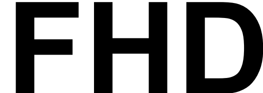


Fachhochschule Darmstadt
Fachbereich Informatik

Softwaretechnik I

Kapitel 0

- Motivation

Aussagen im Umfeld der Software-Technik

- „Programmieren klappt doch auch so!“
- „Wen interessieren schon Boxen und Linien?“
- „Ihr werdet nicht für bunte Folien mit Kästchen bezahlt!“
- „Prozesse haben noch nie ein Produkt geliefert!“
- „Warum können die SW-Chaoten nicht einfach arbeiten wie ordentliche Ingenieure?“

⇨ **Wozu braucht man Software-Technik?**

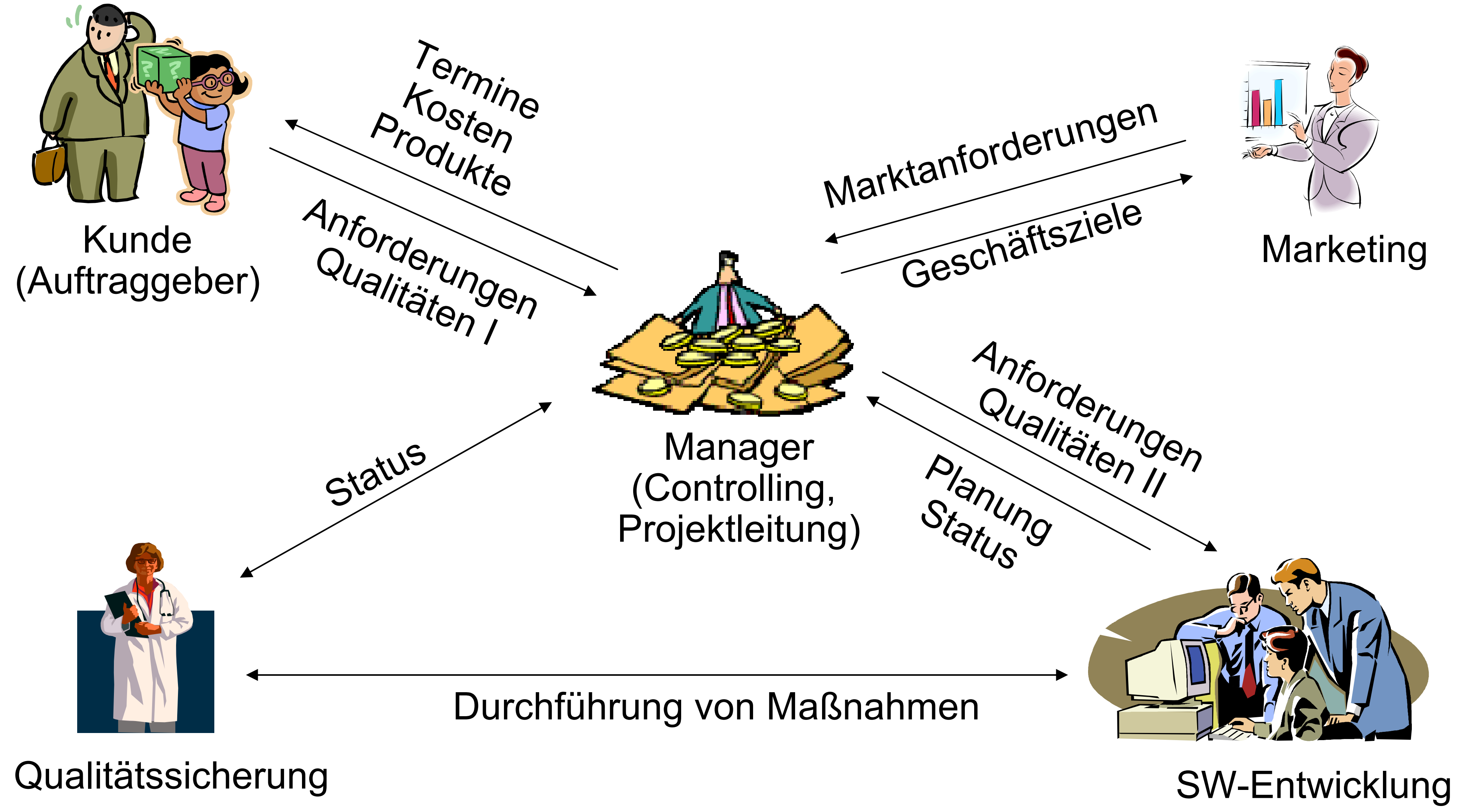
Szenario

Stellen Sie sich bitte folgendes vor:

- Sie sind verantwortlich für ein Software-Projekt im Wert von ca. 100 Mio Euro
- Ihr Gehalt, Ihre Karriere und die Zukunft Ihrer Firma hängen am Erfolg des Projektes

⇨ **Was müssen Sie wissen, damit Sie nachts noch ruhig schlafen können?**

SW-Technik im Projekt (Auszug)



Gründe für den Einsatz von Software-Technik

- Planung: Termine, Umfang, Kosten, Personal, Qualität, Reporting
- Komplexität: Strukturierung, Verbergen von Details
- Funktion: Erfassen der Wünsche von Kunde (und eigener Firma)
- Organisation: Teamgröße, Standortverteilung, unterschiedliche Sprache
- Qualität: Sicherheit, Regressanspruch, Zugänglichkeit
- Wartung: Gewährleistung, Dokumentation
- …

⇨ **Lernen, wie man <u>große Softwaresysteme</u> in guter Qualität im gegebenen Rahmen erstellt!**

⇨ **Vermittlung von Modellen, Methoden und Techniken für die Arbeit mit solchen Projekten**

Inhalt von Softwaretechnik I

<u>Inhaltsverzeichnis im Skript:</u>

1. Was ist SWT, SW-Qualität
2. Objektorientierung
3. UML
4. Vorgehens- und Prozessmodelle

SW-Technik im Projekt (Auszug)

Vorgehen
Prozesse

Termine
Kosten
Produkte

Marktanforderungen

Kunde
(Auftraggeber)

Anforderungen
Qualitäten I

Geschäftsziele

Marketing

UML

Manager
(Controlling,
Projektleitung)

Anforderungen
Qualitäten II

UML

Qualität
Prozesse

Status

Planung
Status

Durchführung von Maßnahmen

Qualitätssicherung

SW-Entwicklung

OO, UML
Prozesse

Lernziele der Gesamtveranstaltung

- Besonderheiten der Softwareentwicklung kennen und verstehen
- Zielsetzung und Inhalte von Softwaretechnik kennen und verstehen
- Überblick über Technik und Methoden der Softwareentwicklung
- Softwaretechnik als interdisziplinären Dreiklang von Mensch(en), Technik(en) und Organisation verstehen

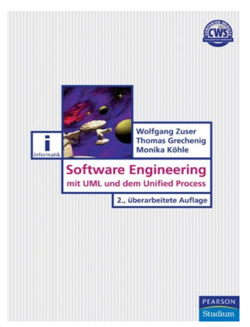
⇨ SWT als ‚Programmieren im Großen‘ ergänzt das bisherige „Programmieren im Kleinen“

⇨ Handwerkszeug für Projekte und Industrie-Praktika

Lehrbücher

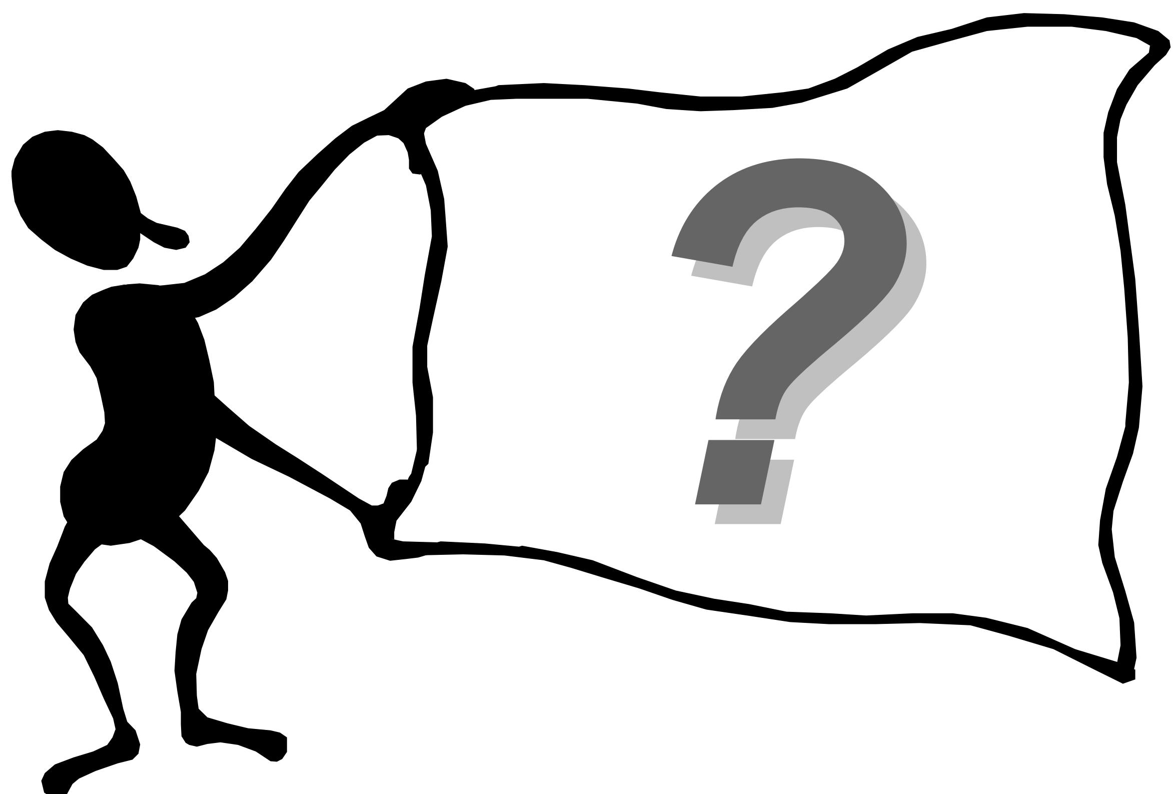
Helmut Balzert: *Lehrbuch der Software-Technik*, Bd. 1 und 2, Spektrum Verlag, Heidelberg 2000 bzw. 1998 (für Bd. 2)

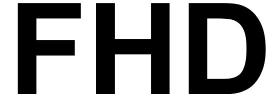
Wolfgang Zuser, Stefan Biffl, Thomas Grechenig, Monika Köhle: *Software Engineering mit UML und dem Unified Process*. Pearson Studium, München, 2. Aufl. 2003



Bernd Brügge, Allen Dutoit: *Objekt-orientierte Softwaretechnik mit UML, Entwurfsmustern und Java*. Pearson Studium, München 2004.

Fragen





Fachhochschule Darmstadt
Fachbereich Informatik

Softwaretechnik I

Kapitel 1

- Einführung Softwaretechnik

1.2 Was ist Softwaretechnik? (0)

Begriffsklärung: Software

Enges Verständnis:

Software: ... unter Software subsumiert man alle immateriellen Teile, d.h. alle auf einer Datenverarbeitungsanlage einsetzbaren Programme
(Lexikon der Informatik und Datenverarbeitung. H.-J. Schneider (Hrsg.), 1986)

Mittleres Verständnis:

Software (engl., eigtl. »weiche Ware«), Abk. SW, Sammelbezeichnung für Programme, die für den Betrieb von Rechensystemen zur Verfügung stehen, einschl. der zugehörigen Dokumentation
(Brockhaus Enzyklopädie)

Weites Verständnis = Unser Verständnis:

Software: Menge von Programmen oder Daten zusammen mit begleitenden Dokumenten, die für ihre Anwendung notwendig oder hilfreich sind
(Ein Begriffssystem für die Softwaretechnik , W. Hesse et al., 1984)

Software: Computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system
(IEEE Standard Glossary of Software Engineering Terminology, ANSI 1983).

1.2 Was ist Softwaretechnik? (1)

Umfangreiche SW-Projekte sind gekennzeichnet durch

- Erstellung von mehrere Versionen. (Software wird weiterentwickelt und erweitert.)
- Lange Entwicklungszeit. (=> Projektplanung und Terminüberwachung).
- Einsatz eines Entwicklerteams (evtl. an mehreren Standorten)
- Lange Lebensdauer, während der das Produkt Software an neue Anforderungen angepasst werden muss.

⇒ Wir benötigen andere Vorgehensweisen als für das Programmieren einer kleinen Anwendung.

⇒ Die Methode VHIT (Vom Hirn Ins Terminal) ist zur Entwicklung komplexer Systeme nicht geeignet.

⇒ **Wir benötigen Prinzipien und Methoden des Software Engineering!**

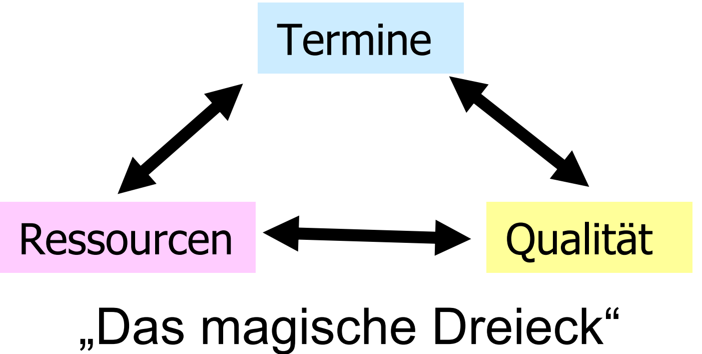
1.2 Was ist Softwaretechnik? (2)

Definition Software-Engineering:

Zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden und Werkzeugen für die arbeitsteilige, ingenieurmäßige Entwicklung und Anwendung von umfangreichen Software-Systemen.

(in Anlehnung an Balzert, SWT-Lehrbuch)

1.2 Was ist Softwaretechnik? (3)



Lernen, wie man <u>große Softwaresysteme</u>
in der erforderlichen Qualität
im gegebenen Zeit- und Ressourcenrahmen erstellt

d. h. Vermittlung von Modellen, Methoden und Techniken für:

- die konstruktiven Phasen der Software Entwicklung (Analyse, Entwurf, Implementierung), die verwendet werden um die Arbeitsergebnisse dieser Phasen zu strukturieren.
- Definition von Qualität (Qualitätseigenschaften, Qualitätsmodelle).
- zeitlichen Strukturierung in verschiedene Entwicklungsphasen (Phasenmodelle) .
- Überprüfung der Phasenergebnisse (Test, Verifikation, Qualitätssicherung).

1.3 Geschichtliche Entwicklung (1)

In der „Steinzeit“ der Software Entwicklung, in den 50er und 60er Jahren:

- Hardware war sehr teuer und wenig leistungsfähig

Wo waren Computer eingesetzt,
welche Tätigkeiten konnte man damals durch den Computer unterstützen?

- Software und Computer fast ausschließlich im naturwissenschaftlichen Bereich.
- Die eingesetzten Programme dienten hauptsächlich der Lösung mathematischer Probleme.
- Die relativ kleinen Programme waren sehr preisgünstig oder wurde von den Anwendern selbst geschrieben.

1.3 Geschichtliche Entwicklung (2)

Durch sinkende Hardwarepreise:

- Computer wurden für eine größere Zielgruppe erschwinglich.

⇒ **Es veränderten sich die Anforderungen an die Software:**

  - Software wurde für die verschiedensten Bereiche gebraucht.
  - Der Umfang der Software wurde immer größer.
  - Ihre Komplexität war mit den bis zu diesem Zeitpunkt bekannten Vorgehensweisen für das „Programmieren im Kleinen“ nicht mehr zu beherrschen.

  ⇒

  - **fehlerhafte** und **unzuverlässige** Programme
  - **nicht termingerechte** Fertigstellung
  - **geplante Kosten** wurden überschritten

Dieser Zustand der Software Industrie wurde charakterisiert mit dem Begriff
„Softwarekrise“

1.3 Geschichtliche Entwicklung (3)

Reaktion darauf:

- strukturierte Programmiersprachen
- Entwicklungsmethoden

Der Begriff „***Software Engineering***“ wurde zuerst auf einer Software-Engineering-Konferenz der NATO 1968 in Garmisch geprägt. Diese Konferenz gilt als Geburtsstunde der Softwaretechnik.

1.3 Geschichtliche Entwicklung (4)

1. **Erster Ansatz zur Entwicklung übersichtlicher Programme (1968)**

"strukturierte Programmierung„ -> Vermeidung von GOTO-Anweisungen etc. (Dijkstra)

2. **Entwicklung von Software-Engineering-Prinzipien (1968 - 1974)**

Strukturiertere Entwicklung von Programmen:

- strukturierte Programmierung,
- schrittweise Verfeinerung,
- Geheimnis-Prinzip,
- Programmmodularisierung,
- Software-Lifecycle,
- Entitiy-Relationship-Modell,
- Software-Ergonomie

Frage: Wissen Sie, was sich hinter den Begriffen verbirgt?

1.3 Geschichtliche Entwicklung (5)

3. **Entwicklung von phasenspezifischen Software-Engineering-Methoden (1972 - 1975)**

Umsetzen der Software-Engineering-Prinzipien in Entwurfsmethoden:
- HIPO
- Jackson
- Constantine-Methode (Structured Design)
- erste Version von Smalltalk

4. **Entwicklung von phasenspezifischen Werkzeugen (1975 - 1985)**

Einsatz von SE-Methoden mit maschineller Unterstützung
z.B.
- Programm-Inversion
- Batchwerkzeuge

1.3 Geschichtliche Entwicklung (6)

5. **Entwicklung von phasenübergreifenden (integrierten) Software-Engineering-Methoden (ab 1980)**

   Automatische Weitergabe der Ergebnisse einer Phase des Software-Lifecycles an die nächste Phase: **Methodenverbund**

6. **Entwicklung von phasenübergreifenden (integrierten) Werkzeuge (ab 1980)**

   Einsatz einer Datenbank als automatischer Schnittstelle zwischen den einzelnen Phasen des Software-Lifecycles.

1.3 Geschichtliche Entwicklung (7)

7. Definition verschiedener, konkurrierender objektorientierter Methoden (ab 1990)

Es entstanden parallel verschiedene Objektorientierte Analyse- und Entwurfsmethoden:

- Booch
- Jacobson
- Rumbaugh
- Shlaer/Mellor
- Coad/Yourdon u. a.

Die Methoden wurden in CASE Tools realisiert.

8. Integration der OO-Methoden zur UML - Unified Modeling Language (ab 1995)

Jacobson, Booch und Rumbaugh schließen sich zusammen und entwickeln die UML.
In der UML sollen

- die Schwächen der frühen OO-Methoden beseitigt und
- ein weltweit gültiger, einheitlicher Standard geschaffen werden.

1.1 Ziel der Softwaretechnik:

Erstellung qualitativ hochwertiger SW-Systeme

Frage an Sie:

- Was ist ein „*qualitativ hochwertiges*“ SW-System?

1.1 Was ist ein qualitativ hochwertiges Softwaresystem?

Es muss

- **nützlich und nutzbar sein.**
  Es muss dem Anwender das Leben möglichst stark erleichtern.
- **zuverlässig sein.**
  Es soll möglichst wenige Fehler enthalten.
- **kostengünstig sein,**
  nicht nur in der Anschaffung, sondern auch im Unterhalt.
- **verfügbar sein.**
  Auf jetzigen und zukünftigen Zielplattformen (Hardware, Betriebssystemen etc.) lauffähig / leicht adaptierbar
  Softwareprodukt existiert zu dem Zeitpunkt, zu dem es zum Einsatz kommen soll.
- **flexibel sein.**
  Das System muss leicht an geänderte Anforderungen des Benutzers anpassbar sein.
  Die Fehler müssen leicht zu beheben sein.
- …

Ist das automatisch auch ein „*gutes*“ SW-System?

1.1 Was ist ein gutes Softwaresystem?

z.B. aus Sicht der Entwicklungsfirma:

Es muss

- **Gewinn bringen**
  wenig Kosten verursachen, viel Geld einbringen
- **sich „verkaufen“**
  Es muss den Markt ansprechen.
- **zuverlässig sein.**
  Es soll keine teure Wartung erfordern
- **kostengünstig sein,**
  geringe Entwicklungs- und Wartungskosten

⇨ **„gut“ hängt stark vom Betrachter ab**

⇨ **nicht jedes gute Softwaresystem muss qualitativ hochwertig sein!**

⇨ **nicht jedes qualitativ hochwertige System ist gut!**

1.4 Softwarequalität

- Die Qualität, die ein Produkt hat, wird häufig in direkter Verbindung mit den Fehlern gesehen.
- Es gibt allerdings noch viele andere Qualitäts*eigenschaften*, welche die Qualität eines Produkts bestimmen.
- Die Zuverlässigkeit ist nur eine dieser Qualitätseigenschaften.

Der Begriff „Qualität“ wird umgangssprachlich für beide Aspekte verwendet!

1.5 Softwarequalitätseigenschaften

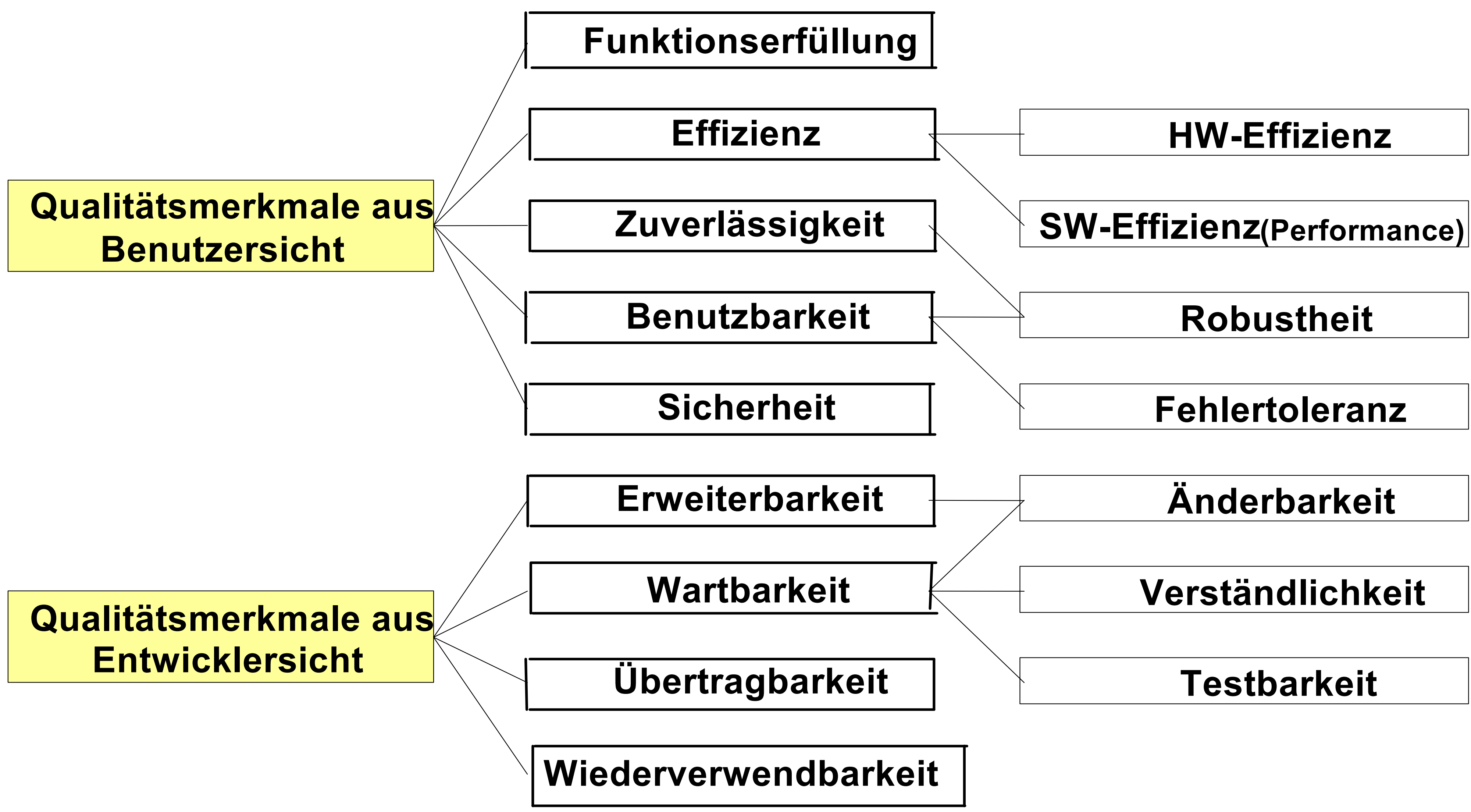
Welche Eigenschaften

machen für SIE die Qualität einer Software aus?

⇨ aus Entwicklersicht?

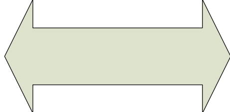
⇨ aus Benutzersicht?

1.5 Softwarequalitätseigenschaften



1.5 Softwarequalitätseigenschaften (Beispiel)

Funktionserfüllung

geplantes System  tatsächlich realisierter Funktionsumfang

Effizienz

- Hohe Ablaufgeschwindigkeit (Performanz)
- wenig HW-Ressourcenverbrauch (z. B. geringer Speicherplatzbedarf)
  - HW wird immer billiger ⇒ verliert immer mehr an Bedeutung
  - Aber: immer noch wichtig bei hohen Stückzahlen z.B. bei embedded Systems
- Schnelle Algorithmen

1.5 Softwarequalitätseigenschaften: Bedeutung

- Die Umsetzung von Qualitätseigenschaften verursacht erhebliche Aufwände
- nachträgliche Änderungen von nicht eingeplanten Qualitätseigenschaften sind in der Regel sehr aufwändig
- Die Festlegung der geforderten Qualitätseigenschaften ist genauso wichtig wie die Festlegung der Funktionalität
- Qualitätseigenschaften sind nicht unabhängig voneinander.
- Werden Maßnahmen zur Verbesserung einer Qualitätseigenschaft getroffen, so wirken sich diese möglicherweise auf andere Qualitätseigenschaften negativ aus.

Beispiel:

Wird die Effizienz verbessert, so kann dies ein unzuverlässigeres System bewirken; die Robustheit kann sinken, das System kann schwerer änderbar sein (weil z. B. Effizienzsteigerung durch Assemblerprogramme erreicht wurde).
Hardwarenahe Programmiersprachen wirken sich natürlich auch auf die Wartbarkeit und Übertragbarkeit aus.

1.5 Maße für die Qualität(seigenschaften)

schon bei der Spezifikation muss die Qualität der zu erstellenden Software definiert werden

⇒ Es kann nach der Fertigstellung nachgeprüft werden, ob die Software den spezifizierten Qualitätsanforderungen genügt.

Funktionserfüllung:
Test: funktioniert die Software so wie vereinbart?
d.h. tut die Software das, was in der vom Auftraggeber abgezeichneten Spezifikation steht?

Effizienz:
spezifiziert durch Angabe von maximalen Antwortzeiten, Speicherverbrauch.

Zuverlässigkeit:
Spezifiziert durch Wahrscheinlichkeit, dass in der Zeit 0-t kein Fehler auftritt.

Andere Maße, wie die z. B. **Erweiterbarkeit** oder **Wartbarkeit:**
sind schwieriger zu definieren ⇒ Softwaremetriken

1.6 Phasen der Softwareentwicklung

Durch den Einsatz von Phasenmodellen wird ein Softwareprojekt zeitlich strukturiert.
Es wird festgelegt:

- Welche Arbeiten sind in welchen Phasen zu erledigen?
- Welche Arbeitsergebnisse müssen in den Phasen erzeugt werden?
- Welche Personen haben die Ergebnisse zu erarbeiten?
- Wie sollen die Ergebnisse dokumentiert werden?
- Welche Termine müssen eingehalten werden?
- Welche Standards und Richtlinien sind einzuhalten?

Phasenmodelle regeln auch die Zusammenarbeit zwischen den am Softwareerstellungsprozess Beteiligten.

Es existieren verschiedene Phasenmodelle für die Softwareentwicklung.

Wasserfallmodell

= „Ur-Modell“

Rein sequentielles Durchlaufen der Phasen.

Arbeiten einer Phase müssen abgeschlossen sein, bevor die Folgephase beginnen kann.

Dieses Vorgehen erwies sich in der Praxis als undurchführbar.
⇒ Modell wurde so erweitert, dass Rücksprünge in zeitlich früher liegende Phasen möglich wurden.

Am Ende jeder Phase werden die Ergebnisse überprüft und an die Folgephase weitergegeben. Wird dort während der Erarbeitung der eigenen Phasenergebnisse, ein Fehler in den Vorgaben festgestellt, so werden die Vorgaben an die vorhergehende Phase zurückgeführt.

andere Phasenmodelle

In den letzten Jahren:
Vielzahl von weiteren Prozessmodellen, die dieses Modell variieren.

z. B.

- durch mehrmaliges Durchlaufen der Phasen des Wasserfallmodells in mehreren Zyklen, wobei am Ende jedes Zyklus eine weitere Ausbaustufe des Systems entsteht

- oder/und durch Integration von Prototyping-Ansätzen in das Phasenmodell.

Unabhängig vom Phasenmodell:
wir durchlaufen während des Softwareentwicklungsprozesses immer verschiedene aufeinander aufbauende Phasen:

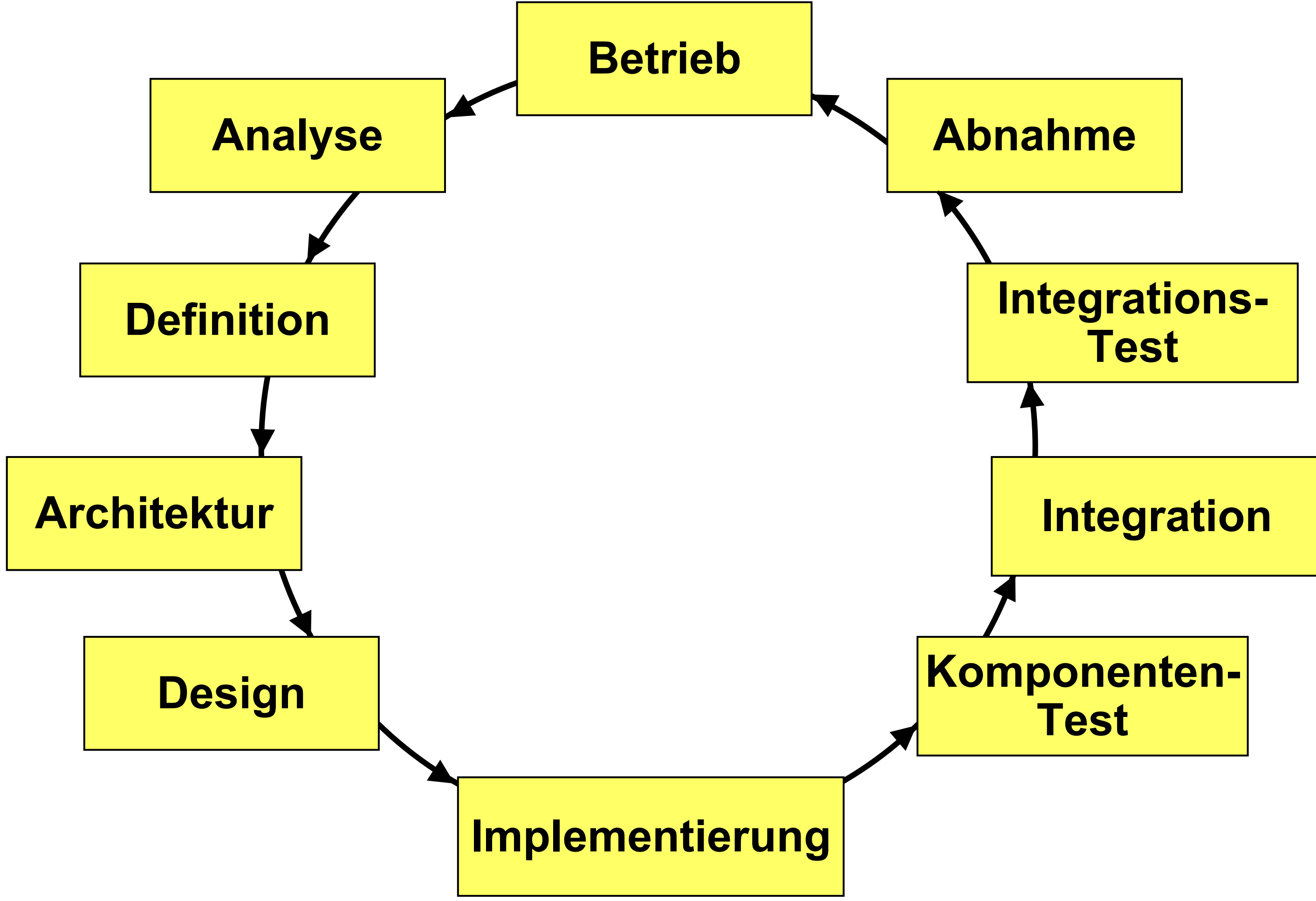
Phasen

Was denken Sie

- Welche Phasen gibt es im SW-Erstellungsprozess?

- Welche Phasen kennen Sie schon?

Phasen – Der Software – Life Cycle



Detaillierung der einzelnen Phasen und des Modells hängen vom Bedarf ab:

- **Zusammenlegung „uninteressanter“ Phasen**
- **Verfeinerung „interessanter“ Phasen**

Phasen: Implementierung, Integration und Test

Implementierung und Test kennen wir schon aus der Lehrveranstaltung Programmieren. (Zumindest kleine Programme wurden von Ihnen bereits implementiert und getestet.)

Bei großen Systemen:

- wir können das System nicht als Ganzes implementieren und testen.
- wir implementieren und testen zuerst die einzelnen Komponenten (**Komponententest**).
- Anschließend: werden diese ausgetesteten Komponenten schrittweise zusammenmontiert (**Integration**) und ausgetestet (**Integrationstest**).

Vor Beginn der eigentlichen Codierung:

- Die Implementierungsstrategie ist festzulegen.
- Es wird entschieden, in welcher Reihenfolge die Komponenten implementiert und zusammengebaut werden (Top-Down- oder Bottom-Up-Vorgehensweise).
- Die einzusetzenden Bibliotheken sind festzulegen.
- Detaillierte Testpläne für die einzelnen Komponenten, für die Teilsysteme und das Gesamtsystem sind, falls nicht schon in vorhergehenden Phasen geschehen, zu erstellen.

Phasen: Abnahme

Kennen wir auch vom Programmierpraktikum.

Es wird geprüft:

- Stimmen die Leistungen des Systems mit den in der Spezifikation vereinbarten Leistungen überein?
  (im Programmierpraktikum: mit den in der Aufgabenstellung geforderten Leistungen).

- Ist ein externer Auftraggeber vorhanden: Dieser überprüft die Systemfunktionen mit einem eigenen Prüfkatalog.

Im Programmierpraktikum: Sie bekamen Ihre Abnahme - fertig -.

Bei der kommerziellen Softwareentwicklung:

Die Software soll danach zum Einsatz kommen.
⇒ Es kommt eine weitere Phase: der eigentliche Einsatz, d.h. der **Betrieb** des Systems.

Phasen: Betrieb

Betriebsphase = längste aller Phasen.

Oft sehr hohe Kosten

d. h. der Aufwand für die Erstellung

- übersichtlich entworfener und
- gut getesteter Programme
- mit guter Dokumentation

rentiert sich

⇨ Fehler und Unzulänglichkeiten am Produkt werden hier bereinigt.

⇨ SW wird an geänderte Einsatzzwecke und betriebliche Abläufe angepasst.

⇨ Oft ist das Zurückgehen über mehrere Phasen hinweg notwendig.

Weitere Phasen bei der Softwareentwicklung

Im Praktikum Programmieren wurde immer eine Aufgabe gestellt,
d.h. die Definition war vorgegeben.

Sie mussten sich nur wenig Gedanken über den Aufbau des Systems machen, und los ging es mit der Implementierung.

z.B. statt graphischer Darstellung der Klassenstruktur
⇨ direktes Hinschreiben als Header-File

Bei Entwicklung großer Systeme:
vor der Implementierung:

⇨ **Analyse**:
Soll ein System erstellt werden?
Ist eine Erstellung unter Kostengesichtspunkten ratsam und möglich?

⇨ **Definition**:
Spezifikation des Funktionsumfang gemeinsam mit Benutzer

⇨ **Architektur und Entwurf**:
Entwurf der IT-Lösung

Phasen: Analyse

Systemanalytiker muss

- Anwendungsgebiet verstehen, Ist-Ablauf aufnehmen,
- begreifen was der Kunde will,
- die Schwachstellen analysieren,
- Verbesserungsvorschläge prüfen,
- zusätzliche Wünsche berücksichtigen.

⇨ **Ableitung was die Software leisten soll**

⇨ Die fachlichen Anforderungen an die Software werden dokumentiert und mit dem Benutzer abgestimmt.
⇨ Es wird analysiert ob die Aufgabe durchführbar (lösbar) ist,
⇨ ob sie finanzierbar ist,
⇨ ob das notwendige Personal vorhanden ist.
⇨ Es wird eine erste Kostenkalkulation vorgenommen.

Fehler in der Analysephase

- sind sehr teuer,
- wirken sich auf alle folgenden Phasen aus.

Phasen: Definition

informelle Anforderungen der Analyse ⇨ vollständige, konsistente Spezifikation

- Beschreibt wie das Produkt aus Benutzersicht funktionieren soll.
- Soll normalerweise implementierungsunabhängig formuliert sein.
- Qualitätseigenschaften werden festgelegt.
- Der Abnahmetest wird mit dem Benutzer festgelegt.

„**Was**“ das System tun soll

Phasen: Entwurf

Entwurf der IT-Lösung

- welche Hardware,
- welches Betriebssystem,
- welche Programmiersprache.

Bei komplexen Systemen:

Unterteilung in

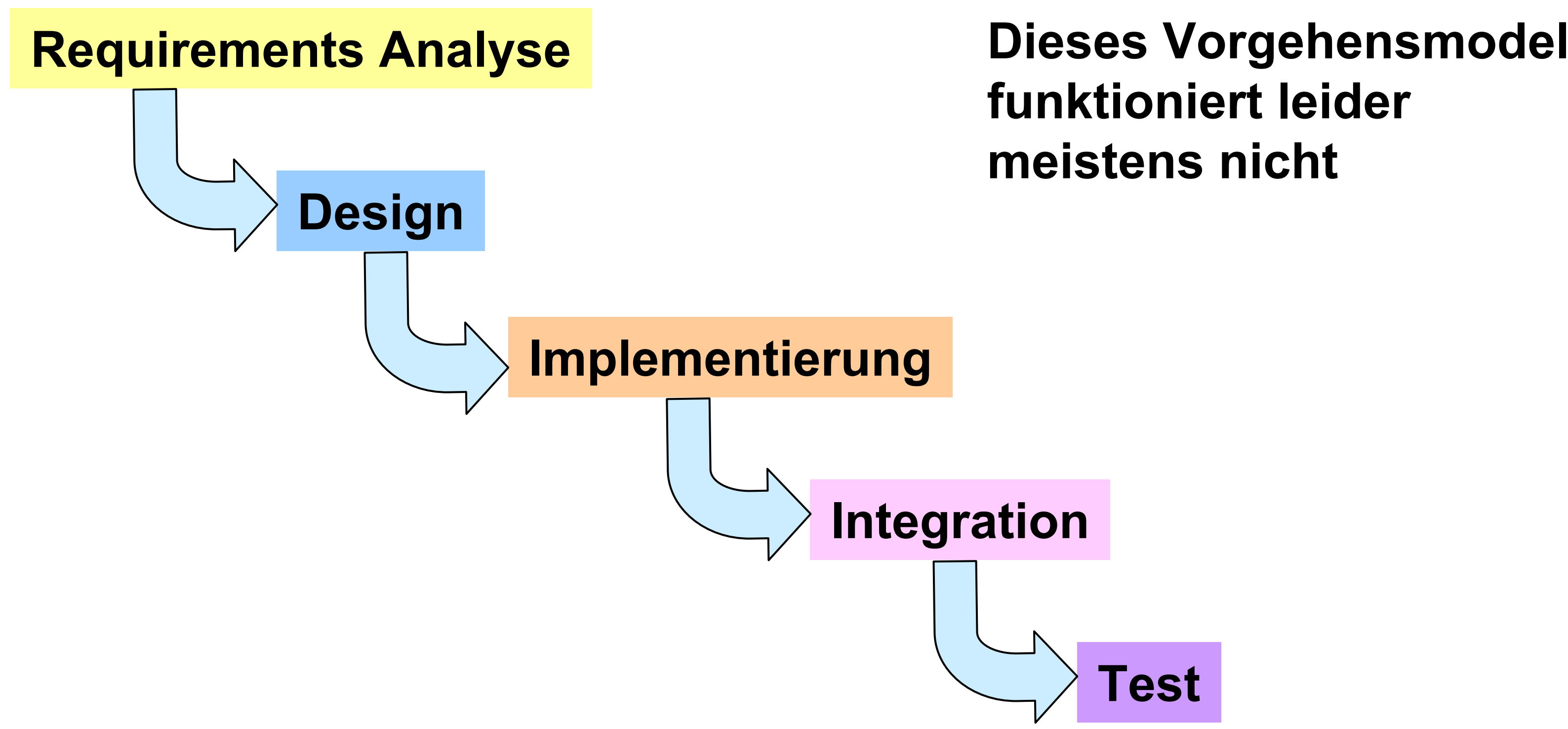
- Grobentwurf (**Architektur**) und
- Feinentwurf (**Design**)

Festlegung

- Benutzerschnittstelle,
- Datenbank,
- notwendige Systemschnittstellen zu Fremdsystemen,
- Komponentenarchitektur,
  (System wird in Komponenten (Klassen, Funktionen) zerlegt, die in Komponenten-architekturdiagrammen (Klassendiagramm, Sequenzdiagramm) dargestellt werden.)
- Komponententestpläne

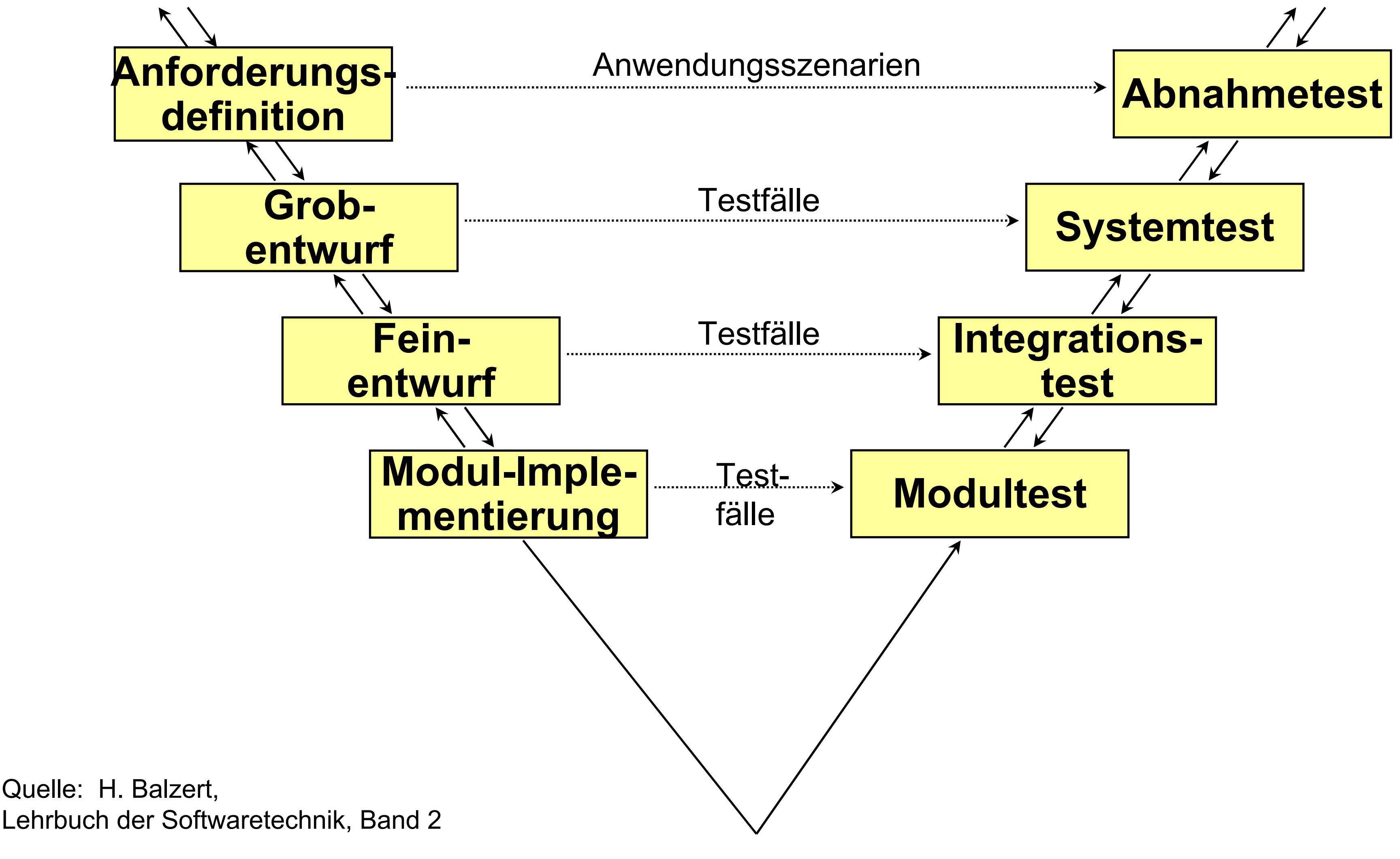
„**Wie**" das System aufgebaut ist

Phasenmodell: Wasserfall



Dieses Vorgehensmodell funktioniert leider meistens nicht

Phasenmodell: V-Modell



Quelle: H. Balzert,
Lehrbuch der Softwaretechnik, Band 2

Ergebnisse der Phasen, Reviews

Als Ergebnis jeder Phase entstehen Dokumente,
die als Grundlage für das Weiterarbeiten dienen.

