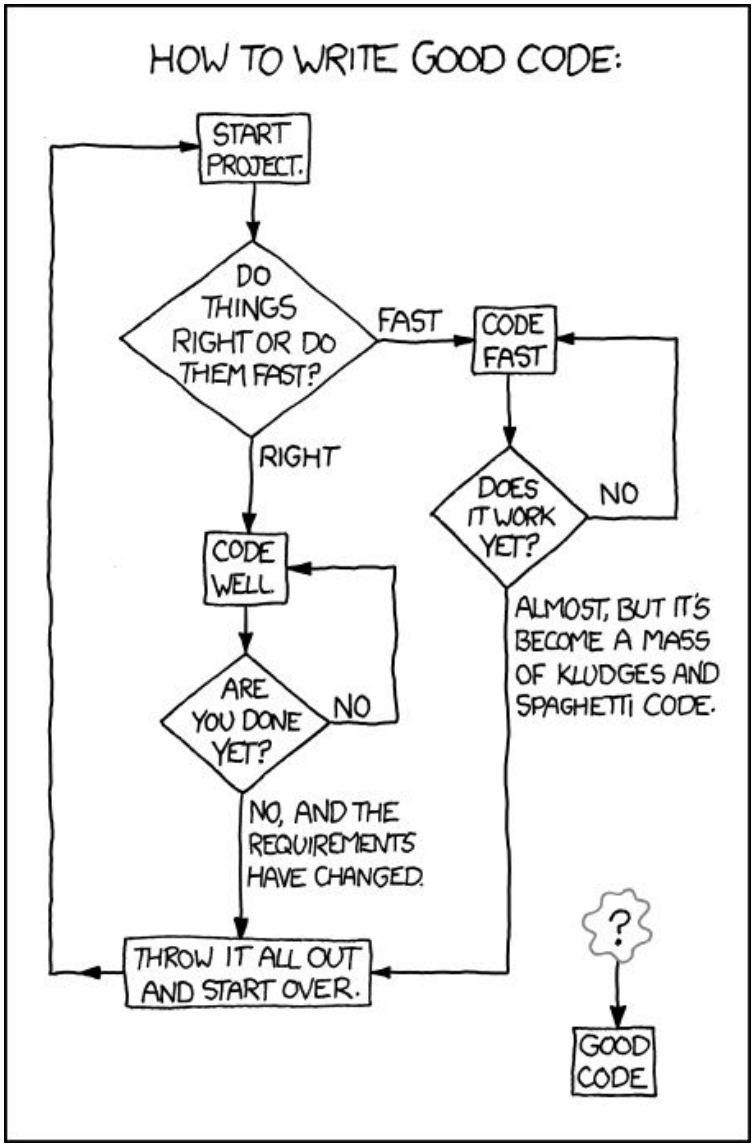
Clean Code

Inhalt

- Metaclasses in Python
- Decorators
- Beziehungen zwischen Objekten
- Best Practices



__new__ vs __init__

```
class T:
    pass

t = T()
```

__new__ vs __init__

```
class T:
    pass

t = T()
```

- wenn T() auftaucht, wird der Interpreter __call__ der Basisklasse aufrufen
- falls keine Basisklassen definiert sind, ruft der Interpreter type.__call__
- danach werden __new__ und __init__ aufgerufen
- __new__ gibt die neue Instanz zurück und __init__ funktioniert wie ein Konstruktor und initialisiert die Attribute der Klasse

__call__

```
class T:
    def __init__(self):
        print ('T::init() called...')

    def __call__(self):
        print ('T::call() called...')

t = T()

t()
```

- direkter Aufruf mit einem Objekt

```
T::init() called...
T::call() called...

Process finished with exit code 0
```

Metaklassen

“Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why).”

- random python guru

Everything in Python is an object because Python has no "primitive"
or "unboxed" types, no machine values. This has nothing to do with
metaclasses. Even if Python had no metaclasses, it could still be true
that "everything is an object".

To summarise:

(1) In Python, "everything is an object" (that is, all values are objects)
is because Python does not include any primitive unboxed values. Anything
which can be used as a value (int, str, float, functions, modules, etc) are
implemented as objects.

