Teil VI

Objektorientierte Programmierung

Was ist ein Objekt?

Was ist ein Objekt?

- Ein Auto
- Eine Katze
- Ein Stuhl
- ...

Was ist ein Objekt?

- Ein Auto
- Eine Katze
- Ein Stuhl
- ...

- Ein Molekül
- Ein Stern
- Eine Galaxie
- ...

Was ist ein Objekt?

- Ein Auto
- Eine Katze
- Ein Stuhl
- ...

- Ein Molekül
- Ein Stern
- Eine Galaxie
- ...

- Alles, was ein reales Objekt repräsentiert
- Alles, was ein abstraktes Objekt repräsentiert
- Alles, was irgendwelche Eigenschaften besitzt

Wie programmiert man objektorientiert?

- Wir müssen unsere Denkweise ändern!
- Wir müssen unseren Blickpunkt ändern!
- Es ist keine rein technische Frage.

Wie programmiert man objektorientiert?

- Wir müssen unsere Denkweise ändern!
- Wir müssen unseren Blickpunkt ändern!
- Es ist keine rein technische Frage.

Besonderheiten in python?

- Wir verwenden schon die ganze Zeit objekorientierte Konzepte.
- Wir merken es nur nicht.
- In Python ist alles ein Objekt (sogar int).
- Woher weiß z.B. print, wie ein int als Text aussieht?

```
>>> a=4
>>> a.__str__()
'4'
```

Theorie: Klassen und Objekte

Klassen:

”Idee“ aller Objekte einer Art;
Beispiel: Auto

Theorie: Klassen und Objekte

Klassen:

”Idee“ aller Objekte einer Art;
Beispiel: Auto

Instanzen / Objekte:

Konkretes Ding;
Beispiel: ”Das rote Auto dort“

Theorie: Klassen und Objekte

Klassen: ”Idee“ aller Objekte einer Art; Bsp.: Auto

- Eine Klasse ist so etwas ähnliches wie die Definition eines Typs.
- Enthält spezifische Eigenschaften der zu erzeugenden Objekte
- Daten, die einmalig für die Gesamtheit an Objekten gespeichert werden (Klassenvariable / statische Variable)
- Daten, die für jedes (in jedem) Objekt gespeichert werden (Objektvariable)
- Operationen, die auf Objekten ausgeführt werden (Methoden)
- Repräsentiert eine Klasse von Dingen/Objekten

Theorie: Klassen und Objekte

Klassen: ”Idee“ aller Objekte einer Art; Bsp.: Auto

- Eine Klasse ist so etwas ähnliches wie die Definition eines Typs.
- Enthält spezifische Eigenschaften der zu erzeugenden Objekte
- Daten, die einmalig für die Gesamtheit an Objekten gespeichert werden (Klassenvariable / statische Variable)
- Daten, die für jedes (in jedem) Objekt gespeichert werden (Objektvariable)
- Operationen, die auf Objekten ausgeführt werden (Methoden)
- Repräsentiert eine Klasse von Dingen/Objekten

Instanzen / Objekte: Konkretes Ding; Bsp.: ”Das rote Auto dort“

- Eine Instanz ist ein konkretes Objekt, das zu einer Klasse gehört.
- Eine Klasse spezifiziert Eigenschaften, ein Objekt hat konkrete Werte für diese Eigenschaften.
- Zu einer Klasse können beliebig viele Objekte gehören.
- Jedes Objekt gehört genau zu einer Klasse (Ausnahme: Polymorphismus)

Grundstruktur einer Klasse

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Typische Statements

- Funktionsdefinitionen $\Rightarrow$ methods
- Variablendefinitionen $\Rightarrow$ data attributes (data members, fields, ...)

Theorie: Konstruktor, Destruktor

Beim Erzeugen einer Instanz wird...

- Speicher allokiert
- eine Variable zum Referenzieren des Objekts erzeugt
- vielleicht manches initialisiert
- die Initialisierung mit einem Konstruktor (`__init__(self,...)`) durchgeführt

Theorie: Konstruktor, Destruktor

Beim Erzeugen einer Instanz wird...

- Speicher allokiert
- eine Variable zum Referenzieren des Objekts erzeugt
- vielleicht manches initialisiert
- die Initialisierung mit einem Konstruktor (`__init__(self,...)`) durchgeführt

Wenn ein Objekt nicht mehr gebraucht wird, ...

