TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa / Arnd Poetzsch-Heffter

AG Softech
FB Informatik
TU Kaiserslautern

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Objektorientierte Modellierung

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Zur objektorientierten Modellierung

Wir geben hier nur eine kurze Einführung; OO-Modellierung ist Gegenstand der Vorlesung SE 2.
(siehe auch Goos, Band 2, 10.2)

Die Analyse einer Aufgabenstellung führt zu einem *Modell* des Ausschnitts der Welt, der zur Lösung geeignet ist.

Das Modell beschreibt die wichtigen Eigenschaften des zu entwickelnden Systems. Es orientiert sich zunächst an der Anwendung und nicht der Realisierung.

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

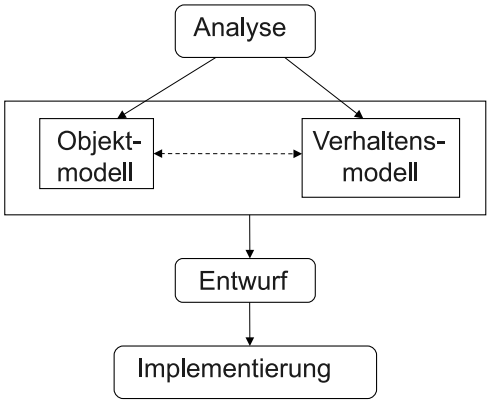
Aspekte der Modellierung

Bei objektorientierter Modellierung ist zu klären:

1. Welche Objekte werden benötigt?
2. Welche Beziehungen gibt es zwischen den Objekten?
3. Welche Eigenschaften besitzen die Objekte?
4. Wie lassen sich die Objekte klassifizieren?
5. Wie werden die Objekte angewendet?
6. Was ist das Verhalten der Objekte?

Das *Objekt-/Klassenmodell* liefert die Antworten zu 1-4.
Anwendungsfälle beantworten Frage 5.
Das *Verhaltensmodell* klärt Frage 6.

Objektorientierte Analyse in Umfeld der objektorientierten Softwareentwicklung



Begriffsklärung: Objekt-, Verhaltensmodell

Das **Objektmodell** beschreibt

- die relevanten Klassen von Objekten,
- welche Dienste/Nachrichten/Operationen bereitgestellt werden,
- die Beziehungen zwischen den Objekten.

Das **Verhaltensmodell** beschreibt

- die Wirkungsweise der Methoden,
- die möglichen Zustände von Objekten,
- das Ablaufverhalten und die Interaktion zwischen den Objekten.

Objektorientierte Analyse: Ein Beispiel

Aufgabe:

Entwickle ein Informationssystem für eine Universität.

Grober Leistungsumfang:

Universitäten bestehen aus Studierenden und Kursen mit folgenden Beziehungen:

- Jede(r) Studierende hat einen Namen, eine eindeutige Matrikelnummer und belegt eine Reihe von Kursen.
- Jeder Kurs hat einen Titel und eine maximale Teilnehmerzahl.

Typische **Anwendungsfälle** (engl. *use cases*) sind:

- Die Universität kann Studierende im- und exmatrikulieren.
- Studierende können angebotene Kurse belegen.
- Man kann eine Liste aller Studierenden der Universität sowie einzelner Kurse ausgeben.

Ermitteln des Verhaltens

Wir verfolgen einen *top-down* Ansatz, d.h. wir modellieren zuerst das Verhalten der Objekte abstrakt, bevor wir uns der Implementierung und den Details der Umsetzung widmen.
Das Verhalten von Objekten wird durch das Versenden von Nachrichten und das Bearbeiten der Nachrichten durch Methoden modelliert.

