Programmierung

Bachelor Informatik

Peter Rossmanith

Wintersemester 2013/14

RWTH AACHEN

Inhalt

1. Einführung in die objektorientierte Programmierung
2. Rekursion
3. Fehler finden und vermeiden
4. Objektorientiertes Design
5. Effiziente Programme
6. Nebenläufigkeit
7. Laufzeitfehler
8. Refactoring
9. Persistenz
10. Eingebettete Systeme
11. Funktionale Programmierung
12. Logische Programmierung

Programmierung

Organisatorisches:

`http://programmierung.informatik.rwth-aachen.de`

Algorithmus

≈ Eindeutige, genaue Anleitung für die Lösung eines Problems.

Programm

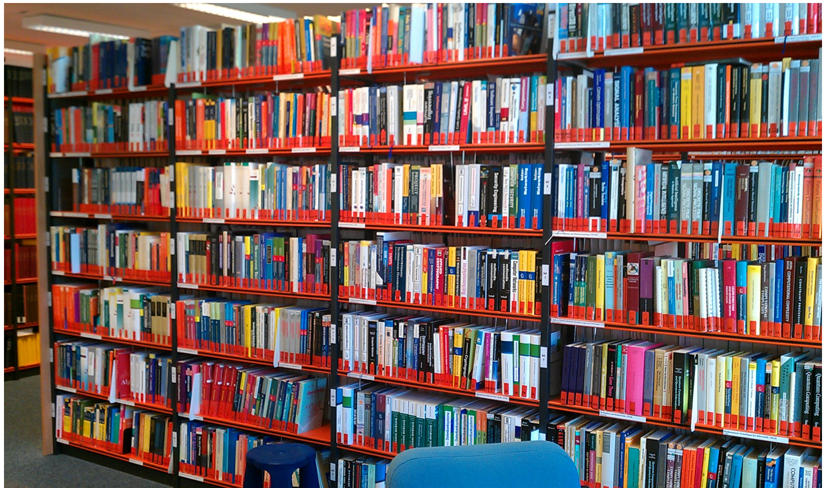
≈ Eindeutige, genaue Anleitung für die Lösung eines Problems in einer *Programmiersprache* formuliert.

Programme und Algorithmen

Es gibt sehr alte Algorithmen:

- Euklidischer Algorithmus
  $\longrightarrow$ größter gemeinsamer Teiler
- Sieb des Erathostenes
  $\longrightarrow$ Berechnung von Primzahlen
- Babylonisches Wurzelziehen
  $\longrightarrow$ Näherung der Quadratwurzel

Literatur zur Vorlesung



Handapparat der Informatik-Bibliothek.

Literatur zur Vorlesung

Ausschnitt zur Programmierung.

Tabelliermaschinen (1890)

Bild von Arnold Reinhold

Neues Kapitel

1. Einführung in die objektorientierte Programmierung
2. Rekursion
3. Fehler finden und vermeiden
4. Objektorientiertes Design
5. Effiziente Programme
6. Nebenläufigkeit
7. Laufzeitfehler
8. Refactoring
9. Persistenz
10. Eingebettete Systeme
11. Funktionale Programmierung
12. Logische Programmierung

Überblick

1. Einführung in die objektorientierte Programmierung
   - Klassen und Objekte
   - Vererbung, Setter, Getter
   - Primitive Datentypen
   - Kontrollstrukturen
   - Arrays

Erstes Beispiel einer Klasse

```
class Person {
  String nachname;
  String vorname;
}
```

Vorname und *Nachname* können gelesen und geschrieben werden.

```
Person fahrer = new Person();
fahrer.vorname = "Karl";
fahrer.nachname = "Ranseier";
String x = fahrer.nachname;
```

Private Attribute

```
class Person {
  private String nachname;
  private String vorname;
  String getNachname() {
    return nachname;
  }
  String getVorname() {
    return vorname;
  }
  void setNachname(String name) {
    nachname = name;
  }
  void setVorname(String name) {
    vorname = name;
  }
}
```

Private Attribute

Auf *vorname* und *nachname* kann nicht direkt zugegriffen werden.

Dies ist nur über dafür vorgesehene Methoden möglich.

```
Person fahrer = new Person();
fahrer.setVorname("Karl");
fahrer.setNachname("Ranseier");
String x = fahrer.getNachname();
```

Konstruktoren

```
class Person {
  private String nachname;
  private String vorname;

  public Person(String vorname, String nachname) {
    this.nachname = nachname;
    this.vorname = vorname;
  }

  String getNachname() {
    return nachname;
  }

  ...
```

Konstruktoren

Nicht mehr möglich:

```
Person fahrer = new Person();
```

Wir müssen unseren Konstruktor verwenden:

```
Person fahrer = new Person("Karl", "Ranseier");
```

Überblick

1. Einführung in die objektorientierte Programmierung
   - Klassen und Objekte
   - Vererbung, Setter, Getter
   - Primitive Datentypen
   - Kontrollstrukturen
   - Arrays

Vererbung

Eine Oberklasse kann Unterklassen haben:

```
class Person {
  String nachname;
  String vorname;
  ...
}
```

Unterklasse:

```
class Student extends Person {
  String matrikelnummer;
  public Student(String nachname, String vorname, String nummer) {
    this.nachname = nachname;
    this.vorname = vorname;
    matrikelnummer = nummer;
  }
}
```

Vererbung

```
Student karl = new Student("Karl", "Ranseier", "123456");
String x = karl.matrikelnummer;
```

Das geht aber nicht:

```
Person karl = new Student("Karl", "Ranseier", "123456");
String x = karl.matrikelnummer;
```

Stattdessen:

```
Person karl = new Student("Karl", "Ranseier", "123456");
String x = ((Student)karl).matrikelnummer;
```

Dies nennen wir einen *cast*.

Vererbung

Eine Unterklasse kann Methoden überschreiben.

```
class Person {
  ...
  String toString() {
    return vorname + "  " + nachname;
  }
}
class Student extends Person {
  ...
  String toString {
    return vorname + "  " + nachname + "  " + matrikelnummer;
  }
}
```

Virtuelle Methoden

```
Person joe = new Person("Joe", "Doe");
Person karl = new Student("Karl", "Ranseier", "123456");
Student sue = new Student("Sue", "Winter", "654321");
String x = joe.toString();
String y = karl.toString();
String z = sue.toString();
```

Was enthalten x, y, z?

In Java sind alle Methoden *virtuell*:

Die aufgerufene Methode richtet sich nicht nach dem Typ der Variablen, sondern des Objekts selbst.

Variablen

Eine Klasse kann *Instanzvariablen* haben. Jede Variable muß einen *Typ* haben.

z. B.

```
private String name;
private int anzahl;
private float geschwindigkeit;
```

Jede Variable wird im Rumpf der Klassendefinition deklariert.

Datentypen

Ein Typ einer Variable ist

- eine selbstdefinierte Klasse oder
- eine Klasse, die von anderen definiert wurde, insbesondere aus einer Softwarebibliothek oder
- ein primitiver Typ:
  boolean, **int**, **short**, **byte**, **char**, **float**, **double**.

Überblick

1. Einführung in die objektorientierte Programmierung
   - Klassen und Objekte
   - Vererbung, Setter, Getter
   - Primitive Datentypen
   - Kontrollstrukturen
   - Arrays

Primitive Datentypen

- **long**: ganze Zahl n mit $-2^{63} \leq n \leq 2^{63} - 1$
  Zweierkomplement-Darstellung mit 32 bit
- **int**: ganze Zahl n mit $-2^{31} \leq n \leq 2^{31} - 1$
  Zweierkomplement-Darstellung mit 32 bit
- **short**: ganze Zahl n mit $-32768 \leq n \leq 32767$
  Zweierkomplementdarstellung mit 16 bit
- **byte**: ganze Zahl n mit $-128 \leq n \leq 127$
  Zweierkomplementdarstellung mit 8 bit
- **char**: Ein Zeichen in 16 bit Unicode-Kodierung.

Fließkommazahlen

- **float**: Fließkommazahl nach IEEE 754–Standard mit 32 bit

- **double**: Fließkommazahl nach IEEE 754–Standard mit 64 bit

Die Mantisse ist eine binär kodierte natürliche Zahl, aber die führende Eins wird *nicht* gespeichert.

Der Exponent wird als binär kodierte natürliche Zahl gespeichert und daher um 127 bzw. 1023 erhöht. Vorteil: Fließkommazahlen können wir normale Binärzahlen verglichen werden.

Zuweisungen

Einer Variable können neue Werte zugewiesen werden:

int x;

$\vdots$

$x = 5;$
$x = y;$
$x = y + z;$
$x = (42 * y + z) - y * y + 2 * x;$

Arithmetische Ausdrücke

In absteigender Priorität:

- $*$, $/$ Multiplikation, Division
- $+$, $-$ Addition, Subtraktion

$x + y * z$ bedeutet $x + (y * z)$

$x - y - z$ bedeutet $(x - y) - z$

Beispiel

Kugeloberfläche und -inhalt

```
class Kugel {
  private double radius;
  public Kugel(double r) {
    radius = r;
  }
  public double getSurface() {
    return 4.0 * Math.PI * radius * radius;
  }
  public double getVolume() {
    return 4.0/3.0 * Math.PI * radius * radius * radius;
  }
}
```

Wahrheitswerte

Primitiver Typ: **boolean**

Vergleichsoperationen liefern als Ergebnis **boolean**.

- $x == y$ Gleichheit
- $x \, != y$ Ungleichheit
- $x < y$ Kleiner
- $x <= y$ Kleiner oder gleich

Es gibt die Konstanten *true* und *false*.

```
int x, y, z;
...
boolean b = (5 * x <= y + z);
```

Wahrheitswerte

Wahrheitswerte können mit *logischen Operationen* verknüpft werden.

- $!a$ Negation
- a && b logisches Und
- a || b logisches Oder

Logische Operatoren haben niedrigere Priorität als Vergleichsoperatoren.

Beispiel

```
boolean b = (0 <= x && x < 10);
```

Überblick

1. Einführung in die objektorientierte Programmierung
   - Klassen und Objekte
   - Vererbung, Setter, Getter
   - Primitive Datentypen
   - Kontrollstrukturen
   - Arrays

Verzweigungen

```
public int maximum(int x, int y) {
    int result;
    if(x < y) {
        result = y;
    }
    else {
        result = x;
    }
    return result;
}
```

Mit einer **if**-Anweisung können wir eine Bedingung prüfen und dann den **then**- oder **else**-Zweig ausführen lassen. Der **else**-Zweig ist optional.

Verzweigungen

```
public int sign(int x) {
  int result;
  if(x < 0) {
    result = -1;
  }
  else if(x > 0) {
    result = +1;
  }
  else {
    result = 0;
  }
  return result;
}
```

Es darf mehrere **else if**-Teile geben.

Schleifen

```
public int dreiecksZahl(int n) {
  int sum = 0;
  int k = 0;
  while(k <= n) {
    num = num + k;
    k = k + 1;
  }
  return sum;
}
```

Wir berechnen hier die Summe

$$\sum_{k=0}^{n} k = \frac{n(n+1)}{2}$$

Schleifen

```
while(bedingung) {
   anweisung1;
   anweisung2;
   ...
}
```

Solange *bedingung* erfüllt ist, wird der Rumpf der Schleife ausgeführt. Ist *bedingung* anfangs nicht erfüllt, wird der Rumpf gar nicht ausgeführt.

for-Schleifen

```
for(anweisung1; bedingung; anweisung2) {
  rumpf;
}
```

ist gleichbedeutend mit

```
anweisung1;
while(bedingung) {
  rumpf;
  anweisung2;
}
```

for-Schleifen

Beispiel

```
public int dreiecksZahl(int n) {
  int sum = 0;
  for(int k = 1; k <= n; k = k + 1) {
    sum = sum + k;
  }
  return sum;
}
```

Wieder berechnen wir

$$n \mapsto \sum_{k=0}^{n} k = \frac{n(n+1)}{2}.$$

for-Schleifen

```
public int tetraederZahl(int n) {
  int sum = 0;
  for(int k = 1; k <= n; k = k + 1) {
    for(int j = 1; j <= k; j = j + 1) {
      sum = sum + j;
    }
  }
  return sum;
}
```

Wir berechnen

$$n \mapsto \sum_{k=1}^{n} \sum_{j=1}^{k} j = \frac{n(n+1)(n+2)}{6}.$$

RWTHAACHEN

Überblick

1. Einführung in die objektorientierte Programmierung
   - Klassen und Objekte
   - Vererbung, Setter, Getter
   - Primitive Datentypen
   - Kontrollstrukturen
   - Arrays

Arrays

Wir wollen häufig viele Dinge gemeinsam speichern.

```
int a[ ] = new int[10]; // Array der Größe 10
a[0] = 7;
a[4] = 3;
a[9] = 10;
a[10] = 12; // nicht erlaubt!
a[3] = a[0] + a[4];
```

Was ist die größte Zahl in einem Array?

```
int a[ ];
```

```
public int maximum() {
  int n = a.length;
  if(n == 0) {
    return 0;
  }
  int max = a[0];
  for(int i = 1; i < n; i++) {
    if(a[i] > max) {
      max = a[i];
    }
  }
  return max;
}
```

Einfacher Sortieralgorithmus

Wir sortieren ein Array $a[\,]$.

```
public void sort() {
  int n = a.length;  // Länge von a
  for(int j = 0; j < n; j++) {
    for(int k = 0; k < n - 1; k++) {
      if(a[k] > a[k + 1]) { // Falsche Reihenfolge
        // Vertausche a[k] mit a[k+1]
        int temp = a[k];
        a[k] = a[k + 1];
        a[k + 1] = temp;
      }
    }
  }
}
```

Neues Kapitel

1. Einführung in die objektorientierte Programmierung
2. Rekursion
3. Fehler finden und vermeiden
4. Objektorientiertes Design
5. Effiziente Programme
6. Nebenläufigkeit
7. Laufzeitfehler
8. Refactoring
9. Persistenz
10. Eingebettete Systeme
11. Funktionale Programmierung
12. Logische Programmierung

Überblick

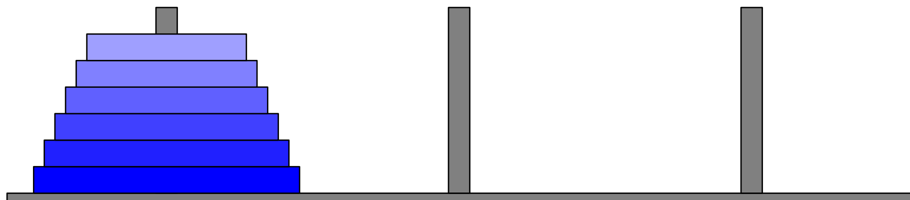
2. Rekursion
   - Rekursive Methoden
   - Backtracking
   - Memorization
   - Einfache rekursive Datenstrukturen
   - Bäume
   - Aufzählen, Untermengen, Permutationen, Bitmengen

Fibonacci-Zahlen

- $F_0 = 0$,
- $F_1 = 1$,
- $F_n = F_{n-1} + F_{n-2}$ falls $n > 1$.

```
int fibo(int n) {
  if(n == 0) {
    return 0;
  }
  else if(n == 1) {
    return 1;
  }
  else {
    return fibo(n - 1) + fibo(n - 2);
  }
}
```

Die Türme von Hanoi



Bewege die Scheiben von der linken Stange auf die rechte Stange.

- Nur je eine Scheibe darf bewegt werden.
- Eine Scheibe darf nicht auf einer kleineren liegen.

Die Türme von Hanoi

Bewege *n* Scheiben von *sourcePeg* nach *destPeg*.

Benutze *thirdPeg* als Ablage.

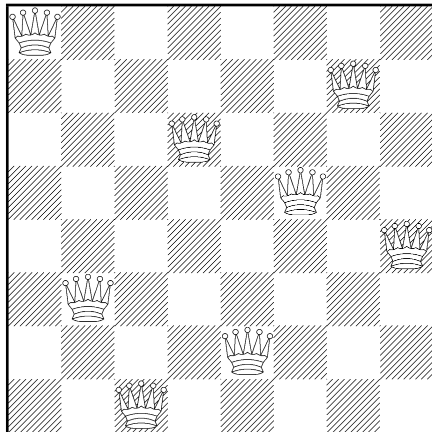
```
static String move(int n, int sourcePeg, int destPeg, int thirdPeg) {
  if(n == 0) {
    return "";
  }
  String phase1 = move(n - 1, sourcePeg, thirdPeg, destPeg);
  String phase2 = "  " + sourcePeg + "->" + destPeg;
  String phase3 = move(n - 1, thirdPeg, destPeg, sourcePeg);
  return phase1 + phase2 + phase3;
}
```

Überblick

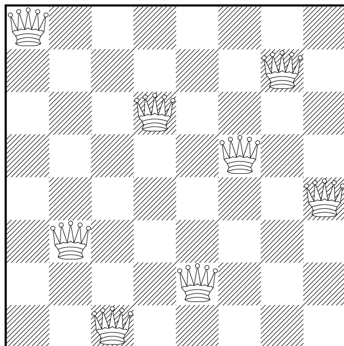
2 Rekursion

- Rekursive Methoden
- Backtracking
- Memorization
- Einfache rekursive Datenstrukturen
- Bäume
- Aufzählen, Untermengen, Permutationen, Bitmengen

Das Acht-Damen-Problem



Backtracking



Wir verwenden *Backtracking*:

Setze Damen in Zeilen von links nach rechts.

Wenn nächstes Setzen unmöglich, nimm letzten Zug zurück und versuche nächsten.

```
void setzeDamenAbZeile(int y) {
  if(y >= n) {
    geloest = true;
    return;
  }
  for(int x = 0; x < n; x++) {
    if(safePosition(y, x)) {
      brett[y][x] = true;
      setzeDamenAbZeile(y + 1);
      if(geloest) {
        return;
      }
      brett[y][x] = false;
    }
  }
}
```

Überblick

2 Rekursion

- Rekursive Methoden
- Backtracking
- Memorization
- Einfache rekursive Datenstrukturen
- Bäume
- Aufzählen, Untermengen, Permutationen, Bitmengen

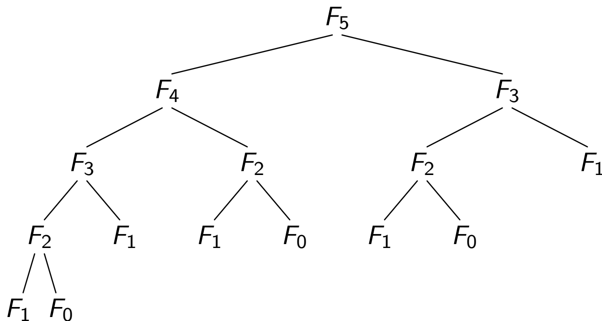
Nochmals Fibonacci-Zahlen

```
int fibo(int n) {
  if(n == 0) {
    return 0;
  }
  else if(n == 1) {
    return 1;
  }
  else {
    return fibo(n − 1) + fibo(n − 2);
  }
}
```

Wie schnell ist dieses Programm? Sehr langsam!

Nochmals Fibonacci-Zahlen

Problem: Gleiche Werte werden *mehrfach* berechnet.



Lösung: *Memorization*

In Wirklichkeit heißt es „Memoization“, aber immer mehr Leute verwenden „Memorization“. In naher Zukunft wird das also der korrekte Begriff durch Mehrheitsmeinung werden. . .

Memorization

Speichere neu berechnete Werte.

Verwende gespeicherte Werte, wenn vorhanden.

```
long fibo2(int n) {
  if(n == 0) {
    return 0;
  }
  else if(n == 1) {
    return 1;
  }
  if(f[n] == 0) {
    f[n] = fibo2(n - 1) + fibo2(n - 2);
  }
  return f[n];
}
```

Überblick

2 Rekursion

- Rekursive Methoden
- Backtracking
- Memorization
- Einfache rekursive Datenstrukturen
- Bäume
- Aufzählen, Untermengen, Permutationen, Bitmengen

Rekursive Datenstrukturen

Wir möchten eine *Liste* bzw. eine *endliche Folge* implementieren.

Operationen:

- **int** *head*(): Was ist das erste Element?
- *Liste* *tail*(): Die Liste ohne das erste Element.
- **boolean** *isempty*(): Ist die Liste leer?
- *Liste* *add*(**int** *elem*): Dieselbe Liste, aber *elem* davor.

Rekursiver Entwurf:

Eine Liste ist entweder

- leer oder
- besteht aus einem Element und einer Liste.

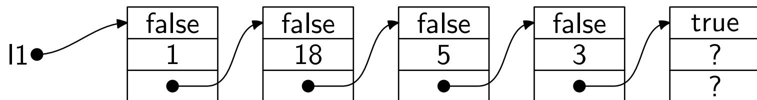
RWTHAACHEN

```
class Liste {
  private boolean empty; // Liste leer?
  private int value; // erstes Element
  private Liste rest; // restliche Elemente
  //
  int head() {...}
  Liste tail() {...}
  boolean isempty() {...}
  // Erzeuge neue Liste [elem, alte Liste]
  Liste add(int elem) {...}
}
```

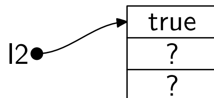
Eine *Liste* darf eine *Liste* „enthalten“!

