Programmiermethoden und Algorithmen

- Algorithmen und Datenstrukturen
- Software-Entwicklung mit Design Patterns
- Aufgaben
- Anhang

Inhaltsverzeichnis

Übersicht

- Algorithmen und Datenstrukturen
  - Rekursion
    - Fibonacci-Zahlen
    - Sortieren
    - Matrixmultiplikation
  - Bewertung von Algorithmen
  - Stack
  - Listen und Bäume
    - Verkettete Listen
    - Binärbäume
    - Ausgeglichene Bäume
  - Hash-Verfahren
- Software-Entwicklung mit Design Patterns
  - Einführung
  - Entwicklung eines UML-Editors
  - Zerlegung in Teile
  - Aufzählungstypen
  - Durchlaufen von Sammlungen
  - Zentrale Kontrollinstanz
  - Speichern von Objekten
  - Graphische Darstellung der Elemente
  - Bearbeiten der UML-Elemente
  - Relationen zwischen UML-Elementen
- Aufgaben
  - Aufgabe 1
    - Lösung von Aufgabe 1
  - Aufgabe 2
    - Lösung von Aufgabe 2
  - Aufgabe 3
    - Lösung von Aufgabe 3
  - Aufgabe 4
    - Lösung von Aufgabe 4
  - Aufgabe 5
    - Lösung von Aufgabe 5
  - Aufgabe 6
    - Lösung von Aufgabe 6
  - Aufgabe 7
    - Lösung von Aufgabe 7
  - Aufgabe 8

        - Lösung von Aufgabe 8
    - Aufgabe 9
- Anhang
    - Literatur
    - Nachweise
    - Prüfungsleistung
    - Sourcen
        - Bubblesort
        - Quicksort
        - MatrixMultiplikation
        - Intstack
        - ListenKnoten
        - Liste
        - BaumKnoten
        - Baum
        - AvlKnoten
        - AvlBaum
        - Class
        - Interface
        - Object
        - UmlElement
        - Field
        - Method
        - Member
        - Visibility (vorläufig)
        - UmlManager
        - Visibility
        - TestUmlManager
        - View (vorläufig)
        - ClassView
        - InterfaceView
        - ObjectView
        - MethodView
        - FieldView
        - UmlGui
        - Editor
        - ClassEditor
        - ListEditorPanel
        - InterfaceEditor
        - ObjectEditor
        - MethodEditor
        - FieldEditor
        - Arrow

    - ArrowType
    - ArrowView
    - ArrowEditor
    - Observer
    - View
    - ObserverAdapter
    - Matrix
    - TestMatrixMultiplikation
    - IntStack 2
    - Token
    - ArithmeticTokenizer
    - ArithmeticEvaluator
    - ListenKnoten 2
    - Liste 2
    - IntStackListe
    - BaumKnoten 2
    - Baum 2
    - BaumKnoten 3
    - Baum 3
    - AvlKnoten 2
    - AvlBaum 2
    - IoTools
    - TestBaeume
    - SimpleHashtable
    - Hashtable
    - TestHashtable
    - WordCounter
  - Applets
    - Bubblesort
    - Quicksort
    - TestUmlManager
    - UmlGui
    - Matrixmultiplikation
    - Integer-Stack
    - Klammerausdrücke
    - Integer-Stack mit Liste
    - Familienstammbaum
    - Vergleich von Baum und AVLBaum
    - Hash-Tabellen

Algorithmen und Datenstrukturen

- Rekursion
- Bewertung von Algorithmen
- Stack
- Listen und Bäume
- Hash-Verfahren

Rekursion

- Fibonacci-Zahlen
- Sortieren
- Matrixmultiplikation

Fibonacci-Zahlen

- Rekursive Funktion:
  - Funktion, die sich selbst aufruft
  - kann auch indirekt geschehen (a ruft b, b ruft a)
  - muss ein Abbruch-Kriterium haben, sonst unendliche Verschachtelung (führt zum Programm-Absturz)
  - Compiler muss viel Information verwalten: lokale Variablen aller aufgerufenen Funktionen (Stack Frames) gemäß Verschachtelungstiefe
  - Anwendungen z.B.
    - dynamische Datenstrukturen (Listen, Bäume)
    - Algorithmen mit Zerlegungsverfahren (Quicksort, Strassen)
  - manche einfachen Rekursionen lassen sich besser mit Schleifen programmieren
- Beispiel Fibonacci-Zahlen:
  - erfunden als Modell zur Kaninchenvermehrung
  - definiert durch
    - $a_0 = 0$
      $a_1 = 1$
      $a_{n+2} = a_{n+1} + a_n$
- Rekursive Implementierung:
  -

```
int fiboRek(int n) {
  switch (n) {
   case 0:
     return 0;
   case 1:
     return 1;
   default:
     return fiboRek(n-1) + fiboRek(n-2);
  }
}
```

  - Ausführung von fiboRek(4):
    -

```
fiboRec(4) = fiboRek(3) + fiboRek(2)
           = (fiboRek(2) + fiboRek(1)) + (fiboRek(1) + fiboRek(0))
           = ((fiboRek(1) + fiboRek(0)) + 1) + (1 + 0)
           = ((1 + 0) + 1) + 1
           = (1 + 1) + 1
           = 2 + 1
           = 3
```

  - 9 Funktionsaufrufe, 4 Additionen
- Iterative Implementierung:
  -

```
int fiboIt(int n) {
  switch (n) {
   case 0:
     return 0;
   case 1:
     return 1;
  }

  int letzter    = 1;
  int vorletzter = 0;
  int neuer      = 0;
  for (i=2; i<=n; i++) {
    neuer = letzter + vorletzter;
    vorletzter = letzter;
    letzter = neuer;
  }
  return neuer;
}
```

- Ausführung von fiboIt(4):

vorletzter	letzter	neuer
0	1	1
1	1	2
1	2	3
2	3	5

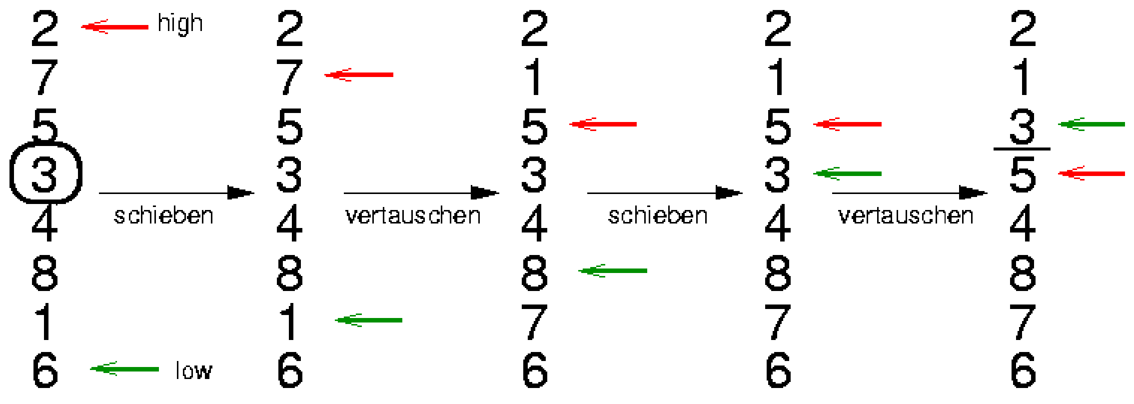
- 1 Funktionsaufruf, 3 Additionen

Sortieren

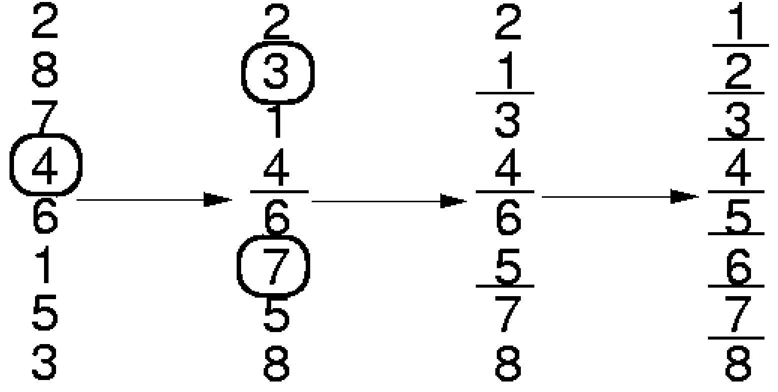
- Sortieren eines Arrays:
  - Voraussetzung: Datenmenge passt in den Speicher
  - Eingabearray mit n Elementen (z.B. Zahlen)
  - soll in aufsteigender Reihenfolge sortiert werden
  - Grundvoraussetzung zum schnellen Suchen in großen
- Bubblesort:
  - einfachstes Sortierverfahren
  - Grundprinzip:
    - Durchlaufe mehrmals das Array
    - vertausche jedesmal aufeinanderfolgende Elemente, falls verkehrt herum
    - so viele Durchgänge, bis keine Elemente mehr vertauscht wurden
  - Sortierroutine für double-Array
- Quicksort:
  - sehr schnelles Sortierverfahren
  - basiert auf Zerlegung der Eingabedaten und rekursiver Verarbeitung der Teile
  - Sortierroutine für double-Array
- Grundverfahren des Quicksort:
  - wähle ein beliebiges Element x aus ("mittleres Element")
  - sortiere alle Elemente kleiner als x nach oben, alle größer als x nach unten
  - verwende das Verfahren rekursiv zum Sortieren der beiden Teile
  - höre auf, wenn eine Teilliste ein oder kein Element enthält
- Verfeinerung des Aufteilens:
  - setze Markierung high nach oben, low nach unten
  - wiederhole

    schiebe high nach unten, bis ein großes Element kommt
    schiebe low nach oben, bis ein kleines Element kommt
    falls high nicht schon unter low:
      vertausche die beiden Elemente bei high und low
      setze high eins runter und low eins rauf

    solange high nicht unter low
- Beispiele:
  - Aufteilen:



- ganzer Quicksort, jeweils "mittleres Element" markiert



Matrixmultiplikation

- Multiplikation von Matrizen:
  - mathematische Definition von C = A * B
    - $$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$
  - zur einfachen Verarbeitung eigene Klasse Matrix
    -

Matrix
double[][] arr
getColumnDimension() getRowDimension() get(i, j) set(i, j, x) matmult(a, b)

  - Implementierung der Routine matmult über drei verschachtelte for-Schleifen
  - Anzahl der Operationen bei 2x2-Matrizen:
    - 8 Multiplikationen
    - 4 Additionen
- Beobachtung von Strassen [1]:
  - Betrachte 2x2-Matrizen
    - $$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} * \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$
  - Berechnung der c_{ij} durch geschickte Kombination von Zwischenergebnissen m_i

$$\begin{aligned}
m1 &= (a12 - a22) * (b21 + b22)\\
m2 &= (a11 + a22) * (b11 + b22)\\
m3 &= (a11 - a21) * (b11 + b12)\\
m4 &= (a11 + a12) * b22\\
m5 &= a11 * (b12 - b22)\\
m6 &= a22 * (b21 - b11)\\
m7 &= (a21 + a22) * b11
\end{aligned}$$

- $$\begin{aligned}
c11 &= m1 + m2 - m4 + m6\\
c12 &= m4 + m5\\
c21 &= m6 + m7\\
c22 &= m2 - m3 + m5 - m7
\end{aligned}$$
    - Anzahl der Operationen:
        - 7 Multiplikationen
        - 18 Additionen
    - Hoffnung: besser, da Multiplikationen "wichtiger" als Additionen
    - inzwischen Verbesserung mit weniger Additionen [2]
- Rekursion im Strassen-Algorithmus:
    - Zerlegung der Matrizen in 2x2 Blöcke
        - $$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} * \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$
    - a_{ij}, b_{ij}, c_{ij} jetzt Teilmatrizen der Dimension n/2
    - Berechnung des Produkts mit gleichen Formeln wie oben
    - Produkte der Teilmatrizen rekursiv mit gleichem Verfahren
    - ständiges Halbieren nur möglich, wenn Matrixdimension = Zweierpotenz
    - Abhilfen
        - Matrix mit Nullen erweitern
        - "ungerade" Zeilen/Spalten extra behandeln
- Strassen-Verfahren im Beispiel:
    - Ausgangsmatrizen
        - $$A = \left(\begin{array}{cc|cc} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 4 \\ \hline 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \end{array}\right) \quad B = \left(\begin{array}{cc|cc} 0 & -1 & -2 & -3 \\ 1 & 0 & -1 & -2 \\ \hline 2 & 1 & 0 & -1 \\ 3 & 2 & 1 & 0 \end{array}\right)$$
    - Berechnung von m1

- $$\begin{aligned} m1 &= \left[\begin{pmatrix} 2 & 3 \\ 3 & 4 \end{pmatrix} - \begin{pmatrix} 4 & 5 \\ 5 & 6 \end{pmatrix}\right] * \left[\begin{pmatrix} 2 & 1 \\ 3 & 2 \end{pmatrix} + \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}\right] \\ &= \begin{pmatrix} -2 & -2 \\ -2 & -2 \end{pmatrix} * \begin{pmatrix} 2 & 0 \\ 4 & 2 \end{pmatrix} \end{aligned}$$

  - Rekursion für das Teilprodukt

- $$\begin{aligned} m1 &= 0 * 6 = 0 \\ m2 &= (-4) * 4 = -16 \\ m3 &= 0 * 2 = 0 \\ m4 &= (-4) * 2 = -8 \\ m5 &= (-2) * (-2) = 4 \\ m6 &= (-2) * 2 = -4 \\ m7 &= (-4) * 2 = -8 \end{aligned}$$

  $$\begin{aligned} c11 &= -12 \qquad c12 = -4 \\ c21 &= -12 \qquad c22 = -4 \end{aligned}$$

  - Rückkehr aus Rekursion liefert

- $$m1 = \begin{pmatrix} -12 & -4 \\ -12 & -4 \end{pmatrix}$$

  - analog für m2 .. m7 und Gesamtprodukt C

Bewertung von Algorithmen

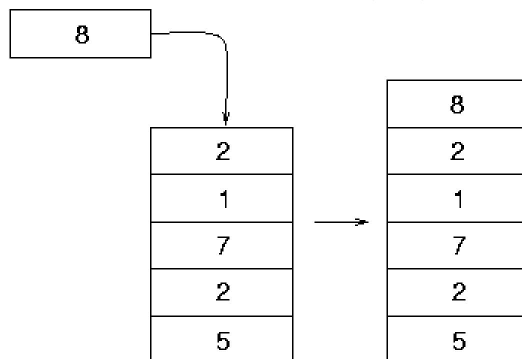
- Komplexität:
  - Größe der Eingabe sei beschrieben durch Parameter n
  - Beispiele für n
    - Zahl der Stellen der Eingabezahlen
    - Anzahl der zu bearbeitenden Werte
    - Zahl der Knoten in einem Graphen
  - Komplexität = Wachstum des Aufwands bei wachsendem n
  - Aufwand z.B.
    - Rechenzeit
    - Speicherbedarf
    - Ein-/Ausgabe-Operationen
  - genauer Aufwand von Rechnertyp, Betriebssystem u.v.a abhängig
  - Bestimmung der Komplexität
    - meistens nur wichtigster Term für großes n
    - teilweise erheblicher mathematischer Aufwand erforderlich
  - Beispiel "Addition zweier großer Zahlen":
    - Zeitaufwand ~ n
    - geschrieben O(n)
  - Beispiel "Multiplikation großer Zahlen":
    - Zeitaufwand ~ n^2
    - geschrieben $O(n^2)$
  - hängt manchmal von den speziellen Eingabewerten ab
- Zeitaufwand beim Sortieren:
  - Bubblesort
    - minimal 1 Durchgang (Liste ist schon sortiert) → O(n)
    - maximal n Durchgänge (kleinstes Element unten) → $O(n^2)$
    - Mittelwert über alle Reihenfolgen: $O(n^2)$
  - Quicksort
    - hängt von guter Wahl des "mittleren Elements" ab
    - schlechtester Fall: $O(n^2)$
    - mittlerer Fall: O(n log(n))
  - Vergleich der Größenordnungen, wenn 10^6 Schritte 1s brauchen:
    -

n	Zeit bei n log(n)	Zeit bei n^2
10^2	0.5 ms	10 ms
10^3	6.9 ms	1 s
10^4	92 ms	1.7 m
10^5	1.2 s	2.8 h
10^6	13.8 s	11.6 d

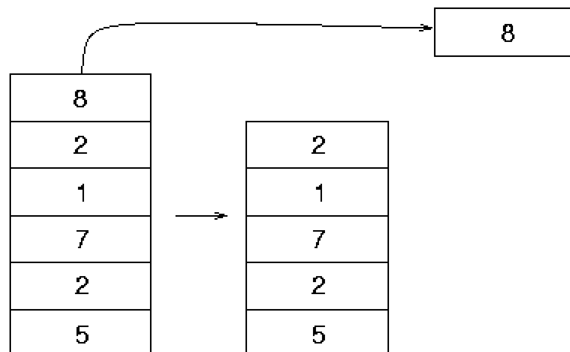
  - direkter Vergleich im Applet mit Zufallszahlen
    - Bubblesort-Applet
    - Quicksort-Applet
- Zeitaufwand der Matrix-Multiplikation:
  - betrachten nur nxn-Matizen
  - untergliedert in
    - Zahl der Multiplikationen M(n)
    - Zahl der Additionen A(n)
  - bestimmbar aus Rekursionsbeziehungen für $M(2^{h+1})$ und $A(2^{h+1})$
  - direktes Verfahren
    - $$\begin{aligned} M(n) &= n^3 \\ A(n) &= n^3 - n^2 \end{aligned}$$
  - Strassen-Algorithmus
    - $$\begin{aligned} M(n) &= n^{\log_2 7} \approx n^{2.807} \\ A(n) &\approx 6 \cdot (n^{2.807} - n^2) \end{aligned}$$
  - für große Matrizen
    - höhere Potenz 3 überwiegt größeren Vorfaktor
    - → Strassen ist schneller
- Strassen-Algorithmus in der Praxis:
  - benötigt zusätzlichen Speicherplatz und Verwaltungsaufwand für Zwischenergebnisse
  - Verhalten abhängig von Aufwand einer Multiplikation gegenüber einer Addition
  - frühere Abschätzung: lohnt sich erst ab n > 10000!
  - Tricks zur Verbesserung [3]
    - Rekursion bei kleinen Matrizen abbrechen und direkt multiplizieren
    - besserer Umgang mit Speicherplatz
    - geschicktes Verwalten der "ungeraden" Zeilen/Spalten
  - damit Strassen-Algorithmus schneller ab n > 1000
- Aufgaben:
  - Aufgabe 1

Stack

- Kellerspeicher (Stack):
  - realisiert einen LIFO-Speicher (Last In First Out)
    - Elemente eines beliebigen (aber jeweils festen) Typs werden abgespeichert
    - neue Elemente werden oben abgelegt



    - ein Element kann nur von oben entnommen werden



  - im Prinzip unendlich groß
- Operationen eines Stacks:
  - push(e)
    - legt das Element e auf dem Stack ab
  - e = pop()
    - holt das oberste Element vom Stack
  - status = empty()
    - prüft, ob der Stack leer ist
    - status ist boolean
  - optional: e = peek()
    - gibt das oberste Element des Stacks zurück, ohne es zu entfernen
    - identisch mit e = pop; push(e)
- Stack in Java:
  - vorhanden als java.util.Stack
  - pop und peek werfen ggf. EmptyStackException
  - Typ der Elemente immer Object

        - → beliebige Typen (auch durcheinander) in einem Stack
        - einfache Typen über Wrapper-Klassen (Double statt double etc.)
    - eigentlich gewünscht
        - Stack mit einem vorgebbaren Datentyp
        - z.B. Stack<int>, Stack<String> (generische Datentypen)
        - geplant für Java 1.5
- Verwaltung von Unterfunktionsaufrufen:
    - alle lokalen Variablen und Parameter einer Funktion bilden ein Element (Stackframe)
    - aktueller Stackframe
        - gehört zur aktuell ablaufenden Unterfunktion
        - enthält alle gerade zugreifbaren Variablen
    - Aufruf einer Unterfunktion
        - Stackframe des Aufrufers wird auf einem Stack abgelegt (push)
        - Unterfunktion erzeugt neuen aktuellen Stackframe
    - Rückkehr aus einer Unterfunktion
        - oberster Stackframe ersetzt aktuellen (pop)
        - Variablen des Aufrufers wieder da
        - Variablen des Unterprogramms unwiderruflich verloren
- Berechnung von arithmetischen Ausdrücken:
    - betrachten Ausdrücke aus
        - Zahlen (der Einfachheit halber positive ganze Zahlen)
        - Klammern ( )
        - Grundoperationen + - * /
    - Ausdrücke müssen vollständig geklammert sein
        - Operationen haben genau zwei Operanden
        - Operanden sind Zahlen oder Klammer-Ausdrücke
        - keine Punkt- vor Strichrechnung
    - Beispiele
        - ok: (((3 + 5) * 2)/(7 - 4))
        - falsch: (12 + 5 + 7), sondern ((12 + 5) + 7)
        - falsch: (2 + 4 * 6), sondern (2 + (4*6))
- Berechnungsverfahren mit Stack
    - Klammern, Operatoren und Zahlen gelten jeweils als ein Element (Token)
    - Eingabeelemente der Reihe nach verarbeiten:
        - alles außer) auf den Stack

```
bei ):
  operand2 = pop()
  operator = pop()
  operand1 = pop()
  dummy = pop()     // gibt (
  ergebnis = operator(operand1, operand2)
  push(ergebnis)
am Ende: ergebnis = pop()
```

- Beispielrechnung:
  - am Anfang
    - Eingabe Stack
      ((2+4)*(1-3)) —
  - nach 5 Schritten
    - Eingabe Stack
)*(1-3)) 4 + 2 ((
  - 6. Schritt
    -) gefunden → 4 Elemente holen 4, +, 2, (
    - berechne 2 + 4 → 6 auf Stack
    - Eingabe Stack
      *(1-3)) 6 (
    - nach 11. Schritt
    - Eingabe Stack
      )) 3 - 1 (* 6 (
  - 12. Schritt
    -) gefunden → 4 Elemente holen 3, -, 1, (
    - berechne 1 - 3 → -2 auf Stack
    - Eingabe Stack
) -2 * 6 (
  - 13. Schritt
    -) gefunden → 4 Elemente holen -2, *, 6, (
    - berechne 6 * -2 → -12 auf Stack
    - Eingabe Stack
      -12
  - Eingabe leer → Stack hat das Ergebnis

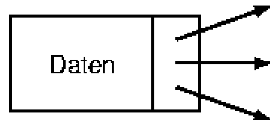
- Beispiel-Implementierung von Stack:
  - benötigte Datenfelder
    - Array als Speicher
    - Zeiger auf oberstes Element
  - Hauptproblem
    - Array hat feste Größe → Stack kann voll werden
    - schnelle Lösung: Ausnahme bei Stack-Überlauf
    - alternativ Liste statt Array verwenden
  - Methoden leicht geschrieben
    - IntStack.java
    - Ausnahmen bei Zugriff auf leeren Stack oder Stack-Überlauf fehlen
- Wachsender Stack mit Array:
  - Problem
    - Arraygröße am Anfang festgelegt
    - Array kann später nicht vergrößert werden
  - Lösung
    - bei Überlauf größeres Array anlegen
    - Werte vom alten ins neue kopieren
  - um wieviel vergrößern?
    - um einen → zu viel Kopieraufwand bei häufigem Wachstum
    - jedesmal um einen festen Wert (z.B. 100 neue Werte)
    - jedesmal auf doppelte Größe
- Warteschlange (Queue):
  - FIFO-Speicher (First In First Out)
  - Grundoperationen analog zum Stack
    - enter(e) // hinten anstellen
    - e = leave() // vorne herausnehmen
    - empty() // ist die Warteschlange leer?
  - Grundstruktur bei der Verteilung von Ressourcen, z.B.
    - Prozessverwaltung im Betriebssystem
    - Druckaufträge
    - Puffer bei nachrichtenübertragung
  - Implementierung etwas schwieriger als Stack, da es zwei Zugriffsenden gibt
- Aufgaben:
  - Aufgabe 2
  - Aufgabe 3

Listen und Bäume

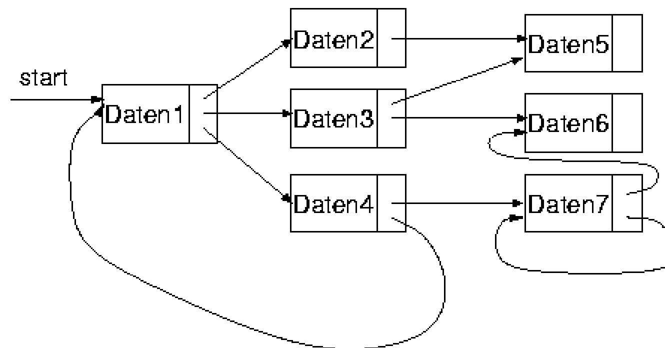
- Verkettete Listen
- Binärbäume
- Ausgeglichene Bäume

Verkettete Listen

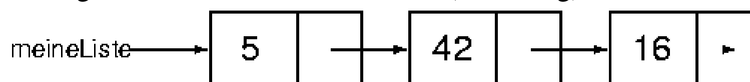
- Dynamische Datenstrukturen:
  - Datenstrukturen, deren Größe sich während der Laufzeit des Programms ändert
  - bestehen aus Basiseinheiten (Knoten) mit Daten und Zeigern



  - werden über die Zeiger zusammengehängt



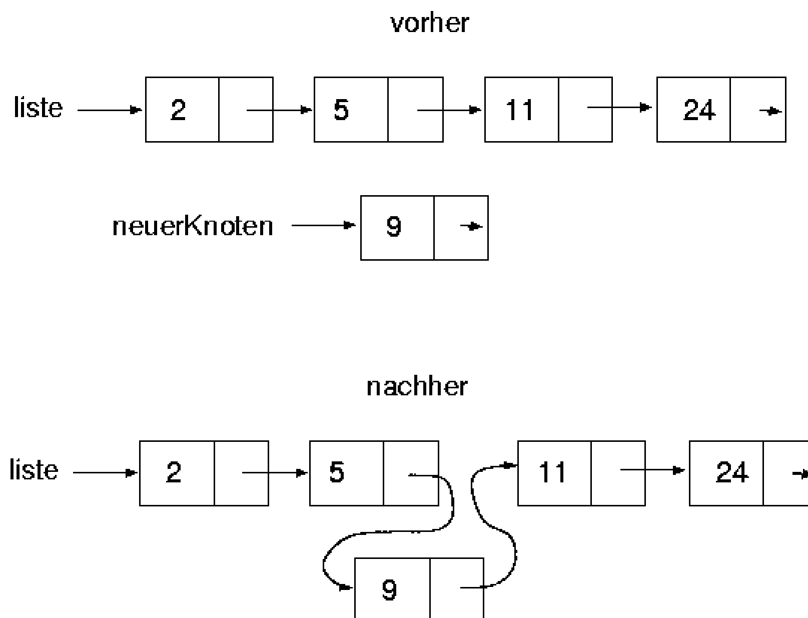
  - Knoten können verschiedene Anzahl von Zeigern besitzen
  - Zeiger können auf nichts verweisen (Wert null)
  - Zeiger können rückwärts oder auf den eigenen Knoten zeigen
  - mindestens ein Zeiger muss "von außen" auf die Struktur verweisen (im Bild der Zeiger start)
  - grundsätzlich beliebig viele Daten im Knoten
    - i.f. aber nur eine (Fließkomma-)Zahl
    - dient als Suchschlüssel für die anderen (nicht betrachteten) Daten
- Einfach verkettete Listen:
  - Knoten haben genau einen Zeiger
  - auf jeden Knoten zeigt genau ein Zeiger
  - der Zeiger des letzten Knotens ist null (kein Ring)



  - Knoten für Listen als Java-Klasse: ListenKnoten.java
    - Datenfelder von ListenKnoten haben Standard-Zugriffsklasse ("package")
    - direkter Zugriff möglich für Klassen aus dem selben Paket (z.B. Liste)
    - kein Zugriff für externe Klassen
  - Beispiel: Aufbau der obenstehenden Liste

```
ListenKnoten meineListe = new ListenKnoten(16.0);

ListenKnoten neuerKnoten = new ListenKnoten(42.0);
neuerKnoten.naechster = meineListe;
meineListe = neuerKnoten;

neuerKnoten = new ListenKnoten(5.0);
neuerKnoten.naechster = meineListe;
meineListe = neuerKnoten;
```

- Sortieren mit Listen:
  - Grundidee: Knoten direkt an die passende Stelle einfügen

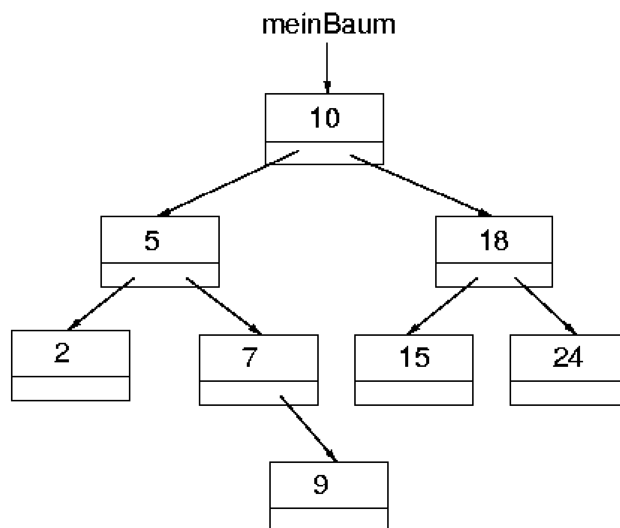


  - dazu zunächst durch die Liste laufen, um Einfügestelle zu finden
  - Problem:
    - Stelle gefunden, sobald größeres Element auftaucht
    - neues Element davor einfügen
    - Rückwärtslaufen geht nicht
  - Lösung: zweiten Zeiger mitnehmen, der immer einen zurückhinkt
  - Implementierung in Java:
    - zusätzliche Klasse Liste
    - enthält Zeiger auf 1. Element
    - enthält Methoden zum Einfügen, Suchen etc.
  - Fallunterscheidungen beim Einfügen nötig:
    - Liste ist leer → neues Element wird Start-Element
    - Start-Element ist kleinstes → wird einfach vorgehängt
    - Start-Element ist größtes → wird hinten angehängt
    - sonst richtig einfädeln
  - Beispielprogramm: Liste.java

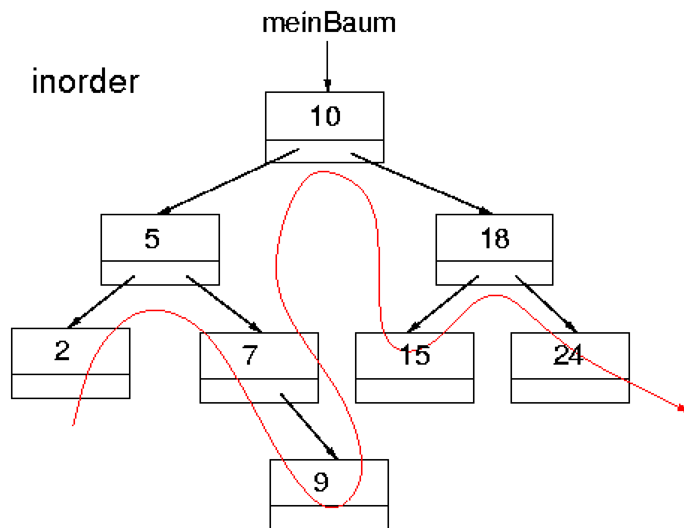
    - Nachteil: Suchen dauert lange (immer durch Liste durchlaufen)
- Standardklasse java.util.Vector:
    - sehr nützliche Kombination aus Liste und Array
    - verwendbar als wachsendes/schrumpfendes Array
    - implementiert Interface List →
        - add(Object o) fügt o an das Ende der Liste an
        - remove(Object o) entfernt (erstes Auftreten von) o aus der Liste
        - isEmpty() ist true, falls Liste leer
    - erlaubt Array-Zugriff
        - get(int index) gibt Objekt an Position index zurück
        - set(int index, Object o) speichert o an Position index
        - size() gibt Anzahl der Elemente zurück
- Aufgaben:
    - Aufgabe 4

Binärbäume

- Baum:
  - auf jeden Knoten zeigt genau ein Zeiger (keine Schlingen)
  - genau ein Knoten (Wurzel) hat keinen Vorgängerknoten
  - Blatt = Knoten ohne Nachfolger
  - Höhe eines Baumes h(B) = maximale Länge eines Weges im Baum B
  - Binärbaum = Knoten haben genau zwei Zeiger (darunter ggf. null-Zeiger)
  - Beispiel: Baum der Höhe 4 (in der Informatik wachsen Bäume nach unten)

