Benutzerdefinierte Typen III

Zwischenprüfung

- …ist kein Drama … man sollte nur mehr üben
- wie schon erwähnt
  - projecteuler.net → kostenlos, viele gute Übungen
  - leetcode → sollte reichen
  - edabit → als Quelle/Repo für Übungen (nicht als Plattform)

Dateien und Python

Methoden

- `write(str)`
- `readline()`
- `readlines()`
- `read()`
- `close()`

Exception

- `IOError`

```
#open file for write (overwrite if exists, create if not)
f = open("test.txt","w")
f.write("Test data\n")
f.close()
```

```
#open file for write (append if exist, create if not)
f = open("test.txt","a")
f.write("Test data line 2\n")
f.close()
```

```
#open for read
f = open("test.txt","r")
#read a line from the file
line = f.readline()
print line
f.close()
```

```
#open for read
f = open("test.txt","r")
#read a line from the file
line = f.readline().strip()
while line!="":
    print line
    line = f.readline().strip()
f.close()
```

```
#open for read
f = open("test.txt","r")
#read the entire content from the file
line = f.read()
print line
f.close()
```

Pickle

```
studenten = [('Peter', 123), ('Paula', 988), ('Freddi', 851)]

datei = open("studenten.dat", "w")
for student in studenten:
    s = str(student) + '\n'
    datei.write(s)
datei.close()

datei = open("studenten.dat", "r")
for z in datei:
    name, matnum = z.strip('()\n ').split(',')
    name = name.strip("\' ")
    matnum = matnum.strip()
    print "Name: %s  Matrikelnummer: %s" % (name, matnum)
datei.close()
```

konvertiert Objekte in einen Stream, damit sie gespeichert und erneut gelesen werden können

```
import pickle

studenten = [('Peter', 123), ('Paula', 988), ('Freddi', 851)]

datei = open("studenten.dat", "w")
pickle.dump(studenten, datei)
datei.close()

datei = open("studenten.dat", "r")
meine_studenten = pickle.load(datei)
datei.close()

print meine_studenten
```

Private Methoden

- wie bei Attributen

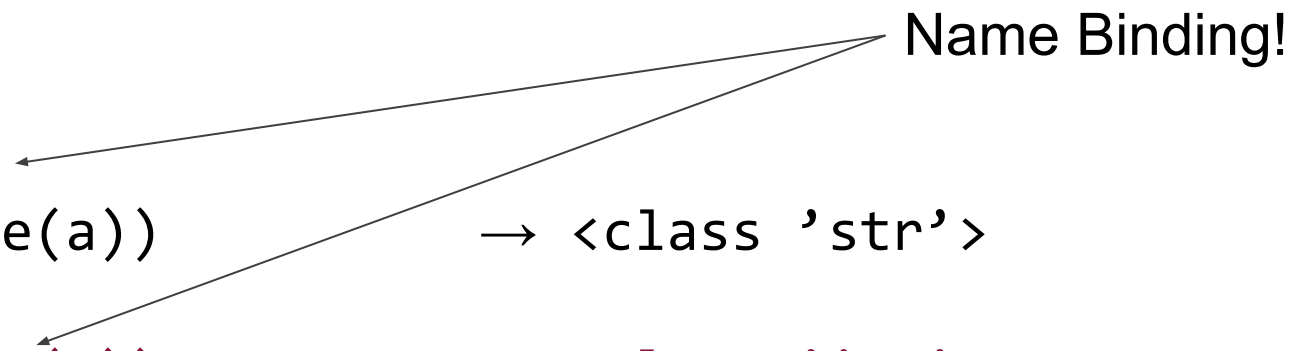
```
class Bruch:
   def __init__(self, n, m):
      self.n = n
      self.m = m

   def __ggT(self, a, b): pass
   def simplify(self):
      c = self.__ggT(self.n, self.m)
      return Bruch(self.n//c, self.m//c)

b = Bruch(2,4)
b.simplify()          → ok
b.__ggT()             → not ok
```

OOP

- named parameters
- Statische Elemente
- Operatoren, Funktionen und Hooks
- Sortieren
- Dataclass

Dynamischer vs. Statischer Typcheck

Dynamisch




```
a = "1"
print(type(a))          → <class 'str'>
a = 1
print(type(a))          → <class 'int'>
```

Statisch

```
std::string a = "1";
cout << typeid(variable).name();      → string
a = 1; //Fehler
```

Dynamischer vs. Statischer Typcheck

dynamische Typisierung “kompliziert” den Code

Welche Zeile führt zu einem Fehler ?

```
a = 1
b = a + 1
c = a + foo(2)
```

falls wie foo wie folgt definiert ist:

```
def foo(p):
    return p/2
```

```
die Lesbarkeit des Codes ist niedrig
```

Dynamische Typisierung

- “variable” definition: man muss Typeinfo nicht explizit angeben
- Typprüfungen werden hauptsächlich zur Laufzeit eines Programms durchgeführt

```
def f(a):
    a.a()
```