In Reviews werden diese Ergebnisse geprüft mit den Erstellern diskutiert
(und evtl. auch innerhalb der Phasen an so genannten Meilensteinen anfallende Zwischenergebnisse).

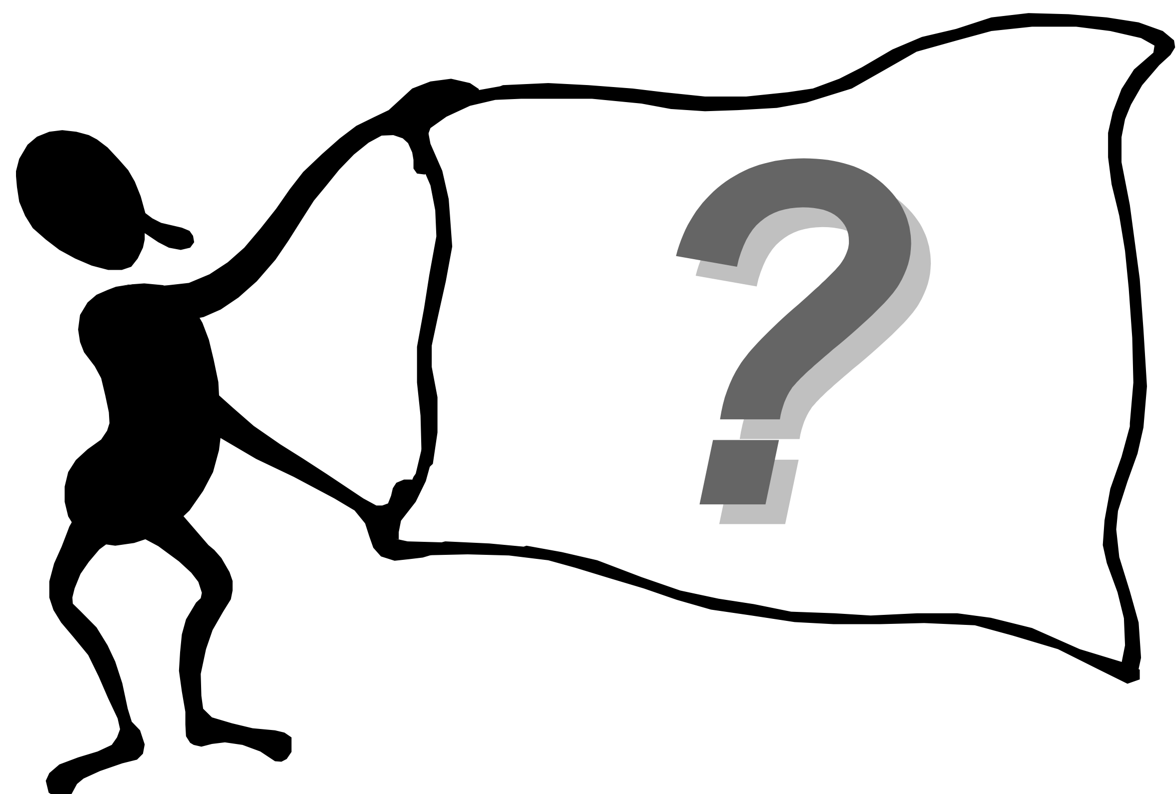
Reviews sind sehr wichtig, da
Fehler in frühen Phasen sehr hohe Kosten in späteren Phasen verursachen

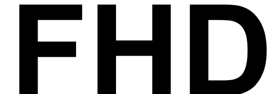
z. B. Fehler oder missverständliche Formulierungen in der Spezifikation erst zum Zeitpunkt der Abnahme festgestellt werden

⇒ alle Phasen müssen nochmals durchlaufen werden,
⇒ man muss sich in den Aufbau des Programms hineindenken und Änderungen im Code vornehmen.
⇒ System ist neu zu testen (Komponenten- und Integrationstest des gesamten Systems)
⇒ Dokumentation ist anzupassen

Eine ausführliche Betrachtung von Phasenmodellen ist in einem gesonderten Kapitel weiter hinten im Skript durchgeführt.

Fragen





Fachhochschule Darmstadt
Fachbereich Informatik

Softwaretechnik I

Kapitel 2

- Grundkonzepte der objektorientierten SW-Entwicklung

Quellenhinweis:
Teile dieser Präsentation enthalten Inhalte und Grafiken des Lehrbuchs der Software-Technik (Band 1), 2. Auflage von Helmut Balzert, Spektrum Akademischer Verlag, Heidelberg 2001

Objektorientierte Programmierung (OOP)

⇨ seit den 90-ern das zentrale Programmierparadigma

- Objekte
- Klassen
- Attribute
- Operationen / Methoden

„Erfinder" der objektorientierten SW-Entwicklung

- Prof. Dr. e. h. Kristen Nygaard
  (1926 – 2002), Oslo, Norwegen
  Department of Informatics
  University of Oslo

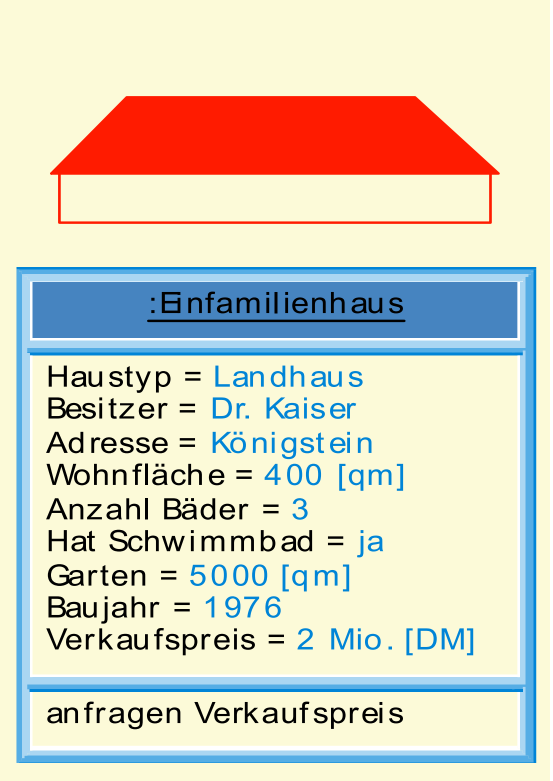
- Erfinder der Programmiersprache SIMULA 67,
  die das Klassenkonzept in die Programmiersprachenwelt einführte (zusammen mit Ole-Johan Dahl)

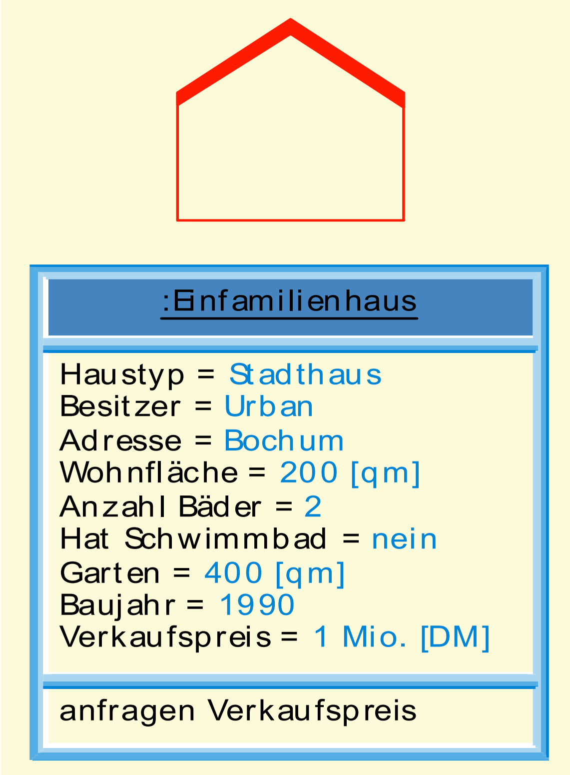
- Erfinder der objektorientierten Programmiersprache BETA
  (zusammen mit B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen).

Objekte

- Haben Attribute
- Bieten Operationen an

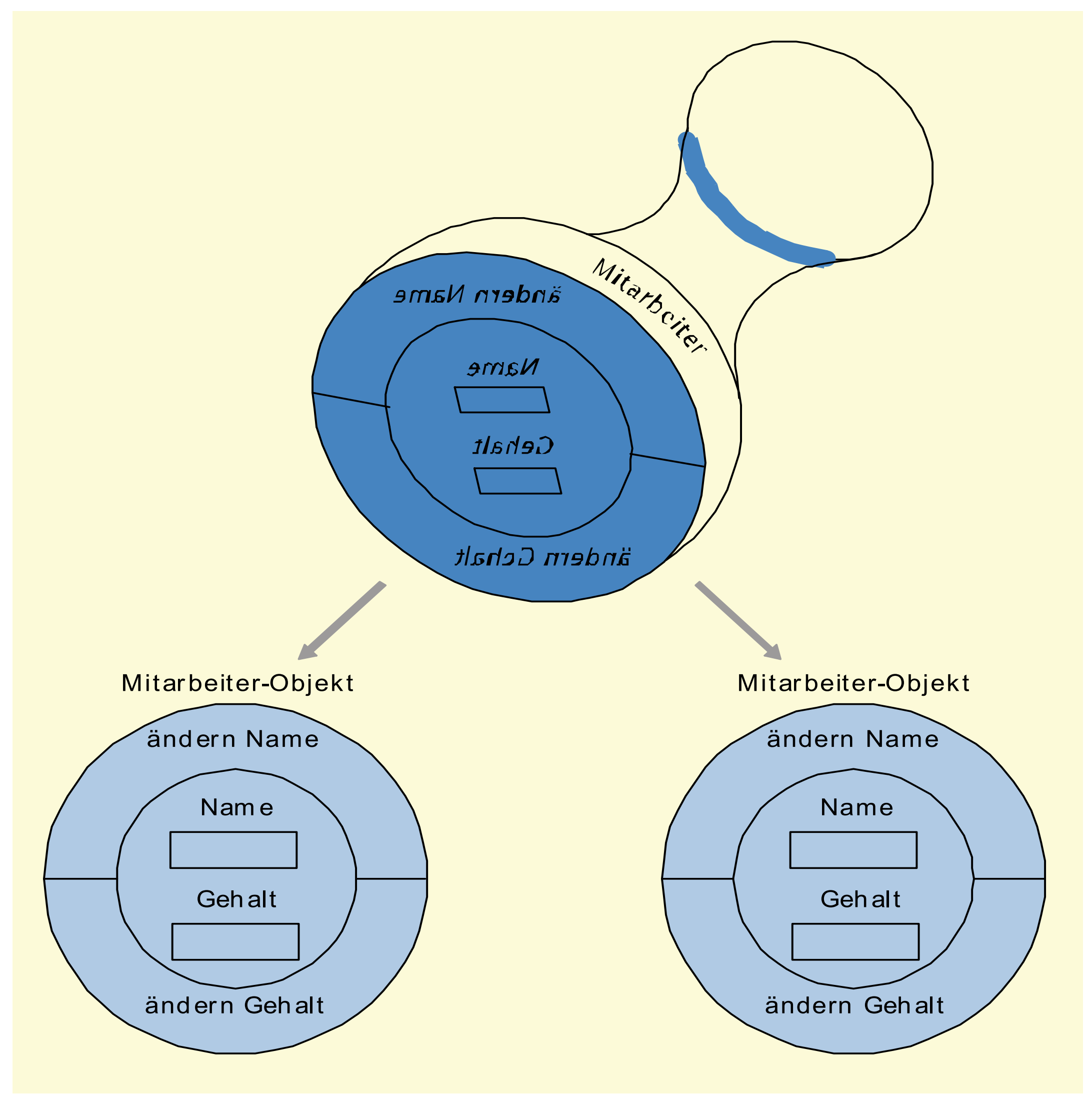
Einführungsbeispiel: Einfamilienhaus-Objekte





Von Objekten zu Klassen ...

Klasse als Schablone



Abstraktion

⇨ Eine Klasse definiert die (gemeinsamen) ...

- Attribute ihrer Objekte
- Operationen ihrer Objekte

⇨ Eine Klasse trennt Konzept und Umsetzung („Abstraktion“)

⇨ Objekte werden oft auch „Instanzen der Klasse“ genannt!

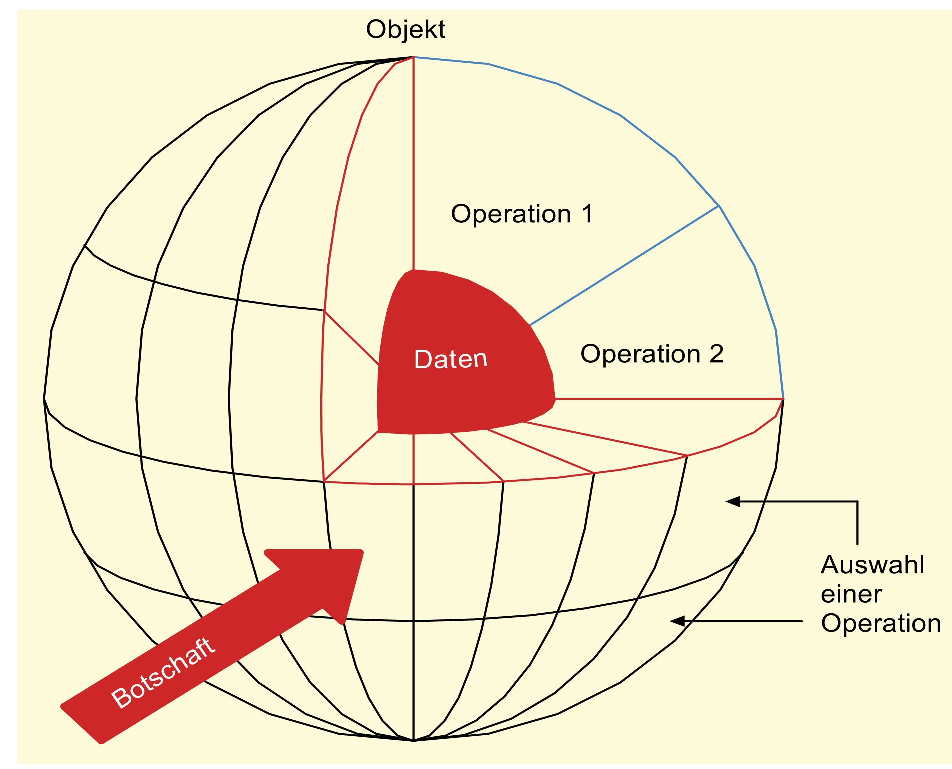
⇨ Je nach Anwendungen sind unterschiedliche Abstraktionen sinnvoll.

Kapselung

⇨ Eine Klasse „kapselt“ Attribute und Operationen

⇨ Zugriff auf die Daten erfolgt nur über die Operationen

⇨ Interne Realisierung ist von außen nicht sichtbar („Geheimnisprinzip“)



Wiederverwendung

- ⇨ Klassen können in verschiedenen Kontexten eingesetzt werden
- ⇨ Klassen verbergen die Realisierung und bieten zum Objekt passende Schnittstellen
- ⇨ Klassen fördern die (systematische) Wiederverwendung

Beziehungen (Assoziationen)

Objekte (bzw. Klassen)

- ⇨ stehen oft untereinander in Beziehung
- ⇨ bestehen oft aus mehreren anderen Objekten (Aggregation)
- ⇨ Konkretisieren die Eigenschaften einer anderen Klasse („Vererbung“ - „Spezialisierung“; invers: „Generalisierung“)
- ⇨ können als Hierarchie dargestellt werden („Vererbungshierarchie“)

Polymorphismus

„Vielgestaltigkeit“

⇨ gleiche Methoden anwendbar auf unterschiedliche Objekte (innerhalb einer Vererbungshierarchie)

⇨ Vermeidung von typbasierten Fallunterscheidungen bzw. Casts

⇨ konkrete Umsetzung erfolgt über Compiler bzw. Linker

⇨ erleichtern die Benutzung von Klassen

Prinzipien der Objektorientierung: Zusammenfassung

⇨ Kenntnisse werden vorausgesetzt über

- Objekte
- Klassen
- Attribute
- Operationen / Methoden
- Abstraktion
- Kapselung
- Vererbung
- Polymorphismus
- …

2.3 Überblick Objektorientierte Programmiersprachen

C++

- Bjarne Stroustrup (198?)
- Ziel: Erweiterung von C um Objektorientierung
- Viele Bibliotheken und Klassen verfügbar
- Pointer
- Speicherplatzverwaltung in Selbst-Regie

$\Rightarrow$ Performanz statt Entwicklungskomfort

2.3 Überblick Objektorientierte Programmiersprachen

Java

- SUN Microsystems (1995)
- Ziel: Plattformunabhängigkeit (Java Virtual Machine)
- Keine (fehlerträchtigen) Pointer
- Komponententechnik integriert
- Viele Bibliotheken und Klassen verfügbar
- Einsatz im Web-Umfeld
- Automatische Speicherverwaltung
- Rechnerleistung wird vorausgesetzt

$\Rightarrow$ Entwicklungskomfort statt Performanz

2.3 Überblick Objektorientierte Programmiersprachen

C#

- Microsoft (2001?)
- Ziel: Kombination der Vorteile von C++ & Java
- keine Header, keine Makros
- Compilierung des Codes in eine Zwischensprache
- Ausführung in der proprietären .NET-Laufzeitumgebung

$\Rightarrow$ Entwicklungskomfort und Performanz (?)

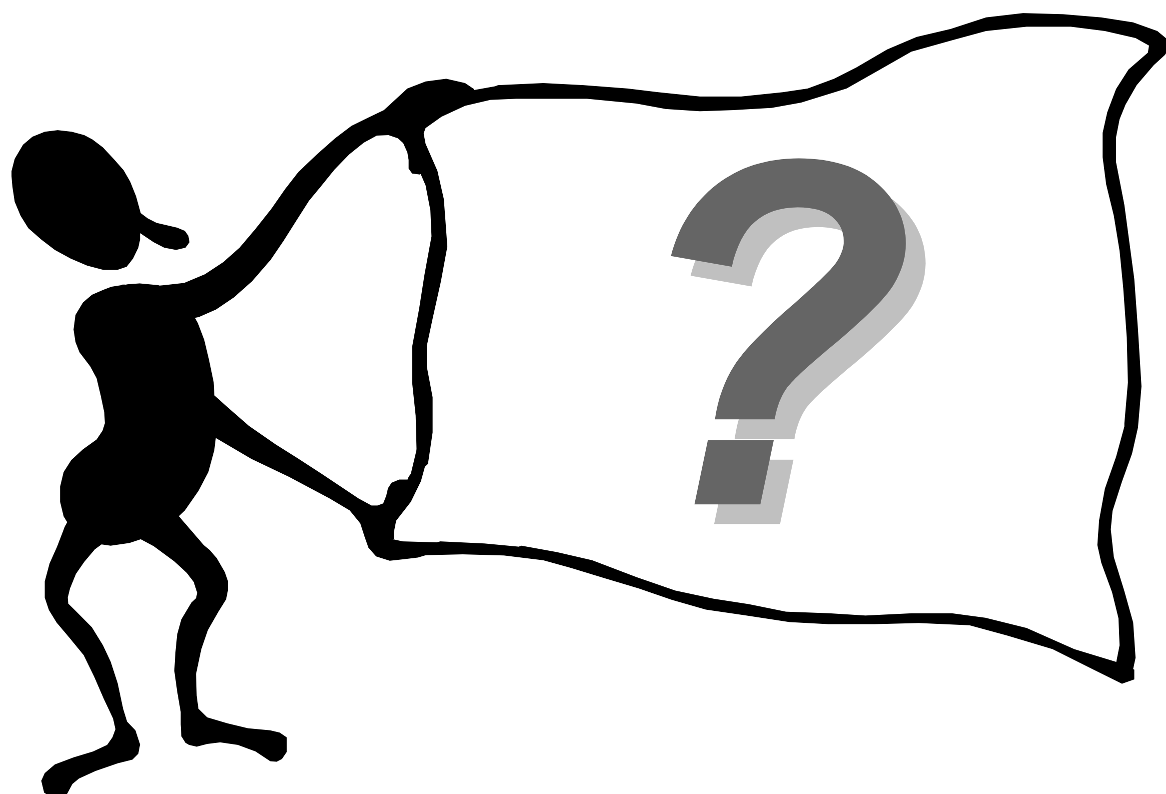
2.4 Werkzeuge für die objektorientierte Entwicklung

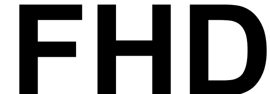
- lange Zeit nur Werkzeuge für spezifische Teile des Enwicklungszyklus‘ verfügbar
  - Editoren
  - Programmierumgebungen
  - Analyse
  - Design

meist proprietär!
kein echter Im- / Export

- Heute zunehmend „CASE-Tools“, die den gesamten SW-Lebenszyklus unterstützen
  - z.B. Innovator, Eclipse, Rational Rose, Visual Studio .Net …
  - Generieren Modell aus Code („Reverse Engineering“)
  - Generieren Code(fragmente) aus Modell („Forward Engineering“)
    - Zusammen „Round-Trip-Engineering“
  - Unterstützen die UML

Fragen





Fachhochschule Darmstadt
Fachbereich Informatik

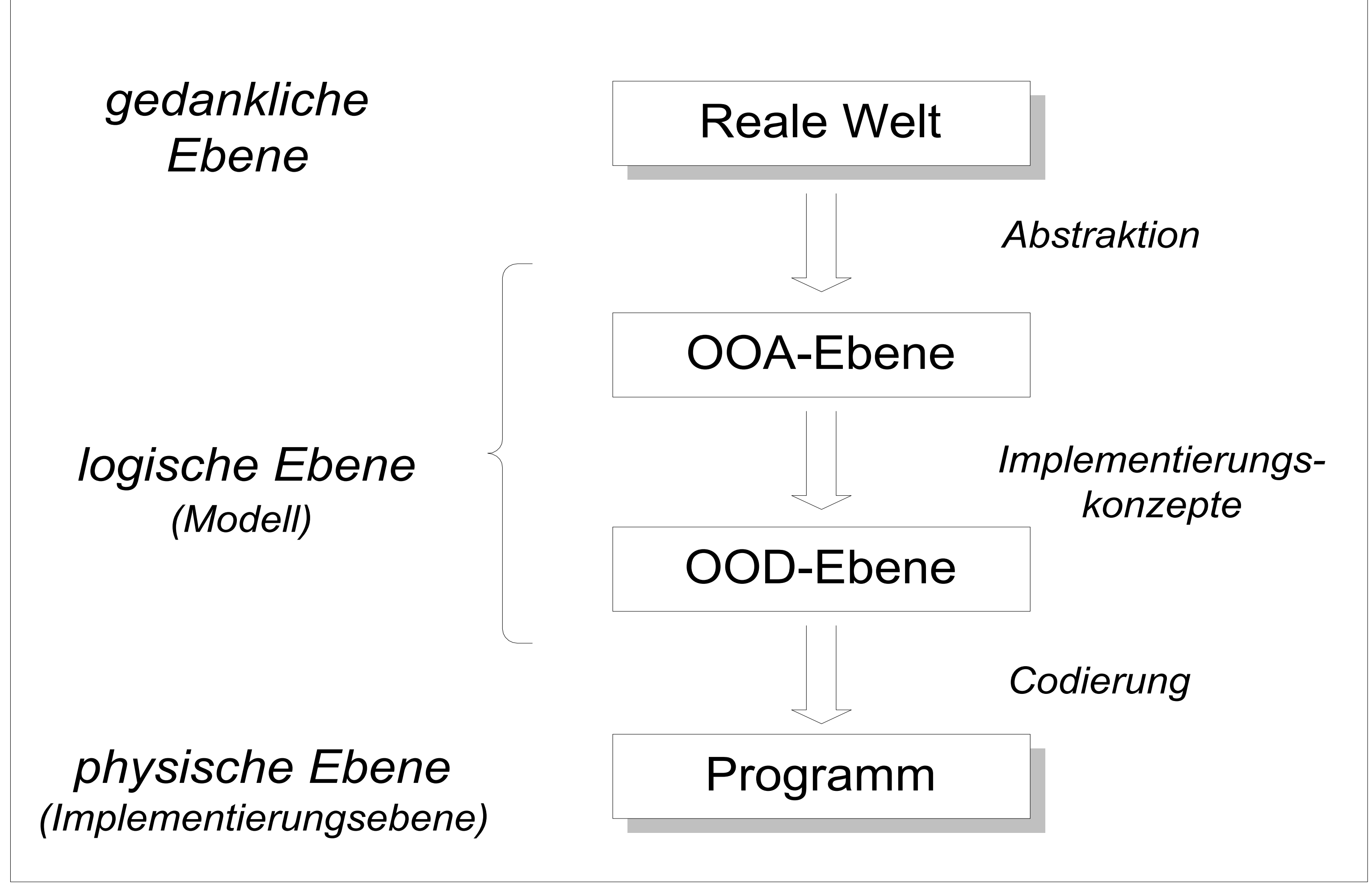
Softwaretechnik I

Kapitel 3

- Objektorientierte Analyse, Design und UML

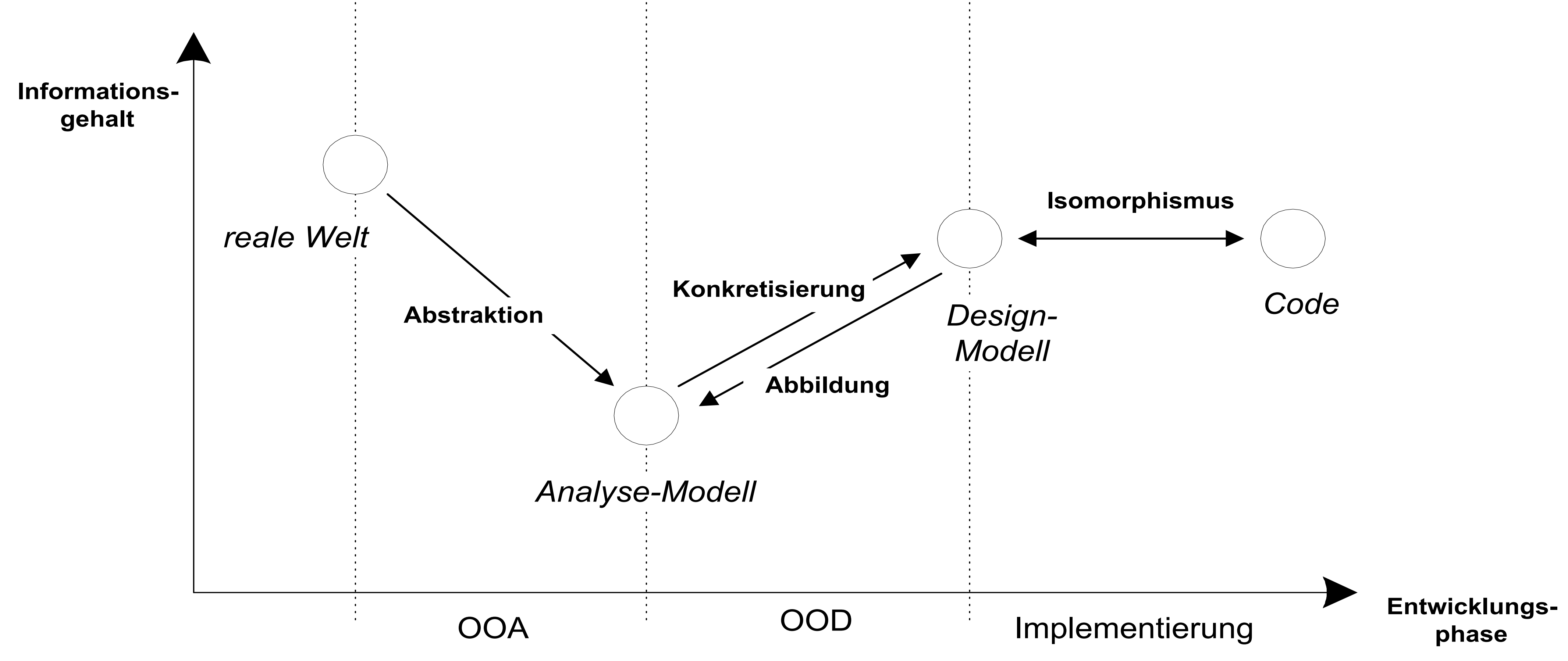
Quellenhinweis:
Einige Folien zu dieser Vorlesung entstammen Präsentationen von Prof. U. Andelfinger, Prof. G. Raffius und Prof. W. Weber

Abstraktionsebenen



Abstraktionsebenen der objektorientierten Systementwicklung (4-Schichten-Modell)

Phasenmodell der objektorientierten Systementwicklung



Warum ist schrittweises Vorgehen sinnvoll?

- Komplexität wird organisiert/beherrschbar.
- Das System ist strukturiert.
- Das System ist gut dokumentiert

“Round-Trip-Engineering“:

⇨ Tiefe Abbildungsschritte (Codierung/Impl.-Konz.) soweit automatisieren, dass Codierung auf OOA-/OOD-Ebene erfolgen kann.

Änderungen auf Codeebene

⇑

automatisch

⇓

Änderungen OOD-/OOA-Ebene

Die Entstehungsgeschichte der UML

OML
Firesmith, Henderson-Sellers, Page-Jones; 1998

Unified Modeling Language v1.0
Booch, Jacobson, Rumbaugh; 1997

Catalysis
D'Souza, Willes; 1996

Unified Method
Booch, Rumbaugh; 1995

Fusion
Coleman, et al.; 1994

MOSES
Henderson-Sellers; 1994

SOMA
Graham; 1994

OBA
Rubin; 1992

BON
Nerson; 1992

OOA&D
Martin, Odell; 1992

OOD
Booch; 1992

OOSE
Jacobson; 1992

OOAD&I
Henderson-Sellers, Macrone; 1992

OOA
Coad, Yourdan; 1991

OMT
Rumbaugh, et al.; 1991

OSA
Embley; 1991

OOSA
Shlear, Mellor; 1991

SCOOP
Cherry; 1990

HOOD
ESA; 1990

RDD
Wirfs-Brock; 1990

OBA
Bailin; 1989

State Charts
Harel; 1987

Quelle und Literaturhinweis: http://www.uml.org

UML als Vorgehensmodell ?

Es existieren verschiedene objektorientierte Methoden
Am bekanntesten:
OMT: Object Modelling Technique (James Rumbaugh)

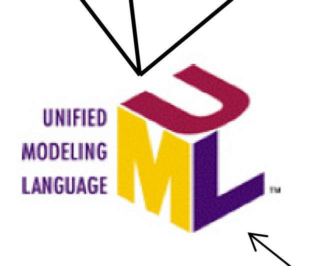
OMT: Object Modelling Technique
v. Rumbaugh,...

OOSE: Object Oriented
Software-Eng.
v. Jackobson

OOSE: Object Oriented
System Analysis
v. Shlaer/Mellor

OOD: Object Oriented Design
v. Booch

OOA: Object Oriented Analysis
v. Coach/Yourdon



Rumbaugh, Booch, Jacobson, Firma Rational.
Standardisierungsvorschlag bei OMG eingereicht: 1997

Kein Vorgehensmodell !
Nur Sammlung von
Beschreibungsmitteln.

UML Inhalte

Die UML eignet sich zur durchgängigen (OO)-Modellierung statischer und dynamischer Aspekte bis zur Implementierung

Struktur-Diagramme

1. Klassendiagramm
2. Objektdiagramm
3. Komponentendiagramm
4. Paketdiagramm
12. Verteilungsdiagramm
13. Kompositionsstrukturdiagramm

Verhaltens-Diagramme

5. Use-Case-Diagramm
10. Zustandsautomat
11. Aktivitätsdiagramm

Interaktionsdiagramme

6. Sequenzdiagramm
7. Kommunikationsdiagramm
8. Zeitverlaufsdiagramm
9. Interaktionsübersichtsdiagramm

Inhalt von SWT II

Diagramme der UML 2

- Diagramme der UML 2
  - Struktur-diagramme
    - Klassendiagramm
    - Kompositions-strukturdiagramm
    - Komponenten-diagramm
    - Verteilungs-diagramm
    - Objektdiagramm
    - Paketdiagramm
  - Verhaltens-diagramme
    - Aktivitäts-diagramm
    - Interaktions-diagramme
      - Sequenz-diagramm
      - Kommunikations-diagramm
      - Interaktionsüber-sichtsdiagramm
      - Zeitverlaufs-diagramm
    - Use-Case-Diagramm
    - Zustandsautomat

Quelle: „UML 2 –Ballast oder Befreiung?“
von Chris Rupp, SOPHIST GROUP, Agility Days 2003

Diagramme der UML – Anwendung I

Diagrammtyp	Diese zentrale Frage beantwortet das Diagramm	Stärken
Klassendiagramm	Aus welchen Klassen besteht mein System und wie stehen diese untereinander in Beziehung?	Beschreibt die statische Struktur des Systems. Enthält alle relevanten Struktur-zusammenhänge/Datentypen. Brücke zu dynamischen Diagrammen. Normalerweise unverzichtbar.
Paketdiagramm	Wie kann ich mein Modell so schneiden, dass ich den Überblick bewahre?	Logische Zusammenfassung von Modellelementen. Modellierung von Abhängigkeiten/ Inklusion möglich.
Objektdiagramm	Welche innere Struktur besitzt mein System zu einem bestimmten Zeitpunkt zur Laufzeit (Klassendiagramm-schnappschuss)?	Zeigt Objekte u. Attributbelegungen zu einem bestimmten Zeitpunkt. Verwendung beispielhaft zur Veranschaulichung Detailniveau wie im Klassen-diagramm. Sehr gute Darstellung von Mengenverhältnissen.

Quelle: „UML 2 –Ballast oder Befreiung?“
von Chris Rupp, SOPHIST GROUP, Agility Days 2003

Diagramme der UML – Anwendung II

Diagrammtyp	Diese zentrale Frage beantwortet das Diagramm	Stärken
Kompositionsstruktur-diagramm SWT 2	Wie sieht das Innenleben einer Klasse, einer Komponente, eines Systemteils aus?	Ideal für die Top-Down-Modellierung des Systems (Ganz-Teil-Hierarchien). Zeigt Teile eines „Gesamtelements" und deren Mengenverhältnisse. Präzise Modellierung der Teile-Beziehungen über spezielle Schnittstellen (Ports) möglich.
Komponentendiagramm	Wie werden meine Klassen zu wieder verwendbaren, verwaltbaren Komponenten zusammengefasst und wie stehen diese in Beziehung?	Zeigt Organisation und Abhängig-keiten einzelner technischer Systemkomponenten. Modellierung angebotener und benötigter Schnittstellen möglich.
Verteilungsdiagramm SWT 2	Wie sieht das Einsatzumfeld (Hardware, Server, Datenbanken, ...) des Systems aus? Wie werden die Komponenten zur Laufzeit wohin verteilt?	Zeigt das Laufzeitumfeld des Systems mit den „greifbaren" Systemteilen. Darstellung von „Softwareservern" möglich. Hohes Abstraktionsniveau, kaum Notationselemente.

Quelle: „UML 2 –Ballast oder Befreiung?"
von Chris Rupp, SOPHIST GROUP, Agility Days 2003

Diagramme der UML – Anwendung III

Diagrammtyp	Diese zentrale Frage beantwortet das Diagramm	Stärken
Use-Case-Diagramm	Was leistet mein System für seine Umwelt (Nachbarsysteme, Stakeholder)?	Außensicht auf das System. Geeignet zur Kontextabgrenzung. Hohes Abstraktionsniveau, einfache Notationsmittel.
Aktivitätsdiagramm SWT 2	Wie läuft ein bestimmter fluss-orientierter Prozess oder ein Algorithmus ab?	Sehr detaillierte Visualisierung von Abläufen mit Bedingungen, Schleifen, Verzweigungen. Parallelisierung und Synchronisation. Darstellung von Datenflüssen.
Zustandsautomat SWT 2	Welche Zustände kann ein Objekt, eine Schnittstelle, ein Use Case, … bei welchen Ereignissen annehmen?	Präzise Abbildung eines Zustands-modells mit Zuständen, Ereignissen, Nebenläufigkeiten, Bedingungen, Ein- und Austrittsaktionen. Schachtelung möglich.
Sequenzdiagramm	Wer tauscht mit wem welche Informationen in welcher Reihenfolge aus?	Darstellung d. Informationsaustauschs zwischen Kommunikationspartnern Sehr präzise Darstellung der zeitlichen Abfolge auch mit Nebenläufigkeiten.

Quelle: „UML 2 –Ballast oder Befreiung?“
von Chris Rupp, SOPHIST GROUP, Agility Days 2003

Diagramme der UML und ihre Anwendung IV

Diagrammtyp	Diese zentrale Frage beantwortet das Diagramm	Stärken
Kommunikations-diagramm	Wer kommuniziert mit wem? Wer „arbeitet“ im System zusammen?	Stellt den Informationsaustausch zwischen Kommunikationspartnern dar. Überblick steht im Vordergrund (Details und zeitliche Abfolge weniger wichtig).
Timingdiagramm SWT 2	Wann befinden sich verschiedene Interaktionspartner in welchem Zustand?	Visualisiert das exakte zeitliche Verhalten von Klassen, Schnittstellen,.. Geeignet für die Detailbetrachtungen, bei denen es wichtig ist, dass ein Ereignis zum richtigen Zeitpunkt eintritt.
Interaktionsübersichts-diagramm SWT 2	Wann läuft welche Interaktion ab?	Verbindet Interaktionsdiagramme (Sequenz-, Kommunikation- und Timingdiagramme) auf Top-Level-Ebene. Hohes Abstraktionsniveau.

Quelle: „UML 2 –Ballast oder Befreiung?“
von Chris Rupp, SOPHIST GROUP, Agility Days 2003

Wann verwende ich welches Diagramm?

- Es gibt verschiedene Diagramme um ähnliche Sachverhalte darzustellen
  – z.B. Sequenzdiagramm vs. Timingdiagramm
- Die Notwendigkeit einiger Diagramme hängt vom konkreten Projekt ab
  – z.B. Verteilungsdiagramm bei Einzelplatzanwendung nicht nötig
- Die Anzahl der zu erstellenden Diagramme hängt von der Komplexität / dem Kommunikationsbedarf uvm. ab
- Ein Diagramm stellt oft nur einen speziellen Ausschnitt des Systems dar

  ⇒ Die Diagramme entstehen *nach Bedarf*
  ⇒ Grobe Festlegung durch das gewählte Vorgehensmodell

Begriffe und Übersetzungen

http://www.oose.de

bietet

- ⇨ Übersetzungstabelle Deutsch – English
- ⇨ UML-Notationsübersicht
- ⇨ Glossar
- ⇨ und vieles mehr…

Einschub: UML-Syntax der Klassen im Analysemodell (für das Praktikum)

Attribute:

[Sichtbarkeit] Name [:Typ] [=Wert]

Methoden:

[Sichtbarkeit] Name ([Parameterliste]) [:Returntyp]

mit

Parameterliste als Folge von [In | Out | Inout] Name [:Typ] [=Wert]

Sichtbarkeit als [public|privat|protected|package]

- **Beispiele:**

  Kontonummer: Real

  protected GeldAbheben (Betrag: Real): Boolean

Weiteres Vorgehen in der Vorlesung

⇨ Sequentielles Durchschreiten der konstruktiven Phasen im SW-Lifecycle

  – Anforderungsanalyse
  – (Definition)
  – (Architektur)
  – Design
  – (Implementierung)

- …mit den entsprechenden Diagrammen der UML
-

Anforderungsanalyse (reale Welt ⇨ OOA)

- Warum?

⇨ Informatiker sind nur Experten für Software !

  ⇨ Anwendungsgebiet begreifen (um was geht´s ?)

  ⇨ Arbeitsabläufe kennenlernen (wie läuft das ?)

  ⇨ gegenseitig kompetent machen

  ⇨ Kommunizieren lernen !

- Ziele:

⇨ Schwachstellen und Probleme der Anwendungswelt nachvollziehen

⇨ Verstehen, was genau gewünscht wird und was nicht

Anforderungsanalyse (reale Welt ⇨ OOA)

- Wie?

