Vererbung

Florian Adamsky, B. Sc. (PhD cand.)

florian.adamsky@iem.thm.de
http://florian.adamsky.it/

Softwareentwicklung im WS 2014/15

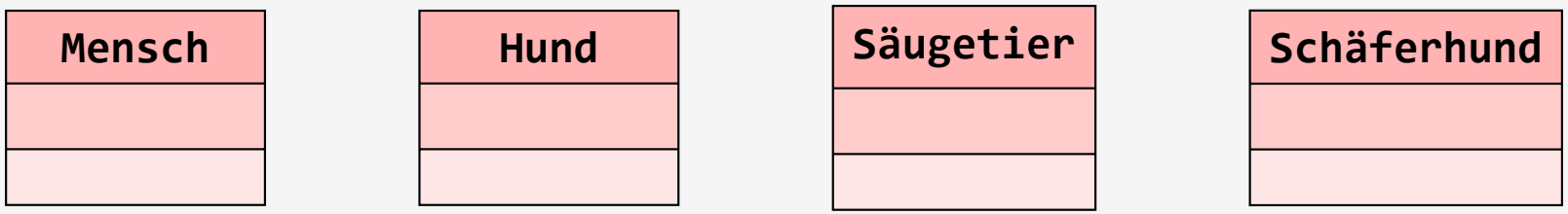
TECHNISCHE HOCHSCHULE MITTELHESSEN

Outline

Inhaltsverzeichnis

Einführung

- Vererbung und Polymorphie sind fundamentale Konzepte der Objektorientierten Programmierung
- menschliche Intelligenz findet, erkennt und erzeugt Beziehungen zwischen Begriffen



- diesen Abstraktionsvorgang bildet C++ über Vererbung ab

Inhaltsverzeichnis

Was ist Vererbung?

- Vererbung bildet in der OOP eine *ist-ein*-Beziehung ab
  - Der Mensch *ist ein* Säugetier
- Systematische Spezialisierung von oben nach unten
- Die *Unterklasse* erbt damit alle Merkmale der *Oberklasse*

Terminologie

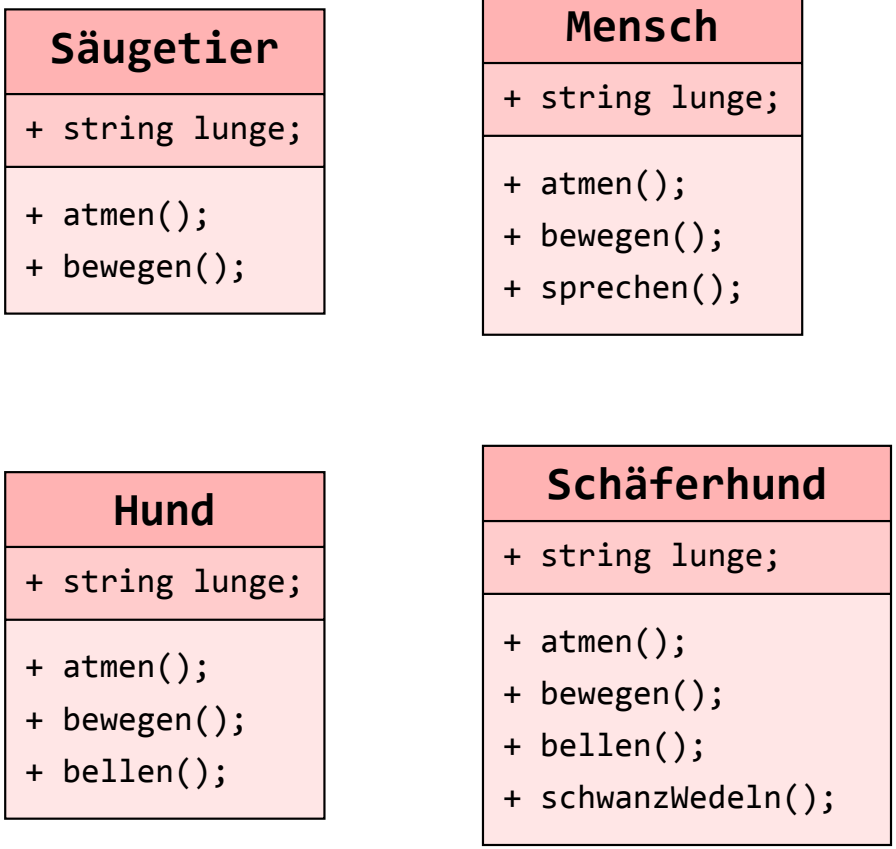
Oberklasse	Basisklasse, Elternklasse, Superklasse
Unterklasse	Kindklasse, Subklasse
Vererbung	Ableitung, Spezialisierung

Was wird vererbt und was nicht?