Dynamische Typisierung

- “variable” definition: man muss Typeinfo nicht explizit angeben
- Typprüfungen werden hauptsächlich zur Laufzeit eines Programms durchgeführt

```
def f(a):
    a.a()
print('HAHA')
f(10)
```

```
HAHA
Traceback (most recent call last):
  File "<string>", line 5, in <module>
File "<string>", line 2, in a
AttributeError: 'int' object has no attribute 'a'
```

Dynamische Typisierung

- “variable” definition: man muss Typeinfo nicht explizit angeben
- Typprüfungen werden hauptsächlich zur Laufzeit eines Programms durchgeführt

Duck-Typing

- der Typ eines Objektes wird nicht durch seine Klasse beschrieben wird
- sondern durch vorhandene Methoden oder Attribute

Duck-Typing/Analogie

'Wenn ich einen Vogel sehe,

der wie eine Ente läuft,

wie eine Ente schwimmt

und wie eine Ente schnattert,

→ *dann nenne ich diesen Vogel eine Ente.*'

Duck-Typing

```
class Computer:
    def sleep(self):
        print('Energy Saving Mode')

class Student:
    def sleep(self):
        print('Student is sleeping')

class Bird:
    def fly(self):
        print(Bird is flying)

def main():
    objects  = [Computer(), Student(), Bird()]
    for obj in objects:
        obj.sleep()
main()
```

```
Energy Saving Mode
Student is sleeping
Traceback (most recent call last):
  File "/Users/cat/Documents/python/mul_inh/main3.py", line 52, in <module>
    main()
  File "/Users/cat/Documents/python/mul_inh/main3.py", line 51, in main
    obj.sleep()
AttributeError: 'Bird' object has no attribute 'sleep'

Process finished with exit code 1
```

Named Parameters

- man kann Parameters mit Position übergeben

```
def f(a, b):
    return a + b

f(1,2)
```

Named Parameters

- wieso könnte das problematisch sein?
- die Reihenfolge der Parametern ist wichtig
- man könnte theoretisch Parameter mit gleichen Typ wechseln
- Alternative: named parameters

```
f(a=1, b=2)
f(b=2, a=1)
```

Named Parameters

- mit named parameters kann man auch die Lesbarkeit des Code erhöhen
- self-documenting Code
  - good code is its own documentation

```
send_packet(buf, 30, intvl)

send_packet(
    data=buf,
    timeout=30,
    interval=intvl
)
```

Method Overloading (überladen)

- mehrere Methoden mit dem gleichen Namen
  - aber eine unterschiedliche Anzahl von Parametern
- existiert in vielen anderen Sprachen (C++, Java, C#, etc.)
- funktioniert in Python nicht

```
def sum(a, b):
    return a + b
def sum(a, b, c):
    return a + b + c

sum(1,2,3)      → ok
sum(1,2)        → not ok
```

Parameter

Kann man einer Funktion eine beliebige Anzahl von Parametern übergeben?

- ja
- *args, **kwargs

kwargs

- *args = Non Keyword Arguments
- **kwargs = Keyword Arguments

```
def my_function(*args):
    pass

my_function(1, 2)


def my_function(**kwargs):
    pass

my_function(b=3, c=4)
```

kwargs

- *args = Non Keyword Arguments
- **kwargs = Keyword Arguments
- mit args/kwargs kann die Funktion eine beliebige Anzahl von Parametern bekommen

```
def my_function(*args, **kwargs):
    pass

my_function(1, 2, b=3, c=4)

def funk(**data):
     for key, value in data.items():
        print("{} is {}".format(key,value))

funk(name="bob", uni="ubb", Age=20)
```

args

```
def adder(*num):
    sum = 0

    for n in num:
        sum = sum + n

    print("Sum:",sum)

adder(3,5)              → Sum: 8
adder(4,5,6,7)          → Sum: 22
```

args

- man kann named parameters durchsetzen
- funk kann nur mit solcher Parametern aufgerufen werden

```
def funk(*, a, b):
    return a + b


funk(a=1, b=2) → ok

funk(1,2)
```

```
Traceback (most recent call last):
  File "/Users/cat/Documents/python/tttt/m.py", line 107, in <module>
    funk(1,2)
TypeError: funk() takes 0 positional arguments but 2 were given

Process finished with exit code 1
```

Klassenvariablen

Eine Klassenvariable kann nur mit Hilfe des Klassennamens verändert werden (theoretisch)

Der Klassenname und das Attribut werden durch einen Punkt miteinander verbunden

```
Modul.Klasse.Attribut = Wert
```

```
class T:
    counter = 0

    def __init__(self):
        type(self).counter += 1
```

Statische Methoden

```
class T:
    __counter = 0

    def __init__(self):
        type(self).__counter += 1

    @staticmethod
    def TotalInstances():
        return T.__counter

>>> T.TotalInstances()
>>> x = T()
>>> x.TotalInstances()
```

Monkey Patching

Kann man das Verhalten einer Methode dynamisch zum Laufzeit ändern?