⇨ Mit kleinem Expertenteam zusammensitzen

⇨ Geschäftsprozesse (Workflow) untersuchen

⇨ fragen, diskutieren, zuhören, lesen, dokumentieren

⇨ Glossar für Fachbegriffe

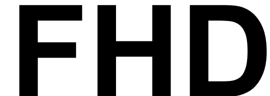
– **Dem Anwender soll ein Gefühl für Komplexität und Empfindlichkeit aber auch Möglichkeiten und Flexibilität von SW vermittelt werden !**

Grundsätzliches zur Anforderungsanalyse

- Das Herausarbeiten und Dokumentieren von Requirements ist eine sehr schwierige Aufgabe
- Übliche Aussagen der Auftraggeber:
  - "Wenn ich den Knopf drücke, soll … passieren."
  - "An einer Kreuzung soll der Bildschirm … zeigen„
  - „Das 7. Bit signalisiert den Zustand, dass...“

⇨ Menschen denken nicht in Requirements - Menschen denken in Anwendungsfällen (und Lösungen)

⇨ Der Wege zur Erstellung von Requirementsdokumenten ist daher über die Beschreibung von **Anwendungsfällen** am einfachsten



Fachhochschule Darmstadt
Fachbereich Informatik

Softwaretechnik I

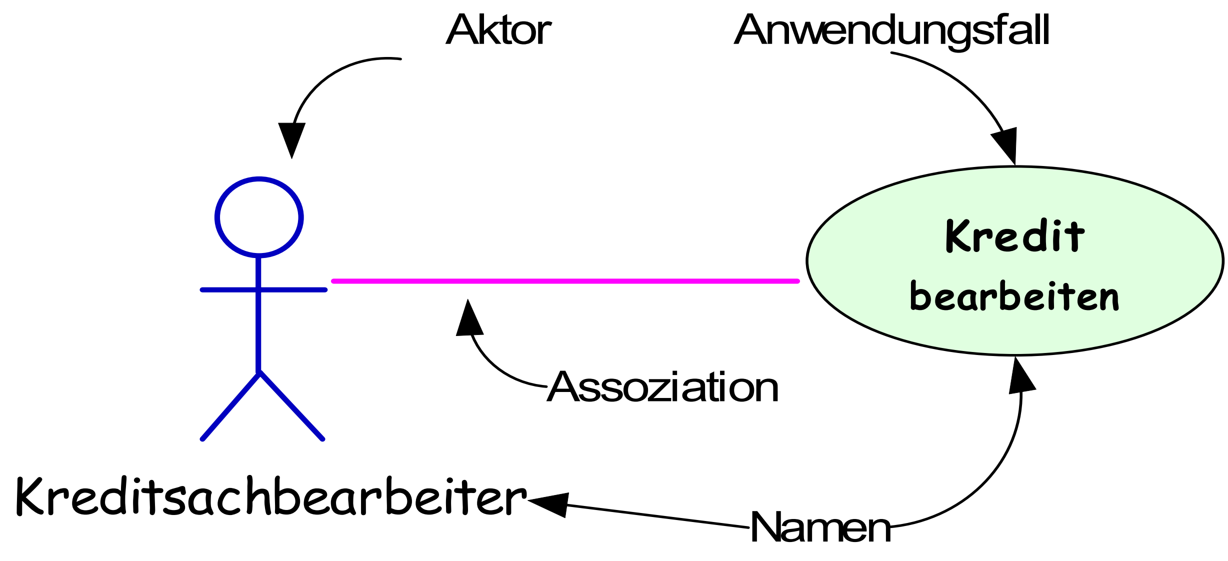
Kapitel 3 ff

- Use Cases & UML

Quellenhinweis:
Einige Folien zu dieser Vorlesung entstammen Präsentationen von Prof. G. Raffius und Prof. W. Weber

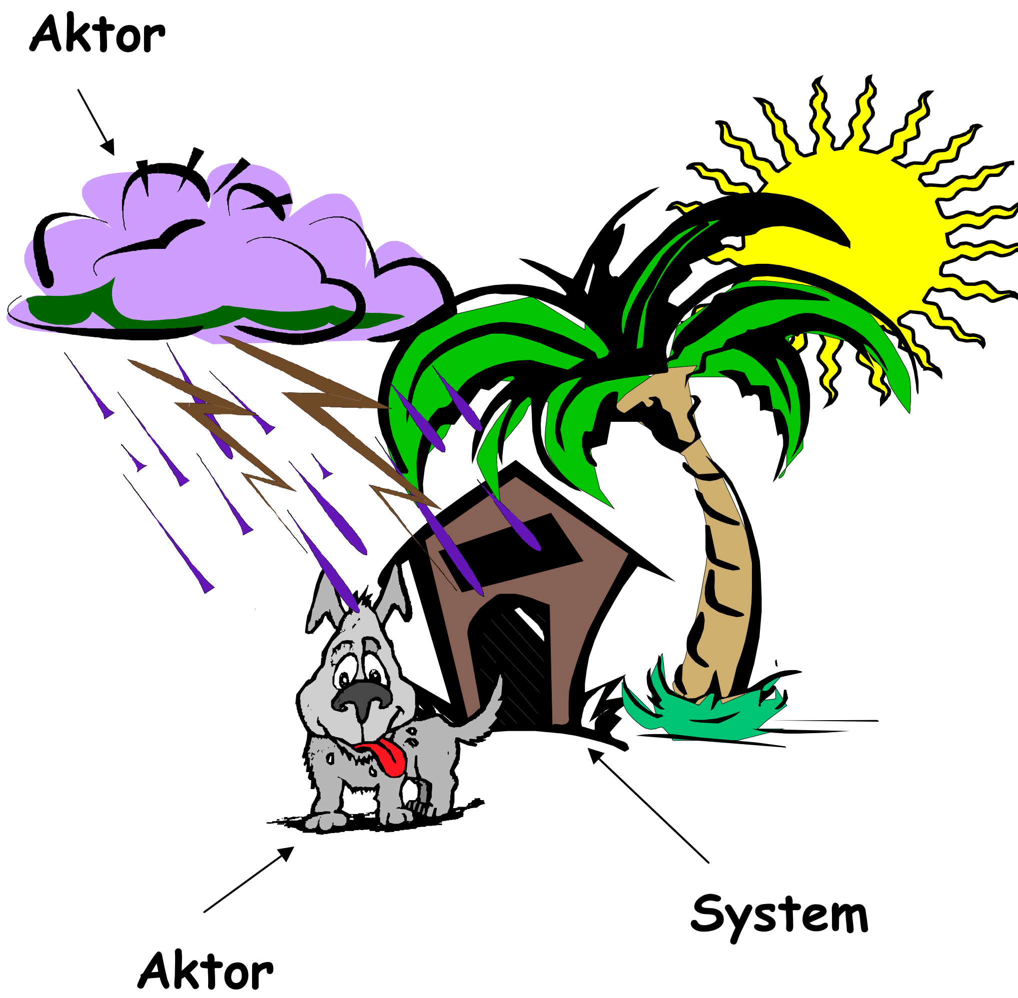
© Softwaretechnik I, SS 2005, FHD - Fachbereich Informatik, Ralf Hahn

Namen und Akteure



- Jeder Anwendungsfall muss einen Namen haben, der ihn von anderen Anwendungsfällen unterscheidet
- Namen sollten kurze Formulierungen mit aktiven Verben aus dem Vokabular des modellierten Systems sein

Der Kontext eines Systems



- Alle Dinge außerhalb des Systems, die mit dem System interagieren, bilden den Kontext eines Systems.
- Der Kontext definiert die Umgebung, in der das System lebt.
- Ein Produkt kann häufig in unterschiedlichen Kontexten betrieben werden.
- Innerhalb eines Kontextes kann es verschiedene Anwendungsfälle geben
- Durch den Kontext werden die nicht funktionalen Anforderungen vorgegeben

Definitionen (I)

- Aktor
  - irgend jemand oder etwas mit Verhalten. Auch das zu entwickelnde System ist ein Aktor
- Stakeholder
  - jemand oder etwas mit einem persönlichen Interesse am Verhalten des Systems unter Diskussion (SuD)
- Primary Actor
  - der Stakeholder, der die Interaktion mit dem SuD initiiert um ein Ziel zu erreichen
- Use Case
  - ein Vertrag für das Verhalten des SuD
- Scope
  - identifiziert das System über das wir diskutieren

Definitionen (II)

- Level
  - Auf welcher Ebene ist das Ziel angesiedelt. Gesamtziel, Userziel oder Teilfunktion?
- Preconditions and guarantees
  - was wahr sein muss vor und nach dem Ablauf eines Use Case
- Trigger
  - Auslöser der Use Cases
- Main Success scenario
  - Ein Use Case in dem alles funkioniert (nichts falsch wird)
- Extensions
  - was unterschiedlich imAblauf sein kann

Auslöser von Use Cases

- Der Start eines Use Cases wird durch ein Ereignis (Trigger) ausgelöst.
- Ereignisse können sowohl **asynchrone Ereignisse**, ausgelöst durch Akteure oder Systeme in der Umgebung sein, als auch **Zeitereignisse**
- Fragestellungen: **„Wer tut Was“** und **„Es ist Zeit für ...“**
- Beispiele
  - Bediener schaltet System ein
  - Es ist Zeit für ein Backup

Stakeholder und Akteure

- Ein Stakeholder ist jemand, der ein Interesse am Verhalten eines Use Cases hat.
- Ein Aktor ist jemand oder etwas, das ein eigenes Verhalten besitzt.
  - Ein Aktor (actor) repräsentiert eine Rolle, die ein Mensch ein Gerät oder ein anderes System gegenüber dem zu entwickelnden System spielt
  - Jeder Aktor ist gleichzeitig auch ein Stakeholder des Systems, aber nicht umgekehrt
  - Das System selbst (SuD, System under Design) ist ebenfalls ein Aktor
  - In einem Use Case können verschiedene Aktoren auftreten
    - Haupt Aktor (primary actor)
    - Hilfs Aktor (supporting actor)
    - interner Aktor (internal Aktor)
- Akteure können mit Anwendungsfällen durch Assoziationen verbunden werden. Dies zeigt, dass Anwendungsfall und Aktor miteinander kommunizieren und jeder Nachrichten sendet und empfängt

Dokumentation: Aktor Profil Tabelle

Name	Profil: Hintergrund und Fähigkeiten
Kunde	Person von der Straße, kann einen Touch Screen benutzen, aber es wird nicht angenommen, dass er Erfahrung mit GUIs hat. Kann Schwierigkeiten mit dem Lesen haben, kann kurzsichtig sein
Operator	Person, die ständig mit der Software arbeitet. Ist ein erfahrener Benutzer und kann den Wunsch haben, sich die Oberfläche anzupassen
Manager	Gelegentlicher Nutzer, kennt GUIs, ist aber nicht vertraut mit irgendeiner speziellen Software Funktion. Ungeduldig

Dokumentation: Muster für Use Case Spezifikation (I)

- **Use Case ## „Use Case Name“ <name>**
- **Context of Use:**
- <a longer statement of the context of use if needed>
- **Scope:**
- <what system is being considered black-box under design>
- **Level:**
- <one of: summary, user-goal, subfunction>
- **Primary Actor:**
- < a role name for the primary actor or description>
- **Stakeholder and Interests:**
  <list of stakeholders and key interests in the use case>
- **Precondition:**
- <what we expect is already the state of the world>
- **Minimal Guarantees:**
- <how the interests are protected under all exits>
- **Sucess Guarantees:**
- <the state of the world if goal succeeds>
- **Trigger:**
- <what starts the Use Case, may be time event>

Dokumentation: Muster für Use Case Spezifikation (II)

- **Main Sucess Scenario:**
  - <put here the steps of the scenario from trigger to goal delivery and any cleanup after>
  - <step #> <action description>
  - <step #> <action description>
- **Extensions:**
  - <put here the extensions one at time, referring to the steps of the main scenario>
  - <step altered><condition>: <action or sub use case>
  - <step altered><condition>: <action or sub use case>
- **Technologie & Data Variations:**
  - <put here the variations that will cause eventual bifurcation in the scenario>
  - <step or variation # > < list of variations>
  - <step or variation # > < list of variations>
- **Related Information:**
  - <whatever your project needs for additional information>

Beispiel Geldautomat (Ablauf als Text-Template)

- **Use Case Benutzer validieren**
- **Hauptablauf:**
  - Der Anwendungsfall beginnt, wenn das System den Kunden nach seiner PIN Nummer fragt.
  - Der Kunde kann nun eine PIN Nummer über die Tastatur eingeben.
  - Der Kunde schließt die Eingabe durch das Betätigen der Eingabetaste ab.
  - Das System überprüft dann die PIN Nummer auf ihre Gültigkeit.
  - Ist die PIN Nummer gültig, akzeptiert das System die Eingabe und der Anwendungsfall ist beendet.
- **Ausnahmeablauf:**
  - Der Kunde kann eine Transaktion jederzeit durch Betätigen der Abbruch Taste beenden und den Anwendungsfall so neu beginnen
- **Ausnahmeablauf:**
  - Der Kunde kann die PIN Nummer jederzeit löschen, bevor er die Eingabetaste betätigt und eine neue PIN Nummer eingeben
- **Ausnahmeablauf:**
  - Wenn der Kunde eine ungültige PIN Nummer eingibt, beginnt der Anwendungsfall erneut. Passiert dies dreimal in Folge, bricht das System die gesamte Transaktion ab und unterbindet für 60 Sekunden jede Interaktion mit dem Benutzer

Template in Tabellenform

<table>
<tr><td>USE CASE # / Name</td><td></td><td colspan="2"></td></tr>
<tr><td>Context of Use</td><td colspan="3"></td></tr>
<tr><td>Scope</td><td colspan="3"></td></tr>
<tr><td>Level</td><td colspan="3"></td></tr>
<tr><td>Primary Actor</td><td colspan="3"></td></tr>
<tr><td rowspan="2">Stakeholder and Interests</td><td>Stakeholder</td><td colspan="2">Interest</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>Precondition</td><td colspan="3"></td></tr>
<tr><td>Minimal Guarantee</td><td colspan="3"></td></tr>
<tr><td>Sucess Guarantees</td><td colspan="3"></td></tr>
<tr><td>Trigger</td><td colspan="3"></td></tr>
<tr><td rowspan="2">Main Sucess Scenario</td><td>Step #</td><td colspan="2">Action</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td rowspan="2">Extension</td><td>Step #</td><td>Condition</td><td>Action</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td rowspan="2">Technologie and Data Variations</td><td>Step #</td><td colspan="2">Variation</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>Related Information</td><td colspan="3"></td></tr>
</table>

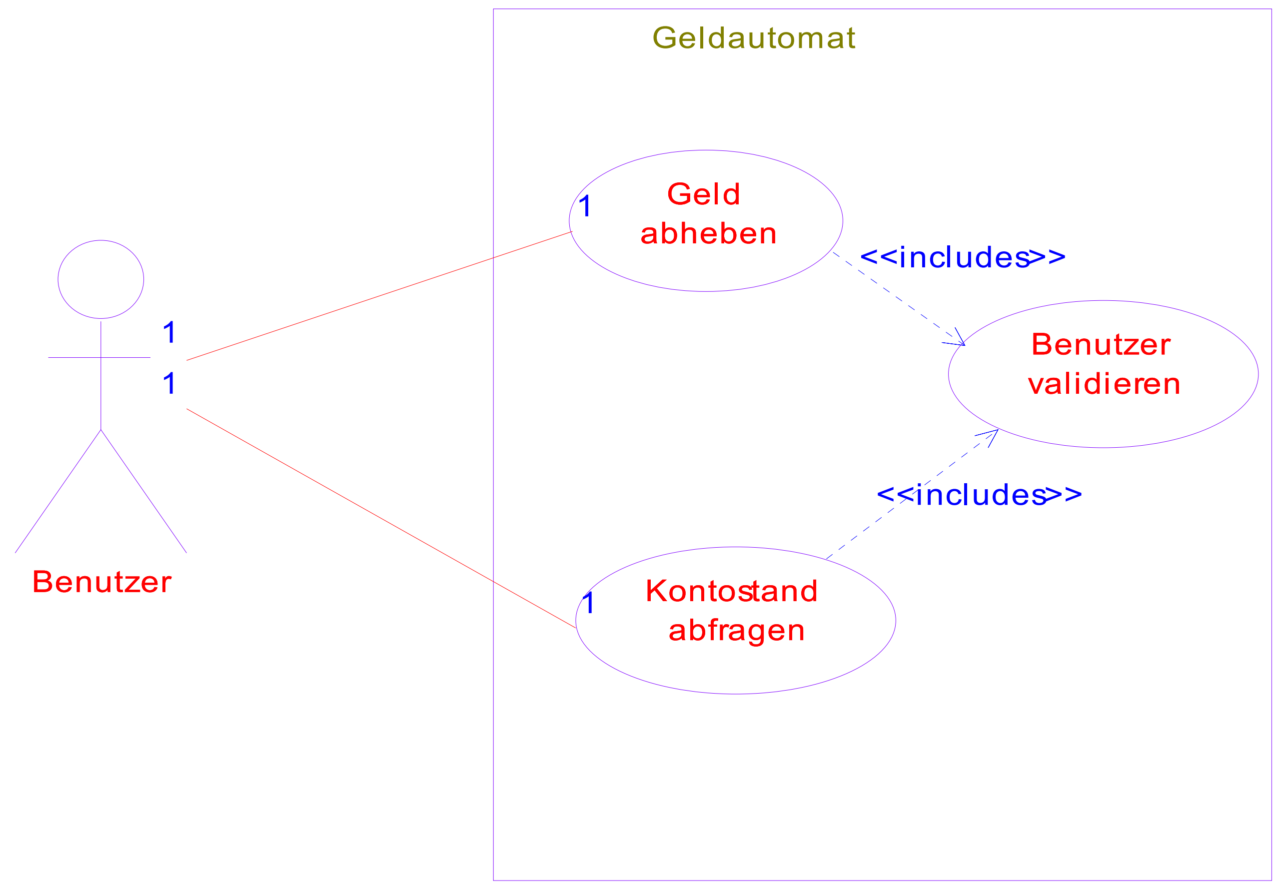
Beispiel Geldautomat (I) – als Tabelle

<table>
<tr><td>USE CASE # / Name</td><td>1</td><td colspan="2">Benutzer validieren</td></tr>
<tr><td>Context of Use</td><td colspan="3">Geldautomat einer Bank</td></tr>
<tr><td>Scope</td><td colspan="3">System under Design (Black Box)</td></tr>
<tr><td>Level</td><td colspan="3">Subfunction</td></tr>
<tr><td>Primary Actor</td><td colspan="3">System</td></tr>
<tr><td rowspan="3">Stakeholder and Interests</td><td colspan="2">Stakeholder</td><td>Interest</td></tr>
<tr><td colspan="2">Bank</td><td>Schutz vor unberechtigtem Zugriff</td></tr>
<tr><td colspan="2"></td><td></td></tr>
<tr><td>Precondition</td><td colspan="3">Kunde hat Kreditkarte eingeschoben</td></tr>
<tr><td>Minimal Guarantee</td><td colspan="3">Kein unberechtigter Zugang</td></tr>
<tr><td>Sucess Guarantees</td><td colspan="3">Zugriff wird gewährt</td></tr>
</table>

Beispiel Geldautomat (II) – als Tabelle

Trigger	System fragt Kunden nach PIN		
Main Sucess Scenario	step #	Action	
	1	Kunde gibt PIN ein	
	2	System validiert PIN	
	3	System gibt Zugang frei	
Extension	step #	Condition	Action
	*	Kunde bricht Vorgang ab	Beenden des Anwendungsfalls
	2a	PIN stimmt nicht und Versuche < 3	Weiter mit Schritt 1
	2b	PIN stimmt nicht und Versuche >= 3	System für 60s sperren
Technologie and Data Variations	step #	Variation	
Related Information			

Use Cases Geldautomat



Gedanken zu Use Cases (I)

- Use Cases sind geeignet um das Verhalten eines Systems zu visualisieren, zu spezifizieren, und zu dokumentieren

- Ein Use Case beschreibt eine Menge von Aktionsfolgen, einschließlich Varianten, die ein System ausführt um ein erkennbares, für einen Akteur nützliches Ergebnis zu erarbeiten.

- Ein Use Case beschreibt nicht, wie das Verhalten implementiert wird

- Use Cases geben eine Außenansicht von Systemen. Sie zeigen wie das System im Zusammenhang verwendet wird

Gedanken zu Use Cases (II)

- Use Cases geben eine Möglichkeit für den Entwickler mit den Endanwendern und den Fachleuten für den Anwendungs-bereich zu einem gemeinsamen Verständnis zu gelangen
- Sie schaffen die Basis für die Requirements die Überprüfung der Architektur und den späteren Test
- Ein Anwendungsfall stellt eine funktionale Anforderung an das System als Ganzes dar
- Der Kontext eines Anwendungsfalls liefert non-functional Requirements
- Die Sammlung der Use cases ergeben noch keine vollständige Sammlung der Requirements

© Softwaretechnik I, SS 2005, FHD - Fachbereich Informatik, Ralf Hahn

Übung: Use Case "Geld abheben" (I)

USE CASE # / Name	2	Geld abheben
Context of Use		
Scope		
Level		
Primary Actor		
Stakeholder and Interests	Stakeholder	Interest
Precondition		
Minimal Guarantee		
Sucess Guarantees		

Übung: Use Case "Geld abheben" (I)

<table>
<tr><td>USE CASE # / Name</td><td>2</td><td colspan="2">Geld abheben</td></tr>
<tr><td>Context of Use</td><td colspan="3">Geldautomat einer Bank</td></tr>
<tr><td>Scope</td><td colspan="3">System under Design (Black Box)</td></tr>
<tr><td>Level</td><td colspan="3">User</td></tr>
<tr><td>Primary Actor</td><td colspan="3">User</td></tr>
<tr><td rowspan="3">Stakeholder and Interests</td><td colspan="2">Stakeholder</td><td>Interest</td></tr>
<tr><td colspan="2">Bank</td><td>Schutz vor unberechtigtem Zugriff</td></tr>
<tr><td colspan="2">User</td><td>Abbuchung nur nach Auszahlung</td></tr>
<tr><td>Precondition</td><td colspan="3">keine</td></tr>
<tr><td>Minimal Guarantee</td><td colspan="3">Keine unberechtigte Auszahlung</td></tr>
<tr><td>Sucess Guarantees</td><td colspan="3">Geld wird ausgezahlt, Karte zurückgegeben</td></tr>
</table>

Übung: Use Case "Geld abheben" (II)

<table>
<tr><td>Trigger</td><td colspan="3"></td></tr>
<tr><td rowspan="5">Main Sucess Scenario</td><td>step #</td><td colspan="2">Action</td></tr>
<tr><td>1</td><td colspan="2"></td></tr>
<tr><td>2</td><td colspan="2"></td></tr>
<tr><td>3</td><td colspan="2"></td></tr>
<tr><td>4</td><td colspan="2"></td></tr>
<tr><td rowspan="4">Extension</td><td>step #</td><td>Condition</td><td>Action</td></tr>
<tr><td>*</td><td></td><td></td></tr>
<tr><td>1a</td><td></td><td></td></tr>
<tr><td>2a</td><td></td><td></td></tr>
<tr><td rowspan="2">Technologie and Data Variations</td><td>step #</td><td colspan="2">Variation</td></tr>
<tr><td>2</td><td colspan="2"></td></tr>
<tr><td>Related Information</td><td colspan="3"></td></tr>
<tr><td></td><td colspan="3"></td></tr>
</table>

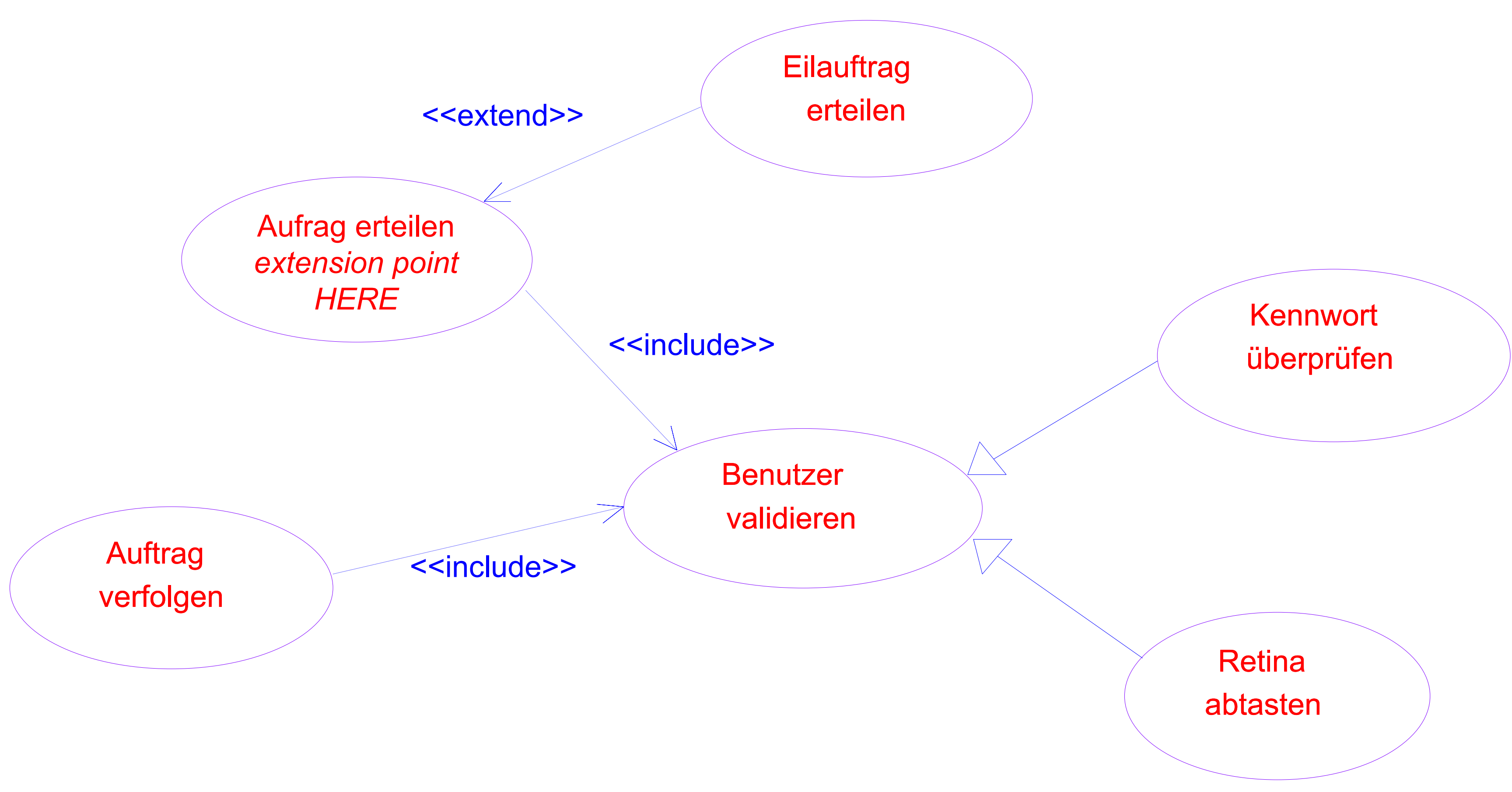
Übung: Use Case "Geld abheben" (II)

<table>
<tr><td>Trigger</td><td colspan="3">Benutzer schiebt Karte in das System</td></tr>
<tr><td rowspan="5">Main Sucess Scenario</td><td>step #</td><td colspan="2">Action</td></tr>
<tr><td>1</td><td colspan="2">Use Case: <u>Benutzer validieren</u></td></tr>
<tr><td>2</td><td colspan="2">Kunde gibt Wunschbetrag ein</td></tr>
<tr><td>3</td><td colspan="2">System gibt Karte aus, Kunde entnimmt Karte</td></tr>
<tr><td>4</td><td colspan="2">System gibt Geld aus, Kunde entnimmt Geld</td></tr>
<tr><td rowspan="4">Extension</td><td>step #</td><td>Condition</td><td>Action</td></tr>
<tr><td>*</td><td>Kunde bricht Vorgang ab</td><td>Beenden des Anwendungsfalls</td></tr>
<tr><td>1a</td><td>Validierung schlägt fehl</td><td>Karte ausgeben und Beenden des Anwendungsfalls</td></tr>
<tr><td>2a</td><td>Betrag über Limit</td><td>Weiter mit Schritt 2</td></tr>
<tr><td rowspan="2">Technologie and Data Variations</td><td>step #</td><td colspan="2">Variation</td></tr>
<tr><td>2</td><td colspan="2">Bei Summen über 1000€ stellt das System eine Überprüfungsfrage (im Stil „Sind Sie sicher?)</td></tr>
<tr><td>Related Information</td><td colspan="3">Die Karte wird VOR dem Geld ausgegeben. Ansonsten wird die Karte oft nicht mitgenommen.</td></tr>
</table>

Beziehungen zwischen Use Cases

- Extend
  - Die erweitert (extend) Beziehung sagt aus, dass der Basisanwendungsfall durch den anderen Anwendungsfall erweitert wird (z.B. optionales Verhalten).
- Include
  - Der Basisanwendungsfall bezieht explizit das Verhalten des anderen Anwendugsfalls ein
- Generalisierung
  - der spezialisierte Anwendungsfall ererbt das Verhalten des allgemeineren Anwendungsfalls und kann dieses ergänzen oder überschreiben

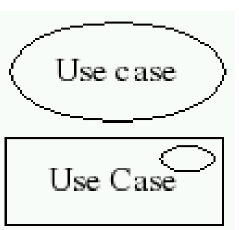
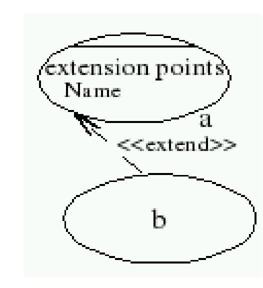
Beziehungen zwischen Use Cases am Beispiel



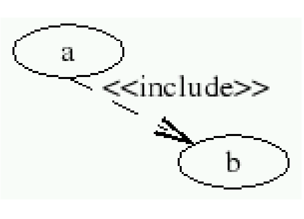
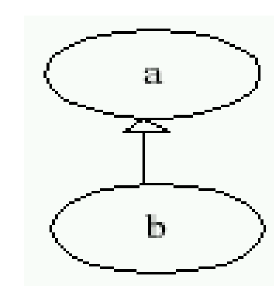
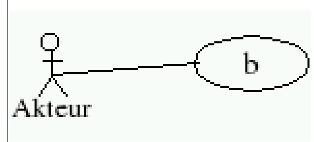
Notation UML 2.0 (I)

Symbol	Bedeutung
Name	**System** Das Rechteck stellt das geplante System dar. Das System erhält einen Namen. Ein Use Case Diagramm kann auch mehrere Systeme enthalten, die ineinander geschachtelt sein dürfen. Dadurch kann ein System in Teilsysteme gegliedert werden.
Akteur <<actor>> Akteur	**Akteur** Ein Akteur ist ein Element, das nicht zum geplanten System gehört. Er kann eine Person sein, die auf das System zugreift, oder ein anderes System, das mit dem geplanten System kommuniziert. Die UML erlaubt mehrere Symbole zur Darstellung eines Akteurs. Er kann als Strichmännchen dargestellt werden. Es ist optional erlaubt ein Klassensymbol zu verwenden, das mit dem Stereotyp <<actor>> markiert wird. Zusätzlich können eigene Symbole verwendet werden um nicht menschliche Akteure darzustellen.

Notation UML 2.0 (II)

	**Use Case** Eine Ellipse stellt einen Anwendungsfall des Systems dar. Ein Anwendungsfall ist ein in sich abgeschlossener Vorgang, der für einen oder mehrere Akteure ein beobachtbares Ergebnis liefert. Er beschreibt aus Sicht der Akteure welche Leistungen das System für den Anwender zur Verfügung stellt. Ein Use Case stellt somit einen Teil der Gesamtfunktionalität des Systems dar. In UML 2.0 kann auch ein Rechteck, das mit einer Ellipse markiert wird, als Use-Case-Symbol verwendet werden. Der Name kann innerhalb oder außerhalb des Symbols stehen.
	**Extension points** Das Verhalten eines Use Case kann durch einen weiteren Use Case erweitert werden. Dies wird durch die extend-Beziehung und den extension point angegeben. Die Beziehung wird gestrichelt dargestellt; der Pfeil zeigt auf den erweiterten Use Case und trägt den Stereotyp <<extend>>. Der extension point gibt den Ort im erweiterten Use Case an, an dem der erweiternde Use Case eingefügt wird. b erweitert das Verhalten von a. b wird an dem extension point "Name" in a eingefügt.

Notation UML 2.0 (III)

	**include-Assoziation** Die include Beziehung definiert einen Use Case, der die Funktionalität, die ein anderer Use Case zur Verfügung stellt, importiert. Der Use Case a importiert die Funktionalität des Use Case b.
	**Vererbungsbeziehung** In Use Case Diagrammen ist die Verebungsbeziehung erlaubt. Sie kann verwendet werden um Verhalten zwischen Use Cases zu vererben. Sie kann aber auch verwendet werden, um Vererbungsbeziehungen zwischen Akteuren aufzubauen.
	**Assoziation** Eine Linie stellt eine Assoziation zwischen einem Akteur und einem Use Case dar. Sie beschreibt den Zugriff des Akteurs auf die Funktionalität, die das System in diesem Use Case zur Verfügung stellt, bzw. eine Antwort des Systems an einen Akteur.

Use Cases für ein Gesamtsystem: Beispiel Hochschulverwaltung (I)

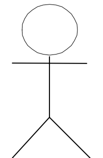
- Aufgabe:
  - ***Modellieren Sie ein Use Case Diagramm für das System Hochschulverwaltung.***
- Anforderungen:
- **Das System Hochschulverwaltung soll folgende Bereiche abdecken:**
- **Benutzertypen sind**
- Auslandsreferats-Angestellter
- Prüfungsamtsangestellter
- Hausmeister
- Verwaltungsangestellter
- Dekan
- Fachbereichssekretärin

Use Cases für ein Gesamtsystem: Beispiel Hochschulverwaltung (II)

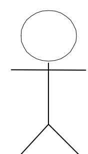
- Immatrikulation von Studenten,
- Exmatrikulation von Studenten,
- Rückmeldung von Studenten
- Erfassen von Prüfungsergebnissen,
- Anmeldung zur Diplomarbeit und Registrierung der Gutachten
- Verwaltung von Studienaustausch u.Immatrikulation von ausländischen Studierenden im Rahmen eines Austauschs
- Das Erfassen und Verwalten von Lehrbeauftragten und Tutoren
- Das Verwalten von verliehenem Hochschuleigentum an Studenten, Tutoren und Lehrbeauftragte (Schlüssel, Bücher etc.)
- Das Erstellen von Lehrveranstaltungskalendern mit Zeit- und Raumbelegung

Use Cases für ein Gesamtsystem: Beispiel Hochschulverwaltung (III)

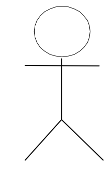
- **Übung**



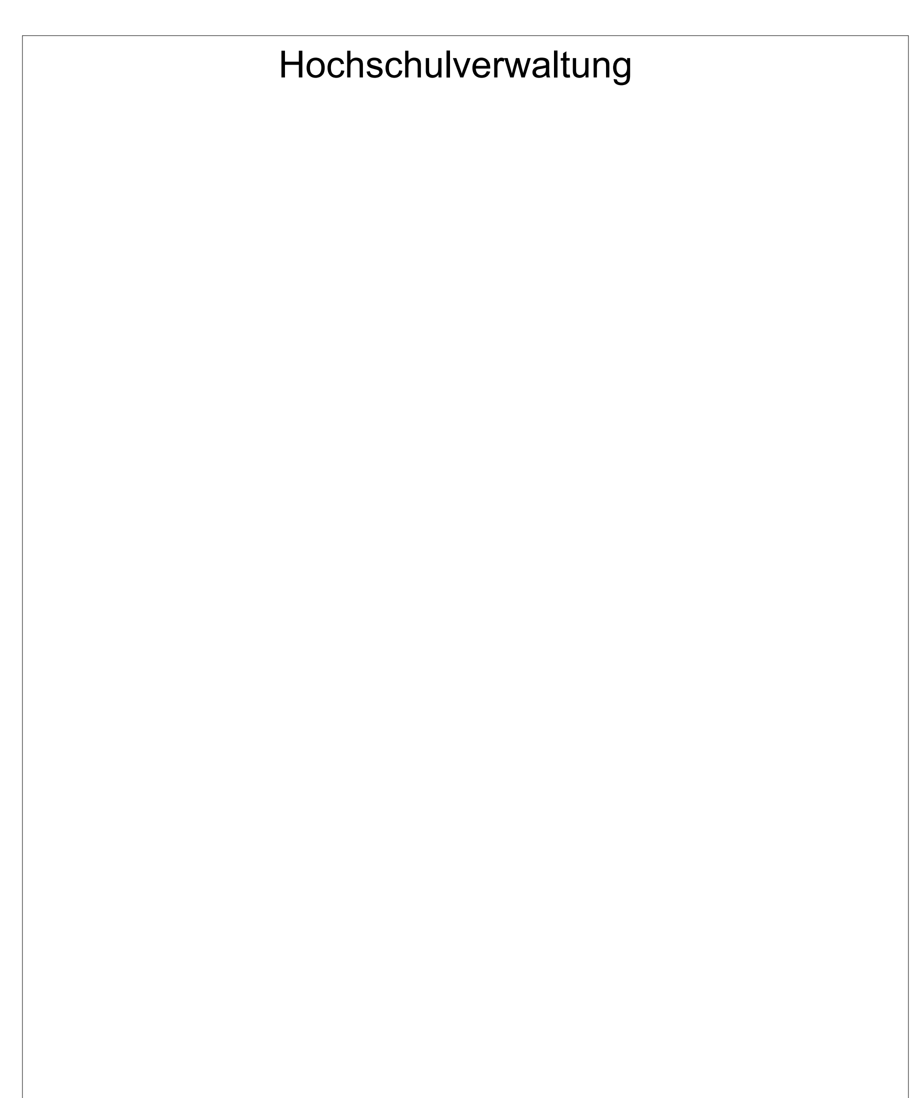
Auslandsreferats
Angestellter



Dekan



Fachbereichs
Sekretärin



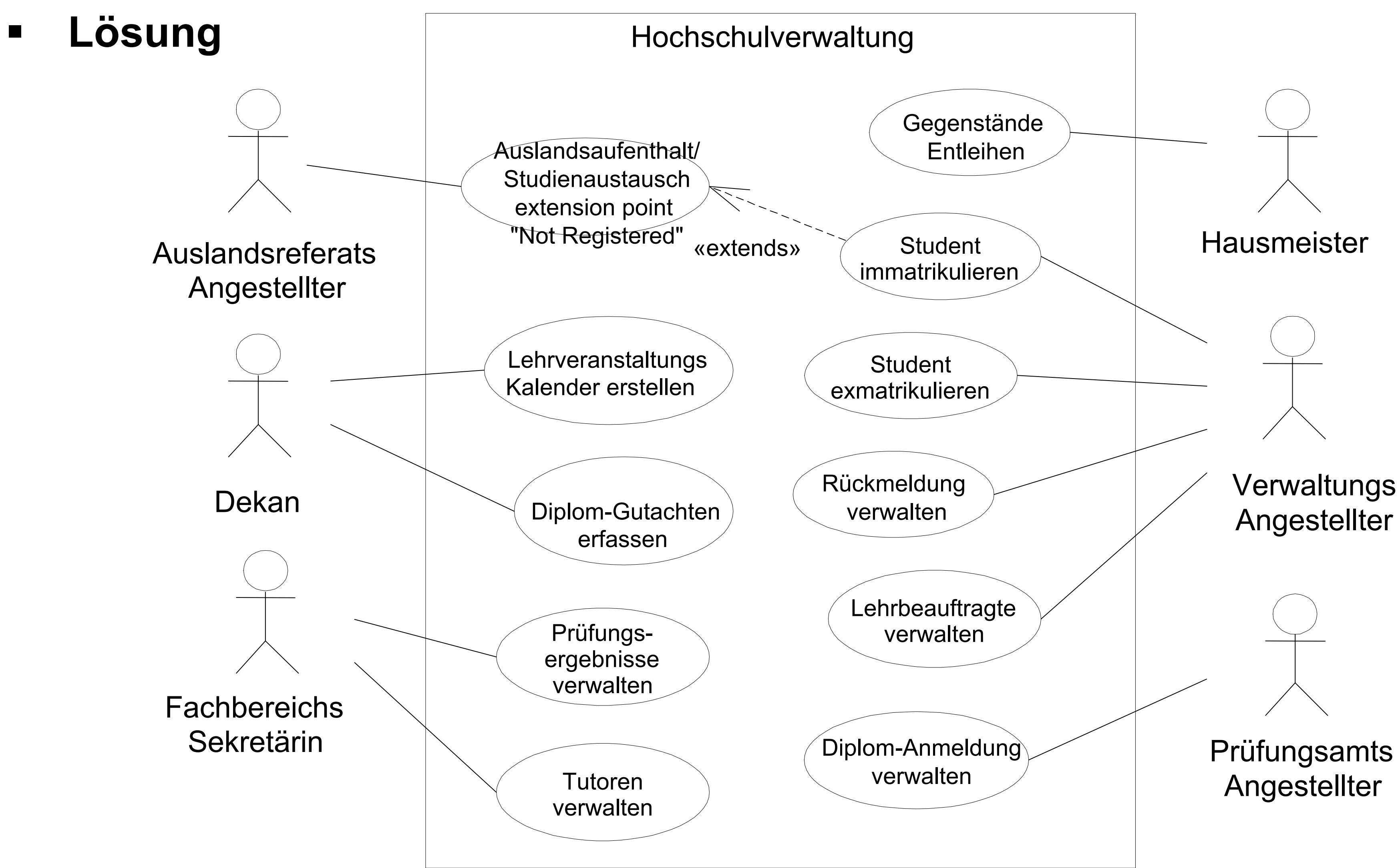
Hausmeister

Verwaltungs
Angestellter

Prüfungsamts
Angestellter

Use Cases für ein Gesamtsystem: Beispiel Hochschulverwaltung (III)

- **Lösung**



Zwischenstand Use Cases

- Bekannt:
  - Darstellung eines einzelnen Use Cases
  - Darstellung eines Gesamtsystems mit Use Cases
  - Darstellung mit UML 2.0

- Noch (weitgehend) offen:
  - Wie finde ich die Aufteilung der Use Cases?
  - Was lagere ich als Unter-Use Case aus?
  - Welche Granularität / welchen Level brauche ich?
  - Was gehört in der Beschreibung wohin?

- Lösung:
  - Regeln, Tipps...
  - ... und Iterationen

12 Schritte zu Use Cases für ein System (I)

1. Finde die Grenzen des Systems
   (Kontext Diagramm, IN/OUT Liste)
2. Brainstorme und liste die Aktoren (Aktor Profil Tabelle)
3. Brainstorme und liste die Ziele der Aktoren in Bezug auf das System
   (Aktor-Ziele Liste)
4. Schreibe die äußersten Summary Use Cases, die das gefundene zusammenfassen
5. Überdenke und verbessere die strategischen Use Cases. Füge Ziele hinzu, entferne Ziele und fasse zusammen
6. Nehme einen Use Case und schreibe eine Geschichte, um mit dem Material vertraut zu werden
7. Füge die Stakeholder hinzu, deren Interessen, Vorbedingungen und Garantien. Überprüfe sie doppelt.

12 Schritte zu Use Cases für ein System (II)

8. Schreibe das Hauptererfolgsszenario (main success scenario). Vergleiche es mit den Interessen und Garantien
9. Brainstorme und liste die möglichen Fehlerbedingungen und die alternativen Erfolgsbedingungen
10. Beschreibe wie die Aktoren und das System sich in den Erweiterungen (extensions) Verhalten sollen
11. Breche jeden Unter Use Case heraus, der seinen eigenen Raum braucht
12. Beginne vom Anfang und verbessere die Use Cases. Füge hinzu, entferne oder fasse zusammen wenn angebracht. Überprüfe Vollständigkeit, Lesbarkeit und Fehlerbedingungen.

Übung: Use Cases für das System „Geldautomat“

- ⇨ Spielen Sie die 12 Schritte für das Beispiel „Geldautomat“ durch
- ⇨ Erstellen Sie keine Use Cases im Detail
- ⇨ Bestimmen Sie nur die Gesamtstruktur der Use-Cases
- ⇨ Verwenden Sie auch die Assoziationen zwischen Use Cases

Übung: Use Cases „Geldautomat“ – Ergebnis (1. Iteration, unvollständig)

1. Grenzen des Systems

IN	**OUT**
Glasscheibe	Geld
Tastatur	Unterbrechungsfreie Stromversorgung
Kartenleser	EC-Karte
Bildschirm	

Was ist mit einem Dieb?

2. Aktoren und 3. Ziele

Aktoren	**Ziele**	**Zu klären**
Bankkunde	Geld abheben	
	Kontostand abfragen	
Service Techniker	Fehlerprotokoll auslesen	Remote?
	Hardware-Selbsttest	Remote?
Geld-Bote	Geld einfüllen	
Bank-Verantw. GA	Remote-Abfrage Status	
?? System??	Fehler-Benachrichtigung	

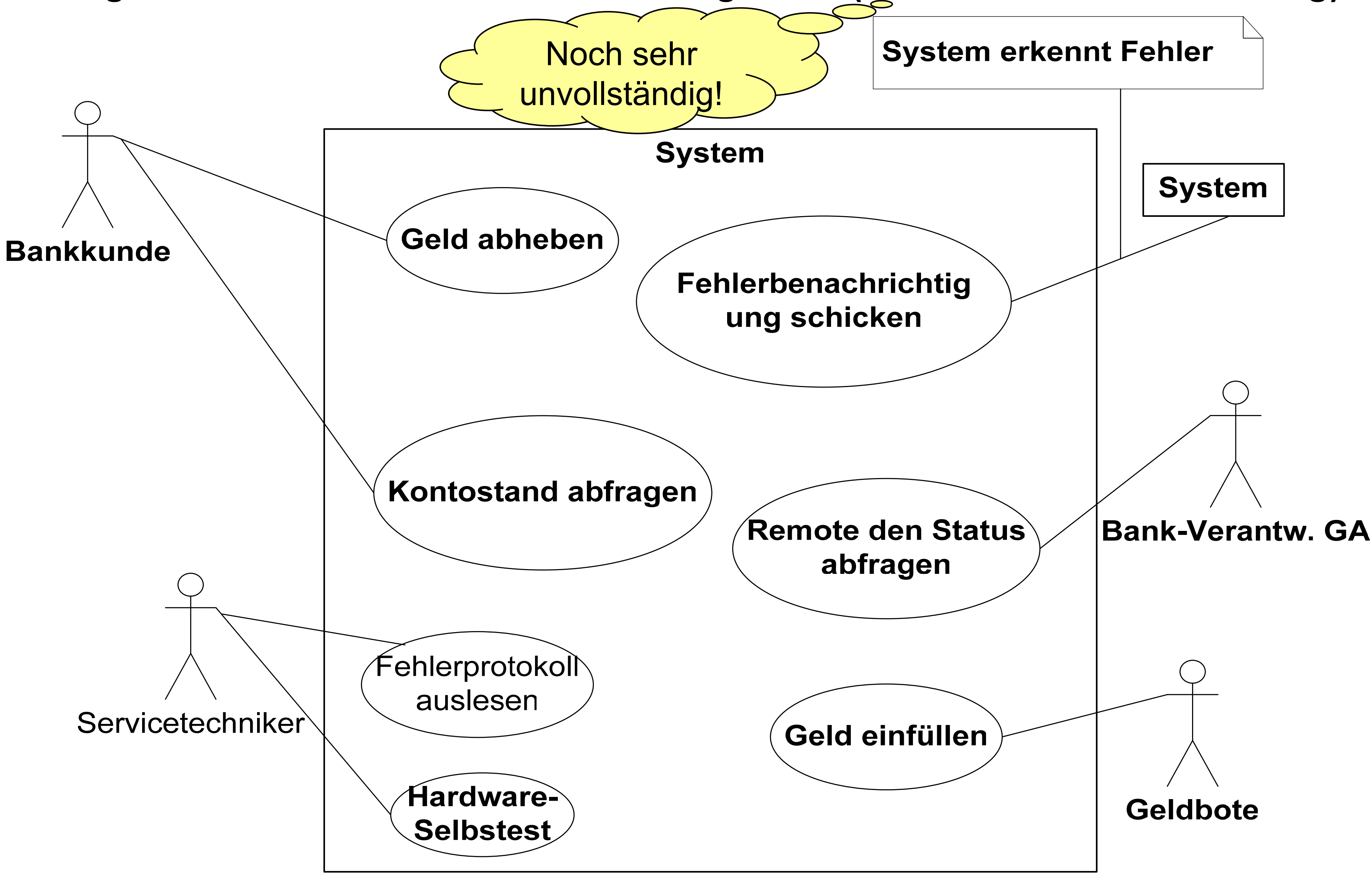
Übung: Use Cases „Geldautomat“ – Ergebnis (1. Iteration, unvollständig)

4. Summary Use Cases
   – Bank: Kostengünstigen Bargeldverkehr realisieren
   – Kunde: (und Bank): Rund-um-die-Uhr verfügbarer Bargeldverkehr

• …

7. Stakeholder, Interessen, Garantien
   – Gesetzgeber: Dokumentieren der Zahlungsvorgänge
   – usw.
8. Haupterfolgsszenario:
   – Geldauszahlung: Karte rein, richtige PIN eingeben, Betrag eingeben, Karte raus, Geld raus
9. Fehler, Alternativen:
   – Karte falsch, PIN falsch, Konto überzogen, Geld-Stau, Karte nicht entnommen, Geld nicht entnommen
10. Verhalten der Aktoren / des Systems in den Extensions
    – Karte ist keine EC-Karte: System wirft Karte aus, gibt Meldung aus
    – usw.
11. Use Cases rauziehen?
12. Iterierieren? Unbedingt – und vervollständigen!

