Gymnasium Bayern

Informatik 11

Liebe Schülerinnen und Schüler!

Dieses Buch soll Dir helfen, Dich auf die Abiturprüfung vorzubereiten. Dazu ist es notwendig, dass Du die Grundzüge einer objektorientierten Programmiersprache lernst. „Programmieren lernt man nur durch Programmieren.“ Wenn dieser Spruch zutreffen sollte, dann wird es Dir nicht genügen, dich nur durch Lesen dieses Buches auf die Abiturprüfung vorzubereiten. Informatik-Erkenntnisse erschließen sich Dir oft viel leichter, wenn du die Strukturen selbst programmiert hast. In diesem Buch finden sich daher alle Programm-Listings zum Nachprogrammieren. Ich rate Dir dringend, die Listings selbst abzuschreiben, dabei lernst du es am leichtesten.

Das Pflichtprogramm der 11. Jahrgangsstufe ist gar nicht so schwer. Im Unterricht habt ihr den Stoff vielleicht erweitert und erheblich mehr Beispiele durchgesprochen als in diesem Buch vorhanden sind. Aber darauf kommt es mir hier nicht an. Ich möchte Dich begleiten, Dich zielsicher auf Deine schriftliche Abiturprüfung vorzubereiten. Für die mündliche Abiturprüfung wird es notwendig sein, mit dem Kursleiter das prüfungsrelevante Stoffgebiet genauer abzugrenzen.

Ich verwende hier ausschließlich die Programmiersprache Java. Natürlich lassen sich die Konzepte auch ohne Programm-Beispiele lernen, aber wie ich oben bereits angemerkt habe glaube ich, dass Beispiele besser geeignet sind, Konzepte zu verdeutlichen. Ich verwende in diesem Buch BlueJ als Entwicklungsumgebung. Diese solltest Du im Internet gefunden und auf Deinem Rechner installiert haben. Dann kannst du die Beispiele aus dem Buch auch von meiner Homepage laden und ausprobieren.

Für Deine Abiturvorbereitung wünsche ich Dir alles Gute!

Manuel Friedrich

1. Rekursive Datenstruktur Liste

1.1 Software zur Verwaltung der wartenden Besucher bei der Agentur für Arbeit in Neustadt

In der Neustädter Arbeitsagentur geht es immer drunter und drüber. Immer wieder versuchen Besucher, sich vorzudrängeln, wenn der Sachbearbeiter „Der nächste bitte…“ ruft.

Eine Software soll nun helfen, Ordnung zu schaffen. Am Empfang sollen die Besucher mithilfe einer Software namentlich erfasst werden. Neue Besucher werden eingetragen, auf Knopfdruck wird der Name des am längsten wartenden Besuchers angezeigt und auch aus der Liste gelöscht.

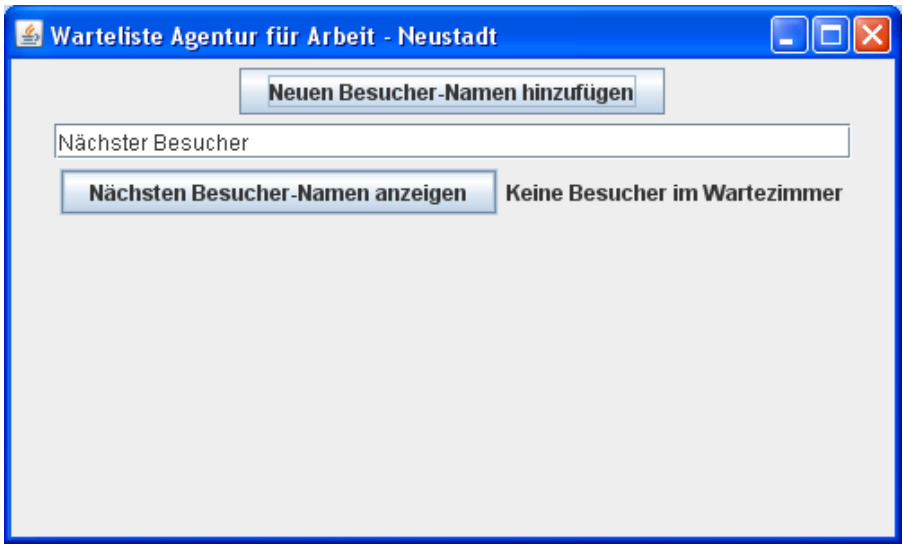


Abb. 1: Software für die Agentur für Arbeit in Neustadt

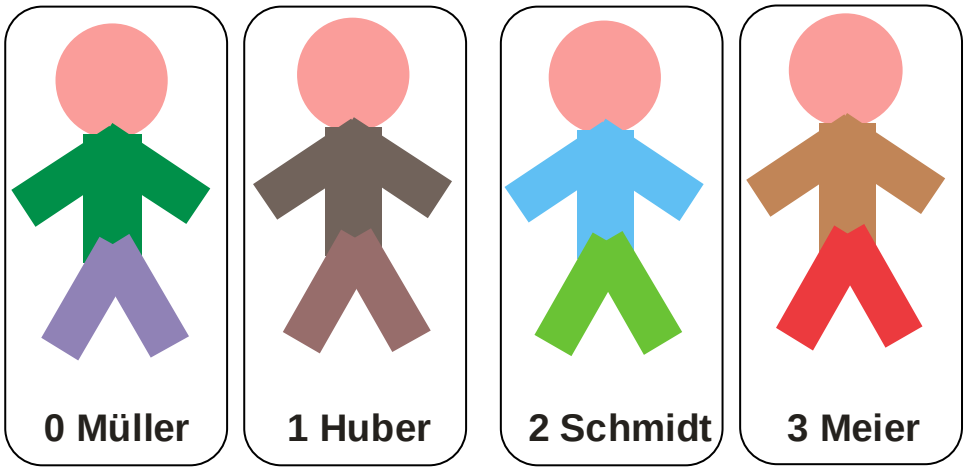


Abb. 2: *Vier Personen in einer Warteschlange, Informatiker beginnen ihr Zählen oft mit der 0. In der Informatik wird die Zahl 0 sogar zu den natürlichen Zahlen gerechnet. Bei Feldern und Listen erhält der erste Eintrag daher meist den Index 0.*

Das Klassendiagramm gibt Auskunft über die Struktur des Programms. Bei diesem Beispiel werden die Daten in einem Array (Feld) gespeichert.

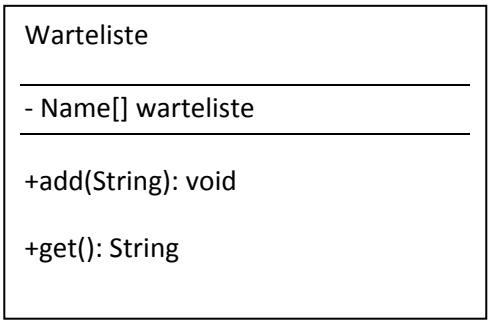


Abb. 3: Klassendiagramm für die Warteliste

1.2 Datenhaltung

Die Daten können auf verschiedene Weise verwaltet werden:

- als Feld;
- in einer ArrayList;
- in einer selbst erstellten Liste;

Wir sehen wir uns alle drei Beispiele an. Die ersten beiden dienen zur Wiederholung des Stoffes aus dem vergangenen Schuljahr.

1.2.1 Alternative 1 - Implementierung als Feld (*Array*)

Zur Wiederholung: Ein Feld unterscheidet sich von einem primitiven Datentyp dadurch, dass es mit eckigen Klammern deklariert wird und mit new instanziiert werden muss. Instanzieren bedeutet, dass das Objekt erzeugt wird. Beim Istanziieren wird in den eckigen Klammern angegeben, für wie viele Variablen bzw. Objektreferenzen Speicherplatz reserviert werden soll.

Listing 1 - Allgemeine Form des Feldes:

```
public class Feld{
   // Feld für Strings
   private String[] warteliste;   // ein Feld von Strings wird deklariert

   public Feld()
   {
      arteliste=new String[100];   // neues Feld für 100 Strings instanziieren

   }

   public string get(int i){
       return warteliste[i];     // Zugriff auf einen Eintrag im Feld mit Index i

   }
}
```

Nun aber zurück zum Beispiel der Verwaltung der Warteliste:

Listing 2 - Feld (Array) zur Verwaltung der Besucher

```
/*
 * Klasse zur Verwaltung von Namen in einem Feld
 * @author Manuel Friedrich
 * Version 2009-09-30
 */
public class Feld
{   // Feld
    private String[] warteliste;
    private int aktPosition;  // Index, bei welchem der nächste Besucher
                              // eingetragen wird


    public Feld()
    {   // neues Feld erzeugen
        warteliste=new String[100];
        // erste Position bestimmen
        aktPosition=0;
        for (int i=0; i<100; i=i+1){ // alle Strings auf "" setzen
            warteliste[i]="";
        }
    }

    /*
     * Neuen Besucher in das Feld eintragen
     */
    public void add(String besucher)
    {
          if (aktPosition<100) {
          warteliste[aktPosition]=besucher;
          aktPosition=aktPosition+1;
       }
    }

    /*
     * Nächsten Besucher ausgeben und Eintrag vorne löschen
     */
    public String get()
    { String n="Niemand im Wartezimmer!";    // Für den Fall, dass niemand im
                                             //Wartezimmer ist

      if (aktPosition>0)     // es ist doch jemand da, ersten Eintrag auswählen
      {
        n=warteliste[0];
        for (int i=0; i<aktPosition-1; i=i+1){ // alle Einträge nach vorne
           warteliste[i]=warteliste[i+1];      // rutschen
        }
        aktPosition=aktPosition-1; // aktuelle Position um ein Stelle verkleinern
      }
      // Ergebnis ausgeben
      return n;
    }
}
```

Die Verwendung eines Feldes bringt aber Nachteile mit sich: Bei einem Feld muss **vorher** festgelegt werden, wie viele Besucher maximal verwaltet werden sollen. In diesem Beispiel gehen wir von einer maximalen Anzahl von 100 Besuchern aus. Als weiteres Argument gegen die Verwendung eines Feldes wird oft argumentiert, dass es sehr umständlich sei, dass nach dem Löschen eines Eintrages

die gesamte Liste um eine Position nach vorne sortiert werden muss. Dies ließe sich allerdings leicht vermeiden, indem man einen zweiten Integer-Wert mitzählt, der die Position des nächsten Besuchers bestimmt. Beide Werte würden eine Art Ring-Puffer ergeben. Nach der Position 99 müssten die Zähler wieder vorne auf den ersten Eintrag 0 zeigen. Ein weiteres Problem bleibt aber: Werden nur wenige Besucher verwaltet, so wurde bei einem Feld Speicherplatz sinnlos belegt. Auch dieses Argument scheint bei der Leistungsfähigkeit moderner Computer nicht wirklich zu stechen. Dennoch bleibt das ungute Gefühl, dass die Festlegung auf eine maximale Anzahl an Wartenden nicht wirklich sinnvoll ist.

1.2.2 Alternative 2 - Implementierung unter Verwendung einer ArrayList

Die genannten Nachteile des Feldes existieren bei Verwendung einer *ArrayList* nicht. Eine *ArrayList* ist genau wie ein Feld eine Möglichkeit, mehr als eine Variable bzw. Objektreferenz zu verwalten. Anders als bei einem *Array* können aber beliebig viele Einträge hinzugefügt werden und es wird nicht unnötig Speicherplatz reserviert. Bei der Instanziierung einer *ArrayList* muss daher die maximale Anzahl des verwalteten Datentyps nicht angegeben werden.

Die *ArrayList* ist eine Klasse der Java-Bibliothek. Alle Klassen der Java-Bibliothek stehen dem Programmierer zur Verfügung. Sie müssen lediglich mit einer Import-Anweisung zum Beginn des Programmcodes eingebunden werden. Eine leere *ArrayList* kann ohne Angabe eines Parameters durch Aufrufen des Konstruktors `ArrayList<…>()` erzeugt werden. In spitzen Klammern wird der Datentyp der in der *ArrayList* verwalteten Objekte/Variablen bestimmt. Die Methoden `add(…)`, `get(int):…`, *`remove(int)`* und `size():int` sind (beinahe) selbsterklärend , können aber natürlich in der Beschreibung zur Java-API nachgelesen werden. (In BlueJ findest du den Link auf die Beschreibung der Java-API im Hilfe-Menü http://download.oracle.com/javase/6/docs/api/index.html .)

Bei den Methoden wird als Index ein `integer`-Wert verwendet. Der erste Eintrag in der *ArrayList* hat den Index 0, der zweite den Index 1 usw. Der n-te Eintrag hat den Index n-1. Die Methode `size()` liefert die Anzahl der vorhandenen Variablen bzw. Objekte, die aktuell in der *ArrayList* gespeichert sind.

Listing 3 - Allgemeine Verwendung der ArrayList:

```
import java.util.ArrayList;     Import-Anweisung für den Zugriff auf die API

public class Liste_ArrayList{
    // ArrayList für Strings
    private ArrayList<String> warteliste;

    public Liste_ArrayList()
    {   // neue ArrayList erzeugen
        warteliste=new ArrayList<String>();  Konstruktor aufrufen
    }

    // Zugriff auf einen Eintrag im Feld
    public String get(int i){
        return warteliste.get(i);   der Index liegt zwischen 0 und
                                    warteliste.size()-1
    }
 }
```

Listing 4 - ArrayList zur Verwaltung der Besucher:

```
import java.util.ArrayList;
/**
 * Wartelistenverwaltung der Agentur für Arbeit in Neustadt
 * unter Verwendung einer ArrayList
 *
 * @author Manuel Friedrich
 * @version 2009-09-29
 */
public class Liste_ArrayList
{
    private ArrayList<String> warteliste;

    /**
     * Konstruktor für Objekte der Klasse Liste_ArrayList
     */
    public Liste_ArrayList()
    {
        warteliste=new ArrayList<String>();
    }

    /**
     * Einen Besucher hinzufügen
     *
     * @param  besucher    Name des Besuchers
     */
    public void add(String besucher)
    {
       warteliste.add(besucher);
    }

   /*
    * Nächsten Besucher ausgeben und Eintrag vorne löschen
    */
    public String get()
    {  // für den Fall, dass niemand im Wartezimmer ist
       String n="Niemand im Wartezimmer!";
       if (warteliste.size()>0)
       {
          // es ist doch jemand da, dann Eintrag auswählen
          n=warteliste.get(0);
          // Eintrag aus der Liste löschen
          warteliste.remove(0);
       }
       // Ergebnis ausgeben
       return n;
    }
}
```

Die Methode `get()` liefert als Ergebnis den Namen zurück, der in der ArrayList den Index 0 besitzt. Da der Eintrag durch den Aufruf der Methode `warteliste.remove(0)` gelöscht wird muss vorher der Name in einer lokalen Variablen `n` gespeichert werden, um ihn am Methoden-Ende mit der `return`-Anweisung zurückzuliefern. Standardmäßig erhält `n` den Wert „Niemand im Wartezimmer“. Befinden sich keine Besucher im Wartezimmer wird der Wert für `n` nicht überschrieben und als Meldung ausgegeben.

1.2.3 Alternative 3 - Eine selbst erstellte Liste verwenden

Eine Liste ist ja nichts anderes als eine Reihe von Elementen desselben Typs. In diesem Kapitel werden wir also eine Klasse ähnlich der ArrayList selbst programmieren. Dazu speichern wir eine Information in einer Klasse Besucher.

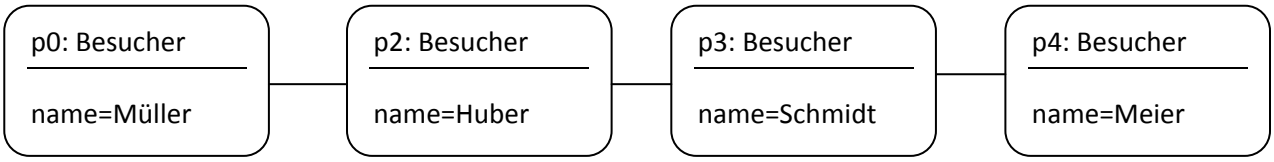


Abb. 3: Das Objektdiagramm zu der Warteschlange, zwischen den Objekten besteht eine 1:1-Beziehung.

Das Klassendiagramm, das sich aus den Objekten ergibt ist schnell erstellt. Später werden wir die Klasse nicht Besucher nennen, sondern allgemein *Knoten*, da wir neben Strings auch andere Datentypen und Objekte verwalten wollen.

Ein Objekt der Klasse `Besucher` benötigt eine Referenz auf seinen Nachfolger. Bis auf das letzte Objekt der Liste haben alle Besucher einen Nachfolger.

Ein Objekt der Klasse `Besucher` zeigt also auf ein Objekt ebenfalls der Klasse `Besucher` usw. Weil die Beziehung *hatNachfolger* zwischen ein und derselben Klasse verläuft sprechen wir von einer **reflexiven Beziehung**.

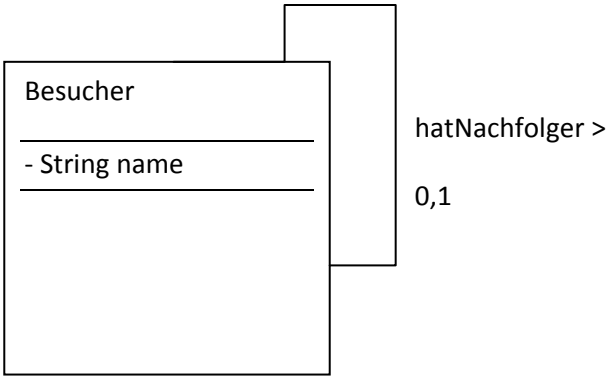


Abb. 4: Das Klassendiagramm Besucher

Eine Warteliste soll es auch geben, wenn noch keine Besucher vorhanden sind. Wir wollen dann von einer leeren Liste sprechen. Daher benötigen wir eine Klasse Warteliste, mit der wir die Besucher verwalten. Hier befinden sich auch die Methoden zum Einfügen und Herausholen/Löschen von Besuchern. Auch zwischen der Warteliste und dem ersten Besucher benötigen wir eine Referenz. Einfacher ist es, zwei Referenzen für die Verwaltung der Besucher zu verwenden. Eine Referenz zeigt auf den ersten Besucher und damit auf den Anfang der Warteschlange, die zweite Referenz auf das Ende der Schlange, also den zuletzt gespeicherten Besucher.

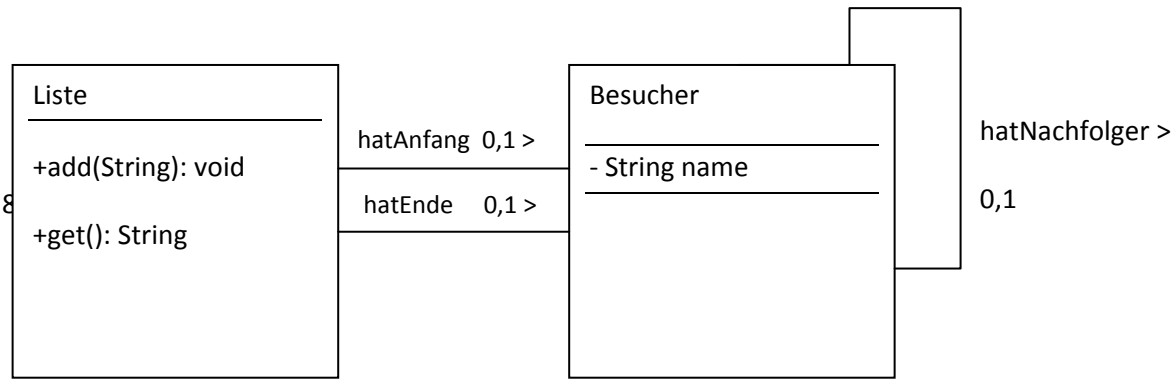


Abb. 4: Das um die Klasse Liste ergänzte Klassendiagramm, eine Klasse Liste verwaltet die Knoten. Alle Referenzen haben die Kardinalität 0,1: In der leeren Liste sind die Referenzen hatAnfang und hatEnde jeweils 0, alle Besucher-Objekte außer dem letzten haben eine Referenz auf den Nachfolger, das letzte Besucherobjekt hat hier den Eintrag null.

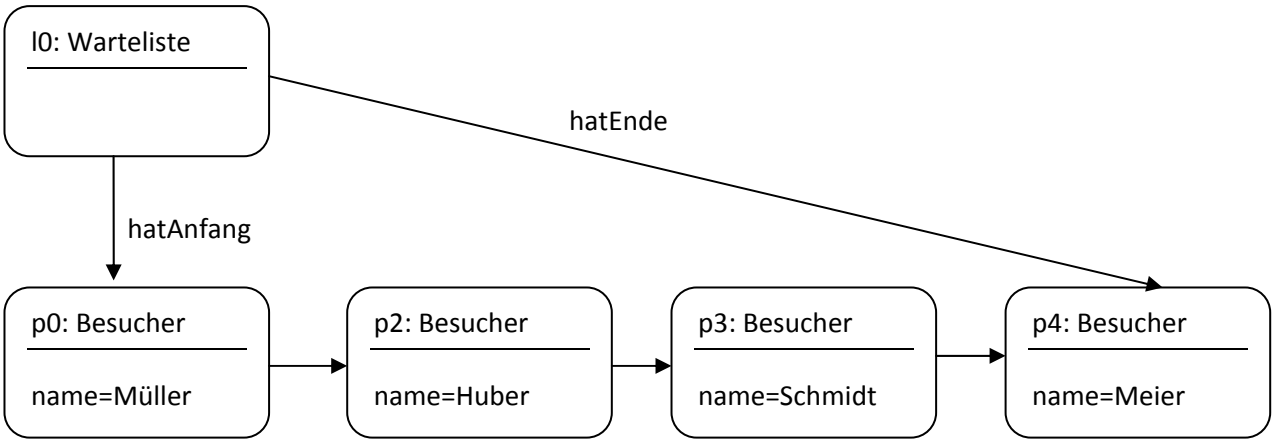


Abb. 5: Objektdiagramm aus Abb. 3 um das Objekt l0 der Klasse Warteliste erweitert.

Die Klasse Besucher

Die Klasse ist schnell implementiert und sollte Dir keine Schwierigkeiten bereiten. Wir brauchen eine verändernde Methoden zum Setzen eines Nachfolgers und zwei sondierende Methoden für den Namen und den Nachfolger.