- ja, das heißt monkey patching
- nutzbar insb. in Testing/Debugging

Monkey Patching

```
class Auto:
    def __init__(self, modell): self.__modell = modell
    def start(self): print('VROOM VROOM!')
```

```
dacia = Auto('Dacia1310')
dacia.start()
```

```
VROOM VROOM!
```

Monkey Patching

```
class Auto:
    def __init__(self, modell): self.__modell = modell
    def start(self): print('VROOM VROOM!')

def willnotstart():
    print ('No, I dont think I will')



dacia = Auto('Dacia1310')
dacia.start = willnotstart
dacia.start()
```

```
No, I dont think I will
```

Monkey Patching

```
class T:
    def __init__(self, attr):
        self.attr = attr

t = T(10)
print (t.attr, type(t))

old_init = T.__init__
def new_init(self, *k):
    print('fancy new init')
    old_init(self, *k)

    self.new_attr = 101

new_init(t, 10)

print(t.attr, t.new_attr, type(t))
```

Printing Objects

Wie kann man in Python die Information bzw. eines Objektes ausgeben?

Printing Objects

- mit einer Print-Methode

```
class Student:
    def __init__(self, name): self.__name = name

    @property
    def name(self): return self.__name



bob = Student('bob')
print(bob)
```

Printing Objects

- mit einer Print-Methode

```
class Student:
    def __init__(self, name): self.__name = name

    @property
    def name(self): return self.__name
```

```
bob = Student('bob')
print(bob)      <__main__.Student object at
0x7fb0efec5048>
```

Printing Objects

- mit einer Print-Methode

```
class Student:
    def __init__(self, name): self.__name = name

    @property
    def name(self): return self.__name

    def print(self): print(f'Name: {self.__name}')

bob = Student('bob')
bob.print()          Name: bob
```

Printing Objects

- __str__ Standard-Methode, die eine String Repräsentation liefert

```
class Student:
    def __init__(self, name): self.__name = name

    @property
    def name(self): return self.__name

    def __str__(self): return f'Name: {self.__name}'

bob = Student('bob')
print(bob)
```

Printing Objects

- __str__ Standard-Methode, die eine String Repräsentation liefert

__str__ ist beim print automatisch aufgerufen

```
class Student:
    def __init__(self, name): self.__name = name

    @property
    def name(self): return self.__name

    def __str__(self): return f'Name: {self.__name}'

bob = Student('bob')
print(bob)                                  Name: bob
```

Printing Objects

- __str__ = eine String Repräsentation
- __repr__ = eine eindeutige Repräsentation

```
class Student:
    def __init__(self, name): self.__name = name
    def __str__(self): return f'Name: {self.__name}'
```

```
l = [Student('bob'), Student('dob')]
print(l)
```

Printing Objects

- __str__ = eine String Repräsentation
- __repr__ = eine eindeutige Repräsentation

```
class Student:
    def __init__(self, name): self.__name = name
    def __str__(self): return f'Name: {self.__name}'
```

```
l = [Student('bob'), Student('dob')]
print(l)
```

```
[<__main__.Student object at 0x7fc3ee4a0048>, <__main__.Student object at 0x7fc3ee4a0048>]
```

Printing Objects

- __str__ = eine String Repräsentation
- __repr__ = eine eindeutige Repräsentation

```
class Student:
    def __init__(self, name): self.__name = name
    def __str__(self): return f'Name: {self.__name}'
    def __repr__(self): return f'Name: {self.__name}'


l = [Student('bob'), Student('dob')]
print(l)
```

```
[Name: bob, Name: dob]
```

Hooks

- bisher wurden einige Operatoren vorgestellt: `+`, `-`, …
- die funktionieren für bestimmten Typen, wie zB
  - + für int, string, list, etc.
  - - für int, set, etc.
  - etc.
- in Python gibt es aber gar keine Operatoren, sondern nur Operationen:
  - der „*“-`Operator` ruft beispielsweise intern die `__mul__` Methode des ersten Operanden auf
  - diese speziellen Methoden kann man selbst definieren, um damit die Funktionalität zu ändern oder zu erweitern
- man kann solche Operatoren für Klasse implementieren

Hooks

```
class Point:
   def __init__(self, x, y):
      self.x = x
      self.y = y

   def move(self, delta):
      self.x += delta
      self.y += delta

   def __str__(self): return f’Point({self.x}, {self.y})’

p = Point(1,2)
p.move(10)
print(p)
```

