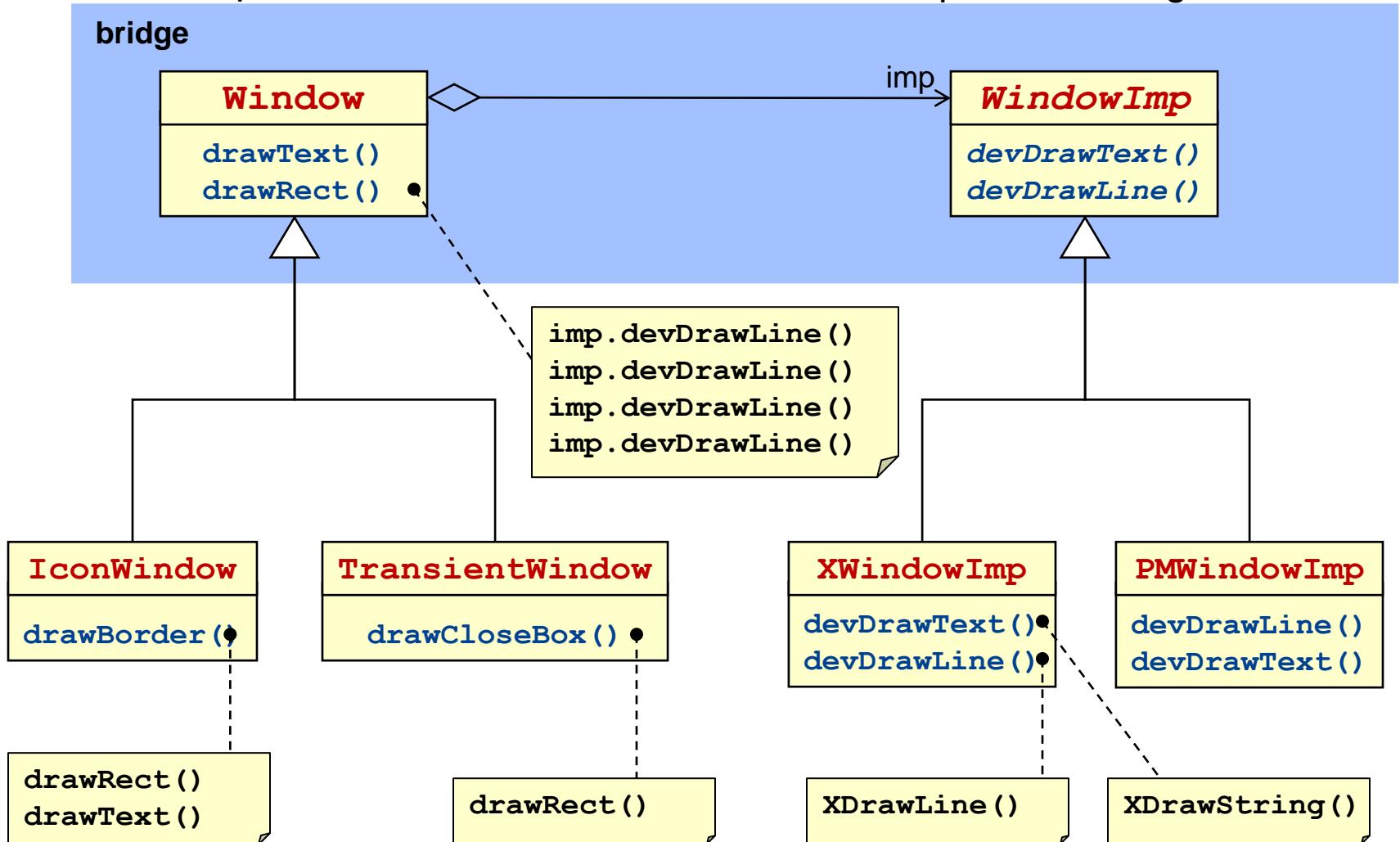
Bridge Pattern: Warum nicht einfach Vererbung einsetzen?



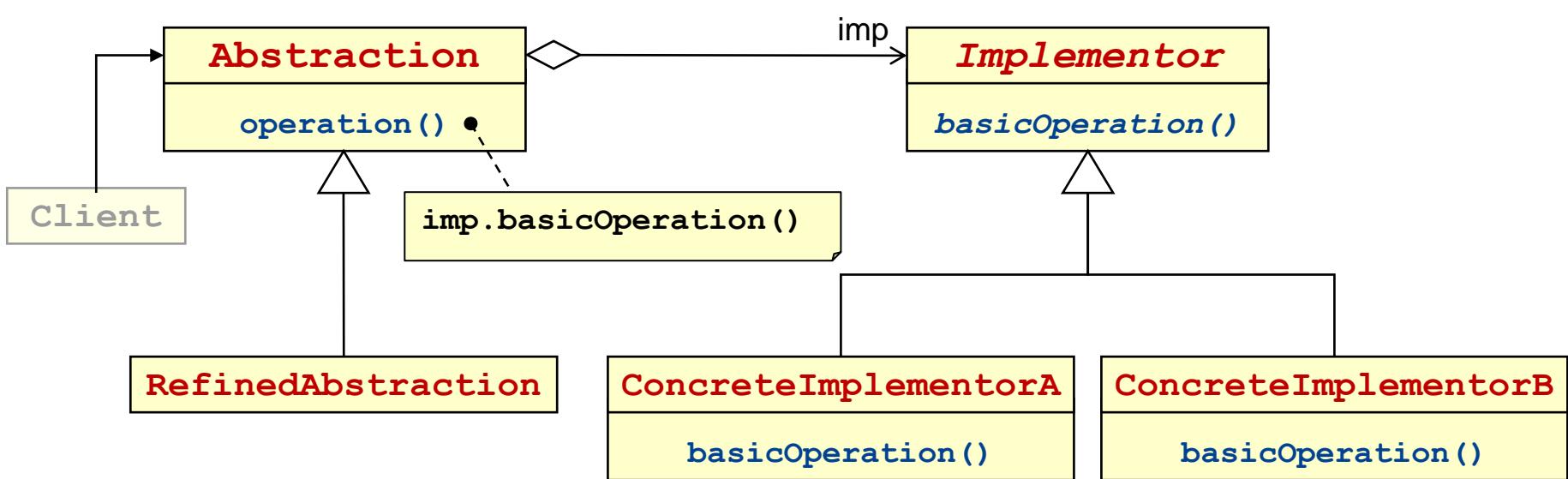
- Probleme dieses Lösungsversuchs
 - ◆ Eigene Klasse für jede Kombination Fenster-Art / Plattform
 - ⇒ unerwartbar
 - ◆ Client wählt Kombination Fenster-Art / Plattform (bei der Objekterzeugung)
 - ⇒ plattformabhängiger Client-Code

Bridge Pattern: Idee

- Trennung von
 - ◆ Konzeptueller Hierarchie
 - Implementierungs-Hierarchie



Bridge Pattern: Schema

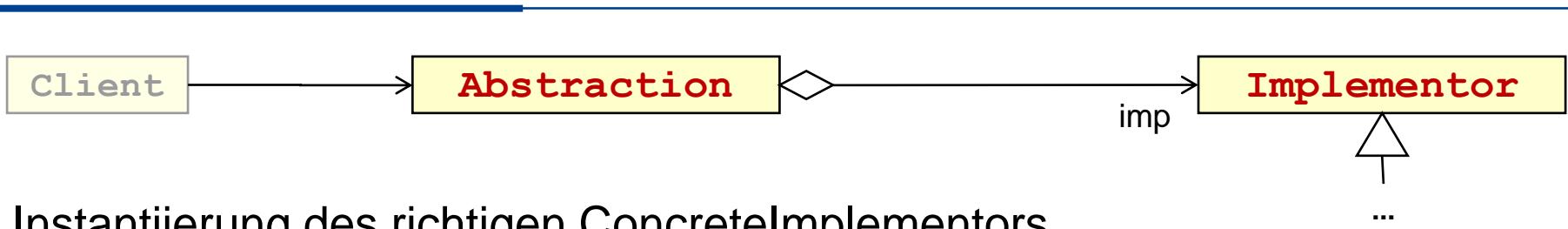


- Abstraction
 - ◆ definiert Interface
 - ◆ referenziert Implementierung
- RefinedAbstraction
 - ◆ erweitert das Interface
- Implementor
 - ◆ Interface der Implementierungs-Hierarchie
 - ◆ kann von Abstraction abweichen
- ConcreteImplementor
 - ◆ wirkliche Implementierung

Bridge Pattern: Anwendbarkeit

- Dynamische Änderbarkeit
 - ◆ Implementierung erst zur Laufzeit auswählen
- Unabhängige Variabilität
 - ◆ neue Unterklassen in beiden Hierarchien beliebig kombinierbar
- Implementierungs-Transparenz für Clients
 - ◆ Änderungen der Implementierung erfordern keine Änderung / Neuübersetzung der Clients
- Vermeidung von "Nested Generalisations"
 - ◆ keine Hierarchien der Art wie in der Motivations-Folie gezeigt
 - ◆ keine kombinatorische Explosion der Klassenanzahl
- Sharing
 - ◆ mehrere Clients nutzen gleiche Implementierung
 - ◆ z.B. Strings

Bridge Pattern: Implementierung



- Falls Abstraction alle ConcreteImplementor-Klassen kennt:
 - ◆ Fallunterscheidung im Konstruktor der ConcreteAbstraction
 - ◆ Auswahl des ConcreteImplementor anhand von Parametern des Konstruktors
 - ◆ Alternativ: Default-Initialisierung und spätere situationsbedingte Änderung
- Falls Abstraction völlig unabhängig von ConcreteImplementor-Klassen sein soll:
 - ◆ Entscheidung anderem Objekt überlassen
 - Abstract Factory Pattern

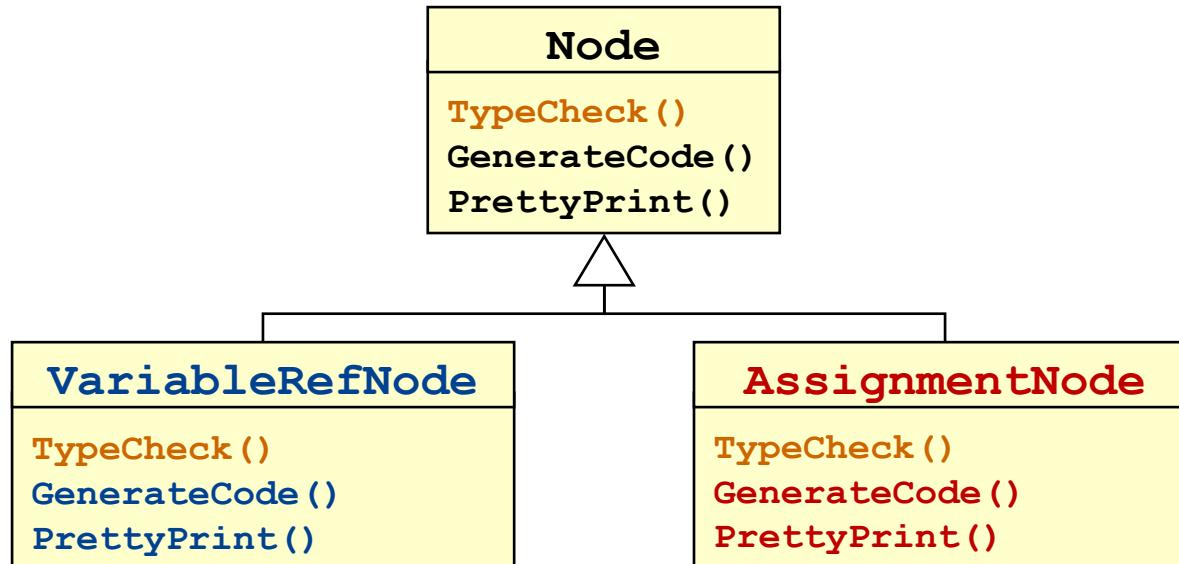
Das Visitor Pattern

Das Visitor Pattern

- Absicht
 - ◆ Repräsentation von Operationen auf Elementen einer Objektstruktur
 - ◆ Neue Operationen definieren, ohne die Klassen dieser Objekte zu ändern

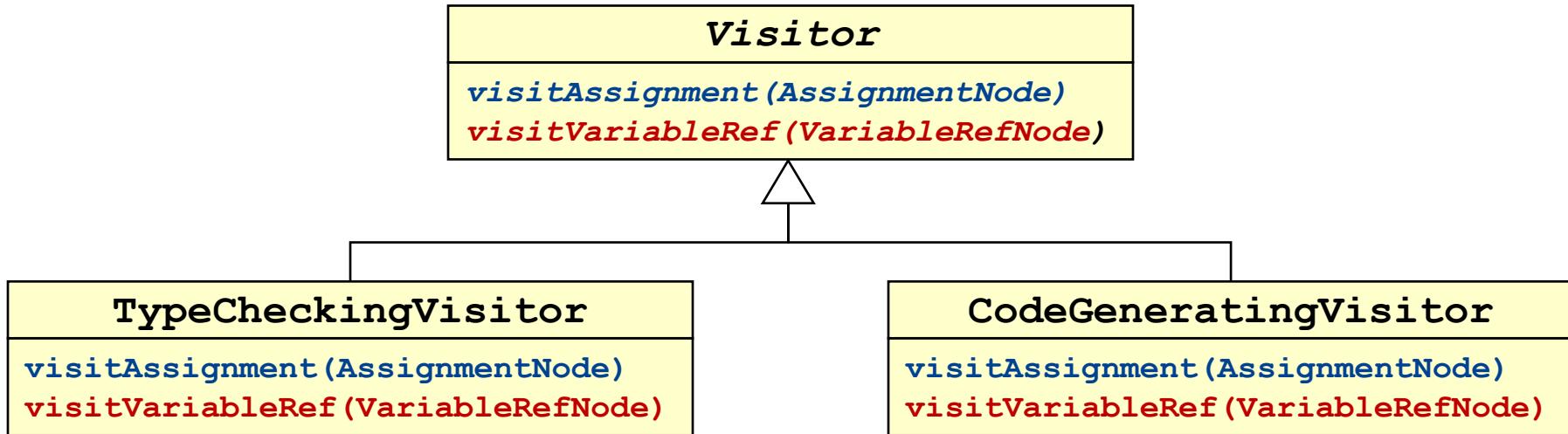
Visitor Pattern: Motivation

- Beispiel: Compiler
 - ◆ enthält Klassenhierarchie für Ausdrücke einer Programmiersprache
- Problem
 - ◆ Code einer Operation (z.B. Typüberprüfung) ist über mehrere Klassen einer Datenstruktur verteilt
 - ◆ Hinzufügen oder ändern einer Operation erfordert Änderung der gesamten Hierarchie



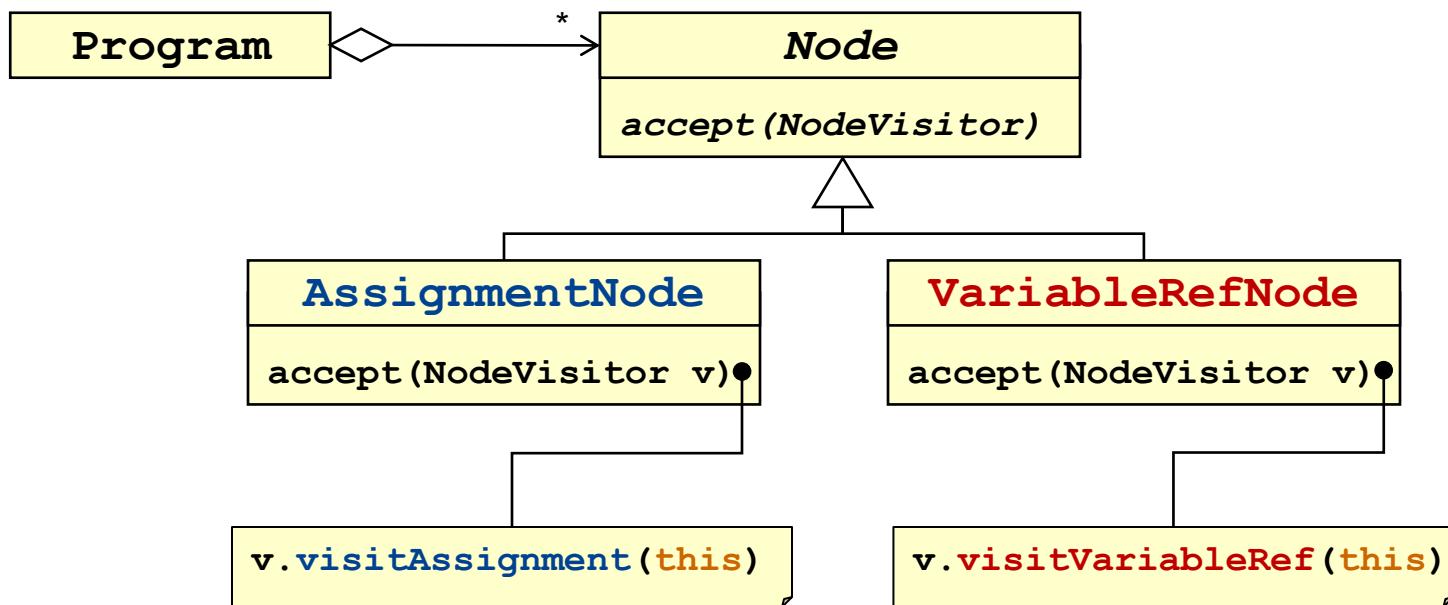
Visitor Pattern: Idee (1)

- Visitor
 - ◆ Klasse die alle visit...-Methoden zusammenfasst, die gemeinsam eine Operation auf einer Objektstruktur realisieren
- Visit...-Methode
 - ◆ Ausprägung der Operation auf einem bestimmtem Objekttyp
 - ◆ Hat Parameter vom entsprechenden Objekttyp
 - ⇒ Kann somit alle Funktionalitäten des Typs nutzen um das zu besuchende Objekt zu bearbeiten

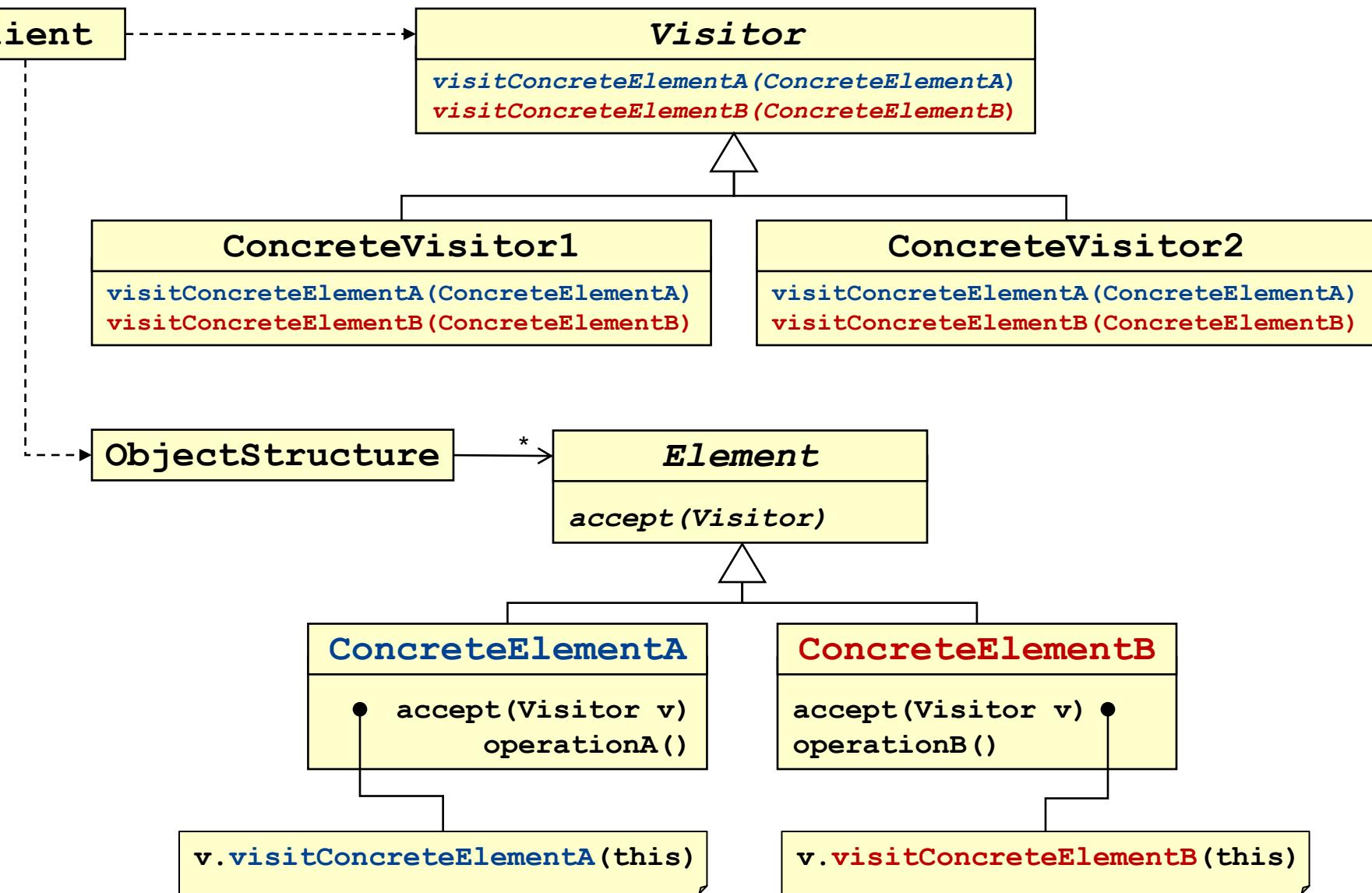


Visitor Pattern: Idee (2)

- Accept-Methoden in allen Klassen der betroffenen Klassenhierarchie
 - ◆ Haben **Visitor** als Parameter
 - ⇒ „Diese Operation soll auf mir ausgeführt werden!“
 - ◆ Rufen die jeweils zu ihnen passende visit...-Methode auf
 - ⇒ „Diese Variante der Operation muss auf **mir** ausgeführt werden!“
 - ⇒ Übergeben „**this**“ als Parameter



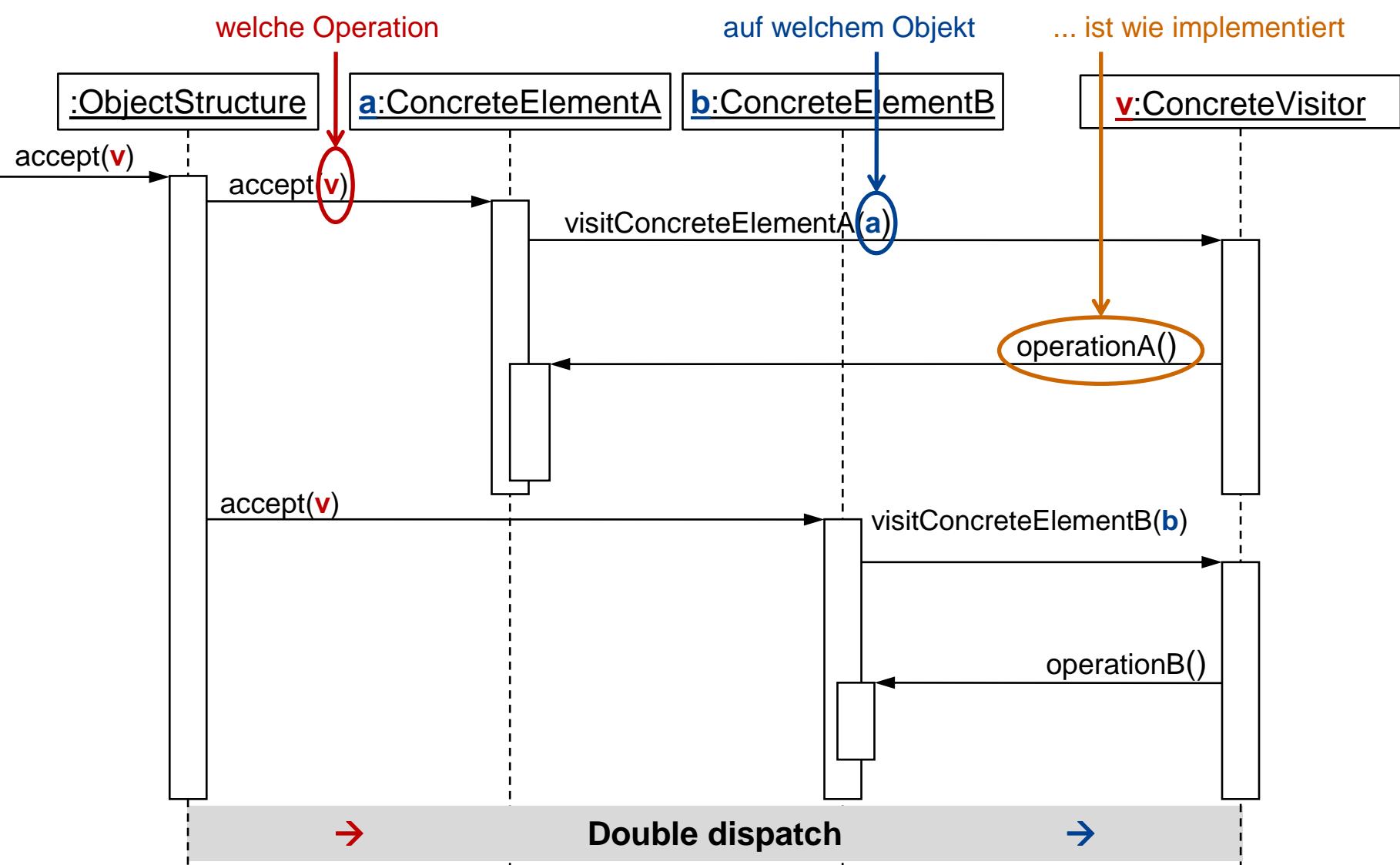
Visitor Pattern: Schema (Statisch)



Visitor Pattern: Verantwortlichkeiten / Implementation

- Objektstruktur-Hierarchie
 - ◆ Einheitliche accept-Methode
- Visitor-Hierarchie
 - ◆ Je eine visit-Methode für jede Klasse der Objektstruktur mit Parameter vom Typ der jeweilige Klasse
- Traversierung der Objektstruktur kann definiert sein in
 - ◆ der besuchten Objektstruktur (Methode accept(...)) oder
 - ◆ dem Visitor-Objekt (Method visit...(...))

Visitor Pattern: Zusammenarbeit



Das Visitor Pattern: Konsequenzen

- Hinzufügen neuer Operationen ist einfach.
 - ◆ Neue Visitor-Klasse
- Hinzufügen neuer Objektklassen ist sehr aufwendig.
 - ◆ Neue Objekt-Klasse
 - ◆ Neue visit... - Methode in allen Visitors
- Sammeln von Zuständen
 - ◆ Visitor-Objekte können Zustände der besuchten Objekte aufsammeln und (evtl. später) auswerten.
 - ◆ Eine Übergabe von zusätzlichen Parametern ist dafür nicht nötig.
- Verletzung von Kapselung
 - ◆ Die Schnittstellen der Klassen der Objektstruktur müssen ausreichend Funktionalität bieten, damit Visitor-Objekte ihre Aufgaben erledigen können.
 - ◆ Oft muss mehr preisgegeben werden als (ohne Visitor) nötig wäre.

Das Visitor Pattern: Anwendbarkeit

- Funktionale Dekomposition
 - ◆ Zusammengehörige Operationen sollen zusammengefasst werden
 - ◆ ... statt in einer Klassenhierarchie verteilt zu sein
- Stabile Objekt-Hierarchie
 - ◆ selten neue Klassen
 - ◆ aber oft neue Operationen
- Heterogene Hierarchie
 - ◆ viele Klassen mit unterschiedlichen Schnittstellen
 - ◆ Operationen die von den Klassen abhängen
- Anwendungsbeispiel
 - ◆ Java-Compiler des JDK 1.3

Das Visitor Pattern: Implementierung

- Abstrakter Visitor
 - ◆ Jede Objektstruktur besitzt **eine** (abstrakte) Visitor-Klasse.
 - ◆ **Für jeden Typ T** in der Objektstruktur, enthält die Visitor-Klasse **je eine** Methode mit einem Parameter vom Typ T → **visitT(T)**
- Konkrete Visitors
 - ◆ Jede UnterkLASSE der Visitor-Klasse redefiniert die visit-METHODEN, um ihre jeweilige Funktionalität zu realisieren.
 - ◆ Jede konkrete **visitT(T)** Methode benutzt dabei die spezifischen Operationen des besuchten Objekttyps T
- Traversierung der Objektstruktur
 - ◆ kann in der Objektstruktur (**accept(...)** Methoden) definiert sein
 - ◆ ... oder im Visitor-Objekt (**visit...(...)** Methoden).

Kapitel 5 Refactoring

Stand: 4.12.2023

Themenbereiche und Vorlesungs-Kapitel

III. PROZESSE

12

Agile Softwareentwicklung

11

Software-Prozess-Modelle

II. AKTIVITÄTEN

10

Test

9

Objekt-Design

8

System-Design

7

Anforderungs-Analyse

6

Anforderungs-Erhebung



5

Refactoring

4

Entwurfsmuster

3

Unified Modelling Language (UML)

2

OO-Modellierung

1

OO-Programmierung ++

I. WERKZEUGE

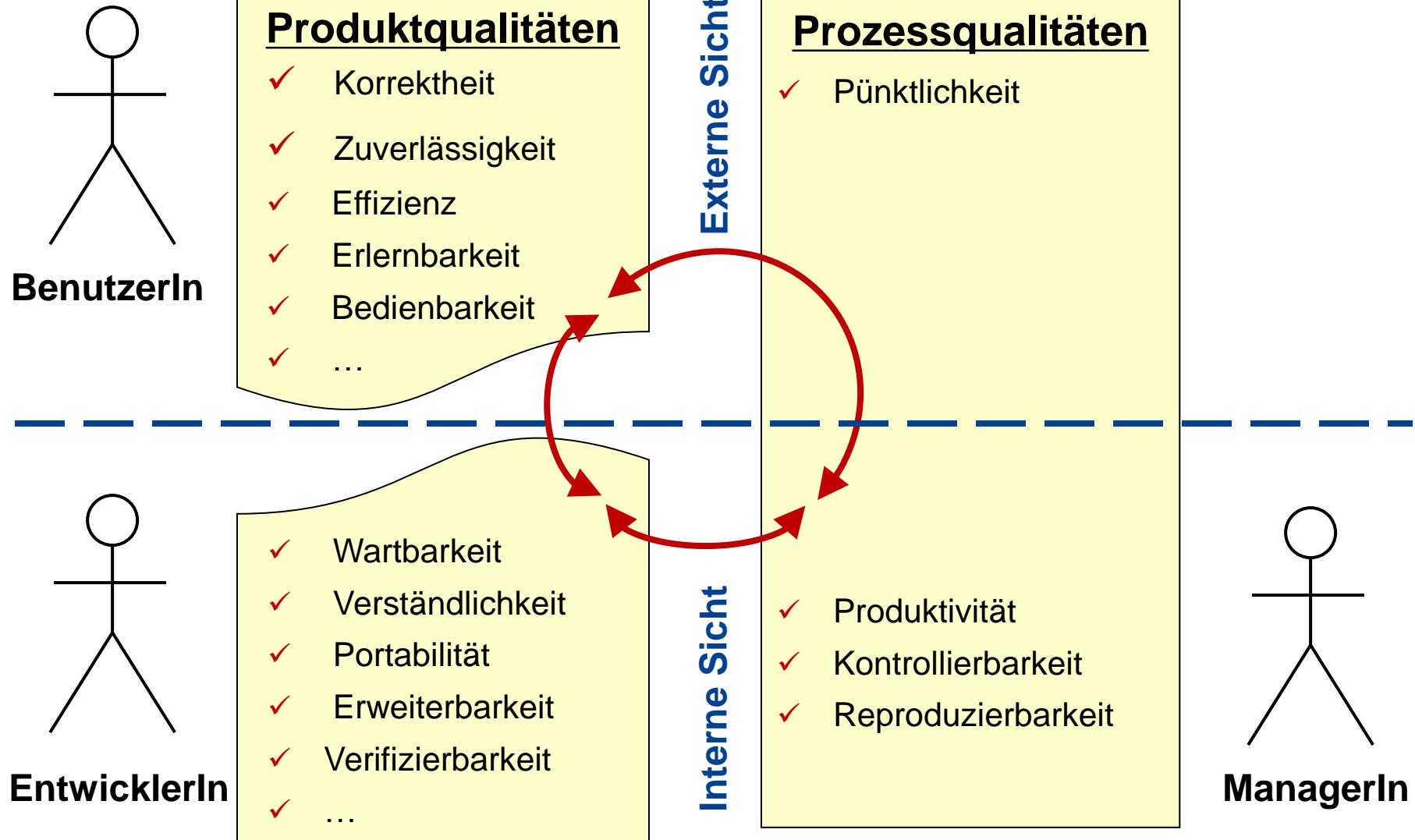
VORWISSEN

Software Configuration Management (mit Git)

OO-Programmierung Grundlagen

Imperative-Programmierung

Software-Qualitäten



Was ist überhaupt “Refactoring”?

Refactoring (noun):

a change made to the internal structure of software
to make it easier to understand and cheaper to modify
without changing its observable behavior.

Refactor (verb):

to restructure software by applying a series of refactorings.

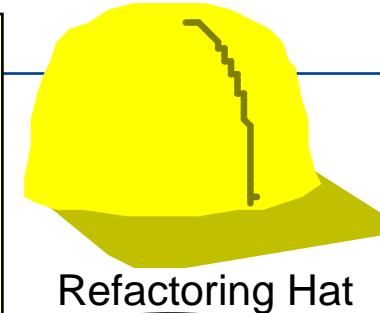
- Definition
 - ◆ Systematische Umstrukturierung des Codes
ohne das Verhalten nach außen zu ändern

Was heißt "Systematische Umstrukturierung"?

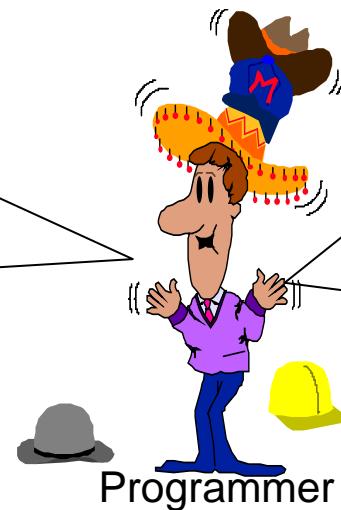
- Klare Anweisungen
 - ◆ was → welche Programmelemente
 - ◆ wie → wie verändert / hinzugefügt werden
 - ◆ wann → wann überhaupt (→ „Bad Smells“) & in welcher Reihenfolge
- Festgelegter Ablauf
 - ◆ kleine Schritte
 - ◆ Tests nach jedem Schritt
 - ◆ strikte Trennung von Refactoring und Funktionsänderungen
- Disziplin!!!

Ablauf: Kent Beck's "Hüte-Metapher"

- Was will ich **umstrukturieren**?
- Gibt es einen Test? → Test schreiben!
- Refactoring durchführen
- Testen → Fehler beheben!



Immer **zuerst alle Refactorings** durchführen!
Sobald du Funktionserweiterungen durchführst verzichtest du auf Feedback durch deine bestehenden Tests.



Denn die testen das alte Verhalten.

Wenn sie nach einer Funktionsänderung fehlschlagen, weißt du nicht ob das wegen eines Fehlers ist oder weil sie nicht zum neuen Verhalten passen.



- Was will ich **hinzufügen**?
- Test schreiben
- Funktionserweiterung durchführen
- Testen → Fehler beheben!

Warum soll man "Refactoring" anwenden?

- Programme sind schwer zu warten wenn sie
 - ◆ ... unverständlich sind
 - ◆ ... Redundanzen enthalten
 - ◆ ... komplexe Fallunterscheidungen enthalten
 - ◆ ... Änderungen bestehenden Codes erfordern, um Erweiterungen zu implementieren
- Probleme
 - ◆ Oft geänderte Software verliert ihre Struktur.
 - ◆ Je mehr Code um so unverständlicher
 - ◆ Redundanter Code / Inkonsistenzen

Warum soll man "Refactoring" anwenden?

- “Refactoring“ macht Software leichter wartbar
 - ◆ Extraktion gemeinsamer Teile: jede Änderung an genau einer Stelle durchführen
 - ◆ "Rule of three": Wenn man das zweite Mal das gleiche tut ist Code-Duplikation noch OK. Beim dritten Mal sollte spätestens Umstrukturiert werden.
- “Refactoring“ macht Software leichter verständlich
 - ◆ Einarbeitung ist leichter, wenn man sofort restrukturiert
 - ◆ hilft einem selbst und denen die später kommen
- “Refactoring“ hilft Bugs zu finden
 - ◆ besseres Verständnis für den Code erleichtert Fehlersuche
- “Refactoring“ macht das Programmieren schneller
 - ◆ in verständlichen und fehlerarmen Code kann Neues schneller eingebaut werden

Wann soll man Refactorings anwenden? → „Bad Smells“

- „Bad Smell“
 - ◆ Zustand des Codes, der hinsichtlich internen Softwarequalitäten problematisch ist
 - ◆ Klare, verifizierbare Kriterien
 - ◆ Hinweis auf anzuwendendes Refactoring, das dieses Problem behebt
- Medizinische Analogie
 - ◆ Symptom ≈ Bad Smell
 - ◆ Therapie ≈ spezifisches Refactoring
 - ◆ Diagnose ≈ Ursache ← dies wird beim Refactoring nicht betrachtet
(Ursachen zu beheben ist Teil des Prozesses)

Refactoring – Schritt für Schritt

„Encapsulate Field“

„Rename ...“

„Extract Method“

„Bad Smell“: Public Field

„Bad Smell“

- ◆ Klasse hat öffentliche Felder
- Korrektheit ☹
- Wartbarkeit ☹

Zielzustand

- Klasse hat private Felder und Zugriffsmethoden
- Alle direkten Zugriffe sind durch Zugriffsmethodenaufrufe ersetzt
 - ◆ auch innerhalb der Klasse!

Schritte

1. Getter und Setter implementieren
2. Compilieren
3. Zugriffe finden
 - a) lokal
 - b) global
4. Keine Tests? → Tests schreiben!
5. Zugriffe ersetzen
6. Compilieren
7. Testen

„Bad Smell“: Schlechter Name

„Bad Smell“

- ◆ Name nicht aussagekräftig
- ◆ Zu allgemein oder zu speziell
- ◆ Veraltet (gibt nicht aktuelle Funktion wieder)
- ◆ ...
- Verständlichkeit ☹
- Wartbarkeit ☹

Zielzustand

- „gute“ Namen
- Alle Deklarationen und alle Aufrufe / Nutzungen sind umbenannt
- Durch die Umbenennung entstehen keine unabsichtlichen Zugriffe / Aufrufe, die es vorher nicht gegeben hat und es fallen keine weg die vorher stattfanden

Extract Method

- Indikation („Bad Smell“ / „Code Smell“):

Code-Fragment das logisch zusammengehört ...

- ◆ soll woanders wiederverwendet werden
→ ohne Kopien / Redundanzen!!!

oder

- ◆ behindert das Verständnis der enthaltenden Methode

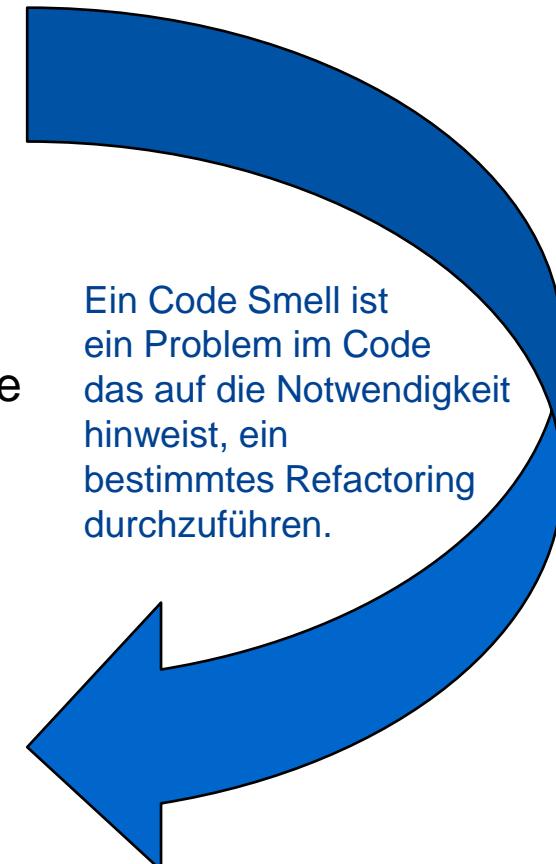
oder

- ◆ hat eine unklare Bedeutung

- Behandlung („Refactoring“)

Durch aussagekräftig benannte Methode ersetzen, in kleinen Schritten, die Fehler vermeiden!!!

- ◆ Varianten beachten!!!
- ◆ Beispiel s. nächste Folien



Ein Code Smell ist ein Problem im Code das auf die Notwendigkeit hinweist, ein bestimmtes Refactoring durchzuführen.

Beispiel: Drei Extraktionen

1. Banner-Druck → printBanner()
2. Ergebnis-Ausgabe → printDetails()
3. Berechnung → getOutstanding()

```
void printOwing(double amount) {  
  
    Enumeration e = orders.elements();  
    double outstanding = 0.0;  
  
    // print banner  
    System.out.println("*****");  
    System.out.println(" *** Customer owes ***");  
    System.out.println("*****");  
  
    // calculate outstanding  
    while (e.hasMoreElements()) {  
        Order each = (Order) e.nextElement();  
        outstanding += each.getAmount();  
    }  
    // print details  
    System.out.println("name"+ _name);  
    System.out.println("amount"+ outstanding);  
}
```

Fall 1 (Extraktion von printBanner): Keine lokale Variable im extrahierten Block

- a) Neue Methode erzeugen
- b) Zu extrahierenden Code kopieren
- c) Neue Methode compilieren! → „Soweit alles OK?“

```
void printOwing(double amount) {  
  
    Enumeration e = orders.elements();  
    double outstanding = 0.0;  
  
    // print banner  
    System.out.println("*****");  
    System.out.println(" *** Customer owes ***");  
    System.out.println("*****");  
  
    // calculate outstanding  
    while (e.hasMoreElements()) {  
        Order each = (Order) e.nextElement();  
        outstanding += each.getAmount();  
    }  
    // print details  
    System.out.println("name"+ _name);  
    System.out.println("amount"+ outstanding);  
}
```

```
private void printBanner() {  
    System.out.println("*****");  
    System.out.println(" *** Customer owes ***");  
    System.out.println("*****");  
}
```

Fall 1 (Extraktion von printBanner): Keine lokale Variable im extrahierten Block

- d) Zu extrahierenden Code durch Methodenaufruf ersetzen
 - ◆ Destruktiver Schritt ist atomar! (möglich durch Vorbereitung a-c)

```
void printOwing(double amount) {  
  
    Enumeration e = orders.elements();  
    double outstanding = 0.0;  
  
    printBanner();  
  
    // calculate outstanding  
    while (e.hasMoreElements()) {  
        Order each = (Order) e.nextElement();  
        outstanding += each.getAmount();  
    }  
    // print details  
    System.out.println("name"+ _name);  
    System.out.println("amount"+ outstanding);  
}  
  
private void printBanner() {  
    System.out.println("*****");  
    System.out.println("*** Customer owes ***");  
    System.out.println("*****");  
}
```

Fall 2 (Extraktion von printDetails): Lokale Variable, die gelesen wird

Der extrahierte Code enthält Lesezugriff(e) auf eine lokale Variable der Ursprungsmethode („outstanding“), aber keine Schreibzugriffe:

- Die Variable wird zum Parameter der extrahierten Methode.

```
void printOwing(double amount) {  
  
    Enumeration e = orders.elements();  
    double outstanding = 0.0;  
  
    printBanner();  
  
    // calculate outstanding  
    while (e.hasMoreElements()) {  
        Order each = (Order) e.nextElement();  
        outstanding += each.getAmount();  
    }  
    // print details  
    System.out.println("name"+ name);  
    System.out.println("amount"+ outstanding);  
}
```

```
private void printDetails(double outstanding) {  
  
    name);  
    outstanding);  
}
```

Fall 2 (Extraktion von printDetails): Lokale Variable, die gelesen wird

Danach alle anderen Schritte wie sonst auch, bis hin zur Ersetzung des extrahierten Codes durch den Aufruf der neuen Methode:

```
void printOwing(double amount) {  
  
    Enumeration e = orders.elements();  
    double outstanding = 0.0;  
  
    printBanner();  
  
    // calculate outstanding  
    while (e.hasMoreElements()) {  
        Order each = (Order) e.nextElement();  
        outstanding += each.getAmount();  
    }  
  
    printDetails(outstanding);  
}  
  
private void printDetails(double outstanding) {  
    System.out.println("name"+ _name);  
    System.out.println("amount"+ outstanding);  
}
```

Fall 3 (Extraktion der Berechnung): Lokale Variable, die geschrieben wird

- Berechnung enthält
 - ◆ Zuweisung an Variable „outstanding“, die nachher weiter genutzt wird
 - ⇒ outstanding muss als Parameter übergeben und als Ergebnis zurückgegeben werden
 - ◆ einzige Zugriffe auf Variable „e“ (außerhalb der Deklaration / Initialisierung)
 - ⇒ Deklaration und Initialisierung wird mit in die neue Methode kopiert

```
void printOwing(double amount) {  
  
    Enumeration e = orders.elements();  
    double outstanding = 0.0;  
  
    printBanner();  
  
    // calculate outstanding  
    while (e.hasMoreElements())  
        Order each = (Order) e.  
        outstanding += each.getA  
    }  
  
    printDetails(outstanding);  
}
```

```
private double getOutstanding(double outstanding) {  
    Enumeration e = orders.elements();  
  
    while (e.hasMoreElements()) {  
        Order each = (Order) e.nextElement();  
        outstanding += each.getAmount();  
    }  
    return outstanding;  
}
```

Fall 3 (Extraktion der Berechnung): Lokale Variable, die geschrieben wird

- In Ursprungsmethode wird schließlich
 - ◆ der extrahierte Code ersetzt durch eine Zuweisung an „outstanding“ deren Rechte Seite der Aufruf der neuen Methode ist
 - ◆ die Deklaration der nun überflüssigen Variable „e“ gelöscht (**in dieser Reihenfolge!**)

```
void printOwing(double amount) {  
  
    Enumeration e = orders.elements();  
    double outstanding = 0.0;  
  
    printBanner();  
  
    outstanding = getOutstandi  
  
    printDetails(outstanding);  
}
```

```
private double getOutstanding(double outstanding) {  
    Enumeration e = orders.elements();  
  
    while (e.hasMoreElements()) {  
        Order each = (Order) e.nextElement();  
        outstanding += each.getAmount();  
    }  
    return outstanding;  
}
```

Fall 3 (Extraktion der Berechnung): Lokale Variable, die geschrieben wird

- Compilierung und Test
- Dann Aufräumen der extrahierten Methode
 - ◆ Zuweisung an Parameter ist schlechter Stil (eigener „Code Smell“)
 - ⇒ lokale Variable „result“ erzeugen
 - ◆ Parametername passt nicht mehr
 - ⇒ sinnvoll umbenennen → „startValue“

```
void printOwing(double amount) {  
  
    double outstanding = 0.0;  
  
    printBanner();  
  
    outstanding = getOutstanding();  
  
    printDetails(outstanding);  
}
```

```
private double getOutstanding(double outstanding) {  
    Enumeration e = orders.elements();  
  
    while (e.hasMoreElements()) {  
        Order each = (Order) e.nextElement();  
        outstanding += each.getAmount();  
    }  
    return outstanding;  
}
```

Fall 3 (Extraktion der Berechnung): Lokale Variable, die geschrieben wird

- Compilierung und Test
- Dann Aufräumen der extrahierten Methode
 - ◆ Zuweisung an Parameter ist schlechter Stil (eigener „Code Smell“)
 - ⇒ lokale Variable „result“ erzeugen
 - ◆ Parametername passt nicht mehr
 - ⇒ sinnvoll umbenennen → „startValue“

```
void printOwing(double amount) {  
  
    double outstanding = 0.0;  
  
    printBanner();  
  
    outstanding = getOutstanding();  
  
    printDetails(outstanding);  
}  
  


```
private double getOutstanding(double startValue) {
 Enumeration e = orders.elements();
 double result = startValue;
 while (e.hasMoreElements()) {
 Order each = (Order) e.nextElement();
 result += each.getAmount();
 }
 return result;
}
```


```

Zwischenzustand nach drei Methoden-Extraktionen

```
private void printBanner() {  
    System.out.println("*****");  
    System.out.println(" *** Customer owes ***");  
    System.out.println("*****");  
}
```

```
void printOwing(double amount) {  
  
    double outstanding = 0.0;  
  
    printBanner();  
  
    outstanding = getOutstanding(outstanding);  
  
    printDetails(outstanding);  
}
```

```
private void printDetails(double outstanding) {  
    System.out.println("name"+ _name);  
    System.out.println("amount"+ outstanding);  
}
```

```
private double getOutstanding(double startValue) {  
    Enumeration e = orders.elements();  
    double result = startValue;  
    while (e.hasMoreElements()) {  
        Order each = (Order) e.nextElement();  
        result += each.getAmount();  
    }  
    return result;  
}
```

Zusammenfassung: Schritte des „Extract Method“ Refactorings

1. Vorbedingungen prüfen: Der zu extrahierende Code enthält keine

- ◆ unvollständigen Anweisungen oder Blöcke

⇒ if(a == b) { doA(); } else { doB(); doC(); }
⇒ if(a == b) { doA(); } else { doB(); doC(); }
⇒ doA(); doB(); doC();

- ◆ „return“-Anweisungen

⇒ if(a == b) { doA(); return; } else { doB(); doC(); }
... weiterer Code ...

- ◆ Zuweisungen an mehr als eine lokale Variable, die nach dem extrahierten Teilstück noch benutzt wird

⇒ if(a == b) { e=f(); } else { c=d; }
if(c == e) ...

⇒ Falls mehr als eine: Vorbereitend Refactorings probieren, die

- Variablen eliminieren (z.B. „Replace Temp by Query“)
oder
- Gruppen von Variablen zu einem Objekt zusammenfassen (z.B. „Introduce Parameter Object“)

```
void printOwing(double amount) {  
  
    Iterator<Order> e =  
        orders.iterator();  
    double outstanding = 0.0;  
  
    printBanner();  
  
    while (e.hasNext()) {  
        outstanding +=  
            e.next().getAmount();  
    }  
  
    printDetails(outstanding);  
}
```

Zusammenfassung: Schritte des „Extract Method“ Refactorings

Falls Methode extrahierbar:

2. Lokale Variablen des zu extrahierenden Codes in Ursprungsmethode suchen

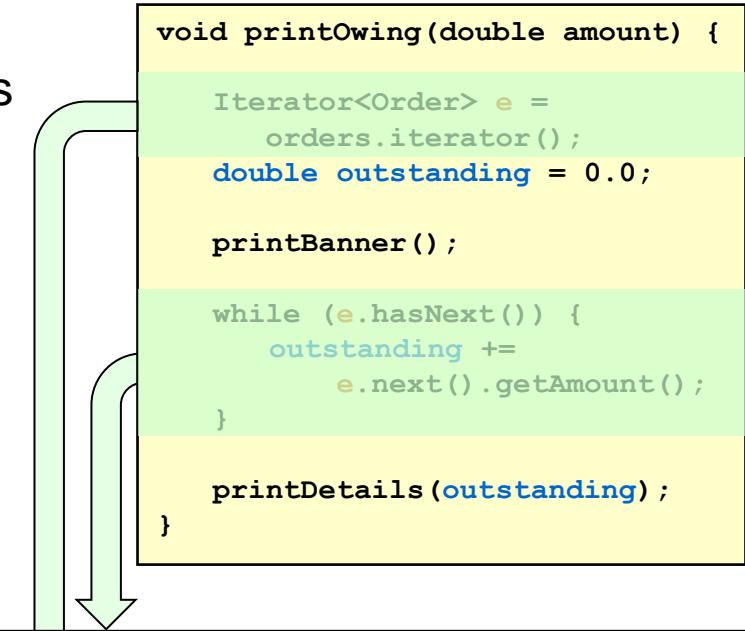
- ◆ Variablen, die in der neuen Methode gelesen werden
 - ⇒ Deklaration als Parameter der neuen Methode
- ◆ Variable, die in neuer Methode verändert und in der alten weiter benutzt wird
 - ⇒ als Ergebnis der neuen Methode zurückgeben
 - ⇒ es kann keine zweite geben (sonst wäre die Methode nicht extrahierbar, siehe 1.).

3. Neue Methode mit passender Signatur erzeugen und sinnvoll benennen

- ◆ immer „private“

4. Lokale Variablen der Ursprungsmethode suchen die nur in der neuen Methode benutzt werden

- ⇒ Deklaration als lokale Variablen der neuen Methode



```
private double getOutstanding(double outstanding) {  
    return 0;  
}
```

```
private double getOutstanding(double outstanding) {  
    Iterator<Order> e = orders.iterator();  
    return 0;  
}
```

Zusammenfassung: Schritte des „Extract Method“ Refactorings

5. Zu extrahierenden Code in die neue Methode kopieren (in der Alten noch nicht löschen!)

6. Kompilieren der extrahierten Methode

```
void printOwing(double amount) {  
  
    Iterator<Order> e =  
        orders.iterator();  
    double outstanding = 0.0;  
  
    printBanner();  
  
    while (e.hasNext()) {  
        outstanding +=  
            e.next().getAmount();  
    }  
  
    printDetails(outstanding);  
}
```

```
private double getOutstanding(double outstanding) {  
    Iterator<Order> e = orders.iterator();  
  
    while (e.hasNext()) {  
        outstanding += e.next().getAmount();  
    }  
    return outstanding;  
}
```

Zusammenfassung: Schritte des „Extract Method“ Refactorings

7. In Ursprungsmethode

- ◆ Deklaration nicht mehr benötigter lokaler Variablen löschen
- ◆ Extrahierte Code durch Aufruf der neuen Methode ersetzen

8. Kompilieren der Ursprungsmethode

9. Testen!!!

10. Extrahierte Methode aufräumen (z.B. Lokale Variablen sinnvoll umbenennen)

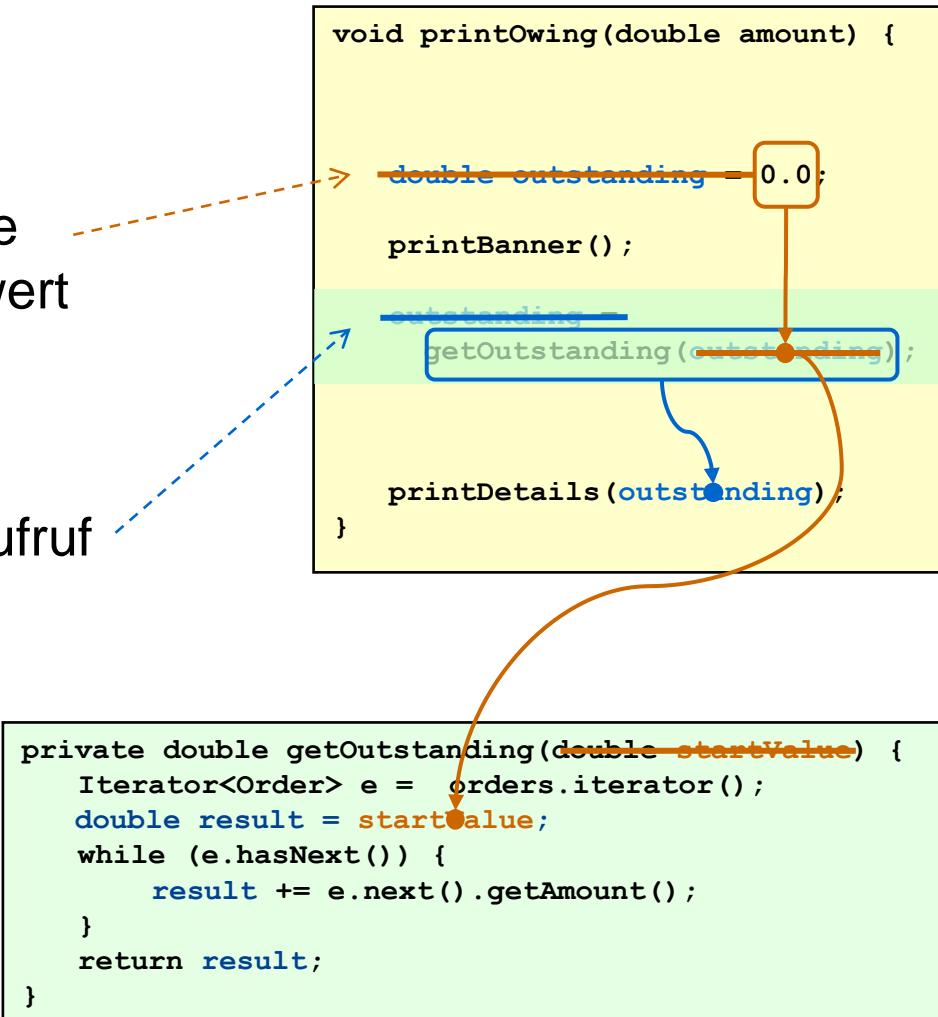
```
void printOwing(double amount) {  
  
    double outstanding = 0.0;  
  
    printBanner();  
  
    outstanding =  
        getOutstanding(outstanding);  
  
    printDetails(outstanding);  
}
```

```
private double getOutstanding(double startValue) {  
    Iterator<Order> e = orders.iterator();  
    double result = startValue;  
    while (e.hasNext()) {  
        result += e.next().getAmount();  
    }  
    return result;  
}
```

Die Aufwärtsspirale: Folge-Refactorings

Bessere Code-Struktur ermöglicht Folge-Refactorings:

- Eine nur einmal genutzte Variable kann durch ihren Initialisierungswert ersetzt werden
 - ◆ → „Inline Temp“ Refactoring
- Ein seiteneffektfreier Methodenaufruf kann selbst als Parameter übergeben werden
 - ◆ → „Replace Temp with Query“ Refactoring
- Schliesslich kann der Wert von `startValue` (der immer 0 ist) in `getOutstanding()` gesetzt werden



Die Aufwärtsspirale: Folge-Refactorings

- Jede Methode hat genau eine Aufgabe
- Kurz und klar
- Die getOutstanding()-Methode implementiert Anwendungslogik, die evtl. an anderer Stelle auch gebraucht wird
 - ◆ wiederverwendbar ☺

```
void printOwing(double amount) {  
    printBanner();  
    printDetails(getOutstanding());  
}
```

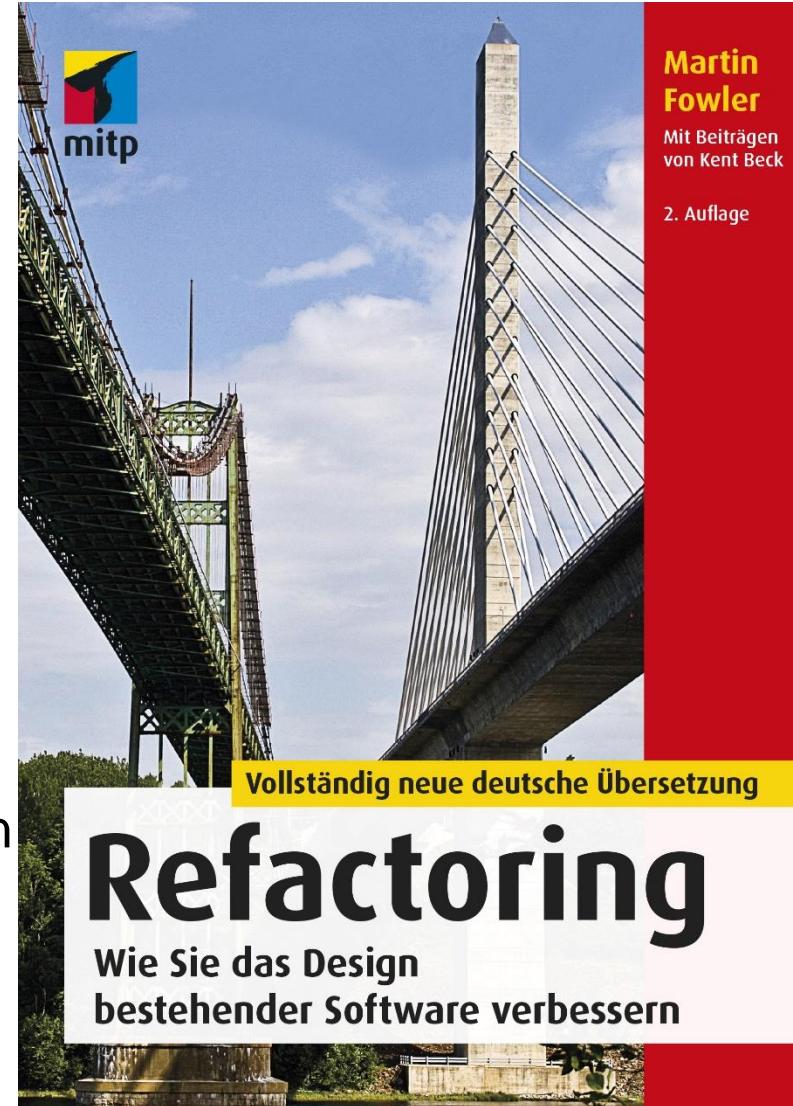
```
private void printBanner() {  
    System.out.println("*****");  
    System.out.println(" *** Customer owes ***");  
    System.out.println("*****");  
}
```

```
private void printDetails(double outstanding) {  
    System.out.println("name"+ _name);  
    System.out.println("amount"+ outstanding);  
}
```

```
private double getOutstanding() {  
    Iterator<Order> e = orders.iterator();  
    double result = 0.0;  
    while (e.hasNext()) {  
        result += e.next().getAmount();  
    }  
    return result;  
}
```

Refactoring-Katalog (Buch von Martin Fowler)

- Komposition von Methoden
 - ◆ Extract Method
 - ◆ Inline Method
 - ◆ Inline Temp
 - ◆ Replace Temp with Query
 - ◆ Split Temporary Variable
 - ◆ Remove Assignments to Parameters
 - ◆ Replace Method with Method Object
 - ◆ ...
- Verlagerung von Methoden
 - ◆ ...
- Strukturierung von Daten
 - ◆ ...
- Vereinfachung von Fallunterscheidungen
 - ◆ ...
- Vereinfachung von Methodenaufrufen
 - ◆ ...
- Vererbung
 - ◆ ...



Automatisierte Refactorings

► Beispiel: Eclipse

- Prinzip: Die Entwicklungsumgebung
 - ◆ ... prüft Vorbedingungen, die gelten müssen, damit das Refactoring verhaltenserhaltend ist
 - ◆ ... inferiert notwendige Informationen
 - ⇒ z.B. die Signatur einer zu extrahierenden Methode
 - ◆ ... führt die Änderung überall konsistent durch
 - ⇒ z.B. werden bei einem „Rename Refactoring“ alle Vorkommnisse des geänderten Namens geändert – in dessen Deklaration und in allen Referenzen darauf – aber keine zufällig gleichen Namen!
 - ◆ ... bietet Vorschau der Änderungen
- Auch andere IDEs (zB IntelliJ) unterstützen viele Refactorings

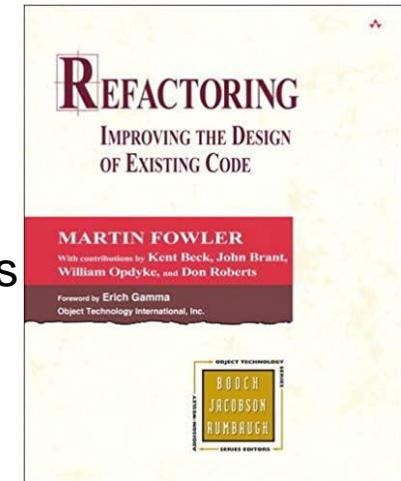
File	Edit	Source	Refactor	Navigate	Search	Project
			Rename...			Alt+Shift+R
			Move...			Alt+Shift+V
			Change Method Signature...			Alt+Shift+C
			Extract Method...			Alt+Shift+M
			Extract Local Variable...			Alt+Shift+L
			Extract Constant...			
			Inline...			Alt+Shift+I
			Convert Local Variable to Field...			
			Convert Anonymous Class to Nested...			
			Move Type to New File...			
			Extract Superclass...			
			Extract Interface...			
			Use Supertype Where Possible...			
			Push Down...			
			Pull Up...			
			Extract Class...			
			Introduce Parameter Object...			
			Introduce Indirection...			
			Introduce Factory...			
			Introduce Parameter...			
			Encapsulate Field...			
			Generalize Declared Type...			
			Infer Generic Type Arguments...			
			Migrate JAR File...			
			Create Script...			
			Apply Script...			
			History...			

Refactoring – Gesamtzusammenfassung

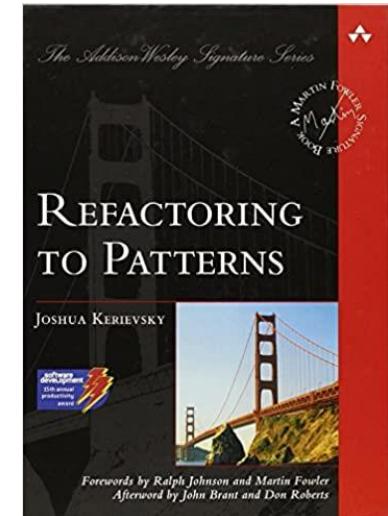
- Refactoring = verhaltenserhaltende Strukturverbesserung
 - ◆ Manuell → systematisches Vorgehen in kleinen Schritten und mit Tests
 - ◆ Automatisiert → Werkzeug überprüft Vorbedingungen und führt die entsprechenden Transformationen im gesamten Programm durch
- Bedeutung von Strukturverbesserung für Software-Qualität
 - ◆ Lesbarkeit, Verständlichkeit, Änderbarkeit
- Bedeutung von automatisierten Tests
 - ◆ Schnelles Feedback, dass Strukturänderung das Verhalten nicht geändert hat
- Bedeutung von verhaltenserhaltender Strukturverbesserung
 - ◆ Fehlerfreiheit
 - ◆ Maximale Nutzung vorhandener Tests
- Bedeutung von feingranularer Strukturverbesserung
 - ◆ Leicht Fehlerursache zu identifizieren

Weiterführende Informationen zum Thema Refactoring

- Martin Fowler:
„Refactoring – Improving the Design of Existing Code“, Addison-Wesley, 1999.
 - ◆ Umfangreicher Katalog von Refactorings und Code Smells
 - ◆ Grundlage dieses Kapitels der Vorlesung



- Joshua Kerievsky:
„Refactoring to Patterns“, Addison-Wesley
 - ◆ Wie man Patterns durch eine Sequenz von Refactorings implementieren kann
 - ◆ ... verhaltenserhaltend und somit sicher
- Alexander Shvets: <http://refactoring.guru>
 - ◆ Gute Website zu Refactorings und Patterns



Selbsttest

- Was ist Refactoring?
- Warum ist es erforderlich?
- Was ist der Nutzen von Refactoring?
- Was ist der Zusammenhang von Refactoring und Tests?
- Warum sollte man Umstrukturierung und Funktionserweiterung nicht mischen?
- Extract Method Refactoring im Detail (incl. Problemfälle)

Auf Anhang bezogen

- Weitere Beispiele von Refactorings (aus Videoverleih Fallstudie)
- Was sind „Bad smells“?
- Warum sollte man „Bad smells“ kennen?
- Beispiel von Bad Smells?
- Refactoring von APIs versus Refactoring von „geschlossenen“ Systemen

Anhang

Ab hier nicht in Vorlesung gezeigt.
Nicht prüfungs- aber praxisrelevant.

„Code Smells / Bad Smells“ – Indikationen für Refactoring

Woran erkenne ich, dass und wie ich Code ändern soll?

→ “Code Smells” beschreiben Design-Probleme und empfehlen Refactorings mit denen man sie lösen kann

Kommentare im Methoden-Rumpf

- Symptom
 - ◆ Kommentar erklärt was als nächstes geschieht
- Problem
 - ◆ Code ist offensichtlich nicht verständlich genug
- Behandlung
 - ◆ Teilmethode extrahieren (mit aussagekräftigem Namen)
 - ◆ Teilmethoden umbenennen (aussagekräftigere Namen)
 - ◆ „Assertions“ benutzen um Randbedingungen explizit zu machen
- Effekt
 - ◆ selbstdokumentierender Code
 - ◆ selbstcheckender Code (Assertions)

Fallunterscheidungen (Switch-Statements)

- Symptom
 - ◆ Fallunterscheidungen selektiert Methodenaufrufe
 - ◆ oft in Verbindung mit „Typ-Code“
- Problem
 - ◆ Redundanz: oft gleiche Fallunterscheidungen an vielen Stellen
 - ◆ schlechte Erweiterbarkeit
- Behandlung
 - ◆ Fallunterscheidungen als Teilmethode extrahieren
 - ◆ ... in Klasse verlagern zu der der Typ-Code logisch gehört
 - ◆ ... Typ-Code durch Unterklassen ersetzen
 - ⇒ „Replace Type Code with Subclasses“
 - ◆ ... jeden Fall in entsprechende Methode einer Unterklassen verlagern
 - ⇒ „Replace Conditional with Polymorphism“
 - ◆ Wenn dabei eine neue Klassenhierarchie für Typ-Code erzeugt wird
 - ⇒ „Replace Type Code with State / Strategy“

Fallunterscheidung (Fortsetzung)

- Behandlung bei wenigen, festgelegten Alternativen
 - ◆ ... wenn also keine Erweiterbarkeit erforderlich ist
 - ◆ „Replace Parameter with explicit Methods“
 - ⇒ Eigene Methode für jeden Fall
 - ⇒ Fallunterscheidung eliminieren
 - ⇒ Aufrufer ruft spezifische Methode auf, statt spezifischen Typ-Code zu setzen
- Behandlung von Tests auf „null“
 - ◆ „Introduce Null Object“
 - ⇒ Erwarteten Objekttyp um eine Unterklasse erweitern
 - ⇒ ... deren Methoden das tun, was im „null“ Fall getan werden soll
 - ⇒ Statt „null“ solche „Null-Objekte“ übergeben
 - ⇒ Fallunterscheidung eliminieren
- Effekte
 - ◆ einfacherer Code
 - ◆ keine Redundanzen
 - ◆ bessere Erweiterbarkeit

Lange Parameterliste

- Problem
 - ◆ Verständlichkeit
 - ◆ Fehleranfälligkeit
 - ◆ dauernde Änderungen
- Idee
 - ◆ Parameter-Werte aus bereits bekannten Objekten besorgen
 - ⇒ Parameter ersetzen durch Methodenaufruf an anderen Parameter oder Instanz-Variablen
 - ⇒ Parametergruppe ersetzen durch Objekt aus dem die Werte stammen, anschließend Methodenaufrufe an diesen einen Parameter
 - ⇒ Parametergruppe durch Objekt einer neuen Klasse ersetzen für ansonsten nicht zusammengehörige Parameter
- Ausnahme
 - ◆ wenn man bewusst keine Abhängigkeit zu einer bestimmten Klasse erzeugen will

Lange Parameterliste: Beispiel

Vorher

```
obj.method(w, w.get2(), x.get3(), x.get4(), y, z);
```

```
void method(a1, a2, a3, a4, a5, a6) {  
    ...  
}
```

Nachher

```
obj.method(w, x, newParam);
```

```
void method(a1, a34, a56) {  
    a2 = w.get2();  
    a3 = a34.get3();  
    a4 = a34.get4();  
    a5 = a56.getY();  
    a6 = a56.getZ();  
    ...  
}
```

- Abhängigkeiten an jeder Aufrufstelle
 - ◆ Typ von w
 - ◆ Typ von x
 - ◆ Typ von y
 - ◆ Typ von z

- Abhängigkeiten in Methode
 - ◆ Typ von w
 - ◆ Typ von x
 - ◆ Typ von newParam

Änderungsanfälligkeit („Divergente Änderungen“)

- Symptom
 - ◆ verschiedene Änderungsarten betreffen gleiche Klasse
 - ◆ Beispiel
 - ⇒ neue Datenbank: Methode 1 bis 3 in Klasse C ändern
 - ⇒ neue Kontoart: Methode 6 bis 8 in Klasse C ändern
- Behandlung
 - ◆ Klasse aufteilen
 - ⇒ C_DB Methode 1 bis 3
 - ⇒ C_Konto Methode 6 bis 8
 - ⇒ C Restliche Methoden
- Effekt
 - ◆ Lokalisierung von Änderungen

Verteilte Änderungen

- Symptom
 - ◆ eine Änderung betrifft viele Klassen
- Problem
 - ◆ schlechte Modularisierung
 - ◆ Fehleranfälligkeit
- Behandlung
 - ◆ Methoden verlagern
 - ◆ Felder verlagern
 - ◆ ... so dass Änderungen in nur einer Klasse erforderlich sind
 - ◆ Evtl. geeignete Klasse erzeugen
 - ◆ Evtl. Klassen zusammenfassen
- Effekt
 - ◆ Lokalisierung von Änderungen

Neid: „Begehre nicht deines Nächsten Hab und Gut!“

- Symptom
 - ◆ Methode die sich vorwiegend um eine bestimmte anderen Klasse „kümmert“
 - ◆ Typisch: viele „get...()-Aufrufe an andere Klasse
- Behandlung allgemein
 - ◆ „Neidische Methode“ in andere Klasse verlagern
 - ◆ evtl. „neidischen Teil“ der Methode extrahieren und verlagern
- Behandlung nicht-eindeutiger Fälle
 - ◆ Methode in Klasse verlagern die „am stärksten beneidet wird“ oder
 - ◆ verschiedene Teilmethoden in verschiedene Klassen verlagern
- Ausnahmen
 - ◆ Strategy und Visitor Pattern
 - ◆ allgemein: Bewusste Dekomposition um divergente Änderungen zu bekämpfen

Verweigertes Vermächtnis

- Symptom
 - ◆ Unterklassen nutzen geerbte Variablen nicht
 - ◆ Unterklassen implementieren geerbte Methoden so dass sie nichts tun oder Exceptions werfen
- Problem
 - ◆ Vererbung falsch angewendet
 - ◆ Subtypbeziehung nicht angebracht
- Behandlung
 - ◆ Vererbungs-Hierarchie verändern
 - ⇒ „verweigerte Anteile“ aus Oberklasse in neue Unterklasse auslagern
 - ⇒ „verweigernde Methoden“ aus anderen Unterklassen eliminieren
 - oder
 - ◆ Vererbung durch Aggregation und Forwarding ersetzen

Abschließende Bemerkungen

Refactoring und Design

Refactoring und Effizienz

Wie sag ich's dem Chef?

Wo sind die Grenzen?

Wann soll man „Refactoring“ anwenden?

- „Refactoring“ ist ständiger Teil des Entwicklungsprozesses
- „Refactoring“ findet nach Bedarf statt ...
 - ◆ keine geplante Aktivität
- ... wenn man neue Funktionen hinzufügt
 - ◆ Wenn man neue Funktionen hinzufügt und denkt, es wäre einfacher, wenn diese Funktion da und jene dort wäre, dann soll man erst umstrukturieren
- ... wenn man einen Fehler sucht
 - ◆ Durch „Refactoring“ wird der Code einfacher. Man findet leichter Bugs.

Wann soll man „Refactoring“ anwenden?

- Code-Review
 - ◆ fremden Programmierern den eigenen Code erklären
- Sinn von Code-Reviews
 - ◆ Know-how-Transfer
 - ◆ Qualitätssicherung (Fehlersuche, besseres Design, ...)
 - ◆ Verbesserungs-Vorschläge
- Refactoring während eines Code-Reviews
 - ◆ Der Code wird einfacher verständlich.
 - ◆ Verbesserungs-Vorschläge werden sofort umgesetzt.
 - ◆ Man erkennt neue Möglichkeiten
- Extreme Programming
 - ◆ permanenter Code-Review („programming in pairs“)
 - ◆ permanentes Refactoring

Refactoring und Design

- Bisher
 - ◆ Refactoring zur Wartung existierender Software
- Nun
 - ◆ Refactoring als alternative SW-Entwicklungsmethode
- Traditionelle Phaseneinteilung
 - ◆ erst Design
 - ◆ während der Implementierung das Design strikt einhalten
 - ◆ iterativ vorgehen
- „Extreme Programming“
 - ◆ minimales Design
 - ◆ während der Implementierung bei Bedarf Refactoring anwenden
 - ◆ Design und Implementierung jederzeit gemischt in kleinen Schritten
 - ◆ „extrem iterative“ Vorgehensweise

Refactoring und Indirektion

- Refactoring bedeutet meistens Indirektion
 - ◆ Zerlegung großer Methoden in kleine Methoden
 - ◆ Zerlegung großer Objekte in kleine Objekte
- Vorteile von Indirektion
 - ◆ Gemeinsame Nutzung („sharing“) von Programmlogik
 - ⇒ Methoden aus Oberklassen
 - ⇒ Teilmethoden
 - ◆ Trennung von Intention und Implementation
 - ⇒ Intention = aussagekräftige Namen
 - ⇒ Implementation nutzt Namen von Teilmethoden
 - ⇒ größtenteils selbstdokumentierender Code
 - ◆ Lokalisation von Änderungen
 - ⇒ Teilmethode
 - ⇒ Unterklasse
 - ◆ Ersatz für Fallunterscheidungen
 - ⇒ Nachrichten
- Nachteile von Indirektion
 - ◆ erhöhte Komplexität

Refactoring und Performance

- Indirektion geht auf Kosten der Laufzeit
 - ◆ Also doch kein Refactoring?