- wird Speicher freigegeben
- müssen einige Dinge aufgeräumt werden
- wird der Destruktor (`__del__(self,...)`) aufgerufen
- ABER: Es ist nicht garantiert, wann er aufgerufen wird

```
class Example:
  def __init__(self, ...):
    do_something
  def __del__(self, ...):
    do_something
```

Theorie: self

Verwenden eines Objekts

- Zugriff auf Methoden: `objekt.methode(...)`
- Beispiel: `liste.append('x')`
- Zugriff auf Variablen: `objekt.variable`
- Beispiel: `complex.real`

Theorie: self

Verwenden eines Objekts

- Zugriff auf Methoden: `objekt.methode(...)`
- Beispiel: `liste.append('x')`
- Zugriff auf Variablen: `objekt.variable`
- Beispiel: `complex.real`

Zugriff innerhalb der Objektmethoden

- Konkretes Objekt wird außerhalb der Klassendefinition erzeugt

$\Rightarrow$ Den Klassenmethoden ist der Name nicht bekannt

$\Rightarrow$ Alle Methoden bekommen zusätzlichen Formalparameter `self`

- `self` ist immer der erste Formalparameter, dieser wird beim Aufruf (Aktualparameter) weggelassen
- Innerhalb der Methoden Zugriff auf andere Methoden und Variablen mit `self.variable` bzw. `self.methode()`

Erstes Klassenbeispiel

```
class Student:
  "Einfache␣Beschreibung␣eines␣Studierenden"
  def __init__(self, name): # Konstruktor
    self.name = name

  def getName(self):          # Methode
    return self.name

person1 = Student("Alex")
person2 = Student("Michaela")
print "Hier␣kommt␣" + person1.getName()
```

Theorie: Methoden

- . . . implementieren das Verhalten von Objekten.
- . . . repräsentieren alle Arten von Veränderungen eines Objekts nach der Initialisierung.
- accessor methods: geben Info über Zustand des Objekts
- mutator methods: verändern Zustand des Objekts

$\Rightarrow$ verändern (mind.) eine Objektvariable

Theorie: Objektübergreifende (statische) Daten

Statische Variablen (Klassenvariablen)

- Variablen pro Objekt speicherbar: Objektvariable
  Beispiel: Name in `Student`
  $\Rightarrow$ in Konstruktor definieren mit self
- Variablen pro Klasse speicherbar: Klassenvariable
  $\Rightarrow$ nur 1x pro Klasse existent
  Klassisches Beispiel: Counter

Theorie: Objektübergreifende (statische) Daten

Statische Variablen (Klassenvariablen)

- Variablen pro Objekt speicherbar: Objektvariable
  Beispiel: Name in `Student`
  $\Rightarrow$ in Konstruktor definieren mit self
- Variablen pro Klasse speicherbar: Klassenvariable
  $\Rightarrow$ nur 1x pro Klasse existent
  Klassisches Beispiel: Counter

Statische Methoden

- Methoden, die lediglich statische Variablen (Klassenvariablen) benutzen und unabhängig von konkreten Objekten verwendbar sein sollen
- Definition via Schlüsselwort `@staticmethod`

Beispiel: Student reloaded

```
class Student:
  "Einfache␣Beschreibung␣eines␣Studierenden"
  anzahl = 0                    # Klassenvariable
  def __init__(self, name):     # Konstruktor
    self.name = name            # Objektvariable
    Student.anzahl += 1
  def getName(self):            # Methode
    return self.name
  @staticmethod
  def getAnzahl():              # statische Methode
    return Student.anzahl

person1 = Student("Alex")
person2 = Student("Michaela")
print "Hier␣kommt␣" + person1.getName()
print "Anzahl␣Studis=", Student.getAnzahl()
```

Prozeduraler Ansatz: Ofen

```
#!/usr/bin/python
def backe(temperature, mode):
    Anweisungsblock

ofentemperatur = 180
ofenmodus = 2
backe(ofentemperatur, ofenmodus)
```

Prozeduraler Ansatz: Ofen

```
#!/usr/bin/python
def backe(temperature, mode):
    Anweisungsblock

ofentemperatur = 180
ofenmodus = 2
backe(ofentemperatur, ofenmodus)
```

Bei einer ganzen Bäckerei?

```
ofentemperaturen = []
ofenmodi = []
ofentemperaturen.append(180)
ofenmodi.append(2)
...
for i in range(len(ofentemperaturen)):
  backe(ofentemperaturen[i], ofenmodi[i])
```

Objektorientierter Ansatz: Ofen

```
#!/usr/bin/python
class Ofen(object):
  def __init__(self, temperatur, modus):
    self.temperatur=temperatur
    self.modus=modus
  def backe(self):
    Anweisungsblock
meinOfen = Ofen(180,2)
meinOfen.backe()
```

Objektorientierter Ansatz: Ofen

```
#!/usr/bin/python
class Ofen(object):
  def __init__(self, temperatur, modus):
    self.temperatur=temperatur
    self.modus=modus
  def backe(self):
    Anweisungsblock
meinOfen = Ofen(180,2)
meinOfen.backe()
```

Und wieder die ganze Bäckerei

```
ofenliste = []
ofenliste.append(Ofen(180,2))
...
for ofen in ofenliste:
  ofen.backe()
```

Was sind die Unterschiede?

