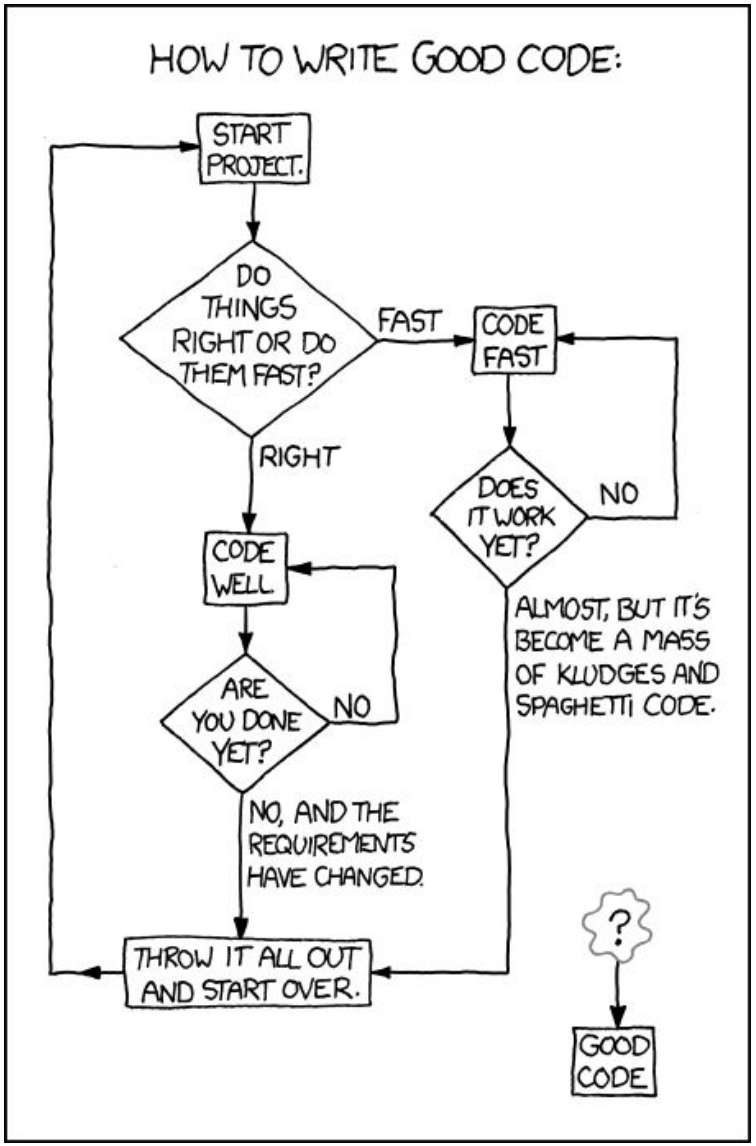
Clean Code

Intro

Inhalt

- Schichtenarchitektur
- GUI
- Assoziationen vs Vererbung
- Testing
- Refactoring
- Clean Code



Dynamische Typisierung

- 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

- '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 Bird:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.__class__.__name__ + ''  + self.name

class Duck(Bird):
    def quak(self):
        print(str(self) + ': quak')

def main():
    ducks  = [Bird('Bob'), Duck('Donald'), object()]
    for duck in ducks:
        try:
            duck.quak()
        except AttributeError:
            print('Keine Ente', duck)
```

```
Keine Ente BirdBob
DuckDonald: quak
Keine Ente <object object at 0x7fbbd6e37070>
```

Dateien und Python

- benötigt man die `open()-Funktion`
- Mit der open-Funktion erzeugt man ein Dateiobjekt
  - und liefert eine Referenz auf dieses Objekt als Ergebniswert zurück

```
open(filename,mode)
Mode: "r" , "w", "a"
```

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

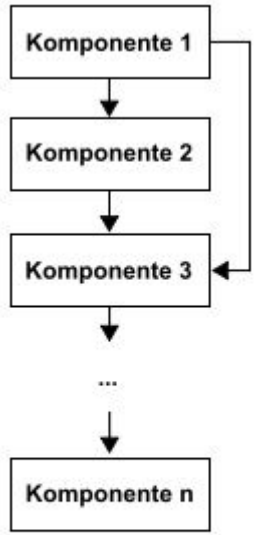
```
studenten = [('Peter', 123), ('Paula', 988), ('Freddi', 851)]