Empfehlung

- Zuerst auf gutes Design konzentrieren
 - ◆ Auch “Refactoring” gehört dazu
- Grundregeln der Optimierung (nach Dijkstra)
 - ◆ 1. Don't do it.
 - ◆ 2. Don't do it yet.
- Optimierung so spät wie möglich!
 - ◆ Voreilige Optimierungen müssen oft mit viel Aufwand rückgängig gemacht werden
- Nur häufig durchlaufene Programmteile („hot spots“) optimieren
 - ◆ Statistik: für ca. 80% der Laufzeit sind 10-20% des Codes verantwortlich
 - ◆ den Rest zu optimieren ist Zeitverschwendung
 - ◆ Profiling-Tools nutzen

Umgang mit API-Änderungen

Viele Refactorings ändern Schnittstellen: Umbenennung, Signatur-Änderung, Verschiebung:

- Ist die Menge der Clients bekannt?
 - ◆ Ja: Prüfen ob sie von der Änderung betroffen sind
 - ◆ Nein: Reparatur oder Vorbeugung
- Reparatur
 - ◆ Methoden: Forwarding-Methode mit alter Signatur einfügen („Bridge Methode“)
 - ◆ Felder: ???
- Vorbeugung
 - ◆ „published Interface“ schlank halten
 - ⇒ Variablen und Methoden so lange wie möglich „privat“ belassen
 - ◆ „code ownership“ flexibler gestalten
 - ⇒ „privates“ soll jeder ändern dürfen

Wie sag ich's dem Chef?

- Technisch kompetenter Chef
 - ◆ meist kein Problem
- Qualitäts-orientierter Chef
 - ◆ Qualitätsgewinn betonen
- Termin-orientierter Chef (Achtung: gibt evtl. vor qualitäts-orientiert zu sein)
 - ◆ Chef will termingerechte Fertigstellung
 - ◆ Wie ich das hinkriege ist meine Sache
 - Nichts sagen, einfach tun!

Grenzen des Refactoring

- Komplexe Design-Änderungen
 - ◆ Erst wohldurchdachtes Redesign versuchen!
 - ◆ Kriterium: „Was sind die Folgen und wie leicht kann die Änderung rückgängig gemacht werden?“
- „Völlig vermurkste Software“
 - ◆ Komplett neu schreiben!
- Abgabetermin steht kurz bevor
 - ◆ Nutzen des Refactoring würde zu spät zum tragen kommen

Refactoring: Ein Anwendungsbeispiel

Das motivierende Beispiel aus dem Buch von Martin Fowler.

Viele kleine Refactorings transformieren ein schlechtes Design
in ein gutes Design, das ein komplexes Entwurfsmuster umsetzt

Grundlage für die „Refactoring to Patterns“-Idee von Joshua Keriavsky (s. Literatur)

Anwendungsbeispiel: Videofilm-Verleih

- **Videos** können im Laufe der Zeit zu verschiedenen **Preiskategorien** gehören (Normal, Jugend, Neuerscheinung).
- Jede Preiskategorie beinhaltet eine andere **Preisberechnung**.
- Jede Ausleihe führt zur **Gutschrift von Bonuspunkten**, die am Jahresende abgerechnet werden.
- Der Umfang der **Gutschrift** hängt ebenfalls von der **Preiskategorie** ab.
- Für jeden Kunden soll es möglich sein, eine **Rechnung** für die ausgeliehenen Videos auszudrucken
 - ◆ Titel und Preis eines jeden ausgeliehenen Videos
 - ◆ Summe der Ausleihgebühren
 - ◆ Summe der Bonuspunkte

Erster Versuch

- Jeder **Film (Movie)** kennt seinen Titel und seine Preiskategorie
- Jede **Ausleihe (Rental)** kennt den ausgeliehenen Film und die Leihdauer
- Jeder **Kunde (Customer)** kennt die Menge seiner aktuellen Ausleihen
- ... kann den Text der dafür fälligen Rechnung selbst ermitteln (Methode `invoice()`)



Movie und Rental

```
public class Movie {  
  
    public static final int CHILDRENS = 2;  
    public static final int REGULAR = 0;  
    public static final int NEW_RELEASE = 1;  
  
    public String _title;  
    public int _priceCode;  
  
    public Movie(String title, int priceCode) {  
        _title = title;  
        _priceCode = priceCode;  
    }  
  
}
```

```
class Rental {  
  
    public Movie _movie;  
    public int _daysRented;  
  
    public Rental(Movie movie, int daysRented) {  
        _movie = movie;  
        _daysRented = daysRented;  
    }  
}
```

Customer

```
class Customer {  
  
    public String _name;  
    public Vector _rentals = new Vector();  
  
    public Customer(String name) {  
        _name = name;  
    }  
}
```

Die invoice()-Methode von Customer (Teil 1)

```
public String invoice() {  
    double totalAmount = 0;  
    int bonusPoints = 0;  
    Enumeration rentals = _rentals.elements();  
    String result = "Rental Record for "+getName()+"\n";  
  
    while (rentals.hasMoreElements()) {  
        double thisAmount = 0;  
        Rental each = (Rental) rentals.nextElement();  
  
        // determine amounts for each line  
        switch (each._movie._priceCode) {  
            case Movie.REGULAR:  
                thisAmount += 2;  
                if (each._daysRented > 2)  
                    thisAmount += (each._daysRented-2)*1.5;  
                break;  
            case Movie.NEW_RELEASE:  
                thisAmount += each._daysRented*3;  
                break;  
            case Movie.CHILDRENS:  
                thisAmount += 1.5;  
                if (each._daysRented > 3)  
                    thisAmount += (each._daysRented-3)*1.5;  
                break;  
        }  
    }  
}
```

Die invoice()-Methode von Customer (Teil 2)

```
// add frequent renter points
bonusPoints++;

// add bonus for a two day new release rental
if ((each._movie()._priceCode() == Movie.NEW_RELEASE) &&
    each._daysRented>1) bonusPoints++;

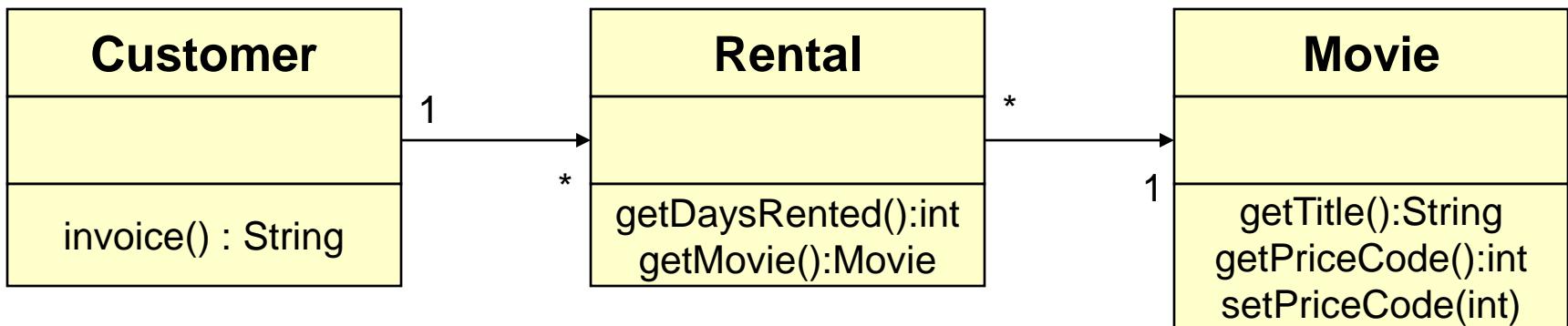
// show figures for this rental
result += "\t" + each.getMovie().getTitle()+"\t" +
          String.valueOf(thisAmount)+"\n";
totalAmount += thisAmount;
}

// add footer lines
result += "Amount owed is "+ String.valueOf(totalAmount) + "\n";
result += "You earned "+ String.valueOf(bonusPoints) +
          " frequent renter points";
return result;
}
```

Daten-Kapselung: Records mit Accessor-Methoden

Erster Versuch

- Jeder **Film (Movie)** kennt seinen Titel und seine Preiskategorie
- Jede **Ausleihe (Rental)** kennt den ausgeliehenen Film und die Leihdauer
- Jeder **Kunde (Customer)** kennt die Menge seiner aktuellen Ausleihen
- ... kann den Text der dafür fälligen Rechnung selbst ermitteln (Methode invoice())
- Instanz-Variablen sind privat, auf ihre Werte wird via Methoden zugegriffen



Movie



```
public class Movie {

    public static final int CHILDRENS = 2;
    public static final int REGULAR = 0;
    public static final int NEW_RELEASE =1;

    private String _title;
    private int _priceCode;

    public Movie(String title, int priceCode) {
        _title = title;
        _priceCode = priceCode;
    }

    public int getPriceCode() {
        return _priceCode;
    }

    public void setPriceCode(int arg) {
        _priceCode = arg;
    }

    public String getTitle() {
        return _title;
    }
}
```

Rental



```
class Rental {

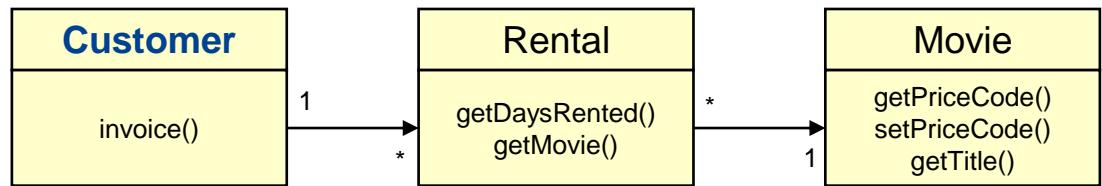
    private Movie _movie;
    private int _daysRented;

    public Rental(Movie movie, int daysRented) {
        _movie = movie;
        _daysRented = daysRented;
    }

    public int getDaysRented() {
        return _daysRented;
    }

    public Movie getMovie() {
        return _movie;
    }
}
```

Customer



```
class Customer {

    private String _name;
    private Vector _rentals = new Vector();

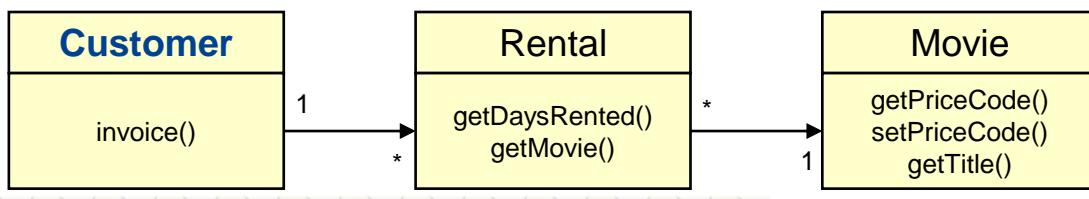
    public Customer(String name) {
        _name = name;
    }

    public void addRental(Rental arg) {
        _rentals.addElement(arg);
    }

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

Auch wenn der Typ von `_rentals` sich mal ändert, bleibt für Clients von `Customer` das Hinzufügen eines neuen Ausleihobjekts gleich.

Die invoice()-Methode (Teil 1)

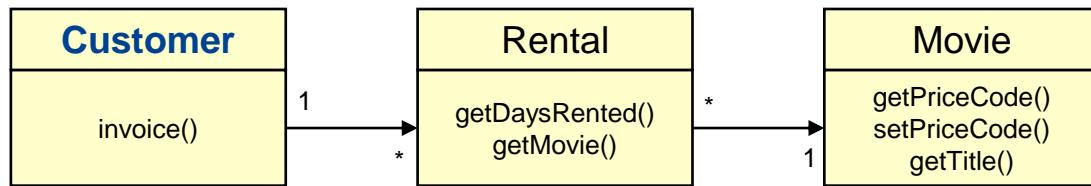


```
public String invoice() {
    double totalAmount = 0;
    int bonusPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for "+getName()+"\n";

    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        // determine amounts for each line
        switch (each.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2)
                    thisAmount += (each.getDaysRented() - 2)*1.5;
                break;
            case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented()*3;
                break;
            case Movie.CHILDRENS:
                thisAmount += 1.5;
                if (each.getDaysRented() > 3)
                    thisAmount += (each.getDaysRented() - 3)*1.5;
                break;
        }
        totalAmount += thisAmount;
        bonusPoints += each.getDaysRented();
    }
    result += "Amount owed is "+totalAmount+
              ".00\n";
    result += "You earned "+bonusPoints+
              " in bonus points";
    return result;
}
```

Die invoice()-Methode (Teil 2)



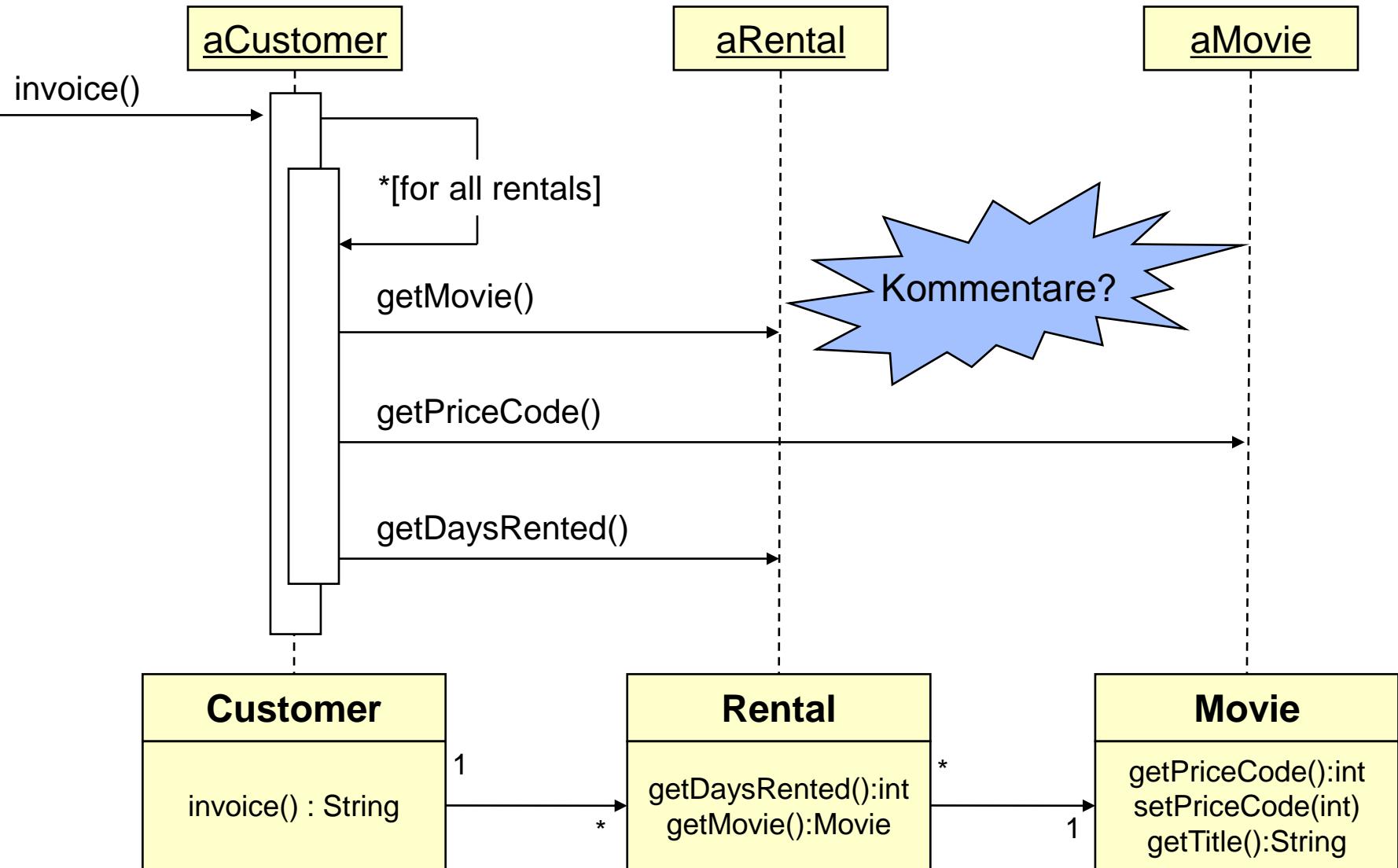
```
// add frequent renter points
bonusPoints++;

// add bonus for a two day new release rental
if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
    each.getDaysRented()>1) bonusPoints++;

// show figures for this rental
result += "\t" + each.getMovie().getTitle() + "\t" +
          String.valueOf(thisAmount) + "\n";
totalAmount += thisAmount;
}

// add footer lines
result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
result += "You earned " + String.valueOf(bonusPoints) +
          " frequent renter points";
return result;
}
```

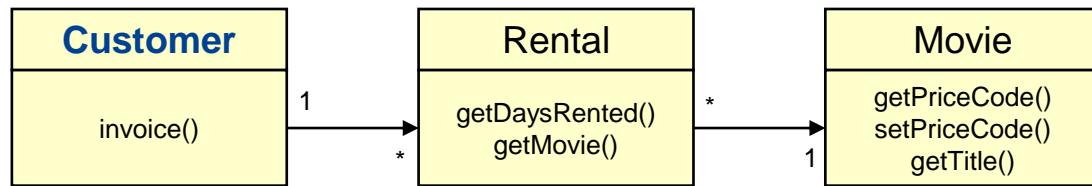
Interaktionen der invoice() Methode



Umverteilung der Verantwortlichkeiten (1)

Was gehört alles NICHT in die Invoice-Methode
... und wohin gehört es dann?

Schritte zur Besserung (1)



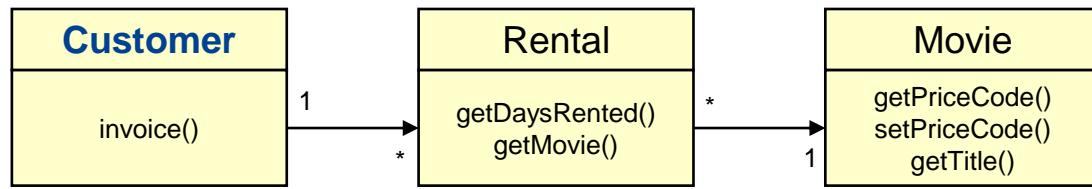
1) Invoice()-Methode aufteilen

- ◆ Berechnung von Beitrags- und Bonuspunkten **pro Ausleihe** extrahieren
- leichter verständlich
- Teile eventuell in anderen Kontexten wieder verwendbar

2) Extrahierte Methoden in passendere Klassen verlagern

- ◆ Methoden näher an "ihre" Daten (Beitrags- und Bonuspunktberechnung hängen von Preis-Code ab)
- weniger Abhängigkeiten zwischen Klassen

Schritte zur Besserung (2)



3) Invoice()-Methode weiter aufteilen

- ◆ Berechnung von **Gesamt**-Beitrags- und Bonuspunkten **für alle Ausleihen** des Kunden aus invoice() extrahieren
- Entkopplung / Trennen von Belangen („separation of concerns“)
 - Bessere Verständlichkeit, Änderbarkeit, Wiederverwendbarkeit

Voraussetzung für 3: Temporäre Variablen eliminieren

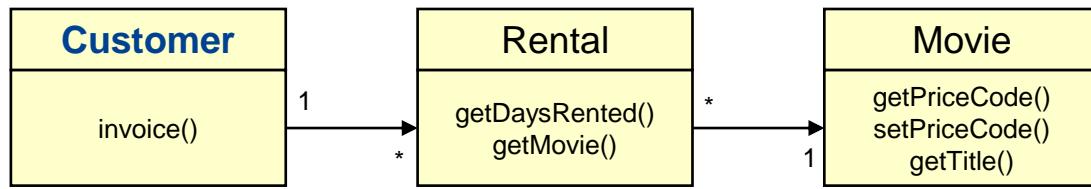
- ◆ Auslagerung von Methoden erleichtern (z.B. Summierung der Bonuspunkte)
- ◆ Vereinfachung

3) Ersetzung von Fallunterscheidungen (switch-statement) durch Nachrichten

- ◆ kleinere, klarere Methoden
- ◆ Erweiterbarkeit um zusätzliche Fälle ohne Änderung der clients einer Klasse
- ◆ z.B. zusätzliche Preiskategorien einführen ohne Änderung von invoice()

Extrahieren der Betragsberechnung

→ amountFor(Rental)

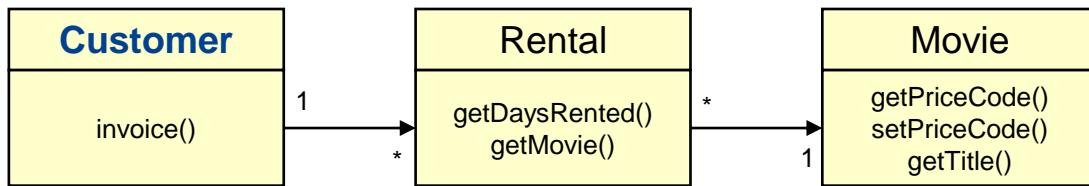


```
public String invoice() {
    double totalAmount = 0;
    int bonusPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for "+getName()+"\n";

    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        // determine amounts for each rental
        switch (each.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2)
                    thisAmount += (each.getDaysRented()-2)*1.5;
                break;
            case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented()*3;
                break;
            case Movie.CHILDRENS:
                thisAmount += 1.5;
                if (each.getDaysRented() > 3)
                    thisAmount += (each.getDaysRented()-3)*1.5;
                break;
        }
    }
}
```

Extrahieren der Betragsberechnung → amountFor(Rental)



```
public String invoice() {
    double totalAmount = 0;
    int bonusPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for "+getName()+"\n";

    while (rentals.hasMoreElements()) {
```

```
Rental each = (Rental) rentals.nextElement();
double thisAmount =
amountFor(each);
```

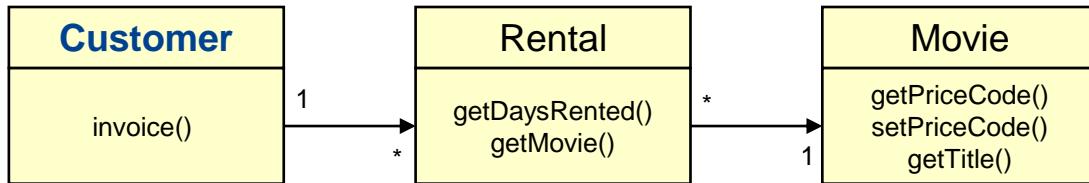
```
private double amountFor(Rental each) {
    double thisAmount = 0;
    switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            thisAmount += 2;
            if (each.getDaysRented() > 2)
                thisAmount += (each.getDaysRented()-2)*1.5;
            break;
        case Movie.NEW_RELEASE:
            thisAmount += each.getDaysRented()*3;
            break;
        case Movie.CHILDRENS:
            thisAmount += 1.5;
            if (each.getDaysRented() > 3)
                thisAmount += (each.getDaysRented()-3)*1.5;
            break;
    }
    return thisAmount;
}
```

Ändern lokaler Variablenamen:

Vorher

Im neuen Kontext
sinnvolle Namen:

- **each** → **aRental**
- **thisAmount** → **result**



```
class Customer { ...
    private double amountFor(Rental each) {
        double thisAmount = 0;
        switch (each.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2)
                    thisAmount += (each.getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented() * 3;
                break;
            case Movie.CHILDRENS:
                thisAmount += 1.5;
                if (each.getDaysRented() > 3)
                    thisAmount += (each.getDaysRented() - 3) * 1.5;
                break;
        }
        return thisAmount;
    }
}
```

Ändern lokaler Variablennamen: Nachher

Im neuen Kontext
sinnvolle Namen:

- **each** → **aRental**
- **thisAmount** → **result**



```
class Customer { ...
    private double amountFor(Rental aRental) {
        double result = 0;
        switch (aRental.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (aRental.getDaysRented() > 2)
                    result += (aRental.getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += aRental.getDaysRented() * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (aRental.getDaysRented() > 3)
                    result += (aRental.getDaysRented() - 3) * 1.5;
                break;
        }
        return result;
    }
}
```

Beitragsberechnung nach "Rental"

verlagern: Vorher



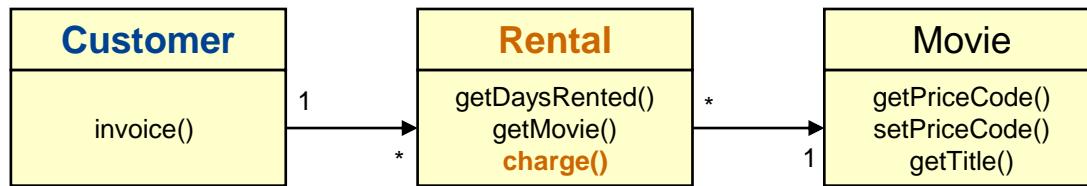
```
class Customer { ...
    private double amountFor(Rental aRental) {
        double result      = 0;
        switch (aRental.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                result      += 2;
                if (aRental.getDaysRented() > 2)
                    result      += (aRental.getDaysRented()-2)*1.5;
                break;
            case Movie.NEW_RELEASE:
                result      += aRental.getDaysRented()*3;
                break;
            case Movie.CHILDRENS:
                result      += 1.5;
                if (aRental.getDaysRented() > 3)
                    result      += (aRental.getDaysRented()-3)*1.5;
                break;
        }
        return result;
    }
}
```

Beitragsberechnung nach "Rental"

verlagern: Nachher

Private

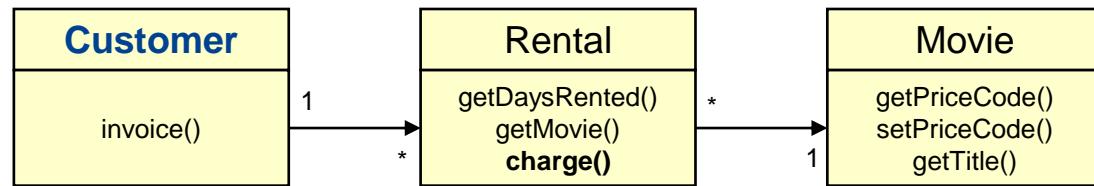
Weiterleitungsmethode kann eliminiert werden



```
class Customer { ...
    private double amountFor(Rental aRental) {
        return aRental.charge();
    }
}
```

```
class Rental { ...
    private double charge() {
        double result = 0;
        switch (getMovie().getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (getDaysRented() > 2)
                    result += (getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += getDaysRented() * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (getDaysRented() > 3)
                    result += (getDaysRented() - 3) * 1.5;
                break;
        }
        return result;
    }
}
```

Aufrufstelle anpassen: Vorher



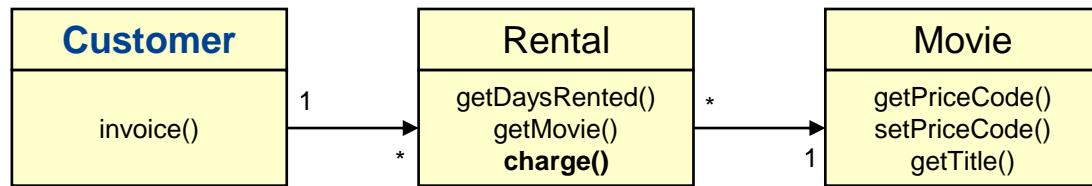
```
public String invoice() {
    double totalAmount = 0;
    int bonusPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for "+getName()+"\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();

        double thisAmount= amountFor(each);

        // add frequent renter points
        bonusPoints++;
        // add bonus for a two day new release rental
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
            each.getDaysRented()>1) bonusPoints++;

        // show figures for this rental
        result += "\t" + each.getMovie().getTitle()+"\t" +
                  String.valueOf(thisAmount)+"\n";
        totalAmount += thisAmount;
    }
    // add footer lines
    result += "Amount owed is "+ String.valueOf(totalAmount) + "\n";
    result += "You earned "+ String.valueOf(bonusPoints) +
              " frequent renter points";
    return result;
}
```

Aufrufstelle anpassen: Nachher



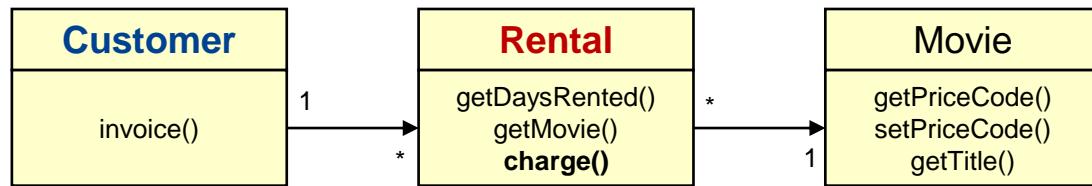
```
public String invoice() {
    double totalAmount = 0;
    int bonusPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for "+getName()+"\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();

        double thisAmount= each.charge();

        // add frequent renter points
        bonusPoints++;
        // add bonus for a two day new release rental
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
            each.getDaysRented()>1) bonusPoints++;

        // show figures for this rental
        result += "\t" + each.getMovie().getTitle()+"\t" +
                  String.valueOf(thisAmount)+"\n";
        totalAmount += thisAmount;
    }
    // add footer lines
    result += "Amount owed is "+ String.valueOf(totalAmount) + "\n";
    result += "You earned "+ String.valueOf(bonusPoints) +
              " frequent renter points";
    return result;
}
```

Bonuspunkt-Berechnung extrahieren



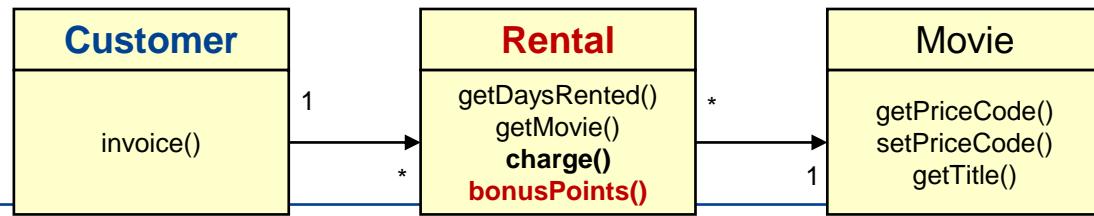
```
public String invoice() {
    double totalAmount = 0;
    int bonusPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for "+getName()+"\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();

        double thisAmount= each.charge();

        // add frequent renter points
        bonusPoints++;
        // add bonus for a two day new release rental
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
            each.getDaysRented()>1) bonusPoints++;

        // show figures for this rental
        result += "\t" + each.getMovie().getTitle()+"\t" +
                  String.valueOf(thisAmount)+"\n";
        totalAmount += thisAmount;
    }
    // add footer lines
    result += "Amount owed is "+ String.valueOf(totalAmount) + "\n";
    result += "You earned "+ String.valueOf(bonusPoints) +
              " frequent renter points";
    return result;
}
```

Bonuspunkt-Berechnung extrahieren



```

public String invoice() {
    double totalAmount = 0;
    int bonusPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for "+getName()+"\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();

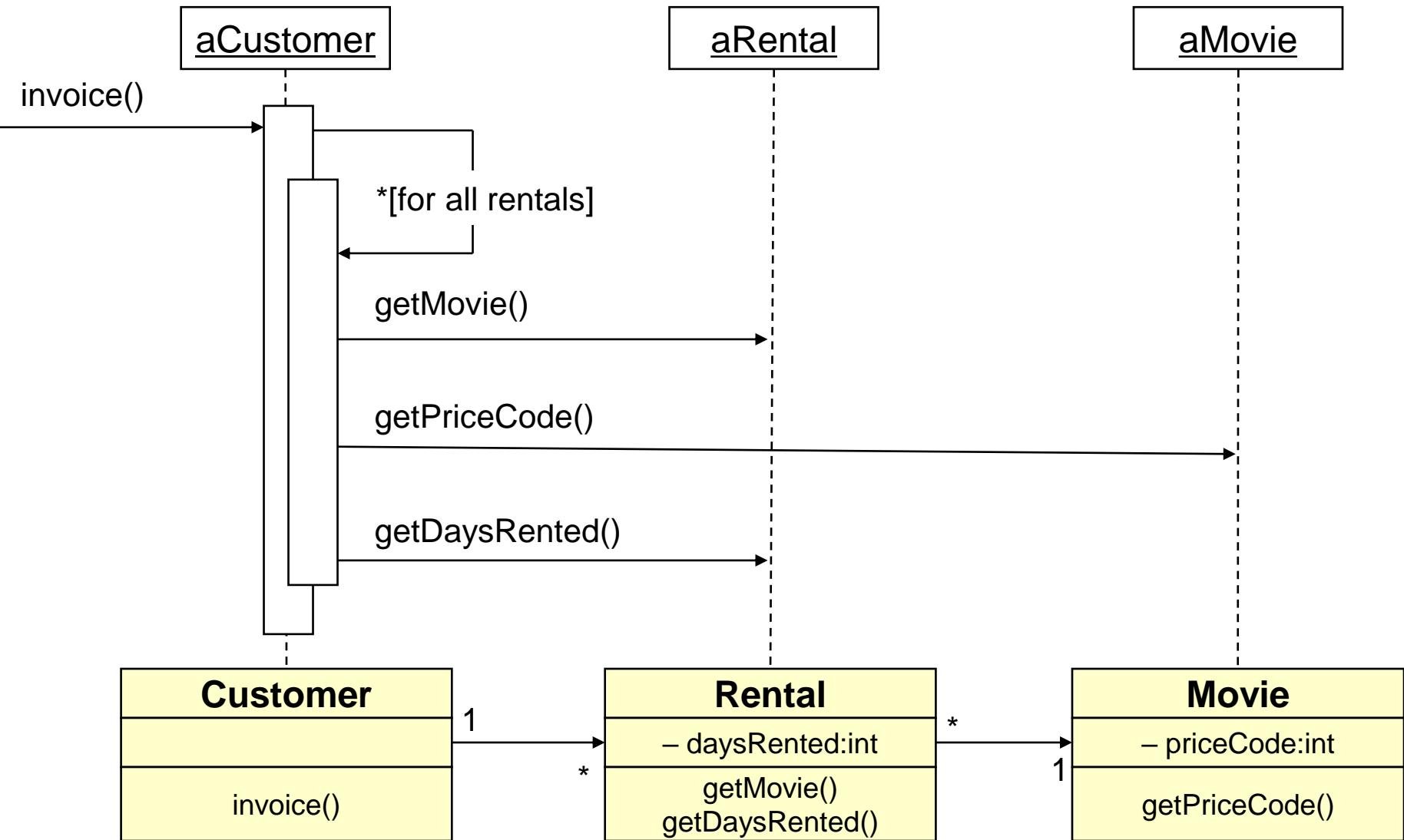
        double thisAmount= each.charge();

        bonusPoints += each.bonusPoints();

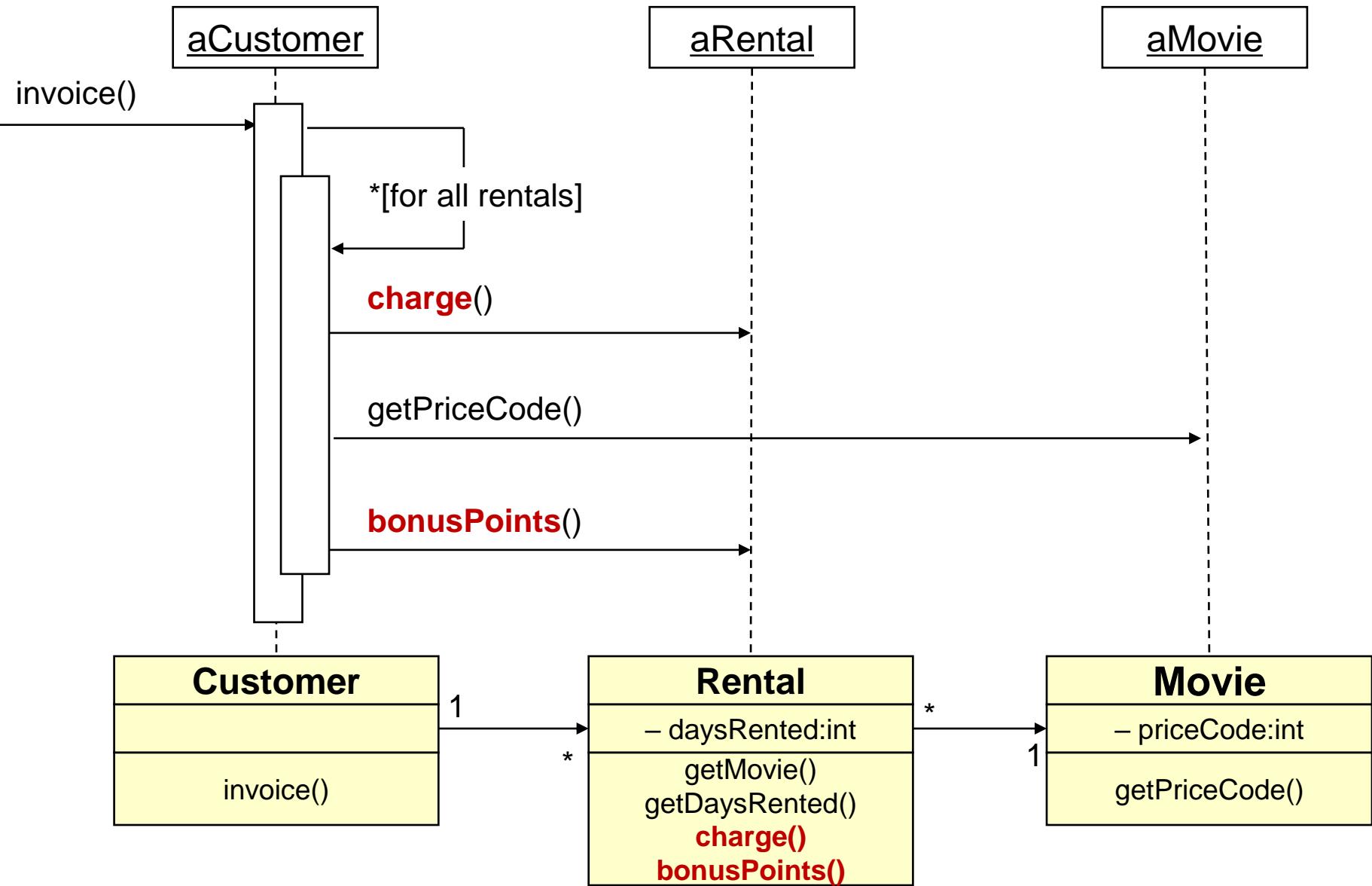
        class Rental ...
        int bonusPoints() {
            if ((getMovie().getPriceCode()==Movie.NEW_RELEASE)
                && getDaysRented() > 1)
                return 2;
            else
                return 1;
        }

        // show figure
        result += "\t"
        Str
        totalAmount +=
    }
    // add footer line
    result += "Amount owed is "+ String.valueOf(totalAmount) + "\n";
    result += "You earned "+ String.valueOf(bonusPoints) +
        " frequent renter points";
    return result;
}
  
```

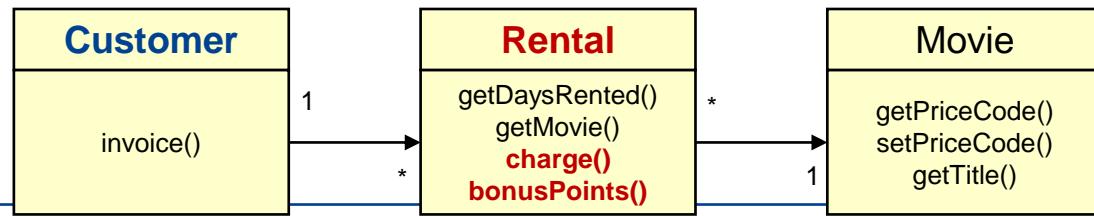
Kosten- und Bonuspunkt-Berechnung extrahieren und verlagern: Vorher



Kosten- und Bonuspunkt-Berechnung extrahieren und verlagern: Nachher

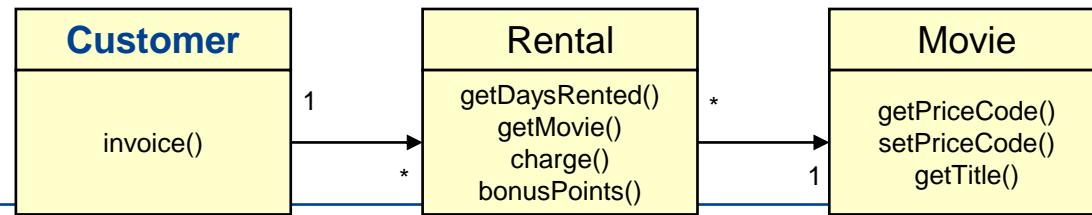


Schritte zur Besserung



- ✓ `invoice()`-Methode aufteilen
- ✓ Teilmethoden in passendere Klassen verlagern
- Temporäre Variablen eliminieren
 - ◆ Vereinfachung
 - ◆ Extraktion und Verlagerung von weiteren Methoden ermöglichen (z.B. Summierung der Bonuspunkte)
- Ersetzung von Fallunterscheidungen durch Nachrichten

Temporäre Variablen eliminieren: `thisAmount`



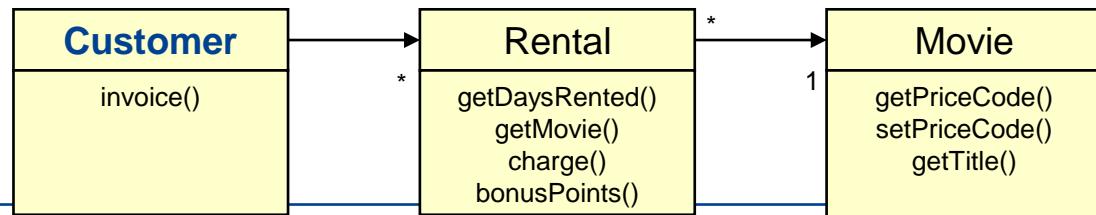
```
public String invoice() {
    double totalAmount = 0;
    int bonusPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for "+getName()+"\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();

        double thisAmount = each.charge();

        bonusPoints += each.bonusPoints();

        // show figures for this rental
        result += "\t" + each.getMovie().getTitle()+"\t" +
                  String.valueOf(thisAmount)+ "\n";
        totalAmount += thisAmount;
    }
    // add footer lines
    result += "Amount owed is "+ String.valueOf(totalAmount) + "\n";
    result += "You earned "+ String.valueOf(bonusPoints) +
              " frequent renter points";
    return result;
}
```

Temporäre Variablen eliminieren: `thisAmount`



```
public String invoice() {
    double totalAmount = 0;
    int bonusPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for "+getName()+"\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();

        bonusPoints += each.bonusPoints();

        // show figures for this rental
        result += "\t" + each.getMovie().getTitle()+"\t" +
                  String.valueOf(each.charge())+"\n";
        totalAmount += each.charge();
    }
    // add footer lines
    result += "Amount owed is "+ String.valueOf(totalAmount) + "\n";
    result += "You earned "+ String.valueOf(bonusPoints) +
              " frequent renter points";
    return result;
}
```

Eine lokale Variable wird eliminiert indem alle Verwendungen der Variablen durch den ihr zugewiesenen Ausdruck ersetzt werden.

Diesen Vorgang nennt man „*Inlining*“.

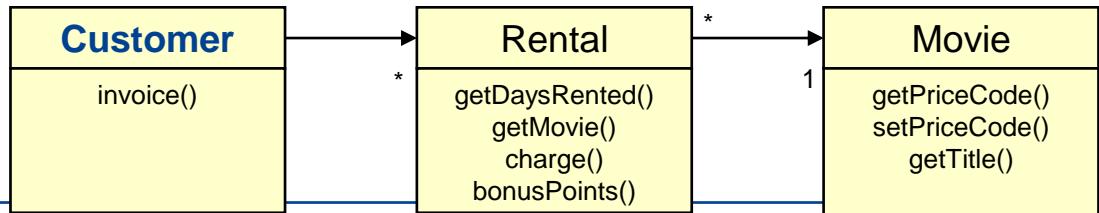
Achtung: „*Inlining*“ ist nur dann verhaltenserhaltend, falls

- der Variablen nur ein mal etwas zugewiesen wird und
- der zugewiesene Ausdruck keine Seiteneffekte hat!

Konsequenzen des Inlining

- Bessere Lesbarkeit (Kein suchen: „Wo kommt dieser Wert her?“)
- Evtl. redundante Berechnungen

Schleife splitten und temporäre Variablen eliminieren



```
public String invoice() {
    double totalAmount = 0;
    int bonusPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for "+getName()+"\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();

        bonusPoints += each.bonusPoints();

        // show figures for this rental
        result += "\t" + each.getMovie().getTitle()+"\t" +
                  String.valueOf(each.charge())+"\n";
        totalAmount += each.charge();
    }
    // add footer lines
    result += "Amount owed is "+ String.valueOf(totalAmount) + "\n";
    result += "You earned "+ String.valueOf(bonusPoints) +
              " frequent renter points";
    return result;
}
```

Wir wollen aus dieser Schleife zwei getrennte machen, die sich jeweils nur um ein Anliegen kümmern:
Bonuspunkte ODER Rechnungsbetrag.

- Klarere Verantwortlichkeiten
- Bessere Wiederverwendbarkeit (jede Funktionalität einzeln)

Schleife splitten und temporäre Variablen eliminieren

```
public String invoice() {  
  
    Enumeration rentals = _rentals.  
    String result = "Rental Record  
    while (rentals.hasMoreElements()) {  
        Rental each = (Rental) rentals.nextElement();  
  
        // show figures for this rental  
        result += "\t" + each.getMovie().getTitle() + "\t" +  
            String.valueOf(each.charge()) + "\n";  
  
    }  
    // add footer lines  
    result += "Amount owed is " + String.valueOf(totalCharge()) + "\n";  
    result += "You earned " + String.valueOf(totalBonusPoints()) +  
        " frequent renter points";  
    return result;  
}
```

Customer → Rental * → Movie

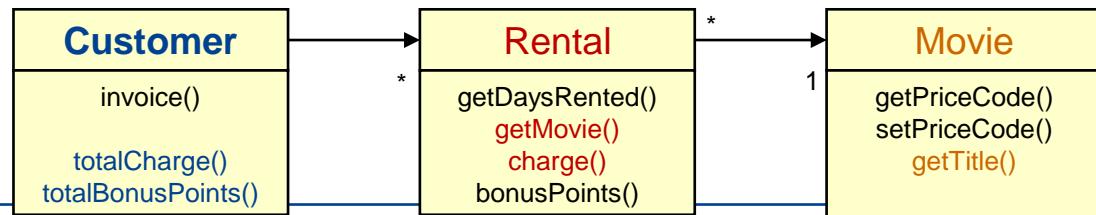
```
private double totalCharge() {  
    double result = 0;  
    Enumeration rentals = _rentals.elements();  
    while (rentals.hasMoreElements()) {  
        Rental each = (Rental) rentals.nextElement();  
        result += each.charge();  
    }  
    return result;
```

Ersatz für
totalAmount

```
private double totalBonusPoints() {  
    double result = 0;  
    Enumeration rentals = _rentals.elements();  
    while (rentals.hasMoreElements()) {  
        Rental each = (Rental) rentals.nextElement();  
        result += each.bonusPoints();  
    }  
    return result;
```

Ersatz für
bonusPoints

Endzustand der invoice()-Methode



```
public String invoice() {
    // add header line
    String result = "Rental Record for " +
                    getName() + "\n";

    // show figures for each rental
    Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        result += "\t" +
                  each.getMovie().getTitle() +
                  "\t" +
                  String.valueOf(each.charge()) +
                  "\n";
    }

    // add footer lines
    result += "Amount owed is " +
              String.valueOf(totalCharge()) +
              "\n";
    result += "You earned " +
              String.valueOf(totalBonusPoints()) +
              " frequent renter points";

    return result;
}
```

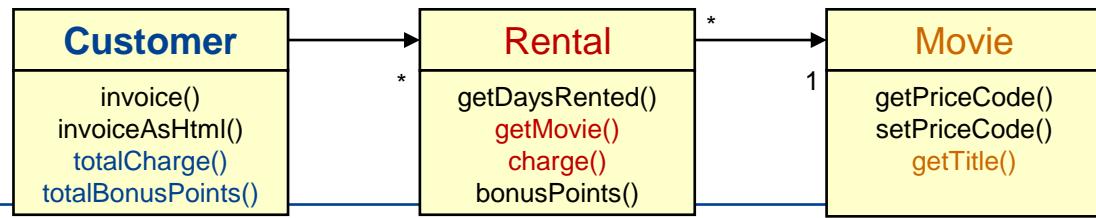
Die neu strukturierte Methode reduziert sich auf Details der Text-Formatierung.

Die eigentlichen Berechnungen wurden ausgelagert

Nutzen der bisherigen Refactorings

- Bessere Lesbarkeit / Verständlichkeit
- Klarere Verantwortlichkeiten / Weniger Abhängigkeiten
- Leichtere Implementierung von Varianten, z.B. Rechnung als HTML (s. nächste Folie)

Einfügen von invoiceAsHtml()



```
public String invoiceAsHtml() {
    // add header lines
    String result = "<H1>Rentals for <EM>" +
                    getName() + "</EM></H1><P>\n";

    // show figures for each rental
    Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        result += each.getMovie().getTitle() +
                  ": " +
                  String.valueOf(each.charge()) +
                  "<BR>\n";
    }

    // add footer lines
    result += "<P> You owe <EM>" +
              String.valueOf(totalCharge()) +
              "</EM></P>\n";
    result += "On this rental you earned <EM>" +
              String.valueOf(totalBonusPoints()) +
              "</EM> frequent renter points<P>";

    return result;
}
```

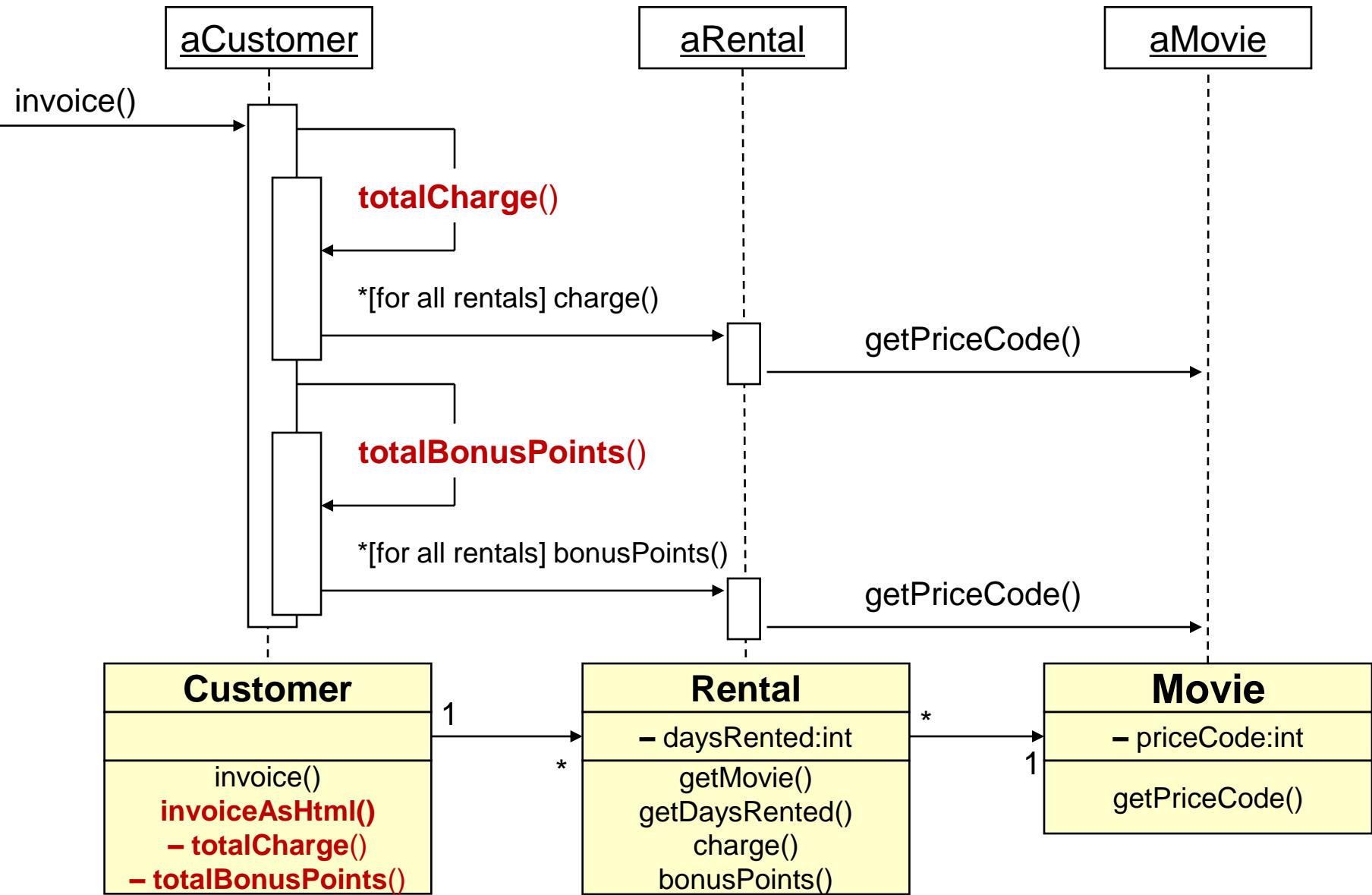
Die Änderungen in der neu erzeugten Methode reduzieren sich auf Details der HTML-Formatierung.
Die eigentlichen Berechnungen werden wiederverwendet.
Es gibt keine Redundanzen.

Umverteilung der Verantwortlichkeiten (2)

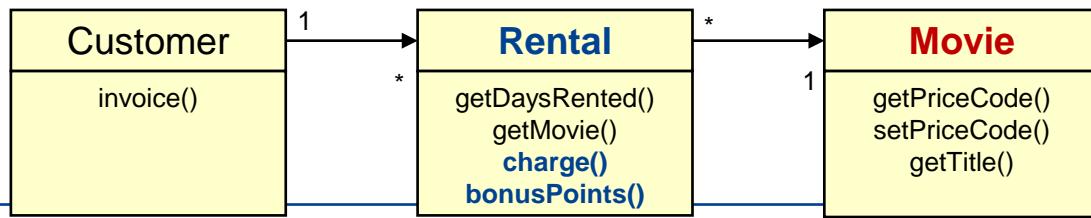
Wie können wir die Abhängigkeiten von Preis-Codes besser modellieren,
als durch case-Anweisungen?

Idee: Polymorphismus nutzen → State Pattern

Zustand nach Extraktion von totalCharge() und totalBonusPoints()



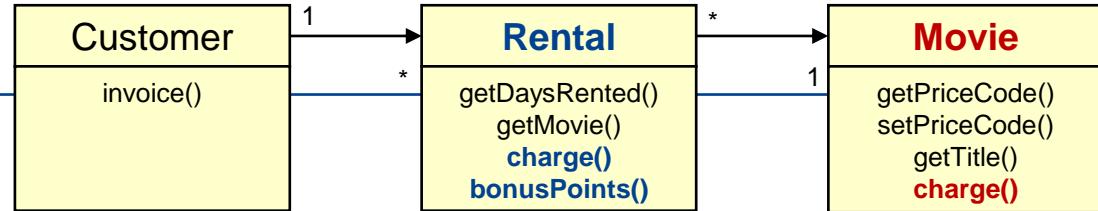
Schritte zur Besserung



- ✓ `invoice()`-Methode aufteilen
- ✓ Teilmethoden in passendere Klassen verlagern
- ✓ Temporäre Variablen eliminieren
- Vorbereitung für Einführung neuer Preiskategorien: Ersetzung der preiscodeabhängigen Fallunterscheidung durch Nachrichten
 - ◆ Verlagern der preiscodeabhängigen Methoden nach Movie (zum Preiscode)
 - ◆ Anwenden des State Patterns (jeder Zustand implementiert Methoden die preiscodeabhängige Berechnungen durchführen auf seine eigene Art)

charge()-Methode aus Rental nach Movie verlagern:

Vorher



```
class Rental ...
public double charge() {
    double result = 0;
    switch (getMovie().getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if (getDaysRented() > 2)
                result += (getDaysRented() - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:
            result += getDaysRented() * 3;
            break;
        case Movie.CHILDRENS:
            result += 1.5;
            if (getDaysRented() > 3)
                result += (getDaysRented() - 3) * 1.5;
            break;
    }
}
```

charge()-Methode aus Rental nach Movie verlagern:

Nachher

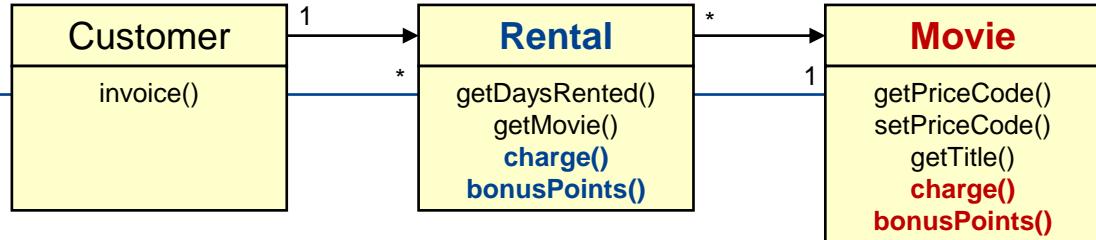


```
class Movie ...  
public double charge(int daysRented) {  
    double result = 0;  
    switch (getPriceCode()) {  
        case Movie.REGULAR:  
            result += 2;  
            if (daysRented > 2)  
                result += (daysRented - 2)*1.5;  
            break;  
        case Movie.NEW_RELEASE:  
            result += daysRented *3;  
            break;  
        case Movie.CHILDRENS:  
            result += 1.5;  
            if (daysRented > 3)  
                result += (daysRented - 3)*1.5;  
            break;  
    }  
}
```

Die in Rental-Instanzen lokal
verfügbare Information
(`daysRented`) wird nun als
Parameter übergeben

```
class Rental ...  
public double charge() {  
    return _movie.charge(_daysRented);  
}
```

bonusPoints()-Methode aus Rental nach Movie verlagern

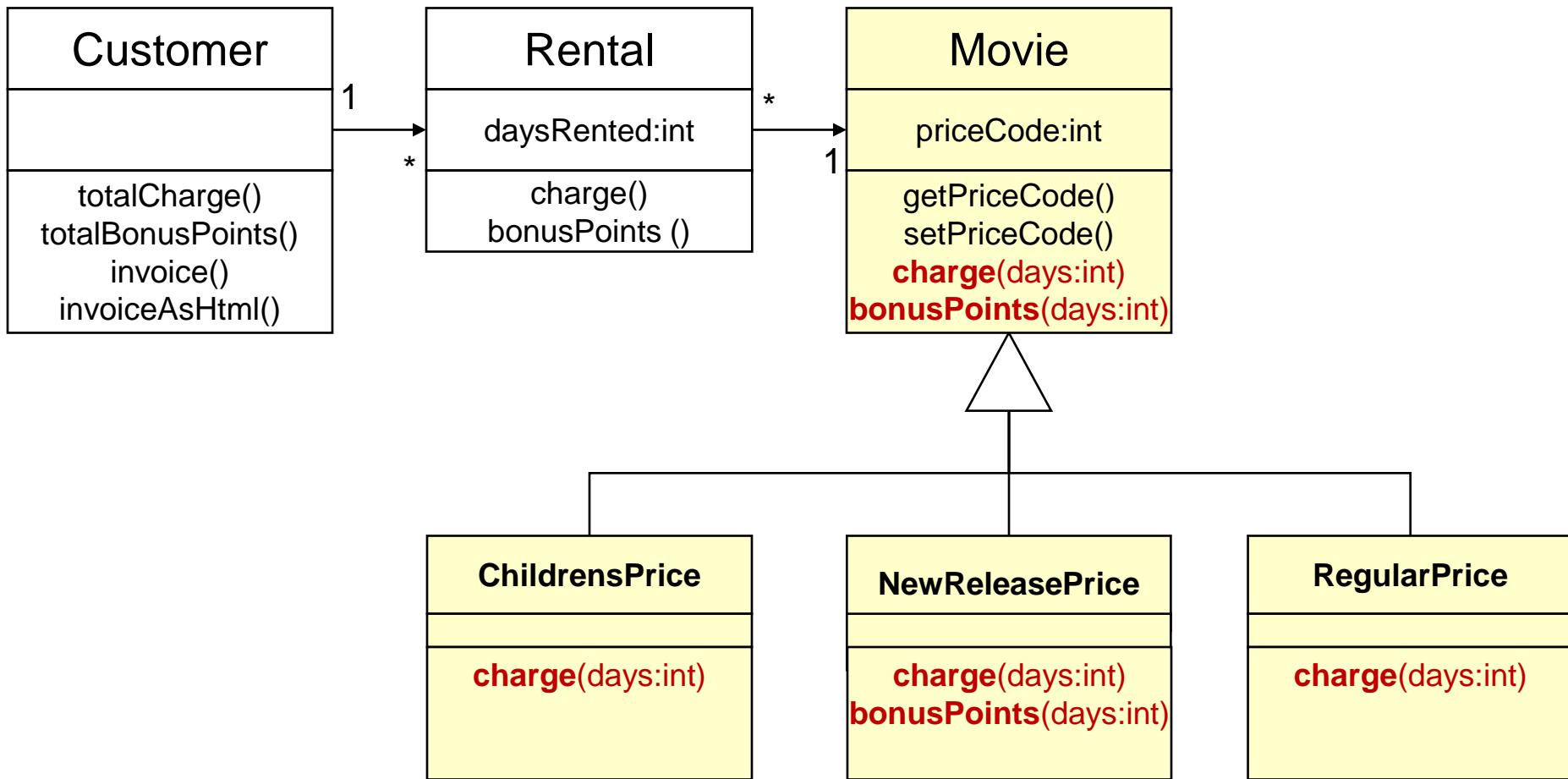


```
class Movie ...
public int bonusPoints(int daysRented) {
    if ( (this.getPriceCode() == NEW_RELEASE)
        && daysRented > 1)
        return 2;
    else
        return 1;
}
```

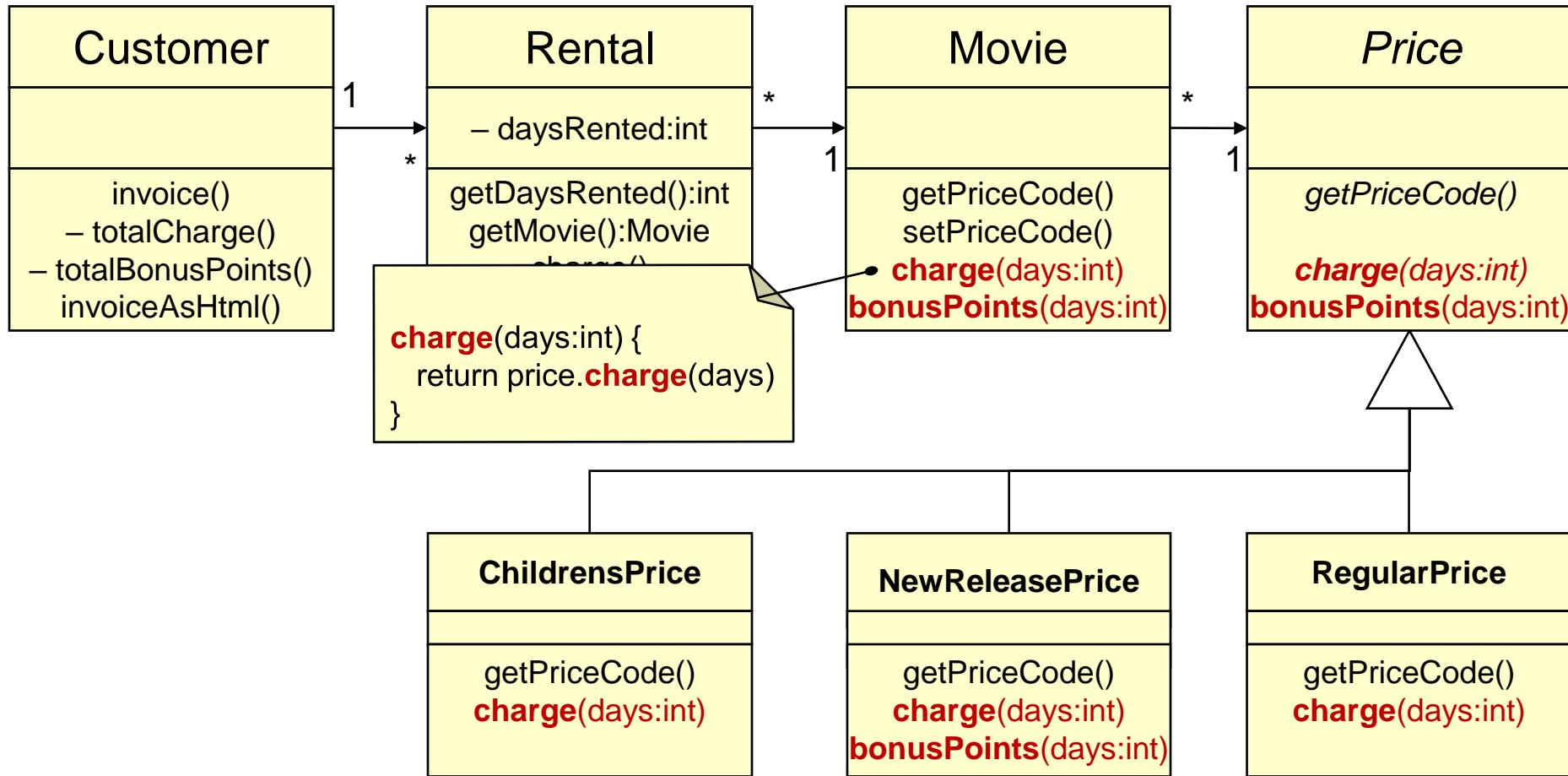
```
class Rental ...
public int bonusPoints() {
    return _movie.bonusPoints(_daysRented);
}
```

Polymorphismus via Vererbung

Hier nicht anwendbar:
ein Film hätte immer eine fixe Preiskategorie



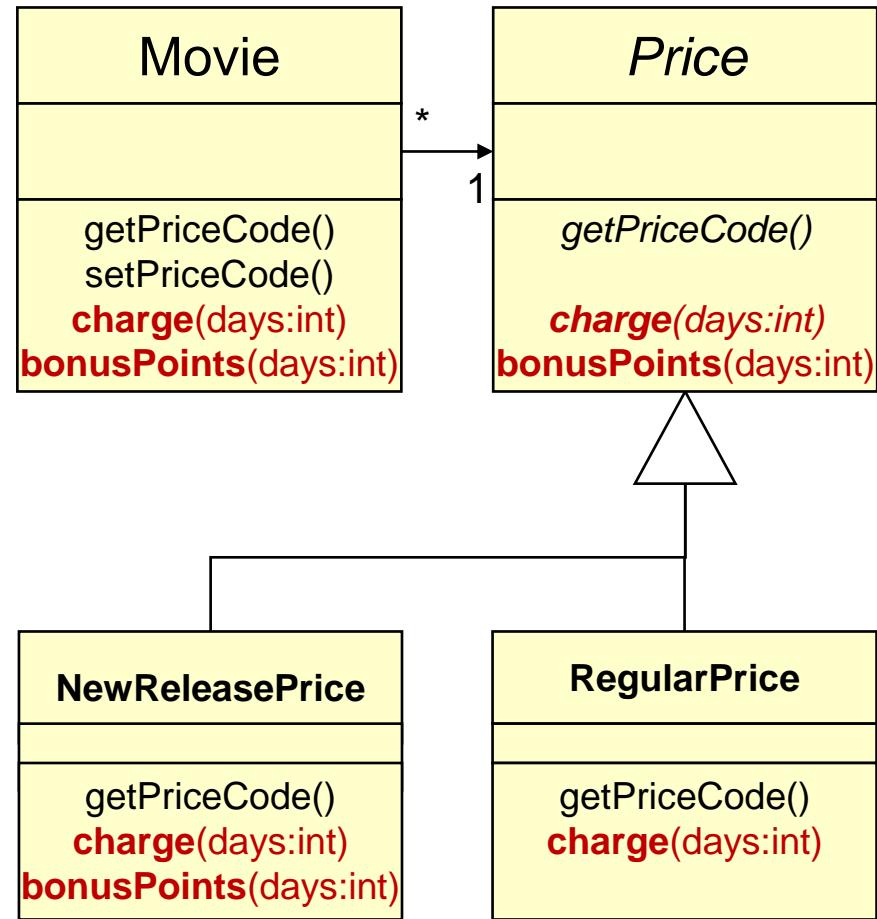
Polymorphismus via State Pattern



Polymorphismus via State Pattern

Refactoring-Schritte (Vorschau)

1. Klassen Price, ..., RegularPrice erzeugen
2. Darin getPriceCode()-Methoden implementieren
3. Ersetzung von Preis-Code durch Preis-Objekt (in Movie)
 - ◆ setPriceCode(int)
 - ◆ getPriceCode
 - ◆ Konstruktor
4. charge() und bonusPoints() von Movie nach Price verlagern
5. Fallunterscheidungen durch Polymorphismus ersetzen
 - ◆ jeden Fall der charge() Methode aus Price in die charge()-Methode einer Unterklasse auslagern
 - ◆ analog für bonusPoints()



Schritt 1-3: Ersetzung von Preis-Code durch Preis-Objekt

```
class Movie { ...  
    private int _priceCode;  
  
    public Movie(String name, int priceCode) {  
        _name = name;  
        _priceCode = priceCode;  
    }  
    public int getPriceCode() {  
        return _priceCode;  
    }  
    public void setPriceCode(int arg) {  
        _priceCode = arg;  
    }  
}
```

```
abstract class Price {  
    public abstract int getPriceCode();  
}  
class RegularPrice extends Price {  
    public int getPriceCode() {  
        return Movie.REGULAR;  
    }  
}  
class ChildrensPrice extends Price {  
    public int getPriceCode() {  
        return Movie.CHILDRENS;  
    }  
}  
class NewReleasePrice extends Price {  
    public int getPriceCode() {  
        return Movie.NEW_RELEASE;  
    }  
}
```

```
class Movie ...  
    private Price _price;  
  
    public Movie(String name, int priceCode) {  
        _name = name;  
        setPriceCode(priceCode);  
    }  
    public int getPriceCode() {  
        return _price.getPriceCode();  
    }  
    public void setPriceCode(int arg) {  
        switch (arg) {  
            case REGULAR:  
                _price = new RegularPrice();  
                break;  
            case CHILDRENS:  
                _price = new ChildrensPrice();  
                break;  
            case NEW_RELEASE:  
                _price = new NewReleasePrice();  
                break;  
            default:  
                throw new  
                    IllegalArgumentException(  
                        "Incorrect price code");  
        }  
    }
```

3

1+2

Schritt 4-5: Fallunterscheidung durch Polymorphismus ersetzen

```
class Movie { ...  
    public double charge(int daysRented) {  
        return _price.charge(daysRented);  
    }  
}
```

4

```
class Price { ...  
    public double charge(int daysRented) {  
        double result = 0;  
        switch (getPriceCode()) {  
            case Movie.REGULAR:  
                result += 2;  
                if (daysRented() > 2)  
                    result += (daysRented() - 2) * 1.5;  
                break;  
            case Movie.CHILDRENS:  
                result += 1.5;  
                if (daysRented() > 3)  
                    result += (daysRented() - 3) * 1.5;  
                break;  
            case Movie.NEW_RELEASE:  
                result += daysRented() * 3;  
                break;  
        }  
    }  
}
```

5

```
abstract class Price ...  
abstract public double charge(int days);
```

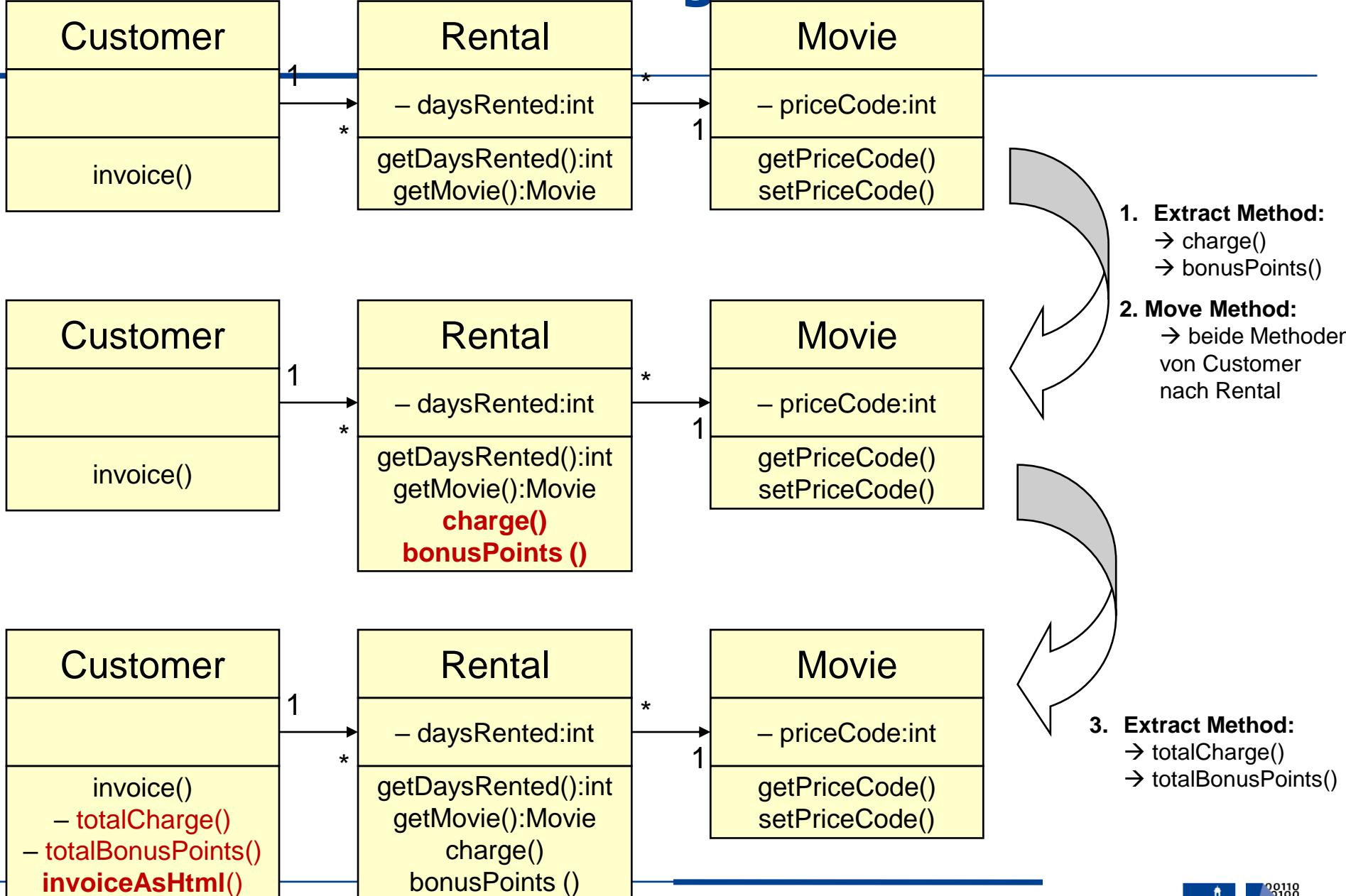
```
class RegularPrice extends Price { ...  
    public double charge(int daysRented) {  
        double result = 2;  
        if (daysRented > 2)  
            result += (daysRented - 2) * 1.5;  
        return result;  
    }  
}
```