(2) Python *also* has metaclasses. This does not follow from the above --
they are independent statements about the language. We could, if we want,
invent new languages:

- one where everything is an object, just like Python, but lacking
metaclasses;

- one which has metaclasses, just like Python, but not everything is an
object;

- and one which *neither* has metaclasses, *nor* everything is an object.

Old-Style(Python2) vs New-Style Classes(Python3)

- eine Instanz einer Old-Style Klasse ist implementiert von einem instance Typ
- für eine Instanz obj, obj.__class__ aber type(obj) ist immer istance

```
Class T:
    pass

>>> t = T()
>>> t.__class__
                <class __main__.T at 0x...>
>>> type(t)
                <type 'instance'>
```

Old Style(Python2) vs New Style Classes(Python3)

- New-Style Klassen vereinigen Klass und Typ

```
class T: #class T(object): / class T():
   pass

>>> t = T()

>>> t.__class__
        <class '__main__.T>
>>> type(T)
        <class '__main__.T>
>>> t.__class__ is type(t)
        True
```

Old Style(Python2) vs New Style Classes(Python3)

```
for t in int, float, dict, list, tuple:
    print(type(t))
```

```
<class 'type'>
<class 'type'>
<class 'type'>
<class 'type'>
<class 'type'>
```

```
>>>type(type)
<class 'type'>
```

Metaklassen

```
>>> type(type)
        <class 'type'>
```

- type ist eine Metaklasse, deren Instanzen sind Klassen

Metaklassen

- alles ist in Python ein Objekt!
- Klassen in Python sind auch nur Objekte
  - also Instanz einer Klasse
- Die Klasse einer Klasse ist type

Metaklassen

- die Klasse einer Benutzerdefinierten Klasse ist also auch type
- man kann eine Metaklasse benutzen, um eine Klasse zu erzeugen

Wie erzeugt man dynamisch eine Klasse

- man kann type mit drei Argumenten aufrufen
  - <name> - der Name der Klasse. In __name__ gespeichert
  - <bases> ein Type von Basisklassen. In __bases__ gespeichert
  - <dct> ein Dictionary, der den Klassenrumpf repräsentiert. In __dict__ gespeichert

- erzeugt eine neue Klasse (Instanz von type)

Metaklassen

```
T = type('T', (), {})
```

```
>>> t = T()
>>> t
<__main__.Foo object at
0x0...>
```

```
class T:
   pass
```

```
>>> t = T()
>>> t
<__main__.Foo object at
0x0...>
```

Metaklassen

```
U = type('U', (T,),
dict(t=10))
```

```
>>> u = U()
>>> u.t
   10
>>> u.__class__
<class '__main__.U'>
>>> u.__class__.__bases__
(<class '__main__.T'>,)
```

```
class U(T):
   t=10
```

```
>>> u = U()
>>> u.t
   10
>>> u.__class__
<class '__main__.U'>
>>> u.__class__.__bases__
(<class '__main__.T'>,)
```

Metaklassen

```
T = type('T', (),
dict(t=10,
get_t=lambda x: x.t))
```

```
>>> t = T()
>>> t.t
   10
>>> t.get_t()
   10
```

```
class T:
   t=10
   def get_t(self):
      return self.t
```

```
>>> t = T()
>>> t.t
   10
>>> t.get_t()
   10
```

Metaklassen

- und genau das passiert auch intern
- Python benutzt nach dem Parsen der Definition einer Klasse type um die Klasse zu erzeugen

Metaklassen

```
class T(object):
    t = 10

class T(object):
    __metaclass__ = type
    t = 10

class Metaklass(type):
    pass
class T(object):
    __metaclass__ = Metaklass
    t = 10
```

Metaklassen (Python2 Syntax)

```
class SimpleMetaClass(type):
    def __new__(cls, name, bases, attr):
        return type(name, bases, attr)

class T(object):
    __metaclass__ = SimpleMetaClass
    def __init__(self):
        pass
```

Metaklassen (Python3 Syntax)


```
class SimpleMetaClass(type):
    def __new__(cls, name, bases, attr):
        return type(name, bases, attr)

class T(object, metclass=SimpleMetaClass):
    def __init__(self):
        pass
```

Metaklassen (Logger Beispiel)