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
```

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

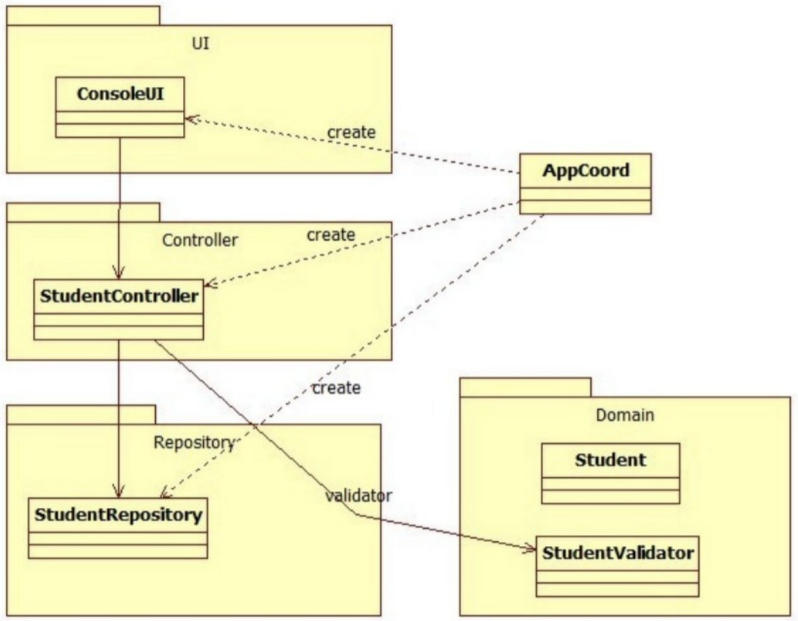
Schichtenarchitektur

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

PSA: GUIs

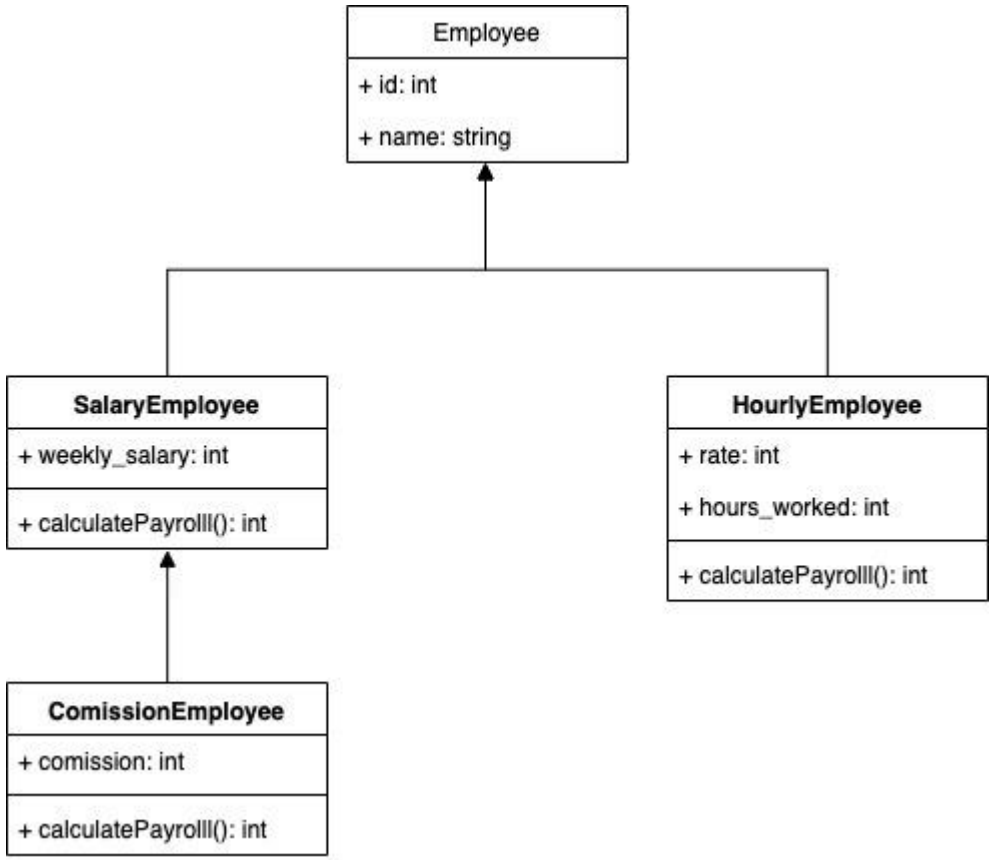
- Tk toolkit

- Tkinter: die Python-Schnittstelle oder Interface zu Tk
- https://python-textbok.readthedocs.io/en/1.0/Introduction_to_GUI_Programming.html
- 2020.V8.zip (code)



Zwischenprüfung - 16.12

Vererbung - ist-a Beziehung



Vererbung - ist-a Beziehung

