LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Skript zur Vorlesung:

Einführung in die Programmierung

WiSe 2009 / 2010

Prof. Dr. Christian Böhm
Annahita Oswald
Bianca Wackersreuther

Ludwig-Maximilians-Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Gliederung

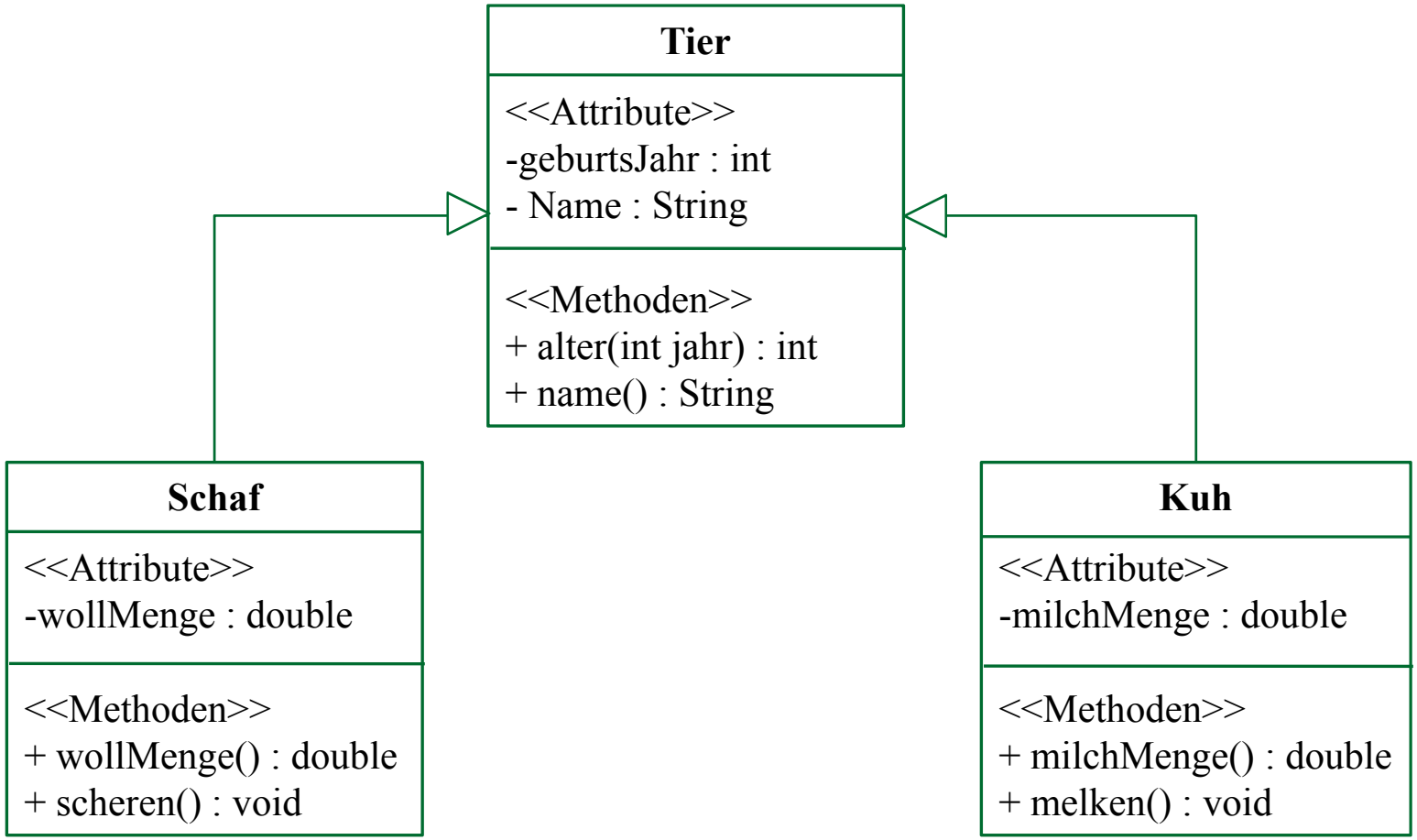
Vererbung

- Vererbung ist die Umsetzung von “is-a”-Beziehungen.
- Alle Elemente (Attribute und Methoden) der Vaterklasse sollen auf die abgeleitete Klasse vererbt werden.
- Zusätzlich kann die abgeleitete Klasse neue Elemente (Attribute und Methoden) definieren.
- Vererbung ist ein wichtiges Mittel zur Wiederverwendung von Programmteilen.