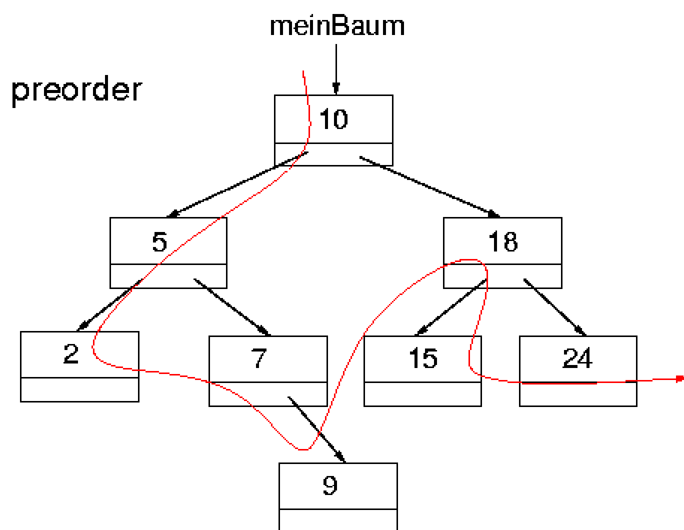


  - Knoten für Binärbäume als Java-Klasse: BaumKnoten.java
- Durchlaufen eines Baums:
  - z.B.zur Ausgabe aller Elemente oder zum Speichern der Daten
  - viele verschiedene Reihenfolgen möglich, drei besonders häufig
  - inorder:
    - zuerst linken Teilbaum, dann Wurzel, dann rechten Teilbaum



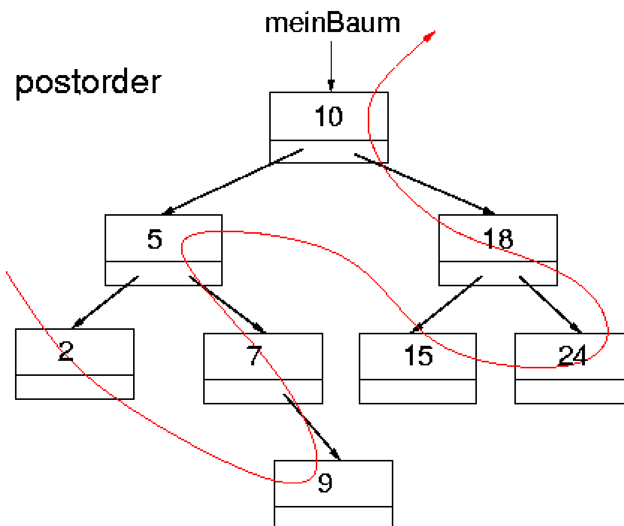
- Ausgabe des Beispielbaums: 2 5 7 9 10 15 18 24

○ preorder:
  - zuerst Wurzel, dann linken Teilbaum, dann rechten Teilbaum



- Ausgabe des Beispielbaums 10 5 2 7 9 18 15 24

○ postorder:
  - zuerst linken Teilbaum, dann rechten Teilbaum, dann Wurzel



- Ausgabe des Beispielbaums 2 9 7 5 15 24 18 10

- Implementierung der Baumausgabe in Reihenfolge inorder:
  - ausdrucken(BaumKnoten b) druckt Teilbaum ab b
  - benutzt Rekursion, um Unterbäume links und rechts auszudrucken
    -

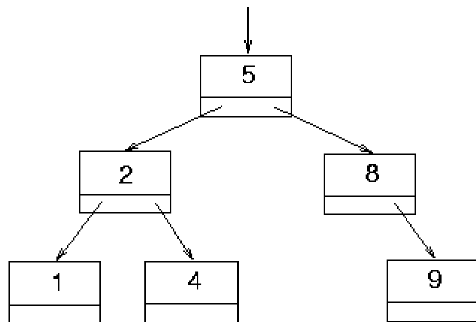
```
ausdrucken(BaumKnoten b) {
  // druckt ganzen Unterbaum ab b
  if (b.links != null) {
    ausdrucken(b.links);
  }
  System.out.print(daten);
  if (b.rechts != null) {
    ausdrucken(b.rechts);
  }
}
```

  - Blatt → beide Zeiger null → Abbruch der Rekursion
  - bequemes Ausdrucken des ganzen Baums mit
    -

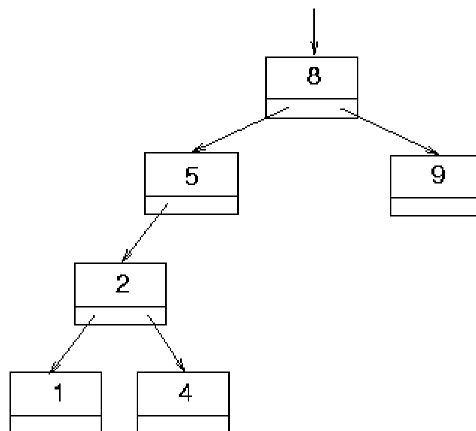
```
public void ausdrucken() {
    if (wurzel != null) {
      ausdrucken(wurzel);
    } else {
      System.out.println("<leerer Baum>");
    }
  }
```

- Sortieren mit Binärbäumen:
  - Sortierbaum: links von einem Knoten nur kleinere Werte, rechts nur größere
  - Aufbau eines Sortierbaums
    - Elemente nacheinander einfügen

    - an jedem Knoten Größenvergleich mit dortigem Wert
    - abhängig vom Ergebnis nach links oder rechts weitergehen
    - unten angekommen → neues Blatt anfügen
  - Beispieldaten 5 2 1 8 9 4 liefern



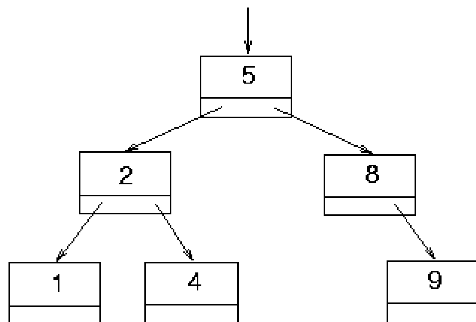
  - Implementierung des Einfügens rekursiv, ähnlich zum Ausdrucken
  - Beispielprogramm Baum.java
  - sehr schnelles Suchen (O(log n)) bei einigermaßen symmetrischem Baum
  - Problem
    - sich aufbauender Baum kann sehr unsymmetrisch werden
    - Suchen im Extremfall wieder O(n)
    - etwa für Beispieldaten 8 5 2 1 9 4



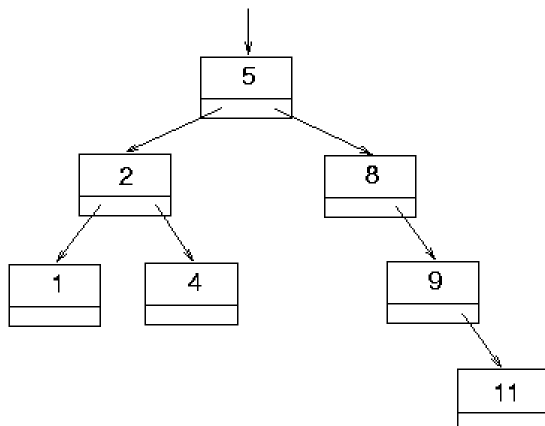
- Aufgaben:
  - Aufgabe 5

Ausgeglichene Bäume

- Verfahren zum Ausgleichen von Bäumen:
    - Ziel: Höhe wächst mit O(log(n))
    - zwei Grundverfahren
        - Umbauen der Zweige beim Einsortieren (z.B. AVL-Bäume)
        - zusätzliche Information im Knoten, Knoten mit mehr Zeigern (z.B. B-Bäume)
    - vielfältige Anwendungen, z.B. Indizierung von Datenbanken
- AVL-Bäume:
    - binärer Suchbaum
    - Höhe von linkem und rechtem Zweig jedes Knotens höchstens um 1 verschieden
    - benannt nach G. M. Adelson-Velskii und E. M. Landis
    - einfache Operationen (wie beim normalen Binärbaum)
        - Ausgeben des Baums
        - Suchen nach einem Element
    - komplizierte Operationen
        - Einfügen eines Elements (s.u.)
        - Löschen eines Elements (ähnlich zum Einfügen)
    - Beispiel für AVL-Baum

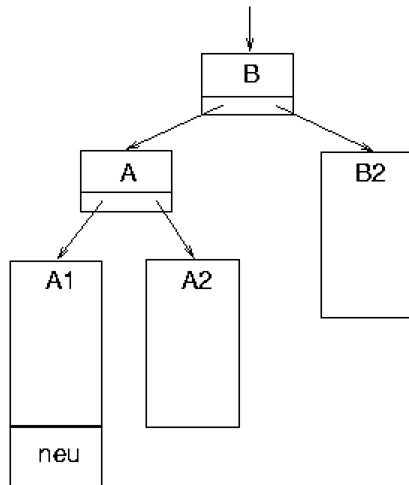


    - kein AVL-Baum

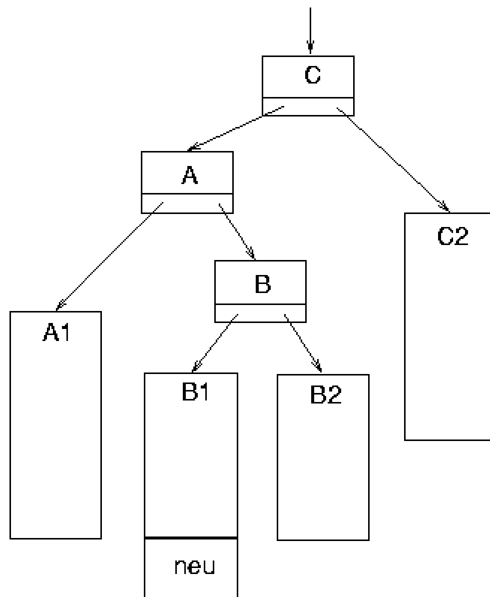


- Einfügen eines neuen Elements:
    - zunächst wie beim Sortierbaum einhängen

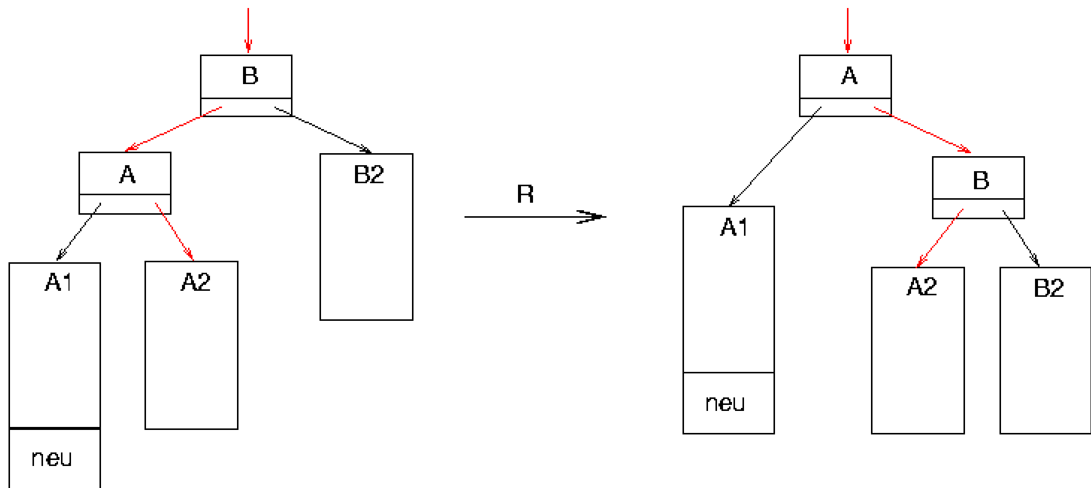
- vom neuen Element aus nach oben unbalanzierten Knoten (Differenz > 1) suchen
- zwei Möglichkeiten
  - zu tiefer Teil ist über links-links oder rechts-rechts zu erreichen ("außen schwer")



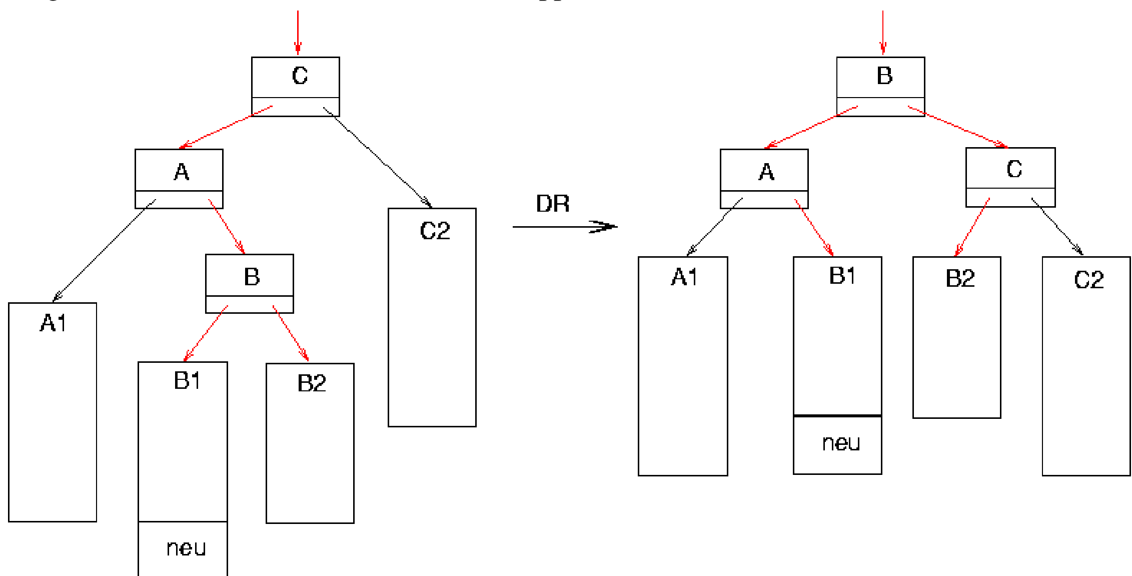
  - zu tiefer Teil ist über links-rechts oder rechts-links zu erreichen ("innen schwer")



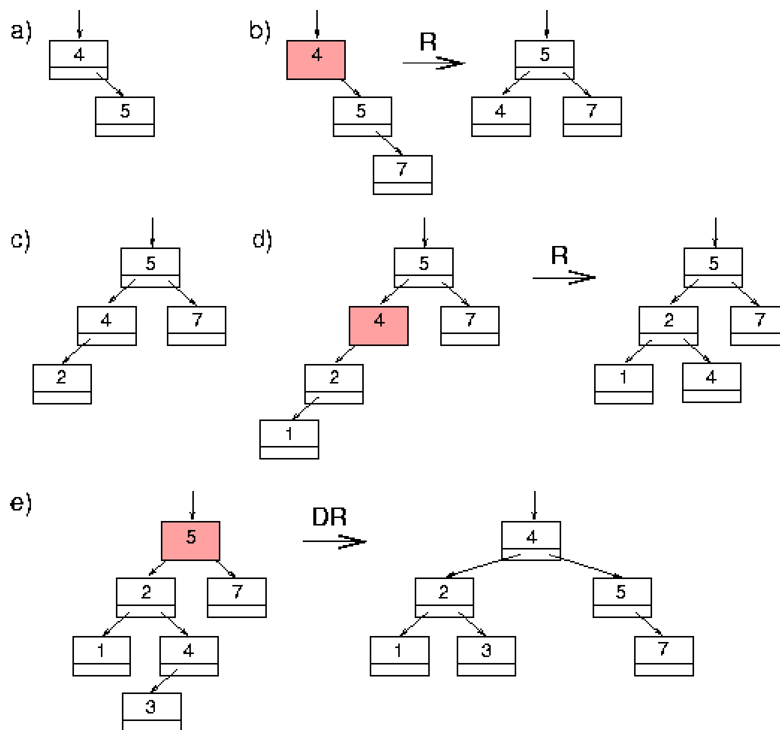
- Ausgleich bei außen schwerem Teilbaum ("einfaches Rotieren" R)



- Ausgleich bei innen schwerem Teilbaum ("doppeltes Rotieren" DR)



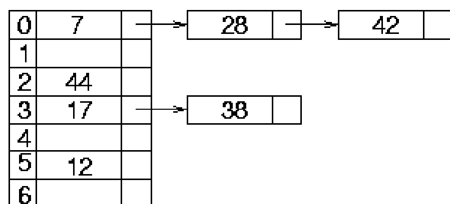
- Beispielfolge 4 5 7 2 1 3



- Implementierung eines AVL-Baums:
  - AVLKnoten
    - enthält zusätzlich Balance = h(rechts) - h(links)
  - AVLBaum
    - toString, suchen etc. wie beim Sortierbaum
  - Methoden rotiere und rotiereDoppelt
    - jeweils in zwei Versionen (linksrum, rechtsrum)
    - verbiegen nur Zeiger
    - bestimmen neue balance-Werte der veränderten Knoten
    - geben Zeiger auf das neue Wurzel-Element zurück
  - Methode einfuegen
    - sucht rekursiv geeignete Stelle und fügt unten neues Blatt an
    - rekursive Aufrufe setzen rebalance, wenn Ungleichgewicht auftreten könnte
    - rebalance gesetzt → Ausgleichen
    - Ausgleich durch Rotation oder einfach durch Korrektur von balance-Werten
- Aufgaben:
  - Aufgabe 6

Hashing und assoziative Arrays

- Grundprinzip:
  - Speicherung grundsätzlich in einem Feld fester Größe N
  - Hashfunktion h berechnet zu einem Schlüssel e den Indexwert h(e) im Array
  - Anforderungen an h
    - verschiedene Elemente haben möglichst verschiedenen Index
    - möglichst gleichmäßige Verteilung der Werte auf den Indexbereich
  - besondere Maßnahmen treffen bei verschiedenenen Schlüsseln mit gleichem Index (Kollision)
- Hashfunktionen:
  - für Integer-Schlüssel am einfachsten
    - h(i) = i % N
    - N am besten Primzahl!
  - alternativ mehr streuende Funktionen, etwa Polynome
  - bei double als Schlüssel z.B.
    - Exponent und Mantisse addieren und dann Hashfunktion für Integer
    - Zerlegung des Bitmusters in mehrere Integer, addieren, hashen
  - bei Strings
    - ASCII/Unicode-Werte der einzelnen Zeichen als Grundlage
    - daraus einen Integerwert ableiten (addieren, Polynom) und hashen
  - Hash-Methode in java.lang.Object
    - int hashCode()
    - liefert int-Hashwert für beliebiges Object
    - mit modulo an N anpassen
    - basiert in der Regel auf interner Speicheradresse
    - kann überschrieben werden durch an Daten angepasste Funktion (z.B. bei String)
- Kollisionsbehandlung:
  - Problem: Feld für neuen Schlüsselwert ist schon belegt (mit anderem Schlüssel)
  - Lösungsmöglichkeit: an Index-Position verkettete Liste der Elemente ansetzen



  - Alternative:
    - neuen Platz suchen (Sondieren)
    - muss ggf. mehrmals probiert werden
  - Sondierungsfunktion
    - liefert neue Schlüssel i_k, falls alte (i_0, i_1, ..) besetzt
    - direkt daneben: $i_k = i_0 + k$
    - linear: $i_k = i_0 + k*step$

        - quadratisch: $i_k = i_0 + k^2$
        - jeweils modulo N zu nehmen
- Beispiel:
    - Eingabewerte 84, 2, 68, 38, 83, 50, 71
    - Integer-Schlüssel mit einfacher Modulo-Hashfunktion
    - Sondierungsfunktion in Einer-Schritten

0	
1	
2	2
3	68
4	
5	38
6	83
7	84
8	50
9	71
10	

        - Bildung von dichtbesetzten Haufen
        - 7 Kollisionen
    - quadratische Sondierungsfunktion

0	
1	
2	2
3	68
4	
5	38
6	83
7	84
8	
9	71
10	50

        - bessere Verteilung der Werte
        - 5 Kollisionen
- Aufwand beim Hashing:
    - stark abhängig vom Füllgrad
    - abhängig von Güte der hash- und Sondierungsfunktionen
    - typische Anzahl S von Suchschritten bei linearem Sondieren [Wirth]

Füllgrad	**S**
0.1	1.06
0.25	1.17
0.5	1.50
0.75	2.50
0.9	5.50
0.95	10.50

- Vorteile gegenüber anderen Suchverfahren
    - bei nicht zu hohem Füllgrad Einfügen und Aufsuchen in konstanter Zeit!
- Nachteile
    - Probleme bei hohem Füllgrad bzw. unbekannter Anzahl auftretender Werte
    - statisch vorgegebene Array-Größe
    - u.U. Schwierigkeiten der Hash-Funktion bei speziellen Daten

- Assoziatives Array (Map):
    - Zuordnung von Schlüsseln und Werten, jeweils von beliebigem Typ
    - vorstellbar als "Array mit nicht-ganzzahligem Index"
    - z.B. Tabelle von Benutzernamen und Passwörtern (in Perl-Syntax)
        - tabelle{"peter"} = "programmieren";
tabelle{"thomas"} = "schweißen";
tabelle{"gaby"} = "beweisen";
print tabelle{"thomas"}; // liefert "schweißen"
    - häufig als Hashtabelle implementiert
    - in Java java.util.HashMap mit (u.a.) folgenden Methoden
        - Object put(Object key, Object value)
legt value unter Schlüssel key ab
        - Object get(Object key)
liefert Wert zum Schlüssel key
        - String toString()
erzeugt Ausgabe-String mit allen Werten in der Form
{key1=wert1, key2=wert2, ...}
- Aufgaben:
    - Aufgabe 7
    - Aufgabe 8

Software-Entwicklung mit Design Patterns

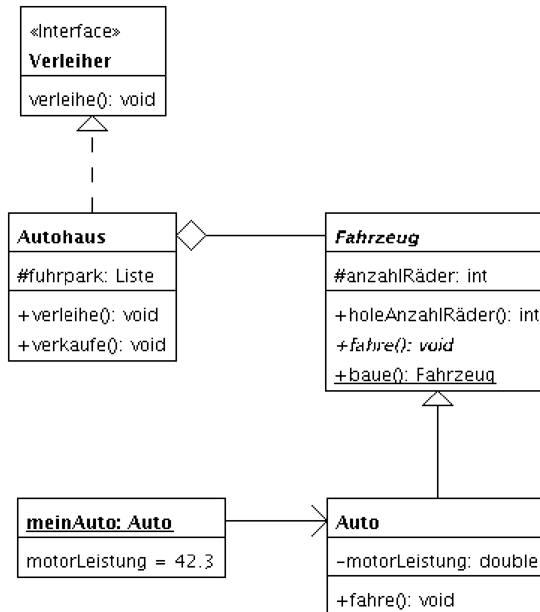
- Einführung
- Entwicklung eines UML-Editors
- Zerlegung in Teile
- Aufzählungstypen
- Durchlaufen von Sammlungen
- Zentrale Kontrollinstanz
- Speichern von Objekten
- Graphische Darstellung der Elemente
- Bearbeiten der UML-Elemente
- Relationen zwischen UML-Elementen

Einführung

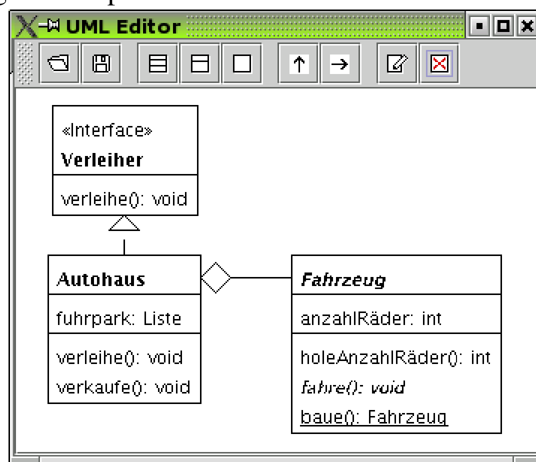
- Design Pattern:
  - allgemeine Formulierung der Lösung eines objektorientierten Design-Problems
  - Versuch, vorhandenes Wissen guter Programmierung zu erfassen
  - Bestandteile
    - Name, erlaubt die Identifizierung und Kommunikation
    - Problem, mit Kontext und Anforderungen
    - Lösung, enthält in in abstrakter Form die Elemente und ihre Beziehungen
    - Konsequenzen, Vor- und Nachteile des Patterns
  - wachsende Listen von Pattern in Literatur und Internet ([Gamma], [Buschmann], [4])
- Eigenschaften von Design Patterns:
  - beschreiben in der Regel Zusammenarbeit mehrerer Klassen
  - unabhängig von einer Programmiersprache
  - Versuche zu einer Standardisierung der Darstellung
  - Beschreibung häufig unterstützt durch UML-Diagramme
- Einteilung in Pattern-Typen:
  - Architectural
    - grundlegende Organisation eines Programms
    - Beispiel: Model-View-Controller
  - Structural
    - Zusammensetzung von Klassen und Objekten
    - Beispiel: Whole-Part, Adapter
  - Creational
    - Erzeugung von Objekten
    - Beispiel: Singleton, FactoryMethod
  - Behavioural
    - Zusammenarbeit von Klassen oder Objekten
    - Beispiel: Iterator, Observer
  - Idiom
    - Implementierung einfacher Design-Probleme
    - häufig sprachabhängig
    - Beispiel: Enumeration, Lazy Initialisation

Entwicklung eines UML-Editors

- Ziel:
  - Entwicklung eines Editors zur Erstellung von UML-Diagrammen
  - einfache Grundversion
  - leicht wartbar und erweiterbar
- Elemente von UML:
  - UML = Unified Modeling Language
  - graphische Notation zur Beschreibung von Software-Modellen
  - Quasistandard der Object Management Group [5]
  - enthält diverse Diagrammarten
  - i.f. nur Klassendiagramme
  - typisches Diagramm mit wichtigsten Elementen

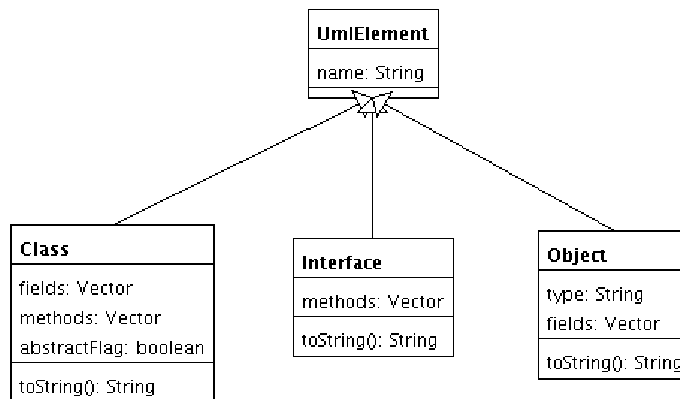


  - Klasse
    - Rechteck mit Bereichen für Namen, Datenfelder und Methoden
    - alles außer Name optional
    - Datenfelder als Name:Typ
    - Methoden ebenso, ggf. mit Parameterliste
    - +, -, #, (nichts) für Zugriffsklassen public, private, protected, default
    - unterstrichen bei static
    - kursiv bei abstrakten Methoden oder Klasse
  - Interface
    - zusätzlich Bezeichnung "Interface"
    - keine Felder
    - nicht kursiv
  - Objekt

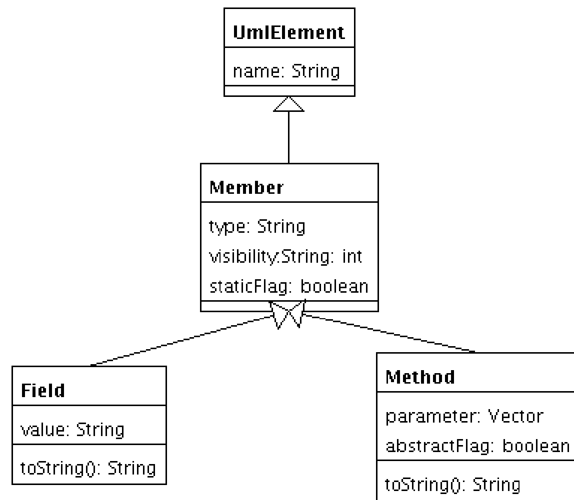
        - Felder und Werte dafür
    - Beziehungen zwischen Elementen
        - Assoziation: einfache Linie
        - Navigierbarkeit: einfacher Pfeil in Richtung der Navigierbarkeit
        - Abhängigkeit: gestrichelter einfacher Pfeil zum unabhängigen Element
        - Aggregation: Linie mit Raute am Anfang vom Ganzem zum Teil
        - Vererbung: großer, nicht ausgefüllter Pfeil zur Oberklasse
        - Implementierung: wie Vererbung, aber Pfeil gestrichelt
- Pflichtenheft des UML-Editors:
    - Darstellung von Klassen, Interfaces und Objekten
    - Elemente verbindbar durch Pfeile wie oben
    - Erzeugung und Löschen von Elementen
    - Elemente in allen Teilen editierbar
    - Elemente incl. Pfeilen verschiebbar
    - Diagramme speicherbar und ladbar
        - 

- Mögliche Erweiterungen:
    - komfortablere Pfeile
        - vielseitigere Andockpunkte
        - automatisches Mitverschieben an die günstigste Ecke
        - Zusammenführen mehrerer Pfeile an Verteilerpunkten
    - Hauptmenü und Popup-Menüs
    - weitere Elemente
    - weitere Diagramm-Typen
    - Aufbohren zu einem Analyse- und Design-Werkzeug (vgl. [6])

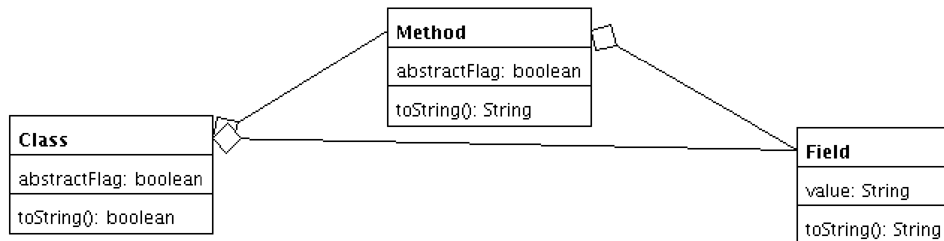
Zerlegung in Teile

- Grundobjekte in betrachteten UML-Diagrammen:
  - liefern Klassen
    - Class
    - Interface
    - Object
  - werden abgeleitet von gemeinsamer Oberklasse UmlElement
  - Grundstruktur
    - 

  - zusätzlich Methoden zum Zugriff auf die Felder (setName, getName)
  - Alternative für Object
    - type zeigt auf eine Class
    - braucht dann keine eigene Feldliste
    - Vorteil: automatischer Abgleich mit seiner Class
    - Nachteil: Objekt nur möglich mit entsprechender Class
- Datenfelder und Methoden:
  - Grundidee: sind selbst Objekte entsprechender Klassen (Field, Method)
  - Gemeinsamkeiten herausgelöst in Oberklasse Member
  - Grundstruktur



- Member abgeleitet von UmlElement (wie Class etc.) wegen Gemeinsamkeiten
  - hat Feld name
  - wird im Diagramm gezeichnet
  - kann editiert werden
  - aber: ist kein selbständiges Objekt im UML-Diagramm
- Method nutzt für die Parameterliste selbst Field als Teile
  - Vorteil des Herausgliederns
  - Class mehrstufig zergliedert



- Implementierung Field:
  - nur langweilige Zugriffsmethoden

```
public String getValue() {
  return value;}public void setValue(String newValue) {  value = newValue;}
```

  - typisch für Klassen, die i.w. Daten enthalten
  - Property
    - Datenfeld mit set- und get-Methode
    - von Entwicklungsumgebungen unterstützt
- Pattern Whole-Part:
  - Pattern-Typ Structural
  - Problem
    - ein Objekt (Whole) besteht aus einer Anzahl verschiedener Teile (Parts)

    - Teile sind selbst recht kompliziert
- Lösung
    - modelliere Teile als eigene Klassen
    - füge ggf. Liste von Parts in Whole ein
    - Whole kapselt die Parts nach außen, falls sinnvoll
- Konsequenzen
    - Teile können in mehreren Klassen verwendet werden (+)
    - Komplexität der Whole-Klasse sinkt (+)
    - Zahl der Klassen steigt an (-)

Aufzählungstypen

- Typ der Zugriffsklasse (Feld visibility):
  - kann 4 Werte annehmen (public, protected, private, default)
  - als String
    - einfach
    - beliebige Werte möglich
    - kein Check
  - als int
    - Werte 0 .. 3 durch Konstruktor erzwungen
    - Bedeutung der Werte nicht festgelegt, beliebig
    - muss woanders festgelegt werden
  - in Pascal als eigener Datentyp (Aufzählungstyp)
    - TYPE Visibility = (public, protected, private, default)
    - geplant für Java 1.5
- Anforderungen an Feld visibility:
  - keine Mischung mit anderen Typen (int, String o.ä.)
  - feste Menge von Werten
  - Ausgabe-Möglichkeit
- Lösung in Java:
  - Klasse Visibility (= eigener Datentyp)
  - Werte als vorgegebene Objekte, static und final:
    - public static final Visibility PRIVATE = new Visibility("private");
  - Konstruktor ist private → keine neuen Werte erzeugbar
  - toString-Methode zur Ausgabe
    - internes Feld name zur Bezeichnung
    - wird im Konstruktor gesetzt
  - kleines Extra
    - komplette Liste der Werte mit getVisibilities()
    - praktisch z.B. für Auswahlboxen
- Enumeration-Pattern:
  - Pattern-Typ Idiom (Java)
  - Problem
    - Feld soll nur Werte aus fester Liste haben
    - Werte sollen auch ausgegeben werden können
  - Lösung
    - eigene Klasse für die Werte
    - privater Konstruktor mit Ausgabe-String als Parameter
    - Werte als Objekte vom Typ static final
  - Konsequenzen
    - typsichere Verwendung der vorgegebenen Werte (+)
    - Ausgabe als Strings in natürlicher Weise (+)
    - leichter Overhead gegenüber einfachen Integern (-)

