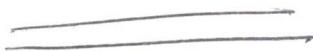
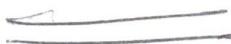


Clean Coding



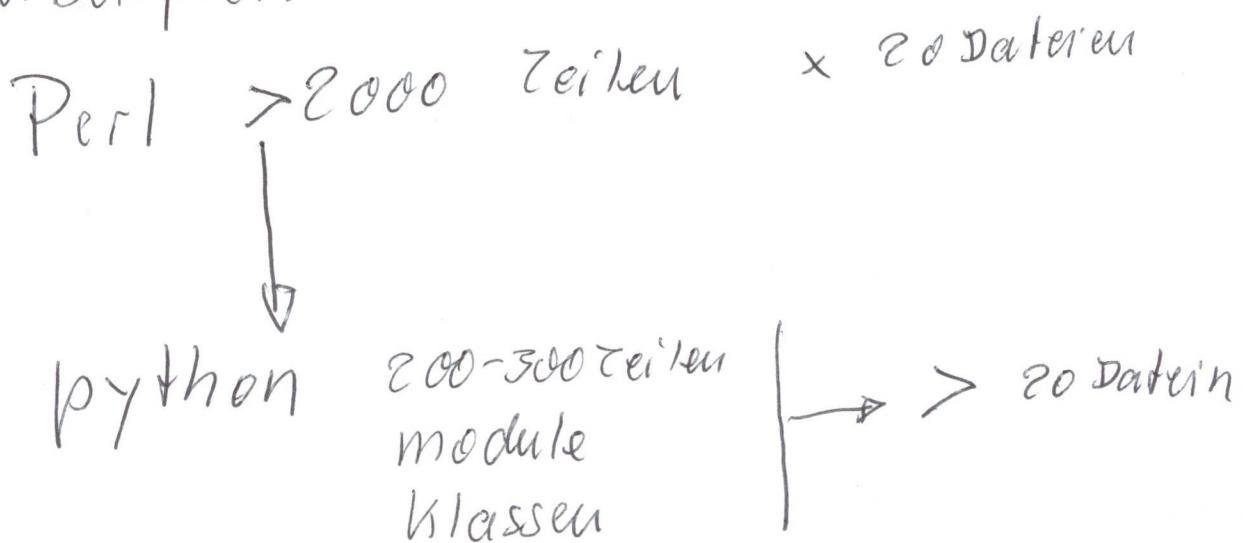
Ulrich Cuber



Start 9:00

CodQualität

Projektbeispiel:



- Lesbarkeit → jeder kann lesen und verstehen
 - Dokumentiert → Sinnvolle Kommentare
 - Testbarkeit → Allg. Doku
 - Wiederbarkeit → Testzugriff
Funktionale Abgrenzung
 - Wartbarkeit → Tests → Regression
Codestyles
Zero Warning
Vollständig
- ↓

Lösen des Programms
Weiterentwicklung -n-

Codequalität

↔ Style guides
Programming Rules } → Tools!
↳ Community / Organisation

↔ Review techniken
↳ Peer-Review
↳ Pairing

↔ Testdriven Design
Testen mit xUnit

↔ Refactoring !!
„Pfadfinder-Regel“

↔ auto-Codecheckung | Tools
Messen / Monitoren

z. B. SonarCube
pylint
Black

↔ immer wieder obiges ergänzen/verbessern

Community

PEP 8 Styleguide

Tool pep8 auf pypi

PEP 20 Zen of Python

Tool import this

Pythonic z. B. Schleife

Aufgabe: Anlage Liste mit Init-Werten
1 - 20

z. B. while

$l = []$

$v = 1$

while $v < 21:$

$l.append(v)$

$v = v + 1 / v += 1$

for

$l = []$

for n in range(1, 21):

$l.append(n)$

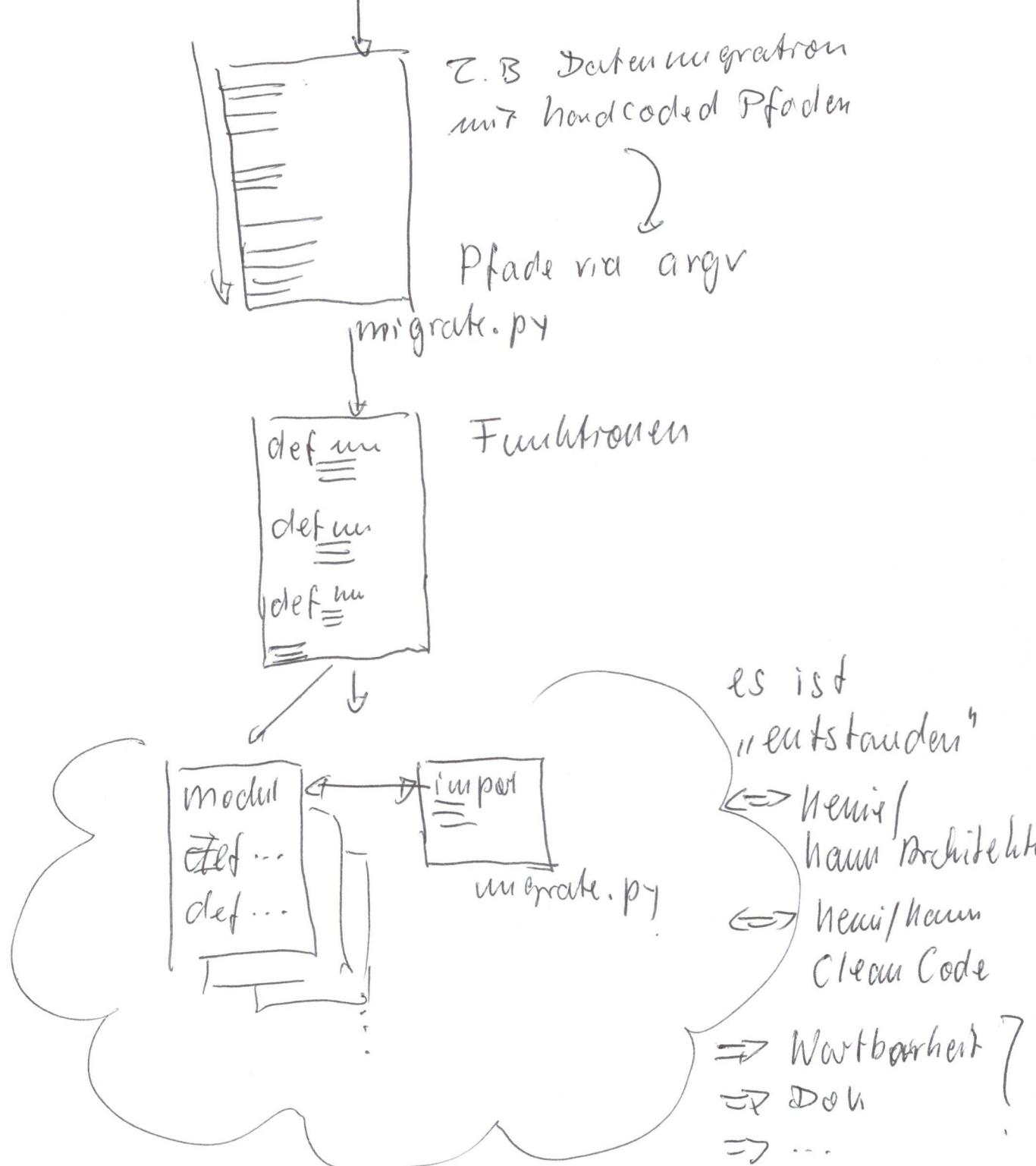
\Rightarrow

pythonic

$l = list(range(1,$

Geschichte eines Skriptes

Start: „Wir brauchten mal ...“



Architekturfragen

E - V - A

Eingabe
Vorarbeiten
Ausgabe } was, wo?

↳ Funktionalen
↳ Module

Daten exl./intern?

↳ JSON, XML, Excel...
Quellen → Verhalten
Ziele →

klass?
?

Interne Datenhaltung:

Liste und / oder Dict

Liste von Dict

Liste von Listen

Dict mit Dict

Dict mit Liste

Rekursive Struktur? z.B. Baum

Erfahrung, Heuristiken

SOLID, DRY, YAGNI, KIS ...

DRY - Don't repeat yourself !!

Vermeide Wiederholung

→ Fehleranfällig!

→ Teuer!

→ Unzuverlässig!

Funktionen mit Parametern helfen
bzw Klassen helfen

YAGNI You ain't gonna need it !!

v.a. bei Klassen: Interface-Wichung
zu viele Methoden von den vermutet
wird dann sie mal gebraucht werden

↔ lass sie weg!

KIS Keep It Simple !

1

EATP: Easier to ask for
forgiveness than Permission

Bsp ext. Rufnr → fällt in 1/100 Fällen

a) for n in range(1000):

```
1000x   v = external-data()
           if v != None:
               if v >= 1 or <= 50:
                   if isinstance(v, int):
                       mach was(v)
                   else:
                       Fehler!
               else:
                   Fehler!
           else:
               Fehler!
```

Viel Testen ⇒ Viel Aufwand
⇒ viel Redundanz
⇒ viel Code
⇒ nicht so schön!



b) EAFP als Lösung

for n in range(1000):

try:

$\{ V = \text{external_data}()$
 $\quad \text{machiwas}(v)$

except ExceptionNone: \Leftrightarrow NoneError
==

except ExceptionType: \hookrightarrow TypeError

10^x - except Exception Range:
=

except Exception as e:
===== Suspect(e)
raise e

erkl. Klassen für Exceptions zu def.
Struktur einfacher

→ alles nah bei einander

Weniger Rechenzeit

ggf sicheres Aufräumen mit
finally

Pattern

Ziel : Standards
Best Practices



- Testen verbessert
- Flexibel gegen Änderungen
- Feiner Modularisiert

Beispielpattern: Factory

boilerplate

$e = E(\text{elb connector})$

$a = A(\text{csv file})$

$v = V()$

$v.set E(e)$

$v.set A(a)$

v steht zur Verfügung

$v = v_factory(\text{elb connector, csv file})$

Pattern / Antipattern

Antipattern - typische "No-gos"



