Teil VIII

Objektorientierte Programmierung

Was ist ein Objekt?

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

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

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

Beispiel: Liste umdrehen

Als Funktion

```
def reverse(liste):
  kopie = list(liste)
  for i in range(len(liste)):
    liste[i] = kopie[len(liste)-i-1]

l = [1, 2, 3]
reverse(l)
```

Als Methode

```
Class list()
  def reverse(self):
    ...
l = [1, 2, 3]
l.reverse()
```

Wie programmiert man objektorientiert?

- Ändert eure Denkweise
- Ändert euren Blickpunkt
- Es ist keine rein technische Frage

Besonderheiten in python?

- Ihr verwendet schon die ganze Zeit objekorientierte Konzepte
- Ihr merkt 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

- 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

- Eine Instanz is 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)

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`

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

```
#!/usr/bin/python
class Student(object):
  anzahl = 0
  def __init__(self, name): # Konstruktor
    self.name = name      # Objektvariable
    Student.anzahl += 1 # Klassenvariable
  def getName(self):      # Methode
    return self.name
  @staticmethod
  def getAnzahl():
    return Student.anzahl

person1 = Student("Alex")
person2 = Student("Michael")
print person1.getName()       # "Alex"
print Student.getAnzahl() # 2
```

Proceduraler 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(190)
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():
    Anweisungsblock
meinOfen = Ofen(180,2)
meinOfen.backe()
```

Und wieder die ganze Bäckerei

```
ofenliste = []
ofenliste.append(Ofen(190,2))
...
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

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

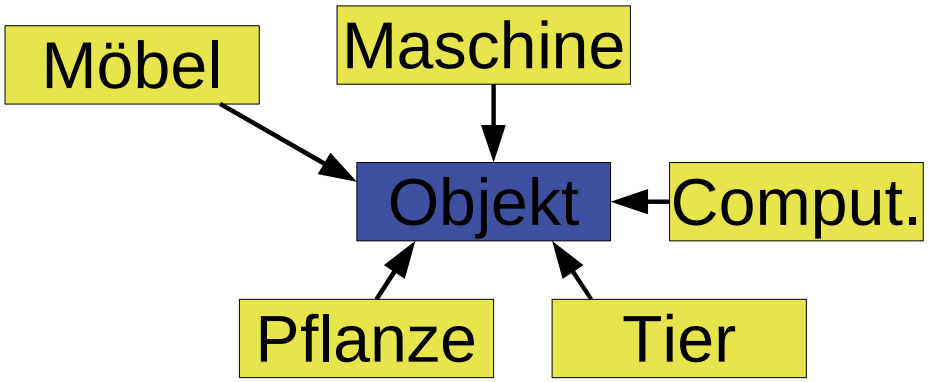
Spezielle Variablen und Methoden

- Variablen und Methoden beginnend mit `__`, aber nicht endend mit `__` (`__privateVar`, `__privateMet()`) sind privat
- Achtung: private Variablen und Methoden sind indirekt zugreifbar (`._Klasse__privateVar`)
- `__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()`

