Grundlagen der Programmierung

Vorlesung und Übung

07 – Assoziation und Vererbung

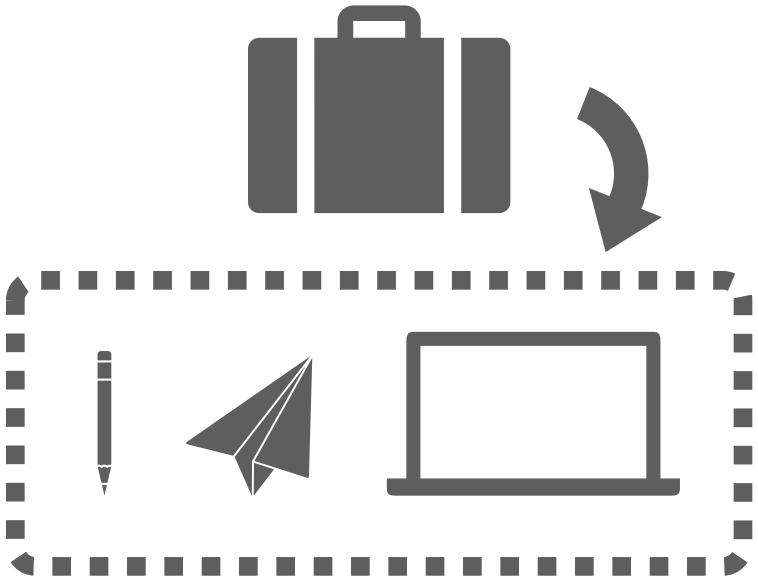
Prof. Dr. Andreas Biesdorf

Wirtschaft
Hauptcampus

H O C H
S C H U L E
T R I E R

ASSOZIATIONEN

Grundidee



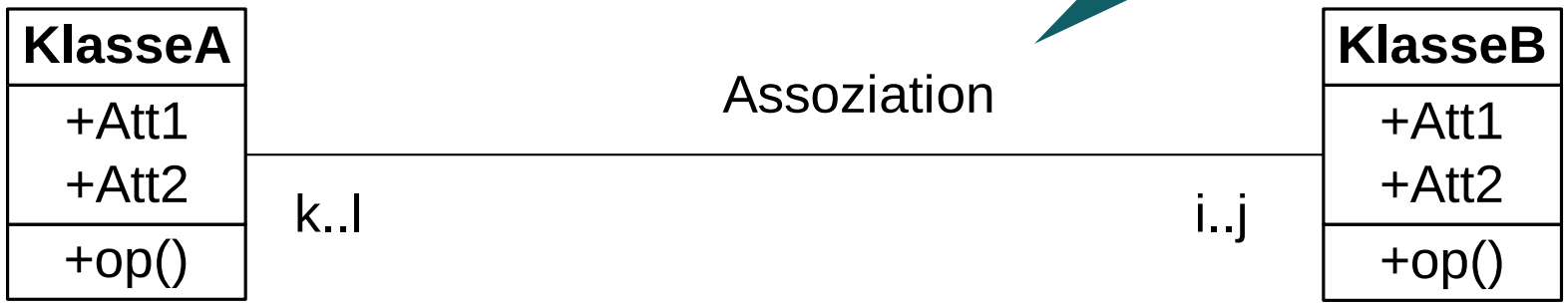
- Objekte für sich genommen, sind nur bedingt nützlich
- Nach dem Vorbild der Realität verbindet man unterschiedliche Objekte miteinander
- Ein Objekt kennt/hat also andere Objekte

DIE UNIFIED MODELING LANGUAGE - UML

Zentrales Konstrukt ist die **Klasse (class)**, mit der gleichartige Objekte hinsichtlich
- **Struktur** (Attribute)
- **Verhalten** (Operationen/Methoden)

modelliert werden.

