



Object-Oriented Analysis / Design (OOAD)



- Angemessene Weise, ein komplexes System zu modellieren
- Modelliere Software-System als Menge kooperierender Objekte
 - Programmverhalten bestimmt durch *Gruppenverhalten*
 - Entsteht aus Verhalten einzelner Objekte
- Objekte werden *antropomorph* betrachtet
 - Jedes hat gewisse "Intelligenz" (Auto kann selbst fahren, Tür kann sich selbst öffnen, ...)
 - Trigger dazu muß von außen kommen
- Jedes Objekt ist "black box":
 - Versteckt Details
 - Erleichtert die Entwicklung / Wartung eines komplexen Systems



Was ist ein Objekt?



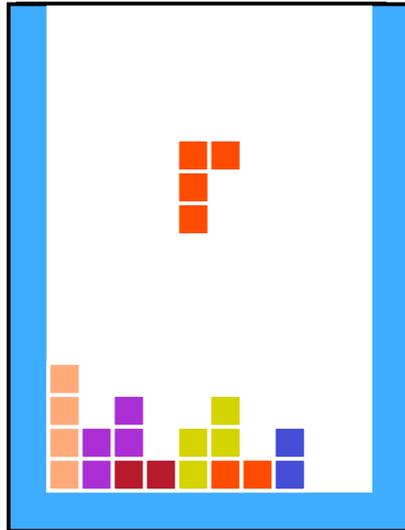
- Typische Kandidaten für Objekte:
 - **Dinge**: Stift, Buch, Mensch
 - **Rollen**: Autor, Leser, Benutzerhandbuch
 - **Ereignisse**: Fehler, Autopanne
 - **Aktionen** (manchmal!): Telefongespräch, Meeting
- Keine Objekte sind:
 - Algorithmen (z.B. Sortieren),
- Ein Objekt hat
 - eine **Struktur**
 - einen **Zustand** (= interne "objekt-eigene" Variablen)
 - ein **Verhalten / Fähigkeiten** (= **Methoden**)
 - eine **Identität** (= Nummer, Zeiger, ...)



Beispiel: Tetris



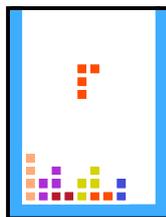
- Was sind die Objekte?
- Was müssen die Objekte können?
- Welche Eigenschaften haben die Objekte?



- Objekte:
 - Brett, Spielsteine

▪ Fähigkeiten:

- Steine:
 - Erzeugt werden
 - Fallen
 - Rotieren
 - Stoppen
- Brett:
 - Erzeugt werden
 - Zeilen löschen
 - Spielende feststellen



▪ Eigenschaften:

- Steine:
 - Orientierung
 - Position
 - Form
 - Farbe
- Brett:
 - Größe
 - Belegung der Zeilen



Definition von Klassen

- Allgemeine Form:

```
class name( object ) :  
    "documentation"  
    Anweisungen
```

- Anweisungen sind i.A. Methodendeklarationen von der Form:

```
def name(self, arg1, arg2, ...):  
    ...
```

- erster Parameter jeder Methode ist eine Referenz auf die aktuelle Instanz der Klasse
- Konvention: **self** !
- Ähnlich dem Keyword **this** in Java oder C++



Zugriff auf Instanzvariablen

- **self** muß man in der Methoden-Deklaration immer angeben, **nicht** aber im Aufruf:

```
def set_age(self, num):
```

```
x.set_age(23)
```

- Zugriff auf Instanzvariablen innerhalb einer Instanzmethode immer über **self**:

```
def set_age( self, num ) :  
    self.age = num
```

- Zugriff von außerhalb einer Instanzmethode über Name der Instanz selbst:

```
x.set_age( 23 )  
x.age = 17
```



Erzeugung von Instanzen



- Durch `ClassName()`
- Beispiel:

```
class Name:  
    ...  
x = Name()  
z = x
```

- Ordentliche Initialisierung:
 - Spezielle Methode `__init__`
 - Wird bei Erzeugung der Instanz aufgerufen
 - Zur Initialisierung der Instanzvariablen
- Beispiel:

```
class Name:  
    def __init__( self ):  
        self.var = 0
```



- `__init__` heißt **Konstruktor**:
- Kann, wie jede andere Funktion, beliebig viele Parameter nehmen zur Initialisierung einer neuen Instanz
- Beispiel:

```
class Atom:  
    def __init__( self, atno, x,y,z ):  
        self.atno = atno  
        self.position = (x,y,z)
```