```python
class Employee:
    def __init__(self, id, name):
        self.id = id
        self.name = name

class SalaryEmployee(Employee):
    def __init__(self, id, name, weekly_salary):
        super().__init__(id, name)
        self.weekly_salary = weekly_salary

    def calculate_payroll(self):
          return self.weekly_salary

class HourlyEmployee(Employee):
    def __init__(self, id, name, hours_worked, hour_rate):
        super().__init__(id, name)
        self.hours_worked = hours_worked
        self.hour_rate = hour_rate

    def calculate_payroll(self):
          return self.hours_worked * self.hour_rate
```

```python
class CommissionEmployee(SalaryEmployee):
    def __init__(self, id, name, weekly_salary, commission):
        super().__init__(id, name, weekly_salary)
        self.commission = commission

    def calculate_payroll(self):
        fixed = super().calculate payroll()
          return fixed + self.commission
```

Vererbung - ist-a Beziehung


```
class PayrollSystem:
    def calculate_payroll(self, employees):
        print('Calculating Payroll')
        print('==================')
        for employee in employees:
            print(f'Payroll for: {employee.id} - {employee.name}')
            print(f'- Check amount: {employee.calculate_payroll()}')
            print('')
```

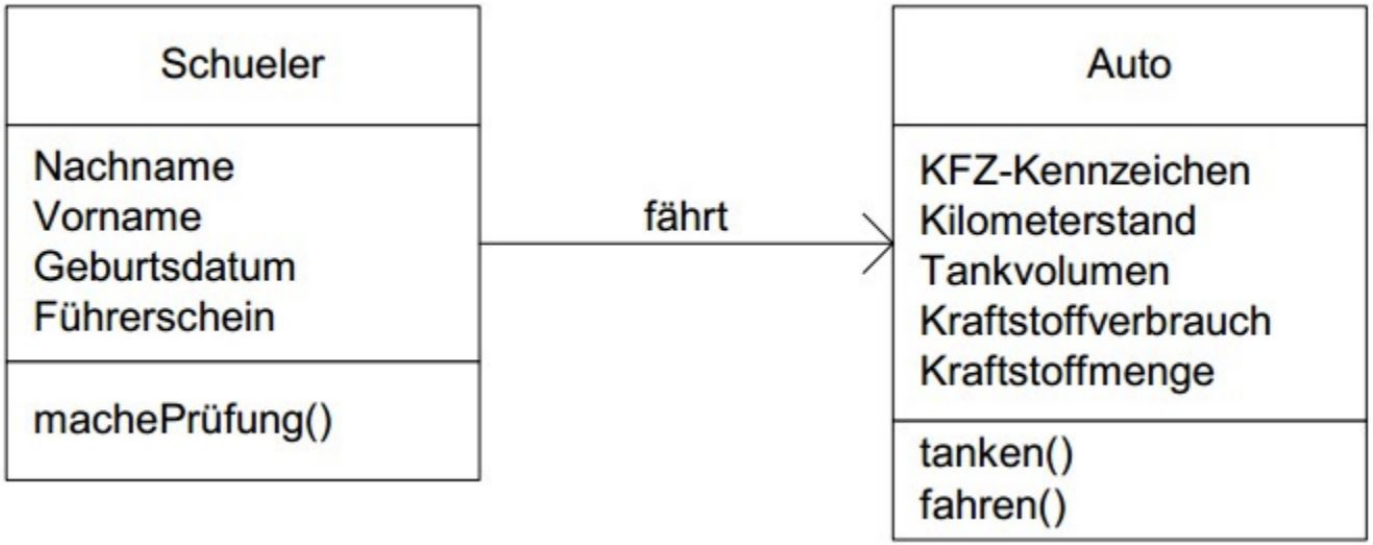
```
salary_employee = hr.SalaryEmployee(1, 'John Smith', 1500)
hourly_employee = hr.HourlyEmployee(2, 'Jane Doe', 40, 15)
commission_employee = hr.CommissionEmployee(3, 'Kevin Bacon', 1000, 250)

payroll_system = PayrollSystem()
payroll_system.calculate_payroll([
    salary_employee,
    hourly_employee,
    commission_employee
])
```

```
Calculating Payroll
==================
Payroll for: 1 - John Smith
- Check amount: 1500