```
class ChildrensPrice extends Price { ...  
    public double charge(int daysRented) {  
        double result = 1.5;  
        if (daysRented > 3)  
            result += (daysRented - 3) * 1.5;  
        return result;  
    }  
}
```

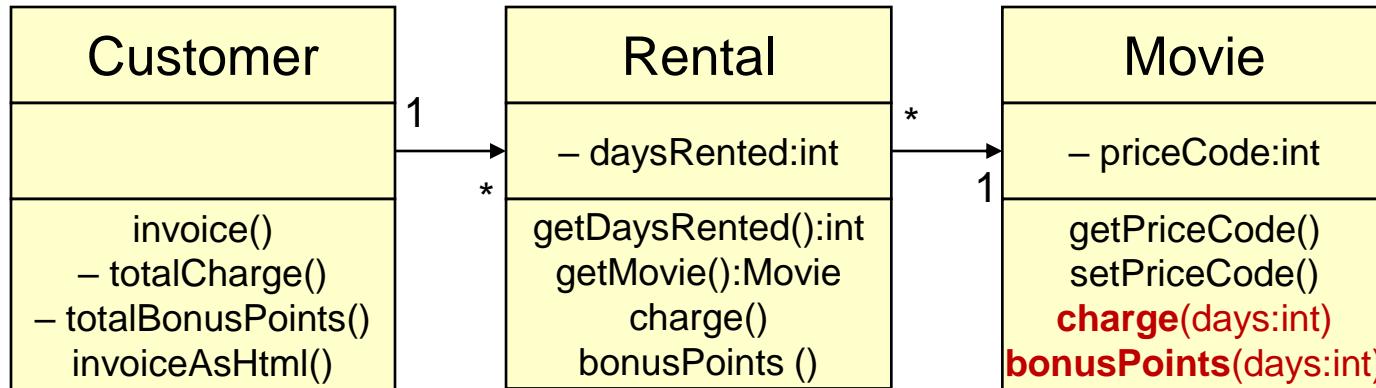
```
class NewReleasePrice extends Price { ...  
    public double charge(int daysRented) {  
        return daysRented * 3;  
    }  
}
```

Rückblick auf das Beispiel: Zusammenfassung der Refactoring-Schritte

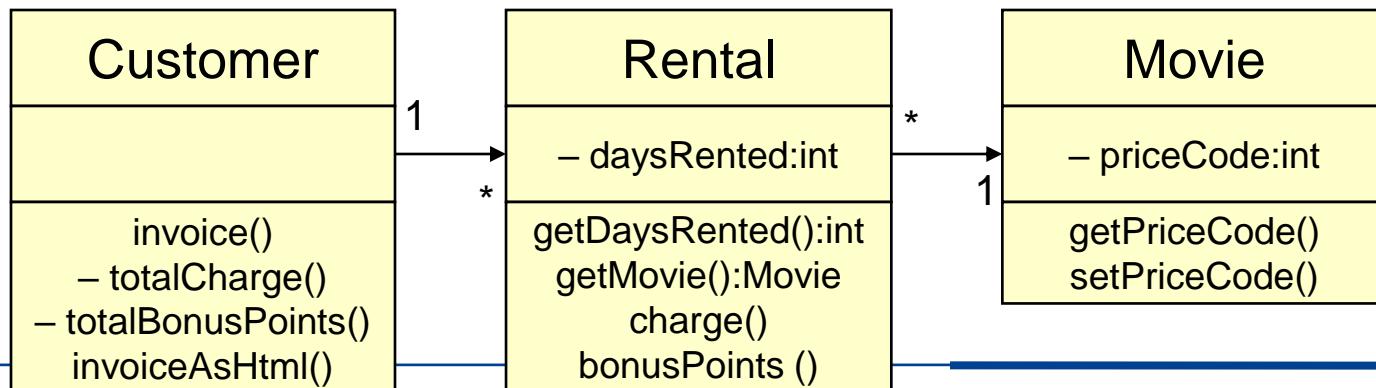
Schritte zur Besserung: Rückblick



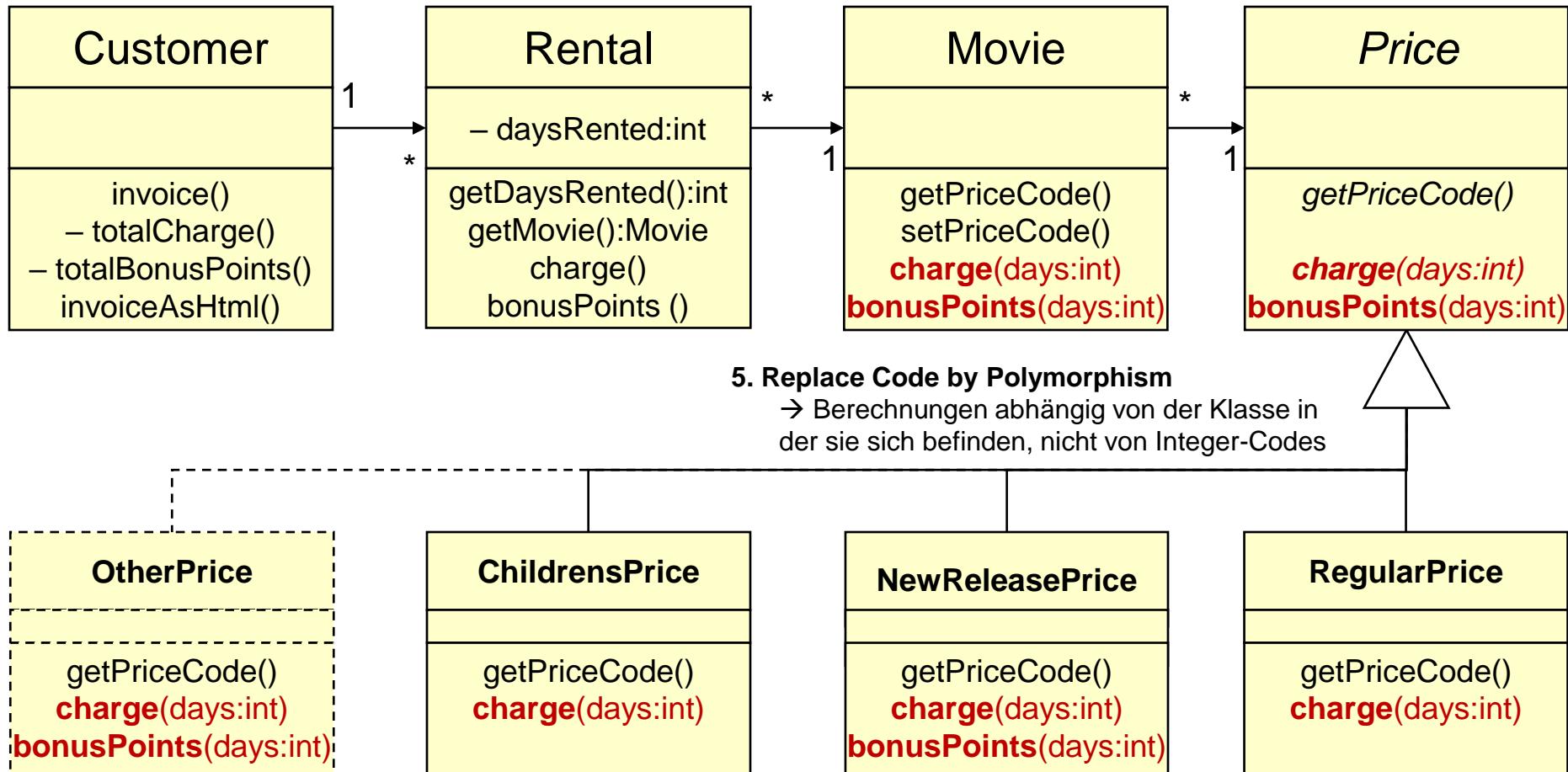
Schritte zur Besserung: Rückblick (2)



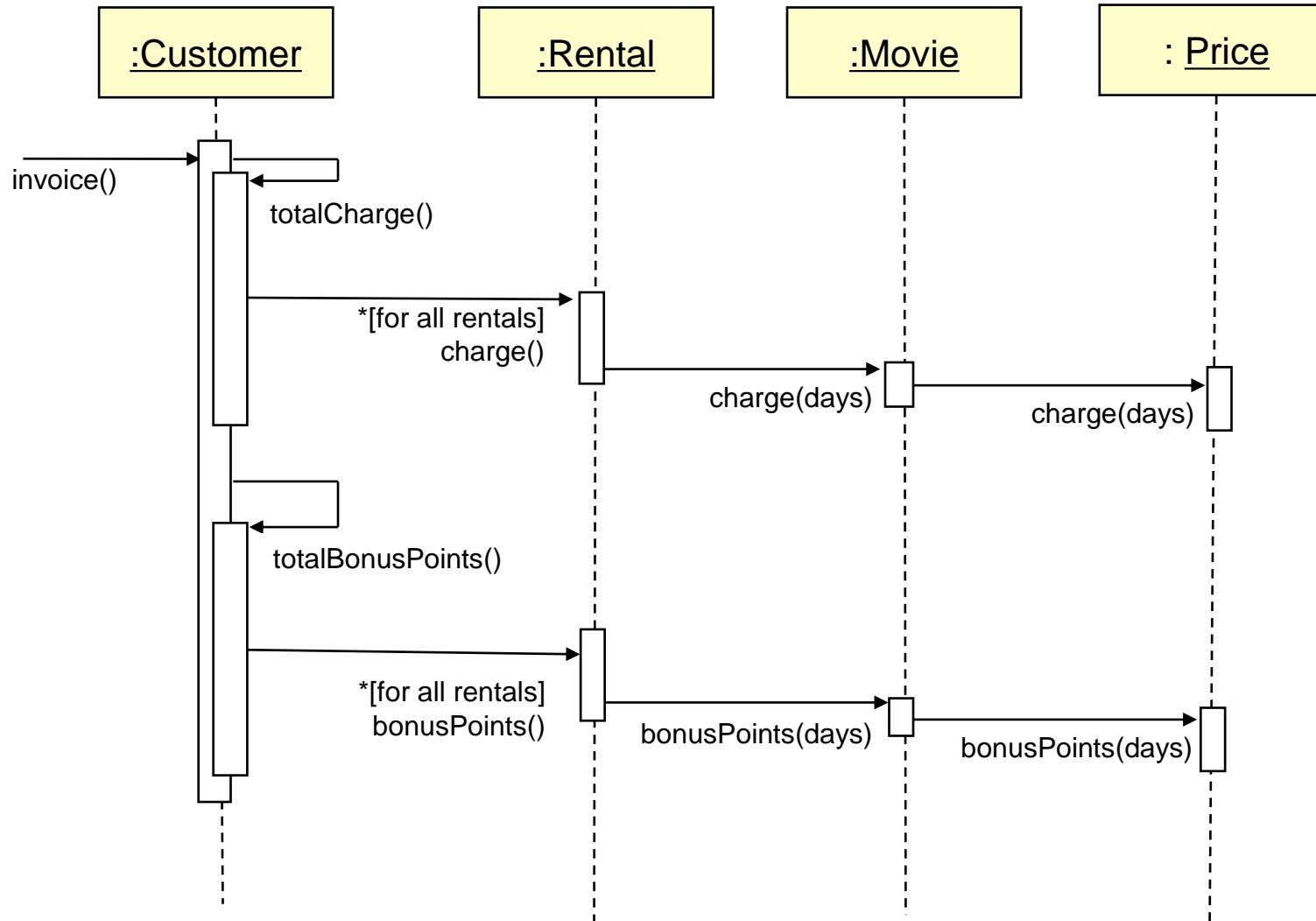
4. Move Method:
→ beide Methoden
von Rental
nach Movie



Schritte zur Besserung: Endzustand



Endzustand: Sequenz-Diagramm für invoice()-Aufruf



Rückblick auf das Beispiel: Nutzen der Refactoring-Schritte

1. Einfache Erweiterbarkeit um neue Formatierung (schon vorher gezeigt) ☺
2. Einfache Erweiterbarkeit um neue Preiskategorien ☺
3. Beide Dimensionen entkoppelt!!! ☺

Nutzen: Einfache Erweiterbarkeit (1)

```
class Customer {  
    ...  
  
    public String invoiceAsHtml() {  
        // add header lines  
        String result = "<H1>Rentals for <EM>" +  
            getName() +  
            "</EM></H1><P>\n";  
  
        Enumeration rentals = _rentals.elements();  
        while (rentals.hasMoreElements()) {  
            Rental each = (Rental) rentals.nextElement();  
            // show figures for each rental  
            result += each.getMovie().getTitle() +  
                ": " +  
                String.valueOf(each.charge()) +  
                "<BR>\n";  
        }  
  
        // add footer lines  
        result += "<P> You owe <EM>" +  
            String.valueOf(totalCharge()) +  
            "</EM></P>\n";  
        result += "On this rental you earned <EM>" +  
            String.valueOf(totalBonusPoints()) +  
            "</EM> frequent renter points<P>";  
  
        return result;  
    }  
}
```

Die neue Methode reduziert sich auf Details der HTML-Formatierung.

Die eigentlichen Berechnungen werden wiederverwendet.

Nutzen: Einfache Erweiterbarkeit (2)

```
class Movie
...
public void setPriceCode(int arg) {
    switch (arg) {
        ...
        case NEUE_KATEGORIE:
            _price = new NewCategoryPrice();
            break;
        ...
    }
}
```

```
abstract class Price {
    abstract int getPriceCode();
    abstract double charge(days:int);
    ...
}
```