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

    def __repr__(self):
        return f"Point(x={self.x}, y={self.y})"
```

Metaklassen (Logger Beispiel)


```
class BadLogger(type):
    def __new__(metacls, cls, bases, classdict):
        print(f"classname: {cls}")
        print(f"baseclasses: {bases}")
        print(f"classdict: {classdict}")

        return super().__new__(metacls, cls, bases,
classdict)
```

Metaklassen (Logger Beispiel)

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

    def __repr__(self):
        return f"Point(x={self.x}, y={self.y})"

p = Point(1,2)
```

```
classname: Point
baseclasses: ()
classdict: {'__module__': '__main__', '__qualname__': 'Point', '__init__': <function Point.__init__ at 0x7fe571dd20d0>, '__repr__': <function Point.__repr__ at 0x7fe571dd2158>}

Process finished with exit code 0
```

Metaklassen

- Eine Klasse, die keine Unterklasse haben darf

```
class FinalClass(type):
    def __new__(metacls, cls, bases, classdict):
        for typ in [type(base) for base in bases]:
            if typ is metacls:
                raise RuntimeError("no subclassing...")
        return super().__new__(metacls, cls, bases,
classdict)
```

Metaklassen

- Eine Klasse, die keine Unterklasse haben darf

```
class T(metaclass=FinalClass):
    pass

class U(T):
    pass

u = u()
```

```
/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 143, in <module>
    class U(T):
  File "/Users/cat/Documents/python/tttt/m.py", line 138, in __new__
    raise RuntimeError("no subclassing...")
RuntimeError: no subclassing...
```

Dekoratoren

- Funktionen sind in Python 1st-Class-Citizens, wie die anderen Objekte
- d.h man kann eine Funktion
  - als Parameter übergeben
  - zurückgeben
- man kann eine Funktion innerhalb einer anderen definieren
- aber wie und wieso ist das wichtig

Dekoratoren

```
def total_price(price):
    def tax(price):
        return 10*price
    return price + tax(price)

print(total_price(100))          → OK
print(tax())                     → Fehler
```

- die innere Tax-Funktion ist nicht von außen aufrufbar

Dekoratoren

- man kann aber auch eine Funktion zurückgeben

```
def outer(param):
    def inner_1(): print('outer::inner1()')
    def inner_2(): print('outer::inner2()')

    return inner_1 if param == 1 else inner_2

f = outer(1)

print(f)
f()
```

```
<function outer.<locals>.inner_1 at 0x7fb9e341ddc0>
outer::inner1()
>
```

Dekoratoren

- ein Dekorator packt eine Funktion ein und dadurch ändert das Verhalten

```
def decorator(f):
    def wrapper():
        print("func() is being called...")
        f()

    return wrapper
def hello(): print("hello world")

decorated_hello = decorator(hello)
decorated_hello()
```

```
func() is being called...
hello world
>
```

Dekoratoren

- die Syntax ist verbose

```
decorated_hello = decorator(hello)
decorated_hello()
```

- Python bietet eine alternative Syntax für die Definition dekorierter Methoden

- man könnte `@` verwenden

Dekoratoren

```
def decorator(f):
    def wrapper():
        print("func() is being called...")
        f()

    return wrapper


def hello(): print("hello world")

hello()
```

```
hello world
```

Dekoratoren

```
def decorator(f):
    def wrapper():
        print("func() is being called...")
        f()

    return wrapper

@decorator
def hello(): print("hello world")

hello()
```

```
func() is being called
hello world
```

Dekoratoren mit Parametern

```
def total_price(f):
    def wrapper(*args, **kwargs):
        return 10 + f(*args, **kwargs)

    return wrapper


@total_price
def price(wo_tax): return wo_tax

print(price(100))                    110
```

Dekoratoren mit Parametern


```
def total_price(f):
    def wrapper(*args, **kwargs):
        return 10 + f(*args, **kwargs)

    return wrapper

@total_price
def price(wo_tax): return wo_tax

print(price)
print(price.__name__)
```

Dekoratoren mit Parametern

- man gibt eine total andere Funktion zurück
- die Identität ist verloren
- @functools.wraps behält die Information über die initiale Identität der Funktion

```
<function total_price.<locals>.wrapper at 0x7f9fa86e1e50>
wrapper
>
```

