

# ***Grundlagen der Programmierung***

Dr. Christian Herzog  
Technische Universität München

Wintersemester 2018/2019

## ***Kapitel 3: Klassen und Objekte***

### ***Überblick über Kapitel 3 der Vorlesung***

- ❖ Objekte
  - Attribute, Operationen
- ❖ Klasse von Objekten (Objekt vs. Klasse)
- ❖ Klassendiagramm und Objektdiagramm in UML
- ❖ Sichtbarkeit von Merkmalen
- ❖ Implementierung von Klassen und Objekten in Java
- ❖ Konstruktoren in Java
- ❖ Die Vererbungs-Beziehung
  - Beispiele von Vererbung
- ❖ Kombination von Aggregation und Vererbung
- ❖ Das Kompositions-Muster

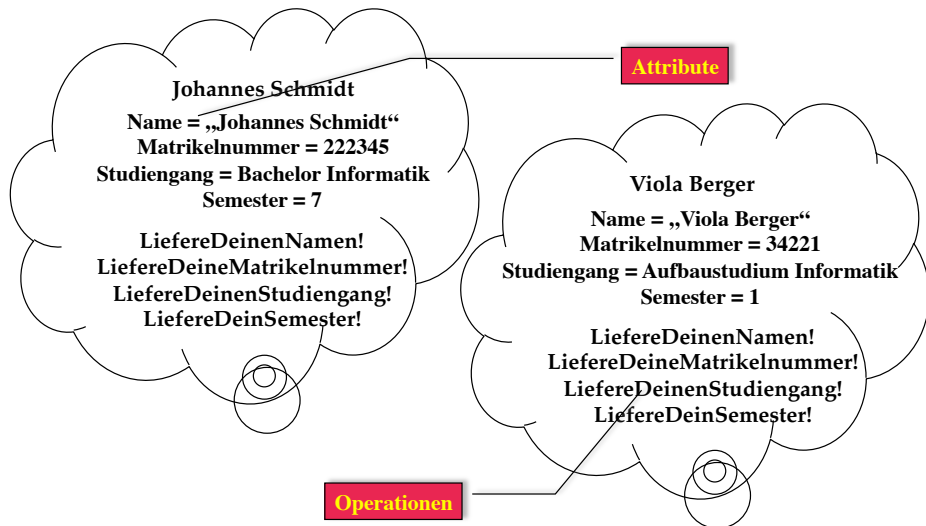
### ***Unser Grundmodell der Modellierung***

- ❖ Ein System besteht aus Subsystemen, die wieder aus Subsystemen bestehen, und diese dann letztendlich aus Gegenständen.
  - Diese Gegenstände bezeichnen wir auch als **Objekte**.
- ❖ Alle zu verarbeitende Informationen in einem System sind auf diese Objekte verteilt.
- ❖ Die Verarbeitung von Information geschieht
  - entweder innerhalb der Objekte
  - oder durch Kommunikation von Nachrichten zwischen zwei Objekten.

### ***Objekt***

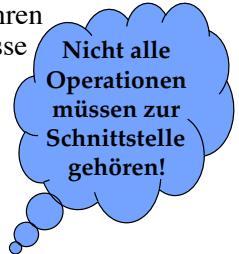
- ❖ Ein **Objekt** ist ein elementares Teilsystem. Es repräsentiert einen beliebigen Gegenstand in einem System.
- ❖ Ein Objekt ist durch seinen **Zustand** und seine **Funktionalität** gegeben.
- ❖ Zustand und Funktionalität setzen sich im Allgemeinen aus Teilzuständen und einzelnen Operationen zusammen.
  - wir nennen die Teilzustände auch **Attribute**
  - wir nennen die einzelnen Operationen auch **Methoden** des Objektes

## Zwei Beispiels-Objekte



## Attribute, Operationen, Merkmale, Schnittstelle

- ❖ Ein Objekt besitzt Attribute und Operationen
  - **Attribut:** Messbare, durch Werte erfassbare Eigenschaft des Objektes.
  - **Operation:** Eine Tätigkeit, die ein Objekt ausführen kann, um Berechnungen durchzuführen, Ereignisse auszulösen sowie Botschaften zu übermitteln.
- ❖ **Merkmale:** Die zu einem Objekt gehörigen Attribute und Operationen.
- ❖ **Schnittstelle:** Die Menge der Operationen eines Objektes, die von anderen Objekten aufgerufen werden können.

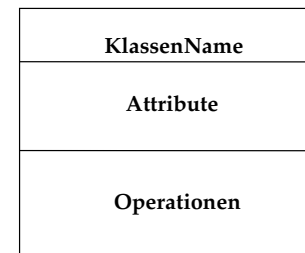


## Klasse und Instanz

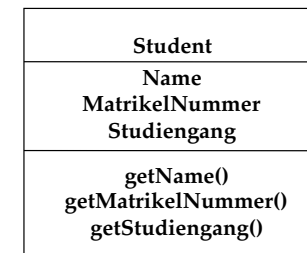
- ❖ Wir können Objekte mit gleichen Merkmalen zusammenfassen bzw. Objekte nach ihren Merkmalen *klassifizieren*:
- ❖ **Klasse:** Die Menge aller Objekte mit denselben Merkmalen, d.h. mit denselben Attributen und denselben Operationen.
- ❖ **Instanz:** Ein Objekt ist eine Instanz einer Klasse K, wenn es Element der Menge aller Objekte der Klasse K ist.
- ❖ Künftig werden wir die Klasse weniger als Menge von Objekten auffassen sondern als *Beschreibung der Merkmale* ihrer Objekte.
  - Dabei wirkt sie wie eine Schablone zur Generierung (*Instanzierung*) von ihr zugeordneten Objekten (Instanzen).