Übung: Use Cases „Geldautomat" – Ergebnis (1. Iteration, unvollständig)



Tipps (I): Mache die Use Cases einfach lesbar (jeden einzelnen)

- Halte dich kurz und bringe es auf den Punkt
- Starte von oben und erzeuge eine stimmige Geschichte
- Oben befindet sich ein strategischer (summary) Use Case
- Die User Goal und Subfunction Use Cases verzweigen von hier aus
- wähle Namen mit kurzen Verben, die das Ziel des Use Cases ausdrücken
- starte vom Trigger und fahre fort bis das Ziel erreicht oder aufgegeben ist
- Ein Use Case hat zwischen 3 und 9 Schritten. Wenn mehr als 9 Schritten vorhanden sind, fasse die Schritte zusammen.
- Schreibe volle Sätze mit aktiven Verben die das Unterziel, das erreicht werden soll, ausdrücken
- Stelle sicher, dass der Aktor und seine Absicht in jedem Schritt sichtbar werden
- Stelle sicher, dass die Fehlerbedingungen gut erkennbar und die recovery Actions gut lesbar sind.
- Stelle sicher, dass gut erkennbar ist was als nächstes passiert, auch ohne die Nummern der Schritte
- Füge alternatives Verhalten in die Extensions ein und nicht als „if statements“ in den Hauptablauf
- Erzeuge Extension Use Cases nur unter ausgewählten Bedingungen

Tipps (II): Erinnerungen für jeden Use Case

- Halte die Benutzeroberfläche heraus
  - Stelle sicher, dass die Schritte die Du beschrieben hast die wahre Absicht des Aktors widerspiegeln und nicht die Bewegungen des Aktors beim manipulieren der Oberfläche.
  - Dies gilt nur, wenn funktionale Requirements beschrieben werden.
  - Man kann Use Cases auch benutzen um die Benutzer-Oberflächen zu dokumentieren

- Jeder Use Case hat zwei Enden: Erfolg und Misserfolg
  - Beachte, dass auch ein Sub Use Case mit Erfolg oder Misserfolg enden kann
  - Taucht der Fehler im Hauptablauf auf, so wird er in den Extensions behandelt.
  - In der Extension werden Erfolg und Fehlerbehandlung in der gleichen Extension beschrieben

Tipps (III): Ceckliste 1

- **Use Case Titel**
  - Ist es eine Formulierung mit einem aktiven Verb, die das Ziel des Primary Actors ausdrückt?
  - Kann das System das Ziel erfüllen?
- **Scope und Level**
  - Sind die Felder ausgefüllt?
- **Scope**
  - Behandelt der Use Case das betrachtete System als ein „Black Box System"?
  - Wenn das betrachtete System das zu entwickelnde System ist, haben dann die Designer alles im System zu entwickeln und nichts außerhalb?
- **Level**
  - Entspricht der Inhalt des Use Cases dem Ziel Level?
  - Ist das Ziel auf dem richtigen Level?
- **Primary Actor**
  - Hat der Primary Actor ein eigenes Verhalten?
  - Hat der Primary Actor ein Ziel gegenüber dem SuD dem ein Service Angebot des SuD entspricht?

Tipps (IV): Checkliste 2

- **Vorbedingungen**
  - Sind die Vorbedingungen obligatorisch und können sie durch das SuD erreicht werden?
  - Ist es wahr, dass sie im Use Case nicht mehr überprüft werden?
- **Stakeholder und Interessen**
  - Sind die Stakeholder und ihre Interessen alle aufgeführt und muss das System die Interessen alle erfüllen?
- **Minimale Garantien**
  - Sind die Interessen aller Stakeholder geschützt?
- **Erfolgsgarantien**
  - Sind die Interessen aller Stakeholder befriedigt?
- **Hauptablauf**
  - Hat der Hauptablauf zwischen 3 und 9 Schritten?
  - Läuft der Hauptablauf vom Trigger bis zur Erfüllung der Erfolgsgarantie?
  - Erlaubt der Ablauf die richtigen Variationen in der Reihenfolge?
- **Erweiterungsbedingungen**
  - Muss das System sie erkennen und behandeln?
  - Braucht das System die Erweiterung wirklich?

Tipps (V): Checkliste 3

- **Jeder Schritt des Szenarios**
  - Ist der Schritt als ein Ziel formuliert, das es zu erreichen gilt?
  - Läuft der Prozess deutlich weiter nach der erfolgreichen Erfüllung des Schritts?
  - Ist es klar, welcher Aktor das Ziel betreibt - wer „spielt den Ball“?
  - Ist die Absicht des Aktors klar?
  - Ist das Ziel des Schrittes niedriger als das Ziel Level des gesamten Use Cases? Ist es vorzugsweise nur ein bisschen unter dem Ziel Level?
  - Ist sicher gestellt, dass der Schritt nicht das User Interface Design des Systems beschreibt?
  - Ist klar welche Information in dem Schritt ausgetauscht wird?
  - Validiert der Schritt (im Gegensatz zu überprüft)?
- **Technologie und Daten Variationen**
  - Ist sichergestellt, dass es sich nicht um eine normale Erweiterung des Verhaltens des Hauptablaufs handelt?
- **Inhalt des Use Cases als Ganzes**
  - Frage an den Auftraggeber und den Nutzer: „Ist es das, was Ihr wollt?"
  - Frage an den Auftraggeber und den Nutzer: „Kannst Du nach der Lieferung bestätigen, dass dies geliefert wurde?“
  - Frage an den Entwickler: „Kannst Du das implementieren?"

Tipps (VI): Guidelines für den Hauptablauf

- Benutze eine einfache Grammatik
  - Subjekt ... Verb ... Objekt ... Präposition
- Zeige klar: „Wer hat den Ball“
- Schreibe aus der Vogelperspektive
- Zeige wie sich der Prozess fortbewegt
- Zeige was der Aktor will, nicht seine Bewegungen
- Benutze eine sinnvolle Menge von Aktionen
- Wähle exakte Begriffe und Wörter
  - z.B. Benutze „validiere“, und nicht „überprüfe“! Das System validiert das Passwort und überprüft es nicht
- wenn sinnvoll, erwähne die Zeit

⇨ Idiom: „User has System A kick System B“

⇨ Idiom: „Tue Schritte x-y bis Bedingung“

Tipps (VII): (Potenzielle) Stakeholder

- Mögliche Stakeholder sind
  - der Primary Actor
  - der Firmenbesitzer
  - gesetzliche Behörden
  - der Testingenieur
  - der Serviceingeneur
- Ein Use Case beschreibt nicht nur die Interaktionen zwischen dem Primary Actor und dem System.
- Ein Use Case beschreibt wie das System die Interessen aller Stakeholder schützt, mit dem Primary Actor als treibender Kraft
- Das Interesse des Primary Actors wird im Use Case Namen fest-gehalten. Normalerweise erhält er etwas
- Das Firmeninteresse besteht meistens darin, dass der Primary Actor nicht etwas umsonst erhält, für die Leistung bezahlen muss oder keinen Schaden anrichten kann
- Das Interesse des Gesetzgebers ist meist, dass die Firma zeigen kann, dass sie die Regeln einhält und dass die Transaktionen protokolliert werden
- Eines der typischen Interesse von Stakeholdern ist die Behandlung von Fehlern und das Recovery bei Fehlern innerhalb einer Transaktion

Tipps (VIII): Vorbedingungen

- Die Vorbedingungen eines Use Cases definieren die gültigen Randbedingungen unter denen der Use Case ablaufen kann
- Die Vorbedingungen müssen erwähnt werden, da sie im Use Case nicht mehr abgeprüft werden
- Es gibt zwei gebräuchliche Situationen für solche Vorbedingungen
  - Der User ist eingelogged und seine Autorisierung verifiziert
  - Ein zweiter Use Case verlässt sich darauf, dass in einem ersten Use Case Randbedingungen geschaffen wurden, auf die sich der zweite Use Case verlassen kann
- Immer wenn man eine Vorbedingung findet, kann man davon ausgehen, dass es einen Use Case auf höherem Level gibt, in dem die Vorbedingung gesichert wurde

Zusammenfassung Use Cases

- Darstellung
  - eines einzelnen Use Cases
  - eines Gesamtsystems mit Use Cases
  - mit UML 2.0

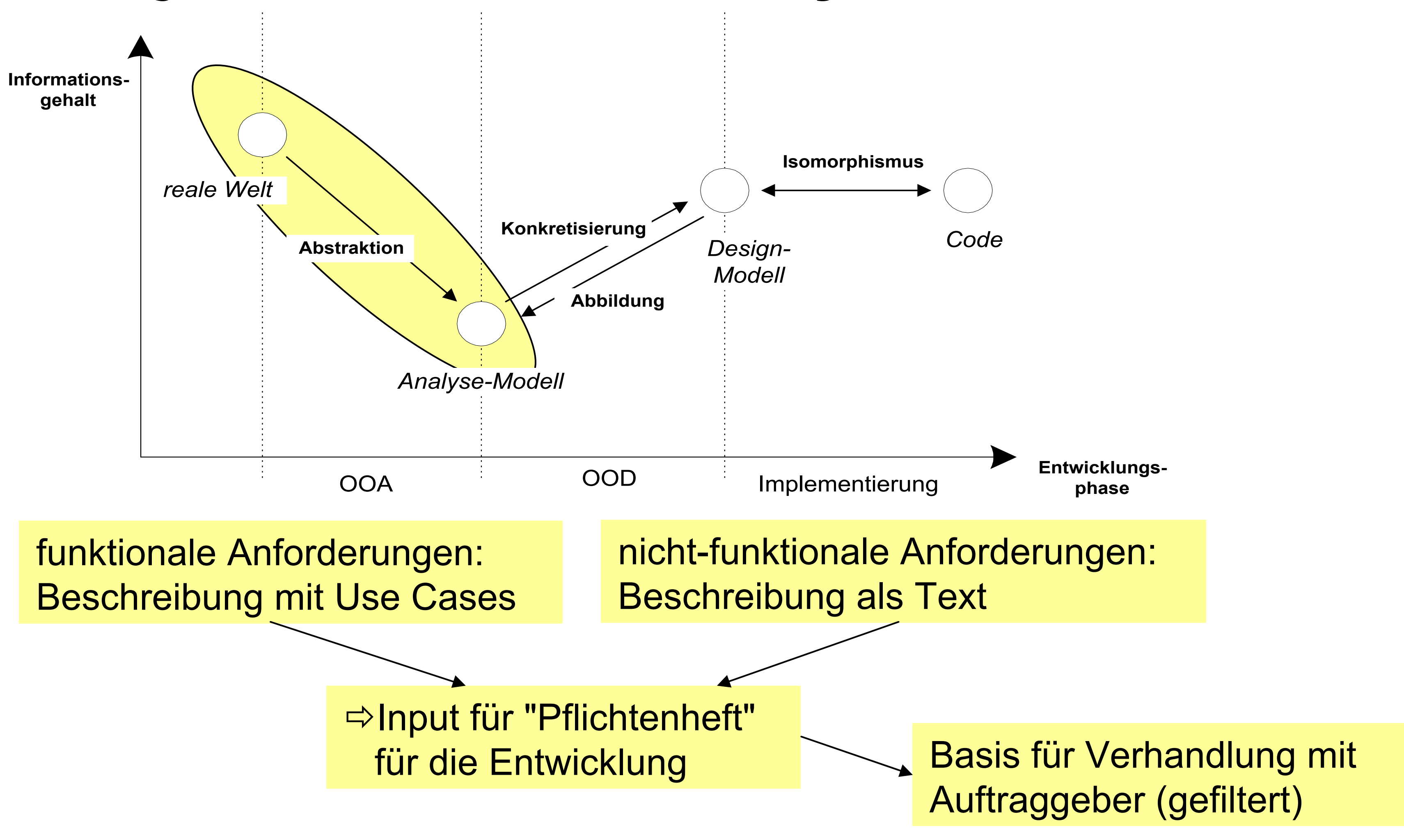
- Regeln und Tipps
  - Aufteilung der Use Cases (12 Schritte)
  - Beschreibung einzelner Use Cases (Geschichte, Grammatik)

⇨ Bei allen Regeln und Tipps – das Finden und Schreiben von Use Cases bleibt ein iterativer Prozess

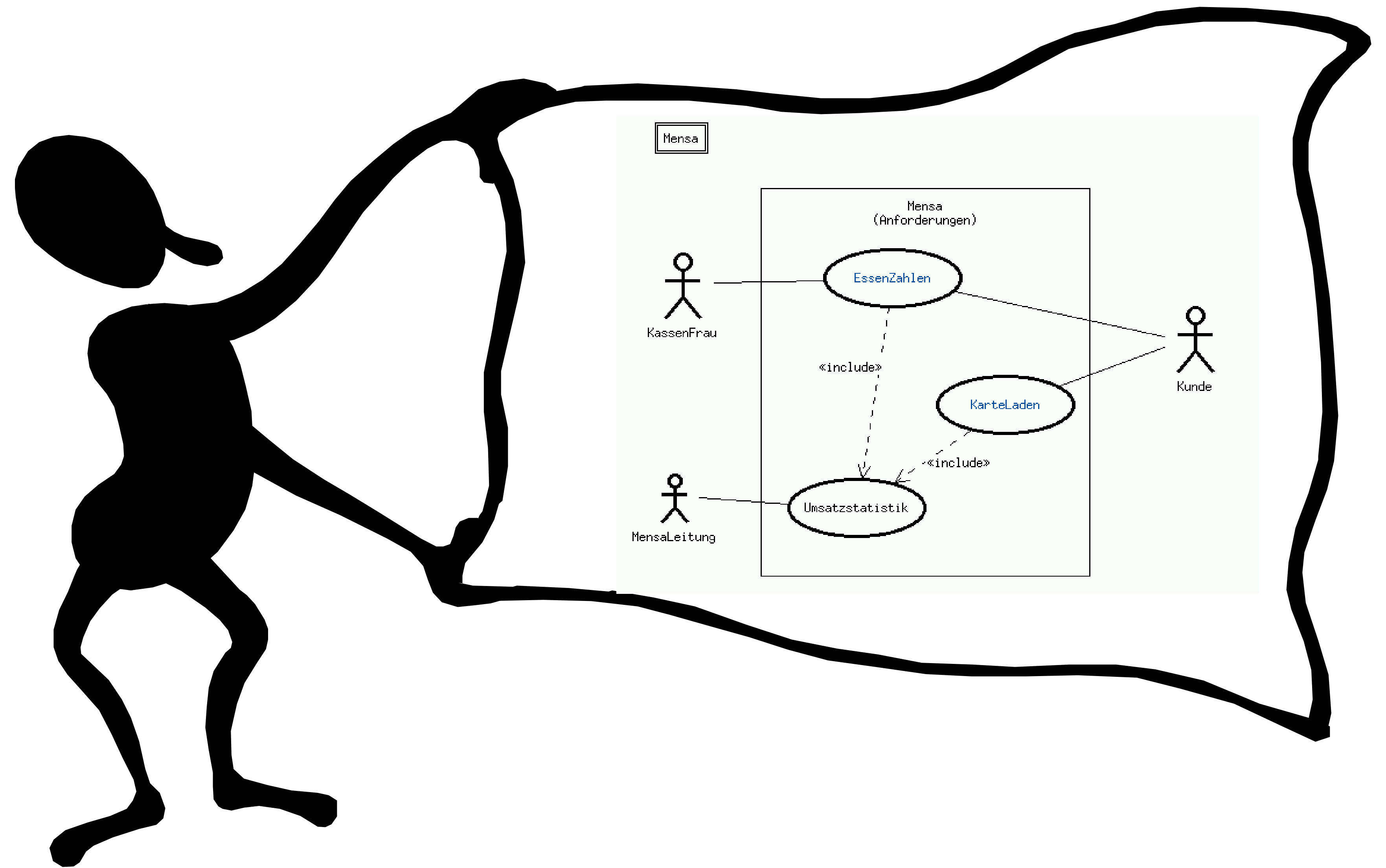
⇨ Versuchen Sie nicht, „perfekte" Use Cases zu schreiben, bevor das Gesamtkonzept stabil erscheint

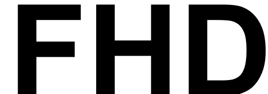
Stand im Phasenmodell

▪ Objektorientierte Analyse



Fragen





Fachhochschule Darmstadt
Fachbereich Informatik

Softwaretechnik I

Kapitel 3 ff

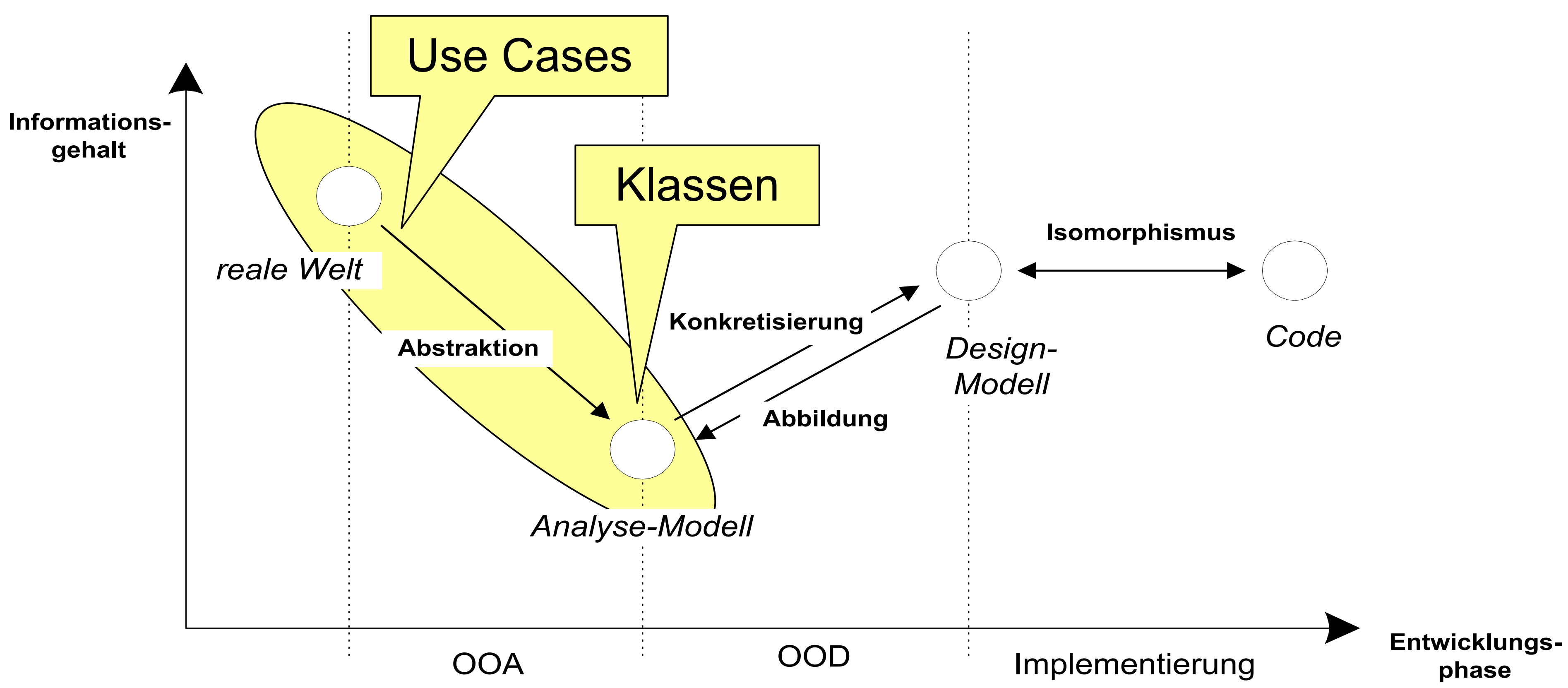
- Klassen & UML

Quellenhinweis:
Einige Folien zu dieser Vorlesung entstammen Präsentationen von Prof. G. Raffius und Prof. W. Weber

Stand im Phasenmodell

Objektorientierte Analyse

⇨ **Klassen**



Klassen und Beziehungen zwischen Klassen

Eigenschaften → **Klasse: Attribute** + *Verhalten* → **Operationen**

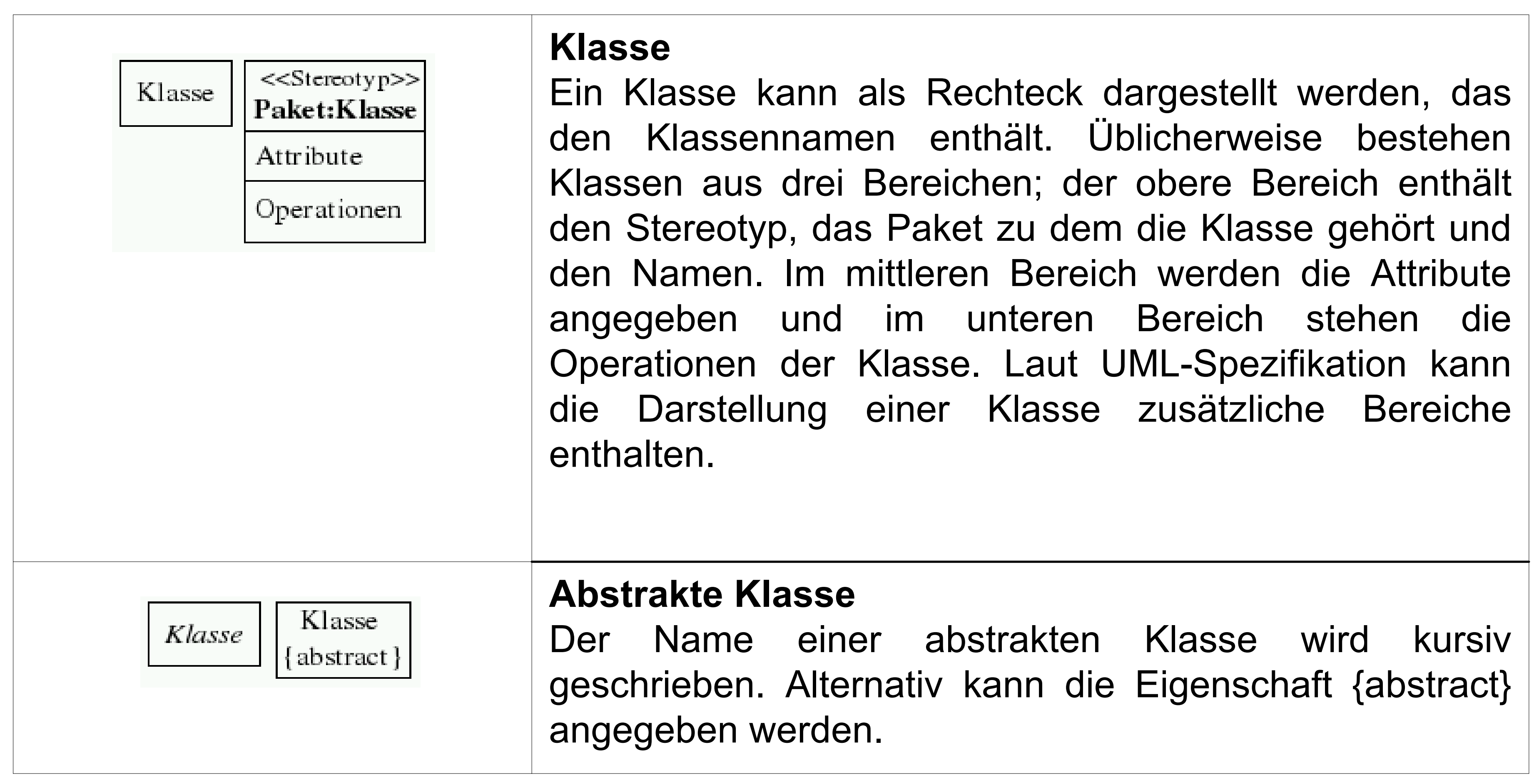
Klasse: **Attribute** **+** **Operationen**

Beziehungen zwischen Klassen:

- Generalisierung / Spezialisierung (Vererbung)
- Assoziation
- Aggregation
- Komposition

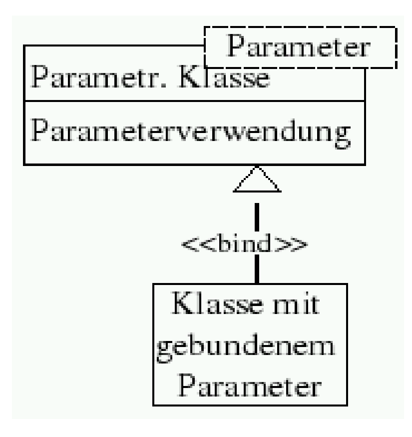
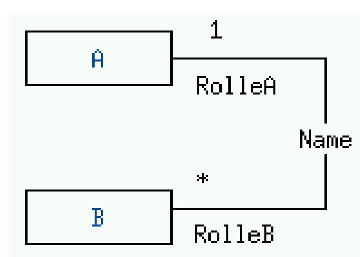
⇨ **Beschreibung der Statik des Systems**

Notation in UML 2.0 (I)

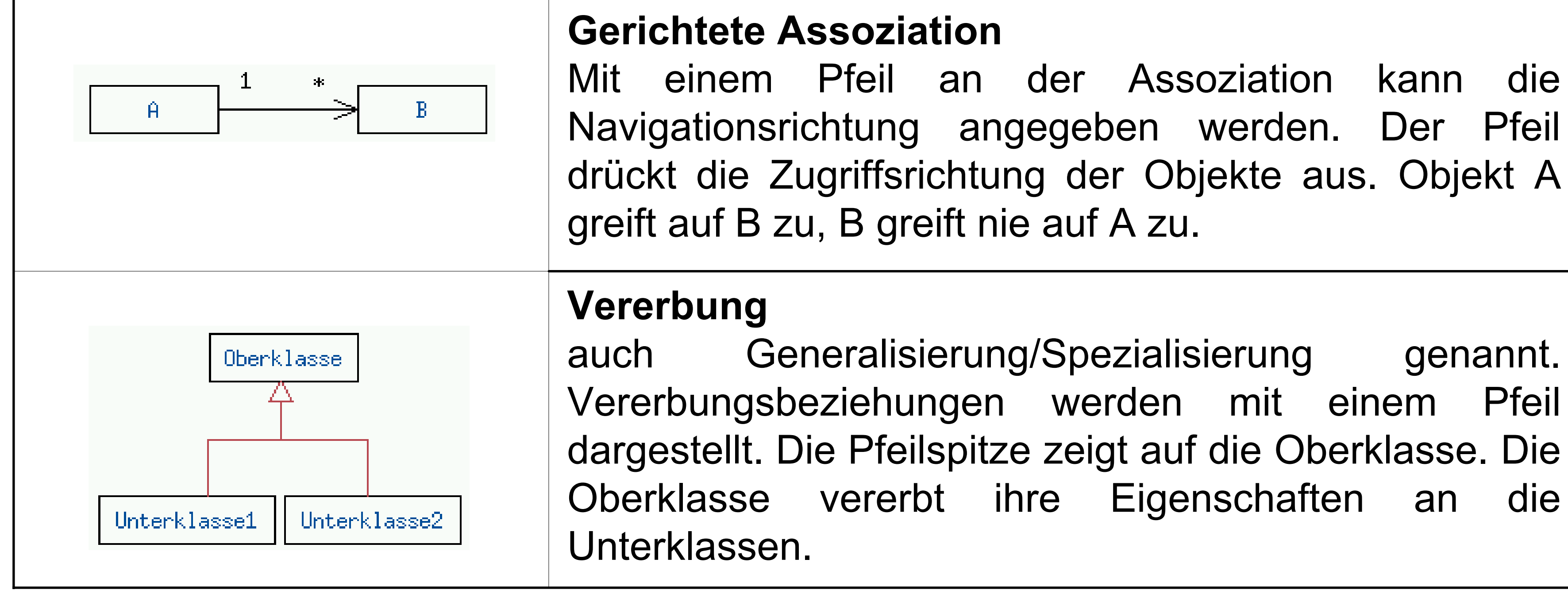


	Klasse Ein Klasse kann als Rechteck dargestellt werden, das den Klassennamen enthält. Üblicherweise bestehen Klassen aus drei Bereichen; der obere Bereich enthält den Stereotyp, das Paket zu dem die Klasse gehört und den Namen. Im mittleren Bereich werden die Attribute angegeben und im unteren Bereich stehen die Operationen der Klasse. Laut UML-Spezifikation kann die Darstellung einer Klasse zusätzliche Bereiche enthalten.
	Abstrakte Klasse Der Name einer abstrakten Klasse wird kursiv geschrieben. Alternativ kann die Eigenschaft {abstract} angegeben werden.

Notation in UML 2.0 (II)

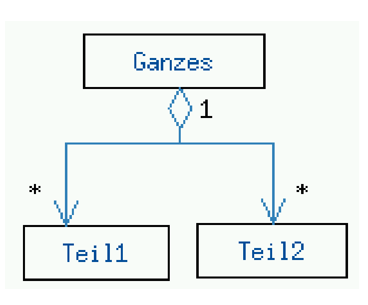
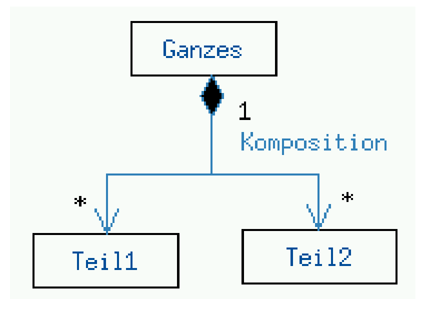
<table>
<tr>
<td>
</td>
<td>Parametrisierte Klasse
auch Template oder Schablone genannt. Die parametrisierte Klasse hat in der rechten, oberen Ecke ein das Klassensymbol überlappendes Rechteck, das die Schablonen-Parameter der Klasse enthält. Die Funktion, die den Parameter verwendet wird angegeben. Die Klasse, die den Parameter bindet, wird über eine gestrichelte Linie mit Pfeil an dem Template verbunden. Diese trägt die Bezeichnung <<bind>>.</td>
</tr>
<tr>
<td>
</td>
<td>Assoziation
Eine Linie zwischen den Klassen stellt eine Assoziation dar. Eine Assoziation ist eine Beziehung zwischen Klassen. Die Objekte der Klassen kommunizieren über die Assoziationen miteinander. Die Assoziation kann einen Namen haben. Ein Pfeil an dem Assoziationsnamen gibt die Leserichtung des Namens an. An den Assoziationsenden können die Rollen der beteiligten Klassen und die Multiplizität angegeben werden. Die zweigliedrige Assoziation kann, wie die mehrgliedrige Assoziation, durch eine Raute markiert werden.</td>
</tr>
</table>

Notation in UML 2.0 (III)

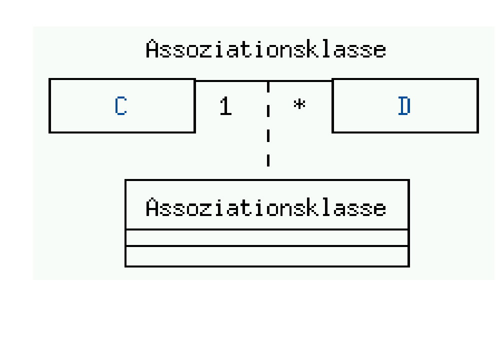
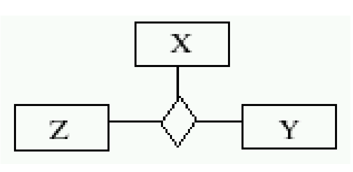


	Gerichtete Assoziation Mit einem Pfeil an der Assoziation kann die Navigationsrichtung angegeben werden. Der Pfeil drückt die Zugriffsrichtung der Objekte aus. Objekt A greift auf B zu, B greift nie auf A zu.
	Vererbung auch Generalisierung/Spezialisierung genannt. Vererbungsbeziehungen werden mit einem Pfeil dargestellt. Die Pfeilspitze zeigt auf die Oberklasse. Die Oberklasse vererbt ihre Eigenschaften an die Unterklassen.

Notation in UML 2.0 (IV)

<table>
<tr><td>
</td><td>Aggregation
Eine Aggregation drückt eine Teile-Ganzes-Beziehung aus. Das Ganze-Objekt besteht aus Teil-Objekten. Die Raute befindet sich an dem Ende des Ganzen. Die Aggregation ist eine spezielle Art der Assoziation. Da das Ganze die Teile enthält, sollten am Assoziationsende der Teile ein Navigationspfeil stehen.</td></tr>
<tr><td>
</td><td>Komposition
Die Komposition ist auch eine Beziehung, die Teile zu einem Ganzen in Beziehung setzt. Die Teile und das Ganze sind bei dieser Beziehung existenzabhängig; die Teile können nicht ohne das Ganze existieren. Wird das Ganze gelöscht, so beenden auch die Teile ihre Existenz.</td></tr>
</table>

Notation in UML 2.0 (V)

	**Assoziationsklasse** Ist eine Klasse von dem Vorhandensein einer Assoziation zwischen zwei Klassen abhängig, so kann dies durch eine Assoziationsklasse ausgedrückt werden. Die Assoziationsklasse beschreibt Eigenschaften, die keiner der an der Assoziation beteiligten Klassen sinnvoll zuordenbar sind. Die Assoziationsklasse wird über eine gestrichelte Linie mit der Assoziation, von der sie abhängt, verbunden. Hat die Assoziation einen Namen, dann muss die Assoziationsklasse den selben Namen erhalten. Die Assoziationsklasse ist ein Analysekonzept.
	**Mehrgliedrige Assoziation** Eine mehrgliedrige Assoziation drückt eine Beziehung zwischen mehr als 2 gleichwertigen Klassen aus. Die Beziehung wird mit einer Raute markiert.

Wie findet man Klassen ?

- ⇨ Aus den Beschreibungen der einzelnen Use Cases:
  Suche nach Substantiven, z.B.
  - Personen, Orte, konkrete Dinge
    (Artikel, Rechnungen, Abteilung, Meßwerte etc.)
  - Schnittstellen eines Systems
    (Masken, Anbindungen an Datenbanken etc.)
  - Abstrakte Dinge (ein Algorithmus, eine Information etc.)
- ⇨ Identifiziert zuerst Objekte
- ⇨ Aussondern überflüssiger Klassen. (Ist Person eigenes Objekt oder nur Rolle? (z.B. Klasse Person. Rollen: Kunde, Vorgesetzter etc.)
- ⇨ Spätere Ergänzungen aus Sequenzdiagrammen, Zustandsdiagrammen.
- ⇨ Weitere Hinweise für Klassen: Anwendungen von Entwurfsmustern (Design).

Wie findet man Operationen und Attribute ?

Attribute ergeben sich aus

⇨ Beobachtungen des Anwendungsgebiets
(Domäne) *(= Eigenschaften der Objekte)*

⇨ oder aus Vorgängersystemen.
(Formulare, Bildschirmmasken, Datenmodell etc.)

⇨ oder aus den Use Cases

⇨ und auch bei dynamischer Modellierung
(Szenarien, Zustandsdiagramme)

Operationen (Methoden)

⇨ Aus dynamischer Modellierung des Systems. *(Hier wird beschrieben, wie sich die Objekte des Systems verhalten und interagieren.)*

⇨ Ableitung aus Sequenzdiagrammen und Zustandsdiagrammen.

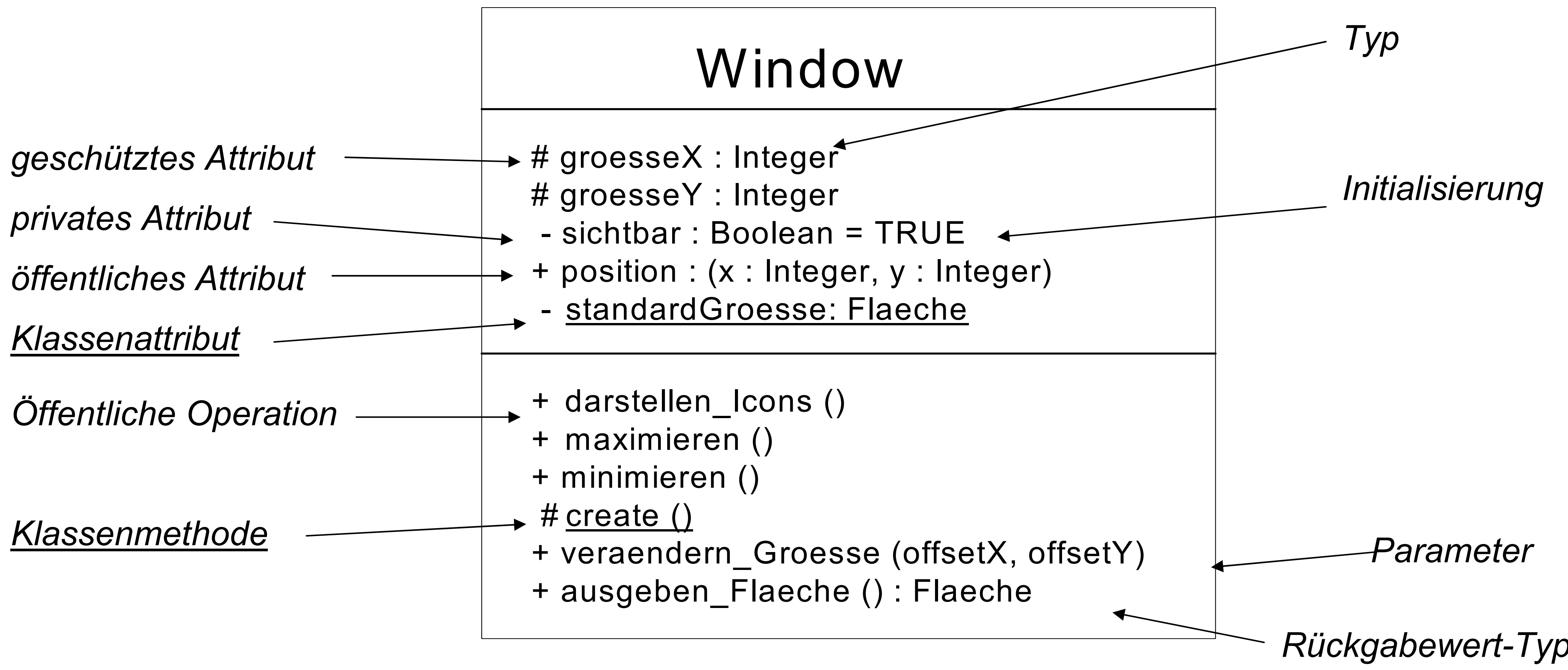
Wie findet man Klassen, Operationen und Attribute ? Beispiel (Diskussion)

Aus einer Use Case Beschreibung:

Ein Dauerauftrag überweist einen bestimmten Betrag in regelmäßigen Zeitabständen von einem Girokonto auf ein Anderes.
Der Dauerauftrag gehört immer zu einem bestimmten Girokonto. Daueraufträge können angelegt und wieder gelöscht werden.

⇨ Dauerauftrag	Klasse
⇨ Betrag	Attribut für Klasse Dauerauftrag
⇨ Zeitabstand	Attribut für Klasse Dauerauftrag
⇨ Girokonto	Assoziation von Dauerauftrag? Oder Attribut?
⇨ Anderes (Girokonto)	Attribut für Klasse Dauerauftrag
⇨ Anlegen	Operation für Klasse Girokonto? Oder Konstruktor?
⇨ Löschen	Operation für Klasse Girokonto? Oder Destruktor?

Darstellung von Klassen, Attributen und Methoden



Anzeigemodus kann im CASE-Tool eingestellt werden!

Darstellung von Klassen, Attributen und Methoden (II)

Klassenmethode:

- ⇨ Bezieht sich nicht auf Instanz (Instanzmethode) sondern auf die Klasse als die Menge aller ihrer Objekte.
  z.B.: Konstruktor, Berechnung Durchschnittsgehalt etc.

Klassenattribute:

- ⇨ Attributwert ist nicht einer Instanz (Instanzattribut) sondern der Klasse zugeordnet und existiert nur 1x für die Klasse.
  z.B.: maxGeschwindigkeit eines Fahrzeugtyps

Methoden für das Schreiben und Lesen von Attributen:

- ⇨ Sind trivial ⇒ werden nicht in die Klassendef. auf Analyse-Ebene aufgenommen.

Bei Verwendung eines CASE-Tools:

- ⇨ Einzelangaben können, um die Übersicht des Gesamtdiagramms zu erhöhen, ausgeblendet werden.

Vererbung / Generalisierung / Spezialisierung

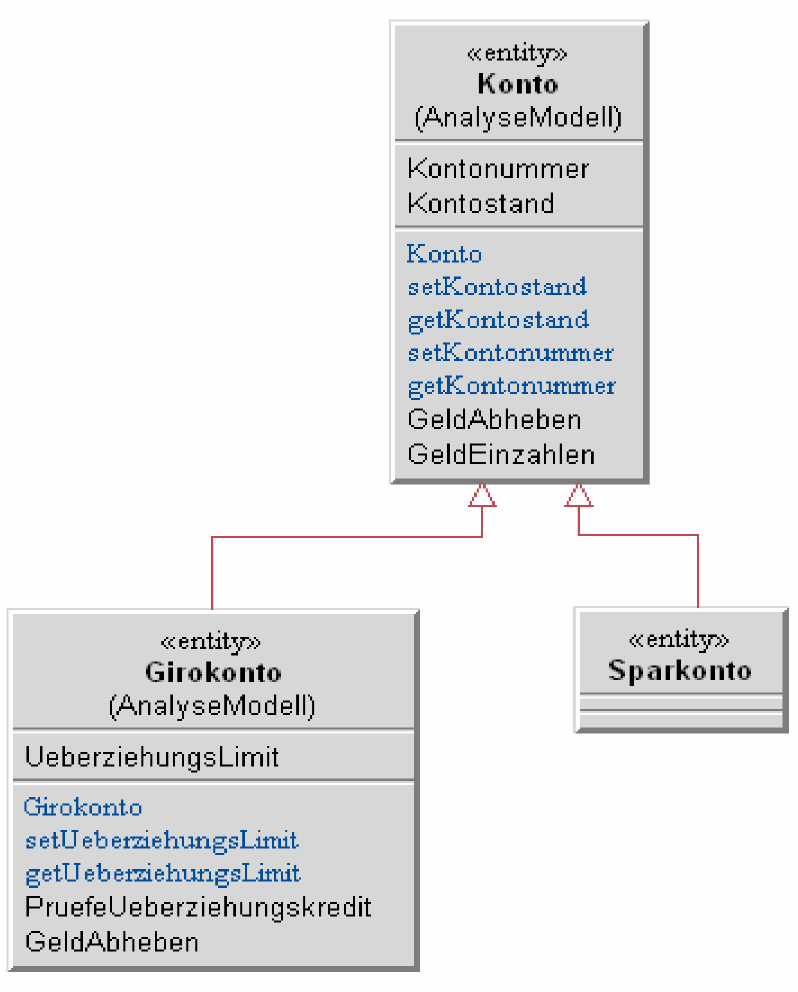
Aus der Menge der Klassen sucht man nach Klassenpaaren wobei:

- ⇨ Die eine Klasse (Superklasse) ist ein allgemeinerer Typ und die andere Klasse (Subklasse) ist ein speziellerer Typ, den die Superklasse erweitert.

