Grundlagen der Programmierung

Vorlesung und Übung

04 – Dekomposition, Abstraktion und Funktionen

Prof. Dr. Andreas Biesdorf

Wirtschaft
Hauptcampus

H O C H
S C H U L E
T R I E R

WAS WIR BISHER GELERNT HABEN ...

Themen bisher:

- Grundlegende Operatoren und Berechnungen
- Kleinere Programme mit einfachen Kontrollmechanismen
- Implementierung einfacherer Suchen

Einschränkungen:

- Jedes Programm ist eine einzelne Datei
- Jeder Code ist immer noch recht sequentiell aufgebaut
- Hohe Komplexität selbst bei einfachen Problemen

Dekomposition & **Abstraktion**

- Zerlegung des Codes in größere Bausteine / Blöcke
- Kapselung von Funktionalität in Bausteinen

  ➔ Einführung von **Funktionen**

RAUCH- UND KOHLENMONOXID-MESSER

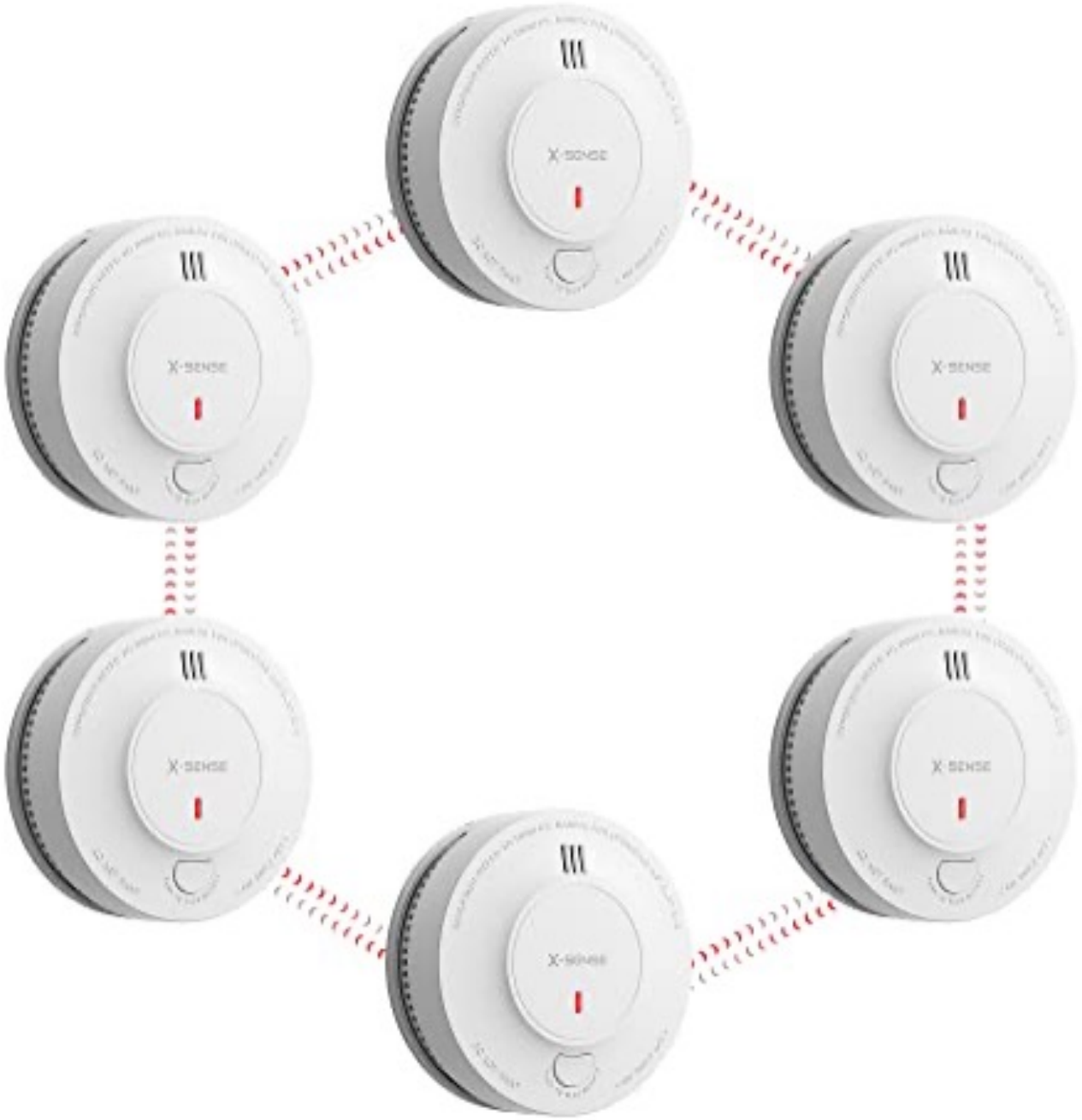
DER RAUCH- UND KOHLENMONOXID-MESSER

- Unklar, wie das Gerät funktioniert
- Interfaces aber recht einfach zu bedienen
- Wenn es mit uns spricht, verstehen wir es ;-)

Abstraktion:
Wir müssen nicht wissen, wie das Gerät funktioniert, um es zu benutzen!