## Graphische Darstellung von Klassen in UML (Klassendiagramm)

Allgemein:

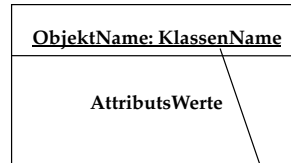


Beispiel:

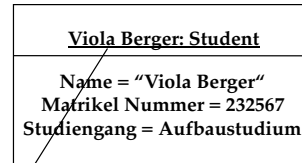


## Graphische Darstellung von Objekten in UML (Objektdiagramm oder Instanzendiagramm)

Allgemein:

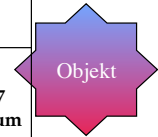
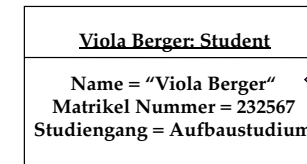
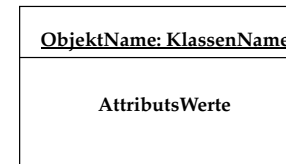
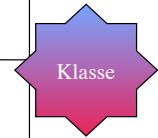
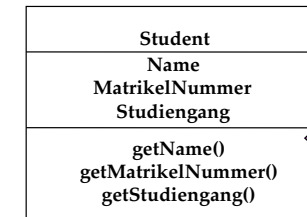
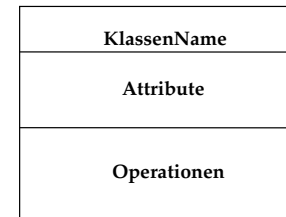


Beispiel:



Die Unterstreichung ist wichtig!

## Graphische Darstellung: Objekt vs. Klasse

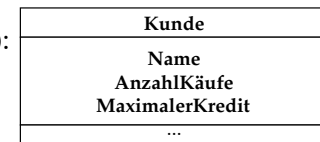


## Attribute

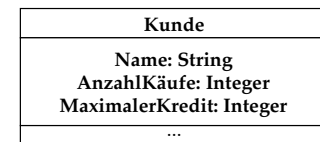
- ❖ Ein **Attribut** einer Klasse wird in der Attributliste der Klasse als
  - *Name* (während der Analyse) oder als
  - *Name: Typ* (während des detaillierten Entwurfs) aufgelistet.
- ❖ Beispiele von Typen sind
  - **String**: Die Menge aller Zeichenketten
  - **Integer**: Die Menge aller ganzen Zahlen
  - **Boolean**: Die Menge der Wahrheitswerte {Wahr, Falsch}
  - **Studiumstyp**: Die Menge der Studiengänge  
{BachelorInf, MasterInf, Aufbaustudium, MasterPhysik}
- ❖ Wird in der Attributliste der Klasse zusätzlich ein Initialisierungswert
  - *Name: Typ = InitialWert*
 angegeben, dann bekommt jedes Objekt dieser Klasse diesen Wert als Anfangswert für dieses Attribut.

## Beispiele für Attributlisten

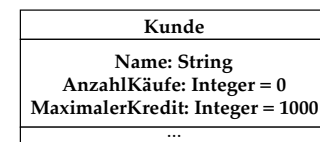
- ❖ Nur Namen (während der Analyse):



- ❖ Namen und Typ (während des detaillierten Entwurfs):



- ❖ Zusätzliche Initialisierungswerte:

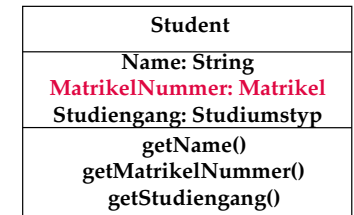


## Attribut als Beziehung zu einer Klasse

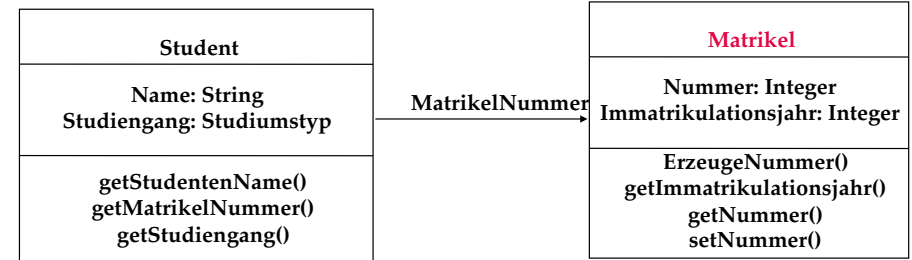
- ❖ Ein Attribut kann auch als Beziehung zu einer Klasse gezeichnet werden, insbesondere wenn die Beziehung zwischen beiden Klassen klargemacht werden soll.
  - Der Name des Attributes ist dann der Name der Beziehung.