Assoziationen zwischen Klassen entsprechen den Beziehungstypen
- Generalisierung
- Aggregation
- Komposition



DIE UNIFIED MODELING LANGUAGE - UML

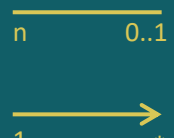
Klasse

Modellierung gleichartiger Objekte hinsichtlich

- Struktur (Attribute)
- Verhalten (Methoden/Operationen)

Modellierung der Sichtbarkeit von Attributen und Methoden

- **+, -, #: public, private, protected**



Assoziation

Darstellung von Beziehung zwischen Klassen

- Gerichtet / ungerichtet
- Berücksichtigung von Kardinalitäten durch Angabe von Multiplizitäten
- Angabe der Rolle einer Assoziation

Aggregation und Komposition

Aggregation: „einfache“ Teil/Ganzes-Beziehung

Komposition: exklusive Zuordnung von existenzabhängigen Teil-Objekten zu einem übergeordneten Objekt. Multiplizität daher immer 1 auf der Seite des übergeordneten Objekts.

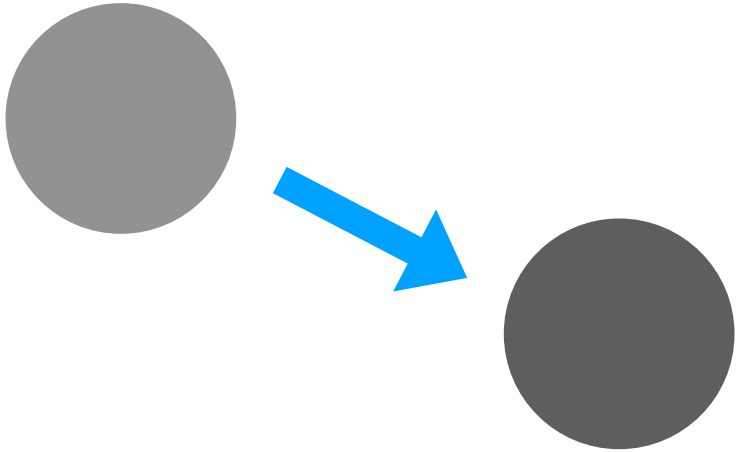
Generalisierung

Hierachien von Klassen

Vererbung von Attributen und Operatoren

Aus: Kemper / Eickler 2015

Unidirektionale Assoziation

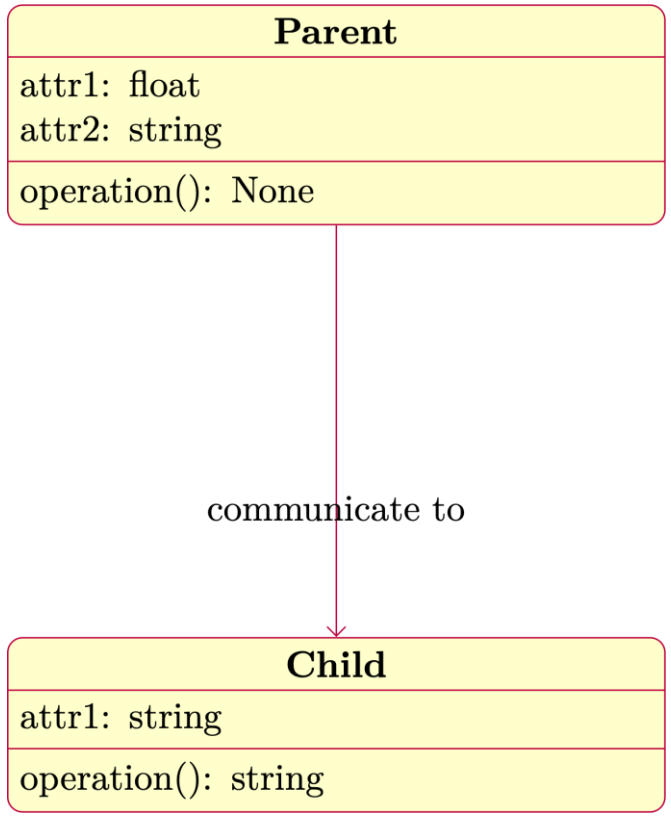


Die Verbindung geht nur in eine Richtung

Beispiele:

- Der Hörer kennt das Radio, aber das Radio nicht den Hörer
- Die Studierenden wissen, wer die Klausur korrigiert, aber der Dozent weiß nicht, welche Klausur zu welchem Studenten gehört