Dekoratoren mit Parametern

```
import functools
def total_price(f):
    @functools.wraps(f)
    def wrapper(*args, **kwargs):
        return 10 + f(*args, **kwargs)
    return wrapper

@total_price
def price(wo_tax): return wo_tax

print(price)
print(price.__name__)
```

```
<function price at 0x7f8ef356e4c0>
price
>
```

Dekoratoren für Klassen

```
import functools

def logger(f):
    @functools.wraps(f)
    def wrapper(*args, **kwargs):
        print(f'{f.__name__} is being called...')
        f(*args, **kwargs)

    return wrapper
```

Dekoratoren für Klassen

```
class T:
    @logger
    def f(self): print("T::f()")
    def g(self): print("T::g()")

t = T()

t.f()
t.g()
```

```
f is being called...
T::f()
T::g()
>
```

Dekoratoren für Klassen

- Manchmal brauchen Dekoratoren zusätzliche Parameter
- Die Dekorator-Funktion selbst darf nur die Funktion als Parameter haben

```
def greedy(times):
    def deco(fun):
        def wrap(*args):
            for i in range(times):
                fun(*args)
        return wrap
    return deco
```

Dekoratoren für Klassen

- Manchmal brauchen Dekoratoren zusätzliche Parameter
- Die Dekorator-Funktion selbst darf nur die Funktion als Parameter haben

```
@greedy(2)
def eat(name):
 print(f'{name} is eating...')

eat('bob')
```

```
bob is eating...
bob is eating...

Process finished with exit code 0
```

Dekoratoren für Klassen

- Manchmal brauchen Dekoratoren zusätzliche Parameter
- Die Dekorator-Funktion selbst darf nur die Funktion als Parameter haben
- darf auch implizite Parameter benutzen

```
def greedy(times=1):
    def deco(fun):
        def wrap(*args):
            for i in range(times):
                fun(*args)
        return wrap
    return deco
```

Dekoratoren für Klassen

- Manchmal brauchen Dekoratoren zusätzliche Parameter
- Die Dekorator-Funktion selbst darf nur die Funktion als Parameter haben
- darf auch implizite Parameter benutzen

```
@greedy()
def eat(name):
 print(f'{name} is eating...')

eat('bob')
```

```
bob is eating...

Process finished with exit code 0
```


Clean Code

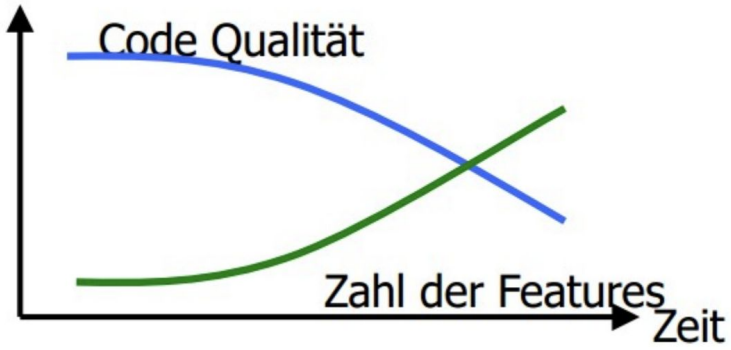
Clean Code

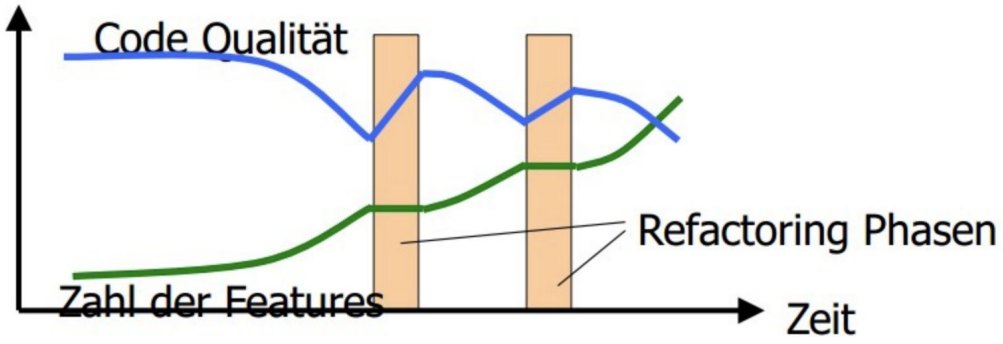

Me
My code works fine
Now time for
Some clean up
Also me
My code works fine
Push it now
Clean it later