## Beispiel: Attribut als Beziehung

Bisher: Attribut mit Klassenname als Typ:



Attribut als Beziehung zwischen Klassen:



## Operation

- ❖ Die **Operationen** arbeiten auf Attributen der Klasse und anderen Objekten, mit denen die Klasse eine Beziehung hat.
- ❖ Notation von Operationen in UML:
  - *Name ()* oder
  - *Name (Parameterliste)* oder
  - *Name (Parameterliste): Ergebnistyp*
- ❖ Die Menge von Operationen, die eine Klasse oder eine Menge von Klassen (Subsystem) zur Verfügung stellt, bezeichnen wir als **Schnittstelle** der Klasse (des Subsystems).
- ❖ Welche der Operationen von einer Klasse zur Verfügung gestellt werden, wird durch die **Sichtbarkeit** der Operationen geregelt.

## Sichtbarkeit von Operationen

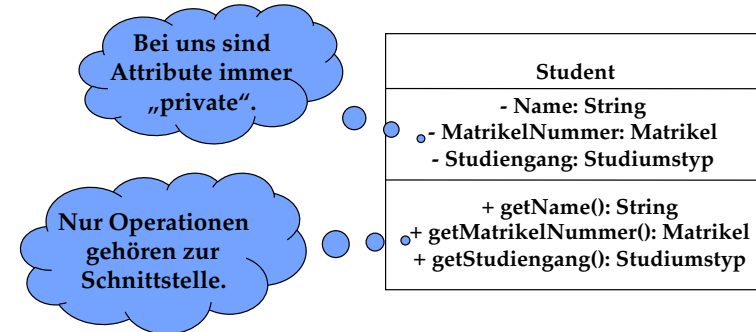
- ❖ Die **Sichtbarkeit** einer Operation regelt, welche Objekte diese Operation verwenden dürfen.
- ❖ Die Sichtbarkeit ist zwischen Klassen definiert, d.h. alle Objekte einer Klasse  $K_1$  haben auf eine Operation eines Objekts einer Klasse  $K_2$  dieselben Zugriffsrechte.
- ❖ Wir unterscheiden zunächst zwei Sichtbarkeiten für Operationen (weitere Sichtbarkeiten werden wir später bei der objektorientierten Programmierung einführen):
  - **public**: jedes Objekt jeder Klasse hat unbeschränkten Zugriff;
  - **private**: nur die Objekte derselben Klasse dürfen die Operation verwenden.
- ❖ In UML wird „public“ durch ein vorangestelltes „+“-Zeichen gekennzeichnet, „private“ durch ein „-“-Zeichen.

## Sichtbarkeit von Attributen

- ❖ In UML (und vielen Programmiersprachen) können auch Attribute dieselben Sichtbarkeiten wie Operationen haben.
- ❖ Auf ein Attribut mit der Sichtbarkeit „public“ kann also von Objekten anderer Klassen aus direkt zugegriffen werden.
- ❖ **Konvention in Grundlagen der Programmierung:**
  - Attribute sind (vorerst) immer „private“.
  - Attribute gehören also nicht zur Schnittstelle.
  - Wenn Attribute von Objekten anderer Klassen aus gelesen oder verändert werden sollen, so müssen dazu „public“-Operationen `setAttribut()` bzw. `getAttribut()` zur Verfügung gestellt werden.

## Sichtbarkeit von Merkmalen in UML

- ❖ In UML wird „public“ durch ein vorangestelltes „+“-Zeichen gekennzeichnet, „private“ durch ein „-“-Zeichen.



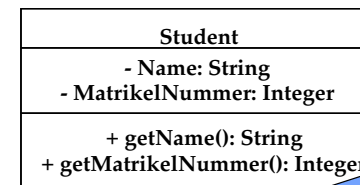
- ❖ Abkürzende Schreibweise in Grundlagen der Programmierung:
  - Wir lassen das „-“ bei Attributen oft weg.
  - Wir lassen das „+“ bei Operationen oft weg (nur das „-“ wird explizit notiert).

## Einschub: Zwei Prinzipien der Didaktik

- ❖ Erstes Prinzip:
  - Verwende keine Konzepte, die du nicht gründlich eingeführt hast!
- ❖ Zweites Prinzip:
  - Programmieren lernt man nur durch Programmieren!
  - Also: so früh wie möglich programmieren!
- ❖ **Problem:**
  - Bereits das kleinste Java-Programm verwendet komplizierte Konzepte.
- ❖ **Kompromiss:**
  - Wir fangen früh an, in Java zu programmieren.
  - Wir betrachten jetzt einiges als „so ist es halt“ und freuen uns auf das Aha-Erlebnis, wenn wir später die dahinter stehenden Konzepte kennen lernen.

Augen zu und durch!

## Umsetzung des Klassendiagramms nach Java



„//“ leitet in Java Kommentarzeilen ein

- ❖ Weitere GrProg - Konventionen:
  - Klassennamen werden in Java groß geschrieben,
  - Merkmalsnamen werden klein geschrieben.

```
public class Student {  
    // Attribute:  
    private String name;  
    private int matrikelNummer;  
  
    // Operationen (Methoden):  
    public String getName() {  
    }  
    public int getMatrikelNummer() {  
    }  
}
```

In Java haben auch Klassen eine Sichtbarkeit. Wir verwenden zunächst nur „public“.

## Implementierung der Operationen in Java

Student
- Name: String - MatrikelNummer: Integer
+ getName(): String + getMatrikelNummer(): Integer

Hier werden jeweils nur die Werte der entsprechenden Attribute als Ergebnis ausgeliefert.

```
public class Student {
    // Konstruktor:
    public Student() {}

    // Attribute:
    private String name;
    private int matrikelNummer;

    // Operationen (Methoden):
    public String getName() {
        return name;
    }
    public int getMatrikelNummer() {
        return matrikelNummer;
    }
}
```

Der Konstruktor ist eine spezielle Operation, die automatisch bei der Instanziierung ausgeführt wird.