Unidirektionale Assoziationen in UML



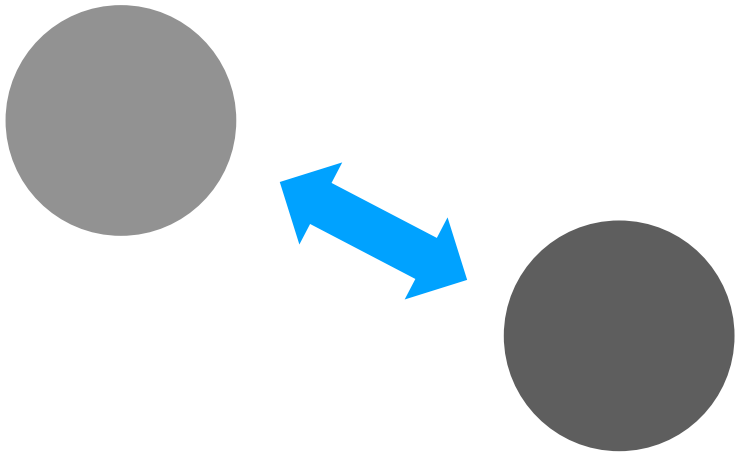
- Einfacher Pfeil von der Klasse mit der Referenz zur Klasse ohne Referenz
- Verbindung kann Erklärung enthalten

Unidirektionale Assoziationen in Python

```
1 class Parent:
2     def __init__(self, child):
3         self._child = child
4
5 class Child:
6     pass
```

- Nur die Klasse von der die Assoziation ausgeht, hält eine Referenz zur anderen Klasse
- Man kann also vom Parent zum Child, aber nicht vom Child zum Parent

Bidirektionale Assoziationen

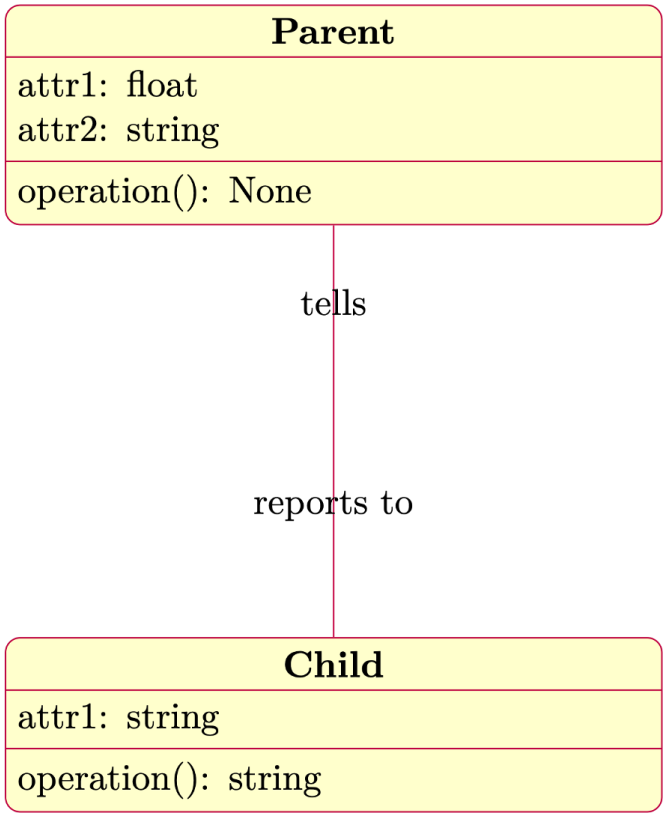


Die Verbindung geht in beide Richtungen

Beispiele:

- Der Anrufer kennt den Angerufenen und der Angerufene kennt den Anrufer
- Der Dozent kennt seine Studierenden und die Studierenden den Dozenten

Bidirektionale Assoziationen in UML



- Verbindung ohne Pfeile zwischen den beiden Klassen
- Ebenfalls mit Erklärung möglich