    - schwierigere Serialisierung (s.u.) (-)
- Erweiterungen
    - Möglichkeiten zum automatischen Durchlaufen der Werte [7]
    - Abspeichern mit einfacher Serialisierung

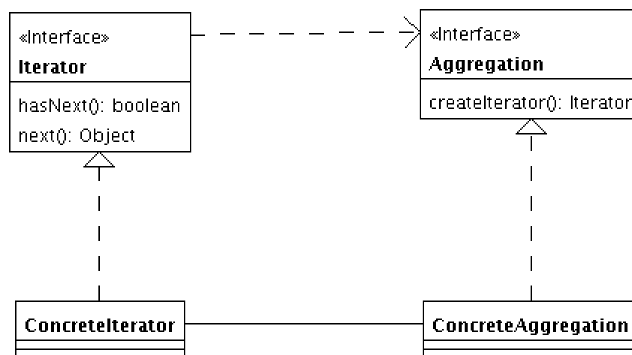
Durchlaufen von Sammlungen

- Textausgabe einer Klasse:
  - toString-Methode
    - erzeugt Textdarstellung in Java-ähnlicher Form
    - verwendet toString von Method und Field für die Teile
    - muss Listen (Vector) von Fields und Methods durchlaufen
  - Durchlaufen abhängig von interner Struktur von Vector
    - bei Array einfach Index hochzählen
    - bei verketteter Liste Zeiger verfolgen
    - bei Baum Zeiger rekursiv durchgehen, verschiedene Reihenfolgen möglich
  - Lösung
    - Durchlauf-Steuerung ist eigene Klasse (Iterator)
    - speichert interne Zeiger, Index o.ä.
    - hat Methoden next() zum Weitergehen und hasNext() zum Beenden
    - wird von einer Sammlung (Collection) wie Vector, List etc. bereitgestellt
  - Implementierung des Feld-Durchlaufs in Class.toString()
    -

```
for (Iterator i = parameters.iterator(); i.hasNext(); ) {
  Field f = (Field) i.next();
  s += f; // ruft implizit f.toString() auf!
}
```

- Pattern Iterator:

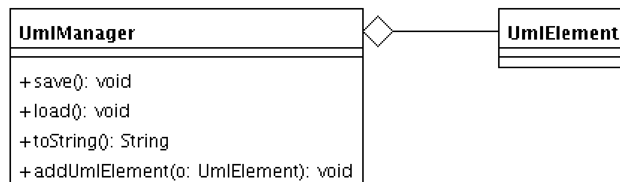
  - Pattern-Typ Behavioural
  - Problem
    - Durchlaufen einer Sammlung von Objekten, ohne die innere Struktur der Sammlung zu kennen
  - Lösung
    - eigene Klasse Iterator zum Durchlaufen
    - Sammlung erzeugt geeignetes Iterator-Objekt
    - Iterator-Objekt kapselt interne Struktur der Sammlung

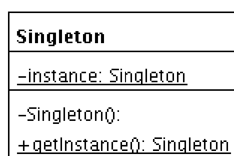


    -
  - Konsequenzen

- vereinfacht die Samlungsklasse (+)
- ermöglicht verschiedene Arten des Durchlaufens (post-, pre-order) (+)
- erlaubt mehrere gleichzeitige Durchläufe (+)
- Overhead gegenüber Index oder Zeiger bei einfachen Sammlungen (-)

Zentrale Kontrollinstanz

- Klasse UMLManager:
  - Oberaufsicht für alle Bestandteile des Diagramms
  - kennt alle Teile
  - zuständig für globale Operationen, wie
    - Laden eines Diagramms
    - Speichern eines Diagramms
    - Textuelle (später graphische) Darstellung des Diagramms
  - Grundstruktur
    - 

  - darf nur einmal vorkommen
- Pattern Singleton:
  - Pattern-Typ Creational
  - Problem
    - von einer Klasse soll genau ein Objekt vorhanden sein
    - auf dieses Objekt kann global zugegriffen werden
  - Lösung
    - Konstruktor ist private
    - ein Objekt ist statisches Element der Klasse
    - öffentliche statische Zugriffsmethode auf das Element
    - 

  - Konsequenzen
    - Kontrolle trotz globalen Zugriffs zentralisiert (+)
    - leicht erweiterbar auf genau N Instanzen (+)
    - Zeitpunkt der Objekt-Entstehung u.U. kritisch (-)
- Pattern Lazy Initialisation:
  - Patterntyp Idiom
  - Problem
    - unklar, wann eine Größe initialisiert werden soll
  - Lösung
    - da initialisieren, wo sie gebraucht wird
    - aber nicht jedes Mal neu, nur dann, wenn es nötig ist ("lazy")
  - Beispiel-Code in UmlManager

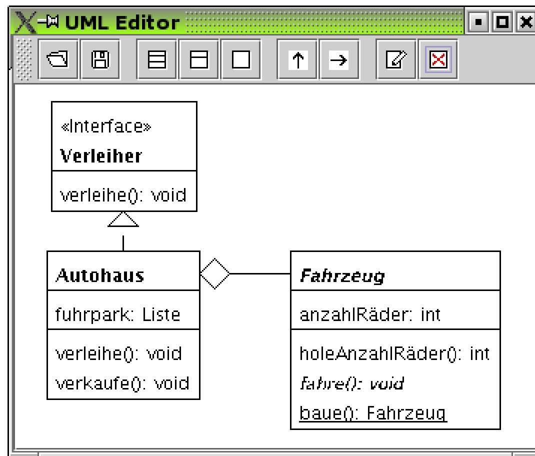
```
private static UmlManager theManager;

public static UmlManager getUmlManager() {
  if (theManager == null) {
    theManager = new UmlManager();
  }
  return theManager;
}
```

Speichern von Objekten

- Serialisierung:
  - Java-Verfahren, um Objekt in Bytemuster zu verwandeln und zurück
  - Anwendungen
    - Abspeichern eines Objekts auf Platte
    - Übertragung eines Objekts über ein Netzwerk
  - Interface java.io.Serializable
    - keine Methoden oder Felder (Marker-Interface)
    - Klassen implementieren Serializable → können serialisiert werden
    - schreibt rekursiv auch alle Unterobjekte (falls Serializable)
    - Ausnahme: statische Felder
  - nicht gewünschte Felder (temporäre oder geheime Daten) mit Modifizierer transient bezeichnen
  - schreiben in Datei mit ObjectOutputStream
    -
      ```
      FileOutputStream fOut = new FileOutputStream("objekte1.out");
      ObjectOutputStream out = new ObjectOutputStream(fOut);
      out.writeObject("juhu");
      out.writeObject(new Integer(4));
      ```
  - lesen aus Datei mit ObjectInputStream
    -
      ```
      FileInputStream fIn = new FileInputStream("objekte1.out");
      ObjectInputStream in = new ObjectInputStream(fIn);
      String s = (String) in.readObject();
      Integer d = (Integer) in.readObject();
      ```
  - Rekonstruktion eines Objekts
    - direkt aus Bytemuster
    - ohne Aufruf eines Konstruktors
- Laden und Speichern:
  - über Serialisierung →
    - Format zum Ablegen von Objekten in Java festgelegt
  - explizit vom UmlManager
    - durch Methoden save/load
    - schreiben/lesen nur umlElements über ObjectStream
  - Klassen der zu speichernden Objekte (incl. Unterobjekten!) sind Serializable
    - ArrayList (Liste umlElements)
    - UmlElement (automatisch auch seine Kinder)
    - Typen von deren Datenfeldern (String, boolean, Vector, Visibility)
- Serialisierung bei Visibility:
  - entscheidende Größen sind statisch
  - manuelle Serialisierung nötig
    - schreibe Methoden writeObject/readObject

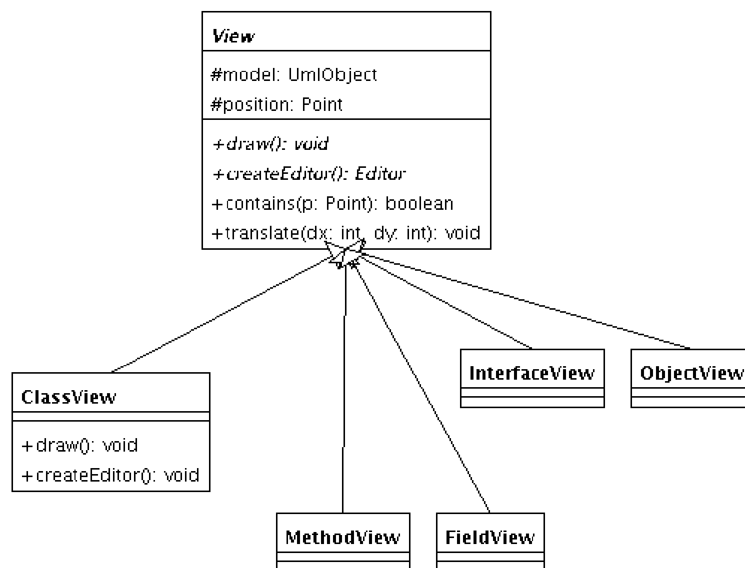
        - Schnittstelle genau vorgegeben (incl. private und throws)
        - verwendet default-Methoden für Standard-Serialisierung
        - schreibt/liest statische Felder explizit
    - Felder können nicht mehr final sein wegen Zuweisung in readObject!
- Prototyp ohne graphische Oberfläche:
    - Test-Programm TestUMLManager mit einfachem main
        - erzeugt "UML-Diagramm" durch explizite Aufrufe
        - gibt es (textuell) aus und speichert es ab
        - erzeugt neues einfaches UML-Diagramm
        - lädt altes Diagramm und gibt es aus
    - Test der bisherigen Konzepte ohne die Komplexität einer GUI
    - vereinfachte Version (ohne Laden/Speichern) als Applet

Graphische Darstellung der Elemente

- GUI für UML-Editor:
  - Gesamtansicht
    - 

  - Zeichenbereich mit UML-Diagramm
  - oben Leiste mit Buttons für Funktionen ("Toolbar"):
    - Laden, Speichern
    - neue Class, Interface, Object
    - neuer Pfeil (vertikal, horizontal)
    - Editieren eines Objekts
    - Löschen eines Objekts
  - Anklicken bei "neuer Pfeil"/"Editieren"/"Löschen"
    - Cursor verändert sich
    - danach entsprechendes Objekt (oder zwei für Pfeil) anklicken
    - leerer Platz auf Zeichenfläche angeklickt → Funktion abgebrochen
  - Editieren
    - Fenster mit Maske für entsprechendes Objekt geht auf
    - enthält Buttons "Abbrechen" und "Übernehmen"
- Pattern Model-View-Controller:
  - Pattern-Typ Architectural
  - Problem
    - Anforderungen an Benutzer-Interfaces und graphische Darstellungen ändern sich häufig
    - mehrere Darstellungs- oder Kontrollmöglichkeiten für ein Objekt
  - Lösung
    - Aufteilung eines Objekttyps in drei Klassen
    - Model: enthält Kerndaten und -funktionen, aber keine UI-Anteile
    - View: zuständig für graphische Darstellung
    - Controller: zuständig für Benutzereingaben zur Änderung des Objekts
    - Views (ggf. auch Controller) erhalten Änderungs-Meldungen vom Model über Observer-Pattern
  - Konsequenzen

        - graphische Darstellung und Kontrollen leicht änderbar (+)
        - mehrere verschiedene Views gleichzeitig möglich (+)
        - Views immer konsistent (da auf das Model-Objekt bezogen) (+)
        - View und Controller sehr eng verknüpft (-)
        - View und Controller stark an Model gekoppelt (-)
        - u.U. verschlechterte Performance (viele Updates) (-)
    - Variante Document-View
        - Document $\hat{=}$ Model
        - View $\hat{=}$ View + Controller
        - beide Richtungen der Oberfläche (Darstellung und Interaktion) in einem Element
        - Standard-Pattern bei Swing-Komponenten
- Anwendung des MVC-Patterns beim UML-Editor:
    - Model (Fachklassen)
        - Top-Level-Elemente (Class, Interface, Object)
        - auch Part-Objekte Field, Method
        - keine Änderungen für GUI nötig
    - View
        - enthält bzw. berechnet Infos wie Position und Größe
        - enthält Referenz auf das Model
        - enthält Methoden zum Zeichnen oder Verschieben

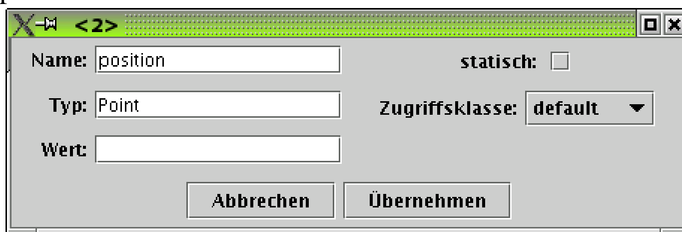


        -
    - Views auch für Part-Elemente
        - Zeichnen der Unterkomponenten von ClassView an diese delegiert
        - verschiedene graphische Darstellungen mit mehreren FieldView's
        - z.B. FieldView mit Zeilenumbruch
    - Controller
        - enthält Referenz auf das Model
        - eigenes Fenster = Editor

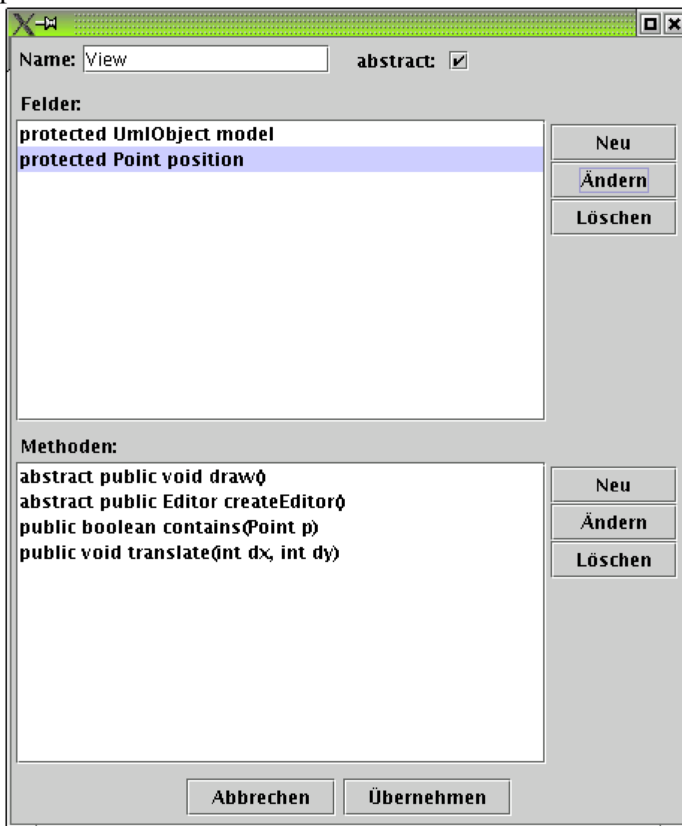
        - View und Controller hier kaum gekoppelt
- Erläuterungen zur Implementierung:
    - View
        - enthält einfache, allen gemeinsame Methoden
        - stellt benötigte Fonts bereit (lazy initialisiert)
    - ClassView
        - implementiert die draw-Routine → Darstellung gemäß UML-Standard
        - verwendet draw der Part-Elemente (Felder und Methoden)
        - berechnet Größe des Darstellungsrechtecks aus String-Größen
        - Breite = Maximum aller Stringbreiten (gemäß Font) + Rand
    - ähnlich InterfaceView, ObjectView, MethodView, FieldView
    - UmlGui
        - kann als Applet oder Applikation laufen (aber als Applet nicht laden/speichern!)
        - enthält Liste aller Views
        - erzeugt Iconleiste als JToolBar
        - fügt Icons hinzu als JButtons, incl. Tooltip-Text und Listenern bei Anklicken
        - bereitet für load/save JFileChooser (Datei-Auswahlbox) vor
        - enthält Zeichenfläche als Unterklasse DisplayPanel
        - implementiert Verschieben der Views über Maus-Listener

Bearbeiten der UML-Elemente

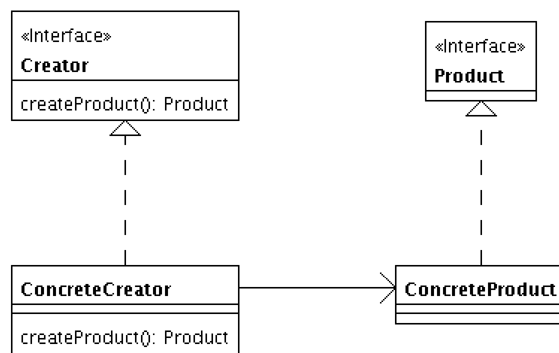
- Editor:
    - zeigt Felder eines UmlElements an
    - erlaubt Editieren aller Felder
    - ruft ggf. Editor für Part-Elemente auf
    - wird beendet mit
        - "Abbrechen" → alte Werte bleiben
        - "Übernehmen" → update() des bearbeiteten Objekts
    - Beispiel FieldEditor

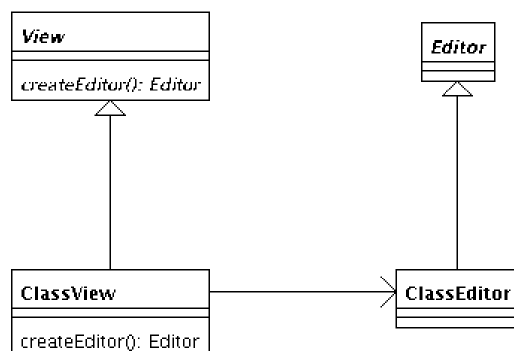


- Beispiel ClassEditor

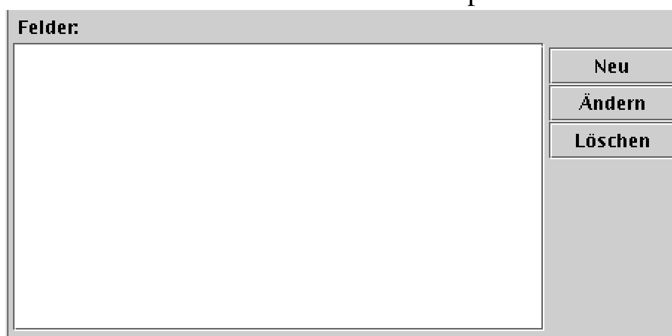


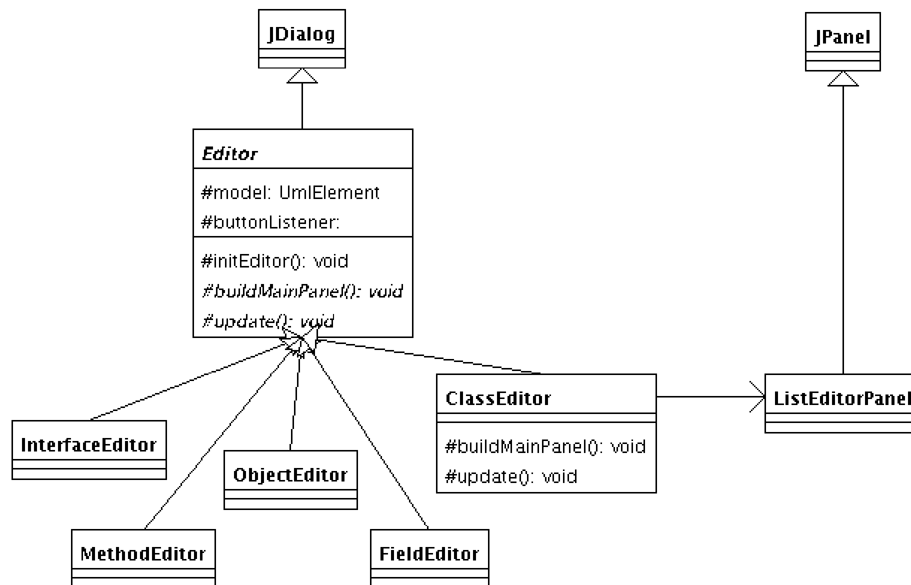
- Erzeugen eines Editors:
    - Benutzer klickt Edit-Button an, danach das zu editierende Objekt
    - DisplayMouseListener von UmlGui

        - findet getroffenen View hitObjectView
        - ruft hitObjectView.createEditor() auf
    - Problem: wie wird der richtige Editor (ClassEditor, InterfaceEditor etc.) gefunden?
    - Lösung mit Hilfe von Polymorphismus
- Pattern FactoryMethod
    - Pattern-Typ Creational
    - Problem
        - ein neues Objekt soll erzeugt werden
        - nur Oberklasse oder Interface ist explizit gegeben
        - genauer Typ des Objekts ist erst zur Laufzeit festgelegt
    - Lösung
        - Erzeuger (Creator) ist ein Interface oder eine abstrakte Klasse
        - Erzeugerfunktion liefert einen abstrakten Obertyp (Product)
        - zur Laufzeit sind der konkrete Erzeugertyp und damit auch das erzeugte Objekt gegeben
        - 

    - Konsequenzen
        - viele Teile des Codes benötigen nur die Oberklasse, nicht die (wechselnden) Details (+)
        - Erzeuger-Hierarchie ist u.U. zusätzlich nötig (-)
- Anwendung des Patterns:
    - mit abstrakten Basisklassen statt Interfaces, etwa bei angeklicktem ClassView:
        - 

    - Implementierung der abstrakten Erzeugerfunktion in ClassView einfach
        - return new ClassEditor((Class) m, p);
    - abstrakte Oberklasse Editor

        - erzeugt gemeinsame Elemente (Buttons und Panels) sowie Listener
        - überlässt Hauptfenster mit buildMainPanel den Kindern
        - überlasst Übertragen der Daten zum Model mit update den Kindern
    - konkrete Unterklasse ClassEditor
        - implementiert update, um Daten an das Model Class weiterzugeben
        - implementiert buildMainPanel, um seine Komponenten einzubauen
        - benutzt 2x ListEditorPanel als Teilkomponente
        - 

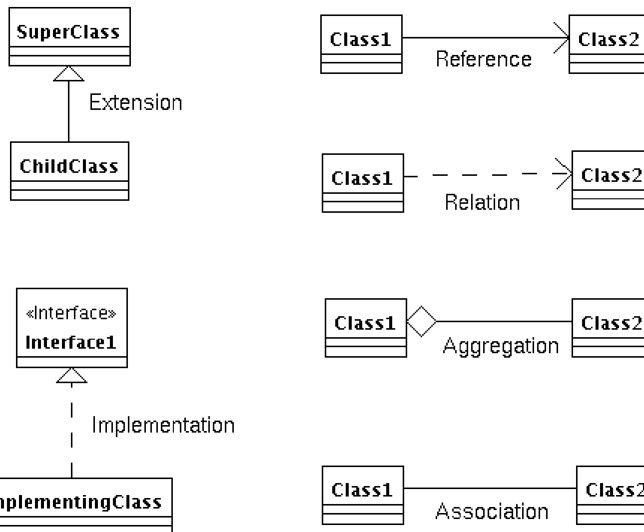
    - analog InterfaceEditor, ObjectEditor, MethodEditor, FieldEditor
        - 

- Klonen:
    - wegen Abbruchmöglichkeit Editieren nur an einer Kopie (Klon) möglich
    - Erzeugen eines Klons in Java
        - Interface Cloneable implementieren
        - Methode public Object clone() implementieren
        - super.clone() (in java.lang.Object) aufrufen (kopiert Elemente flach)
        - Referenzen der Unterobjekte selbst klonen
    - z.B. Klonen der Field-Liste eines Class-Objekts
        - Vektor von Field-Elementen klonen
        - ergibt neuen Vektor mit Referenzen auf originale Field-Objekte

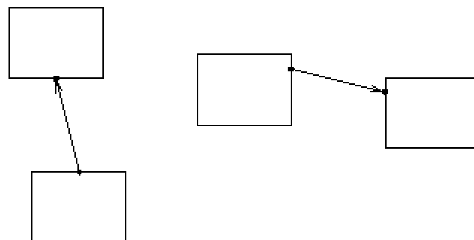
    - danach alle Field-Elemente klonen
- Problem bei ListEditorPanel:
  - wird für Fields und Methods benutzt
  - muss bei "Ändern" einen Editor aufrufen
  - bekommt nur Liste der UmlElements, wissen nichts über Editoren
  - Problem
    - Woher weiss das ListEditorPanel, welchen Editor es erzeugen soll?
  - Lösung
    - vom Aufrufer wird passender View elementView als Konstruktor-Parameter mitgegeben
    - elementView erzeugt passenden Editor
  - elementView wird auch benutzt, um bei "Add" Standard-Element des richtigen Typs zu erzeugen

Relationen zwischen UML-Elementen

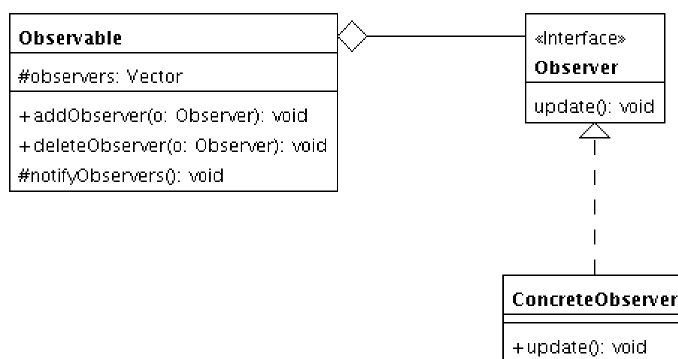
- Design der Pfeile im UML-Diagramm:
  - Beziehungen zwischen UML-Objekten:



  - mögliche Implementierungen
    - als eigene Liste der Pfeile
    - als spezielle UmlElement's
    - als Teil eines vorhandenen Objekts (z.B. Teil des Ausgangsobjekts)
  - passen gut in das MVC-Schema
  - interne Daten in Model-Klasse Arrow (Kind von UmlElement)
    - Sender-, Empfänger-Objekt
    - Art der Relation (ArrowType) als Enumeration
    - Name (über UmlElement)
  - Darstellung mit ArrowView
    - Linien- und Pfeiltyp abhängig vom ArrowType nach UML
    - Anfangs- und Endpunkte durch Start- und Endobjekt festgelegt

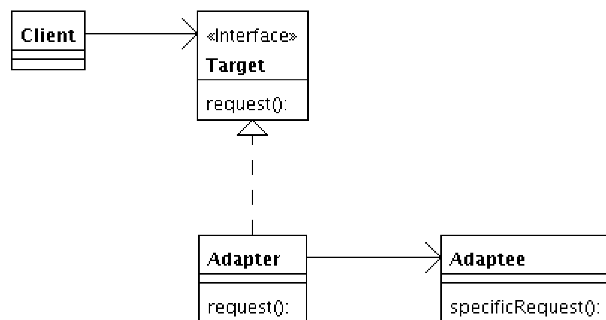


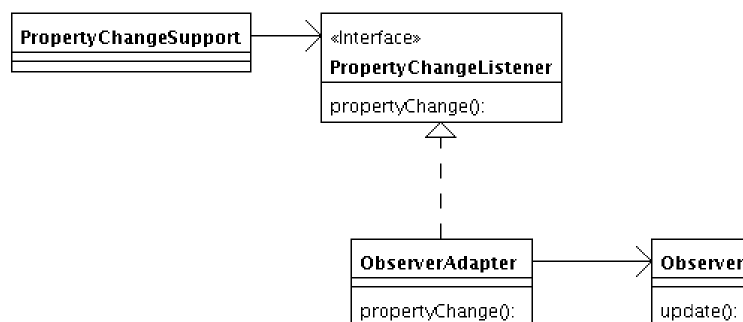
    - Besonderheit: Pfeile werden mit Start-/Endobjekt mitverschoben
    - Erweiterung: Name am Pfeil anzeigen
  - verändern mit ArrowEditor
    - Ändern von ArrowType und Namen
- Pattern Observer:

- Pattern-Typ Behavioural
- Problem
  - Objekt (Observable) soll seine Änderungen an beobachtende Objekte melden
  - Art und Anzahl der Beobachter unbekannt
- Lösung
  - Beobachter implementieren Interface Observer mit Methode update
  - Beobachter melden sich bei Observable an
  - beobachtetes Objekt hat Liste der Observer
  - ruft bei Änderung deren update-Methode auf
  - 

- Konsequenzen
  - beliebige Anzahl von Observern möglich (+)
  - Kopplung zwischen Observable und Observern sehr lose (+)
  - keine Information über Art der Änderung → u.U. unnötige Update-Kaskade (-)
- Bemerkungen
  - Teil des MVC-Patterns
  - bei Swing: Events und Listener
  - in java.util vorhanden

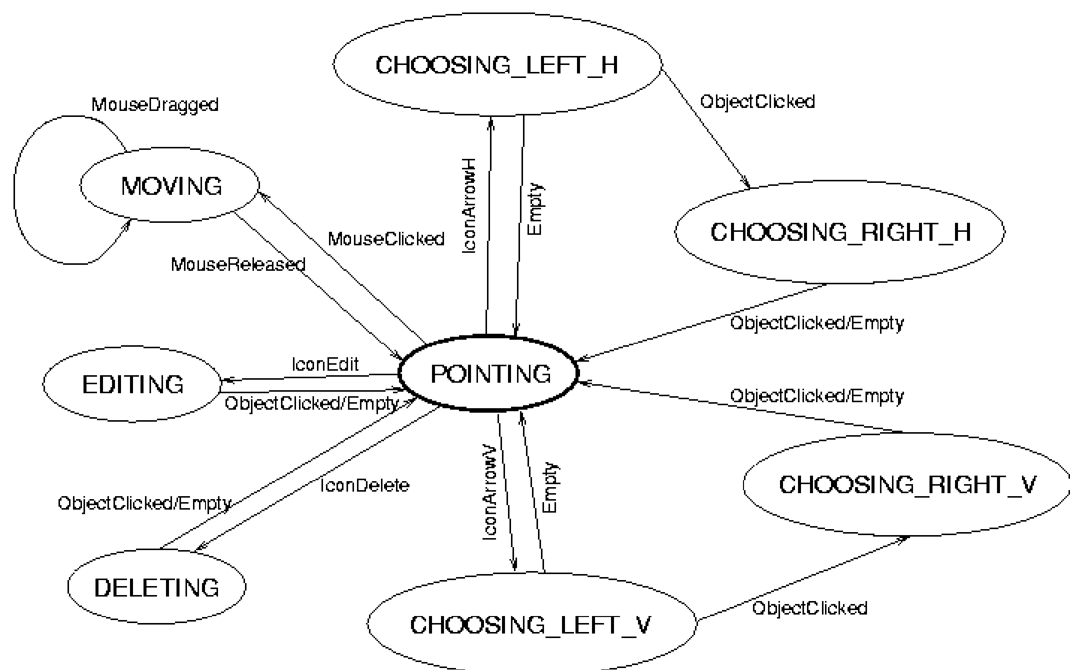
* Implementierung im UML-Editor:
  - ArrowView ist Observer (hat update-Methode)
  - View ist Observable
    - hat Liste der Observer
    - hat Methoden zum An- und Abmelden
    - hat Methode zum Informieren der Observer
  - Problem
    - Was passiert bei mehrfachem Anmelden eines Observers?
  - Unterschiede zu MVC
    - Observable ist hier der View
    - es geht nur ums Verschieben, nicht um das Model
* Alternative Implementierung über java.beans.PropertyChangeSupport:
  - Vorteil: schon vorhanden
  - Alternative java.util.Observable ist nicht serialisierbar
  - Schwierigkeit: View kann nicht von PropertyChangeSupport erben
  - Ausweg: View hat PropertyChangeSupport-Objekt und leitet Anfragen weiter

    - Methodennamen von PropertyChangeSupport werden dabei an Observable angepasst
    - wichtiger Trick: Wen ich nicht beerben kann, den muss ich fressen!
    - Problem: statt Observer wird PropertyChangeListener gebraucht
        - entweder ArrowView anpassen (update → propertyChange)
        - oder eleganter über Adapter-Pattern
- Pattern Adapter:
    - Pattern-Typ Structural
    - Problem
        - Interface einer vorhandenen Klasse (Adaptee) passt nicht zu dem Interface (Target), das eine Kundenklasse (Client) erwartet
    - Lösung
        - neue Adapterklasse (Adapter) hat richtiges Interface Target
        - übersetzt Aufrufe für Adaptee
        - 

    - Konsequenzen
        - eigentlich inkompatible Objekte können zusammenarbeiten (+)
        - u.U. schwierig, Parameter richtig zu übertragen (-)
        - Adapter u.U. nur teilweise Ersatz für Target (-)
    - konkrete Umsetzung mit ObserverAdapter
        - 

- Implementierung des Zeichnens:
    - verwendet einige Funktionen aus 2D-Graphik von Swing
    - zur Benutzung Standard-Graphikkontext einfach "umwidmen"
        - Graphics2D g2 = (Graphics2D) g;
    - gestrichelte Linien mit java.awt.BasicStroke
        - Konstruktor erhält Array mit Längen von Strichen
        - anwenden mit g2.setStroke(BasicStroke);