## Instanziierung von Objekten und Konstruktor

```
public class Student {
    // Konstruktor:
    public Student() {}

    // Attribute:
    private String name;
    private int matrikelNummer;

    // Operationen (Methoden):
    public String getName() {
        return name;
    }
    public int getMatrikelNummer() {
        return matrikelNummer;
    }
}
```

- ❖ Der Konstruktor ist eine spezielle Methode.
- ❖ Sein Name ist identisch mit dem Klassennamen.
- ❖ Der Konstruktor wird ausgeführt, wenn ein Objekt der Klasse instanziiert wird. (Objekte müssen explizit instanziiert werden.)
- ❖ Er wird (u.a) dazu benutzt, um die Attribute zu initialisieren.

❖ Instanziierung eines Objekts der Klasse Student:

```
Student s = new Student();
```

Mit „new“ wird ein Objekt instanziiert und der Konstruktor aufgerufen.

## Vorbesetzung der Attribute durch Angabe von Initialisierungswerten (Variante 1)

```
public class Student {
    // Konstruktor:
    public Student() {}

    // Attribute:
    private String name = "Viola";
    private int matrikelNummer = 1234567;

    // Operationen (Methoden):
    public String getName() {
        return name;
    }
    public int getMatrikelNummer() {
        return matrikelNummer;
    }
}
```

- ❖ Initialisierung der Attribute in der Klassendefinition – Default-Werte

Student
- Name: String = „Viola“ - MatrikelNummer: Integer = 1234567
+ getName(): String + getMatrikelNummer(): Integer

## Vorbesetzung der Attribute im Konstruktor (Variante 2)

```
public class Student {
    // Konstruktor:
    public Student() {
        name = "Viola";
        matrikelNummer = 1234567;
    }

    // Attribute:
    private String name;
    private int matrikelNummer;

    // Operationen (Methoden):
    public String getName() {
        return name;
    }
    public int getMatrikelNummer() {
        return matrikelNummer;
    }
}
```

Student
- Name: String = „Viola“ - MatrikelNummer: Integer = 1234567
+ getName(): String + getMatrikelNummer(): Integer

## Keine Initialisierungswerte sondern Vorbesetzung durch Parameter des Konstruktors (Variante 3)

```
public class Student {
// Konstruktor:
public Student(String n, int m) {
    name = n;
    matrikelNummer = m;
}

// Attribute:
private String name;
private int matrikelNummer;

// Operationen (Methoden):
public String getName() {
    return name;
}
public int getMatrikelNummer() {
    return matrikelNummer;
}
}
```

Student
- Name: String - MatrikelNummer: Integer
+ getName(): String + getMatrikelNummer(): Integer

- ❖ Instanziierung eines Objekts bei einem parametrisierten Konstruktor:

```
Student s =
    new Student("Viola",1234);
```

## System und Umgebung in Java

```
public class Student {
    public Student(String n, int m) {
        name = n; matrikelNummer = m; }
    private String name;
    private int matrikelNummer;
    public String getName() {
        return name; }
    public int getMatrikelNummer() {
        return matrikelNummer; }
}
```

System

Schnittstelle

```
public class Umgebung {
    public static void main (String[] args) {
        Student s = new Student ("Viola", 1234);
        System.out.println(s.getName ());
        System.out.println(s.getMatrikelNummer ());
    }
}
```

Umgebung

## Editieren, Compilieren, Programmablauf

- ❖ **Editieren:**
  - Klassendefinition Student in Datei **Student.java**
  - Klassendefinition Umgebung in Datei **Umgebung.java**
- ❖ **Kompilieren:**
  - **javac Student.java** liefert Datei **Student.class**
  - **javac Umgebung.java** liefert Datei **Umgebung.class**
  - (Es hätte auch nur der Befehl **javac Umgebung.java** genügt. Das Java-System ist so „intelligent“, alle zusätzlich benötigten Klassen mit zu kompilieren.)
- ❖ **Ablauf des Programms (Exekutieren):**
  - **java Umgebung** (ohne Extension **.class**)
  - Das Java-System führt die **main**-Methode der Klasse Umgebung aus

## System und Umgebung mit dem Java-System

```
public class Student {
    public Student(String n, int m) {
        name = n; matrikelNummer = m; }
    private String name;
    private int matrikelNummer;
    public String getName() {
        return name; }
    public int getMatrikelNummer() {
        return matrikelNummer; }
}
```

