STAATSINSTITUT FÜR SCHULQUALITÄT
UND BILDUNGSFORSCHUNG
MÜNCHEN

Informatik am Naturwissenschaftlich-technologischen Gymnasium Jahrgangsstufe 11

München 2009

BRIGG Pädagogik

Erarbeitet im Auftrag des Bayerischen Staatsministeriums für Unterricht und Kultus

Leitung des Arbeitskreises:
Dr. Petra Schwaiger ISB

Mitglieder des Arbeitskreises:
Dr. Andrea Bierschneider-Jacobs Michaeli-Gymnasium München
Ulrich Freiberger Luitpold-Gymnasium München
Albert Wiedemann Erasmus-Grasser-Gymnasium München
Stefan Winter Gymnasium Vilshofen

Externer Berater:
Prof. Dr. Peter Hubwieser TU München

Herausgeber:
Staatsinstitut für Schulqualität und Bildungsforschung

Anschrift:
Staatsinstitut für Schulqualität und Bildungsforschung
Abteilung Gymnasium
Schellingstr. 155
80797 München
Tel.: 089 2170-2304
Fax: 089 2170-2125
Internet: www.isb.bayern.de
E-Mail: petra.schwaiger@isb.bayern.de

Druck und Vertrieb:
Brigg Pädagogik Verlag GmbH
Zusamstraße 5
86165 Augsburg
Tel.: 0821 455494-17
Internet: www.brigg-paedagogik.de
E-Mail: info@brigg-paedagogik.de

Gedruckt auf umweltbewusst gefertigtem, chlorfrei gebleichtem und alterungsbeständigem Papier.

1. Auflage 2009
Nach den seit 2006 amtlich gültigen Regelungen der Rechtschreibung

ISBN 978-3-87101-**585**-4 www.brigg-paedagogik.de

Inhaltsverzeichnis

Vorwort

Die vorliegende Handreichung enthält eine Aufarbeitung von informatischem Fachwissen der Jahrgangsstufe 11, ausführliche didaktisch-methodische Überlegungen zu den einzelnen Lehrplaninhalten, zahlreiche anwendungsorientierte Aufgaben mit Lösungen sowie weiteres umfangreiches digitales Material (verschiedene Versionen der Implementierung der dynamischen Datenstrukturen, Beispielprojekte) auf der beiliegenden CD-ROM.

Aufbauend auf den erworbenen Kenntnissen und Fertigkeiten aus dem Informatikunterricht der Mittelstufe eignen sich die Schülerinnen und Schüler, die in der Qualifikationsphase Informatik belegen, weiterführende Konzepte und Methoden in der Informatik an. Das Fachprofil nennt folgende Zielsetzungen: „Den Schwerpunkt bildet zunächst die Untersuchung dynamischer Informationsstrukturen. Dabei lernen die Jugendlichen das für größere Softwaresysteme unverzichtbare Prinzip der Rekursion kennen. Bei der Planung und arbeitsteiligen Durchführung eines größeren Softwareprojekts wenden sie Konzepte der praktischen Softwareentwicklung an. Hierbei erwerben sie auch auf den Alltag übertragbare Kenntnisse über die Organisation und Synchronisation von Abläufen."

Insgesamt konkretisieren die vorgestellten Konzepte Intention und Anforderungsniveau des Lehrplans, sodass die Auswahl von Aufgaben und Vertiefungsmöglichkeiten aus dem umfangreichen Angebot der Schulbücher unterstützt wird.

An dieser Stelle möchte ich es nicht versäumen, den Mitgliedern und dem Berater des Arbeitskreises für ihr großes Engagement zu danken.

München, August 2009 Petra Schwaiger

Bemerkungen:

- In der Handreichung ist von „Schülerinnen und Schülern" bzw. von „Lehrerinnen und Lehrern" die Rede. In allen anderen Fällen wurde die weibliche Form der Kürze halber mitgedacht.
- Auf der Begleit-CD finden sich die Dateien, auf die im Text verwiesen wird. Diese Dateien enthalten die verschiedenen Versionen der Implementierung der dynamischen Datenstrukturen (Listen, Bäume, Graphen) sowie verschiedene Beispielprojekte zur Softwaretechnik.
- Darüber hinaus befindet sich die elektronische Farbversion des Handreichungstextes auf der Begleit-CD. Zu beachten ist, dass sich Hinweise auf verwendete Farbcodierungen auf die Farbversion der Handreichung bezieht. Entsprechende Textpassagen sind jedoch auch ohne Farbdruck verständlich.
- Die Dateistruktur auf der Begleit-CD korrespondiert im Wesentlichen mit den übergeordneten Bezeichnungen des Inhaltsverzeichnisses der Handreichung.
- Die Handreichung sowie die auf der Begleit-CD enthaltenen Materialien stehen in einer Entwurfsfassung auch auf der Homepage des ISB (ISB-Startseite (www.isb.bayern.de) → Gymnasium → Fach Informatik → Materialien) zur Verfügung.

0 Ausgangslage

> Lehrplan, Jahrgangsstufe 11 (Lp): In den Jahrgangsstufen 9 und 10 haben sich die Schüler insbesondere mit der Analyse sowie mit der für eine maschinelle Verarbeitung geeigneten Darstellung von Information beschäftigt und dabei verschiedene Techniken der Modellierung angewandt. Darauf aufbauend lernen sie in Jahrgangsstufe 11 neue Konzepte kennen, die es ihnen erlauben, größere Systeme effizienter zu modellieren. Sie greifen nun auf rekursive Datenstrukturen wie Listen oder Bäume zurück und erkennen deren Nutzen als häufig verwendbare Modellierungsmuster.

Vor dem Einstieg in die Themen der Jahrgangsstufe 11 sollten an ausgewählten Beispielen Inhalte der Informatik aus den vorherigen Jahrgangsstufen wiederholt werden, wobei die objektorientierte Modellierung den Schwerpunkt bilden sollte. Entsprechend dem Alter der Schülerinnen und Schüler kann dabei deutlich theoretischer vorgegangen werden als in den vorherigen Jahrgangsstufen.

Die objektorientierte Sichtweise spielt sowohl bei der Analyse der Aufgabenstellung (Modellierung) als auch beim Softwaredesign (Implementierung des Modells) die tragende Rolle.

Begriffe wie Klasse, Objekt, Attribut, Methode und Beziehung zwischen Klassen müssen aus früheren Jahrgangsstufen beherrscht werden. Die Kenntnis einer graphischen Notation ist ebenso erforderlich wie die Umsetzung in eine objektorientierte Programmiersprache. Hierbei sind folgende Aspekte relevant: Deklaration einer Klasse mit Attributen und Methoden, Konstruktor als spezielle Methode, Datentypen der Attribute, Realisierung der Methoden mit Algorithmen in der Programmiersprache, Referenzattribute zur Umsetzung der Beziehungen, Vererbungsprinzip zur Realisierung von Klassenhierarchien und abstrakte Klassen.