Hooks

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, delta):
        return Point(self.x + delta, self.y + delta)

    def __str__(self): return f'Point({self.x}, {self.y})'

p = Point(1,2)
p = p + 10
print(p)
```

Hooks

```
class Point:
   def __init__(self, x, y):
      self.x = x
      self.y = y

   def __add__(self, delta):
      return Point(self.x + delta, self.y + delta)

   def __str__(self): return f’Point({self.x}, {self.y})’

p = Point(1,2)
p = 10 + p     #error
print(p)
```

Hooks

```
class Point:
   def __init__(self, x, y):
      self.x = x
      self.y = y

   def __add__(self, delta):
      return Point(self.x + delta, self.y + delta)

   def __radd__(self, delta):
      return Point(self.x + delta, self.y + delta)

   def __str__(self): return f'Point({self.x}, {self.y})'

p = Point(1,2)
p = 10 + p
print(p)
```

Hooks

```
class Point:
   def __init__(self, x, y):
      self.x = x
      self.y = y

   def __add__(self, delta):
      return Point(self.x + delta, self.y + delta)

   __radd__ = __add__

   def __str__(self): return f’Point({self.x}, {self.y})’


p = Point(1,2)
p = 10 + p
print(p)
```

Hooks

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __iadd__(self, delta):
        self.x += delta
        self.y += delta
        return self

    def __str__(self): return f’Point({self.x}, {self.y})’

p = Point(1,2)
p += 10
print(p)
```

Hooks

```
class Rational:
    def __init__(self, num, den):
        self.num = num
        self.den = den

    def __mul__(self, other):
        num = self.num * other.num
        den = self.den * other.den
        return Rational(num, den)

    def __repr__(self):
        return "R("+ str(self.num)+","+ str(self.den) + ")"

    def __str__(self):
        return str(self.num) + "/" + str(self.den)
```

```
>>> r1 = Rational(1,2)
>>> r2 = Rational(3,4)
>>> r1 * r2
R(3, 8)
>>> print(r1 * r2)
3/8
```

Hooks

- Vergleichsoperatoren (Rückgabe: True / False):
  - `__eq__` → `==`
  - `__ge__` → `>=`
  - `__gt__` → `>`
  - `__le__` → `<=`
  - `__lt__` → `<`
  - `__ne__` → `!=`
- `__bool__` : gilt das Objekt als Wahr oder Falsch? (Gibt True oder False zurück)

Hooks

- Numerische Operationen:
  - `__add__` → `+`
  - `__div__` → `/`
  - `__mul__` → `*`
  - `__sub__` → `-`
  - `__mod__` → `%`

- Element-Zugriff für Collections:
  - `__getitem__(self, index)` → `x[i]`
  - `__setItem__(self,index,value)` -> `x[i] = 10`

Hooks

Common Syntax	Special Method Form	
a + b	a.__add__(b);	alternatively b.__radd__(a)
a − b	a.__sub__(b);	alternatively b.__rsub__(a)
a * b	a.__mul__(b);	alternatively b.__rmul__(a)
a / b	a.__truediv__(b);	alternatively b.__rtruediv__(a)
a // b	a.__floordiv__(b);	alternatively b.__rfloordiv__(a)
a % b	a.__mod__(b);	alternatively b.__rmod__(a)
a ** b	a.__pow__(b);	alternatively b.__rpow__(a)
a << b	a.__lshift__(b);	alternatively b.__rlshift__(a)
a >> b	a.__rshift__(b);	alternatively b.__rrshift__(a)
a & b	a.__and__(b);	alternatively b.__rand__(a)
a ^ b	a.__xor__(b);	alternatively b.__rxor__(a)
a | b	a.__or__(b);	alternatively b.__ror__(a)
a += b a −= b a *= b ...	a.__iadd__(b) a.__isub__(b) a.__imul__(b) ...	
+a	a.__pos__()	
−a	a.__neg__()	
~a	a.__invert__()	
abs(a)	a.__abs__()	
a < b	a.__lt__(b)	
a <= b	a.__le__(b)	
a > b	a.__gt__(b)	
a >= b	a.__ge__(b)	
a == b	a.__eq__(b)	
a != b	a.__ne__(b)	

Hooks

```
class Data:
    def __init__(self):
        self.data = []

    def add_element(self, elem):
        self.data.append(elem)

    def print_elements(self):
        for el in self.data:
            print(el)

    def __getitem__(self, index):
        return self.data[index]

    def __setitem__(self, index, value):
        self.data[index] = value

data = Data()

for i in range(10):
    data.add_element(i)

print ('second element', data[1])
data[1] = 101
data.print_elements()
```

```
second element: 1
0
101
2
3
4
5
6
7
8
9
```

Beispiel

```
class R:
  def __init__(self, a, b):
     if b == 0:
          raise ZeroDivisionError("b cannot be 0")
    if isinstance(a, int) and isinstance(b, int):
          self.__a = a
          self.__b = b
    else:
          raise ValueError(“args should be int”)

  def __add__(self, o):
    return R (self.a*o.b+self.b*o.a, self.b*o.b)

  def __lt__(self, o):
    return self.a/self.b < o.a/o.b

  def __eq__(self, o):
    return self.a == o.a and self.b == o.b
```

```
def __str__(self):
    if self.b == 1:
       return str(self._numerator)
     return "%i/%i" % (self.a, self.b)

@property
def a(self):
     return self.__a
@property
def b(self):
     return self.__b
```