  ⇨ Es besteht eine „Ist-Ein“ – Beziehung !

- ⇨ Ein Objekt der Superklasse ist durch ein Objekt der Subklasse ersetzbar!

- ⇨ Unter der Subklasse ist eine Teilmenge der Menge der Instanzen der Superklasse angesiedelt.



Vererbung / Generalisierung / Spezialisierung

Die Subklasse besitzt alle Merkmale (Attribute, Operationen, Assoziationen, Aggregationen) der Superklasse und noch zusätzliche Merkmale.

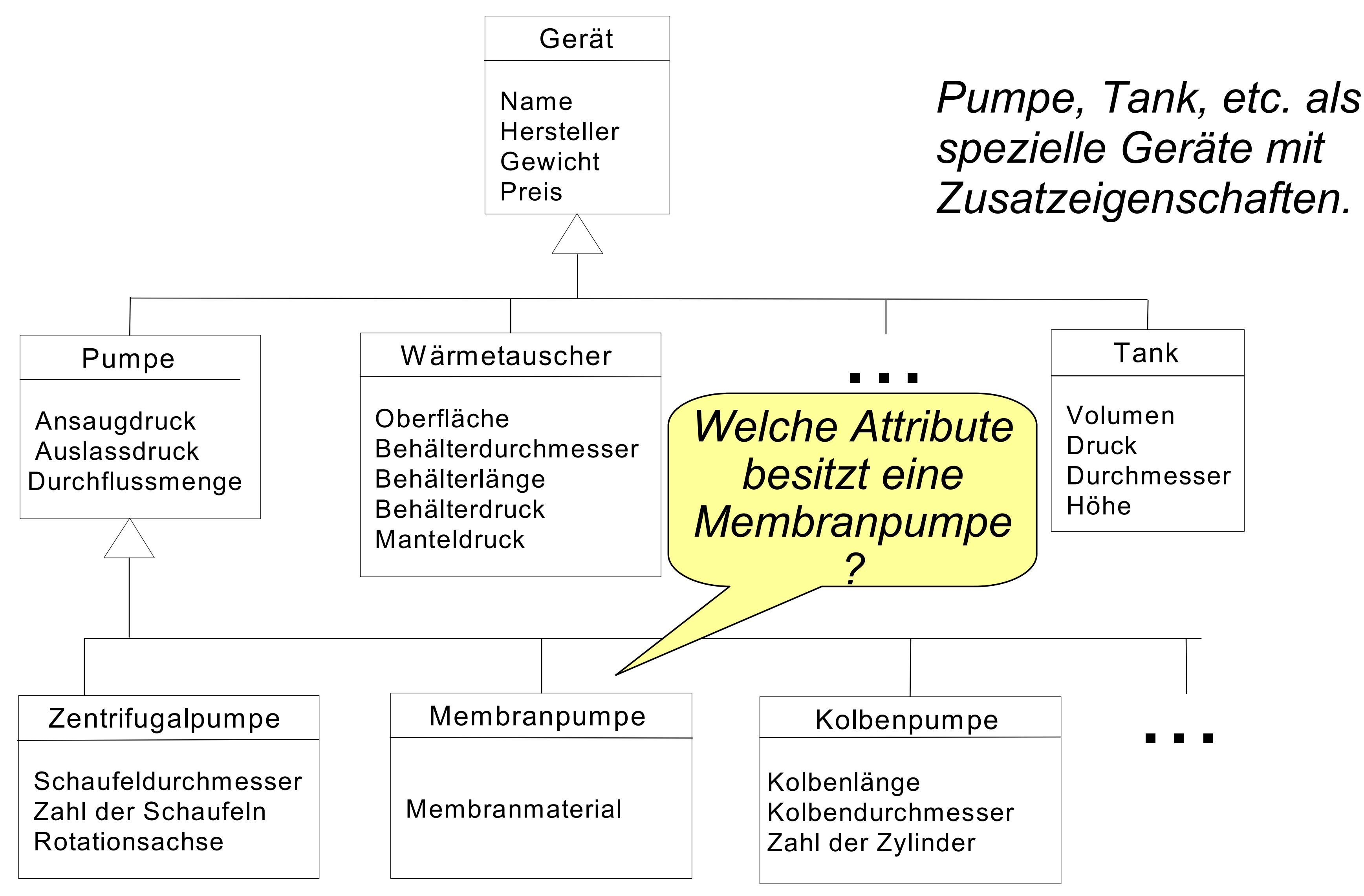
⇨ *Die Subklasse erbt die Merkmale der Superklasse.*

Vererbung: Alle Merkmale werden von der Superklasse auf die Subklasse vererbt.

Generalisierung: "Ist-ein-Beziehung" von der Subklasse aus gesehen.
Ein Konto ist ein (verallgemeinertes) Girokonto

Spezialisierung: "Ist-ein-Beziehung" von der Superklasse aus gesehen.
Ein Girokonto ist ein (spezialisiertes) Konto

Vererbung / Generalisierung / Spezialisierung: Beispiel



Pumpe, Tank, etc. als spezielle Geräte mit Zusatzeigenschaften.

Frage: Welche Attribute besitzt das Exemplar Membranpumpe ?

Instanzen:

(Membranpumpe)

Name = P101
Hersteller = Simplex
Gewicht = 100 kg
Preis = 8.000 €
Ansaugdruck = 1,1 atm
Auslassdruck = 3,3 atm
Durchflussrate = 300 l/h
Membranmat. = Teflon

(Wärmetauscher)

Name = E302
Hersteller = Braun
Gewicht = 5000 kg
Preis = 30.000 €
Oberfläche = 300 m
Behälterdurchm. = 2 cm
Behälterlänge = 6 m
Behälterdruck = 15 atm
Manteldruck = 1,7 atm

(Tank)

Name = T111
Hersteller = Simplex
Gewicht = 10.000 kg
Preis = 80.000 €
Volumen = 400.000 l
Druck = 1,1 atm
Durchmesser = 8 m
Höhe = 9 m

Vererbung / Generalisierung / Spezialisierung: Sichtbarkeit

- Man unterscheidet (allgemein) 4 verschiedene Sichtbarkeitsstufen:
  - Private: außerhalb der Klasse nicht sichtbar
    (in der UML: -), auch nicht für Unterklassen
  - Protected: für alle Unterklassen sichtbar
    (in der UML: #)
  - Public: für alle anderen Klassen sichtbar, d.h. öffentlich
    (in der UML: +)
  - Package: für Klassen im gleichen Paket sichtbar
    (in der UML: ~)
- Verschiedene Stufen der Sichtbarkeit innerhalb einer Vererbungshierarchie
- Jede Klasse „kennt“ nur ihre eigenen Attribute, Operationen und Assoziationen und die ihrer Oberklassen, sofern diese für sie sichtbar sind

Vererbung / Generalisierung / Spezialisierung: Regeln

- In Subklassen dürfen nicht nur zusätzliche Merkmale definiert werden, sondern auch geerbte Merkmale überschrieben werden:

  - Operationen (Reimplementierung / Polymorphie)
  - Attribut-Typen
  - Argumente und Rückgabewerte von Operationen

Aber:

Durch ein Überschreiben darf

nie die Semantik des Attributs bzw. der Operation geändert werden!

(Die Ist-ein-Beziehung darf nie verletzt werden !)

Einschränkungen bezüglich des Überschreibens der Merkmale (I)

- **Der Wertebereich von Attributen darf eingeschränkt, jedoch nicht erweitert werden.**

  z.B.: Superklasse Mitarbeiter: Gehalt 1... 150.000,- €
  Subklasse Arbeiter: Gehalt 1... 80.000,- €

- **Der Typ eines Attributs der Subklasse muss vom Typ der Superklasse isomorph umschlossen sein.**

  z.B.: Superklasse Attribut von Typ REAL
  in Subklasse Attribut von Typ INTEGER

  z.B.: Superklasse Attribut von Typ der Klasse KONTO
  in Subklasse Attribut von Typ einer Subklasse von KONTO (z.B.: GIROKONTO)

Einschränkungen bezüglich des Überschreibens der Merkmale (II)

- **Integritätsbedingungen einer Klasse dürfen nicht verletzt werden:**
  z.B.: Ellipse-Kreis:
  Eine Ellipse ist keine Spezialisierung eines Kreises, da die Kreiseigenschaft zweier gleichlanger Achsen durch die Ellipse verletzt würde. Ein Kreis kann jedoch eine spezielle Ellipse sein, bei der die Achsen gleich lang sind.

- **Überschreiben von Methoden:**
  Andere Implementierungen (z.B. Performance-Verbesserung) mit gleicher Funktionalität sind erlaubt. Die Schnittstelle (Name, Anzahl von Argumenten) der Operation der Superklasse muss jedoch eingehalten werden. Typen von Argumenten und Rückgabewerten müssen den o.g. Bedingungen für das Überschreiben von Attributen genügen.

Hinweise zum Finden von Generalisierungs- & Spezialisierungs-Beziehungen

Gleiche Attribute, Operationen, Assoziationen, Aggregationen in verschiedenen Klassen

⇒ gemeinsame Merkmale in der Superklasse.

Aber: Es muss eine „ist-ein-Beziehung“ bestehen und darf nicht nur alleine der Erleichterung der Implementierung dienen.

Bsp.: *Dieselben Attribute, aber keine „ist-ein-Beziehung“.*

1. Klasse Fenster und Klasse Rechteck:

Attribute: Fläche und Größe

Aber haben aus semantischer Sicht keine gemeinsame Superklasse.

2. Klasse Rollstuhl und Klasse Kraftfahrzeug:

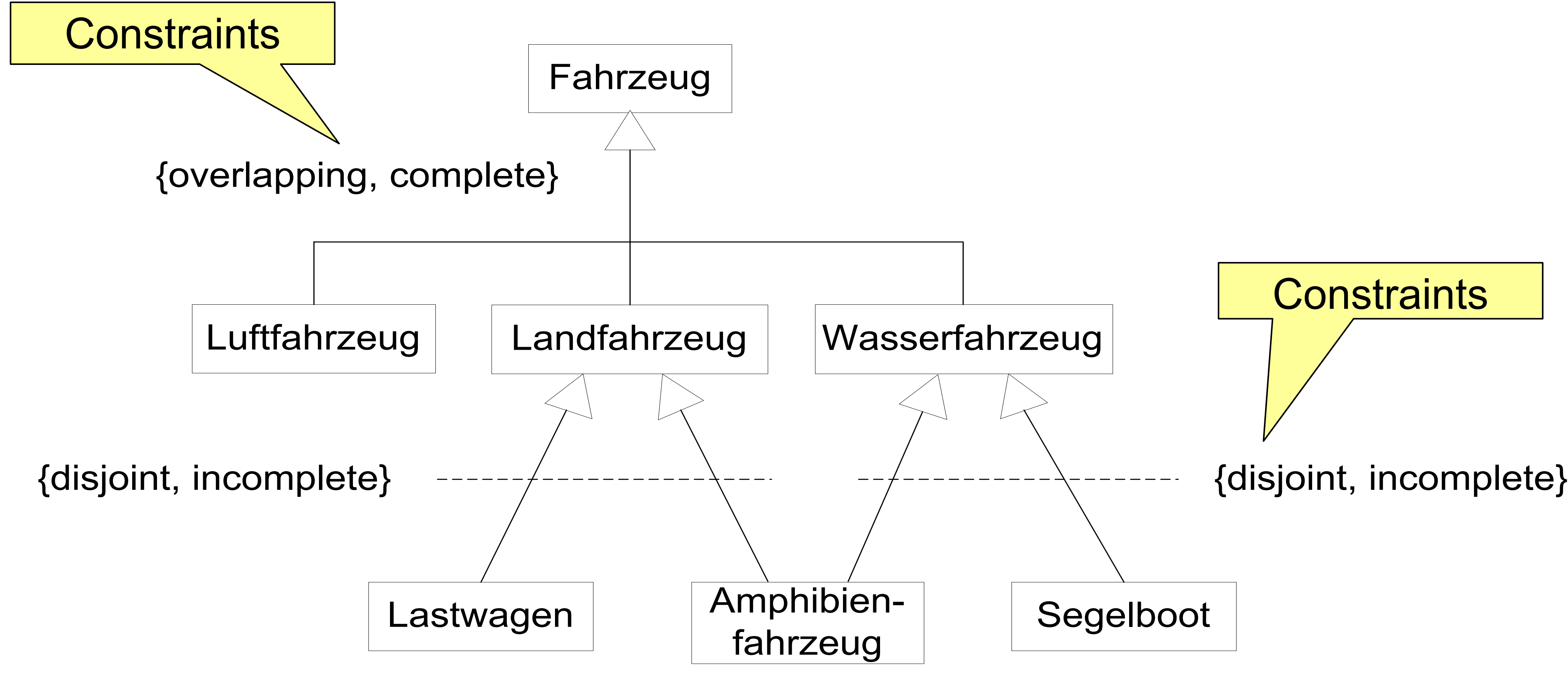
Attribute: Gewicht, Anzahl Räder, Farbe

Operation: fortbewegen

Kraftfahrzeug besitzt zusätzliches Attribut kW

Trotzdem ist Kraftfahrzeug keine Spezialisierung eines Rollstuhls.

Randbedingungen für Vererbung / Mehrfachvererbung



Default: {overlapping, incomplete} – wird weggelassen

Unterschiede zwischen Methoden und Operationen

Operation: Ein best. Verhalten, das eine Klasse für ihre Objekte definiert.
Dazu gehören auch alle geerbten Operationen.
Eine Operation definiert Schnittstelle sowie Vor- und Nachbedingungen eines bestimmten Verhaltens.

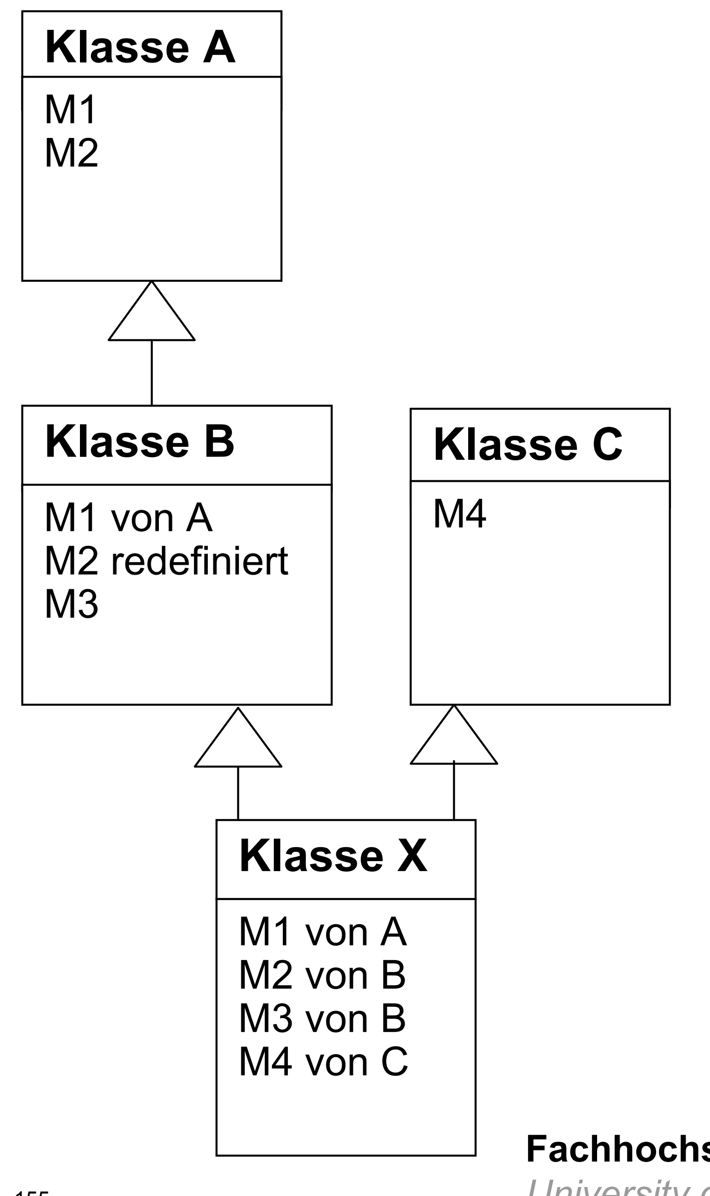
Methode: Die Implementierung/Ausprägung einer Operation.
Die Implementierung muss die Vorgaben der Operationsdefinition einhalten.

Bsp.: graph. Objekt: *Operationen:* *berechneFlaeche(), drehe()*
Methoden: *g*h (Rechteck), π r² (Kreis), entspr. Operation zum Neuzeichnen*

Unterschiede zwischen Methoden und Operationen (Aufgabe)

Frage:

In welchen Klassen sind
M1 und M2 Methoden?
In welchen Operationen?



Die Unterscheidung wird häufig nicht gemacht!

Die Begriffe werden synonym verwendet!

Abstrakte Klassen (I)

Eine abstrakte Klasse ist eine Klasse, in der mindestens eine abstrakte Operation vorkommt.

abstrakte Operation:

- besitzt keine Implementierung (keine Methode)
- definiert nur Schnittstelle, Vor- und Nachbedingungen

⇒ **abstrakte Klassen können keine Instanzen bilden.**

Wozu ?

Klassenhierarchien sind „Begriffshierarchien".

Abstrakte Klassen dienen dem Finden von Oberbegriffen für Klassen, um allgemeingültige Klassenhierarchien und intuitives Verständnis zu erreichen.

⇒ **Abstrakte Klassen definieren Schnittstellen für Verhalten !**

Abstrakte Klassen (II)

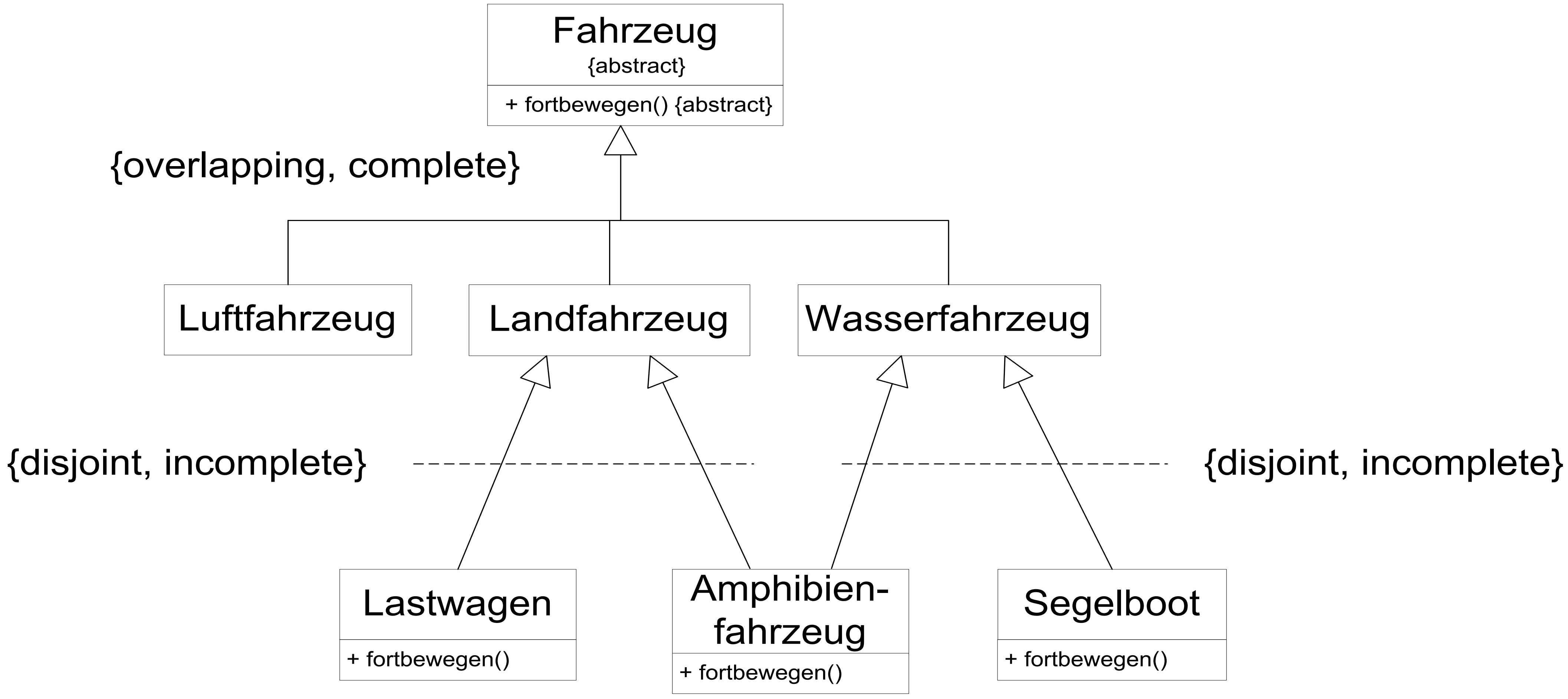
Klassen, die nicht abstrakt sind und Instanzen bilden können, werden *konkrete* Klassen genannt.

Konkrete Klassen, die von abstrakten Klassen erben, definieren die abstrakten Operationen ihrer Superklassen, d.h. sie implementieren Methoden dafür.

⇨ Polymorphie
gleiche Methoden anwendbar auf unterschiedliche Objekte (innerhalb einer Vererbungshierarchie)

⇨ dynamisches / spätes Binden
Die Verknüpfung zwischen Methodenaufruf und Methodenimplementierung findet erst zur Laufzeit statt (Wird nicht von allen Sprachen unterstützt!)

Abstrakte Klassen (Beispiel)



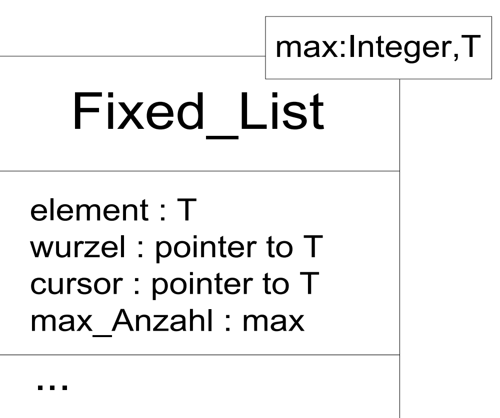
Parametrisierte Klassen (Templates) (I)

Was ist das ?

Bei der Definition einer Instanz einer parametrisierten Klasse:

⇨ Angabe konkreter Parameter ⇒ Ausprägung zur Kompilierzeit.
Dadurch ist es möglich typenunabhängige Algorithmen zu definieren und zur Kompilierzeit zu konkretisieren.

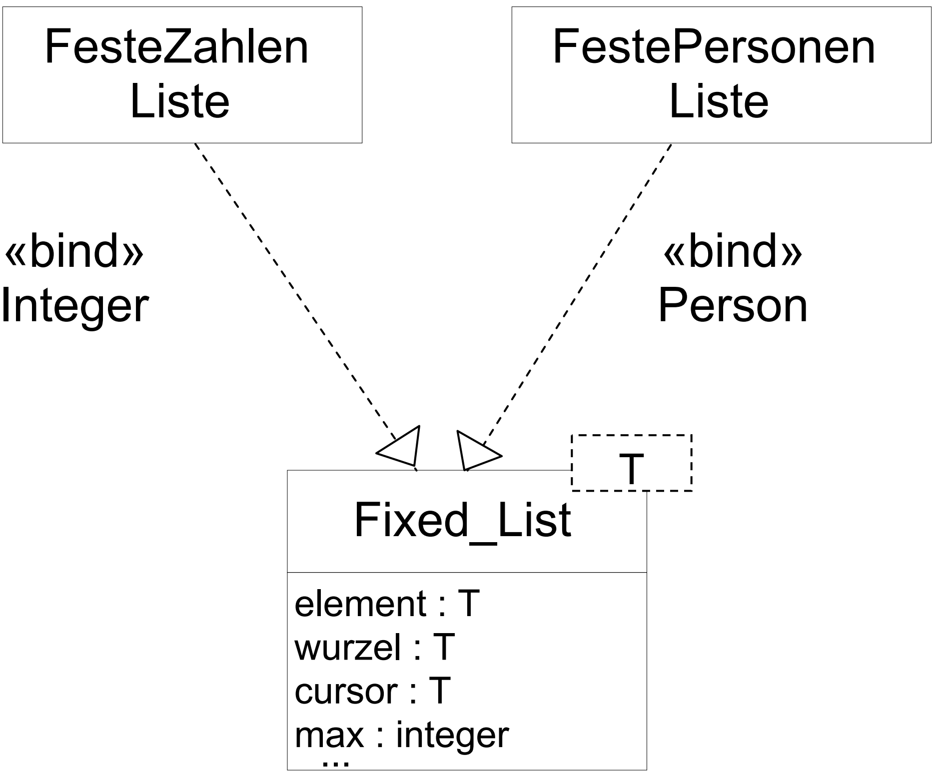
z.B.: Fixed_List:
*Instanz kann die unter **max** angegebene Anzahl von Elementen von dem angegebenen Typ (Integer, String, Kunde,...) enthalten.*



Anwendung:
Standardalgorithmen und Datenstrukturen
z.B. Graphen, Bäume, Listen, Sortierungen, Suche

Parametrisierte Klassen (Templates) (II)

Eine parametrisierte Klasse kann nicht direkt von anderen Klassen benutzt werden. Ihre generischen Parameter müssen erst an einen konkreten Typ „gebunden“ werden. Diese gebundenen Klassen können dann von anderen Klassen assoziiert werden.



Diskriminator (Klassifizierungsmerkmal)

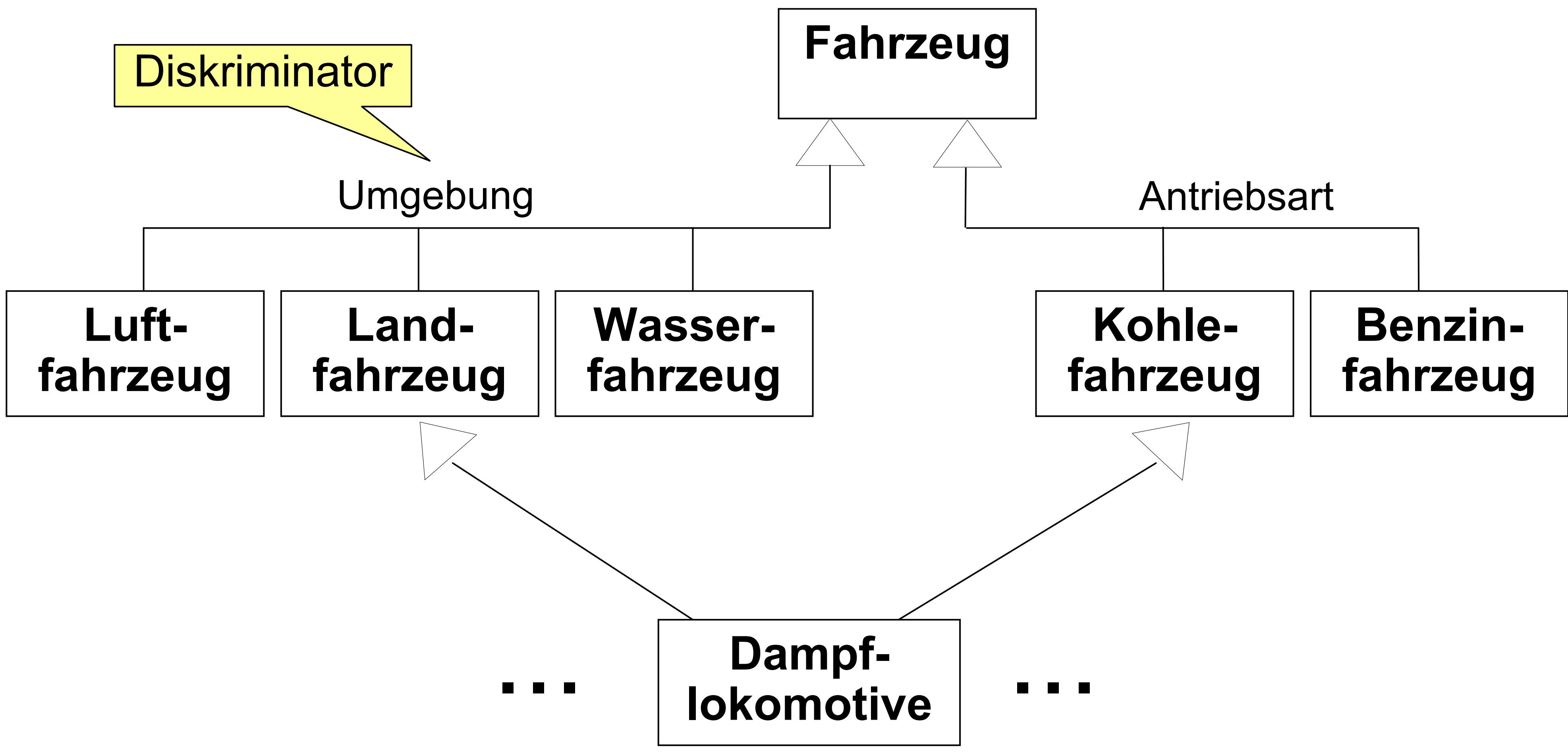
- Unterscheidungsmerkmal für die Strukturierung der Modellsemantik in Generalisierungs- bzw. Spezialisierungsbeziehungen

- Es kann zu einer Klasse eine oder auch mehrere Spezialisierungen (Klassifizierungen) geben.

Bsp.: *Fahrzeug (s.o.):*
Klassifizierung nach Umgebung, in der sich das Fahrzeug bewegt (eine Klassifizierung).

oder:
zusätzliche Klassifizierung nach Antriebsart (zweite Klassifizierung)

Diskriminator (Klassifizierungsmerkmal) (Beispiel)



Durchgängiges Beispiel

- *In diesem Kapitel werden die theoretisch eingeführten Konzepte mit Hilfe eines Beispiels detailliert erläutert. Das Beispiel ist so gewählt, dass es durch das gesamte Kapitel weiterentwickelt und zu einem konsistenten objektorientierten Modell integriert wird.*
- *Die aufeinander aufbauenden Teile des Beispiels in den einzelnen Abschnitten sind so formuliert, dass die Ziele, die zu erreichen sind, in Form einer Aufgabenstellung beschrieben werden. Danach folgt die fachliche Beschreibung der zu modellierenden Inhalte.*
- *Sie sollten die Lösung selbst erarbeiten. Die Lösung wird direkt im Anschluss präsentiert.*

Durchgängiges Beispiel (II)

***Aufgabe**:*

Analysieren Sie die nachfolgenden verbal formulierten Anforderungen an das System Hochschulverwaltung:

- *Identifizieren Sie die Klassen des Problembereichs (Domäne) und ihre Attribute durch Textanalyse.*
- *Identifizieren Sie die Generalisierungs-/Spezialisierungs-Beziehungen zwischen den Klassen.*
- *Integrieren Sie die Klassen zu einer Vererbungshierarchie (Klassendiagramm).*

***Anforderungen**:*

In einer Hochschulverwaltung sind mehrere Personengruppen tätig. Die Hochschule hat Angestellte, die Professoren, Labor-Ingenieure, Lehrbeauftragte, Sekretärinnen oder Tutoren sein können. An einer Hochschule studieren Studenten, die auch als Tutoren in einzelnen Lehrveranstaltungen eingesetzt werden können. Jede Person hat einen Namen, Geburtsdatum und Geburtsort. Alle Angestellten verfügen über ein Gehaltskonto. Dozenten haben einen akademischen Titel und leisten pro Semester eine bestimmte Anzahl Semesterwochenstunden (SWS).
Jeder Student hat eine identifizierende Matrikelnummer.

Durchgängiges Beispiel (Lösung)

*Zur Identifizierung der Klassenkandidaten wird zunächst eine **Liste aller Substantive** der Problembeschreibung erstellt:*

Hochschulverwaltung, Personengruppen, Hochschule, Angestellte, Professoren, Labor-Ingenieure, Lehrbeauftragte, Sekretärinnen, Tutoren, Studenten, Person, Namen, Geburtsdatum, Geburtsort, Gehaltskonto, Dozenten, Titel, Semester, Anzahl, Semesterwochenstunden, Matrikelnummer.

*Als nächstes folgt die **Eliminierung überflüssiger Begriffe** (redundant oder irrelevant):*

Hochschulverwaltung, Personengruppen, Hochschule, Semester, Anzahl

Folgende Begriffe werden als Klassenkandidaten eliminiert, weil sie Eigenschaften (Attribute) anderer Substantive (Klassenkandidaten) bezeichnen:

Namen, Geburtsdatum, Geburtsort, Gehaltskonto, Titel, Semesterwochenstunden, Matrikelnummer.

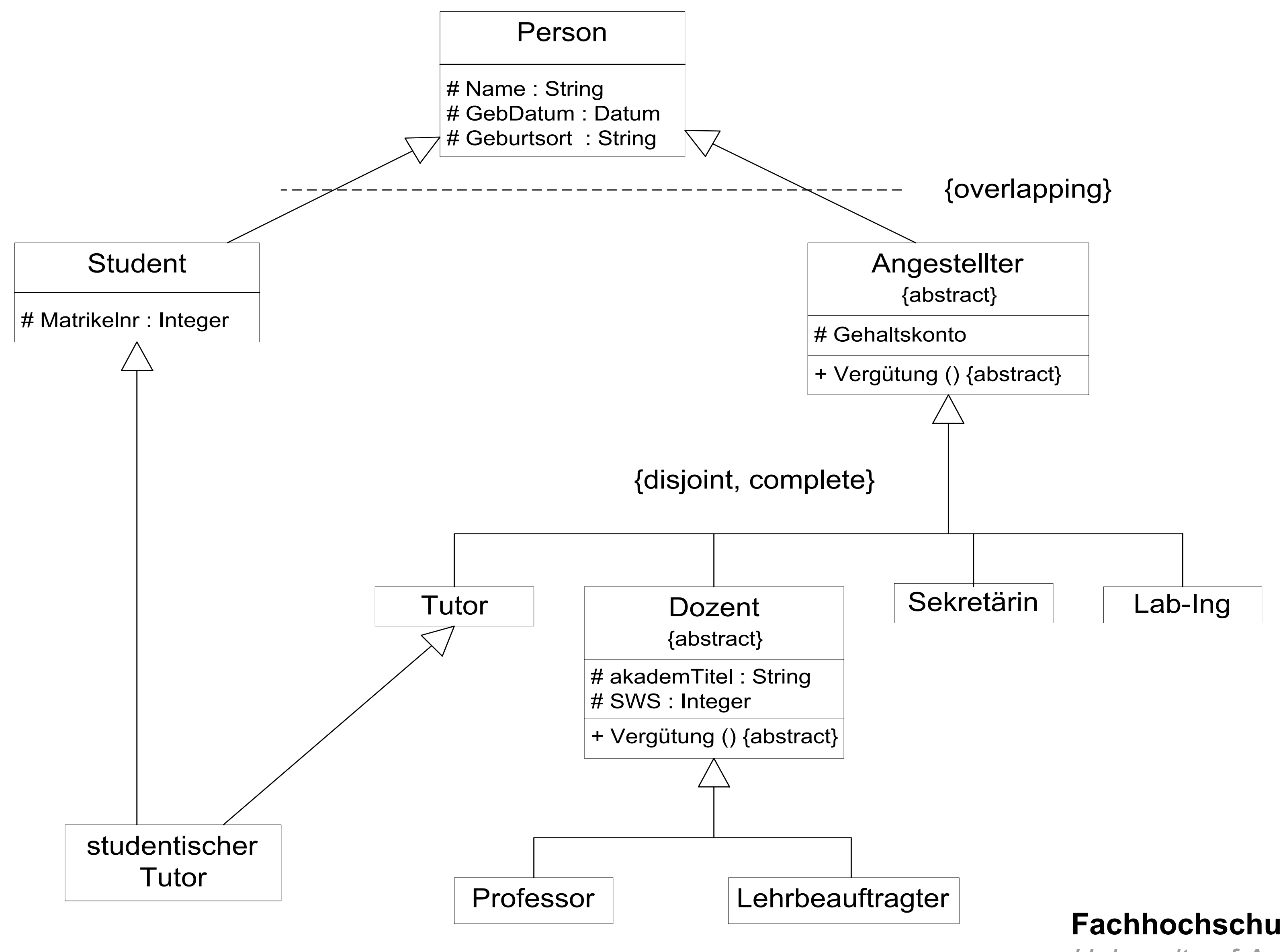
Durchgängiges Beispiel (Lösung)

Daraus ergibt sich dann folgende grobe Klassenstruktur mit Attributen

Klasse	***Attribute***
Person	*Name, Geburtsdatum, Geburtsort*
Student	*Matrikelnummer*
Angestellter	*Gehaltskonto*
Tutor	
Dozent	*Titel, Semesterwochenstunden*
Sekretärin	
Labor-Ingenieur	
Professor	
Lehrbeauftragter	
Studentischer Tutor	

Selbstverständlich haben noch andere Klassen Attribute. Diese sind aber den Dokumenten (Anforderungen), die in dieser Projektphase (Analyse) vorliegen, noch nicht zu entnehmen. Deswegen werden sie zunächst offengelassen und in späteren Projektphasen ergänzt (iterativer Entwicklungsprozess).

Durchgängiges Beispiel (Lösung)



Durchgängiges Beispiel (Offene Punkte)

- *Was passiert, wenn ein Student eine Tutorenstelle annimmt?*
  ⇨ *Das Objekt „Student" müsste zerstört und ein neues Objekt der Klasse „StudentischerTutor" erzeugt und mit denselben Daten belegt werden.*

- *Was passiert, wenn ein Labor-Ingenieur einen Lehrauftrag erhält?*

⇨ Wie kann geschickter modelliert werden???

Assoziationen

- Logische Beziehungen zwischen Klassen („sich-kennen“)

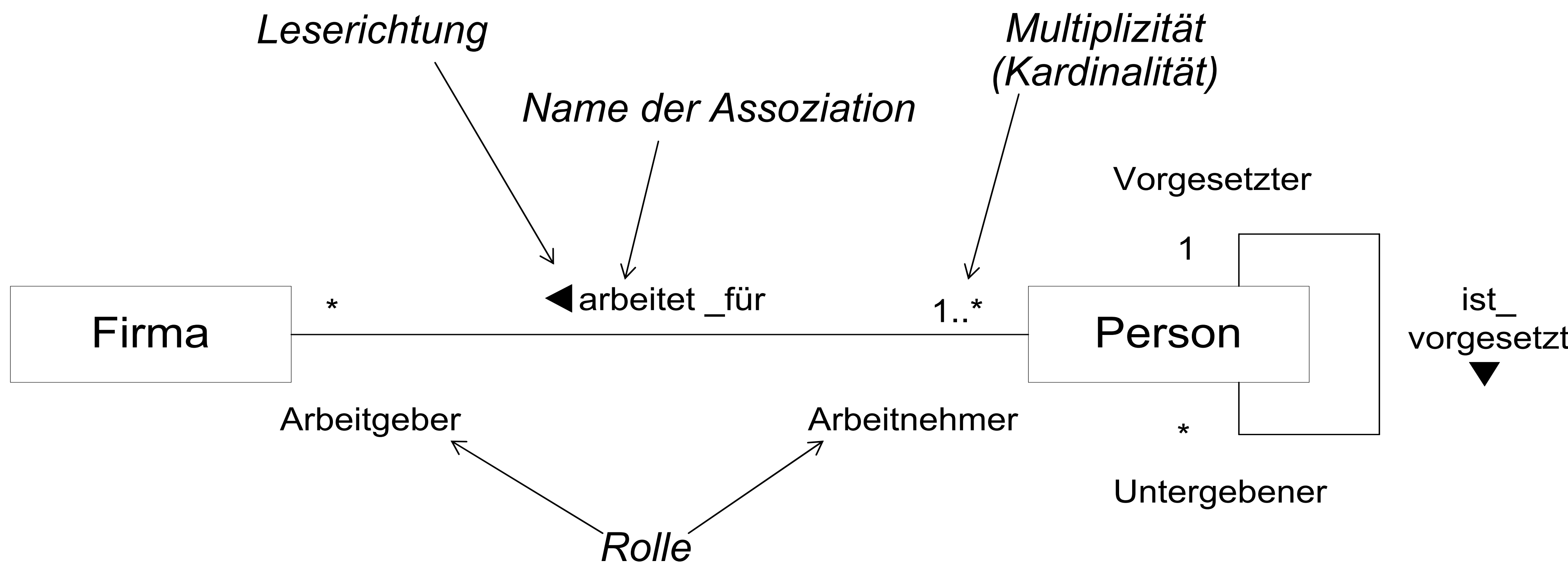
  ⇨ Verknüpfungsmöglichkeit zwischen Objekten dieser Klassen

- Definiert ebenso mögliche Kommunikationswege zwischen Objekten der Klassen.

- **Verbindung zwischen Objekten:**

  ⇨ Stets als Assoziation und nicht als Zeiger-Attribut modellieren! (Abstraktionsebenen)
  ⇨ Implementierung erst später auf niedrigerer Abstraktionsebene
  ⇨ Multiplizität kann angegeben werden
  ⇨ Leserichtung angeben, wenn sie von links⇨rechts abweicht

Assoziationen (Beispiel)



- *1 Firma (als Arbeitgeber) kennt 1..n Personen (als Arbeitnehmer)*
- *1 Person (als Arbeitnehmer) kennt bzw. arbeitet für 0..n Firmen (als Arbeitgeber)*
- *1 Person (als Vorgesetzter) kennt bzw. ist_vorgesetzt für 0..n Personen (als Untergebene)*
- *1 Person (als Untergebener) kennt 1 Person (als Vorgesetzten)*

Wie findet man Assoziationen?

a) Use Case Analyse und Dokumentenanalyse

Verben in Dokumenten (Beschreibungen, Dokumentationen, Spezifikationen Interview-Protokoll etc.)