- beliebige Polygone mit Klasse java.awt.Polygon
  - Konstruktor mit Punkten als Arrays für x- und y-Koordinaten
  - zeichnen mit g2.draw (Umriss) oder g2.fill (ausgefüllt)
  - prüfen, ob Punkt im Polygon liegt, mit pol.contains(Point)
  - pol.invalidate() nach Änderungen an den Koordinaten wegen Caching
- Verwaltung von GUI-Zuständen:
  - verschiedene Aktionen beim Anklicken eines Objekts, je nach vorheriger Icon-Auswahl
  - normalerweise zweischrittig
    - Aktion auswählen (Icons "Edit", "Delete")
    - zugehöriges Objekt auswählen
  - bei Pfeilen dreischrittig
    - Pfeilart auswählen (horizontal/vertikal)
    - Startobjekt und Zielobjekt auswählen
  - Fehler (kein Objekt getroffen) → Grundzustand
  - außerdem noch Verschieben von Objekten mit der Maus
  - Implementierung "hemdsärmlig" mit Konstanten und großem switch
  - zugrundeliegendes Modell:
    - endlicher Automat mit Zuständen aus UmlGui
    - Übergänge gemäß Transitionsdiagramm



- Weitere Bemerkungen zur Implementierung:
  - Problem mit Stroke
    - Stroke wird im Konstruktor initialisiert
    - kann nicht serialisiert werden (Stroke ist nicht Serializable)
    - beim Laden nicht initialisiert (läuft nicht durch den Konstruktor)
    - Lösung: Lazy Initialisation

- Gleichheit von ArrowType-Werten
  - type.equals() (von java.lang.Object geerbt) testet nur Gleichheit der Referenz
  - beim Laden entstehen neue Objekte → Adressen sind verschieden
  - daher: equals() testet auf gleichen name-Wert
  - auch hashCode() anpassen!
- das fertige Programm als Applet (ohne Laden/Speichern)

Aufgaben

- Aufgabe 1
- Aufgabe 2
- Aufgabe 3
- Aufgabe 4
- Aufgabe 5
- Aufgabe 6
- Aufgabe 7
- Aufgabe 8
- Aufgabe 9

Aufgabe 1

- Erweitern Sie die Klasse Matrix zunächst um folgende Methoden:
  - Matrix(Matrix a)
    // erzeugt Matrix als Kopie von a
  - Matrix getMatrix(int i0, int i1, int j0, int j1)
    // erzeugt eine Teilmatrix A(i0..i1, j0..j1)
  - void setMatrix(int i0, int j0, Matrix teil)
    // belegt die Matrix mit einer Teilmatrix
  - Matrix minus(Matrix a)
    // gibt Differenz von aktueller Matrix und a zurück
  - Matrix plus(Matrix a)
    // gibt Summe von aktueller Matrix und a zurück
  - void plusGleich(Matrix a)
    // addiert a zur aktuellen Matrix
  - void minusGleich(Matrix a)
    // subtrahiert a von der aktuellen Matrix
  - String toString()
    // Ausgabestring: Zeilen durch Newline getrennt, Werte durch Komma
- Implementieren Sie nun den Strassen-Algorithmus und vergleichen Sie die Ausführungszeiten für verschiedene Matrixgrößen mit denen der normalen Matrixmultiplikation. Beschränken Sie sich auf den einfachen Fall von quadratischen Matrizen, deren Dimension eine Zweierpotenz ist.
- Fügen Sie noch eine minimale Matrixgröße ein, unterhalb derer die Rekursion abbricht und stattdessen die normale Matrixmultiplikation verwendet wird.
- Lösung

Aufgabe 2

- Erweitern Sie die Implementierung der Klasse IntStack um eine automatische Vergrößerung des Arrays bei Überlauf. Schreiben Sie eine kleine Testroutine dazu.
- Lösung

Aufgabe 3

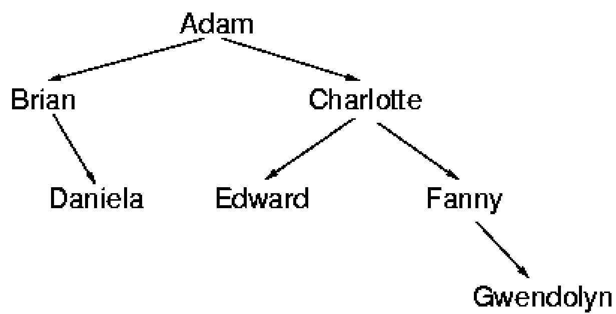
- Schreiben Sie ein Programm, das einen vollständig geklammerten arithmetischen Ausdruck mit positiven Zahlen einliest, das Ergebnis berechnet und ausgibt. Gehen Sie der Einfachheit halber davon aus, dass der Eingabestring nur aus Ziffern, Klammern und den 4 Operatorzeichen besteht.
- Hinweise:
  - Zur Zerlegung des Eingabestrings könnte die Klasse java.util.StringTokenizer nützlich sein.
  - Auf dem Stack müssen sowohl einzelne Zeichen als auch ganze Zahlen abgelegt werden. Definieren Sie eine Klasse Token als gemeinsame Klasse der verschiedenen Elementtypen.
- Lösung

Aufgabe 4

- Implementieren Sie einen Integer-Stack mit Hilfe einer verketteten Liste und testen Sie ihn mit dem Testprogramm von Aufgabe 2.
- Lösung

Aufgabe 5

- Schreiben Sie ein Programm zur Darstellung eines einfachen Familienstammbaums:
  - Definieren Sie dazu einen Knoten mit einem String für den Namen als Datenfeld und Zeigern für Vater und Mutter.
  - Schreiben Sie Methoden zum Einfügen von Vater oder Mutter. Gehen Sie dabei davon aus, dass jeder Name nur einmal auftaucht. Erzeugen Sie damit den folgenden Stammbaum (wie immer in der Informatik auf dem Kopf):



  - Überlegen Sie sich ein Verfahren für eine schöne Ausgabe des Stammbaums.
- Lösung

Aufgabe 6

- Erweitern Sie Baum und AVL-Baum jeweils um Methoden, die die Höhe eines Baumes bestimmen und um eine Ausgaberoutine, die die Baumstruktur deutlich werden lässt.
- Lesen Sie nun einige Werte vom Benutzer ein und geben Sie den entsprechenden Sortierbaum und AVL-Baum sowie deren Höhen aus. Um auch größere Bäume zu erzeugen, verwenden Sie alternativ Zufallszahlen zum Aufbau der Bäume.
- Lösung

Aufgabe 7

- Implementieren Sie einen Hash-Speicher für double-Werte mit verschiedenen Hash- und Sondierfunktionen. Vergleichen Sie deren Auswirkungen, indem Sie die Zahl der Kollisionen bei verschiedenem Füllungsgrad bestimmen.
- Lösung

Aufgabe 8

- Schreiben Sie ein Programm, das einen Beispieltext aus einer Datei einliest und eine Häufigkeitstabelle der verwendeten Wörter erstellt, sortiert nach dem Alphabet.
- Testen Sie Ihr Programm an der Datei sample.txt.
- Lösung

Aufgabe 9

- Verbessern Sie die Klasse UmlGui und begründen Sie Ihre Änderungen!
- Erweitern Sie den UML-Editor um folgende Funktionen:
  1. Die Namen der Pfeile werden am Pfeil angezeigt.
  2. Zusätzlich zu den Icons können Befehle auch über Menüs abgesetzt werden.
  3. Einige Parameter können konfigurieriert werden, z.B. Linienbreite, Fonts, Abstandsgrößen, maximale/variable Breite.
  4. Die Breite der Rechtecke ergibt sich nicht aus der Stringlänge, sondern kann vom Benutzer vorgegeben werden. Passen Sie dazu Field- und MethodView an, damit sie Felder und Methoden über mehr als eine Zeile anzeigen können.

Anhang

- Literatur
- Nachweise
- Prüfungsleistung
- Sourcen
- Applets

Literatur

1. A.Solymosi, U.Grude: Grundkurs Algorithmen und Datenstrukturen
   Vieweg, 3. Aufl. 2002, ISBN 3528257431
2. N. Wirth: Algorithmen und Datenstrukturen
   Teubner, 4. Aufl. 1995, ISBN 3519122502
3. G.Saake, K.-U.Sattler: Algorithmen und Datenstrukturen
   Dpunkt Verlag 2002, ISBN 3898641228
4. H.W.Lang: Algorithmen in Java
   Oldenbourg 2003, ISBN 3486259008
5. B. Oestereich: Objektorientierte Softwareentwicklung: Analyse und Design mit der UML
   Oldenbourg, 5. Auflage 2001, ISBN 3-486-25573-8
6. Erich Gamma, Richard Helm, Ralph Johnson: Design Patterns
   Addison-Wesley Professional 1997, ISBN 0201633612
7. S.J.Metsker: Design Patterns Java Workbook
   Addison-Wesley Professional 2002, ISBN 0201743973
8. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal:
   Pattern-Oriented Software Architecture 1. A System of Patterns
   John Wiley 1996, ISBN 0471958697
9. S. A. Stelting, O. Maassen-Van Leeuwen: Applied Java Patterns
   Prentice Hall 2002, ISBN 0130935387

Nachweise

1. V. Strassen. Gaussian Elimination is Not Optimal.
   Numer. Math., 13:354-356, 1969.
2. D. Coppersmith, S. Winograd: Matrix Multiplication via Arithmetic Progression.
   J. Symb. Comp. 9:251-80, 1990.
3. Huss-Lederman, S., E. M. Jacobson, J. R. Johnson, A. Tsao, T. Turnbull:
   Implementation of Strassen's Algorithm for Matrix Multiplication
   Proceedings of Supercomputing '96
   http://www-unix.mcs.anl.gov/prism/lib/techsrc/wn33.ps.Z
4. Pattern Home Page
   http://hillside.net/patterns/
5. UML Resource Page der OMG
   http://www.omg.org/uml/
6. Open-Source-Projekt ArgoUML
   http://argouml.tigris.org/
7. E. Armstrong: Create enumerated constants in Java
   JavaWorld, July 1997
   http://www.javaworld.com/javaworld/jw-07-1997/jw-07-enumerated.html

Prüfungsleistung

Die Prüfungsleistung ist vom Typ RP (Rechnerprogramm) der Prüfungsordnung. Von jeder Arbeitsgruppe sind abzugeben:

- eine CD mit
    - sämtlichen zur Lösung gehörenden Sourcen,
    - einer HTML-Seite mit lauffähigem Applet,
    - der Dokumentation,
- ein Exemplar der Dokumentation in gedruckter Form.

Die Dokumentation sollte einen Umfang von etwa 15 - 20 Seiten haben und sämtliche selbst erstellten Programmteile beschreiben, u.a. durch

- Beschreibung des Designs und der gefällten Design-Entscheidungen, unterstützt durch UML-Diagramme und Design-Patterns,
- Erklärung zusätzlich verwendeter Patterns aus der Literatur,
- Darstellung der Schnittstellen aller neuen oder geänderten Klassen,
- Hinweise zu Besonderheiten der Implementierung,
- Begründung der Änderungen gegenüber der Beispiel-Version.

Sourcen

- Algorithmen und Datenstrukturen
  - Bubblesort, Quicksort, MatrixMultiplikation, Intstack, ListenKnoten, Liste, BaumKnoten, Baum, AvlKnoten, AvlBaum
- Software-Entwicklung mit Design Patterns
  - Class, Interface, Object, UmlElement, Field, Method, Member, Visibility (vorläufig), UmlManager, Visibility, TestUmlManager, View (vorläufig), ClassView, InterfaceView, ObjectView, MethodView, FieldView, UmlGui, Editor, ClassEditor, ListEditorPanel, InterfaceEditor, ObjectEditor, MethodEditor, FieldEditor, Arrow, ArrowType, ArrowView, ArrowEditor, Observer, View, ObserverAdapter
- Aufgaben
  - Matrix, TestMatrixMultiplikation
  - IntStack 2
  - Token, ArithmeticTokenizer, ArithmeticEvaluator
  - ListenKnoten 2, Liste 2, IntStackListe
  - BaumKnoten 2, Baum 2
  - BaumKnoten 3, Baum 3, AvlKnoten 2, AvlBaum 2, IoTools, TestBaeume
  - SimpleHashtable, Hashtable, TestHashtable
  - WordCounter

Sortierroutine für Bubblesort

```
static void sort(double[] a) {
  boolean nochmal = true;

  while (nochmal) {
    // laufe solange durch das Array, bis keine Vertauschungen mehr nötig waren
    nochmal = false;
    for (int i = 0; i < a.length - 1; i++) {
      if (a[i] > a[i + 1]) {
        vertausche(a, i, i + 1);
        nochmal = true;
      }
    }
  }
}

static void vertausche(double[] a, int i, int j) {
  // vertausche a[i] und a[j]
  double temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
```

Sortierroutine für QuickSort

```
static void sort(double[] liste) {
  quicksort(liste, 0, liste.length - 1);
}

static void quicksort(double[] a, int oben, int unten) {
  int high = oben;
  int low = unten;
  double x = a[(high+low)/2];   // "mittleres" Element

  // Aufteilen
  while (high <= low) {
    while (a[high] < x) {  // suche großes Element bei den Kleinen
      high++;
    }
    while (a[low] > x) {  // suche kleines Element bei den Großen
      low--;
    }
    if (high <= low) {
      vertausche(a, high, low);
      high++;
      low--;
    }
  }

  // Rekursion
  if (oben < low) {  // Abbruchbedingung
    quicksort(a, oben, low);
  }

  if (high < unten) {  // Abbruchbedingung
    quicksort(a, high, unten);
  }
}

static void vertausche(double[] a, int i, int j) {
  // vertausche a[i] und a[j]
  double temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
```

Einfache Routine zur Matrix-Multiplikation

```
public static Matrix matmult(Matrix a, Matrix b) {
  // einfache Matrix-Multiplikation, ohne Checks

  int m = a.getColumnDimension();
  int l = a.getRowDimension();
  int n = b.getRowDimension();
  Matrix c = new Matrix(n, m);

  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      c.arr[i][j] = 0.0;
      for (int k = 0; k < l; k++) {
        c.arr[i][j] += a.arr[i][k] * b.arr[k][j];
      }
    }
  }
  return c;
}
```

Stack mit Integer-Daten

```
public class IntStack {

  private int[] a;    // Liste der Elemente
  private int last;   // Zahl der Elemente

  public IntStack() {
    // erzeugt einen Stack von Standardgröße
    this(100);
  }

  public IntStack(int size) {
    // erzeugt einen Stack der Größe size
    a = new int[size];
    last = 0;
  }

  public void push(int d) {
    // legt d auf dem Stack ab
    a[last] = d;
    last++;
  }

  public double pop() {
    // holt oberstes Element vom Stack
    last--;
    return a[last];
  }

  public boolean empty() {
    // true, wenn kein Element implements Stack ist
    return (last == 0);
  }

  public double peek() {
    // gibt oberstes Element aus, ohne es zu entfernen
    return a[last - 1];
  }
}
```

ListenKnoten.java

```
public class ListenKnoten {

  double daten;
  ListenKnoten naechster;

  // einfacher Konstruktor
  public ListenKnoten(double n) {
    daten = n;
    naechster = null;   // zeigt noch nirgendwo hin
  }

  // Konstruktor mit Ziel
  public ListenKnoten(double n, ListenKnoten next) {
    daten = n;
    naechster = next;
  }
}
```

Liste.java

```
import java.io.*;

public class Liste {

  private ListenKnoten start;

  public Liste() {
    // Konstruktor für leere Liste
    start = null;
  }

  public void einfügen(ListenKnoten neu) {
    // fügt Knoten neu an die richtige Stelle ein

    // 1. Fall: Liste ist noch leer
    if (start == null) {
      start = neu;
      return;
    }

    // 2. Fall: neues Element kommt an den Anfang der Liste
    if (start.daten > neu.daten) {
      neu.naechster = start;
      start = neu;
      return;
    }

    // 3. Fall: Liste durchlaufen
    ListenKnoten letzter = start;
    ListenKnoten aktuell = start.naechster;

    while ((aktuell != null) && (aktuell.daten < neu.daten)) {
      aktuell = aktuell.naechster;
      letzter = letzter.naechster;
    }

    // gefunden, jetzt einfügen
    letzter.naechster = neu;
    if (aktuell != null) {    // neu ist nicht letztes Element
      neu.naechster = aktuell;
    }
  }

  public void ausdrucken() {
    ListenKnoten aktuell = start;
    while (aktuell != null) {
      System.out.println(aktuell.daten);
      aktuell = aktuell.naechster;
    }
  }

  public static void main(String[] args) throws IOException {
    // Testprogramm mit Zufallszahlen
```

```
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

    // Anzahl der Werte abfragen
    System.out.println("Anzahl der Listenelemente eingeben:");
    String s = in.readLine();
    int anzahl = Integer.parseInt(s);

    // starte mit leerer Liste
    Liste sortierListe = new Liste();

    // fülle die Liste
    for (int i = 0; i < anzahl; i++) {
      double wert = Math.random();
      ListenKnoten neu = new ListenKnoten(wert);
      sortierListe.einfügen(neu);
    }

    // gib die Liste aus
    sortierListe.ausdrucken();
  }

}
```

BaumKnoten.java

```
public class BaumKnoten {

  public double daten;
  public BaumKnoten links;
  public BaumKnoten rechts;

  // einfacher Konstruktor
  public BaumKnoten(double n) {
    daten  = n;
    links  = null;   // zeigen noch nirgendwo hin
    rechts = null;
  }

  // Konstruktor mit Zielen
  public BaumKnoten(double n, BaumKnoten l, BaumKnoten r) {
    daten = n;
    links = l;
    rechts = r;
  }
}
```

Baum.java

```
package algorithmen;

import java.io.*;

public class Baum {

  public BaumKnoten wurzel;

  public Baum() {
    // Konstruktor erzeugt leeren Baum
    wurzel = null;
  }

  public void einfügen(BaumKnoten neu) {
    // fügt Knoten neu an die richtige Stelle ein

    // 1. Fall: Baum ist noch leer
    if (wurzel == null) {
      wurzel = neu;
    } else {
      // sonst rekursiv durch
      einfügen(wurzel, neu);
    }
  }

  private void einfügen(BaumKnoten spitze, BaumKnoten neu) {
    // fügt Knoten neu an die richtige Stelle unter Teilbaum spitze ein

    if (spitze.daten > neu.daten) {
      // links einfügen
      if (spitze.links == null) {
        // neu einfach anhängen
        spitze.links = neu;
      } else {
        // neu im linken Unterbaum unterbringen
        einfügen(spitze.links, neu);
      }
    } else {
      // rechts einfügen
      if (spitze.rechts == null) {
        // neu einfach anhängen
        spitze.rechts = neu;
      } else {
        // neu im rechten Unterbaum unterbringen
        einfügen(spitze.rechts, neu);
      }
    }
  }

  public String toString() {
    if (wurzel != null) {
      return toString(wurzel);
    } else {
      return "<leerer Baum>";
```

```
    }
  }

  private String toString(BaumKnoten b) {
    // Ausdrucken des Teilbaums ab b, Reihenfolge inorder
    String s = "";

    if (b.links != null) {
      s += toString(b.links);
    }
    s += b.daten + "\n";
    if (b.rechts != null) {
      s += toString(b.rechts);
    }

    return s;
  }

  // Testprogramm mit Zufallszahlen
  public static void main(String[] args) throws IOException {

    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

    // Anzahl der Werte abfragen
    System.out.println("Anzahl der Listenelemente eingeben:");
    String s = in.readLine();
    int anzahl = Integer.parseInt(s);

    // starte mit leerem Baum
    Baum sortierBaum = new Baum();

    // fülle den Baum
    for (int i = 0; i < anzahl; i++) {
      double wert = Math.random();
      BaumKnoten neu = new BaumKnoten(wert);
      sortierBaum.einfügen(neu);
    }

    // gib den Baum aus
    System.out.println(sortierBaum);
  }
}
```

AVLKnoten.java

```
package algorithmen;

public class AVLKnoten {

  double daten;
  AVLKnoten links;
  AVLKnoten rechts;
  int balance;

  // einfacher Konstruktor
  public AVLKnoten(double n) {
    daten  = n;
    balance = 0;
    links  = null;   // zeigen noch nirgendwo hin
    rechts = null;
  }
}
```

AVLBaum.java

```
package algorithmen;

import java.io.*;

public class AVLBaum {

  private AVLKnoten wurzel;

  public AVLBaum() {
    // Konstruktor erzeugt leeren Baum
    wurzel = null;
  }

  private AVLKnoten rotiereLinks(AVLKnoten b) {
    // vollführt einfaches Rotieren mit b, Fall links - links
    // gibt die geänderte Wurzel zurück
    // Vorbedingung: b ist ein nichtleerer Baum mit den richtigen Teilbäumen
    //    zum entsprechenden Rotieren

    AVLKnoten a = b.links;
    b.links = a.rechts;
    a.rechts = b;

    b.balance = 0;
    a.balance = 0;

    return a;
  }


  private AVLKnoten rotiereRechts(AVLKnoten b) {
    // vollführt einfaches Rotieren mit b, Fall rechts - rechts
    // gibt die geänderte Wurzel zurück
    // Vorbedingung: b ist ein nichtleerer Baum mit den richtigen Teilbäumen
    //    zum entsprechenden Rotieren

    AVLKnoten a = b.rechts;
    b.rechts = a.links;
    a.links = b;

    b.balance = 0;
    a.balance = 0;

    return a;
  }

  private AVLKnoten rotiereDoppeltLinksRechts(AVLKnoten c) {
    // vollführt doppeltes Rotieren mit c, Fall links - rechts
    // gibt die geänderte Wurzel zurück
    // Vorbedingung: c ist ein nichtleerer Baum mit den richtigen Teilbäumen
    //    zum entsprechenden Rotieren

    AVLKnoten a = c.links;
    AVLKnoten b = a.rechts;
```

```
  a.rechts = b.links;
  b.links = a;
  c.links = b.rechts;
  b.rechts = c;

  c.balance = (b.balance == -1)? 1 : 0;
  a.balance = (b.balance ==  1)? -1 : 0;
  b.balance = 0;

  return b;
}

private AVLKnoten rotiereDoppeltRechtsLinks(AVLKnoten c) {
  // vollführt doppeltes Rotieren mit c, Fall rechts - links
  // gibt die geänderte Wurzel zurück
  // Vorbedingung: c ist ein nichtleerer Baum mit den richtigen Teilbäumen
  //    zum entsprechenden Rotieren

  AVLKnoten a = c.rechts;
  AVLKnoten b = a.links;
  a.links = b.rechts;
  b.rechts = a;
  c.rechts = b.links;
  b.links = c;

  c.balance = (b.balance == 1)? -1 : 0;
  a.balance = (b.balance ==  -1)? 1 : 0;
  b.balance = 0;

  return b;
}

public void einfuegen(AVLKnoten neu) {
  // fügt Knoten neu an die richtige Stelle ein

  // 1. Fall: Baum ist noch leer
  if (wurzel == null) {
    wurzel = neu;
  } else {
    // sonst rekursiv durch
    wurzel = einfuegen(wurzel, neu);
  }
}

boolean rebalance;   // muss der Baum umsortiert werden?

private AVLKnoten einfuegen(AVLKnoten spitze, AVLKnoten neu) {
  // fügt Knoten neu an die richtige Stelle unter Teilbaum spitze ein
  // gibt Zeiger auf neuen Wert für spitze zurück

  AVLKnoten temp;

  if (neu.daten > spitze.daten) {
    // rechten Teilbaum betrachten
    if (spitze.rechts == null) {
      // neuen Knoten einfach rechts anhängen
      spitze.rechts = neu;
```

```
      spitze.balance++;
      rebalance = (spitze.balance >= 1);
      return spitze;
    } else {
      // im rechten Teilbaum einfügen
      spitze.rechts = einfuegen(spitze.rechts, neu);  // ggf. spitze ändern
      if (rebalance) {
        // Ausgleichen
        switch (spitze.balance) {
          case -1:
            // war links-lastig, jetzt ausgewogen
            spitze.balance = 0;
            rebalance = false;
            return spitze;
          case 0:
            // war ausgeglichen, jetzt hier rechtslastig. Weiter oben?
            spitze.balance = 1;
            return spitze;
          case 1:
            // ok, ich bin rechts schief
            rebalance = false;  // wird hier gleich bereinigt
            if (spitze.rechts.balance == 1) {
              // Fall rechts - rechts
              return rotiereRechts(spitze);
            } else {
              // Fall rechts - links
              return rotiereDoppeltRechtsLinks(spitze);
            }
        }
      } else {
        return spitze;   // alles ok, einfach hochreichen
      }
    }
  } else {
    // linken Teilbaum betrachten, genau wie rechts
    if (spitze.links == null) {
      // neuen Knoten einfach links anhängen
      spitze.links = neu;
      spitze.balance--;
      rebalance = (spitze.balance <= -1);
      return spitze;
    } else {
      // im linken Teilbaum einfügen
      spitze.links = einfuegen(spitze.links, neu);  // ggf. spitze ändern
      if (rebalance) {
        // Ausgleichen
        switch (spitze.balance) {
          case 1:
            // war rechts-lastig, jetzt ausgewogen
            spitze.balance = 0;
            rebalance = false;
            return spitze;
          case 0:
            // war ausgeglichen, jetzt hier linkslastig. Weiter oben?
            spitze.balance = -1;
            return spitze;
          case -1:
```

```
              // ok, ich bin links schief
              rebalance = false;  // wird hier gleich bereinigt
              if (spitze.links.balance == -1) {
                // Fall links - links
                return rotiereLinks(spitze);
              } else {
                // Fall links - rechts
                return rotiereDoppeltLinksRechts(spitze);
              }
          }
        } else {
          return spitze;   // alles ok, einfach hochreichen
        }
      }
    }

    return null;  // der Form halber - sollte hier nie ankommen!
  }
}
```

Class.java

```
package algorithmen.umleditor;

import java.util.*;

/**
 * Class with Methods and Fields.
 */
public class Class extends UmlElement implements Cloneable {

  protected Vector fields;
  protected Vector methods;
  protected boolean abstractFlag;

  /**
   * Creates a new instance of Class with given name
   */
  public Class(String name) {
    super(name);
    fields = new Vector();
    methods = new Vector();
    abstractFlag = false;
  }

  /**
   * Standard-Konstruktor mit Namen "NewClass"
   */
  public Class() {
    this("NewClass");
  }


  /**
   * Fügt das Field f der Feldliste (am Ende) hinzu.
   */
  public void addField(Field f) {
    fields.add(f);
  }

  /**
   * Entfernt das Field an der Position index aus der Feldliste.
   */
  public void removeField(int index) {
    fields.remove(index);
  }

  /*
   * Gibt die Anzahl der Felder zurück.
   */
  public int getNFields() {
    return fields.size();
  }

  /**
   * Gibt das n-te Feld zurück.
```

```
     */
    public Field getField(int n) {
      return (Field) fields.get(n);
    }

    /**
     * Setzt das n-te Feld.
     */
    public void setField(int n, Field f) {
      fields.set(n, f);
    }

    /**
     * Gibt die Feldliste zurück.
     */
    public Vector getFields() {
      return fields;
    }

    /**
     * Ersetzt die komplette Feldliste.
     */
    public void setFields(Vector f) {
      fields = f;
    }

    /**
     * Fügt die Method m der Methodenliste (am Ende) hinzu.
     */
    public void addMethod(Method m) {
      methods.add(m);
    }

    /**
     * Entfernt die Method an der Position index aus der Methodenliste.
     */
    public void removeMethod(int index) {
      methods.remove(index);
    }

    /*
     * Gibt die Anzahl der Methoden zurück.
     */
    public int getNMethods() {
      return methods.size();
    }

    /**
     * Gibt die n-te Methode zurück.
     */
    public Method getMethod(int n) {
      return (Method) methods.get(n);
    }

    /**
     * Setzt die n-te Methode.
     */
```

```
public void setMethod(int n, Method f) {
  methods.set(n, f);
}

/**
 * Gibt die Methodenliste zurück.
 */
public Vector getMethods() {
  return methods;
}

/**
 * Ersetzt die komplette Methodenliste.
 */
public void setMethods(Vector m) {
  methods = m;
}

/**
 * Gets the abstract flag of the Class.
 */
public boolean isAbstract() {
  return abstractFlag;
}

/**
 * Sets the abstract flag of the Class.
 */
public void setAbstract(boolean flag) {
  abstractFlag = flag;
}

/**
 * Liefert eine String-Darstellung eines Class-Objekts.
 * Typisches Beispiel:
 *    "public class Point {
 *       private double x = 0.0;
 *       private double y = 0.0;
 *
 *       public Point(double x0, double y0);
 *       public void move(double dx, double dy);
 *    }"
 */
public String toString() {
  String s = "public ";
  if (abstractFlag) {
    s += "abstract ";
  }
  s += "class " + name + " {\n";

  // laufe durch die Feldliste
  for (Iterator i = fields.iterator(); i.hasNext(); ) {
    Field f = (Field) i.next();
    s += "  " + f + ";\n";
  }
  s += "\n";
```

```
    // laufe durch die Methodenliste
    for (Iterator i = methods.iterator(); i.hasNext(); ) {
      Method m = (Method) i.next();
      s += "  " + m + ";\n";
    }

    s += "}";
    return s;
  }
}
```

Interface.java

```
package algorithmen.umleditor;

import java.util.*;

/**
 * Interface defining methods.
 */
public class Interface extends UmlElement {

  protected Vector methods;

  /**
   * Erzeugt ein neues Objekt vom Typ Interface mit vorgegebenem Namen.
   */
  public Interface(String name) {
    super(name);
    methods = new Vector();
  }

  /**
   * Standard-Konstruktor mit Namen "NewInterface"
   */
  public Interface() {
    this("NewInterface");
  }

  /**
   * Fügt die Method m der Methodenliste (am Ende) hinzu.
   */
  public void addMethod(Method m) {
    methods.add(m);
  }

  /**
   * Entfernt die Method an der Position index aus der Methodenliste.
   */
  public void removeMethod(int index) {
    methods.remove(index);
  }

  /*
   * Gibt die Anzahl der Methoden zurück.
   */
  public int getNMethods() {
    return methods.size();
  }

  /**
   * Gibt die n-te Methode zurück.
   */
  public Method getMethod(int n) {
    return (Method) methods.get(n);
  }
```

```
  /**
   * Setzt die n-te Methode.
   */
  public void setMethod(int n, Method f) {
    methods.set(n, f);
  }

  /**
   * Gibt die Methodenliste zurück.
   */
  public Vector getMethods() {
    return methods;
  }

  /**
   * Ersetzt die komplette Methodenliste.
   */
  public void setMethods(Vector m) {
    methods = m;
  }

  /**
   * Liefert eine String-Darstellung eines Interface-Objekts.
   * Typisches Beispiel:
   *    "public interface Drawable {
   *       void draw();
   *       void move(double dx, double dy);
   *    }"
   */
  public String toString() {
    String s = "public interface " + name + " {\n";

    // laufe durch die Methodenliste
    for (Iterator i = methods.iterator(); i.hasNext(); ) {
      Method m = (Method) i.next();
      s += "  " + m + ";\n";
    }

    s += "}";
    return s;
  }
}
```

Object.java

```
package algorithmen.umleditor;

import java.util.*;

/**
 * Object einer Class, mit Werten für die Felder.
 */
public class Object extends UmlElement {
  protected String type;
  protected Vector fields;

  /**
   * Erzeugt ein neues Objekt vomTyp Interface mit vorgegebenem Namen.
   */
  public Object(String name, String type) {
    super(name);
    this.type = type;
    fields = new Vector();
  }

  /**
   * Standard-Konstruktor mit Namen "NewClass"
   */
  public Object() {
    this("newObject", "String");
  }

  /**
   * Holt den Typ des Objekts.
   */
  public String getType() {
    return type;
  }

  /**
   * Setzt den Typ des Objekts.
   */
  public void setType(String newType) {
    type = newType;
  }

  /**
   * Fügt das Field f der Feldliste (am Ende) hinzu.
   */
  public void addField(Field f) {
    fields.add(f);
  }

  /**
   * Entfernt das Field an der Position index aus der Feldliste.
   */
  public void removeField(int index) {
    fields.remove(index);
  }
```

```
  /*
   * Gibt die Anzahl der Felder zurück.
   */
  public int getNFields() {
    return fields.size();
  }

  /**
   * Gibt das n-te Feld zurück.
   */
  public Field getField(int n) {
    return (Field) fields.get(n);
  }

  /**
   * Setzt das n-te Feld.
   */
  public void setField(int n, Field f) {
    fields.set(n, f);
  }

  /**
   * Gibt die Feldliste zurück.
   */
  public Vector getFields() {
    return fields;
  }

  /**
   * Ersetzt die komplette Feldliste.
   */
  public void setFields(Vector f) {
    fields = f;
  }

  /**
   * Liefert eine String-Darstellung eines Object-Objekts.
   * Typisches Beispiel:
   *    "Dreieck meinDreieck {
   *       private double x = 0.0;
   *       private double y = 0.0;
   *    }"
   */
  public String toString() {
    String s = type + " "  + name + " {\n";

    // laufe durch die Feldliste
    for (Iterator i = fields.iterator(); i.hasNext(); ) {
      Field f = (Field) i.next();
      s += "  " + f + ";\n";
    }
    s += "}";
    return s;
  }
}
```