Beispiel

```
def main():
     r = R(1,2)

     print(R(1,2) == R(1,2))

     print (r + R(1,2))

     print (r < R(2,3))

     try:
          p = R(1,0)
     except ZeroDivisionError:
          print(Fraction not Valid’)
     except ValueError:
          print(Types not Valid’)

     print(‘here’)
main()
```

Exkurs

Sortiere eine Liste von Studenten nach Alter

Dataclass

- struktuierte Daten
- vorhandene Typen, die erlauben, komplexe Daten effizient zu speichern und zugreifen
- abhangig von der konkreten Situation
- dataclass
  - eine Klasse, die nur ‘Daten’ repräsentiert

Tuple

```
red = (255, 0, 0)
green = 0, 255, 0
white  = tuple(255, 255, 255)

>>> red[0]
255
>>> r,g,b = white
>>>b
255
```

List

```
red = list((255, 0, 0))
green = [0, 255, 0]

>>>green[0]
0
```

Dict

```
red = dict (red=255, green=0, blue=0)
green = {‘red’: 0,
         ‘green’: 255,
         ‘blue’: 0}

>>> green[‘red’]
0
```

namedtuple

```
from collections import namedtuple

Color = namedtuple('Color', 'red green blue')
red = Color(255, 0, 0)
green = Color (red=0,
               green=255,
               blue=0)

>>>green.red
0
```

namedtuple mit Typing

```
from typing import NamedTuple

class Color (NamedTuple):
    red: int
    green: int
    blue: int = 0 #default

red = Color(255, 0)
green = Color (red=0,
               green=255,
               blue=0)

>>>green.red
0
>>red.blue
0
```

TypedDict

SimpleNamespace

- funktioniert wie ein Dict
- del
- man kann Werte ändern

```
from types import SimpleNamespace

red = SimpleNamespace(red=255, green=0, blue=0)

>>> red.red
255
```

Class

```
class Color:
    def __init__(self, red, green, blue=0)
        self.red = red
        self.green = green
        self.blue = blue


red = Color(255, 0, 0)
green = Color (red=0,
               green=255,
               blue=0)

>>>green.red
0
```

Class

für eine Klasse muss man aber auch viele Methoden umsetzen

- __str__ / __repr__
- oder: __lt__, etc.
- __eq__
- …

Class

```
class Color:
   def __init__(self, red, green, blue=0):
      self.red = red
      self.green = green
      self.blue = blue
   def __str__(self): return f’Color(red={self.red},
green={self.green}, blue={self.blue})’
   def eq(self, other):
      return (self.red, self.green, self.blue) ==
(other.red, other.green, other.blue)
   def gt(self, other):
      return (self.red, self.green, self.blue) >
(other.red, other.green, other.blue)
   #order: __lt__(), __le__(), __gt__(), __ge__()
```

Immutability

- by-default Klassen sind mutable
- immutability = bedeutet, jede änderung erzeugt ein neues Objekt
- klassisches Beispiel:
  - string.replace(...)
- wie implementiert man eine solche Klasse in Python
  - einfachste Variante: namedtuple
- __setattr__
- __delattr__

Immutability

```
def __setattr__(self, name, value):
    raise TypeError(f‘cannot set {name}’)

def __delattr__(self, name):
    raise TypeError(f‘cannot delete {name}’)


>>> red = Color(255, 0, 0)
```

```
/usr/local/bin/python3.7 /Users/cat/Documents/python/tttt/m.py
Traceback (most recent call last):
  File "/Users/cat/Documents/python/tttt/m.py", line 98, in <module>
    red = Color(255, 0, 0)
  File "/Users/cat/Documents/python/tttt/m.py", line 83, in __init__
    self.red = red
  File "/Users/cat/Documents/python/tttt/m.py", line 93, in __setattr__
    raise TypeError(f'cannot set {name}')
TypeError: cannot set red
```

Immutability

```
class Color:
    def __init__(self, red, green, blue=0):
        object.__setattr__(self, "red", red)
        object.__setattr__(self, "green", green)
        object.__setattr__(self, "blue", blue)

>>> red = Color(255, 0, 0)
>>> red.red
255

>>>red.red=255
```

```
Traceback (most recent call last):
  File "/Users/cat/Documents/python/tttt/m.py", line 103, in <module>
    red.red = 255
  File "/Users/cat/Documents/python/tttt/m.py", line 95, in __setattr__
    raise TypeError(f'cannot set {name}')
TypeError: cannot set red
```

it’s complicated…

- man muss viel boilerplate Code schreiben
- __init__, __str__, __repr__
- kostet viel Zeit
- dataclass ist eine gute Alternative

it’s complicated…

```
from dataclasses import dataclass

@dataclass
class Color:
    red: int
    green: int
    blue: int

red = Color(255, 0, 0)
blue = Color(red=0, green=0, blue=255)

>>> red
Color(red=255, green=0, blue=0)
```