```
class NewCategoryPrice extends Price {
    ...
}
```

-
- Neue Preiskategorie erfordert nur
- zusätzliche Klasse
 - Änderung bestehenden Codes an genau einer Stelle (locality of change ☺)

Themenbereiche und Vorlesungs-Kapitel

III. PROZESSE

12

Agile Softwareentwicklung

11

Software-Prozess-Modelle

II. AKTIVITÄTEN



10

Test

9

Objekt-Design

8

System-Design

7

Anforderungs-Analyse

6

Anforderungs-Erhebung

I. WERKZEUGE

5

Refactoring

4

Entwurfsmuster

3

Unified Modelling Language (UML)

2

OO-Modellierung

1

OO-Programmierung



Objektorientierte-Programmierung

Imperative-Programmierung

Automatisiertes Testing (mit jUnit)

Software Configuration Management (mit Git)

Kapitel 6

Anforderungserhebung („Requirements Elicitation“)

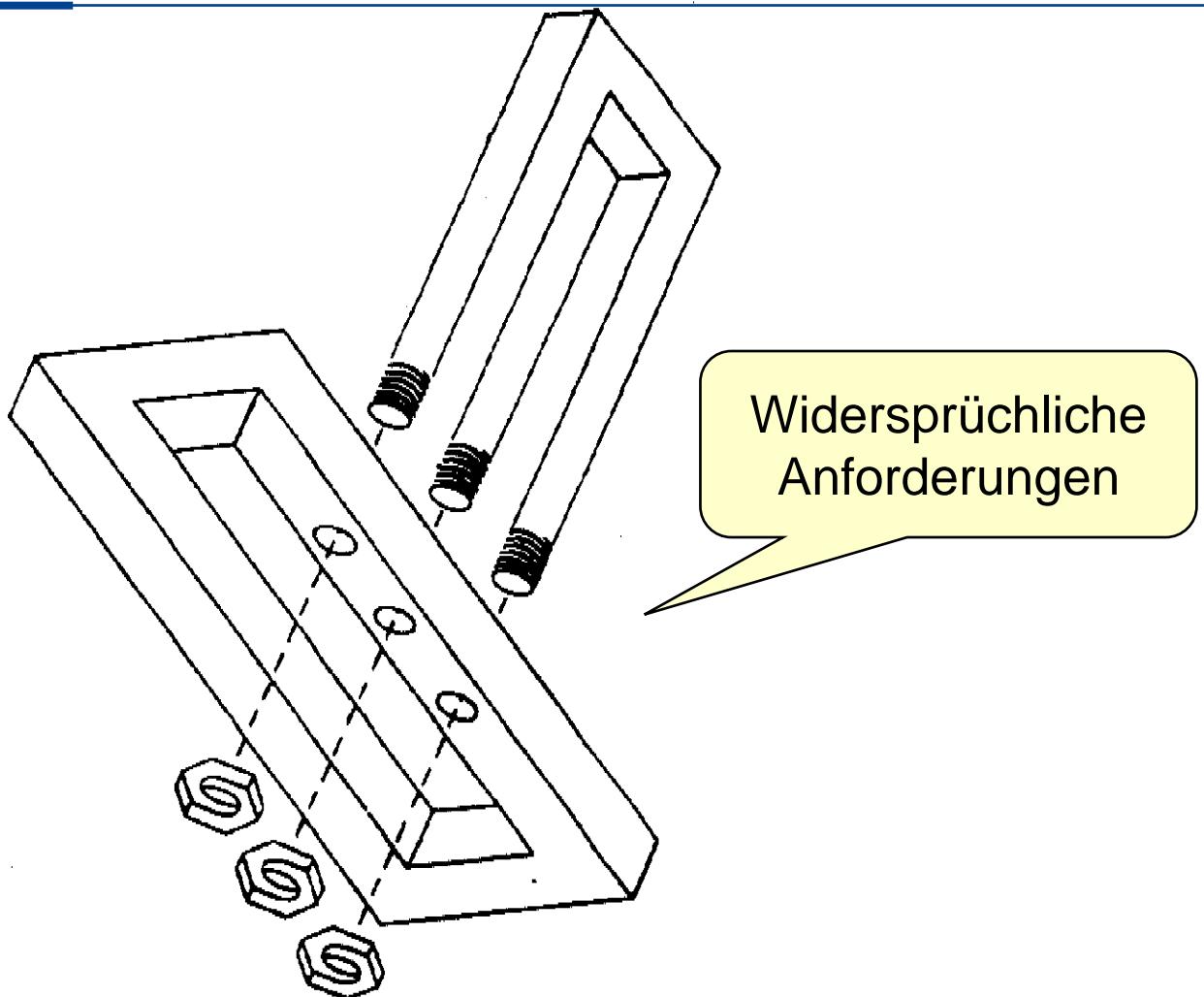
- Stand: 11.12.2023 -

-
- 6.1 Motivation und Übersicht
 - 6.2 Anforderungen, Szenarien und Anwendungsfällen („Use Cases“)
 - 6.3 Anwendungsfall-Diagramme
 - 6.4 Statische Modellierung der Anwendungsdomäne („Domain Object Model“)
 - 6.5 Dynamische Modellierung der Anwendungsfälle
 - 6.6 Zusammenfassung

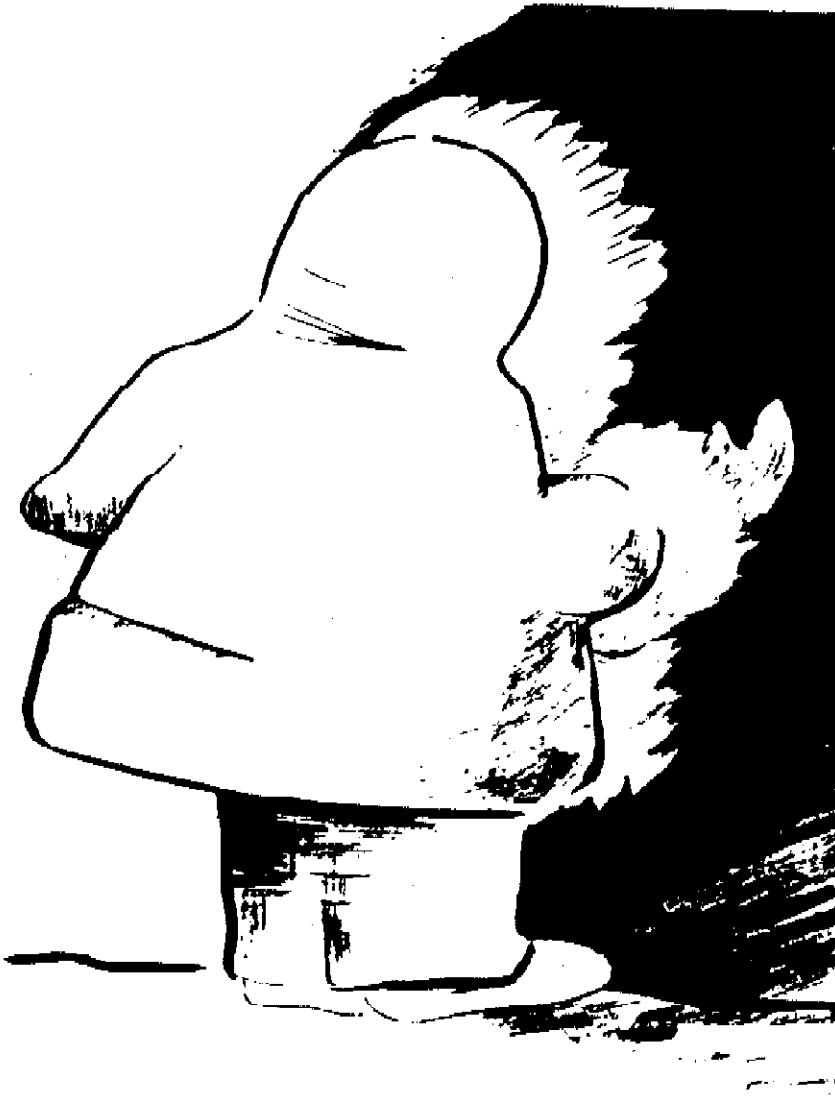
6. 1 Anforderungserhebung – Was und warum? –

Motivation und Übersicht

Können Sie so etwas bauen?

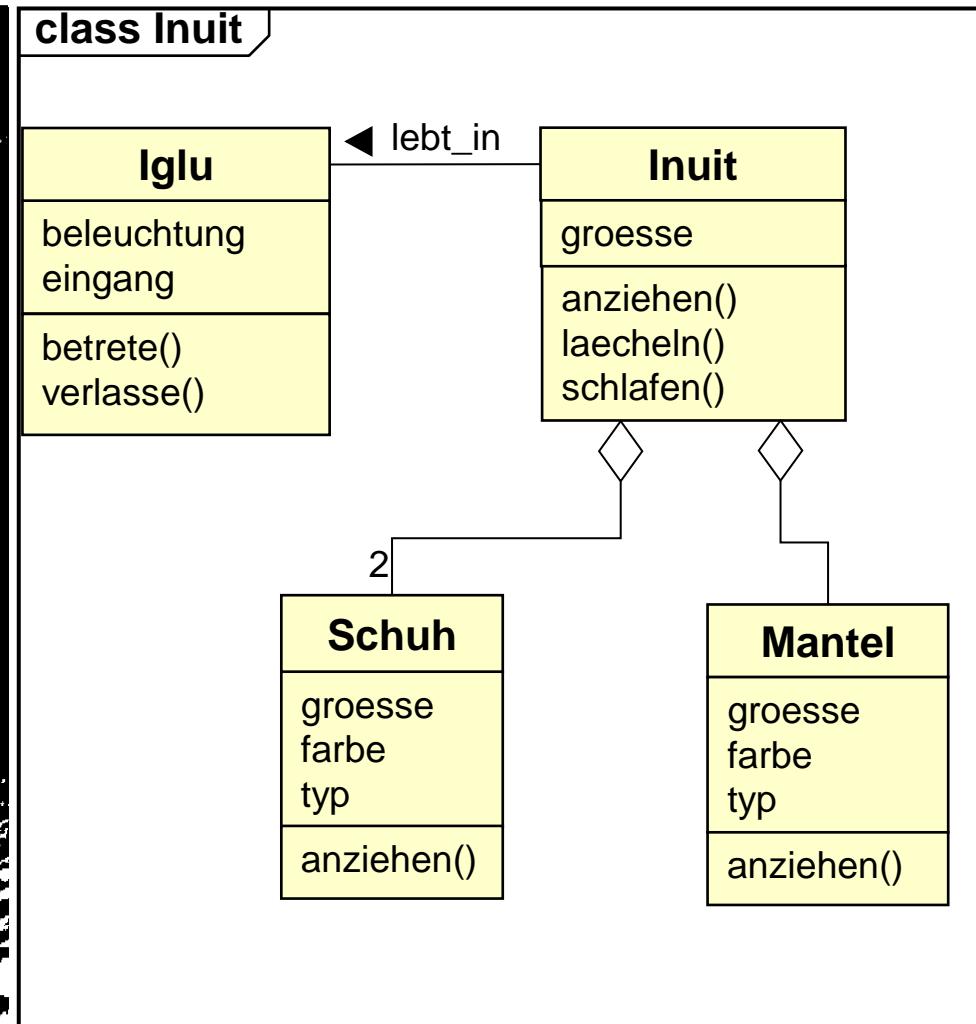
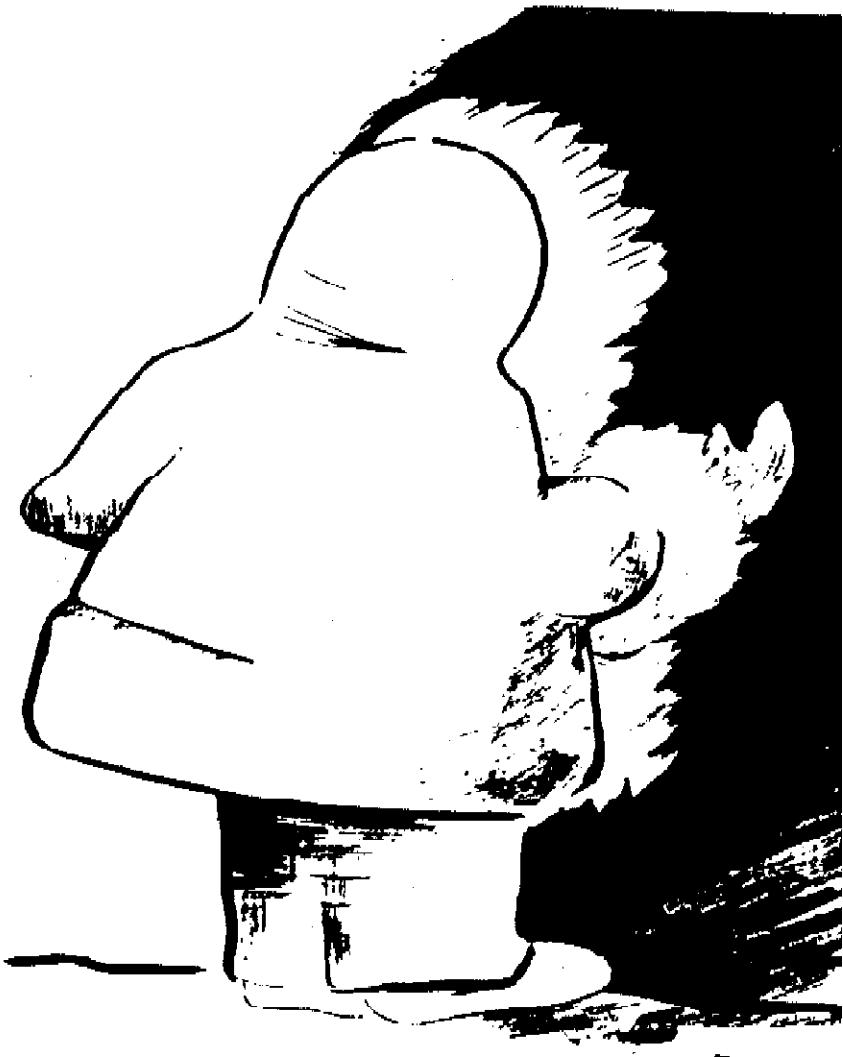


Was ist das?

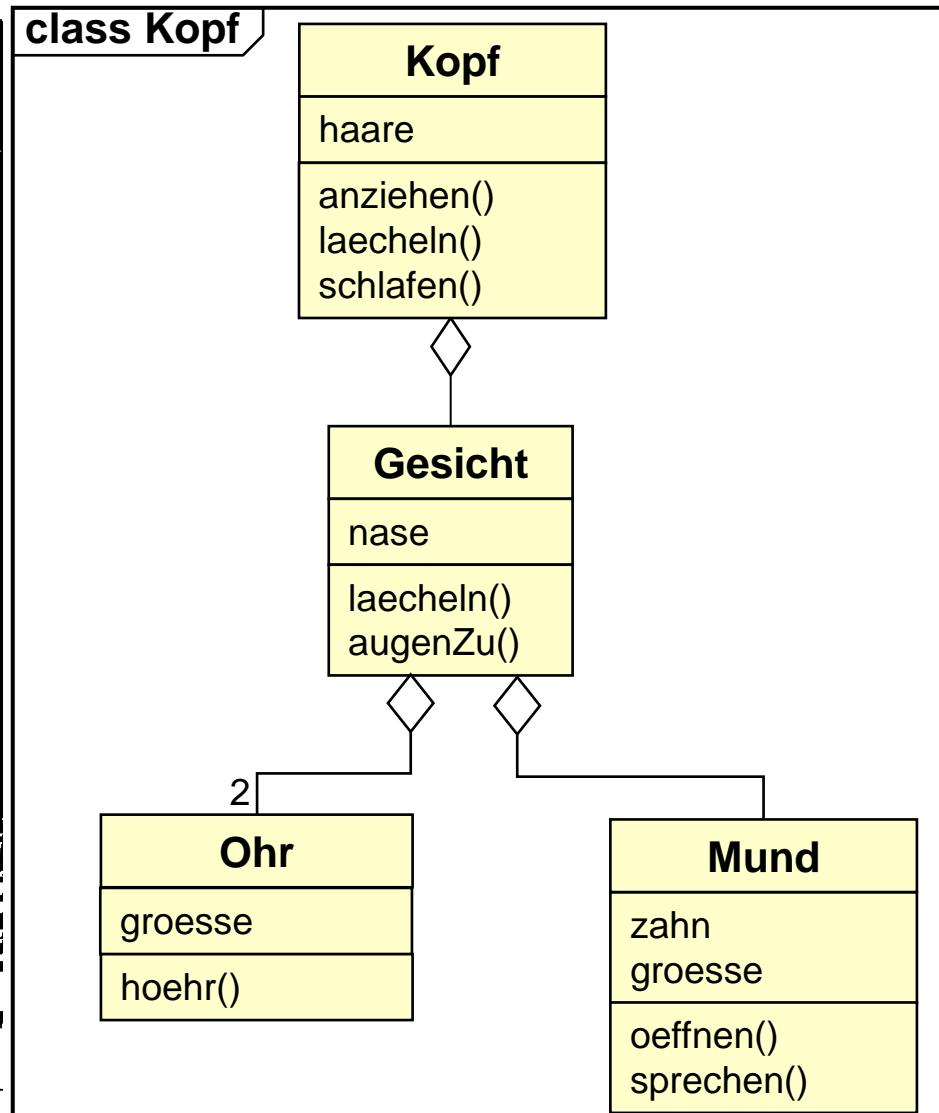
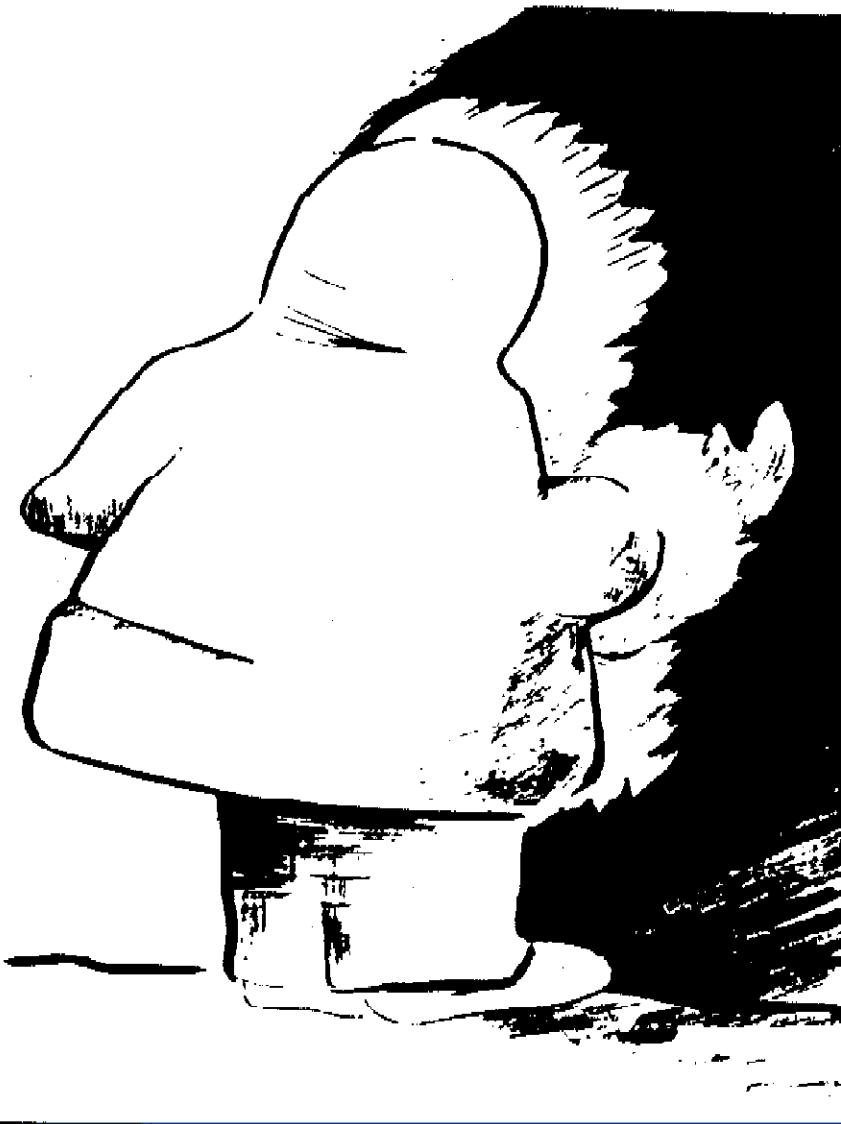


Mehrdeutige
Anforderungen

Mögliches Objektmodell ▶ Inuit

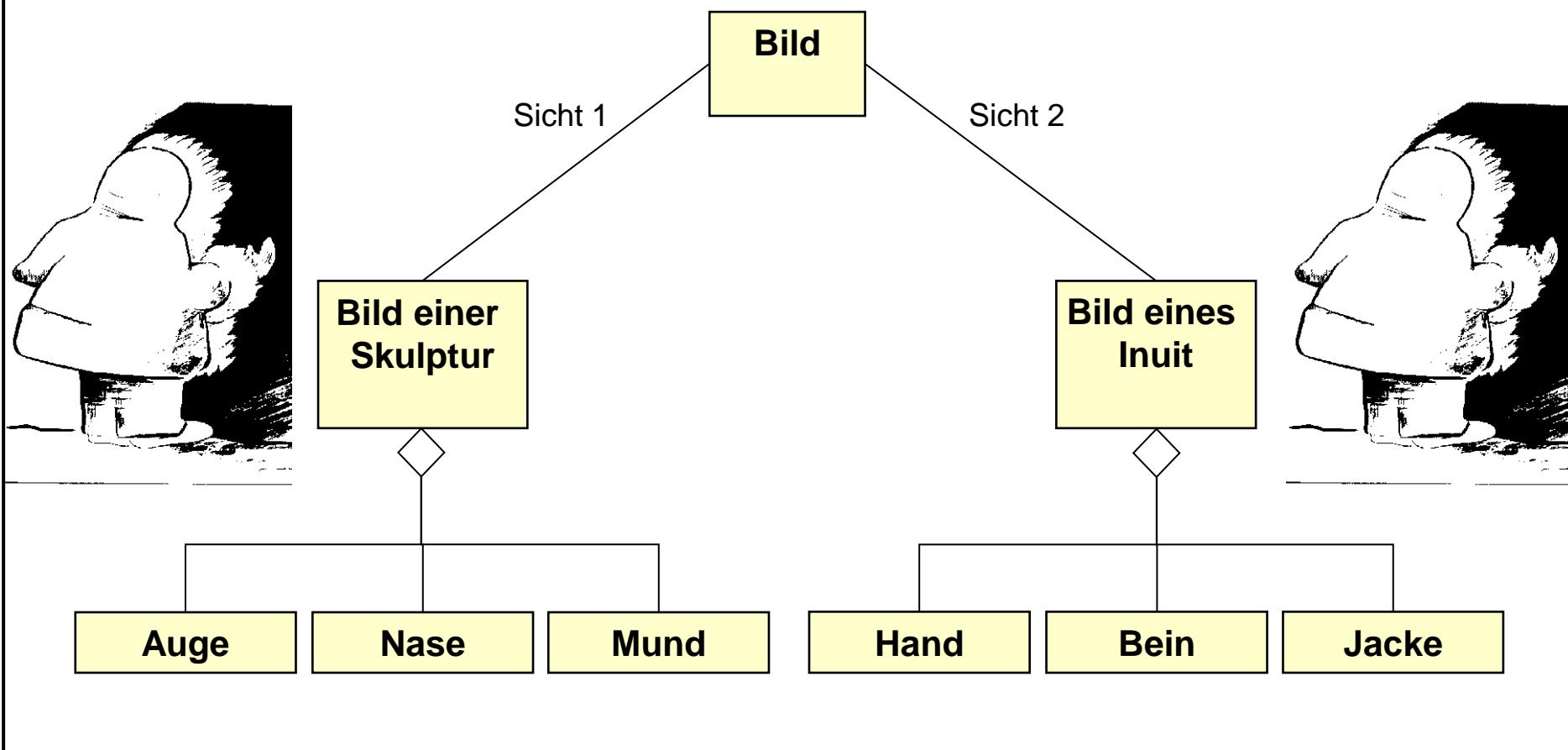


Genauso mögliches Objektmodell ▶ Kopf



Objektmodell aus der Sicht des Künstlers

class Bild



Welches System sollen Sie bauen?

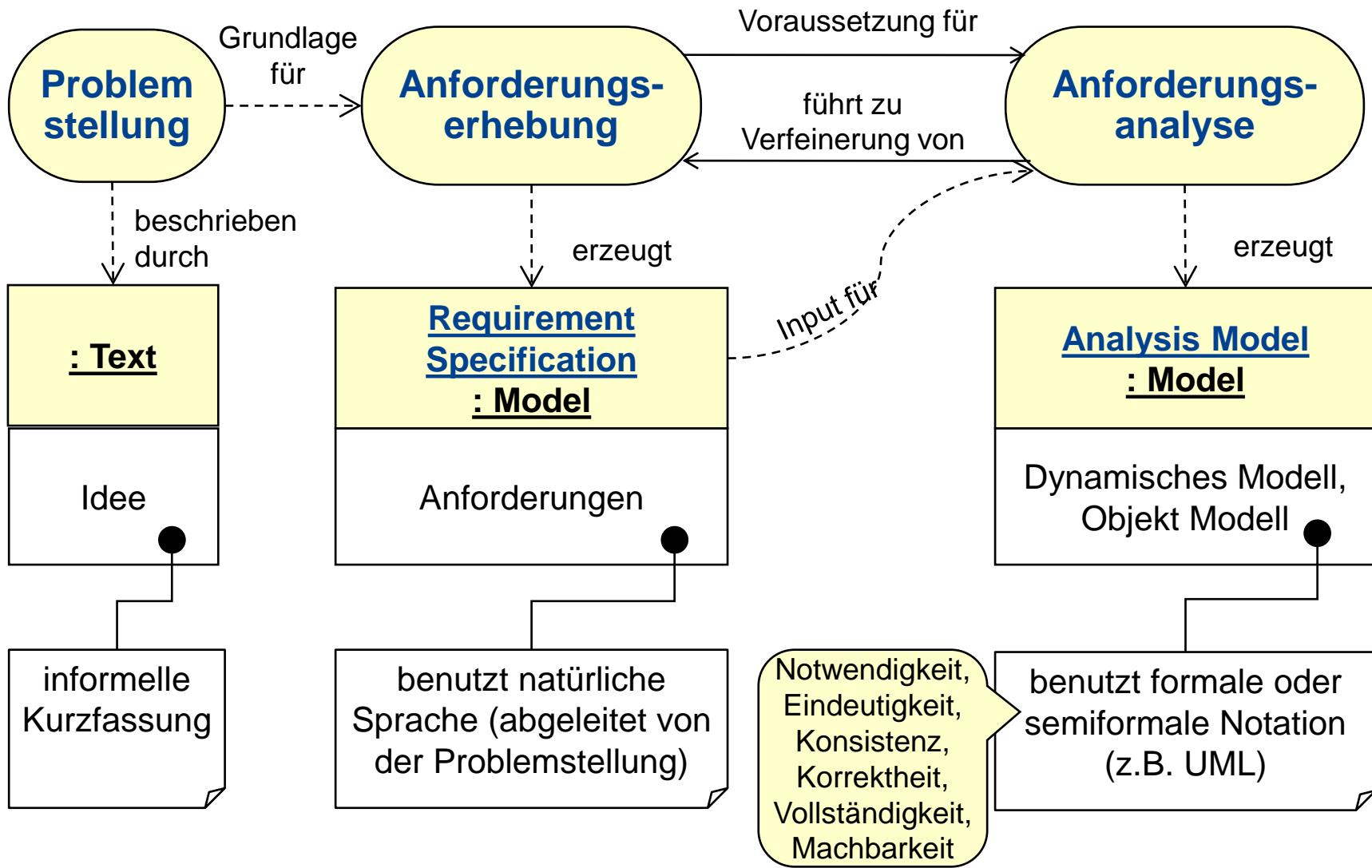


Unscharfe
Anforderungen

Was ist Requirements Engineering (RE)?

- Ziele
 - ◆ Bestimmung der **Anforderungen an das System**
 - ⇒ Identifikation der **Geschäftsprozesse**, die das System unterstützen soll
 - ⇒ Identifikation der **Funktionen** die das System dafür bieten soll
 - ◆ Kennen lernen der **Anwendungsdomäne**
 - ⇒ Identifikation von Konzepten, Beziehungen, grundlegendem Verhalten
- Aktivitäten („Workflows“)
 - ◆ Anforderungserhebung („Requirements Elicitation“) → Dieses Kapitel:
 - ⇒ Definition des Systems in einer Form, die Kunden und Entwickler verstehen
 - ◆ Anforderungsanalyse („Requirements Analysis“) → Kapitel 7:
 - ⇒ Technische Spezifikation des Systems in einer für die Entwickler verständlichen, handhabbaren und realisierbaren Form ← frei von Widersprüchen, Mehrdeutigkeiten, etc.

Der Requirements Engineering Prozess: Aktivitäten und Produkte



Arten der Anforderungserhebung

- Greenfield Engineering („Planung auf der grünen Wiese“)
 - ◆ Es existiert kein vorheriges System
 - ◆ Die Anforderungen werden vom Kunden und den Endbenutzern bestimmt
 - ◆ Ausgelöst durch Nutzerbedarf
- Reengineering
 - ◆ Re-Design und/oder Re-Implementierung eines existierenden Systems mit neuerer Technologie
 - ◆ Ausgelöst durch neue verfügbare Technologie
- Interface Engineering
 - ◆ Bietet die Dienste eines existierenden Systems in neuer Umgebung
 - ◆ Ausgelöst durch neue verfügbare Technologie oder neuen Marktbedarf

6.2 Anforderungen, Szenarien und Anwendungsfälle („Use Cases“)

Arten von Anforderungen

Arten der Anforderungserhebung

Szenarien und Anwendungsfälle („Use Cases“)

Anforderungstypen

● Funktionale Anforderungen: WAS tut das System?

- ◆ Beschreiben die Interaktionen zwischen dem System und seiner Umgebung unabhängig von der Implementierung:
 - ⇒ „Die Uhr muss die Zeit ortsabhängig anzeigen“

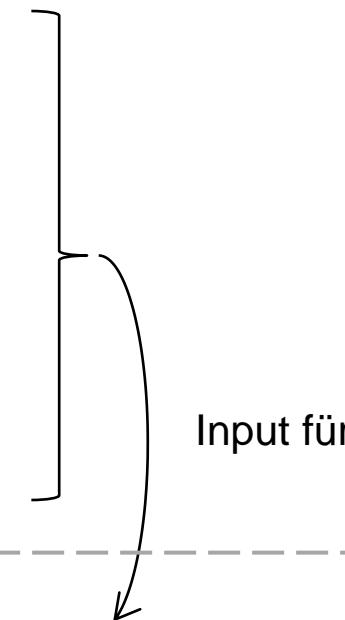
● Nichtfunktionale Anforderungen: WIE tut es das?

- ◆ Für den Nutzer sichtbare Aspekte des Systems, welche kein funktionales Verhalten sind:
 - ⇒ „Die Reaktionszeit muss unter einer Sekunde liegen“
 - ⇒ „Die Genauigkeit muss innerhalb einer Sekunde sein“

● Nebenbedingungen: Vorgaben hinsichtlich Umsetzung

- ◆ Alles was uns in der Umsetzung des Was und Wie einschränkt
 - ⇒ „Muss mit dem Abfertigungssystem von 1956 zusammenarbeiten.“
 - ⇒ „Die Implementierung muss in COBOL erfolgen unter Verwendung von DB2.“

Anforderungserhebung ▶ Gesamtüberblick

- 1. Sammle Anforderungen
 - ◆ Identifiziere funktionale Anforderungen
 - ◆ Identifiziere nichtfunktionale Anforderungen
 - ◆ Identifiziere Nebenbedingungen
 - 2. Entwickle das funktionale Modell
 - ◆ Entwickle Use Cases zur Illustration der funktionalen Anforderungen
-
- 3. Entwickle das Objektmodell der Anwendungsdomäne
 - ◆ Klassendiagramme, die die relevante Konzepte der Domäne beschreiben
 - 4. Entwickle das dynamische Modell der Anwendungsdomäne
 - ◆ Interaktionsdiagramme für das Zusammenspiel von Objekten
 - ◆ Zustandsdiagramme für die internen Abläufe von Objekten
 - ◆ Aktivitätsdiagramme für Geschäftsprozesse
- 
- Input für

Motivation: Szenarien und Anwendungsfälle („Use Cases“)

- Herausforderung der Anforderungserhebung:
Zusammenarbeit von Leuten mit unterschiedlichem Hintergrund
 - ◆ Nutzer mit Wissen über die Anwendungsdomäne
 - ◆ Entwickler mit Wissen über die Implementierungsdomäne
- Problem: Überbrücken der Lücke zwischen Nutzer und Entwickler
- Lösungsansatz: Beispielbasiertes Vorgehen (bottom-up)
 - ◆ Szenario:
Beispiel der Nutzung des Systems in Form einer Reihe von Interaktionen zwischen externen Akteuren und dem System
 - ◆ Anwendungsfall (engl. Use Case, UC):
Abstraktion, welche eine Klasse von Szenarien beschreibt

Szenario

- “Eine erzählende Beschreibung dessen, was Menschen tun und erfahren wenn sie versuchen, Computersysteme und Anwendungen zu benutzen”
[John M. Carroll, Scenario-based Design, Wiley, 1995]
- Eine Beschreibung einer einzelnen Funktionalität eines Systems, aus Sicht eines einzelnen Akteurs die
 - ◆ konkret = ein konkretes Beispiel
 - ◆ fokussiert = nur auf eine Funktionalität bezogen
 - ◆ und informell = umgangssprachlichist.

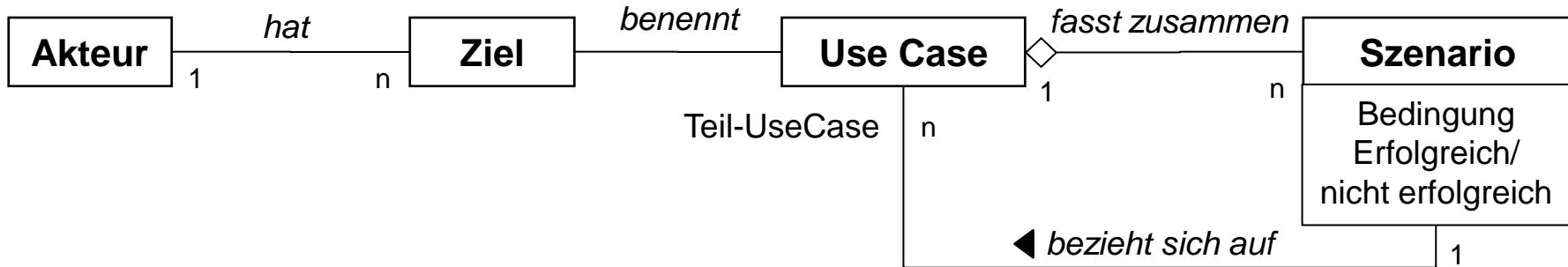
Arten von Szenarien

Szenarien können während eines Software-Lebenszyklus an vielen Stellen Anwendung finden.

- As-is Szenario
 - ◆ Zur Beschreibung einer momentanen Situation. Normalerweise während des Reengineering genutzt. Der Benutzer beschreibt das System.
- Visionäres Szenario
 - ◆ Zur Beschreibung eines zukünftigen Systems (beim Greenfield Engineering oder Reengineering). Erstellung erfordert meistens Kooperation von Benutzer und Entwickler
- Evaluationsszenario
 - ◆ Benutzeraufgaben, anhand derer das System bewertet wird
- Trainingsszenario
 - ◆ Schritt-für-Schritt Führung eines Neulings durch das System

Akteur – Ziel – UseCase - Szenario

- Ein Akteur hat Verantwortlichkeiten aus denen sich **Ziele** ergeben
- Ein **Ziel** fasst eine Systemfunktion in einer verständlichen, überprüfbaren Weise zusammen

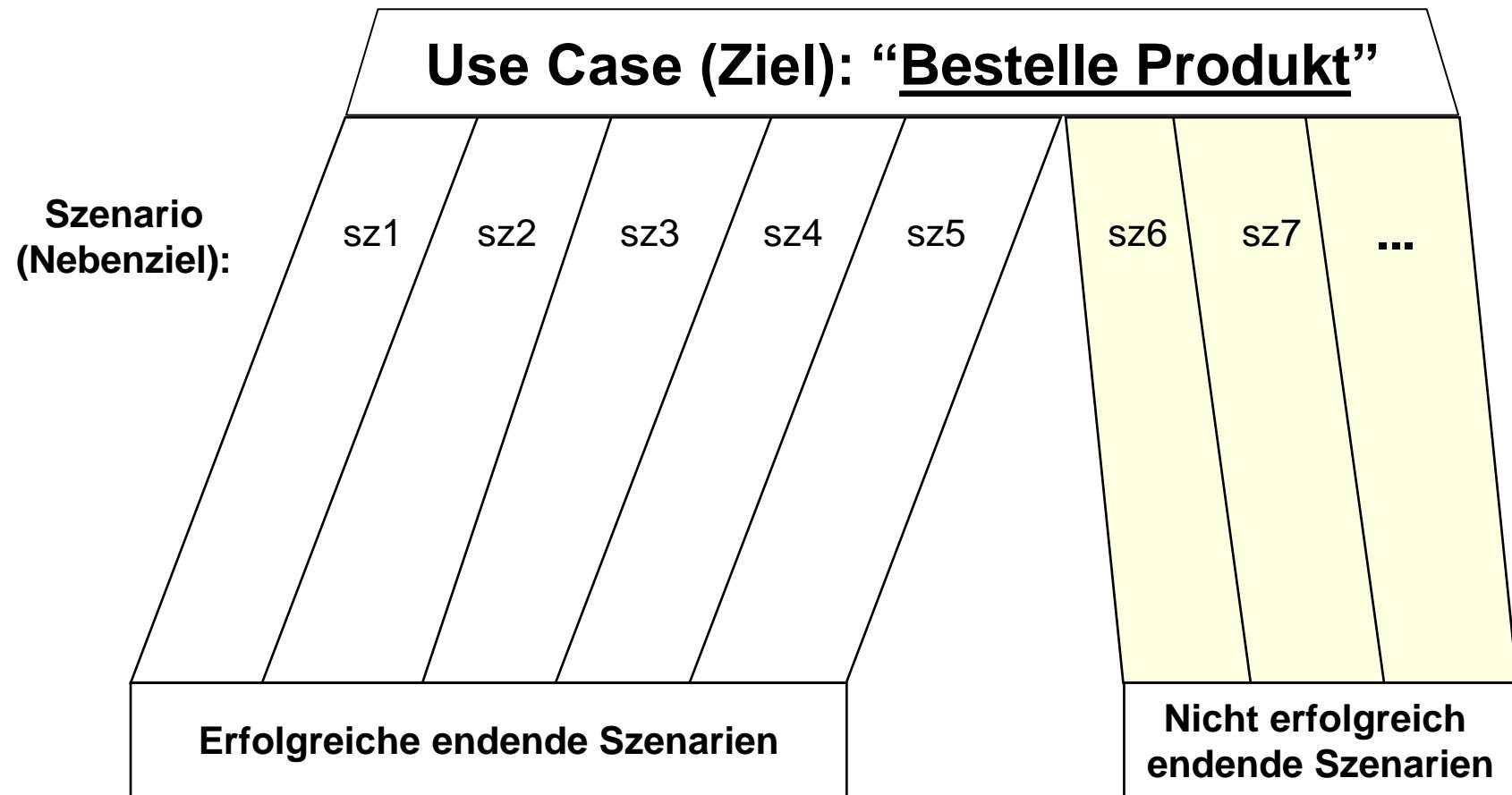


- Im System umgesetzte **Use Cases** dienen dem Erreichen der **Ziele**
 - ◆ Die Use Cases sind nach den Zielen benannt
- Jeder Use Case fasst eine Menge von **Szenarien** zusammen
- Ein Szenario beschreibt einen Ablauf des Use Case.

Use Cases und Szenarien (1)

- Ein Use Case verbindet ein Ziel mit den zugehörigen Szenarien
 - ◆ Der Name des Use Case ist seine Zielsetzung:
“Bestelle Produkt”
 - ◆ Beachte die Grammatik: das aktive Verb steht am Anfang
- Ein Szenario spezifiziert wie sich eine Voraussetzung auswirkt
 - ◆ Szenario (1): Alles funktioniert wie vorhergesehen...
 - ◆ Szenario (2): Zu wenig Guthaben...
 - ◆ Szenario (3): Produkt nicht mehr vorrätig...

Ziele, Use Cases und Szenarien (2)



Zusammenfassung: Wozu Use Cases?

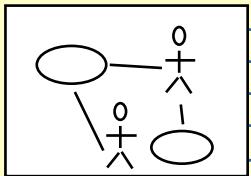
- Nachvollziehbarkeit für Nutzer
 - ◆ Use Cases modellieren ein System aus Sicht des Nutzers
 - ⇒ Bestimmung jedes möglichen Ereignisflusses durch das System
 - ⇒ Beschreibung von Interaktionen
 - ⇒ Beschreibung von nichtfunktionalen Anforderungen und Nebenbedingungen
- Werkzeug um ein Projekt zu managen
 - ◆ Basis für den gesamten Entwicklungsprozess
 - ⇒ Entwurf
 - ⇒ Implementierung
 - ⇒ Testspezifikation
 - ⇒ Akzeptanztest beim Kunden
 - ⇒ Gebrauchsanleitung
 - ◆ Exzellente Grundlage für inkrementelle & iterative Entwicklung
 - ⇒ Alle Prozessschritte erst für eine Teilmenge der Use-Cases durchlaufen
 - ⇒ ... dann für die nächste Teilmenge, u.s.w.

Use-Case-getriebener Softwareentwicklungsprozeß

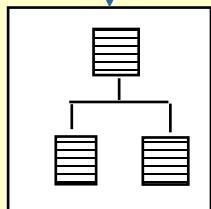
Requirements Engineering

Anforderungs-erhebung

Use Case Model

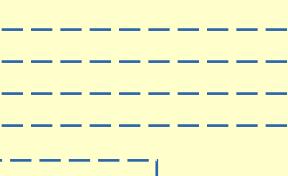


ausgedrückt als

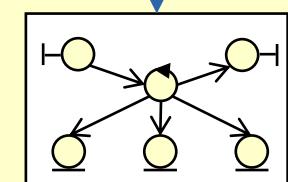


Domain Object Model

Anforderungs-analyse

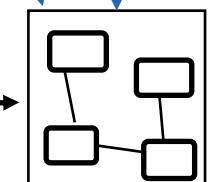


verfeinert zu



Analysis Model

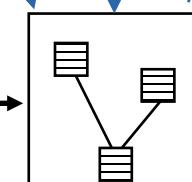
System Design



strukturiert durch

Subsysteme

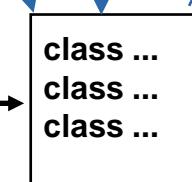
Object Design



realisiert durch

Implementation Domain Objects

Implementa-tion

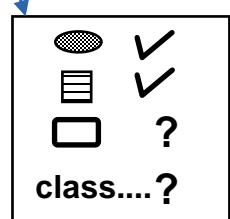


implementiert von

class ...
class ...
class ...

Quell Code

Testen



verifiziert durch

Test Fällen

* Die erzeugten dynamischen Modelle (und Andere) sind hier aus Platzgründen nicht explizit dargestellt.

6.3. Strukturierte Erfassung von Use Cases und Anwendungsfalldiagramme ("Use Case Diagrams")

1. Wie erfassen wir EINEN Use Case?

2. Wie behalten wir die Übersicht über VIELE Use Cases???

Strukturierte Use Case Beschreibung

► Schema

- Name des Use Case und Kurzbeschreibung
- Akteure
 - ◆ Beschreibung der am Use Case beteiligten Akteure
- Vorbedingung
 - ◆ Was gelten muss, damit der Use Case durchgeführt werden kann
- Ereignisfluss
 - ◆ Schritte die im Standardablauf des Use Case durchgeführt werden
- Nachbedingung
 - ◆ Was nach erfolgreichem Ende des Ereignisflusses gilt
- Sonderfälle (Alternativabläufe und Fehlerfälle)
 - ◆ Jeweils Name, Verzweigungspunkt („extension point“) im Standardablauf, auslösende Bedingung, Ereignisfluss im Sonderfall und Nachbedingung des Sonderfalls (→ siehe „extends“-Beziehung)
- Spezielle Anforderungen
 - ◆ Nichtfunktionale Anforderungen und Nebenbedingungen

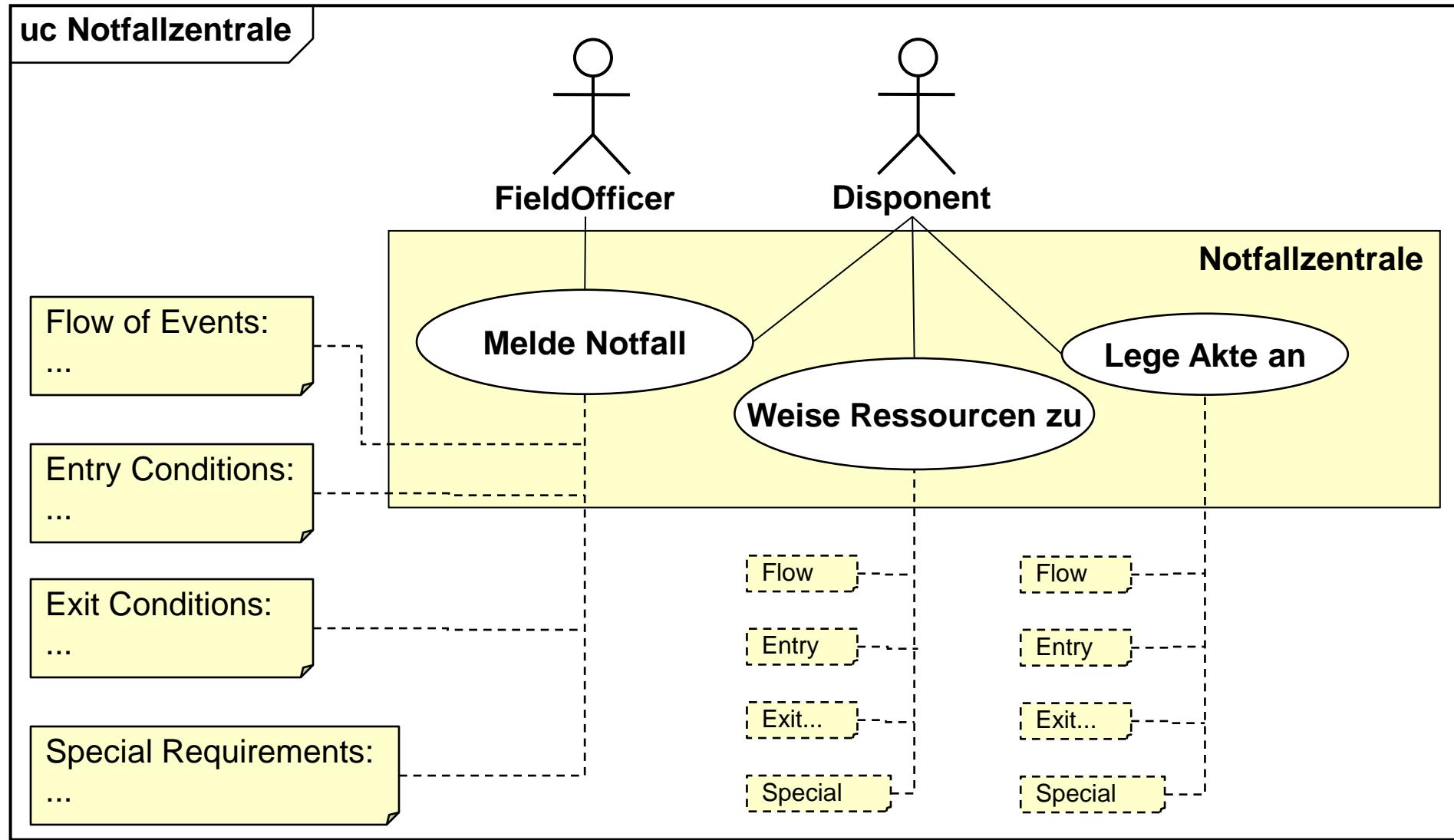
Strukturierte Use Case Beschreibung

► Beispiel „Termin erfassen“

Standardablauf

Kurzbeschreibung:	Ein Termin wird für einen oder mehrere Teilnehmer eingetragen.
Akteure:	Benutzer E-Mail-System (zum Versenden von Benachrichtigungen) ...
Vorbedingung:	Benutzer ist dem System bekannt und eingeloggt.
Ereignisfluss:	1. Neuer Termin wird erfasst (Zeit, Ort, ...) 2. Teilnehmer werden zugeordnet. 3. Benutzer ist berechtigt für alle Teilnehmer Termine zu erfassen. 4. Benachrichtigungen werden verschickt. 5. Sichten werden aktualisiert
Nachbedingung:	Neuer Termin ist erfasst. Alle Teilnehmer sind verständigt und alle Sichten sind aktualisiert.
Alternativablauf A1:	3'. Benutzer hat für mind. einen Teilnehmer keine Berechtigung. ...
Fehlerfall F1:	Benutzer hat für keinen Teilnehmer die Berechtigung, einen ...
Nachbedingung zu F1:	Neuer Termin erfasst, aber ohne Zuordnung zu Teilnehmern. ...

Use Case Diagramm „Notfallzentrale“



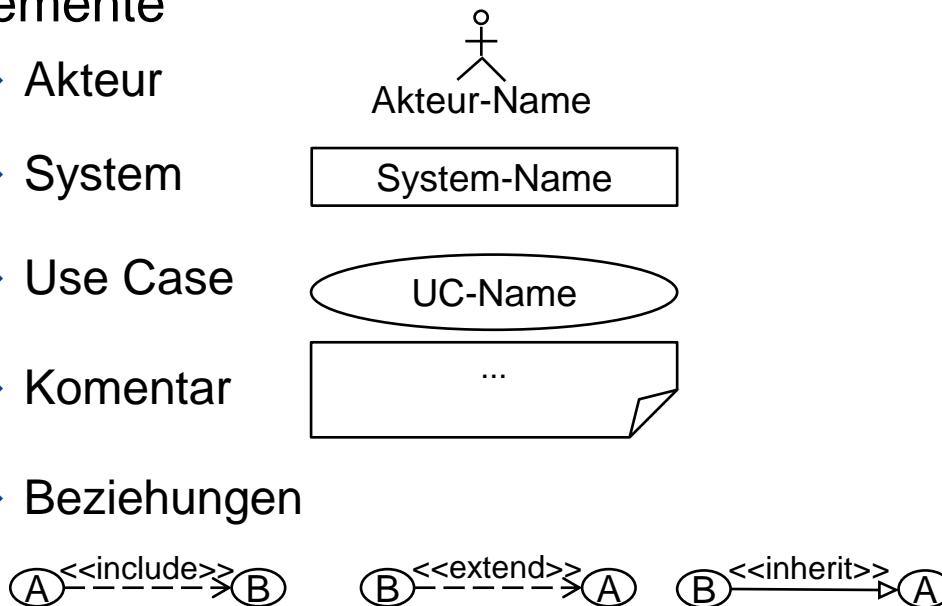
Use-Case Diagramm ▶ Elemente

● Einsatz

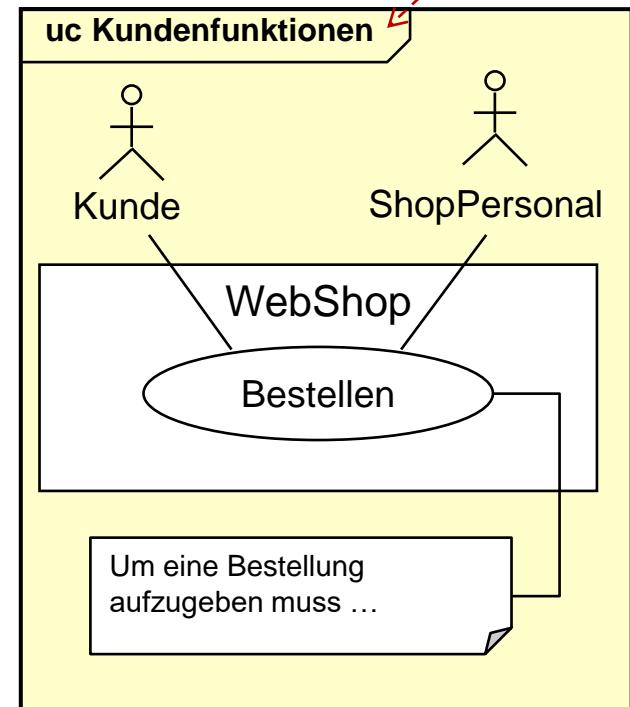
- ◆ Kommunikation mit Kunde / Benutzer während Anforderungserhebung
- ◆ Erfassung von Beziehungen zwischen
 - ⇒ Actors untereinander
 - ⇒ Use Cases untereinander
 - ⇒ Actors und Use Cases

● Elemente

- ◆ Akteur
- ◆ System
- ◆ Use Case
- ◆ Komentar
- ◆ Beziehungen



Art (uc) und Name des Diagramms



Use-Case Diagramm ▶ Elemente

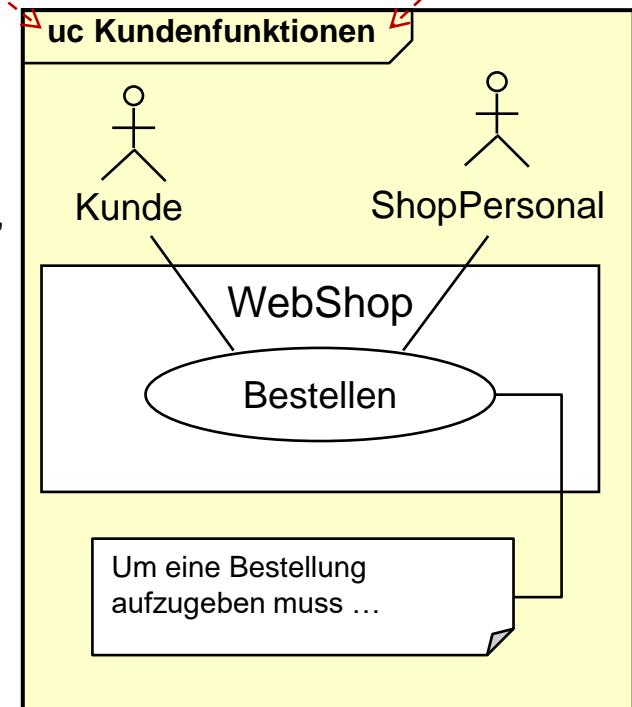
- Wie jedes UML-Diagramm ist auch ein Use-Case-Diagramm in einem Rahmen enthalten, in dessen oberen linken Ecke die Diagrammart (hier „uc“ für Use Case) und der Name des Diagramms angegeben ist.
- Der Name des Diagramms ist nicht gleich dem Namen des Systems. Es kann viele Diagramme geben, die Ausschnitte des gleichen Systems darstellen. Z.B. kann das System „WebShop“ dargestellt werden, durch

- ein Diagramm , das alle UseCases zusammenfasst an denen Kunden und ShopPersonal beteiligt sind,
- eines für die UseCases an denen nur ShopPersonal beteiligt ist,
- etc.

Es steht uns frei nach welchen Kriterien wir Use Cases in Diagramme gruppieren.

- Im Folgenden werden aus Platzgründen die Rahmen die das Gesamtdiagramm darstellen und manchmal auch die für das System weggelassen. Das gleiche gilt für Akteure, da wo der Fokus des Diagramms auf etwas anderen liegt. Bitte denken Sie trotzdem dran wenn Sie Aufgaben bearbeiten, denn all diese Elemente gehören zu einer korrekten Notation dazu und werden entsprechend bewertet (in den Übungsaufgaben wie in der Klausur)!

Art (uc) und Name des Diagramms



Verfeinerung und Strukturierung von Use Cases

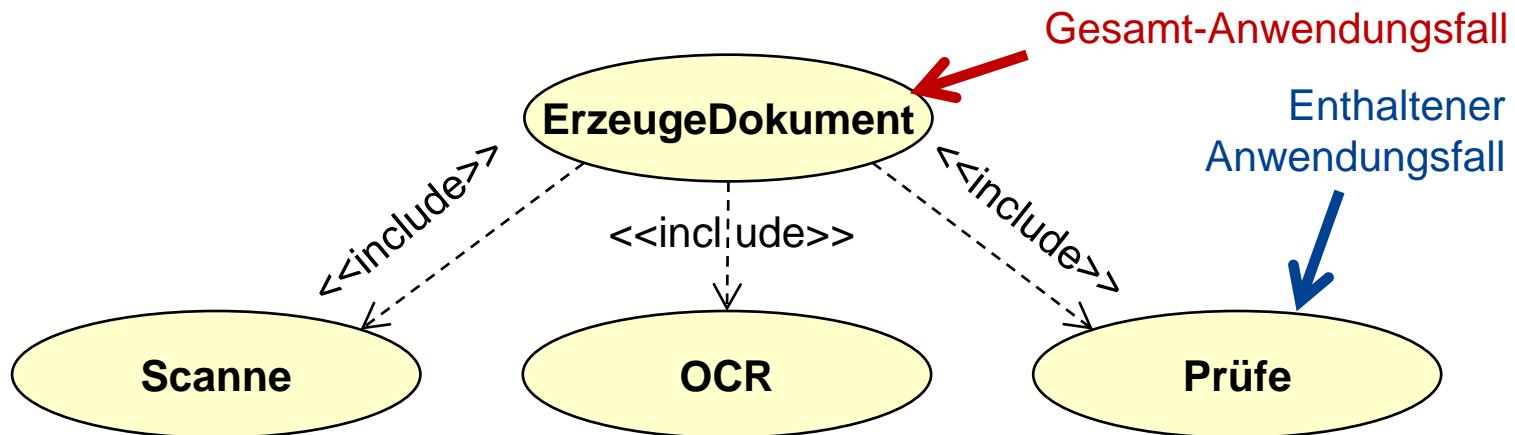
- Ausgangspunkt: Mehrere Use Cases erfasst
- Verfeinere und strukturiere die Use Cases durch Assoziationen
 - ◆ Use Case Assoziation = Beziehung zwischen Use Cases

Beziehungen zwischen Use-Cases

- Include (funktionale Dekomposition)
 - ◆ Ein Use Case benutzt einen anderen
- Extend (Sonderfall)
 - ◆ Ein Use Case ergänzt einen anderen unter bestimmten Bedingungen
- Inherit (Generalisierung/Spezialisierung)
 - ◆ Ein abstrakter Use Case mit verschiedenen Spezialisierungen

<<include>>-Beziehung ▶ Semantik

- Jeder Ablauf des Gesamt-Anwendungsfalls beinhaltet einen Ablauf des enthaltenen Anwendungsfalls
- Die include-Beziehung von A nach B bedeutet, dass A all das Verhalten von B ausführt ("A delegiert an B").
- A ist abhängig von B (daher auch die Richtung und Art des Pfeiles: der übliche Abhängigkeitspfeil, lediglich mit einem passenden Stereotyp)



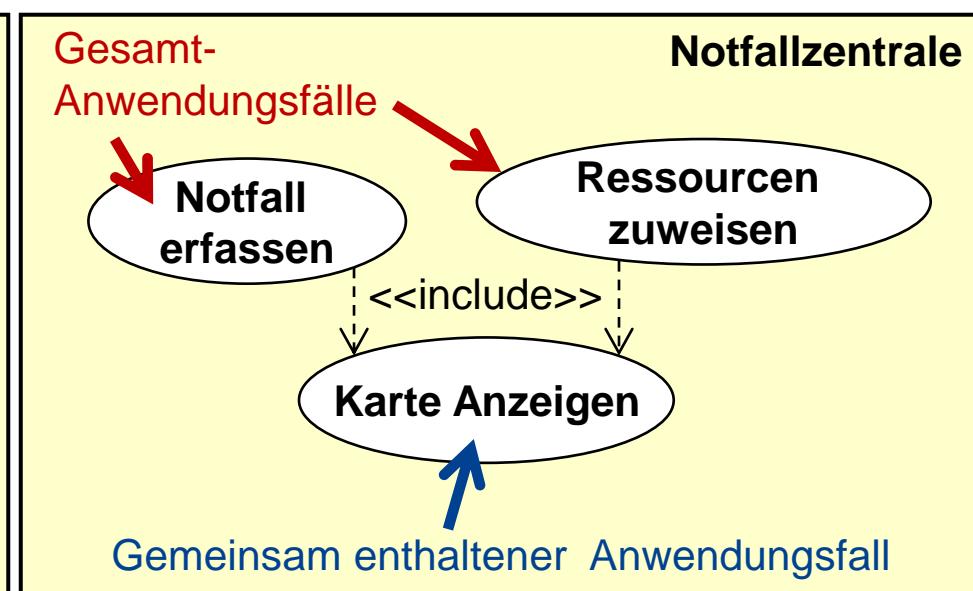
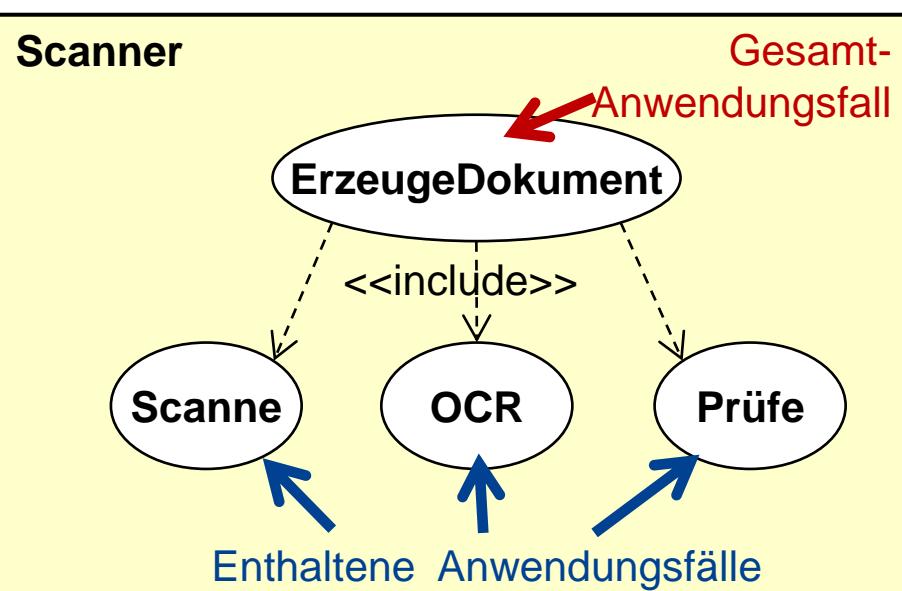
<<include>>-Beziehung ▶ Verwendung

Funktionale Dekomposition

Ein Use Case ist zu komplex und muss zerlegt werden.

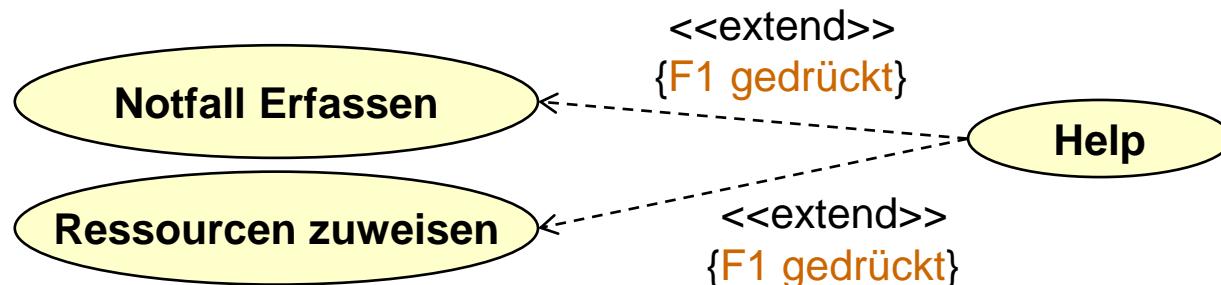
Gemeinsame Verwendung

Gemeinsames Verhalten verschiedener Use Cases muss ausgedrückt werden



<<extend>>-Beziehung

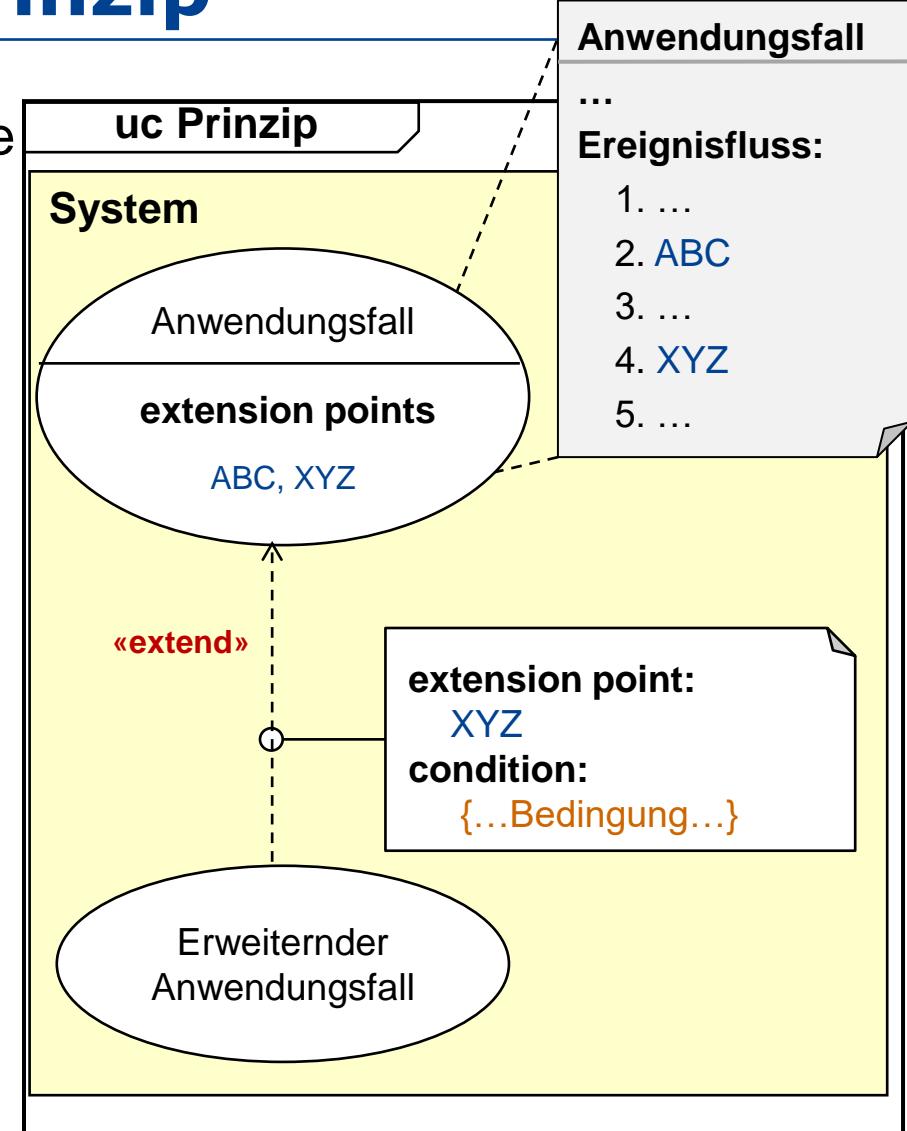
- Problem
 - ◆ Unter bestimmten Bedingungen (Sonderfälle, Fehlerfälle, ...) muss der Ablauf eines Use-Cases erweitert werden
- Lösung / Semantik
 - ◆ Eine extend-Beziehung von Use Case A nach B bedeutet, dass A eine Erweiterung von B ist, die nur unter der **angegebenen Bedingung** aktiv ist.
 - ◆ Der erweiterte UC ist unabhängig vom erweiternden UC.
- Beispiel
 - ◆ “Notfall erfassen” ist komplett, kann aber in einem Szenario, in dem der Benutzer Hilfe braucht, durch „Help“ erweitert werden.



<<extend>>-Beziehung

► Extension Points: Prinzip

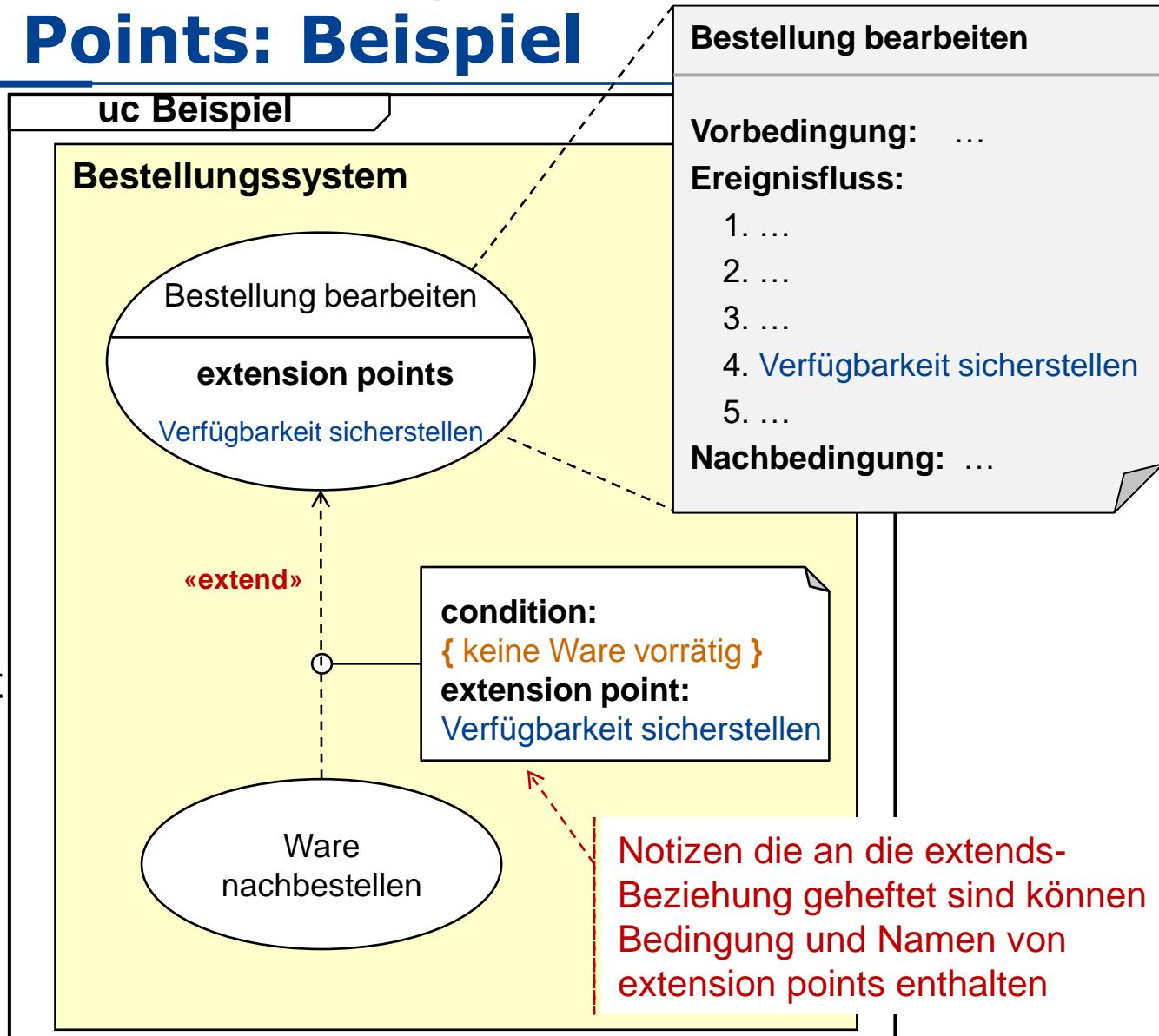
- Deklaration im erweiterten Use Case
 - ◆ Benannte Stellen im **Ereignisfluss** des erweiterten Use Case, wo der Ablauf des erweiternden Use Cases einzufügen ist
- Bezugnahme in <<extends>>-Beziehung
 - ◆ Erweiternder Use Case wird am **Extension Point** aktiviert
 - ◆ ... falls die spezifizierte **extends-Bedingung** gilt



<<extend>>-Beziehung

► Extension Points: Beispiel

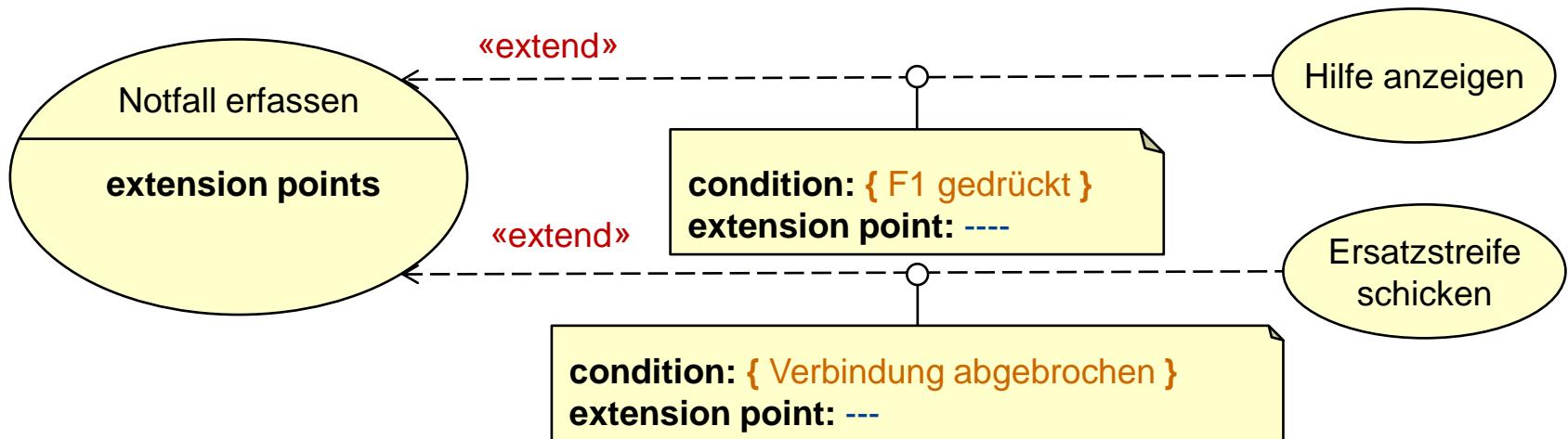
- „Ware nachbestellen“ ist ein Sonderfall
- ... der an der Stelle „Verfügbarkeit sicherstellen“ im Ereignisfluss von „Bestellung bearbeiten“ auftritt
- ... falls dann die Bedingung „keine Ware vorrätig“ gilt



<<extend>>-Beziehung

► Keine Extension Points für „Ereignisse“

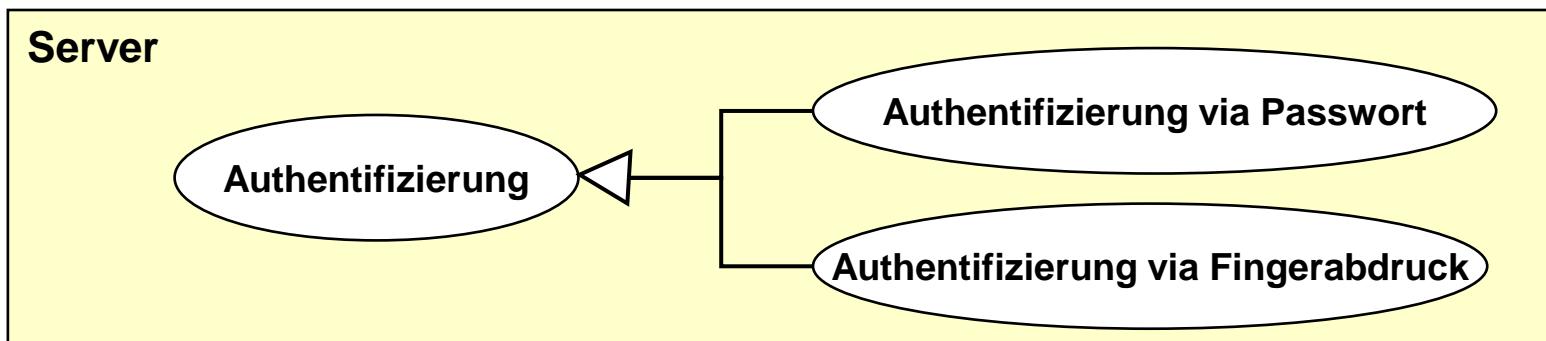
Wenn der Auslöser ein „Ereignis“ ist, werden Extension Points weggelassen, denn Ereignisse können jederzeit auftreten:



Merke: Die Bedingung kann nicht weggelassen werden!

Generalisierung von Use Cases

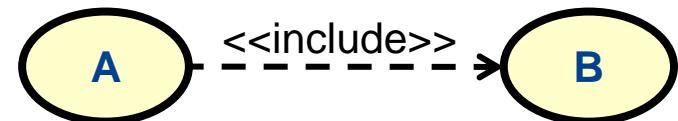
- Ziel
 - ◆ Modellieren, dass verschiedene Use Cases Varianten eines allgemeineren Ablaufs sind
- Prinzip
 - ◆ Die „Kinder“ erben die Kommunikationsbeziehungen, die Bedeutung und das Verhalten der „Eltern“, modifizieren es aber im Detail.
- Beispiel
 - ◆ „Authentifizierung“ prüft die Benutzeridentität. Der Kunde könnte zwei Umsetzungen fordern: „Via Passwort“ und „Via Fingerabdruck“



Beziehungen zwischen Use-Cases

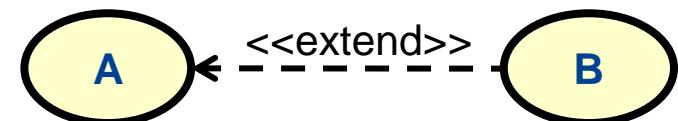
● Include-Beziehung

- ◆ Immer wenn A ausgeführt wird, **muss** auch B ausgeführt werden
- ◆ B ist unbedingt nötig, um A auszuführen



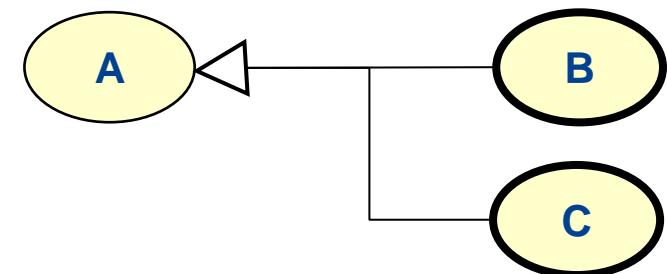
● Extend-Beziehung

- ◆ Wenn A ausgeführt wird, **kann** auch B ausgeführt werden
- ◆ B ist nicht zwingend nötig, um A auszuführen

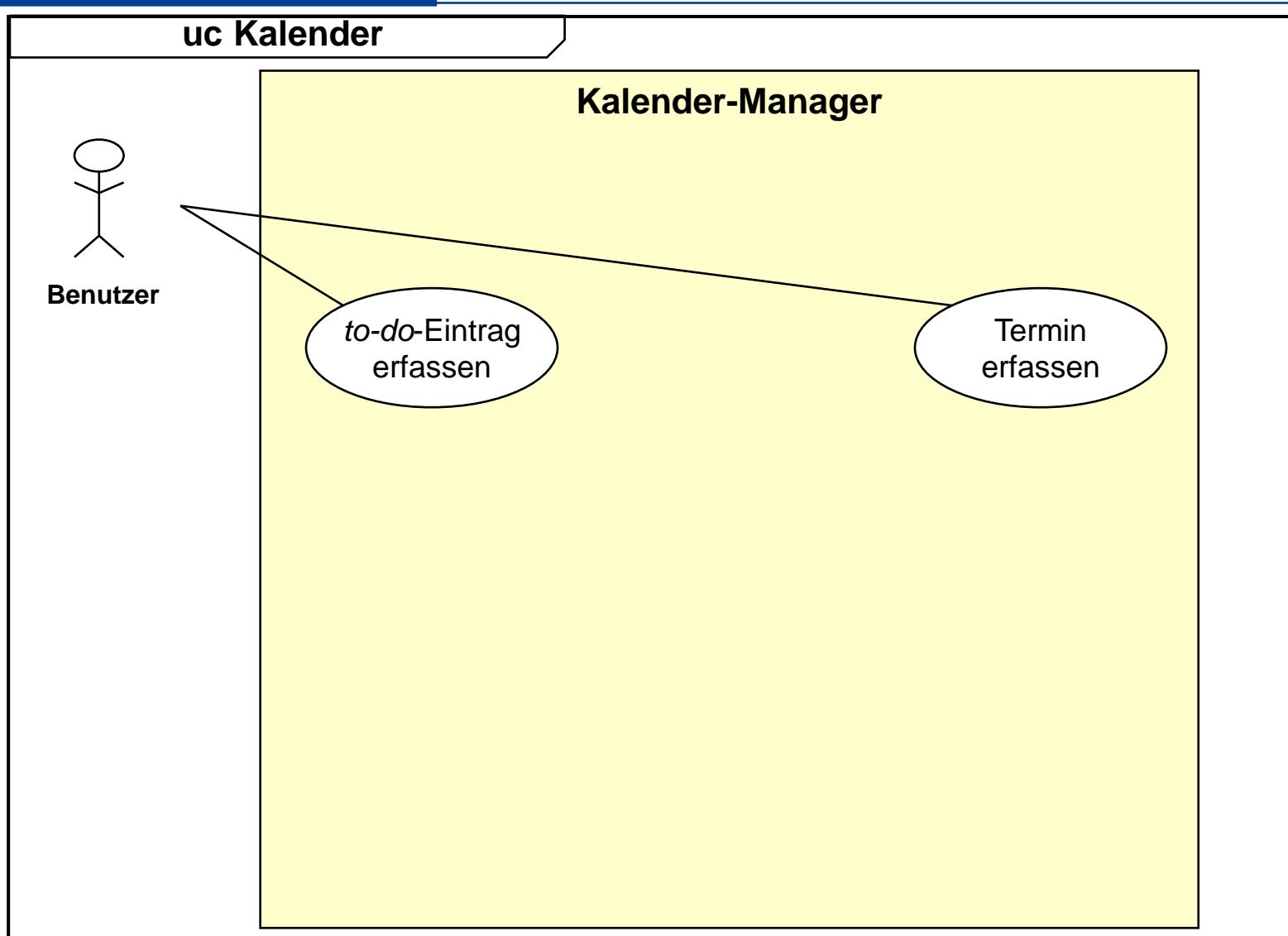


● Generalisierungs-Beziehung

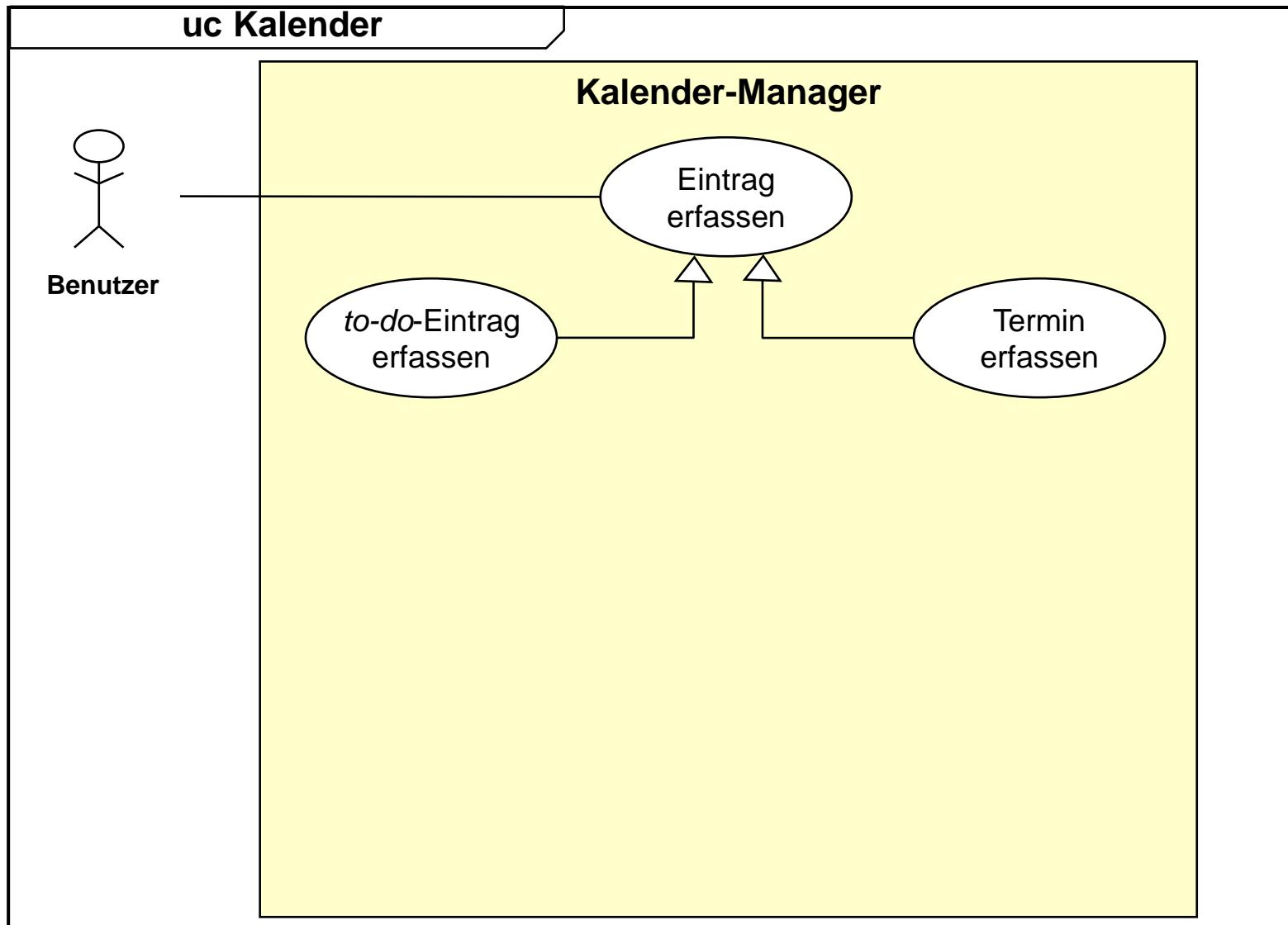
- ◆ B und C sind jeweils spezielle Ausprägungen des **gesamten** Ablaufs von A
- ◆ Nur eine der verschiedenen Spezialisierungen wird durchgeführt, die aber **komplett**



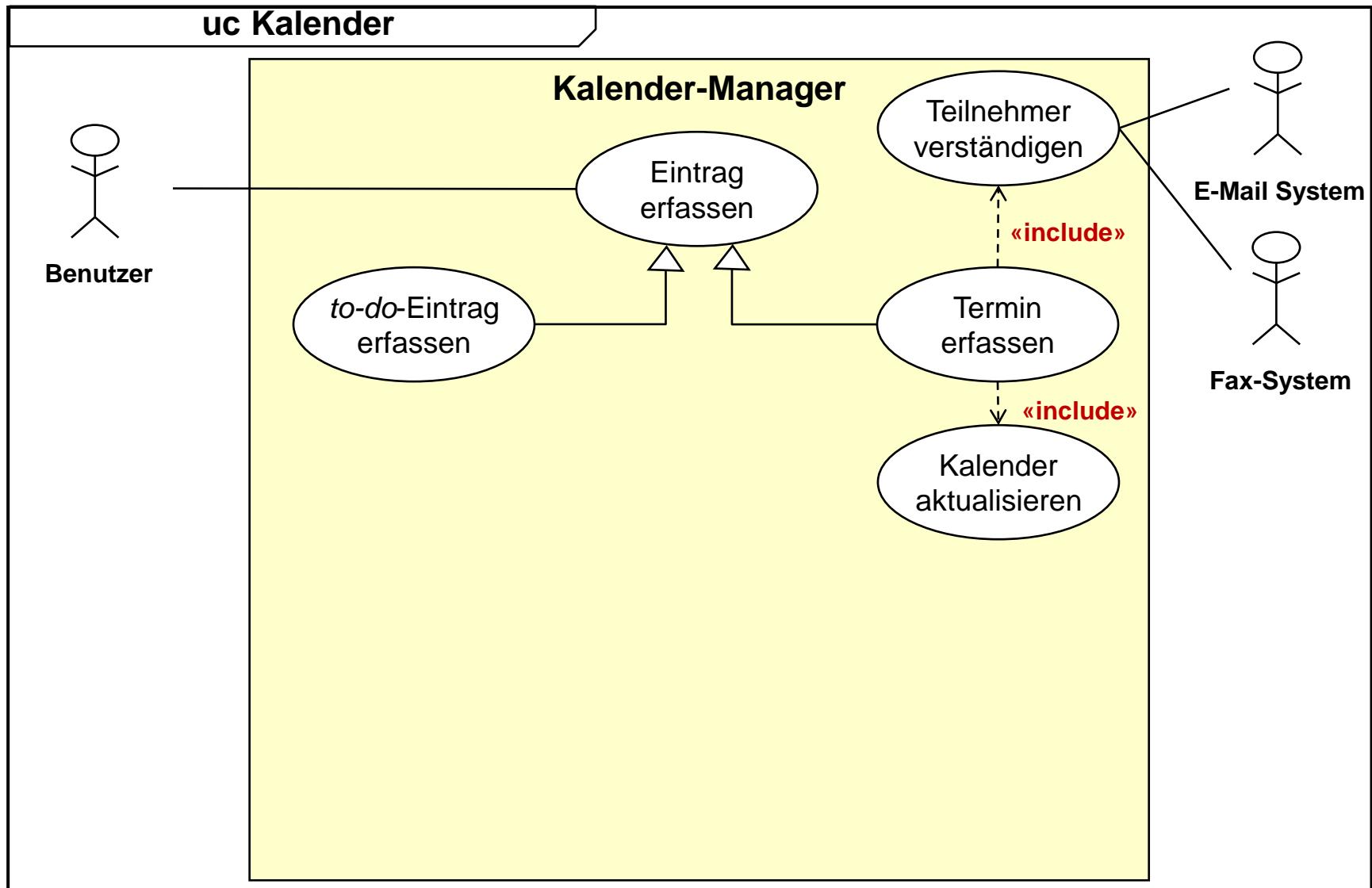
Beispiel „Kalender-Manager“



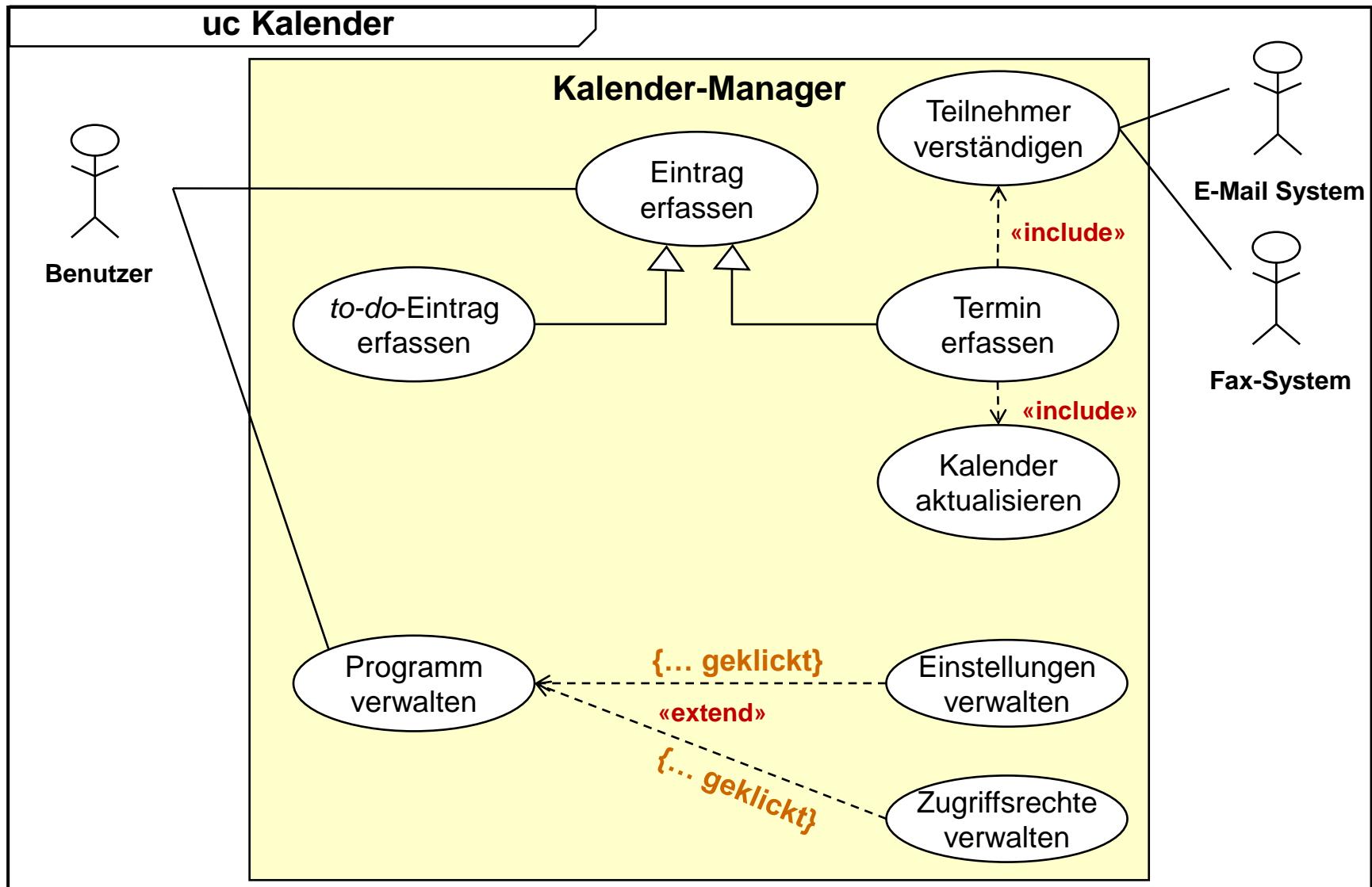
Beispiel „Kalender-Manager“ ▶ Varianten der Eintragserfassung als Generalisierung



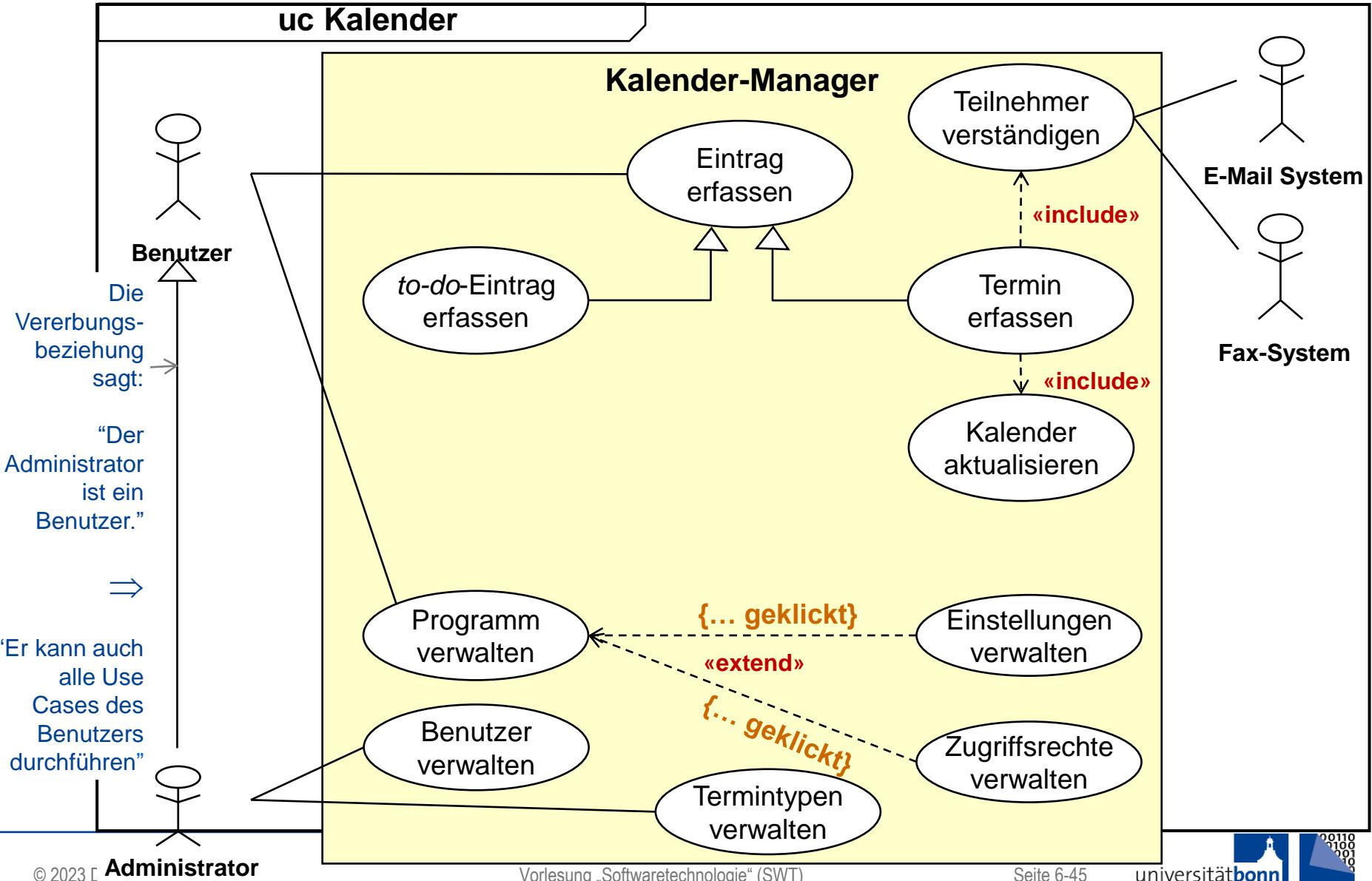
Beispiel „Kalender-Manager“ ▶ <<include>> für Teilschritte der Terminerfassung



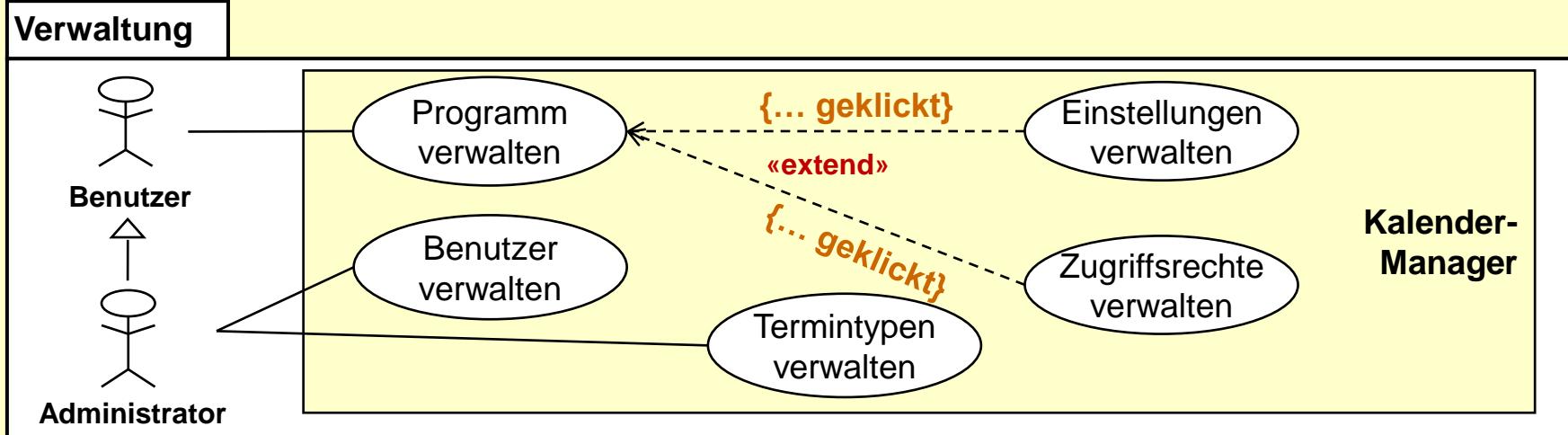
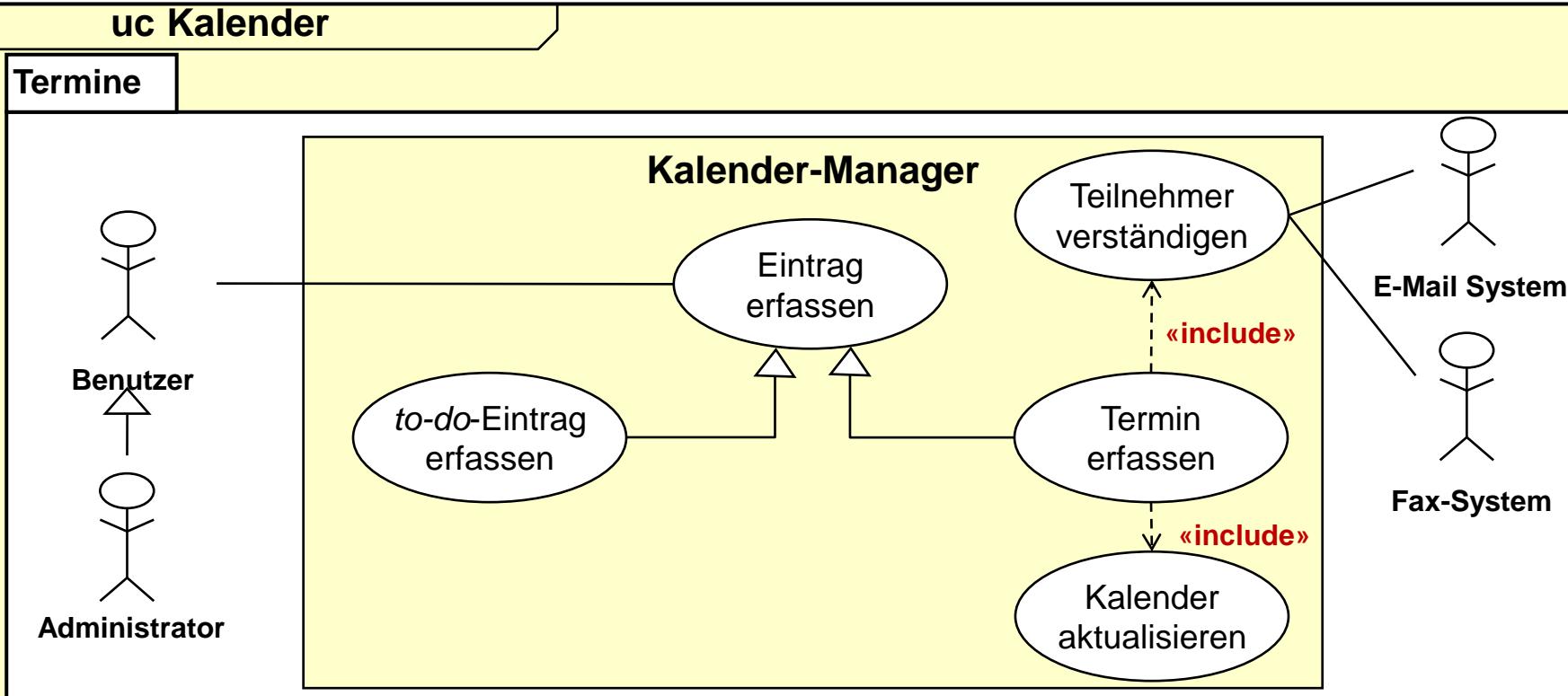
Beispiel „Kalender-Manager“ ▶ <<extend>> für Sonderfälle der Programmverwaltung



Beispiel „Kalender-Manager“ ▶ Generalisierung zwischen Akteuren



Beispiel ▶ Modularisierung durch Packages



Anforderungserhebung ▶ Gesamtüberblick



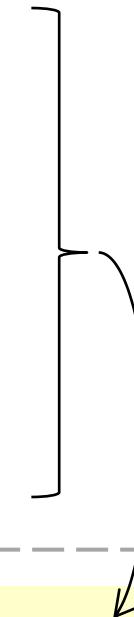
1. Sammle Anforderungen

- ◆ Identifiziere funktionale Anforderungen
- ◆ Identifiziere nichtfunktionale Anforderungen
- ◆ Identifiziere Nebenbedingungen



2. Entwickle das funktionale Modell

- ◆ Entwickle Use Cases zur Illustration der funktionalen Anforderungen



Input für

● 3. Entwickle das Objektmodell der Anwendungsdomäne

- ◆ Klassendiagramme, die die relevante Konzepte der Domäne beschreiben

● 4. Entwickle das dynamische Modell der Anwendungsdomäne

- ◆ Interaktionsdiagramme für das Zusammenspiel von Objekten
- ◆ Zustandsdiagramme für die internen Abläufe von Objekten
- ◆ Aktivitätsdiagramme für Geschäftsprozesse

6.4 Domain Object Model

Abbott's Technik zur Objektidentifikation

Domain Object Model (DOM)

- Das DOM ist eine Menge von Klassendiagrammen, die Konzepte der Anwendungsdomäne beschreiben
 - ◆ Klassen
 - ◆ Attribute
 - ◆ Beziehungen
 - ◆ (wenige Operationen)
- Vorgehensweise
 - ◆ Dialog mit Benutzer → Textuelle Anforderungsspezifikation (Use Cases)
 - ◆ Anschließend Textanalyse nach Abbott → Initiales Klassendiagramm
 - ◆ Anschließend Verfeinerung des Modells anhand allgemeiner Modellierungsprinzipien (s. Kapitel 2. „Objektorientierte Modellierung“)

Domain Object Model ▶ Beispiel „Terminverwaltung“

class Terminverwaltung



Beispiel ▶ Ein Szenario aus der Problembeschreibung

- Jim Smith ist Kunde bei einer großen Bank.
- Dort besitzt er ein kostenpflichtiges Konto mit einem Kontostand von derzeit 2.000 Euro.
- Jim Smith geht zum Schalter und zahlt 100,- Euro ein.
- Am nächsten Tag betritt Jim Smith die Bank erneut. Er geht zum Geldautomat und hebt 400,- Euro ab.

Gibt uns diese Beschreibung Hinweise,
wie unser Objektmodell aussehen sollte?

Beispiel ▶ Ein Szenario aus der Problembeschreibung

- Jim Smith ist Kunde bei einer großen Bank.
- Dort besitzt er ein kostenpflichtiges Konto mit einem Kontostand von derzeit 2.000 Euro.
- Jim Smith geht zum Schalter und zahlt 100,- Euro ein.
- Am nächsten Tag betritt Jim Smith die Bank erneut. Er geht zum Geldautomat und hebt 400,- Euro ab.

Was haben wir hier gemacht? Satzelemente kategorisiert!
→ Abbott's Textanalyse

Textanalyse von Abbott [Abbott 1983]

- Zuordnung von Teilen der Sprache zu Komponenten des Objektmodells
- Wird vor allem genutzt bei Erstellung des Domain Object Models und Analysemodells

Sprachelement	Modellelement	Beispiel
Eigenname	Objekt	Jim Smith
Nomen	Klasse	Kunde, Konto, Bank
„ist“	Generalisierung	Sparkonto ist ein Konto
„hat“, „enthält“, ...	Aggregation	Ein Konto hat eine Nummer
Modalverb („müssen“, „können“, „dürfen“, „sollen“)	Einschränkung (Constraint)	Kontostand darf bestimmte Grenze nicht unterschreiten
Adjektiv	Attribut	kostenpflichtig
Transitives Verb	Methode	bringen
Intransitives Verb	Methode (Event)	erscheinen

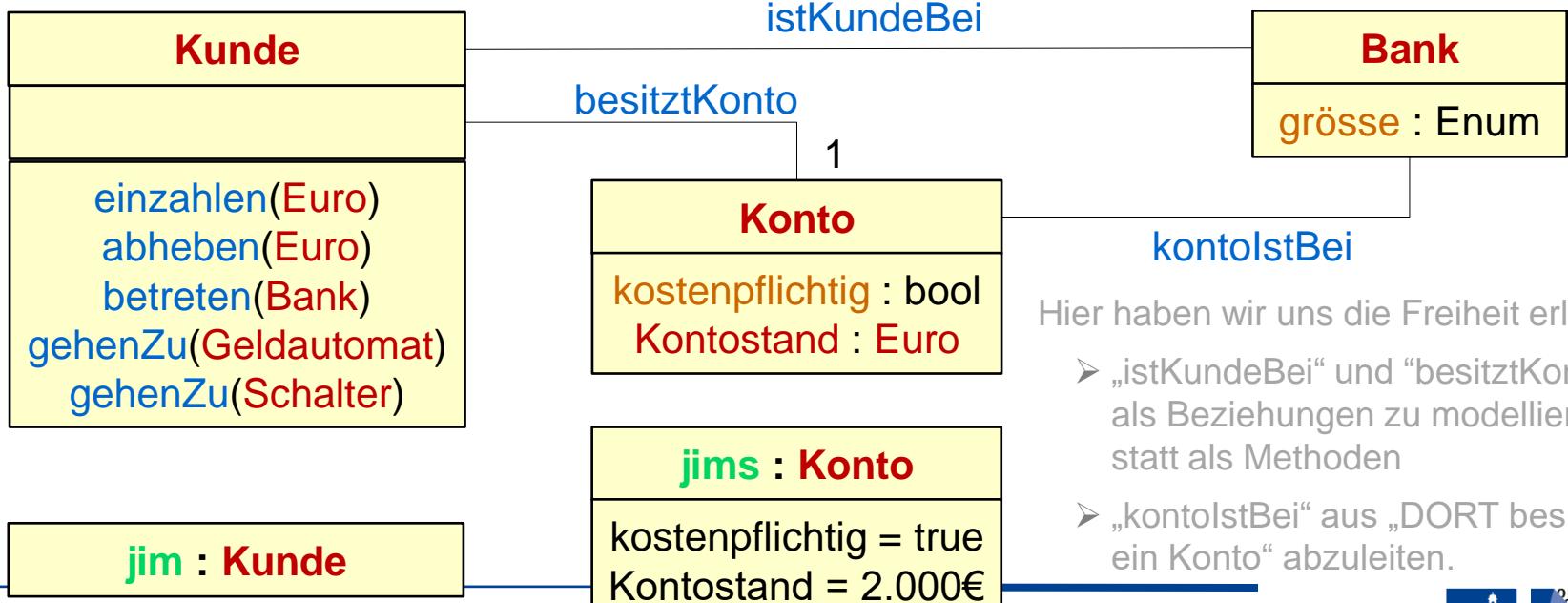
Textanalyse von Abbott [Abbott 1983]

Abbott's Technik ist eine Hilfestellung für denkende Menschen, nicht ein starres Regelwerk für Automaten!

- Die Entsprechungen sind **Hinweise, keine Gesetze!** Selbst entscheiden, was im konkreten Fall zutrifft ist immer noch erforderlich:
 - ◆ Z. B.: „Tisch **hat** Beine“ → Aggregation
 - ◆ Aber: „Kunde **hat** Konto“ → einfache Assoziation.
 - ◆ Z. B.: „Kind **ist** Mensch“ → Generalisierung
 - ◆ Aber: „Thomas **ist** Kind“ → Instanz!
- Die Entsprechungen aus der Abbott-Tabelle sind **nicht vollständig!** Sie sollten sie sinngemäß ergänzen.
 - ◆ Z.B: „hat“ ≈ „beinhaltet“ ≈ „enthält“ ≈ „ist Teil von“ ≈ ...

Beispiel ▶ Ein mögliches erstes Ergebnis Ihrer Abbot-Analyse

- Jim Smith ist Kunde bei einer großen Bank.
- Dort besitzt er ein kostenpflichtiges Konto mit einem Kontostand von derzeit 2.000 Euro.
- Jim Smith geht zum Schalter und zahlt 100,- Euro ein.
- Am nächsten Tag betritt Jim Smith die Bank erneut.
- Er geht zum Geldautomat und hebt 400,- Euro ab.



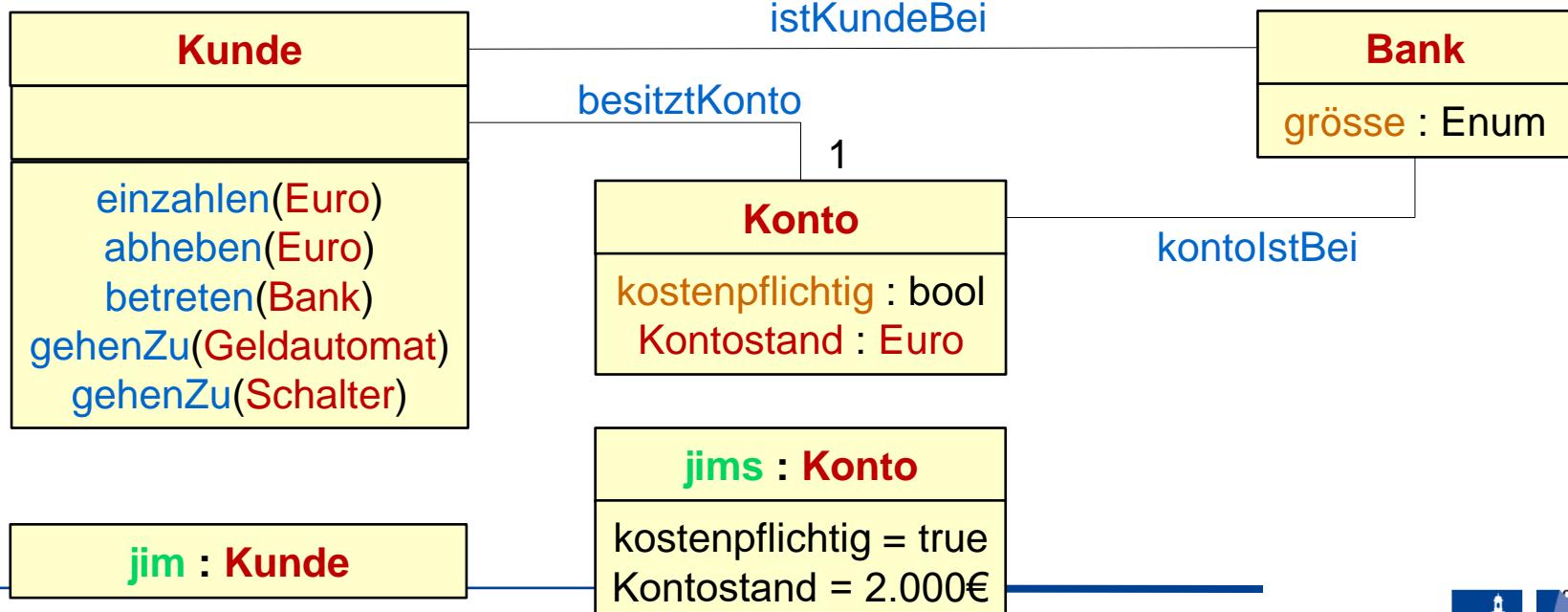
Hier haben wir uns die Freiheit erlaubt:

- „istKundeBei“ und „besitztKonto“ als Beziehungen zu modellieren, statt als Methoden
- „kontolstBei“ aus „DORT besitzt er ein Konto“ abzuleiten.

Beispiel ▶ Sich aus dem ersten Modell ergebende Klärungsfragen

- Kardinalitätsklärungen, zB: „Kann ein Kunde mehrere Konten haben?“
- Ergibt sich „istKundeBei“ aus „besitztKonto“ und „kontolIstBei“?
 - Technisch: „Könnte ich hier eine abgeleitete Beziehung modellieren?“
- Können Einzahlungen nur am Schalter und Abhebungen nur am Automaten erfolgen?
- Sind die geheZu()-Aktionen relevant?

➤ Technisch: „Sollte ich die gehZu() Methoden streichen und stattdessen einzahlen() einen Schalter-Parameter geben und abheben einen Automaten-Parameter?“