Interne Repräsentation

Liste 1, 18, 5, 3:



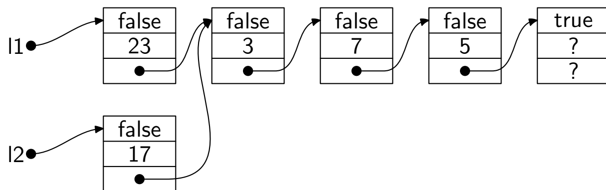
Leere Liste:



Beispiel

```
public static void main(String args[ ]) {
   Liste l1 = new Liste();
   l1 = l1.add(5);
   l1 = l1.add(7);
   l1 = l1.add(3);
   System.out.println(l1);  // [3, 7, 5]
   Liste l2 = l1;
   l2 = l2.add(17);
   l1 = l1.add(23);
   System.out.println(l1);
   System.out.println(l2);
}
```

```
public static void main(String args[ ]) {
  Liste l1 = new Liste();
  l1 = l1.add(5);
  l1 = l1.add(7);
  l1 = l1.add(3);
  System.out.println(l1); // [3, 7, 5]
  Liste l2 = l1;
  l2 = l2.add(17);
  l1 = l1.add(23);
  System.out.println(l1);
  System.out.println(l2);
}
```



Reference- und Value-Semantik

```
Student karl = new Student("Karl", "Ranseier", "123456");
...
Student klausurTeilnehmer = karl;
...
karl.setVorname("Charles");
klausurTeilnehmer.getVorname();  // ergibt Charles
```

Sowohl *karl* als auch *klausurTeilnehmer* beziehen sich auf dieselbe Person.

Bei einer Zuweisung wird keine Kopie angefertigt.

Dieses Verhalten ist natürlich für eine Klasse *Student*.

Ein Algorithmus, wie er in Lehrbüchern steht:

procedure $Dijkstra(s)$:
$Q := V - \{s\}$;
for $v \in Q$ **do** $d[v] := \infty$ **od**;
$d[s] := 0$;
while $Q \mathrel{!=} \emptyset$ **do**
  $choose\ v \in Q\ with\ minimal\ d[v]$;
  $Q := Q - \{v\}$;
  forall u *adjacent* **to** v **do**
    $d[u] := min\{d[u], d[v] + length(v, u)\}$
  od
od

Q und V sind Mengen.

In Java z.B. $V.subtract(s)$ statt $V - \{s\}$.

Bei Mengen erwarten wir eine Value-Semantik.

Immutable Klassen

Es seien A, B, C Mengen.

$$A := \{1, 2, 3\}$$
$$B := A$$
$$A := A \cup \{4\}$$

Mit Mengen sind wir gewohnt so zu arbeiten.

Wir erwarten, daß $B = \{1, 2, 3\}$ und $A = \{1, 2, 3, 4\}$.

Dieses Verhalten können wir durch immutable Klassen erreichen.

Immutable Klassen

Liste ist immutable:

- Es gibt keine Methode, die ein Objekt des Typs *Liste* ändern kann.
- Auf die Instanzvariablen kann man nicht direkt zugreifen.

Daher verhält sich *Liste* genauso wie **int** und **double**.

```
int n = 5;
int k = n + 3;  // n ist immer noch 5
Liste list = new Liste();
list = list.add(5);
list = list.add(3);  // list ist [3, 5]
Liste lang = list.add(1);  // list bleibt [3, 5]
```

Immutable Klassen

Wollen wir eine Klasse *immutable* machen, dann können wir alle Instanzvariablen **final** deklarieren:

```
class Liste {
  private final boolean empty; // ist diese Liste leer?
  private final int value; // das erste Element dieser Liste
  private final Liste rest; // restliche Liste ohne erstes Element
```

Eine **final**-Variable kann nur in einem Konstruktor verändert werden.

So ist die immutable-Eigenschaft garantiert.

Überblick

2. Rekursion
   - Rekursive Methoden
   - Backtracking
   - Memorization
   - Einfache rekursive Datenstrukturen
   - Bäume
   - Aufzählen, Untermengen, Permutationen, Bitmengen

Weitere rekursive Datenstrukturen

Ein *Baum* ist eine Wurzel, die mehrere Kinder haben kann, welche selbst Bäume sind. Einen Baum ohne Kinder nennt man *Blatt*.

```
public class Tree {
   private int root; // Ein Baum aus Zahlen
   private List⟨Tree⟩ children;
   public Tree(int r) {
      root = r;
      children = new List⟨Tree⟩();
   }
   public List⟨Tree⟩ getChildren() {return children; }
   public int getRoot() {return root; }
   public void addChild(Tree newChild) {
      children = children.add(newChild);
   }
```

Was ist List⟨Tree⟩ genau?

```
class List<T> {
  private final boolean empty; // ist diese Liste leer?
  private final T value; // das erste Element dieser Liste
  private final List<T> rest; // restliche Liste ohne erstes Element
  public List() { // erzeuge eine neue leere Liste
    empty = true;
    value = null;
    rest = null;
  }
  private List(T elem, List<T> rest) {
    this.empty = false;
    this.value = elem;
    this.rest = rest;
  }
  public boolean isempty() {
    return empty;
  }
```

Generische Klassen

$List\langle Tree\rangle$ verwendet die *generische Klasse* **class** $List\langle T\rangle$.

Eine generische Klasse hat einen zusätzlichen *Typparameter*.

Auf diese Weise können wir z.B. folgendes schreiben:

```
List<Person> freunde = new List<Person>();
Person karl = new Student("Karl", "Ranseier", "123456");
Person joe = new Person("Joe", "Miller");
freunde.add(karl);
freunde.add(joe);
```

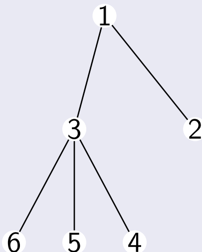
So haben wir *Liste* sehr verallgemeinert!

Beispiel

```
Tree t1 = new Tree(1);
Tree t2 = new Tree(2);
Tree t3 = new Tree(3);
Tree t4 = new Tree(4);
Tree t5 = new Tree(5);
Tree t6 = new Tree(6);
t1.addChild(t2);
t1.addChild(t3);
t3.addChild(t4);
t3.addChild(t5);
t3.addChild(t6);
```

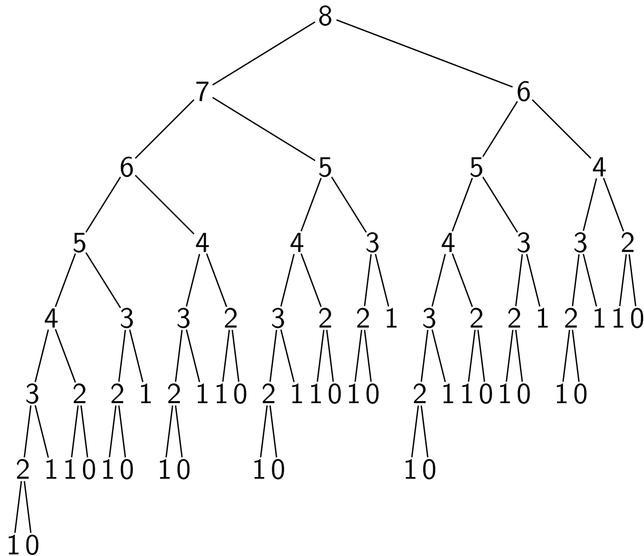
Beispiel

```
Tree t1 = new Tree(1);
Tree t2 = new Tree(2);
Tree t3 = new Tree(3);
Tree t4 = new Tree(4);
Tree t5 = new Tree(5);
Tree t6 = new Tree(6);
t1.addChild(t2);
t1.addChild(t3);
t3.addChild(t4);
t3.addChild(t5);
t3.addChild(t6);
```



Beispiel Fibonacci-Bäume

```
static Tree fiboTree(int n) {
  if(n < 2) {
    return new Tree(n);
  }
  else {
    Tree t = new Tree(n);
    t.addChild(fiboTree(n − 2));
    t.addChild(fiboTree(n − 1));
    return t;
  }
}
```



RWTH AACHEN

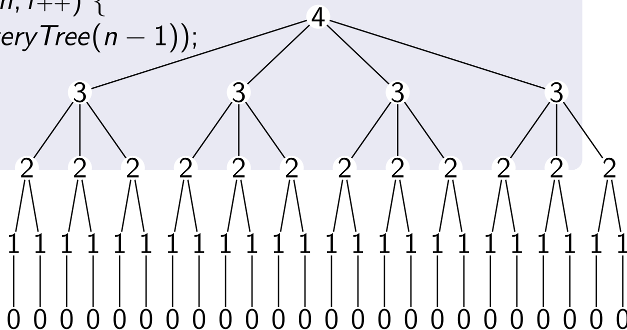
Noch ein Beispiel

```
static Tree misteryTree(int n) {
  Tree t = new Tree(n);
  for(int i = 0; i < n; i++) {
    t.addChild(misteryTree(n - 1));
  }
  return t;
}
```

Welchen Baum erzeugt *misteryTree*(4)?

Noch ein Beispiel

```
static Tree misteryTree(int n) {
  Tree t = new Tree(n);
  for(int i = 0; i < n; i++) {
    t.addChild(misteryTree(n − 1));
  }
  return t;
}
```



Wir berechnen die Summe aller Knoten eines Baumes.

```
public int totalSumRecursive() {
  int sum = getRoot();
  List⟨Tree⟩ children = getChildren();
  while(!children.isempty()) {
    sum = sum + children.head().totalSumRecursive();
    children = children.tail();
  }
  return sum;
}
```

Diese Funktion ist halbwegs übersichtlich und einfach.

Später werden wir dies noch verbessern!

Jetzt eine (schlechtere) Lösung ohne Rekursion.

```
public int totalSumIterative() {
  int sum = 0;
  List⟨Tree⟩ treesToSum = new List⟨Tree⟩();
  treesToSum = treesToSum.add(this);
  while(!treesToSum.isempty()) {
    Tree tree = treesToSum.head();
    treesToSum = treesToSum.tail();
    sum = sum + tree.getRoot();
    List⟨Tree⟩ children = tree.getChildren();
    while(!children.isempty()) {
      treesToSum = treesToSum.add(children.head());
      children = children.tail();
    }
  }
  return sum;
}
```

Überblick

2 Rekursion

- Rekursive Methoden
- Backtracking
- Memorization
- Einfache rekursive Datenstrukturen
- Bäume
- Aufzählen, Untermengen, Permutationen, Bitmengen

Aufzählen

Oft müssen wir verschiedene Dinge *aufzählen*.

Zum Beispiel die Untermengen einer Menge:

Sei $M = \{3, 7, 25\}$.

Die Untermengen von M sind

$$\emptyset, \{3\}, \{7\}, \{25\}, \{3, 7\}, \{3, 25\}, \{7, 25\}, \{3, 7, 25\}.$$

RWTH AACHEN

Neues Kapitel

1. Einführung in die objektorientierte Programmierung
2. Rekursion
3. **Fehler finden und vermeiden**
4. Objektorientiertes Design
5. Effiziente Programme
6. Nebenläufigkeit
7. Laufzeitfehler
8. Refactoring
9. Persistenz
10. Eingebettete Systeme
11. Funktionale Programmierung
12. Logische Programmierung

Überblick

3. Fehler finden und vermeiden
   - Contract Programming und der Hoare-Kalkül
   - Debugging
   - Testen

Hoare-Tripel

Ein *Hoare-Tripel* ist $\langle\phi\rangle\ P\ \langle\psi\rangle$, wobei

- ϕ ist die *Vorbedingung*,
- P ist ein Programm,
- ψ ist die *Nachbedingung*.

Falls immer, wenn die Vorbedingung vor dem Ausführen von P wahr ist, die Nachbedingung nach dem Ausführen von P wahr ist, dann ist das Tripel *korrekt*.

Hoare-Tripel

Beispiele

- $\langle x = 5 \rangle\ x = x * x;\ \langle x = 25 \rangle$
- $\langle x = 5 \wedge i \geq 4 \rangle$
  while$(i > 0)\ \{x = x - x * x; i = i - 1; \}$
  $\langle x \leq -176820 \rangle$
- $\langle x = 5 \rangle$ **if**$(y > 0)\ \{x = x + y; \}\ \langle x \geq 4 \rangle$
- $\langle x = 3 \rangle$ **while**$(x > 0)\ \{x = x + 1; \}\ \langle x = 10 \rangle$

Vorsicht: Nur *nach* dem Ausführen des Programms muß die Nachbedingung erfüllt sein.

Obige Hoare-Tripel sind alle korrekt.

Hoare-Tripel

Ein größeres Beispiel: Fibonacci-Zahlen

$\langle n \geq 0 \rangle$

$\textbf{int}\ a = 0;$

$\textbf{int}\ b = 1;$

$\textbf{int}\ k = n;$

$\textbf{while}(k > 0)\ \{$

$\quad b = a + b;$

$\quad a = b - a;$

$\quad k = k - 1;$

$\}$

$\langle a = F_n \rangle$

Wir können die Korrektheit durch Induktion beweisen.
Der *Hoare-Kalkül* dient zum mechanischen Herleiten von korrekten Hoare-Tripeln.
Er besteht aus verschiedenen *Schlußregeln*.

RWTHAACHEN

1. $$\frac{}{\langle\phi\rangle\ \{\}\ \langle\phi\rangle}$$ leere Anweisung
2. $$\frac{}{\langle\phi[x/t]\rangle\ x = t;\ \langle\phi\rangle}$$ Zuweisungsregel
3. $$\frac{\langle\phi\rangle\ P\ \langle\psi\rangle}{\langle\alpha\rangle\ P\ \langle\psi\rangle}\ \alpha \rightarrow \phi$$ Konsequenzregel 1
4. $$\frac{\langle\phi\rangle\ P\ \langle\psi\rangle}{\langle\phi\rangle\ P\ \langle\beta\rangle}\ \psi \rightarrow \beta$$ Konsequenzregel 2
5. $$\frac{\langle\phi\rangle\ P\ \langle\psi\rangle \qquad \langle\psi\rangle\ Q\ \langle\beta\rangle}{\langle\phi\rangle\ P; Q\ \langle\beta\rangle}$$ Sequenzregel
6. $$\frac{\langle\phi \wedge B\rangle\ P\ \langle\psi\rangle}{\langle\phi\rangle\ \textbf{if}(B)\ \{P\}\ \langle\psi\rangle}\ \phi \wedge \neg B \rightarrow \psi$$ Bedingungsregel 1
7. $$\frac{\langle\phi \wedge B\rangle\ P\ \langle\psi\rangle \qquad \langle\phi \wedge \neg B\rangle\ Q\ \langle\psi\rangle}{\langle\phi\rangle\ \textbf{if}(B)\ \{P\}\ \textbf{else}\ \{Q\}\ \langle\psi\rangle}$$ Bedingungsregel 2
8. $$\frac{\langle\phi \wedge B\rangle\ P\ \langle\phi\rangle}{\langle\phi\rangle\ \textbf{while}(B)\ \{P\}\ \langle\phi \wedge \neg B\rangle}$$ Schleifenregel

Herleitung des Fibonacci-Programms

Wir verwenden zweimal die Zuweisungsregel:

$$\frac{}{\langle 0 = 0 \wedge 1 = 1 \rangle \; \textbf{int} \; a = 0; \; \langle a = 0 \wedge 1 = 1 \rangle}$$

$$\frac{}{\langle a = 0 \wedge 1 = 1 \rangle \; \textbf{int} \; b = 1; \; \langle a = 0 \wedge b = 1 \rangle}$$

Und zweimal Konsequenzregeln (eigentlich unnötig):

$$\frac{\langle 0 = 0 \wedge 1 = 1 \rangle \; \textbf{int} \; a = 0; \; \langle a = 0 \wedge 1 = 1 \rangle}{\langle 0 = 0 \rangle \; \textbf{int} \; a = 0; \; \langle a = 0 \rangle}$$

$$\frac{\langle a = 0 \wedge 1 = 1 \rangle \; \textbf{int} \; b = 1; \; \langle a = 0 \wedge b = 1 \rangle}{\langle a = 0 \rangle \; \textbf{int} \; b = 1; \; \langle a = 0 \wedge b = 1 \rangle}$$

Und zweimal Konsequenzregeln (Wiederholung):

$$\frac{\langle 0 = 0 \land 1 = 1 \rangle \ \textbf{int}\ a = 0; \ \langle a = 0 \land 1 = 1 \rangle}{\langle 0 = 0 \rangle \ \textbf{int}\ a = 0; \ \langle a = 0 \rangle}$$

$$\frac{\langle a = 0 \land 1 = 1 \rangle \ \textbf{int}\ b = 1; \ \langle a = 0 \land b = 1 \rangle}{\langle a = 0 \rangle \ \textbf{int}\ b = 1; \ \langle a = 0 \land b = 1 \rangle}$$

Jetzt kommt die Sequenzregel:

$$\frac{\langle 0 = 0 \rangle \ \textbf{int}\ a = 0; \ \langle a = 0 \rangle \qquad \langle a = 0 \rangle \ \textbf{int}\ b = 1; \ \langle a = 0 \land b = 1 \rangle}{\langle 0 = 0 \rangle \ \textbf{int}\ a = 0; \textbf{int}\ b = 1; \ \langle a = 0 \land b = 1 \rangle}$$

Dies ergibt das erste Zwischenergebnis T_1:

$$T_1 \equiv \boxed{\langle 0 = 0 \rangle \ \textbf{int}\ a = 0; \textbf{int}\ b = 1; \ \langle a = 0 \land b = 1 \rangle}$$

Weiter mit Zuweisung und Konsequenz:

$$\frac{}{\langle a = 0 \land b = 1 \land n = n \rangle \ \textbf{int} \ k = n; \ \langle a = 0 \land b = 1 \land k = n \rangle}$$

$$\frac{\langle a = 0 \land b = 1 \land n = n \rangle \ \textbf{int} \ k = n; \ \langle a = 0 \land b = 1 \land k = n \rangle}{\langle a = 0 \land b = 1 \rangle \ \textbf{int} \ k = n; \ \langle a = 0 \land b = 1 \land k = n \rangle}$$

Wir verwenden T_1 in der Sequenzregel:

$$\frac{T_1 \qquad \langle a = 0 \land b = 1 \rangle \ \textbf{int} \ k = n; \ \langle a = 0 \land b = 1 \land k = n \rangle}{\langle 0 = 0 \rangle \ \textbf{int} \ a = 0; \textbf{int} \ b = 1; \textbf{int} \ k = n; \ \langle a = 0 \land b = 1 \land k = n \rangle}$$

Dies sei unser zweites Zwischenergebnis T_2:

$$\boxed{\langle 0 = 0 \rangle \ \textbf{int} \ a = 0; \textbf{int} \ b = 1; \textbf{int} \ k = n; \ \langle a = 0 \land b = 1 \land k = n \rangle}$$

Etwas übersichtlicher schreibt sich T_2 so:

$\langle 0 = 0 \rangle$
int $a = 0;$
int $b = 1;$
int $k = n;$
$\langle a = 0 \wedge b = 1 \wedge k = n \rangle$

Die Vorbedingung ist „leer“, sie gilt immer.

Die Nachbedingung garantiert, daß die Variablen die richtigen Inhalte haben.

Wir versuchen nun, folgendes Hoare-Tripel herzuleiten:

$\langle a = 0 \wedge b = 1 \wedge k = n \wedge n \geq 0 \rangle$
while$(k > 0)$ {
  $b = a + b;$
  $a = b - a;$
  $k = k - 1;$
}
$\langle a = F_n \rangle$

Zusammen mit T_2 und der Sequenzregel können wir dann fast das ganze Fibonacci-Programm herleiten. T_2 lautete:

$\langle 0 = 0 \rangle$ **int** $a = 0;$ **int** $b = 1;$ **int** $k = n;$ $\langle a = 0 \wedge b = 1 \wedge k = n \rangle$

Leider passen die zwei Teile nicht genau zusammen.

RWTHAACHEN

Statt

$$\langle 0 = 0 \rangle\ \textbf{int}\ a = 0; \textbf{int}\ b = 1; \textbf{int}\ k = n;\ \langle a = 0 \wedge b = 1 \wedge k = n \rangle$$

benötigen wir

$$\langle n \geq 0 \rangle\ \textbf{int}\ a = 0; \textbf{int}\ b = 1; \textbf{int}\ k = n;\ \langle a = 0 \wedge b = 1 \wedge k = n \wedge n \geq 0 \rangle.$$

Wenn so etwas passiert, kann man es oft so reparieren:

Wir gehen die ganze Herleitung von T_2 noch einmal durch und fügen $n \geq 0$ sowohl bei allen Vor- als auch Nachbedingungen ein.

Hier geht das ohne Problem.

Auf diese Weise erhalten wir das Zwischenergebnis T_3:

$\langle n \geq 0 \rangle$
int $a = 0;$
int $b = 1;$
int $k = n;$
$\langle a = 0 \wedge b = 1 \wedge k = n \wedge n \geq 0 \rangle$

und wir versuchen jetzt dies herzuleiten:

$\langle a = 0 \wedge b = 1 \wedge k = n \wedge n \geq 0 \rangle$
while$(k > 0)$ {
  $b = a + b;$
  $a = b - a;$
  $k = k - 1;$
}
$\langle a = F_n \rangle$

Mit einer Konsequenzregel erhalten wir:

$\langle a = F_{n-k} \wedge b = F_{n-k+1} \wedge k \geq 0 \rangle \equiv: \phi$

while$(k > 0)$ $\{$

$\quad b = a + b;$

$\quad a = b - a;$

$\quad k = k - 1;$

$\}$

$\langle a = F_n \rangle$

$\langle a = 0 \wedge b = 1 \wedge k = n \wedge n \geq 0 \rangle \equiv: \psi$

while$(k > 0)$ $\{$

$\quad b = a + b;$

$\quad a = b - a;$

$\quad k = k - 1;$

$\}$

$\langle a = F_n \rangle$

Denn $\psi \rightarrow \phi$. (Vorbedingung „aufgeweicht“.)

RWTHAACHEN

Wir müssen also noch herleiten:

$$\langle a = F_{n-k} \wedge b = F_{n-k+1} \wedge k \geq 0 \rangle$$

```
while(k > 0) {
   b = a + b;
   a = b - a;
   k = k - 1;
}
```

$$\langle a = F_n \rangle$$

Um dies zu schaffen, müssen wir die Schleifenregel verwenden:

$$\frac{\langle \phi \wedge B \rangle \; P \; \langle \phi \rangle}{\langle \phi \rangle \; \textbf{while}(B) \; \{P\} \; \langle \phi \wedge \neg B \rangle}$$

Wir wählen $a = F_{n-k} \wedge b = F_{n-k+1} \wedge k \geq 0$ als ϕ und $k > 0$ als B.