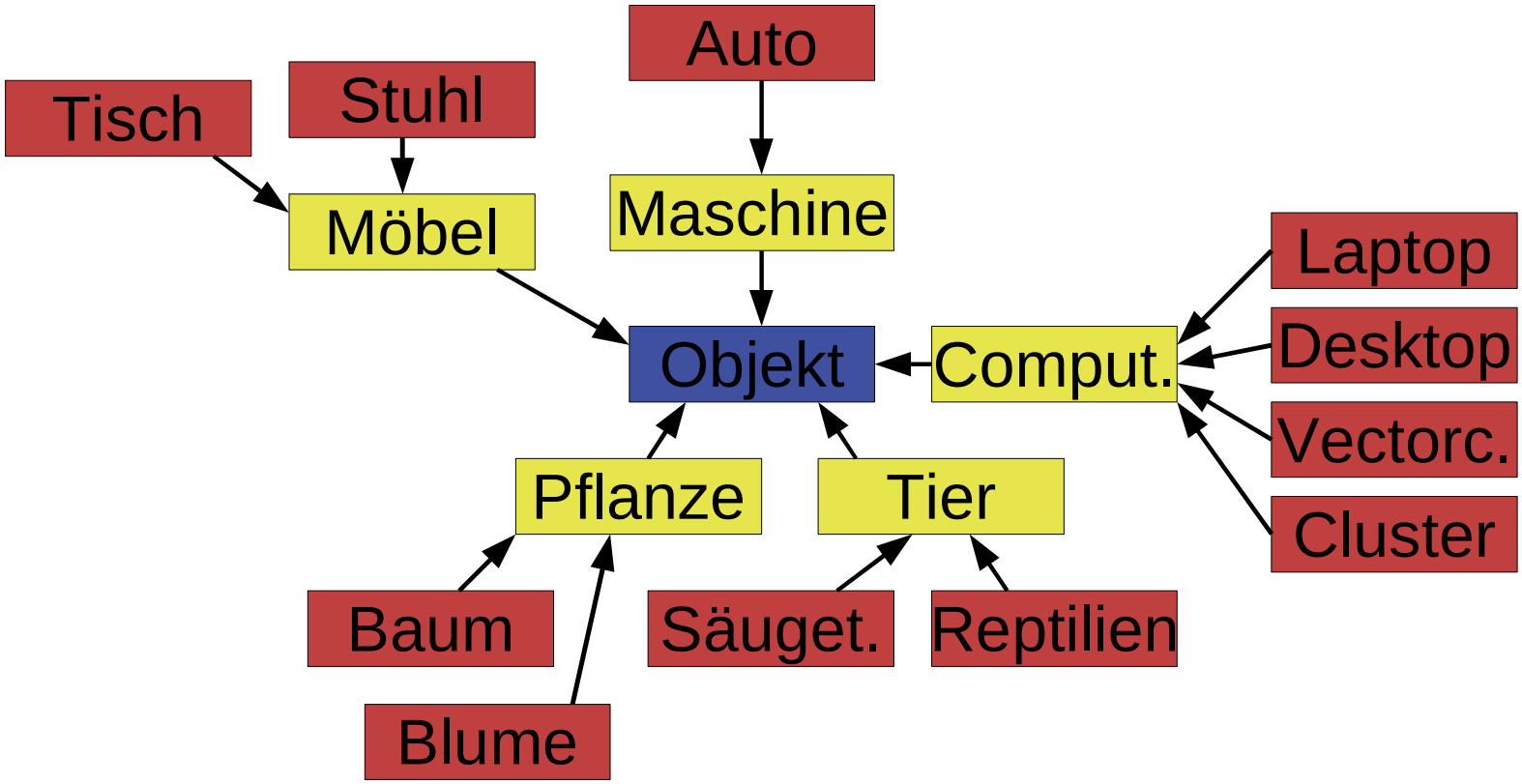
```
class Test:
  def __init__(self, val):
    self.value = val
  def __add__(self, b):
    return self.value+b.value+1
  def __mul__(self,b):
    return self.value*b.value*2
```