In Jahrgangsstufe 10 haben die Schülerinnen und Schüler die Grundprinzipien des objektorientierten Softwareentwurfs kennengelernt. Dabei muss der Entwickler zusätzliche Festlegungen treffen, die sich nicht direkt aus der Modellierung ableiten lassen. Einige Gesichtspunkte sind:

- Bildung von Klassen: Hier muss der Entwickler die Daten und die darauf erforderlichen Operationen in Klassen abbilden. Dabei ist in erster Linie festzulegen, wie detailliert die Aufteilung der Daten/Methoden auf die verschiedenen Klassen erfolgen soll.
- Gegebenenfalls Festlegung einer Vererbungshierarchie zwischen den gebildeten Klassen: Der Entwickler hat zu entscheiden, welche Klassen sinnvollerweise voneinander erben sollen. Als oberste Klasse in einer Hierarchie wird dabei oft eine abstrakte Klasse gewählt.
- Neben Vererbungsbeziehungen („ist ein“- Beziehung) zwischen den Klassen gibt es natürlich auch Beziehungen zwischen den Objekten der Klassen. Ein Objekt kann beispielsweise ein anderes Objekt erzeugen oder eine Referenz auf das andere Objekt verwalten. Diese Beziehungen sind ebenso wichtig wie die Vererbungsbeziehungen, weil erst durch sie ein guter objektorientierter Softwareentwurf mit flexibler Struktur entsteht. Der Entwickler muss entscheiden, welche Referenzattribute hierzu in den einzelnen Klassen erforderlich sind.
- Im Rahmen der Schnittstellengestaltung muss festgelegt werden, welche Methoden und welche Daten ein Objekt anderen Objekten zur Verfügung stellen soll.

Bei ihrer Arbeit erkennen die Schülerinnen und Schüler, dass verschiedene Softwarestücke ähnliche (oder sogar gleiche) Strukturen enthalten. In der Informatik wurden für bestimmte, immer wieder vorkommende Muster bei Abläufen und Datenstrukturen allgemeine Vorgehensweisen entwickelt. Softwareentwurfsmuster beschreiben dies in abstrakter, von der speziellen Aufgabenstellung unabhängigen Form. Analog wurden in der Vergangenheit bewährte Algorithmen katalogisiert, veröffentlicht und standardisiert, beispielsweise verschiedene Sortieralgorithmen. Bei den Datenstrukturen sind für vergleichbare Situationen

sogenannte abstrakte Datentypen (ADT) entwickelt worden, die von der konkreten Aufgabenstellung unabhängig formuliert sind und für die eine standardisierte Implementierung vorliegt. Sie sind universell einsetzbar und lassen sich leicht an die spezielle Modellsituation anpassen. Abstrakte Datentypen bilden nur eine Spezifikation der Schnittstelle nach außen, indem sie Operationen und deren Funktionalität festlegen. Bei der Realisierung eines ADT ist das Prinzip der Kapselung (der ADT darf nur über die Schnittstelle benutzt werden) und das Geheimnisprinzip (die interne Realisierung des ADT bleibt verborgen) zu beachten. Das Konzept der ADT lässt sich mit den Möglichkeiten der objektorientierten Modellierung leicht umsetzen.

Einfache Softwareentwurfsmuster haben die Schülerinnen und Schüler schon in der Jahrgangsstufe 10 angewandt und dabei eine Strukturierung von Daten durch Reihungen (Feld, Array) vorgenommen. Voraussetzung für diese Jahrgangsstufe ist ein stabiles Verständnis für eine Reihung von Referenzen und deren Umsetzung in einer Programmiersprache.

In dieser Jahrgangsstufe werden weitere Softwaremuster zur Strukturierung von Daten (sog. Datenstrukturen) hinzukommen. Die Jugendlichen werden mit abstrakten Datentypen wie Listen (mit den Spezialformen Stapel und Schlange), Bäumen und Graphen arbeiten. Hauptziel ist es, eine möglichst effiziente, jedoch allgemein verwendbare Datenstruktur für die jeweilige Aufgabensituation einzusetzen.

1 Rekursive Datenstrukturen

Lp: In Jahrgangsstufe 10 haben die Schüler im Rahmen der objektorientierten Modellierung schwerpunktmäßig am Erstellen geeigneter Methoden von Objekten sowie am Erkennen und Umsetzen von Objektbeziehungen gearbeitet. Verschiedenartige Aufgabenstellungen aus der Praxis verdeutlichen den Schülern nun, dass immer wieder Strukturen von Daten auftreten, die mit den bisher bekannten Datentypen nicht effizient beschrieben werden können. Die Schüler beschäftigen sich daher eingehend mit typischen rekursiven Datentypen und lernen deren Vorteile und breites Spektrum von Einsatzmöglichkeiten kennen; dabei verwenden sie erstmals rekursive Methoden.

1.1 Listen (ca. 29 Std.)

Lp: Die Schüler untersuchen die grundlegenden Eigenschaften der Datenstruktur Schlange, deren grundsätzlichen Aufbau sie bereits aus ihrem Alltag, z. B. von Warteschlangen, kennen. Eine erste Implementierung mit einem Feld zeigt schnell die Grenzen dieser statischen Lösung auf und führt die Jugendlichen zu einer dynamischen Datenstruktur wie der einfach verketteten Liste. Sie erarbeiten deren prinzipielle Funktionsweise sowie deren rekursiven Aufbau und wenden hierbei das Prinzip der Referenz auf Objekte an. Die Jugendlichen erkennen, dass die rekursive Struktur der Liste für viele ihrer Methoden einen rekursiven Algorithmus nahelegt. Sie verstehen, dass eine universelle Verwendbarkeit der Klasse Liste nur möglich ist, wenn auf eine klare Trennung von Struktur und Daten geachtet wird. An einfachen Beispielen aus der Praxis und deren Implementierung vertiefen die Schüler ihr Wissen und erfahren die flexible Verwendbarkeit dieses Datentyps.

1.1.1 Warteschlange

Am Beispiel der Warteschlange sollen die Schülerinnen und Schüler erkennen, dass es sich lohnt, beim Entwickeln einer Software gründlicher über eine effektive Umsetzung des Modells nachzudenken. Erste einfache Implementierungen mit Feldern werden sehr schnell als ungeeignet eingestuft. Am Ende kristallisiert sich ein allgemein verwendbares Softwaremuster (eine Datenstruktur) für eine Warteschlange heraus. Zunächst wird man die Warteschlange unter Nutzung von einfachen Feldern umsetzen. Da diese Implementierung für den praktischen Einsatz nur bedingt geeignet ist, wird mit den verketteten Listen eine Datenstruktur erarbeitet, die sehr häufig verwendet wird und auch zur Implementierung von Warteschlangen genutzt werden kann.

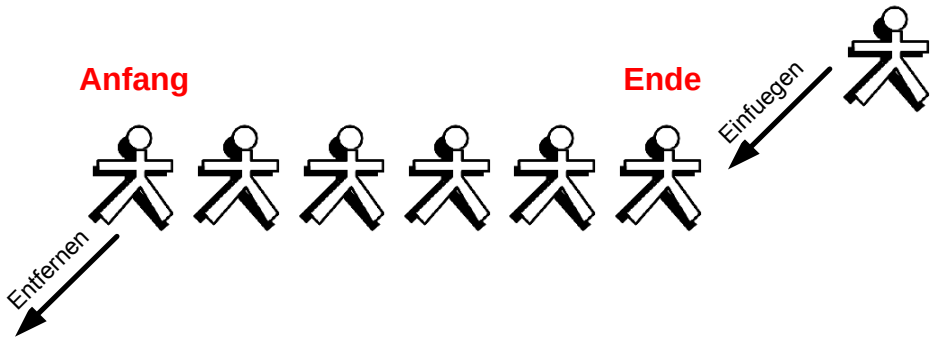
Ausgangspunkt sind Beispiele aus dem Alltag, anhand derer die grundsätzlichen Anforderungen an eine Warteschlange erarbeitet werden. Typisch sind Warteschlangen in Dienstleistungsbetrieben und beim sequenziellen Abarbeiten von Arbeitsvorgängen. Zum Beispiel: Warteschlangen von Kunden beim Postamt oder in der Bankfiliale sowie an den Kassen von Supermärkten; sequenzielle Bearbeitung von Anträgen in einer Behörde; Abarbeitung von Druckaufträgen an einem Drucker; Reservierung von Eintrittskarten (Konzert, Theater, Fußballspiel) nach der Reihenfolge der eingegangenen Bestellungen; Abarbeiten von Anrufen in einem Callcenter oder an einer Hotline; ordentliches Anstehen am Pausenverkauf einer Schule. Generell ist eine Warteschlange erforderlich, wenn zwei Prozesse an einem Ablauf beteiligt sind, die unterschiedliche Verarbeitungsgeschwindigkeit aufweisen.