Listing 5: Knoten zur Speicherung einer Information, hier eines Strings

```
/*
 * Klasse zur Speicherung einer Information,
 * hier eines Strings - In der Liste benötigen
 * wir noch die Referenz auf den Nachfolger
 */

public class Besucher
{

    private String name;
    private Besucher nachfolger;

    public Besucher(String n)
    { name=n;
      nachfolger=null;
    }

    public void setNachfolger(Knoten k){
      nachfolger=k;
```

```
    }

    public Knoten getNachfolger(){
        return nachfolger;   // liefert den Nachfolger
    }

    public String getName(){
        return name; // gibt das Attribut name zurück
    }
}
```

Die Klasse Liste

Schwieriger gestaltet sich das Einfügen von Elementen in die Liste. Dazu muss man etwas wissen: Objekte, die in BlueJ nicht auf der Objektleiste stehen werden in Java automatisch gelöscht, wenn keine Referenz mehr auf sie zeigt. Dies übernimmt ein Prozess im Hintergrund, der sogenannte Müllsammler (garbage collector, kurz gc). Dies ist sehr anwenderfreundlich, da Du dich als Programmierer nicht darum kümmern musst, dass die Objekte wieder gelöscht werden, wenn sie nicht mehr benötigt werden. Der belegte Speicher, der für Objekte bei seiner Instanziierung zugewiesen wurde, wird automatisch wieder freigegeben und steht dem Programm wieder zur Verfügung.

Für unsere Liste bedeutet dies, dass wir die Referenzen der Liste beim Einfügen eines neuen Elementes so verändern müssen, dass kein Objekt verloren geht. Dazu müssen wir die Referenzen „umbiegen“. Mit „umbiegen“ meine ich das Verändern der Referenzen, dass sie auf ein anderes Objekt zeigen. Ich will Dir das am Beispiel der Methode add(String s) zeigen. Mit der Methode add(String s) sollen neue Knoten erzeugt und der Liste hinzugefügt werden. Wir gehen zunächst davon aus, dass eine Liste mit Besuchern bereits besteht.

Zum besseren Verständnis werden wir die Referenzen, die bisher nur durch die Linien dargestellt wurden, nun auch mit Attributen direkt in das Objekt- bzw. Klassendiagramm aufnehmen. Man spricht dann von einem **Erweiterten Objekt- bzw. Klassendiagramm**.

Definition: Bei einem **Erweiterten Objekt- bzw. Klassendiagramm**, werden auch die Referenzattribute direkt mit in die Objekte bzw. Klassen aufgenommen.

Die Methode `add(String)`

In wenigen Schritten wird die Liste um einen Besucher erweitert.

Liste mit 3 Knoten

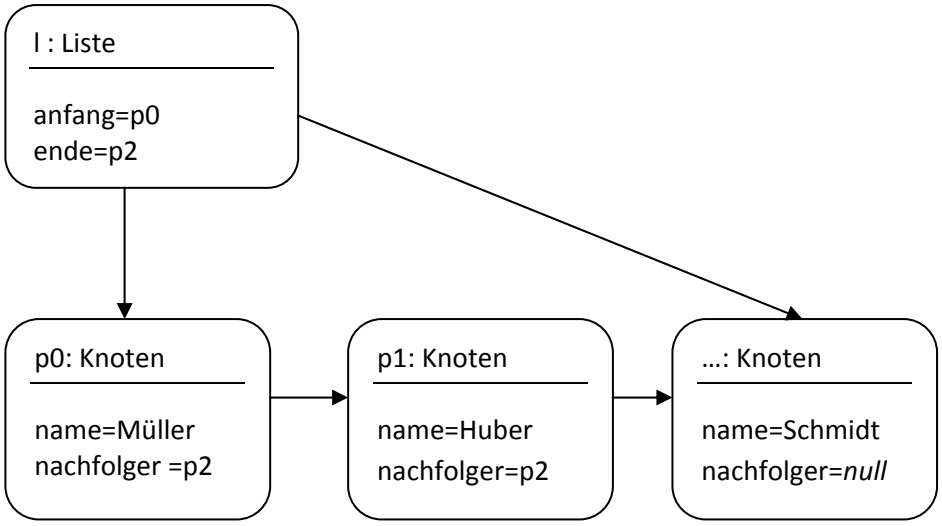


Abb. 6: Ausgangslage: Liste mit drei Knoten

Erster Schritt: Neuen Knoten erzeugen

Zunächst muss ein neuer Knoten erzeugt werden: `Knoten k=new Knoten("Meier");`

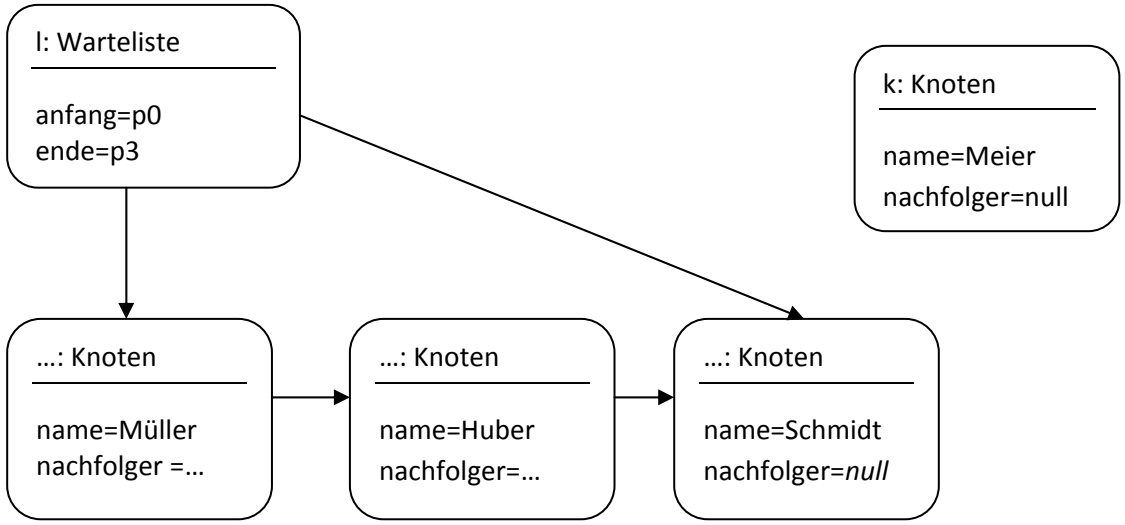


Abb. 7: Ein neuer Knoten wurde erzeugt, ist aber noch nicht in die Liste eingefügt. Beachte, dass der letzte Knoten der Liste keine Referenz auf einen Nachfolger besitzt.

Der neue Knoten wird solange nicht vom *Garbage Collector* gelöscht, solange noch die Referenz k vorhanden ist, also solange die Methode `add(String)` ausgeführt wird.

Zweiter Schritt: Nachfolger setzen

Um kein Objekt zu verlieren, wird nun als erstes der Nachfolger des bisherigen Endes gesetzt.

Der Name des Objektes ist nicht direkt verfügbar. Durch die Referenz `ende` haben wir aber Zugriff auf das Objekt.

Die Anweisung lautet daher `ende.setNachfolger(k);`

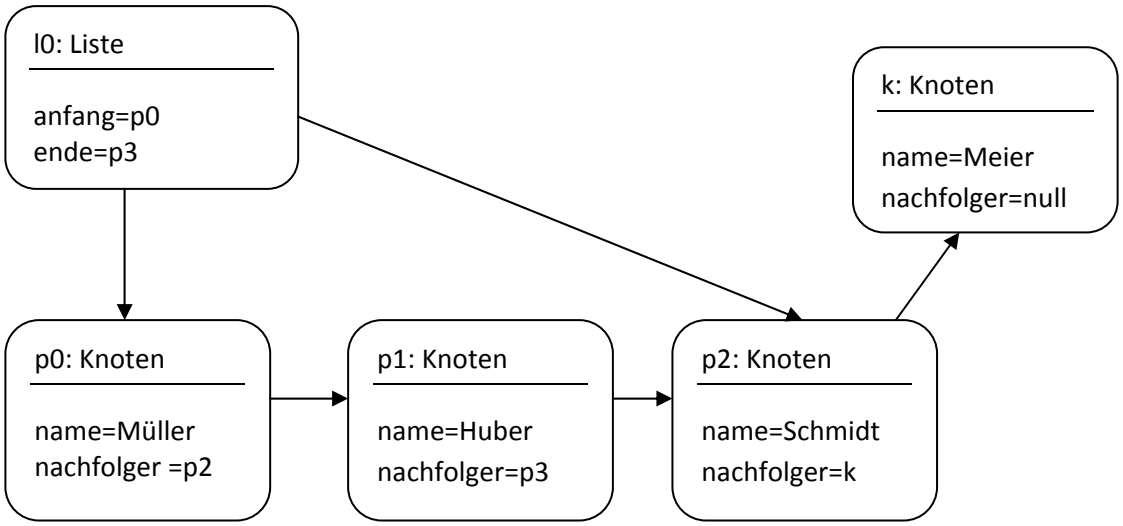


Abb. 8: Beim ursprünglich letzten Element der Liste wird die Referenz des Nachfolgers auf den neuen Knoten gesetzt.

Dritter Schritt: Neues Ende festlegen

Nun wird noch die Referenz `ende` des Objektes Liste auf das neue Ende „umgebogen“ mit der Anweisung `ende=k;` Da `ende` und `k` Attribut bzw. eine lokale Variable der Klasse Liste sind, kann die Wertzuweisung direkt vorgenommen werden und bedarf nicht des Aufrufes einer verändernden Methode, wie das in der Anweisung `ende.setNachfolger (k)` im zweiten Schritt notwendig ist, weil hier von der Klasse `Liste` auf das Attribut der Klasse `Knoten` zugegriffen wird.

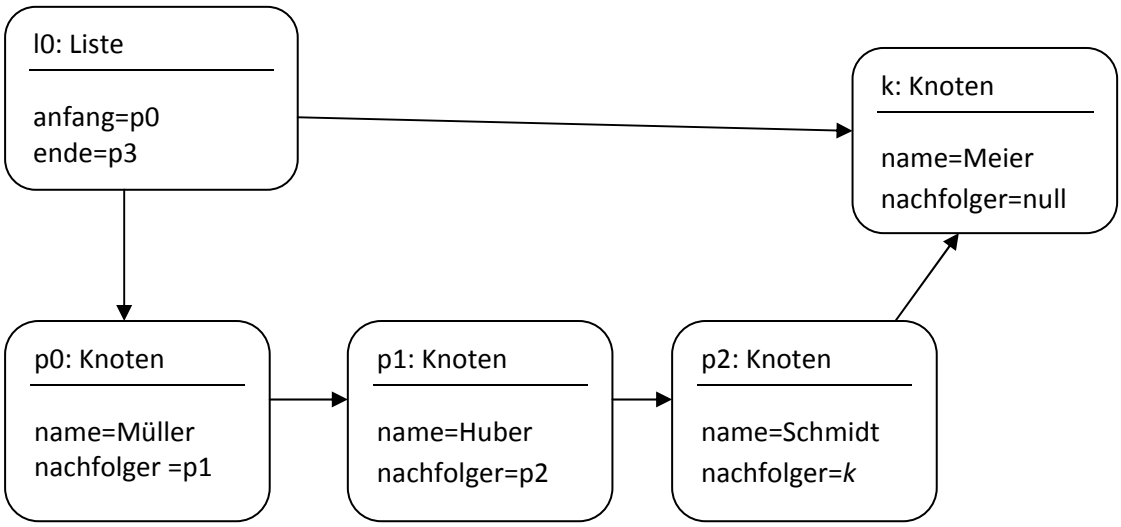


Abb. 9: Nachdem auch die Referenz ende auf den neuen Knoten gesetzt ist, ist die Liste wieder komplett.

Die Liste ist damit um ein Element erweitert worden. Diese Vorgehensweise funktioniert aber nur, wenn mindestens schon ein Knoten vorhanden ist, da die Methode ende.nachfolgerSetzen(Knoten) in einer leeren Liste nicht aufgerufen werden kann.

Sollte die Liste leer sein, so wird sowohl `anfang` als auch `ende` auf den neuen Knoten gesetzt.

Die Anweisung dafür lautet: `if (anfang==null) { anfang=k; ende=k; }`

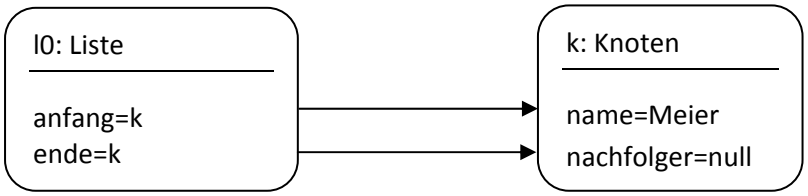


Abb. 10: Bei einer leeren Liste wird das neue Objekte referenziert, indem `anfang` und `ende` auf den Knoten zeigen.

Die Methode `isEmpty():boolean`

Diese Methode liefert `true` zurück, wenn die Liste leer ist, sonst `false`.

Die Methode `get():String`

Die Methode `get()` liefert den Namen der am längsten wartenden Person und löscht sodann den Knoten.

Der Name der Person wird zunächst, vorausgesetzt die Liste ist nicht leer, ermittelt und in einer lokalen Variablen `n` gespeichert. Der Wert von `n` wird am Ende der Methode mit `return` zurückgeliefert. Das Löschen des Knotens erfolgt durch das Umlenken der Referenz auf den Anfangsknoten. Auf den ursprünglich ersten Knoten zeigt dann keine Referenz mehr und er wird automatisch vom *garbage collector (gc)* gelöscht.

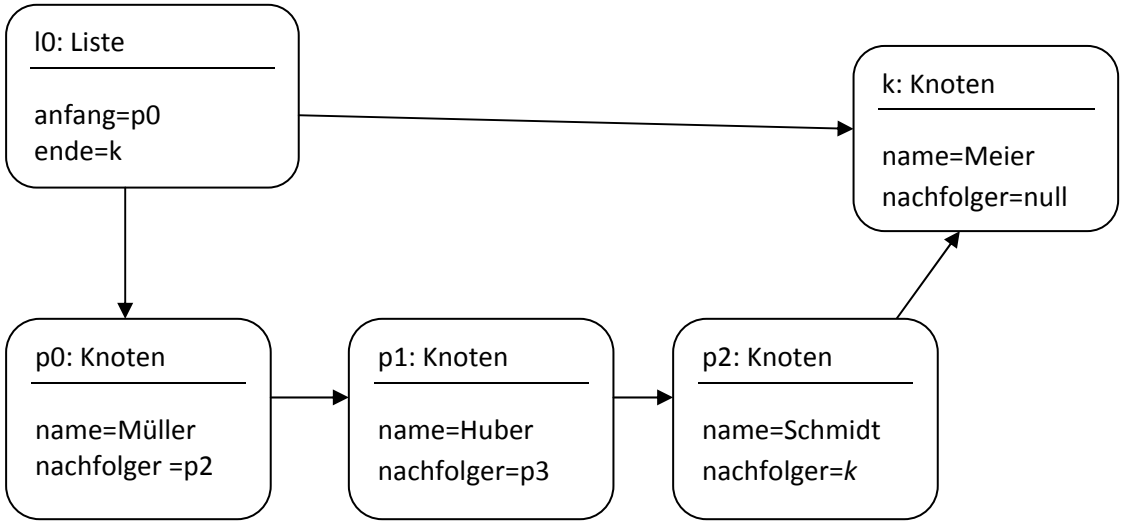


Abb. 11: Ausgangslage: Liste mit vier Knoten

Die Referenz `anfang` wird auf den ersten Knoten „umgebogen“ mit der Anweisung `anfang=anfang.getNachfolger();` Bei einer Wertzuweisung wird stets zuerst der rechte Teil ausgewertet und erst danach der linken Seite zugewiesen. Somit kann die Referenz `anfang` im Listenobjekt tatsächlich auf den Nachfolger des „noch“ ersten Objektes gesetzt werden.

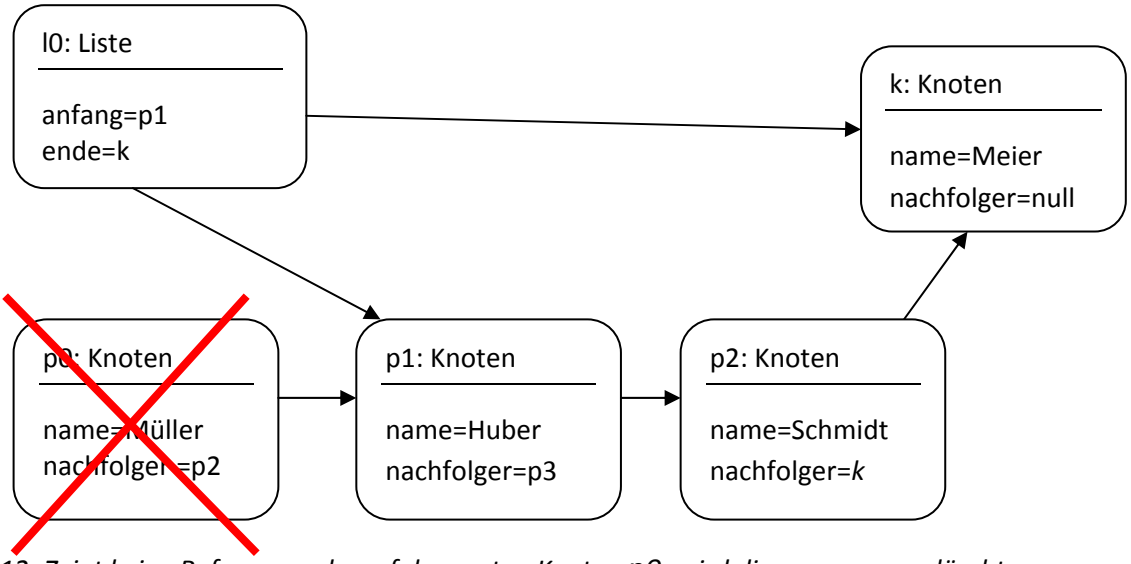


Abb. 12: Zeigt keine Referenz mehr auf den ersten Knoten `p0`, wird dieser vom gc gelöscht.

Listing 6: Die Klasse Liste

```
public class Liste
{
    private Knoten anfang;
    private Knoten ende;

    public Liste()
    {
        anfang=null;
        ende=null;
    }

    public boolean isEmpty(){
        if (anfang==null) return true; else return false;
    }

    public void add(String name){
      Knoten k=new Knoten(name);
      if (anfang==null) { anfang=k; ende=k; } else
      { ende.nachfolgerSetzen(k);  ende=k; }
    }

    public String get()
    {
      if (!isEmpty())  // Alternative für (anfang!=null)
      {
        String n=anfang.getName();
        if (anfang.getNachfolger()!=null) anfang=anfang.getNachfolger(); else
        { anfang=null; ende=null;
          return n;
```

```
        }
      }
      return "Kein Besucher im Wartezimmer";
    }
}
```

1.3 Erstellen einer GUI* (* = nicht prüfungsrelevant)

Eine *Graphical User Interface* (GUI = Graphische Benutzerschnittstelle) ist heute Standard bei der Bedienung einer Software. Die Implementierung in Java ist leider nicht so einfach wie in anderen Programmiersprachen.

Wir werden es an diesem einfachen Beispiel Schritt für Schritt erklären.

Zunächst benötigen wir überhaupt ein Fenster. Dies erhalten wir, indem wir eine neue Klasse „Fenster“ programmieren und dies von JFrame ableiten. JFrame gehört zur Java-Bibliothek swing. Daneben benötigen wir zwei Buttons, ein Textfeld zum Eintragen des neuen Besuchers und ein Textlabel zur Ausgabe des Namens des am längsten wartenden Besuchers. Alle Elemente sind ebenfalls in swing vorhanden und können deklariert werden. Zur Ereignisbehandlung benötigen wir noch zwei Importanweisungen für benötigte awt-Klassen zur Steuerung der Benutzereingaben.