Vererbung in der Realität

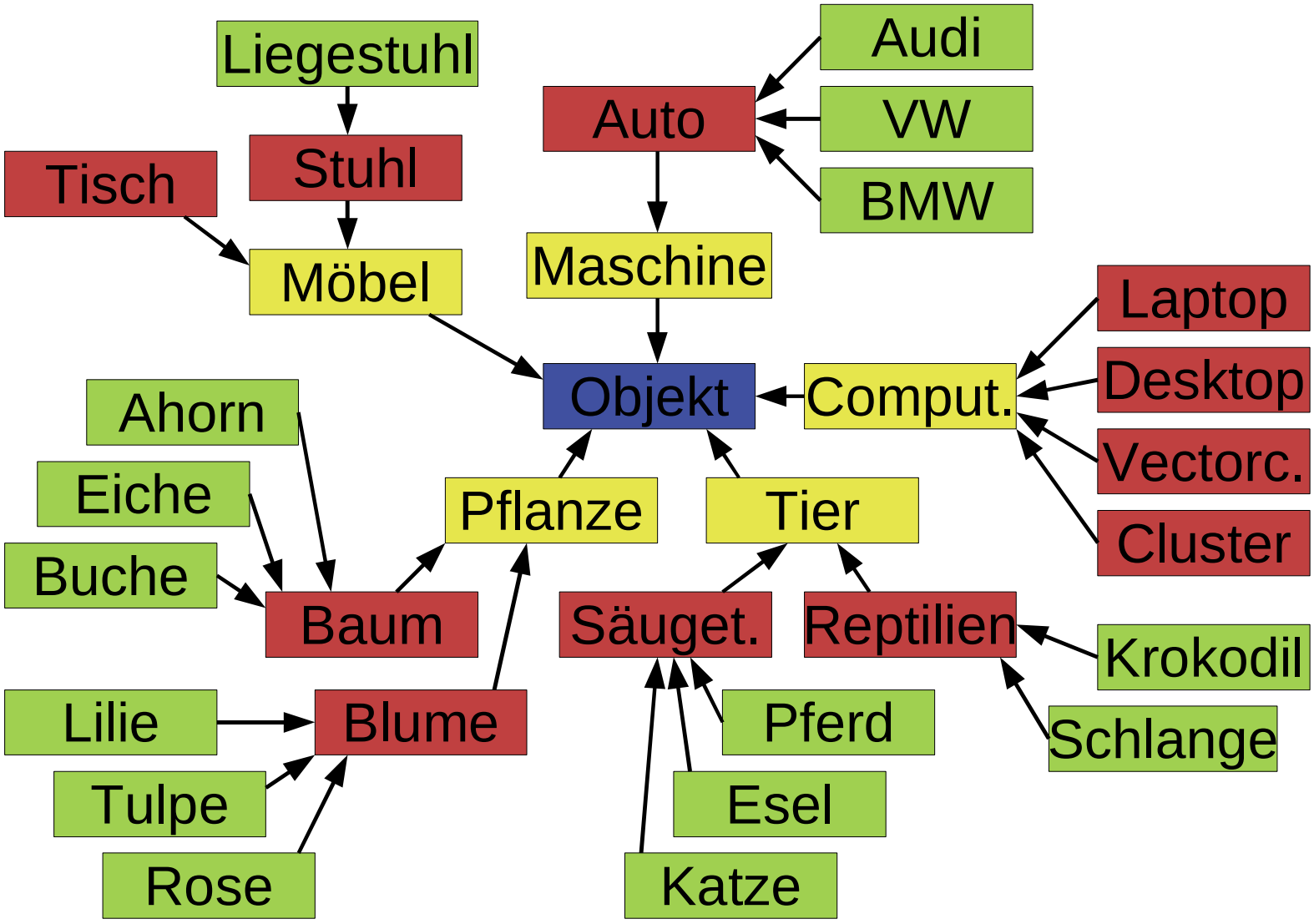
Vererbung in der Realität



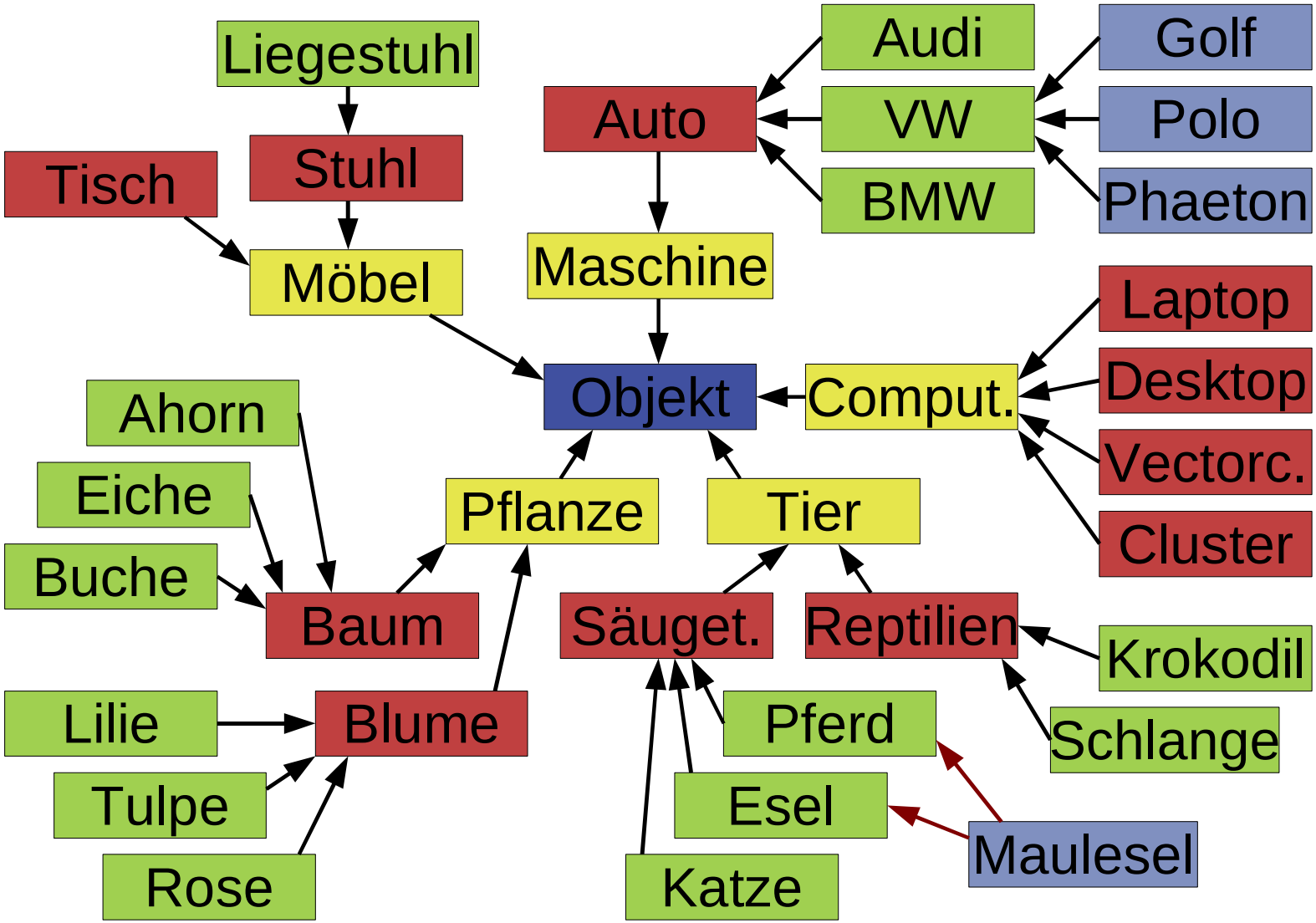
Vererbung in der Realität