Prozeduraler Ansatz

- Prozeduren/Funktionen legen fest, wie etwas „produziert“wird
- Bei der Implementierung denkt man an durchzuführende Aktionen
- Um die Speicherung der Daten muss man sich selbst kümmern

Was sind die Unterschiede?

Prozeduraler Ansatz

- Prozeduren/Funktionen legen fest, wie etwas „produziert“wird
- Bei der Implementierung denkt man an durchzuführende Aktionen
- Um die Speicherung der Daten muss man sich selbst kümmern

Objektorientierter Ansatz

- Objekte spezifizieren „Dinge“(real oder abstrakt)
- Bei der Implementierung muss man sich das Ding und seine Eigenschaften vorhalten
- Sämtliche Daten sind im Objekt selbst gespeichert (und damit gekapselt!)

$\Rightarrow$ Wissen über das Was, aber nicht über das Wie

$\Rightarrow$ Modularisierung! (größere) Unabhängigkeit versch. Code-Teile

- Wenn viele Daten/Variablen mit einer Aktion verbunden sind, lohnt sich der objektorientierte Ansatz besonders

Informationskapselung / Information Hiding

public

- per default erstmal alles (alle Variablen/Methoden)
- komfortabel (Zugriff von außen)
- oft gefährlich / unerwünscht

Informationskapselung / Information Hiding

public

- per default erstmal alles (alle Variablen/Methoden)
- komfortabel (Zugriff von außen)
- oft gefährlich / unerwünscht

private

- Zugriff von außen (d.h. außerhalb der Klasse) verboten

$\Rightarrow$ Variablen/Methoden außen nicht sichtbar

$\Rightarrow$ Variablen/Methoden außen nicht nutzbar

- Workaround in python: name mangling

$\Rightarrow$ Syntax: beginnend mit `__`, aber nicht endend mit `__` (`__privateVar`, `__privateMet()`)

- Achtung: indirekt zugreifbar (`._Klasse__privateVar`)

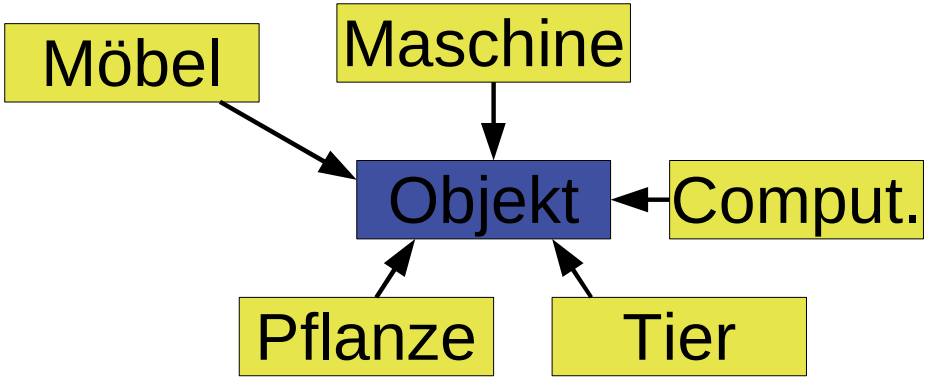
Spezielle Variablen und Methoden

- `__get__(value)`: Erlaubt Lesezugriff mit `[]`
- `__set__(i, value)`: Erlaubt Schreibzugriff mit `[]`
- `__add__(value)`: Erlaubt addition mit `+`
- `__str__()`: Wird von `print` verwendet
- `__call__()`: Erlaubt Aufruf mit `objekt()`