UmlElement.java

```
package algorithmen.umleditor;

import java.io.*;

/**
 * Base class for all objects that can be included in an UML diagram.
 */
public class UmlElement implements Serializable {

  protected String name;

  public UmlElement(String s) {
    name = s;
  }

  /**
   * Gets the name of the UmlElement.
   */
  public String getName() {
    return name;
  }

  /**
   * Sets the name of the UmlElement.
   */
  public void setName(String newName) {
    name = newName;
  }
}
```

Field.java

```
package algorithmen.umleditor;

/**
 * Data field with type and value.
 * Part of Class and Object objects as well as parameter lists.
 */
public class Field extends Member implements Cloneable {

  protected String value;

  /** Creates a new instance with given name and type */
  public Field(String name, String type) {
    super(name, type);
  }

  /**
   * Standard-Konstruktor mit Namen "NewField" und Typ int.
   */
  public Field() {
    this("newField", "int");
  }

  /**
   * Gets the value of the Field.
   */
  public String getValue() {
    return value;
  }

  /**
   * Sets the value of the Field.
   */
  public void setValue(String newValue) {
    value = newValue;
  }

  /**
   * Returns String representation of a Field in Java form.
   * Typical examples:
   *    "protected int alter=18"
   *    "static public double PI=3.1415926"
   */
  public String toString() {
    String s = "";
    if (staticFlag) {
      s += "static ";
    }

    if (visibility != Visibility.DEFAULT) {
      s += visibility + " ";
    }

    s += type + " " + name;
    if (value != null && !value.equals("")) {
```

```
      s += " = " + value;
    }

    return s;
  }

  /**
   * support for cloning.
   */
  public java.lang.Object clone() {
    java.lang.Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Field kann nicht klonen.");
    }
    return o;
  }
}
```

Method.java

```
package algorithmen.umleditor;

import java.util.*;

/**
 * Method with parameter list and return type.
 * Part of Class and Interface objects.
 */
public class Method extends Member implements Cloneable {

  protected Vector parameters;    // parameter list, contains Fields
  protected boolean abstractFlag;

  /**
   * Creates a new instance of Method with given name and return type.
   * The parameter list is empty.
   */
  public Method(String name, String type) {
    super(name, type);
    parameters = new Vector();
  }

  /**
   * Standard-Konstruktor mit Namen "NewMethod" und Typ void
   */
  public Method() {
    this("newMethod", "void");
  }

  /**
   * Fügt das Field f der Parameterliste (am Ende) hinzu.
   */
  public void addParameter(Field f) {
    parameters.add(f);
  }

  /**
   * Entfernt das Field an der Position index aus der Parameterliste.
   */
  public void removeParameter(int index) {
    parameters.remove(index);
  }

  /**
   * Holt die Liste aller Parameter.
   */
  public Vector getParameter() {
    return parameters;
  }

  /**
   * Setzt die Liste aller Parameter auf die neue Liste.
   */
  public void setParameter(Vector newList) {
```

```
    parameters = newList;
  }

  /**
   * Gets the abstract flag of the Class.
   */
  public boolean isAbstract() {
    return abstractFlag;
  }

  /**
   * Sets the abstract flag of the Class.
   */
  public void setAbstract(boolean flag) {
    abstractFlag = flag;
  }

  /**
   * Liefert eine String-Darstellung eines Method-Objekts in Java-Form.
   * Typische Beispiele:
   *    "static public double add(double d1=5,double d2=3)"
   */
  public String toString() {
    String s = "";

    if (abstractFlag) {
      s += "abstract ";
    }

    if (staticFlag) {
      s += "static ";
    }

    if (visibility != Visibility.DEFAULT) {
      s += visibility + " ";
    }

    s += type + " " + name + "(";

    // laufe durch die Parameterliste
    Iterator i = parameters.iterator();
    // erstes Feld ohne Komma am Anfang
    if (i.hasNext()) {   // Liste ist nicht leer
      Field f = (Field) i.next();
      s += f;
    }
    // übrige Elemente, mit Komma getrennt
    while (i.hasNext()) {
      Field f = (Field) i.next();
      s += ", " + f;
    }

    s += ")";
    return s;
  }

  /**
```

```
   * support for cloning.
   */
  public java.lang.Object clone() {
    java.lang.Object o = null;
    try {
      Method m = (Method) super.clone();
      for (int i = 0; i < m.parameters.size(); i++) {
        m.parameters.set(i, ((Field) m.parameters.get(i)).clone());
      }
      o = m;
    } catch(CloneNotSupportedException e) {
      System.err.println("Method kann nicht klonen.");
    }
    return o;
  }
}
```

Member.java

```
package algorithmen.umleditor;

/**
 * Generic Part of a Class.
 * Base class of Field and Method.
 */
abstract public class Member extends UmlElement {

  protected String type;
  protected Visibility visibility;
  protected boolean staticFlag;

  /**
   * Creates a new instance with given name and type,
   * default visibility, not static.
   */
  public Member(String name, String type) {
    super(name);
    this.type = type;
    staticFlag = false;
    visibility = Visibility.DEFAULT;
  }

  /**
   * Gets the type of the Field.
   */
  public String getType() {
    return type;
  }

  /**
   * Sets the type of the Field.
   */
  public void setType(String newType) {
    type = newType;
  }

  /**
   * Gets the static flag of the Field.
   */
  public boolean isStatic() {
    return staticFlag;
  }

  /**
   * Sets the static flag of the Field.
   */
  public void setStatic(boolean flag) {
    staticFlag = flag;
  }

  /**
   * Gets the visibility of the Field.
   */
```

```
  public Visibility getVisibility() {
    return visibility;
  }

  /**
   * Sets the visibility of the Field.
   */
  public void setVisibility(Visibility v) {
    visibility = v;
  }
}
```

Visibility.java

(vorläufig)

```
package algorithmen.umleditor;

import java.io.*;

/**
 * Enumeration type for the visibility of a field/method.
 * Possible values: PRIVATE, PROTECTED, PUBLIC, DEFAULT
 */
public class Visibility implements Serializable {

  private String name;

  /**
   * Creates a new instance of Visibility with a given String.
   * Since it is private, only the predefined values can be used.
   */
  private Visibility(String s) {
    name = s;
  }

  public String toString() {
    return name;
  }

  /**
   * Gibt das UML-Zeichen der Zugriffsklasse zurück.
   */
  public String getUmlSymbol() {
    if (this == PRIVATE) {
      return "-";
    } else if (this == PROTECTED) {
       return "#";
    } else if (this == PUBLIC) {
       return "+";
    } else {
      return "";
    }
  }

  /**
   * Liefert ein Array mit allen Zugriffswerten
   */
  public static Visibility[] getVisibilities() {
    return list;
  }

  /**
   * list of predefined values
   */
  public static final Visibility PRIVATE = new Visibility("private");
  public static final Visibility PROTECTED = new Visibility("protected");
```

```
  public static final Visibility PUBLIC = new Visibility("public");
  public static final Visibility DEFAULT = new Visibility("default");

  protected static Visibility[] list = {DEFAULT, PRIVATE, PROTECTED, PUBLIC};
}
```

UmlManager.java

```
package algorithmen.umleditor;

import java.util.*;
import java.io.*;

/**
 * Manager für alle Elemente eines UML-Diagramms.
 * Ist zuständig für alle globalen Operationen wie laden und speichern.
 * Gehorcht dem Singleton-Pattern.
 */
public class UmlManager implements Serializable {

  private static UmlManager theManager;
  protected List UmlElements;

  /** Creates a new instance of UMLManager */
  private UmlManager() {
    UmlElements = new ArrayList();
  }

  public static UmlManager getUmlManager() {
    if (theManager == null) {
      theManager = new UmlManager();
    }
    return theManager;
  }

  /**
   * fügt ein UmlElement in die Liste des UmlManagers ein.
   */
  public void addUmlElement(UmlElement o) {
    UmlElements.add(o);
  }

  /**
   * löscht alle UmlElements aus der Liste des UmlManagers.
   */
  public void clear() {
    UmlElements.clear();
  }

  /**
   * Speichert die Liste aller UmlObjekte in der Datei file.
   */
  public void save(String file) {
    try {
      ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(file));
      out.writeObject(UmlElements);
      out.close();    // flushes output
    } catch (Exception e) {
      System.err.println("Fehler beim Speichern in " + file);
      e.printStackTrace();
    }
  }
```

```
  /**
   * Lädt die Liste aller UmlObjekte aus der Datei file.
   */
  public void load(String file) {
    try {
      ObjectInputStream in = new ObjectInputStream(new FileInputStream(file));
      UmlElements = (List) in.readObject();
    } catch (Exception e) {
      System.err.println("Fehler beim Laden von " + file);
      e.printStackTrace();
    }
  }


  /**
   * Liefert eine String-Darstellung eines UmlManager-Objekts.
   */
  public String toString() {
    String s = "";
    for (Iterator i = UmlElements.iterator(); i.hasNext(); ) {
      UmlElement o = (UmlElement) i.next();
      s += o + "\n";
    }
    return s;
  }
}
```

Visibility.java

```
package algorithmen.umleditor;

import java.io.*;

/**
 * Enumeration type for the visibility of a field/method.
 * Possible values: PRIVATE, PROTECTED, PUBLIC, DEFAULT
 */
public class Visibility implements Serializable {

  private String name;

  /**
   * Creates a new instance of Visibility with a given String.
   * Since it is private, only the predefined values can be used.
   */
  private Visibility(String s) {
    name = s;
  }

  public String toString() {
    return name;
  }

  /**
   * Gibt das UML-Zeichen der Zugriffsklasse zurück.
   */
  public String getUmlSymbol() {
    if (this == PRIVATE) {
      return "-";
    } else if (this == PROTECTED) {
       return "#";
    } else if (this == PUBLIC) {
       return "+";
    } else {
      return "";
    }
  }

  /**
   * Liefert ein Array mit allen Zugriffswerten
   */
  public static Visibility[] getVisibilities() {
    return list;
  }

  /**
   * list of predefined values
   */
  public static Visibility PRIVATE = new Visibility("private");
  public static Visibility PROTECTED = new Visibility("protected");
  public static Visibility PUBLIC = new Visibility("public");
  public static Visibility DEFAULT = new Visibility("default");
```

```
  protected static Visibility[] list = {DEFAULT, PRIVATE, PROTECTED, PUBLIC};

  /**
   * Serialisation is not automatic for static objects.
   * It has to be augmented with writeObject and readObject.
   */
  private void writeObject(ObjectOutputStream out) throws IOException {
    out.defaultWriteObject();
    out.writeObject(PRIVATE);
    out.writeObject(PROTECTED);
    out.writeObject(PUBLIC);
    out.writeObject(DEFAULT);
  }

  private void readObject(ObjectInputStream in)
  throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    PRIVATE = (Visibility) in.readObject();
    PROTECTED = (Visibility) in.readObject();
    PUBLIC = (Visibility) in.readObject();
    DEFAULT = (Visibility) in.readObject();
  }
}
```

TestUmlManager.java

```
package algorithmen.umleditor;

/**
 * Simple test class for UmlManager
 */
public class TestUmlManager {

  /**
   * Creates a simple demo class
   */
  protected static Class createSampleClass() {
    Class c = new Class("Baum");

    // erzeuge Felder
    Field indentPrint = new Field("indentPrint", "int");
    indentPrint.setVisibility(Visibility.PRIVATE);
    indentPrint.setStatic(true);
    indentPrint.setValue("3");

    Field wurzel = new Field("wurzel", "BaumKnoten");
    wurzel.setVisibility(Visibility.PROTECTED);
    wurzel.setValue("null");

    // Felder für die Parameterlisten
    Field parMutter = new Field("nameMutter", "String");
    Field parKind = new Field("nameKind", "String");
    Field parBaumKnoten = new Field("s", "BaumKnoten");
    Field parPos = new Field("pos", "int");

    // erzeuge Methoden
    Method einfuegeMutter = new Method("einfuegeMutter", "boolean");
    einfuegeMutter.setVisibility(Visibility.PUBLIC);
    einfuegeMutter.addParameter(parMutter);
    einfuegeMutter.addParameter(parKind);

    Method einfuegeMutter2 = new Method("einfuegeMutter", "boolean");
    einfuegeMutter2.setVisibility(Visibility.PRIVATE);
    einfuegeMutter2.addParameter(parBaumKnoten);
    einfuegeMutter2.addParameter(parMutter);
    einfuegeMutter2.addParameter(parKind);

    Method toString = new Method("toString", "String");
    toString.setVisibility(Visibility.PUBLIC);

    Method toString2 = new Method("toString", "String");
    toString2.setVisibility(Visibility.PRIVATE);
    toString2.addParameter(parBaumKnoten);
    toString2.addParameter(parPos);

    c.addField(indentPrint);
    c.addField(wurzel);
    c.addMethod(einfuegeMutter);
    c.addMethod(einfuegeMutter2);
    c.addMethod(toString);
```

```
  c.addMethod(toString2);
  return c;
}

/**
 * Erzeugt ein einfaches Beispiel-Interface
 */
protected static Interface createSampleInterface() {
  Interface demoInterface = new Interface("TimerListener");

  // Felder für die Parameterlisten
  Field parEvent1 = new Field("e", "NextStepEvent");
  Field parEvent2 = new Field("e", "RestartEvent");

  Method nextStep = new Method("nextStep", "void");
  nextStep.addParameter(parEvent1);

  Method restart = new Method("restart", "void");
  restart.addParameter(parEvent2);

  demoInterface.addMethod(nextStep);
  demoInterface.addMethod(restart);
  return demoInterface;
}

/**
 * Erzeugt ein Beispiel-Objekt mit zufälligen Werten
 */
protected static int nObject = 0;

protected static Object createRandomObject() {
  // erzeuge Felder fuer das Objekt
  Field xField = new Field("x", "double");
  xField.setValue("" + Math.random());

  Field yField = new Field("y", "double");
  yField.setValue("" + Math.random());

  nObject++;
  Object o = new Object("z" + nObject, "Complex");
  o.addField(xField);
  o.addField(yField);
  return o;
}

/**
 * Erzeugt einen UML-Manager mit einigen UMLObjekten.
 */
public static UmlManager createSampleManager() {
  UmlManager theManager = UmlManager.getUmlManager();

  theManager.addUmlElement(createSampleClass());
  theManager.addUmlElement(createSampleInterface());
  theManager.addUmlElement(new Class());
  theManager.addUmlElement(createRandomObject());
  theManager.addUmlElement(createRandomObject());
  theManager.addUmlElement(createRandomObject());
```

```
    return theManager;
  }

  /**
   * Testprogramm für UmlManager.
   * Erzeugt UmlManager mit kleiner Liste von Objekten, gibt ihn aus
   *    und speichert ihn.
   * Erzeugt neue Miniliste und gibt sie aus.
   * Lädt alte Liste aus der Datei und gibt sie aus.
   */
  public static void main(String[] args) {
    String file = "umltest.ser";

    // Demo-Liste
    UmlManager theManager = createSampleManager();
    System.out.println("Dump der Demo-Liste:");
    System.out.println("====================\n\n" + theManager);
    theManager.save(file);

    // einfache Liste
    theManager.clear();
    theManager.addUmlElement(new Object());
    theManager.addUmlElement(createRandomObject());
    theManager.addUmlElement(new Interface());
    System.out.println("\nDump der Simple-Liste:");
    System.out.println("======================\n\n" + theManager);

    // Liste einladen
    theManager.load(file);
    System.out.println("\nDump der eingelesenen Liste:");
    System.out.println("============================\n\n" + theManager);
  }
}
```

View.java

(vorläufig)

```
package algorithmen.umleditor;

import java.awt.*;
import javax.swing.*;
import java.io.*;
import java.util.*;

/**
 * Basisklasse aller Views von UML-Objekten.
 * Enthält das Modell und die Position
 *     (= linke obere Ecke des umgebenden Rechtecks)
 * Die Größe ergibt sich aus dem Modell.
 */
abstract public class View implements Serializable {
  protected Point pos;
  protected UmlElement model;
  protected Dimension size;    // aus den Strings berechnet und gecached

  /** Abstand zwischen Strings oder Strings und Linien in Pixel */
  protected int sepSize = 5;

  /** div. Font, werden lazy initialisiert,
   * da dazu Graphics-Objekt nötig */
  protected Font standardFont;
  protected Font cursiveFont;
  protected Font boldFont;
  protected Font curBoldFont;

  /**
   * Erzeugt einen View für das model an der Position p.
   */
  public View(UmlElement model, Point p) {
    this.model = model;
    pos = p;
    size = new Dimension();

    // Liste der Observer bereitmachen
    observers = new Vector();
  }

  /**
   * Zeichnet das Objekt.
   */
  abstract public void draw(Graphics g);

  /**
   * Erzeugt einen Editor für das Objekt.
   * Er ist modal bzgl. des angegebenen Frames bzw. Dialogs.
   */
  abstract public Editor createEditor(UmlElement m, Frame p);
  abstract public Editor createEditor(UmlElement m, Dialog p);
```

```
/**
 * Erzeugt ein Standard-UmlElement.
 * Dies ist Teil des FactoryMethod Patterns und wird verwendet, damit
 *   der ListEditor neue Elemente verschiedenen Typs erzeugen kann.
 */
abstract public UmlElement createStandardObject();

/**
 * Holt die aktuelle Position eines Objekts.
 */
public Point getPosition() {
  return pos;
}

/**
 * Setzt die aktuelle Position eines Objekts.
 */
public void setPosition(Point p) {
  pos = p;
  notifyObservers();
}

/**
 * Holt das aktuelle Modell des Views.
 */
public UmlElement getModel() {
  return model;
}

/**
 * Setzt das aktuelle Modell des Views.
 */
public void setModel(UmlElement m) {
  model = m;
  notifyObservers();
}

/**
 * Holt die Größe des Objekts.
 * Gesetzt wird die Größe vom View selbst durch Bestimmung der Teile,
 *    nicht von außen vorgegeben.
 */
public Dimension getSize() {
  return size;
}


/**
 * Prüft, ob der Punkt p im umschließenden Rechteck des Objekts liegt.
 */
public boolean contains(Point p) {
  Rectangle r = new Rectangle(pos, size);
  return r.contains(p);
}

/**
```

```
 * Verschiebt das Objekt um (dx, dy).
 */
public void translate(int dx, int dy) {
  pos.translate(dx, dy);
  notifyObservers();
}

// Hilfsmethoden

/**
 * Gibt die Höhe von Strings bezogen auf den verwendeten Font zurück.
 */
protected int getStringHeight(Graphics g) {
  return g.getFontMetrics().getHeight();
}

/**
 * Gibt die Breite eines Strings bezogen auf den verwendeten Font zurück.
 */
protected int getStringWidth(String s, Graphics g) {
  return g.getFontMetrics().stringWidth(s);
}

/**
 * Fonts "lazy" initialisieren
 */
protected void initFonts(Graphics g) {
  if (standardFont == null) {
    standardFont = g.getFont();
    cursiveFont = standardFont.deriveFont(Font.ITALIC);
    boldFont = standardFont.deriveFont(Font.BOLD);
    curBoldFont = standardFont.deriveFont(Font.ITALIC | Font.BOLD);
  }
}


// Infrastruktur für das Observer-Pattern

/** Liste der Observer */
protected Vector observers;

/**
 * Registriert einen Observer.
 */
public synchronized void addObserver(Observer o) {
  observers.addElement(o);
}

/**
 * Löscht einen Observer aus der Liste.
 */
public synchronized void deleteObserver(Observer o) {
  observers.removeElement(o);
}

/**
 * Informiert alle Observer.
```

```
   */
  public void notifyObservers() {
    Vector l;

    // Klone die Liste der Listener, damit sich Benachrichtigung und evtl.
    // Änderungen an den Listenern (add/delete) nicht stören.
    synchronized (this) {
      l = (Vector) observers.clone();
    }

    for (int i = 0; i < l.size(); i++) {
      ((Observer) l.elementAt(i)).update();
    }
  }
}
```

ClassView.java

```
package algorithmen.umleditor;

import java.awt.*;
import javax.swing.*;

/**
 * View einer Class.
 * Zuständig für die graphische Darstellung einer Class.
 */
public class ClassView extends View {

  /**
   * Erzeugt einen View für die Class model an der Position p.
   */
  public ClassView(Class model, Point p) {
    super(model, p);
  }

  /**
   * Erzeugt einen Standard-View.
   */
  public ClassView() {
    this(new Class(), new Point(10, 10));
  }

  /**
   * Zeichnet eine Klasse in UML-Notation.
   */
  public void draw(Graphics g) {
    // kann auch mit verschieden hohen Methoden- und Field-Strings umgehen.
    initFonts(g);
    Class c = (Class) model;        // cast nötig für Class-Details
    int maxX = 0;                   // wird die Maximalbreite aller Teilfelder
    int curX = pos.x + sepSize + 1; // Startpunkt aller Strings


    // abstrakte Klasse -> Name wird in bold + italics dargestellt
    if (c.isAbstract()) {
      g.setFont(curBoldFont);
    } else {
      g.setFont(boldFont);
    }

    // erst den Klassen-Namen in bold
    int nameHeight = getStringHeight(g);
    int curY = pos.y + nameHeight + sepSize + 1;
    g.drawString(c.getName(), curX, curY);
    maxX = Math.max(maxX, getStringWidth(c.getName(), g));
    g.setFont(standardFont);

    int yPosLine1  = curY + sepSize;    // Position der 1. Trennlinie merken
    curY += 2*sepSize + 1;    // unter die erste Linie gehen

    // nun die Felder
```

```
    for (int i = 0; i < c.getNFields(); i++) {
      FieldView view = new FieldView(c.getField(i), new Point(curX, curY));
      view.draw(g);
      Dimension d = view.getSize();
      maxX = Math.max(maxX, d.width);
      curY += d.height + sepSize;
    }

    int yPosLine2  = curY;       // Position der 2. Trennlinie merken
    curY += sepSize + 1;         // unter die zweite Linie gehen

    // dann die Methoden
    for (int i = 0; i < c.getNMethods(); i++) {
      MethodView view = new MethodView(c.getMethod(i), new Point(curX, curY));
      view.draw(g);
      Dimension d = view.getSize();
      maxX = Math.max(maxX, d.width);
      curY += d.height + sepSize;
    }

    // schließlich das Rechteck und die Linien
    size.width = maxX + 2*sepSize + 2;
    size.height = curY + 1 - pos.y;
    g.drawRect(pos.x, pos.y, size.width, size.height);
    g.drawLine(pos.x,  yPosLine1, pos.x + size.width, yPosLine1);
    g.drawLine(pos.x,  yPosLine2, pos.x + size.width, yPosLine2);
  }

  public Editor createEditor(UmlElement m, Frame p) {
    return new ClassEditor((Class) m, p);
  }

  public Editor createEditor(UmlElement m, Dialog p) {
    return new ClassEditor((Class) m, p);
  }

  public UmlElement createStandardObject() {
    return new Class();
  }
}
```

InterfaceView.java

```
package algorithmen.umleditor;

import java.awt.*;
import javax.swing.*;

/**
 * View eines Interface.
 * Zuständig für die graphische Darstellung eines Interface.
 */
public class InterfaceView extends View {

  /**
   * Erzeugt einen View für das Interface model an der Position p.
   */
  public InterfaceView(Interface model, Point p) {
    super(model, p);
  }

  /**
   * Erzeugt einen Standard-View.
   */
  public InterfaceView() {
    this(new Interface(), new Point(10, 10));
  }

  /**
   * Zeichnet ein Interface in UML-Notation.
   */
  public void draw(Graphics g) {
    // kann auch mit verschieden hohen Methoden- und Field-Strings umgehen.
    initFonts(g);
    Interface c = (Interface) model; // cast nötig für Interface-Details
    int maxX = 0;                    // wird die Maximalbreite aller Teilfelder
    int curX = pos.x + sepSize + 1;  // Startpunkt aller Strings

    // erst "<> und den Namen
    int nameHeight = getStringHeight(g);
    int curY = pos.y + nameHeight + sepSize + 1;
    String header = "\u00abInterface\u00bb";
    g.drawString(header, curX, curY);
    maxX = Math.max(maxX, getStringWidth(header, g));
    curY += nameHeight + sepSize;
    g.setFont(boldFont);
    g.drawString(c.getName(), curX, curY);
    maxX = Math.max(maxX, getStringWidth(c.getName(), g));
    g.setFont(standardFont);

    int yPosLine1  = curY + sepSize;    // Position der 1. Trennlinie merken
    curY += 2*sepSize + 1;    // unter die erste Linie gehen

    // nur Methoden
    for (int i = 0; i < c.getNMethods(); i++) {
      MethodView view = new MethodView(c.getMethod(i), new Point(curX, curY));
      view.draw(g);
```

```
      Dimension d = view.getSize();
      maxX = Math.max(maxX, d.width);
      curY += d.height + sepSize;
    }

    // schließlich das Rechteck und die Linien
    size.width = maxX + 2*sepSize + 2;
    size.height = curY + 1 - pos.y;
    g.drawRect(pos.x, pos.y, size.width, size.height);
    g.drawLine(pos.x,  yPosLine1, pos.x + size.width, yPosLine1);
  }

  public Editor createEditor(UmlElement m, Frame p) {
    return new InterfaceEditor((Interface) m, p);
  }

  public Editor createEditor(UmlElement m, Dialog p) {
    return new InterfaceEditor((Interface) m, p);
  }

  public UmlElement createStandardObject() {
    return new Interface();
  }
}
```

ObjectView.java

```
package algorithmen.umleditor;

import java.awt.*;
import javax.swing.*;

/**
 * View eines Objects.
 * Zuständig für die graphische Darstellung einer Objects.
 */
public class ObjectView extends View {

  /**
   * Erzeugt einen View für das Object model an der Position p.
   */
  public ObjectView(Object model, Point p) {
    super(model, p);
  }

  /**
   * Erzeugt einen Standard-View.
   */
  public ObjectView() {
    this(new Object(), new Point(10, 10));
  }

  /**
   * Zeichnet eine Klasse in UML-Notation.
   */
  public void draw(Graphics g) {
    // kann auch mit verschieden hohen Methoden- und Field-Strings umgehen.
    initFonts(g);
    Object o = (Object) model;         // cast nötig für Class-Details
    int maxX = 0;                    // wird die Maximalbreite aller Teilfelder
    int curX = pos.x + sepSize + 1; // Startpunkt aller Strings

    // erst den Objekt-Namen und -Typ, unterstrichen
    g.setFont(boldFont);
    int nameHeight = getStringHeight(g);
    int curY = pos.y + nameHeight + sepSize + 1;
    String title = o.getName() + ": " + o.getType();
    g.drawString(title, curX, curY);
    int titleWidth = getStringWidth(title, g);
    maxX = Math.max(maxX, titleWidth);
    g.drawLine(curX, curY + 1, curX + titleWidth, curY + 1);
    g.setFont(standardFont);

    int yPosLine1  = curY + sepSize;    // Position der 1. Trennlinie merken
    curY += sepSize + 1;        // unter die erste Linie gehen

    // nun die Felder
    // in der Form: Name = Wert
    for (int i = 0; i < o.getNFields(); i++) {
      Field f = o.getField(i);
      String fieldLabel = f.getName() + " = " + f.getValue();
```

```
      curY += nameHeight + sepSize;
      g.drawString(fieldLabel, curX, curY);
      maxX = Math.max(maxX, getStringWidth(fieldLabel, g));
    }

    // schließlich das Rechteck und die Linie
    size.width = maxX + 2*sepSize + 2;
    size.height = curY + 1 + sepSize - pos.y;
    g.drawRect(pos.x, pos.y, size.width, size.height);
    g.drawLine(pos.x,  yPosLine1, pos.x + size.width, yPosLine1);
  }

  public Editor createEditor(UmlElement m, Frame p) {
    return new ObjectEditor((Object) m, p);
  }

  public Editor createEditor(UmlElement m, Dialog p) {
    return new ObjectEditor((Object) m, p);
  }

  public UmlElement createStandardObject() {
    return new Object();
  }
}
```

MethodView.java

```
package algorithmen.umleditor;

import java.awt.*;
import javax.swing.*;
import java.util.*;

/**
 * View einer Methode.
 * Zuständig für die graphische Darstellung der Methodendaten.
 */
public class MethodView extends View {

  /**
   * Erzeugt einen View für die Methode model an der Position p.
   */
  public MethodView(Method model, Point p) {
    super(model, p);
  }

  /**
   * Erzeugt einen Standard-View.
   */
  public MethodView() {
    this(new Method(), new Point(10, 10));
  }

  /**
   * Zeichnet das Datenfeld als einfachen String in UML-Notation.
   * Mögliche Weiterentwicklung: Zeilenumbrüche im String.
   */
  public void draw(Graphics g) {
    Method m = (Method) model;

    // abstrakte Methoden werden in italics dargestellt
    Font standardFont = g.getFont();
    if (m.isAbstract()) {
      if (cursiveFont == null) {  // lazy Intialisierung
        cursiveFont = standardFont.deriveFont(Font.ITALIC);
      }
      g.setFont(cursiveFont);
    }

    String s = toUmlString();
    size.width = getStringWidth(s, g);
    size.height = getStringHeight(g);
    g.drawString(s, pos.x, pos.y + size.height);

    // statische Methoden unterstreichen
    if (((Method) model).isStatic()) {
      int yLine = pos.y + size.height + 1;
      g.drawLine(pos.x, yLine, pos.x + size.width, yLine);
    }

    // ggf. Font zurücksetzen
```

```
    if (m.isAbstract()) {
      g.setFont(standardFont);
    }
  }

  /**
   * Liefert eine String-Darstellung eines Method-Objekts in UML-Form.
   * Typische Beispiele:
   *    "+add(d1: double, d2: double): double"
   */
  public String toUmlString() {
    Method m = (Method) model;
    String s = m.getVisibility().getUmlSymbol() + m.getName() + "(";

    // laufe durch die Parameterliste
    // verwendet nicht die draw-Methode, sondern nur den UMLString der Felder
    Iterator i = m.getParameter().iterator();
    // erstes Feld ohne Komma am Anfang
    if (i.hasNext()) {   // Liste ist nicht leer
      FieldView view = new FieldView((Field) i.next(), new Point());
      s += view.toUmlString();
    }
    // übrige Elemente, mit Komma getrennt
    while (i.hasNext()) {
      FieldView view = new FieldView((Field) i.next(), new Point());
      s += ", " + view.toUmlString();
    }

    s += "): " + m.getType();
    return s;
  }

  /**
   * Erzeugt einen Editor für das UmlObjekt m, modal bzgl. dem Dialog p.
   */
  public Editor createEditor(UmlElement m, Dialog p) {
    return new MethodEditor((Method) m, p);
  }

  /**
   * Erzeugt einen Editor für das UmlObjekt m, modal bzgl. dem Frame p.
   */
  public Editor createEditor(UmlElement m, Frame p) {
    return new MethodEditor((Method) m, p);
  }

  public UmlElement createStandardObject() {
    return new Method();
  }
}
```

FieldView.java

```
package algorithmen.umleditor;

import java.awt.*;
import javax.swing.*;

/**
 * View eines Datenfelds.
 * Zuständig für die graphische Darstellung der Felddaten.
 */
public class FieldView extends View {

  /**
   * Erzeugt einen View für das Field model an der Position p.
   */
  public FieldView(Field model, Point p) {
    super(model, p);
  }

  /**
   * Erzeugt einen Standard-View.
   */
  public FieldView() {
    this(new Field(), new Point(10, 10));
  }

  /**
   * Zeichnet das Datenfeld als einfachen String in UML-Notation.
   * Mögliche Weiterentwicklung: Zeilenumbrüche im String.
   */
  public void draw(Graphics g) {
    String s = toUmlString();
    size.width = getStringWidth(s, g);
    size.height = getStringHeight(g);
    g.drawString(s, pos.x, pos.y + size.height);

    // underline, if static
    if (((Field) model).isStatic()) {
      int yLine = pos.y + size.height + 1;
      g.drawLine(pos.x, yLine, pos.x + size.width, yLine);
    }
  }

  /**
   * Returns String representation of a Field in UML form.
   * Typical examples:
   *    "#alter: int"
   *    "+PI: double"
   */
  public String toUmlString() {
    Field f = (Field) model;
    String s = f.getVisibility().getUmlSymbol();
    s += f.getName() + ": " + f.getType();
    return s;
  }
```

```
  public Editor createEditor(UmlElement m, Frame p) {
    return new FieldEditor((Field) m, p);
  }

  public Editor createEditor(UmlElement m, Dialog p) {
    return new FieldEditor((Field) m, p);
  }

  public UmlElement createStandardObject() {
    return new Field();
  }

}
```