Listing 7: Eine GUI programmieren

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

public class Fenster extends JFrame{
  JButton bEintragen;
  JButton bAusgeben;
  JTextField mEingabe01;
  JLabel  mInfo01;

  public Fenster()
  { // Neues Fenster mit Titel
    super("Warteliste Agentur für Arbeit - Neustadt");

    // Button, zwei Texteingabefelder und ein TextLabel erzeugen
    bEintragen=new JButton("Neuen Besucher-Namen hinzufügen");
    bAusgeben=new JButton("Nächsten Besucher-Namen anzeigen");
    mEingabe01=new JTextField("Nächster Besucher",40);
    mInfo01=new JLabel("Keine Besucher im Wartezimmer",SwingConstants.LEFT);
…
```

Die vier Elemente auf dem Fenster und das Fenster selbst sind nun zwar erstellt, aber noch nicht miteinander verbunden. Die Methode `getContentPane()` der Klasse `JFrame` liefert die Zeichenfläche des Fensters zurück. Wir benötigen diese Methode, um einen Layout-Modus festzulegen und die vier Elemente dann in das Fenster aufzunehmen. Das sog. `FlowLayout` reiht die Elemente nacheinander auf dem Fenster an.

Listing 8: Das Layout festlegen und die Elemente zur Fensterzeichenfläche hinzufügen

```
    // Layout-Manager erzeugen
    getContentPane().setLayout(new FlowLayout());
    // Elemente in das Fenster aufnehmen
    getContentPane().add(bEintragen);
```

```
    getContentPane().add(mEingabe01);
    getContentPane().add(bAusgeben);
    getContentPane().add(mInfo01);
```

Die beiden Knöpfe kann man zwar mit der Maus drücken, aber es wird nichts passieren, solange die beiden Buttons nicht bei einem sog. `ActionListener` angemeldet sind. Zudem benötigen wir noch die Fenstergröße und möchten, dass das Fenster sichtbar wird.

Listing 9: Auf Knopfdruck reagieren: Knöpfe einem ActionListener zuordnen

```
    // werden die Knöpfe gedrückt soll eine vorgegebene Methode
    // der Klasse CMeinActionLauscher() aufgerufen werden
    bEintragen.addActionListener(new CMeinActionLauscher());
    bAusgeben.addActionListener(new CMeinActionLauscher());

    // Größe des Fensters festlegen und Sichtbarkeit auf true
    setSize(500,300);
    setVisible(true);
  }

  public static void main(String[] args)
  {
     Fenster fenster=new Fenster();
  }
```

Jetzt fehlt nur noch der letzte Schritt, was soll passieren, wenn auf einen Button gedrückt wird.

Wir schreiben hier eine eigene Klasse `CMeinActionLauscher` direkt in die Klasse `Fenster` hinein. Die Methode `actionPerformed(ActionEvent e)` wird immer dann aufgerufen, wenn einer der beiden Buttons im Fenster gedrückt wird. Wir ermitteln zuerst, welcher Button gedrückt wurde und rufen dann in einer bedingten Anweisung die notwendige Sequenz auf. Die Methode `e.getActionCommand()` liefert den Aufdruck des gedrückten Buttons, `getText()` liefert den String, der in das Textfeld eingetragen wurde und `setText(String)` verändert das angezeigte Textlabel.

Listing 10: Auf Knopfdruck reagieren: Die interne Klasse ActionListener

```
  // Ereignisbehandlung für die Steuerelemente, hier Button
  // Eigene Klasse in der Klasse
  class CMeinActionLauscher implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    {
      String label;
      label=e.getActionCommand();
      if (label.equals("Neuen Besucher-Namen hinzufügen"))
      {
          warteliste.add(mEingabe01.getText());
      }
      else if (label.equals("Nächsten Besucher-Namen anzeigen"))
        {
           mInfo01.setText(warteliste.get());
        }
    }
  }
}
```

Alle anderen Klassen der GUI und die ausführliche Beschreibungen aller Konstruktoren und Methoden kannst du in der Beschreibung der JAVA-API nachesen. Du findest sie in den Paketen `swing` und `awt`.

Arbeitsblatt

Wir schreiben eine kleine Klasse zur Bestimmung der Nullstellen einer quadratischen Gleichung.

Die Lösung kennst du: $$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Zur Lösung dieser Aufgabe müssen die Werte für a, b und c in ein Formular eingegeben werden. Durch Drücken eines Buttons sollen die beiden Werte x1 und x2 ausgegeben werden.

1. Erstelle eine GUI in der Klasse „Fenster“ zur Lösung quadratischer Gleichungen mit drei Textfeldern für a, b und c, einen Button und einem Label. Dir steht die Klasse Fenster des Programms der Warteschlange im Arbeitsamt Neustadt als Vorlage zur Verfügung.

2. Schreibe eine Klasse „QuadGleichung“ mit der Methode
   `public String getQuad(double a, double b, double c)`

3. Verbinde die beiden Gleichungen, indem du in die Klasse `Fenster` ein Referenzattribut zur Klasse `QuadGleichung` einfügst und somit von hier aus die Methode `getQuad(double a, double b, double c):String` aufrufen kannst. Problematisch dabei wird sein, die Einträge in den Textfelder für a, b und c vom Datenformat String in einen Double-Wert zu konvertieren. Recherchiere die Lösung ggf. in den Java-Klassenbibliotheken oder im Internet.

4. Teste die Anwendung ausführlich und erstelle abschließend mit BlueJ eine JAR-Datei zum direkten Ausführen des Programms.

1.4 Wir verbessern die Struktur unserer Liste

1.4.1 Eine eigene Klasse für die Besucher

Unsere Liste funktioniert tadellos, trotzdem gibt es bekanntlich nichts, was man nicht noch besser machen könnte. Listen werden ja nicht nur beim Verwalten von Wartenden in einem Arbeitsamt benötigt. Auch die Druckerwarteschlange, die Tastenanschläge, die mit der Tastatur eines Computers erfolgen, die Verwaltung von Artikeln in einem Lager u.v.a.m. können mithilfe von Listen verwaltet werden.

Unsere Liste soll so erweitert werden, dass sie nicht nur mit Strings umgehen, sondern auch jede andere Art von Daten verwalten kann.

Die Klasse Besucher nimmt in unserer Software im Moment zwei Aufgaben wahr. Zum einen ist ein Objekt dafür verantwortlich, die Information über den Wartenden zu speichern, zum anderen verwaltet das Objekt auch die Information über seinen Nachfolger.

Fortan soll die Klasse Besucher nur noch die letztgenannte Aufgabe erledigen. Wir nenne sie nun auch nicht mehr Besucher, sondern Knoten. Die zu verwaltenden Daten werden in eine eigene Klasse `Daten` ausgelagert.

Objektdiagramm vorher:

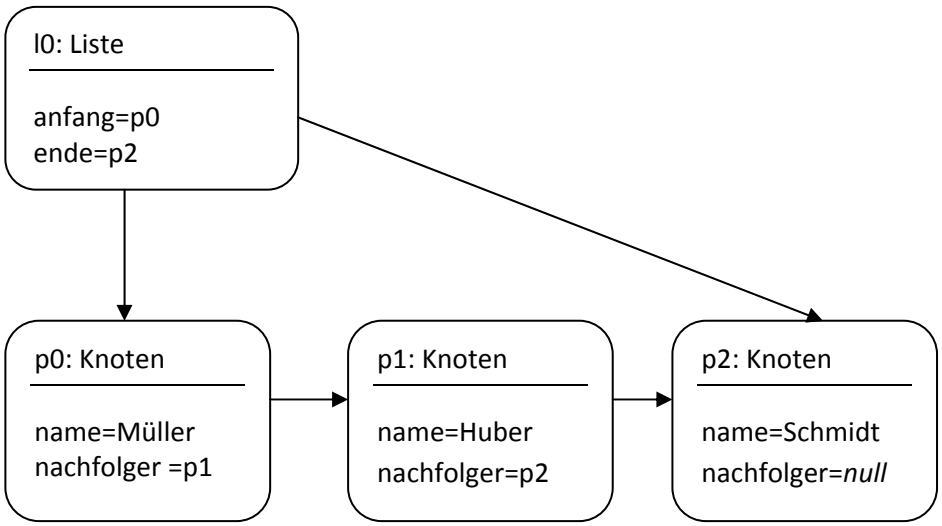


Abb. 13: Ausgangslage ist eine Liste mit drei Elementen

Objektdiagramm nachher:

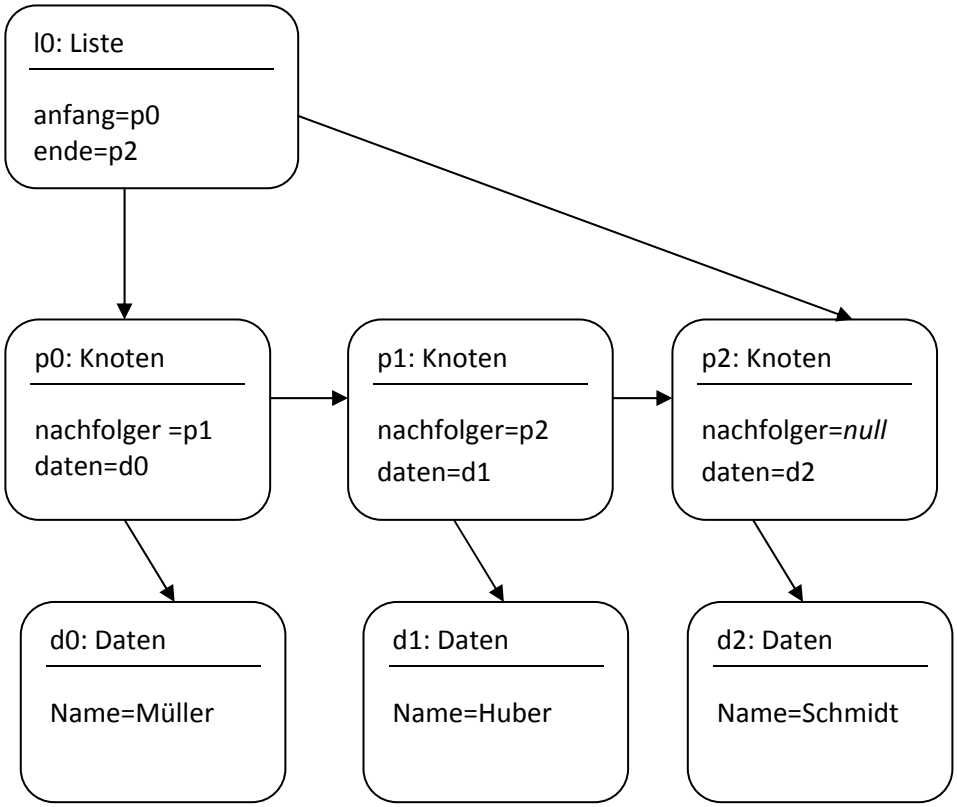


Abb. 14: Liste mit drei Elementen, wobei die Namen der Besucher in eigene Objekte ausgegliedert wurden.

Unser Quelltext muss nun natürlich angepasst werden. Wir benötigen eine eigenen Klasse `Daten`. In der Klasse `Knoten` benötigen wir eine Referenz auf diese Klasse.

Listing 11: Die Klasse Knoten verweist auf ein Objekt der Klasse Daten

```
public class Knoten
{
  private Daten daten;
  private Knoten nachfolger;

  public Knoten(String n)
  {
    daten=new Daten(n);
    nachfolger=null;
  }

  public void setNachfolger(Knoten k){
    nachfolger=k;
  }

  public Knoten getNachfolger(){
    return nachfolger;
  }

  public String getName(){
    return daten.getName();
  }
}
```

Anstelle eines Attributes `Namen` befindet sich in der Klasse `Knoten` nur noch die Referenz auf eine Klasse `Daten`. Im Konstruktor wird daher auch ein Objekt der Klasse `Daten` instanziiert (neu erzeugt). Die sondierende Methode `getName()` kann nun auch nicht mehr das Attribut direkt zurückliefern, sondern ruft die entsprechende Methode der Klasse `Daten` auf.

Die Klasse `Daten` stellt eine einfache Möglichkeit dar, einen String zu speichern und muss nicht weiter erläutert werden.

Listing 12: Die Klasse Besucher zur Verwaltung der Besucherdaten

```
public class Daten
{
    String name;

    public Daten(String n)
    {
       name=n;
    }

    public String getName()
    {
        return name;
    }

    public void setName(String n)
    {
        name=n;
    }

    public void informationAusgeben()
    {
      System.out.println("Der Name lautet: "+name);
    }
}
```

1.4.2 Auf eine Schnittstelle programmieren

Software sollte immer so programmiert werden, dass sie leicht zu warten ist und dass Klassen in späteren Programmen wiederverwendet werden können. Wenn wir in späteren Programmen Listen verwenden, dann werden vermutlich nicht mehr Besucher der Arbeitsagentur verwaltet. Daher werden wir die Klasse Daten vermutlich neu programmieren, während die anderen Klassen *Liste* und *Knoten* sich nicht verändern sollen. Um sicherzustellen, dass die Klasse Daten problemlos ausgetauscht werden kann, empfiehlt es sich, zwischen der Klasse Knoten und der Klasse Daten eine Schnittstelle zu definieren, in der festgelegt wird, welche Methoden der Klasse Daten von den anderen Klassen aufgerufen werden. In einer neuen Klasse „Daten“ muss sichergestellt sein, dass diese Methoden in jedem Fall auch vorhanden sind.

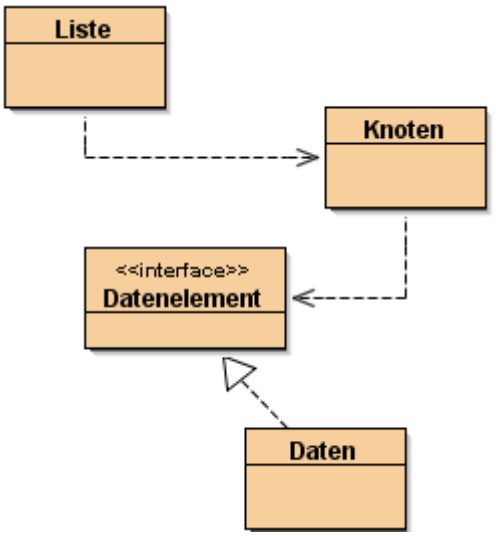


Abb. 15: Darstellung der Schnittstelle für unsere Liste

An dieser Stelle möchte ich das Thema „Vererbung“, das in der letzten Jahrgangsstufe bereits behandelt wurde, in einem kleinen Exkurs wiederholen. Die Kenntnis ist notwendig, um die Schnittstelle zu verstehen. Wenn dir die Vererbung bestens vertraut ist kannst du die Kapitel 1.4.2.1 und 1.4.2.2 überspringen und mit dem Durcharbeiten des Kapitels 1.4.2.3 *Programmierung des Interfaces* fortfahren.

1.4.2.1 Vererbung mit Superklasse (Oberklasse) und Subklasse (Unterklasse)

Listing 13: Aus der Jahrgangsstufe 10 bekannte Art der Vererbung

```
public class Datenelement{
  public void informationAusgeben(){}
}

public class Daten extends Datenelemnt{
  String name;
  …
  Public void informationAusgeben(){
    System.out.println(name);
  }
}
```

Bei einer Vererbung erbt die Subklasse alle Attribute und Methoden der Oberklasse. Dies müssen nicht gesondert deklariert werden. Allerdings können Methoden überschrieben werden. D.h. sie werden mit genau derselben Signatur in der Subklasse implementiert.

Beispiel:
Die Klassen `LeitenderAngestellter`, `Sachbearbeiter` und `Auszubildender` sind von der Klasse `Mitarbeiter` abgeleitet. In der Oberklasse befindet sich u.a. das Attribut `gehalt`, mit dem der Monatslohn bezeichnet wird und die Methode `gehaltserhoehungDurchfuehren()`, durch die das Gehalt eines Mitarbeiters erhöht werden kann. Die Methode kann nun in allen Subklassen überschrieben werden. Von Subklassen kann der Konstruktor mit der Anweisung `super(…)` aufgerufen werden. Methoden der Superklasse können mit der Anweisung `super.methodenname(…)` aufgerufen werden.

Vererbung: Durch Vererbung erhalten die Unterklassen alle Attribute und Methoden der Oberklasse. Die Oberklasse wird auch Superklasse, die Unterklasse Subklasse genannt. Mit dem Wort `super(…)` kann der Konstruktor, mit `super.methodenname()` können auch überschriebenen Methoden der Superklasse aufgerufen werden. Als `private` deklarierte Attribute der Superklasse können nur über ihre get- und set-Methoden eingesehen und verändert werden. Für den direkten Zugriff aus Unterklassen könnten Attribute mit dem Sichtbarkeitsmodifikator `protected` deklariert werden.

Auf eine Anzahl von Mitarbeitern kann nun die Methode `gehaltserhoehungDurchfuehren()` in einem Schleifendurchgang angewendet werden, jedes Objekt ruft dabei seine ihm spezifische Methode auf.

Listing 14: Aus der Jahrgangsstufe 10 bekannte Art der Vererbung

```
public class Mitarbeiter{
  // Attribute
  private String name;
  private double gehalt;

  // Konstruktor
  public Mitarbeiter(String name, double gehalt){
    this.name=name; this.gehalt=gehalt;
  }

  // leere Methode
  public void gehaltserhoehungDurchfuehren(){}

}

public class LeitenderAngestellter extends Mitarbeiter{
  public LeitenderAngestellter(String name, double gehalt){
    super(name, gehalt);
  }
  public void gehaltserhoehungDurchfuehren (){
    grundgehalt=grundgehalt*1.1;
  }
}
public class Sachbearbeiter extends Mitarbeiter{
  …
  public void gehaltserhoehungDurchfuehren (){
    grundgehalt=grundgehalt*1.05;
  }
}
public class Auszubildender extends Mitarbeiter{
  …
  public void gehaltserhoehungDurchfuehren (){
    grundgehalt=grundgehalt+50;
  }
}


public class Mitarbeiterverwaltung{

  …

  // ArrayList mit allen Mitarbeitern

  private ArrayList<Mitarbeiter> mitarbeiter;

  …

  GehaltserhoehungFuerAlleMitarbeiter(){
```

```
    For (int i=0; i<mitarbeiter.size(); i=i+1){
       Mitarbeiter.get(i).gehaltserhoehungDurchfuehren();
    }
  }
}
```

In der Klasse `Mitarbeiterverwaltung` wird eine `ArrayList` vom Typ `Mitarbeiter` verwendet. Als Mitarbeiter können in dieser `ArrayList` sowohl Leitende Angestellte, als auch Sachbearbeiter und Auszubildende verwaltet werden.

Beachte, dass niemals ein Objekt der Klasse `Mitarbeiter` erzeugt wird und dass die Methode `gehaltserhoehungDurchfuehren()` in der Oberklasse auch keine Anweisungen enthält. Trotzdem benötigen wir die Methodendeklaration, um die Methode in der Klasse `Mitarbeiterverwaltung` für alle Elemente der *ArrayList* aufrufen zu können.

1.4.2.2 Abstrakte Klassen

Eine Klasse kann als *abstrakte Klasse* definiert werden. Dazu fügt man in der Oberklasse beim Klassennamen das Wort `abstract` hinzu. **Von abstrakten Klassen selbst kann kein Objekt erzeugt werden.** Dies ist nur von Unterklassen dieser Klassen möglich. Auch Methoden können als *abstrakte Methoden* deklariert werden. **Die abstrakten Methoden müssen dann in der Unterklasse überschrieben werden.** Es muss also in der Unterklasse eine Methode mit gleicher Signatur geben, wobei das Schlüsselwort `abstract` nicht mehr Bestandteil der Signatur ist. Eine abstrakte Methode besitzt in der abstrakten Klasse lediglich eine Signatur und wird ohne geschweifte Klammern aber dafür mit einem Semikolon implementiert.

Für unsere Mitarbeiterverwaltung wäre eine abstrakte Klasse ideal, da von der Klasse `Mitarbeiter` niemals ein Objekt erzeugt wird und die Methode `gehaltserhoehungDurchfuehren()` in dieser Klasse daher auch keine Anweisungen benötigt.

Listing 15:Abstrakte Klasse Mitarbeiter mit einer abstrakten Methode

```
public abstract class Mitarbeiter{
  …
  public abstract void gehaltserhoehungDurchfuehren();

}
```

Auch in unserer Liste können wir diese Struktur verwenden. Wir verwenden eine abstrakte Klasse Datenelement, von der wir die Klasse Besucher ableiten.

Listing 16: Abstrakte Klasse und abstrakte Methode

```
abstract class Datenelement{
  public abstract void informationAusgeben();
}
```

```
class Daten extends Datenelement{
  private String name;
  …
  public void informatinAusgeben(){
     System.out.println(name);
  }
}
```

In Java kann eine Klasse maximal eine Oberklasse haben. Mehrfachvererbung ist nicht zulässig. Daneben gibt es in Java allerdings Vererbungsstrukturen, von denen in eine Subklasse auch mehrere implementiert werden können. Die sog. Interfaces.

1.4.2.3 Interface

Es gibt noch eine weitere Möglichkeit, Vererbungsstrukturen zu implementieren. Über ein Interface. Ein Interface ist eine Schnittstelle. Ein Interface stellt man sich am besten als abstrakte Klasse vor, bei der alle Methoden abstrakt sind. Implementiert wird ein Interface wie im folgenden beschrieben.

Listing 17: Ein Interface implementieren

```
#1  interface Datenelement{
#2    void informationAusgeben();
#3  }
```

Die entsprechende Einbindung in der Unterklasse erfolgt über die Anweisung `implements`.

Listing 18: Ein Interface in eine Klasse einbinden

```
#1  class Besucher implemnts Datenelement{
#2    private String name;
#3    …
#4    public void informationAusgeben(){
#5      System.out.println(name);
#6    }
#7  }
```

Für unsere Zwecke der Programmierung einer Schnittstelle genügt das Interface. Durch die Verwendung wollen wir v.a. sicherstellen, dass in allen von der Schnittstelle abgeleiteten Klassen bestimmte genau festgelegte Methoden implementiert werden.

Definition: Bei einem **Interface** werden Methoden definiert, die zwingend in den Unterklassen übernommen werden müssen. Im Interface selbst werden die Methodenrümpfe nicht implementiert. Dies ist nicht notwendig, da von einem Interface keine Objekte erstellt werden können. Ein Interface wird mit der Anweisung `implements` in einer Unterklasse aufgenommen.

Das Interface im Klassendiagramm hat also weder Attribute noch Methoden, sondern gibt nur an, dass in den Klassen, die dieses Interface implementieren bestimmte Methode vorhanden sein müssen. Dies erleichtert die spätere Wiederverwendung von Programmcode in anderen Prorammen. Merke: „Es ist besser, auf eine Schnittstelle zu programmieren als direkt auf eine Klasse.“

1.5 Rekursive Methoden

1.5.1 Einführung

Methoden, die sich selbst aufrufen, nennt man rekursive Methoden. Um nicht in einer Endlosschleife hängen zu bleiben benötigt jede rekursive Methode auch eine Abbruchbedingung, die nach einer bestimmten Anzahl von Rekursionsschritten die Methode beendet.

Listing 19: Rekursiver Methodenaufruf ohne Abbruchbedingung erzeugt eine Endlosschleife

```
#1  public class Endlosschleife{
#2    int ergebnis;
#3
#4    public int berechne(){
#5      return berechne();
#6    }
#7  }
```

Die Methode `berechne()` in Listing 19 ruft sich selbst neu auf. Allerdings unendlich oft.

Im nächsten Beispiel wollen wir eine Methode wählen, die eine Abbruchbedingung erhält für den sog. Basisfall einer Rekursion. Ziel der Methode berechne soll sein, die Zahlen von 1 bis n zusammenzuzählen.

Listing 20: Rekursiver Methodenaufruf mit Abbruchbedingung

```
#1  public class Summe{
#2    int ergebnis=0;
#3
#4    public void berechne(int n)
#5    {
#6      if (n>0)
#7      {
#8        if (n==1)
#9        {
#10         System.out.println("Ergebnis 1");
#11       } else
#12       {
#13         ergebnis=n+berechne(n-1);
#14         System.out.println("Ergebnis"+ergebnis;);
#15       }
#16       return -1;  // soweit darf es niemals kommen, da oben in der
#17                   // if-else-Struktur jeweils ein return vorhanden ist
#18     }
#19   }
#20 }
```

In Zeile 13 wird, für den Fall dass `(n>0)` wahr ist, die Methode `berechnen()` ein weiteres Mal aufgerufen, ohne dass der erste Aufruf bereits zu Ende wäre. Im Speicher des Computers wird dazu eine Kopie der Methode `berechne()` angelegt und die Anweisungen dort werden vom Anfang der Methode an ein zweites Mal abgearbeitet. Erst wenn diese Methode zu Ende ist, springt das Programm zurück in die vorhergehende Methode und kann die Anweisungen ab Zeile 14 ausführen.

Die Methode `berechne()` wartet also solange, bis ihm das Ergebnis der Wertzuweisung in Zeile 13 bekannt ist, bevor es mit der Zeile 14 fortfährt. Der zweite Aufruf der Methode bleibt aber in den meisten Fällen nicht der letzte. Dies ist abhängig von der Größe `n`. So groß die Zahl ist, so oft wird die Methode `berechne()` aufgerufen und alle vorher aufgerufenen Methoden `berechne()` müssen warten, bis das Ergebnis für den Fall `n==1` berechnet ist. An der Textausgabe wird deutlich, wie oft die Methode berechne() aufgerufen wurde.

Ein rekursiver Aufruf kann daher sehr viel Speicherplatz beanspruchen und unter Umständen sehr viele Rechenoperationen notwendig machen. Vorteil rekursiver Methoden ist aber, dass sich scheinbar schwer zu lösende Probleme mit wenigen Zeilen Programmcode lösen lassen.

Die rekursive Lösung funktioniert dann oft sehr gut, wenn eine Problemstellung gelöst werden kann, indem der Operand leicht modifiziert nochmals aufgerufen wird, solange bis ein Aufruf mit einem Operanden erfolgt der einfach durch Angabe des Ergebnisses gelöst werden kann.

Als weiteres bekanntes Beispiel schauen wir uns die Berechnung der Fakultät an. Bei dieser mathematischen Aufgabe wird das Produkt aller Zahlen von 1 bis n gesucht. Diese Aufgabe ist der obigen sehr ähnlich. Wir wollen sie nutzen, um den Programmablauf zu verstehen.

Die Methode `getFak(int n):int` soll mit dem Wert 5 aufgerufen werden. Das Ergebnis lautet 120, denn `5*4*3*2*1` ergibt 120. Das Problem soll mithilfe einer rekursiven Methode gelöst werden.