Payroll for: 2 - Jane Doe
- Check amount: 600

Payroll for: 3 - Kevin Bacon
- Check amount: 1250
```

Assoziation – kennt-Beziehung

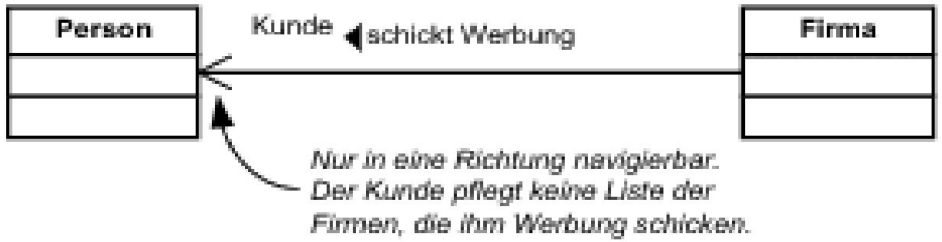
- Zwischen Objekten von Klassen können konkrete Beziehungen bestehen

- Name
- Navigierbarkeit
- Multiplizität



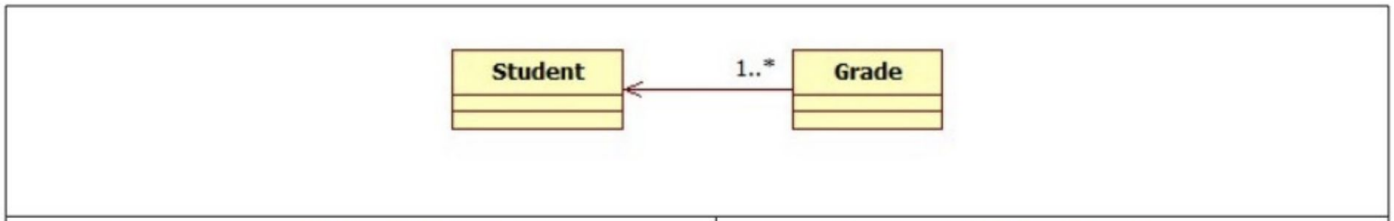
Navigierbarkeit

- die `Firma` kann alle `Personen` auflisten, denen sie Werbung zuschickt
- eine `Person` hat keine Liste der `Firmen`
- die `schickt Werbung` Assoziation ist navigierbar in der Richtung Firma-Person



Multiplizität

- wie viele Objekte des jeweiligen Typs in einer Beziehung stehen können
- Ein Objekt steht in einer Beziehung mit
  - genau einem anderen Objekt
  - keinem oder einem anderen Objekt
  - mindestens einem anderen Objekt
  - beliebig vielen anderen Objekten



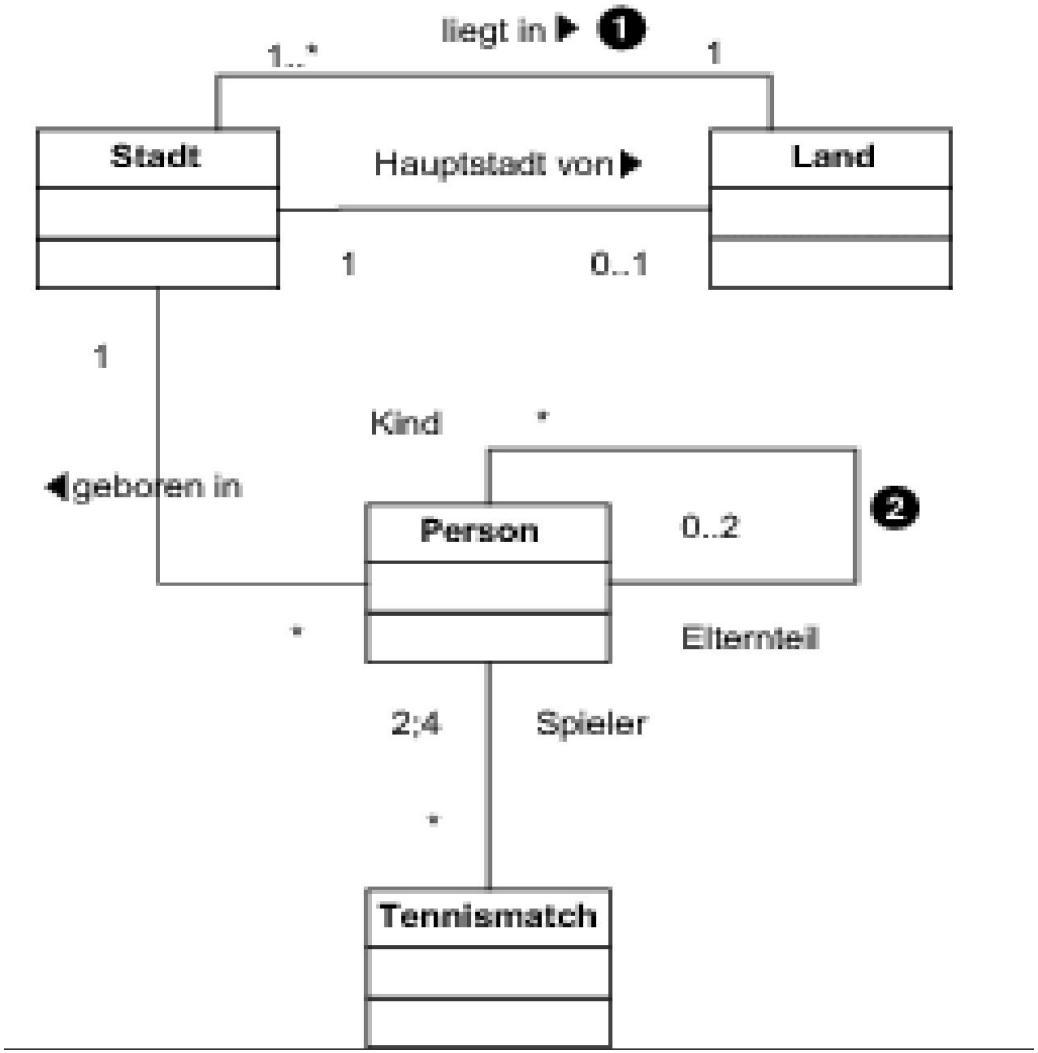