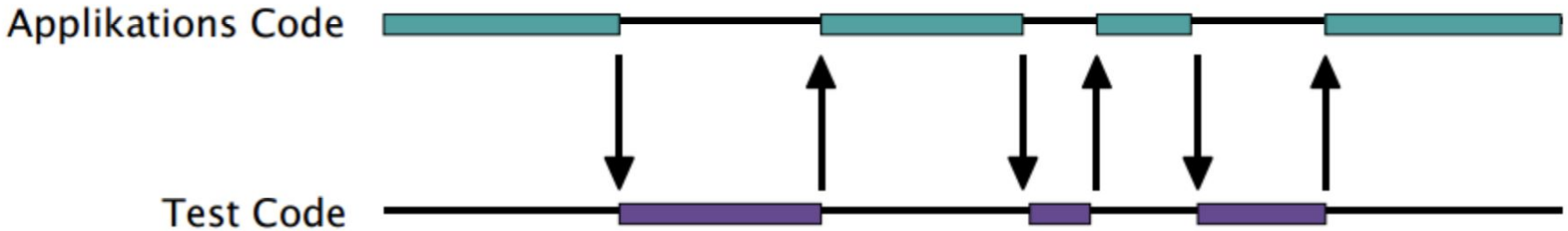
unser Mantra: „Code a little, test a little, refactor a litte!“

Refactoring:

Verbesserung des Codes ohne Änderung des Verhaltens.







Re•fac•to•ring

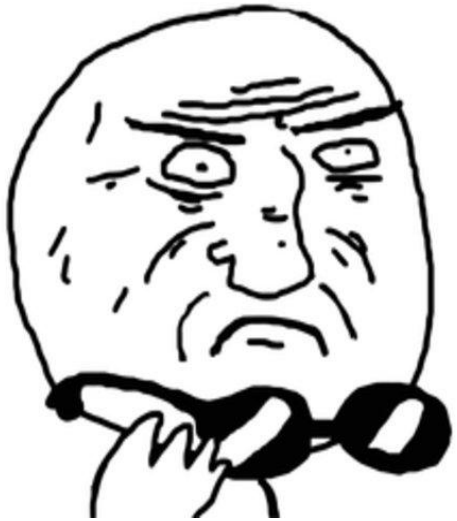
(noun)

„a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior„

-refactoring.com

was bedeutet das genau?

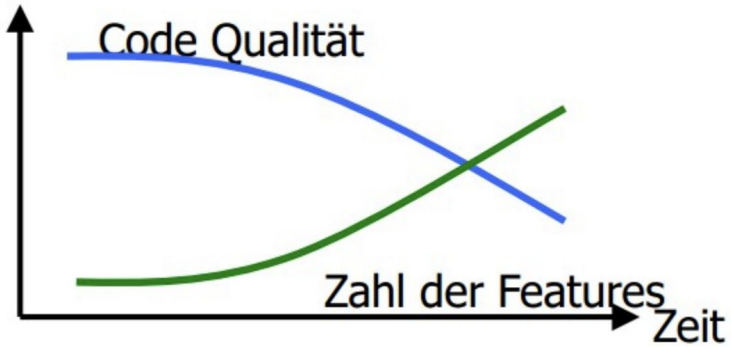
- man versucht den Code nicht direkt perfekt zu schreiben
- der Weg zum Clean Code ist durch Refactoring erreicht
- aber der Code muss immer richtig funktionieren
  - d.h. ohne Bugs ;)
  - testing is wichtig
- ein unendlicher Kreis