```
#1  public class Fakultaet{
#2    public int fak(int n)
#3    {
#4      if (n==1)
#5        {
#6          return 1;
#7        } else
#8        {
#9          return n*fak(n-1);
#10       }
#11    } return -1;
#12 }
```

Die Methode `fak(int n)` kennt nur die Lösung für `n==1`. Aber sie weiß noch, wie man von einem beliebigen `n` der Lösung einen Schritt näher kommt. Man spricht von einem *Rekursionsschritt*, da man dem Basisfall `n==1` durch den Rekursionsschritt näher gekommen ist.

Wird die Methode fak(5) aufgerufen, so weiß das Programm nur, dass es die Lösung findet mithilfe eines weiteren Aufrufs der Methode fak(…), diesmal mit einem um 1 reduzierten Parameter.

```
fak(5) =
5*fak(4)=
5*4*fak(3)=
5*4*3*fak(2)=
5*4*3*2*fak(1)=
5*4*3*2*1
```

Die Methode `fak(int)` muss also insgesamt fünf Mal aufgerufen werden. In unserem Beispiel muss die zuerst aufgerufene Methode warten, bis sie das Ergebnis von `fak(4)` erhalten hat, also bis

zum Ende der Berechnung. Die folgende Abbildung veranschaulicht, wie die Berechnung erfolgt. Jede Fakultätsberechnung muss gleichsam in den Keller hinabsteigen bis der Basisfall erreicht ist, der direkt angegeben werden kann. Der Aufruf der Methode `fak(1)` hat das Ergebnis 1. Nun wird dieses Ergebnis zurück an den vierten Aufruf der Methode gereicht und hier mit 2 multipliziert. Die 2 wandert an den dritten Aufruf zurück und wird mit 3 zu 6 multipliziert usw.

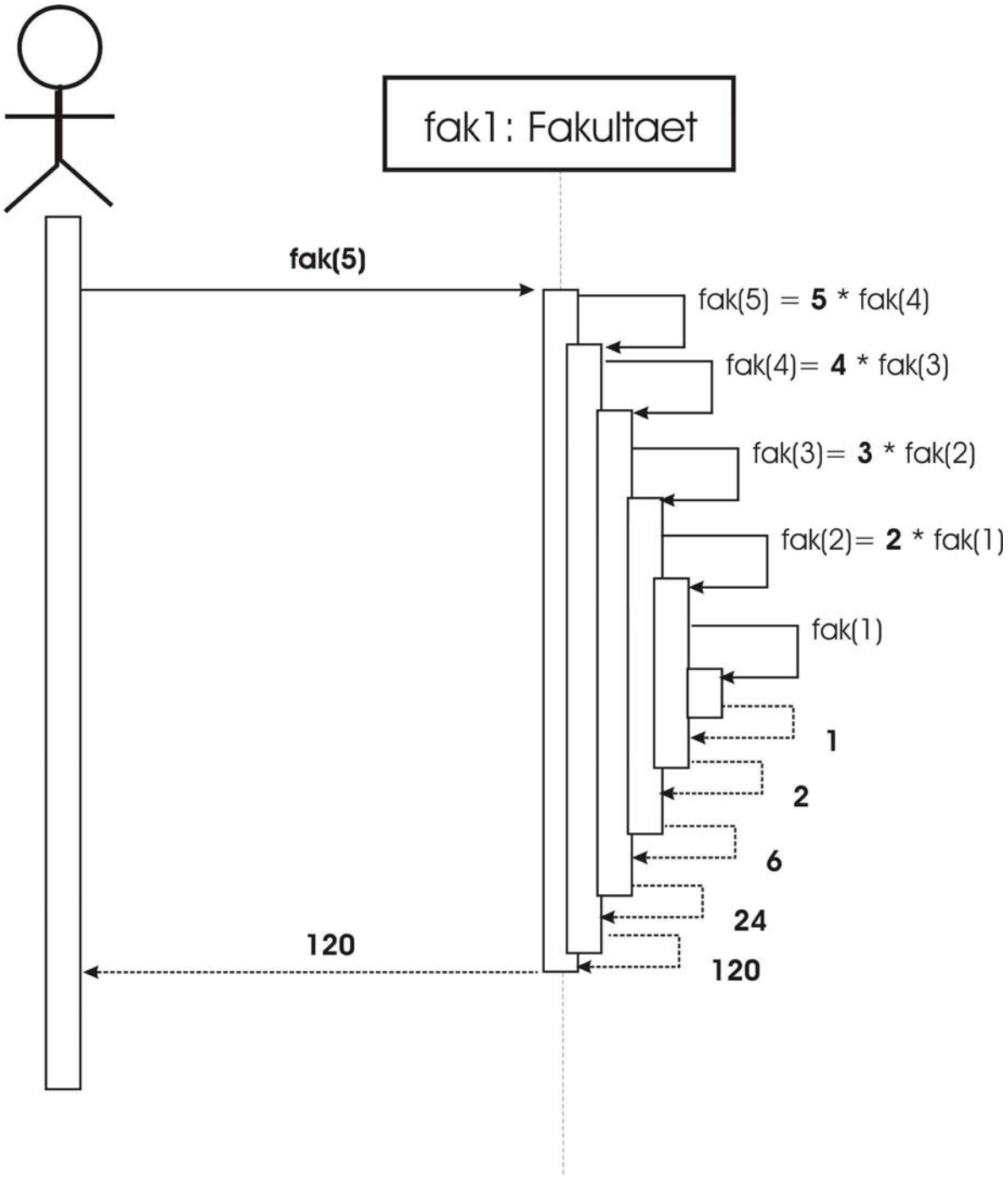


Abb. 16: Veranschaulichung der rekursiven Methodenaufrufe der Fakultätsfunktion fak(5) durch das Sequenzdiagramm

Exkurs: Die Türme von Hanoi

Die Türme von Hanoi sind ein klassisches Problem der Informatik, das mit einem rekursiven Aufruf gelöst werden kann. Unterschiedlich große runde Scheiben befinden sich auf einem von drei Stäben zu einer Pyramide aufgetürmt. Die einzelnen Scheiben sollen nun von einem zu einem anderen Platz verschoben werden. Dabei gilt, dass nur jeweils eine Scheibe bewegt werden darf und niemals eine größere über einer kleineren Scheibe liegen darf. Am Ende des Spiels soll der gesamte Turm von seiner Startposition auf seine Spielposition verschoben sein.

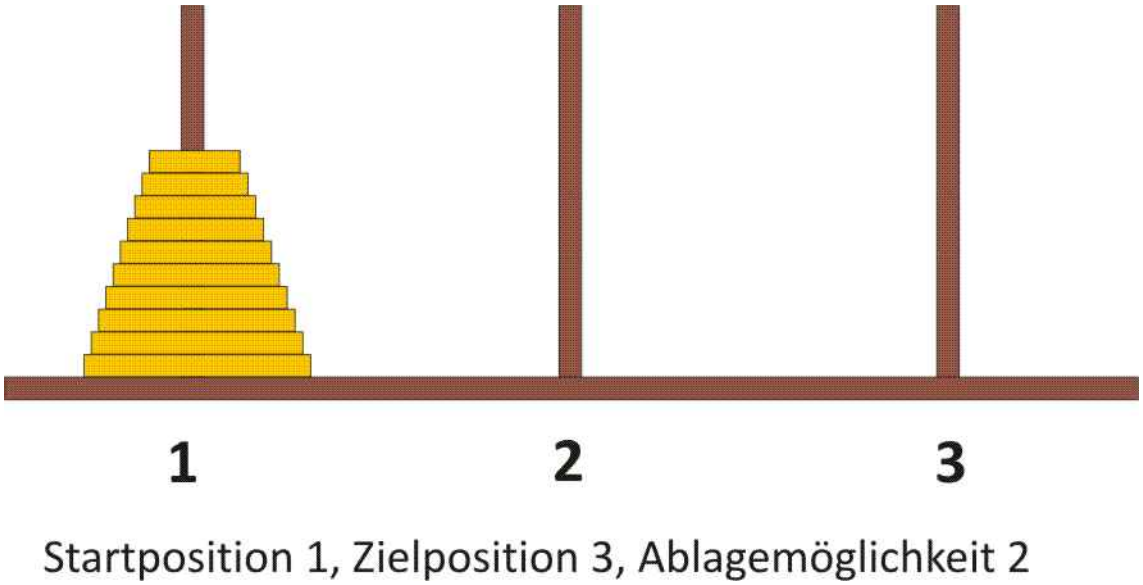


Abb. 17: Spielbrett mit den drei Stäben und 10 Scheiben

Durch Ausprobieren gelingt es einem Spieler mit der Zeit sehr geschickt, Türme bis zu einer Höhe von vielleicht sieben Scheiben von der Start- zur Zielposition zu verschieben. Die Anzahl der notwendigen Scheibenbewegungen nimmt mit der Anzahl der Scheiben des Turmes sehr rasch zu. Im idealen Fall sind 2^n-1 Bewegungen notwendig.

Das Interessante an diesem Problem ist, dass es mithilfe eines rekursiven Methodenaufrufs leicht gelöst werden kann. Wir wollen es kurz skizzieren, weil dadurch das Verständnis für rekursive Prozesse gestärkt wird.

Zunächst soll der Methodenrumpf geschrieben werden, damit klar ist, was von der Software erwartet wird.

Listing 22: Hanoi - Methodenrumpf

```
#1  public class Hanoi{
#2
#3
#4    public void verschiebeTurm(int start, int ziel, int hoehe)
#5    {
#6
#7
#8
#9    }
#10
```

```
#11
```

Als Parameter werden die Startposition und die Zielposition benötigt sowie die Anzahl der Scheiben. Unser Beispielprogramm soll lediglich als Text ausgeben, von wo nach wo die einzelnen Scheiben zu bewegen sind.

In einem nächsten Schritt suchen wir uns den einfachsten Fall aus, den wir sehr leicht beschreiben können und der als Basisfall dienen soll. Es ist das Verschieben eines Turmes der Höhe 1 von der Startposition zur Zielposition. Es sollte uns keinerlei Schwierigkeiten bereiten, den Algorithmus zur Lösung dieses Problems zu finden.

Listing 23: Hanoi - Basisfall

```
#1  public class Hanoi{
#2
#3
#4    public void verschiebeTurm(int start, int ziel, int hoehe)
#5    {
#6      if (hoehe==1)
#7      {
#8         System.out.println("Scheibe von "+start+" nach "+ziel+" bewegen.";
#9      }
#10   }
#11
```

Das Listing 23 sieht ziemlich banal aus. Als nächstes benötigen wir nur noch einen Rekursionsschritt. Diesen zu finden dürfte nicht leicht sein. Dabei geht es um die Frage, ob man einen Turm, der aus mehr als einer Scheibe besteht, durch einen rekursiven Aufruf dem Basisfall einen Schritt näher bringen kann. Und tatsächlich existiert eine solche Möglichkeit wie in der folgenden Abbildung gezeigt wird. Es sollen n=10 Scheiben von der Position 1 nach der Position 3 verschoben werden. Dies ist offensichtlich nicht der Basisfall, da nicht eine sondern zehn Scheiben verschoben werden müssen.

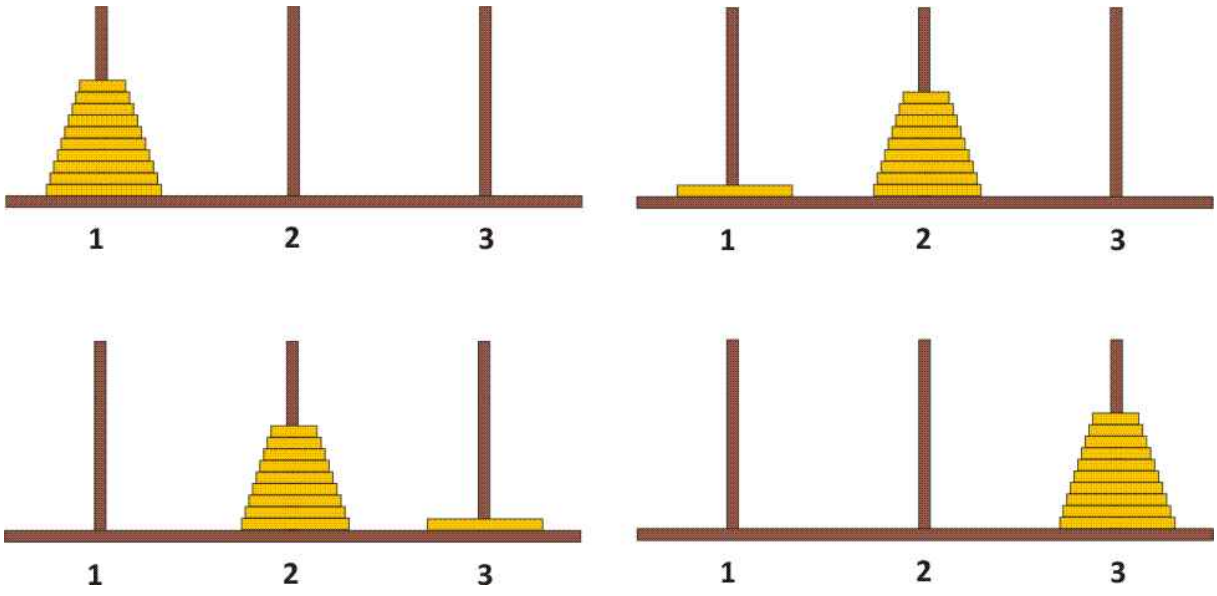


Abb. 18: Veranschaulichung des rekursiven Algorithmus von Türme von Hanoi

Wir wagen ein Gedankenexperiment: Zwar wissen wir nicht, wie wir diese Aufgabe lösen sollen. Wenn wir aber wüssten, wie wir neun Scheiben von der Startposition 1 auf die Zwischenposition 2 verschieben könnten, dann könnten wir das gesamte Problem in drei Schritten lösen.

1. Verschiebe einen Turm mit n-1 Scheiben von der Startposition auf die Zwischenposition.
2. Verschiebe einen Turm der Höhe 1 von der Startposition auf die Zielposition.
3. Verschiebe einen Turm mit n-1 Scheiben von der Zwischenposition auf die Zielposition.

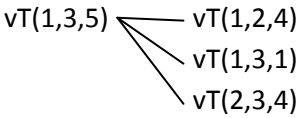
Der zweite Schritt entspricht dem Basisfall, der jetzt schon durch rekursives Aufrufen gelöst werden kann. Aber auch die anderen beiden Anweisungen haben das Problem, einen Turm der Höhe 10 zu verschieben, einen Rekursionsschritt in Richtung Basisfall vereinfacht.

Listing 24: Hanoi – Basisfall und Rekursionsschritt

```
#1  public class Hanoi{
#2
#3
#4    public void verschiebeTurm(int start, int ziel, int hoehe)
#5    {
#6      int zwischen=6-start-ziel;
#7      if (hoehe==1)
#8      {
#9         System.out.println("Scheibe von "+start+" nach "+ziel+" bewegen.";
#10      }
#11     else
#12     {
#13        verschiebeTurm(start, zwischen, hoehe-1);
#14        verschiebeTurm(start, ziel, 1);
#15        verschiebeTurm(zwischen, ziel, hoehe-1);
#16     }
#17   }
#18
#19
```

In Zeile 6 des Listings 24 wird die Position der „Zwischenablage“ berechnet. Wir verwenden für das Beschreiben Scheibenpositionen Nummern 1, 2 und 3. Wenn Start und Ziel bekannt ist kann die Zwischenposition leicht berechnet werden. Dazu werden beide Werte von der Summe der drei Positionen abgezogen. Die Summe der drei Positionen ist 6, so erhält man die Zwischenposition durch die Formel `zwischen=6-start-ziel`.

Der eine oder andere Leser wird nun vielleicht meinen, dass weitere Schritte notwendig wären, um das Problem „Türme von Hanoi“ zu lösen. Dies ist jedoch nicht der Fall. Der Algorithmus ist bereits fertig. Wir wollen uns die rekursiven Methodenaufrufe des Verschiebens eines Turmes der Höhe 5 von der Startposition 1 zur Zielposition 3 einmal genauer anschauen. Der Übersichtlichkeit wegen wird der Methodenname verschiebeTurm(…) mit vT(…) abgekürzt. Jeder Methodenaufruf, der eine Höhe von größer 1 hat, zerfällt in drei Methodenaufrufe, die dem Basisfall um einen Rekursionsschritt näher sind, z. B.



Die Spalte 1 wird in die Spalte 2 aufgelöst, sie sind gleichbedeutend. Die Anweisung vT(1,3,1) kann jedoch erst ausgeführt werden, wenn die vorhergehende Anweisung vT(1,2,4) vollständig beendet ist. Dies erfordert aber den mehrmaligen rekursiven Aufruf.

Funktionenbaum (Aufrufsequenz) der Türme von Hanoi mit der Höhe von fünf Scheiben

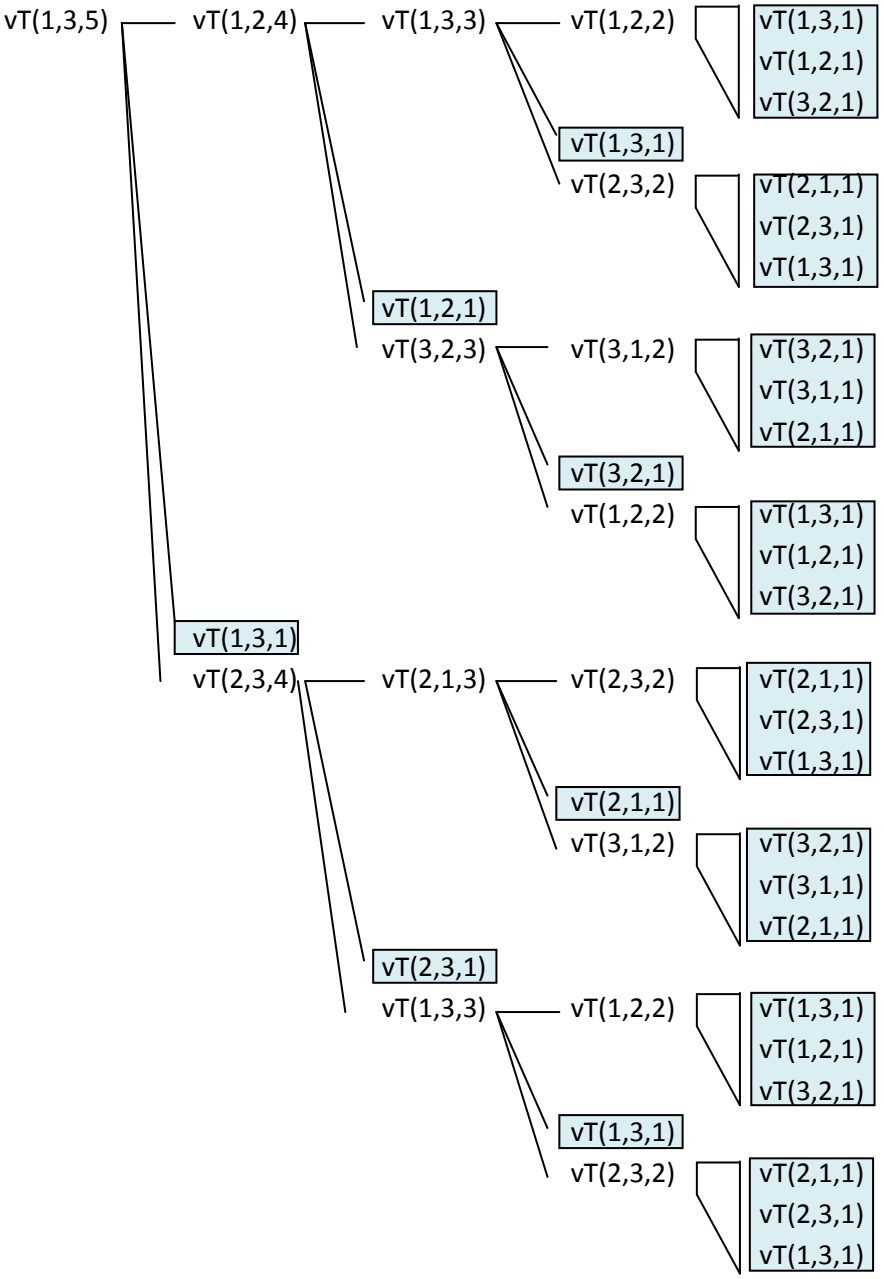
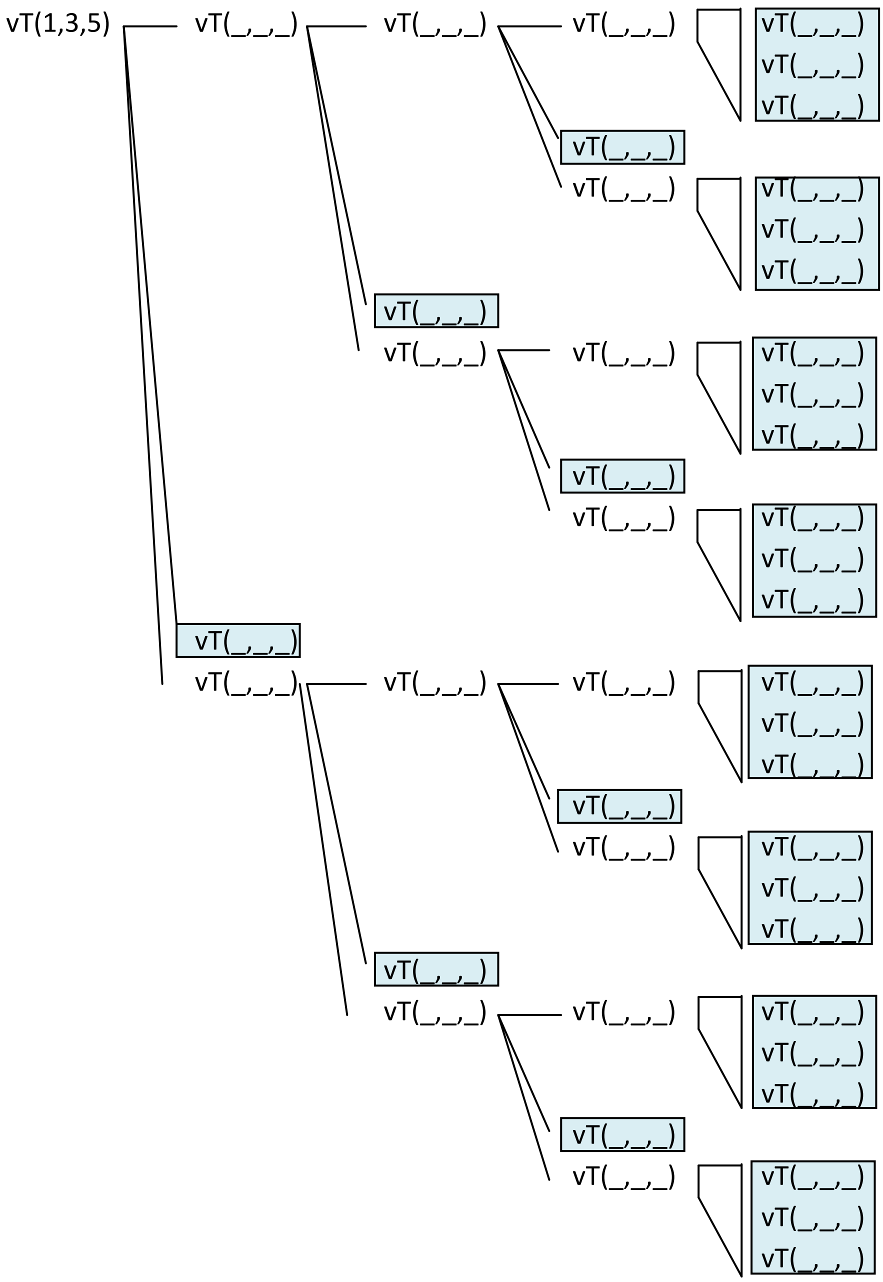


Abb. 19: Es muss 31 Mal eine Scheibe bewegt werden, was hier an den blauen Kästchen zu erkennen ist.

Arbeitsblatt: Funktionenbaum (Aufrufsequenz) der Türme von Hanoi mit der Höhe von fünf Scheiben



Es muss 31 Mal eine Scheibe bewegt werden, was hier an den grauen Kästchen zu erkennen ist.

1.6 Die rekursive Programmierung einer Liste

1.6.1 Die Methode `size()`

Wir wollen nun die Klasse Liste erweitern um eine Methode, die uns die Anzahl der Knoten zurückliefert. Da nirgendwo der Wert gespeichert ist, muss sie berechnet werden. Wir müssen also die ganze Liste von vorne bis hinten durchlaufen und alle Elemente zählen. Dies gelingt, da die Knoten jeweils eine Referenz auf ihren Nachfolger besitzen, über einen rekursiven Methodenaufruf.

Wie in Abbildung 16 für die Fakultätsfunktion wird auch hier jedes Element seinen Nachfolger aufrufen. Wir benötigen eine Idee, wie wir daraus die Länge der Liste ermitteln können. Für den einfachsten Fall der Länge 0 ist das Ergebnis leicht anzugeben. Wenn es kein Anfangselement gibt, dann ist die Länge 0. Für jeden Rekursionsschritt, hier der Aufruf des Nachfolgers, vergrößert sich die Länge um 1. Der Algorithmus kann also leicht angegeben werden.

Listing 25: Der Aufruf der Methode `size()` in der Klasse Liste