Abstraktion = Verstecken von Implementierungsdetails - „Information Hiding“ – Nutzer muss nur den „Vertrag“ kennen

RAUCHMELDER IM NETZWERK



RAUCH- UND KOHLENMONOXID-MESSER IM NETZWERK

- Unterschiedliche Messgeräte interagieren im Netzwerk mit einem gemeinsamen Ziel

Dekomposition:
Unterschiedliche Teile des Systems arbeiten auf ein gemeinsames Ziel hin

Dekomposition = Zerlegung von Problemen in einzelne unabhängige Bausteine

FUNKTIONEN IN PYTHON

Keywort

Name

Parameter

```
def geradeZahl(i):
    """
    Input: i, Zahl
    Returns True if i is even, otherwise Fals
    """
    print("Wir sind in der Funktion geradeZahl und testen die Zahl " + str(i))

    return i % 2 == 0


print("Hallo – wir starten das Programm")

i = int(input("Testzahl: "))
print(geradeZahl(i))
```

Spezifikation

Eigentliche Funktion

Rückgabe Ergebnis

>> Achtung: Scopes!

FUNKTIONEN MIT UND OHNE RÜCKGABEWERTE


```
def justPrint():
    print("test")

def printAndReturn():
    print("test")
    return "test"



print(justPrint())
print(printAndReturn())
```

>> Rückgabewert: None

```
def geradeZahl(i, ausgabeText="Hallo"):
    """
    Input: i, Zahl
    Returns True if i is even, otherwise False
    """
    print(ausgabeText)
    print("Wir sind in der Funktion geradeZahl und testen die Zahl " + str(i))

    return i % 2 == 0

print("Hallo – wir starten das Programm")
i = int(input("Testzahl: "))
print(geradeZahl(i, "Halli"))
print(geradeZahl(i))
```

>> Standardbelegung von Parameter

STANDARDBELEGUNGEN VON PARAMETERN (DEFAULT-WERTE)


```
def geradeZahl(i, ausgabeText="Hallo"):
    """
    Input: i, Zahl
    Returns True if i is even, otherwise False
    """
    print(ausgabeText)
    print("Wir sind in der Funktion geradeZahl und testen die Zahl " + str(i))

    return i % 2 == 0

print("Hallo - wir starten das Programm")
i = int(input("Testzahl: "))
print(geradeZahl(i, "Halli"))
print(geradeZahl(i))
```

>> Standardbelegung von Parameter

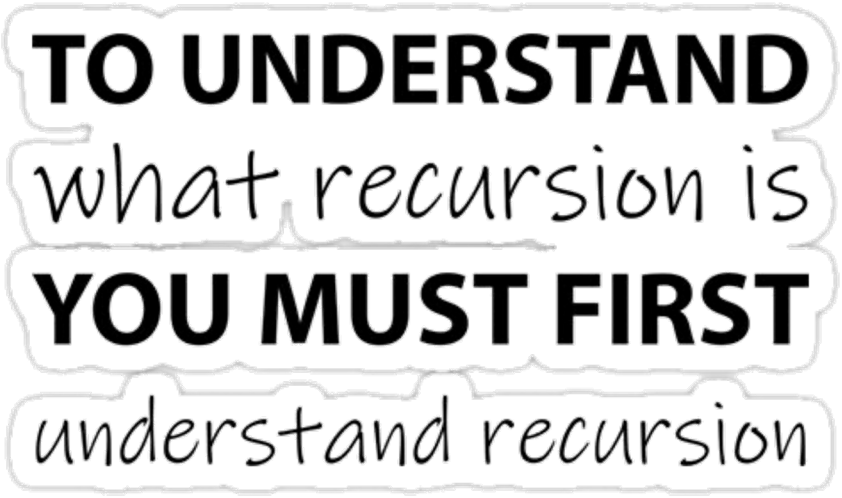
SPEZIFIKATIONEN VON FUNKTIONEN

- Legt einen **Vertrag** zwischen dem Entwickler einer Funktion und den Nutzern fest
- **Formulierung von Annahmen und Garantien**
  - Annahmen: Bedingungen, die vom Nutzer einer Funktion zu erfüllen sind (z.B. Bereiche von Variablen)
  - Garantien: Bedingungen, die die Funktion erfüllt, konsistent mit Vertrag