Die Schleifenregel wird so angewendet:

$$\langle a = F_{n-k} \wedge b = F_{n-k+1} \wedge k \geq 0 \wedge k > 0 \rangle$$
$$b = a + b;$$
$$a = b - a;$$
$$k = k - 1;$$
$$\langle a = F_{n-k} \wedge b = F_{n-k+1} \wedge k \geq 0 \rangle$$

$$\langle a = F_{n-k} \wedge b = F_{n-k+1} \wedge k \geq 0 \rangle$$

while$(k > 0)$ {
  $b = a + b;$
  $a = b - a;$
  $k = k - 1;$
}

$$\langle a = F_{n-k} \wedge b = F_{n-k+1} \wedge k \geq 0 \wedge \neg(k > 0) \rangle$$

Mit einer Abschwächung der Nachbedingung erhalten wir:

$$\langle a = F_{n-k} \wedge b = F_{n-k+1} \wedge k \geq 0 \rangle$$

while$(k > 0)$ $\{$

$b = a + b;$

$a = b - a;$

$k = k - 1;$

$\}$

$$\langle a = F_{n-k} \wedge b = F_{n-k+1} \wedge k \geq 0 \wedge \neg(k > 0) \rangle$$

$$\langle a = F_{n-k} \wedge b = F_{n-k+1} \wedge k \geq 0 \rangle$$

while$(k > 0)$ $\{$

$b = a + b;$

$a = b - a;$

$k = k - 1;$

$\}$

$$\langle a = F_n \rangle$$

RWTHAACHEN

Was müssen jetzt noch dies herleiten:

$\langle a = F_{n-k} \wedge b = F_{n-k+1} \wedge k > 0 \rangle$
$b = a + b;$
$a = b - a;$
$k = k - 1;$
$\langle a = F_{n-k} \wedge b = F_{n-k+1} \wedge k \geq 0 \rangle$

Mit der Zuweisungsregel erhalten wir diese Tripel:

$\langle a = F_{n-k} \wedge a + b = F_{n-k+2} \wedge k > 0 \rangle$
$b = a + b;$
$\langle a = F_{n-k} \wedge b = F_{n-k+2} \wedge k > 0 \rangle$ und
$\langle b - a = F_{n-k+1} \wedge b = F_{n-k+2} \wedge k > 0 \rangle$
$a = b - a;$
$\langle a = F_{n-k+1} \wedge b = F_{n-k+2} \wedge k > 0 \rangle$ und
$\langle a = F_{n-k+1} \wedge b = F_{n-k+1} \wedge k > 0 \rangle$
$k = k - 1;$
$\langle a = F_{n-k} \wedge b = F_{n-k+2} \wedge k + 1 > 0 \rangle$

Stellt man alles zusammen, haben wir dieses Tripel erzeugt:

$\langle n \geq 0 \rangle$
int $a = 0;$
int $b = 1;$
int $k = n;$
while$(k > 0)$ $\{$
  $b = a + b;$
  $a = b - a;$
  $k = k - 1;$
$\}$
$\langle a = F_n \rangle$

Dies war eine lange Herleitung.

In Praxis oft zu schwierig oder teuer.

Übersichtliche Zusammenfassung

(1) Zuweisungsregel

$\langle n \geq 0 \rangle$ **int** $a = 0;$ $\langle n \geq 0 \wedge a = 0 \rangle$

(2) Zuweisungsregel

$\langle n \geq 0 \wedge a = 0 \rangle$ **int** $b = 1;$ $\langle n \geq 0 \wedge a = 0 \wedge b = 1 \rangle$

(3) Sequenzregel aus (1), (2)

$\langle n \geq 0 \rangle$ **int** $a = 0;$ **int** $b = 1;$ $\langle n \geq 0 \wedge a = 0 \wedge b = 1 \rangle$

(4) Zuweisungsregel

$\langle n \geq 0 \wedge a = 0 \wedge b = 1 \rangle$ **int** $k = n;$ $\langle n \geq 0 \wedge a = 0 \wedge b = 1 \wedge k = n \rangle$

(5) Sequenzregel aus (3), (4)

$\langle n \geq 0 \rangle$ **int** $a = 0;$ **int** $b = 1;$ **int** $k = n;$ $\langle n \geq 0 \wedge a = 0 \wedge b = 1 \wedge k = n \rangle$

(6) Zuweisungsregel

$\langle a = F_{n-k} \wedge a + b = F_{n-k+2} \wedge k > 0 \rangle$

$b = a + b;$

$\langle a = F_{n-k} \wedge b = F_{n-k+2} \wedge k > 0 \rangle$

(7) Zuweisungsregel

$\langle b - a = F_{n-k+1} \wedge b = F_{n-k+2} \wedge k > 0 \rangle$

$a = b - a;$

$\langle a = F_{n-k+1} \wedge b = F_{n-k+2} \wedge k > 0 \rangle$

(8) Zuweisungsregel

$\langle a = F_{n-k+1} \wedge b = F_{n-k+2} \wedge k > 0 \rangle$

$k = k - 1;$

$\langle a = F_{n-k} \wedge b = F_{n-k+1} \wedge k + 1 > 0 \rangle$

(9) Sequenzregel aus (6), (7)

$\langle a = F_{n-k} \wedge a + b = F_{n-k+2} \wedge k > 0 \rangle$

$b = a + b;$

$a = b - a;$

$\langle a = F_{n-k+1} \wedge b = F_{n-k+2} \wedge k > 0 \rangle$

(10) Sequenzregel aus (9), (8)

$\langle a = F_{n-k} \wedge b = F_{n-k+1} \wedge k \geq 0 \wedge k > 0 \rangle$

$b = a + b;$

$a = b - a;$

$k = k - 1;$

$\langle a = F_{n-k} \wedge b = F_{n-k+1} \wedge k \geq 0 \rangle$

(11) Schleifenregel aus (10)

$\langle a = F_{n-k} \wedge b = F_{n-k+1} \wedge k \geq 0 \rangle$

while$(k > 0)$ {
  $b = a + b;$
  $a = b - a;$
  $k = k - 1;$
}

$\langle a = F_{n-k} \wedge b = F_{n-k+1} \wedge k \geq 0 \wedge \neg(k > 0) \rangle$

(12) Konsequenzregel aus (11)

$\langle a = 0 \wedge b = 1 \wedge k = n \wedge n \geq 0 \rangle$

while$(k > 0)$ {
  $b = a + b;$
  $a = b - a;$
  $k = k - 1;$
}

$\langle a = F_n \rangle$

(13) Sequenzregel aus (5), (12)

$\langle n \geq 0 \rangle$
int $a = 0;$
int $b = 1;$
int $k = n;$
while$(k > 0)$ {
$\quad b = a + b;$
$\quad a = b - a;$
$\quad k = k - 1;$
}
$\langle a = F_n \rangle$

Damit ist das gesamte Hoare-Tripel in dreizehn Schritten hergeleitet.

Vorsicht: Der Hoare-Kalkül geht von echten ganzen Zahlen aus und nicht von beschränkter Arithmetik.

Das folgende Tripel ist korrekt:

$\langle k \geq 0 \rangle$
while$(k >= 0)$ $\{$
  $k = k + 1;$
$\}$
$\langle k = 25 \rangle$

Das Tripel sagt, daß *nach* Verlassen der **while**-Schleife die Variable k den Wert 25 enthält.

Das ist tatsächlich wahr! (Denn die **while**-Schleife wird nie verlassen.)

Hilfsvariablen

$\langle n \geq 0 \rangle$
$k = n;$
$z = 0;$
while$(k > 0)$ $\{$
$\quad z = z + 1;$
$\quad k = k - 1;$
$\}$
$\langle z = n \rangle$

(1) Zuweisungsregel
$\langle n \geq 0 \rangle \; k = n; \; \langle n \geq 0 \wedge k = n \rangle$

(2) Zuweisungsregel
$\langle n \geq 0 \wedge k = n \rangle \; z = 0; \; \langle n \geq 0 \wedge k = n \wedge z = 0 \rangle$

(3) Sequenzregel aus (1), (2)
$\langle n \geq 0 \rangle \; k = n; z = 0; \; \langle n \geq 0 \wedge k = n \wedge z = 0 \rangle$

(4) Zuweisungsregel

$\langle z + k = n \land k > 0 \rangle\ z = z + 1;\ \langle z - 1 + k = n \land k > 0 \rangle$

(5) Zuweisungsregel

$\langle z + k - 1 = n \land k > 0 \rangle\ k = k - 1;\ \langle z + k = n \land k \geq 0 \rangle$

(6) Sequenzregel aus (4), (5)

$\langle z + k = n \land k > 0 \rangle\ z = z + 1; k = k - 1;\ \langle z + k = n \land k \geq 0 \rangle$

(7) Schleifenregel aus (6)

$\langle z + k = n \land k \geq 0 \rangle$

```
while(k > 0) {
  z = z + 1;
  k = k - 1;
}
```

$\langle z + k = n \land k \geq 0 \land \neg(k > 0) \rangle$

(8) Konsequenzregel aus (7)

$\langle n \geq 0 \wedge k = n \wedge z = 0 \rangle$

```
while(k > 0) {
  z = z + 1;
  k = k - 1;
}
```

$\langle z = n \rangle$

(9) Sequenzregel aus (3), (8)

$\langle n \geq 0 \rangle$

```
k = n;
z = 0;
while(k > 0) {
  z = z + 1;
  k = k - 1;
}
```

$\langle z = n \rangle$

Wir konnten beweisen, daß am Ende $z = n$ gilt.

Wie steht es mit diesem Programm:

```
z = 0;
while(n > 0) {
  z = z + 1;
  n = n − 1;
}
```

Wie beweisen wir, daß z am Ende denselben Wert hat wie n am Anfang?

Dies scheint nicht ausdrückbar zu sein.

Hilfsvariablen

Lösung: Verwende eine Hilfsvariable.

$\langle n' = n \rangle$
$z = 0;$
while$(n > 0)$ $\{$
  $z = z + 1;$
  $n = n - 1;$
$\}$
$\langle z = n' \rangle$

Die Hilfsvariable n' kommt im Programm gar nicht vor.

Die Herleitung im Hoare-Kalkül geschieht analog zum letzten Beispiel.

$\langle true \rangle$
if$(x > y)$ {
  $r = x;$
}
else {
  $r = y;$
}
$\langle r = \max(x, y) \rangle$

(1) Zuweisungsregel
$\langle x = \max(x, y) \rangle \; r = x; \; \langle r = \max(x, y) \rangle$

(2) Konsequenzregel aus (1)
$\langle x > y \rangle \; r = x; \; \langle r = \max(x, y) \rangle$

(3) Zuweisungsregel
$\langle y = \max(x, y) \rangle \; r = y; \; \langle r = \max(x, y) \rangle$

(4) Konsequenzregel aus (3)
$\langle \neg(x > y) \rangle \; r = y; \; \langle r = \max(x, y) \rangle$

RWTHAACHEN

Die 2. Bedingungsregel lautet:

$$\frac{\langle \phi \wedge B \rangle \; P \; \langle \psi \rangle \qquad \langle \phi \wedge \neg B \rangle \; Q \; \langle \psi \rangle}{\langle \phi \rangle \; \textbf{if}(B) \; \{P\} \; \textbf{else} \; \{Q\} \; \langle \psi \rangle}$$

(5) Bedingungsregel aus (2), (4)

$\langle true \rangle$
$\textbf{if}(x > y) \; \{$
$\quad r = x;$
$\}$
$\textbf{else} \; \{$
$\quad r = y;$
$\}$
$\langle r = \max(x, y) \rangle$

Die Herleitung läßt sich gut „rückwärts“ finden.

Kompakte Herleitungen im Hoare-Kalkül

Die Sequenzregel

$$\frac{\langle\phi\rangle\ P\ \langle\psi\rangle \qquad \langle\psi\rangle\ Q\ \langle\beta\rangle}{\langle\phi\rangle\ P; Q\ \langle\beta\rangle}$$

können wir kompakt geschrieben so anwenden:

$\langle\phi\rangle$
P
$\langle\psi\rangle$
Q
$\langle\beta\rangle$

Die Herleitungen von $\langle\phi\rangle\ P\ \langle\psi\rangle$ und $\langle\psi\rangle\ Q\ \langle\beta\rangle$ werden einfach „aneinandergehängt“.

Die Konsequenzregel

$$\frac{\langle\phi\rangle\ P\ \langle\psi\rangle}{\langle\alpha\rangle\ P\ \langle\beta\rangle}\ \alpha \rightarrow \phi \wedge \beta \rightarrow \psi$$

wenden wir kompakt geschrieben so an:

$\langle\alpha\rangle$
$\langle\phi\rangle$
P
$\langle\psi\rangle$
$\langle\beta\rangle$

„Innen“ wird $\langle\phi\rangle\ P\ \langle\psi\rangle$ abgeleitet. Implikationen $\alpha \rightarrow \phi$ und $\psi \rightarrow \beta$ werden „von oben nach unter“ eingefügt.

Die Schleifenregel

$$\frac{\langle \phi \wedge B \rangle \; P \; \langle \phi \rangle}{\langle \phi \rangle \; \textbf{while}(B) \; \{P\} \; \langle \phi \wedge \neg B \rangle}$$

wird kompakt so angewandt:

$\langle \phi \rangle$
while(B) $\{$
  $\langle \phi \wedge B \rangle$
  P
  $\langle \phi \rangle$
$\}$
$\langle \phi \wedge \neg B \rangle$

Beispiel

Wir leiten das folgende Hoare-Tripel durch die kompakte Schreibweise her:

$\langle n = n' \wedge n \geq 0 \rangle$
$k = 1;$
while$(n > 0)$ $\{$
  $k = 2 * k;$
  $n = n - 1;$
$\}$
$\langle k = 2^{n'} \rangle$

In der Herleitung wird keine Programmanweisung mehrfach geschrieben.

$\langle n = n' \wedge n \geq 0 \rangle$
$\langle n = n' \wedge 1 = 1 \wedge n \geq 0 \rangle$
$k = 1;$
$\langle n = n' \wedge k = 1 \wedge n \geq 0 \rangle$
$\langle k = 2^{n'-n} \wedge n \geq 0 \rangle$
while$(n > 0)$ {
  $\langle k = 2^{n'-n} \wedge n \geq 0 \wedge n > 0 \rangle$
  $\langle 2k = 2^{n'-n+1} \wedge n > 0 \rangle$
  $k = 2 * k;$
  $\langle k = 2^{n'-n+1} \wedge n > 0 \rangle$
  $\langle k = 2^{n'-(n-1)} \wedge n - 1 \geq 0 \rangle$
  $n = n - 1;$
  $\langle k = 2^{n'-n} \wedge n \geq 0 \rangle$
}
$\langle k = 2^{n'-n} \wedge n \geq 0 \wedge \neg(n > 0) \rangle$
$\langle k = 2^{n'} \rangle$

Die Bedingungsregel

$$\frac{\langle \phi \wedge B \rangle \; P \; \langle \psi \rangle \qquad \langle \phi \wedge \neg B \rangle \; Q \; \langle \psi \rangle}{\langle \phi \rangle \; \textbf{if}(B) \; \{P\} \; \textbf{else} \; \{Q\} \; \langle \psi \rangle}$$

wird zuletzt kompakt so angewandt:

$\langle \phi \rangle$
if(B) $\{$
  $\langle \phi \wedge B \rangle$
  P
  $\langle \psi \rangle$
$\}$
else $\{$
  $\langle \phi \wedge \neg B \rangle$
  Q
  $\langle \psi \rangle$
$\}$
$\langle \psi \rangle$

Terminierung von Programmen

Wir haben uns mit *partieller Korrektheit* beschäftigt:

Falls das Programm terminiert, *dann* ist das Ergebnis korrekt.

Wie zeigen wir, *daß* es terminiert?

Terminiert dieses Programm? Für alle n??

```
while(n > 1) {
  if(n % 2 == 1) { // ist n ungerade?
    n = 3 * n + 1;
  }
  else {
    n = n/2;
  }
}
```

Einfache Terminierungsbeweise

Nur **while**-Schleifen können nicht terminieren.

Für die totale Korrektheit eines Programms genügt es zu zeigen, daß alle **while**-Schleifen terminieren.

Die Schleife **while**(B) $\{P\}$ terminiert, falls $\langle t = t' \wedge B \rangle \ P \ \langle t < t' \wedge t \geq 0 \rangle$ gilt.

Hier ist t eine ganzzahliger Ausdruck und t' eine Hilfsvariable.

Der Ausdruck t wird immer kleiner, kann aber nicht negativ werden. Daher muß die Schleife irgendwann enden. Wir nennen t die *Variante* der Schleife.

Einfache Terminierungsbeweise

Beispiel:

$$
\textbf{while}(k > 0)\ \{\\
\quad k = k - 1;\\
\}
$$

Wir wählen $t := k$. Dieser Ausdruck wird immer kleiner, kann aber nicht negativ werden.

Wir können zeigen, daß
$\langle k = k' \wedge k > 0 \rangle\ k = k - 1;\ \langle k < k' \wedge k \geq 0 \rangle$ gilt.

Damit terminiert diese Schleife.

Schwierigere Situationen

Für die totale Korrektheit muß (und kann) man manchmal nicht von *allen* **while**-Schleifen isoliert die Terminierung beweisen.

Der Kontext ist wichtig.

Beispiel:

```
if(x > 0) {
  while(y > 0) {
    y = y − x;
  }
}
```

Dieses Programm terminiert immer.

Die Variante ist einfach y.

Contract Programming

Contract Programming ist durch die Hoare-Tripel motiviert.

Wenn wir ein Programm P schreiben, machen wir einen „Vertrag“:

Wir verlangen im Vertrag, daß unser Programm nur bestimmte Eingaben erhält (Vorbedingungen).

Wir garantieren im Vertrag, daß die Ausgabe unseres Programms bestimmte Eigenschaften hat (Nachbedingungen).

Wir versprechen also, daß ein Hoare-Tripel $\langle\phi\rangle\ P\ \langle\psi\rangle$ gültig ist.

Geht etwas schief, dann ist unser Programm „schuldig“, falls ϕ gilt, aber ψ nicht.

Die Vor- und Nachbedingungen lassen sich unter Umständen im Programm leicht testen.

Beispiel

Wir möchten zu $n > 0$ ein k finden mit $k^2 \leq n < (k+1)^2$.

```
static int sqrt(int n) {
  assert n > 0; // Vorbedingung
  int l = 1, r = n;
  while(r - l > 1) {
    int k = (l + r)/2;
    if(k * k <= n) {
      l = k;
    }
    else {
      r = k;
    }
  }
  assert l * l <= n && n < (l + 1) * (l + 1); // Nachbedingung
  return l;
}
```

Programme können neben dem eigentlichen Ergebnis weitere Informationen liefern, die eine Überprüfung der Korrektheit des Ergebnissen erleichtern.

Beispiel: Berechnung des größten gemeinsamen Teilers

```
int gcd1(int a, int b) {
  while(b > 0) {
    int temp = b;
    b = a%b;
    a = temp;
  }
  return a;
}
```

Dies ist der euklidische Algorithmus.

Wie überprüfen, daß Ergebnis korrekt?

Der euklidische Algorithmus berechnet den größten gemeinsamen Teiler.

Der *erweiterte* euklidische Algorithmus berechnet den größten gemeinsamen Teiler zweier Zahlen a und b und zusätzlich zwei ganze Zahlen x und y mit

$$xa + yb = gcd(a, b).$$

Falls a und b Vielfache von r sind und $xa + yb = r$, dann *muß* $r = gcd(a, b)$ sein.

Mithilfe von x und y können wir also die Korrekheit beweisen.

```
int gcd2(int a, int b) {
  assert a > 0 && b > 0;  // Vorbedingung
  int x = 1, y = 0, xx = 0, yy = 1, quot, temp;
  int r = a, rr = b;
  while (rr != 0) {
    quot = r/rr;
    temp = rr; rr = r - quot * rr; r = temp;
    temp = xx; xx = x - quot * xx; x = temp;
    temp = yy; yy = y - quot * yy; y = temp;
  }
   // Nachbedingung
  assert x * a + y * b == r && a%r == 0 && b%r == 0;
  return r;
}
```

Die **assert**-Anweisung überprüft eine Bedingung.

Vorsicht: Assertions müssen im Java-Laufzeitsystem aktiviert werden (durch `java -ea GCD`).

Java unterstützt Contract Programming nicht direkt. Beispiel in D:

```
int gcd(int a, int b, out int x, out int y)
in {assert(a > 0 && b > 0);}
out(r) {assert(x * a + y * b == r && a%r == 0 && b%r == 0);}
body {
  x = 1; y = 0;
  int xx = 0, yy = 1, quot, temp;
  int r = a, rr = b;
  while (rr != 0) {
    quot = r/rr;
    temp = rr; rr = r - quot * rr; r = temp;
    temp = xx; xx = x - quot * xx; x = temp;
    temp = yy; yy = y - quot * yy; y = temp;
  }
  return r;
}
```

Contract Programming – Zusammenfassung

- Hilft Fehler zu lokalisieren.
- Auch wenn Programm schon Jahre in Verwendung.
- Testen der Nachbedingung muß *effizient* sein.
- Benötigt *Zertifizierende Algorithmen*.
- Außer Vor- und Nachbedingung auch *Invarianten*.

Ein Zertifizierender Algorithmus liefert neben dem Ergebnis ein *Zertifikat*, das eine einfache Überprüfung erlaubt.

Selbstkorrigierende Programme

Defensive Programmierung: Fehlertolerant sein.

