Grundlagen der Programmierung

Vom Algorithmus zum Programm:
Objektorientierte Programmierung mit Python

Objekte

- **Im Problembereich** (in Anwendungsdomänen)**:**
  - *Leitgedanke: Die Welt besteht aus Objekten.*
  - Fahrzeuge, Häuser, Hotels, Menschen, Tiere, ..., Lehrveranstaltungen, Algorithmen, ...
  - besitzen Eigenschaften und Verhalten
  - interagieren

- **Im Lösungsbereich** (in Programmiersprachen)**:**
  - Abstrakte Repräsentation der realen Objekte im Programm
  - Dynamik durch Interaktion von Objekten

Klassen

- Zusammenfassung gleichartiger Objekte zu Klassen (Fahrzeuge, Häuser, Hotels, Menschen, ...)
- Festlegung von
  - **Klassennamen** (Fahrzeug, Haus, Hotel, Mensch)
  - **Eigenschaften** (Geschwindigkeit *bzw.* Anzahl der Zimmer)
  - **Verhalten** (beschleunigen *bzw.* Zimmer buchen)
- **Klassen** repräsentieren Mengen aller Objekte, die durch die gleichen Eigenschaften gekennzeichnet sind und die das gleiche Verhalten zeigen können.

Klassen als Mengen von Objekten

Fahrzeug

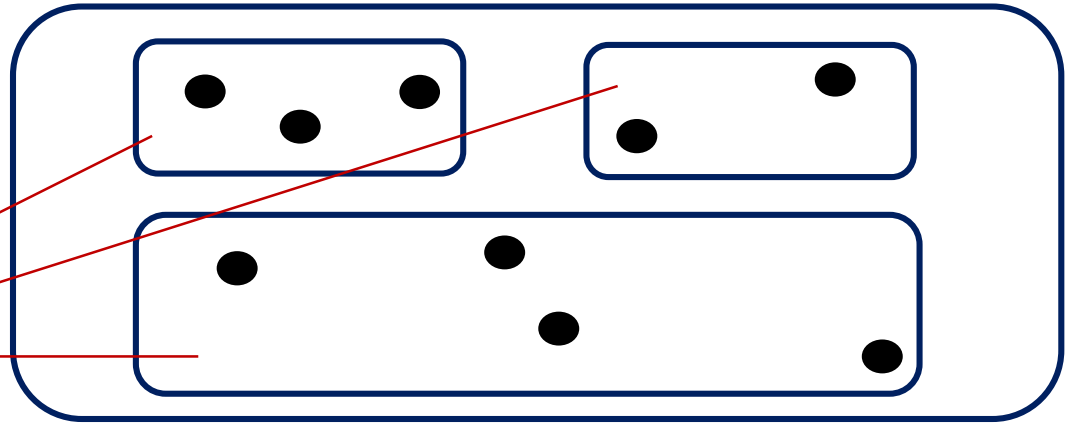
Haus

→ Objekte sind **Instanzen** der Klasse

Klassen und Mengenpartitionen

- Sei M eine nicht leere Menge.
  Eine *Zerlegung/Partition* von M ist eine Teilmenge Z von $\mathcal{P}(M)$ mit folgenden Eigenschaften:

1. $\emptyset \notin Z$
2. Wenn $N_1 \in Z$ und $N_2 \in Z$ und $N_1 \neq N_2$, dann $N_1 \cap N_2 = \emptyset$.
3. $M = \bigcup_{N \in Z} N$



Mengenpartitionen und Äquivalenzrelationen

- Jede Äquivalenzrelation $\sim$ in einer Menge M definiert eindeutig eine Zerlegung:
  $Z/\sim = \{ \{ x \in M \mid x \sim a \} \mid a \in M \}$

- Umgekehrt definiert jede Zerlegung Z einer Menge M eindeutig eine Äquivalenzrelation in M:
  $x \sim_Z y$ **gdw.** $x \in K$ **und** $y \in K$ **für ein** $K \in Z$.

