

# Kapitel 3: Modellierung

1. Informatik: Eine Übersicht

2. Logik (Einführung / Grundlagen)

3. **Modellierung**

3.1 **Logische Modellierung**

3.2 **Einführung in die Software Modellierung**

3.3 **Grundlagen von UML**

4. Algorithmen und ihre Eigenschaften

5. Entwurfsmethoden für Algorithmen

6. Sortieralgorithmen

7. Dynamische Datenstrukturen



# 3.1 Logische Modellierung

## Warum Modellierung?

- Trennung von prozeduralem und deklarativem Wissen (Abstraktionsebene)
  - Bsp. Software:
    - ⇒ deklaratives Wissen ist nicht von Änderungen in der Software betroffen
    - ⇒ deklaratives Wissen kann unabhängig von der Software gewartet werden
- bessere Kommunikationsbasis und Verständnis im Kontext einer gegebenen Domäne



# 3.1 Logische Modellierung

Cyc [Lenat & Guha]

- Modellierung von allgemeinem Weltwissen

Aktuelle Initiativen:

- Standard Upper Ontology (SUO WG, <http://suo.ieee.org>)
- DOLCE (Descriptive Ontology for Linguistic and Cognitive Engineering, Laboratory for Applied Ontology, <http://www.loa-cnr.it/DOLCE.html>)
- WordNet (Princeton, <http://www.cogsci.princeton.edu/~wn/>)
- HALO (Modellierung von biochemischem Wissen)

Interdisziplinär: Informatiker, Linguisten, Logiker,  
Philosophen



AIFB

©AIFB

# 3.1 Logische Modellierung

Anwendungsorientierte Modellierung:

- domänenabhängig
- zweckorientiert

Beispiel: Modellierung eines Autos

Domäne: Auto

Zweck: - Bauanleitung

- Fehlerdiagnose und Erklärung

(z.B. Warum springt es nicht an?)



# 3.1 Logische Modellierung

## Was ist ein Auto?

$$\begin{aligned} \forall x \text{ auto}(x) \Rightarrow \exists m, r, t \text{ motor}(m) \wedge \text{hat}(x, m) \wedge \\ \text{vierRaeder}(r) \wedge \text{hat}(x, r) \wedge \\ \text{tank}(t) \wedge \text{hat}(x, t) \end{aligned}$$

Sind wir fertig?

Nein, wir haben keine ‚Bauanleitung‘ für unser Auto.



# 3.1 Logische Modellierung

Was folgt aus:

$\text{auto}(x) :$   $\exists m, r, t \text{ motor}(m) \wedge \text{hat}(x, m) \wedge$   
 $\text{vierRaeder}(r) \wedge \text{hat}(x, r) \wedge$   
 $\text{tank}(t) \wedge \text{hat}(x, t)$

$\neg \exists \text{ motor}(m) \wedge \text{hat}(x, m) :$   $\neg \text{auto}(x)$

$\neg \text{auto}(x) :$  NICHTS!

Grund: wir haben nur notwendige Bedingungen spezifiziert!





# 3.1 Logische Modellierung

## Hinreichende Bedingungen:

$$\forall x \text{ auto}(x) \Leftrightarrow \exists m, r, t \text{ motor}(m) \wedge \text{hat}(x, m) \wedge \\ \text{vierRaeder}(r) \wedge \text{hat}(x, r) \wedge \\ \text{tank}(t) \wedge \text{hat}(x, t)$$

Jetzt haben wir also unsere ‚Bauanleitung‘ für Autos, da wir nun wissen, welche Teile zum Bau von Autos ‚hinreichend‘ sind.

# 3.1 Modellierung

Modellierung komplexerer Beziehungen:

$$\forall x \text{ kannStarten}(x) \leftarrow \exists t \text{ auto}(x) \wedge \text{tank}(t) \wedge \text{hat}(x, t) \wedge \text{hatTreibstoff}(t)$$

$$\forall x \text{ gestartet}(x) \wedge \text{auto}(x) \Rightarrow \text{kannStarten}(x)$$

$$\forall x \text{ brummt}(x) \leftarrow \text{auto}(x) \wedge \text{gestartet}(x)$$

Diese Modellierung ermöglicht Fehlerdiagnose.

Was folgt zum Beispiel aus:

$$\exists x \exists t \text{ auto}(x) \wedge \text{tank}(t) \wedge \text{hat}(x, t) \wedge \neg \text{kannStarten}(x) \\ \neg \text{hatTreibstoff}(t)$$



# 3.1 Logische Modellierung

## Abstraktion:

$$\forall x \text{ kannStarten}(x) \Leftarrow \exists t, m \text{ motor}(m) \wedge \text{hat}(x, m) \wedge \text{motorgetrieben}(x) \wedge \\ \text{tank}(t) \wedge \text{hat}(x, t) \wedge \text{hatTreibstoff}(t)$$

Hier: unabhängig von „auto“, d.h. z.B. auch für Motorräder gültig!