- Konstrukte die schlecht sind
- die man immer wieder findet
- und selber macht

"Mythische" AP - von vielen und oft als AP bezeichnet, aber in Wirklichkeit ok

def m():

; =

 ← m()

def m(v):

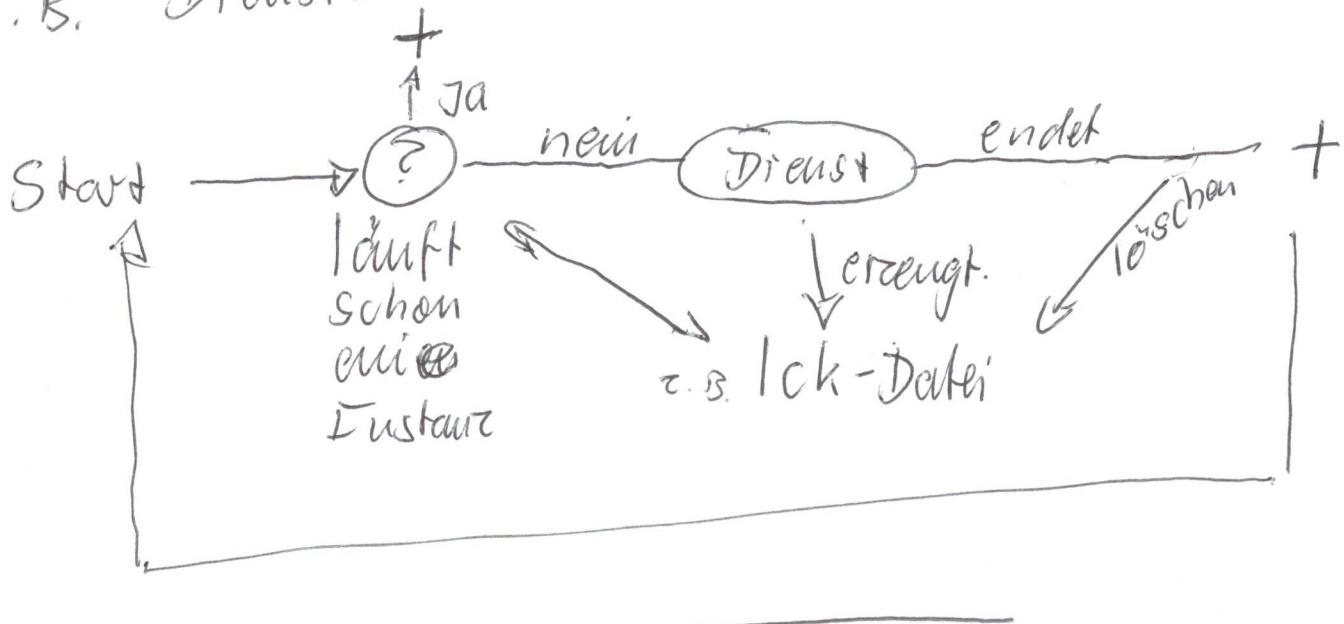
=

 ← m(4711)

Bsp Singleton

Es kann nur ein. geben

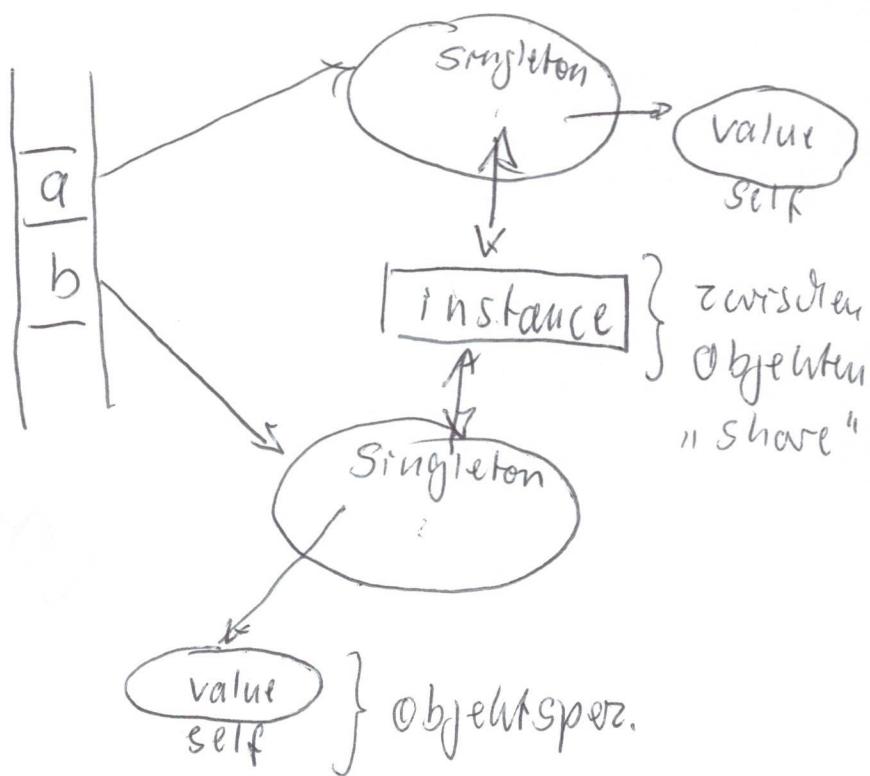
Z. B. Dreiste



in PY

```
def m(self)  
    self.value
```

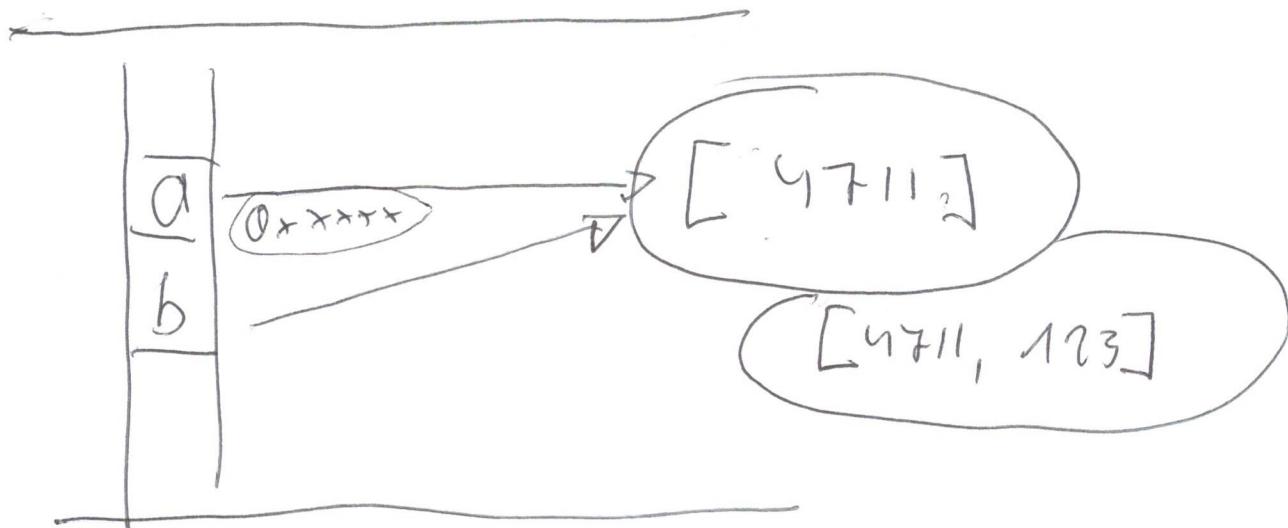
```
a = Singleton()  
b = Singleton(7)
```



$a = [4711]$

zu smell-copy

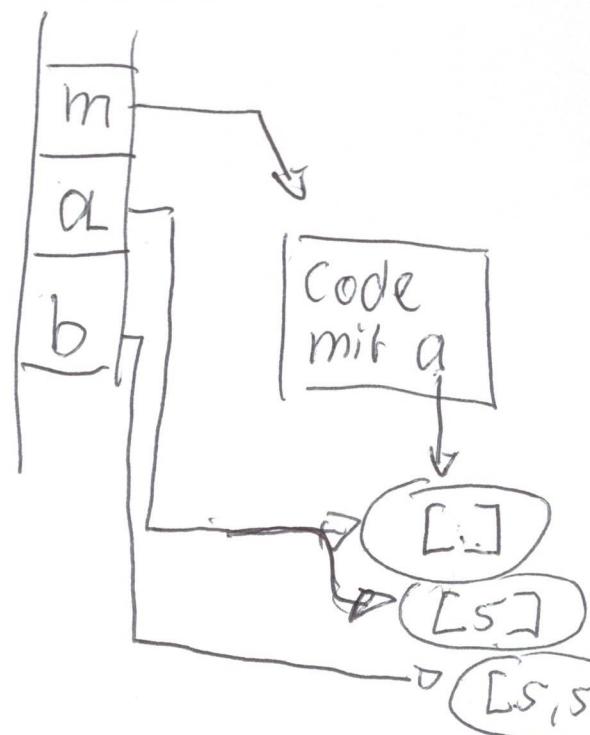
$b = a$



$b.append(123)$

```
def m(a=[ ]):
    a.append(5)
    return a
```

$a = m()$ $b = m()$ $c = m()$



Messungen

Komplettät

Umfänge

Aufrufe

...