Aber: Nicht alle Assoziationen sind für den zu modellierenden Problembereich wichtig!

b) Weitere Hinweise auf Assoziationen:

- Verwaltung von Dingen (Firma hat Kunden, Abteilung gliedert sich in Gruppen)
- Besitz (Motor besitzt Benzinpumpe, Auftrag besteht aus Positionen)
- Kommunikation zwischen Objekten

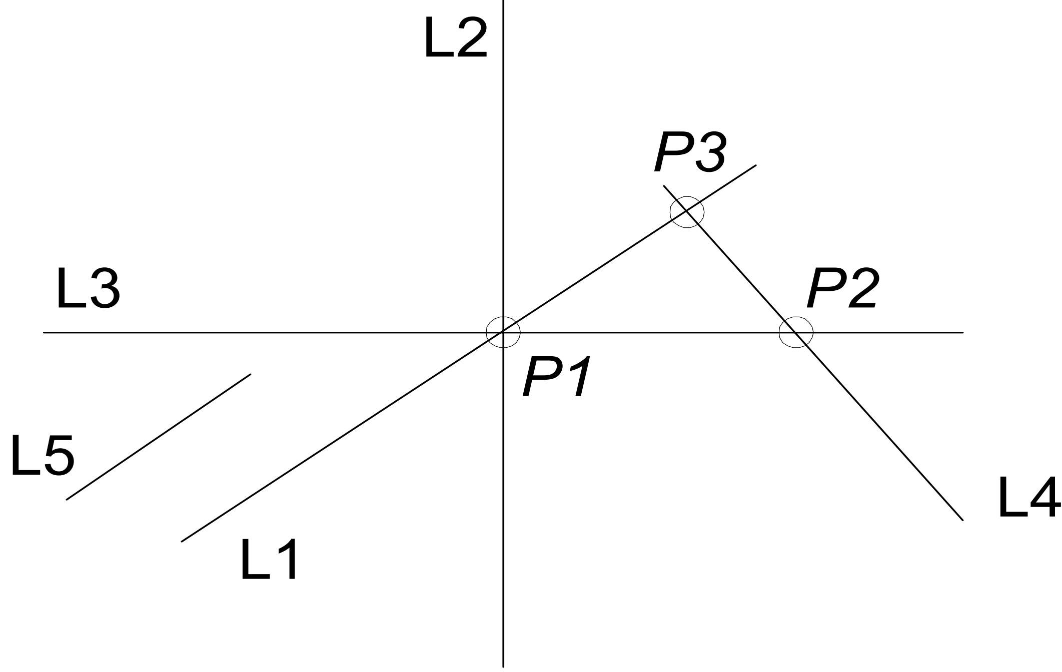
c) In späteren Schritten:

Bei dynamischer Modellierung wird Kommunikation zwischen Klassen dargestellt: Kommunikation läuft über vorhandene Assoziation – oder auch nicht

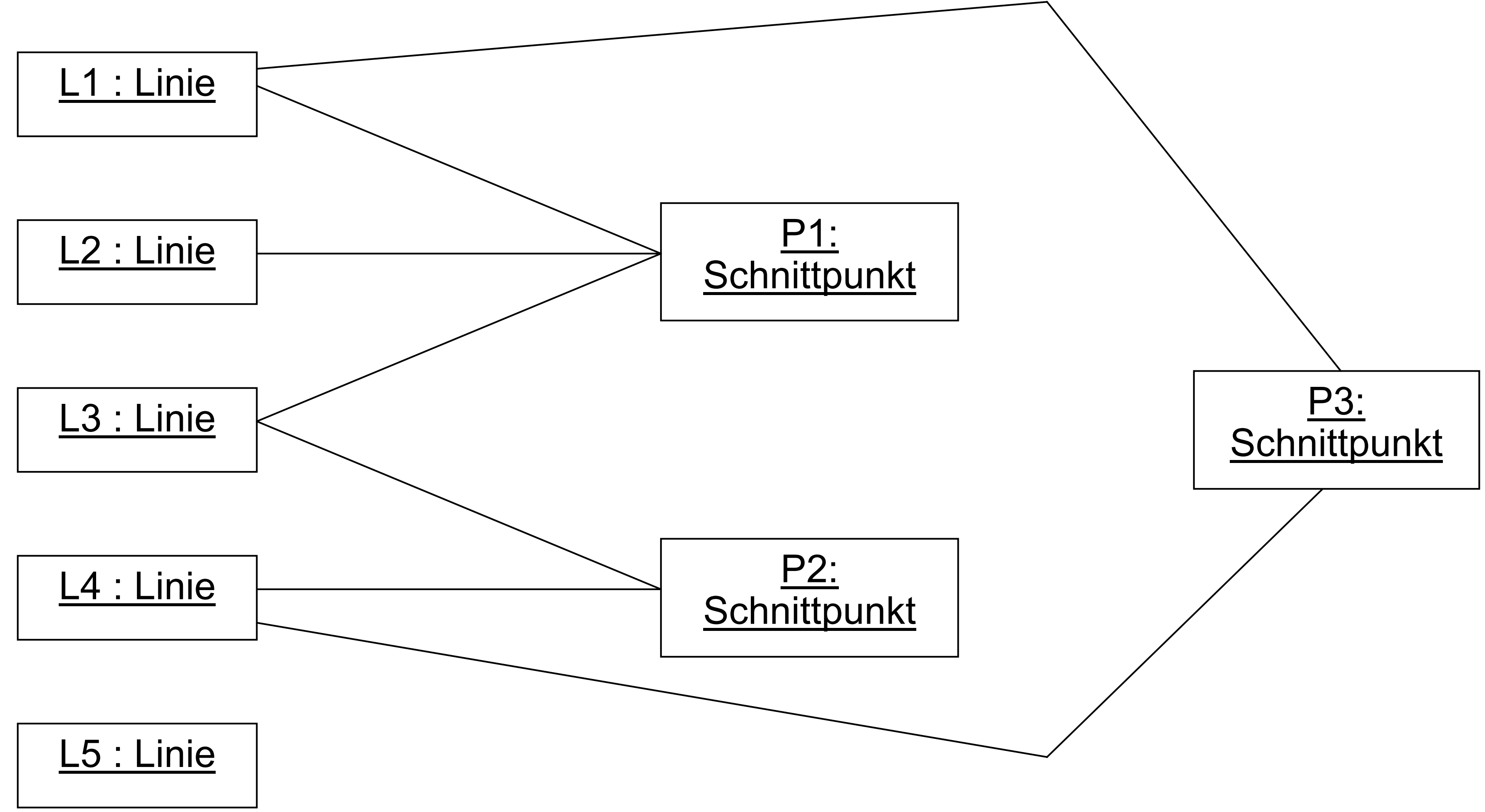
Falls nicht ⇨ zusätzliche Assoziation !

Assoziationen (Übung)

In der nachfolgenden Abbildung sind Linien gegeben, die bestimmte Schnittpunkte haben. Erstellen Sie ein Klassendiagramm mit den Klassen ´Schnittpunkt´ und ´Linie´ und einer Assoziation, die aussagt, dass Linien sich in einem Punkt schneiden.



Assoziationen (Lösung – Vorarbeit durch Betrachtung der Objekte)

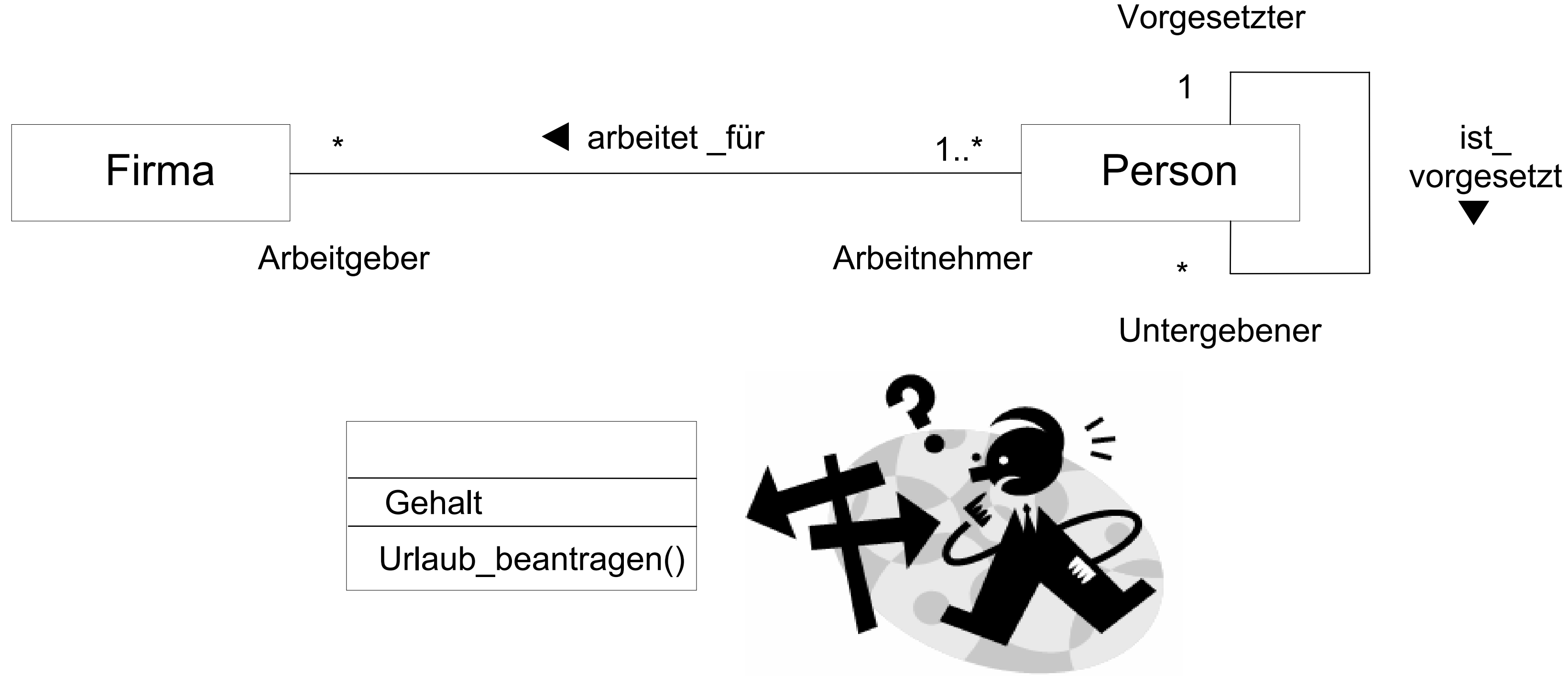


Assoziationen (Lösung - Klassendiagramm)

Erklärung:

Die Linien und Schnittpunkte sind Exemplare der Klassen ´Linie´ und ´Schnittpunkt´. Die Linien L1, L3 und L4 besitzen jeweils zwei Schnittpunkte mit anderen Linien (P1,P3 bzw. P1,P2 bzw. P3,P2), die Linie L2 schneidet zwei Linien im Punkt P1, die Linie L5 schneidet keinen Punkt.

Assoziationen



Betrachten Sie das Attribut *Gehalt* und die Methode *Urlaub_beantragen*

⇨ Wie und wo sollten diese Elemente modelliert werden?

Assoziationsklassen

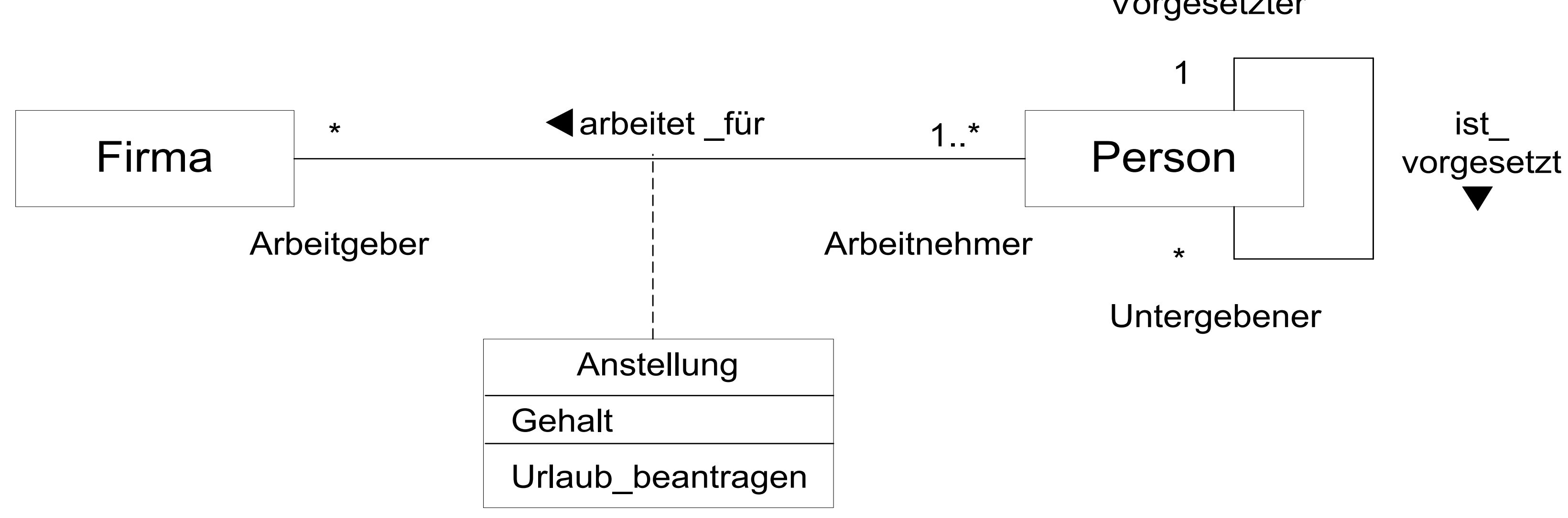
- auch Assoziationen können Merkmale, d.h. Attribute und Operationen besitzen.

- Diese Merkmale sind abhängig vom Vorhandensein der Assoziation. In anderem Kontext nicht nötig.

  $\Rightarrow$ Darstellung als Assoziationsklasse.

- Die Merkmale existieren für jede einzelne aufgebaute Verbindung zwischen zwei Objekten dieser Klassen genau einmal.

Beispiel für eine Assoziationsklasse



Bem.: *Auch wenn jede Person in genau einer Firma arbeiten würde:*
Semantisch gehört Gehalt und Urlaub_beantragen zur Assoziation
$\Rightarrow$ *Assoziationsobjekt*

Durchgängiges Beispiel - Fortsetzung

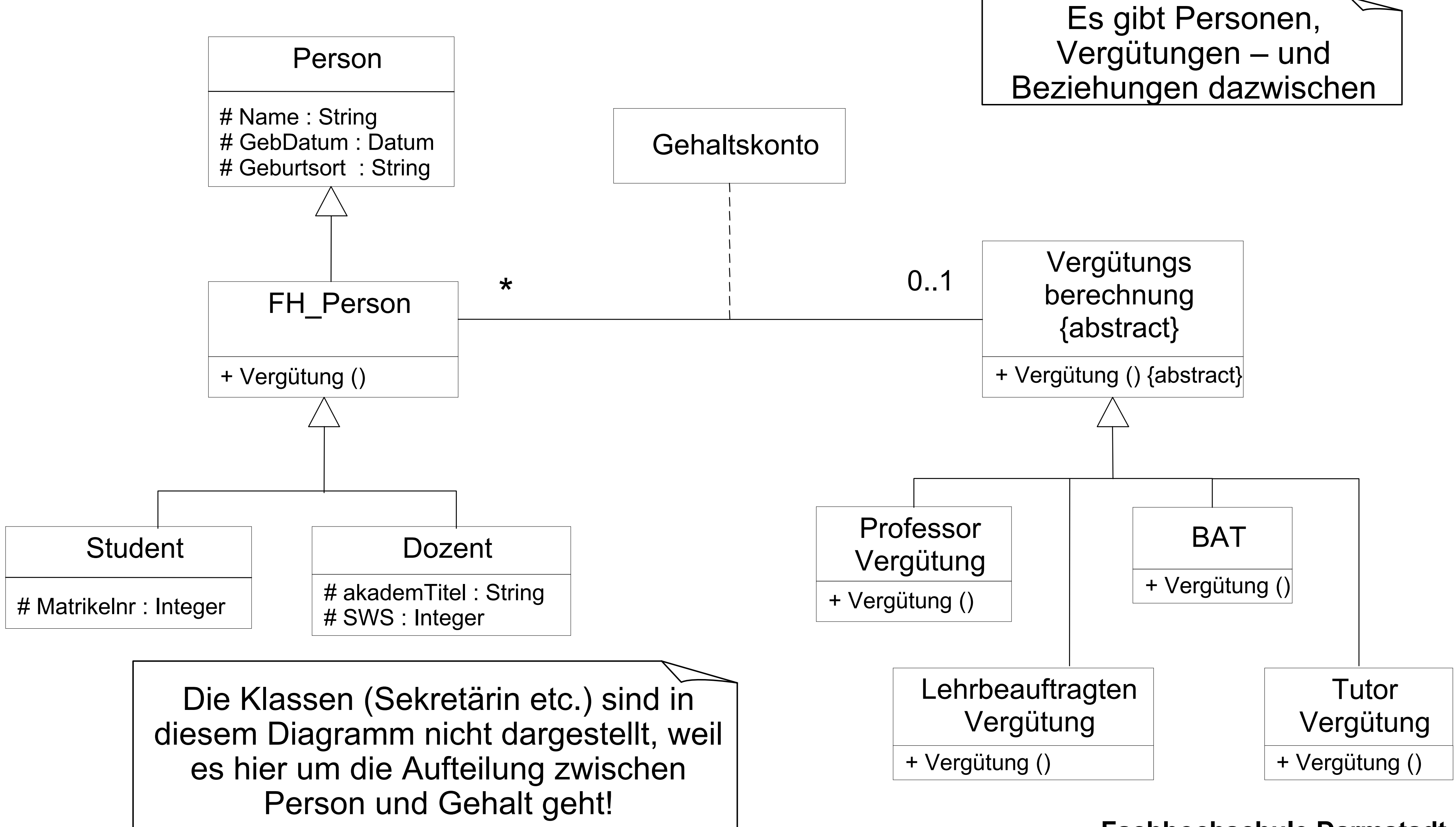
Die bisher modellierte Vererbungshierarchie ist korrekt, jedoch bezüglich Flexibilität und Wiederverwendung ungünstig modelliert:

- *Wird ein Objekt der Klasse Student erzeugt, so müsste dieses Objekt seine Klasse wechseln, sobald der Student Tutor wird. Nach der bisher modellierten Hierarchie müsste das Objekt zerstört und ein neues Objekt der Klasse StudentischerTutor erzeugt und mit denselben Daten belegt werden.*

- *Der Unterschied zwischen den Personengruppen (im Hinblick auf ein Hochschulverwaltungssystem) besteht in der Art der Vergütung.*
  Die Vergütung taucht aber nur "versteckt" als Methode auf.

⇨ *Nachdem wir Assoziationen und Assoziationsklassen kennengelernt haben, können wir durch ein geschicktes Design der bisherigen Klassenhierarchie dieses Problem beseitigen.*

Durchgängiges Beispiel (Lösung mit Assoziationen)

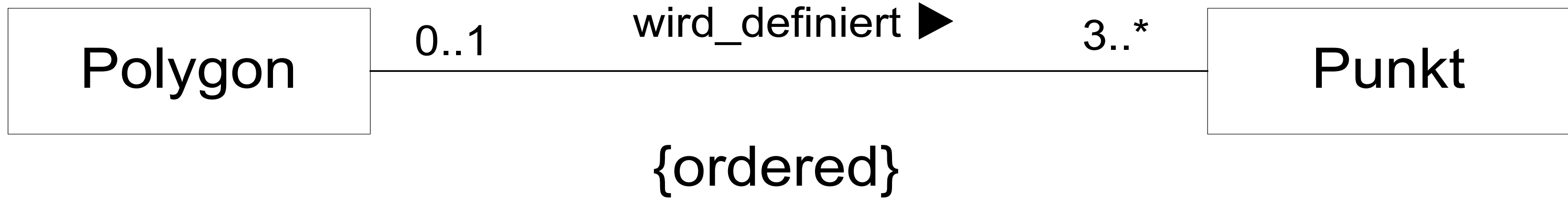


Randbedingungen für Assoziationen (Constraints)

Zu Assoziationen können Randbedingungen hinzudefiniert werden.

- Eine Randbedingung ist eine zusätzliche Information zum vorhandenen Modellelement, um es konsistent zu machen.
- Constraints geben Bedingungen für die Implementierungsphase.

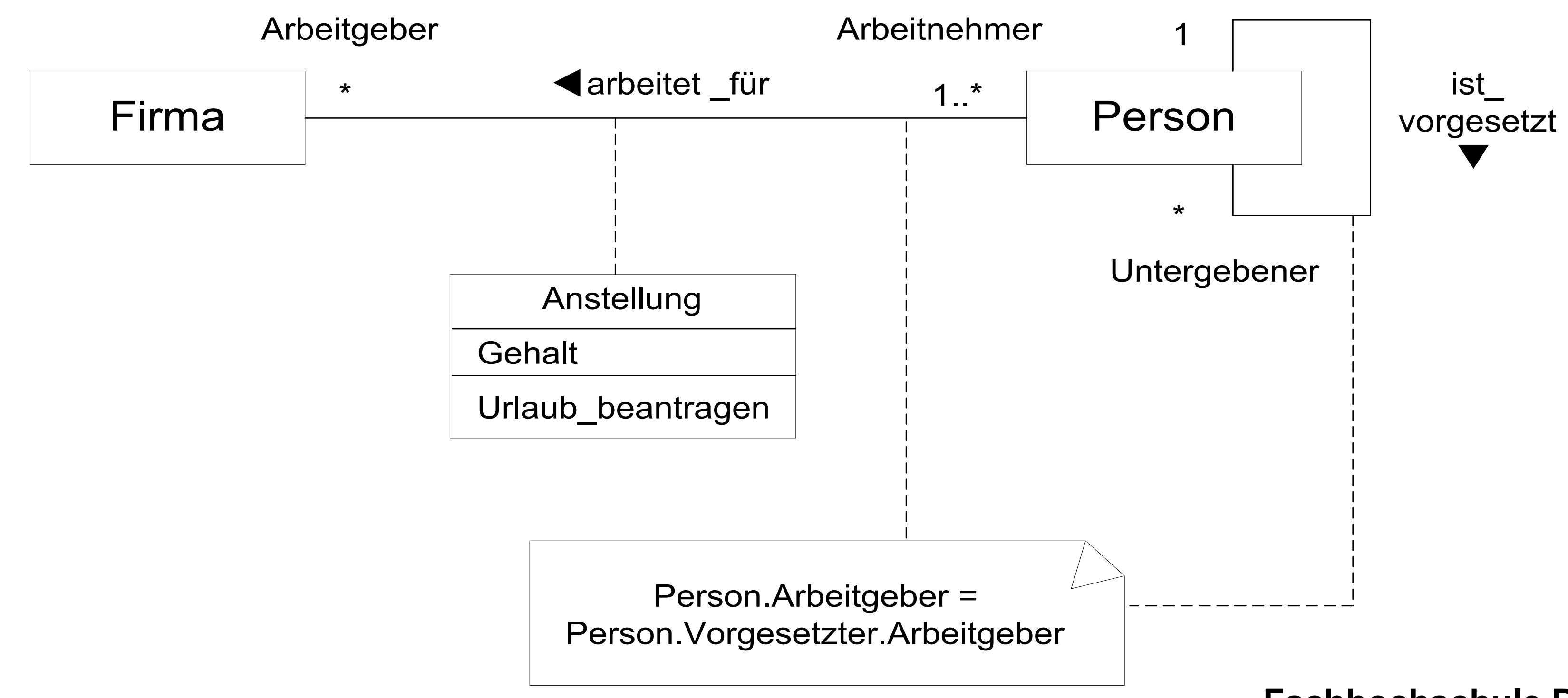
Beispiel 1:
Bei einem Polygon kommt es auf die Reihenfolge der Punkte an



Randbedingungen für Assoziationen (Beispiel)

Randbedingungen mit langem Text können als Kommentar angegeben sein.

z.B.: Der Vorgesetzte und dessen untergebener Arbeitnehmer müssen bei demselben Arbeitgeber beschäftigt sein.

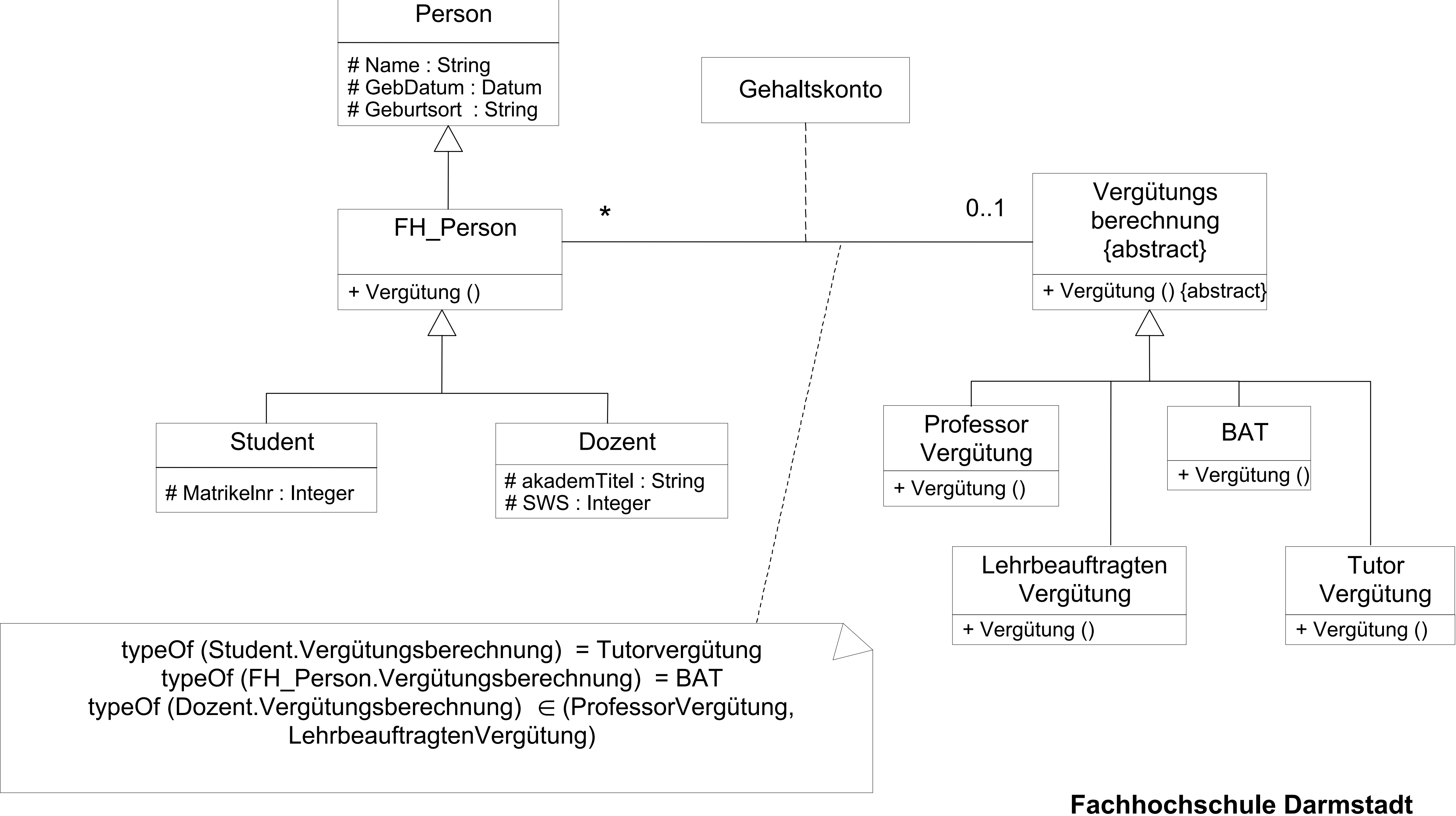


Durchgängiges Beispiel (mit Constraints)

- *Unsere bereits veränderte Vererbungshierarchie des ersten Klassenhierarchieentwurfs kann nun durch die Verwendung von selbstdefinierten Constraints vollständig und widerspruchsfrei modelliert werden.*

- *Es ist klar, dass nicht jedes Objekt willkürlich mit jedem Vergütungsalgorithmus kombiniert werden kann. Durch ein entsprechendes Constraint sind nun (gemäß der Aufgabe eines Modells) alle Regeln des Anwendungsbereichs widergespiegelt und hinreichende Vorgaben für die Implementierung gegeben.*

Durchgängiges Beispiel (Lösung mit Constraints)

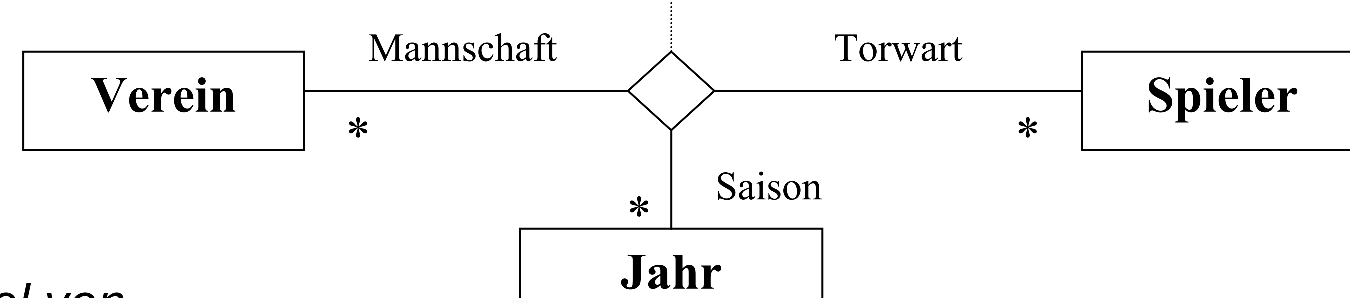


Ternäre und höherwertige Assoziationen: Beispiel

An einer Assoziation können auch mehr als zwei Klassen beteiligt sein.

Daten:

Torwart	Mannschaft	Saison	S	N	U	GT
Peter Gleichauf	FC Sandberg	1964	10	13	02	12
Peter Gleichauf	FC Sandberg	1966	15	10	00	21
Fred Hill	ASC Altheim	1966	08	06	11	13
Fred Hill	ASC Altheim	1972	21	02	02	01
Fred Hill	SV Rohrstock	1972	14	07	04	16
Fred Hill	FC Sandberg	1970	12	06	07	07



Zu einem assoziierten Tripel von

- *Spieler, Verein und Jahr gibt es genau 1 Statistik*
- *Statistik, Spieler und Verein gibt es 0..n Jahre*

Ternäre und höherwertige Assoziationen

- Oft gibt es Alternativen zur 3-er bzw. höherwertigen Assoziation (Einführung einer weiteren Klasse statt 3er-Beziehung)
- Oft sind vermeintliche 3-er Beziehungen in Wahrheit drei 2-er Beziehungen!

⇒ **3-er und höherwertige Assoziationen sollten, falls möglich, vermieden werden !**

Übung

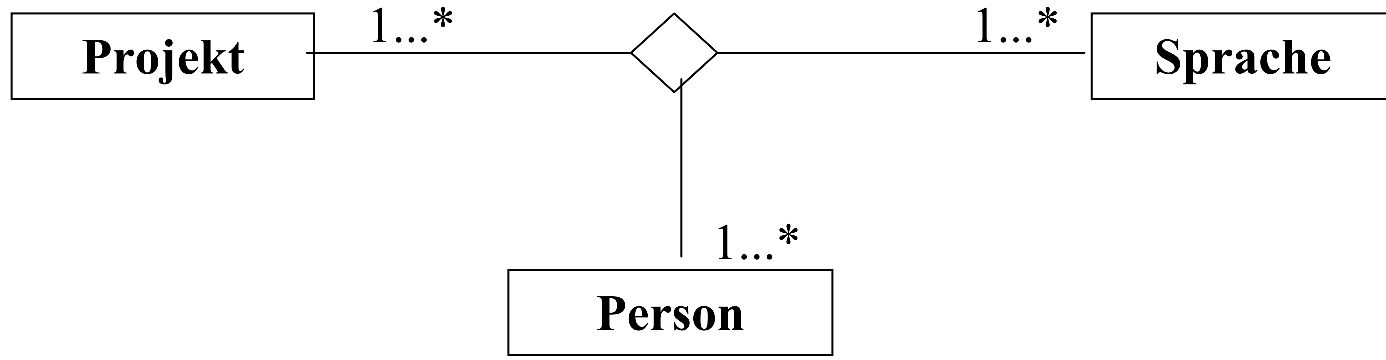
Anforderung:

- *In unserer Firma werden mehrere Projekte abgewickelt.*
- *In einem Projekt arbeiten mehrere Mitarbeiter mit.*
- *Das im Projekt zu erstellende Programmierer-System kann in mehreren Programmiersprachen zu schreiben sein.*
- *Ein Mitarbeiter arbeitet in diesem Projekt in einer oder mehreren Programmiersprachen.*
- *Der gleiche Mitarbeiter kann auch in anderen Projekten mitarbeiten, dort evtl. mit anderen Programmiersprachen.*

Aufgabe:

Erstellen sie das Klassendiagramm.

Übung: Klassendiagramm



- *zu einer Person in einem Projekt gibt es 1..n verwendete Sprachen*
- *zu einer Sprache in einem Projekt gibt es 1..n Personen*
- *zu einer Person, die in einer Sprache arbeitet, gibt es 1..n Projekte*

Qualifizierte Assoziationen

Zweck:

auf <u>logischer Ebene</u> einen Zugriffsschlüssel auf Objekte definieren

⇨ direkter Zugriff auf assoziierte Objekte
⇨ spezifiziert über ein oder mehrere Attribute (**Qualifier**) der Zielklasse
⇨ reduzierte Multiplizität für die Objekte, die über Qualifier festgelegt sind

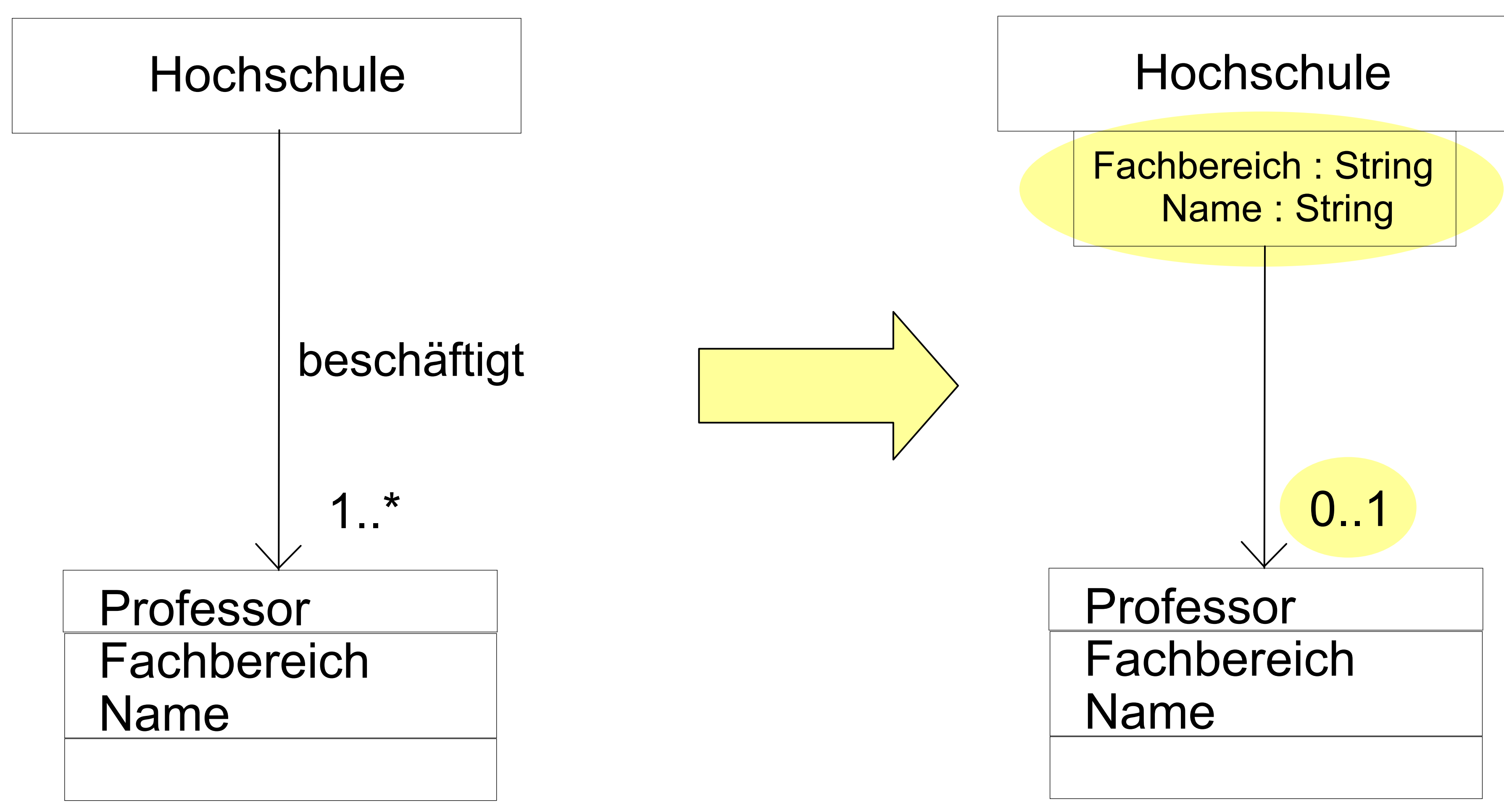
Kardinalität:

- bezieht sich auf den Qualifier

⇨ „wieviele Objekte liefert ein spezifischer Wert des Qualifiers?“

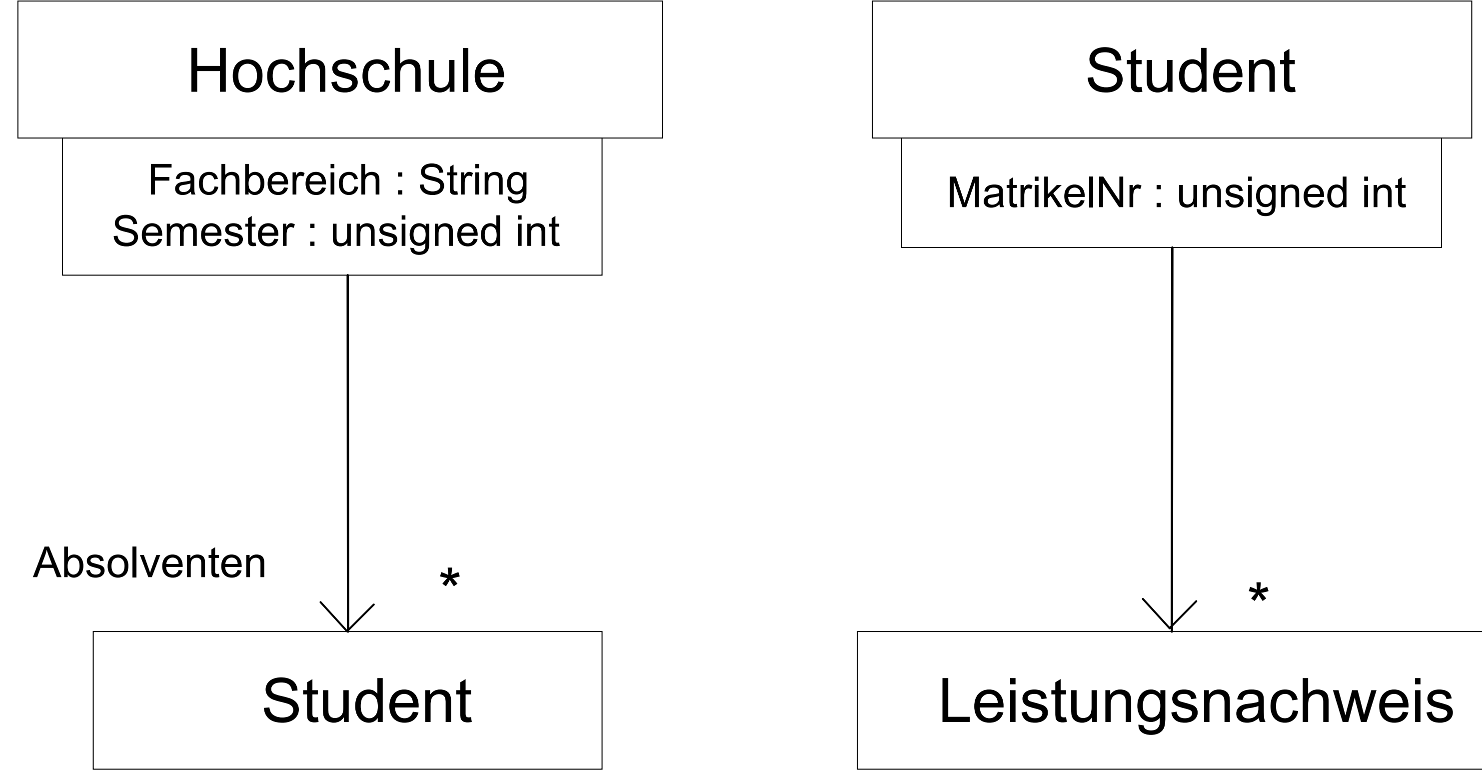
Qualifizierte Assoziationen (Beispiel)

Beispiel: Normale Assoziation und zweifach qualifizierte Assoziation



Qualifizierte Assoziationen (Beispiel)

Beispiel: Einfach und zweifach qualifizierte Assoziation:



Durchgängiges Beispiel

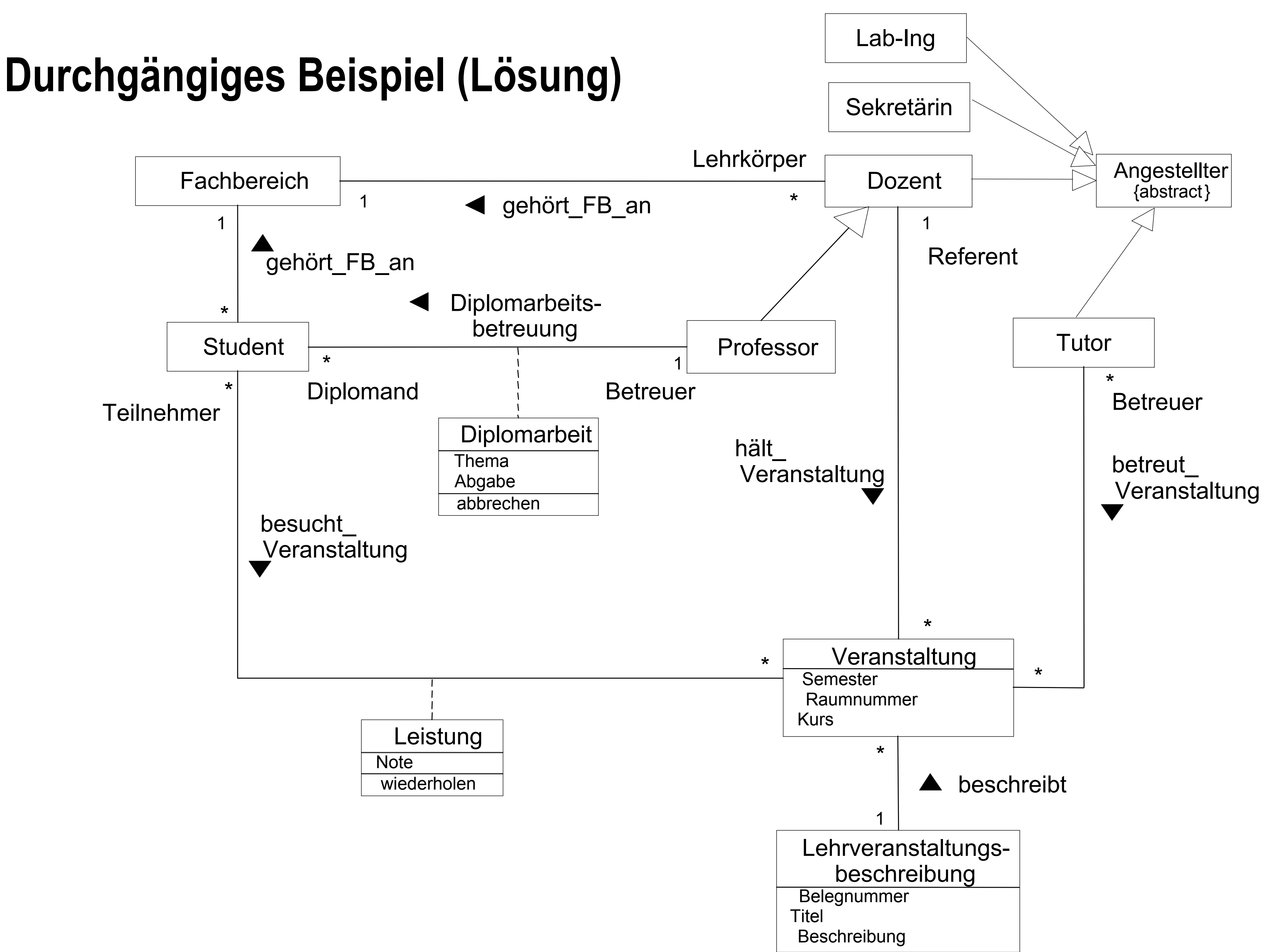
***Aufgabe*:**

- *Erweitern Sie das bisherige Klassendiagramm unseres Hochschulverwaltungssystems so, dass es die nachfolgenden zusätzlichen Klassen und Assoziationen integriert.*
- *Analysieren Sie zur Identifikation der Klassen und Assoziationen den Text. Bestimmen Sie die Kardinalitäten der Assoziationen und benennen Sie die Assoziationen und/oder die Rollen, die die beteiligten Klassen in einer Assoziation spielen, wenn dadurch die Verständlichkeit des Modells verbessert wird.*

***Anforderungen*:**

- *Die Dozenten und die Studenten gehören einem Fachbereich an.*
- *Studenten nehmen an Lehrveranstaltungen teil, die von Dozenten gehalten werden.*
- *Die Lehrveranstaltungen werden mit einem Leistungsnachweis abgeschlossen. Bestehen Studenten die Klausur nicht, können sie an einer Wiederholungsklausur teilnehmen.*
- *Studenten werden während der Diplomarbeit von Professoren betreut. Thema und Abgabetermin der Diplomarbeit sind zu erfassen. Studenten können natürlich eine Diplomarbeit auch unvollendet abbrechen.*
- *Lehrveranstaltungen im Semester können von Tutoren mitbetreut werden.*

Durchgängiges Beispiel (Lösung)



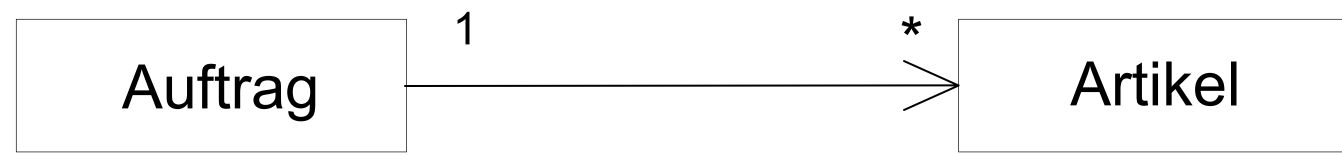
Navigationsrichtungen bei Assoziationen

- Bei der Modellierung des dynamischen Verhaltens (Entwurf der Kommunikation zwischen Klassen) wird festgestellt, in welche Richtung eine Assoziationsbeziehung durchlaufen wird.