Java-System

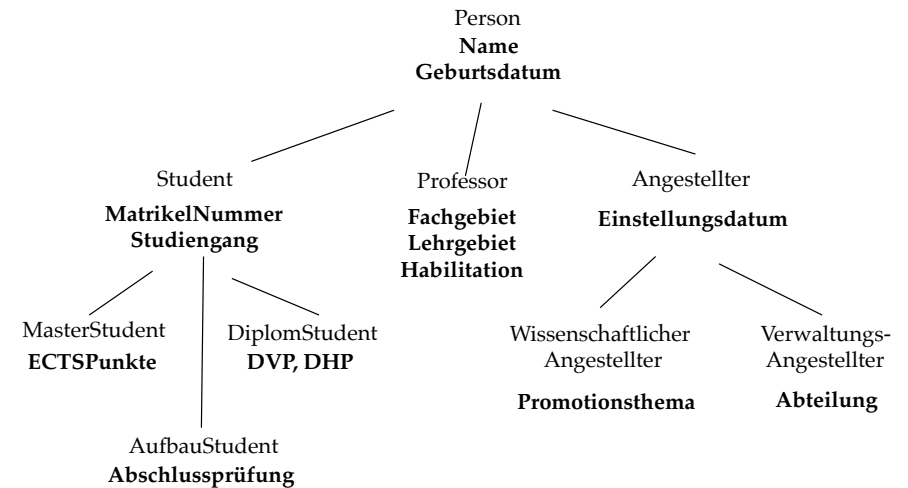
```
public class Umgebung {
    public static void main (String[] args) {
        Student s = new Student ("Viola", 1234);
        System.out.println(s.getName ());
        System.out.println(s.getMatrikelNummer ());
    }
}
```

Schnittstelle zum Java-System

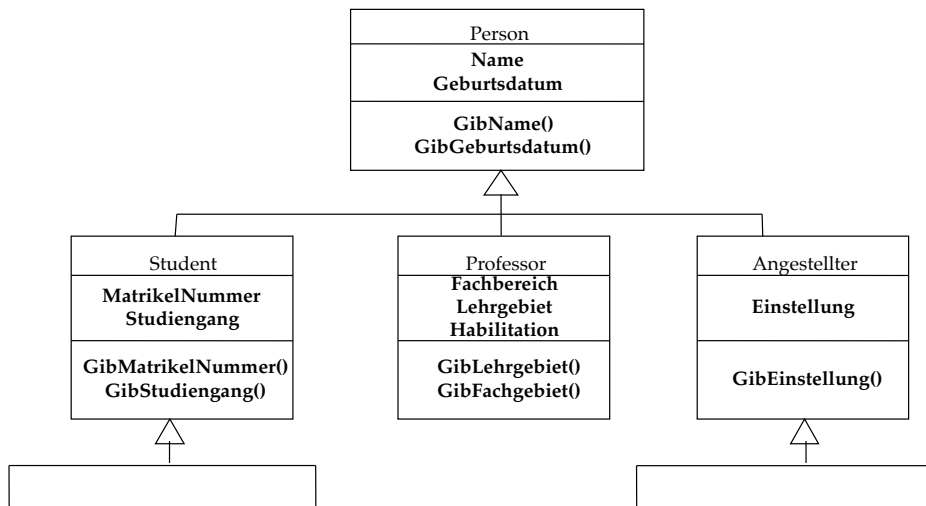
## Zwei wichtige Prinzipien der Modellierung

- ❖ **Informationskapselung** (*information hiding*): Objekte können auf andere Objekte nur über deren Schnittstelle zugreifen.
  - Ein Objekt kann also nicht direkt auf die Attribute eines anderen Objektes zugreifen.
- ❖ **Klassifikation**: Komponenten können nach ihren Merkmalen klassifiziert werden.
  - Beispiel: Zwei Objekte Obj1 und Obj2 können zusammengefasst werden, wenn sie dieselbe Operation *print()* bereit stellen.
- ❖ **Klassifikationen kann man benutzen, um Mengen von Objekten hierarchisch zu strukturieren.**
  - Beispiel: Die Personengruppen an einer Universität.

## Klassifikation von Personengruppen an einer Universität



## Klassifikation von Personengruppen an einer Universität



## Die Vererbungsbeziehung

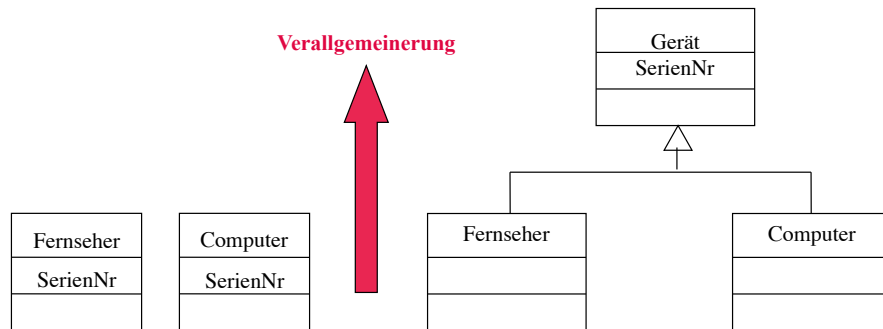
- ❖ Zwei Klassen stehen in einer **Vererbungsbeziehung** (*inheritance relationship*) zueinander, falls die eine Klasse, auch **Unterklasse** (Subklasse) genannt, alle Merkmale der anderen Klasse, auch **Oberklasse** genannt, besitzt, und eventuell darüber hinaus noch zusätzliche Merkmale.
- ❖ Es gilt somit für die Mengen  $A_U, A_O$  der Attribute und die Mengen  $O_U, O_O$  der Operationen der Unterklasse  $U$  und Oberklasse  $O$ :
  - $A_O \subseteq A_U$  und  $O_O \subseteq O_U$
- ❖ Eine Unterklasse wird also durch Hinzufügen von Merkmalen **spezialisiert**.
- ❖ Umgekehrt verallgemeinert die Oberklasse die Unterklasse dadurch, dass sie spezialisierende Eigenschaften weglässt. Wir nennen das auch **Verallgemeinerungsbeziehung** (*generalization relationship*).