- Unterklassen erben alle zugreifbaren Member der Basisklasse:
  - Konstruktor und Destruktor
  - Attribute / Klassenvariablen
  - Methoden und Operatoren
- Was wird nicht vererbt?
  - `friend` Funktionen
  - Member die als `private` deklariert sind

Beispiel

Ohne Vererbung



Beispiel

Ohne Vererbung

Säugetier

```
+ string lunge;
+ atmen();
+ bewegen();
```

Mensch

```
+ string lunge;
+ atmen();
+ bewegen();
+ sprechen();
```

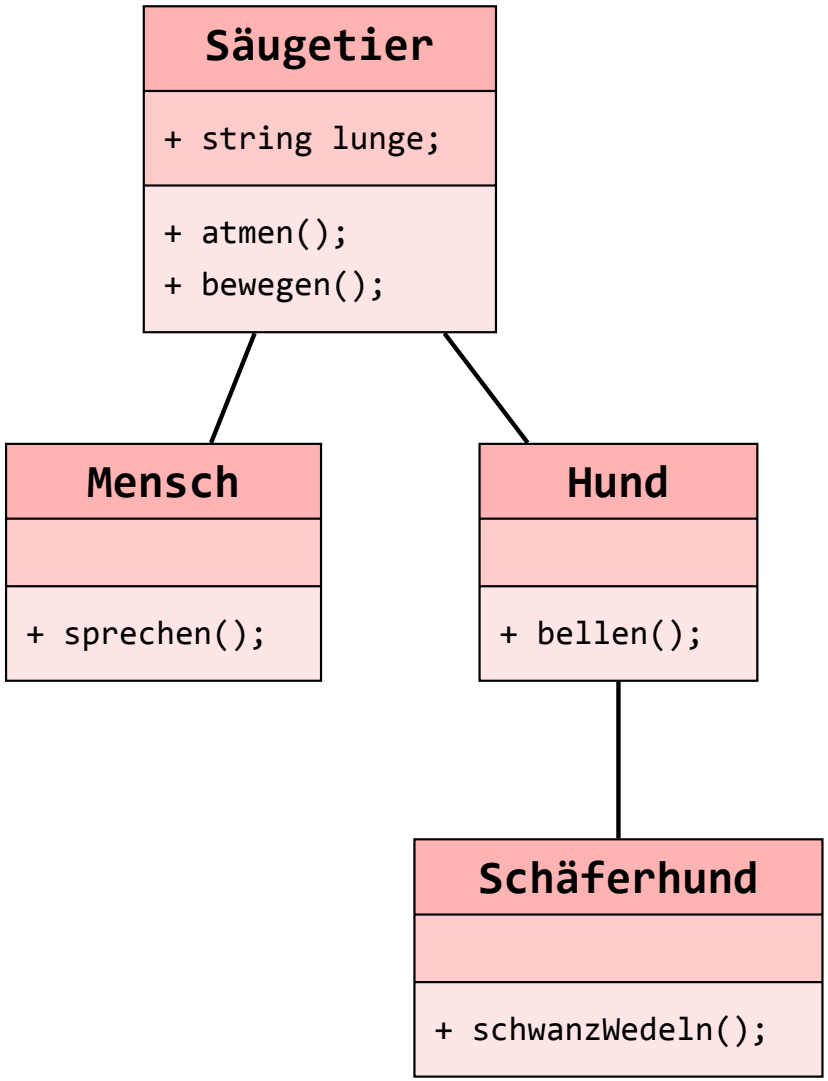
Hund

```
+ string lunge;
+ atmen();
+ bewegen();
+ bellen();
```

Schäferhund

```
+ string lunge;
+ atmen();
+ bewegen();
+ bellen();
+ schwanzWedeln();
```