```
gr = ctr.assign("1", "FP", 10)
assert gr.getDiscipline()=="FP"
assert gr.getGrade()==10
assert gr.getStudent().getId()=="1"
assert
gr.getStudent().getName()=="Ion"
```

```
st = Student("1", "Ion",
       Address("Adr", 1, "Cluj"))

rep = GradeRepository()
grades = rep.getAll(st)
assert grades[0].getStudent()==st
assert grades[0].getGrade()==10
```

Übung

Klassen:

- Stadt
- Land
- Person
- Tennismatch




BUG

Was ist ein `Fehler`?

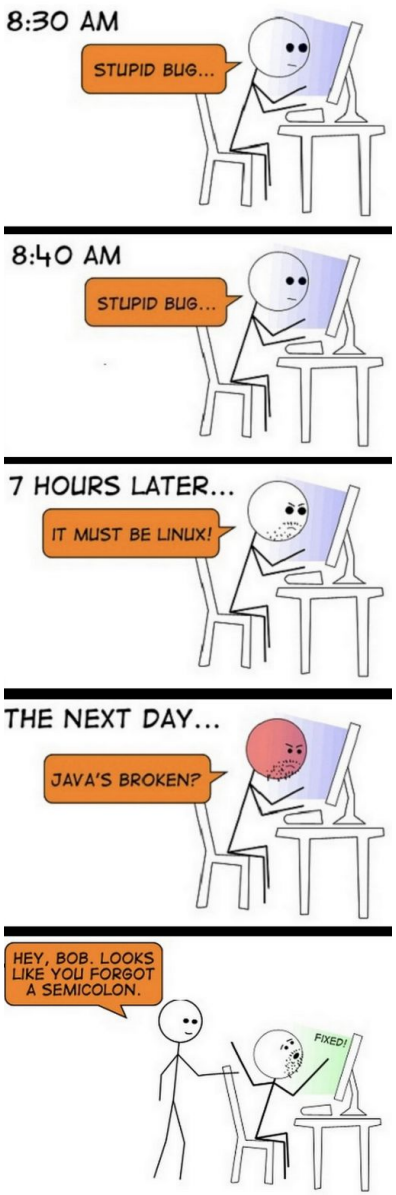
1. der Programmierer macht einen Fehler
2. und hinterlässt den Fehler im Programmcode
3. wird dieser Code ausgeführt, haben wir eine Abnormalität im Programmzustand,
4. die sich als ein Fehler nach außen manifestiert


I DON'T ALWAYS TEST
VIA 9GAG.COM
BUT WHEN I DO, I TEST DURING PRODUCTION

Softwaretest

- Ziel des Testens ist, durch gezielte Programmausführung Fehler zu erkennen
  - Test Cases (input + output + assert)
- Das Testen soll Vertrauen in die Qualität der Software schaffen
- Die Korrektheit eines Programms kann durch Testen (außer in trivialen Fällen) nicht bewiesen werden

Program testing can be used to show the presence of bugs, but never to show their absence. (Dijkstra)



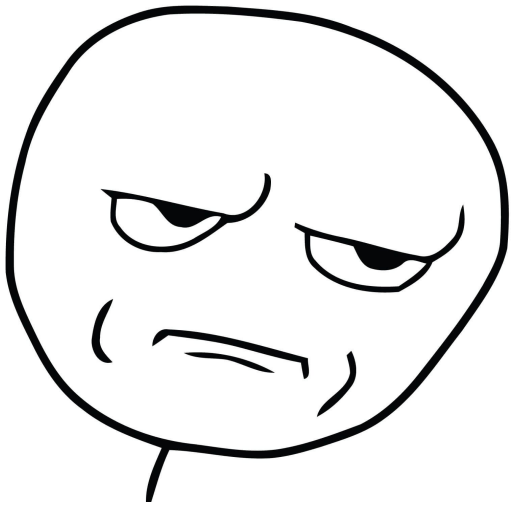
Methoden beim Testen

Black-Box-Test

die Tests ohne Kenntnisse über die innere Funktionsweise des zu testenden Systems entwickelt werden

White-Box-Test

die Tests mit Kenntnissen über die innere Funktionsweise des zu testenden Systems entwickelt werden



Methoden beim Testen

Black-Box-Test	White-Box-Test
• Testfälle gehen von der Spezifikation aus	• Testfälle ausgehend von der Struktur des Testobjekt
• Interna des Testobjekts sind bei der Ermittlung der Testfälle unbekannt	• Testfälle werden vom Entwickler beschrieben
• Testüberdeckung wird an Hand des spezifizierten Ein/Ausgabeverhaltens gemessen	• Testüberdeckung wird an Hand des Codes gemessen