```
#1  public class Liste{
#2
#3  …
#4    public int size()
#5    {
#6      if (anfang==null) { return 0; }
#7      else return
#8      {
#9         anfang.size();
#10     }
#11     // falls Fehler auftreten
#12     return -1;
#13 …
#14
```

In der Klasse Liste wird geprüft, ob die Liste leer ist, in diesem Fall wird der Wert 0 zurückgeliefert. Ansonsten wird die rekursive Methode size() der Klasse Knoten aufgerufen. Die eigentliche rekursive Methode. Die Zeile 12 dient lediglich zur Übersetzung der Klasse, da Java eine Return-Anweisung außerhalb der bedingten Anweisungen erwartet.

Listing 26: Die Länge der Liste ermitteln – Basisfall und Rekursionsschritt

```
#1  public class Knoten{
#2
#3  …
#4    public int size()
#5    {
#6      if (nachfolger==null) { return 1; }  // Basisfall, Ende der Rekursion
#7      else return
#8      {
#9         1+nachfolger.size();  // Rekursiver Aufruf
#10     }
#11     // falls Fehler auftreten
#12     return -1;
#13 …
#14
```

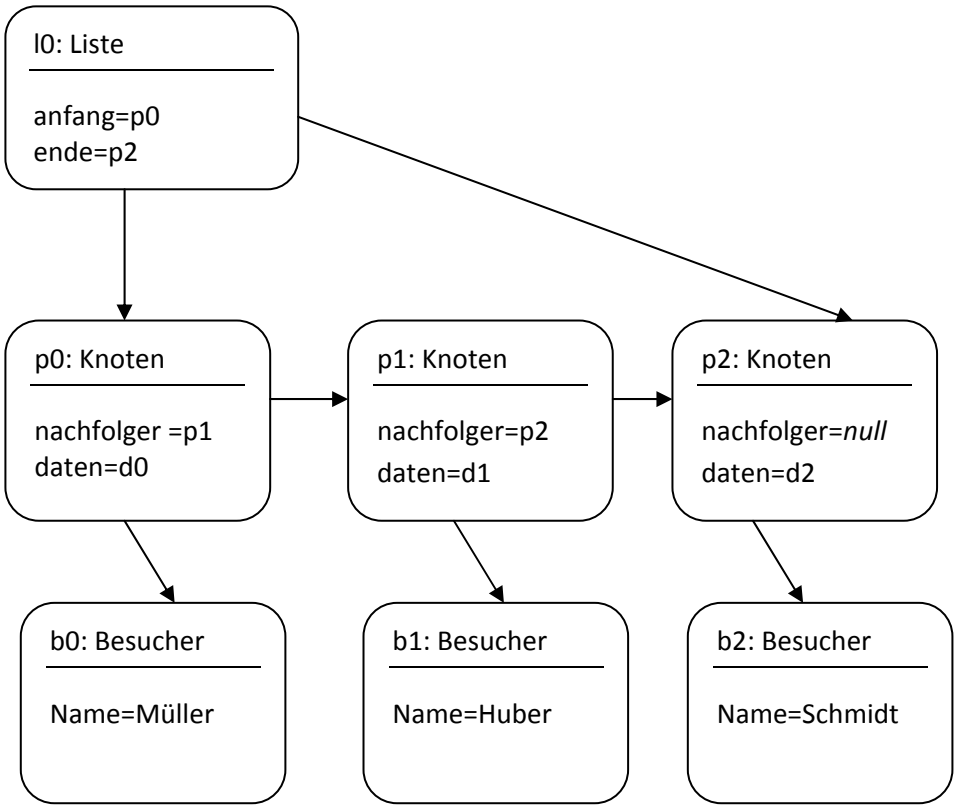


Abb. 20: Objektdiagramm einer Liste mit drei Knoten

Das Sequenzdiagramm zeigt folgendes Bild

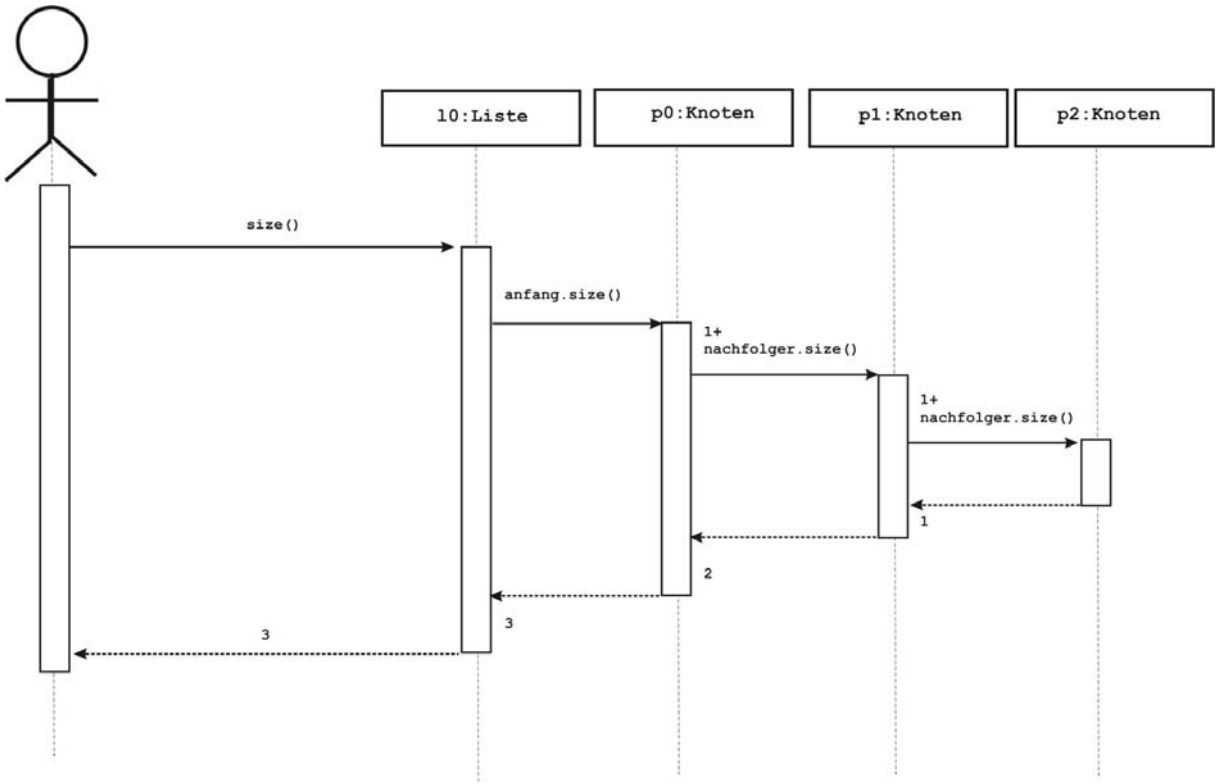


Abb. 21: Sequenzdiagramm zeigt den rekursiven Aufruf der Methode Knoten.size()

Der Methodenaufruf zeigt dasselbe Geschehen

```
l0.size()
  p0.size()
    1+p1.size()
      1+1+p2.size()
      1+1+1
    1+2
  3
3
```

Struktogramm zur Methode

Die Abbruchbedingung, der Basisfall und der rekursive Aufruf (Rekursionsschritt) sind gut nachvollziehbar.

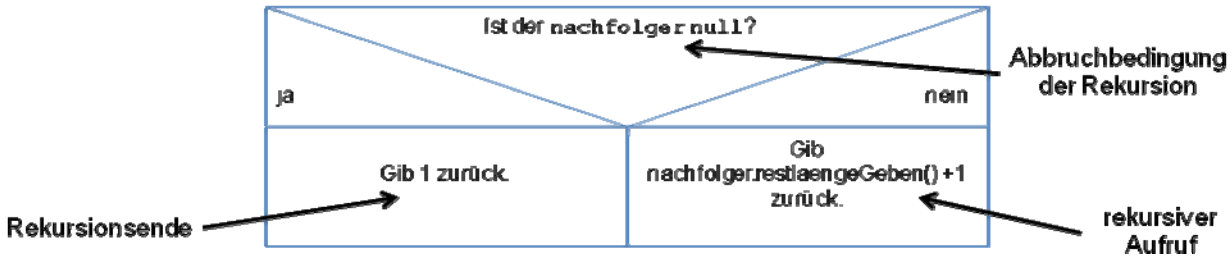


Abb. 22: Struktogramm der Methode size();

1.6.2 Übung zur Programmierung

Im Folgenden wollen wir unsere Warteschlange weiter entwickeln. Dazu benötigen wir – weil wir rekursive Methoden entwickeln werden – keine Referenz mehr von der Klasse Liste auf das letzte Element.

Aufgabe: Implementieren Sie das folgende Klassendiagramm. Beachte bitte, dass keine Referenz mehr auf das letzte Listenelement existiert weil wir diese Information auch über rekursive Methodenaufrufe erhalten können.

Wo dir keine Lösung einfällt, genügt die Implementierung der Methodenrümpfe. Auf den folgenden Seite findest du dann auch den Abdruck der Dokumentation, wie Java sie automatisch erstellt, wenn die Parameter und die Ergebnisdatentypen in einer speziellen Syntax kommentiert in den Quelltext hinzugefügt wurden.

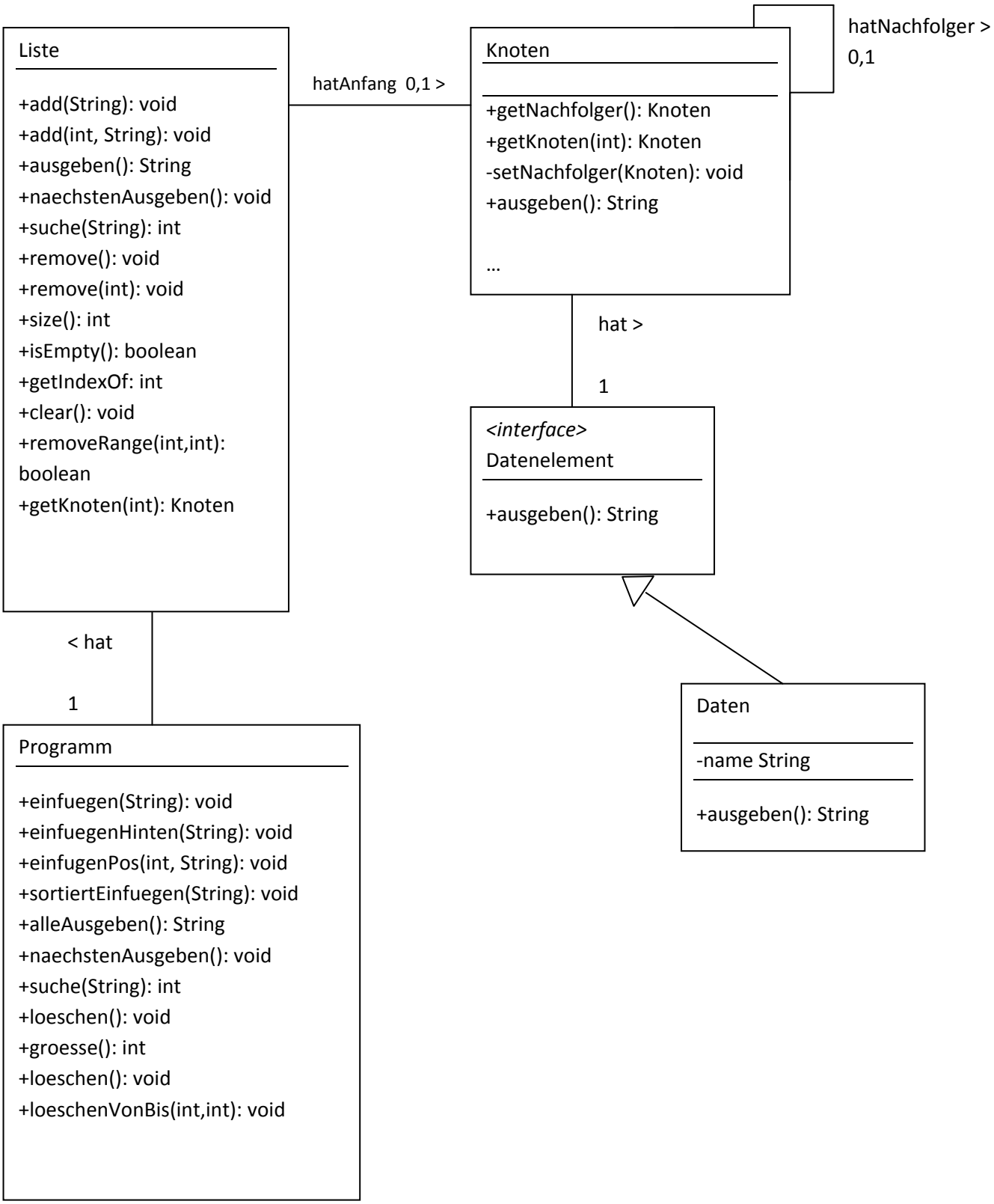


Abb. 23: Klassendiagramm eines Programms mit Verwendung einer Liste

1.6.3 Die Java-Dokumentation

1.6.3.1 Die Java-Dokumentation der Klasse Liste

Class Liste

`java.lang.Object`

`Liste`

`public class` **`Liste`** `extends java.lang.Object`

Diese Klasse implementiert eine Liste, die Individuell für verschiedene Datenelemente verwendet werden kann. Die Daten werden in Knoten verwaltet, die wiederum jeweils eine Referenz auf ein Datenobjekt besitzen.

Version:

2009-11-21

Constructor Summary	
`Liste``()`	Erzeuge eine leere Liste

Method Summary	
`void`	**`add`**`(int i, java.lang.String name)` Vor der mit i angegebenen Position wird ein neuer Knoten eingefügt.
`void`	**`add`**`(java.lang.String name)` Neuer Knoten an das Ende der Liste anhängen.
`java.lang.String`	**`ausgeben`**`()` Den Inhalt des Datenelements ausgeben.
`void`	**`clear`**`()` Löscht alle Knoten in der Liste.
`int`	**`getIndexOf`**`(java.lang.String s)` Liefert den Index des Knotens, in dessen Datenelement sich der Wert eines Strings befindet.
`Knoten`	**`getKnoten`**`(int i)` Eine Referenz auf einen Knoten ausgeben.
`boolean`	**`isEmpty`**`()` Pruefung, ob die Liste leer ist.
`void`	**`remove`**`()` Löscht das Element am Anfang der Liste (mit dem Index 0).
`void`	**`remove`**`(int i)` Löscht das Elemnt an der mit dem Index i angegebenen Stelle.
`boolean`	**`removeRange`**`(int a, int e)` Löscht alle Knoten in der angegebenen Spanne.

int	**size**() Die Anzahl der Knoten in der Liste ermitteln.

Constructor Detail

Liste
```
public Liste()
```
Erzeuge eine leere Liste

Method Detail

add
```
public void add(int i,java.lang.String name)
```
Vor der mit i angegebenen Position wird ein neuer Knoten eingefügt.

Parameters:

i - der Index für die Position, vor dem der neue Knoten eingefügt wird

name - der String, mit dem das neue Datenelement erzeugt wird

---

add
```
public void add(java.lang.String name)
```
Neuer Knoten an das Ende der Liste anhängen.

Parameters:

name - der String, mit dem das neue Datenelement erzeugt wird

---

ausgeben
```
public java.lang.String ausgeben()
```
Den Inhalt des Datenelements ausgeben.

Returns:

den Inhalt eines Datenelements. Erhält den Wert "Keine Information vorhanden" wenn die Liste leer ist.

---

clear
```
public void clear()
```
Löscht alle Knoten in der Liste.

---

getIndexOf
```
public int getIndexOf(java.lang.String s)
```
Liefert den Index des Knotens, in dessen Datenelement sich der Wert eines Strings befindet.

Parameters:

s - ist der String, mit dem das Datenelement gesucht wird.

Returns:

den Index des gesuchten Datenelements, wenn kein Knoten gefunden wurde, wird -1 zurükgeliefert.

---

getKnoten
```
public Knoten getKnoten(int i)
```
Eine Refernz auf einen Knoten ausgeben.

Parameters:

i - der Index für den Knoten, der ausgegeben werden soll.

Returns:

Refernz auf den Knoten

isEmpty
```
public boolean isEmpty()
```
Pruefung, ob die Liste leer ist.

Returns:

true, wenn die Liste leer ist.

---

remove
```
public void remove()
```
Löscht das Element am Anfang der Liste (mit dem Index 0).

---

remove
```
public void remove(int i)
```
Löscht das Elemnt an der mit dem Index i angegebenen Stelle.

Parameters:

`i` - ist der Index des zu löschenden Knotens.

---

removeRange
```
public boolean removeRange(int a,
                           int e)
```
Löscht alle Knoten in der angegebenen Spanne.

Parameters:

a - ist der erste Knoten der gelöscht werden soll.

b - ist der letzte Knoten der gelöscht werden soll. return true, wenn der Löschvorgang erfolgreich war, false, wenn a oder e außerhalb des zulässigen Bereichs lagen.

---

size
```
public int size()
```

1.6.3.2 Die Java-Dokumentation der Klasse Knoten

Class Knoten
```
java.lang.Object
```

Knoten
```
public class Knotenextends java.lang.Object
```

Knoten(java.lang.String n)

java.lang.String	**ausgeben**()
int	**getIndexOf**(int w, java.lang.String s)
Knoten	**getKnoten**(int i)
Knoten	**getNachfolger**()
void	**setNachfolger**(Knoten k)
int	**size**()

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Knoten
`public Knoten(java.lang.String n)`

n - der Bezeichner des Datenelements, nach dem auch gesucht werden kann (Schlüssel)

ausgeben
`public java.lang.String ausgeben()`

getIndexOf
```
public int getIndexOf(int w,
                      java.lang.String s)
```

wsgetKnoten
`public Knoten getKnoten(int i)`

igetNachfolger
`public Knoten getNachfolger()`

setNachfolger
`public void setNachfolger(Knoten k)`

ksize
`public int size()`

1.6.3.3 Die Java-Dokumentation des Interfaces Datenelement

Interface Datenelement

`interface Datenelement`

Interface (entspricht einer abstrakten Klasse, in der alle Methoden abstrakt sind)

java.lang.String	**ausgeben**()

ausgeben
`java.lang.String ausgeben()`

1.6.3.4 Die Java-Dokumentation der Klasse Daten

Class Daten

`java.lang.Object`

Daten

`public class Daten extends java.lang.Object`

Die Klasse speichert die Daten zu jedem Knoten.

(package private) java.lang.String	**name**

Daten(java.lang.String n)

java.lang.String	**ausgeben**()
java.lang.String	**getName**()
void	**setName**(java.lang.String n)

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

name
java.lang.String **name**

Daten
public **Daten**(java.lang.String n)
n - Bezeichnung für den Datenwert.

ausgeben
public java.lang.String **ausgeben**()

getName
public java.lang.String **getName**()

setName
public void **setName**(java.lang.String n)

Diese Dokumentation zeigt lediglich die Signaturen der Konstruktoren und Methoden an. Die Implementierung ist nicht zu sehen. Trotzdem können die Methodenrümpfe schon angelegt werden. Bei Methoden, die einen Rückgabewert haben, stellt sich das Problem, dass ein Kompilieren eines leeren Methodenrumpfes nicht möglich ist. Der Compiler übersetzt das Programm nur, wenn eine Return-Anweisung vorhanden ist. Für Methoden mit Rückgabewert kann aber eine beliebige `return`-Anweisung vorerst den Platzhalter für die spätere Implementierung übernehmen. Dadurch können wir die einzelnen Methoden sofort nach dem Erstellen prüfen. Im Folgenden werden wir uns alle Methoden nacheinander ansehen.

1.6.4 Implementierung der einzelnen Methoden

1.6.4.1 Die Methode getKnoten(int i):Knoten

Einen beliebigen Knoten anhand seines Index zu ermitteln stellt die zentrale Methode der Liste dar. Durch diese Methode lassen sich viele andere Methoden leicht implementieren.

1.6.4.1.1 Implementierung in der Klasse Liste