Erstellung des Domain Object Model – Ausführliches Beispiel –

Use Case als Ausgangspunkt
Anwendung der Technik von Abbott
Verfeinerung des nach Abbot erstellten Domain Object Model

Beispiel ▶ Ereignisfluss aus Use Case als Ausgangspunkt

- Stellen Sie Sich vor, dass Ihre Kundin, eine Großhändlerin, die Softgetränke an Supermärkte liefert, ein System wie folgt beschreibt:

„Die Pfandrückgabemaschine (PRM) wird in Supermärkten aufgestellt um wiederverwendbare Getränkebehälter zurückzunehmen (leere Dosen, Flaschen und Kästen, welche Endbenutzern geliefert werden). Für jeden Behältertyp kann unsere Betreiberin einen individuellen Pfandbetrag einstellen.“

Anstelle der Rückgabe von Münzgeld druckt die PRM eine Quittung über die Summe der entstandenen Pfandbeträge. Diese Quittung kann durch einen Bar Code Scanner eingelöst werden.

Die PRM generiert für unsere Betreiberin täglich einen Bericht. Ein Bericht beinhaltet eine Liste aller an diesem Tag zurückgegangenen Behälter, sowie die Tagessumme an ausgezahltem Pfand-Geld.“

Beispiel ▶ Textanalyse nach Abbott

▶ Hervorheben der Textelemente

- Substantive / Nomen / Hauptwörter → Klassen

„Die Pfandrückgabemaschine (PRM) wird in Supermärkten aufgestellt um wiederverwendbare Getränkebehälter zurückzunehmen (leere Dosen, Flaschen und Kästen, welche Endbenutzern geliefert werden). Für jeden Behältertyp kann unsere Betreiberin einen individuellen Pfandbetrag einstellen.

Anstelle der Rückgabe von Münzgeld druckt die PRM eine Quittung über die Summe der entstandenen Pfandbeträge. Diese Quittung kann durch einen Bar Code Scanner eingelöst werden.

Die PRM generiert für unsere Betreiberin täglich einen Bericht. Ein Bericht beinhaltet eine Liste aller an diesem Tag zurückgegangenen Behälter, sowie die Tagessumme an ausgezahltem Pfand-Geld.“

Beispiel ▶ Textanalyse nach Abbott

▶ Klassen extrahieren

- Substantive / Nomen / Hauptwörter → Klassen

„Die Pfandrückgabemaschine (PRM) wird in Supermärkten aufgestellt um wiederverwendbare Getränkebehälter zurückzunehmen (leere Dosen, Flaschen und Kästen, welche Endbenutzern geliefert werden). Für jeden Behältertyp kann unsere Betreiberin einen individuellen Pfandbetrag einstellen.

Betreiberin

Supermarkt

Aus Platzgründen wird der Rahmen mit Bezeichner hier weggelassen

Pfandrückgabemaschine

Getränkebehälter

Behältertyp

Pfandbetrag

Dose

Flasche

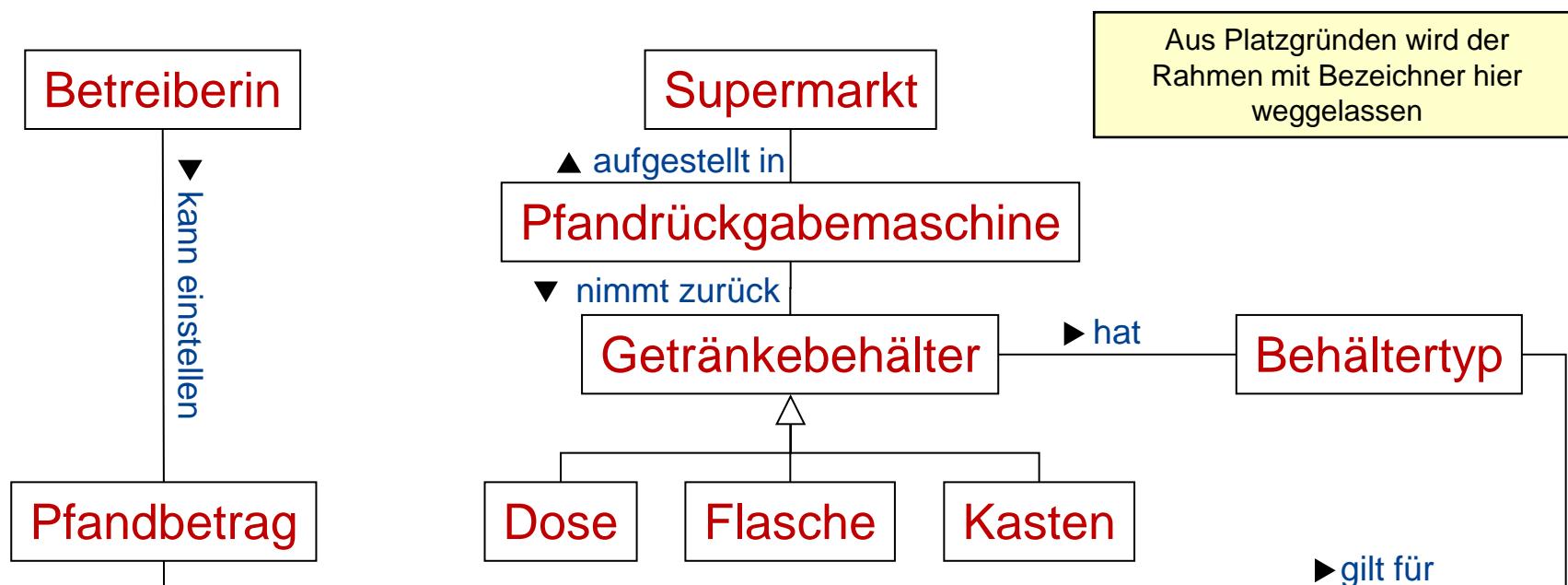
Kasten

Beispiel ▶ Textanalyse nach Abbott

▶ Assoziationen extrahieren

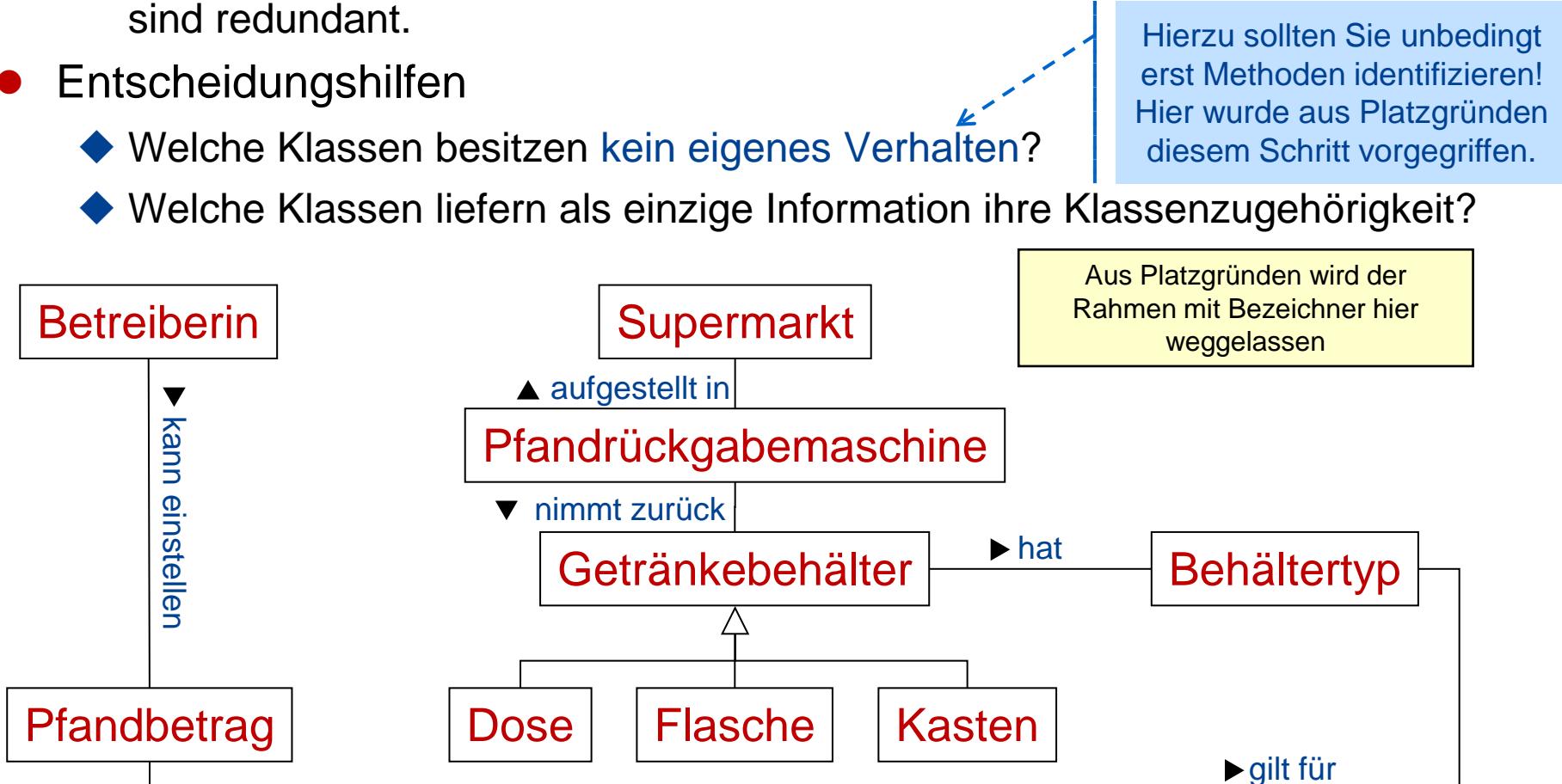
- Substantive + Verben + Attribute → Klassen + Assoziationen

„Die Pfandrückgabemaschine (PRM) wird in Supermärkten aufgestellt um wiederverwendbare Getränkebehälter zurückzunehmen (leere Dosen, Flaschen und Kästen, welche Endbenutzern geliefert werden). Für jeden Behältertyp kann unsere Betreiberin einen individuellen Pfandbetrag einstellen.“



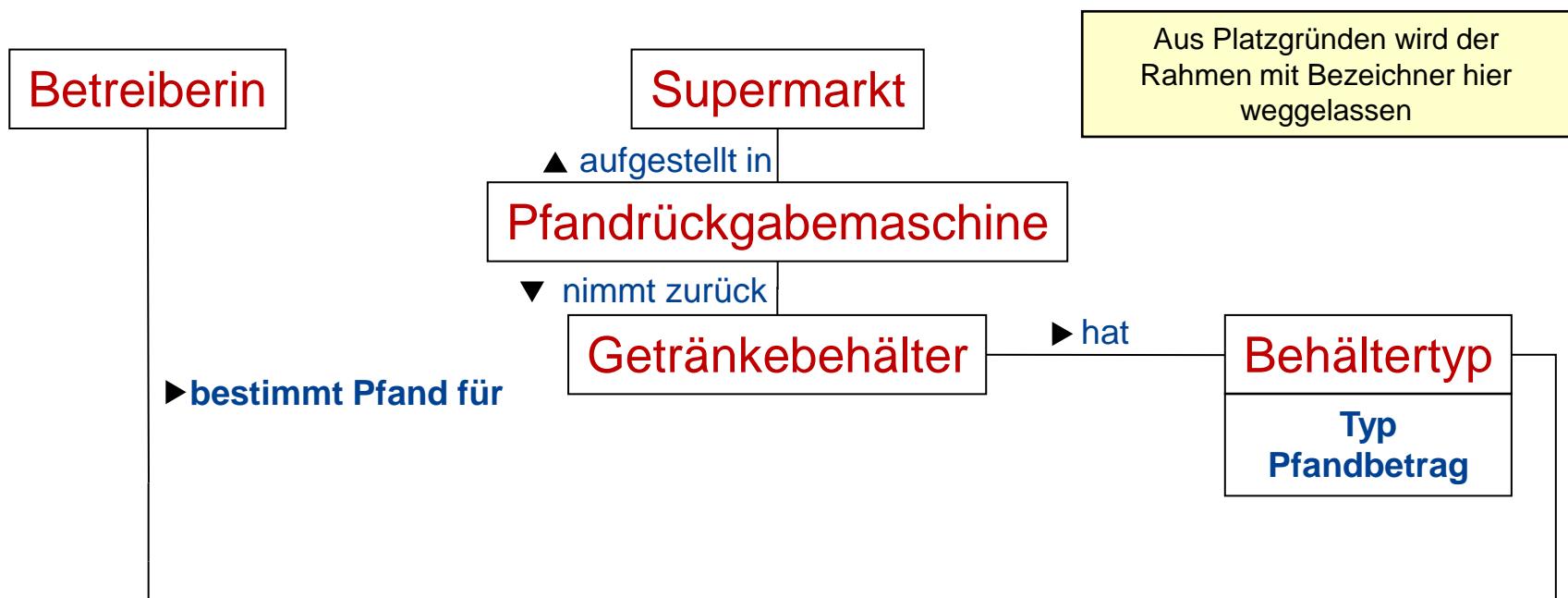
Beispiel ▶ Schematisch erstelltes Domain Object Model überdenken (1)

- Vererbungshierarchie überdenken
 - ◆ Dosen, Flaschen und Kästen sind doch eigentlich Behältertypen!
 - ◆ Die Klasse „Behältertyp“ oder die Unterklassen von „Getränkebehälter“ sind redundant.
- Entscheidungshilfen
 - ◆ Welche Klassen besitzen kein eigenes Verhalten?
 - ◆ Welche Klassen liefern als einzige Information ihre Klassenzugehörigkeit?



Beispiel ▶ Schematisch erstelltes Domain Object Model überdenken (2)

- Korrektur-Maßnahmen: Klasse zu Attribut konvertieren, wenn
 - ◆ sie kein eigenes Verhalten besitzt → Pfandbetrag
 - ◆ sie als einzige Information ihre Klassenzugehörigkeit liefert → Dose, ...
- Anwendung des Prinzips in unserem Beispiel
 - ◆ Pfandbetrag → Attribut von „Behältertyp“
 - ◆ Dose, Flasche, Kasten → Werte des „Typ“-Attributs von „Behältertyp“



Beispiel ▶ Schematisch erstelltes Domain Object Model überdenken (3)

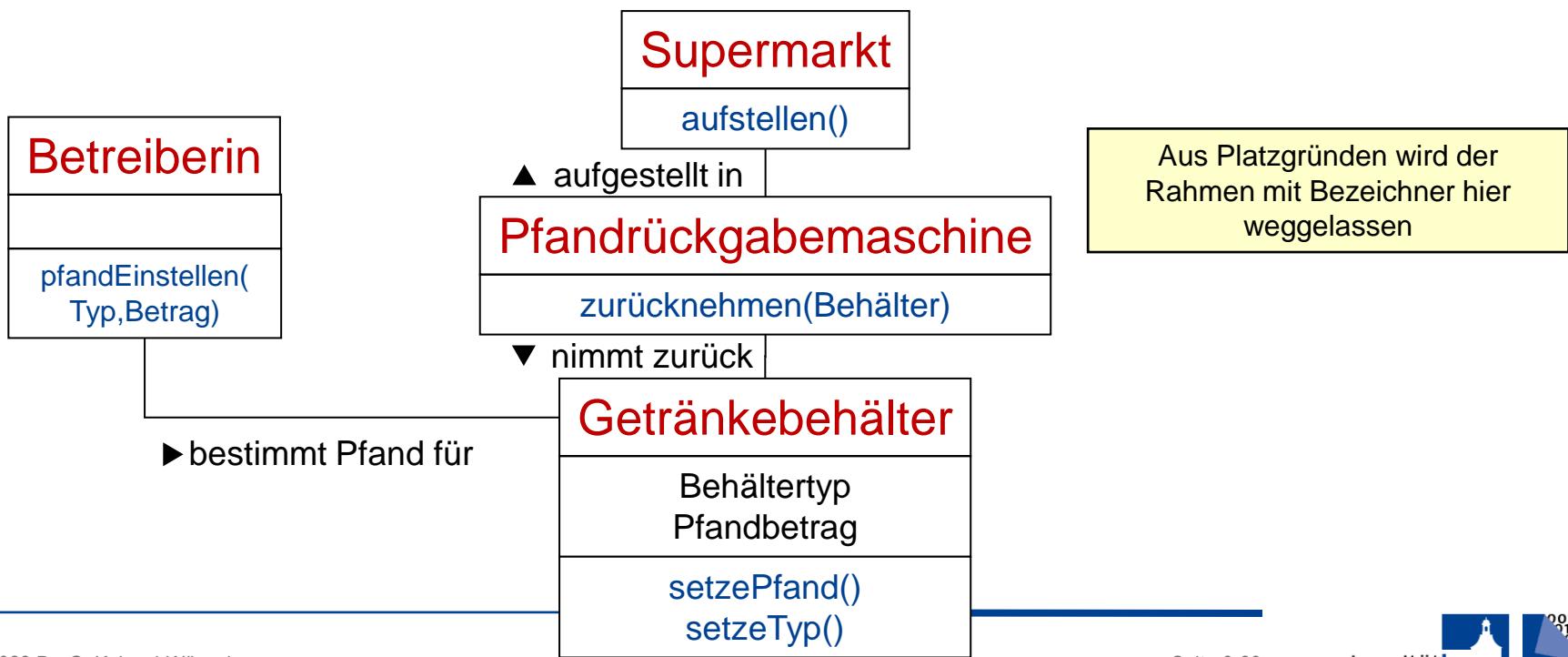
- Korrektur-Maßnahmen: „Inline Class“
 - ◆ Überflüssige Klasse (→ „Behältertyp“) identifizieren und ihre Elemente in eine per Assoziation verbundene Nachbarklasse einbetten
- Anwendung des Prinzips in unserem Beispiel
 - ◆ Die Attribute „Name“ und „Pfandbetrag“ wandern in „Getränkebehälter“
 - ◆ „Name“ wird in „Behältertyp“ umbenannt
 - ◆ Die Klasse „Behältertyp“ wird eliminiert



Beispiel ▶ Fortsetzung der Textanalyse nach Abbott ▶ Methoden extrahieren

● Verben → Methoden

„Die Pfandrückgabemaschine (PRM) wird in Supermärkten aufgestellt um wiederverwendbare Getränkebehälter zurückzunehmen (leere Dosen, Flaschen und Kästen, welche Endbenutzern geliefert werden). Für jeden Behältertyp kann unsere Betreiberin einen individuellen Pfandbetrag einstellen.“



Beispiel ▶ Ausschnitt aus Abbott-Tabelle

Aussage	Wort	Wortart	Modellelement
PRM nimmt Getränkebehälter zurück	zurücknehmen	Verb	Methode "accept"
PRM akzeptiert Dosen, Flaschen, Kästen	PRM	Nomen (Subjekt)	Klasse "PRM" Empfänger der "accept"-Nachricht
PRM akzeptiert Dosen, Flaschen, Kästen	Dosen, Flaschen, Kästen	Nomen (Objekte)	Klassen Parameter der Methode "accept"
PRM druckt Quittung	druckt	Verb	Methode "printReceipt"
PRM erzeugt Bericht	erzeugt	Verb	Methode "generateReport" Report generateReport()
Behältertyp	Behältertyp	Nomen	Klasse "Container"
Getränkebehälter (Dose, ...)	???	???	???
Dose ist Behältertyp	ist	Verb	Behälter ist Generalisierung von Dose
Flasche ist Behältertyp	ist	Verb	Behälter ist Generalisierung von Flasche
Kasten ist Behältertyp	ist	Verb	Behälter ist Generalisierung von

Beispiel ▶ Fortsetzung der Textanalyse nach Abbott ▶ Alles Weitere

- Fortsetzen für weitere Sprachelemente (Z.B. Adjektive)
- Anwendung des Ganzen auf den Rest der Aufgabenstellung

„Die Pfandrückgabemaschine (PRM) wird in Supermärkten aufgestellt um wiederverwendbare Getränkebehälter zurückzunehmen (leere Dosen, Flaschen und Kästen, welche Endbenutzern geliefert werden). Für jeden Behältertyp kann unsere Betreiberin einen individuellen Pfandbetrag einstellen.

Anstelle der Rückgabe von Münzgeld druckt die PRM eine Quittung über die Summe der entstandenen Pfandbeträge. Diese Quittung kann durch einen Bar Code Scanner eingelöst werden.

Die PRM generiert für unsere Betreiberin täglich einen Bericht. Ein Bericht beinhaltet eine Liste aller an diesem Tag zurückgegangenen Behälter, sowie die Tagessumme an ausgezahltem Pfand-Geld.“

Beispiel ▶ Endergebnis für gesamten Text

class Supermarkt-Pfandrücknahme

Betreiberin

pfandEinstellen(
Typ,Betrag)

Supermarkt

aufstellen(PRM)

aufgestellt in

Pfandrückgabemaschine

zurücknehmen(Behälter)
reportErzeugen : Report
quittungDrucken

► bestimmt Pfand für

Getränkebehälter

Behältertyp
Pfandbetrag:Double

setzePfand(Double)
setzeTyp()

Supermarkt

aufstellen(PRM)

aufgestellt in

Bericht

/summe : Double
/Erstattungen
summe():Double

Quittung

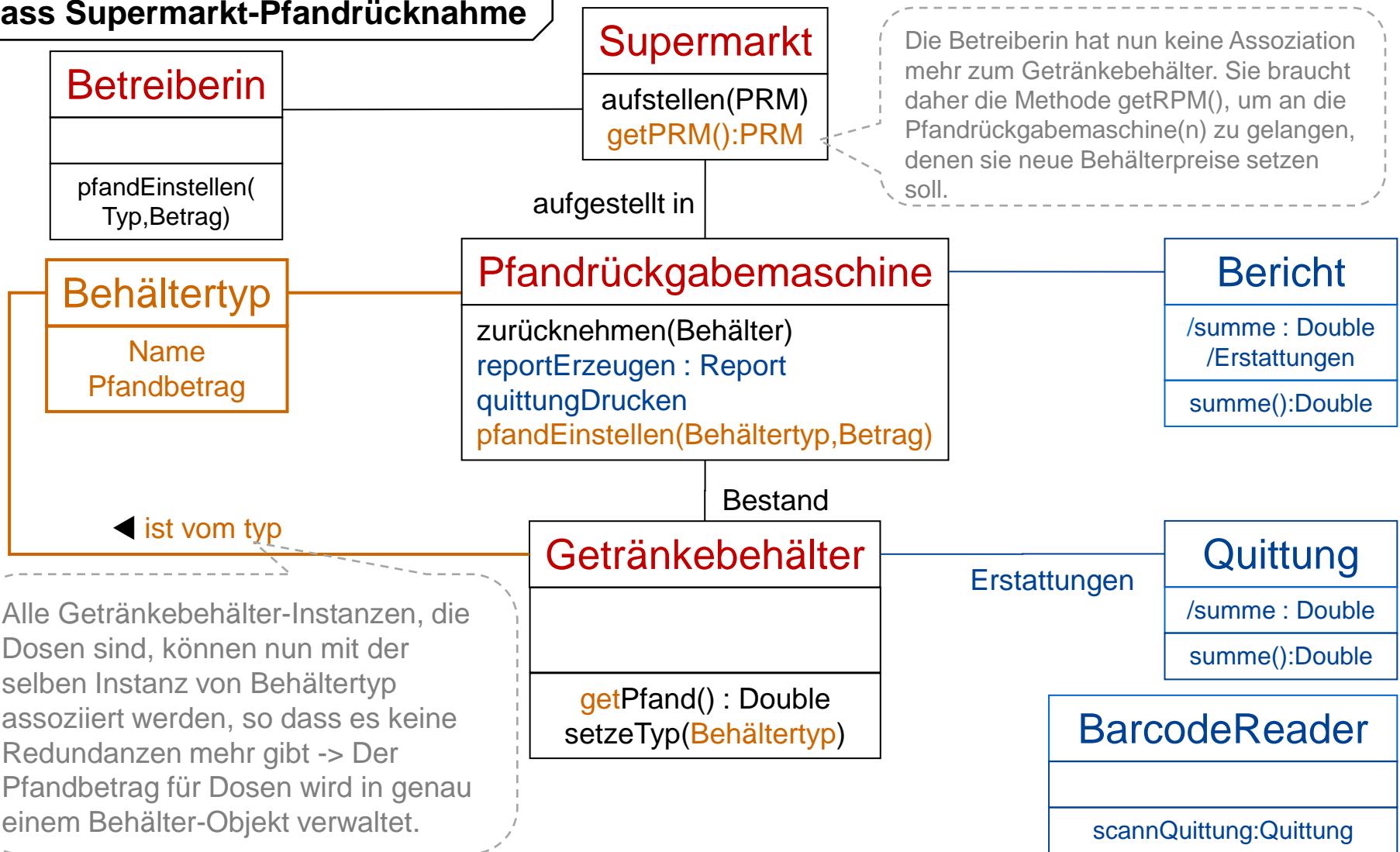
/summe : Double
summe():Double

BarcodeReader

scannQuittung:Quittung

Beispiel ▶ Endergebnis weitergedacht

class Supermarkt-Pfandrücknahme



6.5 Dynamische Modellierung von Anwendungsfällen und Geschäftsprozessen

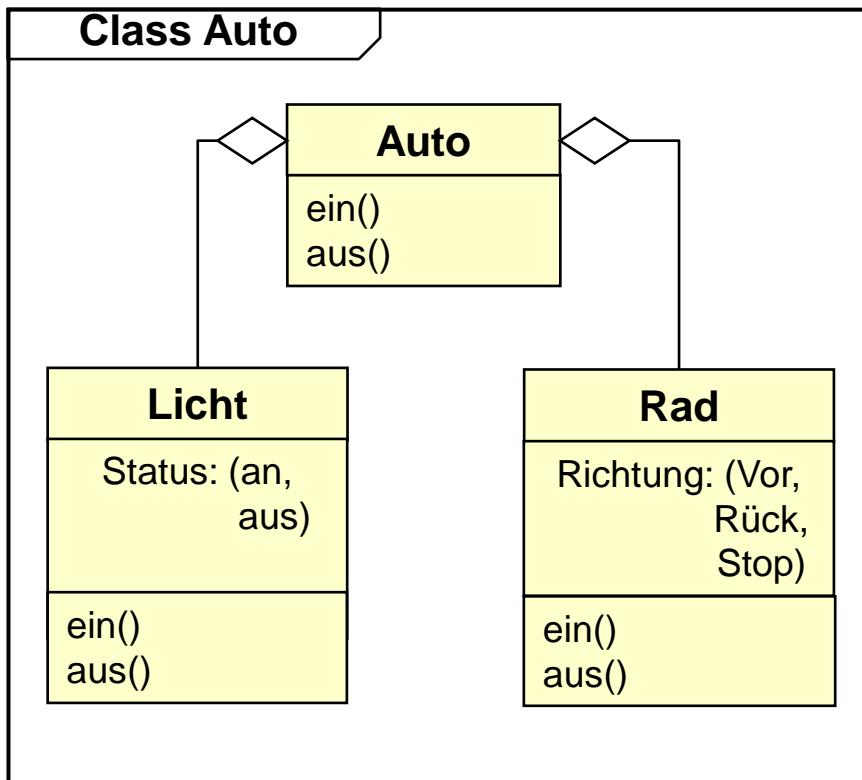
Wir haben die Use Cases und nun auch ein statisches Modell
→ Aber was ist mit dem Verhalten?

Spielzeugauto ▶ Problembeschreibung

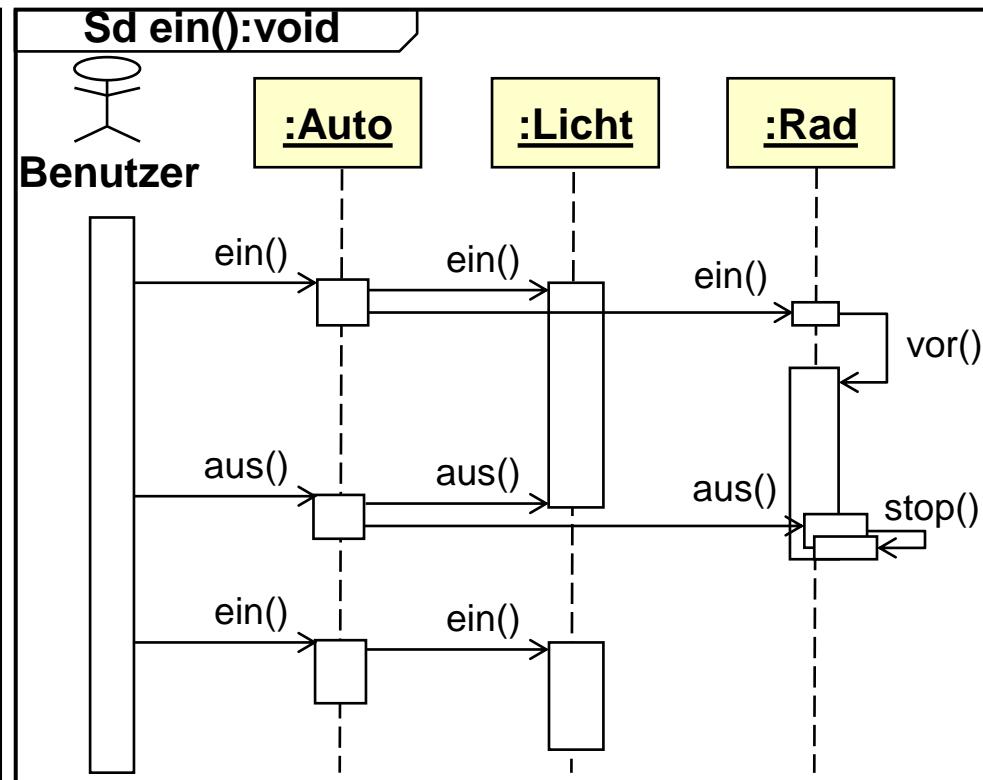
-
- Strom wird angeschaltet
→ Auto fährt vorwärts
 - 🟡 Scheinwerfer leuchtet
 - Strom wird ausgeschaltet
 - 🔴 STOP Auto hält an
 - Scheinwerfer geht aus
 - Strom wird angeschaltet
 - 🟡 Scheinwerfer leuchtet
 - Strom wird ausgeschaltet
 - Scheinwerfer geht aus
-
- Strom wird angeschaltet
← Auto fährt rückwärts
 - 🟡 Scheinwerfer leuchtet
 - Strom wird ausgeschaltet
 - 🔴 STOP Auto hält an
 - Scheinwerfer geht aus
 - Strom wird angeschaltet
 - 🟡 Scheinwerfer leuchtet
 - Strom wird ausgeschaltet
 - Scheinwerfer geht aus

Spielzeugauto ▶ Verschiedene Modelle

Klassendiagramm



Sequenzdiagramm

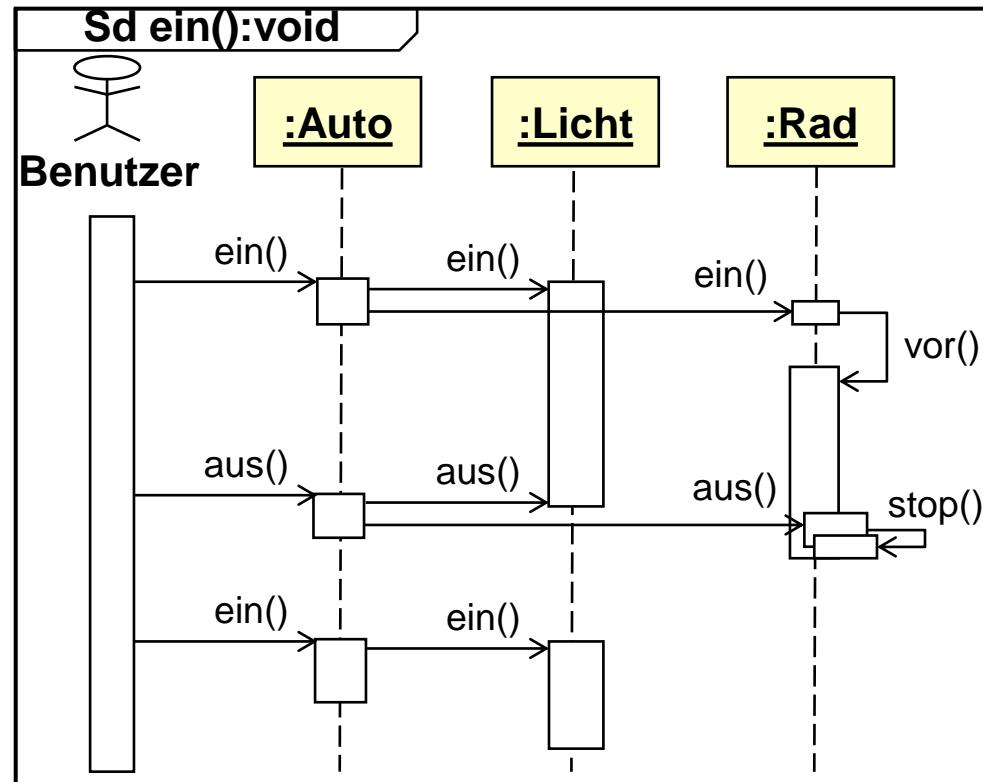


- Sagt nichts über das Verhalten
- Jedes “ein()” tut etwas anderes!

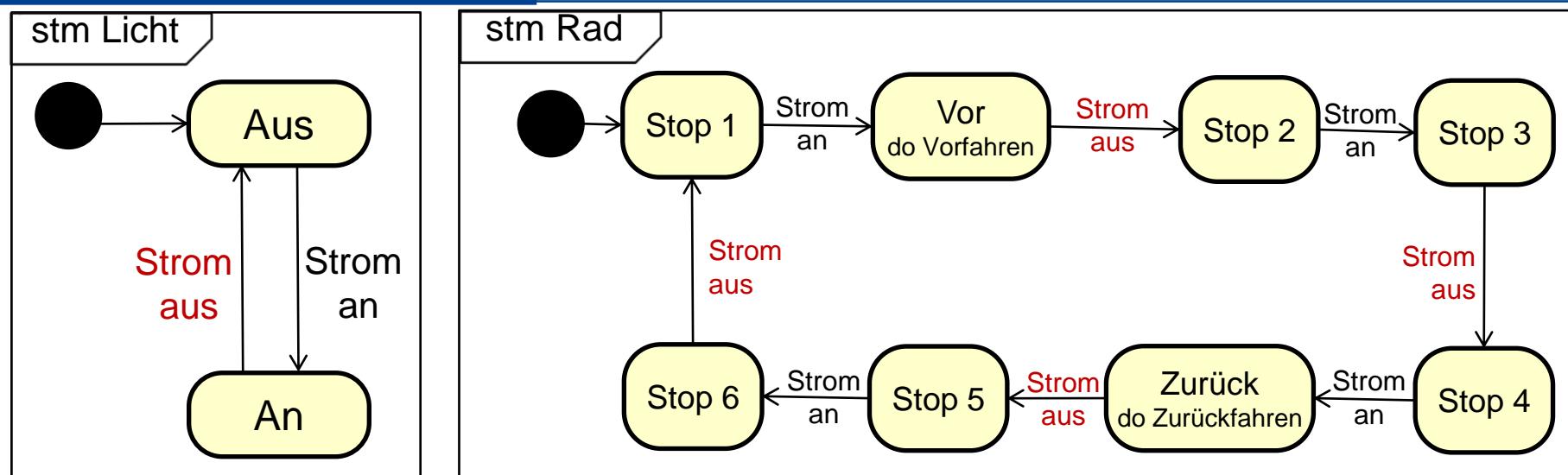
- Modelliert asynchrones Verhalten
- ... allerdings nur einen Ausschnitt

Spielzeugauto ▶ Bewertung des Sequenzdiagramms

- Drückt nicht aus, dass sich die Vorgänge beliebig wiederholen können
 - ◆ keine feste Wiederholungszahl
 - ◆ keine feste Endbedingung
- Dass beim nächsten Einschalten des Stroms nach dem Stop die Räder still stehen wurde im Auto modelliert
 - ◆ Das Auto schickt in diesem Fall keine Nachricht an die Räder
 - Das Auto ist für die Modellierung des Verhaltens der Räder mit zuständig
 - ◆ Ist das gut oder schlecht?



Spielzeugauto ▶ Zustandsdiagramm



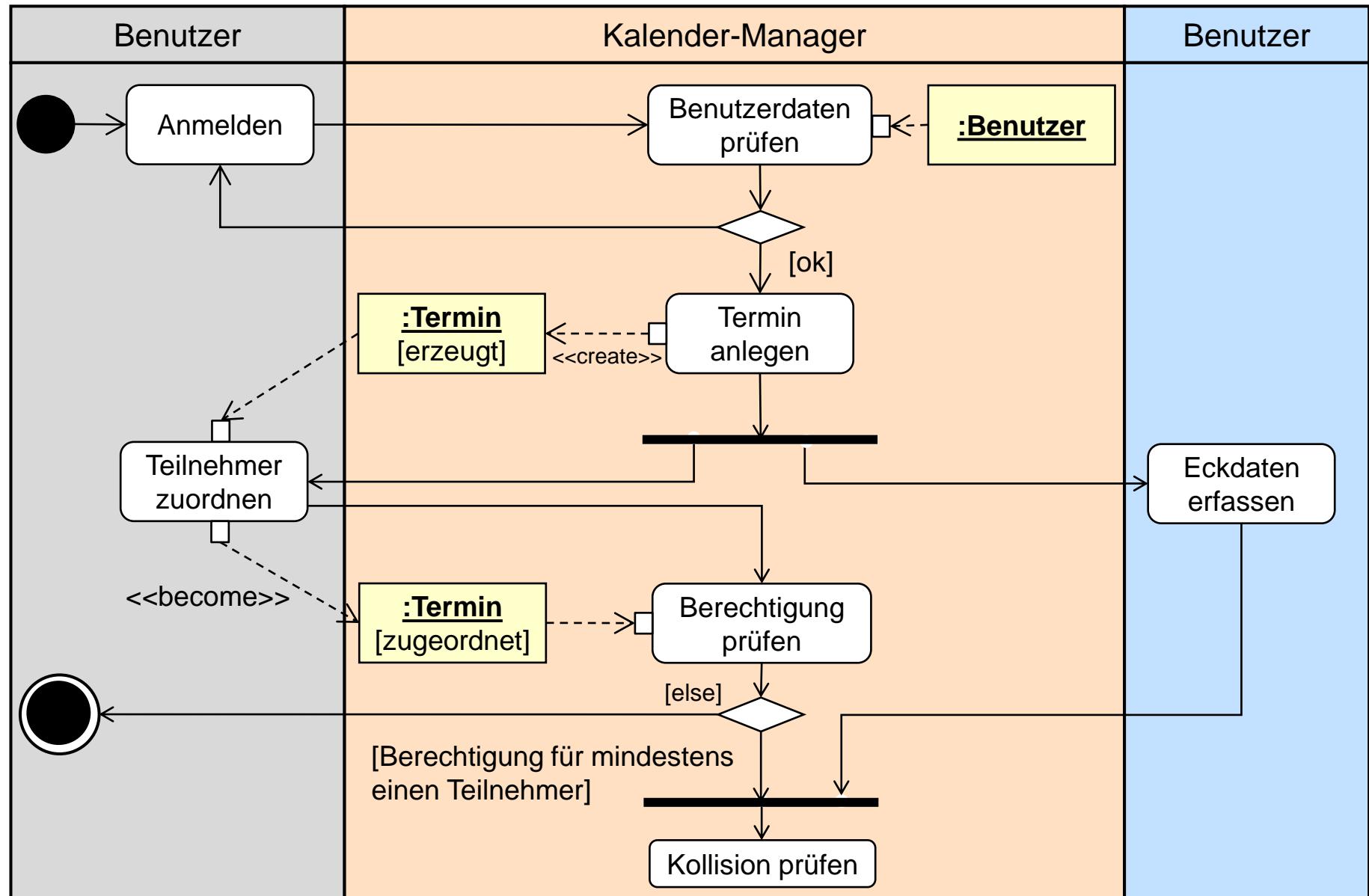
- Hier wird das Verhalten von Licht und Rädern komplett und kompakt ausgedrückt – einschließlich der Unabhängigkeit der beiden Aspekte
 - ◆ „Die Moral von der Geschicht“ ▶ Oft ist die Wahl der richtigen Notation schon der wichtigste Teil der Lösung geleistet.
 - ◆ Um so wichtiger, genau zu verstehen, was welcher Diagrammtyp ausdrücken kann.

Dynamische Diagramme zur Beschreibung von Geschäftsprozessen

- Motivation
 - ◆ Geschäftsprozesses in den der Use Case eingebettet ist, ist noch unklar
 - ◆ Sein Verständnis ist aber wichtig, um denn Sinn des Anwendungsfalls im Gesamtzusammenhang des Unternehmens besser zu verstehen
- Vorgehen bei einfachen Abläufen / wenigen Varianten
 - ◆ Modellierung mit Sequenz- oder Kommunikationsdiagrammen möglich
 - ⇒ eventuell mehrere Diagramme für die wichtigsten Abläufe
 - ◆ Alternative: Aktivitätsdiagramm für eine Gruppe von Ablaufvarianten
 - ⇒ Entscheidungsknoten im Diagramm darstellen
- Vorgehen bei komplexen Abläufen / vielen Varianten
 - ◆ mehrere Aktivitätsdiagramme
- Beispiel
 - ◆ „Termin erfassen“ (Nächste Folie)

Beispiel ▶ Aktivitätsdiag. mit Produkten und „Swimlanes“

act Termin erfassen



6.6 Zusammenfassung

Was Sie sich zur Anforderungserhebung mindestens merken sollten:

- Konzepte
- Aktivitäten
- Produkte

Anforderungserhebung ▶ Wichtigste Konzepte

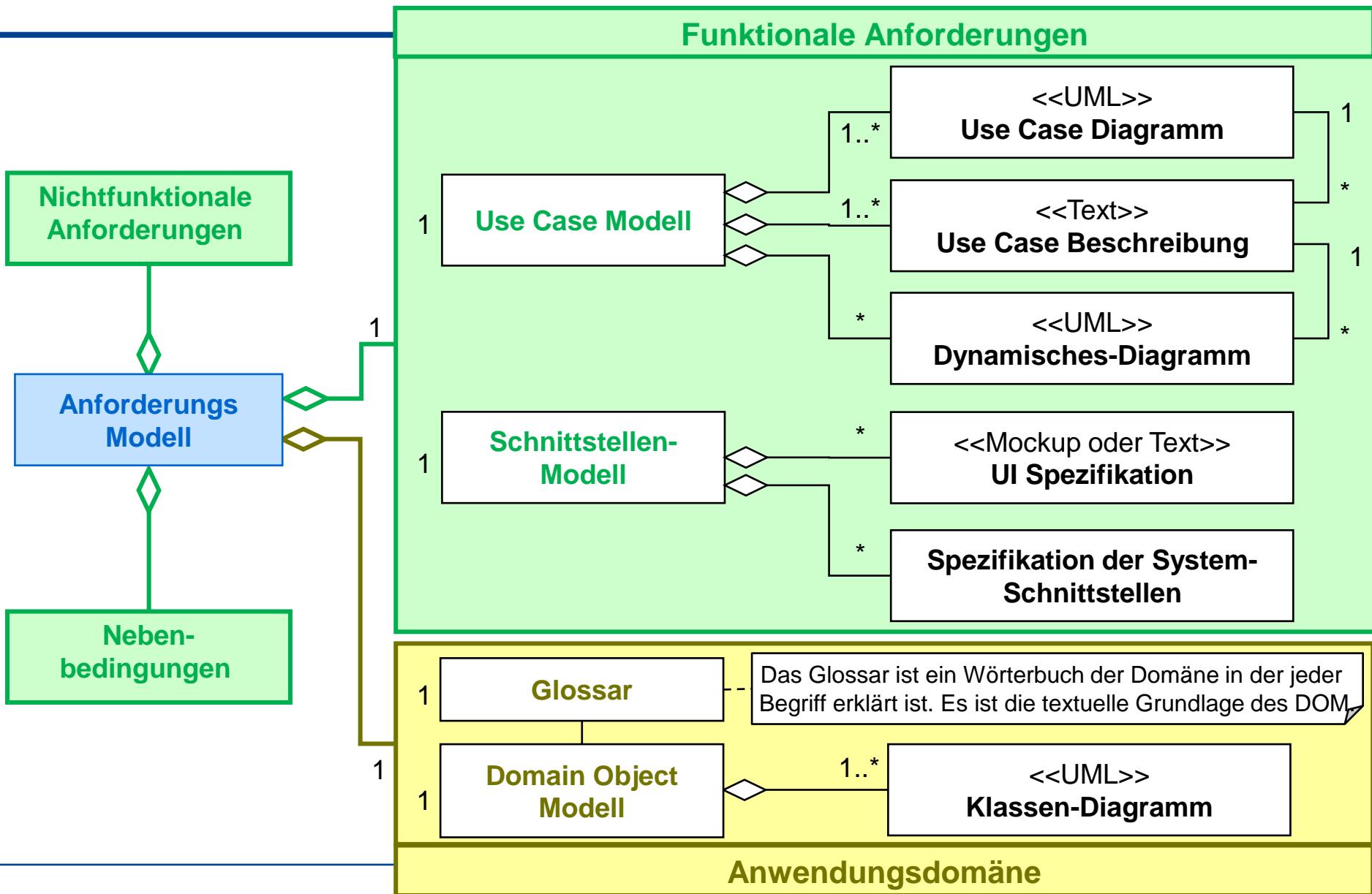
- Szenarien
 - ◆ Großartiger Weg zur Kommunikation mit dem Kunden
 - ◆ As-Is Szenarien, Visionäre Szenarien, Evaluationsszenarien, Training Szenarien
- Use Cases
 - ◆ Abstraktion von Szenarien
 - ◆ Use Case Modell erfasst vor allem funktionale Anforderungen
- Domain Object Modell
 - ◆ Klassendiagramme erfassen Anwendungsdomäne in strukturierter Form
- Verfahren von Abbott
 - ◆ Textanalyse als Hilfe zur Objektidentifikation
 - ◆ Strukturierung / Verfeinerung des Objektmodels anhand simpler Prinzipien

Anforderungserhebung ▶ Aktivitäten

- Aktivitäten der Anforderungserhebung
 - ◆ Erhebung von Szenarien
 - ◆ Abstraktion und Strukturierung von Use Cases
 - ◆ Identifikation beteiligter Objekte (Domain Object Modell)
 - ◆ Spezifikation von elementarem Verhalten

- Sie dienen der
 - ◆ Festlegung der Intention und des Umfanges eines Systems
 - ◆ Erfassung der Anforderungen aus Anwender-Sicht in einer für den Anwender noch nachvollziehbaren Form

Anforderungserhebung ▶ Produkte



Anforderungserhebung ▶ Produkte

● Use Case Modell

- ◆ Beschreibung jedes Anwendungsfalls durch strukturierten Text
- ◆ Beschreibung der Beziehungen der Anwendungsfälle durch ein oder mehrere Diagramme
- ◆ Evtl. Beschreibung der Abläufe komplexer Anwendungsfälle durch dynamische Diagramme
- ◆ Evtl. Beschreibung der Geschäftsprozesse in die die Anwendungsfälle eingebettet sind durch dynamische Diagramme.

● Schnittstellen-Modell

- ◆ Mockups (= mit Papier, Farben, etc. simulierte Benutzeroberflächen)
- ◆ System-Schnittstellen-Beschreibung (textuell oder Klassendiagramm)

● Glossar

- ◆ Wörterbuch der Anwendungsdomäne
- ◆ Begriffe sind durch freien Text definiert

● Domain Object Modell

- ◆ Erfassung der wichtigsten Konzepte des Glossars und ihrer Beziehungen durch ein Klassendiagramm / Klassendiagramme

Kapitel 7

Anforderungsanalyse („Requirements Analysis“)

Stand: 13.12.2023

-
- 7.1 Das Analyse-Modell
 - 7.2 Objektmodellierung im Analyseworkflow
 - 7.3 Erstellung des Analyse-Objektmodells
 - 7.4 Verfeinerung des dynamischen Modells
 - 7.5 Modellierung der Dynamik von GUIs
 - 7.6 Konsolidierung der Analyse

Themenbereiche und Vorlesungs-Kapitel

III. PROZESSE

12

Agile Softwareentwicklung

11

Software-Prozess-Modelle

II. AKTIVITÄTEN



10

Test

9

Objekt-Design

8

System-Design

7

Anforderungs-Analyse

6

Anforderungs-Erhebung

I. WERKZEUGE

5

Refactoring

4

Entwurfsmuster

3

Unified Modelling Language (UML)

2

OO-Modellierung

1

OO-Programmierung

VORWISSEN

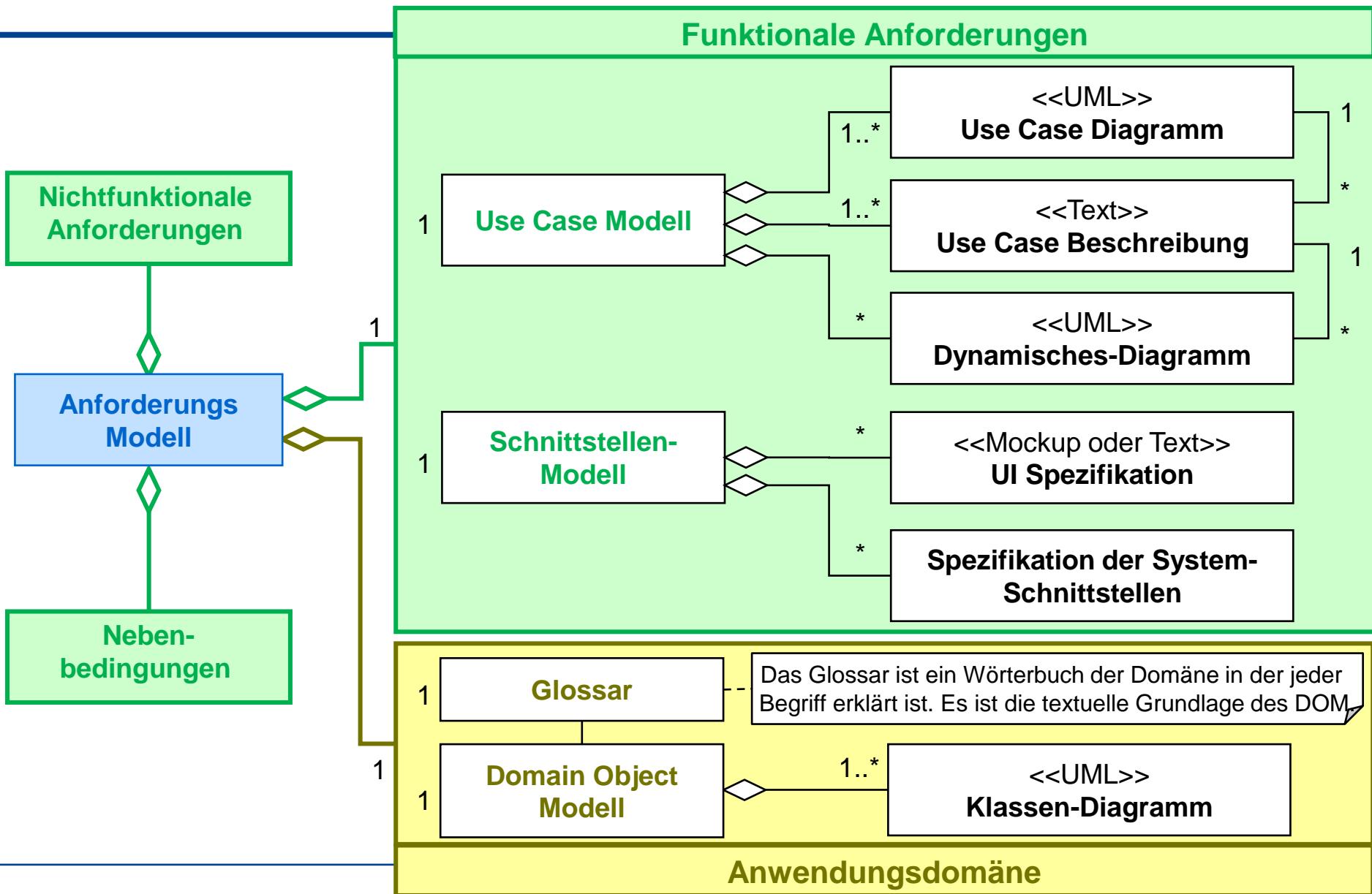
Objektorientierte-Programmierung

Automatisiertes Testing (mit jUnit)

Imperative-Programmierung

Software Configuration Management (mit Git)

Anforderungserhebung ▶ Produkte



Inhaltsübersicht

- 1) Das Analyse-Modell
 - ◆ Unterschiede Use Case Modell ↔ Analysemodell
- 2) Objektmodellierung im Analyse-Workflow
 - ◆ Boundaries, Controller und Entities
- 3) Dynamische Modellierung
 - ◆ Interaktionen zwischen Boundaries, Controller und Entities
 - ◆ Zustandsdiagramme für Typen mit komplexem Verhalten
- 4) Modellierung der GUI
 - ◆ Layout der Boundaries
 - ◆ Interaktion mit und Navigation durch Boundaries
- 5) Konsolidierung der Analyseergebnisse
 - ◆ Redundanzen, Mehrdeutigkeiten, Widersprüche, ... eliminieren

Aktivitäten bei der Anforderungserhebung und -analyse

Anforderungserhebung

- Identifiziere Akteure
- Identifiziere Szenarien
- Identifiziere Use Cases
- Identif. Beziehungen zwischen Use Cases
- Identif. nichtfunktionale Anforderungen
- Identifiziere Nebenbedingungen
- Identifiziere Objekte der Anwendungsdomäne
- Erstelle Glossar

Anforderungsanalyse

- Identifiziere Realisierungs-Objekte:
 - Boundary, Controller, Entity
- Identifiziere Operationen und Methoden
- Identifiziere Assoziationen
- Identifiziere Attribute
- Modelliere Objektinteraktionen
 - Interaktionsdiagramme
- Modelliere nichttriviales internes Verhalten
 - Zustandsdiagramme

Use Cases in Objektstruktur übersetzen

Use Cases in Objektverhalten übersetzen

Von beobachteter

Mehrdeutigkeit, Inkonsistenz, Unkorrektheit, Unvollständigkeit, Undurchführbarkeit
zur Verfeinerung der Anforderungen

7.1 Das Analyse-Modell

Abgrenzung Anforderungserhebung zu Anforderungsanalyse

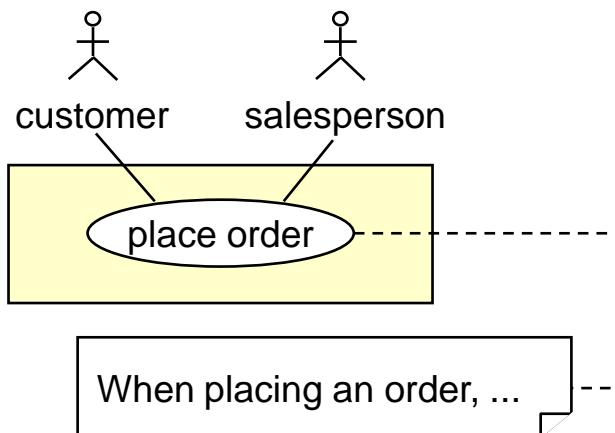
Use Case vs. Analyse Modell (1)

Use Case Modell

- in der Terminologie der Anwender verfasst

- externe Sicht des Systems
(was tut es?)

- in „use cases“ strukturiert

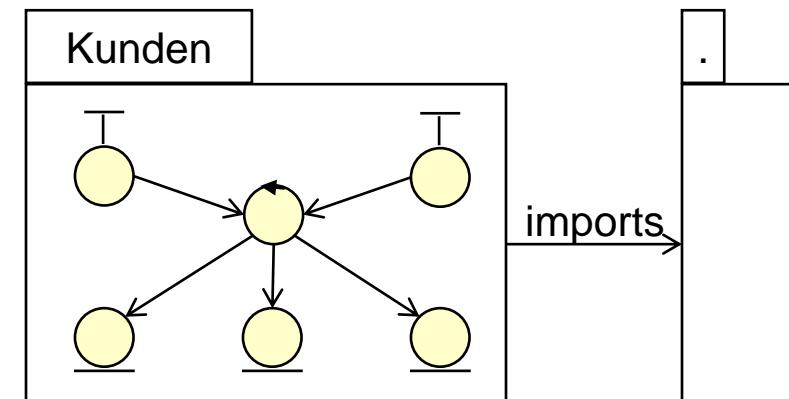


Analyse Modell

- in der Terminologie der Entwickler verfasst

- interne Sicht des Systems
(wie tut es das?)

- in Module und Klassen-Stereotypen strukturiert



Use Case vs. Analyse Modell (2)

Use Case Modell

- dient vorwiegend als Kontrakt zwischen Auftraggeber und Entwickler hinsichtlich der Anforderungen an das System

Analyse Modell

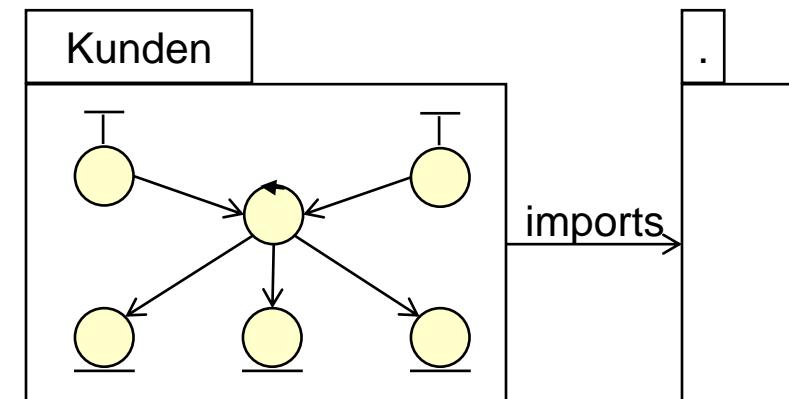
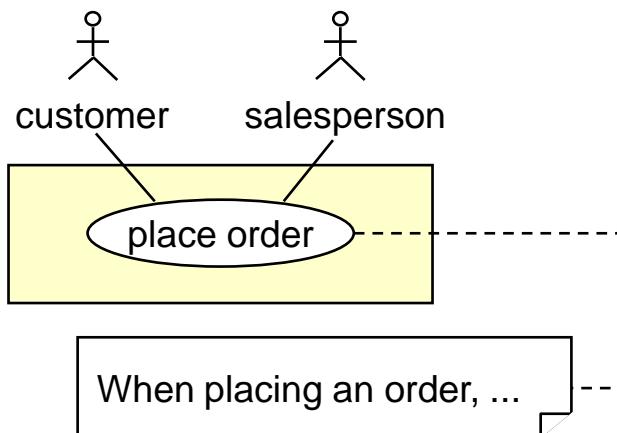
- kann redundante / inkonsistente Anforderungen enthalten

- dient den Entwicklern zum Verständnis der System-Struktur

- definiert Use Cases, die im Analyse-Modell weiter vertieft werden

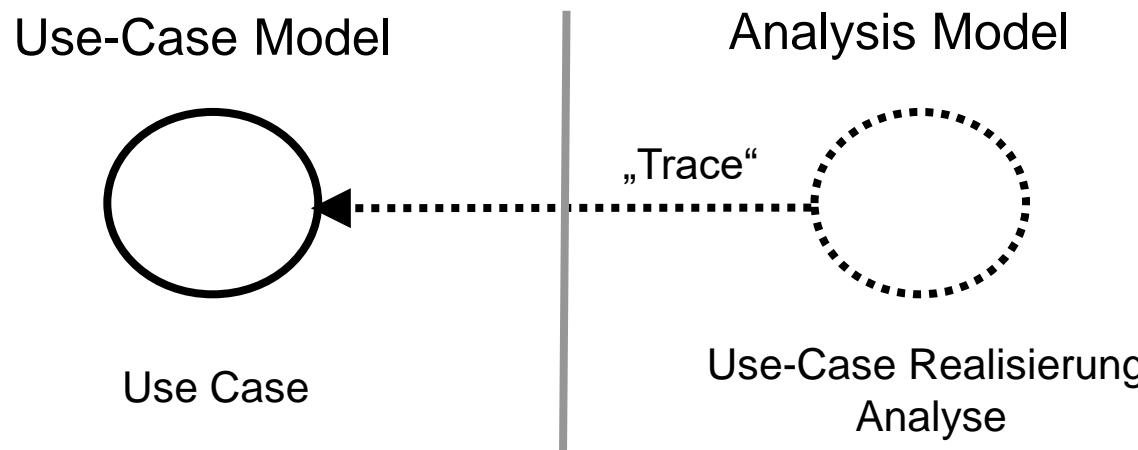
- darf keine Redundanzen / Inkonsistenzen mehr enthalten

- definiert die Use-Case-Realisierungen



Use-Case Realisierung – Analyse

- Beschreibt die Umsetzung eines Use-Case im Analyse-Modell
 - ◆ auf hoher Abstraktionsebene
 - ◆ enthält keine Implementierungsdetails (verwendete Bibliotheken, Application Server, ...)
- Besteht aus
 - ◆ Klassendiagrammen ← Verfeinerung des DOM
 - ◆ Dynamischen Diagrammen ← Verfeinert oder neu
 - ◆ GUI-Mockups und Navigationsverhalten ← neu!



7.2 Objektmodellierung im Analyse-Workflow

→ Aufteilung in Boundary, Controller und Entity Stereotypen

Analysetyp (Klasse oder Interface)

- Realisiert essentielle funktionale Anforderungen
 - ◆ Nicht-funktionale Anforderungen werden erst im Systemdesign behandelt
- Abstrahiert eine oder mehrere Konzepte und / oder Subsysteme
 - ◆ Definiert konzeptuelle Attribute die evtl. später zu eigenen Klassen werden
 - ◆ Beschreibt noch relativ wenige Operationen

Kategorien von Analysetypen

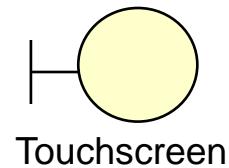
Ein Analysetyp ist stets einer der drei folgenden Stereotypen:

- **Boundary Typ** = Schnittstelle zu Akteur
- **Control Typ** = Ereignisfluss eines Use Cases
- **Entity Typ** = Anwendungskonzept („Business object“)

- Repräsentiert ein für das Anwendungsgebiet relevantes Konzept („Business object“)
 - ◆ Heuristik ▶ Jeder Typ im Domain Object Model ist ein Entity-Kandidat
 - ◆ Heuristik ▶ Jeder Begriff im Glossar ist ein Entity-Kandidat
- Hat oft Use-Case-übergreifende Bedeutung
 - ◆ Heuristik ▶ In verschiedenen Use Cases wiederkehrende Substantive identifizieren

Lebensdauer

- Entspricht oft den persistenten Informationen der Anwendung
 - ◆ Das ergibt sich aus der Use Case übergreifenden Bedeutung
 - ◆ Entities sind aber keine reinen Daten-Klassen
 - ◆ Sie können komplexes Verhalten besitzen (z.B. Zinsberechnung)

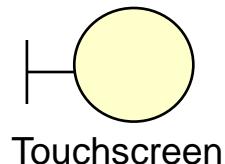


- Beschreibt Interaktionen zwischen dem System und den Akteuren
 - ◆ Heuristik ▶ Jeder Akteur sollte **nur** mit Boundaries verknüpft sein
 - ◆ Konsistenzcheck ▶ Jedes Boundary sollte mit **mindestens einem** Akteur verknüpft sein
- Typische Boundaries
 - ◆ Dateneingabeformulare und Fenster: NotfallberichtBoundary
 - ◆ Fragen und Nachrichten an den Nutzer: BestätigungsBoundary

Benennung

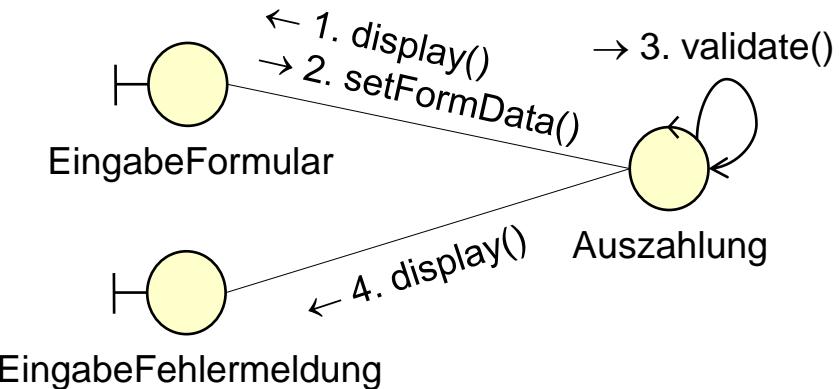
- Verwende Termini der Anwendungsdomäne plus Suffix Boundary zwecks Unterscheidung von Entities
 - ◆ Notfallbericht bezeichnet ein Entity das den Bericht darstellt
 - ◆ NotfallberichtBoundary bezeichnet das Eingabeformular dafür
- Verwende keine Implementierungsbegriffe um keine voreiligen Implementierungsentscheidungen zu suggerieren
 - ◆ **Nicht** NotfallberichtTabelle oder BestätigungsPopUp

Analysetyp ▶ Boundary ▶ Granularität



- So fein wie nötig ▶ Einheiten trennen, die im Ereignisfluss als unterschiedlich erkennbar sein müssen.

- ◆ EingabeFormular von EingabeFehlermeldung trennen, um modellieren zu können, dass letzteres vom Controller als Reaktion auf eine Fehleingabe in ersterem geöffnet wird



- So grob wie möglich ▶ Möglichst große logische Einheiten zusammenfassen.
 - ◆ Nicht jeder Knopf ist ein Boundary!
 - ◆ Eine zu feine Aufteilung würde
 - ⇒ die Modellierung unnötig komplex machen (sich in Details verlieren)
 - ⇒ dauernde Änderungen des Analysemodells für jede Änderung der graphischen Oberfläche nach sich ziehen



- Kapselt den Ereignisfluss eines Use-Case
 - ◆ Vermittlungsstelle zwischen Entities und Boundaries
 - ◆ Komplexe Berechnungen oder logische Zusammenhänge, die keiner Entity zugeordnet werden können
- Typische Aufgaben von Kontrollobjekten
 - ◆ Ablaufsteuerung in Formularen (z.B. „wizards“)
 - ◆ Command History und Undo-Funktionalität
 - ◆ Verteilung und Weiterleitung von Informationen

Benennung

- Name des Use-Case plus Suffix Control, Controller oder Command



- Standard ▶ Ein Kontrollobjekt pro Use Case
- Ausnahme ▶ Mehr als ein Kontrollobjekt pro Use Case um problemgebundene physikalische Verteilung auszudrücken
 - ◆ NotfallberichtControl ▶ im Einsatzfahrzeug
 - ◆ NotfallerfassungControl ▶ in der Notfallzentrale

Lebensdauer

- Die Lebensdauer eines Kontrollobjektes entspricht der des zugehörigen Anwendungsfalls

7.2 Objektmodellierung im Analyse-Workflow

→ Aufteilung in Boundary, Controller und Entity Stereotypen

Warum genau diese drei Stereotypen?

Die Trennung der drei Kategorien ermöglicht

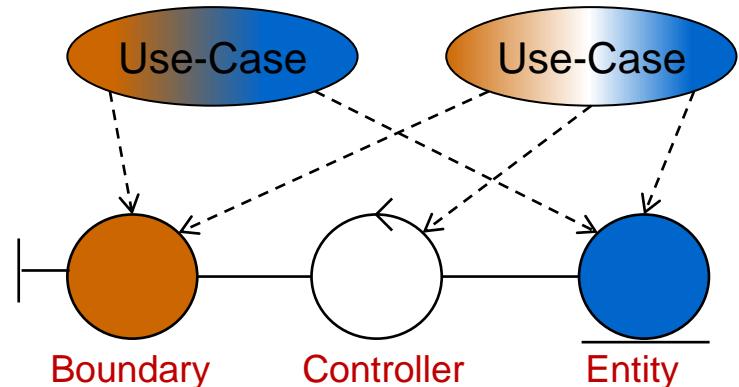
- **Stabilität** ▶ Modelle, die **weniger änderungsanfällig** sind
 - ◆ Die Grenzen eines Systems (boundaries) ändern sich häufig
 - ⇒ Darstellung auf zeilenorientiertem Terminal, im Webbrowser, ...
 - ◆ Die Geschäftsabläufe (controller) ändern sich häufig
 - ⇒ Ablauf der Kreditvergabe, Kreditwürdigkeitskriterien, ...
 - ◆ Jede dieser Änderungen soll den Rest des Systems nicht beeinflussen
 - ⇒ Insbesondere sollte die Anwendungsdomäne (entities) möglichst stabil bleiben
 - ⇒ Konten, Kredite, Zinsen, Tilgung, etc.
 - **Flexibilität** ▶ Modelle die **leichter kombinierbar** sind
 - ◆ Verschiedene Geschäftsabläufe und Oberflächen sollten möglichst frei miteinander kombinierbar sein
 - ⇒ Beispiel: Kreditwürdigkeitsprüfung via Webbrowser oder Java-Oberfläche
- Herkunft ▶ MVC Framework in Smalltalk 76
- „Entity, Boundary, Controller“ = „Model, View, Controller“ (MVC)

Analyse-Objektmodell

- Weiterentwicklung des "Domain Object Model"
 - ◆ Entity-Typen ▶ Aus DOM übernommen (evtl. auch ein paar Neue)
 - ◆ Kontroll- und Boundary-Typen ▶ Während der Analyse hinzugefügt

Aufteilung von Use-Cases auf Typen im Analyse-Objektmodell

- Jeder Use-Case verteilt sich auf mehrere Analysetypen
 - ◆ Boundary, Controller, Entity
- Gleicher Analysetyp kann Aspekte mehrerer Use Cases beinhalten

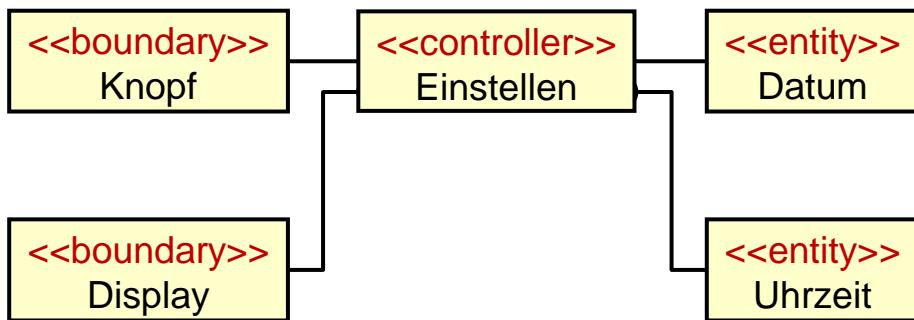


Layout von Analyse-Klassendiagrammen

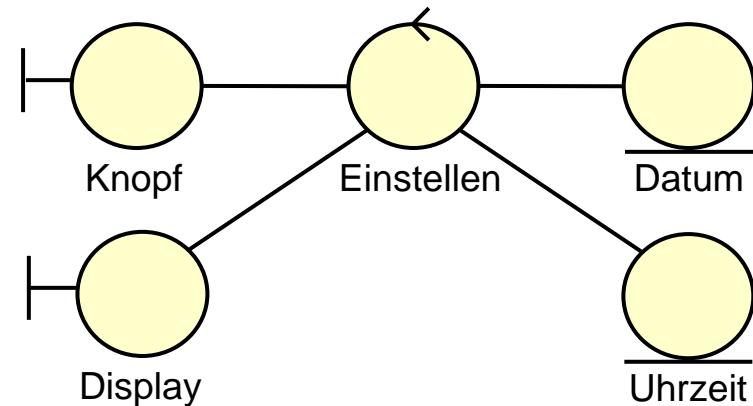
- Boundaries links, Controller in der Mitte, Entities rechts

Analyse-Objektmodell „Digitaluhr“

Textuelle Notation*



Graphische Notation*



* = Beziehungsnamen, Rollen, Kardinalitäten, Attribute und Operationen aus Platzmangel unterschlagen.

- Man kann die Stereotypen für Analyse-Klassen graphisch oder textuell notieren
- Klassendiagramme des Analysemodells die die drei Stereotypen nutzen heißen auch „**Robustness Diagrams**“ → Warum wohl?
- Vergleichen Sie obiges Diagramm mit dem Klassendiagramm der Digitaluhr im OOM-Foliensatz und diskutieren Sie die Unterschiede!

Analyse-Objektmodell → Verbotene Abhängigkeiten

- Verbotene Abhängigkeiten zwischen Analyse-Typen

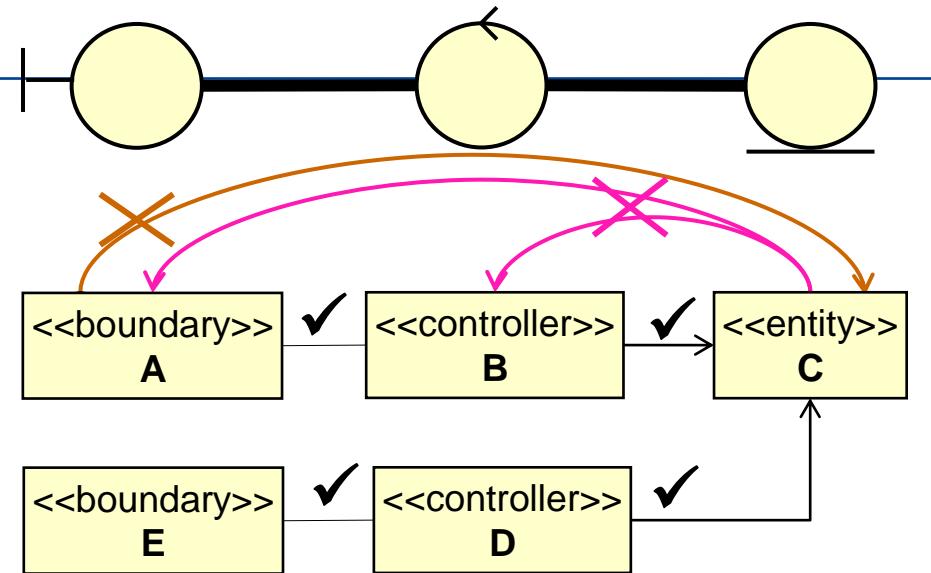
- a) Boundary zu Entity