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

person3 = Werkstudent("Stefan", "Kaffee␣kochen")
person3.printInfo()
```

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

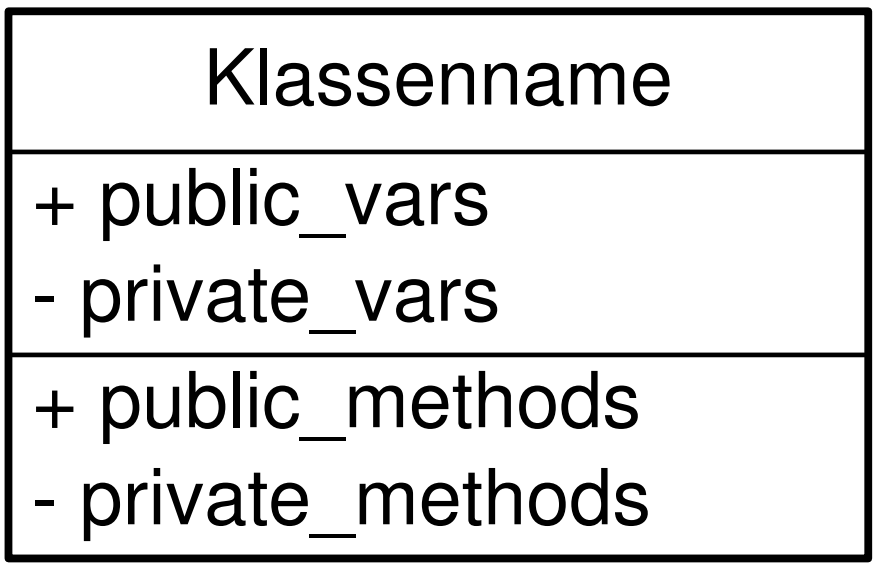
Verhaltensdiagramme

- 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

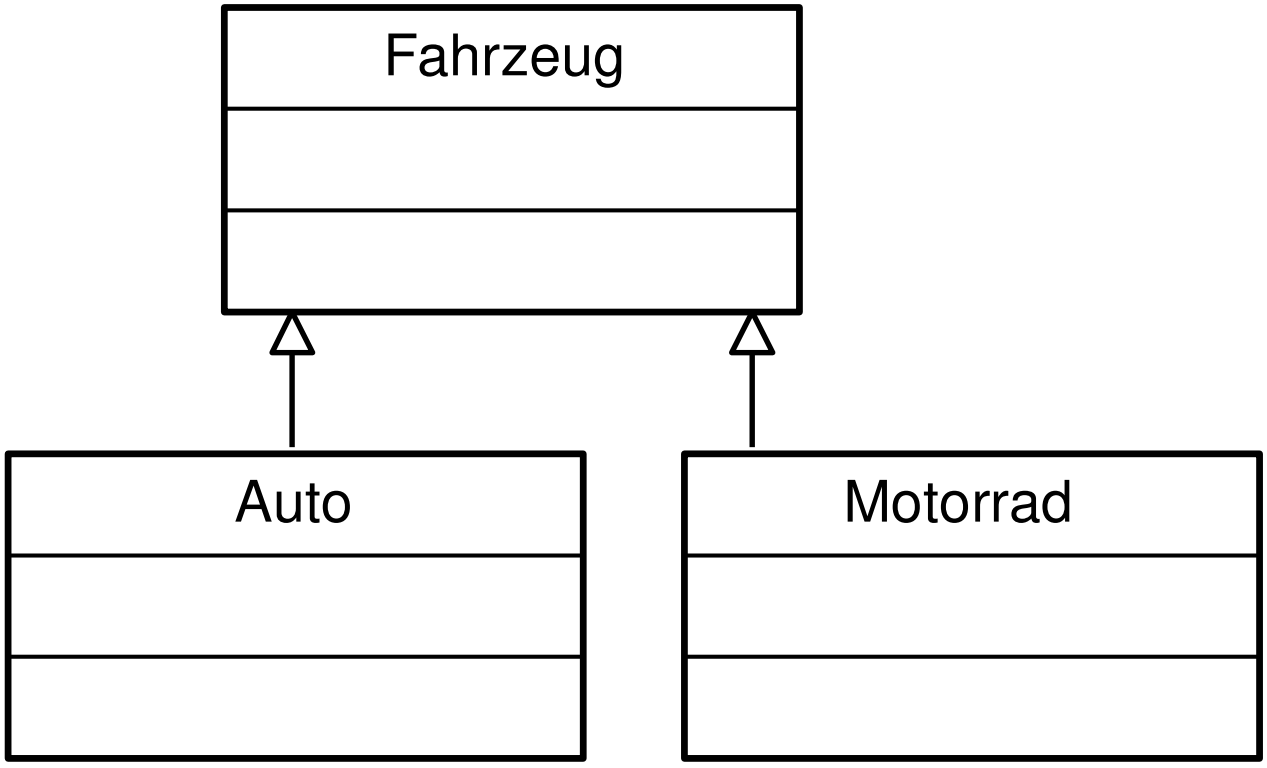


Syntax der Klassenbeschreibung