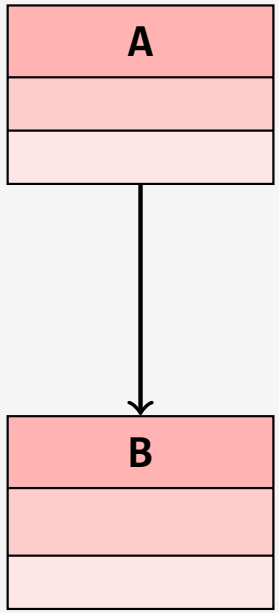
Vererbung



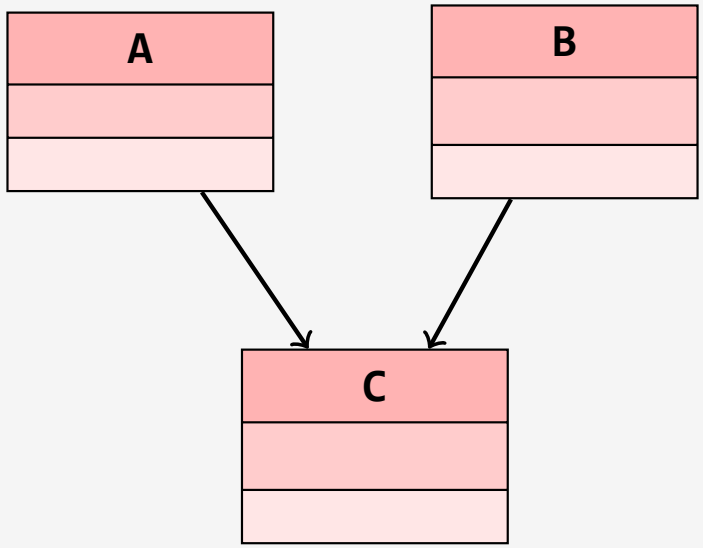
Vorteile von Vererbung

- Klassen sind besser wiederverwendbar
- bessere Wartbarkeit
- weniger Code
  - dadurch weniger Bugs