Bidirektionale Assoziationen in Python

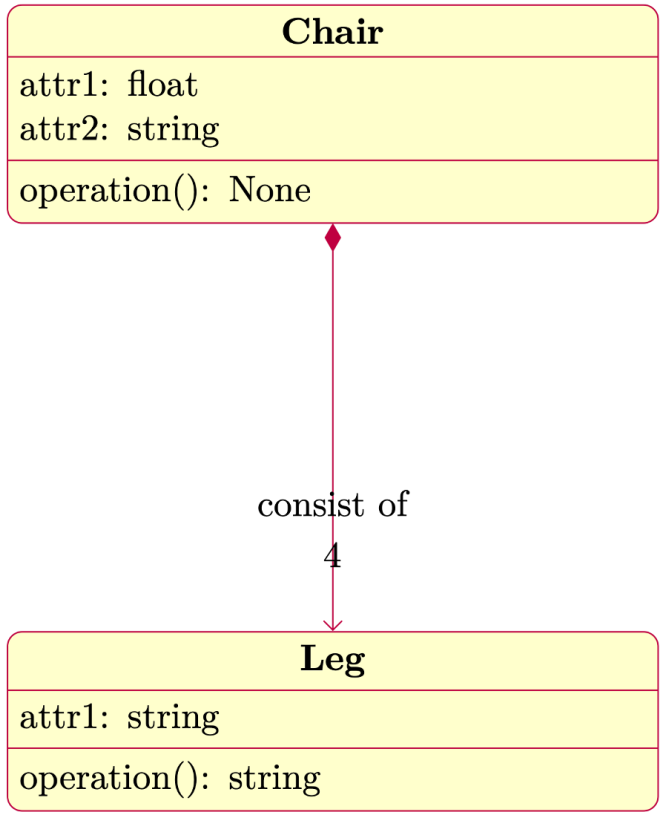
```
1  class Parent:
2      def __init__(self, child):
3          self._child = child
4          self._child.setParent(self)
5
6  class Child:
7      def __init__(self):
8          self._parent = None
9
10     def setParent(self, parent):
11         self._parent = parent
```

- Beide Klassen 'wissen' voneinander und halten eine Referenz
- Man kann also sowohl vom Parent zum Child, als auch von Child zum Parent

Komposition – die „ist-Teil-von“-Beziehung

- Sonderform der Assoziation
- Eine 'ist Teil von'-Beziehung
- Beispiele:
- Ein Stuhl hat 4 Beine. Entsorgt man den Stuhl, entsorgt man auch die Beine

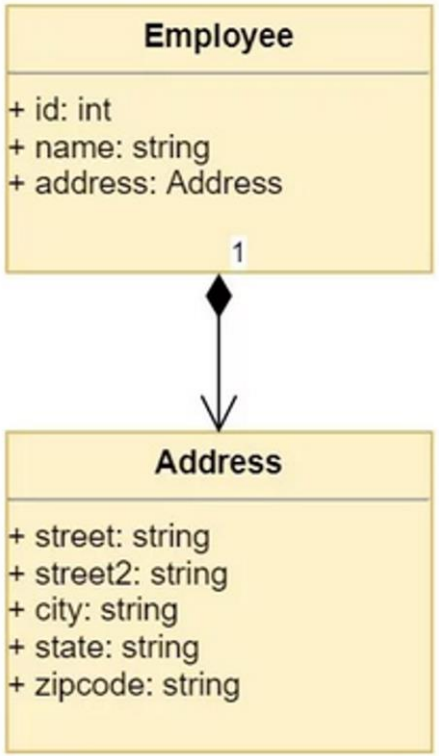
Komposition in UML



- Eine gefüllte Raute markiert den Halter der Komposition
- Der Pfeil zeigt auf die Teile

KOMPOSITION -BEISPIEL

Komposition in UML und Python



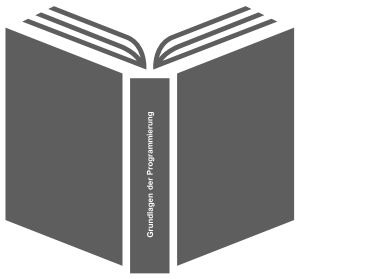

```
class Employee:
    def __init__(self, id, name, street, city, state,
                                                zipcode):
        self.id = id
        self.name = name
        self.address = Address(street, city, state, zipcode)
                    #Klasse wird hier aufgerufen

class Address:
    def __init__(self, street, city, state, zipcode):
        self.street = street
        self.city = city
        self.state = state
        self.zipcode = zipcode

    def print_address(self):
       print(self.street)

E1 = Employee(1,'Max',Mustermann', 'Trier', 'RLP`, '54290')
E1.address.print_address()
```