Diese Regel kann also in verschiedenen Domänen verwendet werden!

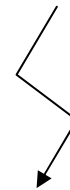
=> Ökonomie in der Modellierung



# 3.1 Logische Modellierung

## Modellrevision:

$\exists x, t, m \text{ motor}(m) \wedge \text{hat}(x, m) \wedge \text{motorgetrieben}(x) \wedge$   
 $\text{tank}(t) \wedge \text{hat}(x, t) \wedge \text{hatTreibstoff}(t) \wedge$   
 $\neg \text{kannStarten}(x)$



Was ist wenn der Motor defekt ist? Dann kann so eine Situation durchaus eintreten. Also müssen wir unser Modell modifizieren:

$\forall x \text{ kannStarten}(x) \Leftarrow \exists t, m \text{ motor}(m) \wedge \text{OK}(m) \wedge \text{hat}(x, m) \wedge$   
 $\text{tank}(t) \wedge \text{hat}(x, t) \wedge \text{hatTreibstoff}(t)$



# 3.1 Logische Modellierung



- Anomalien:

$$\exists a_1, a_2, m \text{ auto}(a_1) \wedge \text{auto}(a_2) \wedge \text{motor}(m) \wedge \\ \text{hat}(a_1, m) \wedge \text{hat}(a_2, m) \wedge a_1 \neq a_2$$

d.h. zwei (unterschiedliche) Autos, die den selben Motor haben?

Ist in unserem bisherigen Modell erfüllbar, also fügen wir folgende Regel hinzu:

$$\forall a_1, a_2, m_1, m_2 \text{ auto}(a_1) \wedge \text{auto}(a_2) \wedge a_1 \neq a_2 \wedge \\ \text{motor}(m_1) \wedge \text{motor}(m_2) \wedge \\ \text{hat}(a_1, m_1) \wedge \text{hat}(a_2, m_2) \Rightarrow m_1 \neq m_2$$

# 3.1 Logische Modellierung

Weitere Anomalien:

$$\exists a, x \text{ auto}(a) \wedge \text{motor}(x) \wedge \text{tank}(x) \wedge \text{hat}(a, x)$$

ist in unserem Modell erfüllbar, d.h. der Motor und der Tank sind das gleiche Bauteil! Das verbieten wird durch folgende Axiome:

$$\forall x \text{ tank}(x) \Rightarrow \neg \text{motor}(x)$$

$$\forall x \text{ motor}(x) \Rightarrow \neg \text{tank}(x)$$

d.h. *motor* und *tank* sind disjunkt!



# 3.1 Logische Modellierung

HALO Projekt (<http://www.projecthalo.com>)

- finanziert von Paul Allen, einem Microsoft-Mitgründer
- konzipiert als Wettbewerb
- Modellierung von biochemischen Wissen zur Beantwortung von Klausuraufgaben im Fach Chemie
- Ziel: automatische Generierung von Antworten inklusive Erklärung des Lösungswegs
- Bewertung durch 3 Professoren
- erstaunlich gute und skalierbare Ergebnisse!



AIFB

©AIFB

# Project Halo Downloads

Das Projekt ist sehr ausführlich dokumentiert, alle Systeme können heruntergeladen und ausprobiert werden!

Projekt Webseite:

<http://www.projecthalo.com/>

OntoNova System:

[http://www.projecthalo.com/downloads/improved/ontonovasetup\\_opt.exe](http://www.projecthalo.com/downloads/improved/ontonovasetup_opt.exe)

Dokumentation vom System:

[http://www.projecthalo.com/doc/teamdocs/ontoprise/Halo\\_Pilot\\_System\\_Instructions.pdf](http://www.projecthalo.com/doc/teamdocs/ontoprise/Halo_Pilot_System_Instructions.pdf)

Die Thematik wird vertiefend behandelt in den Vorlesungen

**Intelligente Systeme im WWW**

**Wissensmanagement**



AIFB

©AIFB

# Kapitel 3: Modellierung

1. Informatik: Eine Übersicht

2. Logik (Einführung / Grundlagen)

3. **Modellierung**

3.1 Logische Modellierung

3.2 **Einführung in die Software Modellierung**

3.3 Grundlagen von UML

4. Algorithmen und ihre Eigenschaften

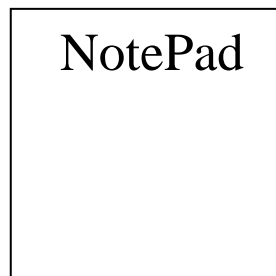
5. Entwurfsmethoden für Algorithmen

6. Sortieralgorithmen

7. Dynamische Datenstrukturen

# 3.2 Software-Modellierung

- Die Probleme bei der Entwicklung von großen Softwaresystemen sind nicht einfach Vergrößerungen der Probleme bei kleinen Softwaresystemen:
  - Es ist schwer eine große Zahl von beteiligten Softwareentwicklern (oft sogar mit unterschiedlichen Kenntnissen) zu koordinieren.
  - Es ist schwer den Aufwand der Realisierung des Systems zu berechnen, weil man nicht leicht feststellen kann, welche Aufgaben gleichzeitig gelöst werden können.
  - Idee: Bausteine erkennen und geschickt verteilen.
  - Lösung: Methodik zum Spezifizieren von Systemarchitektur.



vs.