Selbstkorrigierendes Programm:

1. Ergebnis und Zertifikat mit einem effizienten, aber komplizierten Programm berechnen.
2. Zertifikat überprüfen.
3. Überprüfung negativ: Ergebnis neu berechnen mit einem langsamen, aber einfachen und sicheren Programm.

```
int gcd3(int a, int b) {
  int x = 1, y = 0, xx = 0, yy = 1, quot, temp;
  int r = a, rr = b;
  while (rr != 0) {
    quot = r/rr;
    temp = rr; rr = r - quot * rr; r = temp;
    temp = xx; xx = x - quot * xx; x = temp;
    temp = yy; yy = y - quot * yy; y = temp;
  }
  if(!(x * a + y * b == r && a%r == 0 && b%r == 0)) {
    r = a;
    while(a%r != 0 || b%r != 0) {
      r = r - 1;
    }
  }
  return r;
}
```

Überblick

3 Fehler finden und vermeiden

- Contract Programming und der Hoare-Kalkül
- Debugging
- Testen

Debugging – Fehler beseitigen

Fehler entstehen weniger, wenn das Programm gut geschrieben wird.

- Gute Dokumentation
- Gute Klassenstruktur
- Aussagekräftige Namen (Klassen, Methoden, Variablen)
- Einheitliche Namensstruktur
- Kurze Methoden
- Code Duplication vermeiden
- ... (später mehr)

Debugging – Fehler beseitigen

Fehler können leichter entdeckt werden, wenn das Programm gut geschrieben wird.

- Debug-Ausgaben, „verbose mode“ (abschaltbar)
- **asserts** einbauen
- Contract Programming
- Tests parallel entwickeln
- Methoden einzeln testbar auslegen
- Zusätzliche Visualisierung
- ...

Debugging – Fehler beseitigen

Fehler finden:

- Programm genau inspizieren
- Code Walkthrough
- Teile nacheinander auskommentieren
- Debug-Ausgaben einbauen
- **asserts** einbauen
- Breakpoints setzen, Variablen inspizieren
- Einzelschrittmodus

Einschub: Statische Methoden

```
class Kegelbahn {
  ...
  public static double sqdistance(Point p, Point q) {
    double dx = p.x - q.x;
    double dy = p.y - q.y;
    return dx * dx + dy * dy;
  }
  ...
}
```

Eine statische Methode gehört zur Klasse, nicht zum einzelnen Objekt. Sie kann daher nicht auf Instanzvariablen zugreifen.

Ausserhalb der Klasse *Kegelbahn* können wir sie so aufrufen:

```
...
double dist = Kegelbahn.sqdistance(point1, point2);
...
```

Einschub: Mengen in der Java-Bibliothek

```
Set<Person> freunde = new HashSet<Student>();
freunde.add(karl);
freunde.add(eva);
int anzahl = freunde.size();
for(Person p : freunde) {
  System.out.println(p);
}
```

Programmieren mit solchen „Container-Klassen“ ist sehr bequem.

Später mehr dazu.

Vorsicht: Es fehlen noch wichtige Voraussetzungen.

Arten von Fehlern

Es gibt viele verschiedene Arten von Fehlern in Programmen.

- Typos (z.B. $n + 1$ statt $n - 1$)
- Fall vergessen
- Schnittstelle falsch erinnert
- Fehler im Algorithmus
- Mißverständnis
- ...

Sie sind unterschiedlich schwer zu finden und zu beseitigen.

Überblick

3. Fehler finden und vermeiden
   - Contract Programming und der Hoare-Kalkül
   - Debugging
   - Testen

Neues Kapitel

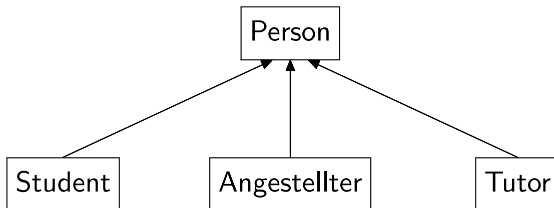
1. Einführung in die objektorientierte Programmierung
2. Rekursion
3. Fehler finden und vermeiden
4. **Objektorientiertes Design**
5. Effiziente Programme
6. Nebenläufigkeit
7. Laufzeitfehler
8. Refactoring
9. Persistenz
10. Eingebettete Systeme
11. Funktionale Programmierung
12. Logische Programmierung

Überblick

4. Objektorientiertes Design
   - Interfaces, Container, Iterator, Visitor
   - Composition, Decorator
   - Schichtenmodell und Filter-Pipelines

Mehrfachvererbung

Eine Klasse kann mehrere Unterklassen haben:

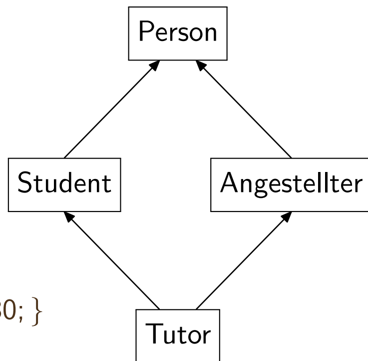


Aber: Ein *Tutor* ist ein *Student* und *Angestellter*.

Darf eine Klasse mehrere Oberklassen haben?

Das nennen wir *Mehrfachvererbung*.

```
class Student extends Person {
   ...
   double getGehalt() {return 0.0; }
   ...
}
class Angestellter extends Person {
   ...
   double getGehalt() {return 458.80; }
   ...
}
class Tutor extends Student, Angestellter {
   ...
}
...
Tutor egon = new Tutor(...);
double geld = egon.getGehalt();
```



Mehrfachvererbung

Mehrfachvererbung kann Probleme bereiten.

Java verbietet Mehrfachvererbung.

Andere Sprachen erlauben sie aber (z.B. C++).

Können wir ohne Mehrfachvererbung leben?

Ja, durch den **interface**-Mechanismus.

Abstrakte Klassen

Eine *abstrakte Klasse* kann Methoden enthalten, die nicht implementiert sind: Ihr Rumpf fehlt.

```
abstract class Tonerzeuger {
  void macheTon();
}
class Katze extends Tonerzeuger {
  void macheTon() {
    System.out.println("Miau");
  }
}
class Trompete extends Tonerzeuger {
  void macheTon() {
    System.out.println("Trara");
  }
}
```

Jeder *Tonerzeuger* hat so sicherlich *macheTon*().

Abstrakte Klassen

Abstrakte Klassen werden oft für die Spezifikation abstrakter Datentypen verwendet:

```
abstract class Stack⟨T⟩ {
  public boolean isempty();
  public T top();
  public void pop();
  public void push(T x);
}
```

Jede Unterklasse von *Stack*⟨*T*⟩ hat jetzt garantiert diese Methoden.

(Sie sollte auch die Semantik eines Stacks richtig umsetzen.)

```
class LinkedStack<T> implements Stack<T> {
  private T element = null;
  private LinkedStack<T> next = null;
  public boolean isempty() {return element == null; }
  public T top() {return element; }
  public void pop() {
    if(next.isempty()) {element = null; }
    else {
      element = next.element;
      next = next.next;
    }
  }
  public void push(T x) {
    LinkedStack<T> s = new LinkedStack<T>();
    s.element = element; s.next = next;
    next = s; element = x;
  }
}
```

LinkedStack ist eine konkrete Implementierung eines *Stack*.
(Sie ist nicht besonders schön, aber kurz...)

Es kann auch andere Implementierungen geben.

```
Stack<String> s = new LinkedStack<String>();
s.push("A");
s.push("B");
System.out.println(s.top());
```

```
LinkedStack<String> s = new LinkedStack<String>();
s.push("A");
s.push("B");
System.out.println(s.top());
```

Die erste Möglichkeit ist besser!

Warum???

Abstrakte Klassen und Interfaces

Eine abstrakte Klassen kann auch *einige* Methoden vollständig implementieren.

Wenn dies *nicht* so ist, dann gibt die abstrakte Klasse nur ein *Versprechen* ab, daß es bestimmte Methoden gibt.

So etwas nennt man auch ein *Interface*.

Ein Interface bestimmt, daß bestimmte Methoden existieren müssen.

```
interface Tonerzeuger {
   void macheTon();
}
```

Java stellt explizit **interface**s zur Verfügung.

In Sprachen mit Mehrfachvererbung werden sie durch abstrakte Klassen umgesetzt.

```
interface Tonerzeuger {
  void macheTon();
}
interface Stack<T> {
  public boolean isempty();
  public T top();
  public void pop();
  public void push(T x);
}
class LinkedStack<T> implements Stack<T> {
  ...
}
class LinkedStackMitTon<T>
      extends LinkedStack<T> implements Tonerzeuger {
  void macheTon() {
    System.out.println("Klack");
  }
}
```

Iterator

Iteratoren sind ein Standardverfahren, um auf Daten zuzugreifen.

```
public interface Iterator<E> {
  boolean hasNext(); // gibt es ein weiteres Element?
  E next(); // gib das nächste Element zurück
}
public interface Iterable<E> {
  Iterator<E> iterator(); // gibt einen Iterator über alle Elemente zurück
}
```

Ist *coll* ein *Iterable⟨String⟩*, dann sind solche Schleifen möglich:

```
Iterator<String> it = coll.iterator();
while(it.hasNext()) {
  String s = it.next();
  System.out.println(s);
}
```

Iterator

In Java gibt es eine hübsche Abkürzung.

Es sei wieder *coll* ein *Iterable⟨String⟩*. Anstatt

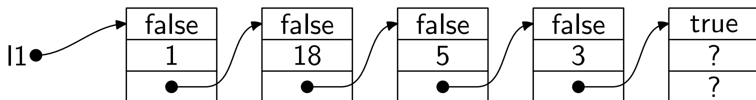
```
Iterator<String> it = coll.iterator();
while(it.hasNext()) {
   String s = it.next();
   System.out.println(s);
}
```

können wir auch dies schreiben:

```
for(String s : coll) {
   System.out.println(s);
}
```

Beispiel: Unsere Liste

Erinnern wir uns an unsere Listenimplementierung $List\langle T\rangle$.



Die öffentlichen Methoden sind

- **void** *isempty*(),
- T *head*(),
- $List\langle T\rangle$ *tail*() und
- $List\langle T\rangle$ *add*(T x).

Wir implementieren einen Iterator.

```
class ListIterator<T> implements Iterator<T> {
  private List<T> currentNode;
  public ListIterator(List<T> list) {
    currentNode = list;
  }
  public T next() {
    T result = currentNode.head();
    currentNode = currentNode.tail();
    return result;
  }
  public boolean hasNext() {
    return!currentNode.isempty();
  }
  public void remove() {}
}
```

Die Klasse *List*⟨*T*⟩ muß nun noch das Interface *Iterable*⟨*T*⟩ implementieren.

```
class List⟨T⟩ implements Iterable⟨T⟩ {
  private final boolean empty; // ist diese Liste leer?
  private final T value; // das erste Element dieser Liste
  private final List⟨T⟩ rest; // restliche Liste ohne das erste Element
  public Iterator⟨T⟩ iterator() {
    return new ListIterator⟨T⟩(this);
  }
```

Nun können wir Iteratoren für die Klasse $List\langle T\rangle$ sofort verwenden.

```
public static void main(String args[]) {
  List<String> list = new List<String>();
  list = list.add("Informatik");
  list = list.add("Physik");
  list = list.add("Mathematik");
  list = list.add("Chemie");
  list = list.add("Biologie");
  for(String s : list) {
    System.out.println(s);
  }
}
```

Einschub: Autoboxing

Können wir *List*⟨**int**⟩ verwenden?

Nein, denn **int** ist keine Klasse.

Java ist keine *reine* okjektorientierte Sprache: Nicht *alles* ist ein Objekt.

Grund: Effizienz.

Lösung in Java: Wrapper-Klassen.

Primitiver Typ	Wrapper-Klasse
int	*Integer*
double	*Double*
boolean	*Boolean*

Autoboxing

```
Set<Integer> zahlen = new HashSet<Integer>();
zahlen.add(34);
zahlen.add(23);
zahlen.add(117);
zahlen.add(3);
zahlen.add(-345);
zahlen.add(23);
zahlen.add(35);
for(int k : zahlen) {
  System.out.println(k * k);
}
```

No children were harmed in the making of these slides.

Autoboxing

```
Map<Short, String> map;
map = new HashMap<Short, String>();
short n = 17;
map.put(n, "Siebzehn");
System.out.println(map.get(17));
```

Vorsicht! Es gibt merkwürdige Effekte und schwer zu findende Fehler.

Tip: Nur *Integer*, *Double*, *Boolean* verwenden.

Nur mit Vorsicht *Short*, *Long*, etc.

Container-Klassen

Klassen die viele Elemente zusammenfassen, sind sehr nützlich.

Die wichtigsten solchen Klassen modellieren Mengen, Listen, Arrays und Abbildungen.

Interfaces in der Java-Bibliothek:

- $Set\langle E\rangle$, Mengen
- $List\langle E\rangle$, Listen und Arrays
- $Map\langle K, E\rangle$, Abbildungen und Wörterbücher

Um auf sie zuzugreifen, müssen wir *java.util.Set* schreiben, oder am Anfang der Datei:

```
import java.util.Set; // importiert Set
import java.util.*; // importiert alles aus java.util
```

Set

Set⟨*E*⟩ ist nur ein Interface.

Implementierungen davon sind z.B. *HashSet*⟨*E*⟩ und *TreeSet*⟨*E*⟩.

Um *Set*⟨*E*⟩ korrekt verwenden zu können, muß die Klasse *E* gewisse Anforderungen erfüllen.

- **boolean** *equals*(*Object o*), ist *o* das gleiche?
- **int** *hashCode*(), berechnet einen Hashwert.
- **int** *compareTo*(*E e*), vergleiche und gib <0, 0, >0 zurück

Jede Klasse erbt *equals* und *hashCode* von *Object*.

compareTo ist im Interface *Comparable*⟨*E*⟩ definiert.

RWTHAACHEN

```
class Person implements Comparable<Person> {
  String nachname, vorname;
  public Person(String v, String n) {
    vorname = v; nachname = n;
  }
  public boolean equals(Object o) {
    if(o == null ||!(o instanceof Person)) {
      return false;
    }
    Person other = (Person)o;
    return nachname.equals(other.nachname) &&
      vorname.equals(other.vorname);
  }
  public int hashCode() {
    return nachname.hashCode() + vorname.hashCode();
  }
```

Wir können nun *HashSet*⟨*Person*⟩ verwenden. Hierzu haben wir *equals* und *hashCode* implementiert.

Für *TreeSet*⟨*Person*⟩ muß *Person* auch noch das Interface *Comparable*⟨*Person*⟩ implementieren.

```
public int compareTo(Person other) {
  int nachnameOrder = nachname.compareTo(other.nachname);
  if(nachnameOrder != 0) {
    return nachnameOrder;
  }
  else {
    return vorname.compareTo(other.vorname);
  }
}
```

Gegeben sei eine Menge von Zahlen $\{3, 5, 10\}$.

Welche Zahlen können wir bilden, indem wir die Zahlen einer Untermenge addieren?

Antwort: $\{0, 3, 5, 8, 10, 13, 15, 18\}$.

```
static Set<Integer> subsetsums(int size[ ]) {
  Set<Integer> sums = new TreeSet<Integer>();
  sums.add(0);
  for(int k : size) {
    Set<Integer> newsums = new TreeSet<Integer>();
    for(int l : sums) {
      newsums.add(k + l);
    }
    sums.addAll(newsums);
  }
  return sums;
}
```

Abbildungen und Wörterbücher

Das Interface $Map\langle K, E\rangle$ modelliert Abbildungen $K \to E$.

So können wir ein Wörterbuch implementieren:

```
Map<String, String> deuita = new HashMap<String, String>();
deuita.put("schneiden", "tagliare");
deuita.put("Butter", "burro");
deuita.put("Igel", "riccio");
String x = deuita.get("Butter");
```

Mit *put* legen wir Werte fest und mit *get* schlagen wir sie nach.

Typische Implementierungen sind *HashMap* und *TreeMap*.

Bei den *Subsetsums* berechneten wir die Summen, aber nicht durch welche Untermenge sie gebildet werden. Hier berechnen wir eine *Map*, welche uns zu jeder Summe *eine* Zahl ihrer Menge gibt.

```
static Map<Integer, Integer> subsetmap(int size[ ]) {
  Map<Integer, Integer> sums = new TreeMap<Integer, Integer>();
  sums.put(0, 0);
  for(int k : size) {
    Map<Integer, Integer> newsums;
    newsums = new TreeMap<Integer, Integer>();
    for(int l : sums.keySet()) {
      newsums.put(k + l, k);
    }
    newsums.putAll(sums);
    sums = newsums;
  }
  return sums;
}
```

List

Es gibt in der Java-Standardbibliothek das Interface $List\langle E\rangle$ (nicht zu verwechseln mit unserer Implementierung).

Folgende Operationen gibt es u.a.:

- **void** $add(E\ x)$, füge x am Ende ein.
- **void** $add(\mathbf{int}\ n, E\ x)$, füge x an Position n ein.
- **void** $set(\mathbf{int}\ n, E\ x)$, ersetzt die $n$te Position durch x.
- $E\ get(\mathbf{int}\ n)$, was ist an Position n?
- **int** $size()$, wieviele Elemente enthält es?

Es wird sowohl ein „wachsendes“ Array als auch eine Liste modelliert.

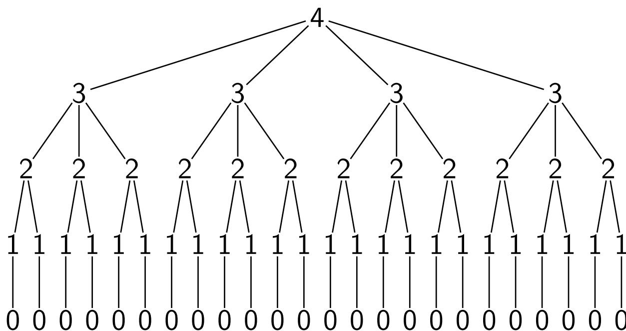
Typische Implementierungen: $ArrayList\langle E\rangle$ und $LinkedList\langle E\rangle$.

Oft ist es bequemer eine *List* als ein normales Array zu verwenden.

Visitor

Erinnern wir uns an die Klasse *Tree*.

Sie implementiert Bäume, deren Knoten Zahlen enthalten.



Öffentlichen Methoden *getRoot*(), *getChildren*(), *addChild*(*Tree t*) und Konstruktor **new** *Tree*(**int** *root*).

Visitor

Wie können wir das Gesamtgewicht eines Baumes berechnen?

Wir machten dies früher durch Schreiben einer geeigneten Methode.

Nachteil: „Enge Kopplung“.

So müssen wir oft die Klasse selbst verändern, was nicht gut ist.

Alternative: Nur die Grundoperationen *getRoot*() und *getChildren*() verwenden.

Weitere Möglichkeit: Manchmal hilft es immens einen Visitor oder Iterator zu verwenden.

Visitor

Wir erweitern die Klasse *Tree* so, daß sie einen *TreeVisitor* „akzeptiert“.

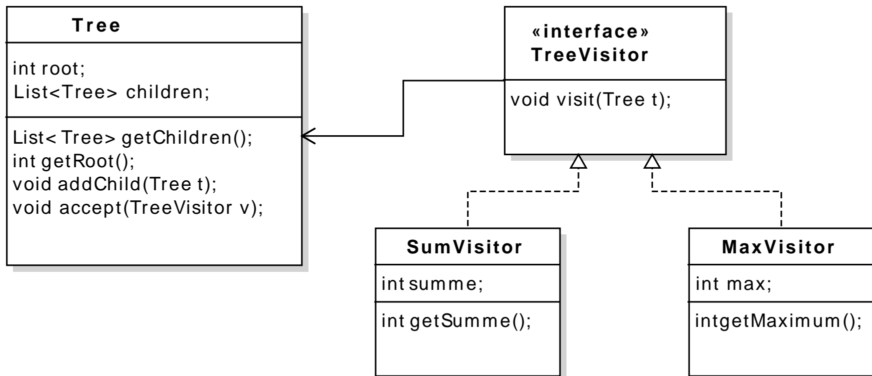
```
interface TreeVisitor {
   public void visit(Tree t);
}
```

Die Klasse *Tree* ermöglicht es einem *TreeVisitor* durch einen einfachen Aufruf alle Knoten des Baumes „zu besuchen“.

```
Tree tree = new Tree(24);
...
TreeVisitor visitor = new MyTreeVisitor();
tree.accept(visitor);
```

Hierin ist *MyTreeVisitor* eine konkrete Klasse, die *TreeVisitor* implementiert.

Klassendiagramm



Visitor

So implementieren wir die Methode *accept*(*Tree t*) in der Klasse *Tree*:

```
public void accept(TreeVisitor v) {
  v.visit(this);
  for(Tree child : children) {
    child.accept(v);
  }
}
```

Die Gesamtsumme aller Knoten können wir jetzt so berechnen:

```
public static void main(String args[ ]) {
    Tree t1 = new Tree(1);
    Tree t2 = new Tree(2);
    Tree t3 = new Tree(3);
    Tree t4 = new Tree(4);
    Tree t5 = new Tree(5);
    Tree t6 = new Tree(6);
    t1.addChild(t2);
    t1.addChild(t3);
    t3.addChild(t4);
    t3.addChild(t5);
    t3.addChild(t6);
    SumVisitor visitor = new SumVisitor();
    t1.accept(visitor);
    System.out.println(visitor.summe());
}
```

Hierfür müssen wir nur den geeigneten Visitor implementieren:

```
class SumVisitor implements TreeVisitor {
    private int sum = 0;
    public void visit(Tree t) {
        sum = sum + t.getRoot();
    }
    public int summe() {
        return sum;
    }
}
```

Dies ist eine Art neue Funktionalität einer Klasse hinzuzufügen.

Es ist jetzt leicht, weitere Visitors zu schreiben.