Wir betrachten hier zwei unterschiedliche Möglichkeiten, Verhaltensaspekte zu beschreiben:

- Skizzieren der wesentlichen Anwendungsfälle
- Beschreibung der Nachrichten, die die Objekte verstehen (*Methodenschnittstelle*)

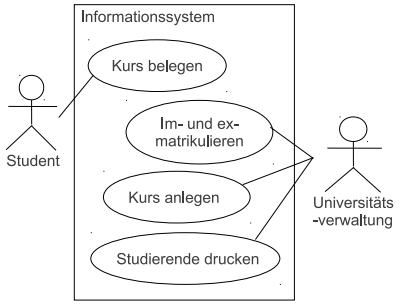
Anwendungsfälle (Use cases)

Ausgangspunkt für die Verhaltensmodellierung sind die wesentlichen Anwendungsfälle des zu entwickelnden Systems.

Ein Anwendungsfall ist ein typischer Vorgang, der mit dem zu realisierenden System durchgeführt wird.

Anwendungsfälle verdeutlichen auch die Abgrenzung des Systems von seiner Umgebung.

Anwendungsfälle des Beispiels



Studierende und Universitätsverwaltung sind Beispiele für **Akteure**.
Akteure können sein:

- Benutzer des Systems
- agierende Softwareteile der Systemumgebung

Methodenschnittstellen

Schnittstellenbeschreibungen sind ein wichtiger Baustein in der Modellierung, aber auch Implementierung von objekt-orientierten Systemen.

- Sie bieten klare Vereinbarungen (*contracts*) über die Interaktion von Klassen und Objekten.
- Sie klassifizieren Objekte nach ihrem Verhalten.
- Sie verbergen die tatsächliche Implementierung ($\rightarrow$ Information hiding).
- Sie verhindern (ungewollte) Abhängigkeiten zwischen Klassen.
- Sie vereinfachen die Zusammenarbeit von Entwicklerteams und ermöglichen eine klare Aufgabenteilung.
- Sie erlauben ein einfaches Verändern von Klassenimplementierungen, da nur wenig im Anwendungscode angepasst werden muss.

Beispiel

<<interface>>
Student

getName() : String
getMatrikel() : int
enroll(String courseTitle) : boolean

<<interface>>
University

register(String name) : Student
deregister(int matrikel)
findCourse(String courseTitle) : Course
addCourse(Course c)
printStudents()

<<interface>>
Course

getTitle() : String
enroll(Student s) : boolean
printStudents()

Interfaces in Java

Java stellt als Sprachmittel **Interfaces** zur Verfügung, um Schnittstellen zu beschreiben.
Ein **Interface** deklariert einen Referenztyp T und beschreibt die öffentliche Schnittstelle, die alle Objekte von T haben.

Syntax:

```
<Modifikatiorenlist> interface <Schnittstellenname>
    [  extends <Liste von Schnittstellennamen> ]
{
   <Liste von Konstantendeklarationen und
    Methodensignaturen>
}
```


Beispiel: Deklaration von Interfaces

```
interface Course {
  String getTitel();
  boolean enroll(Student s);
  void printStudents;
}

interface Student {
  String getName();
  int getMatrikel();
  boolean enroll(String courseTitle);
}
```


Implementierung von Schnittstellen

Zu einem Schnittstellentyp T lassen sich keine Objekte erzeugen, die nur zu T gehören.

Um eine Schnittstelle verwenden zu können, müssen daher Klassen diese Schnittstelle implementieren. Sie stellt dann für jede Methode, die in dem Interface deklariert wird, eine Implementierung bereit.

In Java wird durch `implements` in der Klassendeklaration angezeigt, welche Schnittstellen durch eine Klasse implementiert werden.

Es kann mehr als eine Klasse eine Schnittstelle implementieren.

Beispiel: Implementierung von Schnittstellen I

```
class Seminar implements Course {
  private String title;
  private int capacity;
  private StudentList students;

  Seminar(String title, int capacity) {
    this.title = title;
    this.capacity = capacity;
    this.students = new StudentList();
  }
  String getTitel() {
    return title;
  }
  boolean enroll(Student s) {
    if (students.size() < capacity) {
      students.add(s);
      return true;
    }
    return false;
  }
  void printStudents() {
    StdOut.println(students);
  }
}
```