## Vererbungsbeispiel

Eine Firma stellt sowohl Fernsehgeräte als auch Computer her. Beide haben Seriennummern.

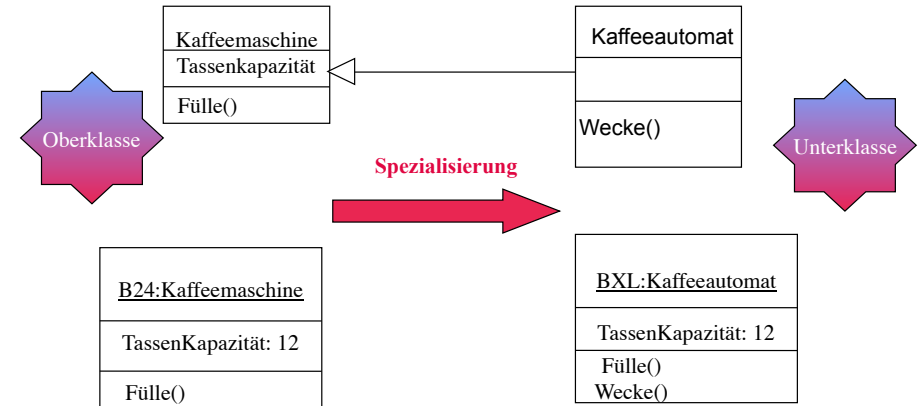
Die Seriennummer wird in einer Oberklasse Gerät angeführt und von dort vererbt.



## Noch ein Vererbungsbeispiel

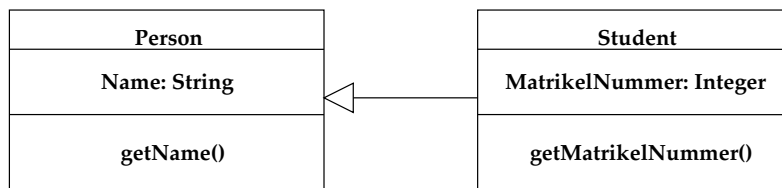
❖ Eine Kaffeemaschine kann eine Anzahl von Tassen füllen.

❖ Das Luxusmodell hat noch eine Weckfunktion



## Java unterstützt Vererbung

❖ UML: Klasse Student als Unterklasse von Person:



❖ Java:

```

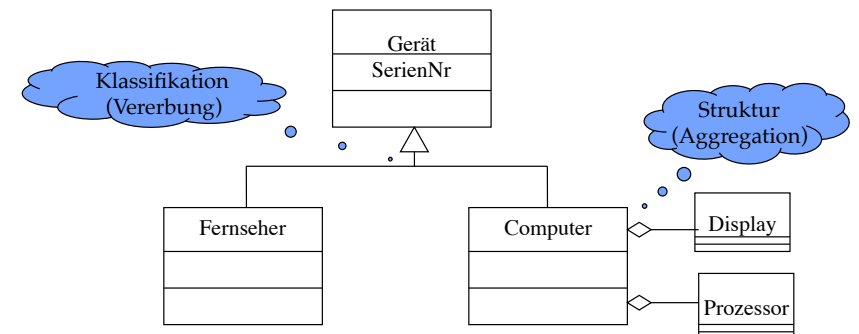
- public class Person { ... }
- public class Student extends Person { ... }
  
```

❖ Näheres dazu im Kapitel „Objektorientierte Programmierung“ später im Semester.

## Kombination von Vererbung und Aggregation

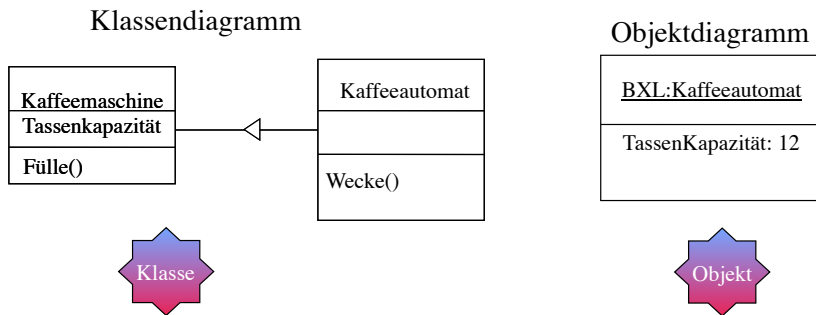
❖ In der Modellierung tritt oft der Fall auf, dass wir Gegenstände klassifizieren müssen, aber gleichzeitig auch deren Struktur erkenntlich machen wollen.