- Es gibt nur diesen einen (in C++ kann man viele deklarieren)
 - Keine wesentliche Einschränkung, da man ja Default-Argumente und Key/Value-Parameter hat



Beispiel: Atom class



```
class Atom:
    """A class representing an atom."""
    def __init__(self,atno,x,y,z):
        self.atno = atno
        self.position = (x,y,z)
    def __repr__(self):          # overloads printing
        return '%d %10.4f %10.4f %10.4f' %
            (self.atno, self.position[0],
             self.position[1],self.position[2])

>>> at = Atom(6,0.0,1.0,2.0)
>>> print at                  # ruft __repr__ auf
6 0.0000 1.0000 2.0000
>>> at.atno                  # Zugriff auf ein Attribut
6
```



```
class Molecule:
    def __init__(self, name='Generic'):
        self.name = name
        self.atomlist = []

    def addatom(self,atom):
        self.atomlist.append(atom)

    def __repr__(self):
        str = 'Molecule named %s\n' % self.name
        str += 'Has %d atoms\n' % len(self.atomlist)
        for atom in self.atomlist:
            str += str(atom) + '\n'
        return str
```

```
>>> mol = Molecule('Water')
>>> at = Atom(8,0.,0.,0.)
>>> mol.addatom(at)
>>> mol.addatom( atom(1,0.0,0.0,1.0) )
>>> mol.addatom( atom(1,0.0,1.0,0.0) )
>>> print mol
Molecule named Water
Has 3 atoms
8  0.000 0.000 0.000
1  0.000 0.000 1.000
1  0.000 1.000 0.000
```

- Bemerkung: `__repr__` wird immer dann aufgerufen, wenn ein Objekt in einen lesbaren String umgewandelt werden soll (z.B. durch `print` oder `str()`)

Prof. Dr. G. Zachmann Informatik 1 - WS 05/06 Einführung in Python, Teil 2 60

Öffentliche (public) und private Daten

- Zur Zeit ist alles in `Atom/Molecule` öffentlich, so könnten wir etwas richtig Dummes machen wie

```
>>> at = Atom(6,0.0,0.0,0.0)
>>> at.position = 'Grape Jelly'
```

dies würde jede Funktion, die `at.position` benutzt, abbrechen
- Aus diesem Grund sollten wir `at.position` **schützen** und Zugriffsmethoden auf dessen Daten bieten
 - *Encapsulation* oder *Data Hiding*
 - Zugriffsmethoden sind "Getters" und "Setters"
- Leider: in Python existiert (noch) kein schöner Mechanismus dafür!
 - Mechanismus: Instanzvariablen, deren Name mit 2 Underscore beginnt, sind privat; Bsp.: `__a` , `__my_name`
 - Üblich ist die Konvention: prinzipiell keinen direkten Zugriff von außen

Prof. Dr. G. Zachmann Informatik 1 - WS 05/06 Einführung in Python, Teil 2 61



Klassen, die wie Arrays und Listen aussehen



- Überladen von `__getitem__(self, index)` damit die Klasse sich wie ein Array/Liste verhält, d.h., der Index-Operator def. ist:

```
class Molecule:
    def __getitem__(self, index):
        return self.atomlist[index]

>>> mol = Molecule('Water') # definiert wie vorhin
>>> for atom in mol:         # benutze wie eine Liste!
    print atom
>>> mol[0].translate(1.,1.,1.)
```

- Bestehende Operatoren in einer Klasse neu/anders zu definieren nennt man **Überladen** (*Overloading*)



Klassen, die wie Funktionen aussehen (Funktoeren)



- Überladen von `__call__(self, arg)` damit sich die Klasse wie eine Funktion verhält, m.a.W., damit der `()`-Operator für Instanzen definiert ist:

```
class gaussian:
    def __init__(self, exponent):
        self.exponent = exponent
    def __call__(self, arg):
        return math.exp(-self.exponent*arg*arg)

>>> func = gaussian(1.0)
>>> func(3.0)
0.0001234
```



Andere Dinge zum Überladen



- `__setitem__(self, index, value)`
 - Analogon zu `__getitem__` für Zuweisung der Form `a[index] = value`
- `__add__(self, other)`
 - Überlädt den "+" Operator: `molecule = molecule + atom`
- `__mul__(self, number)`
 - Überlädt den "*" Operator: `molecule = molecule * 3`
- `__del__(self)`
 - Überlädt den Standarddestruktor
 - Wird aufgerufen, wenn das Objekt nirgendwo im Programm mehr benötigt wird (keine Referenz darauf mehr existiert)