## 3.2 Software-Modellierung

- Die Vorstellungen des Kunden und das erstellte Programm unterscheiden sich oft:
  - **Kommunikationsproblem** zwischen Kunden und Softwareentwickler, weil Programmiersprachen ungeeignet für derartige Kommunikation sind.
  - Idee: Ein nicht technisches und trotzdem präzises Kommunikationsmittel verwenden.
  - Lösung: Methodik, die das System beschreiben läßt, ohne auf Implementierungsdetails eingehen zu müssen.

# 3.2 Software-Modellierung

## Wichtige Phasen in der Softwareentwicklung

- **Anforderungsbeschreibung**
  - Mit Kunden eng zusammenarbeiten und seine Anforderungen spezifizieren
- **Analyse**
  - Detaillierte Analyse des Problembereichs und Anwendungsfällen
- **Entwurf**
  - System entwerfen und implementierungstechnische Entscheidungen treffen.

Hier hilft z.B. der Einsatz von abstrakten Sprachen zur Spezifikation wie UML (siehe später)



# 3.2 Software-Modellierung

## Wichtige Phasen in der Softwareentwicklung

- **Implementierung**
  - System implementieren (programmieren)
- **Test**
  - System testen, Qualitätssicherung
- **Inbetriebnahme**
  - System beim Kunden einführen
- **Einsatz**
  - System warten, pflegen, aktualisieren etc.



# Kapitel 3: Modellierung

1. Informatik: Eine Übersicht

2. Logik (Einführung / Grundlagen)

3. **Modellierung**

3.1 Logische Modellierung

3.2 Einführung in die Software Modellierung

3.3 **Grundlagen von UML**

4. Algorithmen und ihre Eigenschaften

5. Entwurfsmethoden für Algorithmen

6. Sortieralgorithmen

7. Dynamische Datenstrukturen

# 3.3 Grundlagen von UML

## Entwicklungsgeschichte

- Ende 80er, Anfang 90er: erste Bücher zur Modellierung.
- Mehrere verschiedene graphische Notationen, zunächst nur wenige Anwender.
- 1991-1994: Bestrebungen Standards einzuführen (Booch, Rumbaugh und andere).
- 1994: Booch und Rumbaugh einigen sich auf „Unified Method“.
- 1996: Booch, Rumbaugh und Jacobson veröffentlichen „Unified Modeling Language“ (UML) 0.9.
- Juni 1998: UML 1.2
- Juni 1999: UML 1.3
- Nov 2000: UML 1.4 Beta.
- UML 2.0 Working Group arbeitet zur Zeit an UML 2.0 (‘near completion’ laut Webseite)





## 3.3 Grundlagen von UML

Die **Unified Modelling Language** unterstützt u.a.:

- die programmiersprachenunabhängige Repräsentation von statischen Strukturen:  
Klassen, Attribute, Methoden etc.
- die Spezifikation von dynamischen Aspekten:  
Abläufe, Prozesse, Workflows
- die Erfassung und Spezifikation von 'use cases'

Sie findet Anwendung in den ersten Phasen der Softwareentwicklung (Anforderungsbeschreibung, Analyse, Entwurf)

# 3.3 Grundlagen von UML

## Darstellung von Klassen

- Eine Klasse entspricht einer Sorte von Gegenständen in der zu realisierenden Anwendungswelt.
- Fasst zusammengehörende Merkmale (Attribute) und Aktivitäten (Operationen) des Gegenstands zusammen.
- Darstellung einer Klasse enthält
  - Name der Klasse: meistens Substantiv im Singular
  - Liste der Attribute
  - Liste der Operationen

Klassenname	Student/In
Attribute	+ name : String + vorname : String
Operationen	+ matrikelNr : Integer + void anmelden(Tutorium tutorial1)

# 3.3 Grundlagen von UML

## Darstellung von Attributen

- Ein Attribut in einer Klasse beschreibt ein Merkmal des entsprechenden Gegenstands
- Die Darstellung eines Attributes enthält
  - Sichtbarkeit: public (+), protected (#), private (-)
  - Name des Attributes: meistens kleingeschrieben
  - Typ des Attributes: Elementarer Typ oder eine Klasse

```
+ Attribut1 : Typ  
# Attribut2 : Typ  
- Attribut3 : Typ
```

```
+ name : String  
+ vorname : String  
+ matrikelNr : Integer
```