Beispiel

- Wir wollen das Tierreich (bestehend aus Schafen und Kühen) auf einem Bauernhof modellieren.
- Dazu stellen wir fest, dass beide Tierarten gemeinsame Eigenschaften haben, z.B. einen Namen, ein Geburtsjahr, etc.
- Zusätzlich haben beide Tierarten unterschiedliche Eigenschaften, bei den Kühen interessieren wir uns z.B. für die Milchmenge, die sie letztes Jahr abgegeben haben, bei den Schafen interessiert uns die Menge der Wolle, die wir scheren konnten, etc.
- Dies kann man mit Vererbung entsprechend modellieren (siehe nächste Folie).

Beispiel



Was wird hier verwendet?

Ableiten einer Klasse

- In Java wird nur die einfache Vererbung (eine Klasse wird von genau einer Vaterklasse abgeleitet) direkt unterstützt.
- Mehrfachvererbung (eine Klasse wird von mehr als einer Vaterklasse abgeleitet) muss in Java mit Hilfe von *Interfaces* umgesetzt werden (siehe später).
- In Java zeigt des Schlüsselwort **`extends`** Vererbung an. Beispiel:

```
public class Kuh extends Tier
{
  ...
}
```

- Objekte der Klasse `Kuh` erben damit alle Attribute und Methoden der Klasse `Tier`.

Die Klasse `Object`

- Enthält eine Klasse keine **`extends`**-Klausel, so besitzt sie die implizite Vaterklasse `Object` (im Paket `java.lang`).
- Also: Jede Klasse, die keine **`extends`**-Klausel enthält, wird direkt von `Object` abgeleitet.
- Jede explizit abgeleitete Klasse ist am oberen Ende ihrer Vererbungshierarchie von einer Klasse ohne explizite Vaterklasse abgeleitet und ist damit ebenfalls von `Object` abgeleitet.
- Damit ist `Object` die Vaterklasse aller anderen Klassen.

Die Klasse `Object`

- Die Klasse `Object` definiert einige elementare Methoden, die für alle Arten von Objekten nützlich sind, u.a.:
  - **`boolean`** `equals(Object obj)`

    testet die Gleichheit zweier Objekte, d.h. ob zwei Objekte den gleichen Zustand haben.
  - `Object clone()`

    kopiert ein Objekt, d.h. legt eine neues Objekt an, das eine genaue Kopie des ursprünglichen Objekts ist.
  - `String toString()`

    erzeugt eine String-Repräsentation des Objekts.
  - etc.
- Damit diese Methoden in abgeleiteten Klassen sinnvoll funktionieren, müssen sie bei Bedarf überschrieben werden.

Vererbung von Attributen und Methoden

- Neben ererbten Attributen und Methoden dürfen neue Attribute und Methoden in der abgeleiteten Klasse definiert werden.
- Es dürfen aber auch Attribute und Methoden, die von der Vaterklasse geerbt wurden, neu definiert werden.
- Bei Attributen tritt dabei der Effekt des *Versteckens* auf: Das Attribut der Vaterklasse ist in der abgeleiteten Klasse nicht mehr sichtbar.
- Bei Methoden tritt zusätzlich der Effekt des *Überschreibens* (auch: *Überlagerns*) auf: Die Methode modelliert in der abgeleiteten Klasse i.d.R. ein anderes Verhalten als in der Vaterklasse.

Vererbung von Attributen und Methoden

- Ist ein Element (Attribut oder Methode) `x` der Vaterklasse in der abgeleiteten Klasse neu definiert, wird also immer, wenn `x` in der abgeleiteten Klasse aufgerufen wird, das neue Element `x` der abgeleiteten Klasse angesprochen.
- Will man auf das Element `x` der Vaterklasse zugreifen, kann dies mit dem expliziten Hinweis **`super.`**`x` erreicht werden.
- Ein kaskadierender Aufruf von Vaterklassen-Elementen (z.B. **`super.super.`**`x`) ist nicht erlaubt!
- **`super`** ist eine Art Verweis auf die Vaterklasse (Vorsicht: *Kein* Zeiger auf ein ensprechendes Objekt).