Aggregation – die „hat“-Beziehung

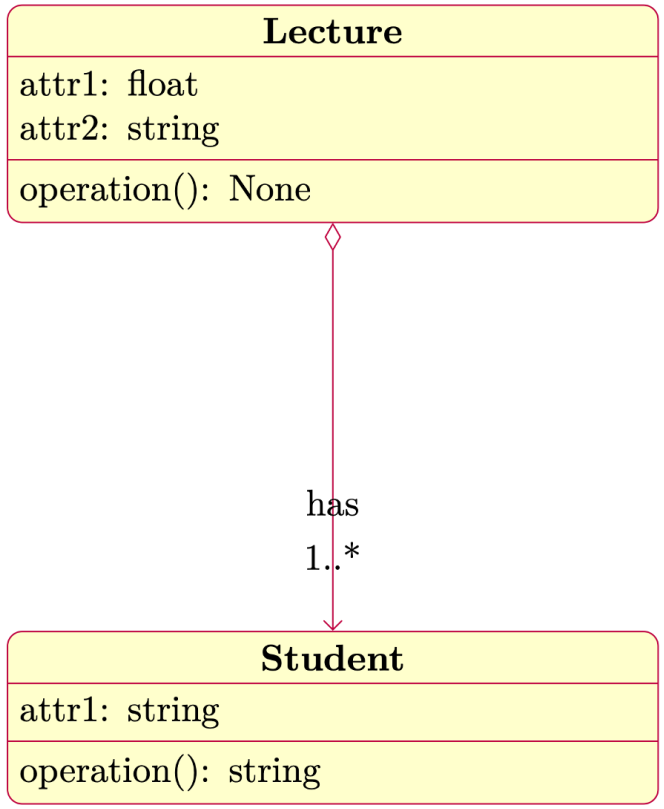


Studierende gibt es auch ohne Vorlesung

- Sonderform der Assoziation
- Eine 'hat'-Beziehung
- Beispiel: Vorlesung wird für viele Studierende gehalten. Wird die Vorlesung entfernt, hat es keine Auswirkung auf die Studierenden

AGGREGATION

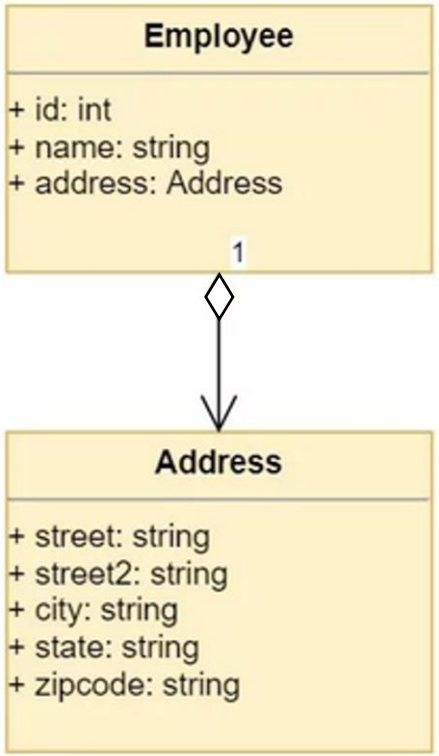
Aggregation in UML



- Eine leere Raute markiert den Halter der Aggregation
- Der Pfeil zeigt auf die Teile

AGGREGATION - BEISPIEL

Aggregation in UML und Python




```python
class Employee:
    def __init__(self, id, name, address):
        self.id = id
        self.name = name
        self.address = address #Objekt wird übergeben

class Address:
    def __init__(self, street, city, state, zipcode):
        self.street = street
        self.city = city
        self.state = state
        self.zipcode = zipcode

    def print_address(self):
       print(self.street)

address1 = Address('Musterstraße', 'Trier', 'RLP', '54294')
E1 = Employee(1, 'Max', address1) #Objekt wird hier übergeben
E1.address.print_address()
```