1.1.1.1 Grundprinzip

Lp: Methoden der Datenstruktur Schlange: Anfügen am Ende, Entfernen am Anfang

Lp: Einsatz der allgemeinen Datenstruktur Liste bei der Bearbeitung eines Beispiels aus der Praxis: Verwaltung von Elementen verschiedener Datentypen mittels Vererbung

Kennzeichnend für die Verwaltung der Objekte in einer (Warteschlange, engl. queue) ist, dass die gespeicherten Objekte nur in der gleichen Reihenfolge wieder ausgegeben werden, wie sie eingetragen wurden (FIFO-Prinzip, „first in first out“). Neue Objekte werden hinter dem Ende der Schlange eingefügt; in der Schlange gespeicherte Objekte werden vom Anfang abgerufen und dabei aus der Schlange entfernt.



I) Allgemeine Forderungen an die Objekte, die von der Klasse SCHLANGE verwaltet werden:

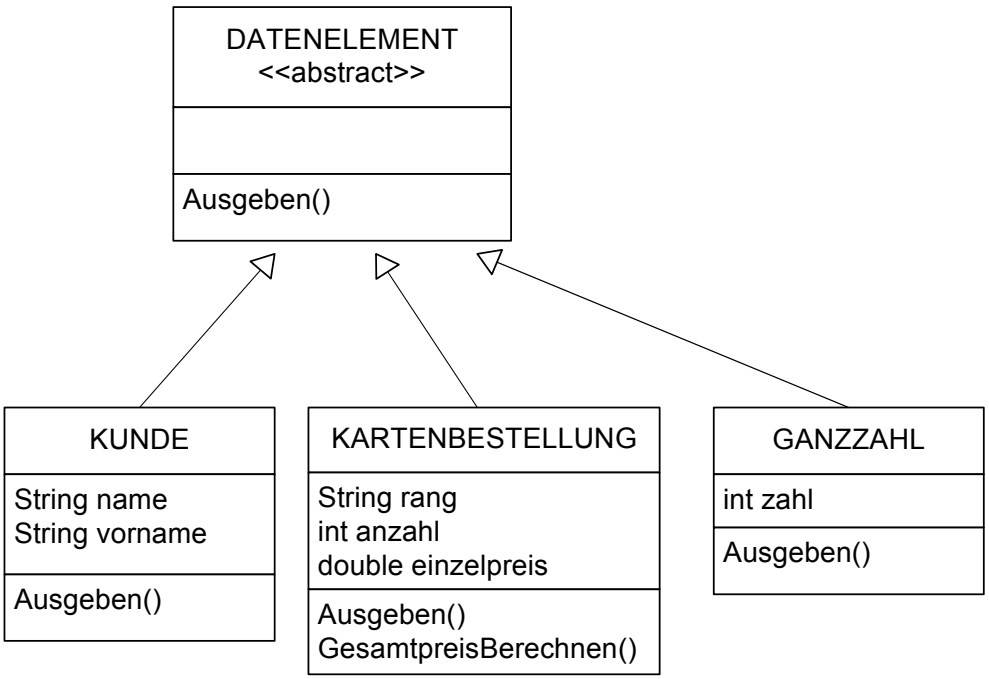
Die Schlange braucht gewisse Kenntnisse über die Objekte, die sie aufnehmen bzw. abgeben soll. Die Schülerinnen und Schüler wissen vom Informatikunterricht des Vorjahrs, dass ein Parameter einer Methode und die interne Speicherung (z. B. in Attributen) je nach Aufgabenstellung die Angabe eines speziellen Datentyps erfordern. Man müsste somit eine Schlange für die Verwaltung von Ganzzahlen, eine andere für Kundenobjekte, eine weitere für Kartenbestellungsobjekte usw. entwickeln. Von einer allgemeinen einsetzbaren Lösung wäre man weit entfernt. Es ist deshalb auf eine klare Trennung zwischen der Struktur an sich (Schlange) und den Daten, die in dieser Struktur verwaltet werden, zu achten.

Deshalb soll die Klasse SCHLANGE so festlegt werden, dass sie beliebige Arten von Objekten verwalten kann, wobei alle Objekte, die in einer Schlange verwaltet werden, von derselben Klasse sein sollen. Dabei setzt man sich als Ziel, dass die konkrete Ausprägung der verwalteten Objekte vor der Schlange verborgen bleiben und für die Schlange alle Objekte immer gleich aussehen. Aus Jahrgangsstufe 10 kennen die Schülerinnen und Schüler ein hierzu einsetzbares Konzept, die Vererbung.

In den folgenden Implementierungen gehen wir davon aus, dass die zu verwaltenden Objekte von einer Klasse stammen, die eine Unterklasse der Klasse DATENELEMENT ist. Für die Schlange schauen alle verwalteten Objekte so aus, als ob sie zur Klasse DATENELEMENT gehören. Da die Warteschlange keine Ahnung vom Aufbau der Unterklassen von DATENELEMENT hat, sind die Objekte der Unterklassen selbst verantwortlich für eine Ausgabe ihrer Daten, d. h., sie benötigen eine Methode *Ausgeben*().

Hinweis: Es wird nicht empfohlen, für die verwalteten Objekte eine Unterklasse der Superklasse `Object` zu verwenden, denn dadurch kann nicht garantiert werden, dass diese Objekte über Methoden verfügen, die die Klasse SCHLANGE erwartet.

Die Klasse DATENELEMENT gibt die wesentlichen Fähigkeiten vor, in den Unterklassen werden diese problemangepasst „mit Leben“ gefüllt. Da es keine Objekte der Klasse DATENELEMENT geben soll, sondern nur von den zugehörigen Unterklassen, wird die Klasse DATENELEMENT als abstrakt deklariert. In Java könnte die Klasse DATENELEMENT auch als Interface realisiert werden.



Die Schlange verwaltet Referenzen auf die Objekte der Klasse DATENELEMENT, genauer sogar nur Referenzen auf Objekte aus Unterklassen von DATENELEMENT.

II) Erforderliche Attribute der Klasse SCHLANGE:

- Referenzen auf die Objekte in der Schlange
  Hier werden die Referenzen auf die Objekte gespeichert. In unserer Implementation sind es Referenzen auf Objekte der Klasse DATENELEMENT.
- Anzahl der Objekte in der Schlange
  Dieses Attribut ist nicht zwingend erforderlich, aber für die ersten Implementierungen mit Feldern hilfreich.