```
/**
   * Eine Refernz auf einen Knoten ausgeben.
   * @param i der Index für den Knoten, der ausgegeben werden soll.
   * @return Refernz auf den Knoten
   */
  public Knoten getKnoten(int i)
  {
     if ((anfang==null) || (i>size()))
     { return null; }
     else
     { return anfang.getKnoten(i); }
  }
```

In der Klasse Liste wird lediglich geprüft, ob die Liste leer ist. Ist dies der Fall wird **`null`** zurückgegeben. Wenn die Liste nicht leer ist, wird die Methode **`getKnoten(int i)`** der Klasse Knoten aufgerufen. Die eigentliche (rekursive) Methode zum Suchen des Knotens befindet sich in der Klasse Knoten.

1.6.4.1.2 Implementierung in der Klasse Knoten

```
/**
   * Getter-Methode eines mit dem Index angegebenen Knotens.
   * @param i Index der Liste
   * @return Objekt des Knotens am Index i
   */
  public Knoten getKnoten(int i){
      if (i==0) return this;
      else return nachfolger.getKnoten(i-1);
  }
```

Die Implementierung erfolgt rekursiv. Der einfachste Fall besteht darin, dass der Knoten mit dem Index 0 gesucht wird. Für diesen Fall wird die Methode eine Referenz auf sich selbst zurückliefern. Ist dies nicht der Knoten so wird das Objekt seinen Nachfolger fragen. Dabei ist der Rekursionsschritt durchzuführen. Der Index wird um 1 erniedrigt, da für den Index n noch n-1 Aufrufe erfolgen müssen, bis der rekursive Aufruf den Basisfall erreicht hat.

Basisfall: `if (i==0) return this`

Rekursionsschritt: `else return nachfolger.getKnoten(i-1)`

`// solange bis der Basisfall erreicht ist`

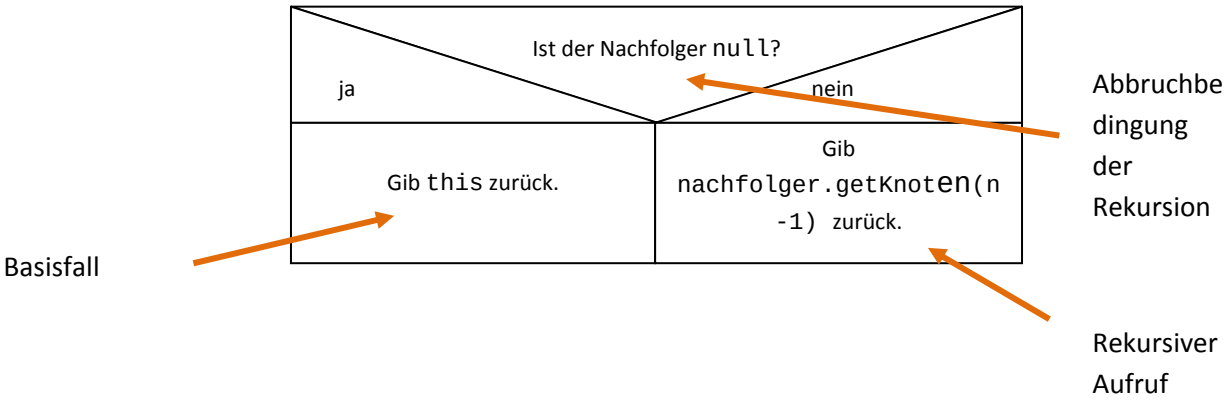


Abb. 24: Strukogramm der Methode getKnoten(int):Knoten;

Listing

```
#01  public void add(int i, String name){
#02      Knoten k=new Knoten(name);
#03
#04      if (anfang==null) {
#05          anfang=k;
#06      } else
#07      if (i==0){
#08          k.setNachfolger(anfang);
#09          anfang=k;
#10        } else
#11      {  Knoten v=getKnoten(i-1);
#12         k.setNachfolger(v.getNachfolger());
#13         v.setNachfolger(k);
#14        }
#15    }
```

In Zeile #02 wird der neue Knoten mit new Erzeugt. Es sind nun zwei Sonderfälle zu berücksichtigen. Für den Fall, dass es sich um eine leere Liste handelt (#04) wird das neue Objekt als **anfang** der Liste referenziert. Für den Fall, dass das neue Objekt am Anfang eingefügt werden soll (i==0) in Zeile #07 wird der Nachfolger des neuen Elements gesetzt. Es ist dies der (alte) Anfang der Liste. Danach wird das Objekt k selbst als anfang der Liste referenziert.

In allen anderen Fällen liefert die Methode getKnoten(i-1) in Zeile #11 eine Referenz auf den Vorgänger des neuen Knotens. Diese Referenz wird in der lokalen Variablen v gespeichert. Dessen Nachfolger v.getNachfolger() ist der neue Nachfolger des Knotens k. Der neue Nachfolger von v ist aber k selbst.

```
#01    /**
#02      * Löscht das Element an der mit dem Index i angegebenen Stelle.
#03      * @param i ist der Index des zu löschenden Knotens.
#04      */
#05    public void remove(int i){
#06      if (i==0) { anfang=anfang.getNachfolger(); } else
#07      if (i==size()-1) {getKnoten(i-1).setNachfolger(null); } else
#08      {
#09        Knoten n=getKnoten(i+1);
#10        getKnoten(i-1).setNachfolger(n);
#11      }
#12    }
```

Der Programmcode im Gesamtzusammenhang

```
/**
 * Diese Klasse implementiert eine Liste, die individuell für verschiedene
 * Datenelemente
 * verwendet werden kann. Die Daten werden in Knoten verwaltet, die wiederum
 * jeweils eine Referenz auf ein Datenobjekt besitzen.
 *
 * @author Manuel F. Friedrich
 * @version 2009-11-21
 */
public class Liste
{
    // Attribut(e)
    private Knoten anfang;

    /**
     * Erzeuge eine leere Liste
     */
    public Liste()
    {
        anfang=null;
    }

    /**
     * Pruefung, ob die Liste leer ist.
     * @return true, wenn die Liste leer ist.
     */
    public boolean isEmpty(){
        if (anfang==null) return true; else return false;
        // einfacher wäre die Anweisung --> return (anfang==null)
    }

    /**
     *  Neuer Knoten an das Ende der Liste anhängen.
     *  @param name der String, mit dem das neue Datenelement erzeugt wird
     */
    public void add(String name){
      int i=size();
      add(i,name);
    }

    /**
     * Vor der mit i angegebenen Position wird ein neuer Knoten
     * eingefügt.
     * @param i der Index für die Position, vor dem der neue Knoten eingefügt wird
     * @param name der String, mit dem das neue Datenelement erzeugt wird
     */
    // An Position einfügen
    public void add(int i, String name){
      Knoten k=new Knoten(name);
      Knoten v=getKnoten(i-1);
      if (anfang==null) {
          anfang=k;
      } else
      {
         k.setNachfolger(v.getNachfolger());
         v.setNachfolger(k);
        }
    }


    // liefert die Information über den Inhalt des ersten Elements
    /**
```

```
     * Den Inhalt des ersten Datenelements ausgeben. Das Element wird nicht
     * gelöscht.
     * @return den Inhalt eines Datenelements. Erhält den Wert "Keine Information
     * vorhanden" wenn die Liste leer ist.
     */
    public String ausgeben(){
      if (!isEmpty()){
      String n=anfang.ausgeben();
      if (anfang.getNachfolger()!=null) anfang=anfang.getNachfolger(); else
      anfang=null;
      return n;
      }
      return "Keine Informationen vorhanden.";
    }

    /**
     * Die Anzahl der Knoten in der Liste ermitteln.
     * @return die Anzahl der Knoten (kann 0 sein).
     */
    public int size(){
        if (anfang==null) { return 0;}
        else
        { return anfang.size(); }
    }

    /**
     * Eine Refernz auf einen Knoten ausgeben.
     * @param i der Index für den Knoten, der ausgegeben werden soll.
     * @return Refernz auf den Knoten
     */
    public Knoten getKnoten(int i)
    {
       if ((anfang==null) || (i>size()))
       { return null; }
       else
       { return anfang.getKnoten(i); }
    }

    /**
     * Liefert den Index des Knotens, in dessen Datenelement sich der Wert eines
     * Strings befindet.
     * @param s ist der String, mit dem das Datenelement gesucht wird.
     * @return den Index des gesuchten Datenelements, wenn kein Knoten gefunden
     * wurde, wird -1 zurükgeliefert.
     */
    public int getIndexOf(String s){
        if (anfang==null) return -1; else
        return anfang.getIndexOf(0,s);
    }

    /**
     * Löscht das Element am Anfang der Liste (mit dem Index 0).
     */
    public void remove(){
        remove(0);
    }

    /**
     * Löscht das Elemnt an der mit dem Index i angegebenen Stelle.
     * @param i ist der Index des zu löschenden Knotens.
     */
    public void remove(int i){
        if (i==0) { anfang=anfang.getNachfolger(); } else
        if (i==size()-1) {getKnoten(i-1).setNachfolger(null); } else
        {
            Knoten n=getKnoten(i+1);
            getKnoten(i-1).setNachfolger(n);
```

```
        }
    }

    /**
     * Löscht alle Knoten in der Liste.
     */
    public void clear(){
        anfang=null;
    }

    /**
     * Löscht alle Knoten in der angegebenen Spanne.
     * @param a ist der erste Knoten der gelöscht werden soll.
     * @param b ist der letzte Knoten der gelöscht werden soll.
     * return true, wenn der Löschvorgang erfolgreich war, false, wenn a oder e
     * außerhalb des zulässigen Bereichs lagen.
     */
    public boolean removeRange(int a, int e){
        if ((a>=0) && (e<=size()-1))
        {
          int b=a;
          for (int i=a; i<=e; i++){
            remove(b);
          }
          return true;
        }
        else return false;

    }

}
```

1.7 Kompositum

1.7.1 Vererbung schafft Vorteile: Intelligente Objekte

Im Folgenden wollen wir eine weitere Vererbungsstruktur aufbauen. Ziel ist es, eine so genannte Komposition zu implementieren. Das *Entwurfsmuster der Komposition* wird in einem späteren Kapitel noch ausführlich behandelt. An dieser Stelle sollen ein paar wenige Worte genügen: Bei einer *Komposition* werden verschiedene Komponenten zu einem ganzen zusammengefügt. Dabei handelt es sich um etwas Ähnliches wie bei der Implementierung einer Beziehung, die über ein Referenz-Attribut realisiert wird. Bei einer Referenz erhält ein Objekt Zugriff auf die Methoden eines anderen Objektes. Als Komposition benennt man die Struktur, wenn die einzelnen Objekte nicht alleine existieren können, sondern in einer baumartigen Struktur nach dem Prinzip „ist-ein-Teil-von“ (engl.: *whole-part*) vorhanden sind. Ein Beispiel für eine *Komposition* stellt z. B. die GUI dar, die im letzten Kapitel erstellt wurde. Ein Textfeld kann ohne Fenster nicht existieren, es ist ein Teil davon. Ähnlich verhält es sich mit allen anderen Bedienelementen des Fensters. Auch unsere Liste besteht aus Elementen, die alleine nicht existieren sollen.

Für eine Komposition gelten folgende Grundsätze:

- Es liegt eine *whole-part*-Beziehung vor.
- Die Kardinalität der Beziehung beträgt 1:n
- Die Lebensdauer der Teile ist maximal so lang, wie die des Ganzen
- Das Ganze ist verantwortlich für das Erzeugen der einzelnen Teile

Anstelle, eine Liste direkt auf den ersten Knoten zeigen zu lassen wollen wir hier wiederum ein Interface bzw. eine abstrakte Klasse verwenden, die wir Listenelement nennen. Davon werden zwei Klassen abgeleitet, die bekannte Klasse *Knoten* eine Klasse *Abschluss*. Das Klassendiagramm hat folgendes Aussehen.

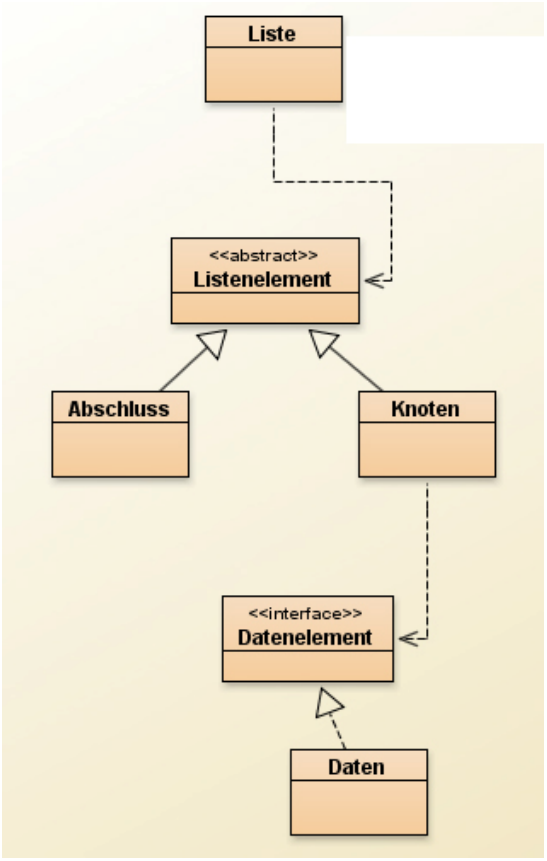


Abb. 25: Klassendiagramm mit Kompositum

Wir wollen damit erreichen, dass unser Programmcode übersichtlicher wird.

1.7.2 Unser bisheriges Problem

Unsere Liste ist so gestaltet, dass eine Referenz der Klasse Liste auf den ersten Knoten zeigt und von dort jeweils auf den Nachfolger. Das Ende der Liste ist dadurch bestimmt, dass der Attributwert für den Nachfolger beim letzten Knoten den Wert *null* besitzt.

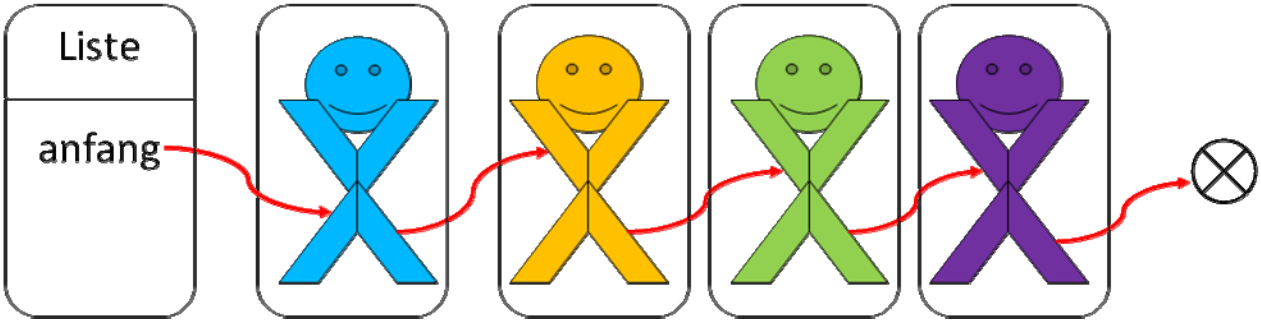


Abb. 26: Vereinfachtes Objektdiagramm ohne Verwendung eines Abschlussobjektes

Dadurch ergibt sich allerdings erheblicher Programmieraufwand, weil in der Klasse Liste stets geprüft werden muss, ob die Liste leer ist und bei den Knoten stets geprüft werden muss, ob das Ende der Liste erreicht ist.

1.7.3 Lösungsansatz: Implementierung eines Abschlusselements in der Liste

Durch Einfügen eines eigenen Objektes einer Klasse Abschluss kann der Programmieraufwand stark vereinfacht werden. Der letzte Knoten zeigt auf ein vorhandenes Objekt.

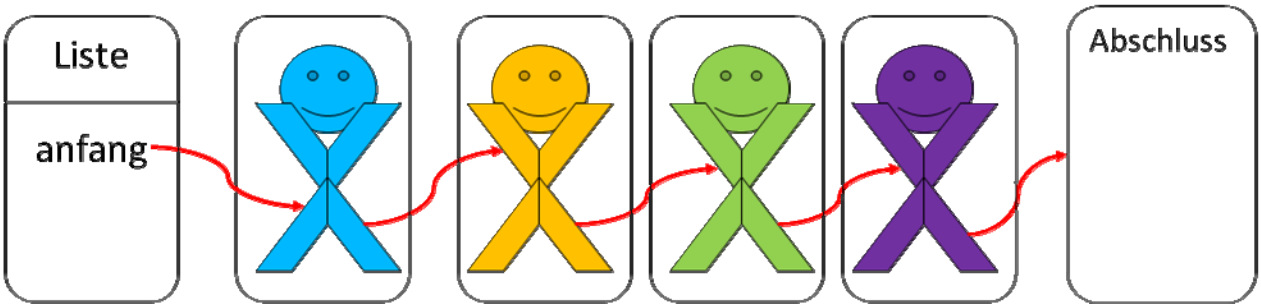


Abb. 27: Vereinfachtes Objektdiagramm mit Verwendung eines Abschlussobjektes

Die Klasse Abschluss muss über alle rekursiven Methoden verfügen, wie sie auch in der Klasse Knoten vorhanden sind. Allerdings ist die Implementierung eine andere. Wie an diesem Beispiel gezeigt wird, vereinfacht sich dadurch die Programmierung, da viele bedingte Anweisungen (if-Anweisungen) wegfallen.

Dies möchte ich an den beiden Methode size() demonstrieren. Bisher kommt die Methode sowohl in der Klasse Liste als auch in der Klasse Knoten vor.

Listing 19: Methode size() in der Klasse Liste ohne Komposition

```
public int size(){
  if (anfang==null) { return 0;}
else
  { return anfang.size(); }
}
```

Listing 20: Methode size() in der Klasse Liste mit Komposition

```
public int size(){
   return anfang.size();
}
```

Da auch die leere Liste über ein Abschlusselement verfügt, kann ohne Prüfung die Methode `size()` der Referenz `anfang` aufgerufen werden.

Listing 20: Methode size() in der Klasse Knoten ohne Komposition

```
public int size(){
  if (nachfolger==null) { return 1; }
  else
  { return 1+nachfolger.size(); }
```

```
}
```

Diese rekursive Methode muss nun in den beiden Klassen Knoten und Abschluss vorhanden sein. Dabei wird sie unterschiedlich implementiert.

Listing 21: Methode size() in der Klasse Knoten mit Komposition

```
public int size(){
      return 1+nachfolger.size();
}
```

Listing 22: Methode size() in der Klasse Abschluss mit Komposition