- b) Entity zu Boundary
- c) Entity zu Controller

- Sinn des Verbots a)

- Boundaries sollen keine Kontrollaufgaben übernehmen
- Wartbarkeit und automatische Codeerzeugung für graph. Oberflächen

- Sinn der Verbote b,c)

- Entity-Typen unempfindlich machen gegen Änderungen in anderen Typen
 - ⇒ C muss nicht geändert werden wenn sich A, B, D oder E ändert
- Nutzung eines Entity-Typs in mehreren Use Cases (= Controller-Typen) möglich
 - ⇒ Wenn C auf Interaktionen mit A und B ausgelegt wäre, könnte es nicht von D genutzt werden



Erstellung des Analyse-Objektmodells

Bisher: Prinzipien und Motivation des Analyse-Objektmodells

Nun: [Workflow](#) zur Erstellung des Analyse-Objektmodells

1. Allgemeiner Weg von Use Cases zu Boundaries, Controllern und Entities
2. Illustration der Workflowschritte anhand einer kleinen Bankanwendung

Workflow

Ausgangspunkt

(0) Vorhandenes Use-Case-Modell und Domain Object Model

Verfahren (pro Use Case)

(1) Identifikation des Controllers und der Boundaries, (und Entities)

- ◆ Erzeugung entsprechender Klassen
- ◆ Erzeugung ihrer Assoziationen

(2) Eventuell Ergänzung von Entities

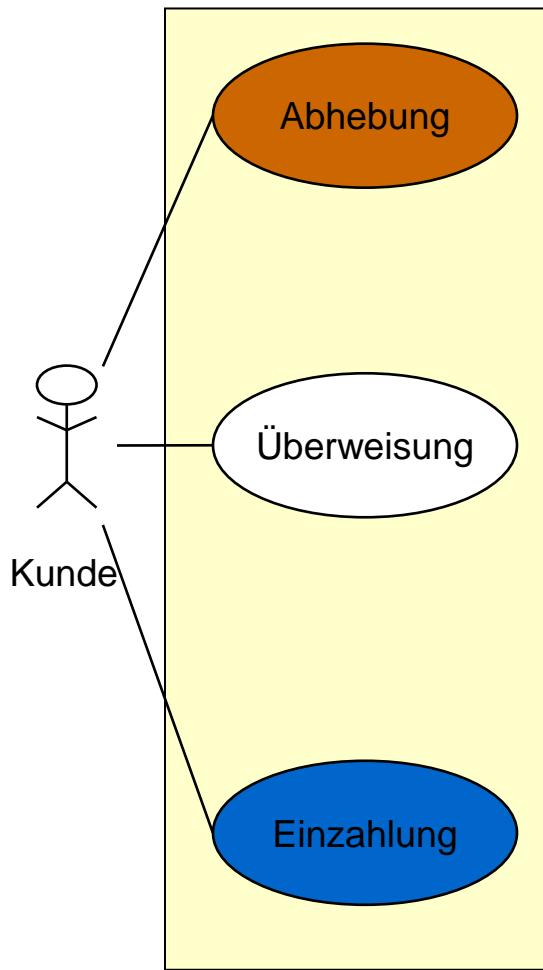
- ◆ Möglicherweise vorher übersehene Domänen-Konzepte

(3) Konsolidierung des erweiterten statischen Modells

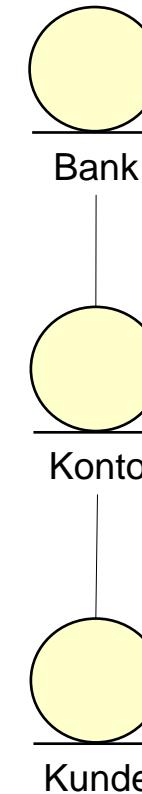
- ◆ Zusammenfassung von Klassen mit ähnlichen oder identischen Funktionen

Beispiel: Ausgangspunkt

Use-Case Modell

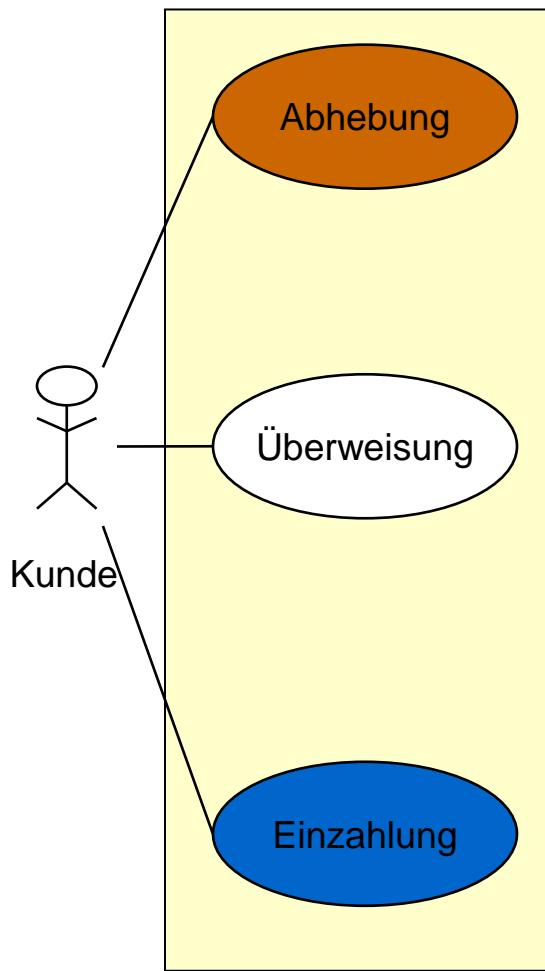


Domain-Objektmodell

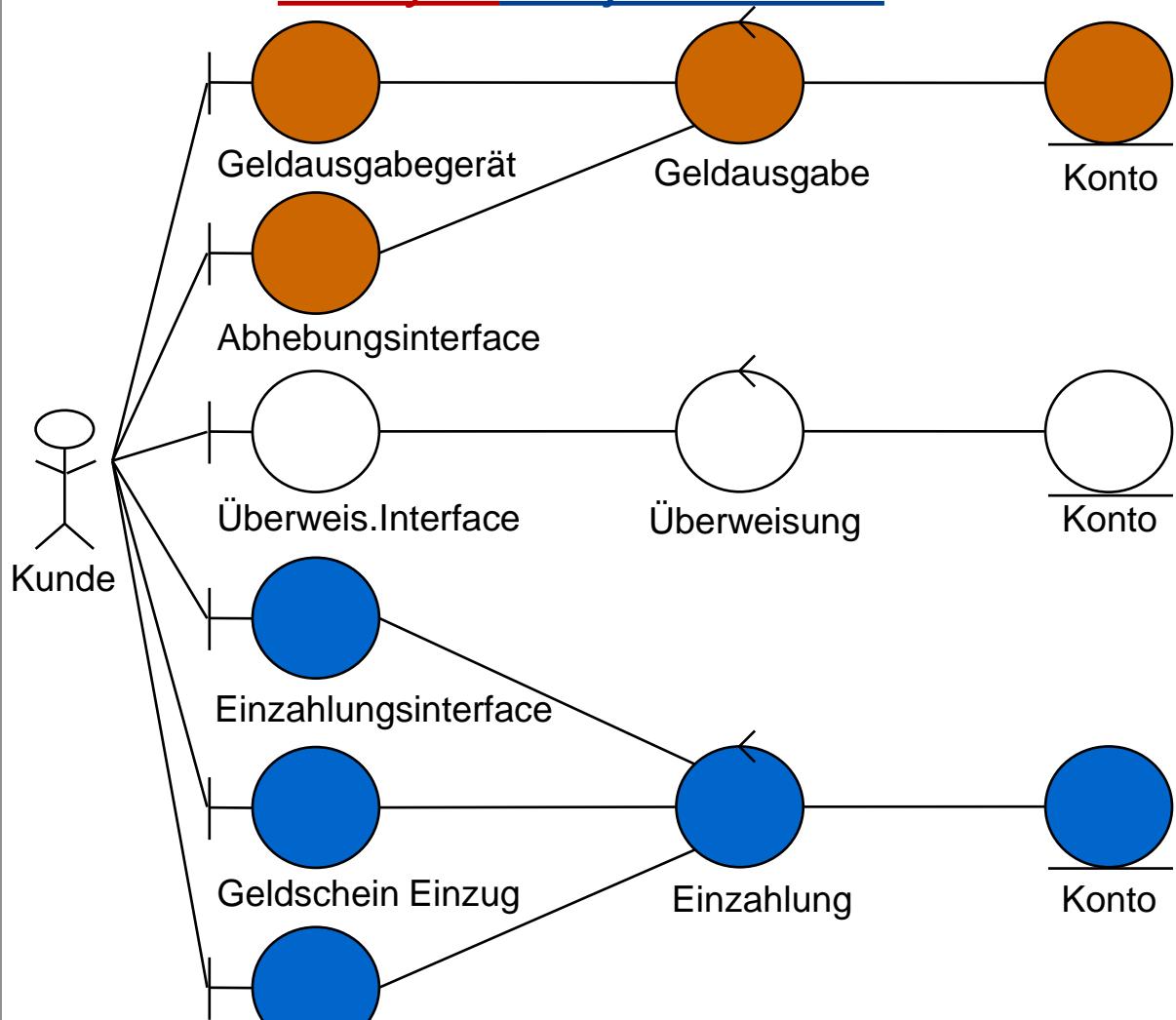


Use-Case Modell → Analyse-Objektmodell

Use-Case Modell



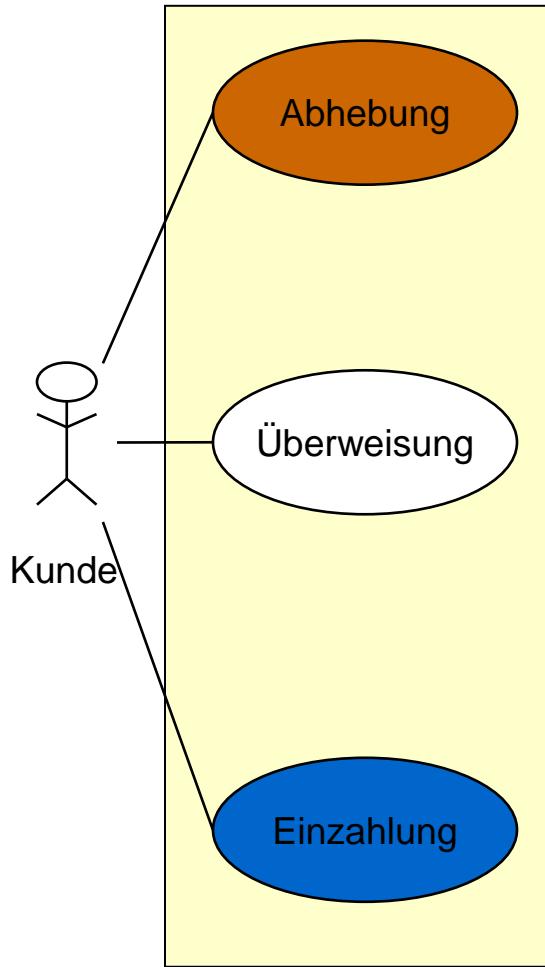
Analyse-Objektmodell



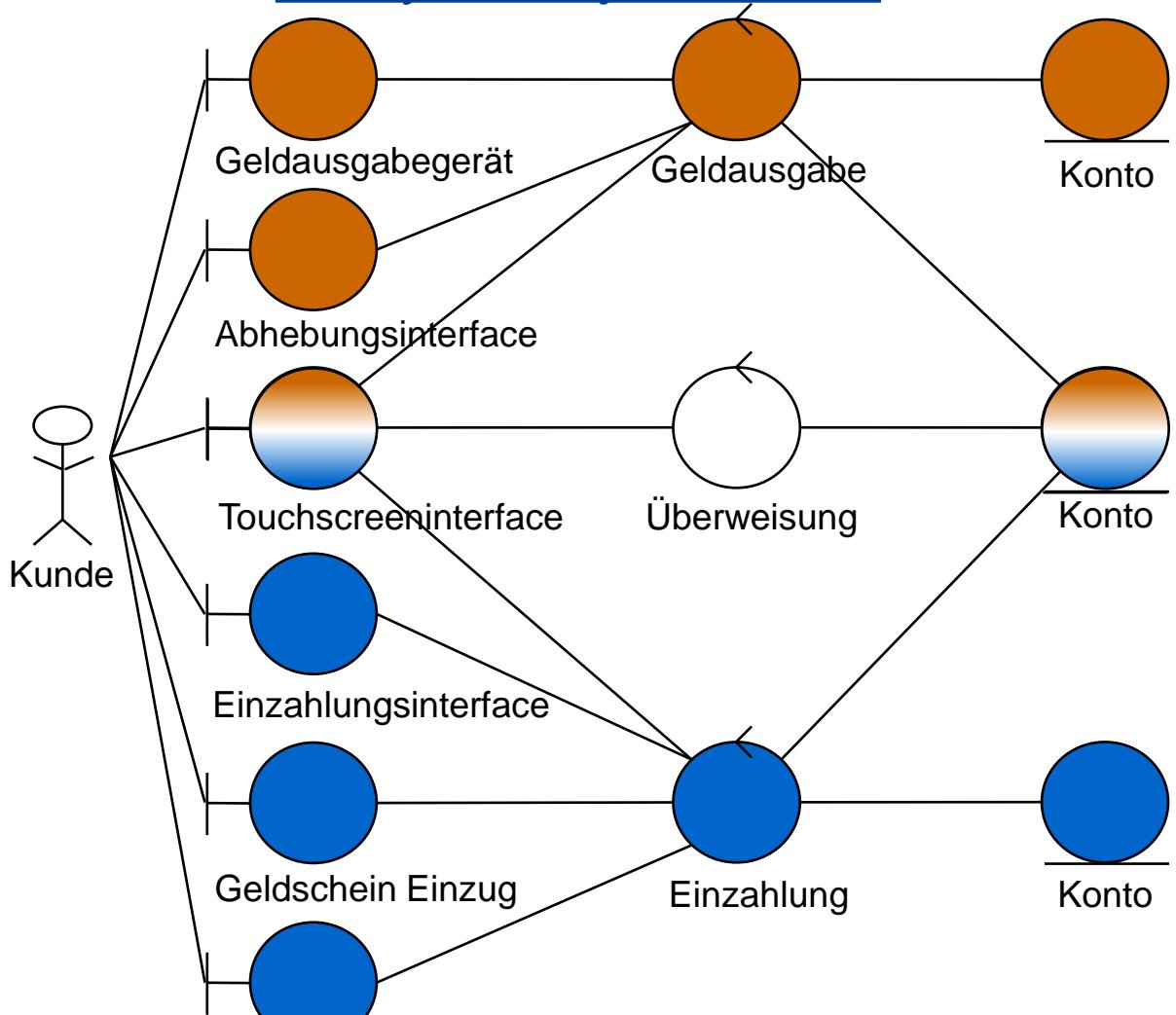
Use-Case Modell → Analyse-Objektmodell

Show

Use-Case Modell

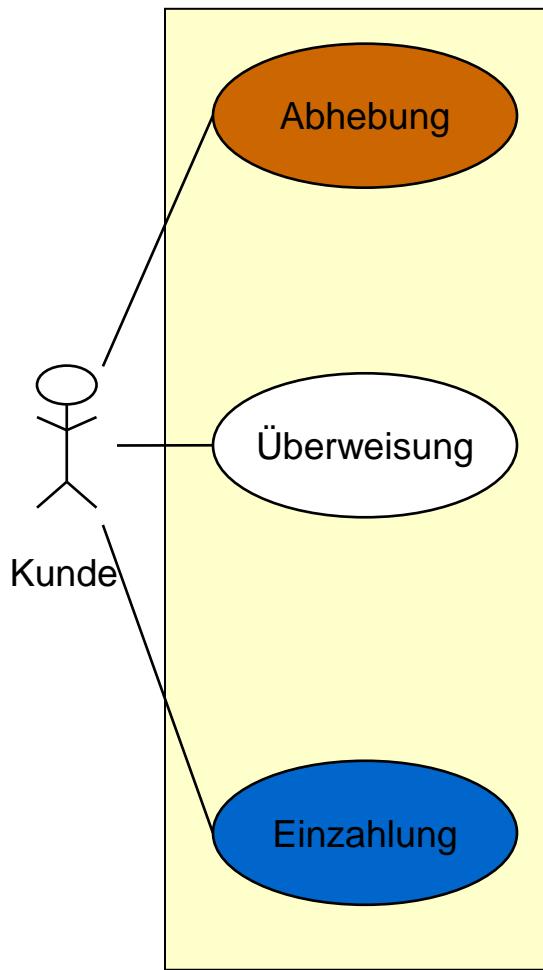


Analyse-Objektmodell

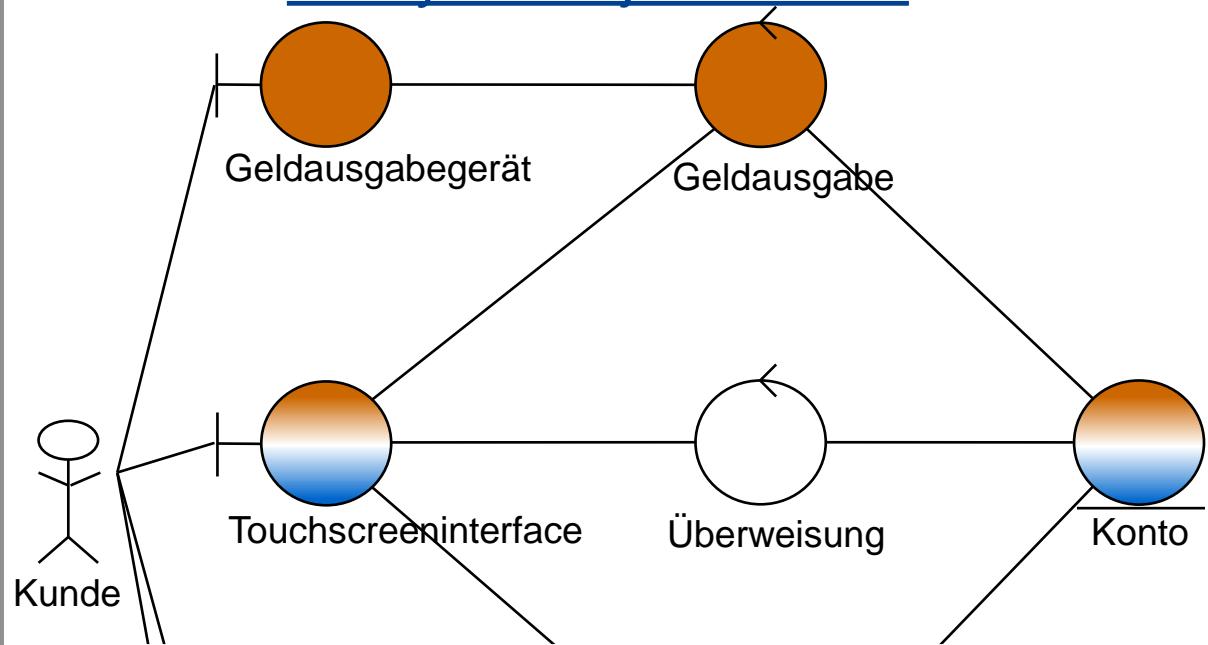


Use-Case Modell → Analyse-Objektmodell

Use-Case Modell



Analyse-Objektmodell



Zusammenfassung von Objekten aus verschiedenen Use-Case-Realisierungen, die ...

- ... konzeptuell identisch sind
 - Konto
- ... ähnliches tun
 - TouchScreen statt getrenntes Abhebungs- und EinzahlungsInterface

7.3 Dynamische Modellierung im Analyse-Workflow

Spezifikation aller Interaktionen der Analysetypen

Vom Analyse-Objektmodell zum Analyse-Verhaltensmodell

- Analyse-Verhaltensmodell
 1. Je ein Interaktionsdiagramm für den Ereignisfluss jedes wichtigen Use Case
 2. Je ein Zustandsdiagramm für Typen mit komplexen Verhalten
 3. Navigationspfade ← vereinfachte Zustandsdiagramme für das Benutzerinterface, die nur die Navigation durch verschiedene Boundaries modellieren
- Zweck
 - ◆ Identifikation von Methoden für das Objektmodell
 - ◆ Präzisierung von Methodenimplementierungen
 - ◆ Verfeinerung des Objektmodells
- Im Folgenden fokussieren wir uns auf Punkt 1 und 3
 - ◆ Verfeinerte Verhaltensmodellierung eines Use Case
 - ◆ Modellierung des Navigationsverhaltens von GUIs (mit Hilfe von Navigationspfaden)

Verfeinerte Verhaltensmodellierung eines Use Case

Ausgangspunkt

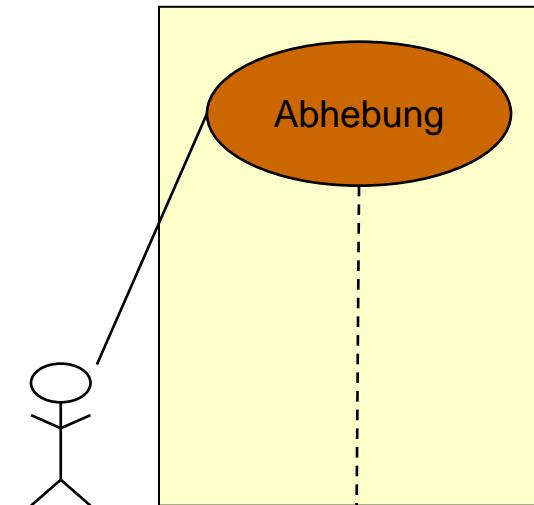
- (0) Vorhandenes Analyse-Objektmodell und dynamisches Modell aus Anforderungserhebung ← letzteres wird an das Analyse-Objektmodell angepasst

Verfahren (pro Use Case)

- (1) Konkretisiere Ereignisfluss
 - ◆ Bilde Schritte im Ereignisfluss auf Methoden oder Ereignisse („events“) ab
- (2) Ergänze Klassendiagramm
 - ◆ Erweitere Klassen um evtl. fehlende Methoden, Parameter und Attribute
- (3) Modelliere Ereignisfluss
 - ◆ Zusammenspiel von Objekten → Interaktionsdiagramm(e)
 - ◆ Verhalten einzelner Objekte → Zustandsdiagramm(e)
- (4) Verfeinere dynamisches Modell und Objektmodell
 - ◆ mehr Zwischenschritte (neue Methoden und Nachrichten)
 - ◆ mehr Strukturdetails (neue Attribute, Parameter, Typen)

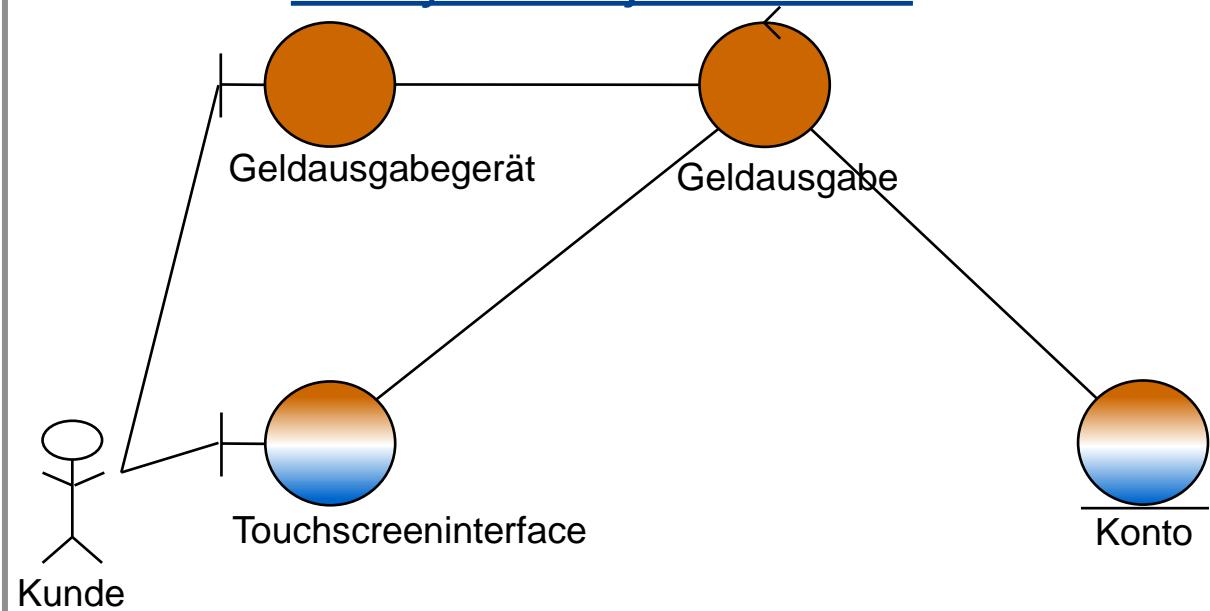
Use-Case Modell → Verhaltensmodell der Analyse

Use-Case Modell



Kunde

Analyse-Objektmodell

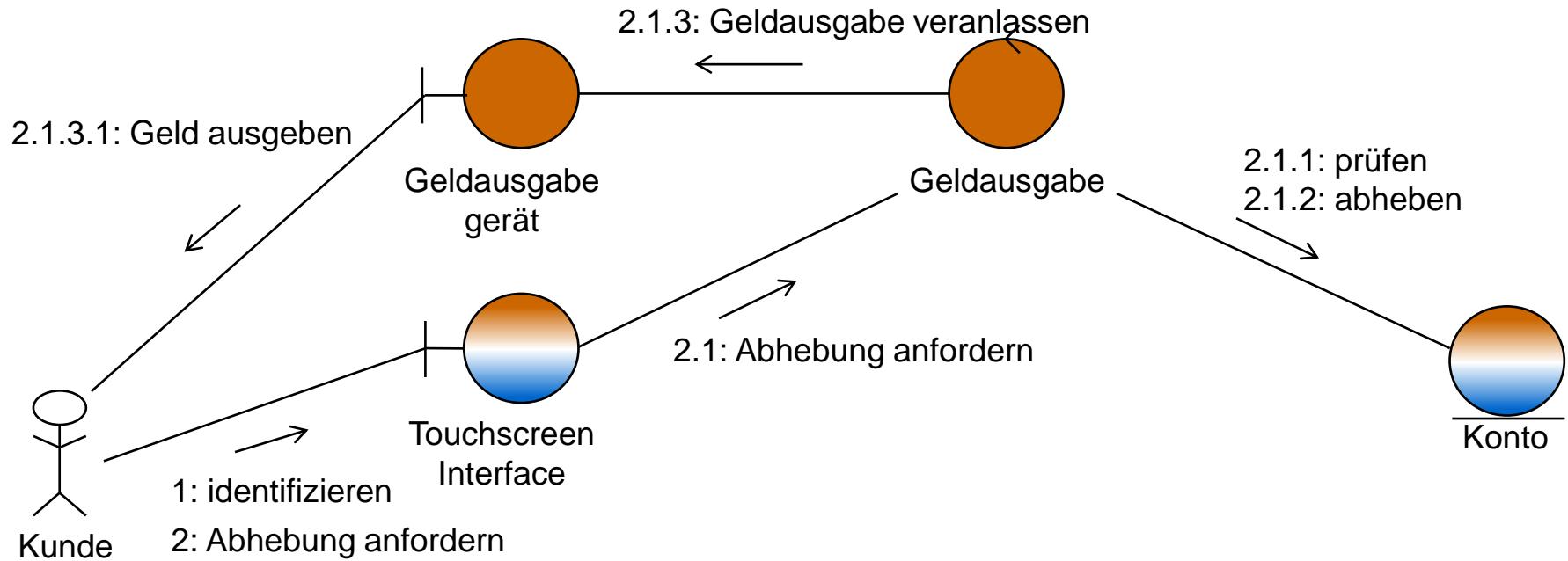


Kunde

Nächster Schritt: Ereignisfluss des Use Case „Abhebung“ auf Interaktionen der betroffenen Analyse-Objekte abbilden!
→ Kommunikationsdiagramm

Use-Case Modell → Verhaltensmodell der Analyse

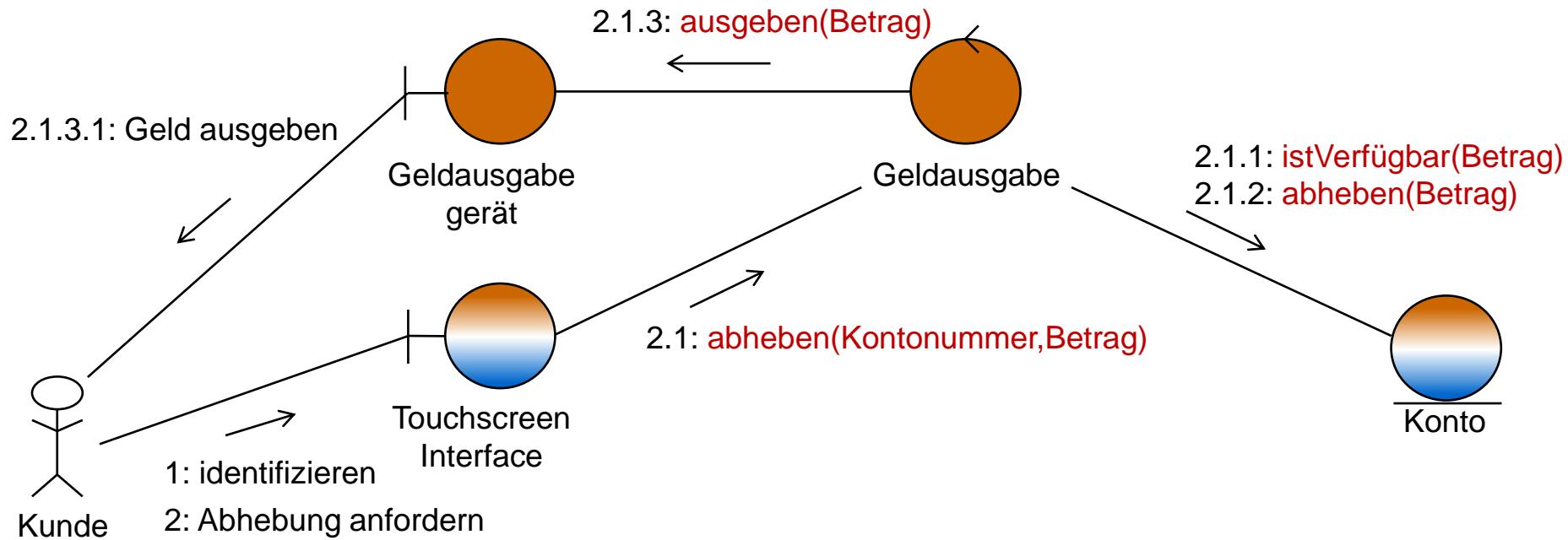
Ereignisfluss "Abhebung" auf Kommunikationsdiagramm abbilden



- Fortsetzung (1)
 - ◆ informelle Aktionen durch konkrete Nachrichten ersetzen

Use-Case Modell → Verhaltensmodell der Analyse

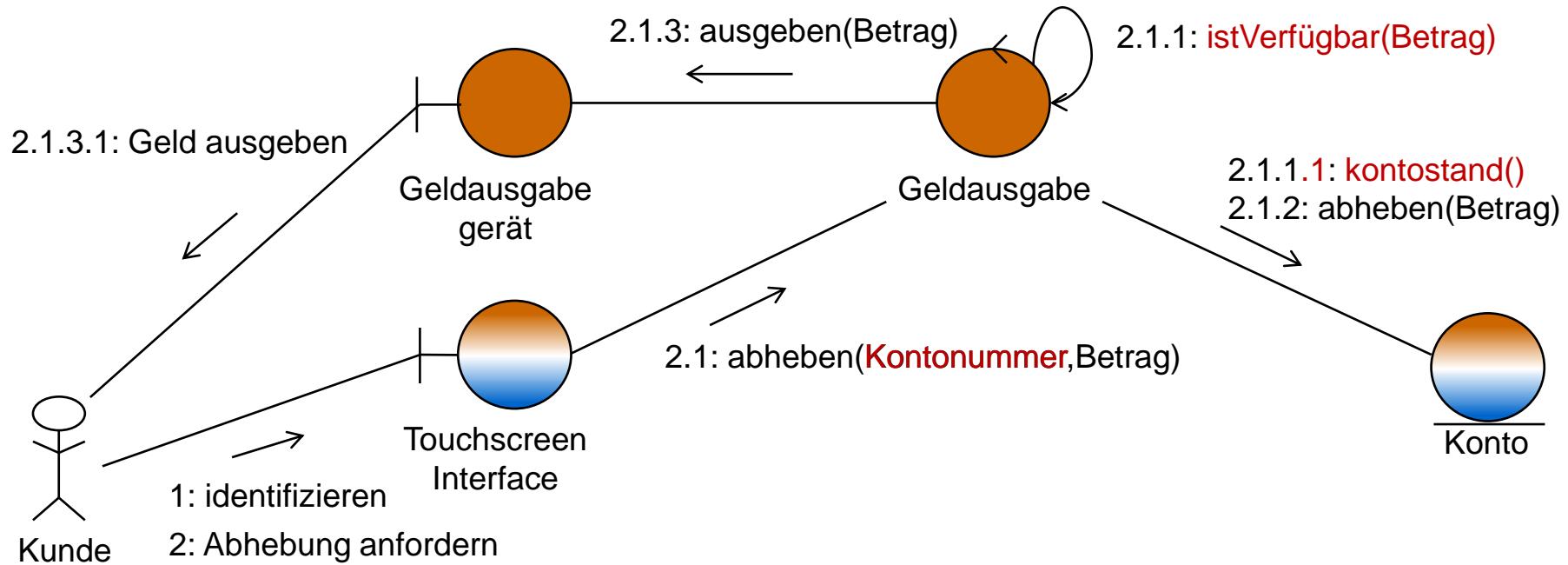
Ereignisfluss "Abhebung" auf Kommunikationsdiagramm abbilden



- Fortsetzung (1)
 - ◆ informelle Aktionen durch konkrete Nachrichten ersetzen

Use-Case Modell → Verhaltensmodell der Analyse

Ereignisfluss "Abhebung" auf Kommunikationsdiagramm abbilden

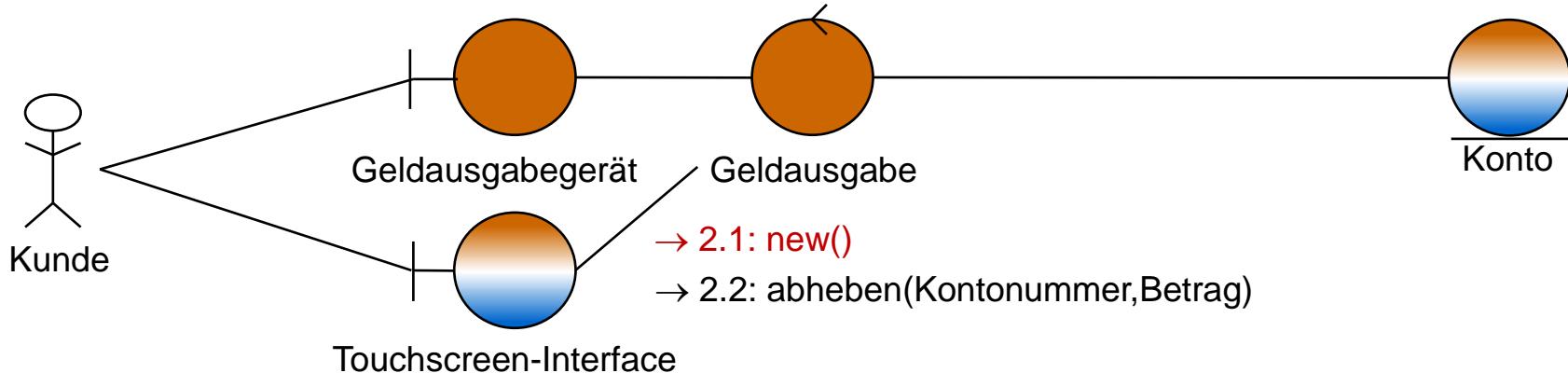


● Fortsetzung (2)

- ◆ Verantwortlichkeiten der Objekte überdenken → Zuordnung von Nachrichten anpassen (Bsp: `istVerfügbar()` in Controller statt in Entität)
- ◆ Methoden der Objekte ergänzen: Was braucht man für „`istVerfügbar()`“?

Analyse-Verhaltensmodell

► Objekt-Erzeugung

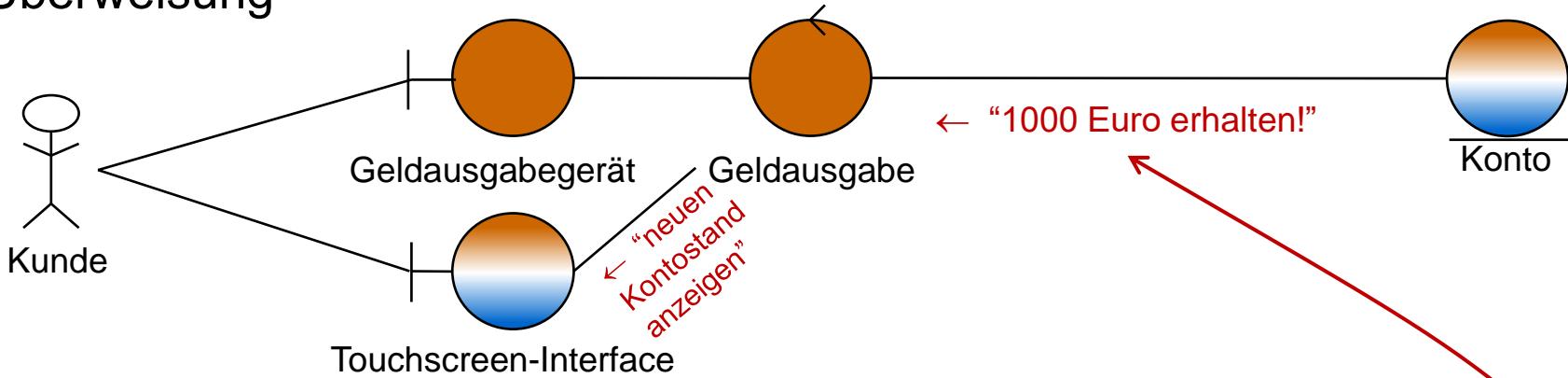


- **Controller-Objekte** werden bei der Initiierung des Use Case erschaffen
 - ◆ **A)** In einem Boundary das der Initiierung des Use Case dient
 - ◆ **B)** In Controller eines Use Case der die Quelle einer <<include>> oder das Ziel einer <<extend>>-Beziehung zum eigenen Use Case ist
- **Boundary-Objekte** werden von Controller-Objekten erschaffen
 - ◆ Im obigen Beispiel ausnahmsweise nicht! → Denksport: Warum???
- **Entity-Objekte** existieren schon (meist persistent) oder werden von Controller-Objekten erschaffen

Analyse-Verhaltensmodell

► Objekt-Zugriff

Szenario: „Sofortige Kontostandsaktualisierung bei eintreffender Überweisung“



● Objekt-Zugriff

- ◆ Trotz der Einschränkungen auf **Typebene** (s. Analyse-Objektmodell „**Verbogene Abhängigkeiten**“) dürfen Entity-, Boundary- und Controller-**Objekte** beliebig miteinander interagieren!
- ◆ Um solche Objektinteraktionen zu modellieren, ohne auf Typebene „**schlechte Abhängigkeiten**“ einzuführen nutzt man typischerweise das Entwurfsmuster „**Observer**“ (→ Siehe Kapitel „**OOM**“ und „**Entwurfsmuster**“)

Analyse-Verhaltensmodell-Verfeinerung

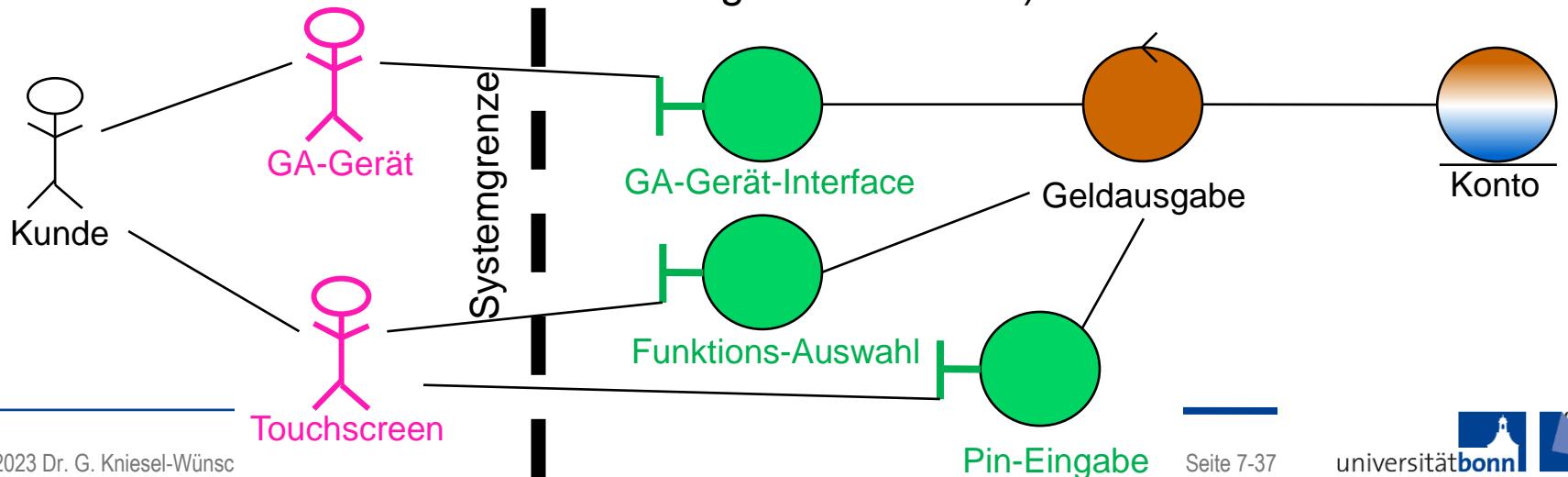
► Akteure versus Boundaries (1)

Akteure: Alles Äußere, womit das System das wir modellieren direkt oder indirekt interagiert.

- Beispiele: Auch die **physischen Geldausgabe-Geräte und Touchscreens** (die wir als Softwareentwickler in einer Bank nicht selber bauen!)

Boundaries: Die Teile unseres Systems, die direkt mit der Außenwelt (= den Akteuren) kommunizieren.

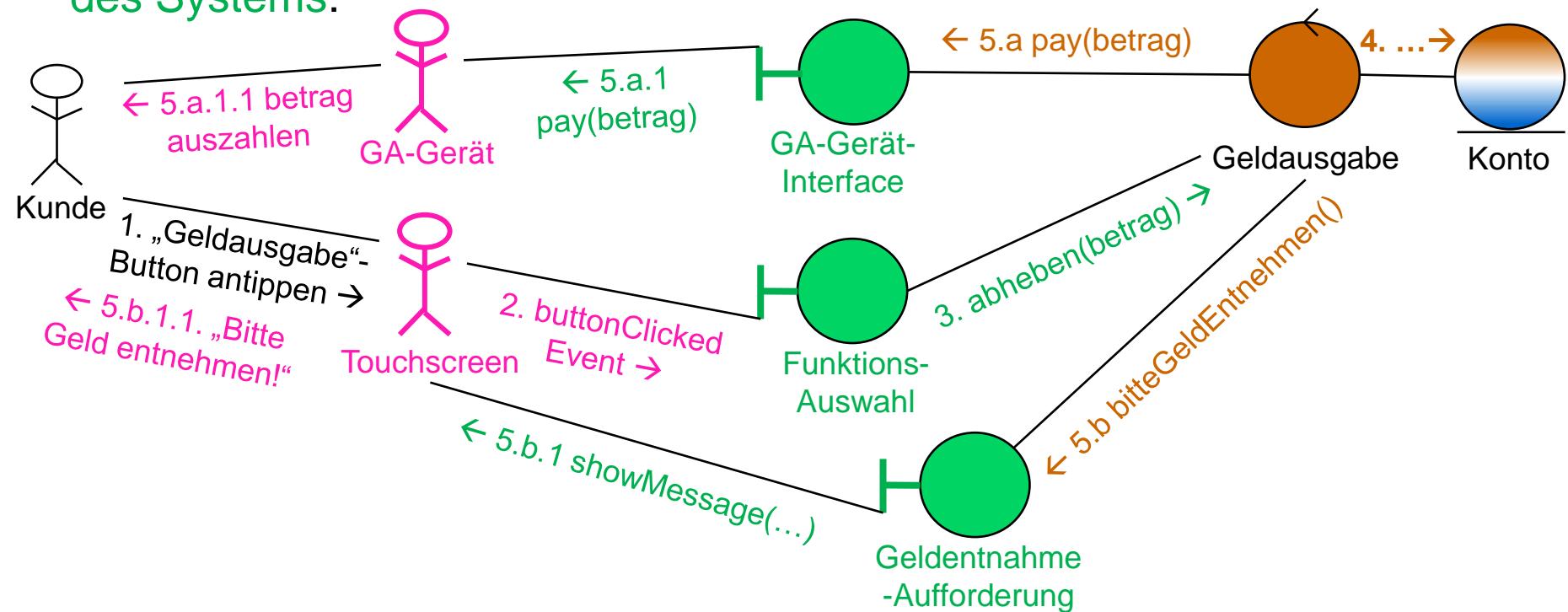
- Beispiele: Unsere **Software-Schnittstelle zu Geldausgabe-Geräten und die GUI** die wir auf den Touchscreens anzeigen lassen (beides müssen wir als Entwickler der Bankanwendung selber bauen!)



Analyse-Verhaltensmodell

► Akteure versus Boundaries (2)

Die Trennung der beiden Konzepte macht Ihr Modell klarer. Sie ermöglicht zu unterscheiden zwischen Aktionen des Kunden, anderer Akteure und des Systems:



Diese Unterscheidung ermöglicht es Abläufe präziser / feiner granular zu spezifizieren (z.B. `buttonClickedEvent`, `pay(betrag)`).

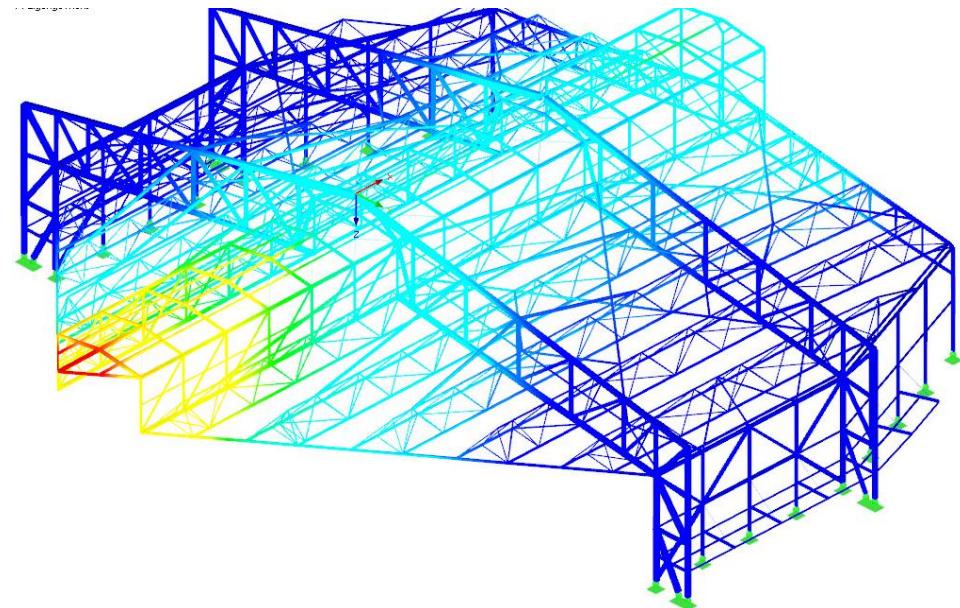
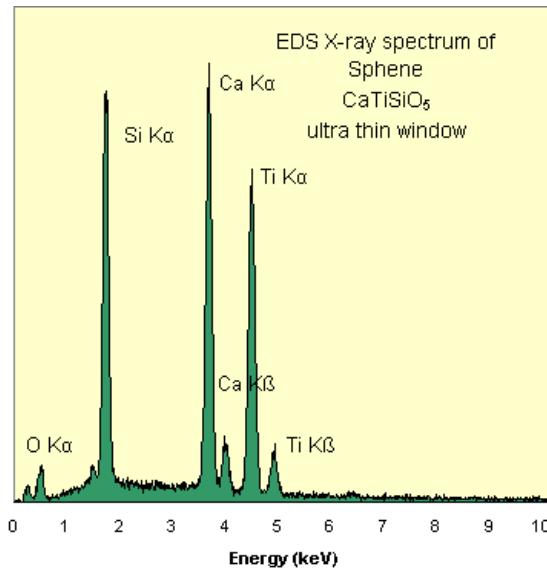
7.4. Entwurf der grafischen Benutzeroberfläche (GUI)

Layout
Interaktionsdesign

Modellierung von Benutzerinterfaces im Analyse-Workflow

Anforderungserhebung: Fokus auf Anwendungsdomäne und Benutzersicht

- Erfassung domänenspezifischer Visualisierungen



- ... aber keine Benutzerschnittstellen (die sind Teil der Realisierung)!

Modellierung von Benutzerinterfaces im Analyse-Workflow

Anforderungsanalyse: Fokus auf Realisierung / Realisierbarkeit

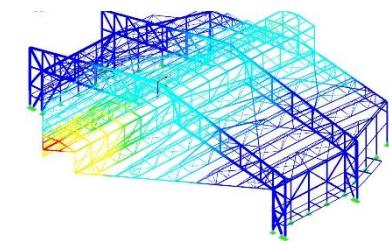
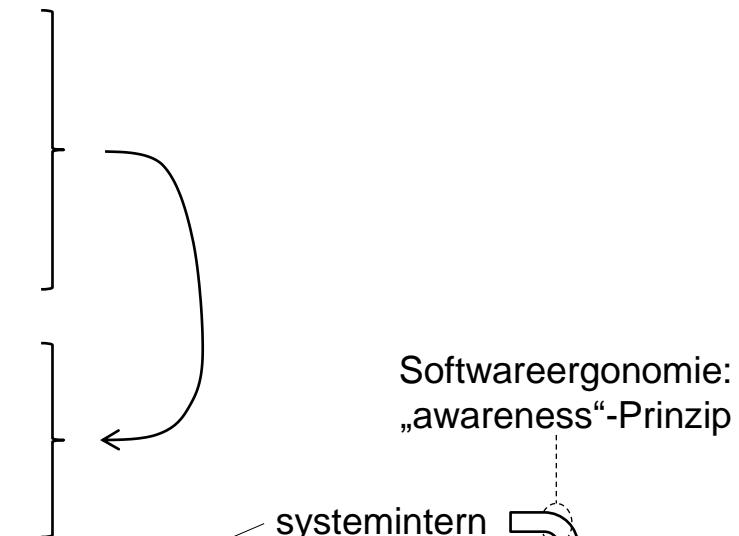
- Spezifikation von Abläufen bei Benutzung
- Erfassung vertrauter HCI/CHI-Konzepte

- ◆ HCI = Human Computer Interaction
- ◆ CHI = Computer Human Interaction

- Spezifikation von Benutzerinterfaces

- ◆ Layout ← Struktur der GUI
- ◆ Interaktionsdesign ← Verhalten der GUI

- ⇒ Event bzw. Aktion auf GUI \wedge Bedingung \Rightarrow Effekt
- ⇒ „Sichtbare Effekte“
 - Navigationsverhalten
 - Öffnen / Ausblenden / Schließen von GUI-Elementen
 - Sonstige Anwendungslogik
 - Zustandsänderung von sichtbaren Elementen



Beispiel: GUI-Layout mit Mockup-Tool

→ <https://balsamiq.com/wireframes/>

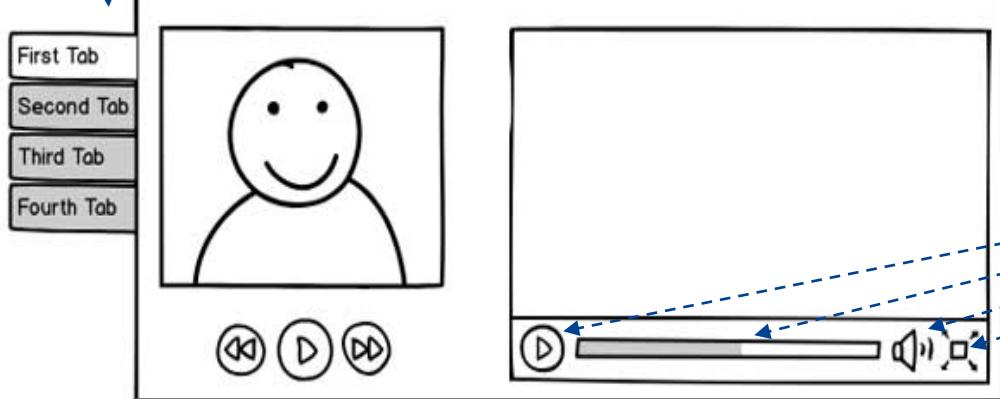
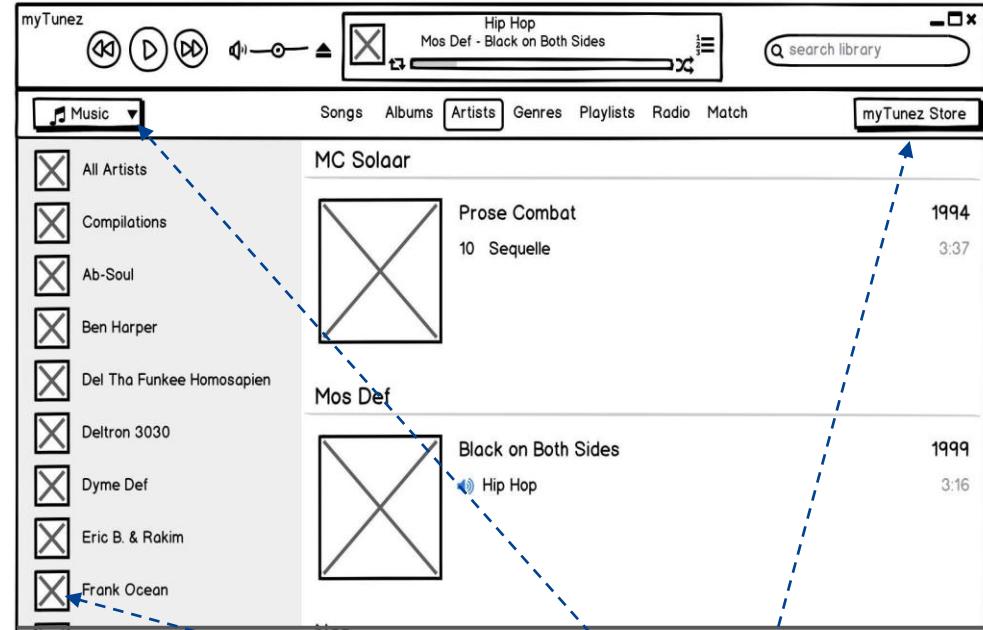
„Wireframes“

erfassen das grobe Layout eines Boundaries:

- Elemente
- Anordnung
- Beschriftung

Keine Details

- genaue Form
- Farben, ...



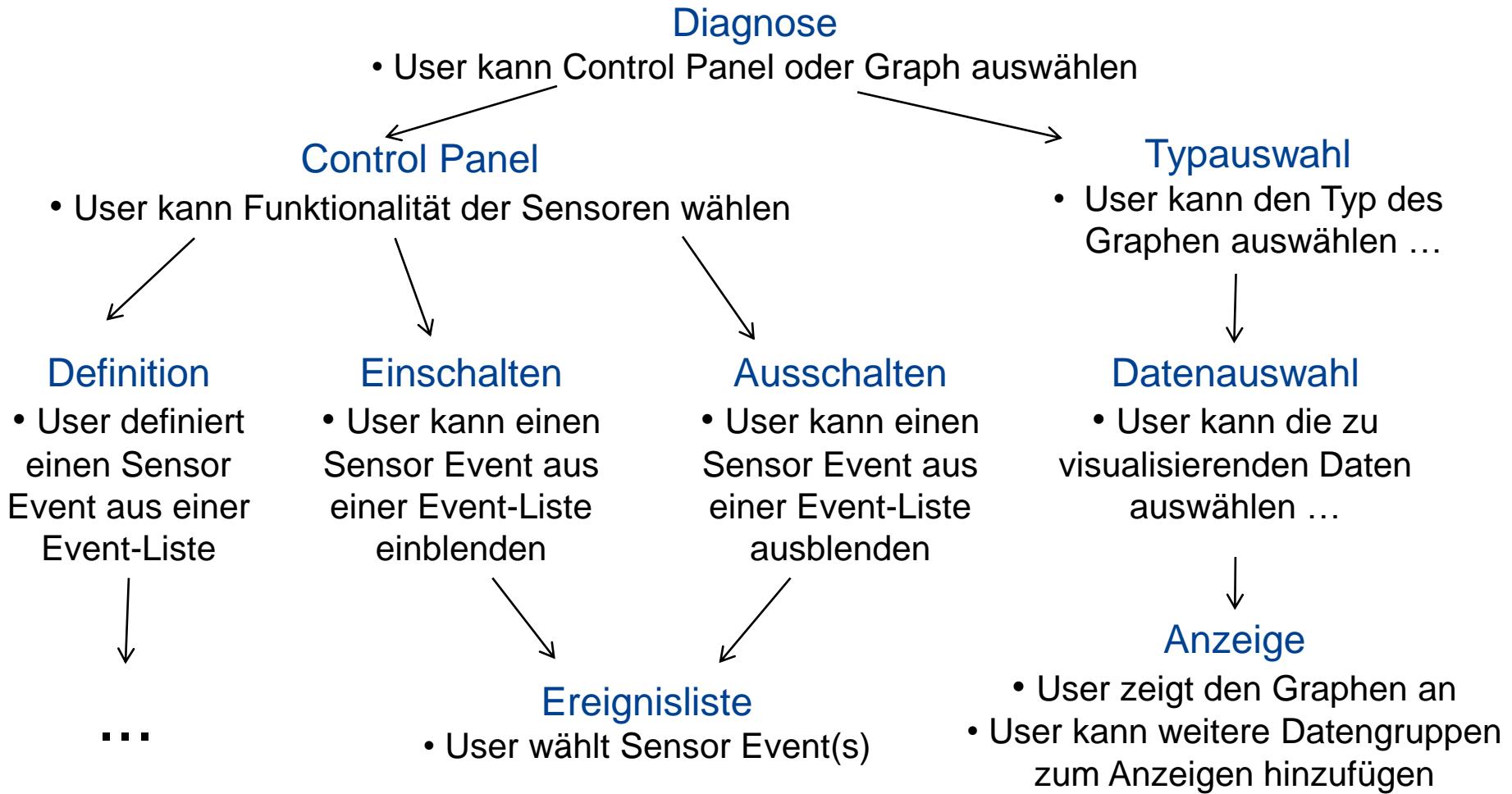
Hinzu kommt das Interaktionsdesign:
„Was passiert, wenn ich hier klicke?“

- Navigationsverhalten
- Sonstige Anwendungslogik

Spezifikation von Navigationsverhalten durch „Navigationspfade“ („Klickpfade“)

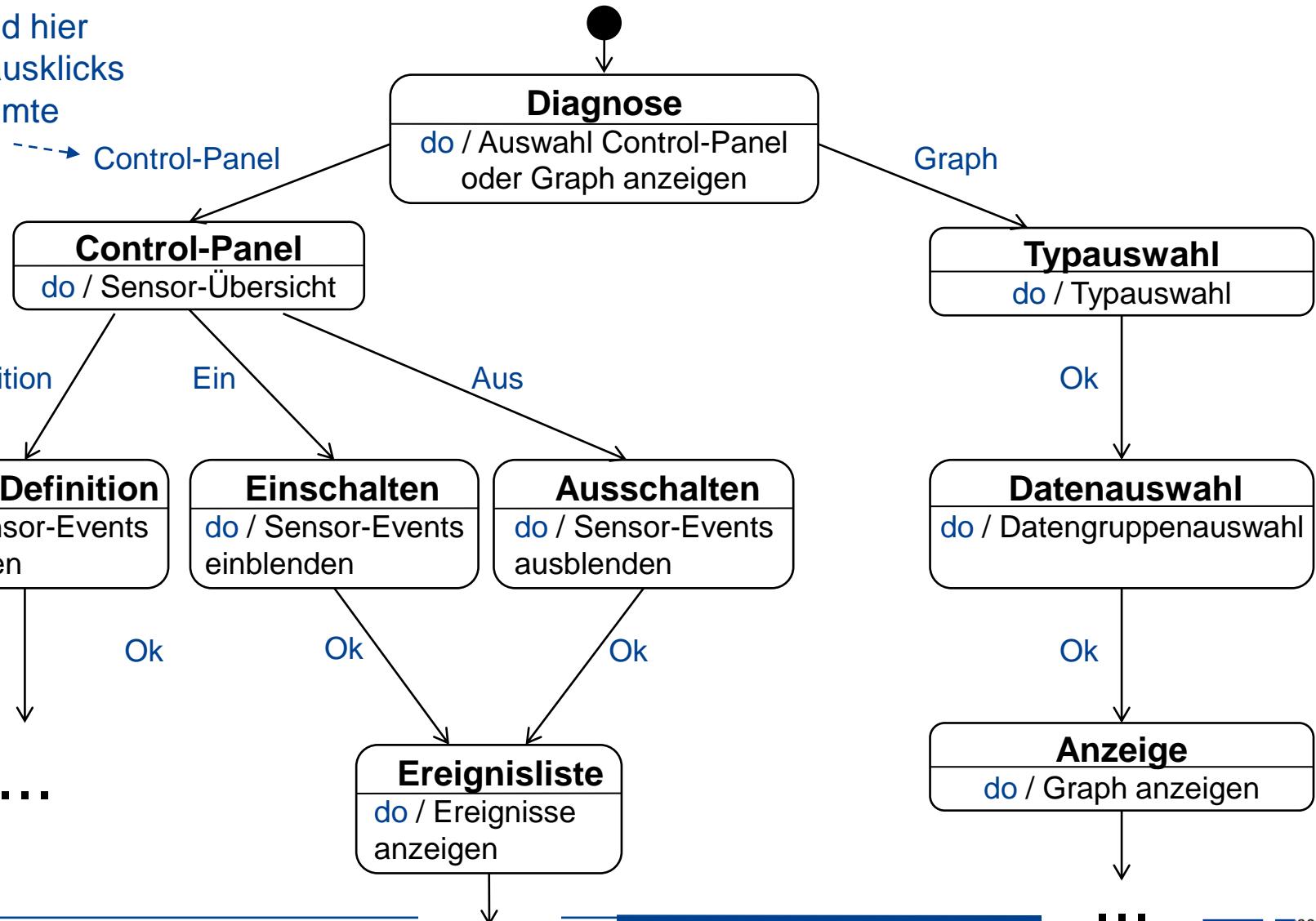
- Navigationspfad = Vereinfachtes Zustandsdiagramm, das nur das Navigationsverhalten einer GUI spezifiziert
- Zustand => Benannte Bildschirmansicht („Screen“)
 - ◆ (Navigations-relevante) Aktivitäten/Aktionen sind unter dem Screen-Namen aufgeführt
- Zustandsübergang => Übergang („Navigation“) zu anderem Screen, die getriggert wurde durch eine Aktion des Akteurs
 - ◆ Klick
 - ◆ Menüauswahl
 - ◆ Cursorbewegung
 - ◆ Sprache
 - ◆ Geste
 - ◆ ...

Navigationspfade als Graph



Navigationspfade als Zustandsdiagramm

Events sind hier
jeweils Mausklicks
auf bestimmte
Elemente





Termine

21.11.2014

23.11.2014

2 Gäste

Art der Unterkunft



Ganze
Unterkunft



Privatzimmer



Gemeinschaftszimmer

Preisspanne

1000+€

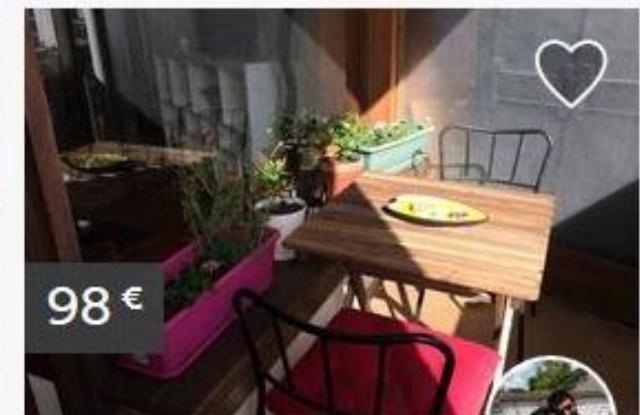
Weitere Filter

1000+ Unterkünfte - Paris

Übung: Malen Sie einige
Navigationspfade für dieses Beispiel
(Airbnb-Buchungsvorgang)



A 2 PAS DES CHAMPS...
Ganze Unterkunft - 34 Bewertungen · ...



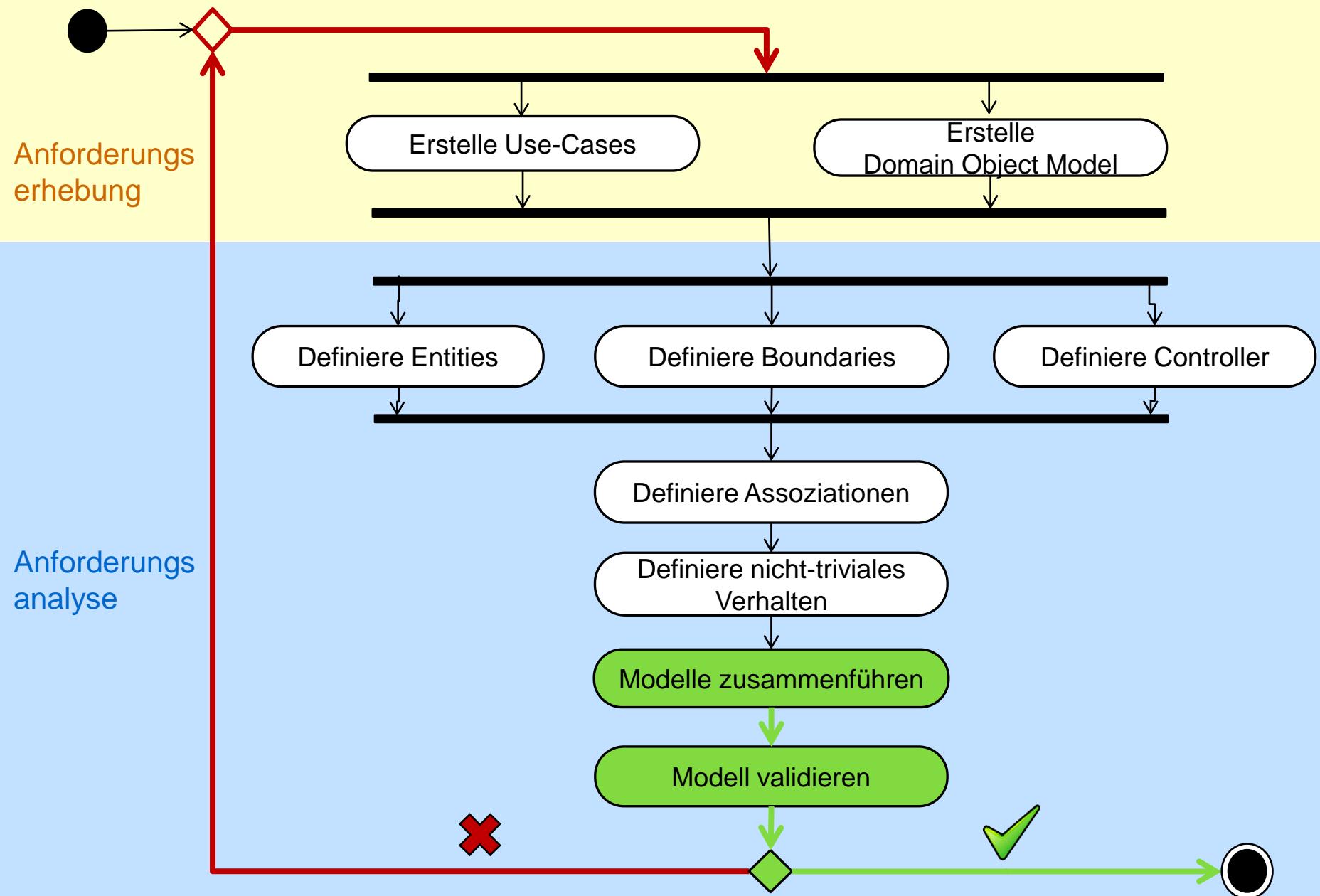
A deux pas de la Tour E...
Ganze Unterkunft - 23 Bewertungen · T...

User-Centered Design und Mockups

- Ziel = ergonomische Software (“Benutzerfreundlichkeit”)
 - ◆ Software die von den Benutzern nicht als hilfreich, leicht zu erlernen, intuitiv, ... empfunden wird scheitert
- Methode = “User-Centered Design”
 - ◆ Akzeptanztesting anhand “Mockups” schon in der Analysephase
- Mockup
 - ◆ Grobes Layout aller Boundaries („Wireframes“)
plus
 - ◆ “ausführbare” Spezifikation der Navigation
plus
 - ◆ “ausführbare” Spezifikation sonstiger Effekte bei Aktionen auf der GUI
- “Ausführbar” = manuell oder mit Hilfe von speziellen Werkzeugen

7.5 Konsolidierung der Analyse

Anforderungs-Erhebung und -Analyse



Kriterien der Anforderungsvalidierung

- Korrektheit
 - ◆ Die Anforderungen repräsentieren die Sicht des Kunden.
- Vollständigkeit
 - ◆ Alle im System möglichen Szenarien sind beschrieben, inklusive Ausnahmeverhalten von System und Benutzer
- Konsistenz
 - ◆ Es gibt keine funktionalen oder nichtfunktionalen Anforderungen die sich widersprechen
- Eindeutigkeit
 - ◆ Es gibt keine Zweideutigkeiten bei den Anforderungen.
- Realismus
 - ◆ Anforderungen können implementiert und ausgeliefert werden
- Zurückverfolgbarkeit
 - ◆ Jede Funktion des Systems kann auf einen Satz entsprechender funktionaler Anforderungen zurückgeführt werden

Vollständigkeit, Konsistenz, Eindeutigkeit des Modells → Technische Sicht

- Vollständigkeit
 - ◆ Alle referenzierten Typen sind definiert
- Konsistenz
 - ◆ Identifikation von doppelt definierten Klassen
 - ◆ Identifikation von vertauschten Rollen zwischen Klassen
 - ⇒ Rolle am falschen Ende der Assoziation
 - ◆ Benennung von Klassen, Attributen, Methoden
 - ⇒ Keine Synonyme, d.h. keine verschiedene Namen für gleiche Bedeutung, z.B. Bank und Kreditinstitut
- Eindeutigkeiten
 - ◆ Eindeutige Benennung von Klassen, Attributen, Methoden
 - ⇒ Keine Homonyme, d.h. keine gleichen Namen für unterschiedliche Bedeutungen, z.B. Bank (= Kreditinstitut) und Bank (Sitzgelegenheit)
 - ◆ Keine Tippfehler in Namen

Formatvorlage Anforderungsanalyse-Dokument

- 1. Einführung
- 2. Momentanes System
- 3. Beabsichtigtes System
 - 3.1 Überblick
 - 3.2 Funktionale Anforderungen
 - 3.3 Nichtfunktionale Anforderungen
 - 3.4 Nebenbedingungen (“Pseudoanforderungen”)
 - 3.5 Systemmodelle
- 4. Glossar



Exkurs „Analyseformatvorlage“

Projektvereinbarung

- Die Projektvereinbarung steht für die Akzeptanz des Analysemodells (dokumentiert durch das Anforderungsanalyse-Dokument) durch den Kunden.
- Kunde und Entwickler einigen sich auf das funktionale und nichtfunktionale Verhalten des Systems, sowie die Einschränkungen hinsichtlich der Umsetzungstechnologien.
Zusätzlich einigen sie sich auf:
 - ◆ Eine Liste der Prioritäten
 - ◆ Einen Revisionsprozess
 - ◆ Eine Liste von Kriterien zur Annahme des Systems
 - ◆ Einen Zeitplan und ein Budget

Anforderungsevolution

- Anforderungen ändern sich rapide während der Anforderungserhebung
- Tools zur Verwaltung von Anforderungen
 - ◆ Speichern Anforderungen in einem Repository
 - ◆ Erstellen automatisch Anforderungsdokumente aus dem Repository
 - ◆ Unterstützen Zurückverfolgbarkeit und Änderungsmanagement für den ganzen Lebenszyklus des Projektes
 - ◆ Unterstützen Multi-user Zugriff
- Beispiele
 - ◆ Requisit Pro (Rational / IBM)
 - ⇒ <http://www.ibm.com/developerworks/rational/products/requisitepro/>
 - ◆ Jira
 - ⇒ <http://www.jira.com>

Zusammenfassung: Anforderungsanalyse

Aus dem Use Case Modell und Domain Object Modell entsteht in der Analyse

- Das statischen Analysemodell
 - ◆ Boundaries
 - ◆ Controllers
 - ◆ Entities
- Das dynamisches Analysemodell
 - ◆ Verhalten der Analysetypen (Boundaries, Controller, Entities)
- Das Modell der GUI
 - ◆ Layout der Boundaries
 - ◆ Verhalten der GUI-Elemente

Abschließend erfolgt die Konsolidierung der Analyse

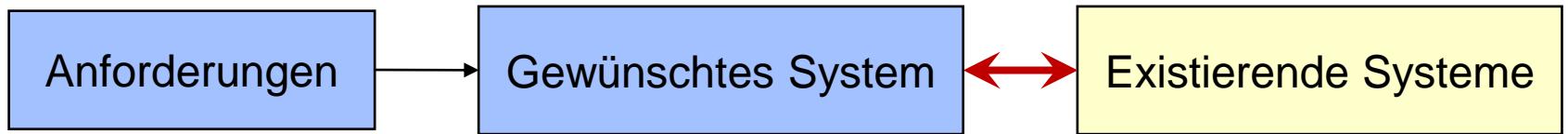
- Integration der Modelle zu allen Use-Cases
- Prüfung von Konsistenz, Vollständigkeit, Eindeutigkeit, ...

Kapitel 8 Systementwurf

Stand: 8.1.2024

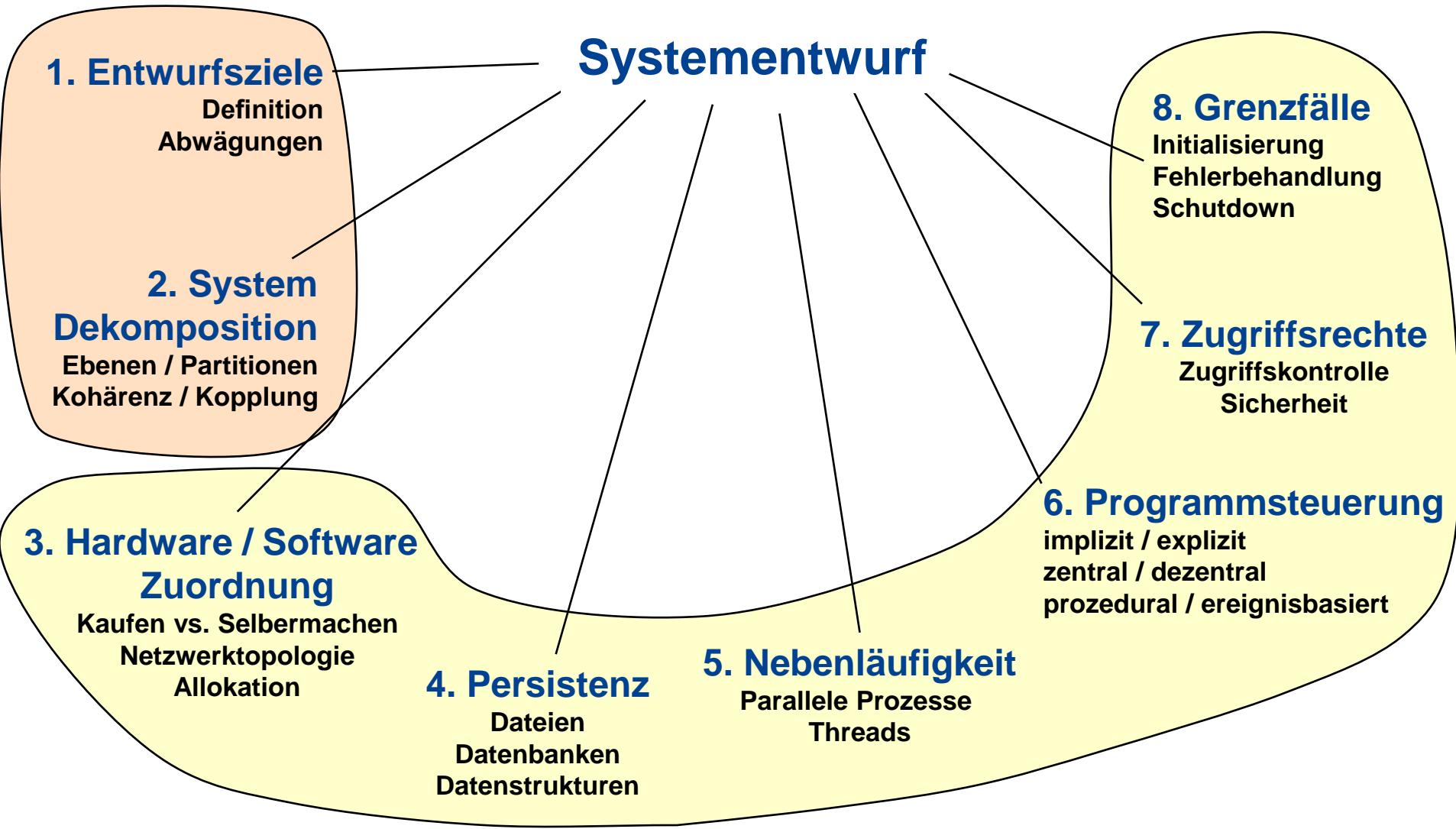
Systementwurf

- **Ziel:** Überbrücken der Lücke zwischen gewünschtem und existierendem System auf handhabbare Weise

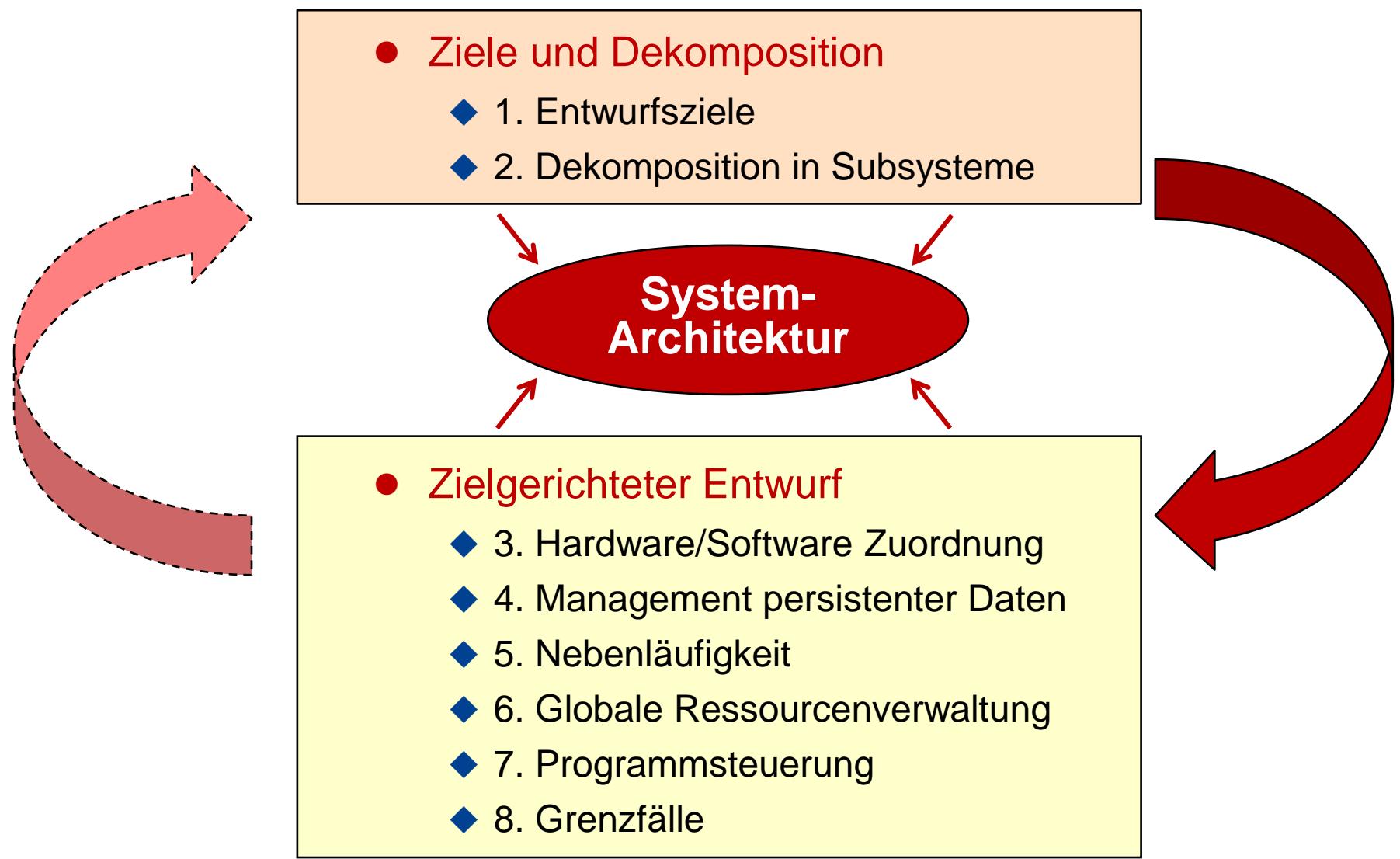


- **Idee:** Anwendung des “Divide and Conquer”-Prinzips
 - ◆ Modellierung des neuen Systems als Menge von Subsystemen
- **Folgeproblem:** „Crosscutting concerns“ – Übergeordnete Belange die viele Subsysteme betreffen (z.B. Persistenz, Nebenläufigkeit, ...)
 - ◆ Erst wenn diese geklärt sind, kann man die Subsysteme unabhängig voneinander bearbeiten
- **Weg:** Zielbestimmung → Dekomposition → Klärung der übergeordneten Belangen ⇒ „Architektur“
 - ◆ Danach erst Detailentwurf der Subsysteme

Systementwurf



Kapitel-Überblick



8.1 Ziele

(→ Brügge & Dutoit, Kap. 6)

Nichtfunktionaler Anforderungen sind Entscheidungshilfen

- **Dilemma** ▶ Zu viele Entwurfsalternativen
 - ◆ Die gleiche Funktionalität ist auf verschiedenste Arten realisierbar
- **Nutzen von NFA** ▶ Auswahlkriterien
 - ◆ Nichtfunktionale Anforderungen (NFA) dienen als Auswahlkriterien
 - ◆ Sie fokussieren die Entwurfsaktivitäten auf die relevanten Alternativen
- **Beispiele** Nichtfunktionale Anforderung → Lösungsmöglichkeiten:
 - ◆ „Hoher Durchsatz“ → Parallelität, optimistische Vorgehensweise, ...
 - ◆ „Zuverlässigkeit“ → Redundanz, einfache GUIs, ...
 - ◆ „dauerhaft“ → Persistenz durch Datenbank, Datei, ...
 - ◆ „sicher“ → Zugriffsrechte, Verschlüsselung, ...

Nutzung der Ergebnisse der Anforderungsanalyse für den Systementwurf

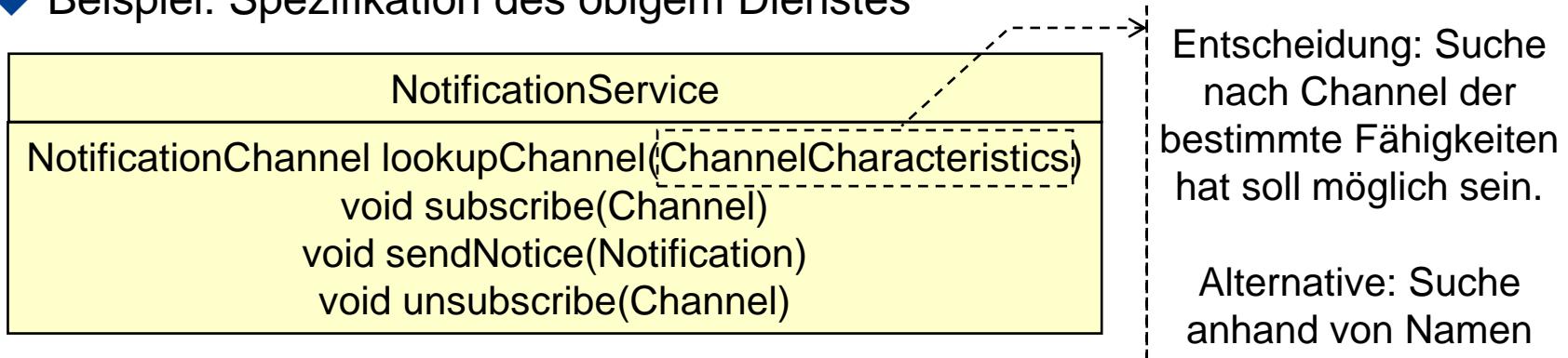
- Nichtfunktionale Anforderungen →
 - ◆ Aktivität 1: Definition der Entwurfsziele
- Statisches Analyse-Modell (Objektmodell mit Stereotypen) →
 - ◆ Aktivität 2: Systemdekomposition (Auswahl von Subsystemen nach funktionalen Anforderungen, Kohärenz und Kopplung)
 - ◆ Aktivität 3: Hardware/Software Zuordnung
 - ◆ Aktivität 4: Persistentes Datenmanagement
- Dynamisches Analyse-Modell →
 - ◆ Aktivität 5: Nebenläufigkeit
 - ◆ Aktivität 6: Globale Ressourcenverwaltung
 - ◆ Aktivität 7: Programmsteuerung
 - ◆ Aktivität 8: Grenzfälle

8.2. Subsystem-Dekomposition

(→ Brügge & Dutoit, Kap. 6)

Dienstidentifikation
Subsystemaufteilung
Kopplung und Kohärenz als Hilfskriterium

- Dienst: Menge von Operationen mit gemeinsamem Zweck
 - ◆ Beispiel: Benachrichtigungsdienst
 - ⇒ lookupChannel(), subscribe(), sendNotice(), unsubscribe()
 - ◆ Dienste werden während des Systementwurfs identifiziert und spezifiziert
- Dienstspezifikation: Vollständig typisierte Menge von Operationen
 - ◆ In UML und Java würde das einem 'Interface' entsprechen
 - ◆ Beispiel: Spezifikation des obigen Dienstes



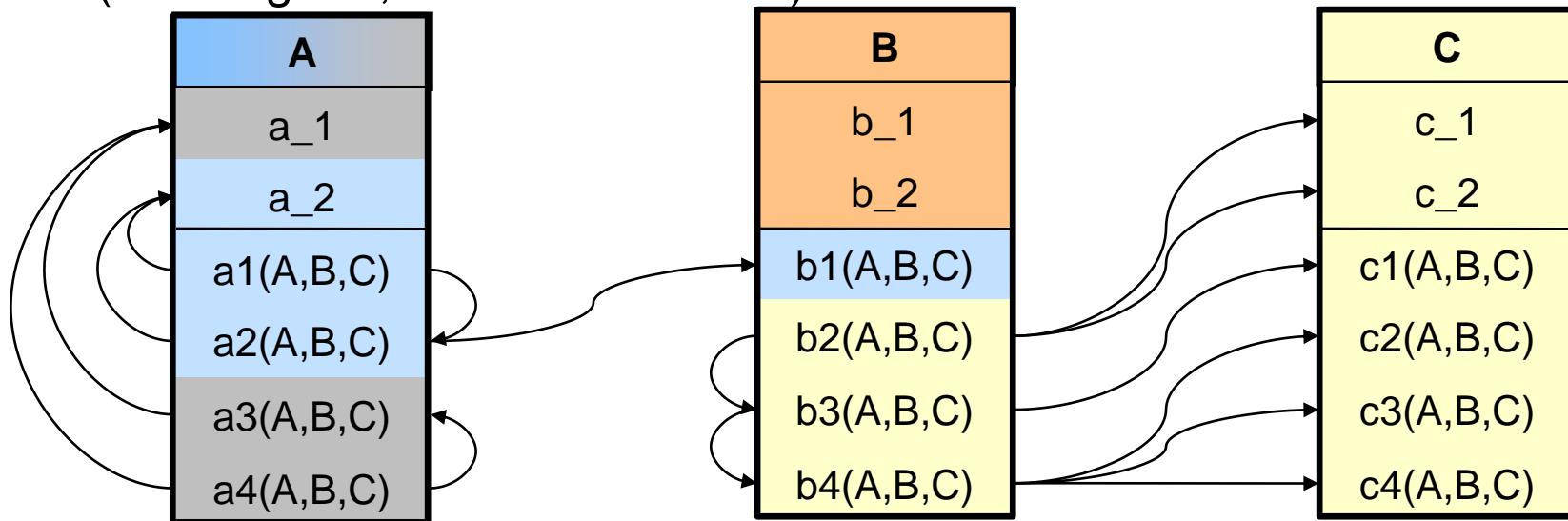
- ◆ Verwendete Schnittstellen (Channel, Notifiation, ...) müssen natürlich auch spezifiziert werden

Subsysteme und Subsystem-Aufteilung

- Subsystem = Stark kohärente Menge von Klassen die einen oder mehrere Dienste realisieren
- **Kohärenz** = Maß der Abhängigkeiten innerhalb der Kapselungsgrenzen (hier: innerhalb eines Subsystems)
- **Kopplung** = Maß der Abhängigkeiten zwischen den Kapselungsgrenzen (hier: zwischen den Subsystemen)
 - ◆ **Starke Kopplung** → Modifikation eines Subsystems hat gravierende Auswirkungen auf die anderen (Wechsel des Modells, breite Neukompilierung, ...)
- Ziel: Wartbarkeit des Systems
 - ◆ Die meisten Abhängigkeiten sollten innerhalb einzelner Subsysteme bestehen, nicht über die Subsystemgrenzen hinweg.
 - ◆ Aufteilung in Subsysteme sollte zu **maximaler Kohärenz** und **minimaler Kopplung** führen

Beispiel: Kopplung und Kohärenz

- Gegeben folgende drei Klassen. Die Pfeile zeigen Abhängigkeiten (Feldzugriffe, Methodenaufrufe):



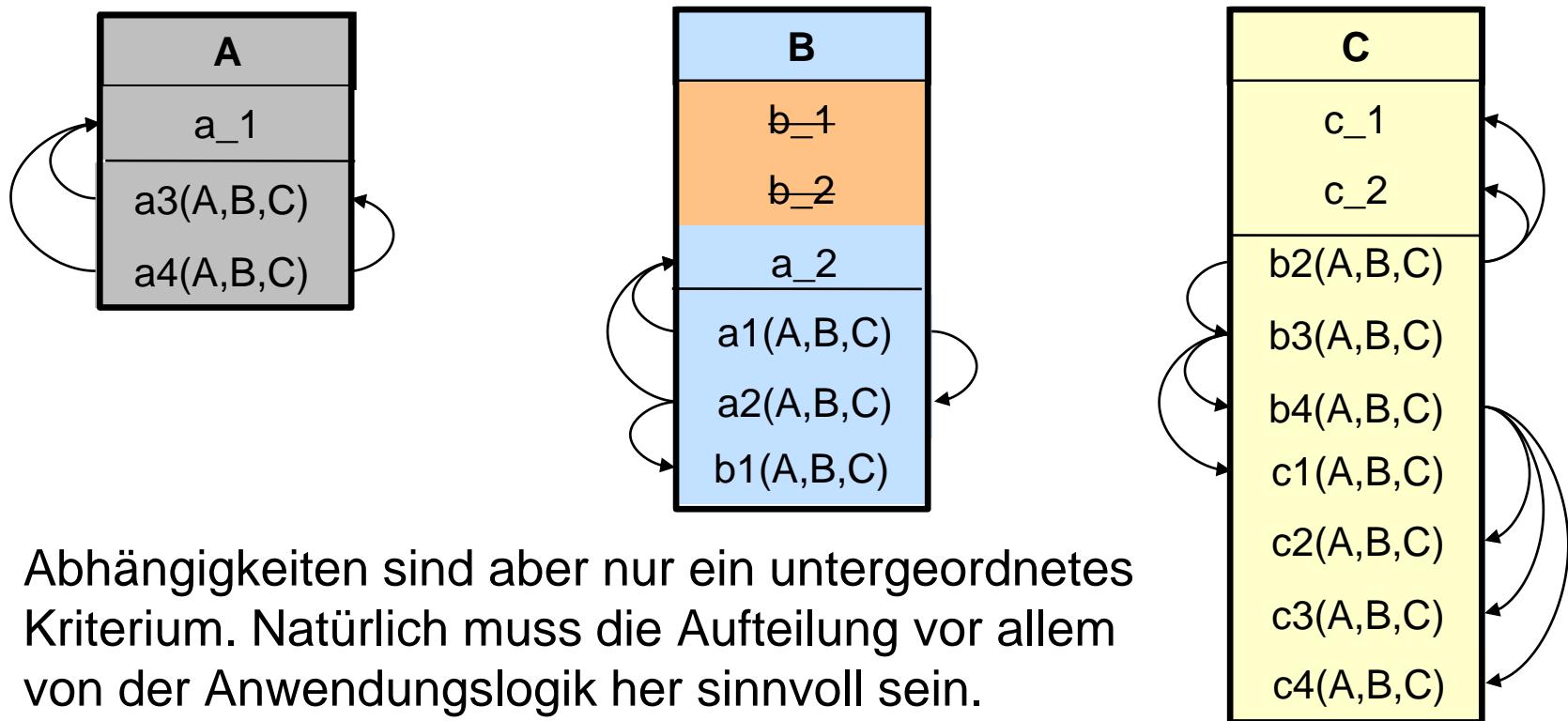
Klasse mit 2
unabhängigen
Kohäsionseinheiten
→ aufsplitten in 2 Klassen!

Kaum Kohäsion.
→ b_1, b_2 ungenutzt
→ b1 gehört nach A!
→ b2 - b4 gehören nach C!

Völlig unkohäsive
Klasse.
B kümmert sich mehr
um C-Elemente als C
selbst!

Beispiel: Kopplung und Kohärenz

- Das wäre aus Abhängigkeitssicht sinnvoller (kohärent + kopplungsfrei):



- Abhängigkeiten sind aber nur ein untergeordnetes Kriterium. Natürlich muss die Aufteilung vor allem von der Anwendungslogik her sinnvoll sein.
- Abhängigkeiten bieten sich also vor allem als Hilfskriterium an, wenn es von der Anwendungslogik her verschiedene Zuteilungen geben kann.

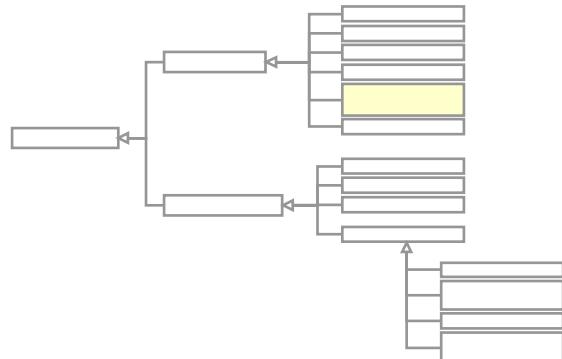
Von Subsystemen zu Komponenten

- Subsysteme sind also Haufen von Klassen die bestimmte Dienste implementieren. OK. Aber was heißt das?
- Sind Subsysteme also eine Art Packages?
- Was sieht ein Subsystem von einem anderen Subsystem?
- Kann es auch auf Dinge zugreifen, die nicht als Dienst spezifiziert sind?
- Wie leicht ist es Subsysteme zu kombinieren?

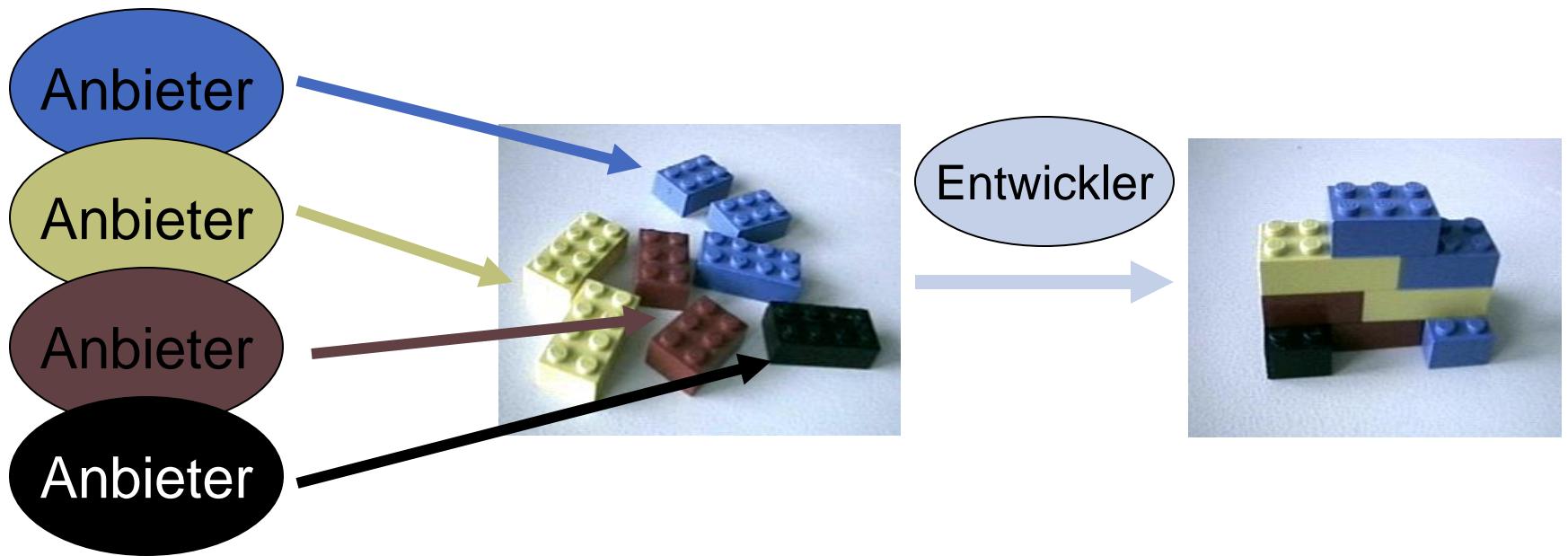
➤ Komponenten!



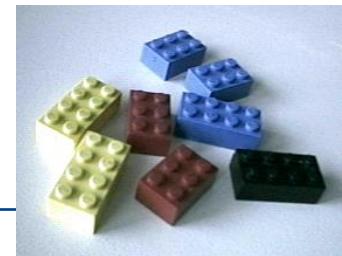
8.3 Komponenten und Komponentendiagramme (UML)



Intuitive Vorstellung



Ziele von Komponenten



- Plattformunabhängige Wiederverwendung von Komponenten
 - günstigere,
 - bessere und
 - schnellere Softwareentwicklung.
- Unterstützung für flexibel anpassbare Geschäftsprozesse
 - ◆ Einfach existierende Dinge zu einem neuen Verbund zusammensetzen
- Fokus auf intelligente Anwendung anstatt der wiederholten (Neu-) Entwicklung des gleichen Basisfunktionalitäten
- Märkte für Komponenten
 - ◆ Möglichkeit Komponenten von Drittanbietern zu kaufen
 - ◆ Möglichkeit Komponenten an andere zu verkaufen

Komponenten-Definition

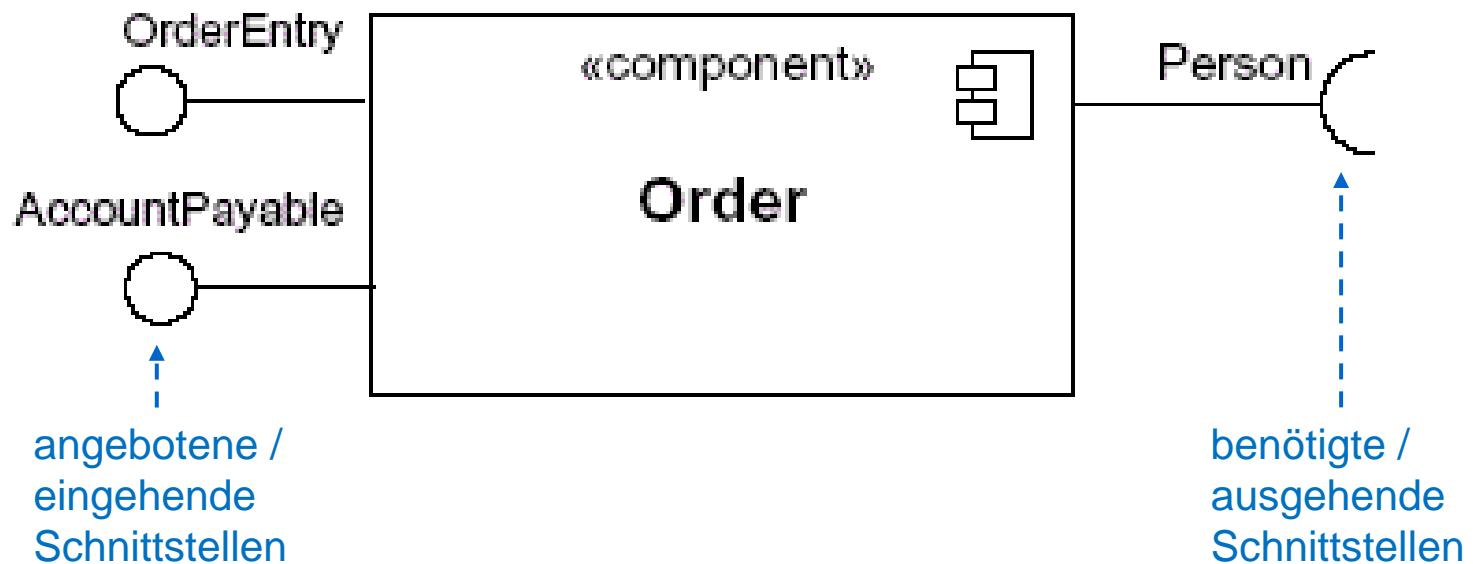


- Clemens Szyperski , WCOP 1996
 - ◆ „Eine Softwarekomponente kann **unabhängig eingesetzt** werden und wird **von Dritten zusammengesetzt**.“
 - ◆ „Eine Softwarekomponente ist eine Kompositionseinheit mit **vertraglich spezifizierten Schnittstellen** die **alle Kontextabhängigkeiten explizit machen**“
- Literatur
 - ◆ Workshop on Component-Based Programming (WCOP) 1996
 - ◆ Clemens Szyperski:
„Component Software – Beyond Object-Oriented Programming“, Addison Wesley Longman, 1998.
 - ◆ Clemens Szyperski, Dominik Gruntz, Stephan Murer:
„Component Software – Beyond Object-Oriented Programming“, Second Edition, Pearson Education, 2002.

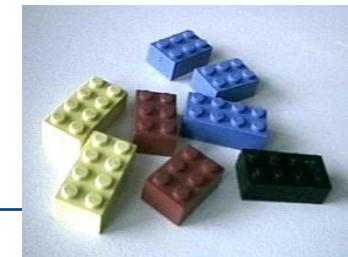
Komponenten



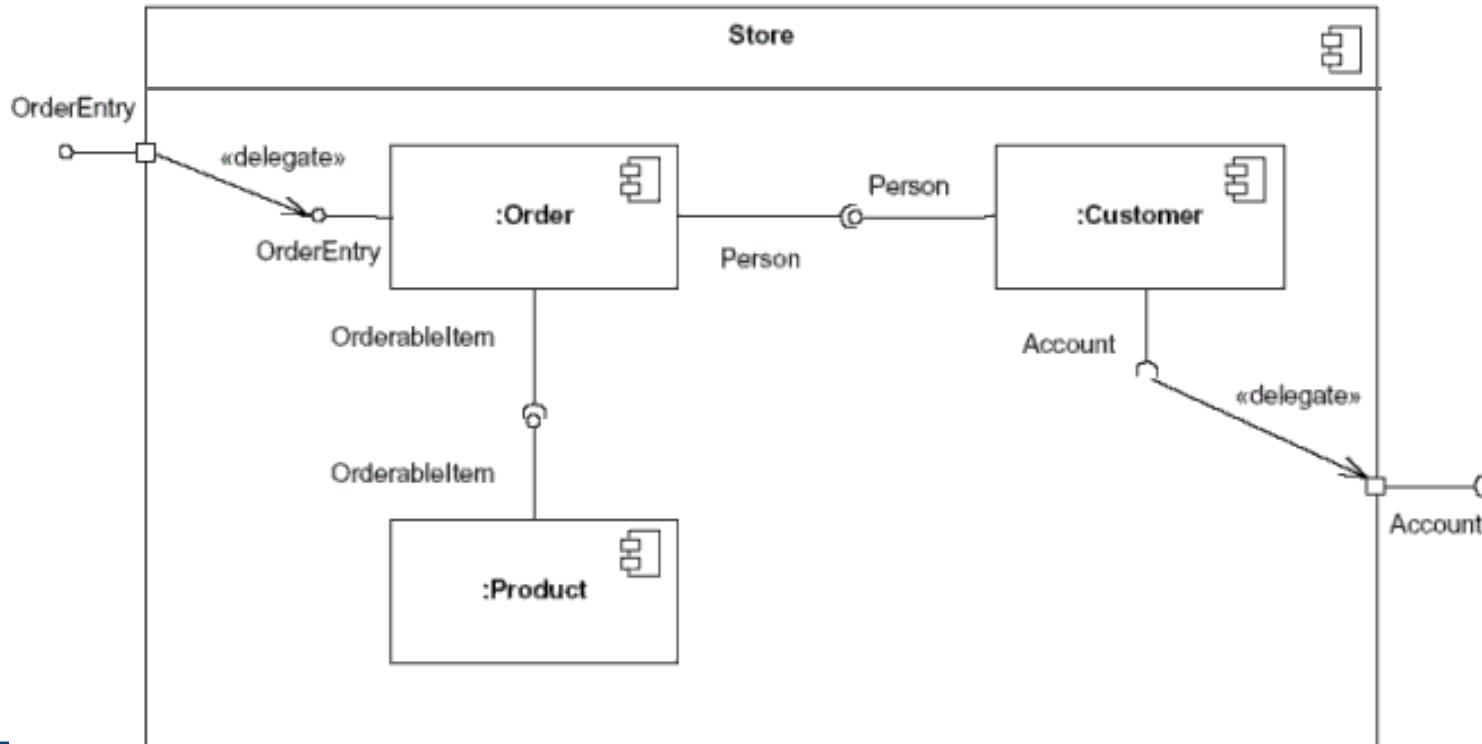
- Kernidee → Nur explizit spezifizierte Kontextabhängigkeiten
 - ◆ Daher nicht nur die „angebotenen Schnittstellen“ beschreiben, sondern auch die „benutzten Schnittstellen“!
- Beispiel
 - ◆ Die Komponente „Bestellung“ (Order) braucht einen „Person“-Dienst um die eigenen Dienste anbieten zu können.



Komponenten: Beispiel



- Komposition
 - ◆ Die „Order“-Komponente nutzt den „Person“-Dienst von „Customer“
- Hierarchische Komponenten
 - ◆ Die „Store“-Komponente besteht ihrerseits aus drei Unterkomponenten
 - ◆ Die intern unverbundenen Schnittstellen werden nach außen durchgereicht



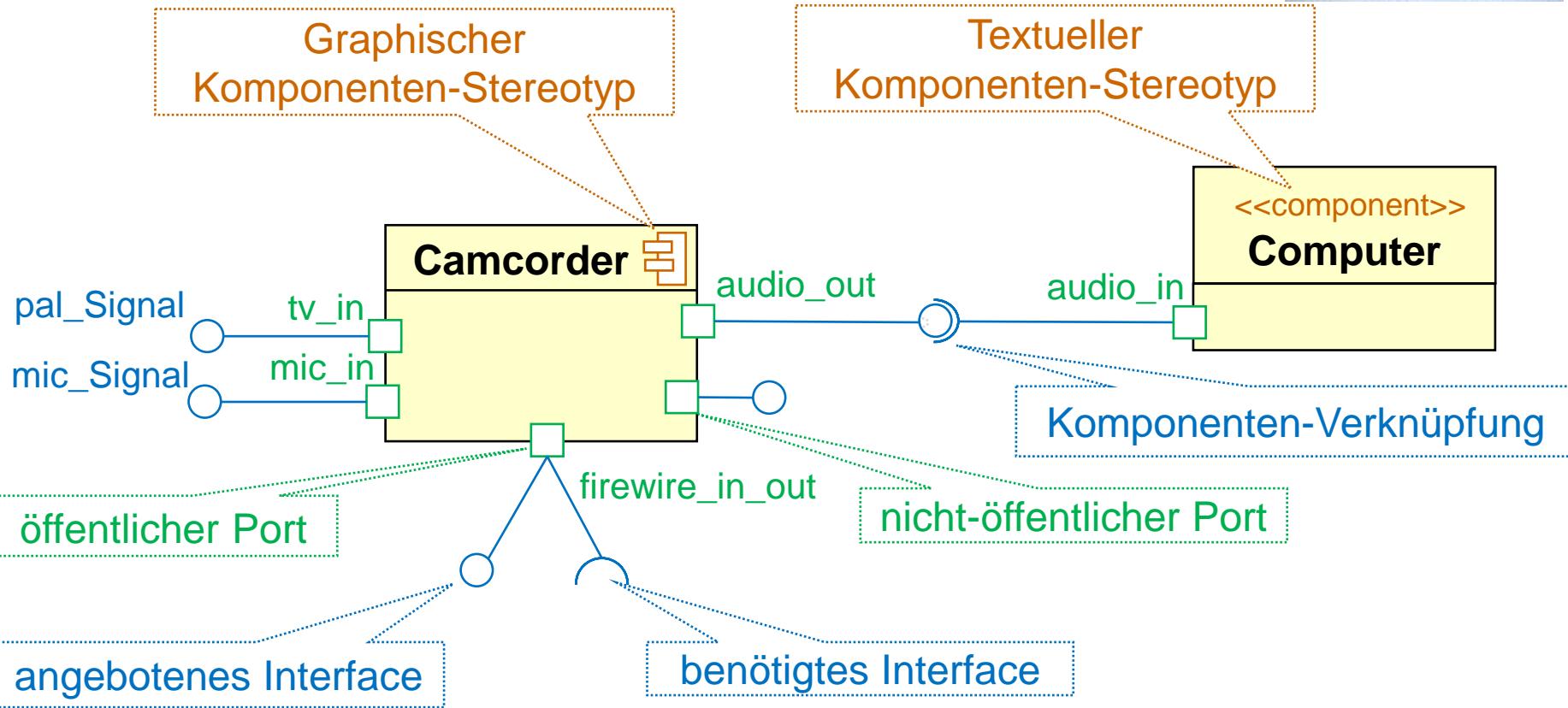
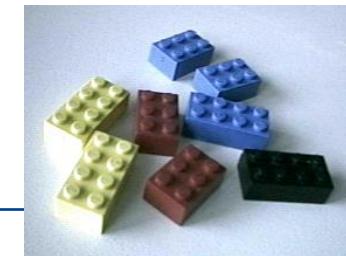
Komponentendiagramme (ab UML 2.0)



Komponentendiagramm zeigt Komponenten und deren Abhängigkeiten

- Komponenten kapseln beliebig komplexe Teilstrukturen
 - ◆ Klassen, Objekte, Beziehungen oder ganze Verbünde von Teilkomponenten (→ hierarchische Komposition)
 - ◆ Quellcode, Laufzeitbibliotheken, ausführbare Dateien, ...
- Komponenten haben wohldefinierten Schnittstellen
 - ◆ Angebotene Schnittstellen (‘provided interfaces’)
 - ◆ Benötigte/benutzte Schnittstellen (‘required interfaces’)
- Komponenten bieten ‘Ports’
 - ◆ Ein Port ist ein Name für eine Menge zusammengehöriger Schnittstellen
 - ◆ Verschiedene Ports (Namen) für mehrfach vorhandene gleiche Schnittstelle (z.B. mehrere USB-Schnittstellen am gleichen Gerät)

Komponentendiagramm-Elemente an einem Beispiel



Erweiterte Interaktionsspezifikation durch „Behaviour Protocoll“

- Gegeben