Fragen:

- Länge einer Methode
- Länge einer Klasse
- Anzahl Methoden/Klasse
(Größe der Schnittstelle)

Anzahl Klassen



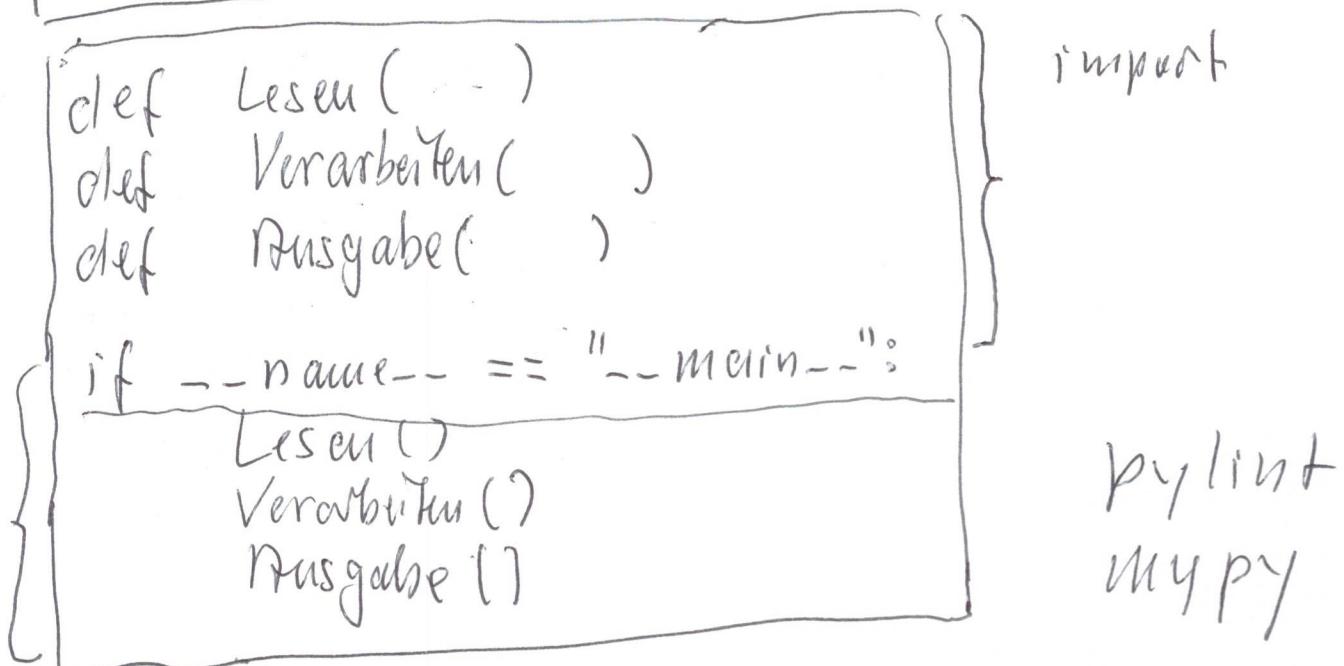
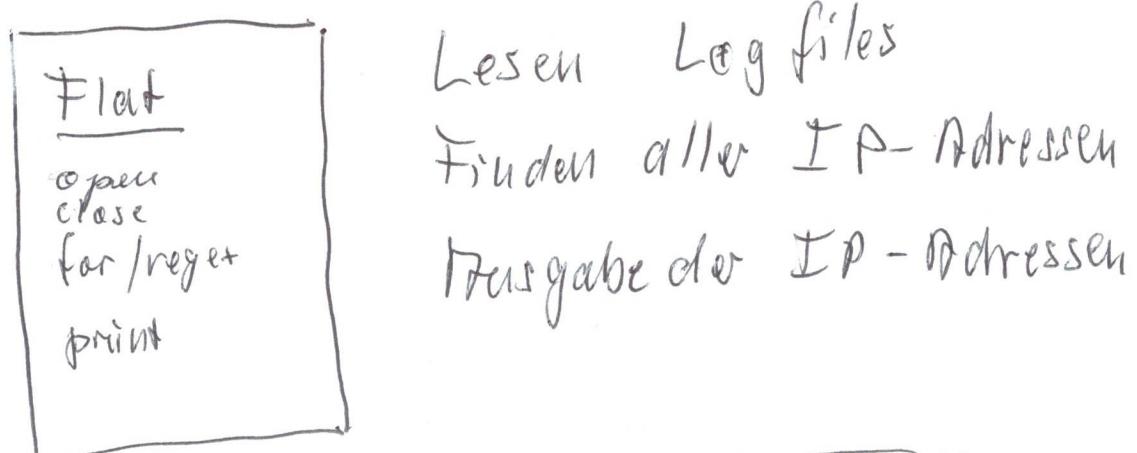
aus der QA:

- Fehlerrateanzahl
- Fehlerhotspots

...

z. B. radon

Programmstruktur



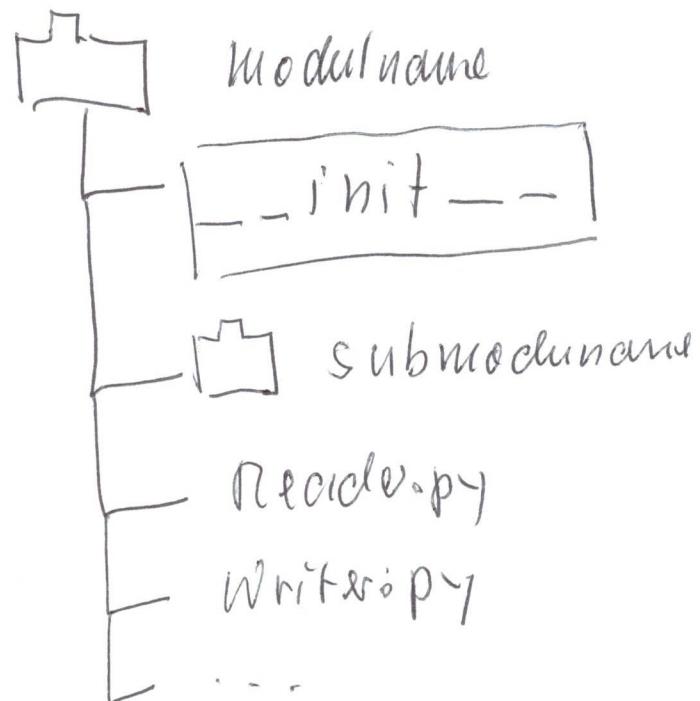
```
class LogfileIPFinder:  
    def __init__  
    def lesen  
    def verarbeiten  
    def ausgabe
```

```
L1 = LogfileIPFinder()  
L2 = LogfileIPFinder()
```

Modul

Single File

Folder basert



Shortcut Eval.

==

Log. Operatoren

Nur ~~z~~ Seiteneffekte

and or

True and True

eval: → True — True

True / bei Trues gesehen

False and True

eval: → False

→ False / 2. Ausdruck nicht
angesehen

fnt1(werte) and fnt2(andere-werte)

True/Truthy

+ 0
+ []
+ None

True/Truthy

T

T

F

F

T

F

T

F

fnt1, fnt2

fnt1, fnt2

fnt1, \neq

fnt1, \neq

Analog or

List Comprehension

$l = \text{mach_was}()$

list von int

soll geformt werden \rightarrow str

$s = ""$
for n in $l:$
 $s = s + \text{str}(n) + "|"$
 $s = s[:-1]$, pythonic

$s = "1|2|3|4"$

for i, n in enumerate(l):
if $i < \text{len}(l)$
 $s = s + \text{str}(n) + "|"$
else:
 $s = s + \text{str}(n)$

↓ pythonic

$s = "|".join([str(n) for n in l])$

list. compr.

Slicing

Liste, String

$s = "ABCD"$

$\text{len}(s) \rightarrow 4$

$s[0] \rightarrow 'A'$

$s[\text{len}(s)-1]$

$s[3]$

$s[-1]$

$\rightarrow 'D'$

$s[:] \rightarrow 'ABCD'$

$s[0:3] \rightarrow 'ABC'$

$s[:3] \rightarrow 'ABC'$

$\underbrace{s[3:-1]}_{\text{slice}} \rightarrow 'BC'$

slice

Direct-Zugriff

```
d = {  
    | 'Name': 'Willi',  
    | 'Ort': 'HH'  
}
```

$d['Name'] \rightarrow \text{Willi}$

$d['PLZ'] \rightarrow \text{KeyError}$

```
try:  
    plz=d['PLZ']  
except Key Error as e:  
    plz=None  
    print(plz)
```

```
plz=d.get('PLZ', None)  
print(plz)
```

Ok, aber ...
nicht nur 3. Variante:

```
if 'PLZ' in d:  
    plz=d['PLZ']  
else:  
    plz=None
```

Typehints / Annotation

import typing

l: List[int]

l: List[str]

def f(arg: Type[CustomClass])

l: List[Type[CustomClass]]

Varablen } Meta-Daten
Parameterlisten } Dokumentation
 | für lesende Personen
 | Neuer Effekt für Interpreter

Function Annotation: PEP 3107

Fehlerbehandlung

nicht "praktisch"

→ viele checks vorweg

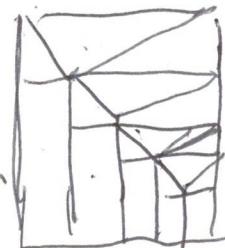
→ irgendetwas vergisst man immer ⇒ doch eine Exception

→ if-Hashtags

Schlecht lesbar

Eintrübung!

→ Fehleranfälligkeit



→ Laufzeitkosten

Weil gg. der Durchläufe nur O.K sind
Warum dann wird auf nur 1. reagiert



try - except - finally

try - except

try - finally

→ Exception / Exception handling

↑
vermeide catch-all!

vermeide "silent errors"

Fehlerbehandlung

try:
=====
"happy path"

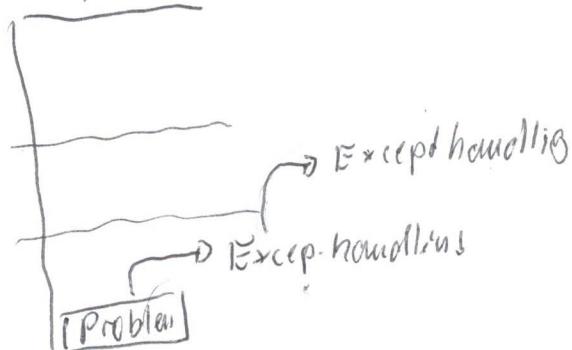
except ETyp1 as e:
 Errorhandling

except ETyp2 as e:
 Errorhandling

manchmal,
meistens except Exception as e: # catch all
keine gute Idee == z.B. für Logging!
 ↳ rasse e aufgebracht

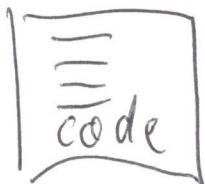
Errorbehandlung: "heilend", kein reraise
dokumentierend, evtl. reraise
z.B. Logging

Programm



Methode

class X:
 feld1
 def f(self):



a = X()

b = X()