# 3.3 Grundlagen von UML

## Darstellung von Operationen

- Eine Operation einer Klasse beschreibt eine Aktivität des entsprechenden Gegenstands
- Die Darstellung einer Operation enthält
  - Sichtbarkeit: public, protected, private
  - Typ des Rückgabewertes: Elementarer Typ oder eine Klasse
  - Name der Operation: meistens kleingeschrieben
  - Liste der Argumente(Typ-Name Paare): Elementarer Typ oder eine Klasse, Name meistens kleingeschrieben

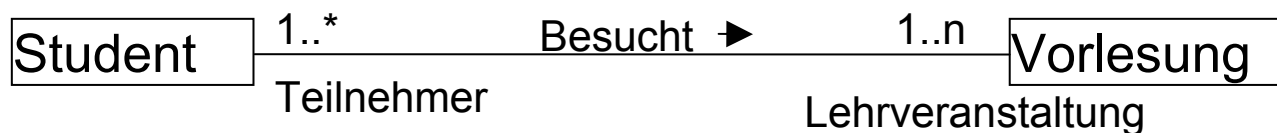
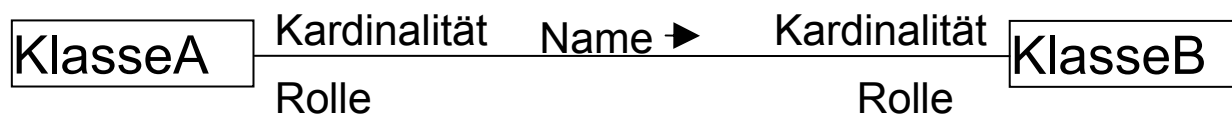
```
+ Typ Operation1(Typ arg1, ...)  
# Typ Operation2(Typ arg1, ...)  
- Typ Operation3(Typ arg1, ...)
```

```
+ void anmelden(Tutorium tutorial1)
```

# 3.3 Grundlagen von UML

## Darstellung von Assoziationen zwischen Klassen I

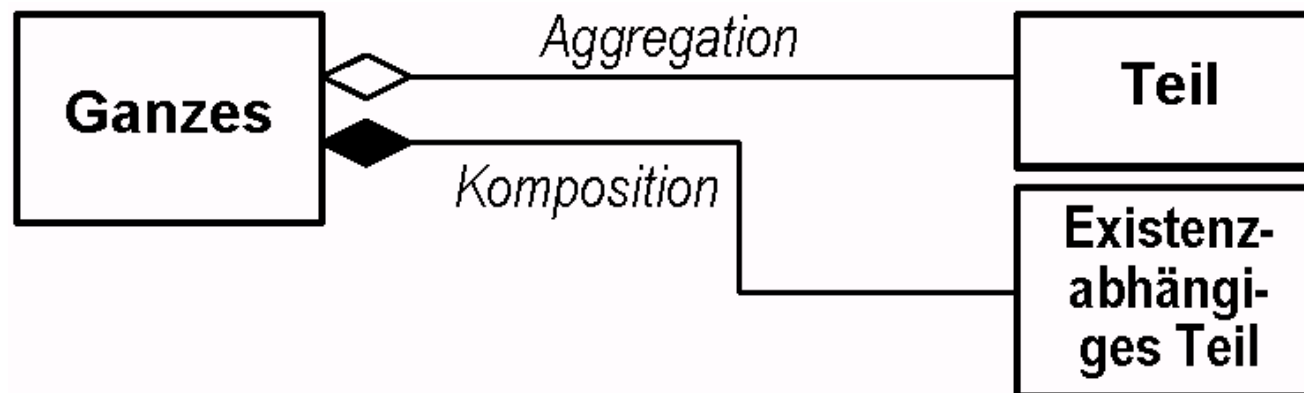
- Assoziation zwischen zwei Klassen beschreibt die Beziehung zwischen den Klassen
- Die Darstellung einer Assoziation enthält
  - Name der Assoziation mit Leserichtung (Pfeilspitze)
  - Rollen: Für jede Klasse eine Rolle, die die Klasse in der Assoziation spielt.
  - Kardinalitäten: Für jede Klasse eine Kardinalität, die angibt, wie viele Objekte dieser Klasse beteiligt sind  
 (\* = beliebig, x = genau x, x..y = Spanne von x bis y)