Beispiel: Implementierung von Schnittstellen II

```
class Lecture implements Course {
  private String title;
  private String lecturer;
  private StudentList students;

  Lecture(String title, String lecturer) {
    this.title = title;
    this.lecturer = lecturer;
    this.students = new StudentList();
  }

  String getTitel() {
    return title;
  }
  String getLecturer() {
    return lecturer;
  }
  boolean enroll(Student s) {
    students.add(s);
  }
  void printStudents() {
    Stdout.println(students);
  }
}
```

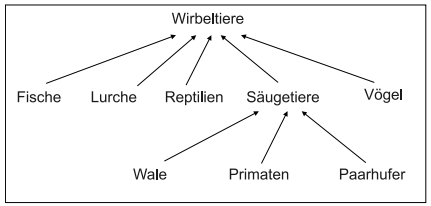

Klassifizieren von Objekten

Begriffsklärung: Klassifikation

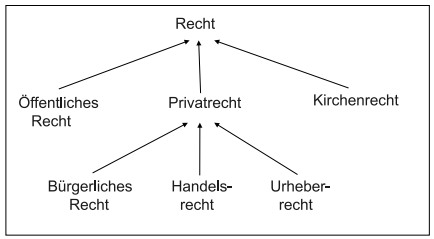
Klassifikation ist eine zentrale Grundlage der objektorientierten Modellierung und Programmierung.

Klassifizieren ist eine allgemeine Technik, mit der Wissen über Begriffe, Dinge und deren Eigenschaften hierarchisch strukturiert wird. Das Ergebnis nennen wir eine **Klassifikation**.

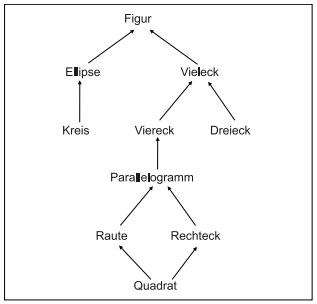
Beispiele: Klassifikationen I

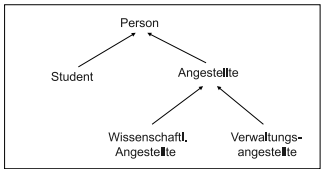


→ _ ist_ein _ (Vogel ist ein Wirbeltier)



Beispiele: Klassifikationen II





Bemerkungen

- Klassifikationen können baumartig oder DAG[1]-artig sein. D.h. Klassifikationen haben eine Richtung ("ist ein"-Beziehung) und dürfen keine Zyklen enthalten.
- Üblicherweise stehen die allgemeineren Begriffe oben, die spezielleren unten.
- Es gibt abstrakte Klassen (ohne "eigene" Objekte) und nicht abstrakte Klassen, von denen Objekte abgeleitet werden können.

[1] DAG = directed acyclic graph

Klassifikation in der Softwaretechnik

Objekte lassen sich nach ihren Eigenschaften klassifizieren:

- Alle Objekte mit ähnlichen Eigenschaften werden zu einer Klasse zusammen gefasst.
- Die Klassen werden hierarchisch geordnet.

Klassifikation beruht auf:

- Schnittstellen der Klassen/Objekte
- Verhalten/Eigenschaften der Objekte

Klassifikationen und Typisierung

Ein Typ beschreibt Eigenschaften von Objekten bzw. Werten.

Ansatz:

- Realisiere jede Klasse / jeden Art einer Klassifikation im Programm durch einen Typ.

Übersicht: Typsystem von Java I

Werte in Java sind

- die Elemente der elementaren Datentypen,
- Referenzen auf Objekte,
- der Wert `null`.

Jeder Wert in Java gehört zu mindestens einem Typ. Man unterscheidet dabei

- die vordefinierten elementaren Basisdatentypen,
- die durch Klassen deklarierten Typen,
- die durch Interfaces deklarierten Typen.

Übersicht: Typsystem von Java II

Darüber hinaus gibt es noch die Feld-/Arraytypen mit Typkonstruktor `[]` .

Feld-, Klassen- und Schnittstellentypen fasst man unter dem Namen Referenztypen zusammen.

(`null` gehört zu allen Referenztypen.)