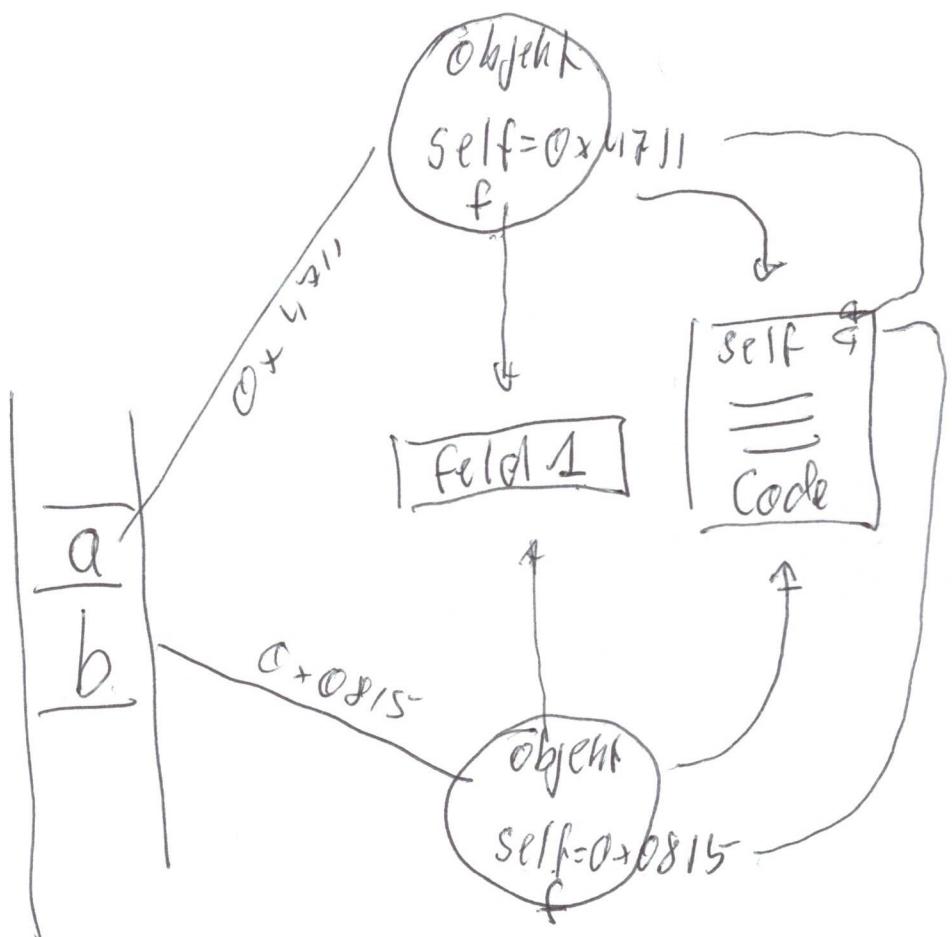
a.f()

b.f()

Modell

X.f(b)

X.f(a)

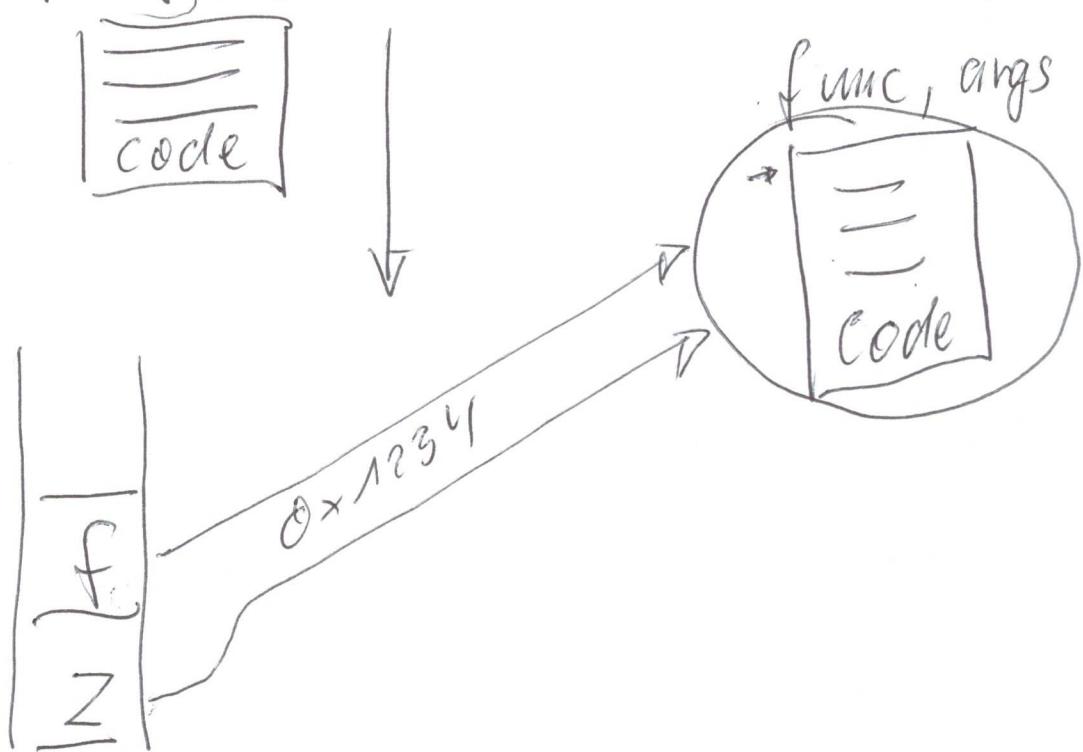


Funktionen /

Tanibdas

„Funktionspointer“ / „Delegat“

def f():



-- main --

$Z = f$ // alias

Z.B.: Funktionen mit F als Arg.
Funktionen mit F als Return

↓
Closures

\Leftrightarrow Funktionale Programmierung

Lambda

„anonyme Funktion“

lambda par : Codezeile
mit auto-return

X = lambda a, b: a + b

print(X(4, 5)) → 9

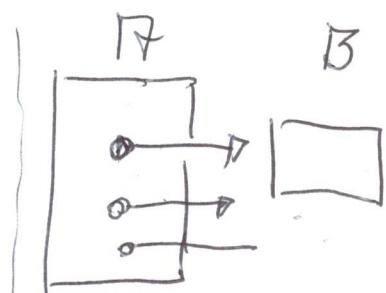
Einsatz bei generischen Fkt

Als Injection bei Fkt → sort
fw Anwendung

Als Filter



has a | is a



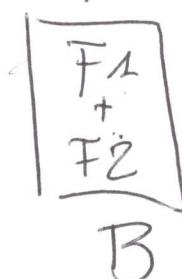
A has a B

Pattern:

"Facade"



is a

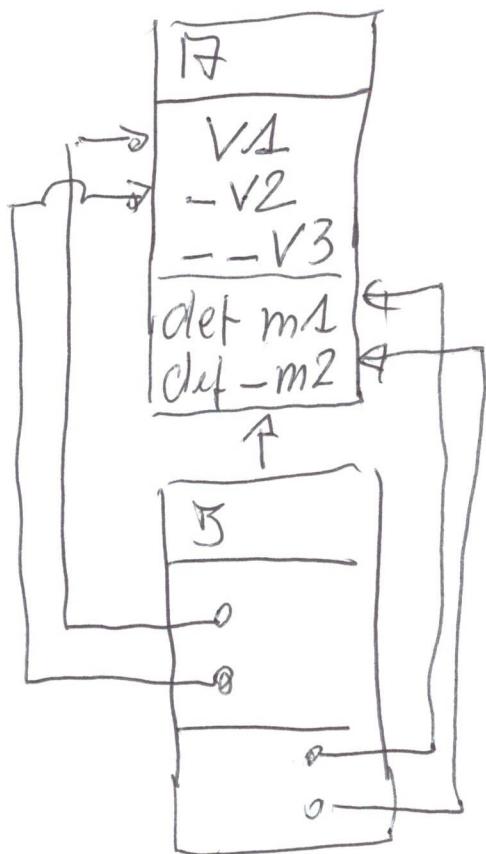


ergänzen
modifizieren
entfernen, bzw
Zugriffstyp ändern

B is a A

Vererbung

Was wird vererbt in py?



-init-() ↗

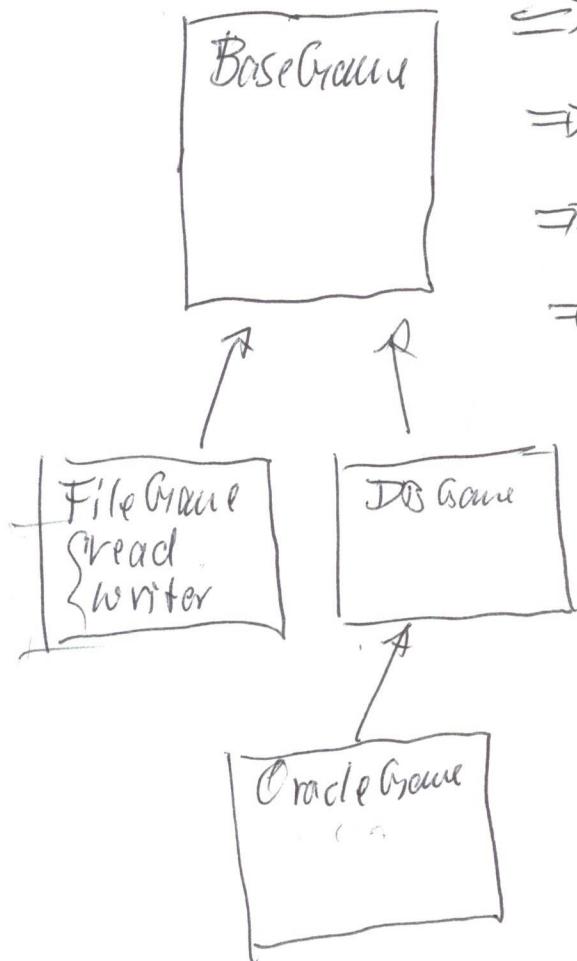
super()

