Objektorientierte Programmierung

OOP

bisher: Programm = Folge von Anweisungen,
was mit Daten zu tun ist

Entwurf:

- Zerlegung in Teilprozesse
- Festlegung der Datenstrukturen

OOP

Entwurf formal z.B. mit:

- Flussdiagramme
- Datenflussdiagramme
- Pseudocode

OOP

OOP grundlegend anders schon im Entwurf

- orientiert an Objekten (aus Anwendungsgebiet)
- keine Trennung von Prozessen und Datenstrukturen
- enge Verbindung von Prozessen und zugehörigen Datenstrukturen im Objekt
- "Kopf" Daten, Prozesse Anhängsel

OOP

Erste objektorientierte Programmiersprachen:

- Simula67 (class, this)
- Smalltalk71 / Smalltalk80

OOP

Erste objektorientierte Programmiersprachen:

- Simula67 (class, this)
- Smalltalk71 / Smalltalk80

heute oft hybrid, wie Python

OOP

Rein objektorientierter Entwurf:

- kein linearer Ablauf
- alles ist Kommunikation zwischen Objekten
- Objekte tauschen Nachrichten aus

OOP

Rein objektorientierter Entwurf:

„1. Alles ist ein Objekt, 2. Objekte kommunizieren durch das Senden und Empfangen von Nachrichten (welche aus Objekten bestehen), 3. Objekte haben ihren eigenen Speicher (strukturiert als Objekte), 4. Jedes Objekt ist die Instanz einer Klasse (welche ein Objekt sein muss), 5. Die Klasse beinhaltet das Verhalten aller ihrer Instanzen (in der Form von Objekten in einer Programmliste), 6. Um eine Programmliste auszuführen, wird die Ausführungskontrolle dem ersten Objekt gegeben und das Verbleibende als dessen Nachricht behandelt“

– Alan Kay: The Early History of Smalltalk (1993)[1]

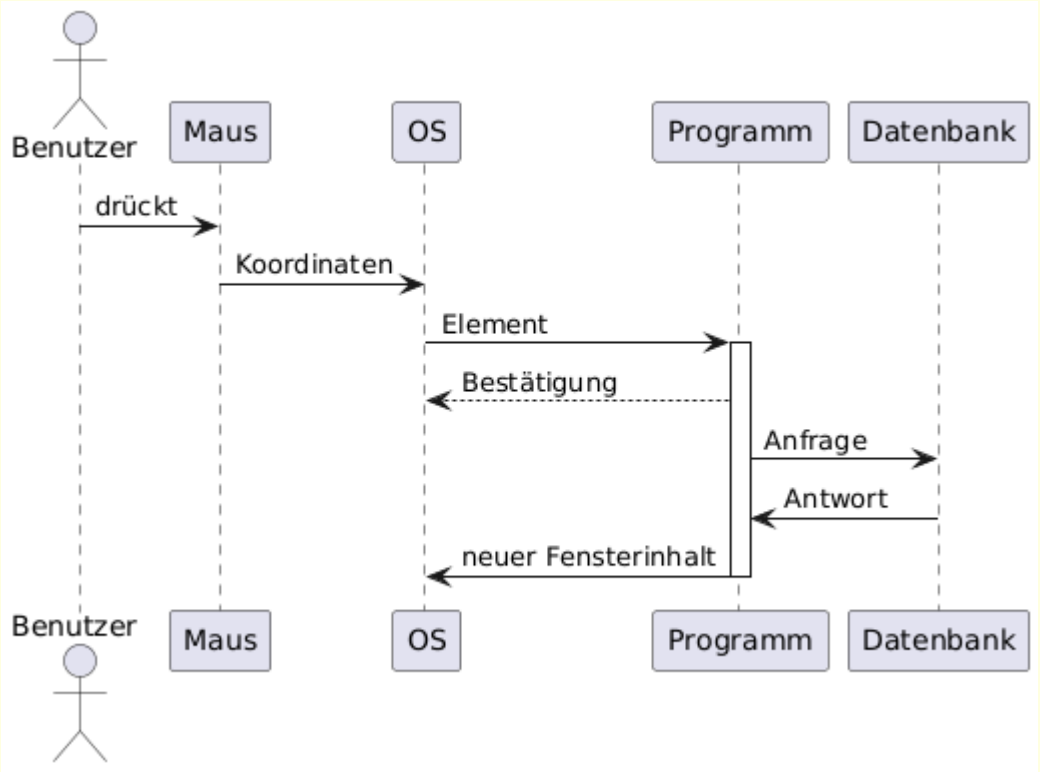
OOP

OOP-Entwurf häufig mit UML

- Unified Modeling Language
- kein einheitlicher Entwurf
- spezifiziert 14 Diagrammtypen
- verschiedene Diagramme modellieren verschiedene Aspekte

OOP

UML- Sequenzdiagramm modelliert Kommunikation



OOP

Wie sehen
Objekte in Python aus?