Zum Beispiel, um das Maximum zu berechnen:

```
class MaxVisitor implements TreeVisitor {
  private int max = −1;
  public void visit(Tree t) {
    int value = t.getRoot();
    if(value > max) {
      max = value;
    }
  }
  public int maximum() {
    return max;
  }
}
```

Überblick

4. Objektorientiertes Design
   - Interfaces, Container, Iterator, Visitor
   - Composition, Decorator
   - Schichtenmodell und Filter-Pipelines

“Composition over Inheritance”

Eine Möglichkeit, daß eine Klasse *B* die Funktionalität von *A* übernimmt, ist *Vererbung*:

```
class B extends A {
    ...
}
```

Eine andere Möglichkeit ist *Komposition*.
Klasse *B* *enthält* ein Object *A*:

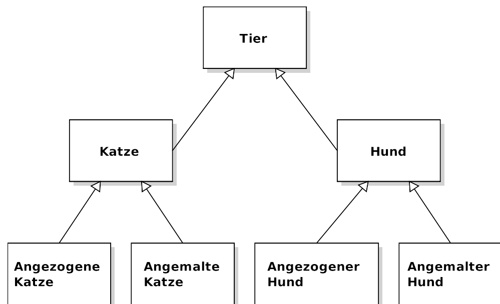
```
class B {
    A a;
    ...
}
```

Dadurch kann *B* im Prinzip auch alles, was *A* kann.

Komposition kann flexibler sein und ist oft besser als Vererbung.

Vererbung

Betrachten wir folgendes Klassendiagramm:



Wenn wir mehr Tiere hinzufügen, gibt es eine Unzahl von spezialisierten Klassen.

Wir können keinen „angezogenen, angemalten Hund“ durch Vererbung erzeugen (keine Mehrfachvererbung).

RWTH AACHEN

```
interface Tier {
   public double getGewicht();
   public String zeichnung();
}
```

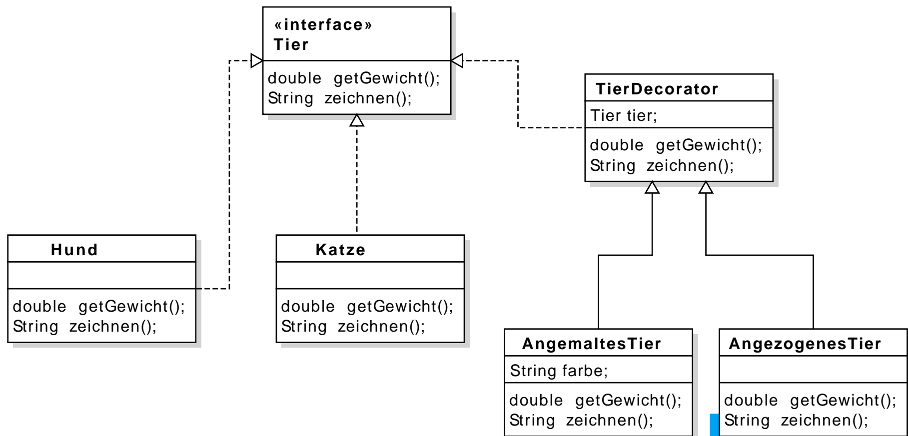
Einige konkrete Tiere:

```
class Katze implements Tier {
  public double getGewicht() {return 2.5; }
  public String zeichnung() {return "Katze"; }
}
```

```
class Hund implements Tier {
  public double getGewicht() {return 3.5; }
  public String zeichnung() {return "Hund"; }
}
```

Composition: Decorator

Eine mögliche Lösung: Wir verwenden einen *Decorator*, der dynamisch Funktionalität zu einer Klasse hinzufügen kann.



```
interface Tier {
  public double getGewicht();
  public String zeichnung();
}
```

Die abstrakte Klasse *TierDecorator* ist auch ein *Tier*:

```
abstract class TierDecorator implements Tier {
 private Tier tier;
 public TierDecorator(Tier tier) {
   this.tier = tier;
 }
 public double getGewicht() {
  return tier.getGewicht();
 }
 public String zeichnung() {
  return tier.zeichnung();
 }
}
```

Wir können jetzt einige *TierDecorator*s implementieren:

```
class AngezogenesTier extends TierDecorator {
 public AngezogenesTier(Tier tier) {
  super(tier);
 }
 public double getGewicht() {
  return super.getGewicht() + 0.5;
 }
 public String zeichnung() {
  return super.zeichnung() + " mit Mantel";
 }
}
```

Ein angezogenes Tier ist ein Tier mit einem Mantel.
Der Mantel wiegt ein bißchen.

Ein „angemaltes Tier“ hat eine Farbe, die wir dem Konstruktor mitteilen müssen:

```
class AngemaltesTier extends TierDecorator {
  private String farbe;
  public AngemaltesTier(Tier tier, String farbe) {
    super(tier);
    this.farbe = farbe;
  }
  public String zeichnung() {
    return super.zeichnung() + "  " + farbe + " angemalt";
  }
}
```

Wir können jetzt beliebige Dekorationen an einem Tier „anbringen“:

```
public static void main(String args[]) {
    Tier hachiko = new Hund();
    hachiko = new AngezogenesTier(hachiko);
    hachiko = new AngemaltesTier(hachiko, "weiss");
    Tier garfield = new Katze();
    garfield = new AngemaltesTier(garfield, "gelb-schwarz");
    System.out.println(hachiko.zeichnung());
    System.out.println(garfield.zeichnung());
}
```

Ausgabe:

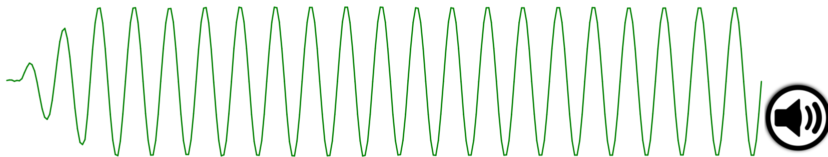
```
Hund mit Mantel blau angemalt
Katze gelb-schwarz angemalt
```

Überblick

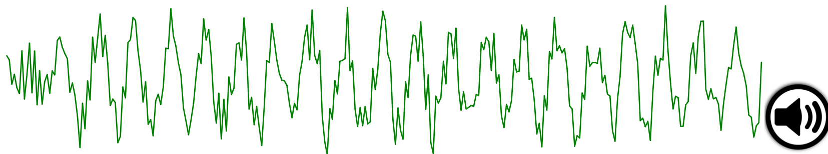
4. Objektorientiertes Design
   - Interfaces, Container, Iterator, Visitor
   - Composition, Decorator
   - Schichtenmodell und Filter-Pipelines

Beispiel: Morsecode dekodieren

Morsecode, sauberer Ton:



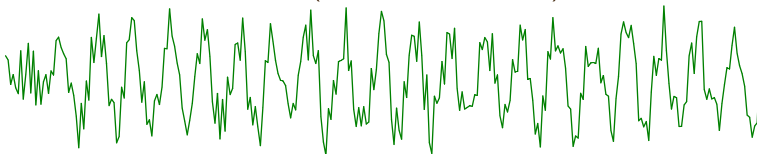
Etwas verrauscht:



Wie können wir den Klartext erhalten?

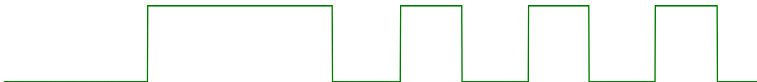
Die Audiodaten werden durch mehrere Filter geschickt.

- $\textit{buffer} = \textit{filter.readFile}(\texttt{"withnoise.raw"});$



- $\textit{buffer} = \textit{filter.average}(200, \textit{buffer})$

- $\textit{buffer} = \textit{filter.threshold}(\textit{buffer})$



- *buffer = filter.threshold(buffer);*

- *buffer = filter.cummulate(buffer);*
  [-1234, 968, -1085, 2951, -44005, 2954, -1076, 977, -1051, 976, -1052, 977, -43977, 2954, -1076, 977, -1077, 2955, -1075, 976, -43978, 2954, -1076, 976, -1053, 976, -43951, 976, -43952, 977, -1051,...]

- *String dotdashes = filter.symbolize(buffer);*
  `.- -... -.-. -.. . ..-. --. .... .. .--- -.- .-.. -- -. --- .--. --.- .-. ... - ..- ...- .-- -..- -.-- --..`

- *List⟨String⟩ tokens = filter.tokenize(dotdashs);*
  `[.-, -..., -.-., -.., ., ..-., --., ...., .., .---, -.-, .-.., --, -., ---, .--., --.-, .-., ..., -, ..-, ...-, .--, -..-, -.--, --..]`

- *String cleartext = filter.decode(tokens);*
  `ABCDEFGHIJKLMNOPQRSTUVWXYZ`

RWTHAACHEN

```
double[ ] average(int k, double a[ ]) {
  int n = a.length;
  assert n > k;
  double sum[ ] = new double[n];
  sum[0] = a[0];
  for (int i = 1; i < n; i++) {
    sum[i] = sum[i - 1] + Math.abs(a[i]);
  }
  double mittel[ ] = new double[n];
  for (int i = k; i < n; i++) {
    mittel[i] = (sum[i] - sum[i - k])/k;
  }
  for (int i = 0; i < k; i++) {
    mittel[i] = mittel[k];
  }
  return mittel;
}
```

```
double[ ] threshold(double a[ ], double mean[ ]) {
  assert mean.length == 2;
  int n = a.length;
  double result[ ] = new double[n];
  int diff = 0;
  for (int i = 0; i < n; i++) {
    if (Math.abs(mean[0] − a[i]) < Math.abs(mean[1] − a[i])) {
      diff = diff − 1;
    } else {
      diff = diff + 1;
    }
    final int d = 10;
    if (diff > d) {diff = d; result[i] = 1; }
    if (diff < −d) {diff = −d; result[i] = 0; }
  }
  return result;
}
```

```
List<Integer> cummulate(double a[ ]) {
  int n = a.length, count = 0;
  List<Integer> result = new ArrayList<Integer>();
  for (int i = 1; i < n; i++) {
    if((a[i] > 0) != (a[i − 1] > 0)) {
      result.add(count);
      count = 0;
    }
    if(a[i] > 0) {
      count = count + 1;
    }
    else {
      count = count − 1;
    }
  }
  result.add(count);
  return result;
}
```

```
String symbolize(List<Integer> len) {
  String s = "";
  double pulse[ ] = kMeans.getCenters(2, signFilter(+1, len));
  double pause[ ] = kMeans.getCenters(2, signFilter(−1, len));
  for(int dur : len) {
    if(dur < 0) {
      if(Math.abs(pause[0] − dur) < Math.abs(pause[1] − dur)) {
        s = s + "  ";
      }
    }
    else {
      if(Math.abs(pulse[0] − dur) < Math.abs(pulse[1] − dur)) {
        s = s + "."; } else {s = s + "-"; }
    }
  }
  return s;
}
```

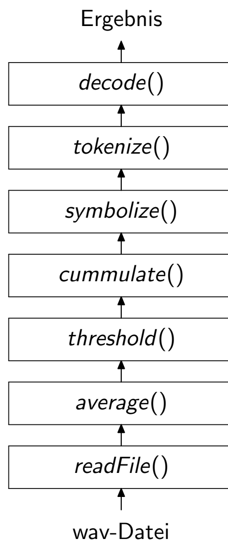
```
double[ ] signFilter(int sign, List<Integer> list) {
  List<Integer> result = new ArrayList<Integer>();
  for (int n : list) {
    if ((sign > 0) == (n > 0)) {
      result.add(n);
    }
  }
  double r[ ] = new double[result.size()];
  for(int i = 0; i < r.length; i++) {
    r[i] = result.get(i);
  }
  return r;
}
```

```
List<String> tokenize(String dotdashs) {
  List<String> tokens = new ArrayList<String>();
  String z = "";
  for (int i = 0; i < dotdashs.length(); i++) {
    if (dotdashs.charAt(i) == ' ') {
      tokens.add(z);
      z = "";
    } else {
      z = z + dotdashs.charAt(i);
    }
  }
  tokens.add(z);
  return tokens;
}
```

```
String decode(List<String> tokens) {
  String result = "";
  for (String morseToken : tokens) {
    result += translateMorseCodeToChar(morseToken);
  }
  return result;
}
```

```
char translateMorseCodeToChar(String code) {
  Character c = morseTable.get(code);
  if(c == null) {
    return '?';
  }
  else {
    return c;
  }
}
```

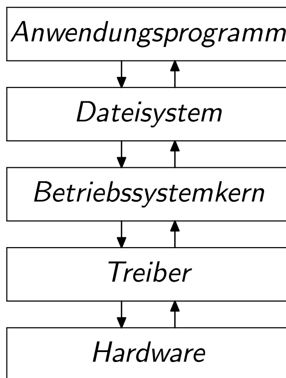
Pipelines, um schrittweise zum Ergebnis zu kommen.



Programmieren in Schichten

Pipeline: Die Daten laufen in einer Richtung.

Verallgemeinerung: Viele Schichten, die immer mehr abstrahieren.



RWTH AACHEN

Neues Kapitel

1. Einführung in die objektorientierte Programmierung
2. Rekursion
3. Fehler finden und vermeiden
4. Objektorientiertes Design
5. Effiziente Programme
6. Nebenläufigkeit
7. Laufzeitfehler
8. Refactoring
9. Persistenz
10. Eingebettete Systeme
11. Funktionale Programmierung
12. Logische Programmierung

Überblick

5. Effiziente Programme
   - Schnelle und langsame Programme
   - Speicherplatz

Wann ist ein Programm schnell?

Die Geschwindigkeit von Programmen ist schwer zu vergleichen.

- Welcher Computer?
- Welcher Compiler?
- Größe des Speichers?
- Cache-Speicher?
- Welche anderen Programme laufen?
- etcetera, etcetera.

Die Geschwindigkeit eines Programms kann auch vom Wetter abhängen.

Schnelle Programme

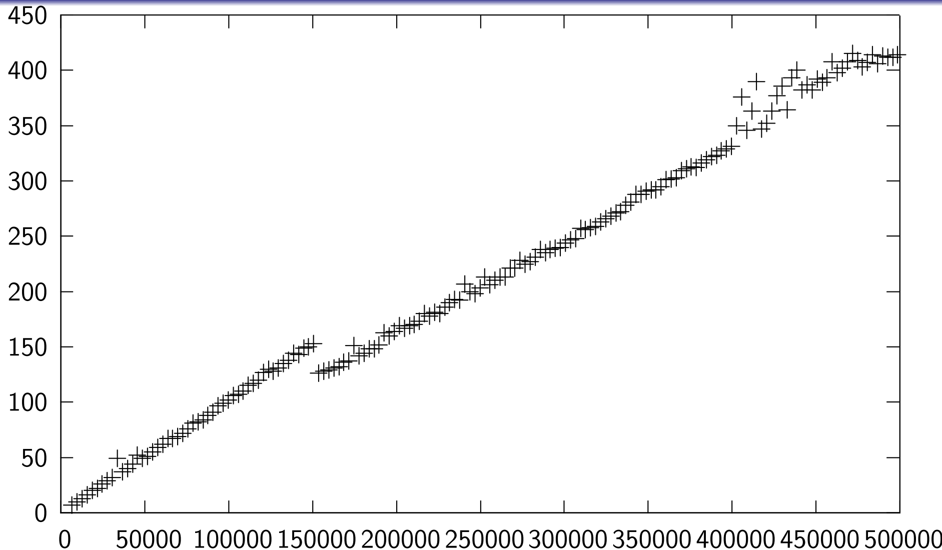
Wir können Programme durch „Tuning“ verbessern.

Dies lohnt sich aber nur bei den Teilen, welche die meiste Zeit verbrauchen: „innere Schleifen“

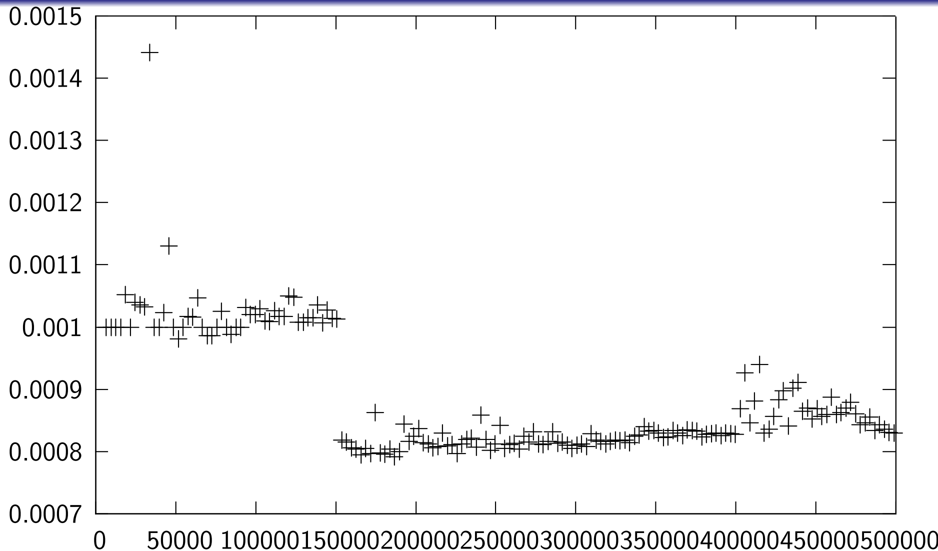
Und:

> *Premature optimization is the root of all evil.*
> *Donald Knuth*

Ein überlegener Algorithmus kann die bessere Wahl sein.

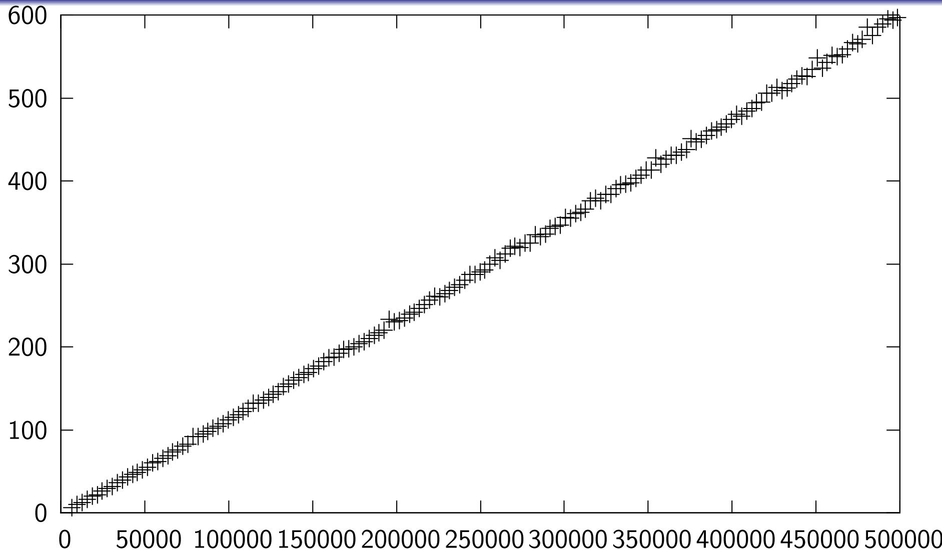


Zehnmalige Wiederholung von Quicksort.

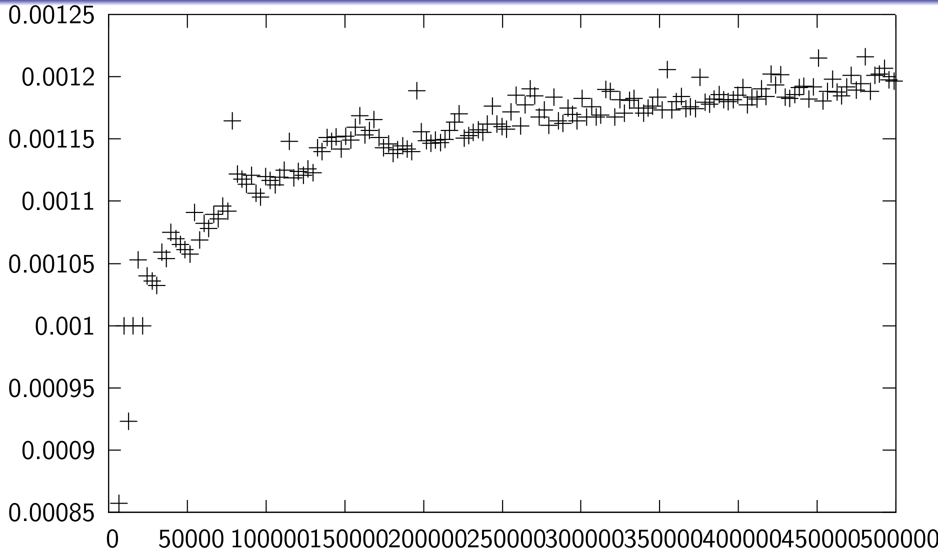


Laufzeit *pro* Datenelement.

RWTHAACHEN

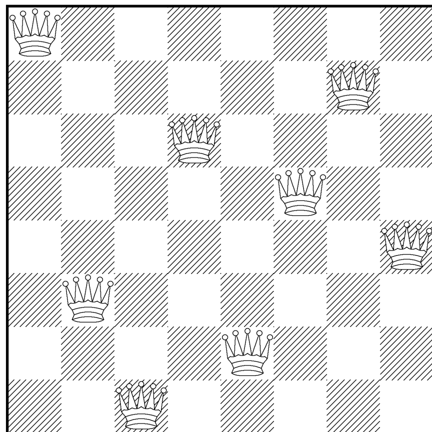


Zehnmalige Wiederholung von Heapsort.



Laufzeit *pro* Datenelement.

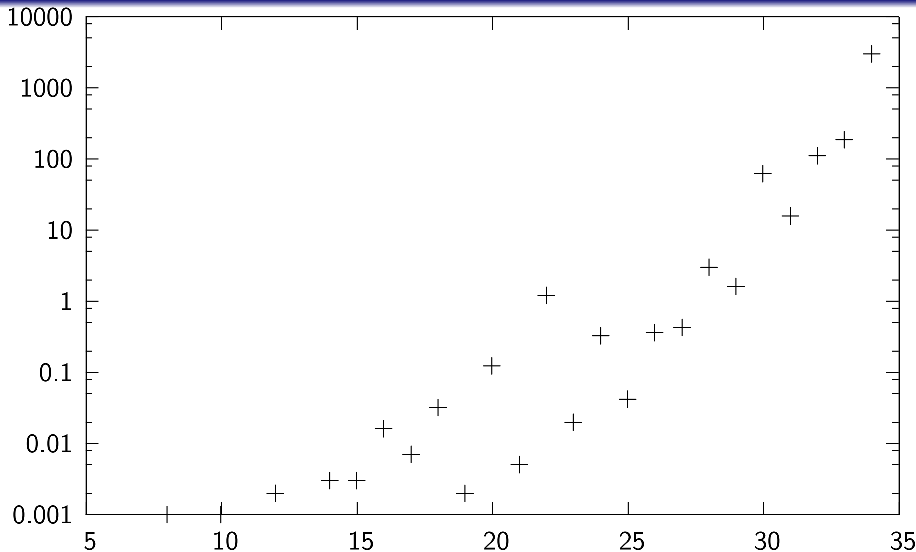
Das n-Damenproblem



Wie können wir das n-Damenproblem *schneller* lösen?