III) Erforderliche Methoden der Klasse SCHLANGE:

- SCHLANGE()
  Anlegen eines Schlange-Objekts (Konstruktor)
- Einfuegen(Referenz auf neues Objekt)
  Die Referenz auf das neue Objekt wird am Ende der Schlange eingefügt.
  „Das angegebene Objekt hinten in die Schlange einfügen!“
- Entfernen() -> Referenz auf vorderstes Objekt
  Das vorderste Objekt in der Schlange wird aus der Schlange entfernt; die Funktion gibt eine Referenz auf dieses Objekt zurück.
  „Das vorderste Objekt aus der Schlange entfernen!“

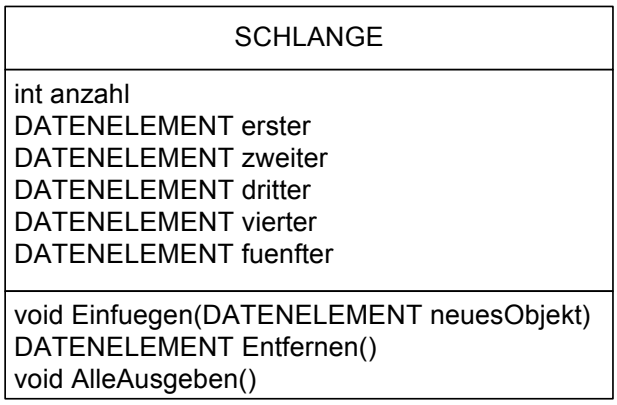
IV) Weitere nützliche Methoden der Klasse SCHLANGE:

- AnfangGeben() -> Referenz auf vorderstes Objekt
  Die Funktion liefert eine Referenz auf das vorderste Objekt (am Anfang der Schlange), ohne es zu entfernen.
  „Wer steht vorne?“
- AlleAusgeben()
  Diese Methode gibt die Inhalte aller in der Schlange verwalteten Objekte aus. Sie ruft hierzu die Methode Ausgeben() der Datenelemente auf, denn die Schlange selbst kennt die Struktur der Objekte nicht.
- AnzahlGeben() -> Ganzzahl
  Ausgegeben wird die Anzahl der Objekte, die aktuell in der Schlange sind.
- IstLeer() -> true/false
  Geprüft wird, ob die Schlange leer ist.

Für die Implementierung der Schlange ist eine geeignete Repräsentation der verwalteten Objekte zu finden. In der Diskussion mit den Schülerinnen und Schülern wird die Entscheidung getroffen, wie die Referenzen auf die zu verwaltenden Objekte in der Schlange hinterlegt werden.

1.1.1.2 Implementierung mit festen Referenzen

Eine erste Realisierung, deren Unbrauchbarkeit für den Jugendlichen sofort einsichtig ist, könnte wie folgt aussehen:



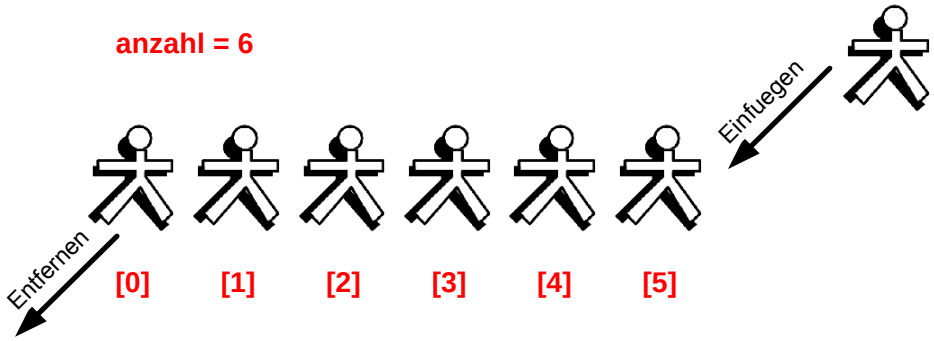
Die Schülerinnen und Schüler versuchen, die Funktionsweise der Methoden in allgemeinen Worten zu beschreiben. Eine Implementierung dieser Lösung ist nicht vorgesehen. Die

Diskussion über die Qualität dieser Lösung geht am besten von der Überlegung aus, dass ein Objekt der Klasse DATENELEMENT erzeugt und dann mit einer Referenz auf dieses Objekt die Methode *Einfuegen* aufgerufen wird. Es stellen sich Fragen der Art:

- Woher weiß man, an welcher Position in der Datenstruktur das neue Objekt eingefügt werden soll?
- Wie schaut diese Abfrage umgangssprachlich formuliert aus?
- Wie kann man verhindern, dass zu viele Referenzen eingefügt werden?
- Nach drei hinzugefügten Objekten soll das erste aus der Schlange entfernt werden. Welches Objekt wird danach das erste sein?
- Wie lässt sich ein Algorithmus fürs Nachrücken umgangssprachlich formulieren?
- Was sind weitere Grenzen dieser Art der Implementierung?

1.1.1.3 Implementierung mit einem Feld

In Jahrgangsstufe 10 haben die Schülerinnen und Schüler die Datenstruktur Feld kennengelernt, die zunächst für eine bessere Implementierung gewählt wird. Man kann sich die in der Schlange verwalteten Objekte wie die Komponenten eines Feldes vorstellen, die von 0 aus durchnummeriert sind. Über diesen Index kann jedes Feldelement einzeln und direkt angesprochen werden.

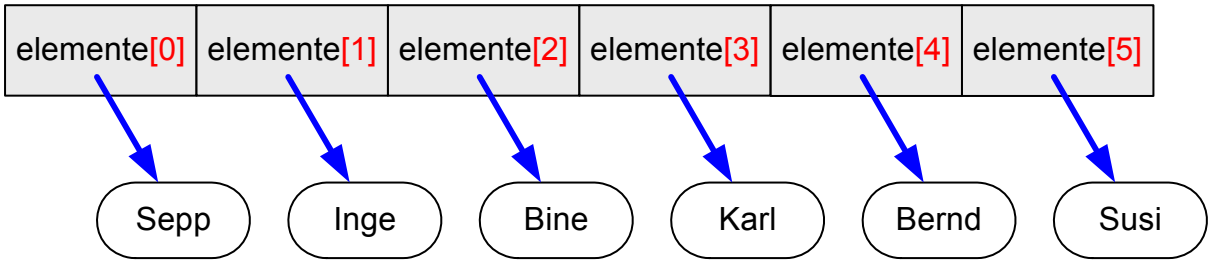


Nur einfache Datentypen (z. B. int, double, boolean) können in einem Feld direkt verwaltet werden. Komplexere Datentypen wie Objekte einer Klasse werden in Form eines Feldes mit Referenzen auf diese Objekte geführt.

Da Objekte der Klasse DATENELEMENT mit der Schlange verwaltet werden sollen, ist ein Feld vom Typ DATENELEMENT[] zu verwenden (im Folgenden *elemente* genannt). Dies erfordert, dass erstens das Feld mit einer festen Größe initialisiert und dass zweites ein ‚Merker' vorhanden ist, der die Position der nächsten freien Einfügestelle kennt. Im folgenden Beispiel wird dies durch das Attribut *anzahl* realisiert, das die Anzahl der Elemente in der Schlange beinhaltet. Der Wert von *anzahl* beschreibt damit die nächste freie Position im Feld *elemente*. Das Element *elemente[0]* ist das vorderste Element der Schlange. Ist die Schlange voll, so werden keine weiteren Objekte aufgenommen.

SCHLANGE
int anzahl DATENELEMENT[ ] elemente
void Einfuegen(DATENELEMENT neuesObjekt) DATENELEMENT Entfernen() void AlleAusgeben() boolean IstLeer()

Die Situation einer Warteschlange für 6 Personen stellt sich in dieser Implementierung wie folgt dar. Es kommt ein Feld *elemente* mit 6 Feldkomponenten zum Einsatz. Jede Feldkomponente beinhaltet eine Referenz auf ein Objekt.



Zur Wiederholung der Inhalte der Jahrgangsstufe 10 sollten die Schülerinnen und Schüler diese Form der Implementierung mit einer Programmiersprache realisieren.