Einfache Vererbung

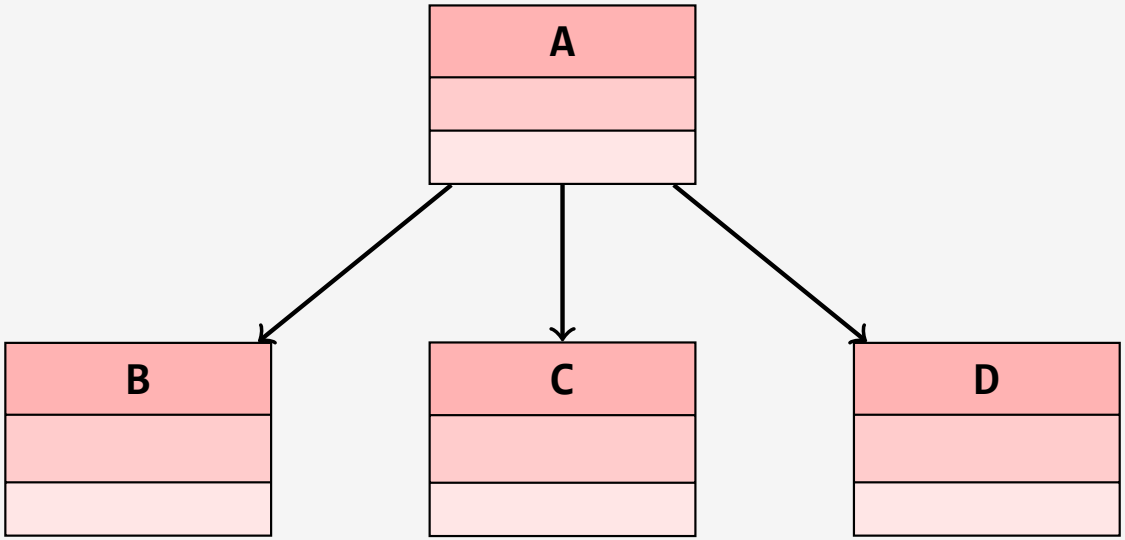


Mehrfach Vererbung

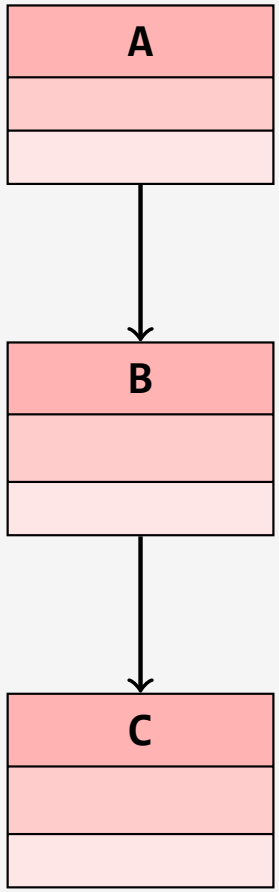


- nur in wenigen Sprachen möglich: C++, Perl, Python
- sollte vermieden werden

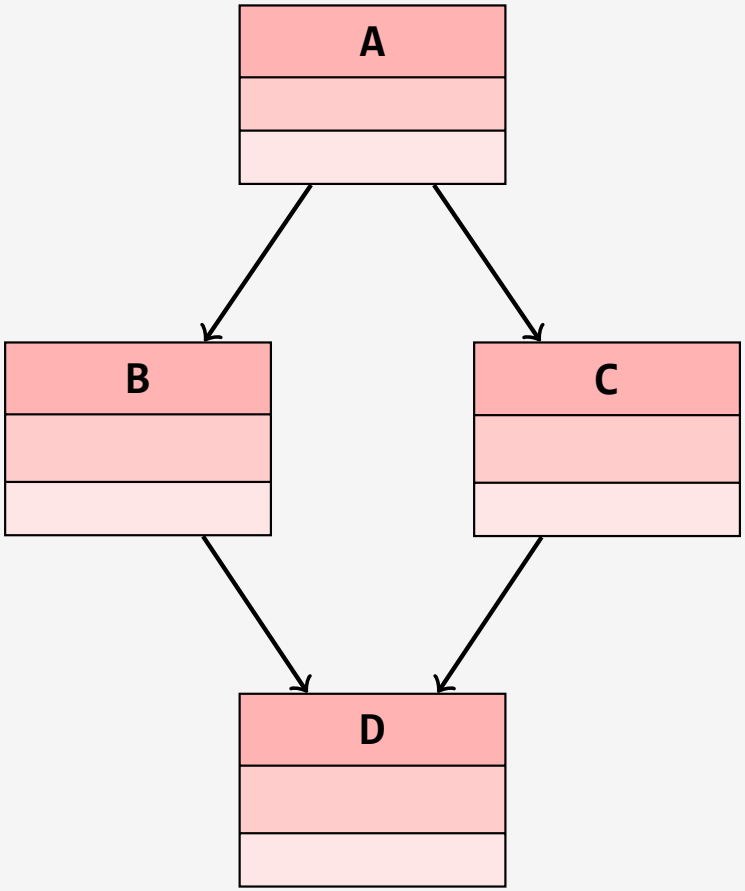
Hierarchische Vererbung



MultiLevel Vererbung



Hybride Vererbung



Public

C++ Syntax

```
class UnterKlasse : public Oberklasse {
    /* ... */
};
```

Oberklasse	Unterklasse
public	public
protected	protected
private	private

Protected

C++ Syntax

```
class UnterKlasse : protected Oberklasse {
    /* ... */
};
```

Oberklasse	Unterklasse
public	protected
protected	protected
private	private

Private

C++ Syntax

```
class UnterKlasse : private OberKlasse {
    /* ... */
};

class UnterKlasse : OberKlasse {
    /* ... */
};
```

Oberklasse	Unterklasse
public	private
protected	private
private	private

Mehrfachvererbung

C++ Syntax

```
class UnterKlasse : public OberKlasse, public OberKlasse2 {
    /* ... */
};
```

Zusammenfassung der Zugriffsschlüsselwörter

private Nur für Attribute/Methoden die innerhalb (intern) der Klassen verwendet werden.

public Stellen Schnittstellen nach außen (Main) dar.

protected Dort zu benutzen, wo eine Subklasse (abgeleitete Klasse) die Methoden/Attributen nochmals benötigt.

Tabelle: Zugriffserlaubnis der einzelnen Typen[1]

Zugriffstyp	Eigene Klasse	Abgeleitete Klasse	Fremde Klasse
`private`	ja	nein	nein
`public`	ja	ja	ja
`protected`	ja	ja	nein

[1] Aus C++-Programmierung, Andre Willms, Addision-Wesley, S.180

Aufruf des Konstruktors der Oberklasse

Example (Klassen)

```
class BasisKlasse {
public:
    BasisKlasse() {
        cout << "Eltern" << endl;
    }
};

class UnterKlasse : public BasisKlasse {
public:
    UnterKlasse(int x) {
        cout << "Kind" << endl;
    }
};
```

Aufruf des Konstruktors der Oberklasse

Example (Klassen)

```
class BasisKlasse {
public:
    BasisKlasse() {
        cout << "Eltern" << endl;
    }
};

class UnterKlasse : public BasisKlasse {
public:
    UnterKlasse(int x) {
        cout << "Kind" << endl;
    }
};
```

Example (Main)

```
int main(void) {
    UnterKlasse kind(23);
}
```

Aufruf des Konstruktors der Oberklasse

Example (Klassen)

```
class BasisKlasse {
public:
    BasisKlasse() {

    }
};

class U
public:
    UnterKlasse(int x) {
        cout << "Kind" << endl;
    }
};
```