Dies war unser Programm:

```
void setzeDamenAbZeile(int y) {
  if(y >= n) {
    geloest = true;
    return;
  }
  for(int x = 0; x < n; x++) {
    if(safePosition(y, x)) {
      brett[y][x] = true;
      setzeDamenAbZeile(y + 1);
      if(geloest) {
        return;
      }
      brett[y][x] = false;
    }
  }
}
```



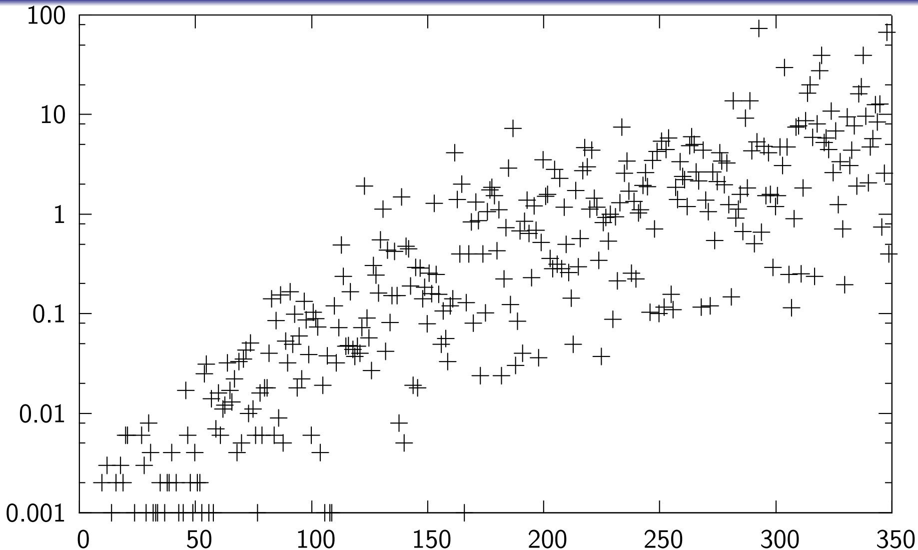
Laufzeit des n-Damenproblems.

```
void setzeDamenAbZeile(int y, int totallives) {
  lives = totallives;
  if(y >= n) {
    geloest = true;
    return;
  }
  lives = lives - 1;
  if(lives < 0) {return; }
  int offset = generator.nextInt(n);
  for(int k = 0; k < n; k++) {
    int x = (k + offset)%n;
    if(safePosition(y, x)) {
      brett[y][x] = true;
      setzeDamenAbZeile(y + 1, lives);
      if(geloest) {return; }
      brett[y][x] = false;
    }
  }
}
```

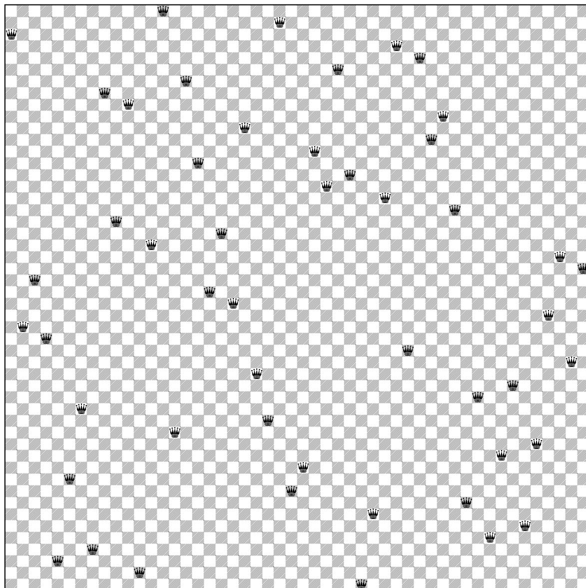
Zwei Innovationen:

- Nach gewisser Zeit aufgeben und neu anfangen.
- Die Damen in zufälliger Reihenfolge setzen.

```
void setzeDamenAbZeile(int y) {
  int totallives = 1000;
  while(!geloest) {
    for(int yy = y; yy < n; yy++) {
      for(int x = 0; x < n; x++) {
        brett[yy][x] = false;
      }
    }
    setzeDamenAbZeile(y, totallives);
    totallives + = 100;
  }
}
```



Laufzeit des n-Damenproblems mit Wiederstart.



RWTH AACHEN

Überblick

5. Effiziente Programme
   - Schnelle und langsame Programme
   - Speicherplatz

Speichersystem

Java verwendet ein sehr bequemes und zuverlässiges Speichersystem.

Mit $x = \textbf{new}\ \mathit{Person}(\ldots)$ wird Speicher für ein neues Objekt alloziert.

Durch eine *garbage collection* werden regelmäßig nicht mehr benötigte Objekte wieder freigegeben.

Dies passiert transparent.

Aber: Jedes **new** kostet Zeit und jedes Objekt verschwendet ein bißchen Speicherplatz.

Wenn dies eine Rolle spielt, dann sind primitive Datentypen und Arrays die schnellste und sparsamste Lösung.

Initialisierung eines Arrays:

```
static int[] arrayTest(int n) {
  int a[] = new int[n];
  for(int i = 0; i < n; i++) {
    a[i] = i;
  }
  return a;
}
```

Dies dauert mit $n = 100.000.000$ z.B. 0.16 Sekunden.

Jetzt verwenden wir eine *ArrayList⟨Integer⟩*.

```
static List<Integer> listTest(int n) {
    List<Integer> l = new ArrayList<Integer>();
    for(int i = 0; i < n; i++) {
        l.add(i);
    }
    return l;
}
```

Laufzeit jetzt 14.2 Sekunden.

Neues Kapitel

1. Einführung in die objektorientierte Programmierung
2. Rekursion
3. Fehler finden und vermeiden
4. Objektorientiertes Design
5. Effiziente Programme
6. **Nebenläufigkeit**
7. Laufzeitfehler
8. Refactoring
9. Persistenz
10. Eingebettete Systeme
11. Funktionale Programmierung
12. Logische Programmierung

Überblick

6. Nebenläufigkeit
   - Threads
   - Synchronisation

Computer haben oft mehrere Prozessoren.

Tendenz ist zunehmend.

Aufgenommen im Deutschen Museum von Clemens Pfeiffer.

Fazit: Paralleles Programmieren immer wichtiger.

Ein Programm kann mehrere laufende *threads* besitzen.

Diese laufen gleichzeitig oder scheinbar gleichzeitig.

So können wir einen *thread* starten:

```
Thread thread = new Thread(somethingRunnable);
thread.start();
```

Hierbei ist *somethingRunnable* ein Objekt, das das Interface *Runnable* implementiert.

Hierfür muß es eine Methode **public void** *run*() besitzen.

Beispiel

```
class ZahlenZeiger implements Runnable {
   private int id;
   private volatile boolean finished = false;
   public ZahlenZeiger(int id) {
      this.id = id;
   }
   public void run() {
      for(int i = 0; i < 100000; i++) {
         System.out.println(id + ":  " + i);
      }
      finished = true;
   }
   public boolean finished() {
      return finished;
   }
}
```

Volatile variables

Wir können eine Variable **volatile** deklarieren.

Damit sagen wir, daß auf diese Variable von mehreren Threads zugegriffen werden kann.

Folgendes Programmstück kann vom Compiler optimiert werden, da sich i niemals ändert.

```
int i = 1;
while(i == 1) {}
```

Die Schleife kann aber beendet werden, wenn i von *einem anderen Thread* verändert wird.

Ein weiteres Problem sind Caches, die den Speicher von verschiedenen Prozessoren verschieden aussehen lassen.

Lösung: **volatile int** $i = 1;$

"Busy waiting" (oder "polling"):

```
static void test1() {
  List<ZahlenZeiger> zeigerList = new ArrayList<ZahlenZeiger>();
  for(int i = 1; i <= 10; i++) {
    ZahlenZeiger z = new ZahlenZeiger(i);
    zeigerList.add(z);
    Thread thread = new Thread(z);
    thread.start();
  }
  boolean stillRunning = true;
  while(stillRunning) {
    stillRunning = false;
    for(ZahlenZeiger z : zeigerList) {
      if(!z.finished()) {stillRunning = true; }
    }
  }
}
```

Warten durch *join*():

```
static void test2() {
  List<Thread> threads = new ArrayList<Thread>();
  for(int i = 1; i <= 10; i++) {
    ZahlenZeiger z = new ZahlenZeiger(i);
    Thread t = new Thread(z);
    t.start();
    threads.add(t);
  }
  try {
    for(Thread t : threads) {t.join();}
  }
  catch(InterruptedException e) {System.out.println(e);}
}
```

join() wartet, bis der Thread endet.

Unterklasse von *Thread* statt *Runnable*:

```
class QueensSolver extends Thread {
  private int n; // Groesse des Bretts
  private volatile Brett brett;
  private volatile boolean finished = false;
  public QueensSolver(int n) {this.n = n;}
  public void run() {
    brett = new RandomBrett(n);
    brett.setzeDamenAbZeile(0);
    finished = true;
  }
  public void printSolution() {brett.print();}
  public boolean finished() {return finished;}
  public void finish() {brett.geloest = true;}
}
```

```
static void test3() {
  final int n = 250;
  List<QueensSolver> solvers = new ArrayList<QueensSolver>();
  for(int i = 0; i < 4; i++) {
    QueensSolver solver = new QueensSolver(n);
    solver.start(); solvers.add(solver);
  }
  while(true) { // Polling!
    for(QueensSolver solver : solvers) {
      if(solver.finished()) {
        solver.printSolution();
        for(QueensSolver s : solvers) {s.finish(); }
        return;
      }
    }
  }
}
```

Überblick

6 Nebenläufigkeit

- Threads
- Synchronisation

Es ist schwer, Daten konsistent zu halten, wenn mehrere Threads auf sie zugreifen:

```
class WortPaarVersuch implements Runnable {
  private String deutsch;
  private String englisch;
  public void run() {
    while(true) {
      setNames("Hallo", "hello");
      setNames("Hund", "dog");
      setNames("Katze", "cat");
    }
  }
  public void setNames(String d, String e) {
    deutsch = d; englisch = e;
  }
  public String getNames() {
    return deutsch + "=" + englisch;
  }
}
```

Deklarieren wir Methoden **synchronized**, dann können keine von ihnen zeitlich überlappt ablaufen.

```
class WortPaar implements Runnable {
  private volatile String deutsch;
  private volatile String englisch;
  public void run() {
    while(true) {
      setNames("Hallo", "hello");
      setNames("Hund", "dog");
      setNames("Katze", "cat");
    }
  }
  public synchronized void setNames(String d, String e) {
    deutsch = d; englisch = e;
  }
  public synchronized String getNames() {
    return deutsch + "=" + englisch;
  }
}
```

Neues Kapitel

1. Einführung in die objektorientierte Programmierung
2. Rekursion
3. Fehler finden und vermeiden
4. Objektorientiertes Design
5. Effiziente Programme
6. Nebenläufigkeit
7. Laufzeitfehler
8. Refactoring
9. Persistenz
10. Eingebettete Systeme
11. Funktionale Programmierung
12. Logische Programmierung

Fehlerbehandlung

Unter einem Laufzeitfehler verstehen wir eine aussergewöhnliche Situtation, die normalerweise nicht auftreten sollte.

Beispiel:

Wir öffnen eine Datei, die eigentlich existieren muß.

```
DataInputStream stream = null;
try {
  stream = new FileInputStream("dateiname");
}
catch(FileNotFoundException exception) {
  ... // Fehlerbehandlung, z.B. Abbruch mit Fehlermeldung
}
... // Lese aus der Datei etc.
```

Gegenbeispiel:

Wie lesen Instruktionen aus einer optionalen Datei.

Falls diese nicht existiert, liegt kein Fehler vor.

Beispiel: Menschen, die heiraten können.

```
class Mensch {
  private int alter;
  private String name;
  private Mensch partner;
  public Mensch(String name, int alter) {
    this.alter = alter;
    this.name = name;
    this.partner = null;
  }
  public int getAlter() {return alter;}
  public String getName() {return name;}
  public Mensch getPartner() {return partner;}
```

Mit **throw** können wir ein Objekt vom Typ *Exception* „werfen“, welches von jemanden mit **catch** gefangen werden muß.

Annahme: Heiraten ist nur erlaubt, wenn das Durchschnittsalter mindestens 18 beträgt.

```
public void heiraten(Mensch partner) throws ZuJungException {
  if(this.alter + partner.alter < 36) {
    throw new ZuJungException();
  }
  this.partner = partner;
}
```

Für das geworfene Objekt erstellen wir eine eigene Klasse:

```
class ZuJungException extends Exception {}
```

RWTHAACHEN

So können wir die Methode *heiraten* jetzt aufrufen:

```
public static void main(String args[ ]) {
  Mensch karl = new Mensch("Karl Ranseier", 17);
  Mensch sue = new Mensch("Sue Winter", 17);
  try {
    karl.heiraten(sue);
  } catch (ZuJungException e) {
    System.out.println("Zu jung. Sie haben nicht geheiratet.");
  }
}
```

Beispiel

Wir berechnen die größte Zahl in einem Array.

```
static int maximum(int a[ ]) throws EmptyArrayException {
  int n = a.length;
  if(n == 0) {
    throw new EmptyArrayException();
  }
  int max = a[0];
  for(int i = 0; i < n; i++) {
    if(max < a[i]) {
      max = a[i];
    }
  }
  return max;
}
```

Neues Kapitel

1. Einführung in die objektorientierte Programmierung
2. Rekursion
3. Fehler finden und vermeiden
4. Objektorientiertes Design
5. Effiziente Programme
6. Nebenläufigkeit
7. Laufzeitfehler
8. Refactoring
9. Persistenz
10. Eingebettete Systeme
11. Funktionale Programmierung
12. Logische Programmierung

Refactoring

Ein (komplexes) Programm muß sich während seiner Lebenszeit Erweiterungen und Veränderungen unterwerfen.

Oft zeigt sich bei einer Erweiterung, daß die derzeitige Struktur ungeeignet ist.

Bestes Vorgehen:

1. Die Struktur des Programms verändern.
2. Die neue Funktionalität hinzufügen.

Den ersten Schritt nennen wir *Refactoring*.

Die Struktur des Programms wird verbessert, ohne Neues hinzuzufügen.

Typische Refactoringschritte (es gibt viel mehr):

- Doppelten Code durch einfachen ersetzen.
- Zu lange Methoden durch kürzere ersetzen.
- Lokale Variablen abschotten (getter und setter).
- Funktionalität verallgemeinern für besseres Wiederbenutzen.
- Neue Unterklassen einführen.
- Unterklassen zusammenfassen.
- Variablen oder Methoden in Ober- oder Unterklassen verschieben.
- Variablen oder Methoden in andere Klassen verschieben.
- ...und sogar einfache Umbenennungen in sinnvollere Namen.

Neues Kapitel

1. Einführung in die objektorientierte Programmierung
2. Rekursion
3. Fehler finden und vermeiden
4. Objektorientiertes Design
5. Effiziente Programme
6. Nebenläufigkeit
7. Laufzeitfehler
8. Refactoring
9. **Persistenz**
10. Eingebettete Systeme
11. Funktionale Programmierung
12. Logische Programmierung

Wie können wir Daten dauerhaft speichern?

Häufigste Methoden:

- In einer Datei speichern.
- In einer Datenbank speichern.

Lesen aus einer Datei:

```
double readDouble(String name) {
  FileInputStream is = null;
  DataInputStream dataStream = null;
  double result;
  try {
    is = new FileInputStream(name);
    dataStream = new DataInputStream(is);
    result = dataStream.readDouble();
  }
  catch (Exception e) {e.printStackTrace(); }
  return result;
}
```

Serializable

Implementiert eine Klasse das Interface *Serializable*, dann kann ihr Inhalt direkt in eine Datei geschrieben und aus ihr gelesen werden.

```
class Person implements Serializable {
  private String vorname;
  private String nachname;
  public Person(String vorname, String nachname) {
    this.vorname = vorname;
    this.nachname = nachname;
  }
  public String toString() {
    return vorname + " " + nachname;
  }
}
```

So schreiben wir eine *Person* in eine Datei:

```
Person karl = new Person("Karl", "Ranseier");
try {
    FileOutputStream stream = new FileOutputStream("karl");
    ObjectOutputStream objstream =
        new ObjectOutputStream(stream);
    objstream.writeObject(karl);
    stream.close();
}
catch(IOException e) {
    System.out.println("Oje:  " + e);
}
```

So lesen wir eine *Person*:

```
Person unbekannt = null;
try {
  FileInputStream stream = new FileInputStream("karl");
  ObjectInputStream objstream =
    new ObjectInputStream(stream);
  unbekannt = (Person) objstream.readObject();
  stream.close();
}
catch(ClassNotFoundException e) {
  System.out.println("Falsche Klasse gelesen: " + e);
}
catch(IOException e) {
  System.out.println("Oje: " + e);
}
System.out.println(unbekannt);
```

Neues Kapitel

1. Einführung in die objektorientierte Programmierung
2. Rekursion
3. Fehler finden und vermeiden
4. Objektorientiertes Design
5. Effiziente Programme
6. Nebenläufigkeit
7. Laufzeitfehler
8. Refactoring
9. Persistenz
10. **Eingebettete Systeme**
11. Funktionale Programmierung
12. Logische Programmierung

Neues Kapitel

1. Einführung in die objektorientierte Programmierung
2. Rekursion
3. Fehler finden und vermeiden
4. Objektorientiertes Design
5. Effiziente Programme
6. Nebenläufigkeit
7. Laufzeitfehler
8. Refactoring
9. Persistenz
10. Eingebettete Systeme
11. Funktionale Programmierung
12. Logische Programmierung

Überblick

11 Funktionale Programmierung

- Allgemeines
- Haskell
- MapReduce

Funktionen

Es gibt verschiedene Notationen für Funktionen:

- $f : \mathbf{N} \to \mathbf{N},\ \ n \mapsto n \cdot n$
- $f : \mathbf{N} \to \mathbf{N},\ \ f(n) = n \cdot n$
- $\lambda n.n \cdot n$ (bedeutet $n \mapsto n \cdot n$)

Die ersten beiden Definitionen geben den Definitions- und Wertebereich an (die „Signatur“ der Funktion).

Sie geben der Funktion auch den Namen f.

Die dritte Definition entstammt dem λ-Kalkül. Sie gibt der Funktion keinen Namen.

Referentielle Integrität

```
class X {
  int n = 0;
  int f(int k) {
    return k * k;
  }
  int g(int k) {
    n = n + 1;
    return n * k;
  }
}
```

Was ergeben $f(5)$ und $g(5)$?

Referentielle Integrität: Das Ergebnis eines Funktionsaufrufs hängt nur von den Parametern ab.

Java hat sie nicht.

Referentielle Integrität

Die referentielle Integrität wird durch das Vorhandensein von Variablen verhindert.

Auch Ein- und Ausgaben verhindern sie:

Die Funktion **double** *readDouble*() liest in Java eine Zahl aus einer Datei.

Diese ist *nicht* immer die gleiche.

Ein weiteres Gegenbeispiel ist eine Methode, die uns eine Zufallszahl gibt.

Referentielle Integrität

Referentielle Integrität hat den Vorteil, daß die Reihenfolge, in der Parameter ausgewertet werden, keine Rolle spielt (Parallelität).

Beispiel:

- $f : n \mapsto n \cdot n$
- $g : (n, m) \mapsto n + m$

Was ist $g(f(3), g(f(2), f(1)))$?

$$\begin{aligned} g(f(3), g(f(2), f(1))) &= g(9, g(f(2), f(1))) \\ &= g(9, g(4, f(1))) \\ &= g(9, g(4, 1)) \\ &= g(9, 5) \\ &= 14 \end{aligned}$$

Andere Auswertungsreihenfolgen ergeben auch 14.

Gilt dies auch für Java-Programme?

```
class Reihenfolge {
  static int n = 0;
  static int f(int x) {
    n = n + x;
    return n;
  }
  static int g(int x, int y) {
    return x - y;
  }
  public static void main(String args[ ]) {
    int result = g(f(20), f(50));
    System.out.println(result);
  }
}
```

Was ist die Ausgabe?

Aus der *Java Language Specification*:

Siehe hier: docs.oracle.com

> *The argument expressions, if any, are evaluated in order, from left to right. If the evaluation of any argument expression completes abruptly, then no part of any argument expression to its right appears to have been evaluated, and the method invocation completes abruptly for the same reason. The result of evaluating the j'th argument expression is the j'th argument value, for $1 \leq j \leq n$. Evaluation then continues, using the argument values, as described below.*

Die Parameter werden *von links nach rechts evaluiert.*

Ein Programm, das sich darauf verläßt, ist nicht ideal.

Die Programmiersprache C gibt keine Garantie für die Reihenfolge, in der Parameter einer Funktion evaluiert werden.

```
#include <stdio.h>
int n = 0;
int f(int x) {
   n = n + x;
   return n;
}
int g(int x, int y) {
   return x - y;
}
void main() {
   printf("%d\n", g(f(20), f(50)));
}
```

Ergebnis unterscheidet sich für SUN Workstations und Intel PCs.