Dataclass

- eine ganz normale Klasse
- implementiert wichtige Methoden
  - __init__()
  - __repr__()
  - __eq__()
  - oder: __lt__(), __le__(), __gt__(), __ge__()
    - @dataclass(order=True)
  - …
- reduziert boilerplate code
- make_dataclass Method
  - generiert automatisch eine Klasse

```
from dataclasses import make_dataclass
Position = make_dataclass('Color', ['r', 'g', 'b'])
```

Types

- erlaubt default value

```
from dataclasses import dataclass

@dataclass
class Color:
    red: int = 0
    green: int = 0
    blue: int = 0
```


import * as serviceWorker from './serviceWorker';
ReactDOM.render((
<BrowserRouter>
<Switch>
<Route path="/login" component={Logi
<ProtectedRoute exact={true} path="/
<ProtectedRoute path="/settings" com
<ProtectedRoute component={Dashboard
</Switch>
</BrowserRouter>
), document.getElementById('root'));
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL

Codestruktur

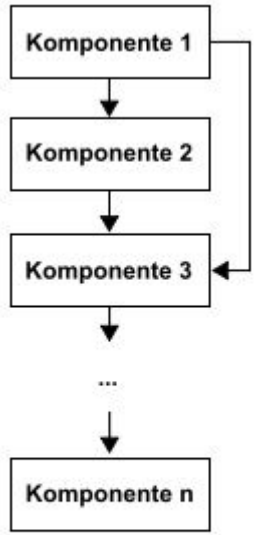
- was ist das genau?

- man benutzt einige Design Prinzipien, um Code besser strukturieren zu können

- Single Responsability Prinzip
- Separation of Concerns
- Dependencies
- Coupling and Cohesion

Schichtenarchitektur

- ein häufig angewandtes Strukturierungsprinzip für die Architektur von Softwaresystemen
- Aspekte einer „höheren“ Schicht nur solche „tieferer“ Schichten verwenden dürfen
- die Trennung von Fachkonzept, Benutzungsoberfläche und Datenhaltung



Aufrufe in einer Schichtenarchitektur

Schichtenarchitektur

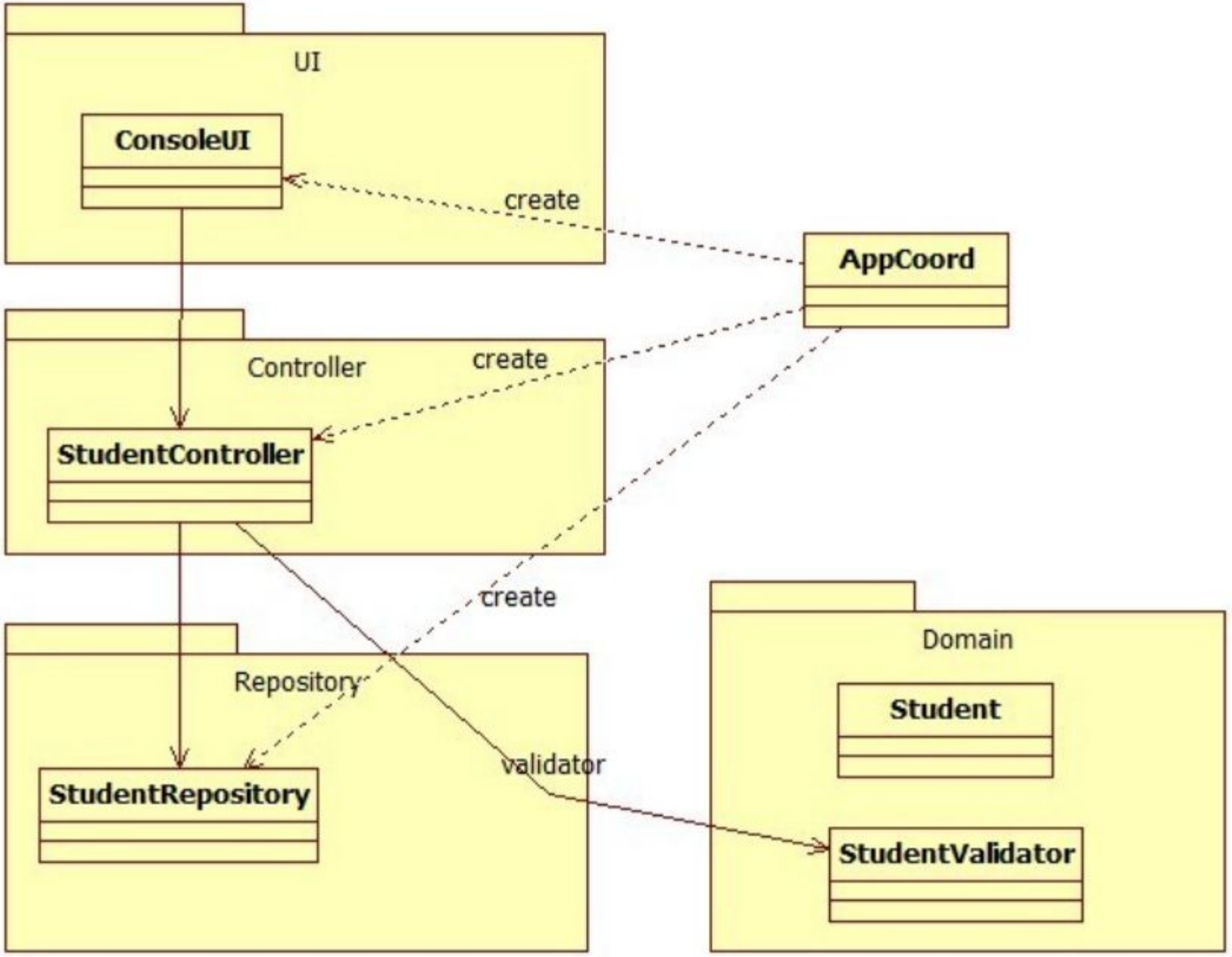
- **Präsentationsschicht**
  - Benutzerschnittstelle
- **Businessschicht**
  - Controller
  - Entities
- **Datenhaltungsschicht**
  - Daten Laden und Speichern
  - Repositories