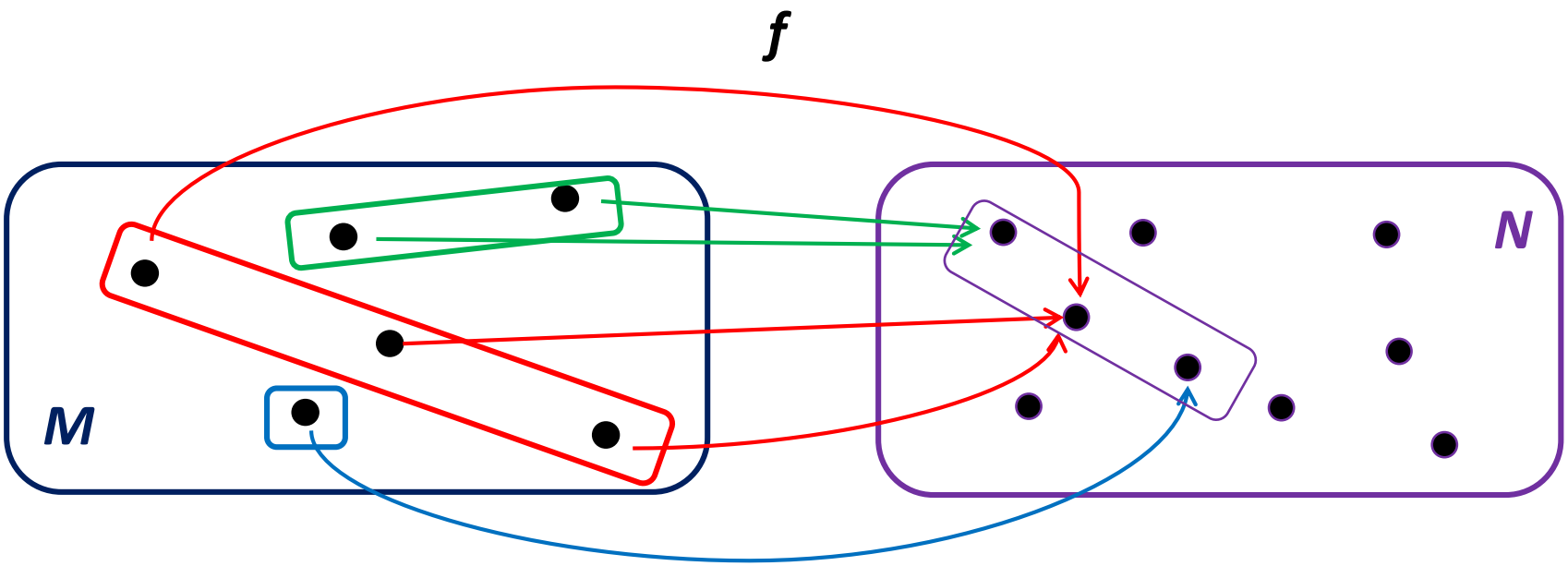
Kern einer Funktion und Faktorisierung

- Sei $f : M \longrightarrow N$ eine Funktion. Die **Kern-Relation** der Funktion f ist die Äquivalenzrelation **ker f** mit

  $$x_1 \sim x_2 \text{ gdw. } f(x_1) = f(x_2) .$$

- Sei $f : M \longrightarrow N$ eine Funktion. Die zu f gehörige **Faktorisierung/Quotientenmenge** ist die Zerlegung $M/\ker f$ der Menge M.

Faktorisierung



- Es gibt eine bijektive Funktion von $M/\ker f$ auf $W(f)$.

$$f: M \longrightarrow W(f)$$
$$\downarrow \quad \nearrow$$
$$M/\ker f$$

Eigenschaften (Attribute) von Objekten

- Objekte einer Klasse unterscheiden sich durch die Werte ihrer Attribute.
  - Fahrzeug: Geschwindigkeit, Typ, ...
  - Hotel: Anzahl der Zimmer, der belegten Zimmer, ...
- Attributwerte bestimmen **Zustand** der Objekte.