Call by Value

Diese Art der Evaluierung nennen wir *Call by Value*:

$$
\begin{aligned}
g(f(3), g(f(2), f(1))) &= g(9, g(f(2), f(1))) \\
&= g(9, g(4, f(1))) \\
&= g(9, g(4, 1)) \\
&= g(9, 5) \\
&= 14
\end{aligned}
$$

Bevor eine Funktion evaluiert wird, werden rekursiv alle ihre Parameter evaluiert.

Normale Evaluierung: Call by Value mit Reihenfolge von links nach rechts.

Call by Value

Call by Value kann ineffizient sein:

```
int h(int x, int y) {
   if(x < 0) {
      return 0;
   }
   else {
      return x * y;
   }
}
...
h(−3, sehrTeuer(25));
```

Um $h(-3, x)$ zu berechnen, ist der Wert von x irrelevant.

Trotztdem wird $sehrTeuer(25)$ aufgerufen.

Wir können eine Funktion auch so weit wie möglich außen evaluieren:

$$
\begin{aligned}
g(f(3), g(f(2), f(1))) &= f(3) + g(f(2), f(1))) \\
&= (3 * 3) + g(f(2), f(1)) \\
&= 9 + g(f(2), f(1)) \\
&= 9 + (f(2) + f(1)) \\
&= 9 + ((2 * 2) + f(1)) \\
&= 9 + (4 + f(1)) \\
&= 9 + (4 + (1 * 1)) \\
&= 9 + (4 + 1) \\
&= 9 + 5 \\
&= 14
\end{aligned}
$$

Dies nennen wir *Call by Name*.

Hat dies irgendeinen Vorteil???

Call by Name

„Evaluierung“ von h mittels Call by Name:

$h(-3, sehrTeuer(25))$
$\textbf{if}(-3 < 0)\ \{\textbf{return}\ 0;\}\ \textbf{else}\ \{\textbf{return}\ (-3) * sehrTeuer(25)\};$
$\textbf{if}(true)\ \{\textbf{return}\ 0;\}\ \textbf{else}\ \{\textbf{return}\ (-3) * sehrTeuer(25);\}$
$\textbf{return}\ 0;$
0

Es stellt sich heraus, daß $sehrTeuer(25)$ *gar nicht* evaluiert wird.

Call by Name und Call by Value sind unter referentieller Integrität *fast* äquivalent.

Wenn $sehrTeuer(25)$ nicht definiert ist (Fehler oder Endlosschleife) dann sind die Ergebnisse verschieden.

$$list(n) = n : list(n+1)$$

Es sei [] eine leere Liste und $x : l$ möge die Liste sein, deren erstes Element x ist gefolgt von der Liste l.

Dann ist $list(4) = [4, 5, 6, 7, 8, 9, \ldots]$ eine unendliche Liste!

$$take(n, x : rest) = \begin{cases} x & \text{falls } n = 1, \\ take(n-1, rest) & \text{sonst.} \end{cases}$$

$take(n, l)$ gibt uns das $n$te Element der Liste l.

Beispiele:

$take(3, [1, 4, 9, 16, 25]) = 9$

$take(6, [1, 2, 3, 4, 5, 6, 7, \ldots]) = 6$

$take(3, list(4)) = 6$

Lazy Evaluation

$$
\begin{aligned}
take(3, list(4)) &= take(3, 4 : list(4+1)) \\
&= take(3-1, list(4+1)) \\
&= take(2, list(4+1)) \\
&= take(2, (4+1) : list((4+1)+1)) \\
&= take(2-1, list((4+1)+1)) \\
&= take(1, list((4+1)+1)) \\
&= take(1, ((4+1)+1) : list(((4+1)+1)+1) \\
&= (4+1)+1 \\
&= 5+1 \\
&= 6
\end{aligned}
$$

Wir nennen dies auch *Lazy Evaluation*, denn es wird nur das evaluiert, was wir wirklich brauchen.

Überblick

11 Funktionale Programmierung

- Allgemeines
- Haskell
- MapReduce

Funktionale Programmierung

Vereinfacht gesagt:

Wir definieren einige Funktionen.

Wir lassen eine von ihnen z.B. mit Lazy Evaluation auswerten und erhalten das Ergebnis.

Das ist funktionale Programmierung.

Ein Programm besteht aus *Funktionsdeklarationen*:

$$list(n) \;=\; n \;:\; list(n+1)$$

$$take(n, x : rest) \;=$$

$$\quad \textbf{if}\; n = 1$$

$$\quad \textbf{then}\; x$$

$$\quad \textbf{else}\; take(n-1, rest)$$

Haskell

Wir verwenden die funktionale Programmiersprache *Haskell*.

Ein Programm besteht aus einer Folge von Deklarationen, die Funktionen definieren.

$$\begin{aligned} \mathit{factorial}(1) &= 1 \\ \mathit{factorial}(n) &= n \ * \ \mathit{factorial}(n-1) \end{aligned}$$

Gibt es mehrere Deklarationen für *factorial*, dann wird die erste verwendet, welche „paßt“.

- Für $\mathit{factorial}(1)$ wird die erste Zeile verwendet.
- Für $\mathit{factorial}(5)$ wird die zweite Zeile verwendet.

Evaluierung

$$
\begin{aligned}
factorial(1) &= 1 \\
factorial(n) &= n * factorial(n-1)
\end{aligned}
$$

Wie wird $factorial(3)$ evaluiert?

$factorial(3)$

$3 * factorial(3-1)$

$3 * factorial(2)$

$3 * (2 * factorial(2-1))$

$3 * (2 * factorial(1))$

$3 * (2 * 1)$

$3 * 2$

6

Es wird die erste passende linke Seite, durch die rechte Seite ersetzt.

Dabei passt n zu jeder natürlichen Zahl.

Typdeklarationen

Eine „anständige“ Definition einer Funktion umfaßt auch ihren Typ:

$$\mathit{factorial} : \mathbf{N} \to \mathbf{N}, \;\; \mathit{factorial}(n) = \ldots$$

In Haskell können wir neben Funktionsdeklarationen auch *Typdeklarationen* erstellen:

$$\begin{aligned} &\mathit{factorial} :: \mathit{Integer} \to \mathit{Integer} \\ &\mathit{factorial}(1) \;=\; 1 \\ &\mathit{factorial}(n) \;=\; n * \mathit{factorial}(n-1) \end{aligned}$$

Typdeklarationen sind nicht immer nötig, aber wir sollten sie immer erstellen. Dadurch wird die Fehlersuche viel leichter.

Typdeklarationen

Es gibt in Haskell bereits primitive Typen:

- *Integer*: ganze Zahlen, z.B. 1289736781236
- *Int*: ganze Zahlen mit Computerarithmetik, z.B. 123
- *Double*: Fließkommazahlen, z.B. 3.14159
- *String*: Zeichenketten, z.B. `"Hello, World"`
- *Char*: ein Zeichen, z.B. `'A'`

Wir können auch neue Typen deklarieren.

Notation

Hat eine Funktion ein Argument muß dieses nicht geklammert werden.

Wir schreiben also lieber

$$\begin{aligned} \mathit{factorial}\ 1 &= 1 \\ \mathit{factorial}\ n &= n * \mathit{factorial}\ (n-1) \end{aligned}$$

Die $(n-1)$ muß geklammert sein, weil sonst implizit so geklammert würde:

$$(n * (\mathit{factorial}(n))) - 1$$

Tupel

Es gibt in Haskell *Tupel*:

Beispiele: `(1,2)`, `("John", "Doe")`, `(17, "plus", 4)`

Eine Funktion hat streng genommen immer nur ein Argument.

Mehrstellige Funktionen können durch Tupel „simuliert“ werden:

$$sum1 :: (Integer, Integer) \to Integer$$
$$sum1\ (x, y) = x + y$$

Man bevorzugt aber solche Funktionen:

$$sum2 :: Integer \to Integer \to Integer$$
$$sum2\ x\ y = x + y$$

Technisch gesehen ist *sum2* eine Funktion, die eine Zahl auf eine andere Funktion abbildet, die dann eine Zahl auf eine Zahl abbildet.

So werden aber praktisch mehrstellige Funktionen umgesetzt.

Was steckt hinter *sum2*?

$sum2 :: Integer \rightarrow Integer \rightarrow Integer$
$sum2\ x\ y\ =\ x + y$

Tatsächlich ist $sum2\ y$ eine Funktion.

Insbesondere ist $sum2\ 5$ die Funktion $x \mapsto x + 5$.

Wir können $(sum2\ 5)(3)$ schreiben.

Die Deklaration $f\ =\ sum2\ 5$ ist auch möglich.

So können wir eine Exponentialfunktion definieren:

$power3 :: Integer \rightarrow Integer \rightarrow Integer$
$power3\ n\ 0\ =\ 1$
$power3\ n\ m\ =\ n * power3\ n\ (m - 1)$

Folgendes Programm ist ineffizient, weil $fib(x-1)$ und $fib(x-2)$ *beide* aufgerufen werden.

$$
\begin{aligned}
&fib :: Integer \rightarrow Integer \\
&fib\ 0 \;=\; 0 \\
&fib\ 1 \;=\; 1 \\
&fib\ x \;=\; fib\ (x-1)\ +\ fib\ (x-2)
\end{aligned}
$$

Die Laufzeit ist exponentiell.

Die Ausführung könnte durch Caching von bereits berechneten *fib*s automatisch optimiert werden.

Dies macht der Compiler aber leider nicht.

In funktionalen Sprachen können wir nicht einfach Werte „speichern“.

Um F_n zu berechnen, benötigen wir F_{n-1} *und* F_{n-2}.

Lösung: Statt nur F_n berechnen wir das Paar $P_n = (F_n, F_{n+1})$.

Jetzt läßt sich P_{n+1} aus P_n allein berechnen.

```
fibpair :: Integer -> (Integer, Integer)
fibpair 0  =  (0, 1)
fibpair n  =  (b, a + b)
  where (a, b)  =  fibpair (n - 1)
```

Durch eine **where**-Anweisung können wir verhindern, *fibpair*$(n-1)$ zweimal aufzurufen. Durch **where**-Anweisungen können wir Zwischenergebnissen einen Namen geben.

Listen

Listen sind in funktionalen Sprachen allgemein eine wichtige Datenstruktur.

In Haskell sind Listen vordefiniert:

1. $[\,]$ ist die leere Liste.
2. $x : xs$ ist die Liste deren Kopf aus x besteht, gefolgt von den Elementen aus der Liste xs.

Beispiel: $1 : 2 : 3 : 4 : 5 : [\,]$

Abkürzung:

$[a, b, c, d]$ ist eine Abkürzung für $a : b : c : d : [\,]$.

Typnamen:

[*Integer*] ist der Typ „Liste von *Integer*s“.

Arbeiten mit Listen

So verdoppeln wir die Einträge einer Liste:

$$verdoppeln :: [Integer] \rightarrow [Integer]$$
$$verdoppeln\ [\,] \ = \ [\,]$$
$$verdoppeln\ (x : xs) \ = \ 2 * x \ : \ verdoppeln\ xs$$

So können wir die Reihenfolge umdrehen:

$$umdrehen :: [Integer] \rightarrow [Integer]$$
$$umdrehen\ [\,] \ = \ [\,]$$
$$umdrehen\ (x : xs) \ = \ (umdrehen\ xs) +\!+ [x]$$

Der Operator $+\!+$ konkateniert zwei Listen.

Beispiel: Mergesort

Ein einfaches Sortierverfahren heißt *Mergesort*.

Es funktioniert so:

1. Teile die Liste in zwei etwa gleich lange Listen $l1$ und $l2$.
2. Sortiere $l1$ und $l2$.
3. Mische $l1$ und $l2$ in eine sortierte Liste.

Wir implementieren daher drei Funktionen *teile*, *mische* und *mergesort*.

Die folgende Funktion verteilt Elemente einer Liste immer abwechselnd auf zwei neue Listen.

```
teile :: [Integer] -> ([Integer], [Integer])
teile [ ] = ([ ], [ ])
teile (x : [ ]) = ([x], [ ])
teile (x : y : [ ]) = ([x], [y])
teile (x : y : xs) = (x : r1, y : r2)
  where (r1, r2) = teile xs
```

```
mische :: [Integer]→[Integer]→[Integer]
mische [ ] l  =  l
mische l [ ]  =  l
mische (x : xs) (y : ys)  =
  if x  <  y
  then x  :  mische xs (y : ys)
  else y  :  mische (x : xs) ys
```

```
mergesort :: [Integer]→[Integer]
mergesort [ ]  =  [ ]
mergesort (x : [ ])  =  [x]
mergesort l  =  mische (mergesort l1) (mergesort l2)
  where (l1, l2)  =  teile l
```

Beispiel: Partition-Exchange-Sort

Ein weiteres schnelles Sortierverfahren heißt Partition-Exchange-Sort oder Quicksort.

Es geht so vor:

1. Wähle ein Pivot-Element p (z.B. das erste Element im Array).
2. Partitioniere das Array in zwei Arrays. Das erste enthält alles was kleiner als p ist, das zweite den Rest.
3. Sortiere beide Arrays rekursive.
4. Packe p in die Mitte.

Bei der imperativen Implementierung mit Arrays geschieht das Partitionieren mithilfe von Vertauschungen, daher der Name Partition-Exchange.

Das Vergleichen mit dem Pivot-Element und weitersuchen benötigt typischerweise nur vier Maschineninstruktionen. Daher der Name *Quicksort*.

Wir implementieren dieses Verfahren mit Listen.

Hierzu implementieren wir die Funktionen *psort* und *partition*, welches ein Pivot-Element und eine Liste erhält.

```
psort :: [Integer] -> [Integer]
psort [ ] = [ ]
psort (x : xs) = (psort l1) ++ [x] ++ (psort l2)
  where (l1, l2) = partition x xs
```

```
partition :: Integer -> [Integer] -> ([Integer], [Integer])
partition x [ ] = ([ ], [ ])
partition x (y : ys) =
  if x < y
  then (l1, y : l2)
  else (y : l1, l2)
    where (l1, l2) = partition x ys
```

Typparameter

Betrachten wir diese Funktion:

$$zweites :: [Integer] \longrightarrow Integer$$
$$zweites\ (x : y : ys) \ = \ y$$

Sie findet das zweite Element in einer Liste von *Integer*.

Die Funktionalität sollte aber auch für beliebige Listen funktionieren.

Wir können *Typvariablen* in einer Signatur verwenden, um solch allgemeine Funktionen zu definieren. Dies ähnelt den generischen Typen von Java.

$$zweites :: [a] \longrightarrow a$$
$$zweites\ (x : y : ys) \ = \ y$$

Eigene Datentypen

So können wir einen einfachen Datentyp *Note* definieren.

Der Typ *Note* kann nur bestimmte Werte annehmen.

```
data Note = Gut | Schlecht | Mies deriving Show
eineschlechter :: Note → Note
eineschlechter Gut = Schlecht
eineschlechter Schlecht = Mies
```

Wir können eigene Datentypen ebenso wie *Integer* etc. verwenden.

deriving *Show* sagt, daß *Note* ein Untertyp von *Show* ist.
Dadurch können seine Werte einfach angezeigt werden.

Eigene Datentypen

Ein Datentyp kann von einem anderen Typ abhängen.

So definieren wir eine eigene Liste von *a*'s, wobei *a* wieder eine Typvariable ist. Wir können also auch so einen eigenen Listentyp für *Integer*-Listen definieren, aber auch für beliebige andere.

```
data List a = Nil | Cons a (List a) deriving Show
kopf :: List a -> a
kopf (Cons x y) = x
schwanz :: List a -> List a
schwanz (Cons x xs) = xs
append :: List a -> List a -> List a
append Nil l = l
append (Cons x xs) l = Cons x (append xs l)
laenge :: List a -> Integer
laenge Nil = 0
laenge (Cons _ xs) = 1 + (laenge xs)
```

Eigene Datentypen

Hier ist ein Beispiel für Binärbäume, die Werte des Typs *a* in ihren Blättern besitzen.

```
data Bintree a = Leaf a | Node (Bintree a) (Bintree a)
  deriving Show
size :: Bintree a → Integer
size (Leaf x) = 1
size (Node l r) = (size l) + (size r)
data OrdBintree a = Ext | In a (OrdBintree a) (OrdBintree a)
  deriving Show
insert :: Ord a => a→OrdBintree a→OrdBintree a
insert x Ext = In x Ext Ext
insert x (In y yl yr) =
  if x < y
  then In y (insert x yl) yr
  else In y yl (insert x yr)
```

Geordnete Binärbäume. Die linken Nachkommen eines Knotens sind stets kleiner, die rechten stets größer (oder gleich).

```
data OrdBintree a = Ext | In a (OrdBintree a) (OrdBintree a)
  deriving Show
insert :: Ord a => a -> OrdBintree a -> OrdBintree a
insert x Ext = In x Ext Ext
insert x (In y yl yr) =
  if x < y
  then In y (insert x yl) yr
  else In y yl (insert x yr)
listtotree :: Ord a => [a] -> OrdBintree a
listtotree [ ] = Ext
listtotree (x : xs) = insert x (listtotree xs)
```

Durch *Ord a => a* wird ein beliebiger Typ bezeichnet, der aber geordnet ist. Sonst kann $<$ nicht verwendet werden.

Funktionen höherer Ordnung

Eine besonders mächtiges Werkzeug der funktionalen Programmierung sind Funktionen, die Funktionen auf Funktionen abbilden. Wir nennen diese auch Funktionen höhrerer Ordnung.

Einfaches Beispiel:

Wandle eine Funktion $f\colon A \times B \to C$ in eine Funktion $g\colon B \times A \to C$ um, so daß $g(x,y) = f(y,x)$ gilt.

```
umdrehen :: (a->b->c)->(b->a->c)
umdrehen f x y  =  f y x
loesche :: Integer->[Integer]->[Integer]
loesche x [ ]  =  [ ]
loesche x (y : ys)  =
   if x == y
   then loesche x ys
   else y  :  loesche x ys
uloesche  =  umdrehen loesche
```

Mit *toweroftwo* 4 können wir $2^{2^{2^{2}}}$ berechnen.

$$
\begin{aligned}
&\mathit{zweimal} :: (a \longrightarrow a) \longrightarrow (a \longrightarrow a)\\
&\mathit{zweimal}\ f\ x \ = \ f\ (f\ x)\\[1em]
&\mathit{kmal} :: \mathit{Integer} \longrightarrow (a \longrightarrow a) \longrightarrow (a \longrightarrow a)\\
&\mathit{kmal}\ 0\ f\ x \ = \ x\\
&\mathit{kmal}\ n\ f\ x \ = \ f\ (\mathit{kmal}\ (n-1)\ f\ x)\\[1em]
&\mathit{zweierpotenz} :: \mathit{Integer} \longrightarrow \mathit{Integer}\\
&\mathit{zweierpotenz}\ k \ = \ \mathit{kmal}\ k\ ({*}2)\ 1\\[1em]
&\mathit{toweroftwo} :: \mathit{Integer} \longrightarrow \mathit{Integer}\\
&\mathit{toweroftwo}\ k \ = \ \mathit{kmal}\ k\ \mathit{zweierpotenz}\ 1
\end{aligned}
$$

Mit *nach* können wir aus f und g ihre Komposition $f \circ g$ berechnen.

```
nach :: (b→c)→(a→b)→(a→c)
nach f g = \x → f (g x)
```

Beispiel:

$f \;=\; nach\ (*2)\ (+4)$ definiert die Funktion

$$f : n \mapsto 2(n+4).$$

Anstatt $f \;=\; nach\ g\ h$ können wir auch $f \;=\; g$ ‘*nach*‘ h schreiben.

Eine solche Funktion ist schon vordefiniert: $f \;=\; g\ .\ h$

lmap wendet eine Funktion auf alle Elemente einer Liste an.

$$lmap :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$
$$lmap\ f\ [\,] \;=\; [\,]$$
$$lmap\ f\ (x : xs) \;=\; f\ x\ :\ lmap\ f\ xs$$

Beispiel:

$lmap\ (*2)\ [1, 2, 3, 4, 5, 6]$

$lmap\ head\ [[1, 2],\ [3, 4],\ [5, 6]]$

$doppeln\ =\ lmap\ (*2)$

Auch *lmap* ist schon vordefiniert und heißt einfach *map*.

Lambda-Ausdrücke

Wir können eine Funktion $n \mapsto 5n$ definieren, ohne ihr einen Namen zu geben:

$\backslash n {\rightarrow} 5 * n$

Dies ist an den λ-Kalkül angelehnt. Dort schreibt man $\lambda n.5n$.

Beispiel:

$map\ (\backslash x {\rightarrow} 2 * x + 5)\ [1, 2, 3, 4, 5, 6]$

$umgekehrt\ f\ =\ \backslash x\ y {\rightarrow} f\ y\ x$

Frage:

Was ergibt $(\backslash x\ y {\rightarrow} (/)\ y\ x)\ 2\ 3$?

RWTHAACHEN

Fragen

$$
\begin{aligned}
f\ [\,] &= [\,] \\
f\ (x : y : ys) &= x * y\ :\ f\ ys \\
f\ (x : xs) &= f\ (x : x : xs)
\end{aligned}
$$

Wozu evaluiert $f\ [1, 2, 3]$ (Abkürzung für $f\ (1 : 2 : 3 : [\,])$)?

Wozu evaluiert $[f\ [\,],\ f[\,]]$?

Weiteres Beispiel:

$$
\begin{aligned}
f\ [\,]\ y &= [\,] \\
f\ (x : xs)\ y &= x : f\ xs\ y \\
g\ [x] &= x : g\ [x]
\end{aligned}
$$

Wozu evaluiert $f\ [5]\ (g\ [3])$?

Evaluierung von links nach rechts

$$f\ [\,]\ x\ =\ [\,]$$
$$g\ =\ [\,]$$
$$h\ =\ 1\ :\ h$$

Wozu evaluiert der Ausdruck $f\ g\ h$?