```python
def isPrime(nr):
    """
      Verify if a number is prime
      return True if nr is prime False if not
      raise ValueError if nr<=0
    """
    if nr<=0:
        raise ValueError("nr need to be positive")
    if nr==1:#1 is not a prime number
        return False
    if nr<=3:
        return True
    for i in range(2,nr):
        if nr%i==0:
            return False
    return True
```

Black Box

- test case for a prime/not prime
- test case for 0
- test case for negative number

White Box (cover all the paths)

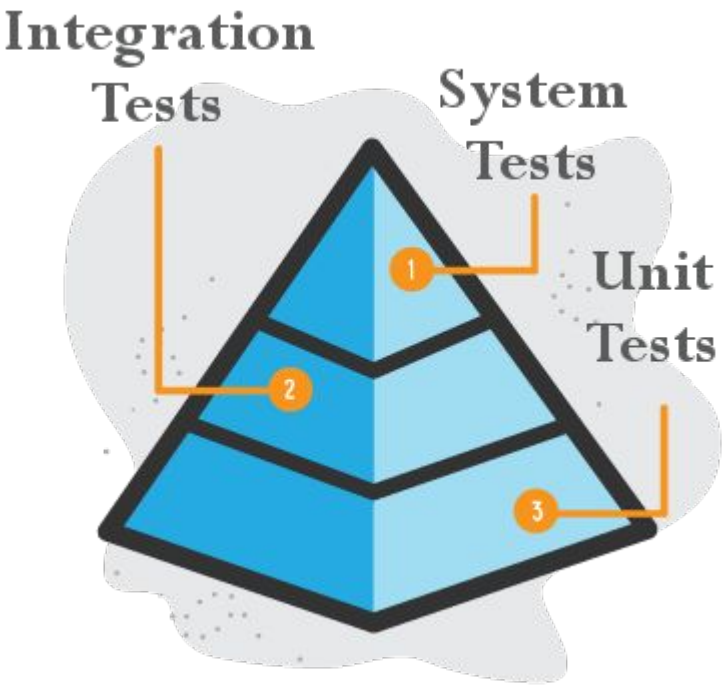
- test case for 0
- test case fot negative
- test case for 1
- test case 3
- test case for prime (no divider)
- test case for not prime

```python
def blackBoxPrimeTest():
    assert (isPrime(5)==True)
    assert (isPrime(9)==False)
    try:
        isPrime(-2)
        assert False
    except ValueError:
        assert True
    try:
        isPrime(0)
        assert False
    except ValueError:
        assert True
```