```
DB_Interface
open(DB_descr) : Connection
close(Connection)
query(Connection,SQL) : ResultSet
getNext(ResultSet) : Result
```

- Problem

- ◆ Wir wissen nicht, wie das beabsichtigte Zusammenspiel der einzelnen Methoden ist.
- ◆ Kann man die Methoden in jeder beliebigen Reihenfolge aufrufen?

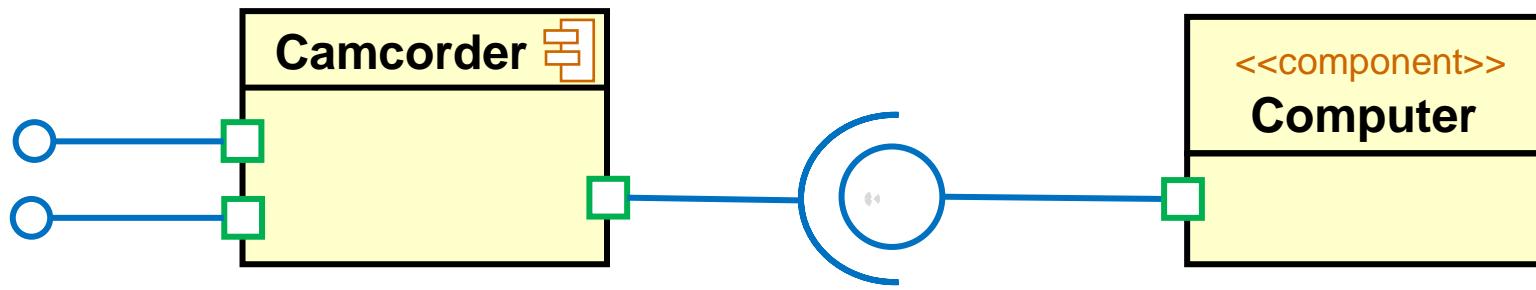
- Lösung

- ◆ Zu jeder Schnittstelle wird ihr „Verhaltensprotokoll“ mit angegeben
- ◆ Es ist ein regulärer Ausdruck der legale Aufrufsequenzen und Wiederholungen spezifiziert

- Beispiel

- ◆ „**Erst** Verbindung zur Datenbank erstellen, **dann** beliebig oft anfragen und in jedem Anfrageergebnis beliebig oft Teilergebnisse abfragen, **dann** Verbindung wieder schließen.“