```
int size(){return 0;}
```

Der rekursive Aufruf der Methode `size()` in der Klasse Liste durchläuft automatisch alle Knoten solange, bis das Abschlusselement erreicht ist. Dort wird der rekursive Aufruf beendet.

Diese Vereinfachungen zeigen sich in allen rekursiven Methoden, da die Fälle der leeren Liste und das Erreichen des Endes der Liste nicht untersucht werden muss. Die Objekte sind selbst intelligent genug, den rekursiven Aufruf zu beenden.

Das Entwurfsmuster des Kompositums bringt aber auch Nachteile. Beim Erstellen einer leeren Liste muss das Abschlusselement bereits hinzugefügt werden und das Hinzufügen und Löschen von Knoten in der Liste muss stets so behandelt werden, dass das Abschlusselement erhalten bleibt.

Im Folgenden sei der gesamte Quelltext unserer Liste unter Verwendung der Struktur des Kompositums abgedruckt.

Rekursive Datenstruktur Baum

2.1 Beispiel für eine Liste eines Englisch-Wörterbuches

Schauen wir uns die Liste in der folgenden Grafik an. Es handelt sich um ein Wörterbuch, bei dem die englischen Begriffe der Reihe nach sortiert sind. Es kann nur nach englischen Begriffen gesucht werden. Das Objektdiagramm zeigt ein Objekt der Klasse Liste, diese besitzt eine Referenz auf das erste Objekt der Klasse Wort. Von dort aus hat jedes Objekt eine Referenz auf genau einen Nachfolger.

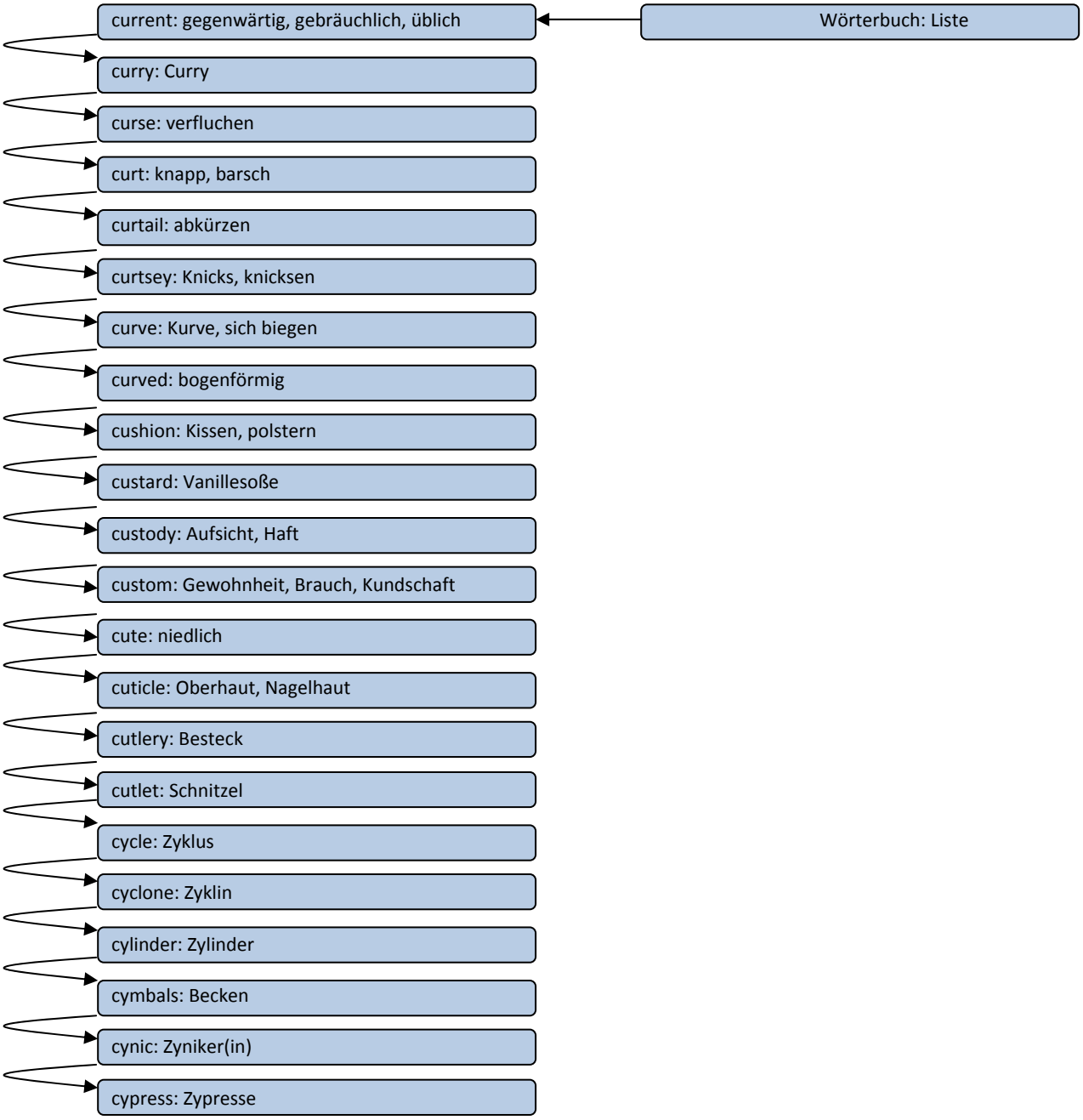


Abb. 28: Objektdiagramm einer Liste Wörterbuch

Um einen Eintrag im Wörterbuch zu finden, muss die Liste nacheinander durchsucht werden. Dabei wird das Suchwort jeweils mit dem englischen Begriff verglichen, bis es Wort gefunden ist. Bei dieser Liste handelt es sich um eine sortierte Liste, dies erleichtert die Suche: Suchen wir beispielsweise das Wort „cycade“ kann die Suche nach dem Wort „cycle“ beendet werden. Es muss nicht die gesamte Liste bis zum Ende durchsucht werden.

Da wir eine Liste verwenden, bei der wir nur eine Referenz auf das erste Wort besitzen, können wir unsere Suche immer nur vom Anfang der Liste aus starten. Um den letzten Eintrag in der Liste zu erreichen muss die gesamte Liste durchsucht werden. Abkürzungen gibt es keine. Bei vielen hunderttausend Einträgen, die ein Wörterbuch haben kann, dauert dies selbst für moderne Computer sehr lange, zu lange für uns. Wir benötigen eine Lösung für dieses Problem.

Ein Vergleich, wie wir in einem Lexikon nach einem gesuchten Wort suchen, bringt uns einem verbesserten Algorithmus näher: Wenn wir einen Eintrag in einem Lexikon suchen, gehen wir meist so vor, dass wir irgendwo in der Mitte das Buch aufschlagen und eines der Wörter mit unserem Suchwort vergleichen. So wissen wir, in welcher Hälfte des aufgeschlagenen Buches wir weitersuchen müssen. Wir nehmen von dieser Hälfte wiederum ungefähr die Mitte und führen den Vergleich ein weiteres Mal durch. Nach sehr wenigen vergleichen finden wir unser Ziel.

Um dies zu verdeutlichen, wollen wir berechnen, wie viele Vergleiche notwendig sind, um nach diesem Algorithmus ein Wort aus 1024 Wörtern zu finden. Die folgende Tabelle gibt uns das Ergebnis an. Am Anfang, also vor dem ersten Vergleich suche ich ein Wort aus 1024 Wörtern. Wenn ich das mittlere Wort mit meinem Suchwort verglichen habe, dann weiß ich, in welcher Hälfte ich weitersuchen muss. Nach einem Vergleich suche ich also noch in einem Teil des Wörterbuches, das nur noch 512 Wörter umfasst. Den beschriebenen Vorgang kann ich nun wiederholen. Ich suche die Mitte der 512 Wörter und vergleich. Nach zwei Vergleichen beschränkt sich die Suche bereits auf 256 verbleibende Worte im Wörterbuch.

Anzahl der Vergleiche	Übrig gebliebene Wörter
0	1024
1	512
2	256
3	128
4	64
5	32
6	16
7	8
8	4
9	2
10	1

Bei der Zahl 1024 handelt es sich um eine Zweierpotenz, nämlich um 2^{10}. Und wie wir sehen sind zehn Vergleiche notwendig, um das Wort zu finden. Dabei handelt es sich sogar um den

schlechtesten Fall („worst case“), das Wort könnte auch schon eher gefunden werden. Da Zweierpotenzen sehr schnell größer werden, steigt die Anzahl der Versuche nicht linear mit der Anzahl der Wörter, die das Wörterbuch enthält. Fragen wir uns, wie viele Vergleiche notwendig sind bei 16.777.216 Wörterbucheinträgen. Diese Zahl ist ebenfalls eine Zweierpotenz, es ist 2^{24}. Daher sind auch bei dieser großen Anzahl an Einträgen nur 24 Vergleiche notwendig, bis das gesuchte Wort gefunden ist. Der zusätzliche Aufwand für eine Verdopplung der Wörter in unserem Lexikon beträgt nur einen zusätzlichen Vergleich!

Diesen Algorithmus möchten wir auch auf unsere Liste anwenden. Dazu muss sie allerdings etwas umgebaut werden. Die Knoten haben nicht mehr nur einen Nachfolger, sondern zwei. Damit kann von jedem Knoten aus durch einen Vergleich die Hälfte der Wörter von der weiteren Suche ausgeschlossen werden.

Zur Veranschaulichung stellen wir uns ein Wörterbuch mit 31 Wörtern vor, die wir zunächst mit den Zahlen 0 bis 30 bezeichnen wollen. Die Liste hätte also die Form

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]

Das mittlere Element ist die 15, dieses bildet den ersten Knoten unseres Baumes. Von jeder Resthälfte suchen wir wiederum das mittlere Element, das wir als linken und rechten Nachfolger einsetzen.

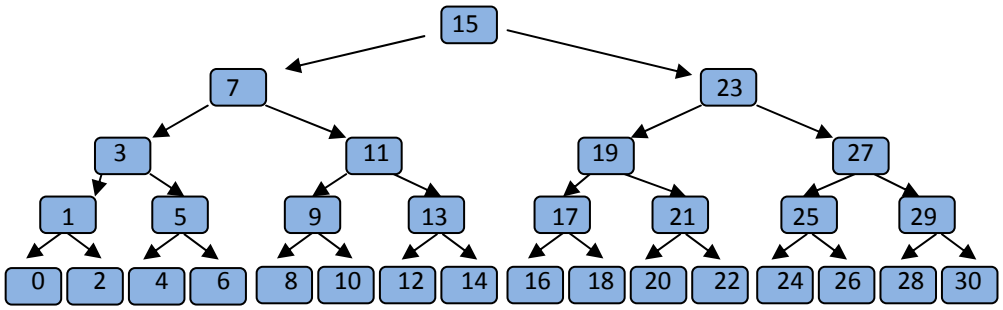


Abb. 29: Baumstruktur

Bei dieser Struktur handelt es sich um eine Baumstruktur. Damit bezeichnet man Objekte, die mit einer Wurzel beginnend folgende Eigenschaften aufweisen:

- Ein Baum besitzt eine **Wurzel**, das ist der *einzige* Knoten, der keinen Vorgänger hat.
- Ein Baum kann mehrere Nachfolger haben, jeder Knoten hat aber maximal *einen* Vorgänger.
- Die Verbindung zwischen zwei Knoten nennt man **Kante**.
- Knoten, die keinen Nachfolger haben, nennt man **Blätter**
- Mit der **Tiefe** eines Knotens wird angegeben, wie viele Vorgänger er hat, die Tiefe der Wurzel ist demnach 0.

Zusätzlich wollen wir vereinbaren, dass ein Wert in einem Baum nur einmal vorkommen soll.

Hat ein Knoten maximal zwei Nachfolger, so spricht man von einem **Binärbaum**.

Von einem **Sortierten Binärbaum** spricht man, wenn die Werte der Knoten so sortiert sind, dass der linke Nachfolger stets einen kleineren, der rechte aber immer einen größeren Wert aufweist als der Knoten selbst.

In einer ersten Aufgabe wollen wir ein Englisch-Deutsches Wörterbuch implementieren. Die Knoten sollen eine eigene Referenz auf ein Datenelement besitzen, für das eine Schnittstelle programmiert wird.

Für das sortierte Einfügen eines neuen Wörterbucheintrages benötigen wir die Methode add(String e, String d), für das Suchen eines Wortes die Methode get(String):Knoten und für die Ausgabe eines Eintrages die Methode ausgeben():String.

Schauen wir uns zuerst das Klassendiagramm an:

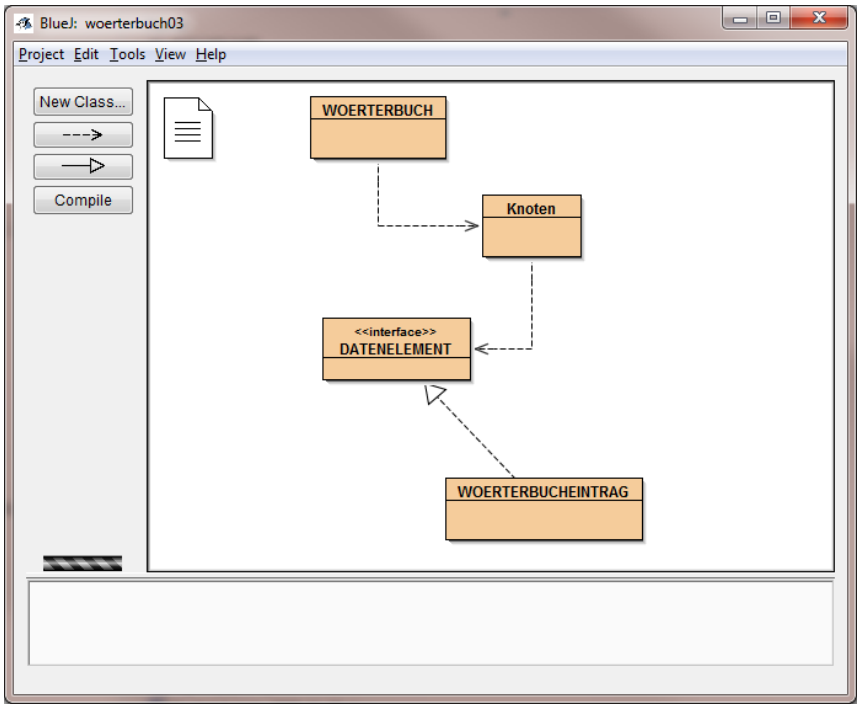


Abb. 30: Klassendiagramm des Baumes

Da es sich hier um eine Liste handelt, wie sie in Kapitel 1 beschrieben ist, sollten Sie keine Schwierigkeit haben, die Klassen zu implementieren. Lediglich das sortierte Einfügen der Knoten wurde noch nicht besprochen. Java stellt Methoden zur Verfügung, um Strings zu vergleichen.

Definition: **Zum Vergleichen von Strings steht in Java in der Klasse String die Methode `int compareTo(String e)` zur Verfügung, die einen Wert größer 0 liefert, wenn e alphabetisch hinter dem String einzuordnen ist, kleiner 0 wird zurückgeliefert für Strings, die im Alphabet vorher einzuordnen sind. Bei einem Rückgabewert von 0 handelt es sich um dasselbe Wort.**

Beispiel:

```
String e="best";
int i=0;
i=e.compareTo("best");  ergibt 0, ist also gleich
i=e.compareTo("and");  ergibt >0, ist also hinter „and“
i=e.compareTo(„zulu");  ergibt <0, ist also vor „zulu“
```

In unserem Beispiel wollen wir eine einfache Implementierung ohne das Entwurfsmuster Komposition für den Abschlussknoten verwenden. In der Klasse WOERTERBUCH müssen wir daher den Fall der leeren Liste wieder mit berücksichtigen.

```
public class WOERTERBUCH
{
    private Knoten anfang;

    public WOERTERBUCH()
    {
        anfang=null;
    }

    public void InformationAusgeben(){
       if (anfang==null)
       System.out.println("Wörterbuch leer!");
       else anfang.informationAusgeben(e);
    }

    public void SortiertEinfuegen(String e, String d){
        // neues Wort sortiert einfügen
        Knoten k=new Knoten(e,d);
        if (anfang==null) {
            k.setNachfolger(anfang);
            anfang=k;
        } else
        if ((anfang.getDatenelement().getE().compareTo(e))>0) {
            k.setNachfolger(anfang);
            anfang=k;
        }


        else
        {
            Knoten v=anfang.getVorgaenger(null,e,d);
            k.setNachfolger(v.getNachfolger());
            v.setNachfolger(k);
        }
    }


    public String Suchen(String e){
        // Wort mit englischem Begriff suchen
        if (anfang==null) return "Wörterbuch leer!"; else
        return anfang.suchen(e);
    }

}
```

Die Liste hat nur eine Referenz auf den Anfangsknoten

Erstelle eine leere Liste.

InformationAusgeben () gibt alle Einträge des Wörterbuches aus.

Neuen Knoten erstellen, wenn das Wörterbuch leer ist, an erster Stelle eintragen.

Wenn die Liste leer ist, prüfen wir, ob es an erster Stelle eingefügt werden muss, dann hat es keinen Vorgänger-Knoten.

Sonst suchen wir den Vorgänger des Knotens und fügen unseren neuen Knoten danach ein.

Wir übergeben die Suche an den Anfangsknoten, es sei denn dieser ist gar nicht vorhanden..

Auch von den anderen Klassen werden wir Dir den gesamten Quelltext zeigen:

```
public class Knoten
{
  private DATENELEMENT d;
  private Knoten nachfolger;
```

Referenzen auf das DATENELEMENT, also das englische Wort und seine Übesetzung, sowie auf den Nachfolgerknoten.

```
    public Knoten(String e,String d)
    {
         this.d=new WOERTERBUCHEINTRAG(e,d);
         nachfolger=null;
    }

    public void setNachfolger(Knoten k){
        nachfolger=k;
    }

    public Knoten getNachfolger(){
        return nachfolger;
    }

     public DATENELEMENT getDatenelement(){
        return d;
    }

    public String suchen(String e){
        if (d.getE().equals(e)) {
            return d.getD();
        } else
        if (nachfolger!=null) {
            return nachfolger.suchen(e);
        } else
        {
            return "nicht gefunden";
        }
    }

    public Knoten getVorgaenger(Knoten v, String e, String d){
        if (this.d.getE().compareTo(e)>0) {
           return v;
        } else if (nachfolger==null) return this; else
        {
            return nachfolger.getVorgaenger(this,e,d);
        }
    }

    public void informationAusgeben(){
        d.ausgeben();
        if (nachfolger!=null)
          nachfolger.informationAusgeben();
    }

}
```

Hier kann „==“ als Vergleich nicht verwendet werden, wir brauchen die Methode equals(). Den Namenskonflikt zwischen dem Parameter e und dem Attribut e müssen wir mit this.e oder mit der get-Methode getE() auflösen. Wir suchen solange rekursiv, bis das Ende der Liste erreicht ist.

Nachdem wir nur Referenzen auf Nachfolger haben, aber nicht auf Vorgänger, geben wir diesen einfach der Methode als Parameter mit! Wie lautet dann wohl der Aufruf in der Liste, wenn die Methode im Objekt anfang aufgerufen wird?

Einfacher rekursiver Aufruf der Methode für alle Nachfolger in der Liste.

```
public interface DATENELEMENT
{
    void ausgeben();
    String getE();
    String getD();
}
```

Ziemlich unspektakulär, so ein Interface. Es gibt ja auch nur an, welche Methoden in den Klassen, die es implementieren, geben muss.

```
public class WOERTERBUCHEINTRAG implements DATENELEMENT
{
    private String e;
    private String d;

    public WOERTERBUCHEINTRAG(String e, String d)
    {
        this.e=e;
        this.d=d;
    }

    public boolean isE(String e){
        if (this.e.equals(e)) return true; else return false;
    }

    public void ausgeben(){
        System.out.println(e+":"+d);
    }

    public String getE(){
        return e;
    }

    public String getD(){
        return d;
    }

}
```

Zwei Strings, alles was ein Wörterbuch braucht. Dazu einen einfachen Konstruktor und bei paar Getter-Methoden.

Ach ja, wir brauchen die Methode ausgeben(), das ist im Interface vertraglich vereinbart. Hier ist es.

2.2 Mit Bäumen geht es besser

Mit Verwendung einer Baumstruktur können wir viel schneller auf einzelne Elemente zugreifen. Wir möchten im Folgenden sortierte binäre Bäume implementieren.

Lastenheft:
In einem binären Baum sollen Elemente des Typs **`int`** sortiert eingefügt, gefunden und gelöscht werden. Später soll es möglich sein, anstelle des Typs **`int`** auch Elemente anderer Typen, z. B. Zeichenketten oder Wörterbucheinträge, zu speichern.

1. Erstelle das Klassendiagramm, verwende dabei ein Interface (Schnittstelle) von der die Klasse Datenelement für die Datenwerte der Knoten abgeleitet wird!

2. Implementiere die Klassen! Orientiere dich an der Liste! Der Unterschied ist gering. Ein Knoten hat zwei Nachfolger. Bei rekursiven Aufrufen muss eine bedingte Anweisung entscheiden, welcher Nachfolger die Aufgabe weiterführt.

   Gehe dabei in folgender Reihenfolge vor:
   a) Erstelle die Klasse **`Zahl`**, die einen **`int`**-Wert speichert!
   b) Schreibe ein <<***`Interface`***>> **`Datenelement`** zu dieser Klasse, das zumindest die

Methode `void ausgeben()` vorgibt. Du wirst ggf. später noch weitere Methoden ergänzen.
c) Schreibe die Klasse **`Baum`**, die nur eine Referenz auf den Anfangsknoten **`Knoten w`** hat
d) Schreibe die Klasse **`Knoten`**, die eine Referenz auf das **`<<Interface>>`** hat und zwei Referenzen auf den linken und rechten Nachfolger **`Knoten l`** und **`Knoten r`**!
e) Implementiere die Methode **`add(int i)`** in den Klassen **`Baum`** und **`Knoten`**!

Algorithmus zum Suchen eines Wertes in einem Baum:

1. Beginne mit der Wurzel. Ist die Wurzel der gesuchte Knoten, so bist du fertig.
2. Sonst prüfe, ob das zu suchende Wort kleiner ist als der Wert des Knotens, dann mache mit dem linken Nachfolger weiter, sonst mit dem rechten.
3. Wenn ein Knoten keinen Nachfolger hat, so stoppe den Algorithmus, das gesuchte Wort kommt nicht vor, sonst wiederhole den Ablauf ab 1. mit diesem Knoten.

Algorithmus zum Suchen eines Wertes in einem Baum:

1. Beginne mit der Wurzel. Ist die aktuelle Position leer, so lege den Knoten hier ab und stoppe.
2. Ansonsten prüfe, ob der neue Knoten kleiner ist als der aktuelle Knoten, so nehme den linken Nachfolger, sonst den rechten.
3. Beginne mit 1.

Algorithmus zum Löschen eines Wertes in einem Baum* (nicht lehrplan- und abiturrelevant)

1. Suche im Baum nach dem zu löschenden Knoten
2. Ist der Knoten nicht vorhanden, so gibt es nichts zu löschen, stoppe den Algorithmus.
3. Ist der gefundene Knoten ein Blatt, so entferne das Blatt und stoppe den Algorithmus.
4. Hat der Knoten nur einen Nachfolger, so ersetze den zu löschenden Knoten durch diesen Nachfolger.
5. Hat der Knoten zwei Nachfolger, so ersetze den zu löschenden Knoten durch den Knoten mit dem kleinsten Wert des rechten Teilbaums. Das Minimum ist zu löschen, dies ist ein eigenständiger Löschvorgang nach diesem Algorithmus. Das Minimum erkennt man daran, dass man den linken Nachfolger entlang folgt, bis es nicht mehr weiter geht.

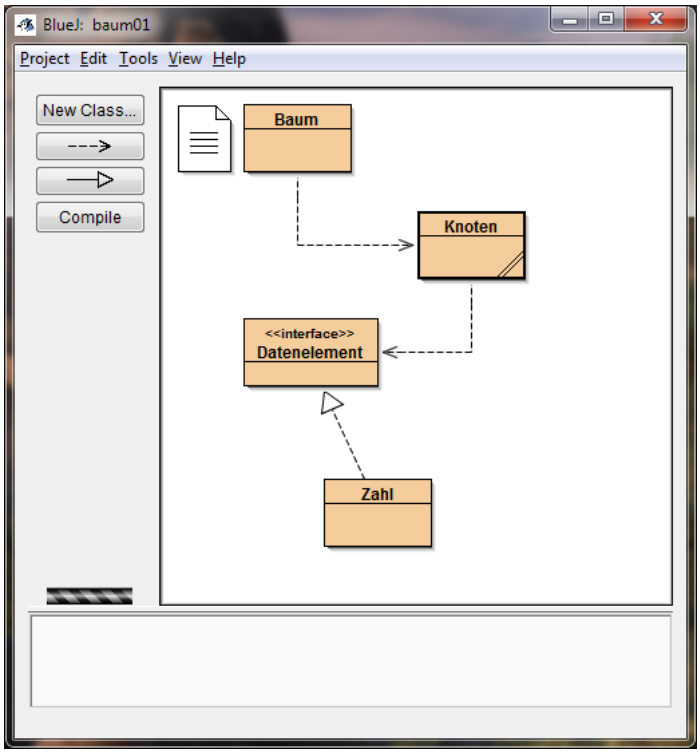


Abb. 31: Klassendiagramm des Baumes

Die Klasse Zahl