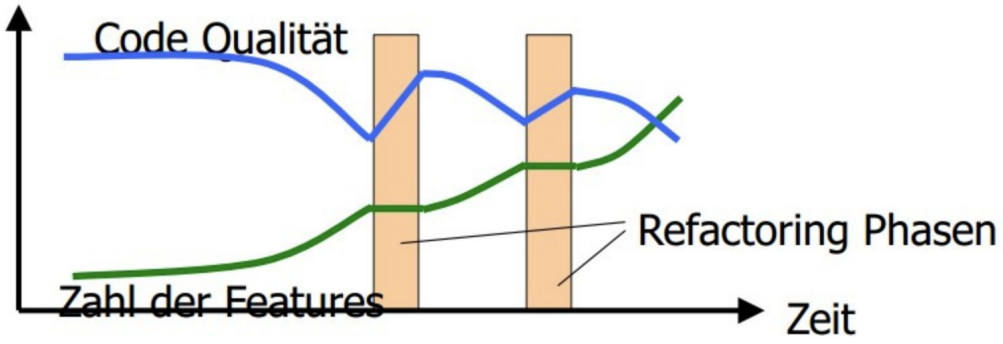


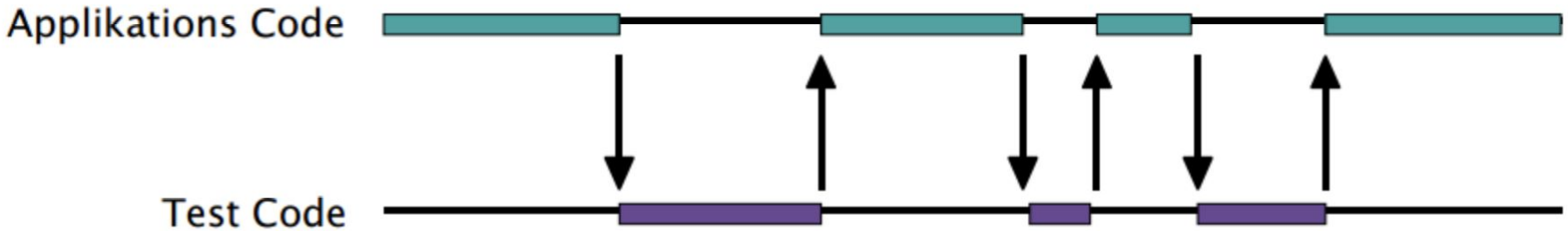
unser Mantra: „Code a little, test a little, refactor a litte!“

Refactoring:

Verbesserung des Codes ohne Änderung des Verhaltens.







4 Prinzipien

- man erstellt Code, den **alle** verstehen können
- die Personen, die den Code reviewen oder verwenden, sollten keine Annahme machen
- manchmal muss man Code löschen (nach dem Motto: *when less is more*)
- Es gibt kein perfekter Code
  - Es gibt immer Raum für Verbesserungen

Ziele des Refactoring

- Lesbarkeit des Codes erhöhen
  - Refactoring kann parallel zu einem Code Review erfolgen
- Design verbessern (sogenannte „Bad Smells“ beseitigen)
- Code so vorbereiten, dass neue Features implementiert werden können

not only good news...

- Refactoring ist riskant
  - Risiko minimieren durch gute Unit Test Abdeckung
  - Testing ist eine Voraussetzung

- Immer in kleinen Schritten:
  - Ein Refactoring Schritt
  - Testen
  - Nächster Refactoring Schritt
  - Testen

- Häufiger Wechsel zwischen Implementation eines neuen Features und Refactoring

Design verbessern/The return: Bad Smell

- ein Anti-Pattern: schlechte Codefragmente, die potenzielle Probleme anzeigen

- Duplizierter Code
  - Hoher Wartungsaufwand da Änderungen überall nachgeführt werden müssen

- Lange Methode / Grosse Klasse
  - Schwierig zu verstehen/hat viele Verantwortungen
  - Schlechte Wiederverwendbarkeit
  - Folge von Code Duplikationen
  - unser Controller: seminar 7-8

The return: Bad Smell

- Lange Parameter Liste
  - Schwierig zu lesen
  - Oft schlechte Wiederverwendbarkeit
  - Gefahr des Vertauschens bei Parametern des gleichen Typs

- Switch Statements bzw. if-else-if Ketten
  - Möglicherweise unflexibel für Erweiterungen
  - Gleichartige Switch Statements: Code Duplikationen
  - main-Methode Seminar 9: wir haben eine neue Methode erstellt

was können wir tun?