```
protocoll DB_Interface_Use =
  open(DB_descr) ,
  ( query(Connection,SQL) : ResultSet ,
    ( getNext(ResultSet) : Result )*
  )* ,
  close(Connection)
```

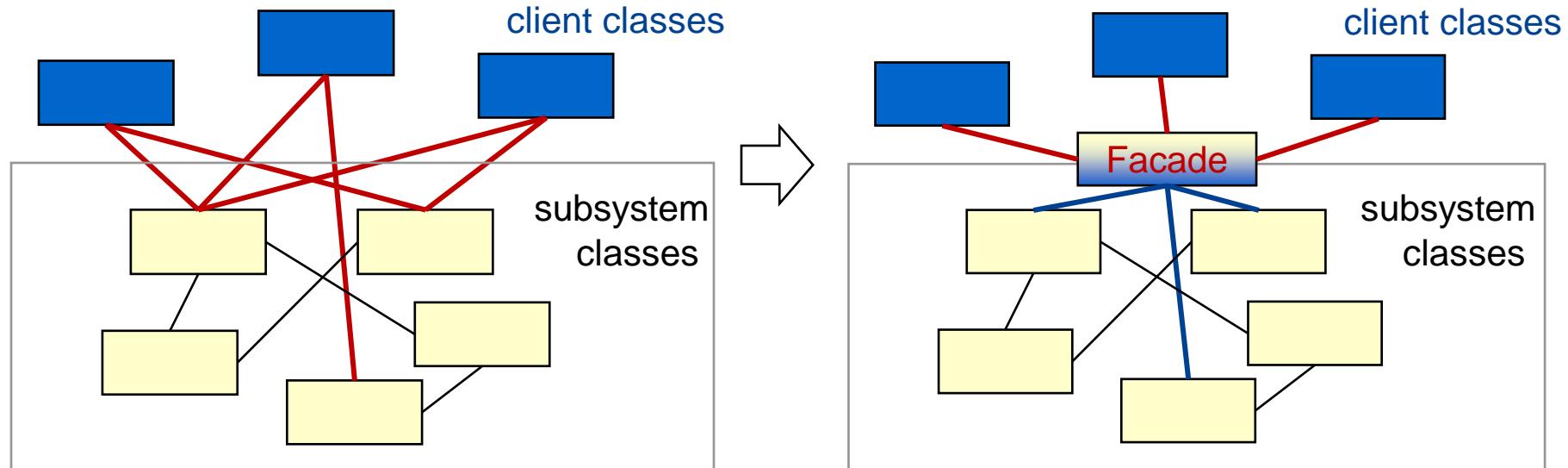


8.4 Implementierung von Subsystemen / Komponenten

Im Systemdesign hilfreiche Patterns

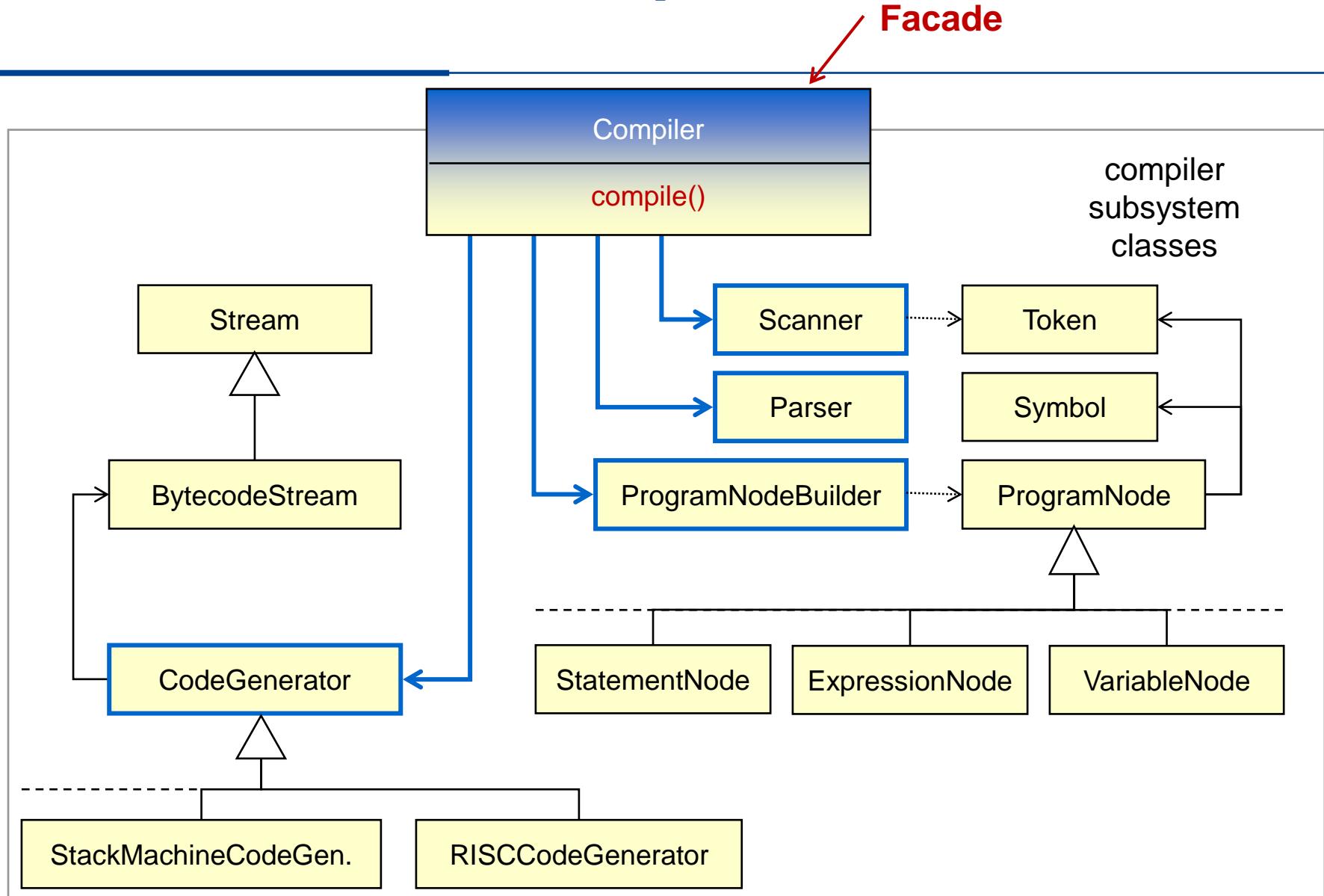
Subsystem-Implementierung mit „Facade Pattern“

- Absicht
 - ◆ Abhängigkeiten der Clients von der Struktur eines Subsystems reduzieren



- Idee: Façade = “Dienst”-Objekt
 - ◆ Funktionen einer Menge von Klassen eines Subsystems zu einem Dienst zusammenfassen
 - ◆ Objekt, das den Dienst eines Subsystems nach außen darstellt
 - ◆ Bietet alle Methoden des Dienstes
 - ◆ Vorteil: Clients müssen nichts über die Interna des Subsystems wissen

Facade Pattern: Beispiel



Facade Pattern: Anwendbarkeit

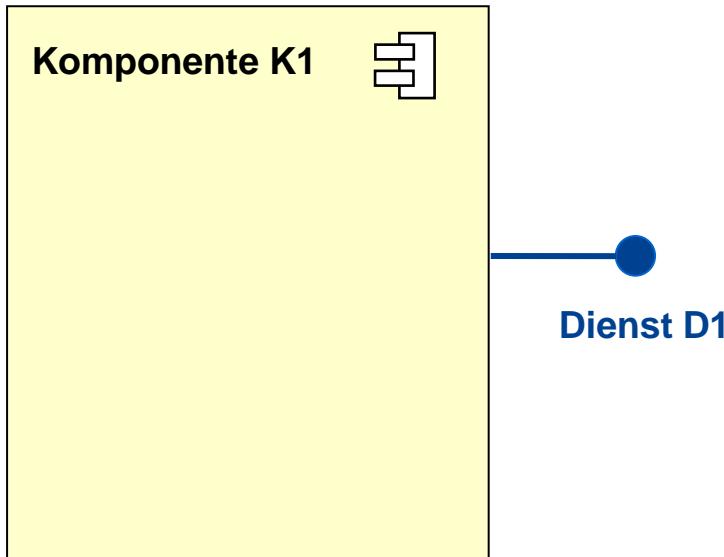
- Viele Abhängigkeiten zwischen Klassen
 - ◆ Reduzieren durch Facade-Objekte
- Einfaches Interface zu einem komplexen Subsystem
 - ◆ Einfache Dinge einfach realisierbar (aus Client-Sicht)
 - ◆ Anspruchsvolle Clients dürfen auch "hinter die Facade schauen"
 - ⇒ zB für seltene, komplexe Anpassungen des Standardverhaltens
- Hierarchische Strukturierung eines Systems
 - ◆ Eine Facade als Einstiegspunkt in jede Ebene

Façade als Realisierung eines Dienstes

Black-Box Sicht

Komponente bietet der Außenwelt einen Dienst

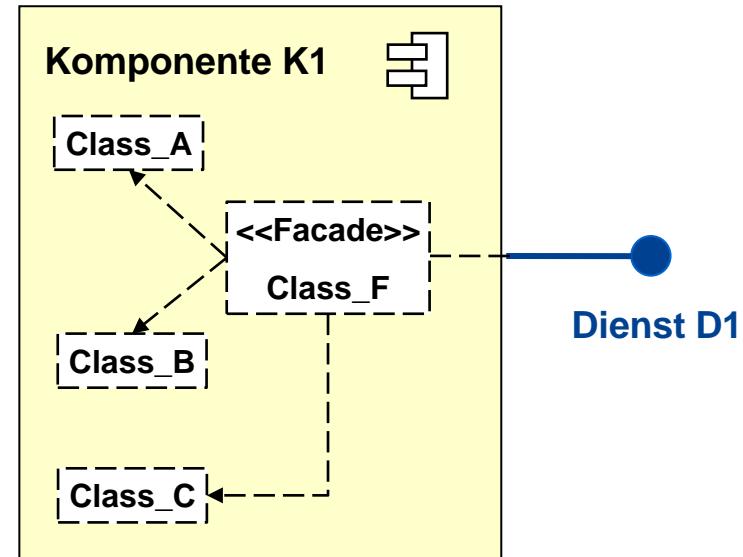
- K1 bietet D1
- Wie das geschieht ist egal



Interne Sicht

Dienst wird intern durch eine Façade implementiert

- Class_F implementiert D1 und agiert als Façade



Patterns für Subsysteme

- Facade
 - ◆ Subsystem abschirmen (gerade vorgeführt)
- Singleton
 - ◆ Nur eine einzige Facade-Instanz erzeugen
- Adapter
 - ◆ Anpassung der realen an die erwartete Schnittstelle
- Proxy
 - ◆ Stellvertreter für entferntes Subsystem
- Bridge
 - ◆ Entkopplung der Schnittstelle von der Implementierung

8.5 Beispiel: Vom Analysemodell zur Systemdekomposition

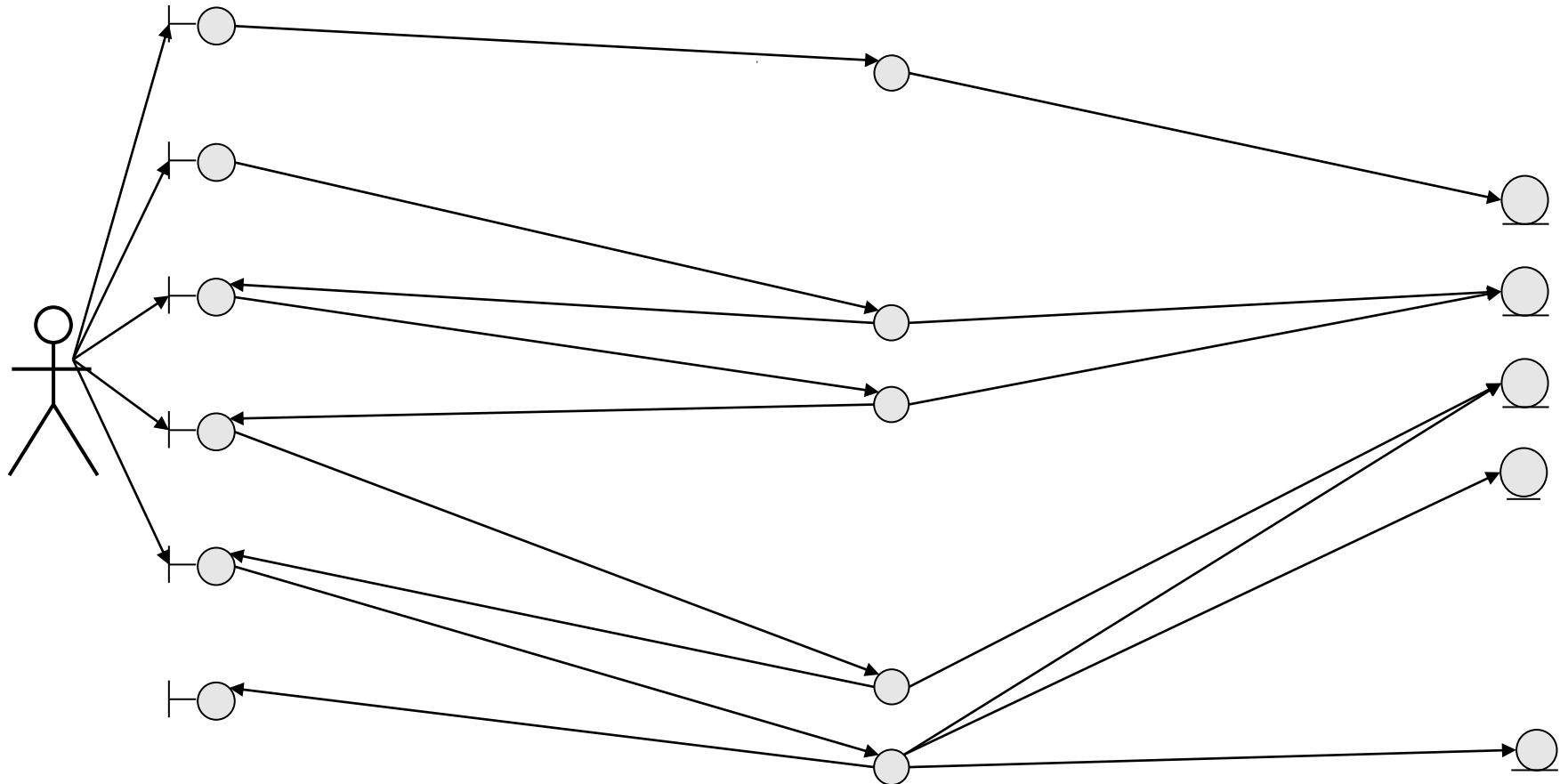
Gruppieren nach ähnlichen Funktionalitäten

Angebotene und genutzte Dienste identifizieren (Schnittstellen)

Subsysteme einführen (Komponenten)

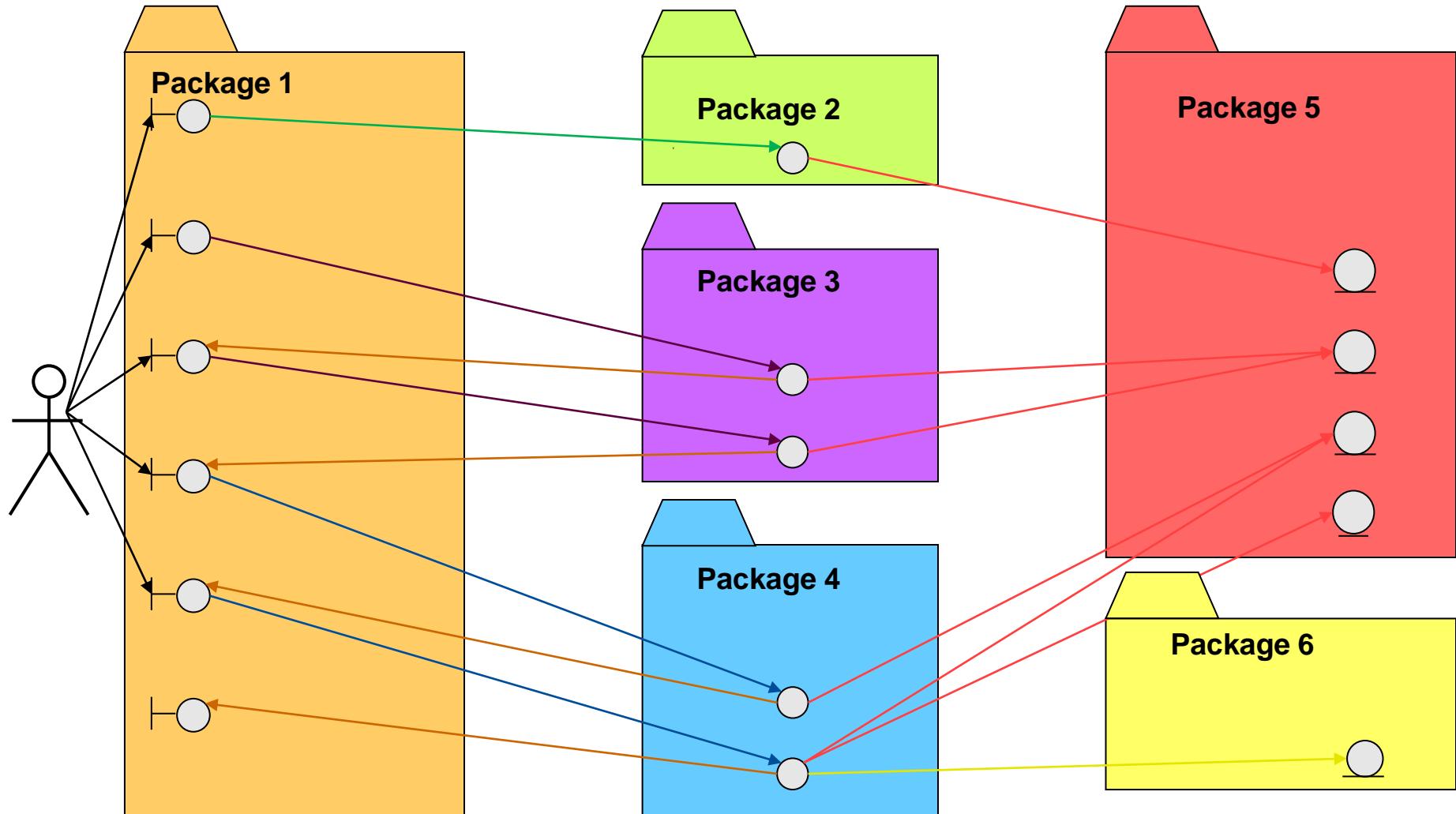
Facades als Einstiegspunkte in die Subsysteme hinzufügen

Ausgangspunkt: Objektmodell der Analyse

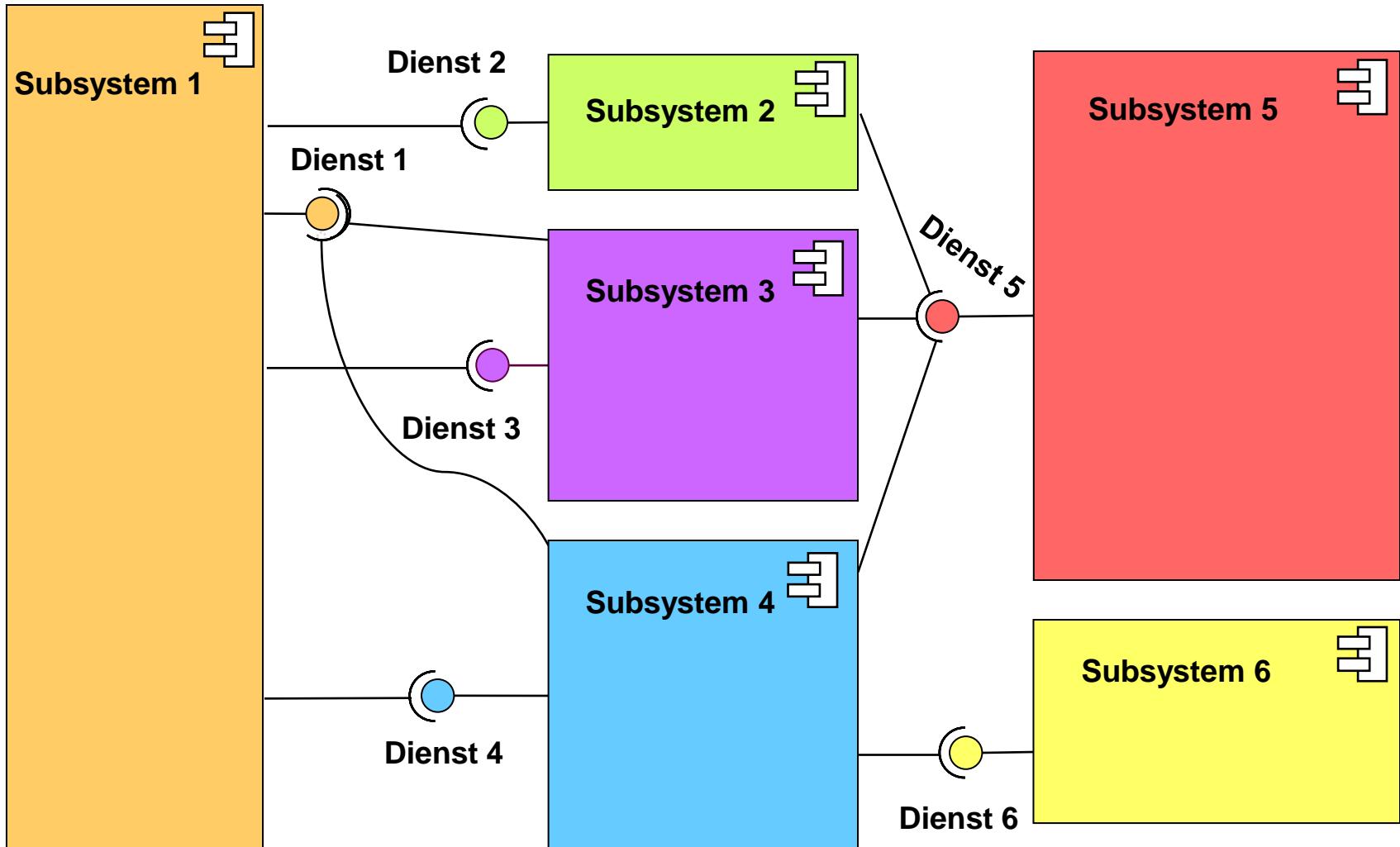
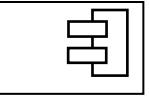


Gruppierung in Packages

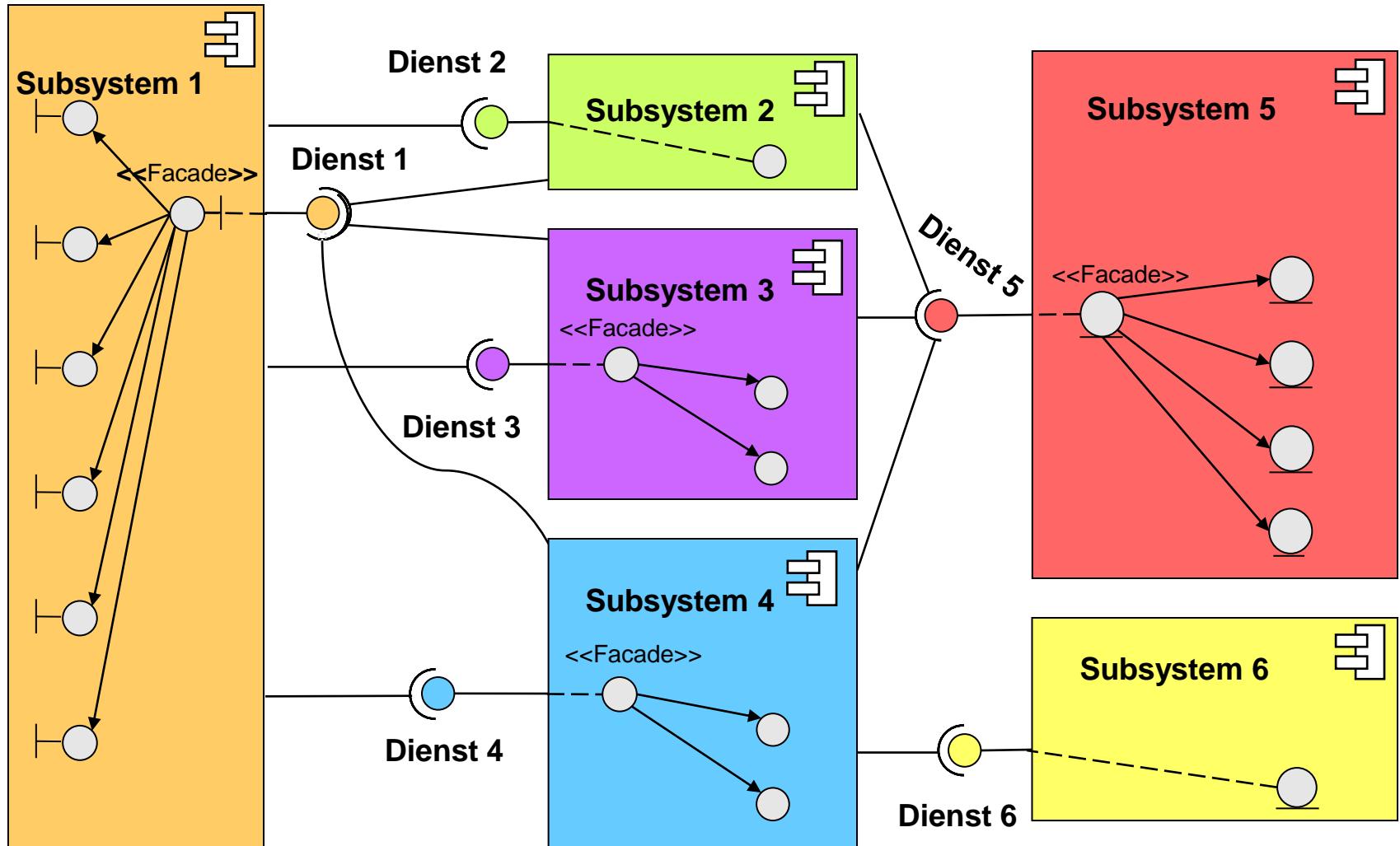
reduziert keine Abhängigkeiten ☹



System-Dekomposition: Komponenten bieten und nutzen Dienste



System-Dekomposition: Dienste-Realisierung mit Facades



Aufgabe (Diskussion mit Kollegen)

- Diskutieren Sie, was für eine Systemdekomposition „gut“ oder „schlecht“ ist.
- Überlegen, Sie ob das vorherige Beispiel eine gute oder schlechte Dekomposition darstellt.
- Kategorisieren Sie die Dekomposition aus dem Beispiel als eine der in der nächsten Vorlesung vorgestellten Software-Architekturen.
- Passt es genau? Brauchen Sie Änderungen damit es passt?

Aufgabe: Was sind Java-Module?

- 1) Versuchen Sie den in Java 9 eingeführten “Modul”-Begriff zu verstehen. Lesen Sie z.B.
 - <https://www.oracle.com/corporate/features/understanding-java-9-modules.html>
 - <https://www.baeldung.com/java-9-modularity>
 - <https://developer.ibm.com/languages/java/tutorials/java-modularity-2/>
- 2) Setzen Sie Java-Module in Beziehung zu dem Komponentenkonzept das hier eingeführt wurde.
Identifizieren Sie mindestens 3 Ähnlichkeiten und 3 Unterschiede.
- 2) Wer korrekte Ergebnisse bis **Montag, 8.01.2024, 16 Uhr** per E-Mail an die Tutorenliste einreicht (Betreff: „Java-Module“), bekommt **10 Übungspunkte** extra.

Systementwurf ▶ Architekturen

Architektur = Alle Subsysteme

- + Beziehungen der Subsysteme (statisch)
- + Interaktion der Subsysteme (dynamisch)

Beispiele in dieser Vorlesung

- Ebenen-Architekturen
- Verteilte Ebenen-Architekturen
 - ◆ Client-Server und Architektur
 - ◆ N-tier
- Peer-To-Peer Architektur
- Repository Architektur
- Model/View/Controller Architektur
- Pipes and Filter Architektur

8.5 Ebenenmodelle, Middleware und Application Server

Schichten und Partitionen
ISO / OSI Referenzmodell
Middleware
Applikationsserver

Schichten und Partitionen

- **Schicht** (=Virtuelle Maschine)
 - ◆ Subsysteme, die Dienste für eine höhere Abstraktionsebene zur Verfügung stellt
 - ◆ Eine Schicht darf nur von tieferen Schichten abhängig sein
 - ◆ Eine Schicht weiß nichts von den darüber liegenden Schichten
- **Partition**
 - ◆ Subsysteme, die Dienste auf der selben Abstraktionsebene zur Verfügung stellen und sich gegenseitig aufeinander beziehen
- Architekturanalysewerkzeuge
 - ◆ Identifikation von Schichten und Partitionen
 - ◆ Warnung vor Abhängigkeiten die der Schichtung entgegenlaufen

Schichten-Architekturen

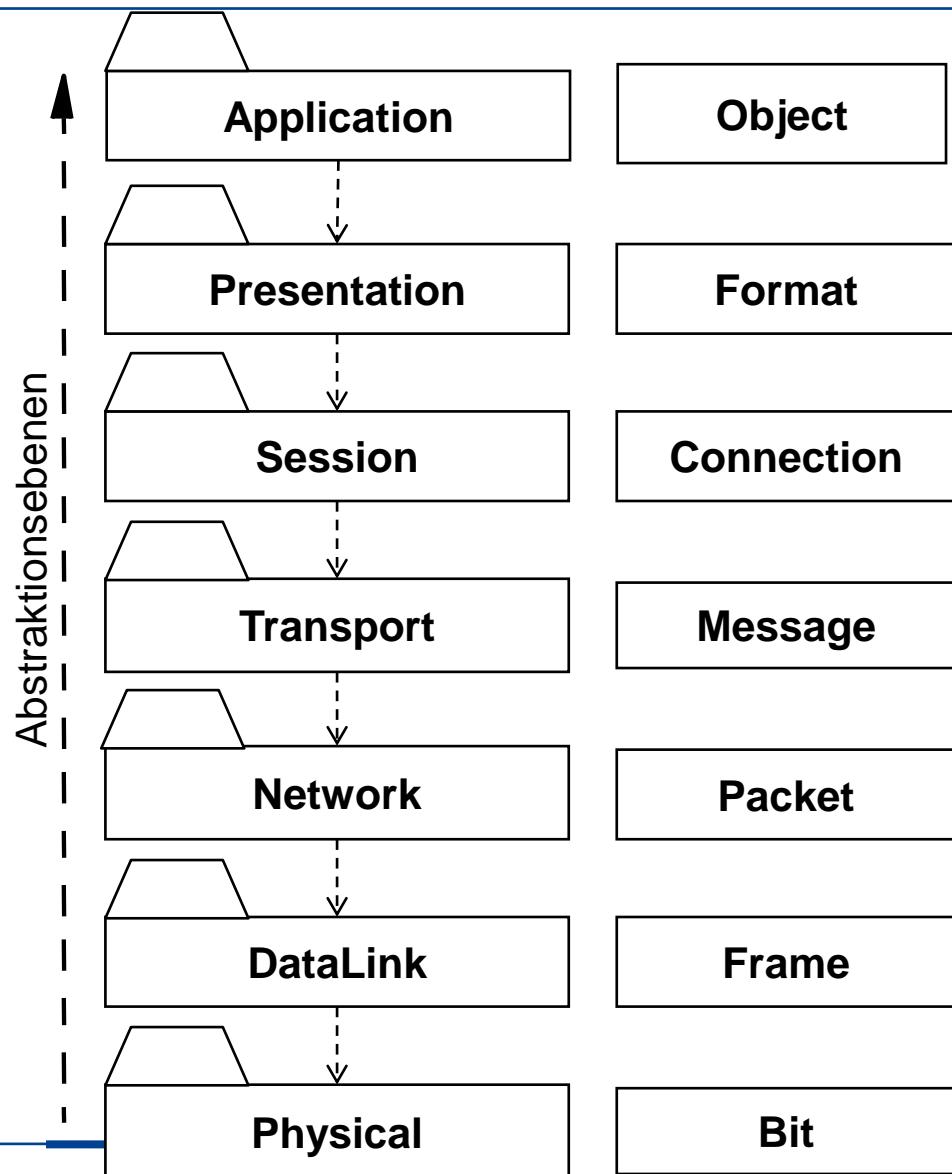
Geschichtete Systeme sind hierarchisch. Das ist wünschenswert, weil Hierarchie die Komplexität reduziert.

- Geschlossene Schichten-Architektur (Opaque Layering)
 - ◆ Jede Schicht kennt nur die nächsttieferne Schicht.
 - ◆ Geschlossene Schichten sind leichter zu pflegen.
- Offene Schichten-Architekturen (Transparent Layering)
 - ◆ Jede Schicht darf alle tiefer liegenden Schichten kennen / benutzen.
 - ◆ Offene Schichten sind effizienter.

Geschlossene Schichten-Architektur

► Beispiel „Verteilte Kommunikation“

- ISO / OSI Referenzmodell
 - ◆ ISO = International Organization for Standardization
 - ◆ OSI = Open System Interconnection
- Das Referenzmodell definiert Netzwerkprotokolle in 7 übereinander liegenden Schichten sowie strikte Regeln zu Kommunikation zwischen diesen Schichten.

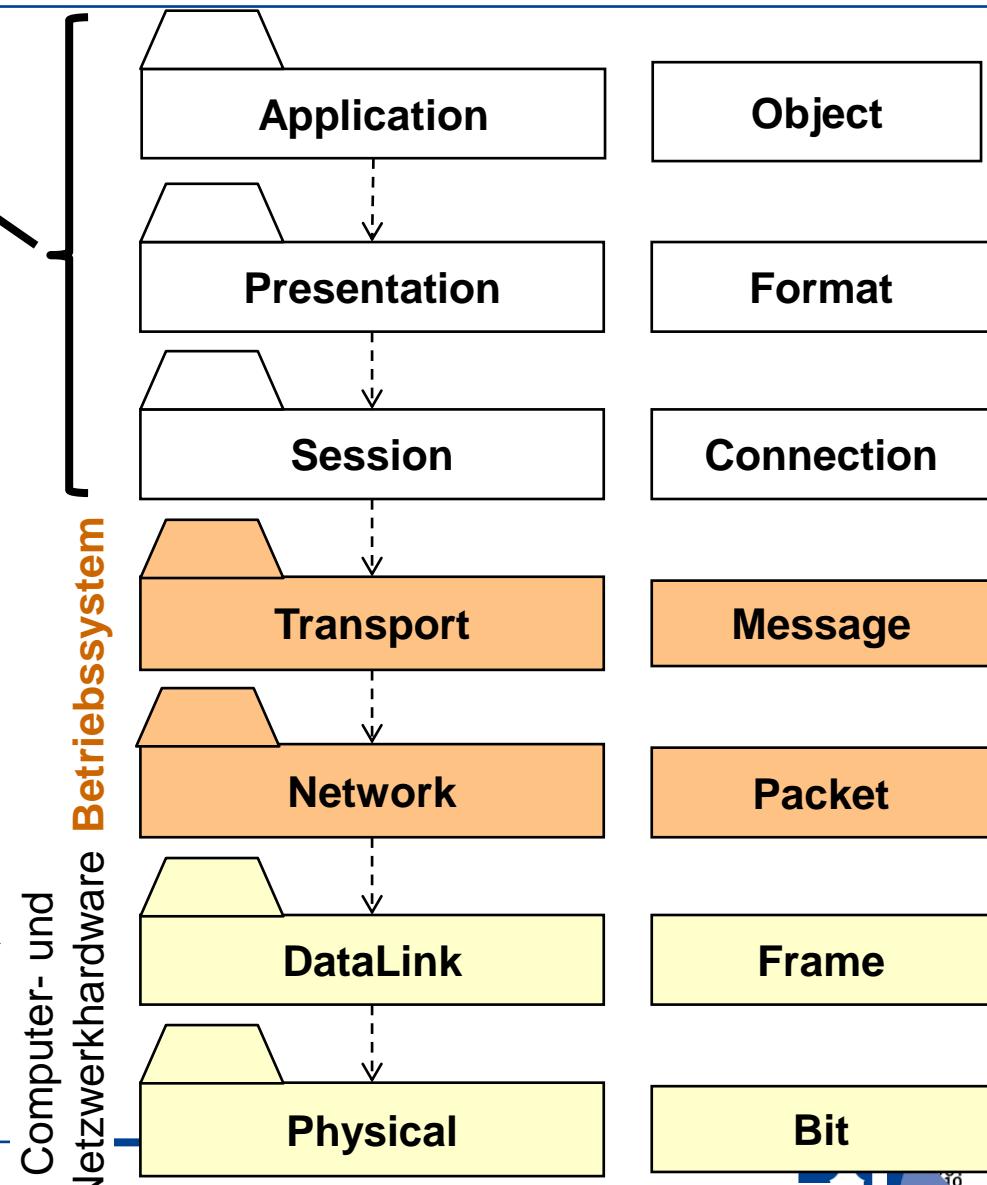


Verteilte Kommunikation im ISO-OSI Modell

Verteilte Programmierung

ist mühsam und fehleranfällig:

- **Verbindungsherstellung** zwischen Prozessen
- **Kommunikation** zwischen Prozessen statt Objekten
- **Packen/Entpacken** von Informationen in Nachrichten statt Parameterübergabe
- **Umcodierung** der Informationen wegen heterogener Platformen
- **Berücksichtigung technischer Spezifika** des Transportsystems



Geschlossene Schichten-Architektur

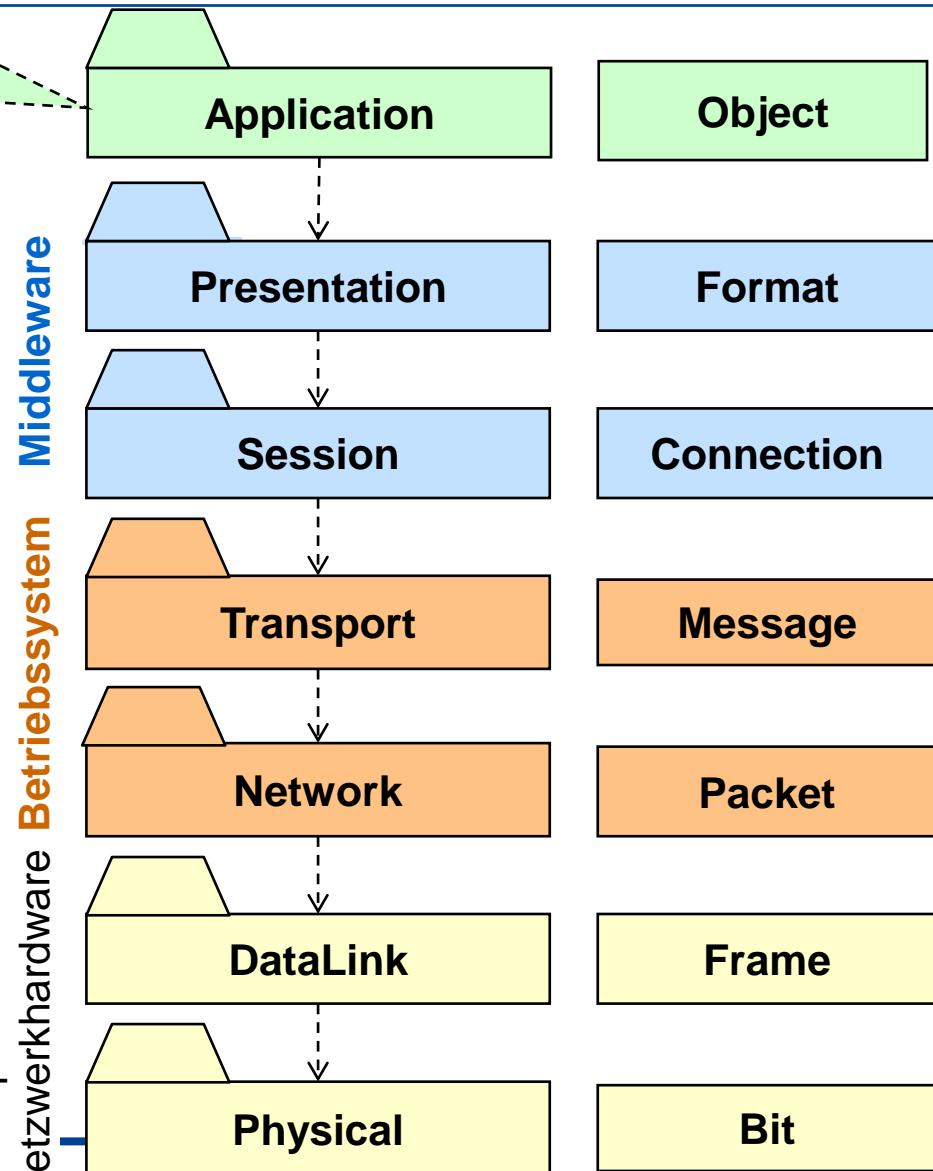
► Middleware erlaubt Konzentration auf Anwendungsschicht

Middleware

Garantiert Transparenz der

- Verteilung
- Plattform

- Plattformunabhängig
- Middlewareabhängig



Middleware

- Definition: Middleware
 - ◆ Softwaresystem auf Basis standardisierter Schnittstellen und Protokolle, das Dienste bietet, die zwischen der Plattform (Betriebssystem + Hardware) und den Anwendungen angesiedelt sind und deren Verteilung unterstützen
- Bekannte Ansätze
 - ◆ Remote Procedure Calls
 - ◆ Java RMI (Remote Method Invocation)
 - ◆ CORBA (Common Object Request Broker Architecture)
- Wünschenswerte Eigenschaften
 - ◆ Gemeinsame Ressourcennutzung
 - ◆ Nebenläufigkeit
 - ◆ Skalierbarkeit
 - ◆ Fehlertoleranz
 - ◆ Sicherheit
 - ◆ Offenheit

Applikationsserver

- Definition: Applicationsserver
 - ◆ Softwaresystem das als Laufzeitumgebung für Anwendungen dient und dabei **über Middleware-Funktionen hinausgehende Fähigkeiten** bietet
 - ⇒ Transparenz der Datenquellen
 - ⇒ Objekt-Relationales Mapping
 - ⇒ Transaktionsverwaltung
 - ⇒ Lebenszyklusmanagement („Deployment“, Updates, Start)
 - ⇒ Verwaltung zur Laufzeit (Monitoring, Kalibrierung, Logging, ...)
- Beispiel-Systeme (kommerziell)
 - ◆ IBM WebSphere
 - ◆ Oracle WebLogic
- Beispiel-Systeme (open source)
 - ◆ JBoss
 - ◆ Sun Glassfish
 - ◆ Apache Tomcat

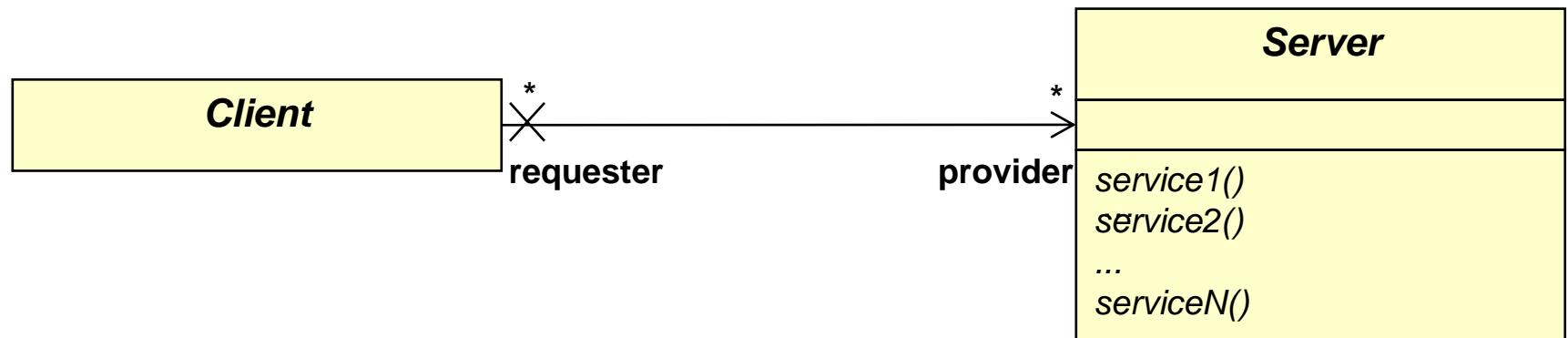
8.6 Ebenen in verteilten Systemen: N-tier Architekturen

Client / Server
Von 2-tier zu n-tier

Schichten-Architektur ▶ Client/Server

Abbildung von Schichten auf Rechner im Netzwerk

- Nutzer interagieren nur mit dem Client
- Server bieten Dienste für Clients
- Clients ruft Operation eines Dienstes auf; die wird ausgeführt und gibt ein Ergebnis zurück
 - ◆ Client kennt das Interface des Servers (seinen Dienst)
 - ◆ Server braucht das Interface des Client nicht zu kennen



Entwurfsziele für Client/Server Systeme

- Portabilität
 - ◆ Server kann auf vielen unterschiedlichen Maschinen und Betriebssystemen installiert werden und funktioniert in vielen Netzwerkumgebungen
- Flexibilität
 - ◆ Server kann mit verschiedenen Front-Ends genutzt werden
- Performance
 - ◆ Client sollte für interaktive, UI-lastig Aufgaben maßgefertigt sein
 - ◆ Server sollte CPU-intensive Operationen bieten
- Skalierbarkeit
 - ◆ Server hat genug Kapazität, um eine größere Anzahl Clients zu bedienen
- Transparenz
 - ◆ Der Server kann selbst verteilt sein, und dennoch den Clients einen einzigen “logischen” Dienst bieten
- Zuverlässigkeit
 - ◆ System kann einzelne Knotenausfälle /Verbindungsprobleme überleben

Schichten-Architektur ▶ Client/Server

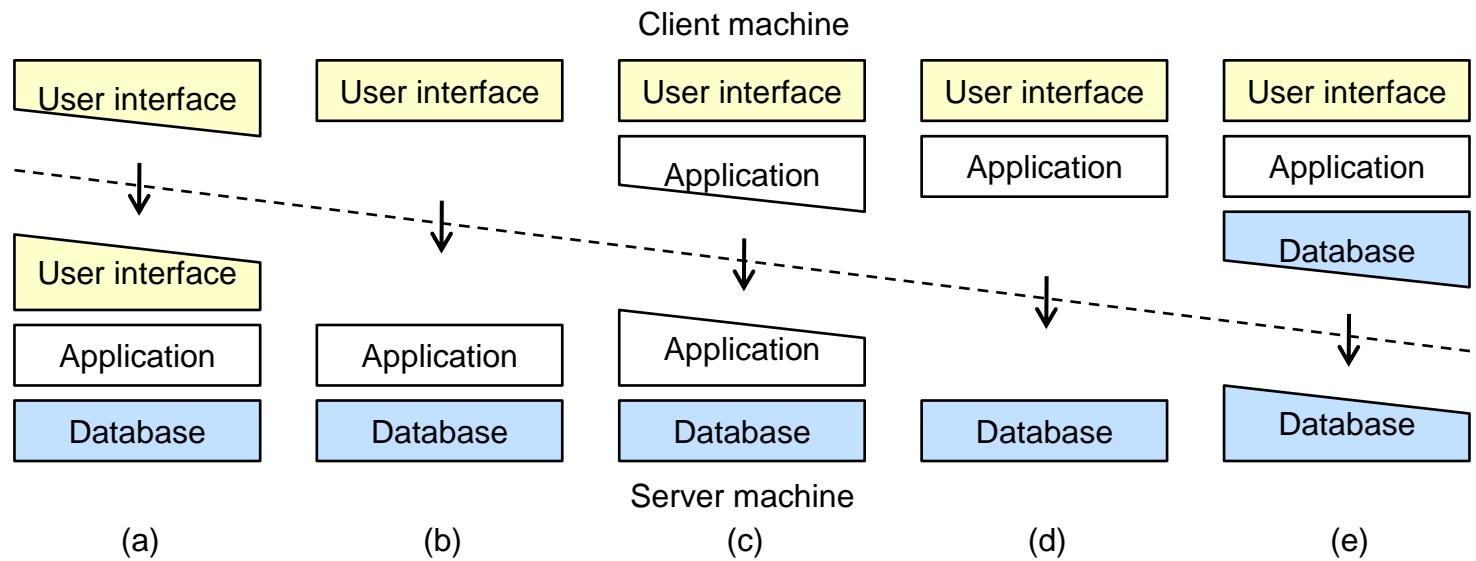
- Oft bei Datenbanksystemen genutzt
 - ◆ Front-End: Nutzeranwendung (Client)
 - ◆ Back-End: Datenbankzugriff und Datenmanipulation (Server)
- Vom Client ausgeführte Funktionen
 - ◆ Maßgeschneiderte Benutzerschnittstelle
 - ◆ Front-end-Verarbeitung der Daten
 - ◆ Aufruf serverseitiger RPCs (Remote Procedure Call)
 - ◆ Zugang zum Datenbankserver über das Netzwerk
- Vom Datenbankserver ausgeführte Funktionen
 - ◆ Zentrales Datenmanagement
 - ⇒ Datenintegrität und Datenbankkonsistenz
 - ⇒ Datenbanksicherheit
 - ◆ Zentrale Verarbeitung
 - ◆ Nebenläufige Operationen (Mehrbenutzerzugriff)

2-Tier Architektur

- Ältestes Verteilungsmodell: Client- und Server-Tier
- Zuordnung von Aufgaben zu ‚tiers‘ (engl. ‚tier‘ = Stufe / Schicht)
 - ◆ Präsentation → Client
 - ◆ Anwendungslogik → Beliebig
 - ◆ Datenhaltung → Server
- Vorteile
 - ◆ Einfach und schnell umzusetzen
 - ◆ Performant
- Probleme
 - ◆ Schwer wartbar
 - ◆ Schwer skalierbar
 - ◆ Software-Update Problem

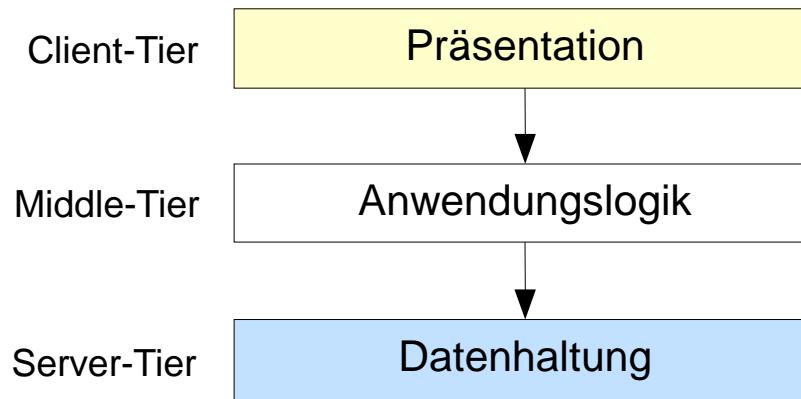
2-Tier Architektur ▶ Varianten

- ◆ Ultra-Thin-Client Architekturen (a):
 - ⇒ Die Client-Tier beschränkt sich auf Anzeige von Dialogen in einem Browser.
- ◆ Thin-Client Architekturen (b):
 - ⇒ Die Client-Tier beschränkt sich auf Anzeige von Dialogen und die Aufbereitung der Daten zur Anzeige.
- ◆ Fat-Client Architekturen (c,d,e):
 - ⇒ Teile der Anwendungslogik liegen zusammen mit der Präsentation auf der Client-Tier.



3-Tier Architekturen

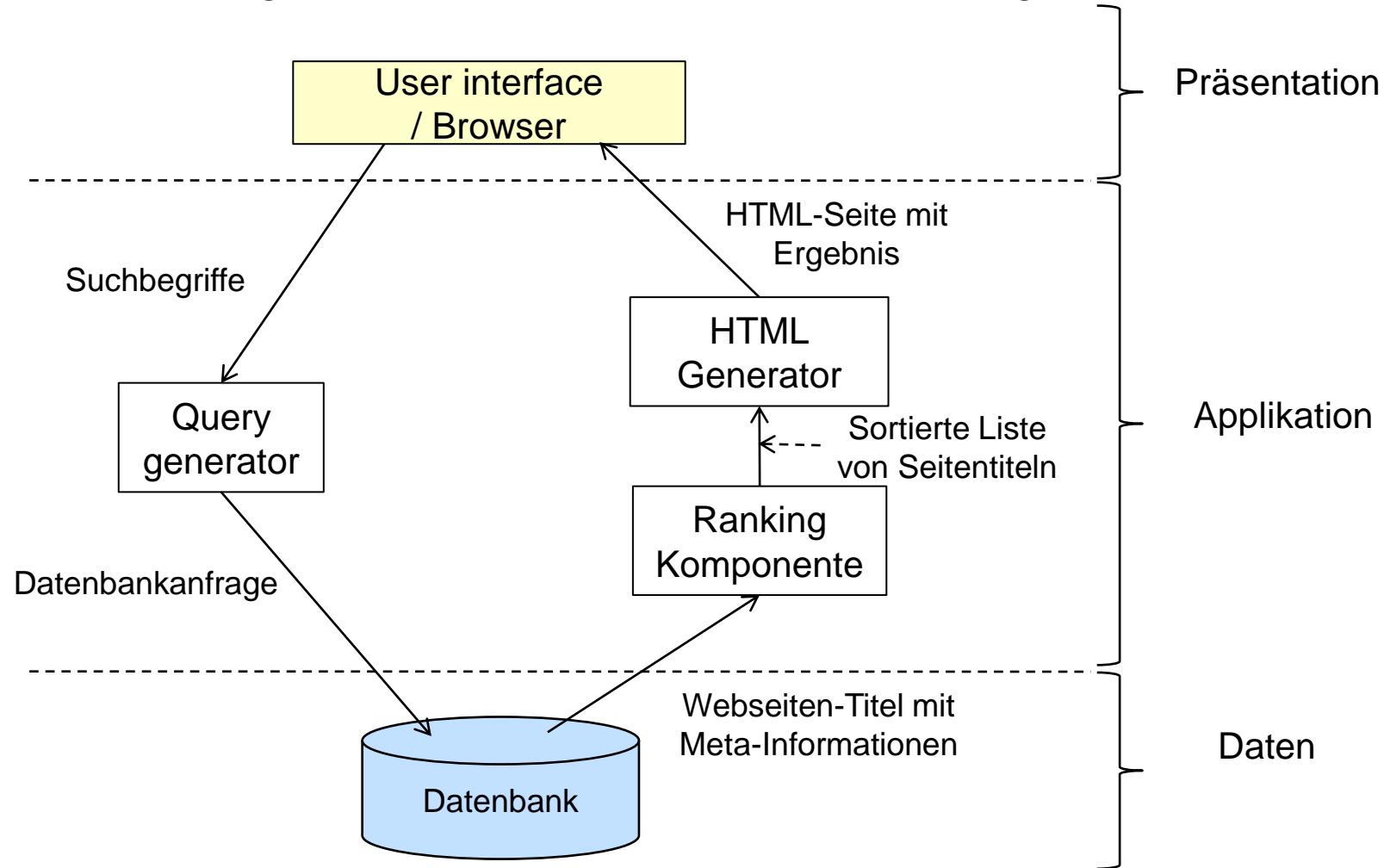
- Zuordnung von Aufgaben zu 3 Tiers



- Standardverteilungsmodell für einfache Webanwendungen
 - ◆ Client-Tier = **Browser** zur Anzeige
 - ◆ Middle-Tier = **Webserver** mit Servlets / ASP / Anwendung
 - ◆ Server-Tier = **Datenbankserver**

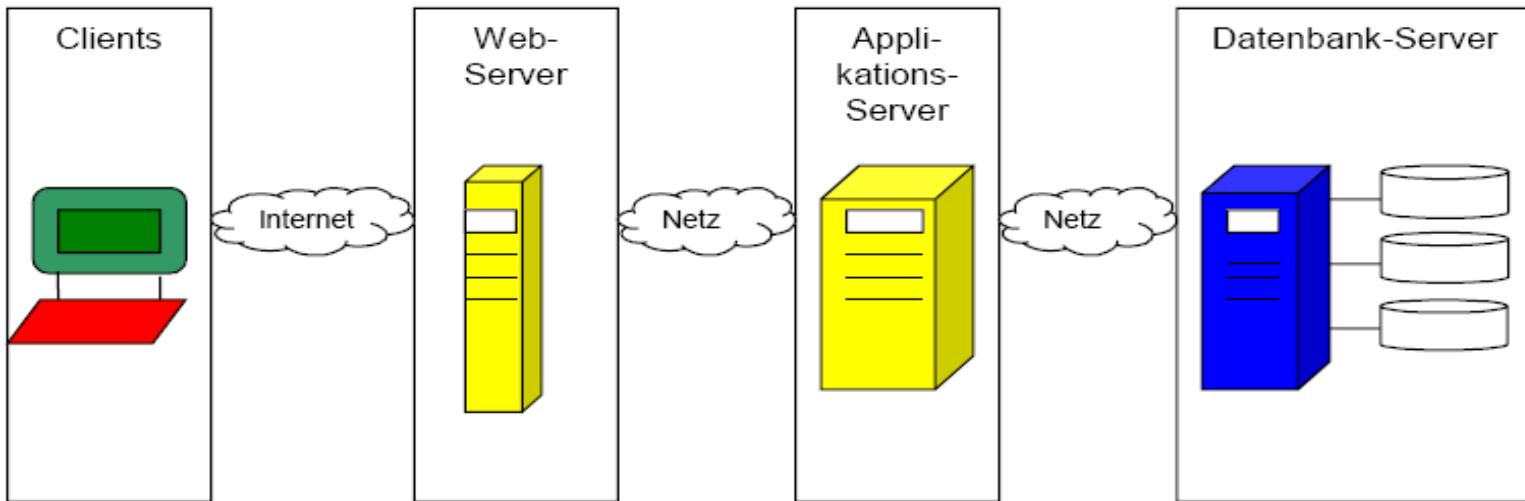
3-Tier Architekturen ▶ Beispiel Webanwendungen

- Standardverteilungsmodell für einfache Webanwendungen:



4- und Mehr-Tier Architekturen

- Unterschied zu 3-Schichten-Architekturen
 - ◆ Die Anwendungslogik wird auf mehrere Schichten verteilt (zB. Webserver, Application Server)



- Motivation
 - ◆ Minimierung der Komplexität („Divide and Conquer“)
 - ◆ Besserer Schutz einzelner Anwendungsteile
- Grundlage für die meisten Applikationen im E-Bereich
 - ◆ E-Business, E-Commerce, E-Government, ...

8.7 Weitere verteilte Architekturen

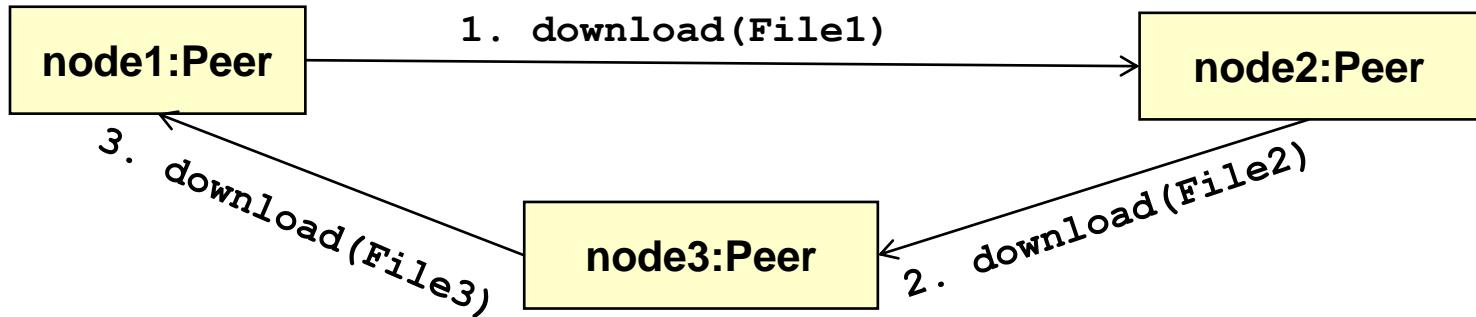
Pipe and Filter

Repository

Model-View-Controller

Probleme mit Client/Server Architekturen

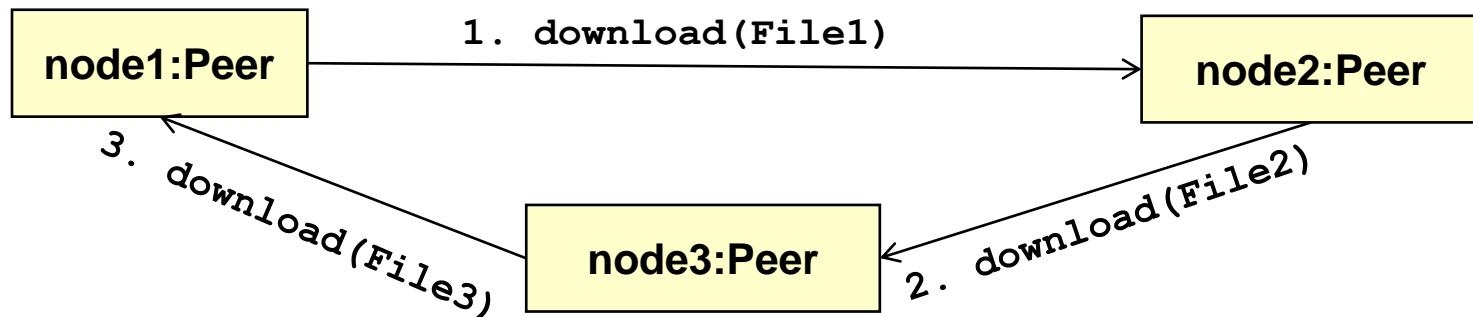
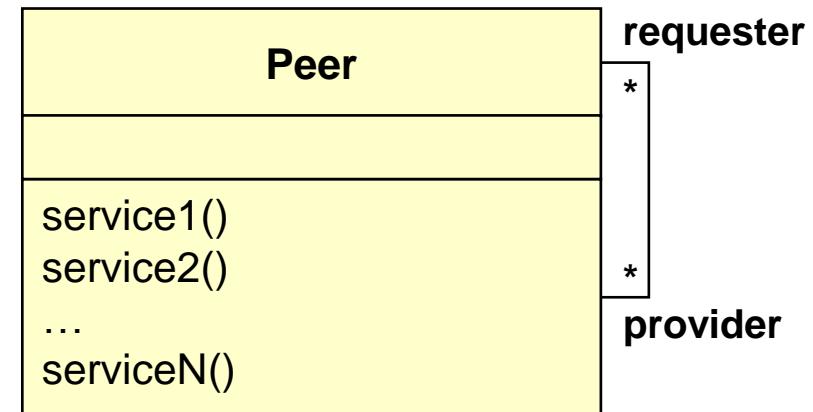
- Geschichtete Systeme unterstützen keine gleichberechtigte gegenseitige („Peer-to-peer“) Kommunikation
- „Peer-to-peer“ Kommunikation wird oft benötigt
 - ◆ Beispiel File-Sharing-Netze: Jeder Netzwerkknoten bietet Dateien zum download an und kann selbst welche herunterladen.



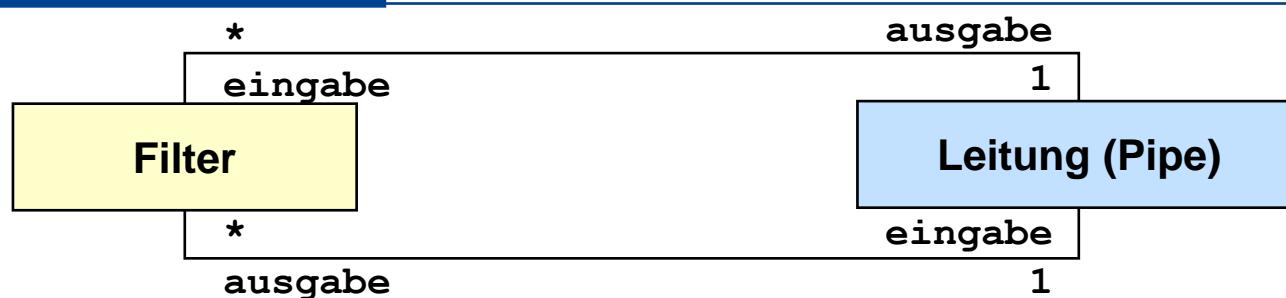
Peer-to-Peer Architektur

Generalisierung der Client/Server Architektur: Clients können Server sein und umgekehrt

- Hochskalierbar
- Ausfallsicherer
- Schwerer anzugreifen
- Schwieriger zu verwalten



Pipe-and-Filter-Architektur



- **Filter-Subsysteme** bearbeiten Daten
 - ◆ Es sind reine Funktionen
 - ◆ Sie sind konzeptionell und implementierungstechnisch unabhängig von
 - ⇒ den Pipes die die Daten zu ihnen bzw. von ihnen Weg leiten
 - ⇒ den Erzeugern und Verbrauchern der Daten
- **Leitungs-Subsysteme** leiten Daten weiter
 - ◆ Sie sammeln Daten von einem oder mehreren Filtern
 - ◆ Sie leiten Daten an einen oder mehrere Filter weiter
 - ◆ Sie dienen der Synchronisation paralleler Filteraktivitäten
 - ◆ Sie sind von allen anderen Subsystemen völlig unabhängig (genau wie die Filter)

Pipe-and-Filter-Architektur: Stärken

- Das Gesamtsystem entsteht einfach durch die Verknüpfung von Pipe- und Filter-Subsystemen

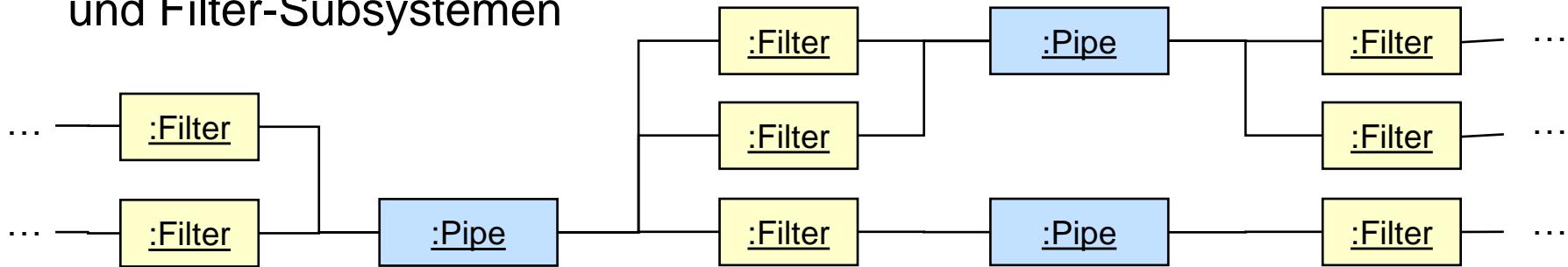


Abb. Beispiel-Ausschnitt aus möglicher Systemkonfiguration

- Vorteile
 - Flexibilität: leichter Austausch von Filtern, Leichte Rekonfiguration der Verbindungen über Pipes
 - Effizienz: Hoher Grad an Parallelität (alle Filter können Parallel arbeiten!)
 - Gut geeignet für automatisierte Transformationen auf Datenströmen
 - ⇒ Beispiel: Sattelitendatenbearbeitung

Pipe-and-Filter-Architektur: Grenzen

- Das Gesamtsystem entsteht einfach durch die Verknüpfung von Pipe- und Filter-Subsystemen

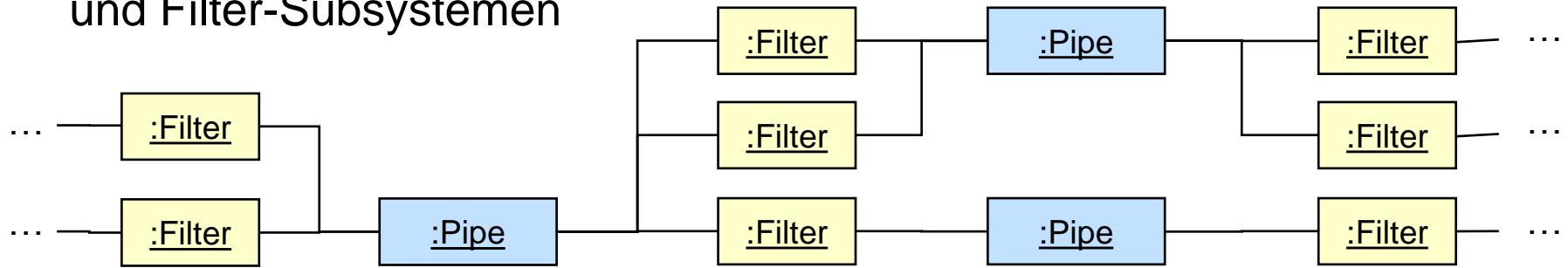
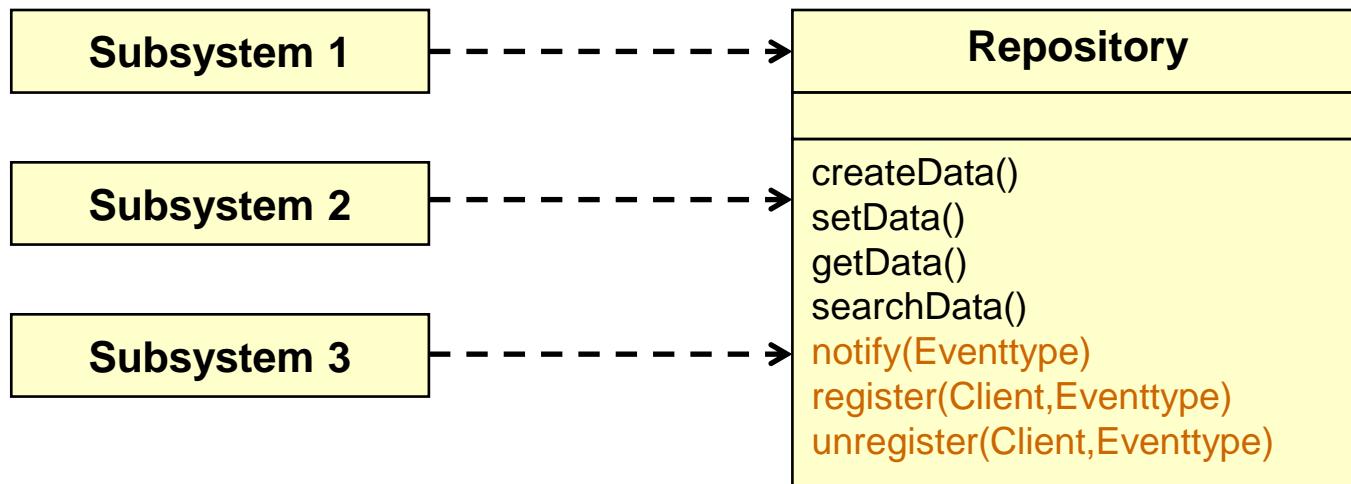


Abb. Beispiel-Ausschnitt aus möglicher Systemkonfiguration

- Weniger geeignet für
 - Hochinteraktive Aufgaben
 - ⇒ Benutzerinteraktion macht die potentielle Parallelität zunicht
 - Aufgaben, wo die Daten sich nicht bzw. nur wenig ändern, da sich dann der Aufwand die Daten ständig zu kopieren nicht lohnt
 - ⇒ In diesem Fall ist eine Repository-Architektur vorteilhafter

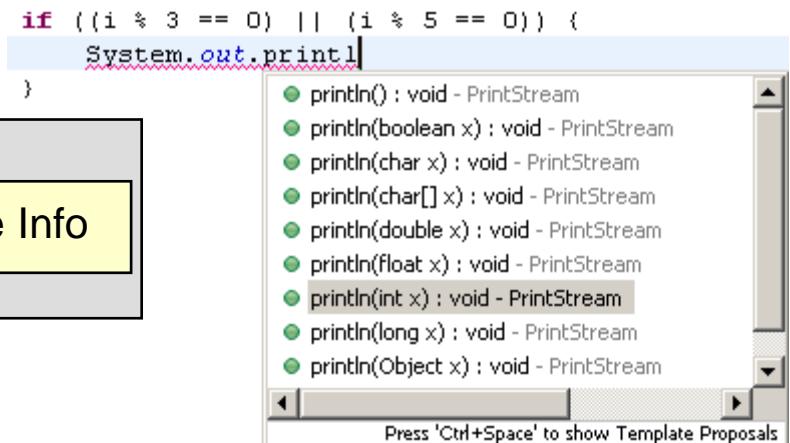
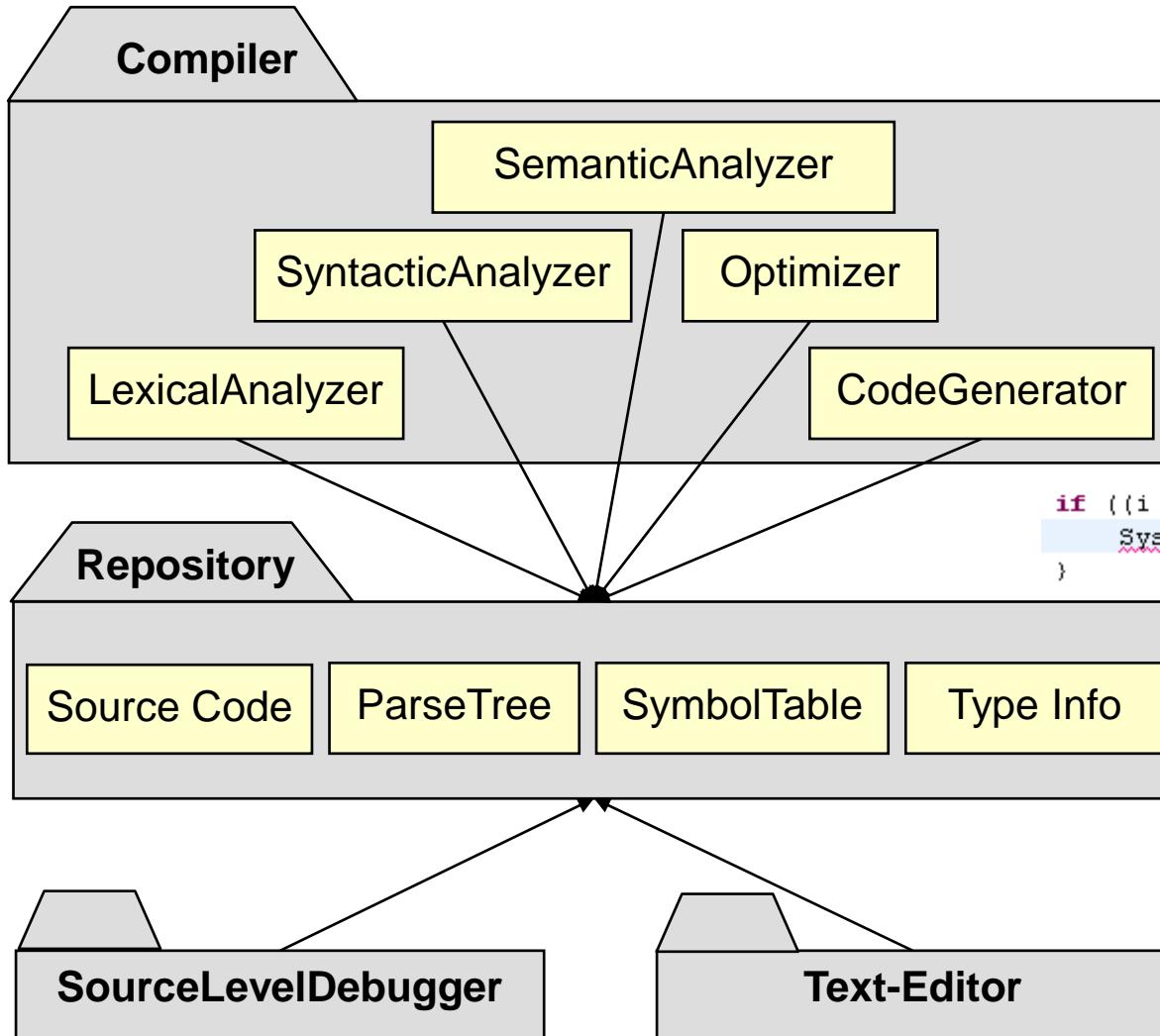
Repository Architektur

- Subsysteme lesen und schreiben Daten einer einzigen, gemeinsamen Datenstruktur
- Subsysteme sind lose gekoppelt (Interaktion nur über das Repository)



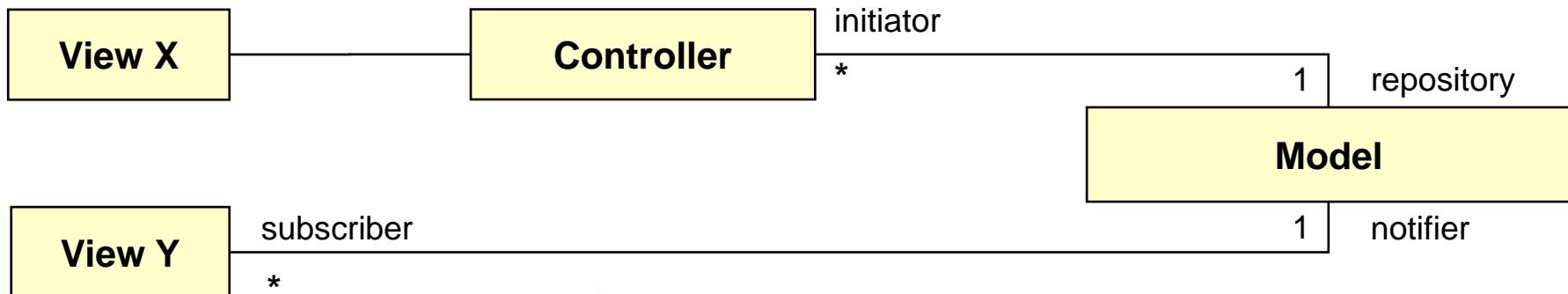
- Repository **benachrichtigt** Clients über Änderungen
 - ◆ Clients können (indirekt, über das Repository) miteinander Kooperieren, ohne sich gegenseitig zu kennen (keine statischen Abhängigkeiten im Programm)
 - ⇒ Observer Pattern

Repository Architektur: Beispiel „Eclipse“

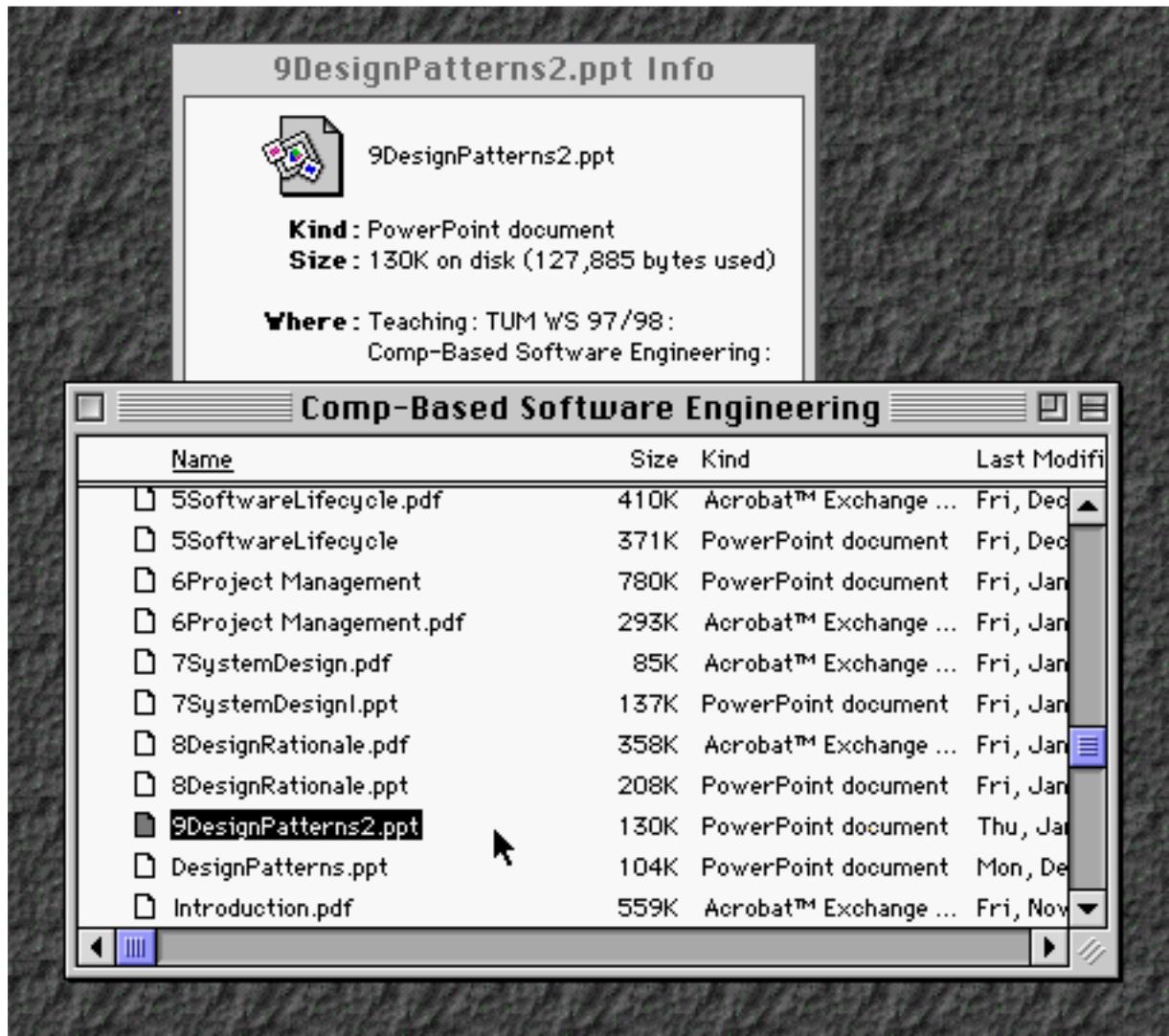


Model/View/Controller

- Subsysteme werden in drei verschiedene Typen unterteilt
 - ◆ **Model** Subsystem: Kapselt Wissen der Anwendungsdomäne und benachrichtigt Views bei Änderungen im Model.
 - ◆ **View** Subsystem: Stellt Objekte der Anwendungsdomäne für den Nutzer dar
 - ◆ **Controller** Subsystem: Steuert Interaktionen mit dem Nutzer.
- MVC ist eine Verallgemeinerung der Repository-Architektur:
 - ◆ Model Subsysteme implementieren die zentrale Datenstrukturen
 - ◆ Controller Subsysteme steuern expliziten Kontrollfluss einzelner Use Cases
 - ◆ Zusätzlich findet eine implizite, eventbasierte Interaktion zwischen unabhängigen Komponenten statt

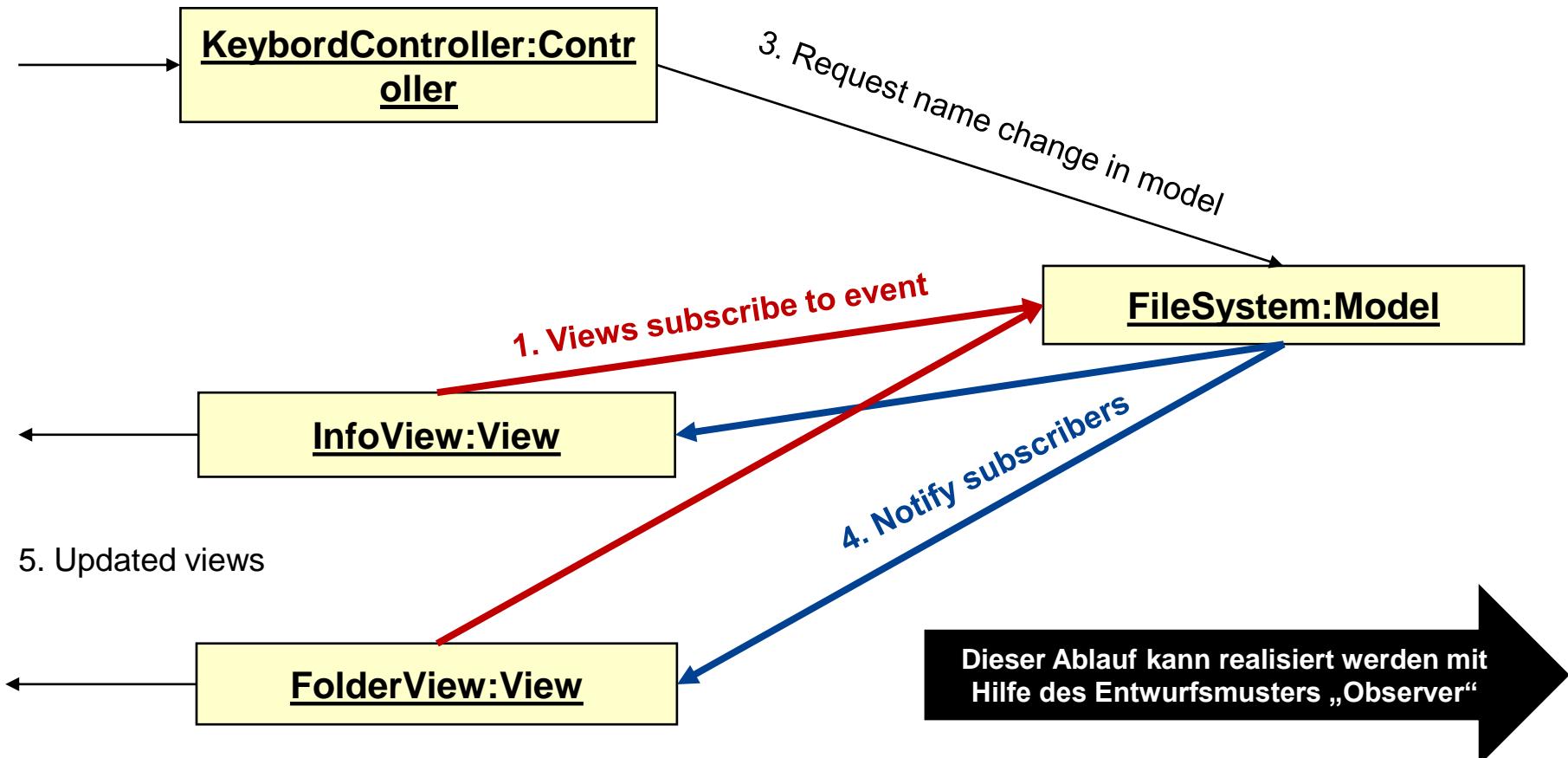


Beispiel einer auf der MVC Architektur basierenden Dateiverwaltung (1984)



Abfolge von Events (im Macintosh 1984 Beispiel)

2. User types new filename



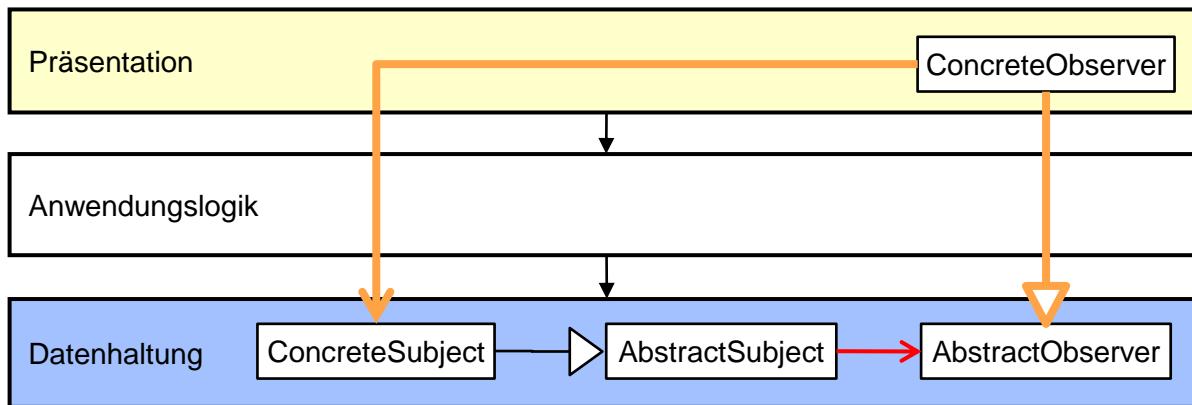
8.8 Software-Architekturen und das Observer-Pattern

Das Observer Pattern: Konsequenzen

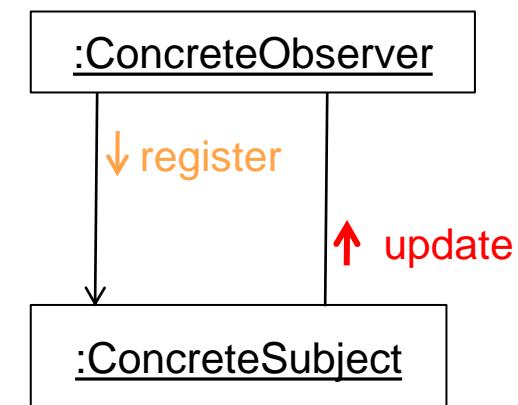
- Abstrakte Kopplung

- ◆ Subjekte aus tieferen Schichten eines Systems können mit Beobachtern aus höheren Schichten **zur Laufzeit** kommunizieren (**update()**-Aufruf), ...
- ◆ ... ohne dass die **statischen** Abhängigkeiten zwischen den Klassen die Ebenenarchitektur verletzen.

Statische Abhängigkeiten:
Hierarchisch

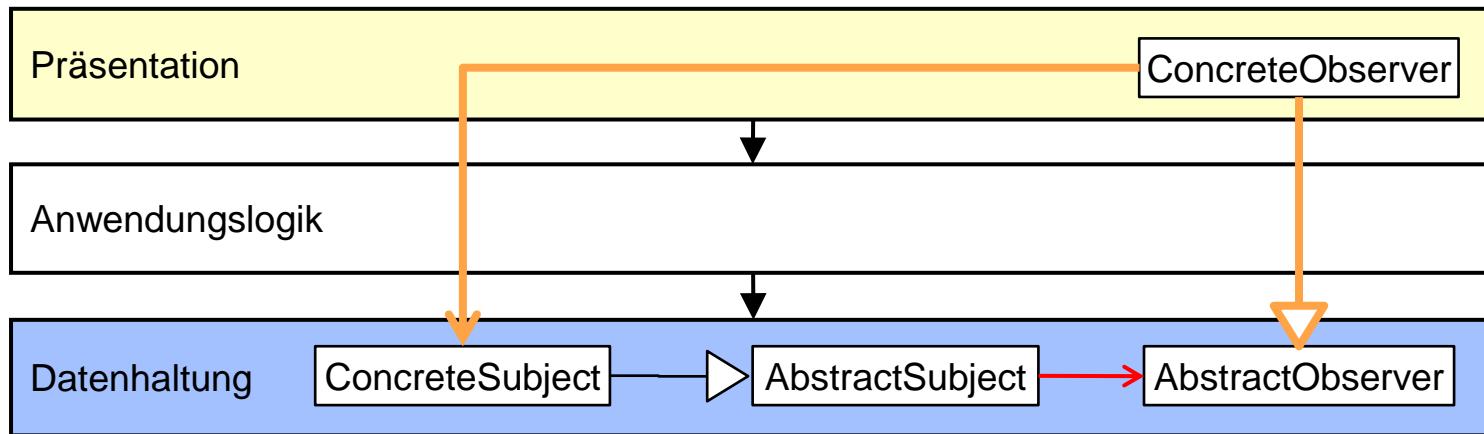


Dynamische Interaktionen:
Bidirektional



Auswirkungen von Observer auf Ebenen: „Presentation-Application-Data“ (1)

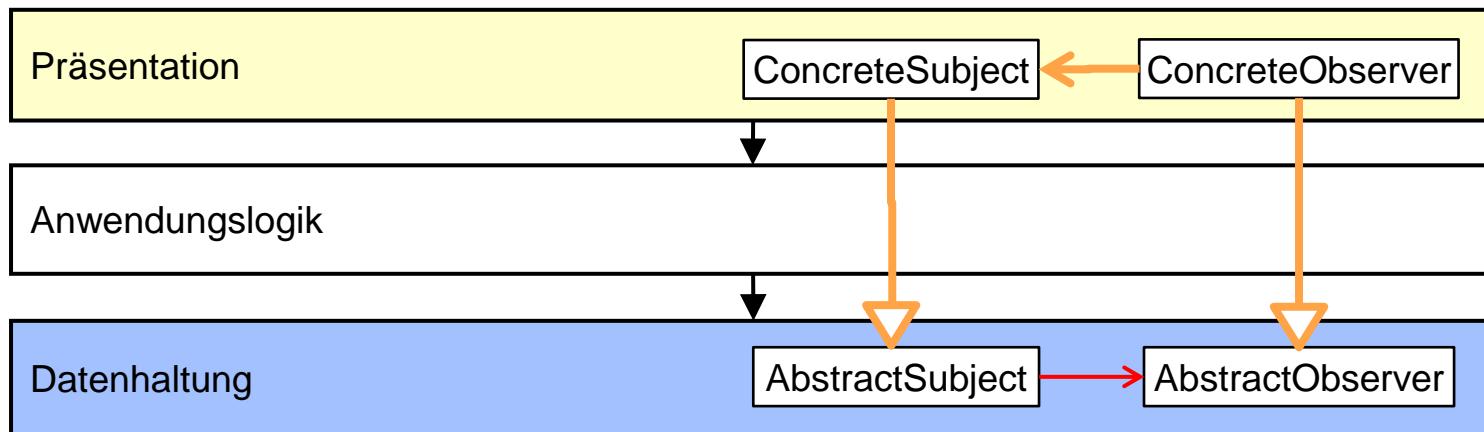
- N-tier-Architekturen basieren auf der rein hierarchische Anordnung von Presentation, Anwendungslogik und Datenhaltung



- Die Aktualisierung der Präsentation bei Änderung der Daten ist durch das Observer-Pattern möglich, ohne dass die Daten von der Präsentation wissen müssen.
 - ◆ Sie sind als AbstractSubjects nur von dem AbstractObserver abhängig.
 - ◆ Da dessen Definition auch in der Datenschicht angesiedelt ist, wird die Ebenenanordnung nicht verletzt

Auswirkungen von Observer auf Ebenen: „Presentation-Application-Data“ (2)

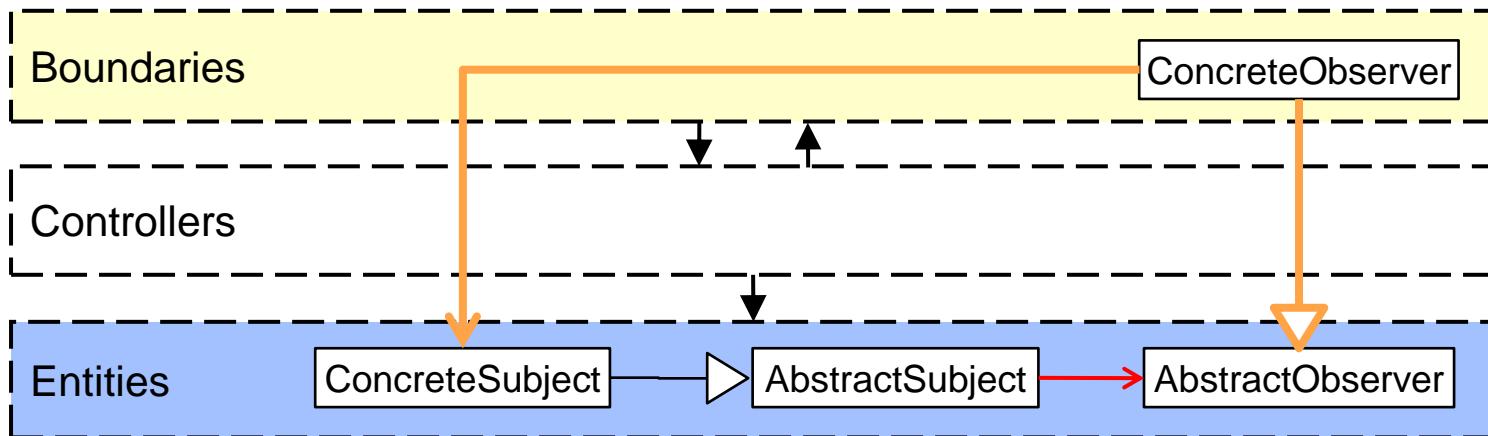
- ConcreteSubjects können in der Präsentations-Ebene angesiedelt sein, ohne die Ebenenarchitektur zu verletzen:



- So können manche Präsentationsobjekte andere Präsentationsobjekte beobachten und sich bei deren Änderung automatisch anpassen
- Der **ConcreteSubject** und/oder **ConcreteObserver** kann auch in der Anwendungslogik liegen
 - ◆ Nur der Fall, dass der **ConcreteObserver** in einer Ebene unter dem **ConcreteSubject** liegt darf nicht auftreten

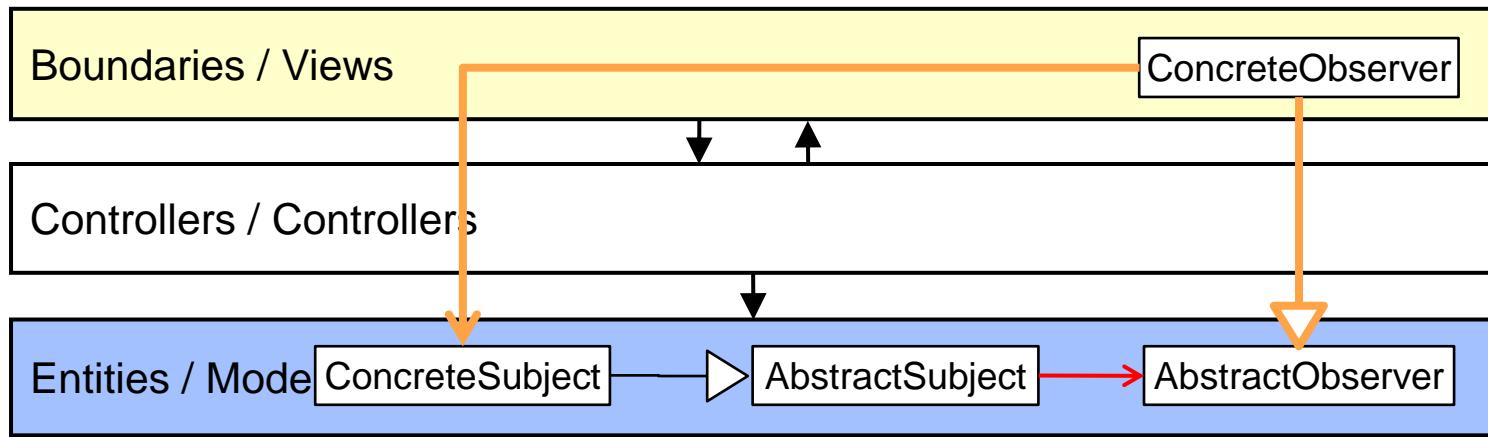
Auswirkungen von Observer für „Boundary-Controller-Entity“ Stereotypen

- Die Abhängigkeitsreduzierung ist die Gleiche wie bei N-Ebenen-Architekturen



- Der einzige Unterschied zwischen BCE und PAD ist die Gruppierung:
 - ◆ BCE beschreibt lediglich die Funktionen einzelner Objekttypen. Es sagt nichts über ihre Gruppierung in Ebenen aus.
 - ◆ PAD sagt etwas über die Gruppierung von Objekttypen gleicher Funktion:
 - ⇒ Alle Boundaries mit GUI-Funktionalität in der Präsentationsschicht
 - ⇒ Controller primär in der Anwendungslogik-Schicht
 - ⇒ Alle Entities in der Datenhaltungs-Schicht

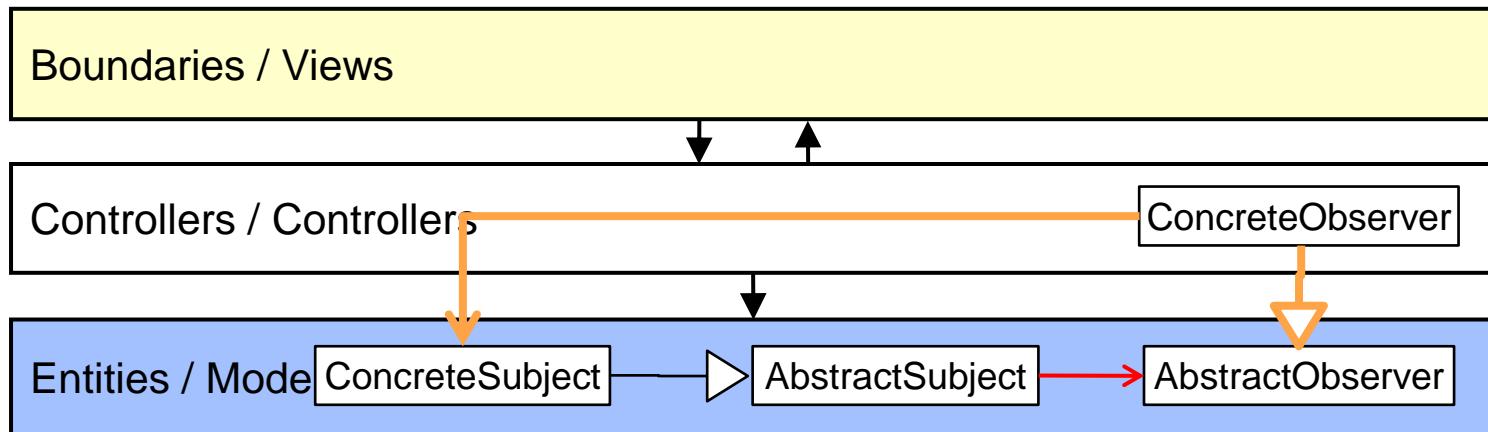
Auswirkungen von Observer für Model-View-Controller



- Views enthalten immer Boundaries die Observer sind
 - ◆ Sonst könnten Sie ihre Funktion nicht erfüllen
 - ◆ Das schließt nicht aus, dass sie eventuell noch andere Rollen spielen
- Boundaries sind nicht nur in Views enthalten
 - ◆ Beispiel: Tastatur

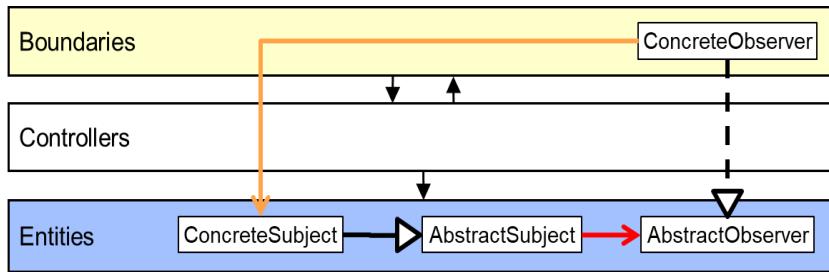
Auswirkungen von Observer für Model-View-Controller

- Observer sind nicht immer Views! Sie können auch Controller sein!



- Sie können sich bei Modellelementen als Observer registrieren, deren Veränderungen sammeln und gefiltert oder kummuliert weitergeben.
 - ◆ Aktive Weitergabe: Aufruf / Steuerung von Aktionen anderer Objekte („Controller“-Rolle)
 - ◆ Passive Weitergabe: Benachrichtigung von eigenen Observern („Modell“-Rolle)
 - ◆ Siehe auch „Mediator-Pattern“ („Vermittler“)

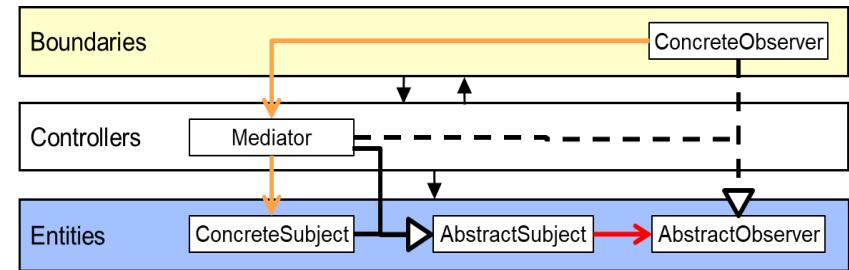
Problem



Die orangene Assoziation verletzt

- eine strikte Ebenenarchitektur
- das Prinzip, dass Boundaries nicht von Entities abhängen sollen

Lösung



- Klasse in „Controllers“-Schicht, die eine Doppelrolle spielt
 - ◆ Als ConcreteSubject aus Sicht des ConcreteObserver
 - ◆ Als ConcreteObserver aus Sicht des ConcreteSubject
- Kann bestehender Controller sein oder eigene Hilfsklasse (s. auch „Mediator“-Pattern)

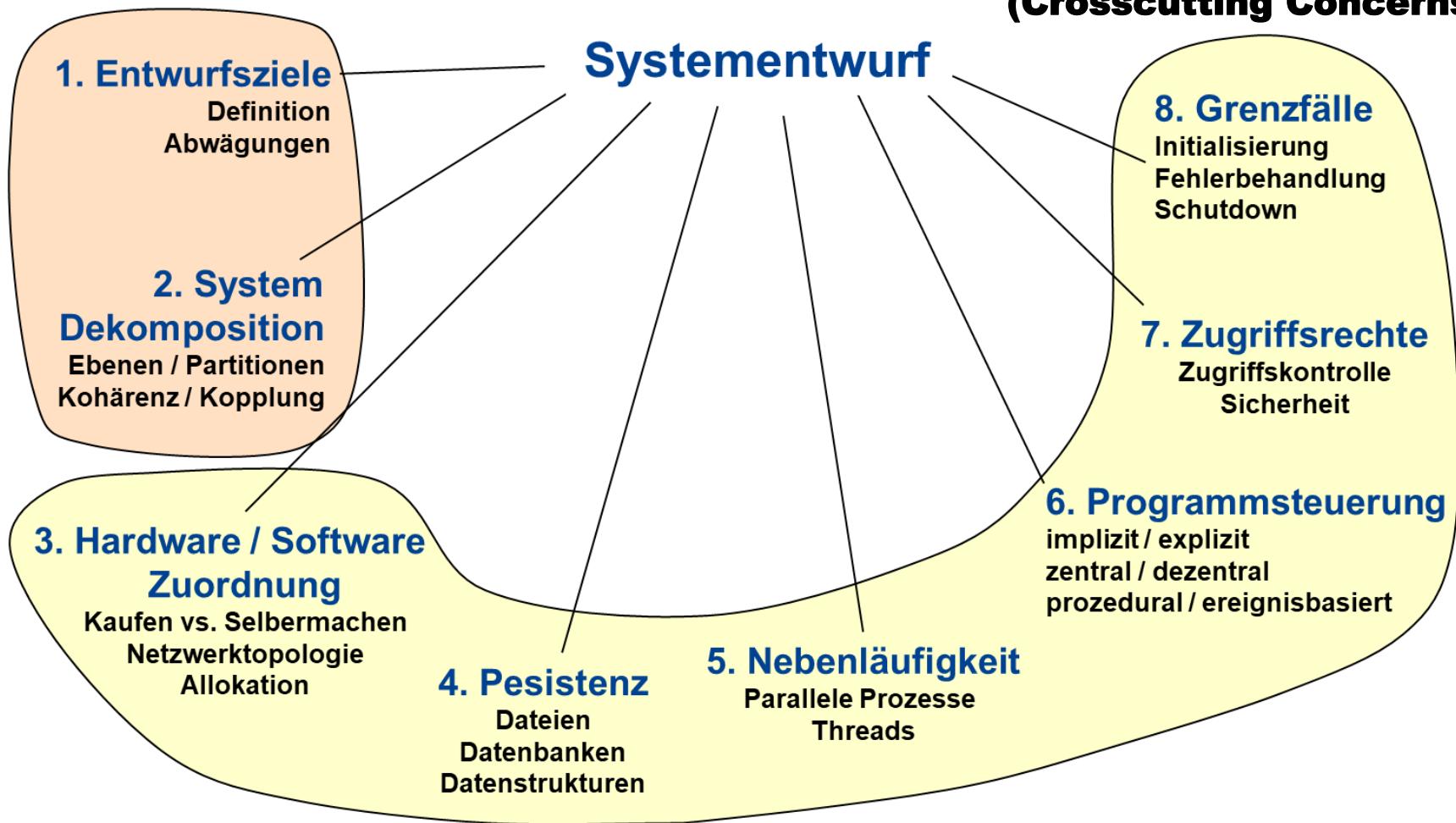
Zusammenfassung (bis hierhin)

- Systementwurf
 - ◆ Verkleinert die Lücke zwischen Anforderungen und der Implementierung
 - ◆ Zerteilt das Gesamtsystem in handhabbare Stücke
- Definition der Entwurfsziele
 - ◆ Beschreibt und priorisiert die für das System wichtigen Qualitäten
 - ◆ Definiert das Wertesystem anhand dessen Optionen überprüft werden
- Subsystemdekomposition
 - ◆ Führt zu einer Menge lose gekoppelter Teile, die das System bilden
- Softwarearchitektur
 - ◆ Beschreibt die Beziehungen / Interaktionen der Subsysteme
- Observer Pattern
 - ◆ Ermöglicht dynamische Abhängigkeiten zu modellieren ohne statische Abhängigkeiten (im Programmcode) einzuführen

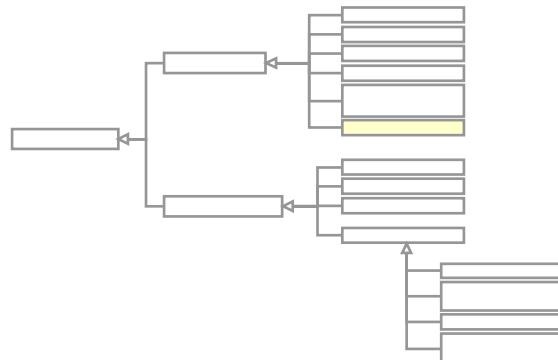
8.9 „Übergeordnete Belange“

Überblick „8.9 Übergeordnete Belange“

Bisher:
Ziele, Dekomposition, Architektur



8.9.1 Hardware-/Software-Zuordnung und Verteilungsdiagramme (Deployment Diagrams)



4. Hardware/Software Zuordnung

- Fragen

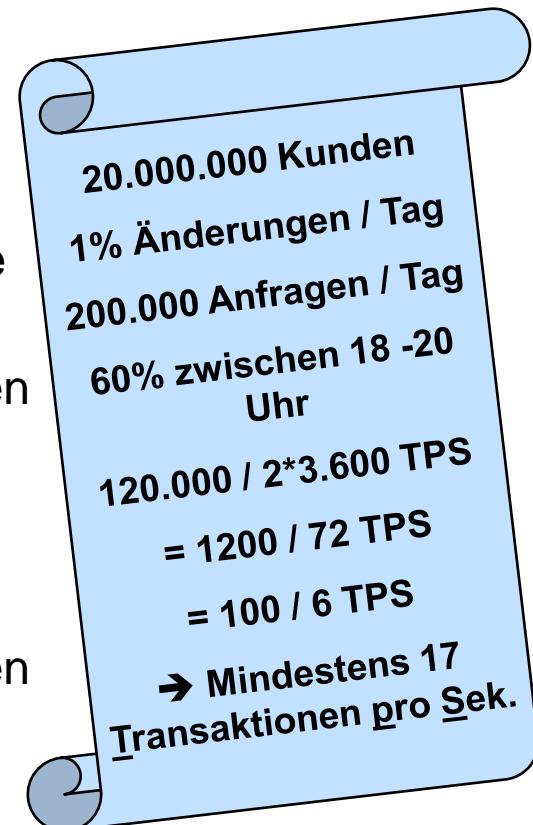
- ◆ Wie sollen wir jedes einzelne Subsystem realisieren
 - ⇒ In Hardware oder in Software?
- ◆ Welche Hard- / Software ist schon verfügbar?
 - ⇒ Altsysteme die man integrieren muss
 - ⇒ Komponenten von Drittanbietern die man nutzen kann
- ◆ Wie wird das Objektmodell auf die gewählte Hard- und Software abgebildet?
 - ⇒ Komponenten abbilden → auf Rechner / Prozessor, Speicher, I/O
 - ⇒ Assoziationen abbilden → auf Netzwerktopologie

- Hard- und Software-Zuordnung oft durch Nebenbedingungen des Kunden eingeschränkt

- ◆ „Wir haben gerade erst N Millionen für System X ausgegeben...“
- ◆ „Aufgabe Y muss von Hard- /Software Z gelöst werden.“

Zuordnung von Objekten / Klassen / Komponenten

- Auf Rechner / Prozessor
 - ◆ Ist die geforderte Berechnungsgeschwindigkeit zu hoch für einen einzelnen Prozessor?
 - ◆ Bringt die Verteilung der Aufgaben auf verschiedene Prozessoren einen Geschwindigkeitsgewinn?
 - ◆ Wie viele Prozessoren sind für den dauerhaft stabilen Betrieb unter Dauerlast notwendig?
- Auf Speicher
 - ◆ Ist genug Speicher vorhanden, um Belastungsspitzen abzufangen?
- Auf I/O
 - ◆ Reicht die Kommunikationsbandbreite zwischen den Hardware-Einheiten, auf denen Subsysteme eingesetzt werden, um die gewünschte Reaktionszeit zu garantieren?



Zuordnung der Assoziationen: Kommunikationstopologie

- Beschreibe die **physikalische Topologie** der Hardware
 - ◆ Entspricht oft der physikalischen Schicht in ISO's OSI Referenzmodell
- Beschreibe die **logische Topologie** der Assoziationen zwischen den Subsystemen
 - ◆ Welche Assoziationen und <>-Beziehungen werden auf physikalische Verbindungen abgebildet?
 - ◆ Welche werden nicht direkt auf physikalische Verbindungen abgebildet? Wie sollen diese implementiert werden?
- Zu stellende Fragen
 - ◆ Was ist das Übertragungsmedium? (Ethernet, Wireless, ...)
 - ◆ Welche "Quality of Service" (QOS) ist erforderlich?
 - ◆ Welche Kommunikationsprotokolle sollen / können genutzt werden?
 - ◆ Sollen Interaktion asynchron, synchron oder sperrend sein?
 - ◆ Welche Bandbreite braucht man zwischen bestimmten Subsystemen?

Verteilungsdiagramm (Einsatzdiagramm / Deployment Diagram)

- Ziel: Spezifikation der
 - ◆ Hardware/Software-Zuordnung
 - ◆ Verteilung im Netzwerk
 - ◆ Einsatz zur Laufzeit
- Elemente
 - ◆ **Manifestation** (=Realisierung) von Komponenten durch Artefakte,
 - ◆ **Einsatz** von Artefakten auf Ausführungsknoten,
 - ◆ **Ausführungsknoten** (Rechner und Laufzeitumgebungen),
 - ◆ **Kommunikationsbeziehungen** zwischen Ausführungsknoten,
 - ◆ **Konfiguration** des Einsatzes,
 - ◆ sonstige Beziehungen (Abhängigkeits-Pfeile mit Stereotypen)

Verteilungsdiagramm ▶ Knoten

- Komponente

- ◆ Logische Einheit mit expliziten Abhängigkeiten
- ◆ Wie im Komponentendiagramm definiert

<<component>>
CALENDAR



- Artefakt

- ◆ Physische Einheit, z.B. Modell, Hilfetext, Quellcode, ausführbarer Code (class-file, jar-Archiv, o-file).
- ◆ Realisierung einer oder mehrerer Komponenten

<<artifact>>
Calendar.jar



- Laufzeitumgebung („execution environment“)

- ◆ Softwaresystem in dem Artefakte zum Einsatz kommen
- ◆ Z.B. Java Virtual Machine, Applikationsserver, ...

<<execution environment>>
:Browser

- Gerät („device“)

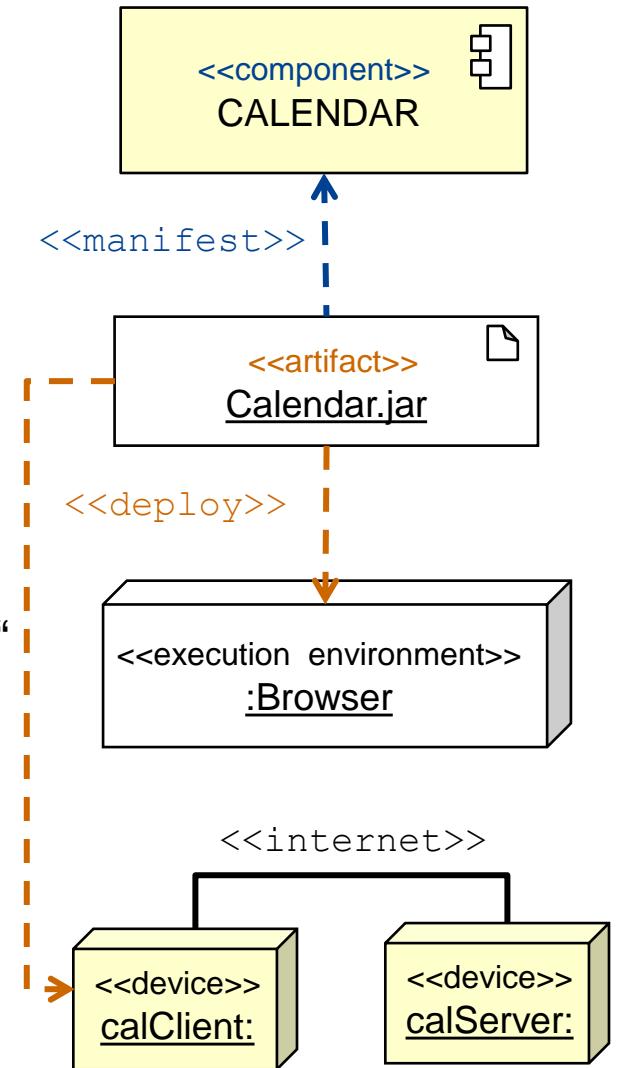
- ◆ Physikalisches Gerät (Rechner) auf dem Artefakte zum Einsatz kommen -- direkt oder indirekt, in einer Laufzeitumgebung

<<device>>
calendarClient:PC

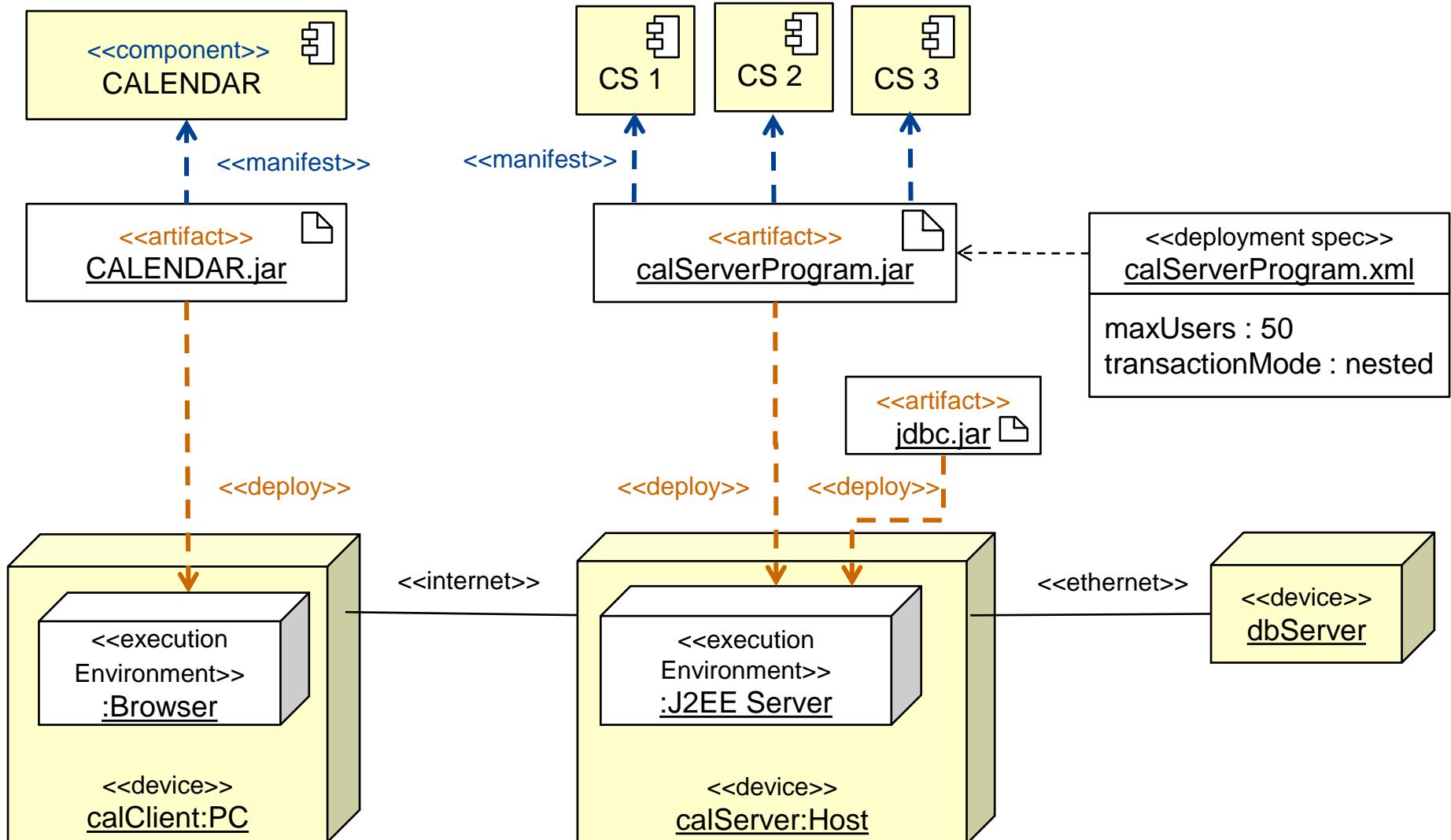


Verteilungsdiagramm ▶ Kanten

- Manifestation (<<manifest>>)
 - ◆ Komponente ist durch Artefakt realisiert
 - ◆ „Artefact is manifestation of component“
- Einsatzbeziehung (<<deploy>>)
 - ◆ Artefakt wird auf Ausführungsumgebung oder Gerät eingesetzt
 - ◆ „Artefact is deployed on device / in environment“
- Kommunikationsbeziehung
 - ◆ Physische Verbindung über die Ausführungsumgebungen kommunizieren
 - ◆ Art wird als Stereotyp angegeben, z.B. <<internet>>, <<ethernet>>, ...



Verteilungsdiagramm ▶ Beispiel



8.9.2 Datenmanagement / Persistenz

Datenmanagement

- Manche Daten / Objekte müssen persistent sein
- Persistenz kann realisiert werden durch
 - ◆ Dateien
 - ⇒ Billig, einfach
 - ⇒ Low level (Lese-/Schreiboperationen)
 - ⇒ Der Anwendung muss gegebenenfalls Code hinzugefügt werden, um eine angemessene Abstraktion zu realisieren
 - ◆ Datenbank
 - ⇒ Mächtig, leicht zu portieren
 - ⇒ Unterstützt mehrere Schreiber und Leser

Datei oder Datenbank?

- Dateien benutzt man für
 - ◆ Große, unstrukturierte Daten (Bitmaps, Core Dumps, Event Traces)
 - ◆ Daten mit geringer Informationsdichte (Archivdateien, Logdateien)
 - ◆ Daten, die nur kurzzeitig zu speichern sind

- Datenbanken benutzt man für
 - ◆ Strukturierte Daten
die in verschiedenen Detailstufen
von vielen Nutzern
zugreifbar sein müssen
(→ Relationen),
(→ Sichten)
(→ Transaktionsmanagement)
 - ◆ Daten, die von vielen Anwendungen
benutzt werden
(→ Transaktionsmanagement)
 - ◆ Daten, die auf verschiedenen Plattformen
zur Verfügung stehen müssen
(→ Datenabstraktion)

Abbildung eines Objektmodells auf eine relationale Datenbank

- Klassendiagramme können auf relationale Datenbanken abgebildet werden
 - ◆ Jede Klasse wird auf eine Tabelle abgebildet
 - ◆ Jedes Attribut einer Klasse wird auf eine Spalte abgebildet
 - ◆ Eine Instanz einer Klasse repräsentiert eine Zeile in der Tabelle
 - ◆ Eine n-zu-m Beziehung wird in eine eigene Tabelle abgebildet
 - ◆ Eine 1-zu-n Beziehung wird als Fremdschlüssel implementiert
- Methoden werden nicht abgebildet ☹
 - ◆ „Stored procedures“ können nicht einzelnen Relationen zugeordnet werden (keine Kapselung, kein dynamisches Binden, ...)
- Vererbung ist schwer abzubilden ☹
 - ◆ Entweder redundante Speicherung von geerbten Feldern in jeder Unterklasse → Update-Probleme
 - ◆ ... oder aufwendige „joins“ → Laufzeitprobleme

Abbildung von Klassen mit Assoziationen (1)

N-zu-M Assoziation: Eigene Tabelle für die Assoziation

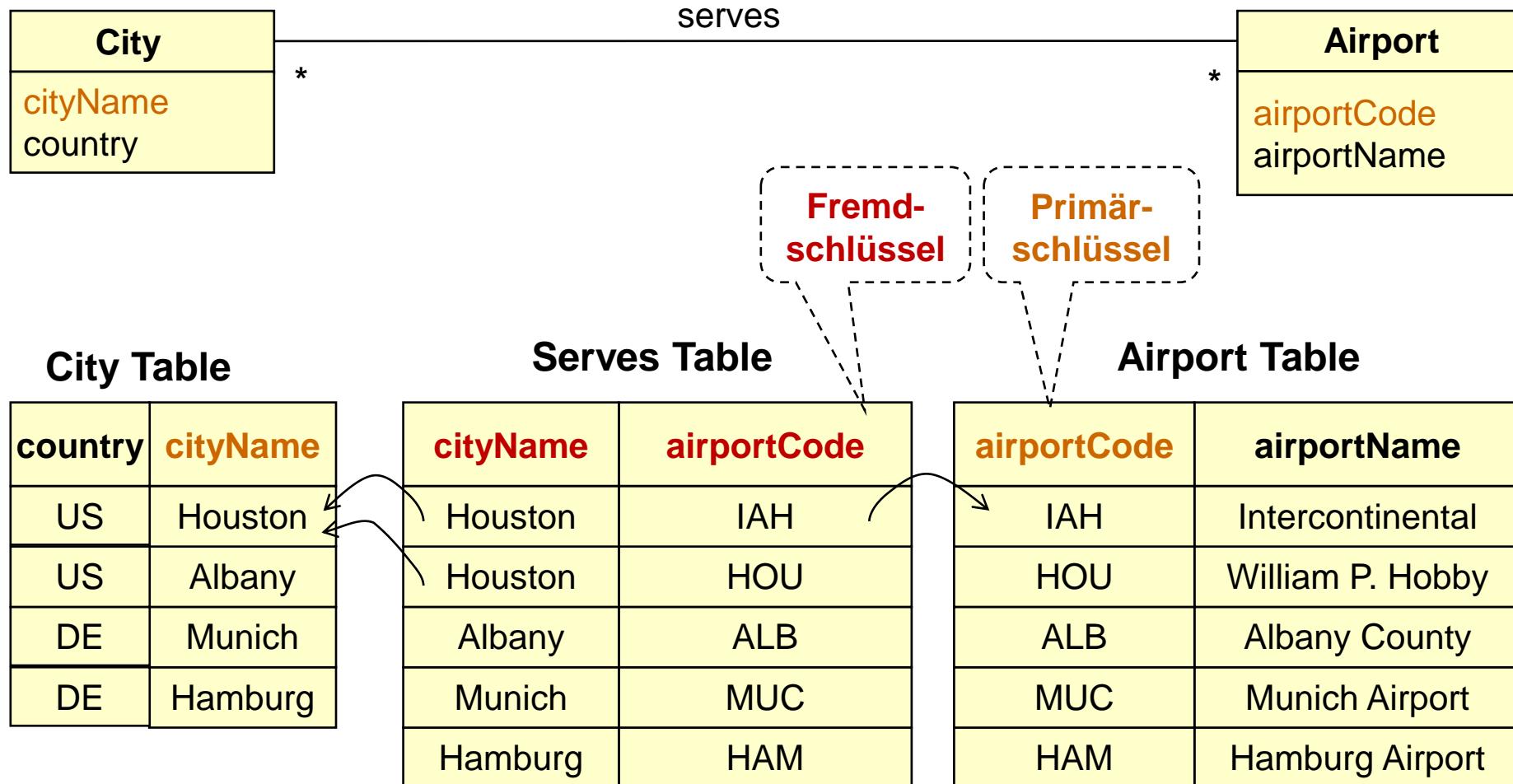


Abbildung von Klassen mit Assoziationen (2)

1-zu-* oder *-zu-1 Assoziationen: Fremdschlüssel auf *-Seite integrieren

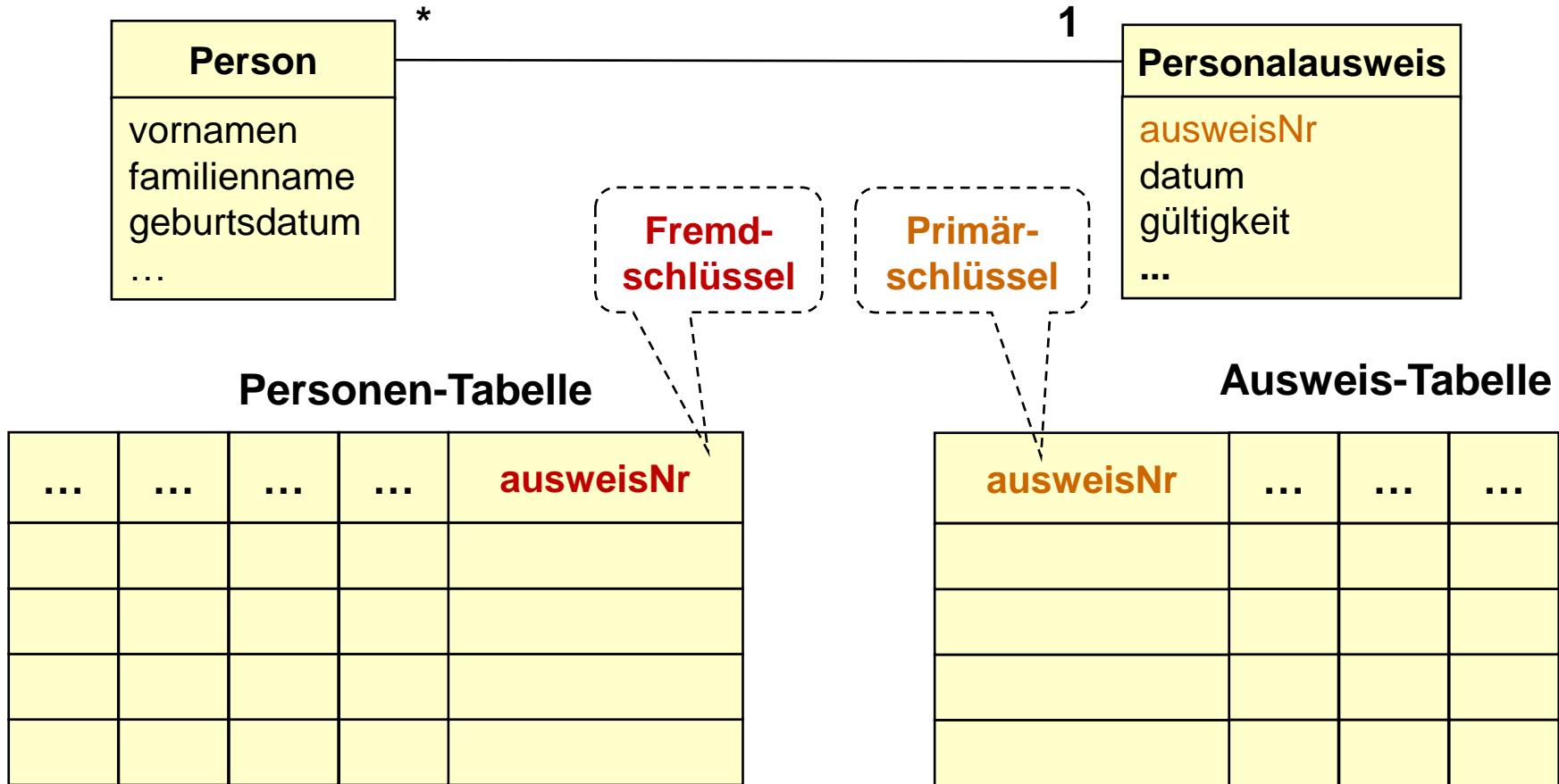
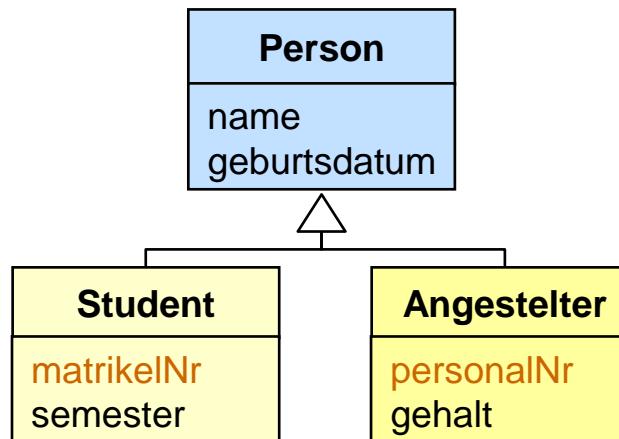


Abbildung von Vererbung (1)

Vererbung (Variante 1 ,flattening')

- Tabellen nur für unterste Unterklassen.
- Jede Tabelle enthält eigene Spalte für jedes Feld der Oberklasse



Studenten-Tabelle

matrNr	sem.	name	gebDatum
201	5	Tom	1.1.1995
202	5	Tim	2.2.1996
203	3	Tara	3.3.1997
204	1	Tanja	4.4.1998

Angestellten-Tabelle

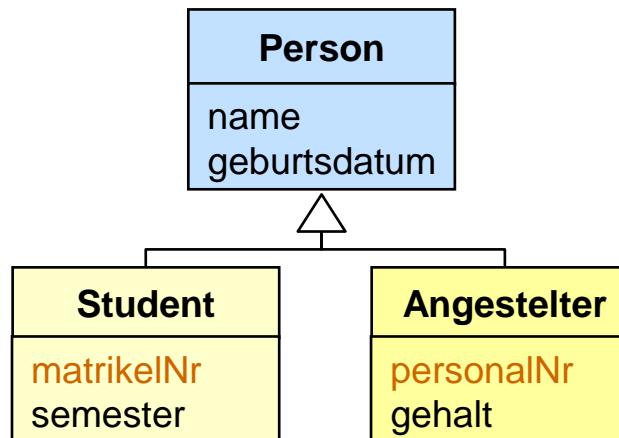
name	gebDatum	persNr	gehalt
Anna	4.5.1960	1	10.000
Adam	5.6.1970	2	5.000
Tara	3.3.1997	10	650
Tim	2.2.1996	11	450

Problem:
Redundanzen

Abbildung von Vererbung (2)

Vererbung Variante 2 (‘join’)

- Eigene Tabelle für jede Klasse.
- Unterklassentabelle enthält Fremdschlüssel der Oberklasse(n).



Studenten-Tabelle

matrNr	sem.	id
201	5	#3
202	5	#4
203	3	#5
204	1	#6

Problem:
Join notwendig

Personen-Tabelle

id	name	gebDatum
#1	Anna	4.5.1960
#2	Adam	5.6.1970
#3	Tom	1.1.1995
#4	Tim	2.2.1996
#5	Tara	3.3.1997
#6	Tanja	4.4.1998

Angestellten-Tabelle

id	persNr	gehalt
#1	1	10.000
#2	2	5.000
#5	10	650
#4	11	450

Von Objektmodellen zu Tabellen ➤

Werkzeuge

- Applikationsserver und ähnliche Werkzeuge erledigen die Abbildung eines Objektmodells auf ein relationales Schema („object relational mapping“) **automatisch**
- Gängige Systeme / Frameworks
 - ◆ Java Persistence API (JPA) – Teil der Java Enterprise APIs
 - ⇒ strikt objekt-relationales Mapping
 - ◆ Java Data Objects (JDO) – Teil der Java Enterprise APIs
 - ⇒ Verallgemeinerung, Mapping auch auf nicht-relationale DB möglich
 - ◆ Hibernate – Teil des Applikationsservers JBoss
 - ⇒ Implementierung von JPA
 - ◆ DataNucleus – Eigenes Produkt
 - ⇒ Implementierung von JDO und JPA
 - ◆ ... Google nach „object relational mapping“ ...

8.9.3 Nebenläufigkeit

Nebenläufigkeit

- Entwurfsziel: Reaktionszeit, Performance
- Aufgabe: Identifizieren potenziell nebenläufiger Ausführungsstränge und Entscheidung über ihre Umsetzung als Threads
- Threads („Fäden“, „Stränge“)
 - ◆ Ein Thread ist ein Pfad durch eine Menge von Zustandsdiagrammen, wobei stets genau ein Objekt zur selben Zeit aktiv ist.
 - ◆ Ein Thread bleibt in einem Zustandsdiagramm bis ein Objekt auf einen Event / eine Nachricht wartet
 - ◆ Thread Abspaltung: Ein Objekt sendet ein asynchrones Event
- Welche Threads sind identifizierbar?

Fragen zur Nebenläufigkeit

Inhärente externe Nebenläufigkeit

- Bietet das System vielen Nutzern Zugriff?
- Kann eine einzelne Anfrage an das System in mehrere Teilanfragen zerlegt werden?
- Können diese Teilanfragen parallel abgearbeitet werden?

Inhärente interne Nebenläufigkeit

- Welche Objekte des Objektmodells sind unabhängig?
 - ◆ Objekte sind inhärent nebenläufig, wenn sie zur gleichen Zeit Events/ Nachrichten empfangen können
 - ◆ Inhärent nebenläufige Objekte sollten verschiedenen Threads zugeordnet werden
 - ◆ Objekte mit sich wechselseitig ausschließenden Aktivitäten sollten demselben Thread zugeordnet werden

Realzeitanforderungen

- Realzeitanforderungen bezeichnen Anforderungen an Software in einer **bestimmten Zeitspanne** zu reagieren oder etwas zu einem **bestimmten Zeitpunkt** zu tun
 - ◆ Meistens geht es um sehr kurze Zeitspannen (Sekundenbruchteile)
 - ◆ Typischerweise bei "Eingebetteten Systemen" (Mischung aus Hard- und Software) im Fahrzeug- und Maschinenbau, Telekommunikation, ...
- Dilemma
 - ◆ Software ist flexibler (leichter austauschbar und wartbar) und kostengünstiger als Hardware
 - ◆ Aussagen über die absolute Zeit, die ein bestimmter Aufruf dauert sind aber sehr schwer zu treffen
 - ⇒ Problem "Dynamisches Binden": Welcher Code wird aufgerufen?
 - ⇒ Problem "Garbage Collection": Wann unterbricht sie evtl. einen Aufruf?
- UML
 - ◆ Spezifikation von Realzeitanforderung durch timing-constraints in Sequenzdiagrammen und durch „Timing Diagramme“ (s. Anhang)

8.9.4 Programmsteuerung (Kontrollparadigma)

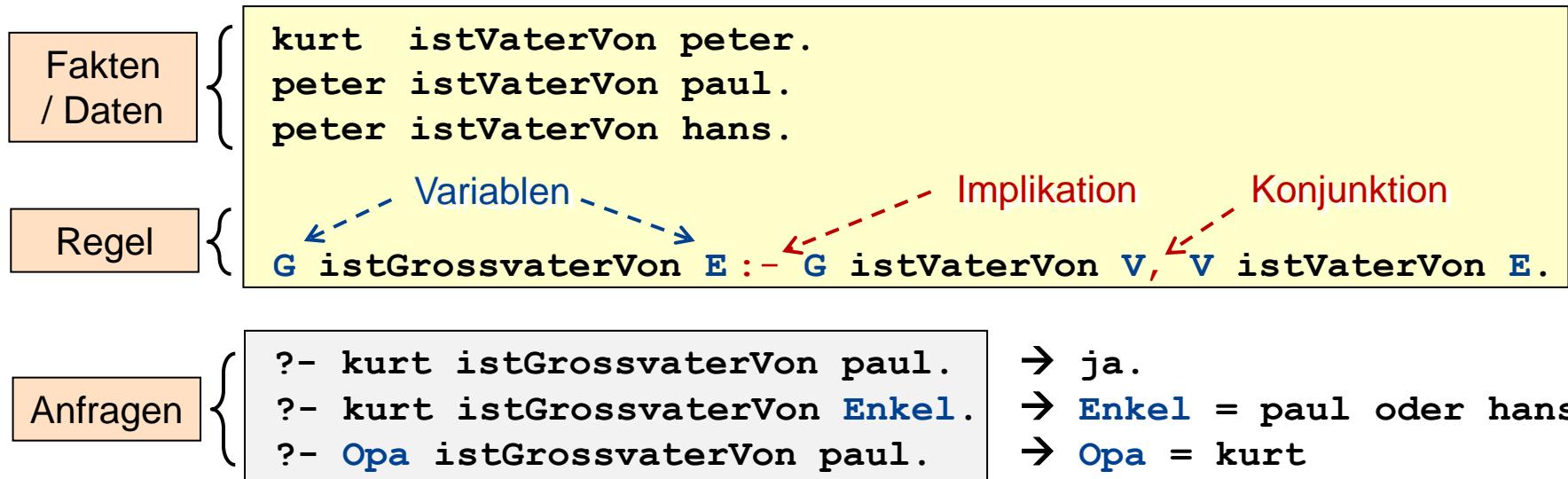
Kontrollparadigma (Programmsteuerung)

A) Implizite Kontrolle (deklarative Sprachen)

- Regelbasierte Systeme
- Logische Programmierung (Prolog)
- Datenbankabfragesprachen (SQL)

Prinzip: Sie programmieren Sachverhalte, nicht Algorithmen.

- Beispiel: Verwandschaftsbeziehungen in „Prolog“



Bestimmung des Kontrollparadigmas (Programmsteuerung)

B. Explizite Kontrolle (prozedurale und objektorientierte Sprachen)

- ◆ Zentrale Kontrolle
 - ⇒ Kontrolle befindet sich in einem Objekt / einer Komponente
- ◆ Dezentrale Kontrolle
 - ⇒ Kontrolle befindet sich in verschiedenen unabhängigen Objekten / Komponenten
 - ⇒ Evtl. Höhere Ausfallsicherheit
 - ⇒ Geschwindigkeitsgewinn durch Parallelität versus mehr Kommunikation.
- ◆ Prozedurgesteuerte Kontrolle
 - ⇒ Kontrolle befindet sich im Programmcode.
 - ⇒ Beispiel: Hauptprogramm ruft Prozeduren in Subsystemen auf.
 - ⇒ Einfach, leicht zu bauen
- ◆ Eventgesteuerte Kontrolle
 - ⇒ Kontrolle sitzt in einem Dispatcher, der Funktionen von Objekten / Komponenten durch Events aufruft.
 - ⇒ Lose Kopplung, Erweiterbarkeit

8.9.5 Zugriffsrechte

Zugriffsrechte

- Es geht darum, welcher **Akteur** auf welche **Objekte** wie zugreifen kann, d.h. zu welchen **Operation** auf den Objekten er berechtigt ist
- Zugriffskontrollmatrix
 - ◆ Zeilen = Akteure
 - ◆ Spalten = Objekte
 - ◆ Feldinhalt = zulässige Operationen

	Objekttyp 1	Objekttyp 2	Objekttyp 3	
Actor A	Op1.1, Op1.2	---	Op3.1	
Actor B	Op1.2	Op2.2, Op2.3	Op3.2	
Actor C	---	Op2.1	Op3.3	

Bsp: Actor C darf Operation Op2.1 auf Objekten des Typs Objekttyp 2 ausführen.

Realisierung der Zugriffskontrollmatrix

- Spaltenweise Aufteilung = Jedes Objekt weiß wer was damit tun darf
 - ◆ Access Control Lists (Beispiel: „Unix“)
- Zeilenweise Aufteilung = Jeder Actor besitzt ein „Ticket“ das besagt, welche Operationen er ausführen darf
 - ◆ Capabilities (Beispiel: „Amoeba“)
 - ◆ Synonyme: „Ticket“ / „Ausweis“/ „Schlüssel“

	Objekttyp 1	Objekttyp 2	Objekttyp 3	
Actor A	Op1.1, Op1.2	---	Op3.1	
Actor B	Op1.2	Op2.2, Op2.3	Op3.2	
Actor C	---	Op2.1	Op3.3	

Bsp: Actor C darf Operation Op2.1 auf Objekten des Typs Objekttyp 2 ausführen.

Fragen zu Zugriffsrechten

- Benötigt das System eine Authentifizierung?
- Wenn ja, welches Authentifizierungsschema?
 - ◆ Nutzernname und Passwort? → Zugriffskontrollliste (ACL)
 - ◆ Tickets? → Capability-based
- Welche Benutzerschnittstelle für die Authentifizierung?
- Wann und wie wird ein Dienst dem Rest des Systems bekannt gemacht?
 - ◆ Zur Laufzeit?
 - ◆ Beim Kompilieren?
 - ◆ Über einen TCP-IP-Port?
 - ◆ Durch einen Namen?
- Benötigt das System einen netzweiten „Name Server“?

8.9.6 Grenzfälle

- Die meiste Zeit beschäftigt man sich beim Systementwurf mit dem Verhalten im Betriebszustand.
- Abschließend muss man sich aber auch mit Grenzfällen befassen.
 - ◆ Initialisierung
 - ⇒ Beschreibt, wie das System aus einem nicht initialisierten Zustand in einen Betriebszustand gebracht wird ("startup use cases").
 - ◆ Terminierung
 - ⇒ Beschreibt, welche Ressourcen vor der Beendigung aufgeräumt werden und welche Systeme benachrichtigt werden ("Terminierungs-Use Cases").
 - ◆ Fehler
 - ⇒ Viele mögliche Gründe: Programmierfehler, externe Probleme (Stromversorgung).
 - ⇒ Guter Systementwurf sieht fatale Fehler voraus ("Fehler-Use Cases").

Fragen zu den Grenzfällen

- Initialisierung
 - ◆ Wie startet das System?
 - ⇒ Auf welche Daten muss beim Hochfahren zugegriffen werden?
 - ⇒ Welche Dienste müssen registriert werden?
 - ◆ Was tut die Benutzerschnittstelle beim Startvorgang?
 - ⇒ Wie präsentiert sie sich dem Nutzer?
- Terminierung
 - ◆ Dürfen einzelne Subsystemen terminieren?
 - ◆ Werden andere Subsysteme benachrichtigt, wenn ein einzelnes terminiert?
 - ◆ Wie werden lokale Updates der Datenbank mitgeteilt?
- Fehler
 - ◆ Wie verhält sich das System, wenn ein Knoten oder eine Kommunikationsverbindung ausfällt? Gibt es Backupverbindungen?
 - ◆ Wie stellt sich das System nach einem Fehler wieder her?
 - ◆ Unterscheidet dieser Vorgang sich von der Initialisierung?

Rückblick ▶ Zielgerichteter Systementwurf

Aktivitäten

- Identifikation von Nebenläufigkeit
- Hardware/Software Zuordnung
- Management persistenter Daten
- Globale Ressourcenverwaltung
- Wahl des Programmsteuerung
- Grenzfälle

Nutzen

- Jede Aktivität überprüft die gewählte Architektur in Hinsicht auf eine bestimmte Frage.
- Nach Beendigung dieser Aktivitäten können die Schnittstellen der Subsysteme **abschließend** definiert werden.
 - Start frei für den Objektentwurf der Subsysteme!