- *Im Programm*:
  Werte von **Instanzvariablen / Datenelementen**

Verhalten von Objekten

- ausgelöst durch **Nachrichten** an das Objekt
- Ändert oft den Objektzustand, also die Attributwerte
- kann vom Zustand des Objekts abhängen

- *Im Programm:* **Methode (Funktion)**
  - Bezeichner der Funktion ist Nachricht
  - Implementierung der Funktion ist das Verhalten (die Reaktion des Objekts)

Beispiel: Fahrzeug

- Klasse: Vehicle
- Eigenschaften:
  1. type: (*z.B.*) Diesellok
  2. speed: (*z.B.*) 0 km/h
- Methoden:
  1. accelerate (um ... km/h)
  2. signal (Ausgabe eines Signaltons ... als String)

Zusammenfassung

- **Im Problembereich:**
  - *Leitgedanke: Die Welt besteht aus Objekten.*
  - Fahrzeuge, Häuser, Hotels, Menschen, Tiere, ..., Algorithmen, ...
  - besitzen Eigenschaften und Verhalten

- **Im Lösungsbereich** (in Programmiersprachen):
  - Abstrakte Repräsentation der realen Objekte im Programm
  - Eigenschaften $\rightarrow$ Werte von **Datenelementen** (Variablen)
  - Verhalten $\rightarrow$ **Methoden** (Funktionen)

Fahrzeug in Python

```
class Vehicle:
    pass
```

eine erste Klassendefinition
(ohne Attribute und Methoden)

```
vehicle_1 = Vehicle()
```

Erzeugen eines Objekts der Klasse

```
vehicle_1.type = "Pkw"
vehicle_1.speed = 0
```

Anlegen von Attributen mit initialen Werten für ein Objekt

Bindung der Datenelemente an Objekte

```
vehicle_1 = Vehicle()
vehicle_1.type = "Pkw"
vehicle_1.speed = 0
```

```
print(vehicle_1.type, vehicle_1.speed)   # Pkw 0
print(speed)                             # Fehler
vehicle_2 = Vehicle()
print(vehicle_2.type, vehicle_2.speed)   # Fehler
```

Konstruktoren

- werden in der Klasse wie eine spezielle Funktion definiert:

```
__init__(self, ...)
```

  dienen zur Festlegung von Aktionen bei der Objekterzeugung (*meist zur Definition von Datenelementen*)

- z.B. für Klasse **`Vehicle`**:

```
def __init__(self,bez,ge):
    self.type = bez
    self.speed = ge
```

- Der Parameter `self` steht für das Objekt, das gerade erzeugt werden soll.

Fahrzeug in Python (2)

Klassendefinition mit Konstruktor

```
class Vehicle:
      def __init__(self, bez, ge):
            self.type = bez
            self.speed = ge
```

Erzeugen eines Objekts der Klasse

```
vehicle_1 = Vehicle()                  # Fehler
# Den parameterlosen Standardkonstruktor
# gibt es nur, wenn kein __init__ definiert ist
```

Fahrzeug in Python (2)

Klassendefinition mit Konstruktor

```
class Vehicle:
      def __init__(self, bez, ge):
            self.type = bez
            self.speed = ge
```

Erzeugen eines Objekts der Klasse

```
vehicle_1 = Vehicle("Pkw", 0)
print(vehicle_1.type, vehicle_1.speed)   # Pkw 0
```

- Aktuelle Parameter initialisieren Datenelemente für ein Objekt
- Kein aktueller Parameter für `self`

Fahrzeug in Python (2)

- Es ist jetzt garantiert, dass alle Objekte diese Attribute haben.
- Werte der Datenelemente machen die Objekte unterscheidbar.

```
vehicle_1 = Vehicle("Pkw", 0)
vehicle_2 = Vehicle("Lkw", 45)

print(vehicle_1.type, vehicle_1.speed)   # Pkw 0
print(vehicle_2.type, vehicle_2.speed)   # Lkw 45
```

- Existenz der Datenelemente erlaubt die Definition von Methoden, die auf die Datenelemente zugreifen.

Methoden der Klasse Fahrzeug

- Eigenschaften:
  1. type
  2. speed
- Methoden:
  1. accelerate (um ... km/h)
  2. signal

Python:

```
class Vehicle:
      def __init__(...)
            ...

      def accelerate(self,g):
            self.speed += g

      def signal(self):
            print(self.type,
              ": tuuut")
```

Parameter `self`

- Methoden haben in Python immer mindestens einen formalen Parameter.
- Er steht für das Objekt, für das die Methode aufgerufen wird.
- Der Name kann frei gewählt werden.
  Konvention: **`self`**
- *Beispiel:*

```
self.speed    # Wert der Instanzvariablen
              # speed vom aktuellen Objekt
```

Methodenaufrufe für Objekte

```
lok = Vehicle("Lok", 0)      # ein Objekt erzeugen
                             # mit speed = 0

lok.accelerate(20)           # Nachricht an lok:
                             # Beschleunige um 20 km/h
                             # speed von lok jetzt 20
```

Kein aktueller Parameter für **`self`** !