Problemstellung

Schreibe ein Programm, das alle Daten von Studenten verwaltet. (CRUD-Operationen)

Feature

F1. Studenten einfügen
F2. Studenten filtern (nach Alter)
F3. Studenten finden
F4. Durchschnittsnote aller Studenten
F5. Sort
…


UI
ConsoleUI
create
AppCoord
Controller
create
StudentController
create
Repository
validator
Domain
Student
StudentRepository
StudentValidator

Entitäten

Objekt != Entität? (eigentlich nicht!)

- ein Objekt hat eine Reihe ihm eigener dynamischer Funktionen, Operationen und Methoden
- Entitäten sind Objekte mit einer eindeutigen Identifizierung

verwechselter Identitäten führt zu Datenfälschung

Entitäten

```
def testIdentity():
    #attributes may change
    st = Student("1", "Ion", "Adr")
    st2 = Student("1", "Ion", "Adr2")
    assert st==st2

    #is defined by its identity
    st = Student("1", "Popescu", "Adr")
    st2 = Student("2", "Popescu", "Adr2")
    assert st!=st2
```

```
class Student:
    def __init__(self, id, name, adr):
        """
         Create a new student
         id, name, address String
        """
        self.__id = id
        self.__name = name
        self.__adr = adr

    def __eq__(self,ot):
        """
          Define equal for students
          ot - student
          return True if ot and the current instance represent the same student
        """
        return self.__id==ot.__id
```

Value Objects

Objekte, die keine eindeutige Identifizierung haben

zB: Farben/Adressen

Es kommt immer auf den Anwendungsfall an!

Value Objects

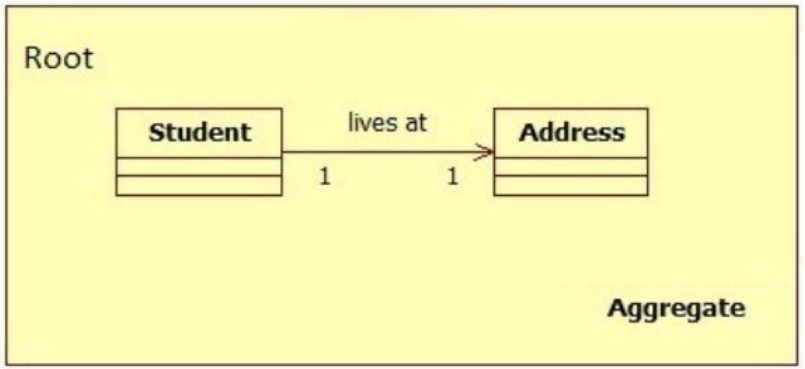
```
def testCreateStudent():
    """
      Testing student creation
      Feature 1 - add a student
      Task 1 - Create student
    """
    st = Student("1", "Ion", Address("Adr", 1, "Cluj"))
    assert st.getId() == "1"
    assert st.getName() == "Ion"
    assert st.getAdr().getStreet()=="Adr"

    st = Student("2", "Ion2", Address("Adr2", 1, "Cluj"))
    assert st.getId() == "2"
    assert st.getName() == "Ion2"
    assert st.getAdr().getStreet() == "Adr2"
    assert st.getAdr().getCity() == "Cluj"
```

```
class Address:
    """
      Represent an address
    """
    def __init__(self,street,nr,city):
        self.__street = street
        self.__nr = nr
        self.__city = city

    def getStreet(self):
        return self.__street

    def getNr(self):
        return self.__nr

    def getCity(self):
        return self.__city
```

```
class Student:
    """
      Represent a student
    """
    def __init__(self, id, name, adr):
        """
         Create a new student
         id, name String
         address - Address
        """
        self.__id = id
        self.__name = name
        self.__adr = adr

def getId(self):
        """
          Getter method for id
        """
        return self.__id
```

Aggregation

die Zusammensetzung einer Entität aus einer Menge von Value Objects



Repositories?