Example (Main)

```
int main(void) {
    UnterKlasse kind(23);
}
```

Ausgabe

```
$ ./programm
Eltern
Kind
```

Aufruf des Konstruktors der Oberklasse

Example (Klassen)

Wird automatisch aufgerufen, wenn Konstruktor keine Argumente erwartet

```
class BasisKlasse {
public:
    BasisKlasse() {
        cout << "Eltern" << endl;
    }
};

class UnterKlasse : public BasisKlasse {
public:
    UnterKlasse(int x) {
        cout << "Kind" << endl;
    }
};
```

Example (Main)

```
int main(void) {
    UnterKlasse kind(23);
}
```

Aufruf des Konstruktors der Oberklasse mit Argumente

Example (Klassen)

```
class BasisKlasse {
public:
    BasisKlasse(int x) {
        cout << "Eltern" << endl;
    }
};

class UnterKlasse : public BasisKlasse {
public:
    UnterKlasse(int x) : BasisKlasse(x) {
        cout << "Kind" << endl;
    }
};
```

Aufruf des Konstruktors der Oberklasse mit Argumente

Example (Klassen)

```
class BasisKlasse {
public:
    BasisKlasse(int x) {
        cout << "Eltern" << endl;
    }
};

class UnterKlasse : public BasisKlasse {
public:
    UnterKlasse(int x) : BasisKlasse(x) {
        cout << "Kind" << endl;
    }
};
```

Example (Main)

```
int main(void) {
    UnterKlasse kind(23);
}
```

Aufruf des Konstruktors der Oberklasse mit Argumente

Example (Klassen)

Konstruktor der Basisklasse mit Argumente

```
class BasisKlasse {
public:
    BasisKlasse(int x) {
        cout << "Eltern" << endl;
    }
};

class UnterKlasse : public BasisKlasse {
public:
    UnterKlasse(int x) : BasisKlasse(x) {
        cout << "Kind" << endl;
    }
};
```

Example (Main)

```
int main(void) {
    UnterKlasse kind(23);
}
```

Aufruf des Konstruktors der Oberklasse mit Argumente

Example (Klassen)

```
class BasisKlasse {
public:
    BasisKlasse(int x) {
        cout << "Eltern" << endl;
    }
};

class UnterKlasse : public BasisKlasse {
public:
    UnterKlasse(int x) : BasisKlasse(x) {
        cout << "Kind" << endl;
    }
};
```

Aufruf des Konstruktors der Basisklasse über die Initialization-List (siehe Folien zu Konstruktor/Destruktor)

Example (Main)

```
int main(void) {
    UnterKlasse kind(23);
}
```

Kompatibilität zur Basisklasse

Example (Klassen)

```
class Person {
public:
    string Name;
};

class Student : public Person {
public:
    int MatrikelNr;
};
```

Kompatibilität zur Basisklasse

Example (Klassen)

```
class Person {
public:
    string Name;
};

class Student : public Person {
public:
    int MatrikelNr;
};
```

Example (Main)

```
int main(void) {
    Person  person;
    Student student;

    student.Name = "Heribert";
    student.Matrikel = 2342;

    person  = student;
    student = person;
}
```

Kompatibilität zur Basisklasse

Example (Klassen)

```
class Person {
public:
    string Name;
};

class Student : public Person {
public:
    int MatrikelNr;
};
```

Example (Main)

```
int main(void) {
    Person  person;
    Student student;

    student.Name = "Heribert";
    student.Matrikel = 2342;

    person  = student;
    student = person;
}
```

Kein Problem: Attribute und Methoden von `Student` werden abgeschnitten (`public` wichtig)

Kompatibilität zur Basisklasse

Example (Klassen)

```
class Person {
public:
    string Name;
};

class Student : public Person {
public:
    int MatrikelNr;
};
```

Example (Main)

```
int main(void) {
    Person  person;
    Student student;

    student.Name = "Heribert";
    student.Matrikel = 2342;

    person  = student;
    student = person;
}
```

Compilefehler!

Zusammenfassung

- Eine Klasse (Oberklasse) kann folgende Elemente an eine andere Klasse (Unterklassen) vererben:
  - Konstruktor und Destruktor
  - Attribute / Klassenvariablen
  - Methoden und Operatoren
- Es gibt einfache, mehrfache, hierarchische und hybride Vererbung
- Mit den access specifiern `public`, `protected` und `private` regelt man den Zugriffsschutz der vererbten Elemente
- Der Konstruktor der Oberklasse wird automatisch aufgerufen wenn dieser keine Argumente erwartet
  - ansonsten muss dieser explizit über die Initialization List aufgerufen werden