❖ Wir benutzen dann sowohl Vererbung als auch Aggregation:

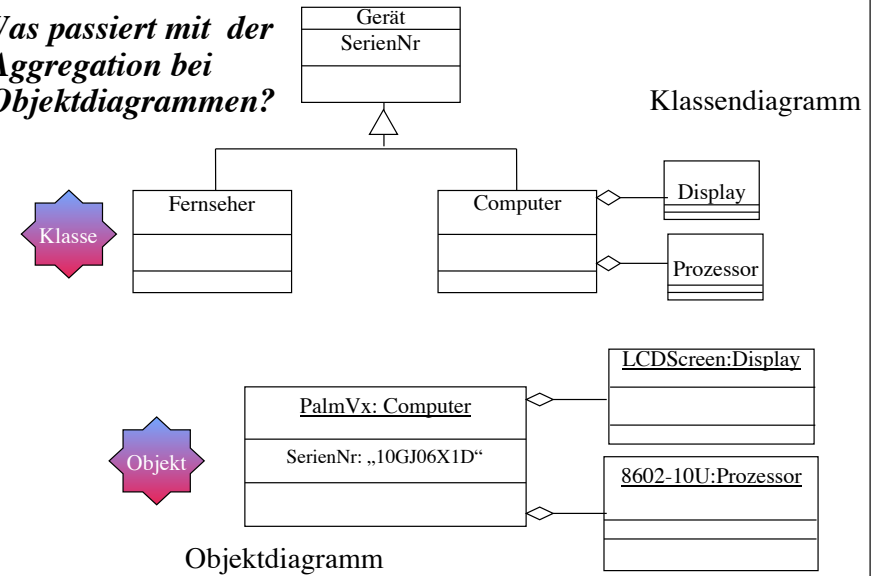


❖ *Heuristik:* In jedem guten Modell gibt es eine Kombination von Klassen, die in Vererbungs- und Aggregationsbeziehungen stehen.

## Was passiert mit der Vererbung bei Objektdiagrammen?



## Was passiert mit der Aggregation bei Objektdiagrammen?

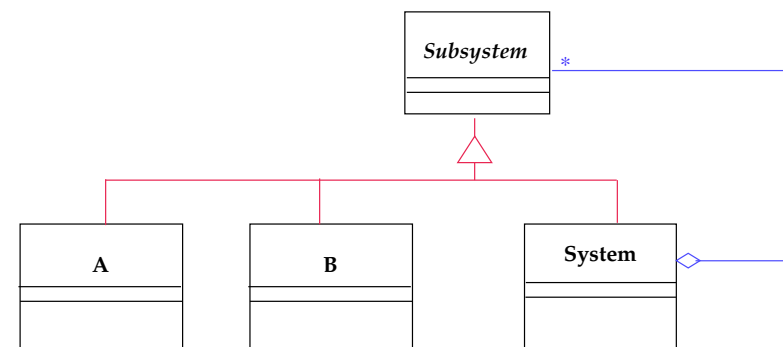


## Modellierung unserer Systemdefinition

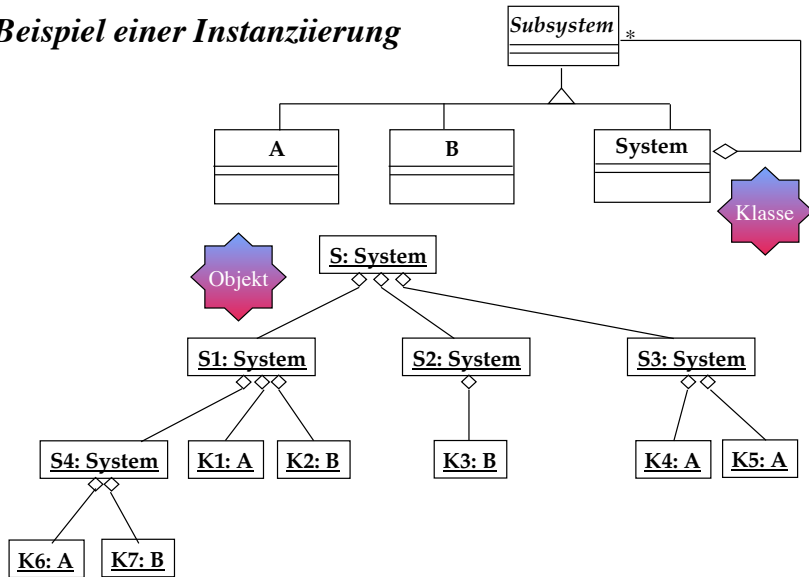
- ❖ **Definition eines Systems (Wiederholung):** Unter einem System versteht man eine Menge von Komponenten (Gegenständen), die in einem gegebenen Bezugssystem in einem Zusammenhang stehen, und die Beziehungen zwischen diesen Komponenten.
- ❖ Die Komponenten eines Systems können selbst wieder (Sub-)Systeme sein.
- ❖ **Heuristik:** In jedem guten Modell gibt es eine Kombination von Klassen, die in Vererbungs- und Aggregationsbeziehungen stehen.

## Modellierung des Systembegriffs

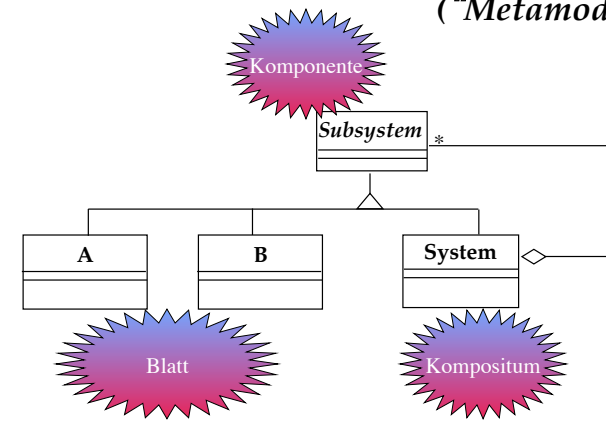
- ❖ Ein System besteht aus beliebig vielen Subsystemen => **Aggregation**.
- ❖ Die Subsysteme sind entweder selbst wieder Systeme oder Komponenten A oder B => **Vererbung**.