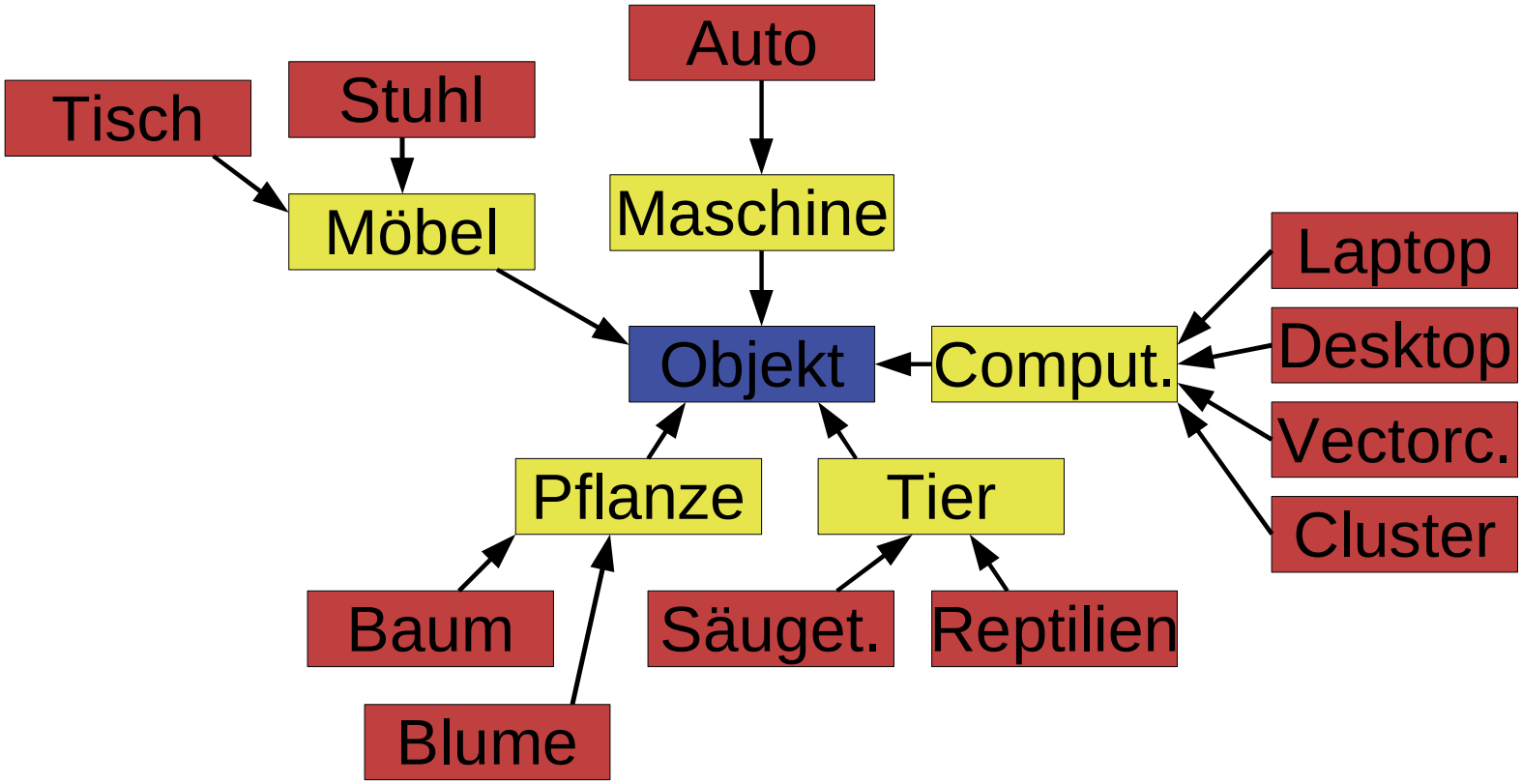
```
class Test:
  def __init__(self, val):
    self.value = val
  def __add__(self, b):
    return self.value + b.value
  def __mul__(self,b):
    return self.value * b.value
```