UmlGui.java

```
package algorithmen.umleditor;

import java.util.*;
import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
 * Baut die graphische Oberfläche auf.
 * Manager für alle Elemente eines UML-Diagramms.
 * Ist zuständig für alle globalen Operationen wie laden und speichern.
 * Enthält eine Liste aller Views
 */
public class UmlGui extends JApplet implements Serializable {

  protected java.util.List views;
  protected JFileChooser fc;

  protected int state;   // action that is just being performed

  // types of actions
  protected static final int POINTING = 0;
  protected static final int EDITING = 1;
  protected static final int DELETING = 2;
  protected static final int CHOOSING_LEFT_H = 3;
  protected static final int CHOOSING_RIGHT_H = 4;
  protected static final int CHOOSING_LEFT_V = 5;
  protected static final int CHOOSING_RIGHT_V = 6;
  protected static final int MOVING = 7;

  protected View leftObject;  // object found with the first click

  /** Creates a new instance of UMLGui */
  public UmlGui() {
    init();
  }

  public void init() {
    Container cp = getContentPane();

    // Erzeuge den Toolbar
    JToolBar toolBar = new JToolBar();
    addButtons(toolBar);

    // Erzeuge die Zeichenfläche
    JPanel drawArea = new DisplayPanel();
    DisplayMouseListener dml = new DisplayMouseListener();
    drawArea.addMouseListener(dml);
    drawArea.addMouseMotionListener(dml);
    JScrollPane scrollPane = new JScrollPane(drawArea);

    cp.add(toolBar, BorderLayout.NORTH);
```

```
  cp.add(scrollPane, BorderLayout.CENTER);

  try {  // keine Dateioperation möglich, wenn UmlGui als Applet läuft
    fc = new JFileChooser();
  } catch(Exception e) {
    // ok, tue nichts
  }

  views = new ArrayList();
  state = POINTING;
}

/**
 * Fügt die Buttons zum Toolbar hinzu
 */
protected void addButtons(JToolBar toolbar) {
  JButton button;

  // Button Laden
  button = new JButton(getIcon("load"));
  button.setToolTipText("Laden");
  button.addActionListener(new LoadListener());
  toolbar.add(button);

  // Button Speichern
  button = new JButton(getIcon("save"));
  button.setToolTipText("Speichern");
  button.addActionListener(new SaveListener());
  toolbar.add(button);
  toolbar.addSeparator();

  // Button Neue Klasse
  button = new JButton(getIcon("class"));
  button.setToolTipText("Neue Klasse");
  button.addActionListener(new AddClassListener());
  toolbar.add(button);

  // Button Neues Interface
  button = new JButton(getIcon("interface"));
  button.setToolTipText("Neues Interface");
  button.addActionListener(new AddInterfaceListener());
  toolbar.add(button);

  // Button Neues Objekt
  button = new JButton(getIcon("object"));
  button.setToolTipText("Neues Objekt");
  button.addActionListener(new AddObjectListener());
  toolbar.add(button);
  toolbar.addSeparator();

  // Button Neuer Pfeil vertikal
  button = new JButton(getIcon("up"));
  button.setToolTipText("Neuer vertikaler Pfeil");
  button.addActionListener(new AddUpArrowListener());
  toolbar.add(button);

  // Button Neuer Pfeil horizontal
```

```
    button = new JButton(getIcon("right"));
    button.setToolTipText("Neuer horizontaler Pfeil");
    button.addActionListener(new AddRightArrowListener());
    toolbar.add(button);
    toolbar.addSeparator();

    // Button Edit
    button = new JButton(getIcon("edit"));
    button.setToolTipText("Editieren");
    button.addActionListener(new EditListener());
    toolbar.add(button);

    // Button Delete
    button = new JButton(getIcon("delete"));
    button.setToolTipText("Löschen");
    button.addActionListener(new DeleteListener());
    toolbar.add(button);
  }

  /**
   * Fügt den View eines UML-Objekts in die Liste ein.
   */
  public void addUmlElement(View o) {
    views.add(o);
    repaint();
  }

  /**
   * löscht alle Views aus der Liste.
   */
  public void clear() {
    views.clear();
  }

  /**
   * Speichert die Liste aller Views in der Datei file.
   */
  public void save() {
    if (fc.showSaveDialog(this) != JFileChooser.APPROVE_OPTION) {
      return;
    }

    File file = fc.getSelectedFile();
    try {
      ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(file));
      out.writeObject(views);
      out.close();    // flushes output
    } catch (Exception e) {
      System.err.println("Fehler beim Speichern in " + file.getName());
      e.printStackTrace();
    }
  }

  /**
   * Lädt die Liste aller Views aus der Datei file.
   */
  public void load() {
```

```
  if (fc.showOpenDialog(this) != JFileChooser.APPROVE_OPTION) {
    return;
  }

  File file = fc.getSelectedFile();
  try {
    ObjectInputStream in = new ObjectInputStream(new FileInputStream(file));
    views = (java.util.List) in.readObject();
  } catch (Exception e) {
    System.err.println("Fehler beim Laden von " + file.getName());
    e.printStackTrace();
  }

  repaint();
}

public static void main(String[] args) {
  // erzeuge ein neues Fenster, das sich richtig beendet
  JFrame meinFenster = new JFrame("UML Editor");
  meinFenster.setDefaultCloseOperation(meinFenster.EXIT_ON_CLOSE);

  // packe ein ZeichenFenster hinein
  Container p = meinFenster.getContentPane();
  p.add(new UmlGui());

  // zeige das Fenster in bestimmter Größe an
  meinFenster.setSize(640, 480);
  meinFenster.setVisible(true);

}

// interne Methoden

/*
 * Erzeugt ein ImageIcon aus der Datei name im images-Verzeichnis.
 * Hängt Endung .gif an name an.
 */
protected ImageIcon getIcon(String name) {
  // Problem:
  // Programm läuft als Applet und Applikation ->
  //   als Applet kann der Pfad nicht direkt verwendet werden,
  //   als Applikation kann getCodebase von Applet nicht benutzt werden.
  //  Daher verwende ich den ClassLoader explizit
  //     -> Basis-URL bezieht sich auf Spitze der Paket-Hierarchie
  ClassLoader myCL = getClass().getClassLoader();
  String urlString = "algorithmen/umleditor/images/" + name + ".gif";
  URL myURL = myCL.getResource(urlString);
  if (myURL != null) {
    return new ImageIcon(myURL);
  } else {
    return null;
  }
}

/**
 * Bestimmt das umgebende Frame.
 * Vor allem für das Applet wichtig.
```

```
 */
protected Frame getFrame() {
  Container c = getParent();
  while (c != null && !(c instanceof Frame)) {
    c = c.getParent();
  }
  return (Frame) c;
}

class DisplayPanel extends JPanel {
  public void paintComponent(Graphics g) {
    setBackground(Color.WHITE);
    super.paintComponent(g);

    // laufe durch die Objektliste
    for (Iterator i = views.iterator(); i.hasNext(); ) {
      View v = (View) i.next();
      v.draw(g);
    }
  }
}

class LoadListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    if (fc != null) {  // no load when running as applet
      load();
    }
  }
}

class SaveListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    if (fc != null) {   // no save when running as applet
      save();

    }}
}

class AddClassListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    addUmlElement(new ClassView());
  }
}

class AddInterfaceListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    addUmlElement(new InterfaceView());
  }
}

class AddObjectListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    addUmlElement(new ObjectView());
  }
}

class AddRightArrowListener implements ActionListener {
```

```
  public void actionPerformed(ActionEvent e) {
    state = CHOOSING_LEFT_H;
    setCursor(new Cursor(Cursor.CROSSHAIR_CURSOR));
  }
}

class AddUpArrowListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    state = CHOOSING_LEFT_V;
    setCursor(new Cursor(Cursor.CROSSHAIR_CURSOR));
  }
}

class EditListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    state = EDITING;
    setCursor(new Cursor(Cursor.CROSSHAIR_CURSOR));
  }
}

class DeleteListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    state = DELETING;
    setCursor(new Cursor(Cursor.CROSSHAIR_CURSOR));
  }
}

class DisplayMouseListener extends MouseAdapter
implements MouseMotionListener {

  // aktives Objekt einer Mausoperation
  protected View hitObjectView = null;
  // Koordinaten des letzten mousePressed
  protected Point lastPoint;

  public void mousePressed(MouseEvent e){
    // prüfe, welches Objekt getroffen wurde
    hitObjectView = null;
    for (Iterator i = views.iterator(); i.hasNext(); ) {
      View v = (View) i.next();
      if (v.contains(e.getPoint())) {
        hitObjectView = v;
        break;
      }
    }

    // nichts getroffen -> zurück in Grundzustand
    if (hitObjectView == null) {
      state = POINTING;
      setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
      return;
    }

    // Aktion je nach Zustand durchführen
    Arrow a;
    switch (state) {
      case DELETING:
```

```
      views.remove(hitObjectView);
      setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
      state = POINTING;
      repaint();
      break;
    case EDITING:
      hitObjectView.createEditor(hitObjectView.getModel(), getFrame());
      setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
      state = POINTING;
      repaint();
      break;
    case POINTING:
      lastPoint = e.getPoint();
      state = MOVING;
      break;
    case CHOOSING_LEFT_H:
      leftObject = hitObjectView;
      state = CHOOSING_RIGHT_H;
      break;
    case CHOOSING_LEFT_V:
      leftObject = hitObjectView;
      state = CHOOSING_RIGHT_V;
      break;
    case CHOOSING_RIGHT_H:
      a = new Arrow(leftObject.getModel(), hitObjectView.getModel(),
      ArrowType.REFERENCE);
      addUmlElement(new ArrowView(a, leftObject, hitObjectView));
      setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
      state = POINTING;
      repaint();
      break;
    case CHOOSING_RIGHT_V:
      a = new Arrow(leftObject.getModel(), hitObjectView.getModel(),
      ArrowType.EXTENSION);
      addUmlElement(new ArrowView(a, leftObject, hitObjectView));
      setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
      state = POINTING;
      repaint();
      break;
  }
}

public void mouseDragged(MouseEvent e) {
  if (hitObjectView != null && state == MOVING) {
    hitObjectView.translate(e.getX() - lastPoint.x, e.getY() - lastPoint.y);
    lastPoint = e.getPoint();
    repaint();
  }
}

public void mouseReleased(MouseEvent e){
  if (hitObjectView != null && state == MOVING) {
    hitObjectView.translate(e.getX() - lastPoint.x, e.getY() - lastPoint.y);
    hitObjectView = null;
    state = POINTING;
    repaint();
  }
```

```
    }

    public void mouseMoved(MouseEvent e) {  // tue nichts
    }

  }
}
```

Editor.java

```
package algorithmen.umleditor;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
 * Base class of all editors.
 * It provides a main panel and a standard button panel for closing
 * with or without updating its model object.
 */
abstract public class Editor extends JDialog {
  protected UmlElement model;

  /**
   * Erzeugt einen Editor der Größe xSize x ySize für ein UmlElement o,
   * der modal ist bzgl. dem Jdialog parent.
   */
  public Editor(UmlElement o, int xSize, int ySize, Dialog parent) {
    super(parent, true);
    initEditor(o, xSize, ySize);
  }

  /**
   * Erzeugt einen Editor der Größe xSize x ySize für ein UmlElement o,
   * der modal ist bzgl. dem JFrame parent.
   */
  public Editor(UmlElement o, int xSize, int ySize, Frame parent) {
    super(parent, true);
    initEditor(o, xSize, ySize);
  }

  /**
   * gemeinsamer Teil der Konstruktoren
   */
  protected void initEditor(UmlElement o, int xSize, int ySize) {
    model = o;

    setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
    Container cp = getContentPane();
    cp.setLayout(new BorderLayout());

    // Haupt-Panel, wird von den Kindern gefüllt
    JPanel mainPanel = new JPanel();
    buildMainPanel(mainPanel);
    cp.add(mainPanel, BorderLayout.CENTER);

    // erzeuge Panel mit Buttons zum Abbrechen und Übernehmen
    JPanel buttonPanel = new JPanel();
    JButton noButton = new JButton("Abbrechen");
    noButton.addActionListener(new NoButtonListener());
    buttonPanel.add(noButton);
    JButton okButton = new JButton("Übernehmen");
    okButton.addActionListener(new OkButtonListener());
```

```
    buttonPanel.add(okButton);
    cp.add(buttonPanel, BorderLayout.SOUTH);

    setSize(xSize, ySize);
    setVisible(true);
  }

  /**
   * Aufbau des Haupt-Editor-Panels.
   */
  abstract protected void buildMainPanel(JPanel mp);

  /**
   * Übernahme der Editorwerte an die Model-Klasse
   */
  abstract protected void update();

  class NoButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      dispose();
    }
  }

  class OkButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      update();
      dispose();
    }
  }
}
```

ClassEditor.java

```
package algorithmen.umleditor;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

/**
 * Editor für eine Klasse.
 */
public class ClassEditor extends Editor {

  protected JTextField nameField;
  protected JCheckBox abstractField;
  protected Vector fieldsClone;
  protected Vector methodsClone;

  public ClassEditor(Class c, Frame p) {
    super(c, 460, 550, p);
  }

  public ClassEditor(Class c, Dialog p) {
    super(c, 460, 550, p);
  }

  protected void buildMainPanel(JPanel mp) {
    mp.setLayout(new BoxLayout(mp, BoxLayout.Y_AXIS));
    Class c = (Class) model;

    JPanel topPanel = new JPanel(new GridLayout(1, 2));

    // add name field
    JPanel namePanel = new JPanel(new FlowLayout(FlowLayout.LEFT));
    namePanel.add(new JLabel("Name:"));
    nameField = new JTextField(c.getName(), 15);
    namePanel.add(nameField);

    // add static field
    JPanel abstractPanel = new JPanel(new FlowLayout(FlowLayout.LEFT));
    abstractPanel.add(new JLabel("abstract:"));
    abstractField = new JCheckBox();
    abstractField.setSelected(c.isAbstract());
    abstractPanel.add(abstractField);

    // Listeditorpanel für die Feldliste hinzufügen
    // bekommt eine (tiefe!) Kopie der Liste, damit beim
    // Abbrechen alle Änderungen verschwinden.
    fieldsClone = (Vector) c.getFields().clone();  // leider flacher Klon!
    for (int i = 0; i < fieldsClone.size(); i++) {
      fieldsClone.set(i, ((Field) fieldsClone.get(i)).clone());
    }

    // Listeditorpanel für die Methodenliste hinzufügen
    methodsClone = (Vector) c.getMethods().clone();  // leider flacher Klon!
```

```
    for (int i = 0; i < methodsClone.size(); i++) {
      methodsClone.set(i, ((Method) methodsClone.get(i)).clone());
    }

    // alles zusammenbauen
    topPanel.add(namePanel);
    topPanel.add(abstractPanel);
    mp.add(topPanel);
    mp.add(new ListEditorPanel(fieldsClone, "Felder", new FieldView(), this));
    mp.add(new ListEditorPanel(methodsClone, "Methoden", new MethodView(), this));

  }

  protected void update() {
    Class c = (Class) model;
    c.setName(nameField.getText());
    c.setAbstract(abstractField.isSelected());
    c.setFields(fieldsClone);
    c.setMethods(methodsClone);
  }

}
```

ListEditorPanel.java

```
package algorithmen.umleditor;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import java.util.*;

/**
 * Panel zum Editieren einer Liste (definiert als Vector).
 * Zeigt die Liste an und Buttons zum Löschen, Ändern und Hinzufügen
 *    von Elementen.
 */
public class ListEditorPanel extends JPanel {

  protected JList listField;
  protected Vector theList;
  protected View elementView;
  protected JDialog parent;

  /**
   * Erzeugt einen Editor für eine Liste list von UmlObjekten mit Titel string.
   * Der View subview ermöglicht, einen passenden Editor für die Objekte zu
   *   erzeugen.
   * Der Parameter dialog ist der aufrufende JDialog (Editor) selbst, er ist
   *    Bezugspunkt für den modalen Untereditor.
   */
  public ListEditorPanel(Vector list, String title, View view, JDialog parent) {
    super(new BorderLayout());
    int bSize = 3;
    Border emptyBorder = BorderFactory.createEmptyBorder(bSize, bSize, bSize, bSize);
    setBorder(emptyBorder);
    theList = list;
    elementView = view;
    this.parent = parent;

    JLabel titleLabel = new JLabel(title + ":");
    titleLabel.setBorder(emptyBorder);

    listField = new JList();
    listField.setListData(list);
    listField.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

    JScrollPane scrollPane = new JScrollPane(listField);

    // rechts kommen drei Buttons dazu
    Box buttonBox = new Box(BoxLayout.Y_AXIS);
    buttonBox.setBorder(emptyBorder);

    JButton newButton = new JButton("Neu");
    newButton.addActionListener(new AddListener());
    newButton.setMaximumSize(new Dimension(100, 25));

    JButton changeButton = new JButton("Ändern");
```

```
    changeButton.addActionListener(new ChangeListener());
    changeButton.setMaximumSize(new Dimension(100, 25));

    JButton deleteButton = new JButton("Löschen");
    deleteButton.addActionListener(new DeleteListener());
    deleteButton.setMaximumSize(new Dimension(100, 25));

    buttonBox.add(newButton);
    buttonBox.add(changeButton);
    buttonBox.add(deleteButton);

    // und jetzt Komponenten zusammensez
    add(titleLabel, BorderLayout.NORTH);
    add(scrollPane, BorderLayout.CENTER);
    add(buttonBox, BorderLayout.EAST);
  }

  class AddListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      UmlElement o = elementView.createStandardObject();
      theList.add(o);
      listField.setListData(theList);
      repaint();
    }
  }

  class ChangeListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      int index = listField.getSelectedIndex();
      if (index != -1) {
        UmlElement o = (UmlElement) theList.get(index);
        elementView.createEditor(o, parent);
        repaint();
      }
    }
  }

  class DeleteListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      int index = listField.getSelectedIndex();
      if (index != -1) {
        theList.remove(index);
        repaint();
      }
    }
  }
}
```

InterfaceEditor.java

```
package algorithmen.umleditor;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

/**
 * Editor für eine Klasse.
 */
public class InterfaceEditor extends Editor {

  protected JTextField nameField;
  protected Vector methodsClone;

  public InterfaceEditor(Interface c, Frame p) {
    super(c, 460, 300, p);
  }

  public InterfaceEditor(Interface c, Dialog p) {
    super(c, 460, 300, p);
  }

  protected void buildMainPanel(JPanel mp) {
    mp.setLayout(new BoxLayout(mp, BoxLayout.Y_AXIS));
    Interface c = (Interface) model;

    JPanel topPanel = new JPanel(new GridLayout(1, 2));

    // add name field
    JPanel namePanel = new JPanel(new FlowLayout(FlowLayout.LEFT));
    namePanel.add(new JLabel("Name:"));
    nameField = new JTextField(c.getName(), 15);
    namePanel.add(nameField);

    // Listeditorpanel für die Methodenliste hinzufügen
    methodsClone = (Vector) c.getMethods().clone();  // leider flacher Klon!
    for (int i = 0; i < methodsClone.size(); i++) {
      methodsClone.set(i, ((Method) methodsClone.get(i)).clone());
    }

    // alles zusammenbauen
    topPanel.add(namePanel);
    mp.add(topPanel);
    mp.add(new ListEditorPanel(methodsClone, "Methoden", new MethodView(), this));
  }

  protected void update() {
    Interface c = (Interface) model;
    c.setName(nameField.getText());
    c.setMethods(methodsClone);
  }

}
```

ObjectEditor.java

```
package algorithmen.umleditor;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

/**
 * Editor für ein Object.
 */
public class ObjectEditor extends Editor {

  protected JTextField nameField;
  protected JTextField typeField;
  protected Vector fieldsClone;

  public ObjectEditor(Object o, Frame p) {
    super(o, 460, 300, p);
  }

  public ObjectEditor(Object o, Dialog p) {
    super(o, 460, 300, p);
  }

  protected void buildMainPanel(JPanel mp) {
    mp.setLayout(new BoxLayout(mp, BoxLayout.Y_AXIS));
    Object o = (Object) model;

    JPanel topPanel = new JPanel(new GridLayout(1, 2));

    // add name field
    JPanel namePanel = new JPanel(new FlowLayout(FlowLayout.LEFT));
    namePanel.add(new JLabel("Name:"));
    nameField = new JTextField(o.getName(), 15);
    namePanel.add(nameField);

    // add type field
    JPanel typePanel = new JPanel(new FlowLayout(FlowLayout.LEFT));
    typePanel.add(new JLabel("Typ:"));
    typeField = new JTextField(o.getType(), 15);
    typePanel.add(typeField);

    // Listeditorpanel bekommt eine (tiefe!) Kopie der Liste, damit beim
    // Abbrechen alle Änderungen verschwinden.
    fieldsClone = (Vector) o.getFields().clone();  // leider flacher Klon!
    for (int i = 0; i < fieldsClone.size(); i++) {
      fieldsClone.set(i, ((Field) fieldsClone.get(i)).clone());
    }

    // alles zusammenbauen
    topPanel.add(namePanel);
    topPanel.add(typePanel);
    mp.add(topPanel);
    mp.add(new ListEditorPanel(fieldsClone, "Felder", new FieldView(), this));
```

```
  }

  protected void update() {
    Object o = (Object) model;
    o.setName(nameField.getText());
    o.setType(typeField.getText());
    o.setFields(fieldsClone);
  }

}
```

MethodEditor.java

```
package algorithmen.umleditor;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

/**
 * Editor für eine Methode.
 */
public class MethodEditor extends Editor {

  protected JTextField nameField;
  protected JTextField typeField;
  protected JCheckBox staticField;
  protected JCheckBox abstractField;
  protected JComboBox accessField;
  protected Vector fieldsClone;

  public MethodEditor(Method m, Frame p) {
    super(m, 460, 300, p);
  }

  public MethodEditor(Method m, Dialog p) {
    super(m, 460, 300, p);
  }

  protected void update() {
    Method m = (Method) model;
    m.setName(nameField.getText());
    m.setStatic(staticField.isSelected());
    m.setAbstract(abstractField.isSelected());
    m.setType(typeField.getText());
    m.setVisibility((Visibility) accessField.getSelectedItem());
    m.setParameter(fieldsClone);
  }

  protected void buildMainPanel(JPanel mp) {
    mp.setLayout(new BoxLayout(mp, BoxLayout.Y_AXIS));
    Method m = (Method) model;

    // topPanel mit 4 kleinen Feldern
    JPanel topPanel = new JPanel(new GridLayout(2, 2));

    // add name field
    JPanel namePanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
    namePanel.add(new JLabel("Name:"));
    nameField = new JTextField(m.getName(), 15);
    namePanel.add(nameField);

    // add static/abstract field
    JPanel staticPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
    staticPanel.add(new JLabel("statisch:"));
    staticField = new JCheckBox();
```

```
    staticField.setSelected(m.isStatic());
    staticPanel.add(staticField);
    staticPanel.add(new JLabel("abstrakt:"));
    abstractField = new JCheckBox();
    abstractField.setSelected(m.isAbstract());
    staticPanel.add(abstractField);

    // add type field
    JPanel typePanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
    typePanel.add(new JLabel("Typ:"));
    typeField = new JTextField(m.getType(), 15);
    typePanel.add(typeField);

    // add access field
    JPanel accessPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
    accessPanel.add(new JLabel("Zugriffsklasse:"));
    accessField = new JComboBox(Visibility.getVisibilities());
    accessField.setSelectedItem(m.getVisibility());
    accessPanel.add(accessField);

    // Listeditorpanel bekommt eine (tiefe!) Kopie der Liste, damit beim
    // Abbrechen alle Änderungen verschwinden.
    fieldsClone = (Vector) m.getParameter().clone();  // leider flacher Klon!
    for (int i = 0; i < fieldsClone.size(); i++) {
      fieldsClone.set(i, ((Field)fieldsClone.get(i)).clone());
    }

    // alle Teile zusammenbauen
    topPanel.add(namePanel);
    topPanel.add(staticPanel);
    topPanel.add(typePanel);
    topPanel.add(accessPanel);
    mp.add(topPanel);
    mp.add(new ListEditorPanel(fieldsClone, "Felder", new FieldView(), this));
  }
}
```

FieldEditor.java

```
package algorithmen.umleditor;

import java.awt.*;
import javax.swing.*;

/**
 * Editor für ein Datenfeld.
 */
public class FieldEditor extends Editor {

  protected JTextField nameField;
  protected JTextField typeField;
  protected JTextField valueField;
  protected JCheckBox staticField;
  protected JComboBox accessField;

  public FieldEditor(Field f, Frame p) {
    super(f, 460, 150, p);
  }

  public FieldEditor(Field f, Dialog p) {
    super(f, 460, 150, p);
  }

  protected void update() {
    Field f = (Field) model;
    f.setName(nameField.getText());
    f.setStatic(staticField.isSelected());
    f.setType(typeField.getText());
    f.setVisibility((Visibility) accessField.getSelectedItem());
    f.setValue(valueField.getText());
  }

  protected void buildMainPanel(JPanel mp) {
    mp.setLayout(new GridLayout(3, 2));
    Field f = (Field) model;

    // add name field
    JPanel namePanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
    namePanel.add(new JLabel("Name:"));
    nameField = new JTextField(f.getName(), 15);
    namePanel.add(nameField);
    mp.add(namePanel);

    // add static field
    JPanel staticPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
    staticPanel.add(new JLabel("statisch:"));
    staticField = new JCheckBox();
    staticField.setSelected(f.isStatic());
    staticPanel.add(staticField);
    mp.add(staticPanel);

    // add type field
    JPanel typePanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
```

```
    typePanel.add(new JLabel("Typ:"));
    typeField = new JTextField(f.getType(), 15);
    typePanel.add(typeField);
    mp.add(typePanel);

    // add access field
    JPanel accessPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
    accessPanel.add(new JLabel("Zugriffsklasse:"));
    accessField = new JComboBox(Visibility.getVisibilities());
    accessPanel.add(accessField);
    mp.add(accessPanel);

    // add value field
    JPanel valuePanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
    valuePanel.add(new JLabel("Wert:"));
    valueField = new JTextField(f.getValue(), 15);
    valuePanel.add(valueField);
    mp.add(valuePanel);


  }

}
```

Arrow.java

```
package algorithmen.umleditor;

import java.awt.*;

/**
 * Beschreibt einen Pfeil zwischen zwei UmlElement’s.
 */
public class Arrow extends UmlElement {

  protected ArrowType type;
  protected UmlElement sender;      // Objekt, an dem der Pfeil beginnt
  protected UmlElement receiver;    // Objekt, an dem der Pfeil endet

  public Arrow(UmlElement send, UmlElement recv, ArrowType t) {
    super("");
    sender = send;
    receiver = recv;
    type = t;
  }

  public Arrow(UmlElement send, UmlElement recv) {
    this(send, recv, ArrowType.REFERENCE);
  }

  /**
   * Gibt das Sender-Objekt des Pfeils zurück.
   */
  public UmlElement getSender() {
    return sender;
  }

  /**
   * Setzt das Sender-Objekt des Pfeils.
   */
  public void setSender(UmlElement s) {
    sender = s;
  }

  /**
   * Gibt das Receiver-Objekt des Pfeils zurück.
   */
  public UmlElement getReceiver() {
    return receiver;
  }

  /**
   * Setzt das Receiver-Objekt des Pfeils.
   */
  public void setReceiver(UmlElement r) {
    receiver = r;
  }

  /**
   * Gibt den Pfeiltyp zurück.
```

```
   */
  public ArrowType getType() {
    return type;
  }

  /**
   * Setzt den Pfeiltyp.
   */
  public void setType(ArrowType t) {
    type = t;
  }
}
```

ArrowType.java

```
package algorithmen.umleditor;

import java.io.*;

/**
 * Enumeration type for the possible arrow types
 */
public class ArrowType implements Serializable {

  private String name;

  /**
   * Creates a new instance of ArrowType with a given String.
   * Since it is private, only the predefined values can be used.
   */
  private ArrowType(String s) {
    name = s;
  }

  public String toString() {
    return name;
  }

  /**
   * Ist true bei vertikalen Beziehungen (Vererbung, Implementierung),
   * false bei den anderen (horizontalen).
   */
  public boolean isVertical() {
    if (equals(EXTENSION) || equals(IMPLEMENTATION)) {
      return true;
    } else {
      return false;
    }
  }

  /**
   * Liefert ein Array mit allen Zugriffswerten
   */
  public static ArrowType[] getArrowTypes() {
    return list;
  }

  /**
   * Liste der möglichen Werte.
   * Im Gegensatz zum Enumeration-Pattern nicht final, da sie durch
   *   Serialisierung neu erzeugt werden sollen.
   */
  public static  ArrowType ASSOCIATION = new ArrowType("association");
  public static  ArrowType REFERENCE = new ArrowType("reference");
  public static  ArrowType RELATION = new ArrowType("relation");
  public static  ArrowType AGGREGATION = new ArrowType("aggregation");
  public static  ArrowType EXTENSION = new ArrowType("extension");
  public static  ArrowType IMPLEMENTATION = new ArrowType("implementation");
```

```
  protected static ArrowType[] list =
  {ASSOCIATION, REFERENCE, RELATION, AGGREGATION, EXTENSION, IMPLEMENTATION};


  /**
   * Serialisierung ist nicht automatisch für static Felder.
   * Daher müssen die Methoden writeObject und readObject geschrieben werden.
   */
  private void writeObject(ObjectOutputStream out) throws IOException {
    out.defaultWriteObject();
    out.writeObject(ASSOCIATION);
    out.writeObject(REFERENCE);
    out.writeObject(RELATION);
    out.writeObject(AGGREGATION);
    out.writeObject(EXTENSION);
    out.writeObject(IMPLEMENTATION);
  }

  private void readObject(ObjectInputStream in)
  throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    ASSOCIATION = (ArrowType) in.readObject();
    REFERENCE = (ArrowType) in.readObject();
    RELATION = (ArrowType) in.readObject();
    AGGREGATION = (ArrowType) in.readObject();
    EXTENSION = (ArrowType) in.readObject();
    IMPLEMENTATION = (ArrowType) in.readObject();
  }


  /*
   * Vergleicht zwei ArrowType-Objekte.
   * Dies ist (im Gegensatz zum normalen Enumeration-Pattern) nötig,
   *   da die Objekte durch Serialisierung neu erzeugt werden können,
   *   was zu neuen Adressen führt, also zum Nicht-Funktionieren von ==
   */
  public boolean equals(ArrowType type) {
    return name.equals(type.toString());
  }

  /*
   * Wird equals geändert, muss auch hashCode angepasst werden, damit gleiche
   * Elemente auch denselben Hashcode produzieren.
   */

  public int hashCode() {
    return name.hashCode();
  }
}
```