Die Sichtbarkeitsklasse `protected`

- Zur Spezifikation der Sichtbarkeit von Attributen und Methoden einer Klasse hatten wir bisher kennengelernt:
  - **`public`**: Das Element ist in allen Klassen sichtbar.
  - **`private`**: Das Element ist nur in der aktuellen Klasse sichtbar, also auch *nicht* in abgeleiteten Klassen!
- Zusätzlich gibt es das Schlüsselwort **`protected`**.
- Elemente des Typs **`protected`** sind in der Klasse selbst und in den Methoden abgeleiteter Klassen sichtbar.

Verstecken von Attributen

- Das Verstecken von Attributen ist eine gefährliche Fehlerquelle und sollte daher grundsätzlich vermieden werden.
- Meist geschieht das Verstecken von Attributen aus Unwissenheit über die Attribute der Vaterklasse.
- Problematisch ist, wenn Methoden aus der Vaterklasse vererbt werden, die auf ein verstecktes Attribut zugreifen, deren Name aber durch den Verstecken-Effekt irreführend wird, weil es ein gleich benanntes neues Attribut in der abgeleiteten Klasse gibt.

Verstecken von Attributen: Beispiel

- Die Klasse `Tier` definiert das Attribut `name` in dem der Name des Tieres gespeichert ist.
- Zudem gibt es eine Methode `public String getName()`, die den Wert des Attributs `name` zurrückgibt.
- In der abgeleiteten Klasse `Kuh` gibt es nun ebenfalls ein Attribut `name`, in dem der Name des Besitzers gespeichert werden soll. Die Methode `getName()` aus der Vaterklasse wird nicht überschrieben.
- Wenn nun ein Objekt der Klasse `Kuh` die Methode `getName()` aufruft, wird der Name der Kuh ausgegeben und nicht der Name des Besitzers.

Überschreiben von Methoden

- Das Überschreiben von Methoden ist dagegen ein gewünschter Effekt, denn eine abgeleitete Klasse zeichnet sich gerade durch ein unterschiedliches Verhalten gegenüber der Vaterklasse aus.
- *Late Binding* bei Methodenaufrufen: Erst zur Laufzeit wird entschieden, welcher Methodenrumpf nun ausgeführt wird, d.h. von welcher Klasse das aufrufende Objekt nun ist.
- Dieses Verhalten bezeichnet man auch als *dynamisches Binden*.
- Beispiel: Eine Variable vom Typ `Tier` kann Objekte vom Typ `Tier`, Objekte vom Typ `Kuh` oder Objekte vom Typ `Schaf` enthalten.
- Es kann erst zur Laufzeit entschieden werden, welchen Typ die Variable aktuell hat.
- Überschreiben von Methoden und dynamisches Binden ist ein wichtiges Merkmal von Polymorphismus.

Warum keine Mehrfachvererbung?

- Wie bereits erwähnt, ist Mehrfachvererbung in Java nicht erlaubt.
- Der Grund hierfür ist, dass bei Mehrfachvererbung verschiedene Probleme auftauchen können.
- Ein solches Problem kann im Zusammenhang mit Methoden entstehen, die aus beiden Vaterklassen vererbt werden und nicht in der abgeleiteten Klasse überschrieben werden.
- In diesem Fall ist unklar, welche Methode ausgeführt werden soll, wenn eine dieser Methoden in der abgeleiteten Klasse aufgerufen wird.

Warum keine Mehrfachvererbung?