OOP

bisherige Beispiele:

- komplexe Datenstrukturen

- Daten + zugehörige Operationen
- z.B: Liste + append, pop…
- Methoden nur für Liste aufrufbar: li.pop()

OOP

Wie wird Objekt definiert?

- Schlüsselwort class
- Name und Doppelpunkt
- eingerückter Körper

```
1  class erstes_objekt:
2      pass
```

OOP

Wie sieht eine Methode aus?

- wie Funktion, aber innerhalb class
- erster Parameter: Instanz des Aufrufs, self

```
1  class erstes_objekt:
2      def erste_methode(self, text):
3          print(text)
```

OOP

Verwendung

- Instanziierung der Klasse, mit Klammern
- Aufruf der Methode, ohne erstes Argument

```
1 class erstes_objekt:
2     def erste_methode(self, text):
3         print(text)
4
5 beispiel_objekt = erstes_objekt()
6
7 beispiel_objekt.erste_methode("Ausgabe")
```

OOP

Verwendung

- Zugriff auf Methoden, Felder:

  Objekt – Punkt – Element

```
1 class erstes_objekt:
2     def erste_methode(self, text):
3         print(text)
4
5 beispiel_objekt = erstes_objekt()
6
7 beispiel_objekt.erste_methode("Ausgabe")
```

OOP

Klasse vs. Objekt

- Definition vs. erzeugte Instanz
- Methode vs. Aufruf
- z.B. list vs. konkrete Liste von int-Werten

OOP

Eine besondere Methode: der Konstruktor

- automatisch aufgerufen bei Erzeugung
- bereitet das Objekt vor für Benutzung
  - Bsp. Liste: legt Speicherort fest
  - optional: fügt erste Werte ein (li = [1,2,3])

OOP

Eine besondere Methode: der Konstruktor

- Name: __init__
- wird bei Erzeugung automatisch aufgerufen

```
1  class erstes_objekt:
2      def __init__(self):
3          print("Es wird konstruiert")
4
5      def erste_methode(self, text):
6          print(text)
7
8  beispiel_objekt = erstes_objekt()
9  beispiel_objekt.erste_methode("Ausgabe")
```

OOP

Eine besondere Methode: der Konstruktor

- Attribute hier definiert
- durch Zuweisung

```
class erstes_objekt:
    def __init__(self, parameter):
        self.def_wert = 10
        self.eing_wert = parameter

beispiel_objekt = erstes_objekt(45)
print( str(beispiel_objekt.def_wert ) , " und ",
      str(beispiel_objekt.eing_wert) )
```

```
10  und  45
```

OOP

Attribute auch in anderen Methoden definierbar

- existieren nicht bis Ausführung der Methode
- durch Zuweisung

```
1 class erstes_objekt:
2     def __init__(self, parameter):
3         self.eing_wert = parameter
4     def neue_methode(self, x):
5         self.met_wert = x
6
7 beispiel_objekt = erstes_objekt(45)
8 beispiel_objekt.neue_methode(33)
9 print(beispiel_objekt.met_wert)
```

```
33
```

OOP

Attribute auch in anderen Methoden definierbar

- ohne self: lokale Variable
- nicht außerhalb Methode

```
1 class erstes_objekt:
2     def __init__(self, parameter):
3         self.eing_wert = parameter
4     def neue_methode(self, x):
5         met_wert = x
6
7 beispiel_objekt = erstes_objekt(45)
8 beispiel_objekt.neue_methode(33)
9 print(beispiel_objekt.met_wert)
```

```
Fehler
```

OOP

Methoden ohne self:

- lokale Funktionen
- nicht aufrufbar von außen

```
1 class erstes_objekt:
2     def __init__(self, parameter):
3         self.eing_wert = parameter
4     def neue_methode(x):
5         met_wert = x
6
7 beispiel_objekt = erstes_objekt(45)
8 beispiel_objekt.neue_methode(33)
```

```
TypeError:
erstes_objekt.neue_me
thode() takes 1
positional argument
but 2 were given
```

OOP

Direkter Zugriff auf Attribute nicht empfohlen

- besser über Methoden

```
1  class erstes_objekt:
2
3      def __init__(self, parameter):
4          self.eing_wert = parameter
5
6      def liefere_eing_wert(self):
7          return self.eing_wert
8
9      def setze_eing_wert(self, neuer_wert):
10         self.eing_wert = neuer_wert
```

OOP

Erstellen Sie eine Klasse ***Hund*** mit

- einem Konstruktor mit einem Parameter, der dem Attribut *name* zugewiesen wird
- einer Methode sprich(), die "Wau!" zurückliefert
- einer Methode friss(etwas), die True zurückgibt falls etwas gleich "Fleisch" ist, sonst False