Methode extrahieren

- Ein Codefragment kann zusammengefasst werden
- Setze das Fragment in eine Methode, deren Name und Zweck bezeichnet

Motivation:

- Verbesserung der Lesbarkeit
- Codeduplikation: Verschiedene Codefragmente tun (fast) dasselbe.

Methode extrahieren

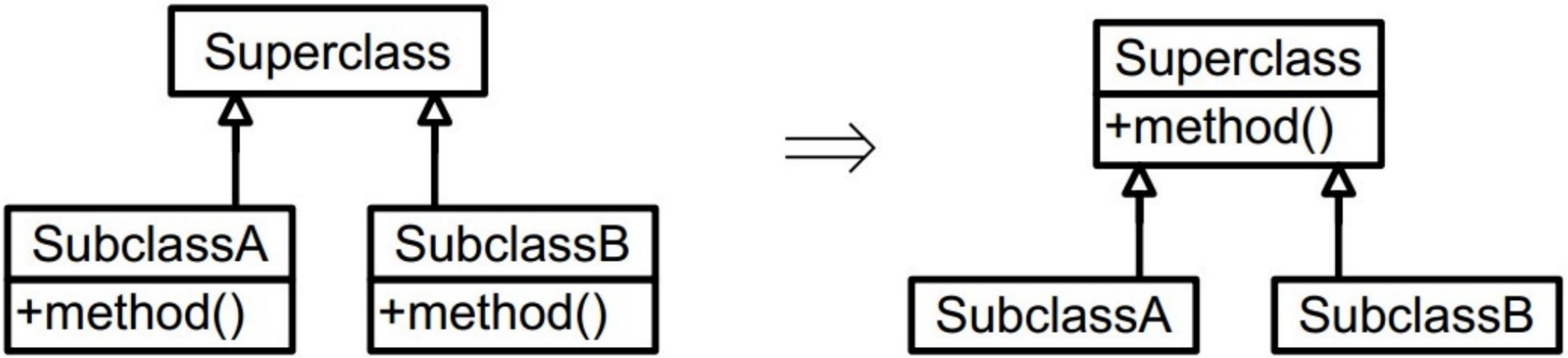
```
void foo()
{
    // berechne Kosten
    kosten = a * b + c;
    kosten -= discount;
}
```

```
void foo()
{
    berechneKosten();
}
```

```
void berechneKosten()
{
    kosten = a * b + c;
    kosten -= discount;
}
```

Methode hochziehen

- Es gibt Methoden mit identischen Ergebnissen in den Unterklassen
- Verschiebe die Methoden in die Oberklasse



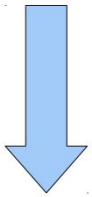
Methode umbenennen

- um die Lesbarkeit zu verbessern
- Der Name einer Methode macht die Absicht nicht deutlich
- Ändere den Name der Methode

Beschreibende Variable

- Es gibt einen komplizierten Ausdruck
- Setze den Ausdruck (oder Teile) in lokale Variable, deren Name den Zweck erklärt.

```
if  platform.toUpperCase().indexOf("MAC") > -1 and
browser.toUpperCase().indexOf("IE") > -1 and wasInitialized() and resize > 0: #stuff
```



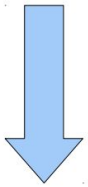
```
isMacOs = platform.toUpperCase().indexOf("MAC") > -1
isIEBrowser = browser.toUpperCase().indexOf("IE") > -1
wasResized = resize > 0;
if isMacOs and isIEBrowser and wasInitialized() and wasResized: #stuff
```

Replace Temp with Query

- typische Situation: eine temporäre Variable speichert das Ergebnis eines Ausdrucks
- man kann den Ausdruck in eine Abfrage-Methode stellen
- und die temporäre Variable durch Aufrufe der Methode ersetzen
- Vorteil: die neue Methode kann in anderen Methoden benutzt werden

Replace Temp with Query

```
basePrice = quantity * itemPrice;
if basePrice > 1000.00:  return basePrice * 0.95
else: return basePrice * 0.98
```



```
if basePrice() > 1000.00:
    return basePrice() * 0.95
else:
    return basePrice() * 0.98

def basePrice():
    return quantity * itemPrice
```