GENERALISIERUNG / SPEZIALISIERUNG

Generalisierung - die “ist-ein“ Beziehung

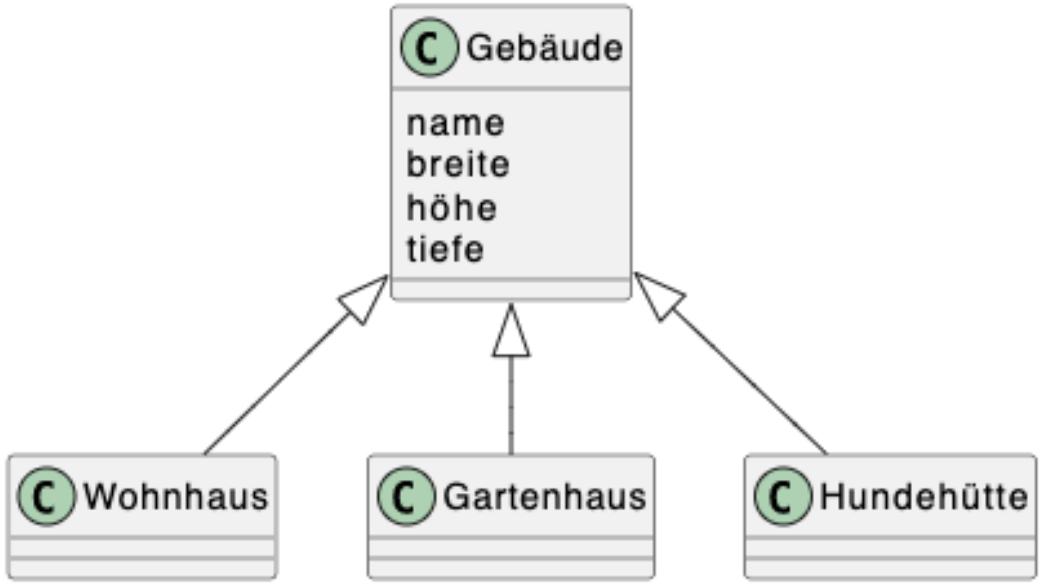
Rechteck

Besonderes Rechteck

- Vererbung drückt eine '**ist ein**' Beziehung aus
- Eine Oberklasse definiert bestimmte Attribute und Methoden
- Eine Unterklasse verfeinert diese Attribute und Methoden („Vererbung“)

GENERALISIERUNG / SPEZIALISIERUNG IN UML

Generalisierung in UML



- Dargestellt durch ein leeres Dreieck am Kopf der Verbindung
- Gelesen:
  - Gebäude generalisiert Wohnhaus, Gartenhaus und Hundehütte
  - Wohnhaus, Gartenhaus und Hundehütte spezialisieren Gebäude