Vererbung in der Realität

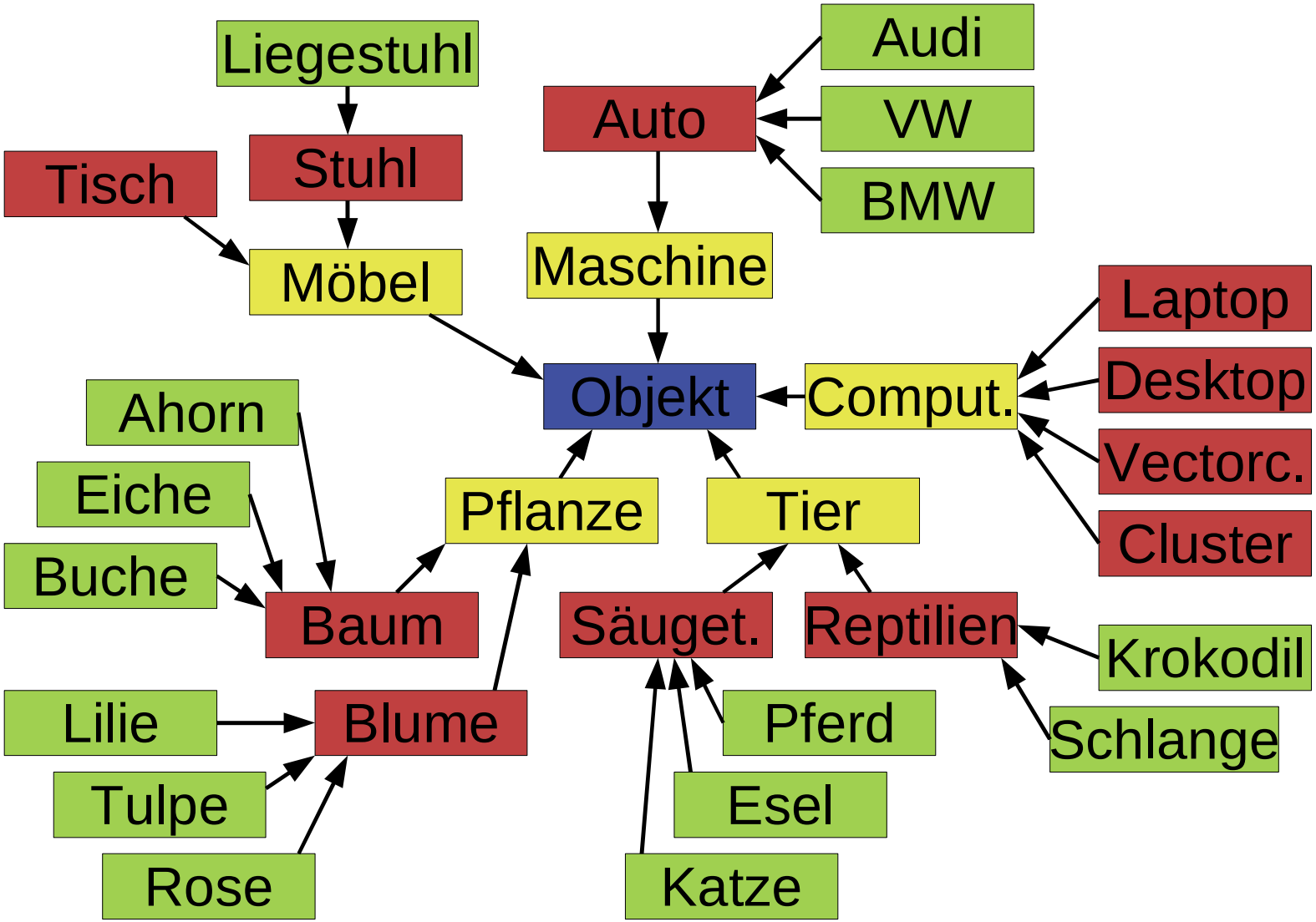
Vererbung in der Realität



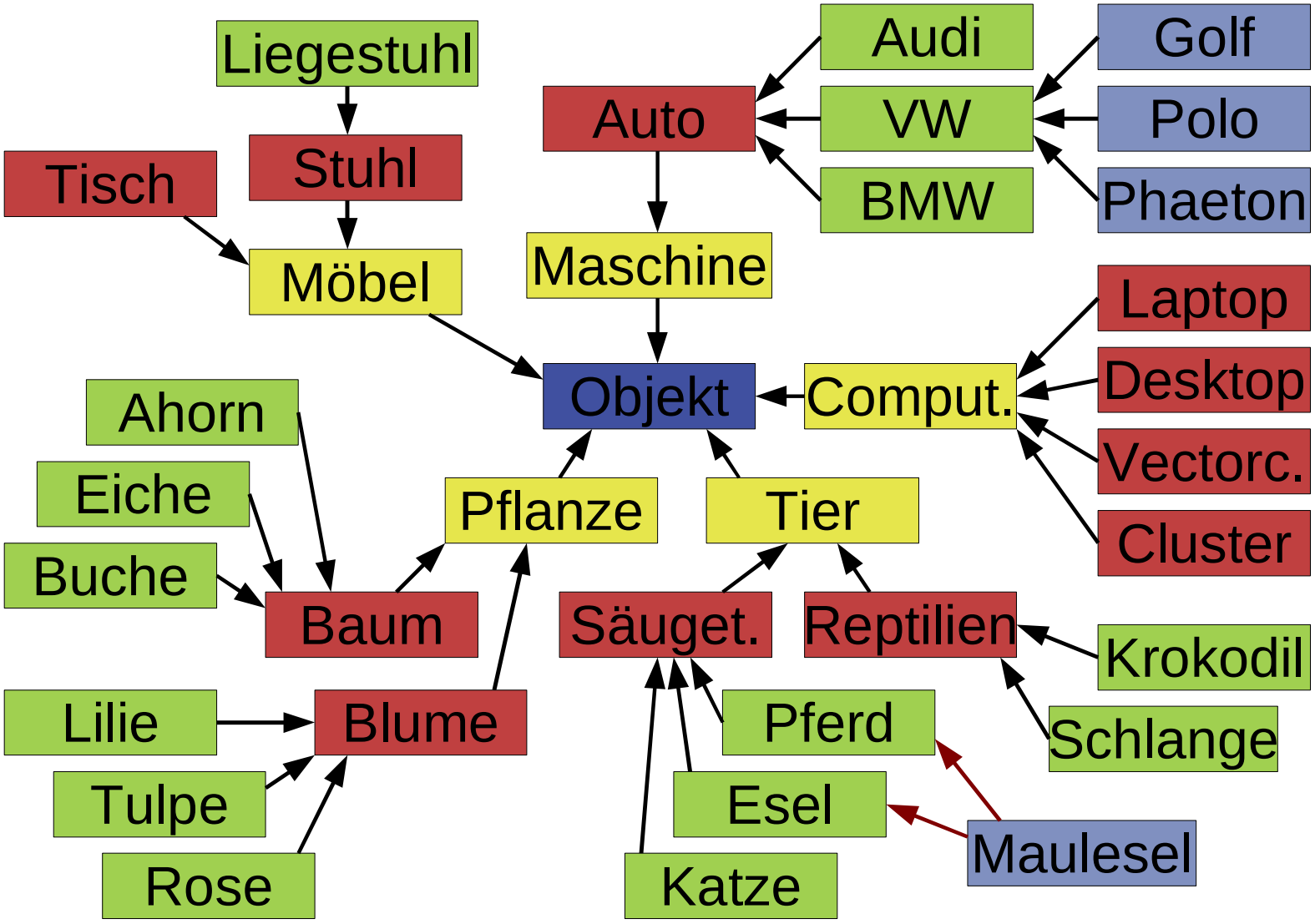
Vererbung in der Realität



Vererbung in der Realität



Vererbung in der Realität



Vererbung

- Klassen können von einer *Basisklasse / Oberklasse* erben
- Dadurch sind sie *abgeleitete Klasse / Unterklasse* der Basisklasse
- Alle Klassen können spezialisiert werden
- Alle Klassen zusammen bilden eine Klassenhierarchie
- Methoden der Oberklasse können übernommen oder neu definiert (überschrieben) werden

```
class Werkstudent(Student):
  def __init__(self, name, thema):
    Student.__init__(self, name)
    self.thema = thema
  def printInfo(self):
    print "%s arbeitet an: %s" % (self.name, self.thema)

person3 = Werkstudent("Stefan", "Tabellenkalkulation")
person3.printInfo()
```

Modellierung mit UML

Flussdiagramme

- Nur geeignet für kleine Algorithmen
- Keine Darstellung von Klassenhierarchien
- Keine Darstellung von Datenabhängigkeiten
- ...

Modellierung mit UML

Flussdiagramme

- Nur geeignet für kleine Algorithmen
- Keine Darstellung von Klassenhierarchien
- Keine Darstellung von Datenabhängigkeiten
- ...

UML: Unified Modeling Language

- „visuelle“Modellierungssprache
- Sprache unterstützt unterschiedliche Diagrammtypen
- Software wird über diese Diagramme/Modelle spezifiziert
- Kann als Basis für die Dokumentation dienen
- Automatische Code-Generierung möglich
- Auch für sonstige betriebliche Abläufe geeignet

Strukturdiagramme

- Klassendiagramm
- Objektdiagramm
- Kompositionsstrukturdiagramm
- Komponentendiagramm
- Verteilungsdiagramm
- Paketdiagramm
- Profildiagramm

Strukturdiagramme

- Klassendiagramm
- Objektdiagramm
- Kompositionsstrukturdiagramm
- Komponentendiagramm
- Verteilungsdiagramm
- Paketdiagramm
- Profildiagramm