```
public class Zahl implements Datenelement
{
    private int x;

    public Zahl(int i)
    {
        x = i;
    }

    public int vergleiche(Datenelement d)
    {
        if (d.getWert()==x) return 0;
        else if (d.getWert()<x) return -1; else
        return 1;
    }

    public void ausgeben(){
        System.out.println(x);
    }

    public int getWert(){
        return x;
    }
}
```

Das Interface Datenelement

```
public interface Datenelement
```

```
{
    void ausgeben();
    int vergleiche(Datenelement d);
    int getWert();
}
```

Die Klasse Baum

```
public class Baum
{
    private Knoten w;

    public Baum() {  w=null; }

    public void add(int i)
    { Knoten k=new Knoten(i);
      if (w==null) w=k; else
      w.add(k);
    }
}
```

Die Klasse Knoten

```
public class Knoten
{
    private Datenelement wert;
    private Knoten l;
    private Knoten r;

    public Knoten(int i) { wert=new Zahl(i);
      l=null;
      r=null;
    }

    public void add(Knoten k){
        if (k.wert.vergleiche(this.wert)==0){} hier keine doppelten Einträge
        else if (k.wert.vergleiche(this.wert)>0)
        {
            if (l==null) l=k; else    wenn das Ende des Astes erreicht ist
            l.add(k);                 sonst rekursiver Aufruf
        } else
        {
            if (r==null) r=k; else    rechte Seite
            r.add(k);
        }
    }
}
```

2.3 Traversieren

Unter Traversieren versteht man, einen Baum zu durchlaufen und jeden Knoten der Reihe nach einmal zu besuchen, z. B. um seinen Wert auszugeben.

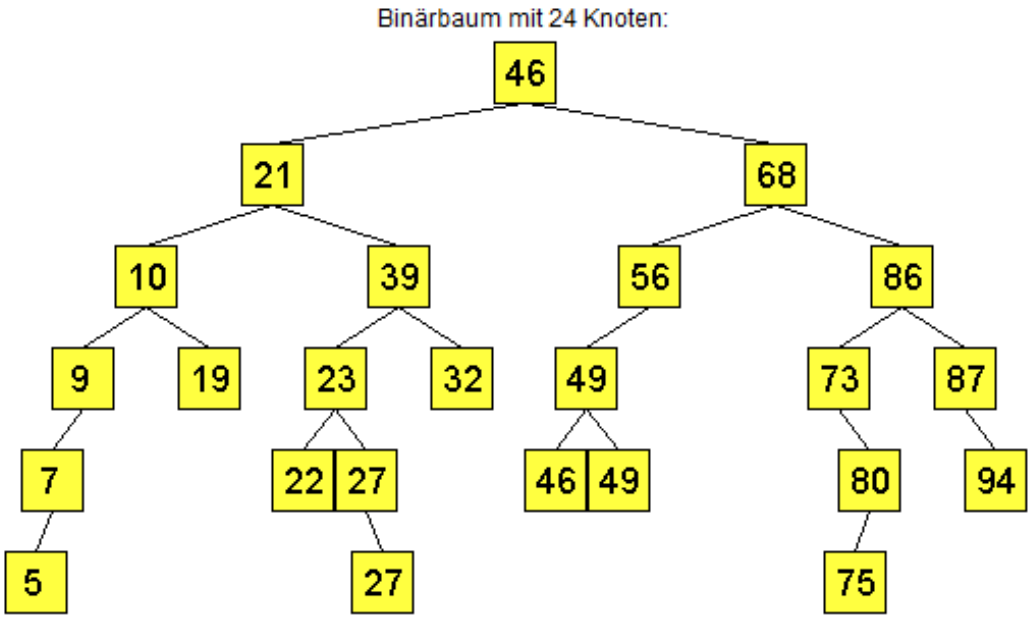


Abb. 32: Binärbaum mit 24 Knoten

Das Traversieren ist durch einen rekursiven Methodenaufruf leicht möglich. Es müssen nacheinander die linken und rechten Knoten besucht werden.

```
#01 xOrder(knoten){
#02
#03 if (knoten.links!=null)  knoten.links.xOrder();
#04
#05 if (knoten.rechts!= null) knoten.rechts.xOrder()
#06
#07 }
```

Von der Wurzel ausgehend wird zuerst jeweils der linke Ast durchsucht bis es nicht mehr weiter geht. Dabei wird aber jeweils nur die erste Anweisung der Methode xOrder ausgeführt. Jeder Besuchte Knoten wird danach ein zweites Mal besucht, um die verbleibende Anweisung auszuführen und dem rechten Ast zu folgen.

Eine Anweisung knoten.ausgeben(); kann nun in Zeile #02, #04 oder #06 eingefügt werden. Man unterscheidet die drei Möglichkeiten des Traversierens durch die Namen preOrder(knoten), inOrder(knoten) und postOrder(knoten), da die Reihenfolge der ausgegebenen Zahlen unterschiedlich ist.

Listing

```
#01 preOrder(knoten){
```

```
#02 knoten.ausgeben();
#03 if (knoten.links!=null)  knoten.links.preOrder();
#04
#05 if (knoten.rechts!= null) knoten.rechts.preOrder()
#06
#07 }
```

```
#01 inOrder(knoten){
#02
#03 if (knoten.links!=null)  knoten.links.inOrder();
#04 knoten.ausgeben();
#05 if (knoten.rechts!= null) knoten.rechts.inOrder()
#06
#07 }
```

```
#01 postOrder(knoten){
#02
#03 if (knoten.links!=null)  knoten.links.postOrder();
#04
#05 if (knoten.rechts!= null) knoten.rechts.postOrder()
#06 knoten.ausgeben();
#07 }
```

Bei einem sortierten Binärbaum führt die inOrder-Methode des Traversierens zu einer sortierten Ausgabe aller Werte.

Für den in Abb. 34 dargestellten Baum ergibt sich folgender Aufrufbaum:

preOrder(46){
knoten.ausgeben();
if (knoten.links
!=**null)**

preOrder(21){
knoten.ausgeben();
if (knoten.links !=**null)**

preOrder(10){
knoten.ausgeben();
if (knoten.links !=**null)**
(… knoten.links.preOrder(9) // dann 7 dann 5;
if (knoten.rechts!= **null)**

preOrder(19) {
knoten.ausgeben();
if (knoten.links !=**null)** …
if (knoten.rechts!= **null) …**

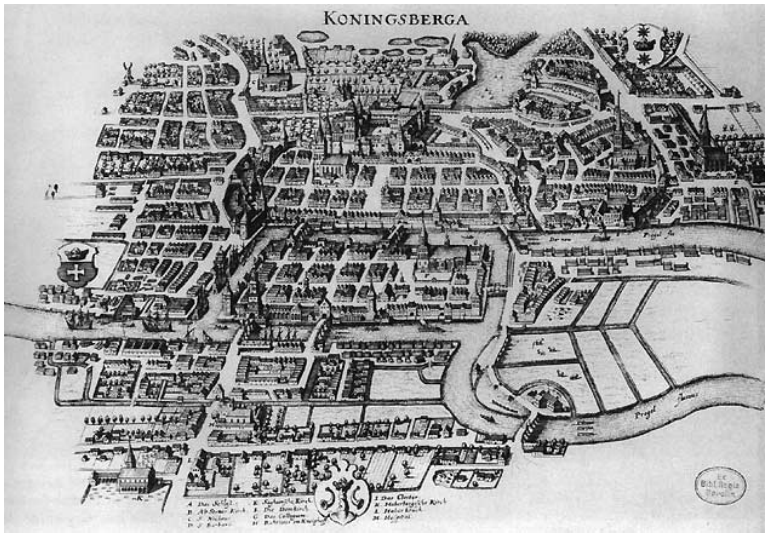
if (knoten.rechts!= **null) …**
}

```
                    if (knoten.rechts  != null) …

                }

 if (knoten.rechts!= null)
        preOrder(68){
        knoten.ausgeben();
        if (knoten.links !=null) …

}
```

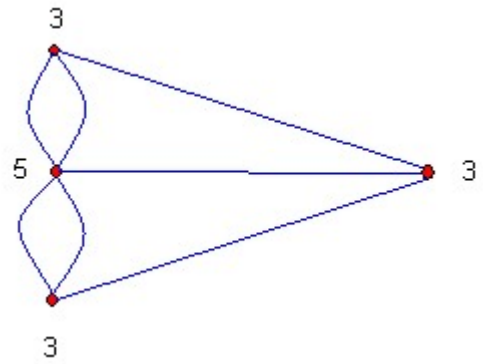
3. Die Datenstruktur Graph

3.1 Einleitung: Das Königsberger Brückenproblem

Das **Königsberger Brückenproblem** ist eine mathematische Fragestellung des frühen 18. Jahrhunderts, die anhand von sieben Brücken der Stadt Königsberg illustriert wurde. Das Problem bestand darin, zu klären, ob es einen Weg gibt, bei dem man alle sieben Brücken über den Pregel genau einmal überquert, und wenn ja, ob auch ein Rundweg möglich ist, bei dem man wieder zum Ausgangspunkt gelangt. Wie Leonhard Euler 1736 bewies, war ein solcher Weg bzw. „Eulerscher Weg“ in Königsberg nicht möglich, da zu allen vier Ufergebieten bzw. Inseln eine ungerade Zahl von Brücken führte. Es dürfte maximal zwei Ufer (Knoten) mit einer ungeraden Zahl von angeschlossenen Brücken (Kanten) geben. Diese zwei Ufer könnten Ausgangs- bzw. Endpunkt sein. Die restlichen Ufer müssten eine gerade Anzahl von Brücken haben, um sie auch wieder verlassen zu können.



Das Brückenproblem ist kein klassisches geometrisches Problem, da es nicht auf die präzise Lage der Brücken ankommt, sondern nur darauf, welche Brücke welche Inseln miteinander verbindet. Es handelt sich deshalb um ein topologisches Problem, das Euler mit Methoden löste, die wir heute der Graphentheorie zurechnen.



3.2 Unterschiede Graph - Baum

Auch ein Graph besteht wie der Baum aus Knoten und Kanten. Allerdings gibt es bei einem Graphen keine Wurzel und die Anzahl der Kanten ist nicht beschränkt. Jeder Knoten kann Kanten zu allen anderen Knoten besitzen, wenn man so will sogar zu sich selbst. Als Kante bezeichnet man wie beim Baum die Verbindung zwischen zwei Knoten.

Der Baum ist also eine spezielle Form eines Graphen. Während wir beim Binärbaum maximal zwei Nachfolger-Knoten hatten, so kann ein Knoten im Graphen Kanten zu allen anderen Knoten besitzen.

Kanten lassen sich sehr leicht durch eine Tabelle darstellen, bei der sowohl in der ersten Zeile als auch in der ersten Spalte alle Knoten aufgelistet sind. Du kennst bestimmt Entfernungstabellen zwischen Städten. Diese sind ja nichts anderes als die Kanten zwischen den Knoten.

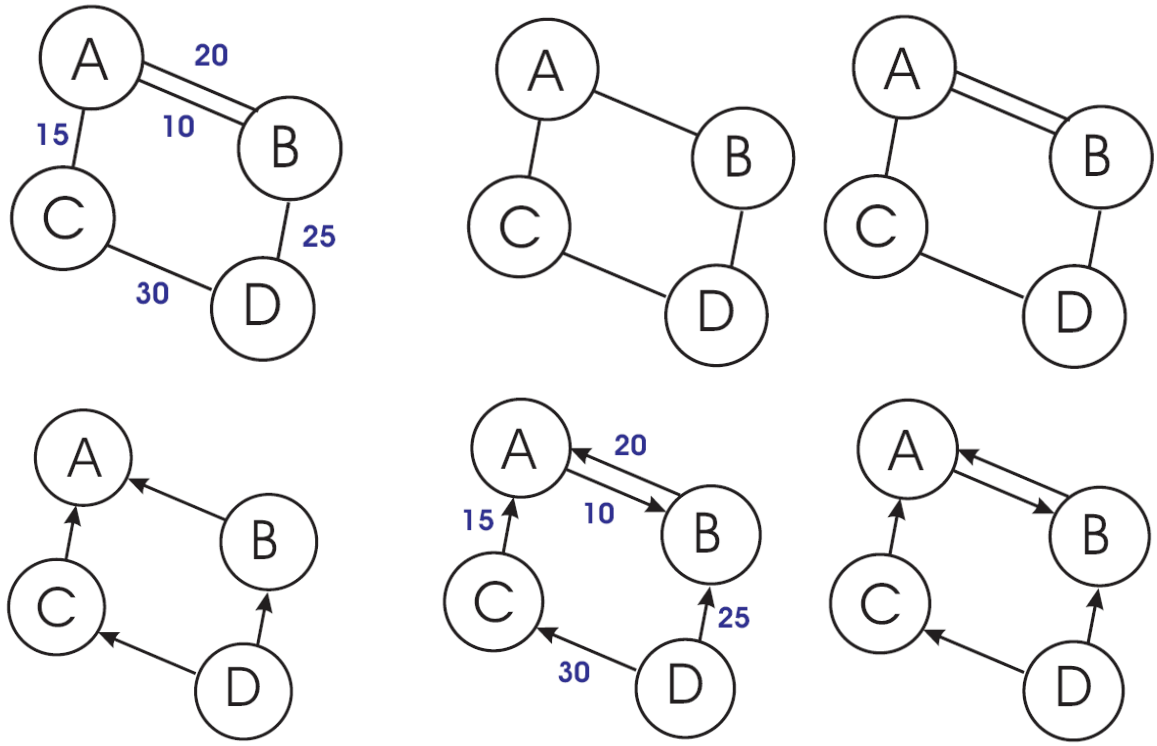
```
   | A   B   C   D   E
---+------------------->
A  | .   2   5   9  14
B  | 2   .   7  15  27
C  | 5   7   .   9  23
D  | 9  15   9   .  12
E  |14  27  23  12   .
   |
   v
```

Adjazenzmatrix

Zwischen den Städten A bis E sind die Entfernungen angegeben. In der Informatik spricht man allerdings nicht von Entfernungstabellen, da nicht zwangsläufig Entfernungen angegeben sind, es kann jedes Kostenmaß Verwendung finden. In der Informatik nennt man eine solche Tabelle eine **Adjazenzmatrix**. In unserer Abbildung bildet die Diagonale eine Symmetrieachse. Das muss aber nicht in jedem Fall so sein. An mehreren Beispielen lässt sich zeigen, dass die Kosten in einer Richtung andere sind als in die andere Richtung sein können:

- Der Transport eines Containers von Shanghai nach Hamburg ist wegen der unterschiedlich hohen Auslastung der Schiffe teurer als von Hamburg nach Shanghai.
- Die Zeit, die man für eine Wanderung zwischen zwei Orten benötigt, ist unterschiedlich lang, wenn es in eine Richtung überwiegend bergauf geht.
- Einbahnstraßen führen zu unterschiedlichen Wegen zwischen zwei Knoten.

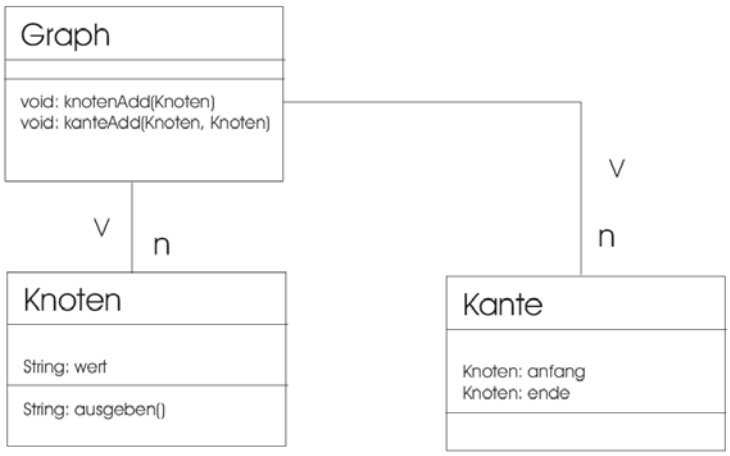
Die Beispiele zeigen, dass es neben ungerichteten Graphen auch gerichtete Graphen geben muss. Daneben kann eine Kante auch mit dem Attribut „Gewicht“ versehen werden, das ein beliebiges Kostenmaß für die Strecke darstellt. Zwischen zwei Knoten kann es auch mehrere Kanten geben. In unseren Beispielen werden wir uns aber darauf beschränken, dass zwischen zwei Knoten in eine Richtung maximal eine Kante verlaufen soll.



Unterschiedliche Graphen, mit und ohne Gewichtung, gerichtet und ungerichtet, mit einer oder mehrerer Kanten zwischen zwei Knoten.

3.3 Modellierung eines Graphen

Die Modellierung eines Graphen erscheint nicht viel anders als die Modellierung eines Baumes. Es gibt aber keine Wurzel und ein Knoten hat keine Nachfolger, sondern eine vergleichsweise unbestimmte Anzahl von Kanten, von der wir nun wissen, dass sie zwischen 0 und der Anzahl der Knoten im Graphen liegt. In einer sehr einfachen Modellierung sieht der Graph dann so aus:



In der Klasse Graph können die Beziehungen zu den Klassen Knoten und Kante durch zwei *ArrayList* realisiert werden. Die Verwaltung der Kanten ist aber ein schwieriges Unterfangen. Daher verwendet man in der Praxis als Datenstruktur eine der beiden Möglichkeiten Adjazenzliste oder Adjazenzmatrix, um die Kanten im Rechner zu repräsentieren.

Bei der Adjazenzliste besitzt jeder Knoten eine Referenz auf eine eigene Liste, in der die Referenzen aller Nachbarn dieses Knotens gespeichert sind. Allerdings ist der Verwaltungsaufwand für diese Datenstruktur nicht so einfach wie bei der Adjazenzmatrix.

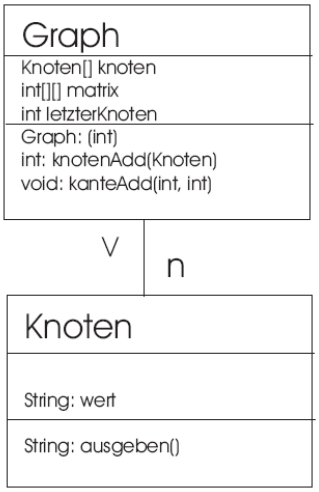
Bei einer Matrix (oder Matrize) handelt es sich um ein zweidimensionales Feld, also eine Nachbarschaftstabelle (oder Entfernungstabelle), wie sie oben schon angesprochen wurde.

In Java lässt sich diese Tabelle leicht als zweidimensionales Array des Datentyps `int` anlegen. Anzumerken ist dabei allerdings, dass dies keine objektorientierte Art der Modellierung darstellt. Die Objekte Kanten repräsentieren nicht mehr eine Klasse Kante. Das Klassendiagramm muss entsprechend angepasst werden.

```
public class Graph
{
  int[][] matrix;
  …

  public Graph(int maxKnoten){
    …
    // zweidimensionales Feld anlegen
    matrix=new int[maxKnoten][maxKnoten];
    for (i=0; i<maxKnoten; i++){
      for (j=0; j<maxKnoten; j++){

         // keine Kante erhält den Wert -1
         if (i!=j) matrix[i][j]=-1;

         // Gewicht für die Kante zu sich selbst ist 0
         else matrix[i][j]=0;

       }
    }
  }

}
```

Nachteilig erweist sich für das Feld, dass schon zu Beginn die maximale Anzahl an Knoten bekannt sein muss.

Die Knoten selbst können als ArrayList oder als Array implementiert werden. Das sollte keine Schwierigkeiten bereiten.



Im Vergleich zum ersten Klassendiagramm hat sich einiges geändert: Die Klasse Kante ist weggefallen, da alle Kanten durch das zweidimensionale Feld `matrix` repräsentiert werden. Die Knoten wurden hier als `Array` vorgegeben und nicht als ArrayList. Der Konstruktor erwartet als Parameter einen int-Wert, mit dem die maximale Anzahl an Knoten vorgegeben wird. Insofern ist es sinnvoll, das Feld für die Knoten genau auf die Länge festzulegen. Kanten werden mit der Methode kanteAdd(int, int) angegeben, die beiden Integer-Werte stehen für den Start und das Ziel der Kante. Gemeint ist jeweils der Index i und j in der Matrix. Da die Stelle eines Knotens kein Attribut des Knotens selbst ist, soll die Methode knotenAdd(Knoten) einen Integer-Wert zurückliefern. Dies übernimmt die Aufgabe einer Referenz, um die Kanten in der Matrize richtig eintragen zu können. Denn wenn Knoten erzeugt werden, dann hält man seine Position in der Matrix als Integer-Wert einer Variable fest. Kanten können dann für diese Knoten leicht eingetragen werden. Das folgende Listing zeigt ein Beispiel, wie in einer Test-Klasse ein Graph aufgebaut wird.

```
public class Test
{
  public los(){
    // der Graph hat maximal 10 Knoten
    Graph graph=new Graph(10);
    // zwei Knoten warden erzeugt
    Knoten k1=new Knoten(“Hamburg”);
    Knoten k2=new Knoten(“Berlin”);

    // die Knoten warden dem Array graph.knoten hinzugefügt
    // in der int-Variable speichern wir die Position des Knotens im Feld
    int hamburg=graph.addKnoten(k1);
    int berlin=graph.addKnoten(k2);

    // in der Adjazenzmatrix wird die Kante eingetragen
    graph.addKante(haburg,berlin);

  }

}
```

Wie bei der Liste ist es aber sinnvoll, nicht direkt auf eine Klasse Knoten zu implementieren, sondern auf eine Schnittstelle und davon ein Datenelement abzuleiten. Später können die Datenelemente leicht ausgetauscht werden, um die Struktur des Graphs auf andere Aufgabenstellungen anzupassen.

Exkurs: Vergleichende Betrachtungen zwischen Adjazenzmatrix und Adjazenzliste

Adjazenzlisten sind zwar aufwändiger zu implementieren und zu verwalten, bieten aber eine Reihe von Vorteilen gegenüber Adjazenzmatrizen. Zum einen verbrauchen sie stets nur linear viel Speicherplatz, was insbesondere bei dünnen Graphen (also Graphen mit wenig Kanten) von Vorteil ist, während die Adjazenzmatrix quadratischen Platzbedarf bezüglich der Anzahl Knoten besitzt (dafür aber kompakter bei dichten Graphen, also Graphen mit vielen Kanten ist). Zum anderen lassen sich viele graphentheoretische Probleme nur mit Adjazenzlisten in linearer Zeit lösen. In der Praxis verwendet man daher meist diese Form der Repräsentation.

http://kik.informatik.fh-dortmund.de/visual/grapheditor.html