GENERALISIERUNG / SPEZIALISIERUNG - BEISPIEL

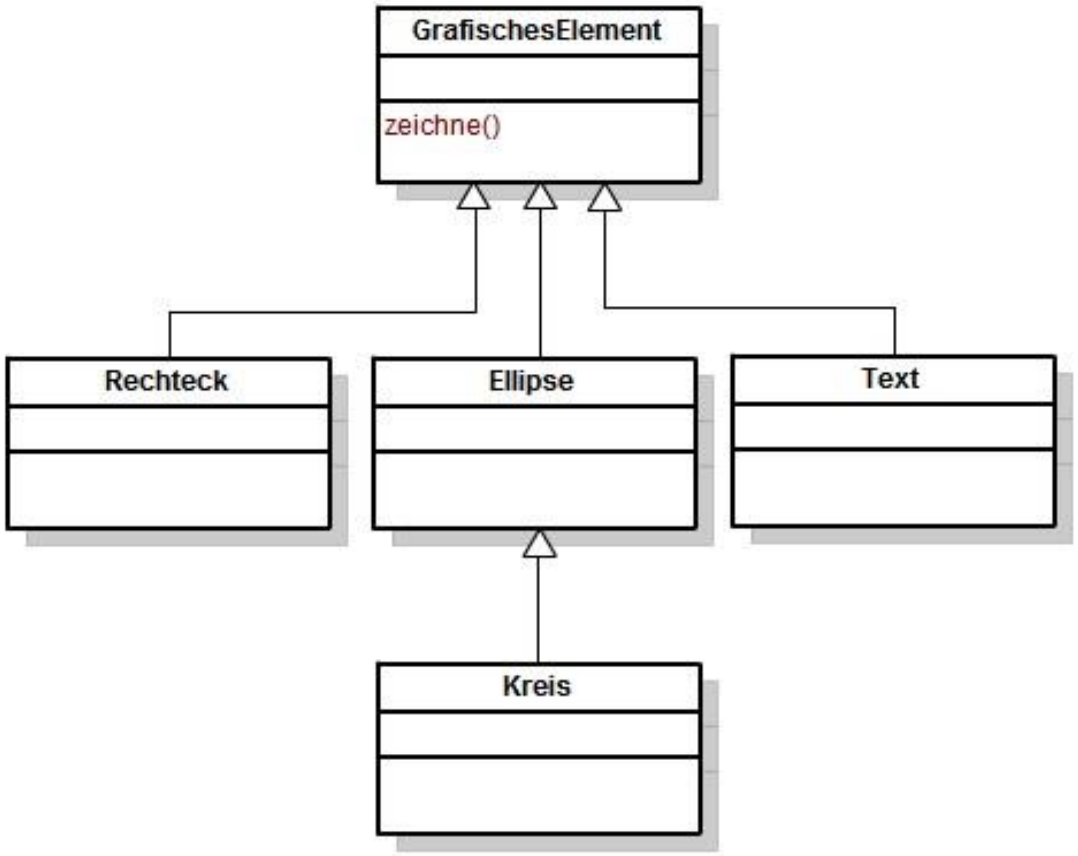
Generalisierung in UML und Python

- Oberklasse wird in runden Klammern nach dem Klassennamen angegeben
- Falls die Unterklasse keine eigene `__init__` definiert, wird die der Oberklasse aufgerufen

```
1 class animal:
2     def __init__(self):
3         self._legs = 4
4
5 class dog(animal):
6     pass
7
8 rex = dog()
9 rex._legs # 4
```

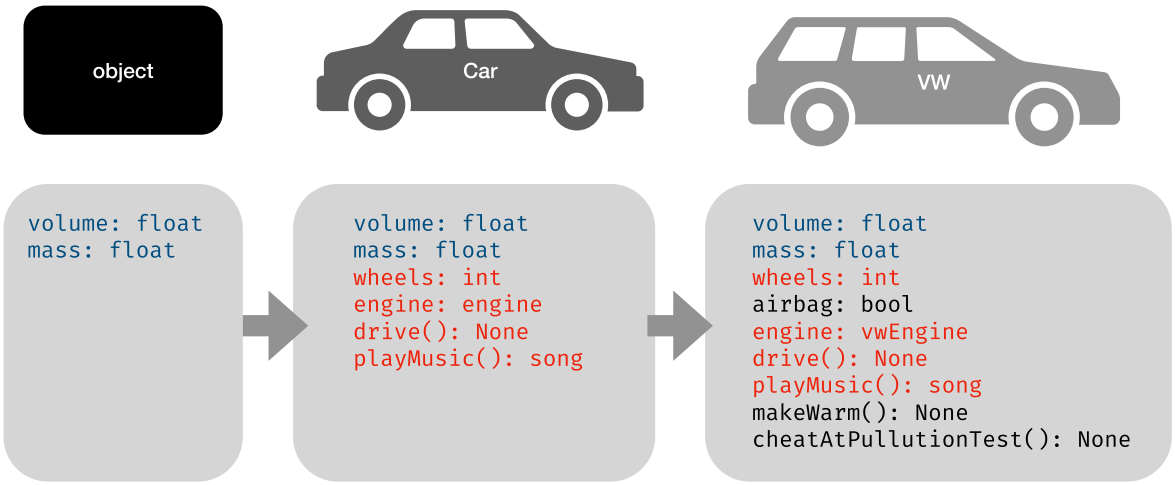
GENERALISIERUNG / SPEZIALISIERUNG IN UML

Beispiel



GENERALISIERUNG / SPEZIALISIERUNG

Wiederverwendung von Code / Vererbung



- Es werden nur einmal Attribute und Methoden definiert und diese dann weitergegeben
- *Alle* Attribute und Methoden einer Klasse werden vererbt
- Jeder Erbe kann dann entscheiden, ob die bisherige Implementierung für ihn passt
- Die erbende Klasse kann beliebig eigene Attribute und Methoden definieren
- Das ist Sinnvoll, da die erbende Klasse spezialisiert und unter Umständen mehr 'kann' als die Vater-Klasse