ArrowView.java

```
package algorithmen.umleditor;

import java.awt.*;

/**
 * View eines UmlArrow.
 * Zuständig für die graphische Darstellung eines Pfeils.
 */
public class ArrowView extends View implements Observer {

  protected View start;
  protected View end;

  protected int tipWidth;  // Breite der Ofeilspitze
  protected int tipLength;  // L?nge der Pfeilspitze
  protected int topOffset;  // Abstand des (horiz.) Pfeils von der oberen Linie
  protected Polygon pol;
  // enth?lt Anfangspunkt, rechte Pfeilspitze, Endpunkte, linke Pfeilspitze
  protected Polygon pol3;  // Pfeilspitze, Teil von pol

  protected static BasicStroke normal;   // Zeichenstil für normale
  protected static BasicStroke dashed;   // und für gestrichelte Linien

  /**
   * Erzeugt eine Pfeilansicht zwischen den Objekten, die durch die Views
   * start und end repräsentiert werden.
   */
  public ArrowView(Arrow model, View start, View end) {
    super(model, new Point());    // Anfangspunkt muss erst berechnet werden
    this.start = start;
    this.end = end;
    tipWidth = 10;
    tipLength = 10;
    topOffset = 15;

    computeCoordinates();
    setupStrokes();

    // achte auf Änderungen der UmlObjekte
    start.addObserver(this);
    end.addObserver(this);
  }

  public Editor createEditor(UmlElement m, Frame p) {
    return new ArrowEditor((Arrow) m, p);
  }

  public Editor createEditor(UmlElement m, Dialog p) {
    return new ArrowEditor((Arrow) m, p);
  }

  /**
   * Problem: Arrow bezieht sich immer auf zwei UmlElement’s,
   * es gibt daher kein Standard-Objekt.
```

```
 * Um dem Interface Gen?ge zu tun, werden einfach zwei Standard-Objekte
 * miterzeugt.
 */
public UmlElement createStandardObject() {
  return new Arrow(new Class(), new Class());
}

/**
 * Pr?ft, ob der Punkt p im umschlie?enden Polygon des Objekts liegt.
 */
public boolean contains(Point p) {
  return pol.contains(p);
}

public void draw(Graphics g) {
  Arrow a = (Arrow) model;
  ArrowType type = a.getType();
  Graphics2D g2 = (Graphics2D) g;

  computeCoordinates();
  // lazy initialisation
  // nötig, da Strokes nicht Serializable sind.
  if (normal == null) {
     setupStrokes();
  }

  // erst die Linie
  if (type.equals(ArrowType.IMPLEMENTATION) || type.equals(ArrowType.RELATION)) {
    g2.setStroke(dashed);
    g2.drawLine(pol.xpoints[0], pol.ypoints[0], pol.xpoints[2], pol.ypoints[2]);
    g2.setStroke(normal);
  } else {
    g2.drawLine(pol.xpoints[0], pol.ypoints[0], pol.xpoints[2], pol.ypoints[2]);
  }

  // dann die Spitze
  if (type.equals(ArrowType.REFERENCE) || type.equals(ArrowType.RELATION)) {
    g2.drawLine(pol.xpoints[2], pol.ypoints[2], pol.xpoints[1], pol.ypoints[1]);
    g2.drawLine(pol.xpoints[2], pol.ypoints[2], pol.xpoints[3], pol.ypoints[3]);
  } else if (!type.equals(ArrowType.ASSOCIATION) ) {
    g2.setPaint(Color.WHITE);
    g2.fill(pol3);
    g2.setPaint(Color.BLACK);
    g2.draw(pol3);
  } // ASSOCIATION hat keinen Pfeil
}

/**
 * Berechnet Punkte des Pfeils.
 */
protected void computeCoordinates() {
  Arrow a = (Arrow) model;

  // 4 Punkte für den Pfeil, definieren auch das Polygon für Maus-Treffer
  int[] x = new int[4];
  int[] y = new int[4];
  pol = new Polygon(x, y, 4);
```

```
// Dreieck für die Pfeilspitze bzw. Raute
if (a.getType().equals(ArrowType.AGGREGATION)) {
  pol3 = new Polygon(x, y, 4);
} else {
  pol3 = new Polygon(x, y, 3);
}

// berechne Start- und Endpunkt
if (a.getType().isVertical()) {
  // Pfeile beginnen und enden auf der Mitte der oberen bzw. unteren
  // Begrenzung des Objekts
  pol.xpoints[0] = start.getPosition().x + start.getSize().width/2;
  pol.ypoints[0] = start.getPosition().y;
  pol.xpoints[2] = end.getPosition().x + end.getSize().width/2;
  pol.ypoints[2] = end.getPosition().y + end.getSize().height;

} else {
  // Pfeile starten rechts oben und enden links oben
  pol.xpoints[0] = start.getPosition().x + start.getSize().width;
  pol.ypoints[0] = start.getPosition().y + topOffset;
  pol.xpoints[2] = end.getPosition().x;
  pol.ypoints[2] = end.getPosition().y + topOffset;
}

// berechne Enden der Pfeilspitze
// e1 = Einheitsvektor in Pfeilrechtung
// e2 = Einheitsvektor orthogonal dazu
double e1x = pol.xpoints[2] - pol.xpoints[0];
double e1y = pol.ypoints[2] - pol.ypoints[0];
double eabs = Math.sqrt(e1x*e1x + e1y*e1y);
e1x /= eabs;
e1y /= eabs;
double e2x = e1y;
double e2y = -e1x;

if (! (a.getType().equals(ArrowType.AGGREGATION) )){
  pol.xpoints[1] = (int)(pol.xpoints[2] + tipWidth*e2x - tipLength*e1x);
  pol.ypoints[1] = (int)(pol.ypoints[2] + tipWidth*e2y - tipLength*e1y);
  pol.xpoints[3] = (int)(pol.xpoints[2] - tipWidth*e2x - tipLength*e1x);
  pol.ypoints[3] = (int)(pol.ypoints[2] - tipWidth*e2y - tipLength*e1y);
  pol.invalidate();  // interne Caches von Polygon aktualisieren

  pol3.xpoints[0] = pol.xpoints[1];
  pol3.ypoints[0] = pol.ypoints[1];
  pol3.xpoints[1] = pol.xpoints[2];
  pol3.ypoints[1] = pol.ypoints[2];
  pol3.xpoints[2] = pol.xpoints[3];
  pol3.ypoints[2] = pol.ypoints[3];
  pol3.invalidate();
} else {
  // Aggregation wird mit Raute am Anfang dargestellt
  pol.xpoints[1] = (int)(pol.xpoints[0] + tipWidth*e2x + tipLength*e1x);
  pol.ypoints[1] = (int)(pol.ypoints[0] + tipWidth*e2y + tipLength*e1y);
  pol.xpoints[3] = (int)(pol.xpoints[0] - tipWidth*e2x + tipLength*e1x);
  pol.ypoints[3] = (int)(pol.ypoints[0] - tipWidth*e2y + tipLength*e1y);
  pol.invalidate();  // interne Caches von Polygon aktualisieren
```

```
      pol3.xpoints[0] = pol.xpoints[0];
      pol3.ypoints[0] = pol.ypoints[0];
      pol3.xpoints[1] = pol.xpoints[1];
      pol3.ypoints[1] = pol.ypoints[1];
      pol3.xpoints[2] = (int)(pol.xpoints[0] + 2*tipLength*e1x);
      pol3.ypoints[2] = (int)(pol.ypoints[0] + 2*tipLength*e1y);
      pol3.xpoints[3] = pol.xpoints[3];
      pol3.ypoints[3] = pol.ypoints[3];
      pol3.invalidate();
    }
  }

  public void update()  {
     computeCoordinates();
  }

  protected void setupStrokes() {
    float dash1[] = {10.0f};
    normal = new BasicStroke();
    dashed = new BasicStroke(1.0f, BasicStroke.CAP_BUTT,
                    BasicStroke.JOIN_MITER, 10.0f, dash1, 0.0f);
  }

}
```

ArrowEditor.java

```
package algorithmen.umleditor;

import java.awt.*;
import javax.swing.*;

/**
 * Editor für einen Pfeil.
 */
public class ArrowEditor extends Editor {

  protected JTextField nameField;
  protected JComboBox typeField;

  public ArrowEditor(Arrow f, Frame p) {
    super(f, 280, 150, p);
  }

  public ArrowEditor(Arrow f, Dialog p) {
    super(f, 280, 150, p);
  }

  protected void update() {
    Arrow a = (Arrow) model;
    a.setName(nameField.getText());
    a.setType((ArrowType) typeField.getSelectedItem());
  }

  protected void buildMainPanel(JPanel mp) {
    mp.setLayout(new BoxLayout(mp, BoxLayout.Y_AXIS));
    Arrow a = (Arrow) model;

    // add name field
    JPanel namePanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
    namePanel.add(new JLabel("Name:"));
    nameField = new JTextField(a.getName(), 15);
    namePanel.add(nameField);
    mp.add(namePanel);

    // add access field
    JPanel typePanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
    typePanel.add(new JLabel("Pfeiltyp:"));
    typeField = new JComboBox(ArrowType.getArrowTypes());
    typeField.setSelectedItem(a.getType());
    typePanel.add(typeField);
    mp.add(typePanel);
  }
}
```

Observer.java

```
package algorithmen.umleditor;

/**
 * Teil des Observer-Patterns:
 * Ein Observer meldet sich bei einer Observable an und wird über
 * seine notify-Methode von Änderungen benachrichtigt.
 */
public interface Observer {
  void update();
}
```

View.java

```
package algorithmen.umleditor;

import java.awt.*;
import javax.swing.*;
import java.io.*;
import java.beans.*;

/**
 * Basisklasse aller Views von UML-Objekten.
 * Enthält das Modell und die Position
 *     (= linke obere Ecke des umgebenden Rechtecks)
 * Die Größe ergibt sich aus dem Modell.
 */
abstract public class View implements Serializable {
  protected Point pos;
  protected UmlElement model;
  protected Dimension size;    // aus den Strings berechnet und gecached

  /** Abstand zwischen Strings oder Strings und Linien in Pixel */
  protected int sepSize = 5;

  /** div. Font, werden lazy initialisiert,
   * da dazu Graphics-Objekt nötig */
  protected Font standardFont;
  protected Font cursiveFont;
  protected Font boldFont;
  protected Font curBoldFont;

  /**
   * Erzeugt einen View für das model an der Position p.
   */
  public View(UmlElement model, Point p) {
    this.model = model;
    pos = p;
    size = new Dimension();

    // Liste der Observer bereitmachen
    observerSupport = new PropertyChangeSupport(this);
  }

  /**
   * Zeichnet das Objekt.
   */
  abstract public void draw(Graphics g);

  /**
   * Erzeugt einen Editor für das Objekt.
   * Er ist modal bzgl. des angegebenen Frames bzw. Dialogs.
   */
  abstract public Editor createEditor(UmlElement m, Frame p);
  abstract public Editor createEditor(UmlElement m, Dialog p);

  /**
   * Erzeugt ein Standard-UmlElement.
```

```
 * Dies ist Teil des FactoryMethod Patterns und wird verwendet, damit
 *   der ListEditor neue Elemente verschiedenen Typs erzeugen kann.
 */
abstract public UmlElement createStandardObject();

/**
 * Holt die aktuelle Position eines Objekts.
 */
public Point getPosition() {
  return pos;
}

/**
 * Setzt die aktuelle Position eines Objekts.
 */
public void setPosition(Point p) {
  pos = p;
  notifyObservers();
}

/**
 * Holt das aktuelle Modell des Views.
 */
public UmlElement getModel() {
  return model;
}

/**
 * Setzt das aktuelle Modell des Views.
 */
public void setModel(UmlElement m) {
  model = m;
  notifyObservers();
}

/**
 * Holt die Größe des Objekts.
 * Gesetzt wird die Größe vom View selbst durch Bestimmung der Teile,
 *    nicht von außen vorgegeben.
 */
public Dimension getSize() {
  return size;
}


/**
 * Prüft, ob der Punkt p im umschließenden Rechteck des Objekts liegt.
 */
public boolean contains(Point p) {
  Rectangle r = new Rectangle(pos, size);
  return r.contains(p);
}

/**
 * Verschiebt das Objekt um (dx, dy).
 */
public void translate(int dx, int dy) {
```

```
    pos.translate(dx, dy);
    notifyObservers();
  }

  // Hilfsmethoden

  /**
   * Gibt die Höhe von Strings bezogen auf den verwendeten Font zurück.
   */
  protected int getStringHeight(Graphics g) {
    return g.getFontMetrics().getHeight();
  }

  /**
   * Gibt die Breite eines Strings bezogen auf den verwendeten Font zurück.
   */
  protected int getStringWidth(String s, Graphics g) {
    return g.getFontMetrics().stringWidth(s);
  }

  /**
   * Fonts "lazy" initialisieren
   */
  protected void initFonts(Graphics g) {
    if (standardFont == null) {
      standardFont = g.getFont();
      cursiveFont = standardFont.deriveFont(Font.ITALIC);
      boldFont = standardFont.deriveFont(Font.BOLD);
      curBoldFont = standardFont.deriveFont(Font.ITALIC | Font.BOLD);
    }
  }


  // Infrastruktur für das Observer-Pattern
  // benutzt java.beans.PropertyChangeSupport

  protected PropertyChangeSupport observerSupport;

  /**
   * Registriert einen Observer.
   */
  public void addObserver(Observer o) {
    observerSupport.addPropertyChangeListener(new ObserverAdapter(o));
  }

  /**
   * Informiert alle Observer.
   */
  protected void notifyObservers() {
    observerSupport.firePropertyChange("", null, null);
  }
}
```

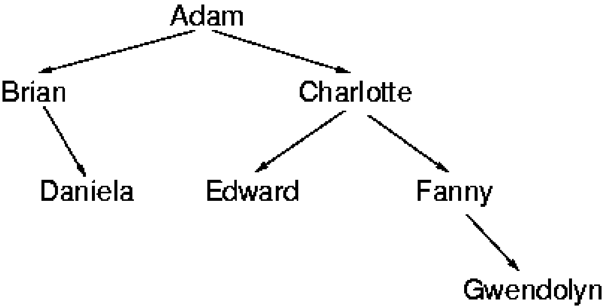
ObserverAdapter.java

```
package algorithmen.umleditor;

import java.beans.*;

/**
 * Adapter to make an Observer fit into a PropertyChangeListener
 */
public class ObserverAdapter implements PropertyChangeListener {

  Observer o;

  /**
   * Creates an ObserverAdapter from a given Observer
   */
  public ObserverAdapter(Observer o) {
    this.o = o;
  }

  /*
   * Relays notification to the observer
   */
  public void propertyChange(PropertyChangeEvent propertyChangeEvent) {
    o.update();
  }

}
```

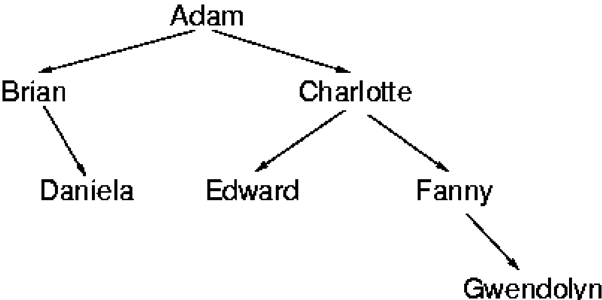
Applets

- Bubblesort
- Quicksort
- TestUmlManager
- UmlGui
- Matrixmultiplikation
- Integer-Stack
- Klammerausdrücke
- Integer-Stack mit Liste
- Familienstammbaum
- Vergleich von Baum und AVLBaum
- Hash-Tabellen

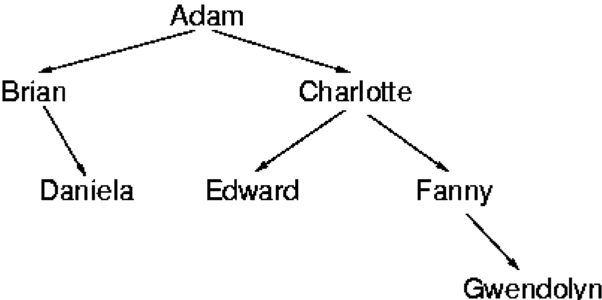
Bubblesort



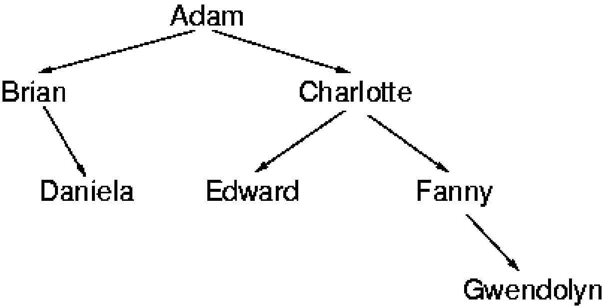
Quicksort



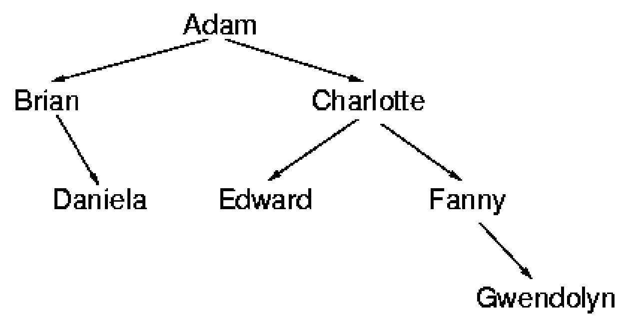
TestUmlManager



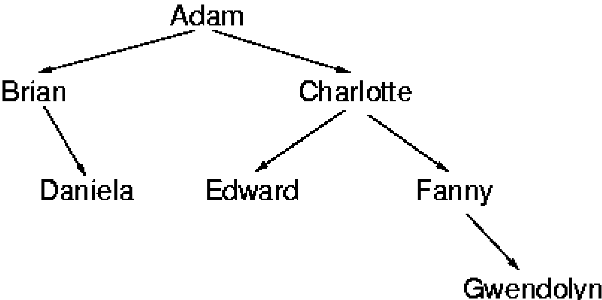
UML-Editor



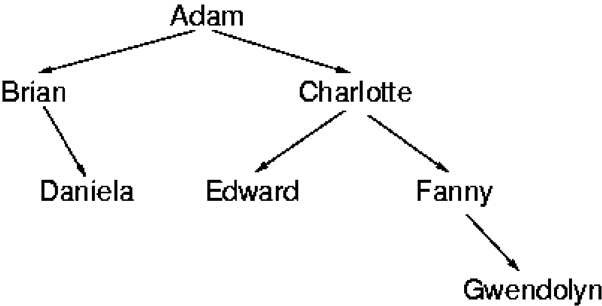
TestMatrixMultiplikation



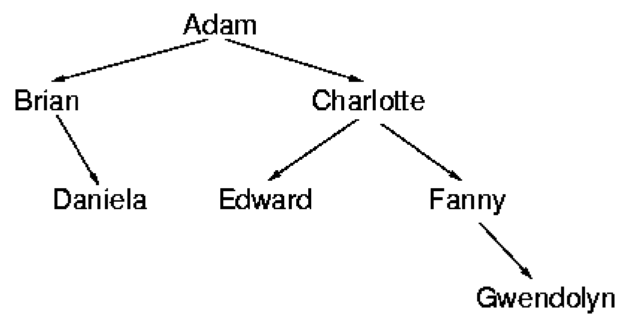
Integer-Stack



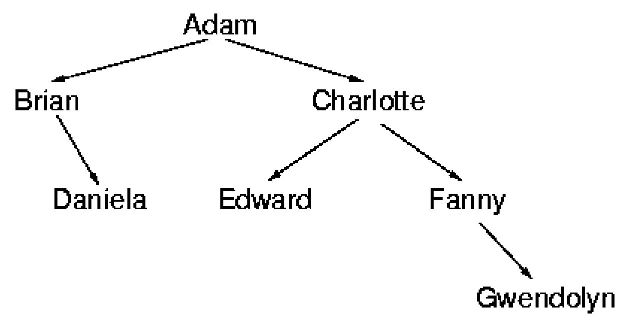
Klammerausdrücke



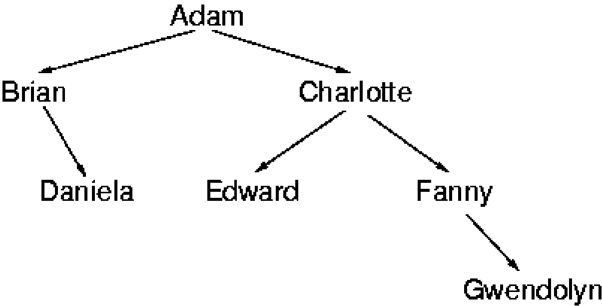
Integer-Stack mit Liste



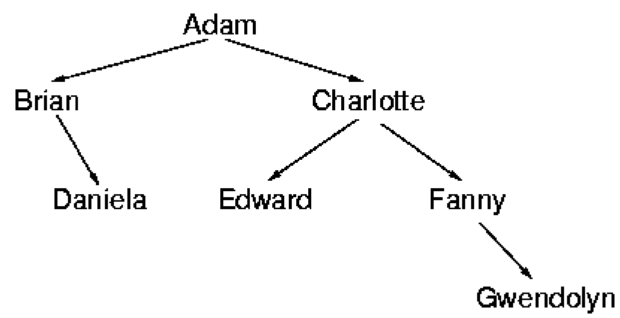
Familienstammbaum



Vergleich von Baum und AVLBaum



Hash-Tabellen



Lösung von Aufgabe 1

- Erläuterungen zum Programm:
  - in allen Routinen fehlen Tests, ob die Dimensionen der Matrizen passen,
  - viele Kopien von Matrizen → sehr viel Speicher verschenkt.
- Sourcen:
  - Matrix.java
  - Testprogramm mit Zeitmessung TestMatrixMultiplikation.java
- als lauffähiges Applet

Matrix.java

```
public class Matrix {
  // mathematische Matrix

  private double[][] arr;

  public Matrix(int m, int n) {
    // erzeuge nxm-Matrix, mit 0 vorbesetzt
    arr = new double[m][n];
  }

  public Matrix(Matrix a) {
    // erzeuge Matrix als Kopie von a

    int m = a.getColumnDimension();
    int n = a.getRowDimension();
    arr = new double[m][n];

    for (int i = 0; i < m; i++) {
      for (int j = 0; j < n; j++) {
        arr[i][j] = a.arr[i][j];
      }
    }
  }

  public int getColumnDimension() {
    // Spaltenlänge
    return arr.length;
  }

  public int getRowDimension() {
    // Zeilenlänge
    return arr[0].length;
  }

  public double get(int i, int j) {
    // hole Array-Element A_ij
    return arr[i][j];
  }

  public void set(int i, int j, double s) {
    // setze Array-Element A_ij
    arr[i][j] = s;
  }

  public Matrix getMatrix(int i0, int i1, int j0, int j1) {
    // erzeugt eine Teilmatrix A(i0..i1, j0..j1)

    int m = i1 - i0 + 1;
    int n = j1 - j0 + 1;

    Matrix teil = new Matrix(m, n);

    for (int i = 0; i < m; i++) {
      for (int j = 0; j < n; j++) {
```

```
      teil.arr[i][j] = arr[i + i0][j + j0];
    }
  }
  return teil;
}

void setMatrix(int i0, int j0, Matrix teil) {
  // belegt die Matrix mit einer Teilmatrix

  int m = teil.getColumnDimension();
  int n = teil.getRowDimension();

  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      arr[i+ i0][j + j0] = teil.arr[i][j];
    }
  }
}

public static Matrix matmult(Matrix a, Matrix b) {
  // einfache Matrix-Multiplikation

  // Achtung: Dies ist eine Version zum Zweck der Zeitmessung, daher
  //   verzichtet sie auf Dimensions-Überprüfungen.

  int m = a.getColumnDimension();
  int l = a.getRowDimension();
  int n = b.getRowDimension();

  Matrix c = new Matrix(n, m);

  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      c.arr[i][j] = 0.0;
      for (int k = 0; k < l; k++) {
        c.arr[i][j] += a.arr[i][k] * b.arr[k][j];
      }
    }
  }

  return c;
}

public static Matrix matmultStrassen(Matrix a, Matrix b) {
  return matmultStrassenMitCutoff(a, b, 1);
}

public static Matrix matmultStrassenMitCutoff(Matrix a, Matrix b, int cutoff) {
  // Matrix-Multiplikation nach dem Strassen-Algorithmus
  // für Matrizen mit Dimension >= cutoff wird normal multipliziert

  // Achtung: Dies ist eine Version zum Zweck der Zeitmessung, daher
  //   - verzichtet sie auf Dimensions-Überprüfungen,
  //    - geht sie von quadratischen Matrizen mit einer Zweierpotenz als Dimension aus

  int m = a.getColumnDimension();

  Matrix c = new Matrix(m, m);
```

```
// Abbruchbedingung der Rekursion: m <= cutoff
if (m <= cutoff) {
  // normal multiplizieren
  c = matmult(a, b);
  return c;
}

// ansonsten rekursiv a la Strassen

// Teilmatrizen herausschneiden
int dim = m/2;     // sollte ohne Rest aufgehen, da m Zweierpotenz
Matrix a11 = a.getMatrix(  0, dim-1,   0, dim-1);
Matrix a12 = a.getMatrix(  0, dim-1, dim,   m-1);
Matrix a21 = a.getMatrix(dim,   m-1,   0, dim-1);
Matrix a22 = a.getMatrix(dim,   m-1, dim,   m-1);
Matrix b11 = b.getMatrix(  0, dim-1,   0, dim-1);
Matrix b12 = b.getMatrix(  0, dim-1, dim,   m-1);
Matrix b21 = b.getMatrix(dim,   m-1,   0, dim-1);
Matrix b22 = b.getMatrix(dim,   m-1, dim,   m-1);

// Matrizen m1 .. m7 berechnen
// dazu zwei Hilfsmatrizen d1, d2 für Zwischenwerte
Matrix d1 = a12.minus(a22);
Matrix d2 = b21.plus(b22);
Matrix m1 = matmultStrassen(d1, d2);

d1 = a11.plus(a22);
d2 = b11.plus(b22);
Matrix m2 = matmultStrassen(d1, d2);

d1 = a11.minus(a21);
d2 = b11.plus(b12);
Matrix m3 = matmultStrassen(d1, d2);

d1 = a11.plus(a12);
Matrix m4 = matmultStrassen(d1, b22);

d1 = b12.minus(b22);
Matrix m5 = matmultStrassen(a11, d1);

d1 = b21.minus(b11);
Matrix m6 = matmultStrassen(a22, d1);

d1 = a21.plus(a22);
Matrix m7 = matmultStrassen(d1, b11);

// Teilmatrizen c11 .. c22 bestimmen
Matrix c11 = new Matrix(m1);
c11.plusGleich(m2);
c11.minusGleich(m4);
c11.plusGleich(m6);

Matrix c12 = new Matrix(m4);
c12.plusGleich(m5);

Matrix c21 = new Matrix(m6);
c21.plusGleich(m7);
```

```
  Matrix c22 = new Matrix(m2);
  c22.minusGleich(m3);
  c22.plusGleich(m5);
  c22.minusGleich(m7);

  // Gesamtmatrix zusammensetzen
  c.setMatrix(  0,   0, c11);
  c.setMatrix(  0, dim, c12);
  c.setMatrix(dim,   0, c21);
  c.setMatrix(dim, dim, c22);

  return c;
}

public  Matrix minus(Matrix a) {
  // gibt Differenz von aktueller Matrix und a zurück

  int m = getColumnDimension();
  int n = getRowDimension();
  Matrix c = new Matrix(m, n);

  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      c.arr[i][j] = arr[i][j] - a.arr[i][j];
    }
  }
  return c;
}

public  Matrix plus(Matrix a) {
  // gibt Summe von aktueller Matrix und a zurück

  int m = getColumnDimension();
  int n = getRowDimension();
  Matrix c = new Matrix(m, n);

  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      c.arr[i][j] = arr[i][j] + a.arr[i][j];
    }
  }
  return c;
}

public void plusGleich(Matrix a) {
  // addiert a zur aktuellen Matrix

  for (int i = 0; i < getColumnDimension(); i++) {
    for (int j = 0; j < getRowDimension(); j++) {
      arr[i][j] += a.arr[i][j];
    }
  }
}

public void minusGleich(Matrix a) {
  // subtrahiert a von der aktuellen Matrix
```

```
    for (int i = 0; i < getColumnDimension(); i++) {
      for (int j = 0; j < getRowDimension(); j++) {
        arr[i][j] -= a.arr[i][j];
      }
    }
  }

  public String toString() {
    // Ausgabestring: Zeilen durch Newline getrennt, Werte durch Komma

    String s = "";
    for (int i = 0; i < getColumnDimension(); i++) {
      for (int j = 0; j < getColumnDimension(); j++) {
        s += arr[i][j] + ", ";
      }
      s += "\n";
    }
    return s;
  }
}
```

TestMatrixMultiplikation.java

```
import java.io.*;

public class TestMatrixMultiplikation {
  // Testprogramm für Strassen-Matrixmultiplikation mit Cutoff

  public static void main(String[] args) throws IOException {
    // erzeugt zwei Matrizen mit beliebigen Zahlen
    // multipliziert sie und gibt das Ergebnis ggf. aus

    BufferedReader in = new BufferedReader(
    new InputStreamReader(System.in));

    // Werte vom Benutzer abfragen
    String s;    // String für eingegebene Zeile
    System.out.println("Matrixdimension eingeben:");
    s = in.readLine();
    int dim = Integer.parseInt(s);

    System.out.println("Soll der Strassen-Algorithmus verwendet werden (j/n):");
    s = in.readLine();
    boolean mitStrassen = false;
    if (s.compareTo("j") == 0) {
      mitStrassen = true;
    }

    int cutoff = 0;
    if (mitStrassen) {
      System.out.println("Größte Dimension für normale Multiplikation:");
      s = in.readLine();
      cutoff = Integer.parseInt(s);
    }

    System.out.println("Soll die Rechnung ausgegeben werden (j/n):");
    s = in.readLine();
    boolean mitAusgabe = false;
    if (s.compareTo("j") == 0) {
      mitAusgabe = true;
    }

    // erzeuge die Arrays
    Matrix a = new Matrix(dim, dim);
    Matrix b = new Matrix(dim, dim);

    // fülle die Arrays mit (beliebigen) Zahlen
    for (int i = 0; i < dim; i++) {
      for (int j = 0; j < dim; j++) {
        a.set(i, j, i+j);
        b.set(i, j, i-j);
      }
    }

    // multipliziere die Matrizen und miss die benötigte Zeit
    Matrix c;
    long timeBefore = 0L;
```

```
    long timeAfter = 0L;
    if (mitStrassen) {
      timeBefore = System.currentTimeMillis();
      c = Matrix.matmultStrassenMitCutoff(a, b, cutoff);
      timeAfter = System.currentTimeMillis();
    } else {
      timeBefore = System.currentTimeMillis();
      c = Matrix.matmult(a, b);
      timeAfter = System.currentTimeMillis();
    }
    double timeSpent = (timeAfter - timeBefore)/1000.0;  // in s
    System.out.println("Zeit zum Multiplizieren: " + timeSpent + "s");

    // gib die Rechnung aus, falls gewünscht
    if (mitAusgabe) {
      System.out.println("Matrix A:\n" + a);
      System.out.println("Matrix B:\n" + b);
      System.out.println("Matrix C = A*B:\n" + c);
    }
  }

}
```

Lösung von Aufgabe 2

- Die Zuweisung
  - a = aNeu;
- überschreibt den Zeiger auf das alte Array, es liegt jetzt verloren im Heap und wird "bei nächster Gelegenheit" vom Garbage-Collector eingesammelt.
- Sourcen:
  - IntStack.java
- als lauffähiges Applet

IntStack.java

```
import java.io.*;

public class IntStack {
  // Stack mit automatischem Wachstum über Verdopplung

  private int[] a;    // Liste der Elemente
  private int last;   // Zahl der Elemente

  public IntStack() {
    // erzeugt einen Stack von Standardgröße
    this(100);
  }

  public IntStack(int size) {
    // erzeugt einen Stack der Größe size
    a = new int[size];
    last = 0;
  }

  public void push(int d) {
    // legt d auf dem Stack ab

    // ggf. internen Speicher vergrößern
    if (last == a.length) {
      int[] aNeu = new int[2*a.length];
      for (int i = 0; i < a.length; i++) {
        aNeu[i] = a[i];
      }
      a = aNeu;  // a ist dann reif für den Garbage Collector
    }

    a[last] = d;
    last++;
  }

  public int pop() {
    // holt oberstes Element vom Stack
    last--;
    return a[last];
  }

  public boolean empty() {
    // true, wenn kein Element implements Stack ist
    return (last == 0);
  }

  public double peek() {
    // gibt oberstes Element aus, ohne es zu entfernen
    return a[last - 1];
  }

  public static void main(String[] args) throws IOException {
    // kleines Testsprogramm
```

```
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

    // Größe des Stackbuffers vom Benutzer abfragen
    String s;    // String für eingegebene Zeile
    System.out.println("Anfangsgröße des Stacks eingeben:");
    s = in.readLine();
    int size = Integer.parseInt(s);

    IntStack meinStack = new IntStack(size);

    // Stack füllen
    System.out.println("Positive Zahlen eingeben (beenden durch -1):");
    int neuerWert;
    do {
      s = in.readLine();
      neuerWert = Integer.parseInt(s);
      meinStack.push(neuerWert);
    } while (neuerWert >= 0);

    // letzten Wert wieder herausnehmen, war ja Ende-Zeichen
    meinStack.pop();

    // Ausgabe des Stacks
    System.out.println("Werte im Stack:");
    while (! meinStack.empty()) {
       System.out.println(meinStack.pop());
    }
  }
}
```

Lösung von Aufgabe 3

- Das Zerlegen eines Strings in seine logischen Bestandteile (Tokens) erledigt i.w. die Klasse StringTokenizer. Sie bekommt eine Liste von Trennzeichen und gibt dann jedesmal mit nextToken das nächste gefundene Token (Trennzeichen oder Bereich dazwischen) zurück.
- Die weitergehende Analyse erledigt ArithmetikTokenizer, der seiner Elternklasse die meiste Arbeit übergibt. Die gefundenen Tokens sind
  - Klammern oder Operatoren, die als entsprechende Tokens geliefert werden,
  - Zahlen, die gleich mit ihrem Wert versehen werden,
  - Leerzeichen, die überlesen werden, d.h. es wird das nächste Token gewählt. Hier gäbe es Probleme mit Leerzeichen am Ende der Eingabe, wenn dies nicht durch das s.trim() im Konstruktor gleich verhindert würde.
  - etwas anderes → Eingabefehler. Wird hier nicht abgefangen, sondern liefert später eine Exception.
- Das Berechnen des Ausdrucks aus den Tokens erfolgt genau wie in der Vorlesung beschrieben. Dabei sorgt das Ausgliedern der Hilfsfunktion applyOp für Übersichtlichkeit. Zu beachten ist noch, dass die pop-Operation von java.util.Stack nur ein Object zurückliefern kann, das man zur Weiterverarbeitung mittels Casting in ein Token umwandeln muss.
- Sourcen:
  - Token.java
  - ArithmeticTokenizer.java
  - ArithmeticEvaluator.java
- als lauffähiges Applet

Token.java

```
public class Token {
  // ein Eingabe-Symbol
  // Möglichkeiten: ( ) + - * / ZAHL
  // Zahl ist (positive) ganze Zahl

  private int   value;   // Wert, falls das Token eine Zahl ist
  private char  symbol;
    // Token ist ein einzelnes Zeichen -> das Zeichen selbst
    // Token ist eine ganze Zahl -> N

  public Token(char c) {
    // erzeugt ein 1-Zeichen-Token
    symbol = c;
    value = 0;
  }

  public Token(int d) {
    // erzeugt ein Zahl-Token
    symbol = 'N';
    value = d;
  }

  public char getSymbol() {
    // gibt das Symbol des Tokens zurück
    return symbol;
  }

  public int getValue() {
    // gibt den Wert des Tokens zurück
    return value;
  }
}
```