  ⇒ Diese Navigationsrichtung kann angegeben werden (offene Pfeilspitze)

- Die Navigationsrichtung gibt beim späteren Programmentwurf Auskunft darüber, in welchen Klassen Verweise auf Objekte anderer Klassen angelegt werden müssen.

Beispiel:



⇨ *Ein Auftrag muss seine assoziierten Artikel kennen*
(z.B. Gesamtsumme berechnen, Nachschauen was disponiert werden muss).

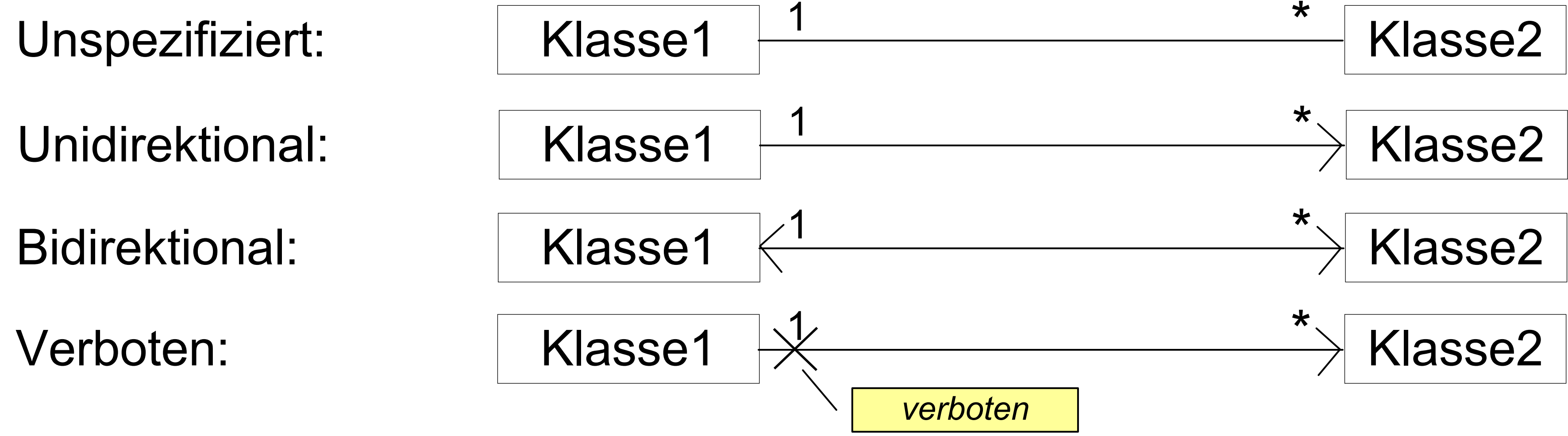
⇨ *Ein einzelner Artikel muss jedoch nicht wissen, welche Kunden ihn jemals bestellt haben.*

Notation bei Navigationsrichtungen

**UML 2 !!
geändert ggü. UML 1.x**

- In UML 1.x stand die Assoziation in Linienform für eine bidirektionale Beziehung
  ⇨ es gab keine Möglichkeit auszudrücken, dass man sich über die Richtung "noch keine Gedanken gemacht" hat

- In UML 2.0:
  ⇨ Notation für Navigierbarkeit und "unspezifiziert" (default)

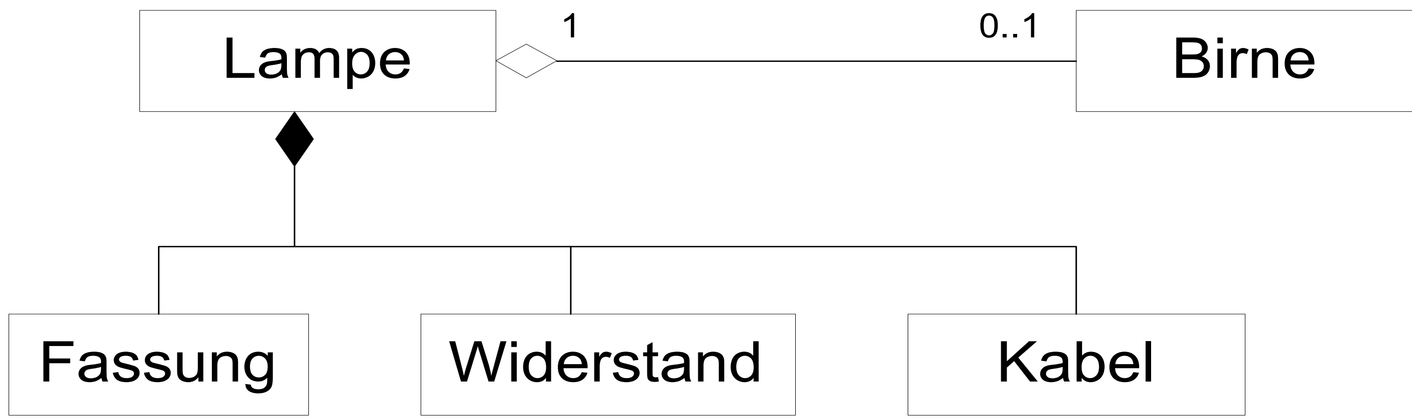


Aggregation (= spezielle Assoziation)

⇨ Aggregation bedeutet „Teil-von-Beziehung“ (oder „besteht-aus-Beziehung“) d.h. ein Objekt ist Bestandteil eines anderen.

Aggregation ist

transitiv:	A ist Teil von B, B ist Teil von C. Somit ist A Teil von C.
asymetrisch:	A ist Teil von B, aber B ist nicht Teil von A.

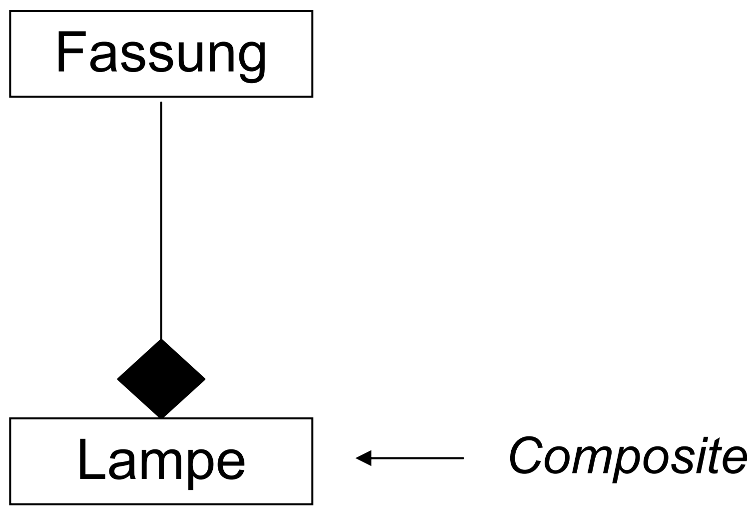


Arten von Aggregationen: Komposition

Komposition

- Symbol: ausgefüllte Raute
- Ist die Verbindung aufgebaut, kann sie nicht mehr verändert werden.
- Ein aggregiertes Objekt existiert während der Lebenszeit des aggregierenden Objekts (Composite-Objekt), aber nicht darüber hinaus.
- Ein aggregiertes Objekt gehört zu genau einem aggregierenden Objekt

⇨ "unshared" Aggregation

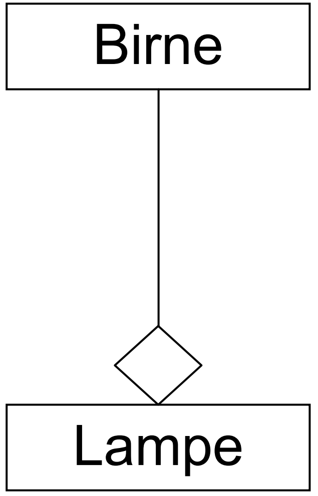


Arten von Aggregationen: Aggregation

Aggregation:

- Symbol: leere Raute
- spezifiziert Referenz auf das aggregierte Objekt.
- Aggregiertes Objekt ist zwar Bestandteil, existiert aber unabhängig vom aggregierenden Objekt.

⇨ "shared" Aggregation



Beispiel: Constraints für Aggregationen

Ein Motor und das Getriebe müssen, um zu funktionieren, mit derselben Kupplung verbunden sein.

Motor

Auto

Kupplung

Getriebe

1

1

1

1

1

Auto.Kupplung = Auto.Motor.Kupplung AND Auto.Kupplung = Auto.Getriebe.Kupplung

- *Ohne ein Constraint könnte laut Modell der Motor sowie das Getriebe ein anderes Kupplungsexemplar assoziieren als das übergeordnete Auto selbst*

  $\Rightarrow$ *Inkonsistenz im Modell !*

- *Das hinzudefinierte Constraint schließt diese Lücke*

Durchgängiges Beispiel mit Aggregationen

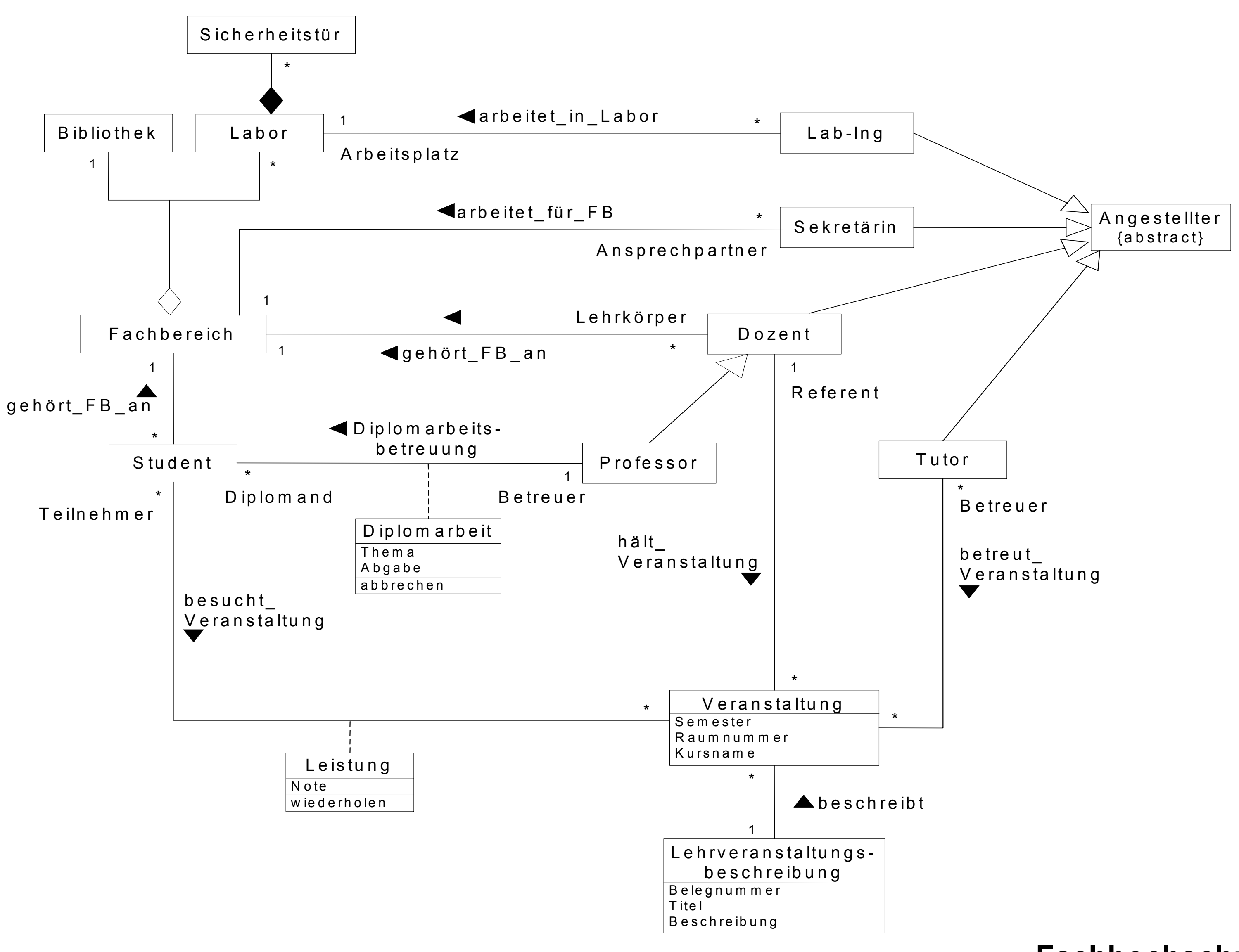
Aufgabe:

Erweitern Sie das Klassendiagramm der Hochschulverwaltung um die nachfolgenden Inhalte.

***Anforderungen**:*

- *Ein Fachbereich kann eine Bibliothek und mehrere Labore besitzen, in denen Laboringenieure arbeiten.*
- *Jeder Fachbereich beschäftigt in der Verwaltung Sekretärinnen.*
- *Die Laborräume sind zur Vermeidung von Diebstählen mit Sicherheitstüren ausgestattet.*
- *Labor-Ingenieure arbeiten in bestimmten Labors.*

Durchgängiges Beispiel mit Aggregationen (Lösung)



Durchgängiges Beispiel (Erklärung)

- *Teile eines Fachbereichs sind Labors und eine Bibliothek, die aber jederzeit einem anderen Fachbereich zugewiesen werden können. Daher ist die Teil-von-Beziehung eine shared-Aggregation.*
- *Eine Sicherheitstür ist fester Bestandteil des Labors und ist deswegen eine unshared-Aggregation.*

Zusammenfassung Klassen(diagramme)

Klassendiagramme enthalten

- Klassen mit Attributen und Methoden
- Vererbung / Generalisierung / Spezialisierung
- Abstrakte Klasse
- Parametrisierte Klasse
- Diskriminator
- Assoziationsklassen
- Constraints
- Assoziationen
- Höherwertige Assoziationen
- Qualifizierte Assoziationen
- Navigationsrichtung
- Aggregation

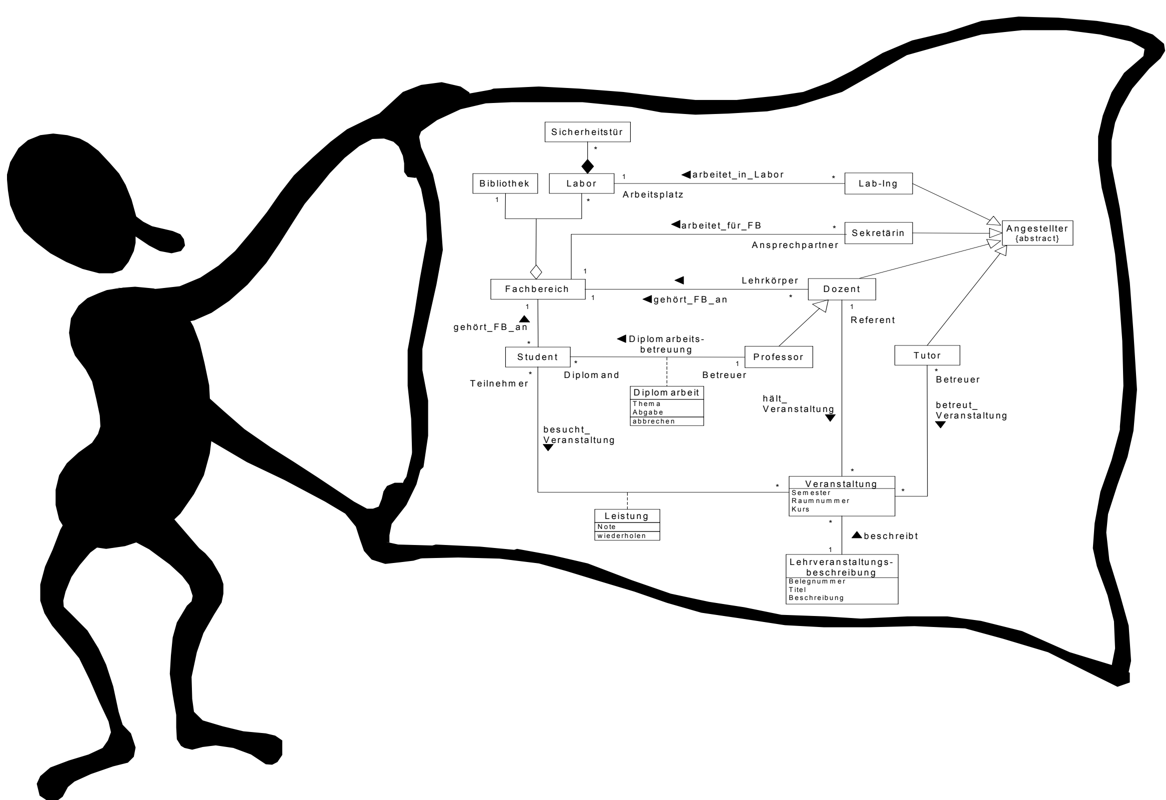
⇨ Das sind die Elemente im "Baukasten"

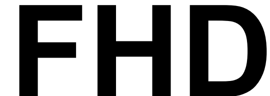
Wert von Klassendiagrammen

- Klassendiagramme sind eines der wichtigsten Beschreibungsmittel in der Modellierung
- Zur Beschreibung eines (komplexen) Systems werden viele Klassendiagramme erstellt

- Hinweise:
  - niemals versuchen gleich alles perfekt zu machen; auch Klassendiagramme werden iterativ entwickelt
  - Details wie Navigationsrichtungen und Multiplizität erst eintragen, wenn die Modellierung hinreichend stabil scheint
  - Bei der Modellierung an die Systemgrenze denken! Nur das System modellieren – nicht die komplette Außenwelt!
  - immer nur einen Aspekt auf einem Diagramm darstellen; wenn das Diagramm nicht mehr auf eine Seite passt – aufteilen!
  - die Begriffe für alle Elemente mit großer Sorgfalt wählen
  - Assoziationen so konkret wie möglich spezifizieren
  - ein Pointer als Attribut einer Klasse ist meist eine falsch modellierte Assoziation
  - keine Pseudoprogramme in Constraints schreiben – es geht um Verständlichkeit

Fragen





Fachhochschule Darmstadt
Fachbereich Informatik

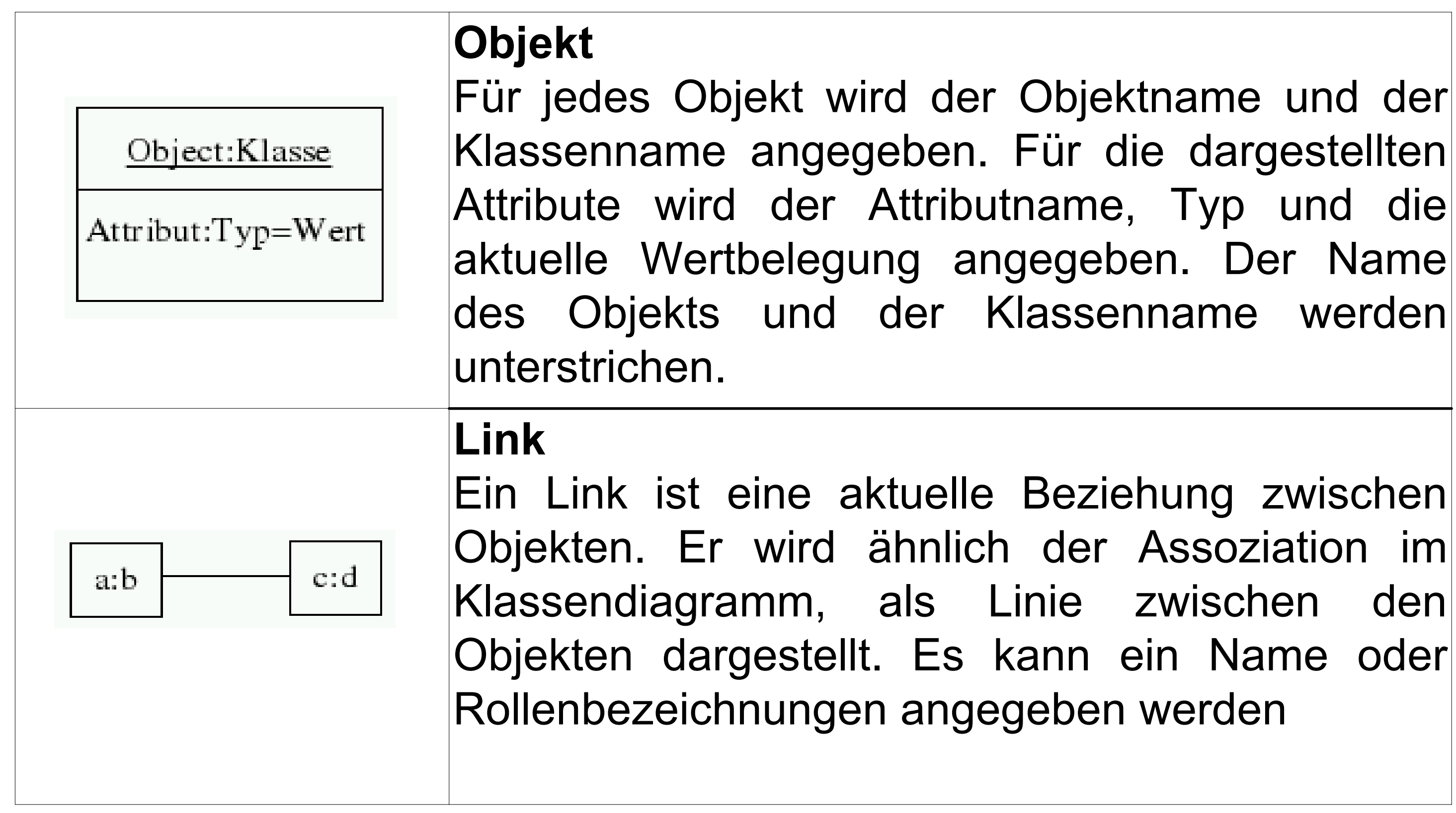
Softwaretechnik I

Kapitel 3 ff

- weitere Diagramme in der UML

Quellenhinweis:
Einige Folien zu dieser Vorlesung entstammen Präsentationen von Prof. G. Raffius und Prof. W. Weber

Objektdiagramm: Notation in UML 2.0



Object:Klasse Attribut:Typ=Wert	**Objekt** Für jedes Objekt wird der Objektname und der Klassenname angegeben. Für die dargestellten Attribute wird der Attributname, Typ und die aktuelle Wertbelegung angegeben. Der Name des Objekts und der Klassenname werden unterstrichen.
a:b — c:d	**Link** Ein Link ist eine aktuelle Beziehung zwischen Objekten. Er wird ähnlich der Assoziation im Klassendiagramm, als Linie zwischen den Objekten dargestellt. Es kann ein Name oder Rollenbezeichnungen angegeben werden

Objektdiagramm: Inhalt

Ein Objektdiagramm enthält folgende Elemente

- Objekt (Instanz von Klassen)
- Wert (Instanz von Attributen)
- Link (Instanz von Assoziationen)

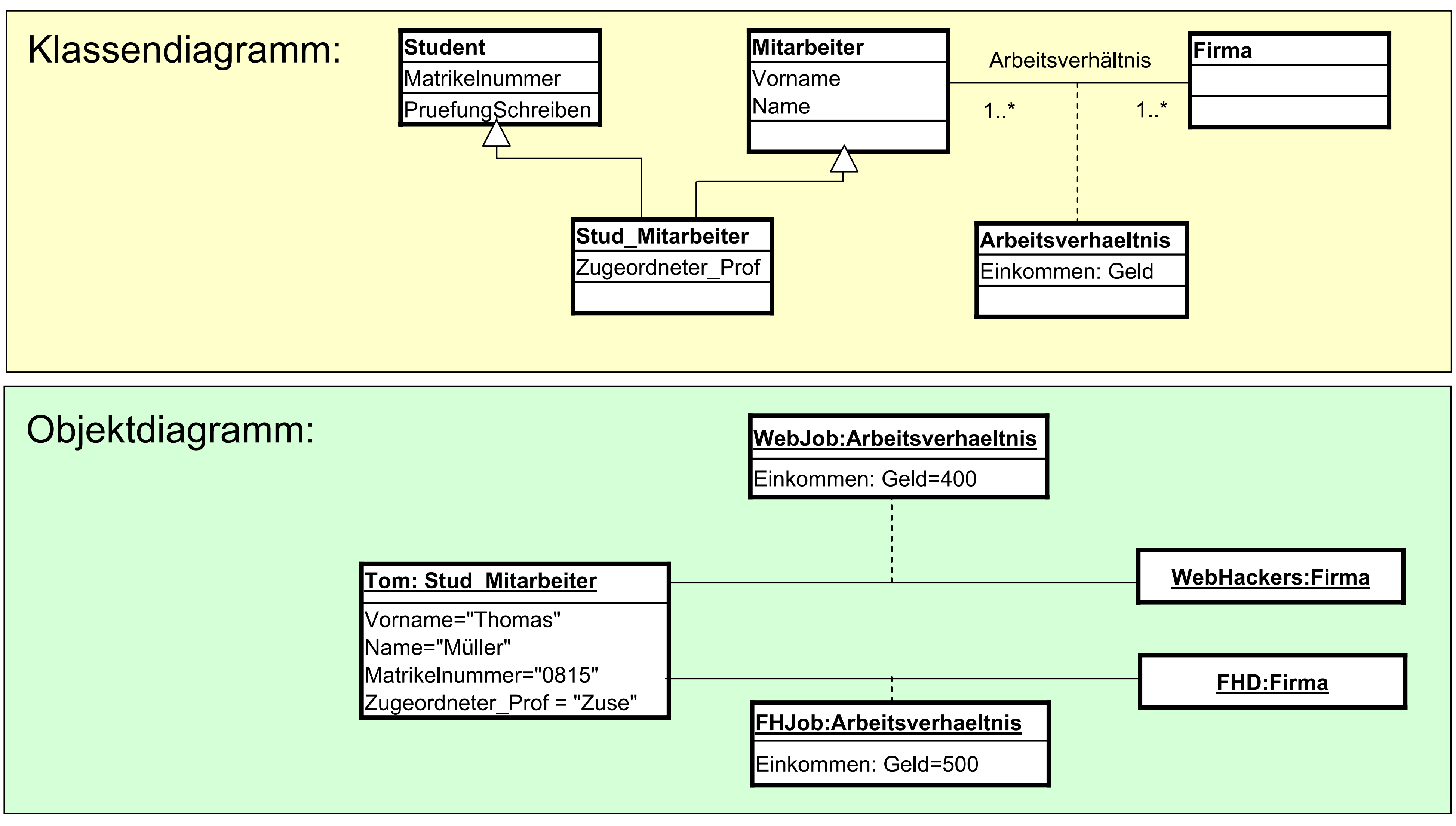
Instanzen von *Methoden* werden in Sequenz-diagrammen dargestellt!

Ein Objektdiagramm zeigt

- **Instanzen zu einem bestimmten Zeitpunkt**

⇨ **einen "Snapshot" des Systems**

Objektdiagramm: Beispiel



Quelle: UML 2 glasklar, C. Rupp et.al.

Objektdiagramm: Diskussion

Vergleich mit dem Klassendiagramm

Klassendiagramm	Objektdiagramm
stellt Vererbungsbeziehungen dar	Stellt das Ergebnis der Vererbung dar
Zeigt Assoziationen in der allgemeinen Form	Zeigt Assoziationen als konkrete Links zu Objekten
Keine Darstellung von Werten (außer Defaultwerten)	Konkrete Werte
Abstraktion teilweise schwer zu verstehen	Unvollständige Darstellung in "Beispielform"

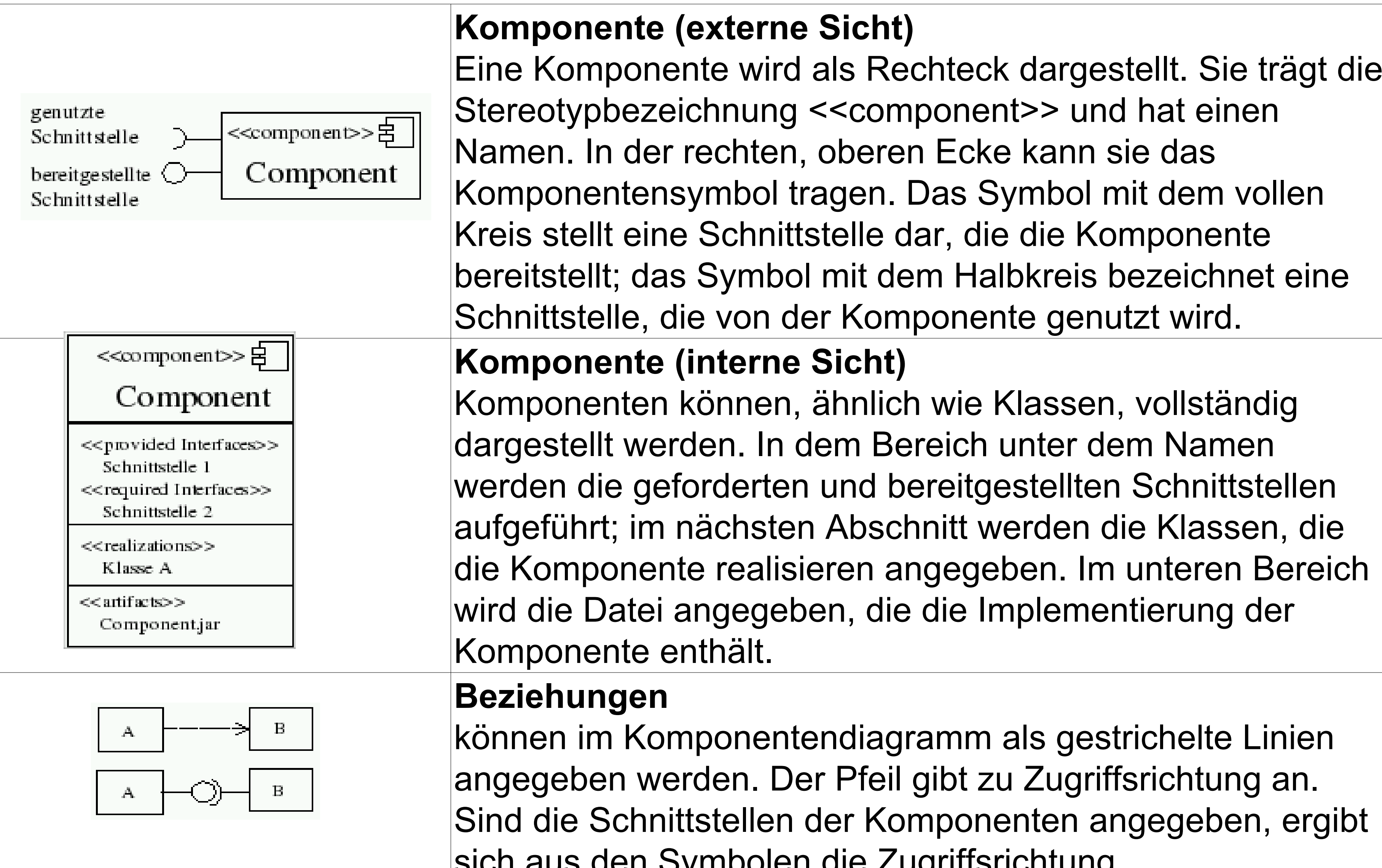
Objektdiagramm: Anwendung

Objektdiagramme sind z.B. in folgenden Situationen nützlich:

- zur Dokumentation von Architekturen mit abstrakten Klassen
- zur Erläuterung von zirkulären Assoziationen
  (z.B. *n* Personen kennen *n* Personen)
- zur Überprüfung der Modellierung mit konkreten Beispielen
- zur Darstellung von konkreten Daten zum Debugging / Testen
- Zum Darstellen der Daten- bzw. Objekt-Verteilung in verteilten Systemen

Komponentendiagramm: Notation in UML 2.0

verändert gegenüber UML1.x!



Komponente (externe Sicht)
Eine Komponente wird als Rechteck dargestellt. Sie trägt die Stereotypbezeichnung <<component>> und hat einen Namen. In der rechten, oberen Ecke kann sie das Komponentensymbol tragen. Das Symbol mit dem vollen Kreis stellt eine Schnittstelle dar, die die Komponente bereitstellt; das Symbol mit dem Halbkreis bezeichnet eine Schnittstelle, die von der Komponente genutzt wird.

Komponente (interne Sicht)
Komponenten können, ähnlich wie Klassen, vollständig dargestellt werden. In dem Bereich unter dem Namen werden die geforderten und bereitgestellten Schnittstellen aufgeführt; im nächsten Abschnitt werden die Klassen, die die Komponente realisieren angegeben. Im unteren Bereich wird die Datei angegeben, die die Implementierung der Komponente enthält.

Beziehungen
können im Komponentendiagramm als gestrichelte Linien angegeben werden. Der Pfeil gibt zu Zugriffsrichtung an. Sind die Schnittstellen der Komponenten angegeben, ergibt sich aus den Symbolen die Zugriffsrichtung.

Komponentendiagramm : Inhalt

mehr als die Zuordnung zu Quellcodedateien in UML1.4!

Komponenten

- sind modulare Systemteile, die ihren Inhalt kapseln (und vor dem Benutzer verbergen)
- sind eigenständige Anwendungen mit einer klar definierten S (bereitgestellte und benutzte Funktionalität)
- bestehen aus enthaltenen Klassen oder Komponenten

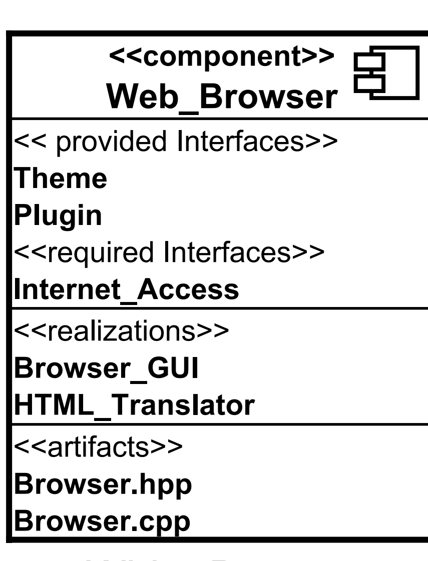
Anpassung an den Komponenten-begriff im Sinne von CORBA, COM, Java,…

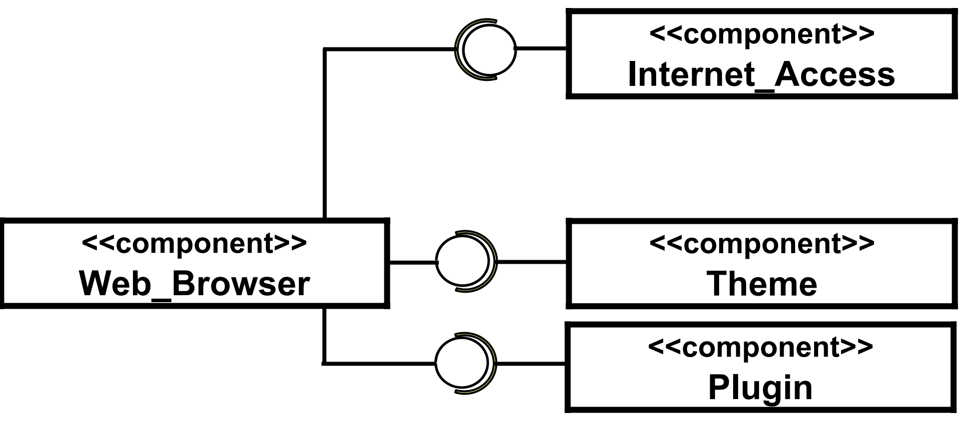
Ein Komponentendiagramm beschreibt

- die Komponenten, ihre Beziehungen und die öffentlichen Sc
- die Struktur eines Systems
- wie diese Strukturen erzeugt werden
- die physikalischen Bestandteile eines Systems

Dateien mit Source-Code, Dokumentation, ByteCode etc. heißen jetzt Artefakte (stereotyp <<artifact>>)

Komponentendiagramm : Beispiel





White-Box
(mit Interna)

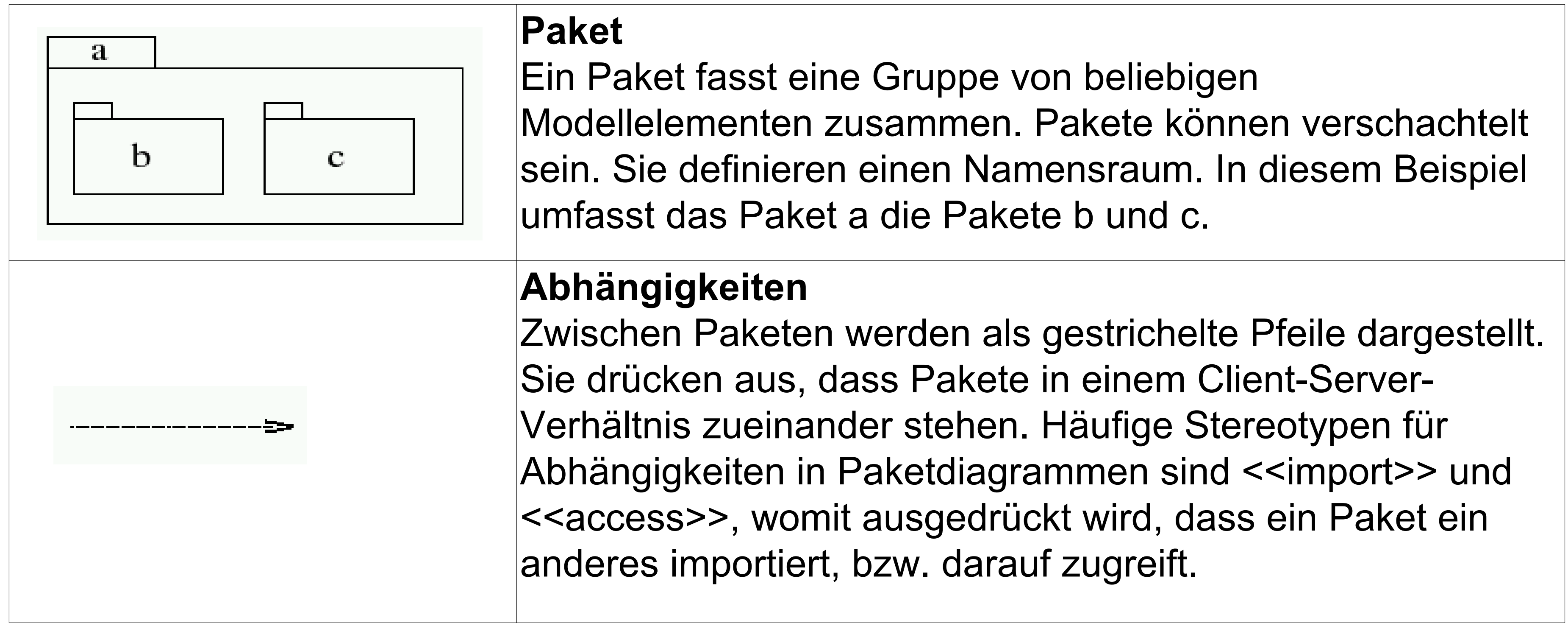
Black-Box
(Überblick)

Komponentendiagramm : Anwendung

Komponentendiagramme sind z.B. in folgenden Situationen nützlich:

- Entwicklung von SW mit Komponententechnologie
- Entwicklung von SW-Komponenten zur Wiederverwendung (Austauschbarkeit von Komponenten als Black-Box)
- Verteilte SW-Entwicklung (Darstellung von Schnittstellen)
- Zuordnung von Quellcodedateien

Paketdiagramm: Notation in UML 2.0



a, b, c	**Paket** Ein Paket fasst eine Gruppe von beliebigen Modellelementen zusammen. Pakete können verschachtelt sein. Sie definieren einen Namensraum. In diesem Beispiel umfasst das Paket a die Pakete b und c.
	Abhängigkeiten Zwischen Paketen werden als gestrichelte Pfeile dargestellt. Sie drücken aus, dass Pakete in einem Client-Server-Verhältnis zueinander stehen. Häufige Stereotypen für Abhängigkeiten in Paketdiagrammen sind <<import>> und <<access>>, womit ausgedrückt wird, dass ein Paket ein anderes importiert, bzw. darauf zugreift.

Paketdiagramm : Inhalt

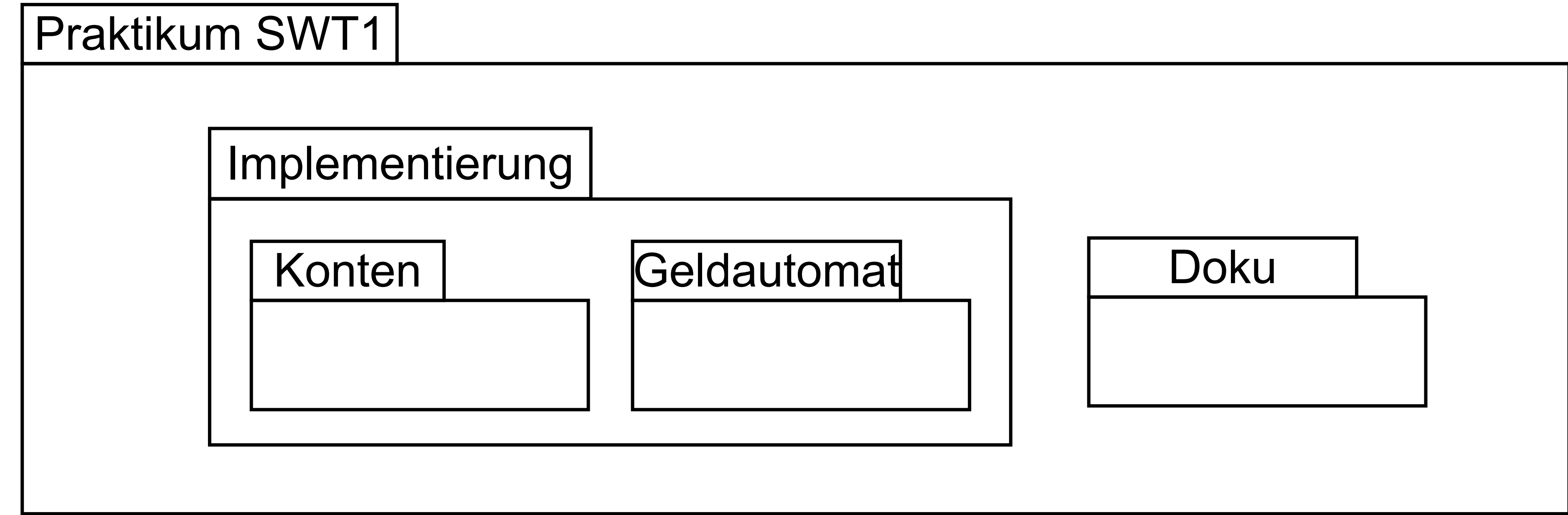
Ein Paketdiagramm beschreibt

- eine Aufteilung des Systems in Gruppen ("Pakete")
  – ähnlich zu Ordnern in Dateisystemen
- die Pakete und deren Beziehungen untereinander

Anmerkungen

- Ein Modellelement darf nur zu einem Paket gehören
- Ein Paket definiert einen Namensraum und eine Sichtbarkeit
- Pakete können verschachtelt sein

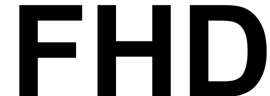
Paketdiagramm : Beispiel



Paketdiagramm : Anwendung

Paketdiagramme sind z.B. in folgenden Situationen nützlich:

- ein großes System soll in handhabbare Arbeitspakete aufgeteilt werden
- Modellelemente werden nach Themen gruppiert (Use Cases, Klassen, Diagramme,…)
  - funktional
  - logisch
  - in Schichten
  - ...