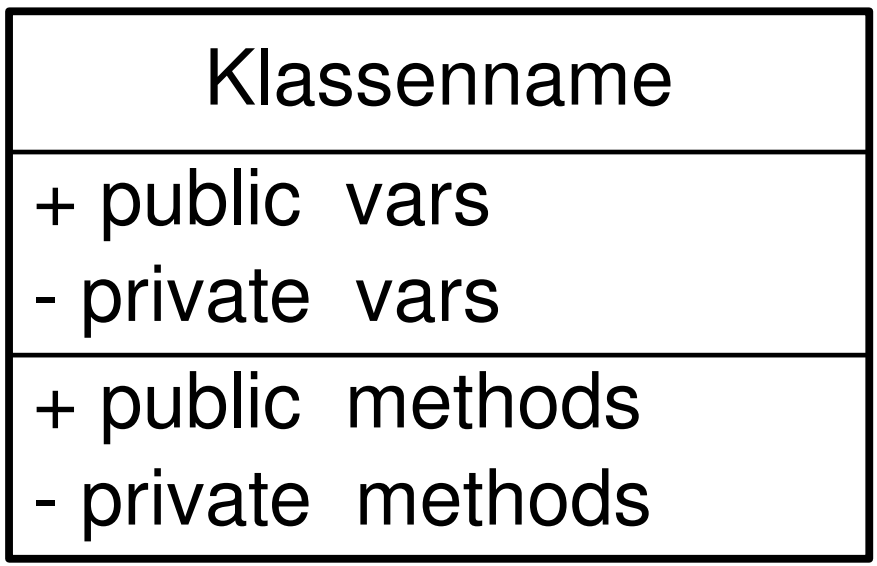
Verhaltensdiagramme

- Aktivitätsdiagramm
- Anwendungsfalldiagramm (Use-Cases)
- Interaktionsübersichtsdiagramm
- Kommunikationsdiagramm
- Sequenzdiagramm
- Zeitverlaufsdiagramm
- Zustandsdiagramm

Klassendiagramm

Einzelne Klasse

- Klasse dargestellt durch Rechteck
- Horizontale Unterteilung in drei Bereiche
- Oberster Bereich: Klassenname
- Mittlerer Bereich: Attribute/Variablen
- Unterer Bereich: Methoden