# 3.3 Grundlagen von UML

## Darstellung von Assoziationen zwischen Klassen II

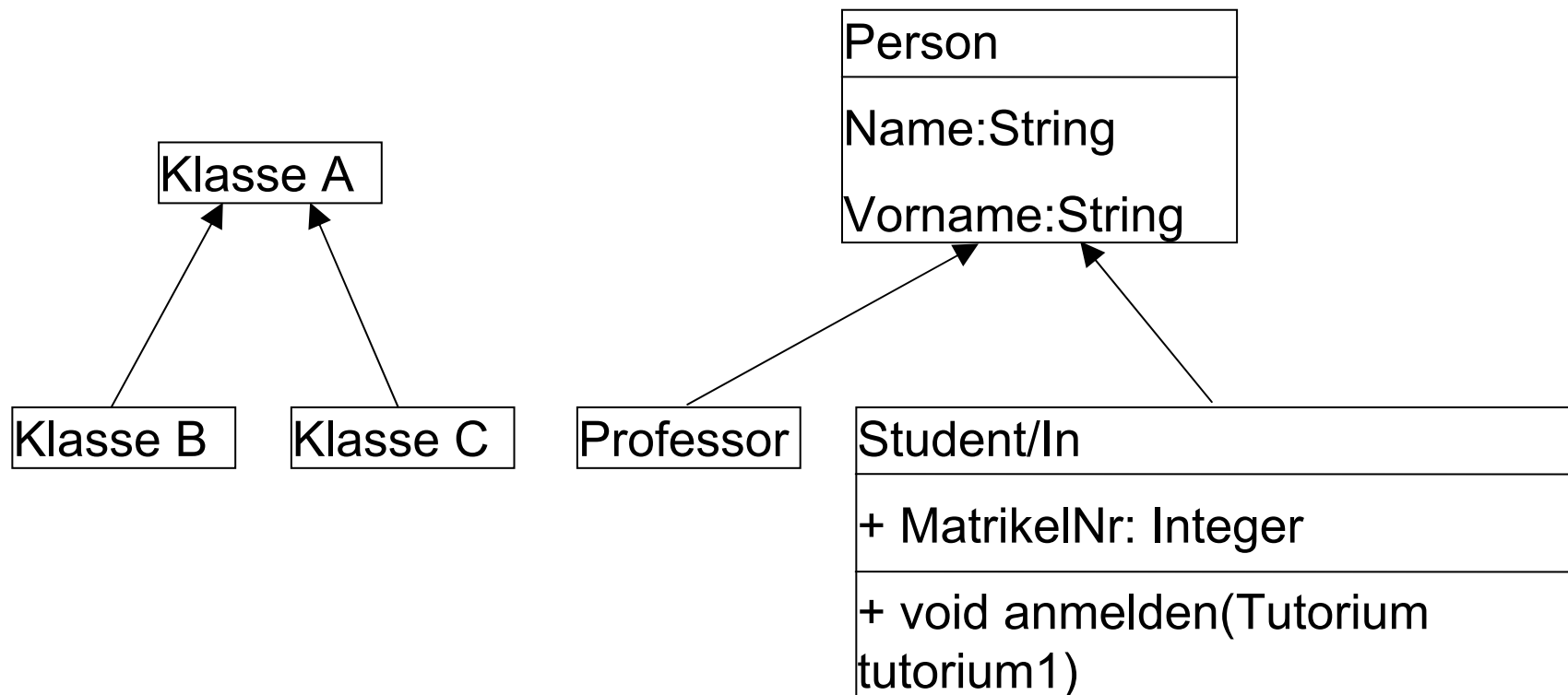
- **Aggregation:** ist eine “hat-ein”-Relation zwischen einem Ganzen und seinen Teilen.
  - Beispiel: Team hat Mitarbeiter. Zerfällt das Team bleiben die Mitarbeiter erhalten.
- **Komposition:** ist die engere “enthält-ein”-Relation, bei der die Teile nicht ohne das Ganze existieren.
  - Beispiel: Firma besteht aus Teams und Mitarbeitern, d.h. wenn die Firma nicht mehr existiert, sind auch die Teams und Mitarbeiter nicht mehr existent.