call ↗

-init-():
super().__init__()
self ↗

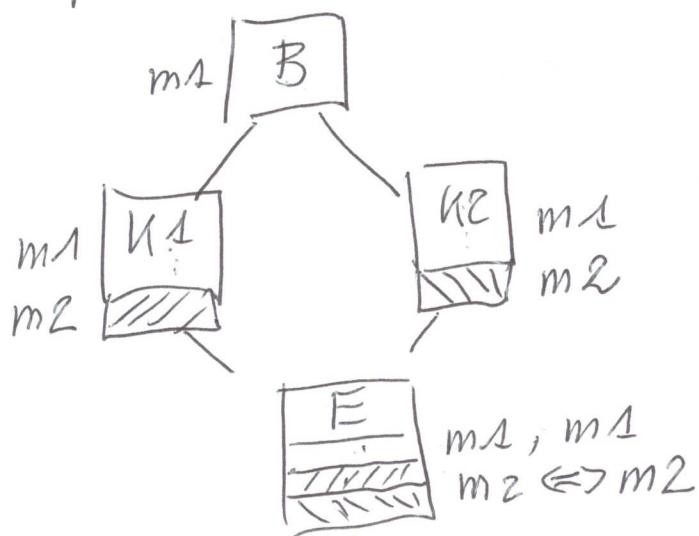
Probleme

- nicht passendes Verfahren,
z.B. has a wäre besser gewesen

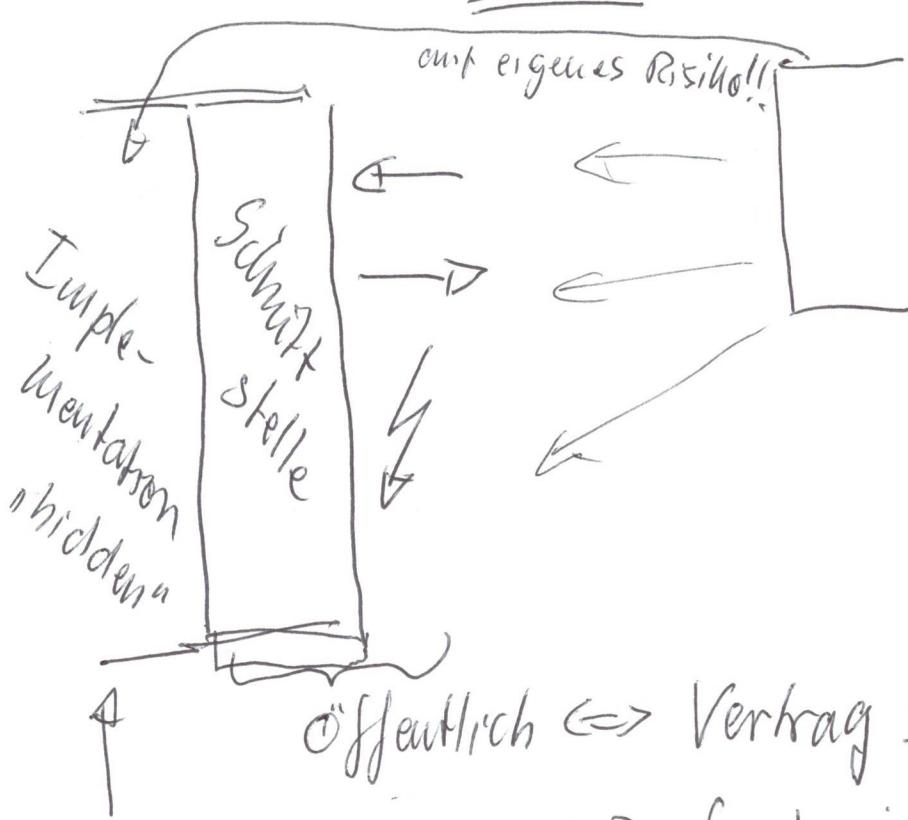


⇒ Logisch nicht zusammengehörig
⇒ Für jede Variable eine Vererbung
⇒ Tiefe Klassenhierarchie
⇒ Sehr feinfühlige Klassen

- py-spezifisch: Mehrfachvererbung



Schnittstellen



Öffentlich \Leftrightarrow Vertrag mit dem Nutzer

\Rightarrow fast immutabel?

- Probleme:
- Änderungen müssen gemacht werden
 - \rightarrow Ergänzungen ✓
 - \rightarrow Lösungen \Rightarrow deprecated
aus deprecated Modul
 - \Rightarrow Docstring vermerken
 - Wuchern gerne!
 - \hookrightarrow neues kommt, altes bleibt
 - \hookrightarrow convenience-Methoden
 - \hookrightarrow Setter/getter

Decorator

Anti pattern

```

@d1
@d2(..., ...)
@d3()
@d4(..., ...)

def machwas(a, b):
    """
    """

```

warum so viele

in Klassen hilfend die Ergänzung, v.a.
wenn diese tempor. sein sollen oder
an DB Systeme nachträglich eingefügt

z.B @logging(logfile)
def machwas(a, b):

"""

decorator + funktion + a, b

[Vorarbeiten]

[Erzeugt Aufruf-Closure für Funktion]

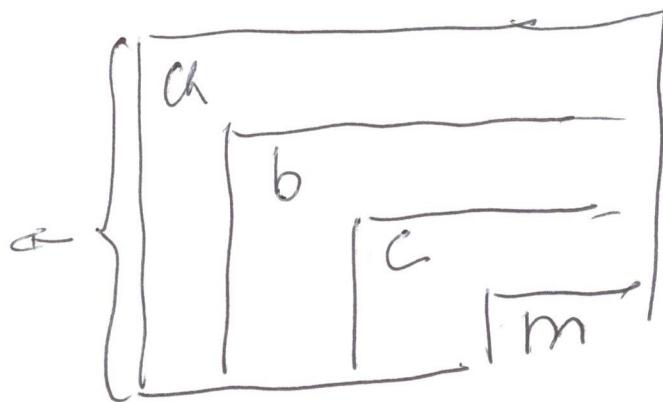
[Nacharbeiten]

return den Closure

Multi declarator

=

@a
@ b
@ c
def m



m() \rightarrow a \rightsquigarrow b \rightsquigarrow c \rightsquigarrow m

=

Generator / Iterator / Iterable

$l = []$ Schrittweise durchlauft bar \Leftrightarrow Iterable

range() erzeugt Iterator-Objekt

Generatorfunktion erzeugt bei jedem Aufruf ein Ergebnis

Bsp my-numbers(start-value);

```

 $i = \text{start-value}$ 
while True:
    yield i
    i += 1
  
```

class Walker:

def __init__(self, start)
 self._start = start

def __iter__(self):
 self._index = \emptyset

def __next__(self):
~~self._start + 1~~; self._index += 1
return (self._index, self._start + self._index)

Testen

Testen, Testen, Testen!

Häufig: „Entwicklerspontanfest“ + point
nur 1 = Manuelle Test

Zusätzlich Butoxan. Testen

much

L	Testfälle Testdaten Drittsysteme	ausdeihen besorgen <u>entbinden</u>
---	--	---

Oder möchten, stützen,

Testfall coden fallen

\Leftrightarrow a) normal Code, evtl.

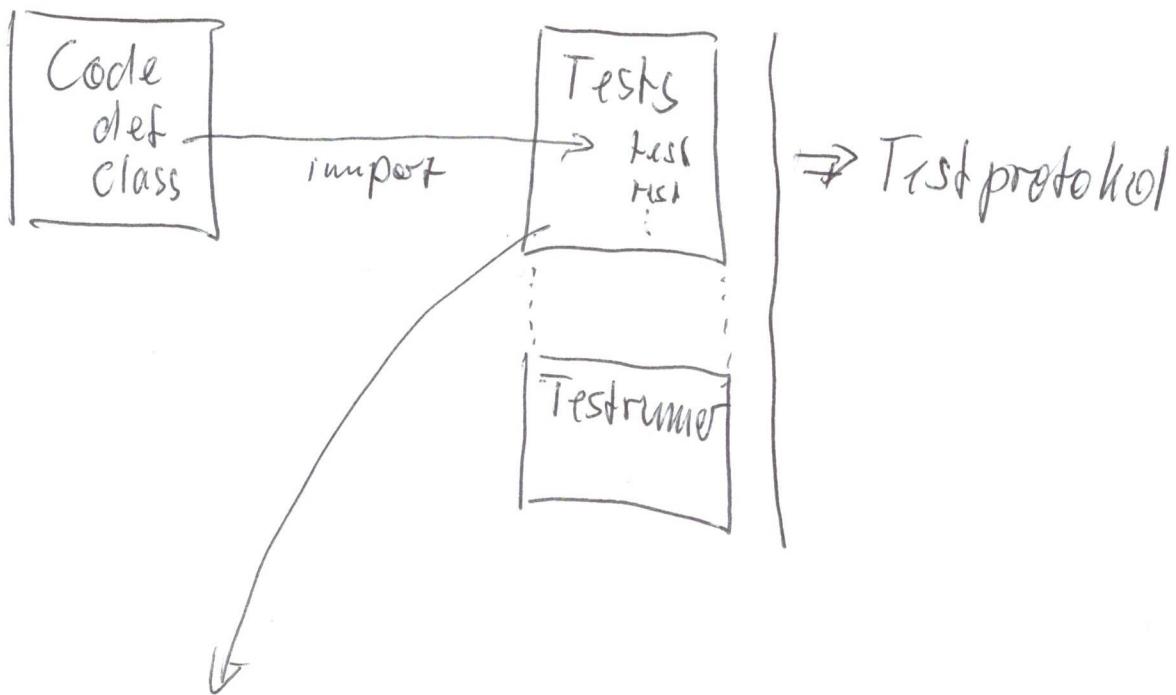
in .if ~-name-- = " - more ~"

b) xUnit-Framework

unittest

pytest

Tesfen



Testklasse (Framework Testklasse)