Ein Auszug in Java:

```
abstract class DATENELEMENT
{
   abstract void Ausgeben();
}
```

```
class SCHLANGE
{
   private int anzahl;
   private DATENELEMENT[ ] elemente;

   SCHLANGE()
   {
      // maximal 100 Elemente möglich
      elemente = new DATENELEMENT[100];
      anzahl = 0;
   }

   void Einfuegen(DATENELEMENT neuesObjekt)
   {
      if (anzahl < elemente.length)
      {
         elemente[anzahl] = neuesObjekt;
         anzahl = anzahl + 1;
      }
   }

   DATENELEMENT Entfernen()
   {
      DATENELEMENT ergebnis = null;

      if (anzahl > 0)
      {
         ergebnis = elemente[0];
         anzahl = anzahl - 1;
         for (int i = 0; i < anzahl; i++)
            elemente[i] = elemente[i+1];
         elemente[anzahl] = null;
      }
      return ergebnis;
   }
```

```
  void AlleAusgeben()
  {
    for (int i = 0; i < anzahl; i++)
      elemente[i].Ausgeben();
  }

  boolean IstLeer()
  {
    return anzahl == 0;
  }
}
```

Hinweis: Im Hinblick auf ein späteres rekursives Vorgehen sollten die Jugendlichen darauf hingewiesen werden, dass hier die Schlange die Ausgabe iterativ durchführt, d. h., die Schlange ruft jedes Element der Reihe nach auf und bittet es, sich auszugeben.

Diese Implementierung einer allgemeinen Datenstruktur Schlange kann man etwa am Beispiel der Kundenwarteschlange testen.

Definition der Klasse KUNDE:

```
class KUNDE extends DATENELEMENT
{
  private String name;
  private String vorname;

  KUNDE(String neuerName, String neuerVorname)
  {
    name = neuerName;
    vorname = neuerVorname;
  }

  void Ausgeben()
  {
    System.out.println("Kunde: " + name + ", " + vorname);
  }
}
```

Die Jugendlichen testen in einem kleinen Programm die Richtigkeit der Implementierung und machen sich mit dem Verhalten der Schlange vertraut. Ein Ausschnitt aus einem Programm zur Verwendung der Schlange könnte lauten:

```
SCHLANGE warteSchlange = new SCHLANGE();

warteSchlange.Einfuegen(new KUNDE("Huber", "Eduard"));
warteSchlange.Einfuegen(new KUNDE("Meier", "Paul"));
warteSchlange.Einfuegen(new KUNDE("Müller", "Inge"));
...
warteSchlange.AlleAusgeben();
warteSchlange.Entfernen();
warteSchlange.AlleAusgeben();
...
```

In diesem Programmausschnitt wurde der Rückgabewert (Referenz auf das entfernte Objekt) der Methode *Entfernen()* nicht verwendet.

Nachdenken sollten die Schülerinnen und Schüler über folgenden Codeausschnitt:

```
KUNDE k =(KUNDE) warteSchlange.Entfernen();
k.Ausgeben();
```

Die Methode *Entfernen()* liefert eine Referenz auf ein „allgemeines“ Datenelement zurück. Zu welcher Unterklasse von DATENELEMENT das Objekt wirklich gehört, ist nicht ersichtlich. Das ist Absicht, denn die Methode *Entfernen()* soll für alle Unterklassen von DATENELEMENT verwendbar sein (Polymorphismus). Durch eine so genannte Typkonvertierung (hier „(KUNDE)“) ist in derartigen Fällen anzugeben, wie die Referenz zu interpretieren ist (im Beispiel als Referenz auf ein Kundenobjekt).[1]

1.1.1.4 Grenzen der Implementierung mit einem Feld

Ein wesentliches Problem der hier vorgestellten Implementierung der Datenstruktur Schlange ist die Größenbeschränkung des Feldes. Es können nur solange Elemente eingefügt werden, bis die Kapazität des Feldes erschöpft ist. Ein wenig sinnvoller Ausweg wäre das Erzeugen eines neuen, größeren Feldes und das Umkopieren der Elemente aus dem alten Feld.

Die Vorgabe, dass sich das vorderste Objekt immer auf der Position 0 befindet, hat zur Folge, dass beim Entfernen eines Elements alle verbleibenden Elemente um eine Position nach vorne verschoben werden müssen. Dies ist ziemlich zeitaufwendig.

Das „Nachvorneverschieben“ könnte man sich mit einer anderen Variante ersparen: Das erste Element ist jetzt nicht immer zwangsweise an der 0-ten Position. Ein neues Attribut *vorne* merkt sich die Position des ersten Elements innerhalb des Felds und die Elemente bleiben immer an der eingefügten Stelle im Array. Beim Einfügen ergibt sich die nächste freie Position aus *anzahl* + *vorne*. In diesem Fall entfällt das Verschieben der Elemente. Ein Auslasten aller Feldkomponenten, auch derer vor der Position *vorne*, erfordert einen nicht trivialen Algorithmus. Interessierte Schülerinnen und Schüler können sich an eine Implementierung dieser Variante wagen.

Aufgrund der Größenbeschränkung ist aber auch die zweite Variante kein großer Fortschritt. Diese Probleme lassen sich nur durch dynamische Datenstrukturen umgehen. Dynamisch bedeutet dabei, dass sie zur Laufzeit ‚wachsen’ oder ‚schrumpfen’ können und sich dadurch an den tatsächlichen Speicherbedarf anpassen.

1.1.2 Grundprinzip der Liste und rekursive Abläufe

Lp: allgemeines Prinzip und rekursive Struktur einer einfach verketteten Liste;

Lp: rekursive Abläufe: rekursiver Methodenaufruf, Abbruchbedingung, Aufrufsequenz

1.1.2.1 Grundprinzip der verketteten Liste

Die Logik einer Warteschlange muss hierzu näher analysiert werden. Am besten ist dies an der Situation einer „Kundenwarteschlange an einer Kasse“ möglich: Jeder Kunde hat einen Nachfolger („Hintermann“ bzw. „Hinterfrau“), einzig der Kunde am Ende der Schlange hat keinen Nachfolger. Mit seinem Hintermann kann ein Kunde in Kontakt treten, indem er sich umdreht und dann den Nachfolger anspricht – er schickt dem Hintermann eine Botschaft. Auf diese Botschaft kann der Hintermann antworten oder nicht, je nach Botschaft. Für die Kontaktmöglichkeit benötigt der Kunde eine Referenz auf seinen Hintermann.

[1] Prinzipiell wäre es auch möglich, k als Variable vom Typ DATENELEMENT zu deklarieren und auf den Typecast zu verzichten oder den direkten Methodenaufruf `warteSchlange.Entfernen().Ausgeben()` anzustoßen, da die Methode Ausgeben() bereits in DATENELEMENT deklariert ist (Polymorphismus). Aus didaktischen Gründen ist es aber sinnvoll, hier konkrete Objekte der Klasse KUNDE zu betrachten.

Die Kunden sind über die Referenz „verkettet“. Man spricht in der Informatik von einer einfach verketteten Liste von Objekten. „Einfach“ drückt aus, dass es nur eine Referenz von einem Objekt zum anderen gibt (in einer Richtung). Doppelt verkettete Listen werden im Lehrplan nicht erwähnt. Ein Objekt ist der Anfang der Liste (in der Abbildung der Kunde ganz links), ein anderes das Ende (in der Abbildung der Kunde ganz rechts).

1.1.2.2 Rekursive Abläufe

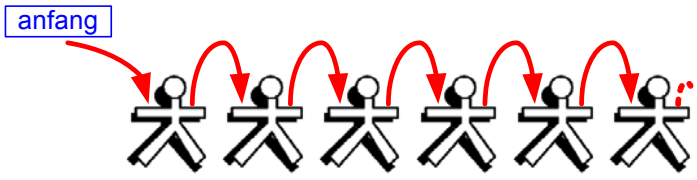
Für die Organisation der Warteschlange ist der Herr[2] an der Kasse zuständig. Er möchte mit allen Kunden in der Warteschlange in Kontakt treten, denn er will ihnen zur Beruhigung mitteilen, dass es nach dem technischen Ausfall der Kasse bald weitergeht. Dazu benötigt er eine Methode *AlleBenachrichtigen(Nachricht)*. Er hat zwei Möglichkeiten, dies zu bewerkstelligen.