# 3.3 Grundlagen von UML

## Darstellung von Vererbung von Klassen

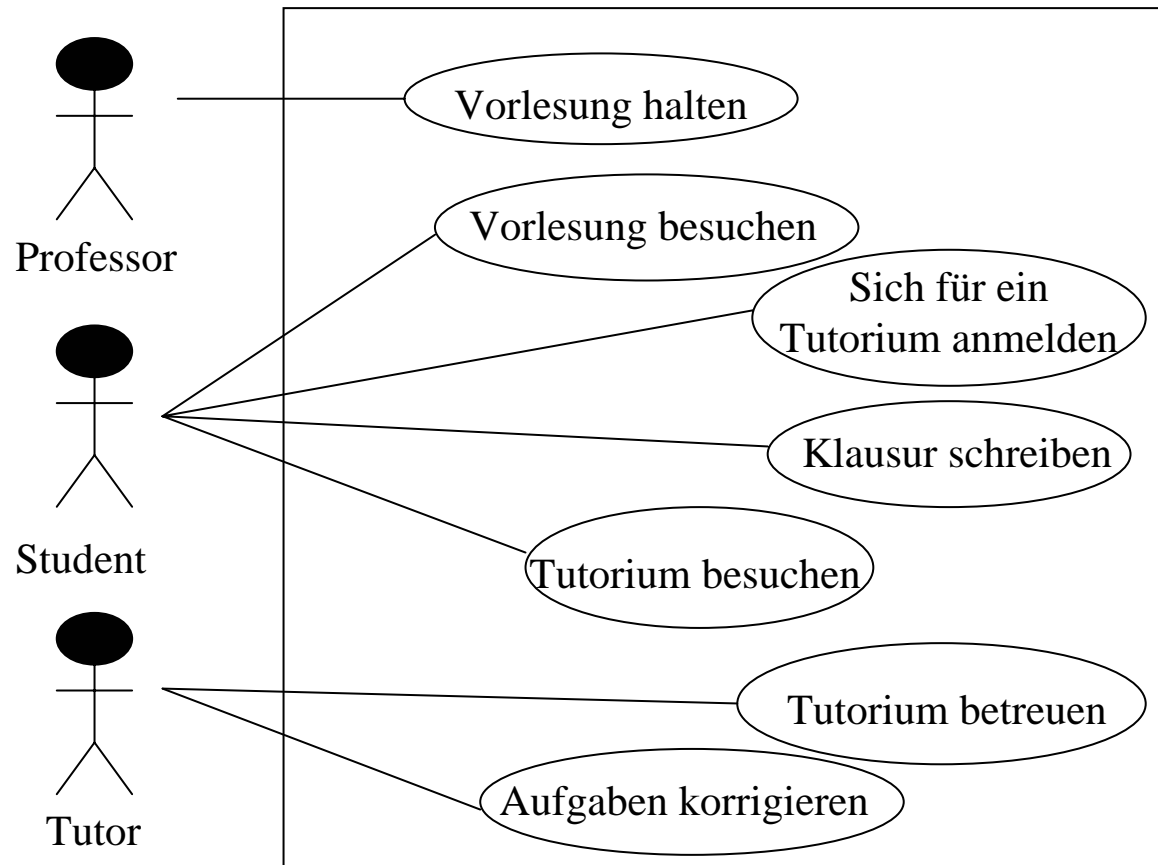
- Gemeinsame Attribute und Operationen von Klassen herauskapseln.
- Nur zusätzliche Attribute und Operationen in Unterklassen spezifizieren.
- Darstellung durch einen Pfeil in Richtung “erbt von”



# 3.3 Grundlagen von UML

## Darstellung von Anwendungsfallmodellen

- Beschreibt die gewünschte Systemfunktionalität aus Benutzungssicht, indem es die einzelnen vom System zu unterstützenden Anwendungsfälle definiert.
- sehr grobe Modellierung, Übersicht über Gesamtsystem