Setup ← Vorbereitung

test1 \Leftrightarrow assert BedL; False == Fail
True == Ok

Tests

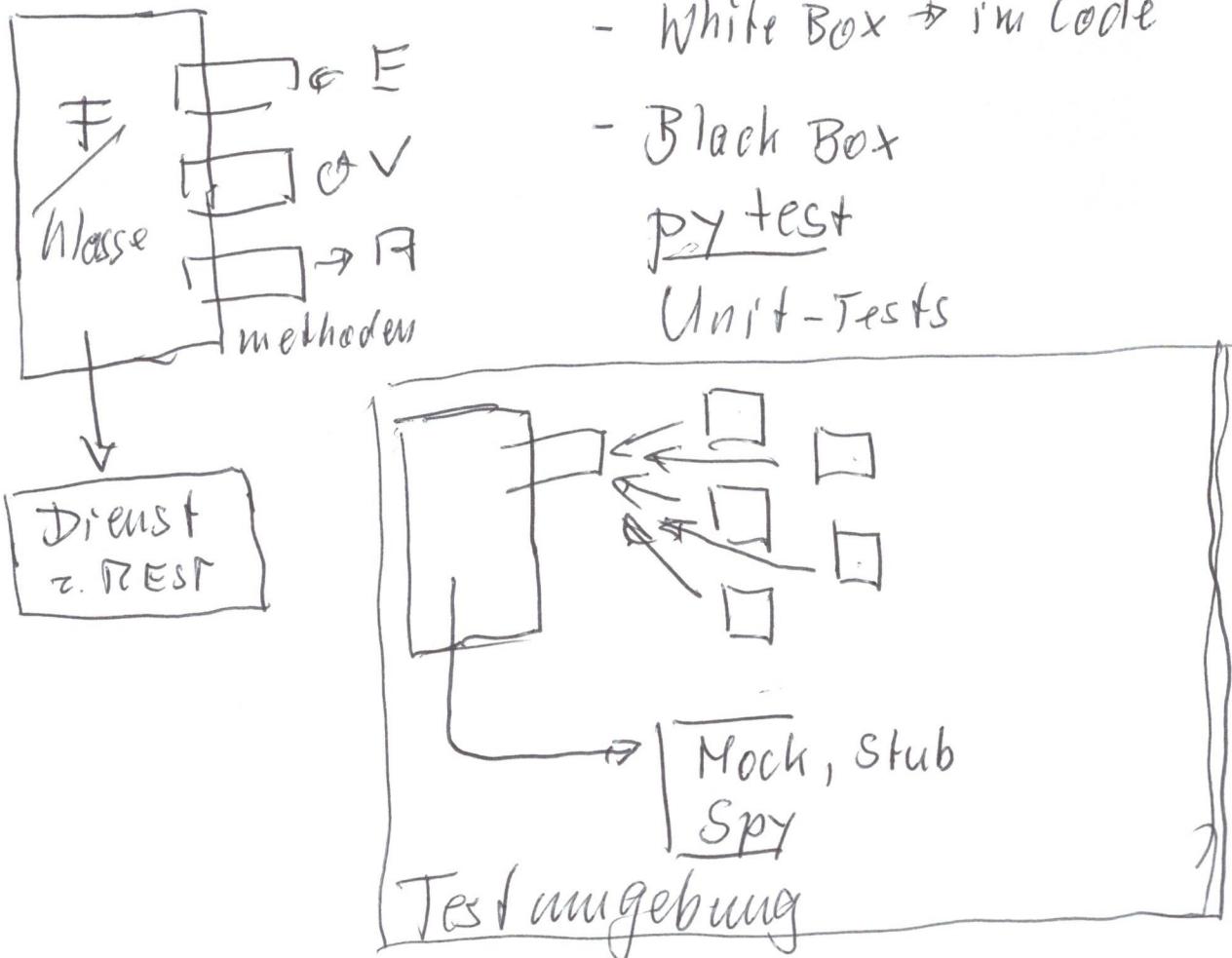
test2

teardown ← Nachbereitung
Frauen

pro Test nur 1 Sache testen / assert

jeder Test ist von anderen unabhängig

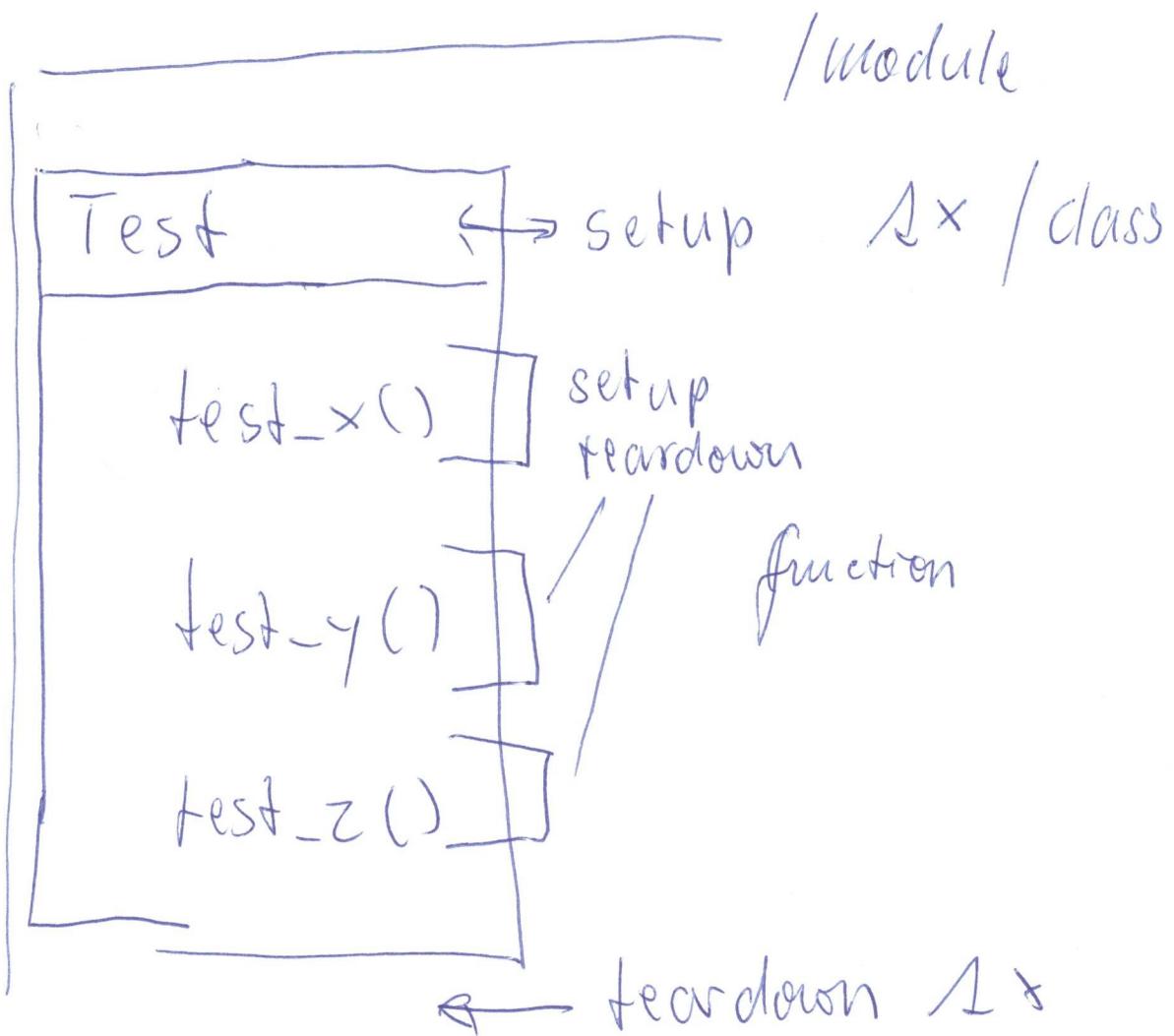
Testvorgehen



* Beispiel für Tag 3

pytest
→ mocking

Fixture Scopes



Setup m

Setup c

Setup f

test_x

Teardown f

Setup f

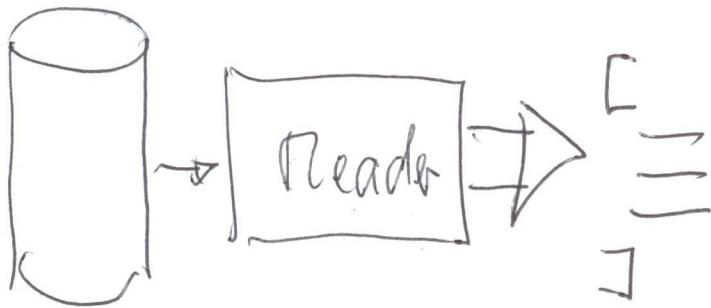
test_y

Teardown

Teardown c

Teardown m

Mocks



class IpFinder:

```
def __init__(self, reader, pattern)
```

~~self~~

```
    self.reader = reader
```

:

```
def read_log(self)
```

```
    self.log = reader.readfile()
```

Test ohne Reader zu realisieren

→ Mock → readfile → gibt mir [-,-,-]

r=Mock()

r.readfile() → get([["un", "un", "un"]])