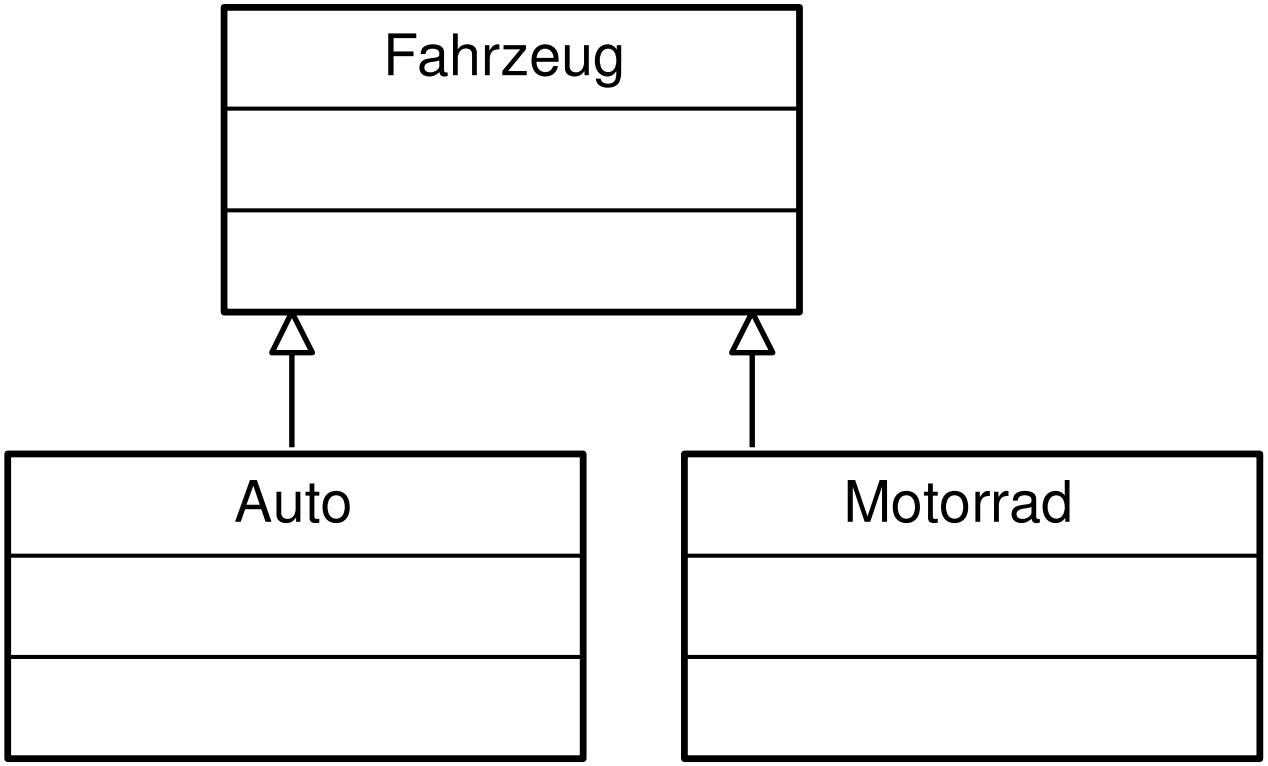


Syntax der Klassenbeschreibung

- Name von Klasse/Variablen/Methoden frei wählbar
- +: public Variable/Methode (optional)
- -: private Variable/Methode (optional)
- Für Variablen kann der Typ angegeben werden (nach :)
- Für Methoden kann der Rückgabewert angegeben werden (nach :)

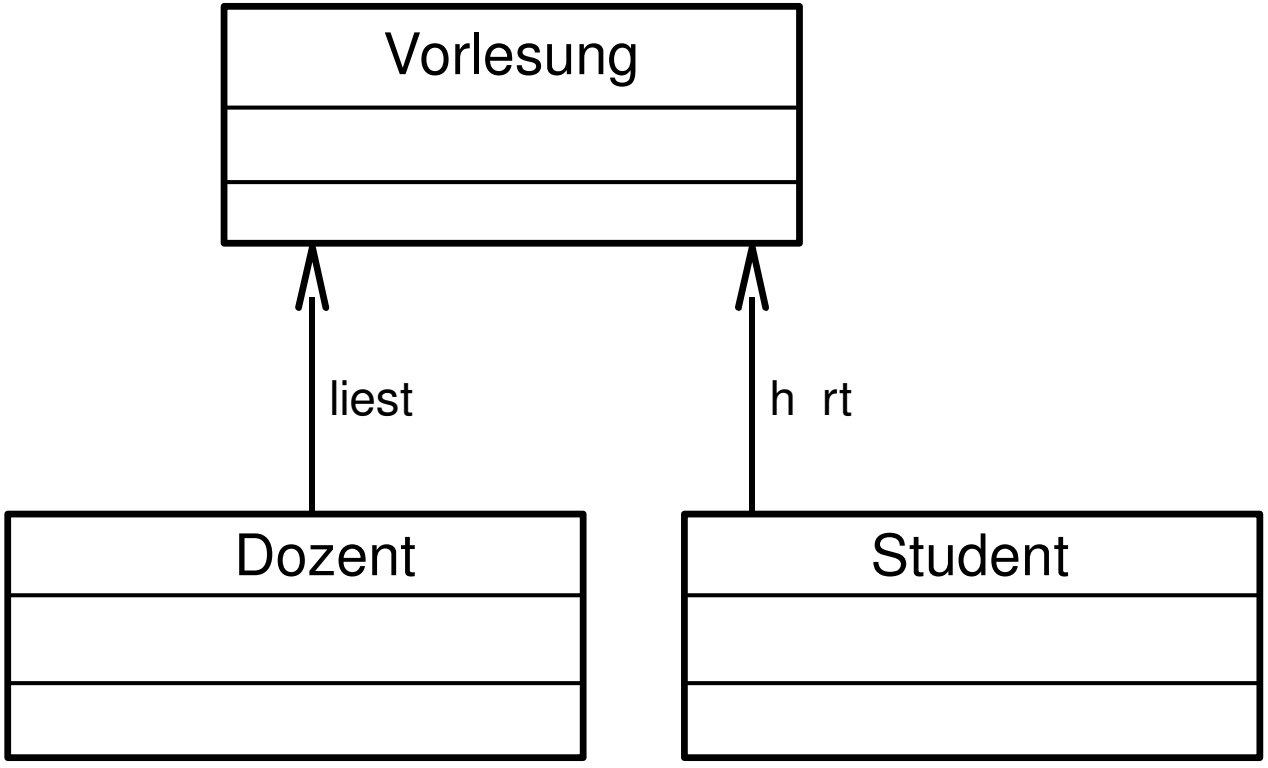
Vererbung

- Darstellung durch Pfeil mit „geschlossener“Spitze
- „ist ein“- Beziehung
- Pfeil von spezialisierter Klasse zur Vaterklasse



Assoziation

- Semantischer Zusammenhang zwischen Klassen
- Beschreibung der Assoziation neben dem Pfeil



Aggregation und Komposition

- Aggregation (leere Raute): „hat ein“
- Komposition (volle Raute): „enthält ein“
- Angabe der Anzahl möglich