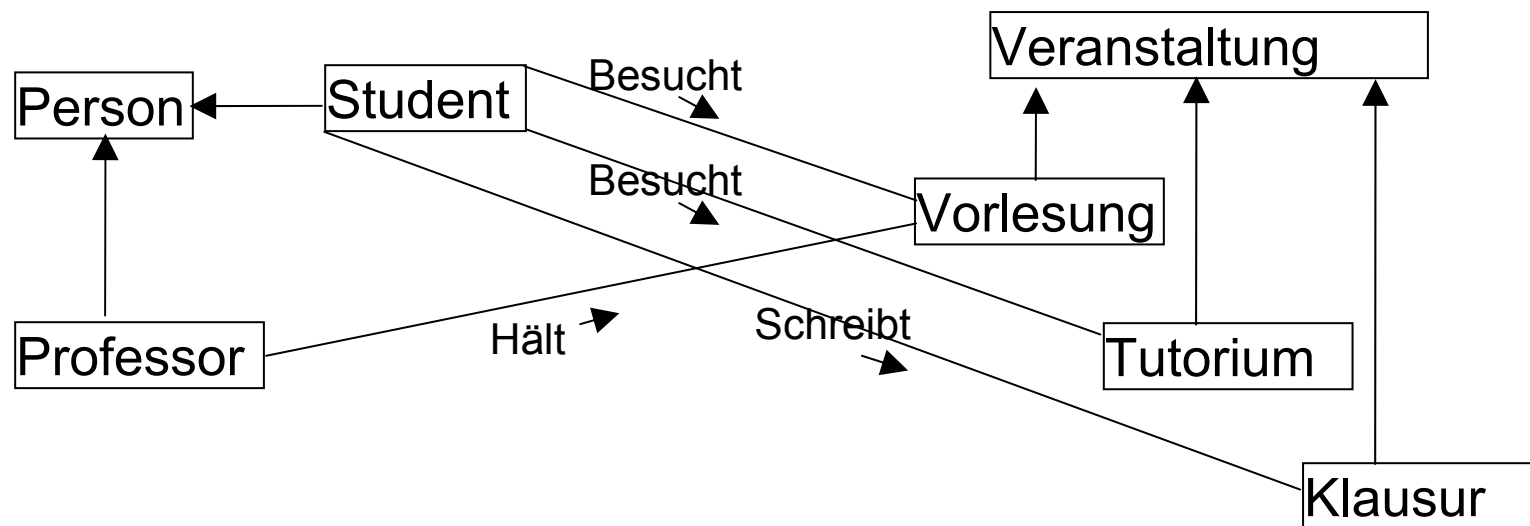


# 3.3 Grundlagen von UML

## Darstellung von statischen Modellen (Strukturmodellen)

• Beschreibt alle Klassen mit ihren Attributen, Operationen, Assoziationen mit anderen Klassen und Vererbungshierarchie

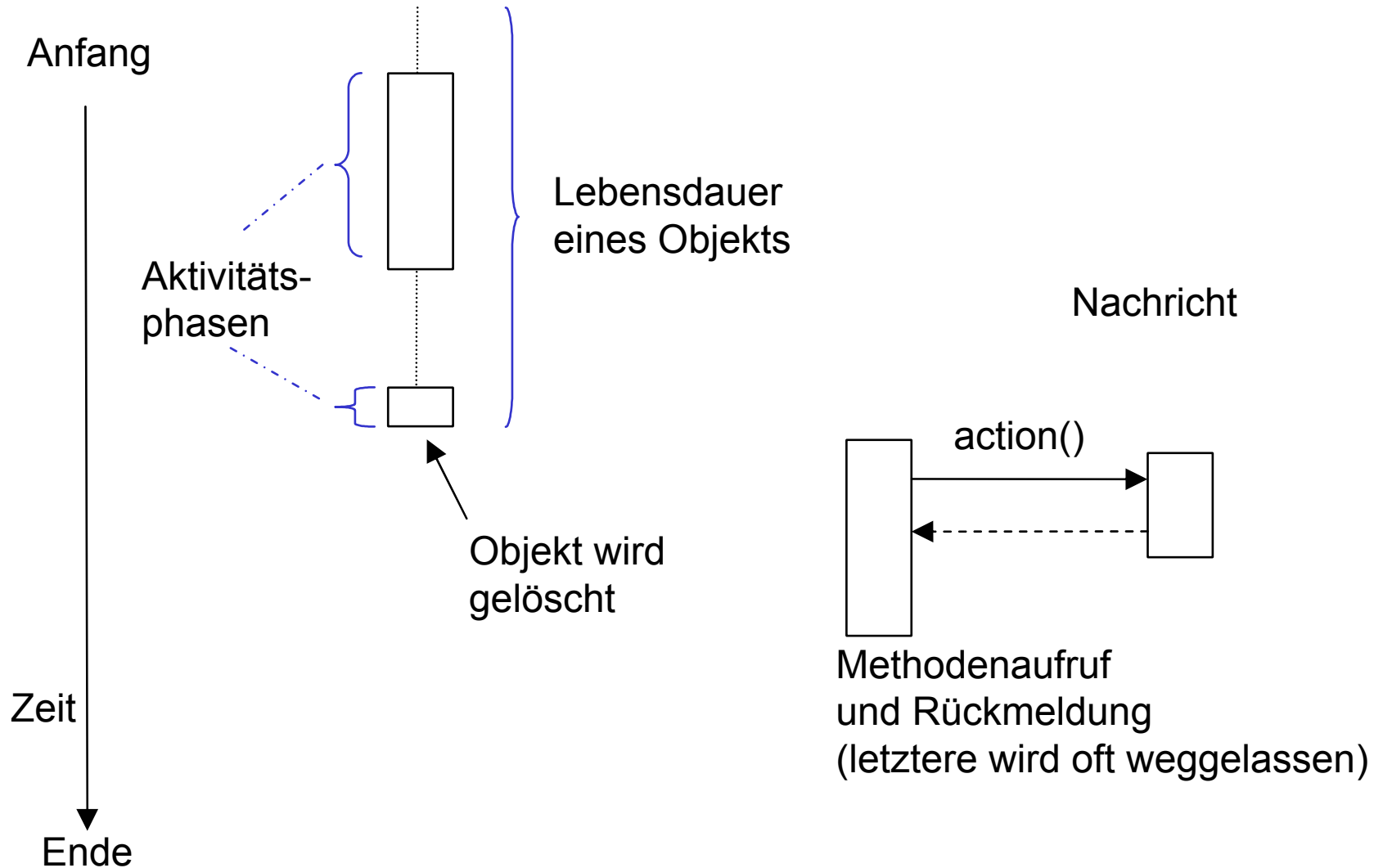
Beispiel: **Klassendiagramm** (sehr häufig verwendetes UML-Diagramm, kann in ersten Entwürfen auch ohne Kardinalitäten verwendet werden):