- Beispiel: Klasse `AmphibienFahrzeug` wird von den beiden Klassen `LandFahrzeug` und `WasserFahrzeug` abgeleitet.
- Die Methode `getPS()`, die die Leistung des Fahrzeugs zurückgibt, könnte bereits in den beiden Vaterklassen implementiert sein und nicht mehr in der Klasse `AmphibienFahrzeug` überschrieben werden.
- Problem:

```
AmphibienFahrzeug a = new AmphibienFahrzeug();
int ps = a.getPS();
```

- **Welche Methode `getPS()` wird ausgeführt?**

Konstruktoren und Vererbung

- Grundsätzlich gilt: Konstruktoren werden *nicht* vererbt!
- Dies ist auch sinnvoll, schließlich kann ein Konstruktor der Klasse `Tier` keine Objekte der spezielleren Klasse `Kuh` erzeugen, sondern eben nur Objekte der generelleren Klasse `Tier`.
- Es müssen also (wenn dies gewünscht ist), in jeder abgeleiteten Klasse eigene explizite Konstruktoren definiert werden.

Konstruktoren und Vererbung

- Wenn ein Objekt mittels des **`new`**-Operators und eines entspr. Konstruktors erzeugt wird, wird grundsätzlich auch (explizit oder implizit) der Konstruktor der Vaterklasse aufgerufen.
- Explizit kann man dies durch Aufruf von **`super`**`(<Parameterliste>)` als *ersten* Befehl in einem expliziten Konstruktor erreichen. `<Parameterliste>` kann leer sein (Default-Konstruktor) oder muss zur Signatur eines expliziten Konstruktors der Vaterklasse passen.
- Steht in einem expliziten Konstruktor der abgeleiteten Klasse kein **`super`**-Aufruf an erster Stelle, wird implizit der Default-Konstruktor der Vaterklasse **`super`**`()` aufgerufen.
- **Achtung**: Es ist nicht erlaubt, den Default-Konstruktor aufzurufen, obwohl ein expliziter Konstruktor in der Vaterklasse vorhanden ist und der Default-Konstruktor nicht existiert.

Konstruktoren und Vererbung: Beispiel

1. Fall:
In `Tier` ist *kein* expliziter Konstruktor definiert.

```
public class Kuh extends Tier
{
  public Kuh(double bisherigeMilchMenge) {
    this.milchMenge = bisherigeMilchMenge;  // (*)
  }

}
```

Beim Aufruf des Konstruktors, z.B.

```
Kuh erni = new Kuh(3.5);
```

wird, bevor Zeile (*) ausgeführt wird, zunächst der Konstruktor `Tier()` implizit aufgerufen.

Konstruktoren und Vererbung: Beispiel

2. Fall:
In `Tier` ist *ein* expliziter Konstruktor

```
Tier(String name, int geburtsJahr)
```

definiert.

```
public class Kuh extends Tier
{
  public Kuh(String name, int geburtsjahr, double  bisherigeMilchMenge) {
    super(name, geburtsjahr);
    this.milchMenge = bisherigeMilchMenge;
  }
}
```

Die Lösung von Fall 1 geht hier nicht, da der (implizit vor Zeile (*) aufgerufene) Default-Konstruktor nicht existiert.

Konstruktoren und Vererbung

- Offenbar dient der Aufruf des Vaterklassen-Konstruktors dazu, die vererbten Attribute in der abgeleiteten Klasse zu initialisieren.
- Natürlich kann dies auch in der abgeleiteten Klasse explizit gemacht werden (ist aber meist wenig sinnvoll).
- Konstruktoren werden nicht vererbt, müssen aber (implizit oder explizit) in abgeleiteten Klassen verwendet werden (können).

Abstrakte Methoden

- *Abstrakte* Methoden enthalten im Gegensatz zu konkreten Methoden nur die Spezifikation der Signatur.
- Abstrakte Methoden enthalten also keinen Methodenrumpf, der die Implementierung der Methode vereinbart.
- Abstrakte Methoden werden mit dem Schlüsselwort **`abstract`** versehen und anstelle der Blockklammern für den Methodenrumpf mit einem simplen Semikolon beendet.
- Beispiel:

```
public abstract <Typ> abstrakteMethode(<Parameterliste>);
```

Abstrakte Klassen

- Abstrakte Methoden können nicht aufgerufen werden, sondern definieren eine Schnittstelle: Erst durch Überschreiben in einer abgeleiteten Klasse und (dortige) Implementierung des Methodenrumpfes wird die Methode konkret und kann aufgerufen werden.
- Klassen, die mindestens eine abstrakte Methode haben, sind selbst abstrakt und müssen ebenfalls mit dem Schlüsselwort **`abstract`** gekennzeichnet werden.
- Eine von einer abstrakten Vaterklasse abgeleiteten Klassen wird konkret, wenn alle abstrakten Methoden der Vaterklasse implementiert sind. Die Konkretisierung kann auch über mehrere Vererbungsstufen erfolgen.
- Es können keine Objekte (Instanzen) von abstrakten Klassen erzeugt werden!