```python
def whiteBoxPrimeTest():
    assert (isPrime(1)==False)
    assert (isPrime(3)==True)
    assert (isPrime(11)==True)
    assert (isPrime(9)==True)
    try:
        isPrime(-2)
        assert False
    except ValueError:
        assert True
    try:
        isPrime(0)
        assert False
    except ValueError:
        assert True
```

Testen

- Komponententest, Modultest (Unit Test)
  - die Tests die wir schon geschrieben haben
- Integrationstest (Integration Test)
- Systemtest (System Test)



Unit Testing

- Unit Test: **Automatischer White-Box Test** welcher eine Einheit (z.B. Modul, Klasse, Komponente etc.) testet
- Unit Testing: Erstellen, Verwalten und Ausführen aller Unit Tests
- Unit Testing ist das Fundament aller agilen Softwareentwicklung Methodologien.


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


Refactor

$$5(x-2)+6(x-2)^2 =$$

$$= (x-2)[5+6(x-2)]$$

$$= (x-2)(6x-7)$$

Re•fac•to•ring

(noun)

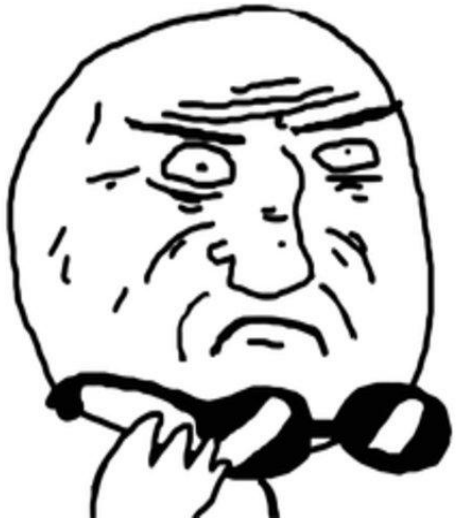
„a disciplined technique for restructuring and existing body of code, altering its internal structure without changing its external behavior.„

-refactoring.com


UNIVERSITAS NAPOCENSIS
TRADITIO NOSTRA
UNACUM EUROPAE
VIRTUTIBUS SPLENDET
1581
ROMÂNIA
BABEŞ-BOLYAI

wie oft?

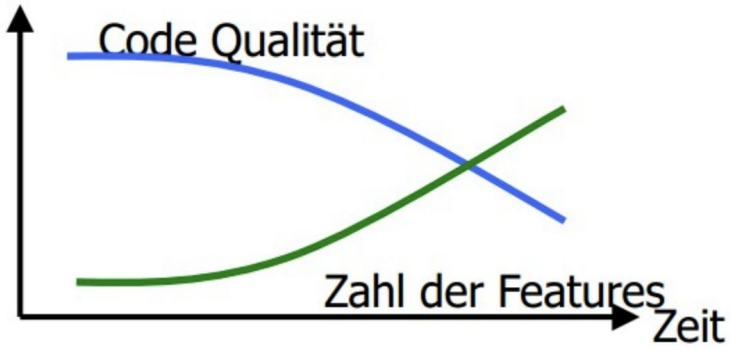
...immer

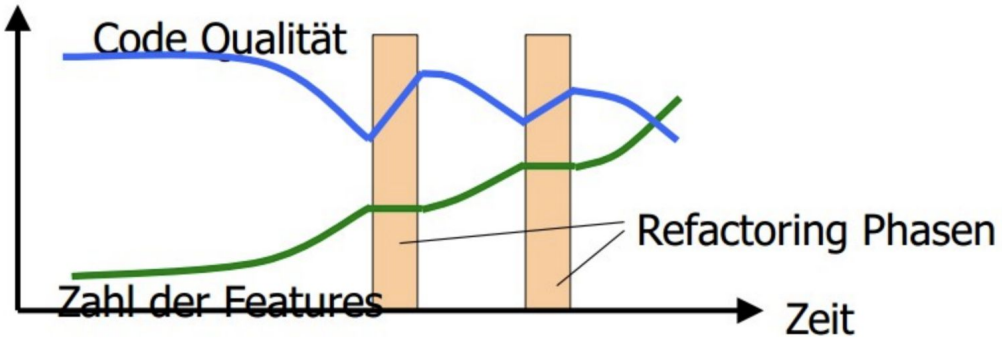


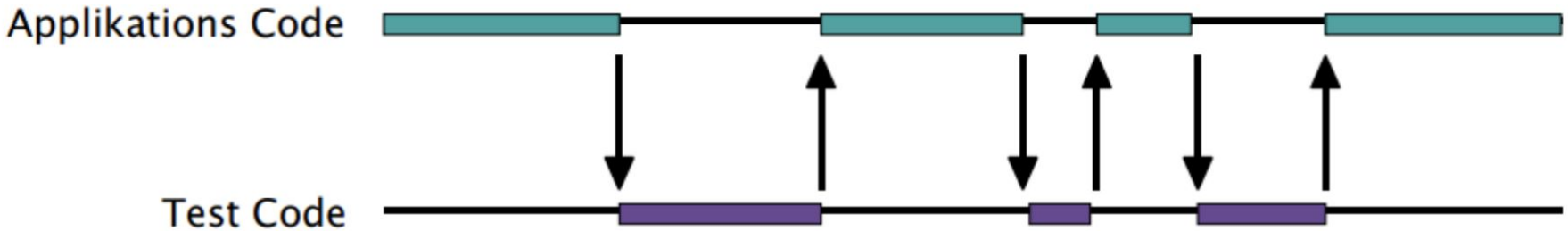
„Code a little, test a little, refactor a litte!“

Refactoring:

Verbesserung des Codes ohne Änderung des Verhaltens.







4 Prinzipien

- erstelle Code, der von allen Softwareentwickler verstanden werden kann
- die Personen, die den Code reviewen/verwenden, sollten nichts annehmen
- Manchmal geht es um das Entfernen von Code (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
- Immer in kleinen Schritten:
  - Ein Refactoring Schritt
  - Testen
  - Nächster Refactoring Schritt
  - Testen
- Häufiger Wechsel zwischen Implementation eines neuen Features und Refactoring

The return: Bad Smell

- Duplizierter Code
  - Hoher Wartungsaufwand da Änderungen überall nachgeführt werden müssen
- Lange Methode
  - Schwierig zu verstehen
  - Schlechte Wiederverwendbarkeit
  - Folge von Code Duplikationen
- Grosse Klasse
  - Oft schlechte Wiederverwendbarkeit da die Klasse viele verschiedene Dinge machte/viele Verantwortungen
  - Folge von Code Duplikationen

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

Methode extrahieren

- Ein Codefragment kann zusammengefasst werden
- Setze das Fragment in eine Methode, deren Name den 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();
}
```