Fachhochschule Darmstadt
Fachbereich Informatik

Softwaretechnik I

Kapitel 3 ff

- Sequenzdiagramme und die UML

Quellenhinweis:
Einige Folien zu dieser Vorlesung entstammen Präsentationen von Prof. G. Raffius und Prof. W. Weber

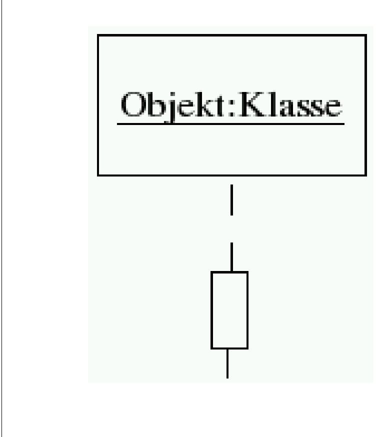
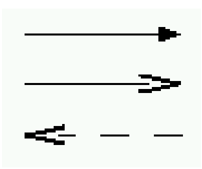
Diagramme der UML 2

Zwischenstand

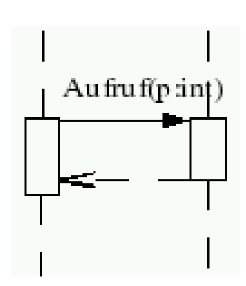
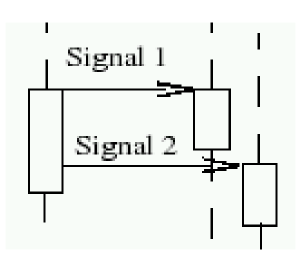
- Diagramme der UML 2
  - Strukturdiagramme
    - Klassendiagramm ✔
    - Komponentendiagramm ✔
    - Objektdiagramm ✔
    - Kompositionsstrukturdiagramm SWT 2
    - Verteilungsdiagramm SWT 2
    - Paketdiagramm ✔
  - Verhaltensdiagramme
    - Aktivitätsdiagramm SWT 2
    - Use-Case-Diagramm ✔
    - Zustandsautomat SWT 2
    - Interaktionsdiagramme
      - Sequenzdiagramm (kommt jetzt)
      - Interaktionsübersichtsdiagramm SWT 2
      - Kommunikationsdiagramm (kommt noch)
      - Zeitverlaufsdiagramm SWT 2

Quelle: „UML 2 –Ballast oder Befreiung?“
von Chris Rupp, SOPHIST GROUP, Agility Days 2003

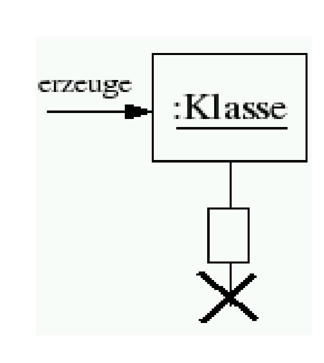
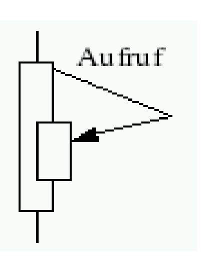
Sequenzdiagramm: Notation in UML 2.0 (I)

	**Klasse, Objekt** In dem Rechteck oberhalb der gestrichelten Linie wird der Objektname und der Klassenname angegeben. Der Name wird unterstrichen. Die senkrechte, gestrichelte Linie stellt die Lebenslinie (lifeline) eines Objekts dar. Im diesem zeitlichen Bereich existiert das Objekt. Das schmale Rechteck auf der gestrichelten Linie stellt eine Aktivierung dar. Eine Aktivierung ist der Bereich, in dem eine Methode des Objektes aktiv ist (ausgeführt wird). Auf einer Lebenslinie können mehrere Aktivierungen enthalten sein.
	**Nachricht, Botschaft** Objekte kommunizieren über Nachrichten. Nachrichten werden als Pfeile zwischen den Aktivierungen eingezeichnet. Der Name der Nachricht steht an dem Pfeil. Eine Nachricht liegt immer zwischen einem sendenden und einem empfangenden Objekt.

Sequenzdiagramm: Notation in UML 2.0 (II)

	**Synchrone Nachricht (Aufruf)** Der Pfeil mit der ausgefüllten Spitze bezeichnet einen synchronen Aufruf. Der Aufruf erfolgt von der Quelle zum Ziel, d.h. die Zielklasse muss eine entsprechende Methode implementieren. Die Quelle wartet mit der weiteren Verarbeitung bis die Zielklasse ihre Verarbeitung beendet hat und setzt die Verarbeitung dann fort. Ein Aufruf muss einen Namen haben; in runden Klammern können Aufrufparameter angegeben werden. Der gestrichelte Pfeil ist der Return. Die Bezeichnung des Return mit einem Namen ist optional.
	**Asynchrone Nachricht** Mit einer offenen Pfeilspitze werden asynchrone Nachrichten gekennzeichnet. Der Aufruf erfolgt von der Quelle zum Ziel. Die Quelle wartet mit der Verarbeitung nicht auf die Zielklasse, sondern setzt ihre Arbeit nach dem Senden der Nachricht fort. Asynchrone Aufrufe werden verwendet, um parallele Threads zu modellieren.

Sequenzdiagramm: Notation in UML 2.0 (III)

	**Objekt erzeugen, zerstören** Das Erzeugen eines Objektes wird durch eine Nachricht, die im Kopf des Objekts endet dargestellt. Das Zerstören (Löschen) eines Objektes wird durch ein x auf der Lebenslinie markiert.
	**Rekursion** Sendet ein Objekt eine Nachricht an sich selbst, so ruft es eine Methode auf, die es selbst implementiert.

Sequenzdiagramm: Inhalt

Ein Sequenzdiagramm beschreibt

- die Interaktion von Objekten
  - ⇨ die Reihenfolge beim Ablauf eines Szenarios (Instanz eines Use Cases)
  - ⇨ die (prinzipiellen) Aufrufe im Code
  - ⇨ die Erzeugung und Zerstörung von Objekten

Anmerkungen

- UML-Tools erzeugen aus Aufrufen die entsprechenden Methoden
- ein Sequenzdiagramm konkretisiert häufig einen Use Cases
- ein Sequenzdiagramm kann aber auch Sachverhalte illustrieren, die nicht als Use Case vorliegen
- Sequenzdiagramme haben die Leserichtung links→rechts, oben →unten
- Überkreuzungen von Aufrufen sind möglichst zu vermeiden
- Aktoren stehen üblicherweise links außen

Sequenzdiagramm: Erstes Beispiel

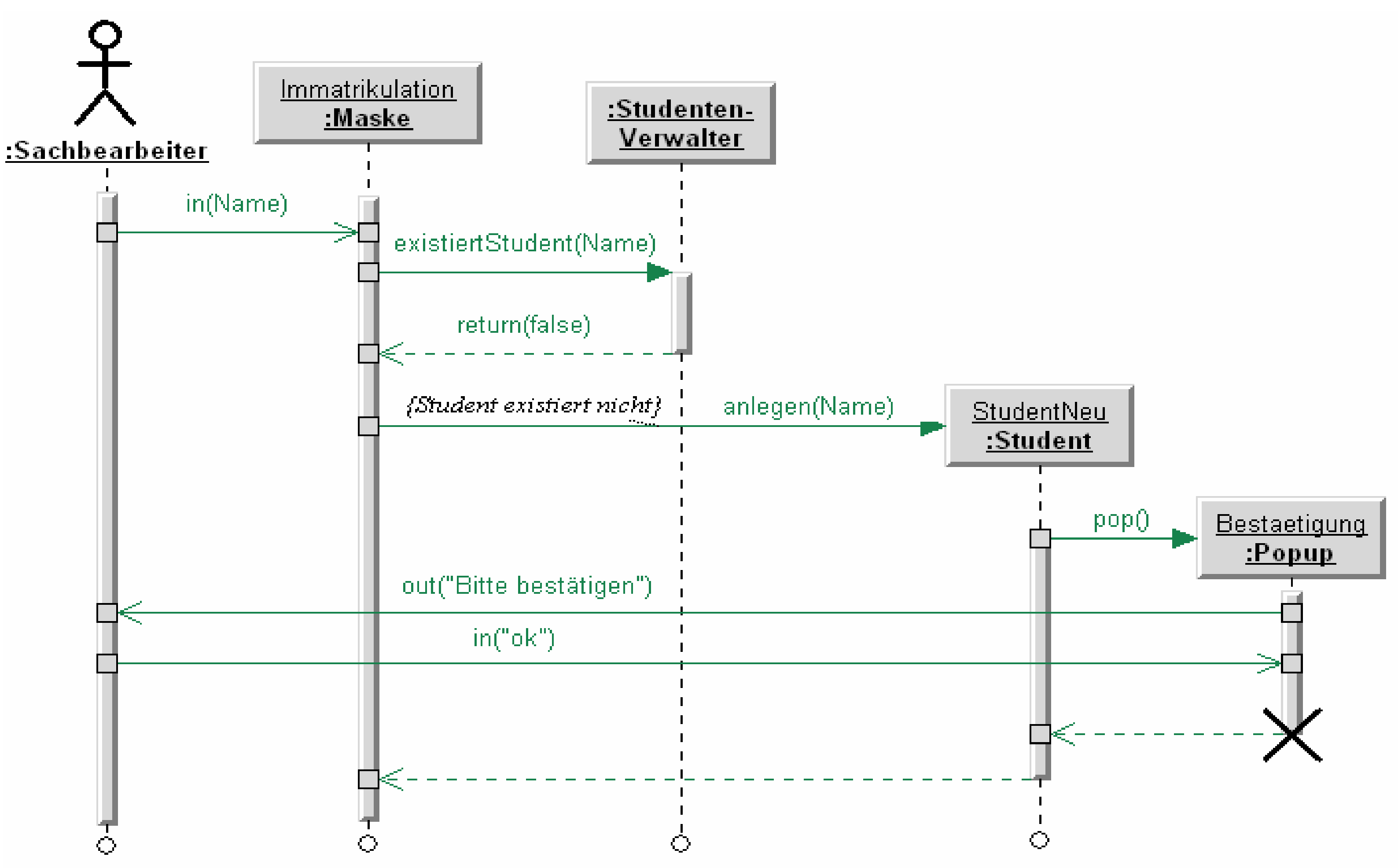
- Wir formulieren nun für Use Cases die entsprechenden Szenarien mit Hilfe von Sequenzdiagrammen.

Use Case *'Student immatrikulieren'*

Ein Sachbearbeiter der Hochschule erfasst am Bildschirm neue Studenten. Wurde ein Student bisher noch nicht erfasst, so wird ein neuer Eintrag angelegt und der Sachbearbeiter erhält eine Rückmeldung über den Erfolg der Erfassung.

Sequenzdiagramm: Erstes Beispiel – Lösung

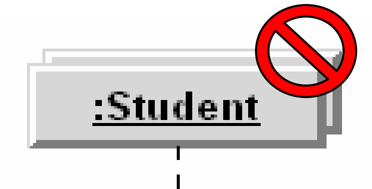
Szenario zu Use Case ´Student immatrikulieren´:



Sequenzdiagramm: Erstes Beispiel – Erläuterung

- *Über eine Erfassungsmaske werden Studentendaten erfasst. Ein Objekt Student wird kreiert, wenn es noch nicht vorhanden ist.*
- *Um dies herauszufinden, wird die Klasse StudentenVerwalter befragt, die alle Studenten kennt und verwaltet.*
- *Ist die Immatrikulation (das Kreieren des Studenten) erfolgreich, so teilt uns dies der neu angelegte Eintrag selbst mit, indem er ein Popup mit der entsprechenden Nachricht öffnet, die bestätigt werden muss.*

In UML 1.x wurden zur Verwaltung der Gesamtheit aller Klasseninstanzen sogenannte Multi Object Icons eingesetzt. Diese gibt es in UML 2.0 nicht mehr.



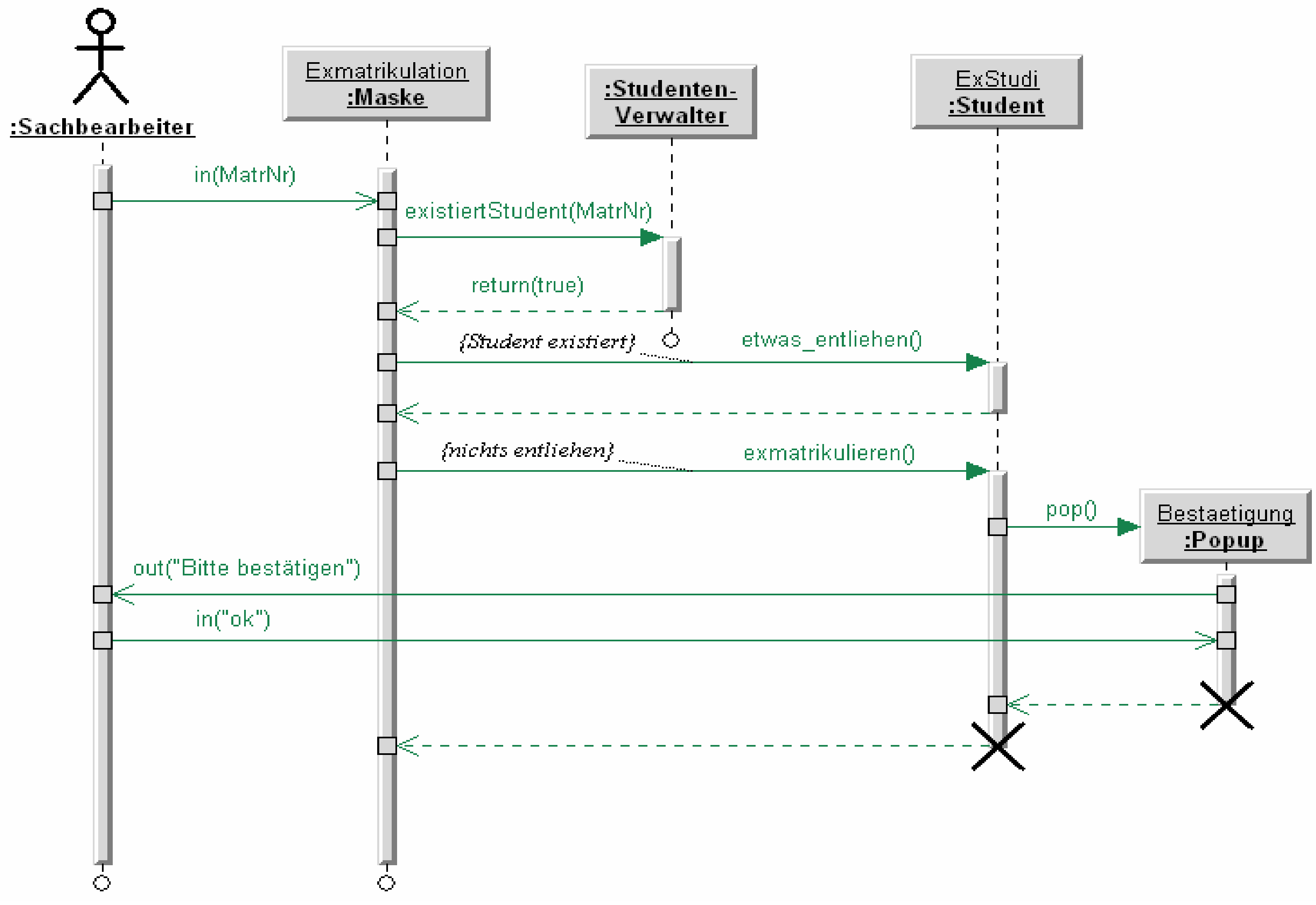
Sequenzdiagramm: Zweites Beispiel

Use Case *Student exmatrikulieren*

Ein Sachbearbeiter der Hochschule erfasst am Bildschirm Studenten, die exmatrikuliert werden. Die Löschung der Studentenakte darf nur erfolgen, wenn darin keine entliehenen Gegenstände mehr eingetragen sind. Der Sachbearbeiter erhält eine Rückmeldung über den Erfolg der Exmatrikulation.

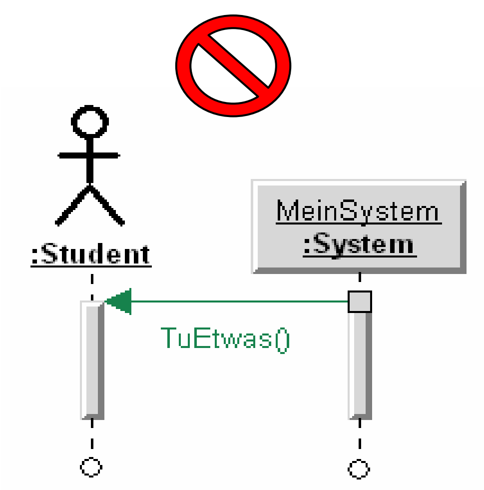
Sequenzdiagramm: Zweites Beispiel – Lösung

Szenario zu Use Case ´Student exmatrikulieren´:



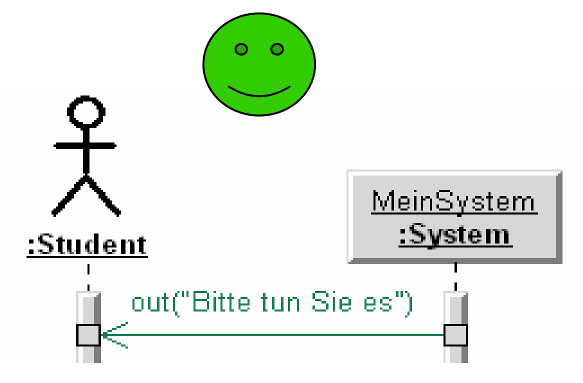
Fachhochschule Darmstadt
University of Applied Sciences

Sequenzdiagramm: Häufige Fehler



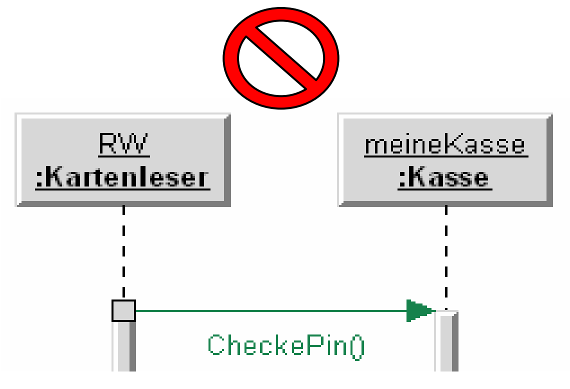
1. **Fehler: Methodenaufruf beim Aktor**
   Ein Aktor ist in der Regel menschlich. Er hat deshalb keine Methoden, die man aufrufen könnte.

2. **Fehler: Synchroner Aufruf bei einem Aktor**
   Ein Aktor führt ein eigenständiges Leben. Er hat seine eigene "CPU". In dieser Modellierung darf das System erst weitermachen, wenn der Aktor ein Return zurückschickt.



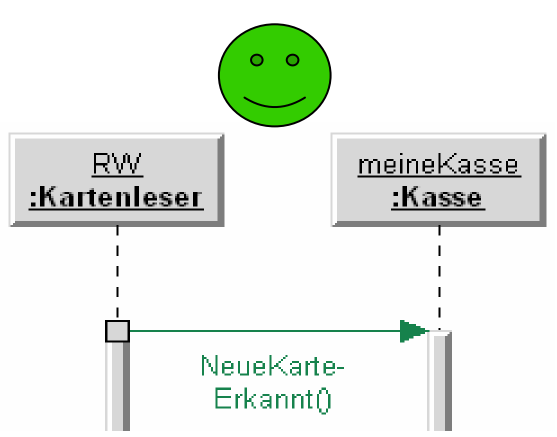
- **Richtig wäre, eine asynchrone Nachricht (als Ausgabe) an den Aktor zu schicken, und zu hoffen, dass er wie gewünscht reagiert (asynchron).**

Sequenzdiagramm: Häufige Ungeschicklichkeiten



1. Kontrolle durch Clients

Der Kartenleser sagt der Kasse was sie tun soll (CheckePin). Eigentlich soll die Kontrolle aber bei der Kasse liegen.



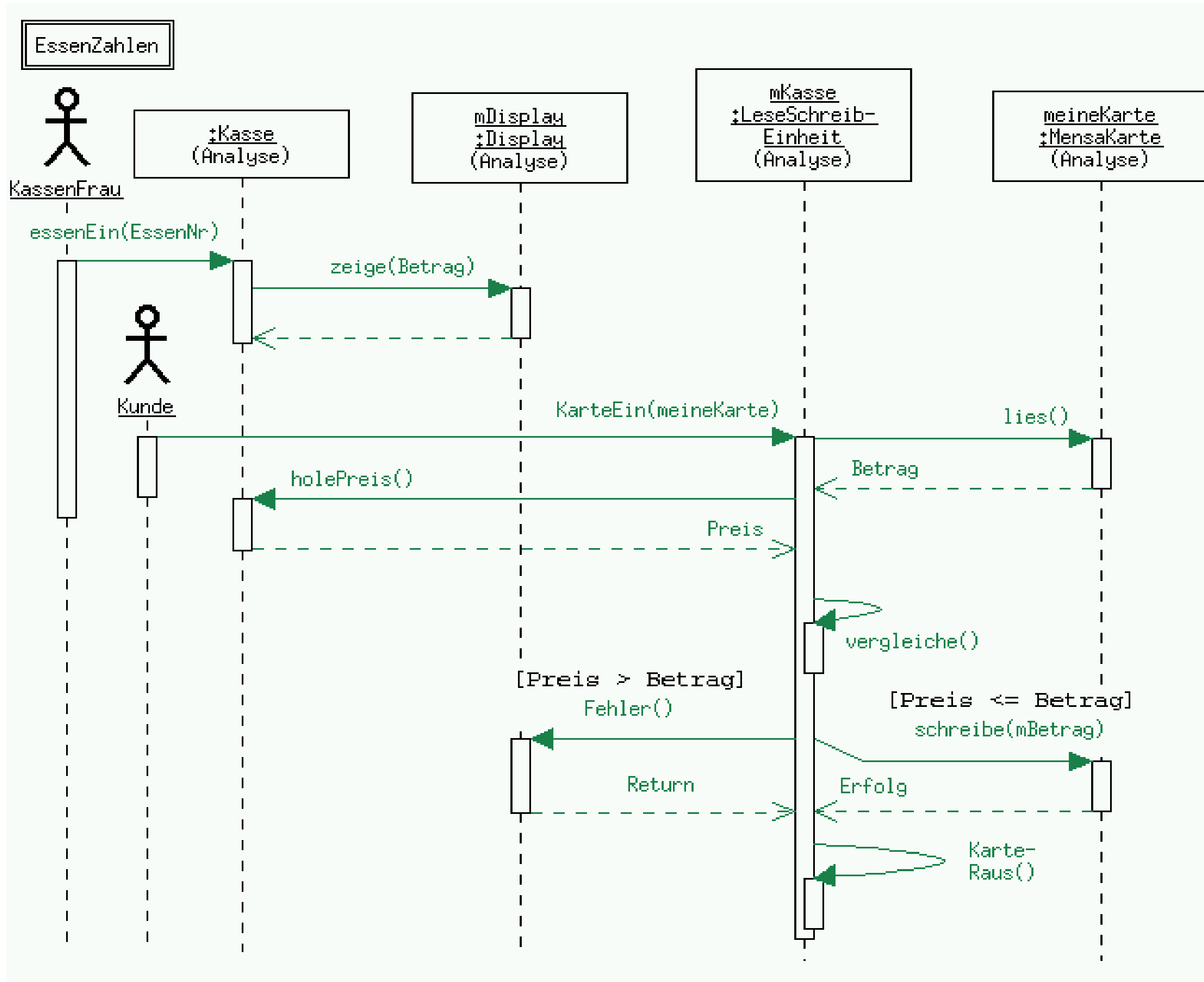
▶ **Besser wäre, nur das zu sagen was in der eigenen Verantwortung liegt. Dann kann die Kasse so reagieren wie sie es für richtig hält.**

Sequenzdiagramm: Schwierigeres Beispiel

Use Case *'Essen zahlen' (in der Mensa)*

Ein Mensakunde kommt mit seinem Essen zur Kasse. Die Kassenfrau gibt die Artikelnummern ein und die Kasse zeigt den Betrag an. Der Kunde schiebt seine Mensakarte in das Lesegerät und der Betrag wird abgebucht. Reicht das Guthaben nicht aus, wird eine Fehlermeldung angezeigt. Der Kunde entnimmt in beiden Fällen seine Karte.

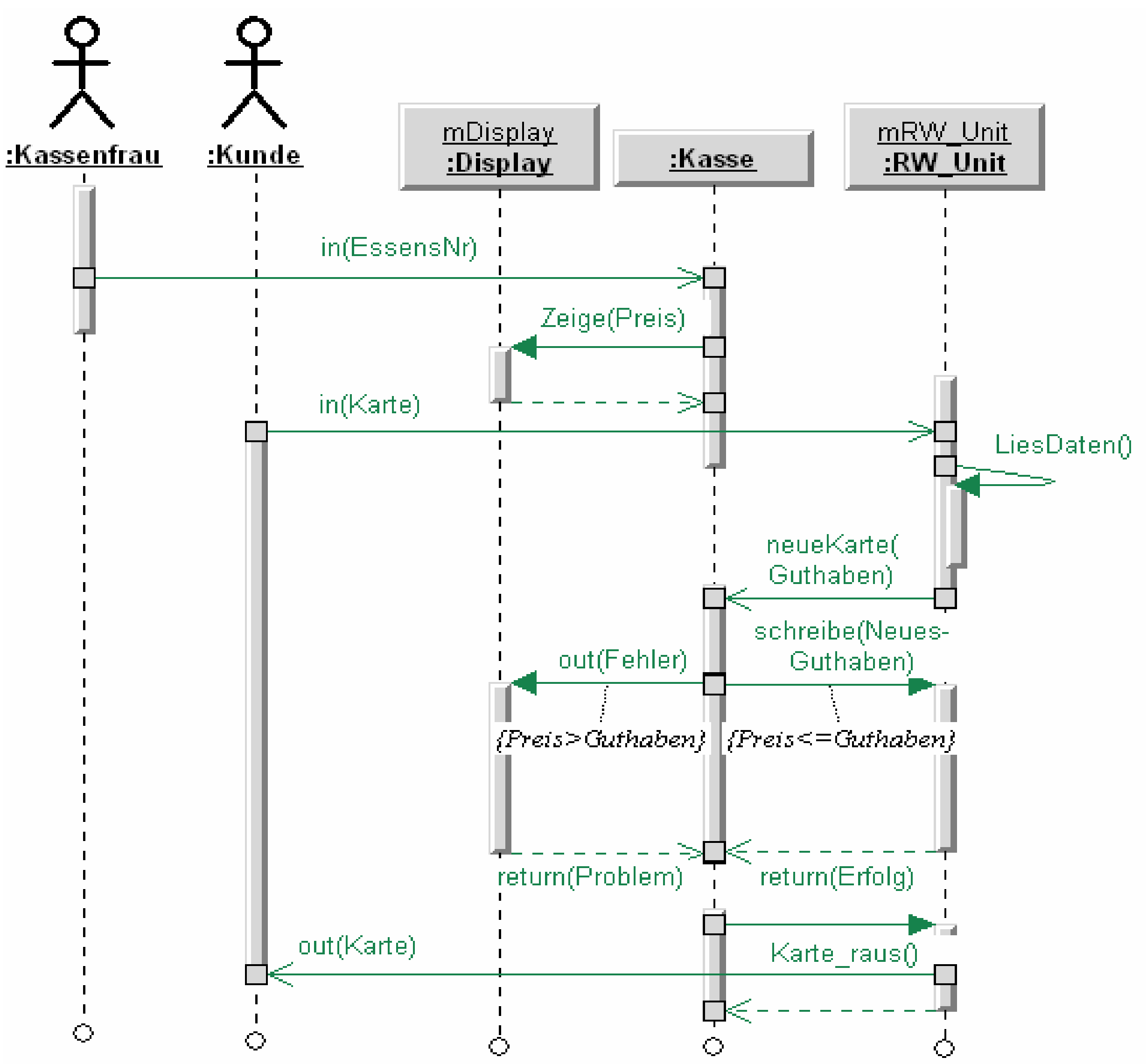
Sequenzdiagramm: Schwierigeres Beispiel – Diskussion einer Lösung



Anmerkungen:

- In dieser Lösung kontrolliert die Leseschreibeinheit den Gesamtvorgang. Alternativ könnte dies auch die *Kasse* tun.
- I/O wird unterschiedlich modelliert
- Gehört die *Karte* zu unserem System?
- Was würde passieren, wenn die Bezahlung durch einen Daumen-abdruck und direkte Abbuchung von einem Bank-Konto ersetzt würde?

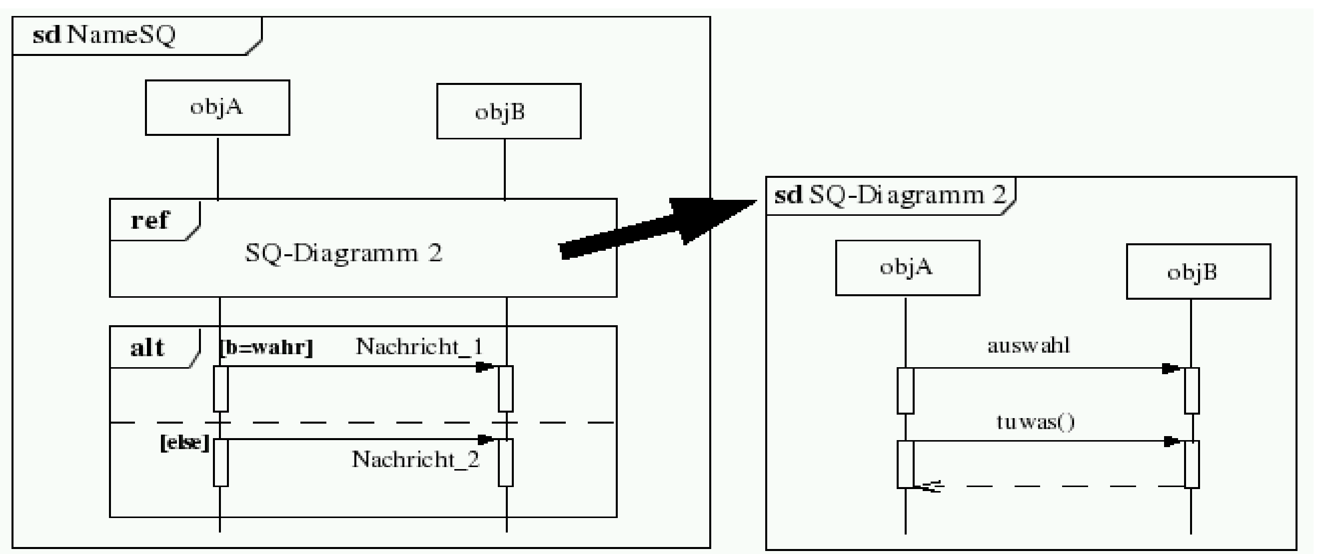
Sequenzdiagramm: Schwierigeres Beispiel – Alternative Lösung



Anmerkungen:

- In dieser Lösung kontrolliert die Kasse den Gesamtvorgang. Die RW_Unit kümmert sich bloß um die Karte.
- Die *Karte* wird nicht mehr modelliert, sondern übergeben
- Was würde passieren, wenn die Bezahlung durch einen Daumenabdruck und direkte Abbuchung von einem Bank-Konto ersetzt würde?

Sequenzdiagramm: Erweiterte Notation in UML 2.0 (I)

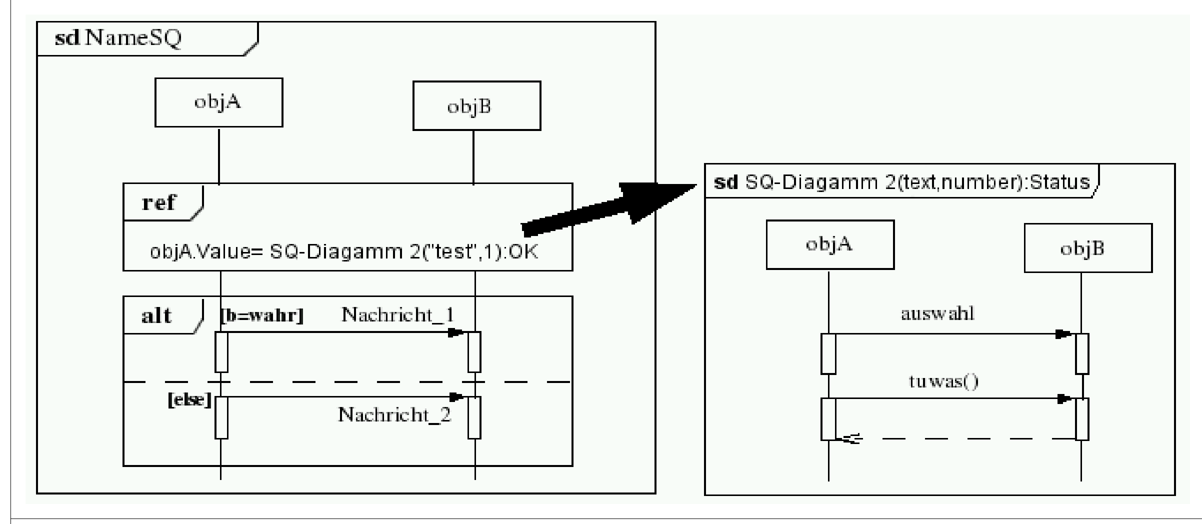


Links oben stehen der Diagrammtyp (sd) und der Diagrammname (NameSQ). Der mit ref bezeichnete Block stellt einen Verweis auf ein Diagramm mit dem Namen "SQ-Diagramm 2" dar, das eine Verfeinerung des Blockes enthält. Der mit alt markierte Block beschreibt eine Alternative (Verzweigung). Ist die Bedingung [b=wahr] erfüllt, wird der Bereich oberhalb der gestrichelten Linie ausgeführt, sonst der Bereich unterhalb der gestrichelten Linie.

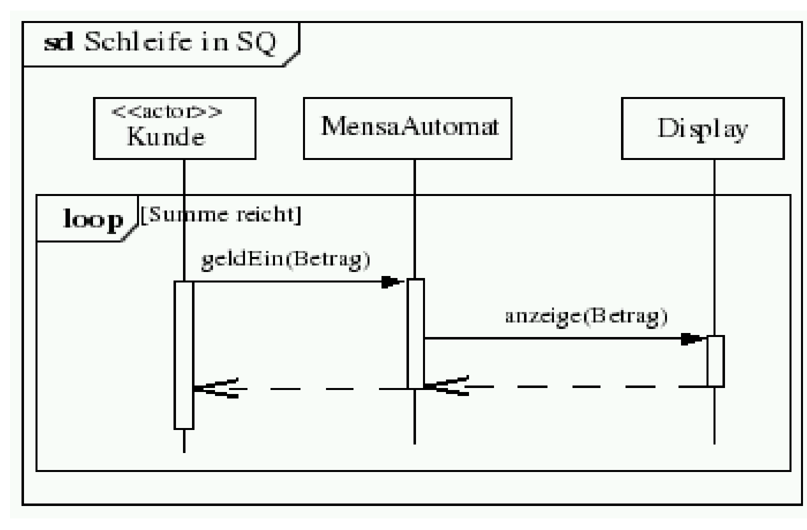
kombinierte Fragmente (combined fragment)

Mit kombinierten Fragmenten können in Sequenzdiagrammen die Kontrollstrukturen höherer Programmiersprachen ausgedrückt werden. Außerdem können damit in der UML 2.0 Sequenzdiagramme verschachtelt werden. Ein kombiniertes Fragment wird als rechteckiger Block mit fehlender Ecke dargestellt. In der linken, oberen Ecke trägt es eine Bezeichnung, die den Typ der Kontrollstruktur angibt (interaction operator). Die wichtigsten Interaktionsoperatoren in der UML 2.0 sind **ref** (Verweis), **alt** (Alternative) und **loop** (Schleife).

Sequenzdiagramm: Erweiterte Notation in UML 2.0 (II)



ref
Sequenzdiagramme können andere Sequenzdiagramme beinhalten (als eine Art Unter-programm). Dabei können beim "Aufruf" und in der "Definition" des Unterprogramms Para-meter übergeben bzw. spezifiziert werden.



loop
Eine Schleife in einem Sequenzdiagramm bewirkt, dass die enthaltene Sequenz von Botschaften solange ausgeführt wird, bis die Abbruchbedingung erfüllt ist.

Sequenzdiagramm: Anwendung

Sequenzdiagramme sind z.B. in folgenden Situationen nützlich:

- vor Implementierungsbeginn
  - zur Verifikation des Designs: besonders interessante (oder schwierige) Systemzustände anschaulich darstellen und "testen"
  - um zu verifizieren ob die Klassenaufteilung (Architektur) gut gewählt ist und eine lose Kopplung bietet
- im Team
  - verbesserte Kommunikation der wesentlichen Abläufe
- beim Test
- kritische Abläufe im System können für Tests spezifiziert (und evtl. generiert) werden

Sequenzdiagramme sind das wichstigste Interaktionsdiagramm in der UML!

Sequenzdiagramm: Anmerkungen

Sequenzdiagramme sind sehr leicht zu lesen…

… aber gar nicht so leicht zu erstellen!

- Welche Objekte sind (nicht) erforderlich?
  - ⇨ *Auf das Beschränken was wichtig ist bzw. dargestellt werden soll*
  - ⇨ *aber auch nicht zu wenige Objekte*
- Wer ruft wen auf?
  - ⇨ *Überlegen, wer den Trigger erhält, der die Aktion startet*
- Welche bzw. wieviele Alternativen sollen dargestellt werden?
  - ⇨ *nur die wichtigsten für das Szenario. Lieber mehrere Diagramme!*
  - ⇨ *oder kombinierte Fragmente verwenden*
- Wie wird die Interaktion mit der Außenwelt (Aktoren) modelliert?
  - ⇨ *Tipp: Asynchrone Aufrufe und I/O-Objekte verwenden*

auch hier iterativ vorgehen!

Diagramme: Diskussion

Vergleich

Use Cases	Sequenzdiagramm	Klassendiagramm
Was passiert?	**Wie** passiert es?	**Woraus** besteht das System?
Verbale Beschreibung	Formale Sequenz	Formale Darstellung
System als Black-Box	System als White-Box	Systembeschreibung
Aktoren und das System	Objekte und Methodenaufrufe	Klassen mit Attributen und Methoden & Assoziationen

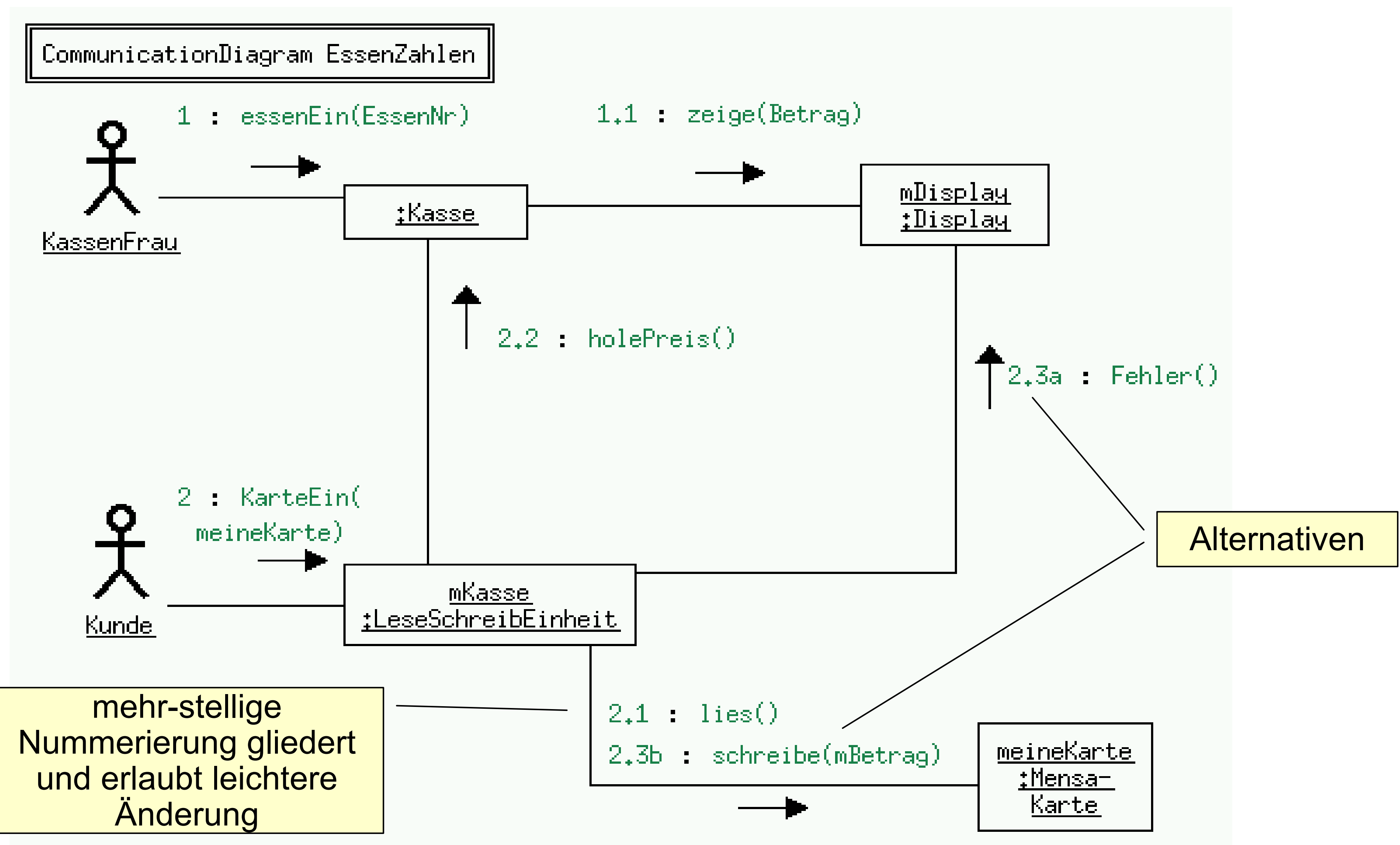
alle diese Diagramme haben Ihre Stärken und Schwächen…
…und sind alle sehr wichtig!

Kommunikationsdiagramm: Notation in UML 2.0

in UML 1.x:
"Kollaborationsdiagramme"

Objekt:Klasse	**Objekt** Das Rechteck stellt ein Objekt dar. Es kann einen Objektnamen und einen Klassennamen tragen.
1: Nachricht1 2: Nachricht2	**Beziehung, Nachricht** Die Kommunikationsbeziehungen zwischen Objekten werden als Linie dargestellt. An die Beziehungen werden die Nachrichten, die darüber übertragen werden geschrieben. Die Nummerierung vor den Nachrichtennamen gibt die zeitliche Reihenfolge der Nachrichten an. Der Pfeil gibt die Richtung der Nachrichten an.

Kommunikationsdiagramm: Beispiel 'Essen zahlen' (Auszug)



Kommunikationsdiagramm: Inhalt

Ein Kommunikationsdiagramm beschreibt

- die Interaktion zwischen Objekten im Überblick
- die zeitliche Reihenfolge, in der die Nachrichten gesendet werden (Nummerierung)
- Es gibt die Beziehungen zwischen den Objekten an und die Nachrichten, die übertragen werden. Dabei steht der Überblick der Kommunikationsstruktur im Vordergrund.

Anmerkungen

- Kommunikationsdiagramme werden für die selben Zwecke eingesetzt wie Sequenzdiagramme.

Kommunikationsdiagramm vs. Sequenzdiagramm

Kommunikationsdiagramm	Sequenzdiagramm
Objekte in der Fläche verteilt	Objekte in einer Linie
Stärke: viele Objekte, die wenige Nachrichten austauschen	Stärke: wenige Objekte, die viele Nachrichten austauschen
Parallelität nicht darstellbar	Parallelität durch Asynchronität (wird mit offenen Pfeilen angedeutet)
Keine Verschachtelung	verschachtelte Diagramme möglich

Kommunikationsdiagramme bieten nur eine Untermenge der Sequenzdiagramme!

Kommunikationsdiagramm: Anwendung

Kommunikationsdiagramme sind z.B. in folgenden Situationen nützlich:

- das Zusammenspiel zwischen vielen interagierenden Teilen soll möglichst einfach dargestellt werden
- Es geht um das grundsätzliche Verständnis des Ablaufs und der Verantwortungen – weniger um Details im Timing