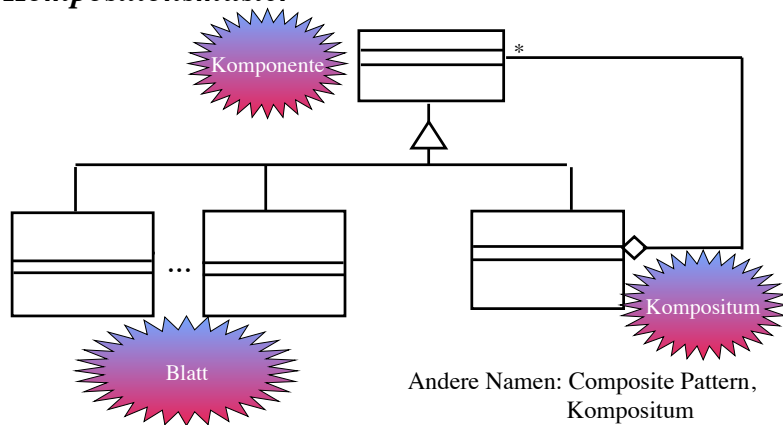
### Beispiel einer Instanziierung



### Betrachtungen zu unserer Modellierung ("Metamodellierung")



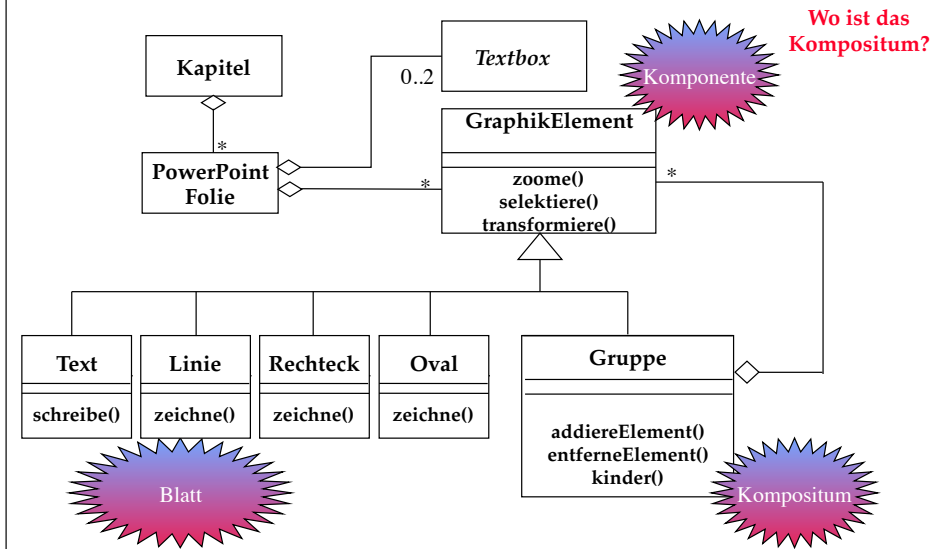
### Kompositionsmuster



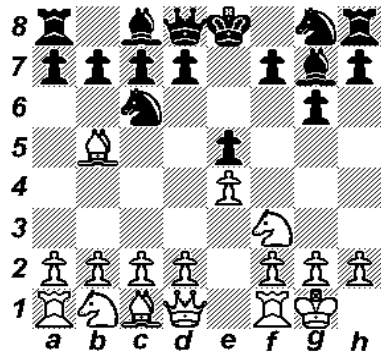
Andere Namen: Composite Pattern, Kompositum

Das Kompositionsmuster ist immer anwendbar, wenn wir Strukturen beschreiben, die eine dynamische Höhe und dynamische Breite haben.

### Beispiel: Modellierung eines Foliensatzes



## Muster im Schach



- ❖ Spanische Eröffnung
- ❖ Schwarz: Läufer Fianchetto
- ❖ Weiss: Kurze Rochade

## Nützlichkeit von Entwurfsmustern

- ❖ Entwurfsmuster sind wieder verwendbares Wissen bei der Entwicklung von Informatik-Systemen, vor allem bei der Analyse und beim Systementwurf.
- ❖ Entwurfsmuster lassen sich zu einem Gesamtentwurf kombinieren, der dann als Grundlage für ein Informatik-System dienen kann.
- ❖ Beispiele für weitere Entwurfsmuster:
  - **Beobachter-Muster** (Observer Pattern): Trennt die Veröffentlichung von Information und das Lesen dieser Information (Publish-Subscribe)
  - **Adapter-Muster** (auch „Wrapper“ genannt): Zum Aufruf von alten, nicht mehr änderbaren Schnittstellen
  - **Brückenmuster** (Bridge Pattern): Zur dynamischen Anpassung an unterschiedliche Implementierungen einer Schnittstelle

## Zusammenfassung

- ❖ Objekt: Attribute, Operationen, Merkmale
- ❖ Objekt als Instanz einer Klasse
- ❖ Instanzendiagramm, Klassendiagramm
- ❖ Klassen und Instanzen in Java
  - die main-Methode
  - Konstruktoren
- ❖ Wichtige Beziehungen in der Modellierung:
  - Aggregation
  - Vererbung
  - Man nennt sie deshalb auch kanonische Assoziationen
- ❖ Kompositionsmuster