Zusammenfassung
Benutzen von Objekten (Syntax)

- `lok = Vehicle("Lok", 0)  # Erzeugen eines Objekts`

  → **Klassenname(*Parameterliste*)**

- `lok.accelerate(20)  # Aufruf einer Methode`

  → **Objektname.Methodenname(*Parameterliste*)**

- `lok.speed += 5  # Zugriff auf ein Datenelement`

  → **Objektname.Datenelement**

  ➢ *vorzugsweise in Methodendefinitionen:*

```
self.speed += g      # in der Definition der Methode
                     # accelerate(self,g)
```

Objekte ausgeben

- *Beispiel Liste:*

```
L = [1,2,3]
print(L)          # [1,2,3]
```

- *Beispiel Vehicle:*

```
lok = Vehicle("Lok", 0)
print(lok)        # Vehicle object at 0x0043D70
```

-

```
def __str__ (self):
    return self.type  + ": "
                      + str(self.speed) + " km/h"
```

OO-Programm

```
lok = Vehicle("Diesellok",0)
vw = Vehicle("Golf",60)
print(lok)
print(vw)
vw.signal()
lok.accelerate(20)
vw.accelerate(-20)
print(lok)
print(vw)
```

Ausgabe:
Diesellok: 0 km/h
Golf: 60 km/h
Golf: tuuut
Diesellok: 20 km/h
Golf: 40 km/h

Klassendefinition und Programm

- Klassendefinition ist reine Sammlung von Variablen- und Funktionsdefinitionen

➢ *Eine Klasse allein „tut nichts“.*

- Objektorientiertes Programm:
  - Erzeugen von Objekten von Klassen
  - Aufruf von Methoden der Objekte (auch gegenseitig → *Interaktion von Objekten*)
  - *eventuell* etwas Ausführungslogik „drum herum“

OO-Programmierung: weitere Features

→ **Praxis der Programmierung**

- **Destruktoren** (in einigen Sprachen, z.B. in Python)
  - werden zum Ende der Lebensdauer aufgerufen
  - werden genutzt, um „aufzuräumen“, z.B. nicht mehr benötigte Objekte zerstören, Dateien zu schließen etc.
  - `__del__(self)`
- **Vererbung**
  - von definierten Klassen abgeleitete/spezialisierte Klassen
  - *Beispiel:* Fahrzeug → Auto *oder* Haus → Hotel
  - **erben** alle Datenelemente und Methoden
  - können weitere Datenelemente und Methoden definieren
  - Objekte bilden Teilmenge der Elternklasse

Wo uns Objekte schon begegneten ...

- Alle Daten werden in Python als Objekte gespeichert.

- Für viele vordefinierte Objekttypen existieren Methoden, *z.B.* für eine Liste `L`

```
L.append(x)
L.remove(x)
L.pop()
L.insert(n,x)
```

Queue als selbst definierte Klasse

```
class Queue:
    def __init __(self):
        self.Q = []

    def __str __(self):
        return str(self.Q)

    def enqueue(self,new):
        self.Q.insert(0,new)

    def dequeue(self):
        return self.Q.pop()
```

```
qu = Queue()
qu.enqueue(1)
qu.enqueue(2)

print(qu) # [2,1]

qu.dequeue()
qu.enqueue(3)

print(qu) # [3,2]

qu.dequeue()
print(qu) # [3]
```

Vorteile OO-Programmierung

- Besonderheiten der Programmiersprachen können versteckt werden
  - klare, einfache **Schnittstellen**
  - einfache Wiederverwendbarkeit
- Nähe zum Problembereich
  - *Software, die tut, was Anwender erwarten* **(valide Software)**
- Vorteile offenbaren sich am besten bei großen Softwareprojekten

  → Software Engineering