Er könnte die ganze Schlange entlanglaufen und jeder Person in der Schlange die Nachricht schicken: „Bald geht es weiter, der Schaden ist behoben“. Diese Arbeitsweise nennt man iterativ (lat. iterare „wiederholen“).

Alternativ könnte er sich dazu die innere Struktur der Warteschlange („jeder kennt seinen Hintermann“) zunutze machen. Er schickt dem Ersten in der Schlange die Botschaft „Bitte weitersagen: ‚Bald geht es weiter, der Schaden ist behoben’“. Dieser reagiert auf die Botschaft mit seiner Methode *Weitersagen(Nachricht)*. Alle Elemente der Warteschlange sind von derselben Sorte (derselben Klasse) und haben alle die Methode *Weitersagen(Nachricht)*. Wie läuft diese Methode ab? Der Empfänger nimmt die Nachricht ‚Bald geht es weiter, der Schaden ist behoben’ zur Kenntnis und schickt dann die Botschaft „Bitte weitersagen: ‚Bald geht es weiter, der Schaden ist behoben’“ an seinen Hintermann. Dies wird solange durchgeführt, bis der Letzte in der Schlange erreicht ist.

Jede Methode *Weitersagen(Nachricht)* ruft die Methode *Weitersagen(Nachricht)* des Nachfolgers auf, um seine Arbeit zu erledigen (Aufrufsequenz). Diese Art des Aufrufs bezeichnet man als rekursiven Aufruf (rekursiver Methodenaufruf). Der rekursive Aufruf wird beendet, wenn die Methode beim Letzten in der Schlange aufgerufen wird (Abbruchbedingung).

Durch eine rekursive Arbeitsweise hat der Kassenangestellte seine Nachricht an den Mann gebracht. Er muss lediglich wissen, wer der Erste in der Schlange ist. Dazu benötigt er lediglich eine Referenz *anfang* („Wen muss ich ansprechen?“) auf das erste Objekt in der Warteschlange, d. h. auf den Anfang der einfach verketteten Liste.



[2] Zu beachten ist hier – und an vielen weiteren Stellen – insbesondere die Anmerkung nach dem Vorwort „In der Handreichung ist von ‚Schülerinnen und Schülern‘ und ‚Lehrerinnen und Lehrern‘ die Rede. In allen anderen Fällen wurde die weibliche Form der Kürze halber mitgedacht.“

Der Herr an der Kasse (als Verwalter der Warteschlange) möchte gerne wissen, wie viele Personen sich in der Warteschlange befinden. Da die Kunden ordentlich in der Reihe stehen und die Schlange schon reichlich lang geworden ist, kann er die Schlange nicht im Ganzen sehen, nur auf den vordersten Kunden (am Anfang der Schlange) hat er Zugriff. Er fragt deshalb den vordersten Kunden: „Wie lange ist die Schlange?“ Der erste Kunde kann dies nicht beantworten und fragt deshalb seinen Hintermann: „Können Sie mir bitte sagen, wie lange ist die Schlange hinter Ihnen, Sie eingeschlossen?“ Falls der Hintermann diese Frage auch nicht beantworten kann, wird er sie seinem Hintermann stellen und so weiter. Der Erste, der die Frage direkt beantworten kann, ist der letzte Kunde in der Schlange. Da keiner hinter ihm steht, beantwortet er die Frage mit „ein Kunde“, d. h., er meldet seinem Vordermann die Zahl 1. Der Vordermann (der vorletzte Kunde in der Schlange) kann die Frage jetzt auch beantworten, denn er braucht nur zum rückgemeldeten Wert seines Hintermanns eins dazu zu addieren. Das wird fortgeführt, bis die schrittweise Rückmeldung beim Ersten angekommen ist. Dieser meldet dem Kassenangestellten die Länge der Schlange, d. h. die Anzahl der Kunden, die in der Schlange stehen.

Bei der Feststellung der Schlangenlänge handelt sich um einen rekursiven Ablauf. Rekursion bedeutet die Anwendung desselben Prinzips auf in gewisser Weise „einfachere“ Situationen, bis die Situation so einfach ist, dass sie direkt gelöst werden kann (Abbruchbedingung). Wenn die einfachste Situation gelöst werden kann, dann können auch die vorherigen Situationen fertig bearbeitet werden, die Aktion läuft wieder zurück. (lat. recurrere „zurücklaufen“).

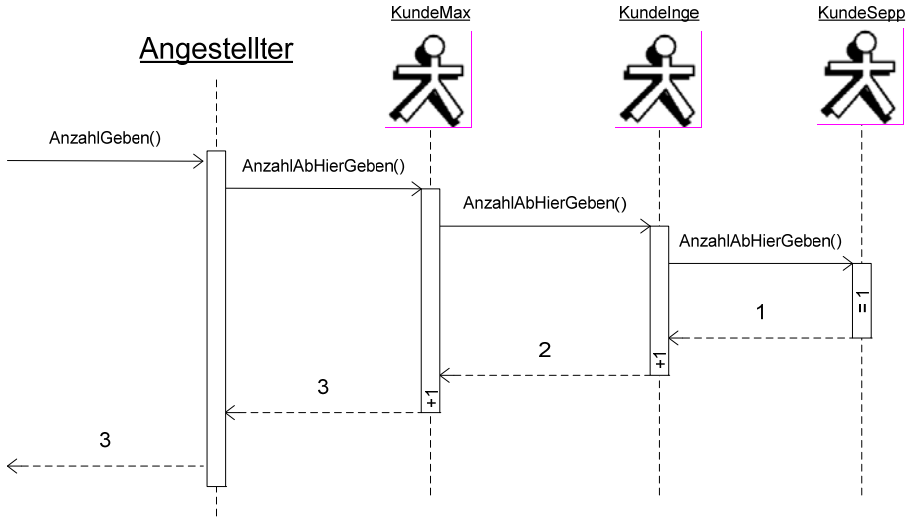
Diesen Ablauf sollten die Schülerinnen und Schüler direkt nachspielen, um ein Gespür für rekursive Vorgänge (rekursive Methodenaufrufe) zu bekommen.

Jetzt wird dieser Vorgang unter dem Gesichtspunkt der Informatik betrachtet. Für das Objekt Warteschlange ist eine Methode *AnzahlGeben()* zu realisieren. Jedes Objekt in der Schlange hat eine Methode *AnzahlAbHierGeben()*, die als Rückgabewert die Länge der Schlange ab der eigenen Position liefert (einschließlich des Objekts). Die Warteschlange schickt an das erste Objekt in der Schlange die Botschaft „Bitte die Länge der Schlange geben“. Dieses Objekt benutzt seine Methode *AnzahlAbHierGeben()*, kann aber noch nicht antworten, da es nicht das Ende der Schlange ist und somit nicht weiß, wie viele noch dahinter kommen. Also schickt es eine Botschaft an sein Nachfolgeobjekt. Das nachfolgende Objekt reagiert mit seiner Methode *AnzahlAbHierGeben()*, handelt gleich und so weiter bis das wirklich letzte Objekt in der Schlange diese Botschaft bekommt. Es ist das einzige Objekt in der Schlange, das die Frage sofort beantworten kann. Es meldet dem Objekt, das ihm die Botschaft geschickt hat, „die Schlange hat (einschließlich mir) die Länge 1“, indem es als Rückgabewert 1 übergibt. Das vorhergehende Objekt addiert zu dem Wert die Zahl 1 und reicht diese Zahl wieder zu seinem vorhergehenden weiter usw., bis die Zahl beim ersten Objekt in der Schlange angekommen ist. Dieses reicht die Zahl an den Aufrufer weiter, der ihm die Botschaft geschickt hat, nämlich das Warteschlangen-Objekt.

```
// Methode der Schlangenobjekte
// Länge der Schlange hinter mir (mich eingeschlossen)
Methode AnzahlAbHierGeben() -> Ganzzahl
    wenn ich einen Nachfolger habe, dann
            // Botschaft weiterschicken, warten bis Antwort kommt,
            //  und diesen Antwortwert in anzahl ablegen
            Nachfolger.AnzahlAbHierGeben() -> anzahl
            anzahl = anzahl + 1   // mich dazuzählen
    sonst   // ich bin der letzte
            anzahl = 1
```

```
        endeWenn
        // diesen Wert gebe ich an meinen Aufrufer zurück
        Rückgabewert = anzahl
    EndeMethode

    // Methode der Warteschlange
    // Anzahl der Objekte in der Warteschlange
    Methode AnzahlGeben() -> Ganzzahl
        // das erste Objekt fragen wie lange die Schlange ist, warten bis Antwort kommt,
        // und diesen Antwortwert in anzahl ablegen
        anfang.AnzahlAbHierGeben() -> anzahl
        Rückgabewert = anzahl
    EndeMethode
```