- h läßt sich nicht evaluieren. Es ergäbe sich die unendliche Liste $[1, 1, 1, ...]$.
- $f\ g\ h$ läßt sich mit keiner Definition matchen.
- Daher wird erst einmal g einmal evaluiert.
- Daraus ergibt sich der Ausdruck $f\ [\,]\ h$, welcher jetzt zur Definition von f paßt.
- Jetzt erhalten wir die rechte Seite $[\,]$ als Ergebnis.
- Es ist wichtig, daß Hakell von links nach rechts evaluiert.

Funktionales Morsedekodieren

Zum Vergleich mit Java dekodieren wir denselben Morsecode nun mit Haskell.

Dies ist eine Aufgabe, die sich gut mit einem funktionalem Programm lösen läßt.

Wir gehen ähnlich vor:

```
main = do
  contents ← readFile "audio.txt"
  (print . decode2 . decode1 . read) contents
```

Wir dekodieren in zwei Stufen:

```
decode1 = (cummulate 0) . threshold . (average 200) . absolute
decode2 = demorse . (crack "") . (foldl (++) "") . (map symbolize)
```

Absolute

Wir berechnen die Absolutwerte einer Liste. Am einfachsten können wir dies mit *map* erledigen und einer Funktion, die den Absolutwert eines *Double* berechnet.

$$\mathit{absolute} :: [\mathit{Double}] \rightarrow [\mathit{Double}]$$
$$\mathit{absolute} \;=\; \mathit{map}\ (\backslash x \rightarrow \textbf{if}\ x > 0\ \textbf{then}\ x\ \textbf{else}\ -x)$$

Ohne Funktionen höherer Ordnung würden wir alternativ schreiben:

$$\mathit{absolute}\ [\,] \;=\; [\,]$$
$$\mathit{absolute}\ (x : \mathit{xs}) \;=\; \mathit{absx}\ :\ \mathit{absolute}\ \mathit{xs}$$
$$\quad \textbf{where}\ \mathit{absx} \;=\; \textbf{if}\ x > 0\ \textbf{then}\ x\ \textbf{else}\ -x$$

Die Folgende Funktion berechnet die Durschnittswerte von allen Unterlisten einer gegebenen Länge.

Zum Beispiel: *average* $2\ [1, 2, 3, 4, 5, 6]$ soll das Ergebnis $[1.5,\ 2.5,\ 3.5,\ 4.5,\ 5.5]$ liefern.

$$
\begin{array}{l}
average :: Double \to [Double] \to [Double] \\
average\ k\ l\ =\ map\ (\backslash x \to x/k)\ (listdiff\ (chop\ (k-1)\ ll)\ (0 : ll)) \\
\quad \textbf{where}\ ll\ =\ prefixsum\ l\ 0
\end{array}
$$

Wenn die Fensterlänge k ist, dann berechnen wir zuerst die Präfixsummen p, von diesen eine neue Liste, von der nur die ersten $k-1$ Elemente fehlen und ziehen von ihr elementweise $0 : p$ ab. Am Ende muß noch durch k geteilt werden.

Beispiel: $p = [1, 3, 6, 10, 15, 21]$

$[3, 6, 10, 15, 21]$ minus $[0, 1, 3, 6, 10, 15, 21]$ ergibt $[3, 5, 7, 9, 11]$.

Die Hilfsfunktionen *prefixsum*, *chop* und *listdiff* sind einfach:

$$
\begin{aligned}
&prefixsum :: [Double] \rightarrow Double \rightarrow [Double] \\
&prefixsum\ [\,]\ a\ =\ [\,] \\
&prefixsum\ (x : xs)\ a\ =\ (x + a)\ :\ prefixsum\ xs\ (x + a)
\end{aligned}
$$

$$
\begin{aligned}
&chop :: Double \rightarrow [Double] \rightarrow [Double] \\
&chop\ 0\ l\ =\ l \\
&chop\ k\ (x : xs)\ =\ chop\ (k - 1)\ xs
\end{aligned}
$$

$$
\begin{aligned}
&listdiff :: [Double] \rightarrow [Double] \rightarrow [Double] \\
&listdiff\ [\,]\ l\ =\ [\,] \\
&listdiff\ (x : xs)\ (y : ys)\ =\ x - y\ :\ (listdiff\ xs\ ys)
\end{aligned}
$$

In *prefixsum* nutzen wir einen Akkumulator, um lineare Laufzeit zu erhalten.

$$
\begin{aligned}
&\mathit{threshold} :: [\mathit{Double}] \rightarrow [\mathit{Int}] \\
&\mathit{threshold}\ [\,] \ = \ [\,] \\
&\mathit{threshold}\ (x : xs) \ = \ (\textbf{if}\ x > 0.2\ \textbf{then}\ 1\ \textbf{else}\ -1)\ :\ (\mathit{threshold}\ xs)
\end{aligned}
$$

Der Kürze wegen übersetzt *threshold* alle Werte in einer Liste zu -1 oder 1, je nachdem ob sie größer als 0.2 sind.

In Java hatten wir einen Clustering-Algorithmus verwendet, um diesen Schwellwert automatisch zu bestimmen. Darauf verzichten wir jetzt.

```
cummulate :: Int -> [Int] -> [Int]
cummulate 0 (x : xs)  =  cummulate x xs
cummulate n [ ]  =  [n]
cummulate n (x : xs)  =
   if (n > 0) == (x > 0)
   then cummulate (n + x) xs
   else n  :  cummulate 0 (x : xs)
```

Die Funktion *cummulate* berechnet die Länge von Blöcken, die nur aus -1 oder 1 bestehen.

Beispiel:

$$cummulate\,[1,1,1,-1,-1,1,1,-1,-1,-1,-1] \;=\; [3,\;-2,\;2,\;-4]$$

```
symbolize :: Int -> String
symbolize x
        | x <  -20000 = " "
        | x <  0 = ""
        | x >  2000 = "-"
        | otherwise = "."
```

In *symbolize* verwenden wir *guarded commands*, welche bisher nicht vorkamen. Mit ihnen lassen sich gut viele verschiedene Fälle behandeln.

Wir übersetzen kurze Signale in einen Punkte und lange in einen Strich. Lange Pausen werden zu einem Leerzeichen übersetzt.

```haskell
crack :: String -> String -> [String]
crack a [ ] = [a]
crack a (x : xs) =
  if x == ' '
  then a : crack "" xs
  else crack (a++[x]) xs
```

Diese Funktion bricht einen String in eine Liste von Teilstrings auf, wobei die Teilstrings von Leerzeichen getrennt wurden.

Beispiel:

```haskell
crack "" "-- . --- .-." = ["--", ".", "---", ".-."]
```

```
demorse :: [String]→String
demorse  =  map (\s → zeichen s tab "ABCDEFGHIJKLMNOPQRSTUVWXYZ")
```

demorse übersetzt eine Liste von Morsecodes in ihre jeweiligen Klartextsymbole.

Die Hilfsfunktion *zeichen s l1 l2* findet einen String *s* in der Liste *l1* auf Position *p* und gibt dann das *p*te Zeichen aus *l2* aus.

```
zeichen :: String→[String]→String→Char
zeichen _ [ ] _  =  '?'
zeichen z (x : xs) (y : ys)  =
  if z  ==  x
  then y
  else zeichen z xs ys
```

```
tab  =  [".-","-...","-.-.","-..",".","..-.","--.",
         "....","..",".---","-.-",".-..","--","-.",
         "---",".--.","--.-",".-.","...","-","..-",
         "...-",".--","-..-","-.--","--.."]
```

Überblick

11 Funktionale Programmierung

- Allgemeines
- Haskell
- MapReduce

Neues Kapitel

1 Einführung in die objektorientierte Programmierung
2 Rekursion
3 Fehler finden und vermeiden
4 Objektorientiertes Design
5 Effiziente Programme
6 Nebenläufigkeit
7 Laufzeitfehler
8 Refactoring
9 Persistenz
10 Eingebettete Systeme
11 Funktionale Programmierung
12 Logische Programmierung

Überblick

12 Logische Programmierung

- Prolog
- Syntax und Semantik

Logische Programmierung

Eine Weise der logischen Programmierung:

- Definition von Fakten
- Definition von Regeln

Wir definieren eine „Welt“ mittels Fakten und Regeln.

Dann stellen wir dem System Fragen über diese Welt.

Das System überlegt selbst, wie es diese Fragen beantworten kann.

Familienbeziehungen

Einige einfache Fakten und eine Schlußregel:

$parent(peter, petik).$
$parent(peter, kenny).$
$parent(dieter, peter).$
$parent(dieter, martin).$
$parent(martin, martinek).$
$grand(X, Y) :- parent(X, Z), parent(Z, Y).$

Die letzte Regel besagt, daß wenn X Elternteil von Z und Z von Y ist, dann ist X ein Großelternteil von Y.

$:-$ ist ein stylisierter Pfeil nach links. Die linke Seite „folgt“ von der rechten.

Variablen werden groß geschrieben.

Überblick

12. Logische Programmierung
    - Prolog
    - Syntax und Semantik

Syntax von Prolog

Atome sind

1. String aus Buchstaben, Ziffern und _, angefangen mit Kleinbuchstaben.
   Beispiele: *petik*, *xB7*, *a_HOPP*
2. String aus Sonderzeichen.
   Beispiele: ===> , .−−., :=:
3. String in einfachen Anführungszeichen.
   Beispiele: `'Petik'`, `'Hello, World'`

Syntax von Prolog

Variablen sind Strings aus Buchstaben, Ziffern und _, angefangen mit einem Großbuchstaben oder _

Beispiele: *X*, *X7*, *X_7*, *_anton*

Es gibt außerdem *Zahlen*.

Strukturen

Strukturen sind Objekte, die aus mehreren Komponenten bestehen.

Beispiele:

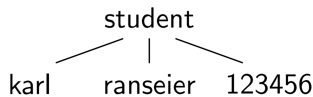
$student(karl, ranseier, 123456)$

$strecke(punkt(0,0), punkt(5,3))$

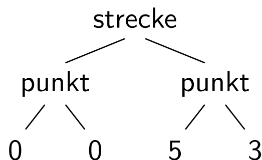
Der *Funktor* einer Struktur (z.B. $student$) ist ein Atom, die Komponenten sind Atome, Strukturen, oder Variablen.

Wir können Strukturen als Bäume darstellen:

$student(karl, ranseier, 123456)$



$strecke(punkt(0,0), punkt(5,3))$



Unifikation oder Matching

Sind S und T zwei Terme, so ist $S = T$ eine Anfrage, sie zu *unifizieren*:

- Sind S und T Konstanten (Atome oder Zahlen), dann unifizieren sie genau dann, wenn sie identisch sind.
- Ist S eine Variable, dann unifizieren S und T und S wird zu T *instantiiert*: Ab jetzt wird S stets durch T ersetzt:
- Ist T eine Variable dann ebenso.
- Sind S und T Strukturen, dann unifizieren sie, falls die Funktionen identisch sind und alle Komponenten unifizieren.

Unifikation – Beispiele

- $X = adam$ ✓
- $punkt(X, 7) = punkt(5, Y)$ ✓
- $punkt(X, 7) = punkt(7, X)$ ✓
- $punkt(X, 7) = punkt(5, X)$ –
- $X = punkt(Y, 3)$ ✓
- $strecke(X, Y) = strecke(Z, punkt(3, 5))$ ✓
- $strecke(X, Y) = X$?

Welche Instantiierungen finden statt?

Backtracking

Prolog versucht mit Backtracking, Anfragen mit seinen gespeicherten Deklarationen zu unifizieren. Scheitert dies an einer Stelle, dann werden die letzten Instantiierungen rückgängig gemacht und bei der nächsten Deklaration weitergemacht.

$punkt(3, 5).$
$punkt(1, 2).$
$punkt(2, 5).$
$nebeneinander(punkt(X1, Y), punkt(X2, Y)).$

Anfrage $punkt(X, Y), punkt(Y, Z)$

Zuerst wird $punkt(X, Y)$ mit der ersten Zeile unifiziert. Dann wird versucht $punkt(Y, Z)$ ebenfalls mit der ersten Zeile zu unifizieren, was scheitert, gefolgt von den anderen Zeilen. Schließlich scheitert $punkt(Y, Z)$ endgültig. Jetzt wird die Instantiierung von X, Y zurückgenommen und $punkt(X, Y)$ mit der zweiten Zeile unifiziert. Schließlich unifiziert $punkt(Y, Z)$ mit der letzten Zeile.

RWTHAACHEN

Backtracking

$punkt(3, 5).$
$punkt(1, 2).$
$punkt(2, 5).$
$nebeneinander(punkt(X1, Y), punkt(X2, Y)).$

Bei der Anfrage $nebeneinander(A, B)$ sind die ersten drei Deklarationen *belanglos*.

$nebeneinander(A, B)$ wird mit der vierten Zeile erfolgreich unifiziert.

Dabei wird $A = punkt(X1, Y)$ und $B = punkt(X2, Y)$ instantiiert.

Backtracking

$punkt(3, 5).$
$punkt(1, 2).$
$punkt(2, 5).$
$nebeneinander(punkt(X1, Y), punkt(X2, Y)).$

Die Anfrage
$nebeneinander(punkt(A, B), punkt(C, D)), punkt(A, B), punkt(C, D)$
liefert dagegen $(A, B) = (3, 5)$ und $(C, D) = (3, 5)$ als erste Lösung.

$nebeneinander(punkt(A, B), punkt(C, D))$ wird mit der letzten Zeile unifiziert, denn die Funktoren stimmen dort überein.

Was für Instantiierungen werden gemacht? Nur $B = D$.

Jetzt wird $punkt(A, B)$ mit der ersten Zeile unifiziert und $A = 3, B = 5$ instantiiert. Dann wird versucht $punkt(C, 5)$ mit einer Deklaration zu unifizieren.

RWTH AACHEN

Listen

In Prolog wird $[X|Y]$ für eine Liste mit Kopf X und Schwanz Y geschrieben.
Die leere Liste ist $[\,]$.
Wir können die Abkürzung $[1, 2, 3, 4, 5]$ benutzen. Beispiel:

```
removed(X, [X|Y], Y).
removed(X, [X1|Y], [X1|Y_ohne_X]) :-
    X \= X1,
    removed(X, Y, Y_ohne_X).
perm([ ], [ ]).
perm([X|Y], Z) :- member(X, Z), removed(X, Z, Z1), perm(Y, Z1).
aufsteigend([ ]).
aufsteigend([_]).
aufsteigend([X|[Y|Z]]) :- X =< Y, aufsteigend([Y|Z]).
sortiert(X, Y) :- perm(X, Y), aufsteigend(X).
```

$\backslash =$ ist ungleich, $=<$ und $>=$ sind in Prolog $\leq$ und $\geq$.

RWTH AACHEN

Beispiel

Es gibt das bekannte Rätsel:

$$\mathrm{SEND} + \mathrm{MORE} = \mathrm{MONEY}$$

Jeder Buchstabe muß durch eine Ziffer ersetzt werden, so daß die Rechnung stimmt.

Verschiedene Buchstaben müssen dabei verschiedene Ziffern erhalten.

Wie lösen wir solche Rätsel in Prolog?

Nehmen wir uns hier eine weniger bekannte Frage vor:

$$\mathrm{FH} \cdot \mathrm{RWTH} = \mathrm{AACHEN}$$

Beispiel

Wir können das Rätsel lösen, indem wir verlangen, daß $[R, W, T, H, F, A, C, E, N, _]$ eine Permutation der Ziffern $0, \ldots, 9$ ist *und* die Rechnung stimmt.

Dafür verwenden wir eine Relation $[num(N, L)]$, die aussagt, daß die Zahl N aus den Ziffern in der Liste L besteht.

$$
\begin{array}{l}
num(0, [\,]). \\
num(N, [X|XS]) :- num(N1, XS), N \text{ is } X + 10 * N1. \\
loesung(FH, RWTH) :- \\
\quad perm([R, W, T, H, F, A, C, E, N, _], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), \\
\quad num(FH, [H, F]), \\
\quad num(RWTH, [H, T, W, R]), \\
\quad num(AACHEN, [N, E, H, C, A, A]), \\
\quad AACHEN \text{ is } FH * RWTH.
\end{array}
$$

Beispiel – Umdrehen einer Liste

Drehen wir eine Liste naiv um, ist die Laufzeit wieder quadratisch.

Die schnelle Art eine Liste umzudrehen, verwendet ähnlich wie in Haskell einen Akkumulator.

Wir definieren eine Regel, die aussagt, daß „ein Element hinübergeschoben wurde“.

$$hinueber([\,], L, L).$$
$$hinueber([E|XS], YS, L) :- hinueber(XS, [E|YS], L).$$
$$umgedreht(A, B) :- hinueber(A, [\,], B).$$

Was ergiebt die Anfrage $umgedreht(X, X)$?

Löschen und Einfügen in eine Liste

Löschen des ersten Elements einer Liste:

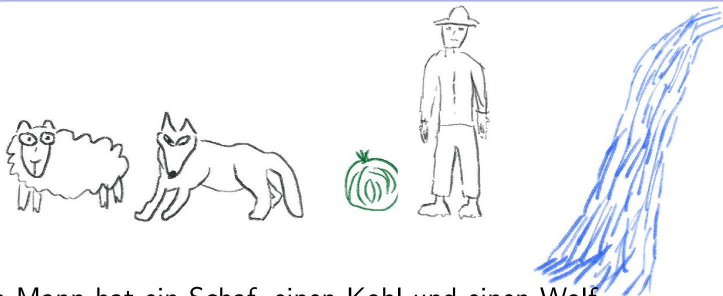
$loesche_erstes(X, [X|R], R).$
$loesche_erstes(X, [Y|R], [Y|RR]) :- X \backslash = Y, loesche_erstes(X, R, RR).$

Löschen *eines* Elements:

$loesche(X, [X|R], R).$
$loesche(X, [Y|R], [Y|RR]) :- loesche(X, R, RR).$

Kann man so auch Elemente *einfügen*?

Das Fährproblem



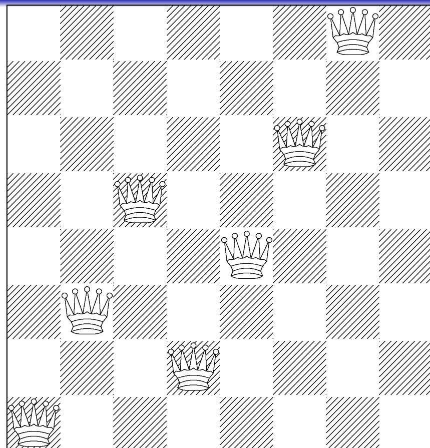
- Ein Mann hat ein Schaf, einen Kohl und einen Wolf.
- Er möchte sie auf die andere Flußseite bringen.
- Er darf das Schaf und den Wolf nicht allein lassen.
- Er darf auch das Schaf und den Kohl nicht allein lassen.
- Im Boot ist nur Platz für den Mann und einen der drei Dinge.

Wie kann er das schaffen?

Das Fährproblem

```
tausch([K, W, S, o], [K, W, S, w]).
tausch([o, W, S, o], [w, W, S, w]).
tausch([K, o, S, o], [K, w, S, w]).
tausch([K, W, o, o], [K, W, w, w]).
fahrt(X, Y) :- tausch(X, Y).
fahrt(X, Y) :- tausch(Y, X).
sicher(S) :- kohl_sicher(S), schaf_sicher(S).
kohl_sicher([K, _, _, K]).
kohl_sicher([w, _, o, _]).
kohl_sicher([o, _, w, _]).
schaf_sicher([_, _, S, S]).
schaf_sicher([_, w, o, _]).
schaf_sicher([_, o, w, _]).
erreichbar(0, [o, o, o, o], [ ]).
erreichbar(N, T, [S|L]) :- N > 0, fahrt(S, T), sicher(T),
                           N1 is N - 1, erreichbar(N1, S, L).
```

Das Acht-Damen-Problem



Löse das Problem rekursiv für die ersten k Spalten.

Dann versuche die $k+1$te Spalte zu besetzen.

Das Acht-Damen-Problem

$sol(0, [\,]).$
$sol(N, [Y|L])$:–
  $N > 0$, $N1$ *is* $N - 1$, $member(Y, [1, 2, 3, 4, 5, 6, 7, 8])$,
  $sol(N1, L)$,
  $not_same_row(Y, L)$, $not_same_diag1(Y, L)$, $not_same_diag2(Y, L)$.

$not_same_row(Y, [\,]).$
$not_same_row(Y, [Y1|R])$:– $Y \backslash = Y1$, $not_same_row(Y, R)$.

$not_same_diag1(Y, [\,]).$
$not_same_diag1(Y, [Y1|R])$:– YY *is* $Y - 1$, $YY \backslash = Y1$,
  $not_same_diag1(YY, R)$.
$not_same_diag2(Y, [\,]).$
$not_same_diag2(Y, [Y1|R])$:– YY *is* $Y + 1$, $YY \backslash = Y1$,
  $not_same_diag1(YY, R)$.

Hinzufügen von Fakten und Regeln

Durch **assert** (und *asserta*, *assertz*) können wir Fakten und Regeln zur Wissensbasis hinzufügen.

Hier ein Beispiel zu Memoization:

$:- \ dynamic \ fib/2.$
$fib(0, 0).$
$fib(1, 1).$
$fib(N, M) \ :-$
$\quad N > 1,$
$\quad N1 \ is \ N - 1, N2 \ is \ N - 2,$
$\quad fib(N1, M1),$
$\quad fib(N2, M2),$
$\quad M \ is \ M1 + M2,$
$\quad asserta(fib(N, M)).$

Die erste Zeile „erlaubt“ *asserta*.

RWTHAACHEN

Ausgabe

Das Ziel *write*(X) ist immer erfüllt.

Als Seiteneffekt wird X auf den Bildschirm ausgegeben.

Durch *nl* wird eine neue Zeile angefangen.

Türme von Hanoi als Beispiel:

```
hanoi(0, A, B, C).
hanoi(N, A, B, C) :-
    N > 0,
    N1 is N - 1,
    hanoi(N1, A, C, B),
    write('Nimm Scheibe von '), write(A),
    write(' und setze sie auf '), write(C), nl,
    hanoi(N1, C, B, A).
```