- **Annahmen** und **Garantien** sind wichtig für die Erstellung von **Test-Cases!**

- Lösungsansatz für Probleme nach dem Teile-und-Herrsche Prinzip

- **Grundidee:**
  Funktion ruft sich selbst auf
- **Herausforderung**:
  Vermeidung von Endlosschleifen
- **Vorteil:**
  Intuitive und effiziente Formulierung
- **Achtung:**
  Rechenkomplexität!



MULTIPLIKATION – ITERATIV VS. REKURSIV


```
def multiplikation_iterativ(var_a, var_b):
    result = 0

    while var_b > 0:
        result += var_a
        var_b -= 1

    return result
```

```
def multiplikation_rekursiv(var_a, var_b):
    if var_b == 1:
        return var_a
```

>> Basis-Fall

```
    else:
        return var_a + multiplikation_rekursiv(var_a, var_b-1)
```

>> Rekursiver Schritt

MULTIPLIKATION – REKURSIV


```
def multiplikation_rekursiv(var_a, var_b):
    if var_b == 1:
        return var_a
    else:
        return var_a + multiplikation_rekursiv(var_a, var_b-1)
```

>> Basis-Fall

>> Rekursiver Schritt

$$\text{var\_a} * \text{var\_b} = \underbrace{\text{var\_a} + \text{var\_a} + \text{var\_a} + \ldots + \text{var\_a}}_{\text{var\_b}}$$

$$= \text{var\_a} + \underbrace{\text{var\_a} + \text{var\_a} + \ldots + \text{var\_a}}_{\text{var\_b -1}}$$

$$= \text{var\_a} + \text{var\_a} * (\text{var\_b} - 1)$$

DIE TÜRME VON HANOI

https://www.youtube.com/watch?v=UMPneeBzQHk

ADDITION – ITERATIV VS. REKURSIV


```
def addition_iterativ(var_a, var_b):
    result = var_a

    while var_b > 0:
        result += 1
        var_b -= 1

    return result
```

```
def addition_rekursiv(var_a, var_b):
    if var_b == 0:
        return var_a
    else:
        return addition_rekursiv(var_a+1, var_b-1)
```

>> Basis-Fall

>> Rekursiver Schritt

ADDITION – REKURSIV


```
def addition_rekursiv(var_a, var_b):
    if var_b == 0:
        return var_a
    else:
        return addition_rekursiv(var_a+1, var_b-1)
```

>> Basis-Fall

>> Rekursiver Schritt

```
var_a + var_b       = (3)+ 4
                    = (3 + 1) + 3
                    = (3 + 1 + 1) + 2
                    = (3 + 1 + 1 + 1) + 1
                    = (3 + 1 + 1 + 1 + 1) + 0
```