…zu allgemein/idiomatic python

- dos and donts in python
- zur Erinnerung. das Mantra von Python
  - was halten Python-Developers als wichtig

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Readability counts.

idiomatic python…

idiomatic adjective

Save Word

id·i·o·mat·ic | \ ˌi-dē-ə-ˈma-tik \

Definition of *idiomatic*

1 : of, relating to, or conforming to idiom

2 : peculiar to a particular group, individual, or style

Schritte

- step0: Tests
- step1: alles zum Laufen zu bringen
- step2: Code Aufräumen (refactoring)
- step3: Code Reuse
- irgendwo mittendrin werden wir anfangen, *idiomatic python* zu schreiben

List Comperehensions

- erlauben eine neue Collection aus einer Iterable zu erstellen
- allgemeine Syntax

```
[expression for x in iterable if any_condition]
```

```
#avoid
numbers = [1, 2, 3, 4, 5]
gerade_numbers_x2 = []
for number in numbers:
    if number%2 == 0:
        gerade_numbers_x2.append(number*2)

#the idiomatic way
gerade_numbers_x2 = [n*2 for n in numbers if n%2 == 0]
```

List Comperehensions

man kann list comprehensions auch mit `dict` und `set` benutzen

```
words = ['ana', 'are', 'mere']
{word: len(word) for word in words} #dict
```

```
{len(word) for word in words} #set
```

String Formatting/F-Strings

- erlauben Strings einfach zu formatieren
- C-Style String Formatting mit %
- String Verkettung mit dem +-Operator
- format-Methode

```
preis = 110
#avoid
print('Preis=%d' % preis)
print('Preis={preis}'.format(preis=preis))
print('Preis=' + str(preis))

#the idiomatic way
print(f'Preis={preis}')
```

Packing/Unpacking

- eine Definition in einer Zeile
- wenn “variablen” semantisch zusammen gehören

```
#avoid
code = 0
message = 'sucessful'

#the idiomatic way
code, message = 0, 'sucessful'
```

Packing/Unpacking

```
data = ('Bob', 10, 9, 8, 10, 5, 9, 'bbig2434')

#avoid
name = data[0]
noten = data[1:-1]
matrikelnr = data[-1]

#the idiomatic way
name, *noten, matrikelnr = data
```

Enumerable

- typische Situation: wir brauchen den Index und auch den Wert beim durchlaufen einer List

```
words = ['ana', 'are', 'mere']

#avoid
for i in range(len(words)):
    print(f'{i}: {words[i]}')

#the idiomatic way
for i, word in enumerate(words, 0):
    print(f'{i}: {word}')
```

falls Fehl → ist 0 implizit

zip

- typische Situation: zwei Listen gleichzeitig durchlaufen

```
words = ['ana', 'are', 'mere', 'si']
leng = [3, 3, 4]
lengths={}

#avoid
for i in range(min(len(words), len(leng))):
    lengths[words[i]] = leng[i]

#the idiomatic way
for w,l in range(min(len(words), len(leng))):
    lengths[w] = leng[l]
```

Ternary Expression

- initialisierung mit einem if
- allgemeine Syntax

```
var = true_value if condition else false_value
```

```
number = 4
#avoid
if number%2 == 0:
    value = number*2
else:
    value = number

#the idiomatic way
value = number*2 if number%2 == 0 else number
```

Counting Frequency


```
#avoid
ls = ['a', 'a', 'b', 'c', 'a']
d = {}

for el in ls:
    if el in d:
        d[el] += 1
    else:
        d[el] = 1

print(d) # {'a': 3, 'b': 1, 'c': 1}
```

Counting Frequency


```
#avoid
ls = ['a', 'a', 'b', 'c', 'a']
d = {}

for el in ls:
    d[el] = d.get(el, 0) + 1

print(d) # {'a': 3, 'b': 1, 'c': 1}
```

Counting Frequency


```
#the idiomatic way
from collections import Counter

ls = ['a', 'a', 'b', 'c', 'a']
d = Counter(ls)
print(d) # Counter({'a': 3, 'b': 1, 'c': 1})
```