StudentRepository AddressRepository

Single-Reponsibility-Prinzip

- Jede Funktion sollte für eine Sache verantwortlich sein
- Jede Klasse sollte eine Entität darstellen
- Jedes Modul sollte einem Aspekt der Anwendung entsprechen

Single-Reponsibility-Prinzip

- lass uns das folgende Beispiel nehmen

```
def filterScore(scoreList):
    st = input(”Start score :”)
    end = input(”End score:”)
    for score in scoreList :
        if score [1] > st and score [1] < end:
        print(score)
```

Single-Reponsibility-Prinzip

- lass uns das folgende Beispiel nehmen

```
def filterScore(scoreList):
    st = input(”Start score :”)
    end = input(”End score:”)

    for score in scoreList :
        if score [1] > st and score [1] < end:
        print(score)
```

- liest etwas von der Tastatur ein
- berechnet was
- gibt das Ergebnis aus

Single-Reponsibility-Prinzip

- kann diese `filterScore` Funktion sich verändern?
- ein komplett anderes Format für Input
  - Konsoleanwendung (Menu)
  - GUI
  - Webseite
- ein neuer Filter
- ein anderes Format für Output
- → die Methode hat 3 Verantwortungen

Single-Reponsibility-Prinzip

- der gleiche gilt für Module
- Module stellen Methoden zusammen, die thematisch miteinander passen
- mehrere Verantwortungen sind schwer zu
  - verstehen
  - verwenden
  - testen
  - warten
  - weiterentwickeln

Separation of Concerns

- man soll das Programm in verschiedenen Abschnitte aufteilen
- jeder Abschnitt adressiert ein bestimmtes Problem
- Concerns - Informationen, die auf Code auswirken
  - z. B. Computerhardware, auf der das Programm ausgeführt wird, Anforderungen, Funktionen und Modulnamen
- richtig implementiert führt zu einem Programm, das einfach zu testen ist und einfach wiederverwendet werden können

Separation of Concerns

- die gleiche Methode

```
def filterScore(scoreList):
    st = input(”Start score :”)
    end = input(”End score:”)
    for score in scoreList :
        if score [1] > st and score [1] < end:
        print(score)
```

Separation of Concerns - UI

- nur UI Funktionalität
- der Rest sind an `filterScore` delegiert

```
def filterScoreUI(scoreList):
    st = input(”Start score :”)
    end = input(”End score:”)

    result = filterScore(scoreList, st, end)

    for score in result :
        print(score)
```

Separation of Concerns - der Rest

- die Methode hat nur eine Verantwortung

```
def filterScore(scoreList, st, end):
    rez = []

    for p in lst:
        if p[1] > st and p[1] < end:
            rez.append(p)

    return rez
```

Separation of Concerns - das Testen

- die `filterScore()` Funktion kann so getestet werden

```
def filterScoreTest():
      lst = [[“Anna”, 100]]

      assert filterScore(lst, 10, 30) == []
      assert filterScore(lst, 1, 300) == lst

      lst == [[“Anna”], 100], [“Ion”], 40], [“P”], 60]]
      assert filterScore(lst, 3, 50) == [[“Ion”], 40]
```

Dependency/Abhängigkeit

- was ist eine Abhängigkeit?

- **Funktionen** - Eine Funktion ruft eine andere Funktion auf
- **Klassen** - Eine Klassenmethode ruft eine Methode einer anderen Klasse auf
- **Module** - Eine Funktion eines Moduls ruft eine Funktion eines anderen Moduls auf

- Gegeben sei die folgenden Funktionen a, b, c und d.
  - a ruft b an, b ruft c an und c ruft d an

- Was kann passieren, wenn wir die Funktion d ändern?

Kohäsion

- wie gut eine Programmeinheit (eine Funktion/ein Modul) eine logische Aufgabe oder Einheit abbildet

- **starke Kohäsion:** alle Teile eines Moduls sollten mit anderen Teilen des Moduls zusammenhängen und voneinander abhängig sein.
- **schwache Kohäsion:** Teile eines Moduls haben keinen Bezug zu anderen Teilen

- viele Teile -> schwer zu verstehen!

Kopplung

ein Maß, das die Stärke die Verknüpfung von verschiedenen Systemen, Anwendungen, oder Softwaremodulen beschreibt

Formen von Kopplung. Am Beispiel von der Klasse X zur Klasse Y:

- X ist direkte oder indirekte Unterklasse von Y
- X hat Attribut bzw. Referenz von Typ Y
- X hat Methode, die Y referenziert (Abhängigkeit)