Klassen- und Schnittstellentypen beschreiben, welche Nachrichten ihre Objekte verstehen bzw. welche Methoden sie besitzen.

Sub-/Supertypen

Wir führen nun eine partielle Ordnung $\leq$ auf Typen ein, so dass

- speziellere Typen gemäß der Ordnung kleiner als ihre allgemeineren Typen sind und
- Objekte speziellerer Typen auch zu den allgemeineren gehören.

Wenn $S \leq T$ gilt, dann sind alle Objekte vom Typ S auch vom Typ T. S ist ein **Subtyp** von T, und T ist ein **Supertyp** von S.

Wenn $S \leq T$ und $S \neq T$, heißt S ein **echter** Subtyp von T, in Zeichen $S < T$.

Wenn $S < T$ und es kein U mit $S < U < T$ gibt, dann heißt S ein **direkter** Subtyp von T.

Typen als Mengen

Vereinfachend betrachtet, kann man Typen als die Menge ihrer Objekte bzw. Werte auffassen.

Bezeichne $M(S)$ die Menge der Objekte vom Typ S.

Für Typen S und T gilt dann:

$$S \leq T \quad \text{impliziert} \quad M(S) \subseteq M(T)$$

Beispiel: Subtypbeziehungen

- Für die elementaren Datentypen in Java gelten z. B. die folgenden Subtypbeziehungen:

$$\texttt{float} < \texttt{double}$$
$$\texttt{int} < \texttt{long}$$
$$\texttt{long} < \texttt{float}$$

- Im Beispiel Universitätsverwaltung bestehen folgende Subtyp-Beziehungen:

$$\texttt{Lecture} < \texttt{Course}$$
$$\texttt{Seminar} < \texttt{Course}$$

Zwischen `Lecture` und `Seminar` gibt es keine Subtyp-Beziehung.

Die Klasse `Object`

Die Klasse `Object` spielt in Java eine spezielle Rolle.

Sie ist die Wurzel der Typhierarchie und stellt einige wichtige Methoden (z.B. `toString()`) bereit.

- Alle Typen sind Subtypen von `Object`.
- Alle Objekte (inkl. Arrays) erben die Methoden von `Object` ($\Rightarrow$ nächste Vorlesung "Vererbung").

Beispiel: Typhierarchie I

```
interface Printable {
  void print();
}

interface Person {
  String getName();
  String getBirthdate();
}

interface Angestellte
   extends Person, Printable { ... }
interface Student
   extends Person, Printable {... }

class WissAngestellte
   implements Angestellte { ... }
class VerwAngestellte
   implements Angestellte { ... }
class RegulaererStudent
   implements Student { ... }
class AustauschStudent
   implements Student { ... }
class Gasthoerer
   implements Student { ... }
```


Beispiel: Typhierarchie II

- Die Typen `Person` und `Printable` haben nur `Object` als Supertypen.
- Der Typ `Angestellter` hat `Person` und `Printable` als direkte Supertypen.
- Eine Schnittstellendeklaration erweitert also die Schnittstelle eines oder mehrerer anderer Typen.
- Methodensignaturen aus den Supertypen brauchen nicht nochmals aufgeführt werden (*Signaturvererbung*).

Dynamische Methodenauswahl

Die Auswertung von Ausdrücken vom (statischen) Typ T kann Ergebnisse haben, die von einem Subtyp sind.

Beispiel: Folgender Code-Schnipsel druckt die Liste der Teilnehmer für die Veranstaltung "Graphtheorie":

```
University u = ....;

Course c = u.findCourse("Graphtheorie");
c.printStudents();
```

Hier ist nicht klar, ob die Veranstaltung "Graphtheorie" eine Vorlesung oder ein Seminar ist.

Damit stellt sich die Frage, wie Methodenaufrufe (im Beispiel `printStudents()`) auszuwerten sind.