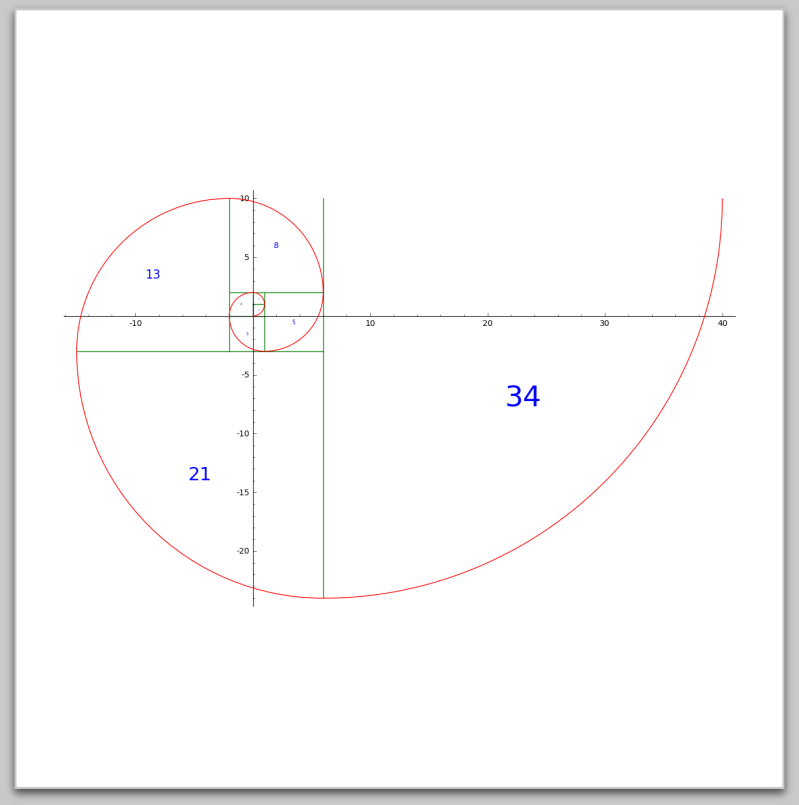
FAKULTÄT – REKURSIV

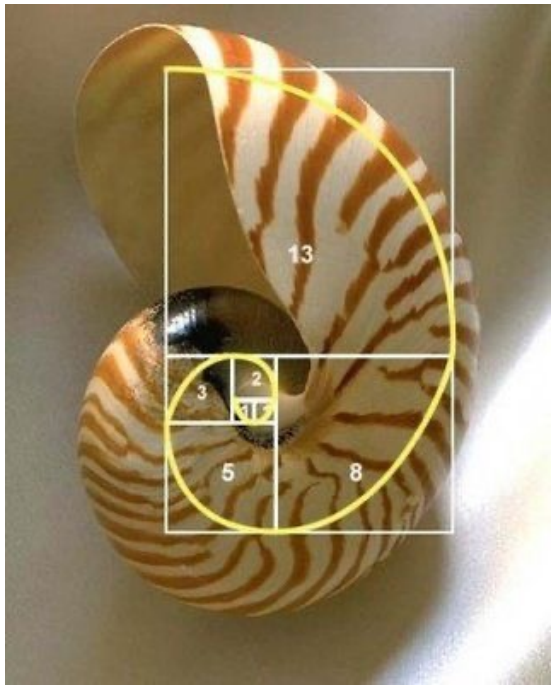

```
def fakultaet(wert):
    if wert == 1:
        return 1
    else:
        return wert * fakultaet(wert-1)
```

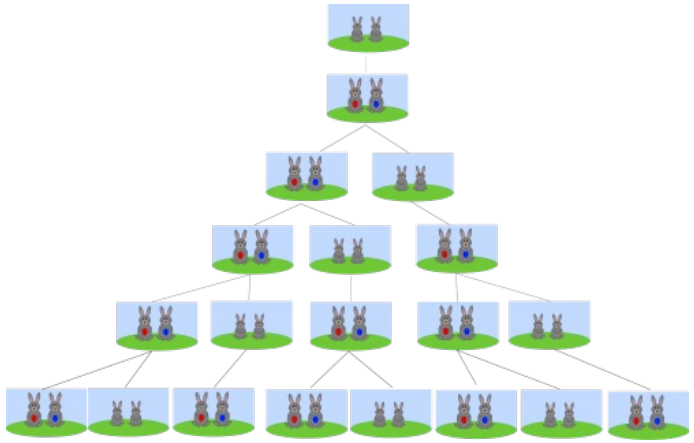
>> Basis-Fall

>> Rekursiver Schritt

```
fakultaet(wert)     = wert * (wert-1) * (wert-2) * … * 1
                      \_____________ wert _____________/
                    = wert * (wert-1) * (wert-2) * … * 1
                             \_____ wert-1 _____/
                    = wert * fakultaet(wert-1)
```







FIBONACCI-FOLGE

$$f_n = f_{n-1} + f_{n-2} \quad \text{für} \quad n \geq 3 \qquad f_1 = f_2 = 1$$

FIBONACCI-FOLGE


```
def fib(pos_fib):
    if pos_fib == 0 or pos_fib == 1:
        return 1
    else:
        return fib(pos_fib-1) + fib(pos_fib-2)
```

>> Basis-Fall

>> Rekursiver Schritt

0, 1, 1, 2, 3, 5, 8, 13, ...

(optional) 0+1 1+1 1+2 2+3 3+5 5+8

FAKTORISIERUNG


```
def faktorisierung(wert):
    if wert == 1:
        return 1
    else:
        return wert * faktorisierung(wert-1)
```

>> Basis-Fall

>> Rekursiver Schritt

$$\text{faktorisierung}(wert) = \underbrace{wert * (wert-1) * (wert-2) * \ldots * 1}_{wert}$$

$$= wert * \underbrace{(wert-1) * (wert-2) * \ldots * 1}_{wert-1}$$

$$= wert * \text{faktorisierung}(wert-1)$$