Sequenzdiagramm für die Warteschlange an der Kasse:



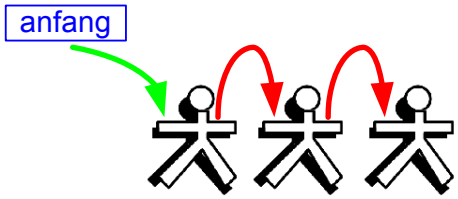
Im Rahmen des Unterrichtsverlaufs muss didaktisch abgewogen werden, ob man an dieser Stelle die Schülerinnen und Schüler z. B. die „Türme von Hanoi“ mit echten Scheiben spielen lässt und dabei den Algorithmus mit einer Methode *Turmbewegen*(*Turmhöhe*, *QuelleStab*, *ZielStab*, *AblageStab*) als rekursiven Ablauf erarbeitet. Das Spiel zeigt nochmals schön das Prinzip von rekursiven Abläufen, andererseits stört dieser Einschub etwas den roten Faden der Listenthematik.

Einige Methoden der Schlange bzw. der verketteten Liste lassen sich sowohl iterativ als auch rekursiv beschreiben. Zur Vertiefung des ‚rekursiven Denkens’ wird in den folgenden Kapiteln den rekursiven Abläufen der Vorzug gegeben. Das hat den Vorteil, dass später der rekursive Durchlauf bei Baumstrukturen leichter verstanden wird.

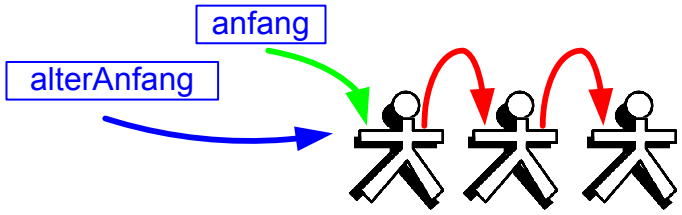
1.1.2.3 Entfernen und Einfügen bei einer Warteschlange

Das Entfernen der vordersten Person aus der Warteschlange ist ein einfaches Verfahren, denn über die Referenz *anfang* hat der Kassenangestellte Zugriff auf den ersten Kunden in der Schlange. Nur muss der erste Kunde vor dem Entfernen gefragt werden: „Wer ist Ihr Hintermann?“ Er antwortet durch Rückgabe einer Referenz auf den Hintermann und diese Referenz wird jetzt der neue Anfang der Warteschlange.

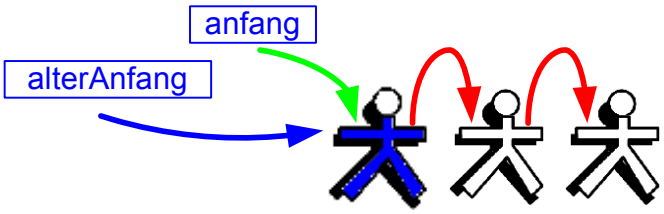
Der Kunde am Anfang der Warteschlange wird aus dieser entfernt.



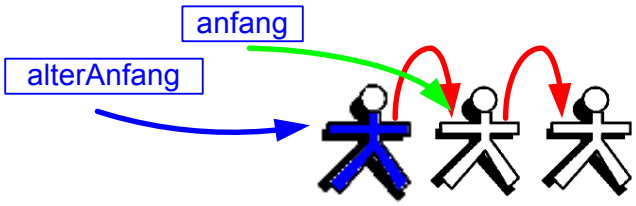
1) Ausgangssituation



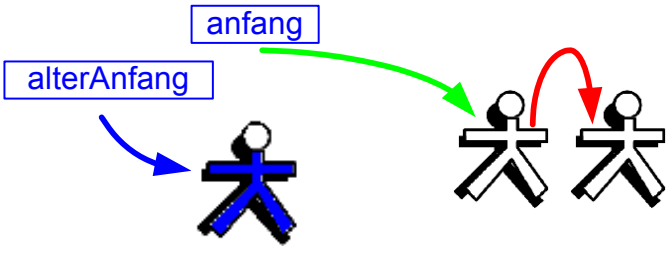
2) Der Verwalter der Warteschlange merkt sich eine weitere Referenz auf den Kunden am Anfang, damit er ihn bei dem Entfernen „nicht aus den Augen verliert“.



3) Er fragt den ersten Kunden (blau)[3] in der Warteschlange: „Wer ist Ihr Nachfolger?“ Dieser gibt eine Referenz auf seinen Nachfolger zurück.



4) Der Verwalter der Warteschlange merkt sich die Referenz auf den Nachfolger (aus 3.) als den neuen Anfang der Warteschlange.



5) Der ehemals erste Kunde ist jetzt aus der Warteschlange isoliert und über die Referenz *alterAnfang* ansprechbar. (Hinweis: Der ehemals erste Kunde besitzt immer noch eine Referenz auf seinen Nachfolger, diese ist aber nicht mehr von Bedeutung und deshalb nicht eingezeichnet.)

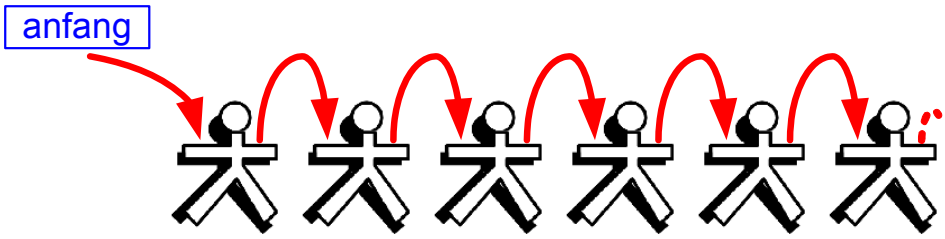
[3] Zu beachten ist hier – und an vielen weiteren Stellen – die Anmerkung nach dem Vorwort über die verwendete Farbcodierung: Die Angaben über verwendete Farben beziehen sich auf die Farbversion der Handreichung. Jedoch ist der Text auch ohne Farbdruck verständlich.

Das Warteschlangenobjekt entfernt das vorderste Objekt aus der Schlange, liefert eine Referenz auf dieses Objekt und setzt seinen Anfang auf das „neue“ erste Objekt.