ÜBERSCHREIBEN VON METHODEN UND ATTRIBUTEN

Override

- Geänderte Methoden und Attribute werden bevorzugt
- Da `private` und `public` in Python nur Konvention ist, kann man theoretisch alles ändern

```
1  class animal:
2      ...
3
4  class dog(animal):
5      def walk(self):
6          print(self._legs*'taps, ')
7
8
9  rex = dog()
10 rex.walk()
11 # taps, taps, taps, taps,
```

ÜBERSCHREIBEN DER __INIT_ METHODE DER SUPERKLASSE

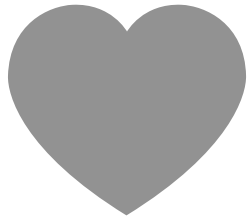
Override

- In einigen Szenarien soll die `__init__` der Oberklasse weiterhin genutzt werden
- Allerdings verhindert die eigen definierte `__init__` der Unterklasse den automatischen Aufruf
- Durch `super()` kann man auf die Oberklasse zugreifen und deren `__init__` nutzen

```
1  class animal:
2      def __init__(self):
3          self._noise = '...'
4          self._legs = 4
5
6
7  class dog(animal):
8      def __init__(self):
9          super().__init__()
10         self._noise = 'wau'
11
12
13 rex = dog()
14 print(rex._noise)
15 # wau
```

Vielfachvererbung

- Eine einzelne Vererbung generalisiert nicht ausreichend
- Beispiel:
  - Ein VW ist ein Auto
  - Ein VW ist auch ein Fortbewegungsmittel und nicht alle Fortbewegungsmittel sind Autos



Oberklasse A

Oberklasse B

Unterklasse A

Vielfachvererbung

- Statt einer Klasse bekommt die Unterklasse zwei Klassen in die runden Klammern
- Reihenfolge ist dabei wichtig

```
1  class rhino:
2      def __init__(self):
3          self._horn = True
4
5  class horse:
6      def __init__(self):
7          self._legs = 4
8
9  class unicorn(horse, rhino):
10     pass
11
12 amalthea = unicorn()
13 print('It has', amalthea._legs,
14                           'legs!')
15 # It has 4 legs!
```

Vielfachvererbung in Python

- Die Superklasse oder dominante Klasse der Vererbung ist immer die erste Klasse in den runden Klammern
- Diese Klasse wird durch super() verfügbar gemacht

```
1  class rhino:
2      def __init__(self):
3          self._horn = True
4
5  class horse:
6      def __init__(self):
7          self._legs = 4
8
9  class unicorn(horse, rhino):
10     def __init__(self):
11         super().__init__()
12
13 amalthea = unicorn()
14 print('It has', amalthea._legs,
15                           'legs!')
16 # It has 4 legs!
```

Priorität der __init__

- Falls keine `__init__` von der Unterklasse implementiert wird, wird die `__init__` von der Superklasse verwendet
- Dadurch entfallen die `__init__` der anderen Eltern-Klassen und damit gegebenenfalls Attribute

```
1  class rhino:
2      ...
3
4  class horse:
5      ...
6
7  class unicorn(horse, rhino):
8      def __init__(self):
9          super().__init__()
10
11 amalthea = unicorn()
12 print('Has it a horn?',
13               amalthea._horn)
14 # Error
```

Konstruktoren der Nicht-Superklasse(n) nutzen

- Damit keinerlei Attribute entfallen, können die `__init__` Methoden der verbleiben Eltern-Klassen separat aufgerufen werden
- Bei diesem Aufruf greift allerdings nicht der Automatismus bisheriger Instanziierungen
- Das bedeutet, es muss der Klassenname genutzt werden, um die `__init__` aufzurufen und man muss selbst die Instanz übergeben

```
1  class rhino:
2      ...
3
4  class horse:
5      ...
6
7  class unicorn(horse, rhino):
8      def __init__(self):
9          super().__init__()
10         rhino.__init__(self)
11
12 amalthea = unicorn()
13 print('Has it a horn?',
14               amalthea._horn)
15 # Has it a horn? True
```