# 3.3 Grundlagen von UML

## Darstellung von dynamischen Modellen

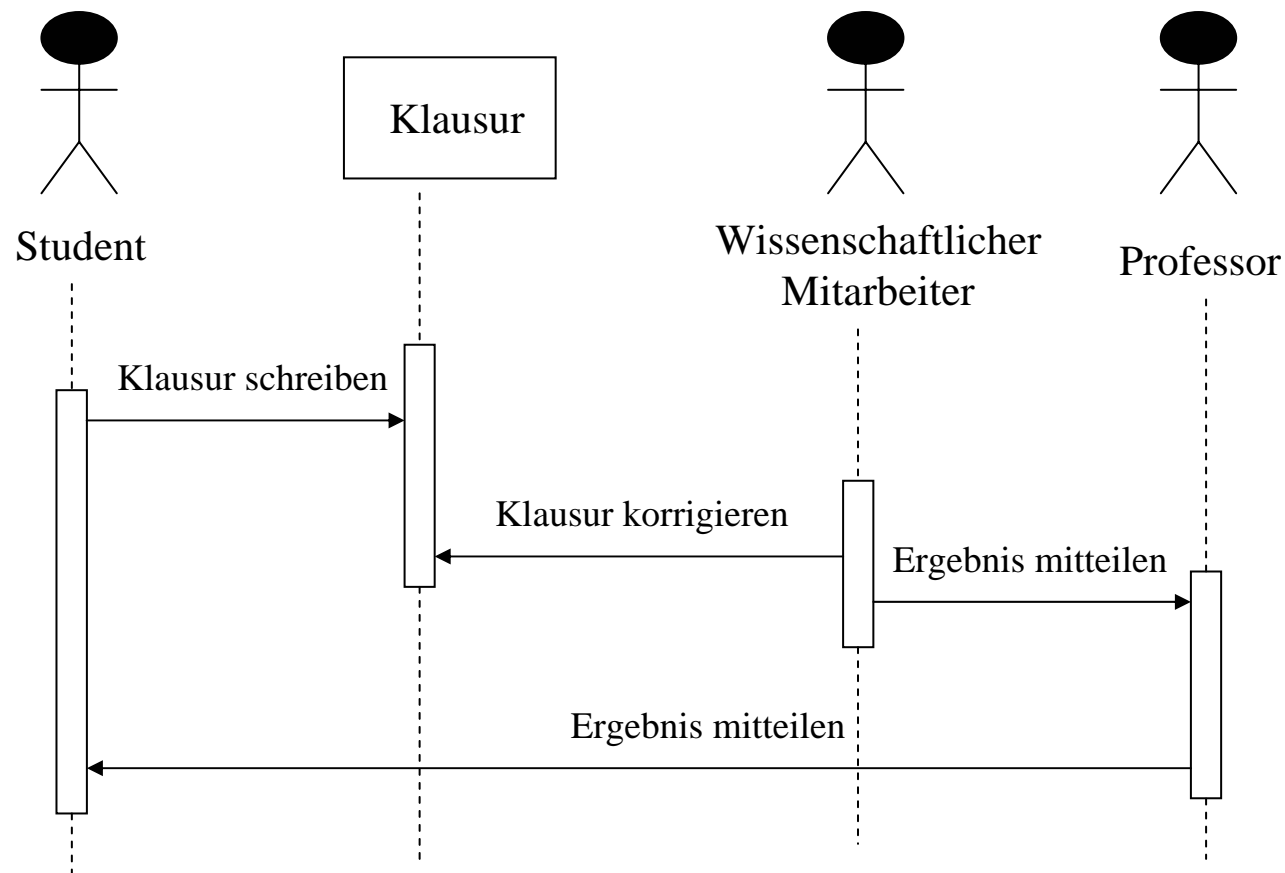
- Beschreibt das dynamische Verhalten des Systems
- Beispiel: **Sequenzdiagramm**



# 3.3 Grundlagen von UML

## Darstellung von dynamischen Modellen

Beispiel: Sequenzdiagramm



# 3.3 Grundlagen von UML

## Abschließende Bemerkungen:

### Vorteile:

- Industriestandard (etabliert)
- Modellierung auf unterschiedlichen Ebenen wird unterstützt (Klassen, Prozesse, etc.)
- Toolunterstützung (z.B. Rational Rose)
- Vereinfacht Kommunikation zwischen Entwicklern erheblich

### Nachteile:

- Bruch zwischen Spezifikation und Implementierung
- lange Entwicklungszyklen  
(vs. Extreme Programming)