Ein Beispiel für Polymorphismus

- Im folgenden sehen wir uns ein Programm an, das die Mitarbeiter der LMU verwaltet, insbesondere deren brutto Monatsgehalt berechnet.
- Dazu wird zunächst die abstrakte Klasse `Mitarbeiter` definiert, die alle grundlegenden Eigenschaften eines Mitarbeiters modelliert. (Auf der VL-Website verfügbar)
- In abgeleiteten Klassen werden dann die einzelnen Mitarbeitertypen `Arbeiter`, `Angestellter` und `Beamter` abgebildet und konkret implementiert. (Auf der VL-Website verfügbar)
- Die Klasse `Gehaltsberechnung` verwendet diese Klassen polymorph. (Auf der VL-Website verfügbar)

Ein Beispiel für Polymorphismus

```
public abstract class Mitarbeiter
{
    private int persNr;
    private String name;
    private int dienstAlter;


    public Mitarbeiter(int persNr, String name)
    {
        this.persNr = persNr;
        this.name = name;
        this.dienstAlter = 0;
    }

    public abstract double monatsBrutto();
}
```

Ein Beispiel für Polymorphismus

```
public class Arbeiter extends Mitarbeiter
{
    private double stundenLohn;
    private double anzahlStunden;
    private double ueberstundenZuschlag;
    private double anzahlUeberstunden;

    public Arbeiter(int persNr, String name,
                    double sL, double aS, double uZ, double aU)
    {
        super(persNr, name);
        this.stundenLohn = sL;
        this.anzahlStunden = aS;
        this.ueberstundenZuschlag = uZ;
        this.anzahlUeberstunden = aU;
    }

    public double monatsBrutto()
    {
        return   stundenLohn * anzahlStunden +
                 (stundenLohn + ueberstundenZuschlag)
                                    * anzahlUeberstunden;
    }
}
```

Ein Beispiel für Polymorphismus

```
public class Angestellter extends Mitarbeiter
{
    private double grundGehalt;
    private double ortsZuschlag;
    private double zulage;

    public Angestellter(int persNr, String name, double gG, double oZ, double z)
    {
        super(persNr, name);
        this.grundGehalt = gG;
        this.ortsZuschlag = oZ;
        this.zulage = z;
    }

    public double monatsBrutto()
    {
        return grundGehalt + ortsZuschlag + zulage;
    }
}
```

Ein Beispiel für Polymorphismus

```
public class Beamter extends Mitarbeiter
{
    private double grundGehalt;
    private double familienZuschlag;
    private double stellenZulage;

    public Beamter(int persNr, String name, double gG, double fZ, double sZ)
    {
        super(persNr, name);
        this.grundGehalt = gG;
        this.familienZuschlag = fZ;
        this.stellenZulage = sZ;
    }

    public double monatsBrutto()
    {
        return grundGehalt + familienZuschlag + stellenZulage;
    }
}
```

Ein Beispiel für Polymorphismus

```
public class Gehaltsberechnung
{
    public static void main(String[] args)
    {
        Mitarbeiter[] ma = new Mitarbeiter[3];

        ma[0] = new Beamter(1, "Meier", 3021.37, 91.50, 10.70);
        ma[1] = new Angestellter(2, "Maier", 2303.21, 502.98, 132.65);
        ma[2] = new Arbeiter(3, "Mayr", 20.0, 113.5, 35.0, 11.0);

        double bruttoSumme = 0.0;

        for(int i=0; i<ma.length; i++)
        {
            bruttoSumme += ma[i].monatsBrutto();
        }

        System.out.println("Bruttosumme = "+bruttoSumme);
    }
}
```