- 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 :)

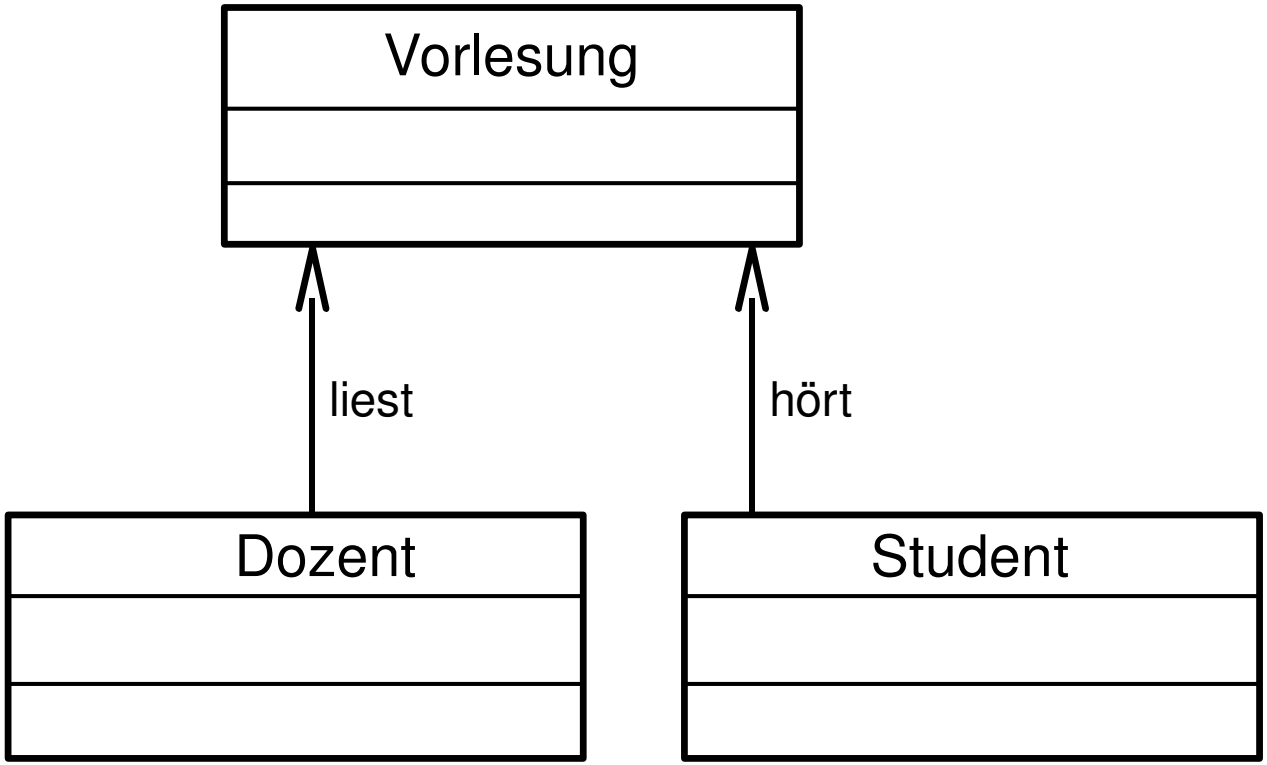
Vererbung

- 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