ArithmeticTokenizer.java

```
import java.util.*;

public class ArithmeticTokenizer extends StringTokenizer {
  // zerlegt einen Eingabestring in Token

  public ArithmeticTokenizer(String s) {
    // erzeugt einen String-Tokenizer, der bei
    //       +-*/() und Leerzeichen auftrennt.
    // Leerzeichen am Anfang oder Ende werden mit trim gleich beseitigt.

    super(s.trim(), "+-*/() ", true);
  }

  public Token getNextToken() {
    // liefert das nächste Token im Eingabestring zurück.

    // zunächst Leerzeichen überlesen
    String t;
    do {
      t = nextToken();  // liefert eines der Trennzeichen oder Rest

      // Achtung: bei Leerzeichen am Ende gibt es hier Probleme!
      // Daher wurden die mit trim schon implements Konstruktor beseitigt.
    } while ( t.equals(" ") );

    // ok, es ist nicht leer
    if (t.equals("+") || t.equals("-") || t.equals("*") ||
    t.equals("/") || t.equals("(") || t.equals(")") ) {
      // Operator oder Klammer
      return new Token(t.charAt(0));
    } else {
      // sollte eine ganze Zahl sein. Wenn nicht -> Exception!
      int value = Integer.parseInt(t);
      return new Token(value);
    }
  }
}
```

ArithmeticEvaluator.java

```
import java.util.*;
import java.io.*;

public class ArithmeticEvaluator {
  // Klasse zum Berechnen von einfachen Klammerausdrücken

  public static int computeExpression(String s) {
    // gibt den Wert des Klammerausdrucks im String s zurück

    ArithmeticTokenizer tok = new ArithmeticTokenizer(s);
    Stack tokenStack = new Stack();

    while (tok.hasMoreTokens()) {
      // gehe durch die Token durch

      Token nextToken = tok.getNextToken();
      if (nextToken.getSymbol() != ')') {
        tokenStack.push(nextToken);
      } else {
        // ')' gefunden, also arbeiten!

        // 2. Operanden holen
        Token t = (Token) tokenStack.pop();
        int operand2 = t.getValue();

        // Operator holen
        t = (Token) tokenStack.pop();
        char operator = t.getSymbol();

        // 1. Operanden holen
        t = (Token) tokenStack.pop();
        int operand1 = t.getValue();

        // Klammer '(' aus dem Stack beseitigen
        tokenStack.pop();

        // Ergebnis berechnen und auf den Stack packen
        int result = applyOp(operator, operand1, operand2);

        tokenStack.push(new Token(result));
      }
    }
    Token t = (Token) tokenStack.pop();
    return t.getValue();
  }

  private static int applyOp(char op, int d1, int d2) {
    // berechnet die Operation  d1 op d2
    switch (op) {
      case '+': return d1 + d2;
      case '-': return d1 - d2;
      case '*': return d1 * d2;
      case '/': return d1/d2;
      default: throw new IllegalArgumentException();
```

```
    }
  }

  public static void main(String[] args) throws IOException {
    // kleines Testprogramm

    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

    // Eingabestring abfragen
    String s;     // String für eingegebene Zeile
    System.out.println("vollständig geklammerten Ausdruck eingeben:");
    s = in.readLine();

    int result = computeExpression(s);

    System.out.println(s + " = " + result);
  }
}
```

Lösung von Aufgabe 4

- Der ListenKnoten enthält Daten vom Typ int sowie Methoden holeDaten und setzeDaten zum Zugriff darauf.
- Liste wird erweitert um die Methoden holeErstesElement, entferneErstesElement und vornEinfügen.
- Damit kann ein Stack leicht implementiert und das Testprogramm von Aufgabe 2 übernommen werden bis auf die Abfrage der Anfangsgröße, die jetzt überflüssig ist.
- Sourcen:
  - ListenKnoten.java
  - Liste.java
  - IntStack.java
- als lauffähiges Applet

ListenKnoten.java

```
public class ListenKnoten {

  int daten;
  ListenKnoten naechster;  // sind "package" für Liste

  // einfacher Konstruktor
  public ListenKnoten(int n) {
    daten = n;
    naechster = null;   // zeigt noch nirgendwo hin
  }

  // Konstruktor mit Ziel
  public ListenKnoten(int n, ListenKnoten next) {
    daten = n;
    naechster = next;
  }

  public int holeDaten() {
    return daten;
  }

  public void setzeDaten(int d) {
    daten = d;
  }
}
```

Liste.java

```
import java.util.*;

public class Liste {

  public ListenKnoten start;

  public Liste() {
    // Konstruktor für leere Liste
    start = null;
  }

  public void einfügen(ListenKnoten neu) {
    // fügt Knoten neu an die richtige Stelle ein

    // 1. Fall: Liste ist noch leer
    if (start == null) {
      start = neu;
      return;
    }

    // 2. Fall: neues Element kommt an den Anfang der Liste
    if (start.daten > neu.daten) {
      neu.naechster = start;
      start = neu;
      return;
    }

    // 3. Fall: Liste durchlaufen
    ListenKnoten letzter = start;
    ListenKnoten aktuell = start.naechster;

    while ((aktuell != null) && (aktuell.daten < neu.daten)) {
      aktuell = aktuell.naechster;
      letzter = letzter.naechster;
    }

    // gefunden, jetzt einfügen
    letzter.naechster = neu;
    if (aktuell != null) {    // neu ist nicht letztes Element
      neu.naechster = aktuell;
    }
  }

  public void ausdrucken() {
    ListenKnoten aktuell = start;
    while (aktuell != null) {
      System.out.println(aktuell.daten);
      aktuell = aktuell.naechster;
    }
  }

  public ListenKnoten holeErstesElement() {
    // gibt erstes Element der Liste zurück
    return start;
```

```
  }

  public void entferneErstesElement() throws NoSuchElementException {
    // entfernt erstes Element, falls vorhanden
    if (start == null) {
     throw new  NoSuchElementException();
    } else {
      start = start.naechster;
    }
  }

  public void vornEinfügen(ListenKnoten neu) {
    // fügt Knoten neu am Beginn der Liste ein

    if (start == null) {
      // Liste ist noch leer
      start = neu;
      return;
    } else {
      // Liste ist nicht leer, vorne anhängen
      neu.naechster = start;
      start = neu;
      return;
    }
  }
}
```

IntStack.java

```
import java.io.*;

public class IntStack {
  // Stack mit automatischem Wachstum über verkettete Liste

  private Liste liste;

  public IntStack() {
    // erzeugt einen Stack
    liste = new Liste();
  }

  public void push(int d) {
    // legt d auf dem Stack ab
    liste.vornEinfügen(new ListenKnoten(d));
  }

  public int pop() {
    // holt oberstes Element vom Stack
    int ergebnis = liste.holeErstesElement().holeDaten();
    liste.entferneErstesElement();
    return ergebnis;
  }

  public boolean empty() {
    // true, wenn kein Element im Stack ist
    return (liste.holeErstesElement() == null);
  }

  public double peek() {
    // gibt oberstes Element aus, ohne es zu entfernen
    int ergebnis = liste.holeErstesElement().holeDaten();
    return ergebnis;
  }

  public static void main(String[] args) throws IOException {
    // kleines Testsprogramm

    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

    IntStack meinStack = new IntStack();

    // Stack füllen
    System.out.println("Positive Zahlen eingeben (beenden durch -1):");
    String s;
    int neuerWert;
    do {
      s = in.readLine();
      neuerWert = Integer.parseInt(s);
      meinStack.push(neuerWert);
    } while (neuerWert >= 0);

    // letzten Wert wieder herausnehmen, war ja Ende-Zeichen
    meinStack.pop();
```

```
    // Ausgabe des Stacks
    System.out.println("Werte im Stack:");
    while (! meinStack.empty()) {
      System.out.println(meinStack.pop());
    }
  }
}
```

Lösung von Aufgabe 5

- Ein Problem beim Suchen in einem unsortierten Baum besteht darin, dass man bei der Rekursion nicht weiß, wenn das Ziel woanders gefunden wurde. Die einfachste Lösung besteht darin, immer den ganzen Baum zu durchsuchen, nach dem Motto: Das bisschen Mehrarbeit ist kein Problem. Hier soll ein besseres Verfahren vorgestellt werden:
  - Die Einfüge-Methoden geben true zurück, wenn sie das Ziel gefunden haben. In der Rekursion kann in diesem Fall auf eine weitere Suche verzichten werden.
  - Ein weiterer Vorteil: Gibt die oberste Routine false zurück, weiß man, dass sich der gesuchte Wert gar nicht im Baum befindet.
- Zur Verschönerung der Ausgabe wird je nach Tiefe ein Stück eingerückt. Dazu bekommt die rekursive print-Methode ein Argument, das die Einrücktiefe angibt, und bei jeder Rekursion entsprechend vergrößert wird.
- Alles weitere kann wieder den Quellen entnommen werden:
  - BaumKnoten.java, Baum.java
- Applet

BaumKnoten.java

```
public class BaumKnoten {

  public String name;
  public BaumKnoten vater;
  public BaumKnoten mutter;

  BaumKnoten(String meinName) {
    name   = meinName;
    vater  = null;
    mutter = null;
  }
}
```

Baum.java

```
public class Baum {
  public BaumKnoten wurzel;

  // Konstruktor
  // legt gleich den Wurzelknoten an
  public Baum(String kind) {
    wurzel = new BaumKnoten(kind);
  }

  // öffentliche Methoden
  public boolean einfuegeMutter(String nameMutter, String nameKind) {
    return einfuegeMutter(wurzel, nameMutter, nameKind);
  }

  public boolean einfuegeVater(String nameVater, String nameKind) {
    return einfuegeVater(wurzel, nameVater,  nameKind);
  }

  public void print() {
    print(wurzel, 0);
  }

  // Hilfsmethoden
  private boolean einfuegeMutter(BaumKnoten s, String nameMutter,
                                 String nameKind) {
    boolean gefunden;

    // suche Kind

    // ist es schon Knoten s?
    if (s.name.equals(nameKind)) {      // gefunden
      s.mutter = new BaumKnoten(nameMutter);  // Mutter eintragen
      return true;
    }

    // oder ein Vorfahre mütterlicherseits ?
    if (s.mutter != null) {
      gefunden = einfuegeMutter(s.mutter, nameMutter, nameKind);
      if (gefunden) {
        return true;
      }
    }

    // oder ein Vorfahre väterlicherseits ?
    if (s.vater != null) {
      gefunden = einfuegeMutter(s.vater, nameMutter, nameKind);
      if (gefunden) {
        return true;
      }
    }

    // im Teilbaum s ist es nicht
    return false;
  }
```

```
private boolean einfuegeVater(BaumKnoten s, String nameVater,
                              String nameKind) {
  boolean gefunden;

  // suche kind

  // ist es schon Knoten s?
  if (s.name.equals(nameKind)) {     // gefunden
    s.vater = new BaumKnoten(nameVater);   // Vater eintragen
    return true;
  }

  // oder ein Vorfahre mütterlicherseits ?
  if (s.mutter != null) {
    gefunden = einfuegeVater(s.mutter, nameVater, nameKind);
    if (gefunden) {
      return true;
    }
  }

  // oder ein Vorfahre väterlicherseits ?
  if (s.vater != null) {
    gefunden = einfuegeVater(s.vater, nameVater, nameKind);
    if (gefunden) {
      return true;
    }
  }

  // im Teilbaum s ist es nicht
  return false;
}

private static int indentPrint = 3;  // Einrückung beim Ausdrucken

void print(BaumKnoten s, int pos) {
  // drucke Baum unter s, um pos nach rechts eingerückt

  // erst pos Leerzeichen
  for (int i=0; i<pos; i++) {
     System.out.print(" ");
  }

  // dann den Namen
  System.out.println(s.name);

  // dann den Rest der Familie
  if (s.vater != null) {
    print(s.vater, pos + indentPrint);
  }
  if (s.mutter != null) {
    print(s.mutter, pos + indentPrint);
  }
}

public static void main(String[] args) {
```

```
    Baum family = new Baum("adam");

    family.einfuegeVater("brian", "adam");
    family.einfuegeMutter("daniela", "brian");
    family.einfuegeMutter("charlotte", "adam");
    family.einfuegeVater("edward", "charlotte");
    family.einfuegeMutter("fanny", "charlotte");
    family.einfuegeMutter("gwendolyn", "fanny");

    family.print();
  }
}
```

Lösung von Aufgabe 6

- Die Höhe eines Baumes wird natürlich rekursiv bestimmt: Ein leerer Baum hat die Höhe 0, ansonsten ist die Höhe eines Baumes um eins größer als das Maximum der Höhe seiner Kinder.
- Die Ausgaberoutine kann man wie in Aufgabe 5 gestalten. Hier wird noch der Elternknoten zwischen den beiden Kindern ausgegeben, um die Baumstruktur deutlicher zu machen. An dieser Stelle wäre es sicher sinnvoll, eine graphische Ausgaberoutine für einen Baum zu programmieren.
- Die neuen Methoden in Baum und AVLBaum sind identisch, es sollte sich also doch irgendwie mit Vererbung die Verdoppelung verhindern lassen. Das ist aber nicht ganz einfach: Man kann ja leicht AVLKnoten als Kind von BaumKnoten implementieren. Der Versuch, AVLBaum als Kind von Baum zu schreiben, scheitert aber daran, dass die Knoten-Typen von AVLBaum nicht mehr passen.
- Um die immer gleichartige Einleseprozedur zu vereinfachen, wurde mit IOTools eine kleine Sammlung nützlicher Routinen zusammengestellt, die auch gleich die Initialisierung des Inputstreams übernimmt.
- Sourcen:
    - BaumKnoten, Baum
    - AVLKnoten, AVLBaum
    - IOTools, TestBaeume
- als lauffähiges Applet

BaumKnoten

```
public class BaumKnoten {

  public double daten;
  public BaumKnoten links;
  public BaumKnoten rechts;

  // einfacher Konstruktor
  public BaumKnoten(double n) {
    daten  = n;
    links  = null;   // zeigen noch nirgendwo hin
    rechts = null;
  }
}
```

Baum.java

```
import java.io.*;

public class Baum {

  public BaumKnoten wurzel;

  public Baum() {
    // Konstruktor erzeugt leeren Baum
    wurzel = null;
  }

  public void einfuegen(double wert) {
    // fügt Knoten neu an die richtige Stelle ein
    BaumKnoten neu = new BaumKnoten(wert);

    // 1. Fall: Baum ist noch leer
    if (wurzel == null) {
      wurzel = neu;
    } else {
      // sonst rekursiv durch
      einfuegen(wurzel, neu);
    }
  }

  public int hoehe() {
    // gibt Höhe des Baums zurück
    return hoehe(wurzel);
  }

  public int hoehe(BaumKnoten spitze) {
    // gibt Höhe des Baums unter spitze zurück
    if (spitze == null) {
      return 0;
    } else {
      int h = Math.max(hoehe(spitze.links), hoehe(spitze.rechts));
      return h + 1;
    }
  }

  private void einfuegen(BaumKnoten spitze, BaumKnoten neu) {
    // fügt Knoten neu an die richtige Stelle unter Teilbaum spitze ein

    if (spitze.daten > neu.daten) {
      // links einfügen
      if (spitze.links == null) {
        // neu einfach anhängen
        spitze.links = neu;
      } else {
        // neu im linken Unterbaum unterbringen
        einfuegen(spitze.links, neu);
      }
    } else {
      // rechts einfügen
      if (spitze.rechts == null) {
```

```
        // neu einfach anhängen
        spitze.rechts = neu;
      } else {
        // neu im rechten Unterbaum unterbringen
        einfuegen(spitze.rechts, neu);
      }
    }
  }

  public String toString() {
    if (wurzel != null) {
      return toString(wurzel, 0);
    } else {
      return "<leerer Baum>";
    }
  }

  private static int indentPrint = 4;  // Einrückung beim Ausdrucken

  private String toString(BaumKnoten b, int pos) {
    // Ausdrucken des Teilbaums ab b, Reihenfolge inorder
    String s = "";

    // pos Leerzeichen für die weitere Ausgabe
    String margin = "";
    for (int i=0; i<pos; i++) {
      margin += " ";
    }

    if (b == null) {
      s += margin + "<null>"+ "\n";
    } else {
      s += toString(b.rechts, pos + indentPrint);
      s += margin + b.daten + "\n";
      s += toString(b.links, pos + indentPrint);
    }

    return s;
  }
}
```

AVLKnoten.java

```
public class AVLKnoten {

  double daten;
  AVLKnoten links;
  AVLKnoten rechts;
  int balance;

  // einfacher Konstruktor
  public AVLKnoten(double n) {
    daten  = n;
    balance = 0;
    links  = null;   // zeigen noch nirgendwo hin
    rechts = null;
  }
}
```

AVLBaum.java

```
import java.io.*;

public class AVLBaum {

  private AVLKnoten wurzel;

  public AVLBaum() {
    // Konstruktor erzeugt leeren Baum
    wurzel = null;
  }

  public int hoehe() {
    // gibt Höhe des Baums zurück
    return hoehe(wurzel);
  }

  public int hoehe(AVLKnoten spitze) {
    // gibt Höhe des Baums unter spitze zurück
    if (spitze == null) {
      return 0;
    } else {
      int h = Math.max(hoehe(spitze.links), hoehe(spitze.rechts));
      return h + 1;
    }
  }

  private AVLKnoten rotiereLinks(AVLKnoten b) {
    // vollführt einfaches Rotieren mit b, Fall links - links
    // gibt die geänderte Wurzel zurück
    // Vorbedingung: b ist ein nichtleerer Baum mit den richtigen Teilbäumen
    //    zum entsprechenden Rotieren

    AVLKnoten a = b.links;
    b.links = a.rechts;
    a.rechts = b;

    b.balance = 0;
    a.balance = 0;

    return a;
  }


  private AVLKnoten rotiereRechts(AVLKnoten b) {
    // vollführt einfaches Rotieren mit b, Fall rechts - rechts
    // gibt die geänderte Wurzel zurück
    // Vorbedingung: b ist ein nichtleerer Baum mit den richtigen Teilbäumen
    //    zum entsprechenden Rotieren

    AVLKnoten a = b.rechts;
    b.rechts = a.links;
    a.links = b;

    b.balance = 0;
```

```
  a.balance = 0;

  return a;
}

private AVLKnoten rotiereDoppeltLinksRechts(AVLKnoten c) {
  // vollführt doppeltes Rotieren mit c, Fall links - rechts
  // gibt die geänderte Wurzel zurück
  // Vorbedingung: c ist ein nichtleerer Baum mit den richtigen Teilbäumen
  //    zum entsprechenden Rotieren

  AVLKnoten a = c.links;
  AVLKnoten b = a.rechts;
  a.rechts = b.links;
  b.links = a;
  c.links = b.rechts;
  b.rechts = c;

  c.balance = (b.balance == -1)? 1 : 0;
  a.balance = (b.balance ==  1)? -1 : 0;
  b.balance = 0;

  return b;
}

private AVLKnoten rotiereDoppeltRechtsLinks(AVLKnoten c) {
  // vollführt doppeltes Rotieren mit c, Fall rechts - links
  // gibt die geänderte Wurzel zurück
  // Vorbedingung: c ist ein nichtleerer Baum mit den richtigen Teilbäumen
  //    zum entsprechenden Rotieren

  AVLKnoten a = c.rechts;
  AVLKnoten b = a.links;
  a.links = b.rechts;
  b.rechts = a;
  c.rechts = b.links;
  b.links = c;

  c.balance = (b.balance == 1)? -1 : 0;
  a.balance = (b.balance ==  -1)? 1 : 0;
  b.balance = 0;

  return b;
}

public void einfuegen(double neuerWert) {
  // fügt Knoten neu an die richtige Stelle ein

  AVLKnoten neu = new AVLKnoten(neuerWert);

  // 1. Fall: Baum ist noch leer
  if (wurzel == null) {
    wurzel = neu;
  } else {
    // sonst rekursiv durch
    wurzel = einfuegen(wurzel, neu);
  }
```

```
}

boolean rebalance;   // muss der Baum umsortiert werden?

private AVLKnoten einfuegen(AVLKnoten spitze, AVLKnoten neu) {
  // fügt Knoten neu an die richtige Stelle unter Teilbaum spitze ein
  // gibt Zeiger auf neuen Wert für spitze zurück

  AVLKnoten temp;

  if (neu.daten > spitze.daten) {
    // rechten Teilbaum betrachten
    if (spitze.rechts == null) {
      // neuen Knoten einfach rechts anhängen
      spitze.rechts = neu;
      spitze.balance++;
      rebalance = (spitze.balance >= 1);
      return spitze;
    } else {
      // im rechten Teilbaum einfügen
      spitze.rechts = einfuegen(spitze.rechts, neu);  // ggf. spitze ändern
      if (rebalance) {
        // Ausgleichen
        switch (spitze.balance) {
          case -1:
            // war links-lastig, jetzt ausgewogen
            spitze.balance = 0;
            rebalance = false;
            return spitze;
          case 0:
            // war ausgeglichen, jetzt hier rechtslastig. Weiter oben?
            spitze.balance = 1;
            return spitze;
          case 1:
            // ok, ich bin rechts schief
            rebalance = false;  // wird hier gleich bereinigt
            if (spitze.rechts.balance == 1) {
              // Fall rechts - rechts
              return rotiereRechts(spitze);
            } else {
              // Fall rechts - links
              return rotiereDoppeltRechtsLinks(spitze);
            }
        }
      } else {
        return spitze;   // alles ok, einfach hochreichen
      }
    }
  } else {
    // linken Teilbaum betrachten, genau wie rechts
    if (spitze.links == null) {
      // neuen Knoten einfach links anhängen
      spitze.links = neu;
      spitze.balance--;
      rebalance = (spitze.balance <= -1);
      return spitze;
    } else {
```

```
      // im linken Teilbaum einfügen
      spitze.links = einfuegen(spitze.links, neu);  // ggf. spitze ändern
      if (rebalance) {
        // Ausgleichen
        switch (spitze.balance) {
          case 1:
            // war rechts-lastig, jetzt ausgewogen
            spitze.balance = 0;
            rebalance = false;
            return spitze;
          case 0:
            // war ausgeglichen, jetzt hier linkslastig. Weiter oben?
            spitze.balance = -1;
            return spitze;
          case -1:
            // ok, ich bin links schief
            rebalance = false;  // wird hier gleich bereinigt
            if (spitze.links.balance == -1) {
              // Fall links - links
              return rotiereLinks(spitze);
            } else {
              // Fall links - rechts
              return rotiereDoppeltLinksRechts(spitze);
            }
        }
      } else {
        return spitze;   // alles ok, einfach hochreichen
      }
    }
  }

  return null;  // der Form halber - sollte hier nie ankommen!
}

public String toString() {
  if (wurzel != null) {
    return toString(wurzel, 0);
  } else {
    return "<leerer Baum>";
  }
}

private static int indentPrint = 4;  // Einrückung beim Ausdrucken

private String toString(AVLKnoten b, int pos) {
  // Ausdrucken des Teilbaums ab b, Reihenfolge inorder
  String s = "";

  // pos Leerzeichen für die weitere Ausgabe
  String margin = "";
  for (int i=0; i<pos; i++) {
    margin += " ";
  }

  if (b == null) {
    s += margin + "<null>"+ "\n";
  } else {
```

```
      s += toString(b.rechts, pos + indentPrint);
      s += margin + b.daten + "\n";
      s += toString(b.links, pos + indentPrint);
    }

    return s;
  }
}
```

IOTools.java

```
import java.io.*;
import java.util.*;

public class IOTools {

  // Hilfsprogramme zur Vereinfachung des Benutzer-Dialogs

  BufferedReader in;

  public IOTools() {
    in = new BufferedReader(new InputStreamReader(System.in));
  }

  public double holeDouble(String bitte) throws IOException {
    // holt vom Benutzer einen double-Wert
    // gibt vorher bitte aus
    System.out.println(bitte);
    String s = in.readLine();
    double neuerWert = Double.parseDouble(s);
    return neuerWert;
  }

  public double[] holeVieleDoubles(String bitte) throws IOException {
    // holt vom Benutzer mehrere double-Werte als Array
    // gibt vorher bitte aus
    // Ende durch leere Eingabe
    System.out.println(bitte + " (Ende: leere Eingabe)");

    // da die Anzahl unbekannt ist, werden die Werte zunächst in eine Liste gepackt
    ArrayList liste = new ArrayList();
    do {
      String s = in.readLine();
      if (s.equals("")) {
        break;
      }
      double neuerWert = Double.parseDouble(s);
      liste.add(new Double(neuerWert));
    } while (true);

    // Werte als Array zurückgeben
    // Problem: double sind keine Objekte, also Umweg über Double
    Object[] oListe = liste.toArray();
    double[] ergebnis = new double[oListe.length];
    for (int i=0; i < oListe.length; i++) {
      ergebnis[i] = ((Double) oListe[i]).doubleValue();
    }

    return ergebnis;
  }

  public int holeInt(String bitte) throws IOException {
    // holt vom Benutzer einen int-Wert
    // gibt vorher bitte aus
    System.out.println(bitte);
```

```
    String s = in.readLine();
    int neuerWert = Integer.parseInt(s);
    return neuerWert;
  }

  public boolean jaOderNein(String frage) throws IOException {
    // holt vom Benutzer die Antwort auf eine Alternativfrage
    // und gibt sie als boolean zurück

    System.out.println(frage + " (j/n)?");
    String s = in.readLine();

    boolean antwort = false;
    if (s.compareTo("j") == 0) {
      antwort = true;
    }
    return antwort;
  }
}
```

TestBaeume.java

```
import java.io.*;
import algorithmen.*;

public class TestBaeume {

  public static void main(String[] args) throws IOException {

    IOTools io = new IOTools();

    // starte mit leeren Bäumen
    Baum sortierBaum = new Baum();
    AVLBaum avlBaum = new AVLBaum();

    // Werteingabe oder Zufallszahlen?
    boolean werteErzeugen = io.jaOderNein("Sollen Werte für die Bäume erzeugt werden");

    // Baum füllen
    double[] werte;
    if (werteErzeugen) {
      // Anzahl der Werte abfragen
      int anzahl = io.holeInt("Anzahl der Listenelemente eingeben:");
      werte = new double[anzahl];

      // Zufallszahlen erzeugen
      for (int i = 0; i < anzahl; i++) {
        werte[i] = Math.random();
      }
    } else {
      // Werte vom Benutzer erfragen
      werte = io.holeVieleDoubles("Positive Zahlen eingeben");
    }

    // fülle die Bäume
    for (int i = 0; i < werte.length; i++) {
      sortierBaum.einfuegen(werte[i]);
      avlBaum.einfuegen(werte[i]);
    }

    // gib die Höhe der Bäume und ggf. die Bäume selbst aus
    System.out.println("Höhe des Sortierbaums: " + sortierBaum.hoehe());
    System.out.println("Höhe des AVL-Baums   : " + avlBaum.hoehe());

    if (io.jaOderNein("Sollen die Bäume ausgegeben werden")) {
      System.out.println("Sortierbaum:");
      System.out.println(sortierBaum);
      System.out.println("\nAVL-Baum:");
      System.out.println(avlBaum);
    }
  }
}
```

Lösung von Aufgabe 7

- Als einfache Hashfunktion wird hier der Modulo verwendet, wobei Double-Werte durch Runden in Integer verwandelt werden. Was das für Konsequenzen haben kann, erkennt man, wenn man die SimpleHashtable mit (z.B.) 15000 Werten betrachtet: Da die Eingabewerte alle im Bereich [0, 10000] liegen, tritt bei 2/3 Füllgrad eine dramatische Performance-Einbuße auf!
- Einen anderen Effekt sieht man, wenn man die (bessere) Hashtable mit 20 Werten betrachtet: Das Programm bricht ab, weil ein Element keinen Platz fand, obwohl wir doch nur bis zu einem Füllgrad von 90% gehen! Ursache sind Eigenschaften der Quadratfunktion, die zusammen mit dem Modulo von 20 nur bestimmte Werte erreicht. Man sollte als Hashgröße daher am besten eine Primzahl wählen.
- Ein Problem beim Hashen von Double-Werten ist die Frage, wie ein leerer Platz erkannt wird. Bei höheren Datentypen erkennt man dies einfach an der null-Referenz. Beim double könnte man auf die Double-Klasse
ausweichen. Alternativ verwendet man einen speziellen double-Wert, der unter den auftretenden Werten nicht vorkommt, hier z.B. Double.POSITIVE_INFINITY.
- Bei der Implementierung von put ist zu beachten, dass die Suche nach freiem Platz irgendwann aufgegeben werden muss. Dies wird hier durch die for-Schleife garantiert, die maximale Zahl von Versuchen ist also gleich der Größe der Tabelle.
- Sourcen:
  - SimpleHashtable
  - Hashtable
  - TestHashtable
- als lauffähiges Applet

SimpleHashtable

```
public class SimpleHashtable {

  protected double[] ar;
  protected int nCollisions;

  protected static final double EMPTY = Double.POSITIVE_INFINITY;

  public SimpleHashtable(int n) {
    ar = new double[n];
    clear();
  }

  public void put(double d) {
    // legt d in der Hashtable ab
    int first = hashCode(d);
    int aktuell = first;

    // suche freien Platz
    int k;
    for (k = 0; k < ar.length; k++) {
      if (ar[aktuell] == EMPTY) {
        break;
      }
      aktuell = sond(first, k);
    }

    if (k == ar.length) {
      // Hashtable ist voll
      throw new IllegalStateException();
    }

    ar[aktuell] = d;
    nCollisions += k;
  }

  public int anzahlKollisionen() {
    return nCollisions;
  }

  public void clear() {
    // leert die Hashtabelle

    nCollisions = 0;
    for (int i = 0; i < ar.length; i++) {
      ar[i] = EMPTY;
    }
  }

  protected int hashCode(double d) {
    // Hashfunktion
    // liefert Position in der Hashtabelle

    // run zu int
    int i = (int) Math.round(d);
```

```
    return  i % ar.length;
  }

  protected int sond(int i0, int k) {
    // Sondierungsfunktion
    // liefert k-ten Versuch für einen freien Platz, wenn i0 besetzt ist

    // gehe einfach k Schritte weiter
    return (i0 + k) % ar.length;
  }
}
```

Hashtable

```
public class Hashtable extends SimpleHashtable {

  public Hashtable(int n) {
    super(n);
  }

  protected int hashCode(double d) {
    // nutzt hashCode für Double
    Double dd = new Double(d);
    return Math.abs(dd.hashCode()) % ar.length;
  }

  protected int sond(int i0, int k) {
    // Sondierungsfunktion
    // liefert k-ten Versuch für einen freien Platz, wenn i0 besetzt ist

    // quadratisch
    return (i0 + k*k) % ar.length;
  }
}
```

TestHashtable

```
import java.io.*;
import algorithmen.*;

public class TestHashtable {

  public static void main(String[] args) throws IOException {

    IOTools io = new IOTools();

    // Art und Größe der Hashtabelle abfragen
    boolean simple = io.jaOderNein("Einfache Hashtabelle");
    int größe = io.holeInt("Größe der Hashtabelle eingeben:");
    SimpleHashtable tabelle;
    if (simple) {
      tabelle = new SimpleHashtable(größe);
    } else {
      tabelle = new Hashtable(größe);
    }

    // Ergebnisse = Werte jeweils bei 10%, 20% etc bis 90%
    double[] ergebnisse = new double[9];
    int chunksize = größe/10;

    // häppchenweise mit Zufallszahlen füttern
    double key;
    for (int i = 0; i < 9; i++) {
      // entsprechend viele Zufallszahlen besorgen
      for (int j = 0; j < chunksize; j++) {
        key = 10000*Math.random();
        tabelle.put(key);
      }
      ergebnisse[i] = tabelle.anzahlKollisionen();
    }

    // Ergebnisse ausgeben
    System.out.println("Füllgrad    Zahl der Kollisionen pro put-Operation");
    for (int i = 0; i < 9; i++) {
      double val = ergebnisse[i]/(chunksize*(i+1));
      System.out.println("  " + (10*(i+1)) + "%               " + val);
    }
  }
}
```

Lösung von Aufgabe 8

- Das Programm verwendet eine TreeMap anstelle eines HashMap für das assoziative Array. Die TreeMap verwendet intern einen sortierten Baum
  anstelle einer Hashtabelle, was automatisch zu alphabetisch sortierter Ausgabe führt.
- Die Datei wird hier als Kommandozeilenargument übergeben und zeilenweise eingelesen. Jede Zeile wird (anhand bestimmter Trennzeichen) in Wörter zerlegt, die dann im TreeMap als Werte (Wort, Anzahl) gespeichert werden.
- Sourcen:
  - WordCounter.java
- keine Applet-Version wegen Dateibearbeitung

WordCounter

```
import java.io.*;
import java.util.*;

/*
 * Zählt die Wörter in einer Datei
 * bekommt Dateinamen als Parameter
 */
public class WordCounter {

  public static void main(String args[]) throws IOException {

    String eingabeDatei = args[0];
    BufferedReader in  = new BufferedReader(new FileReader(eingabeDatei));
    Map m = new TreeMap();

    String s;
    while ((s = in.readLine()) != null) {

      // Zerlege die Zeile in Worte
      String[] worte = s.split("[ \n\t,;.()+=<>!?]");

      // und laufe durch alle Worte
      for (int i=0; i < worte.length; i++) {
        // packe das Wort in die Tabelle bzw. erhöhe seinen Zähler

        // ist das Wort schon in der Tabelle?
        Integer anzahl= (Integer) m.get(worte[i]);
        if (anzahl != null) {
          // erhöhe seinen Zähler
          m.put(worte[i], new Integer(anzahl.intValue() + 1));
        } else {
          m.put(worte[i], new Integer(1));
        }
      }
    }
    in.close();

    // gib die fertige Tabelle aus
    System.out.println("Häufigkeitstabelle der Wörter");
    System.out.println(m);
  }
}
```