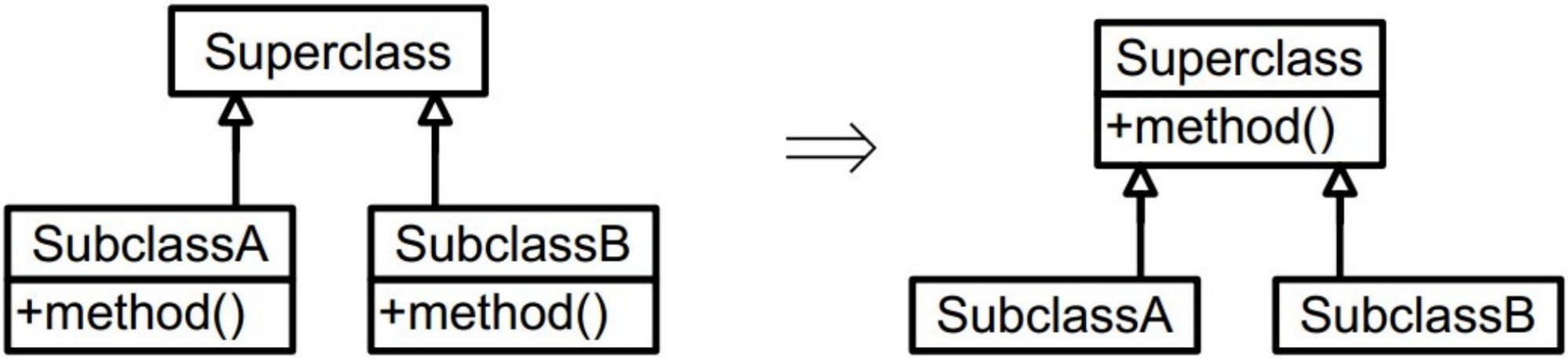
```
void berechneKosten()
{
    kosten = a * b + c;
    kosten -= discount;
}
```

Methode umbenennen

- Der Name einer Methode macht ihre Absicht nicht deutlich
- Ändere den Name der Methode

Methode hochziehen

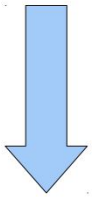
- Es gibt Methoden mit identischen Ergebnissen in den Unterklassen
- Verschiebe die Methoden in die Oberklasse



Beschreibende Variable

- Es gibt einen komplizierten Ausdruck
- Setze den Ausdruck (oder Teile) in eine 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
```



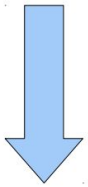
```
isMacOs = platform.toUpperCase().indexOf("MAC") > -1
isIEBrowser = browser.toUpperCase().indexOf("IE") > -1
wasResized = resize > 0;
if isMacOs and isIEBrowser and wasInitialized() and wasResized: #stuff
```

Replace Temp with Query

- Eine temporäre Variable speichert das Ergebnis eines Ausdrucks
- Stelle den Ausdruck in eine Abfrage-Methode
- ersetze die temporäre Variable durch Aufrufe der Methode
- 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
```

Refactoring - Methoden

- ~~Control Flag~~
- ~~Double Negative~~
- ~~Zuweisung an Parametervariable~~