Aufgabe

Programm mit flexibler Ein- und Ausgabeschnittstelle

Eingabe → 6 Zahlen

→ Prüfung → 6 aus 49 / kein Doppelten / int

↓
Tipp

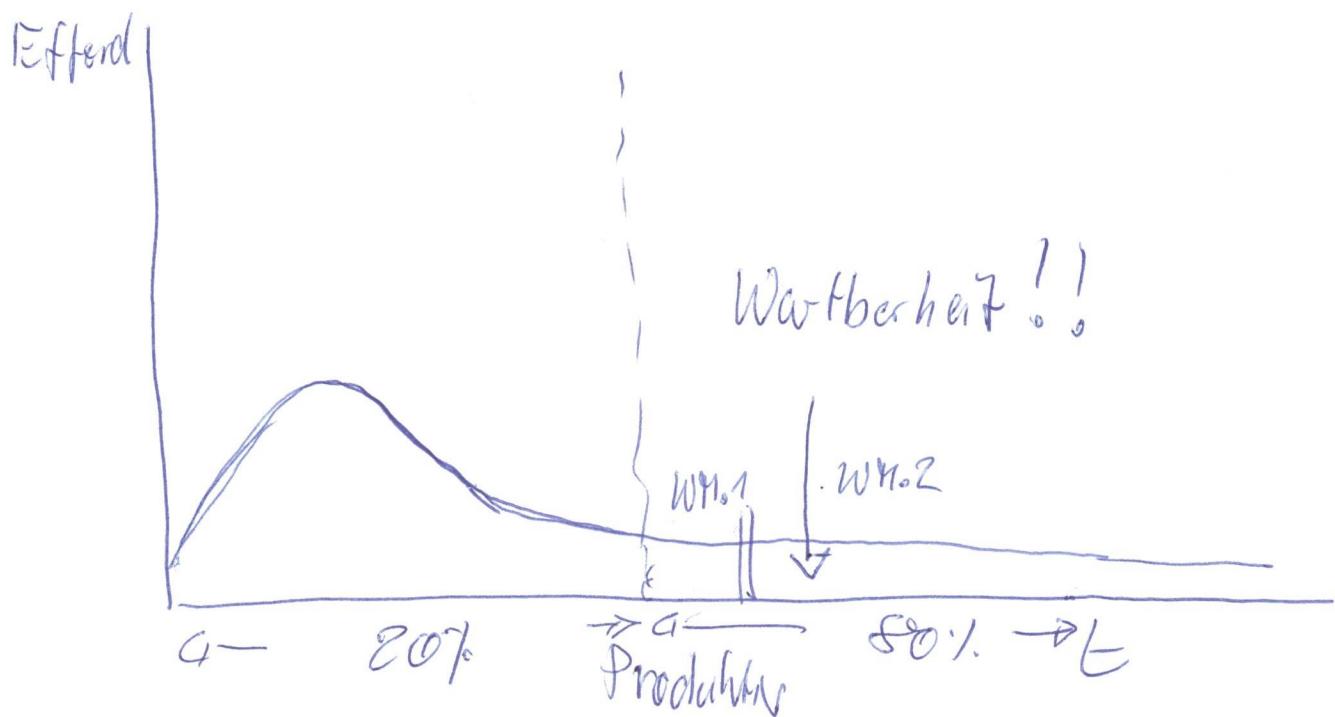
Ziehung
↓
Ziehung

Vergleich

{ Ausgabe
↓ ↓ ↓
Datei Stelout ...

Tests?

Programm leben



Wartung Materialien:
=

* Quellcode

↳ vollständig

↳ lesbar == Clean Coding

* Test Code!

↳ Testdaten

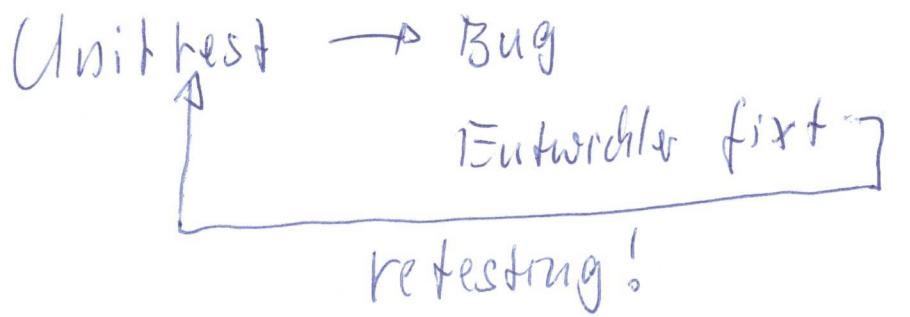
↳ Tests: xUnit, ggf: Funktionsale
Tests
Nichtfunktionsale
Tests

* Protokolle & Dokumente

z.B Handbücher => pydoc/

nach Änderungen zu aktualisieren! docstrings

Bug solving



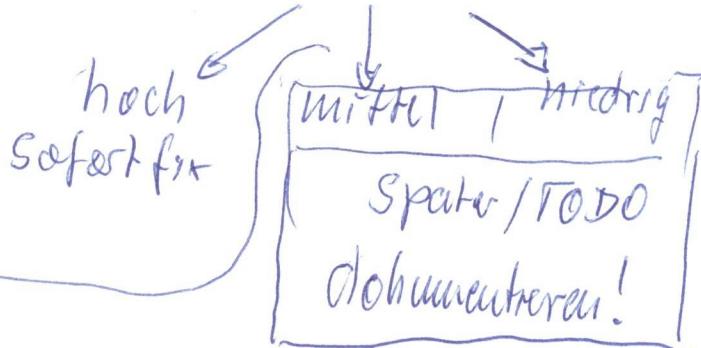
Funktions Test

Testfallbeschreibung

Durchfahrt → Bug / Bug tracking

Bewerten: Risiko, Schwere

Pro



Smells

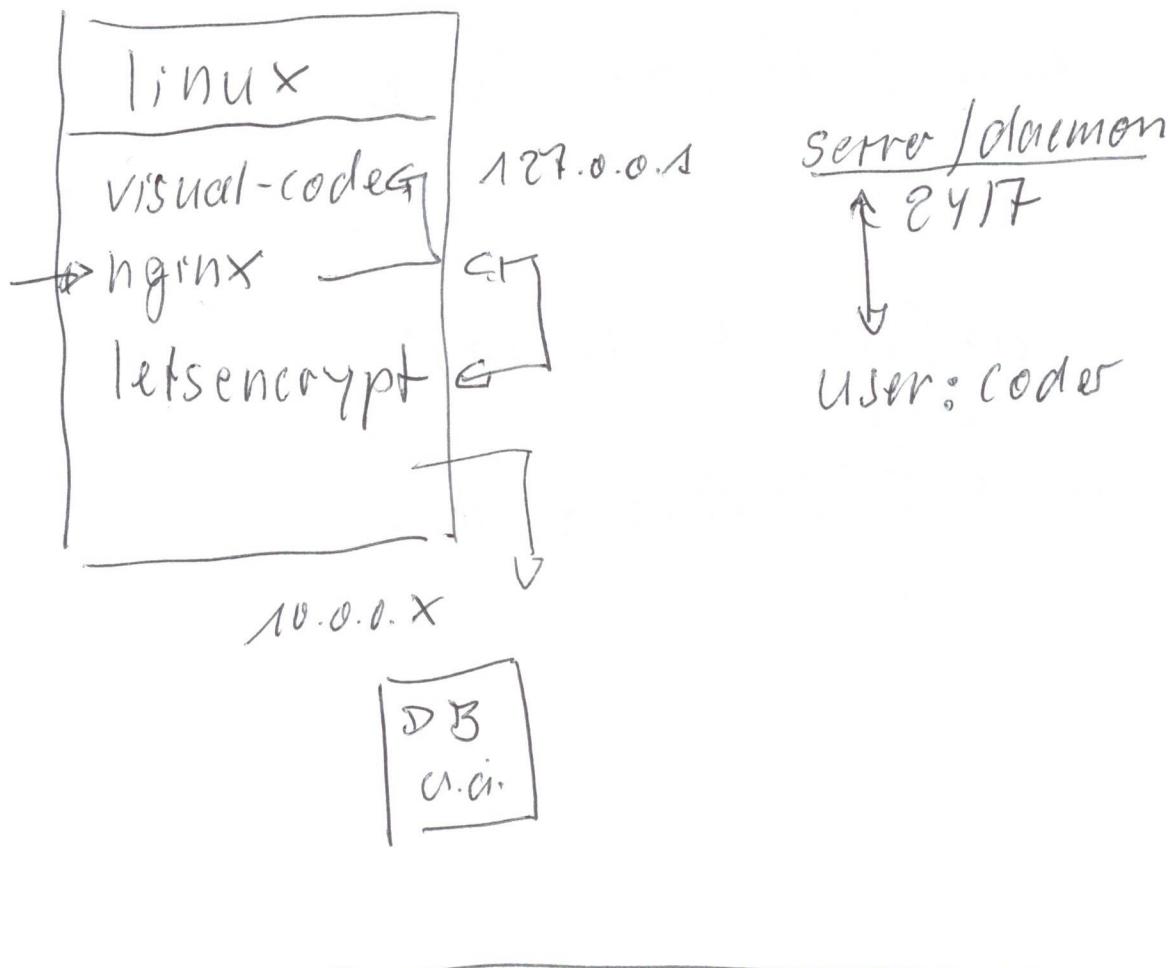
Verschobene zeit
of -n- fixings

Technische Schulden!

Beseitigung techn. Schulden

- ⇒ ⚡ garnicht erst ~~zulassen~~ zu lassen!
- Refactoring \Leftrightarrow Regressionstesting
↓
Testautomatisierung
- ⇒ ⚡ während der Entwicklung
 - "Pfadfinderregel"
 - Refactoring Truebox
= TODOs abarbeiten \Rightarrow techn. Schuld verringern
 - Peer Reviewing /
Pair Review / Programming