Begriffsklärung: Dynamische Methodenauswahl

Die auszuführende Methode zu einem Methodenaufruf:

`<Ausdruck>.<methodenName>( <AktParam1>,...);`

wird wie folgt bestimmt:

1. Werte <Ausdruck> aus; Ergebnis ist das *Zielobjekt*.
2. Werte die aktuellen Parameter <AktParam1>, ... aus.
3. Führe die Methode mit Namen <methodenName> des Zielobjekts mit den aktuellen Parametern aus.

Dieses Verfahren nennt man **dynamische Methodenauswahl** oder **dynamisches Binden** (engl. *dynamic method binding*).

Bemerkung

Die Unterstützung von Subtypen und dynamischer Methodenauswahl ist entscheidend für die verbesserte Wiederverwendbarkeit und Erweiterbarkeit, die durch Objektorientierung erreicht wird.

Zusätzlich werden diese Aspekte auch durch Vererbung unterstützt ($\Rightarrow$ nächste Vorlesung).

Beispiel: Erweiterbarkeit I

Wir gehen von einem Programm aus mit der Methode:

```
void printAll(Printable[] df) {
   int i;
   for(i = 0; i < df.length; i++) {
      df[i].print();
   }
}
```

`Printable` ist dabei Supertyp von `Student`, `Angestellte`, `WissAngestellte` und `VerwAngestellte`.

Beispiel: Erweiterbarkeit II

Das Programm soll nun erweitert werden, um auch Professoren/-innen und studentische Hilfskräfte behandeln zu können. Es reicht, zwei Klassen hinzuzufügen:

```
class Professor
   implements Person, Printable  {... }

class StudHilfskraft
   extends Student { ... }
```

Eine Änderung der Methode `printAll` ist **NICHT** nötig!

Weitere Aspekte der Subtypbildung

Zyklenfreiheit

Die Subtyprelation darf keine Zyklen enthalten (sonst wäre sie keine Ordnung).

Folgendes Fragment ist also in Java nicht zulässig und liefert einen Fehler beim Compilieren:

```
interface C extends A { ... }
interface B extends C { ... }
interface A extends B { ... }
```


Subtyprelation bei Feldern I

Jeder Feldtyp mit Komponenten vom Typ S ist ein Subtyp von `Object`:

$$\texttt{S[]} \leq \texttt{Object}$$

D.h. folgende Zuweisung ist zulässig:

```
Object ov = new String[3] ;
```

Ist $S \leq T$, dann ist `S[]` $\leq$ `T[]`.
D.h. folgende Zuweisung ist zulässig:

```
Person[] pv = new Student[3] ;
```

Diese Festlegung der Subtypbeziehung zwischen Feldtypen ist in vielen Fällen praktisch, aber birgt auch Probleme.

Subtyprelation bei Feldern II

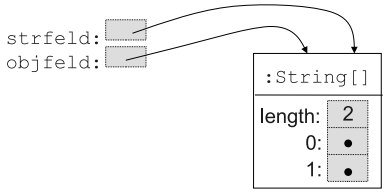
Statische Typsicherheit ist nicht mehr gegeben:

```
String[] strfeld = new String[2];

Object[] objfeld = strfeld;
objfeld[0] = new Object(); // Laufzeitfehler
                           // ArrayStoreException

int strl   =  strfeld[0].length();
```

Speicherzustand nach der zweiten Zeile:



Polymorphie

Ein Typsystem heißt **polymorph**, wenn es Werte bzw. Objekte gibt, die zu mehreren Typen gehören.

Die Form der Polymorphie in Typsystemen mit Subtypen heißt **Subtyp-Polymorphie**.

Subtyp-Polymorphie erlaubt es z.B. *inhomogene* Arrays oder Listen zu verwenden, d.h. mit Elementen unterschiedlichen Typs (s.o.).

Typtest mit `instanceof`

Java bietet den Operator `instanceof` an, um den dynamischen Typen eines Objekts zu testen.

```
class MyUniversity implements University {
  private CourseList cl;
  ...
  int getNumberOfSeminars() {
    int count = 0;
    CourseListIterator ci = cl.iterator();
    while (ci.hasNext()) {
      Course c = ci.next();
      if (c instanceof Seminar) {
        count++;
      }
    }
  }
}
```