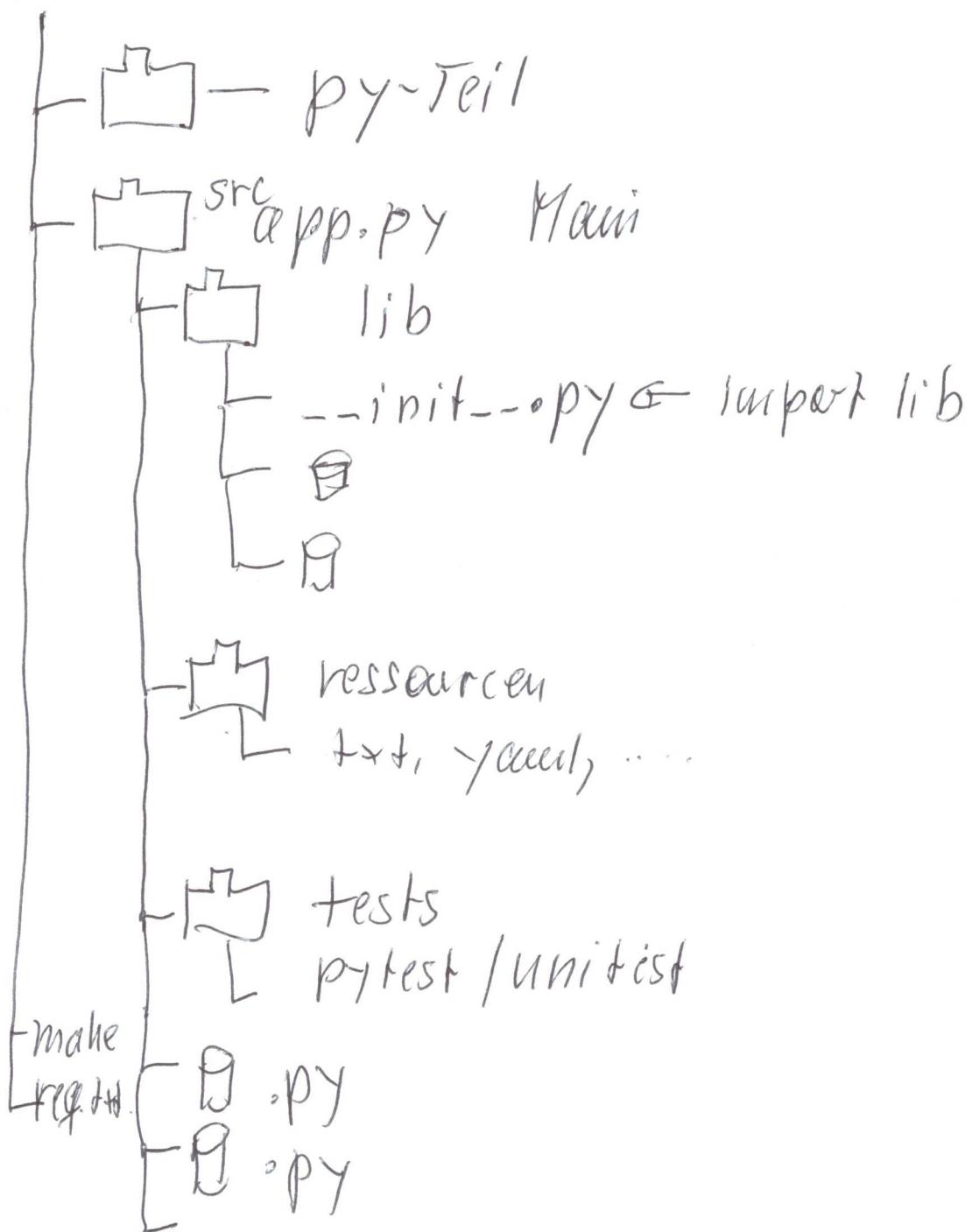
VS code / Code-server



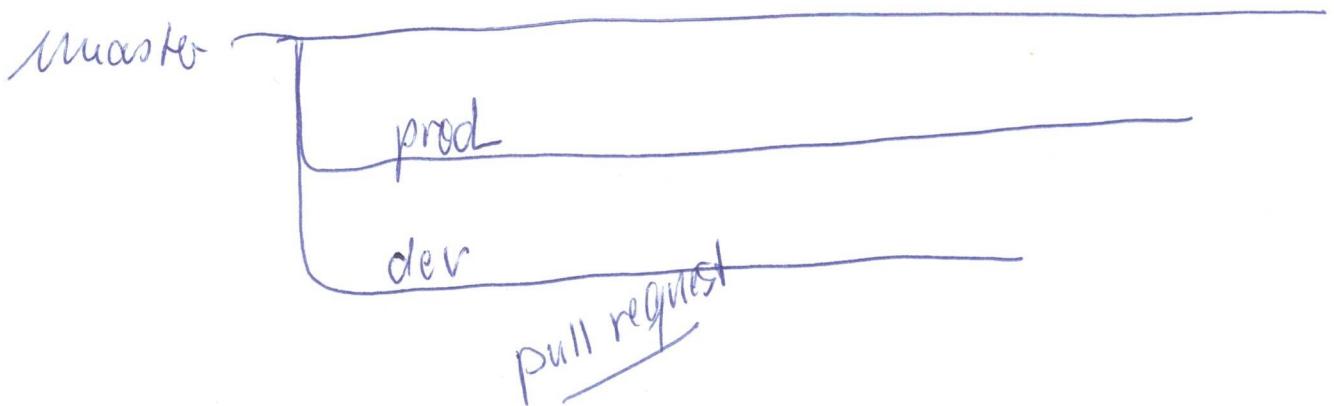
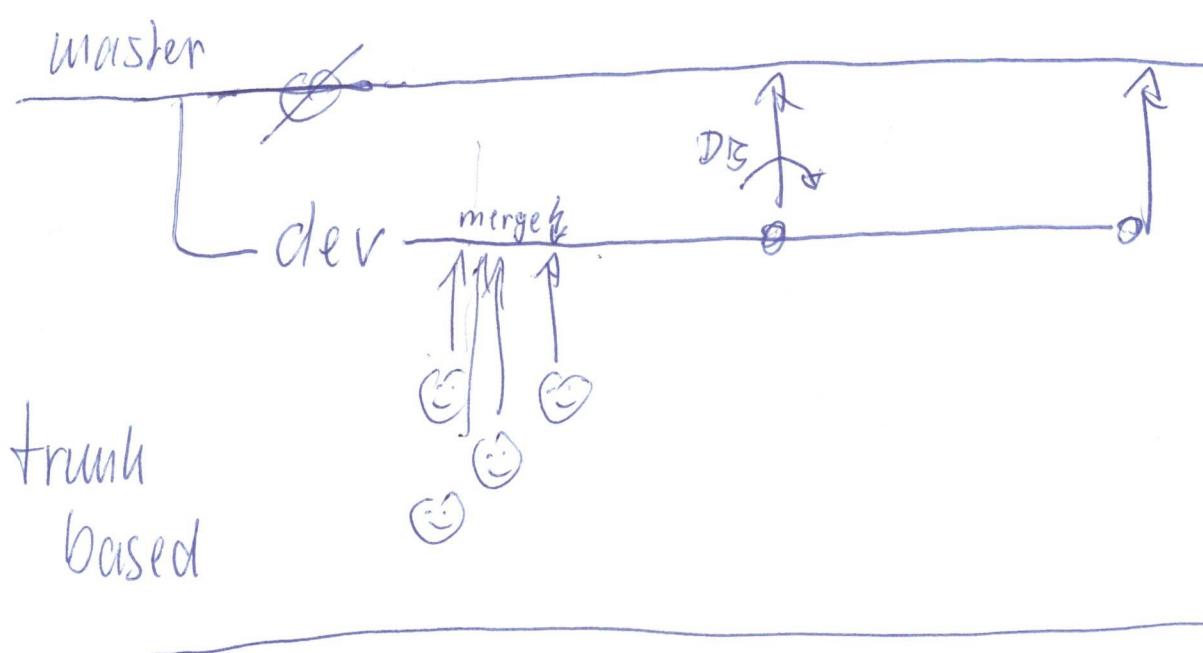
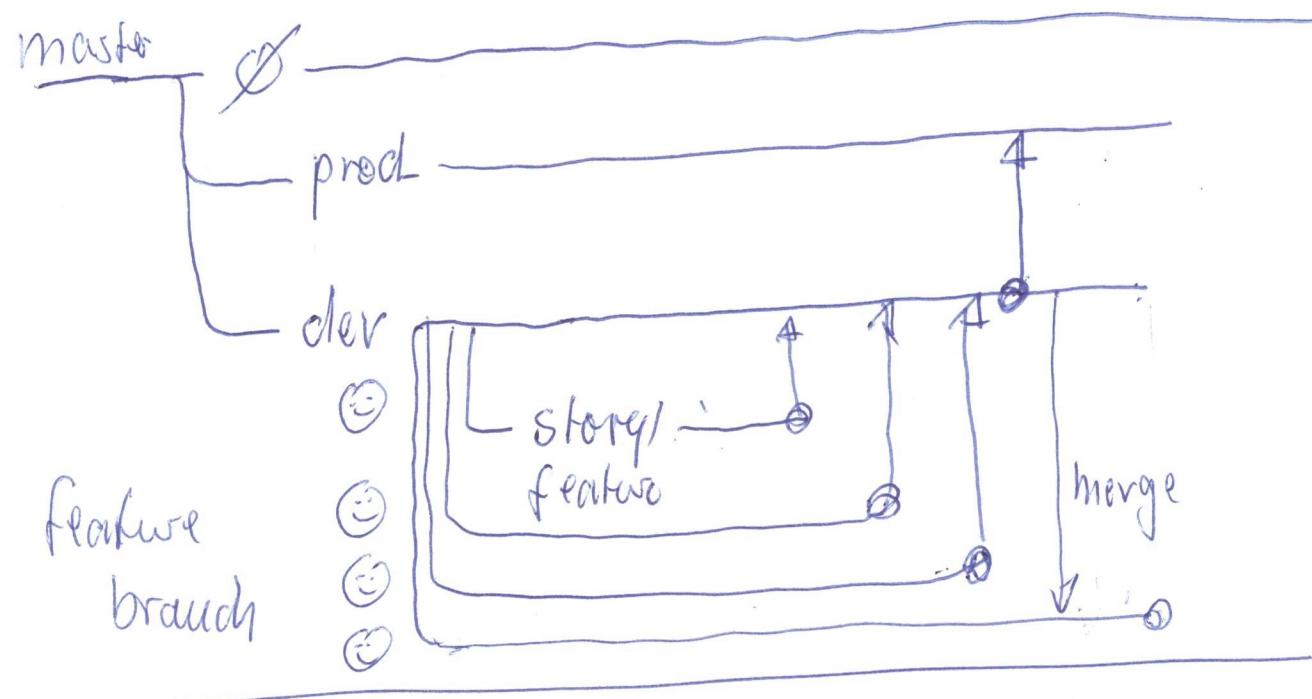
Programm

```
import  
Konst  
def/class  
if name==main
```

Virtuenv ?



(git)



Bsp

"Lotto bude"

Wer — Kunde Adressen etc

Was — Tipp $\frac{6 \text{ aus } 49}{\text{keine Doublets}}$

Von Wo? → Testzwecke Mock
Produktion HTML Seite

Funktion: Lese Tipp

Ziehung → 6 aus 49 | Random

Analysse Treffer →

Ergebnisausgabe → Testzwecke Stdout
Produktion HTML

Funktionen:

read-Tipp(source, user)

tipp = source.get-Tipp()

user = user

Validate-Tipp()

: Fehlerbehandlung?