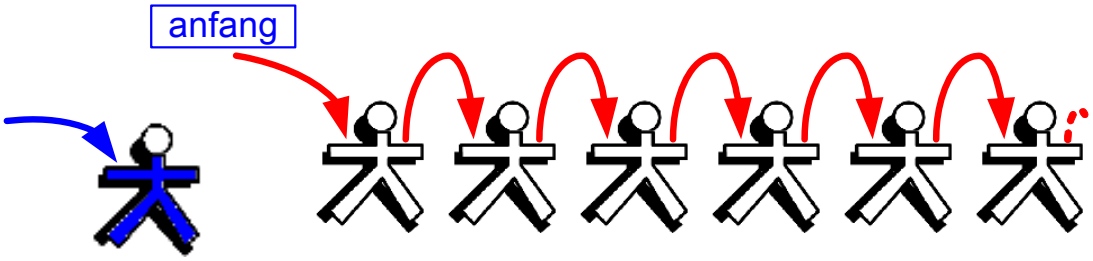
```
// Methode der Warteschlange
// Den ersten entfernen
Methode AnfangEntfernen() -> Referenz auf den Entfernten
    // Referenz auf den alten Anfang merken
    alterAnfang = anfang
    // Referenz auf den zweiten in der Schlange erfragen
    anfang.NachfolgerGeben() -> Referenz auf den Nachfolger des ersten Objekts
    // das ist jetzt der Anfang
    anfang = Referenz auf den Nachfolger des ersten Objekts
    // Referenz auf den alten Anfang zurückgeben
    Rückgabewert = alterAnfang
EndeMethode
```

Der Kassenangestellte ist auch zuständig, dass sich neu hinzukommende Personen „regelkonform“ in der Schlange anstellen. Das Einfügen eines neuen Kunden in der Warteschlange gestaltet sich etwas aufwendiger. Auch hier bedient sich der Angestellte der Fähigkeiten der Kunden. Da er nur Zugriff auf den ersten Kunden in der Warteschlange hat, schickt er diesem die Botschaft „Hier ist ein neuer Kunde. Würden Sie diesen bitte hinten einordnen?“. Der erste Kunde reagiert mit seiner Methode *HintenEinfuegen*(*neuer Kunde*), die als Eingangsparameter eine Referenz auf den neuen Kunden hat. Da der erste Kunde nicht der letzte Kunde in der Warteschlange ist, kann er ihn nicht direkt hinter sich einfügen, das wäre unfair. Deshalb schickt er an seinen Hintermann die Botschaft „Hier ist ein neuer Kunde. Würden Sie diesen bitte hinten einordnen?“ usw. Erst der letzte Kunde in der Reihe kann den neuen Kunden hinter sich einfügen, der jetzt sein Hintermann ist.

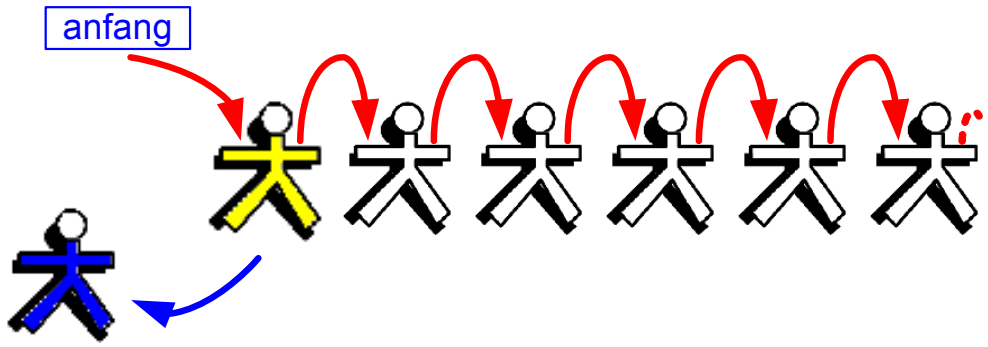
Ein neuer Kunde wird am Ende der Warteschlange eingefügt.



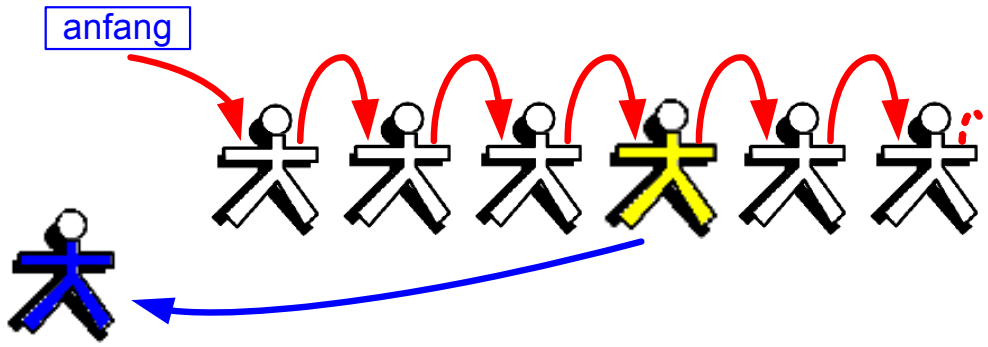
1) Ausgangssituation



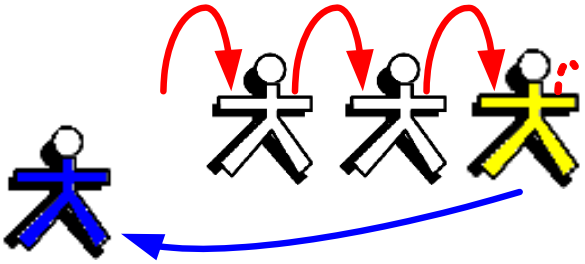
2) Ein neuer Kunde kommt (hier links im Bild mit blauer Farbe markiert). Es existiert eine Referenz auf diesen Kunden.



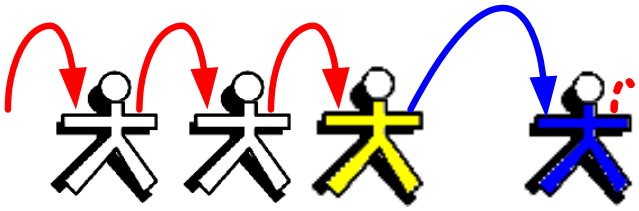
3) Der Verwalter der Warteschlange schickt die Botschaft „Bitte hinten einfügen“ an den ersten Kunden (hier gelb) und gibt als Zusatzinformation eine Referenz auf den neuen Kunden mit.



4) Die Botschaft „Hinten einfügen“ wird immer weiter nach hinten weitergeschickt und jeweils die Referenz auf den neuen Kunden mitgegeben …



5) … bis die Botschaft beim Letzten (gelb) eingetroffen ist.



6) Der Letzte erklärt den neuen Kunden zu seinem Nachfolger und teilt diesem mit, dass er jetzt der Letzte ist (d. h. ohne Nachfolger).

```
// Methode der Schlangenobjekte
// Hinten, am Ende der Schlange einfügen
Methode HintenEinfuegen(Referenz auf neuen Kunden)
    wenn ich einen Nachfolger habe, dann
        // Botschaft weiterschicken zusammen mit der Referenz auf den neuen Kunden
        Nachfolger.HintenEinfuegen(Referenz auf neuen Kunden)
    sonst   // ich bin der letzte
        // Der neue Kunde wird zum Nachfolger
        Nachfolger = Referenz auf neuen Kunden
        Nachfolger.SetzeDeineNachfolgerReferenz auf keinen
    endewenn
EndeMethode
```

```
// Methode der Warteschlange
// Einen neuen Kunden am Ende einfügen
Methode EndeEinfuegen(Referenz auf neuen Kunden)
    // dem ersten Objekt sagen, dass sich der neue Kunde hinten anstellen soll
    anfang.HintenEinfuegen(Referenz auf neuen Kunden)
EndeMethode
```

1.1.3 Die einfach verkettete Liste

1.1.3.1 Von der Warteschlange zur Liste

In der Praxis treten bei einer Warteschlange, etwa vor einer Kasse, beispielsweise auch folgende Situationen auf:

- Einem Kunden in der Warteschlange dauert das Anstehen zu lange, er möchte aus der Warteschlange ausscheren. Bei der Modellierung dieser Situation ist eine Methode Entfernen erforderlich, die ein beliebiges Objekt aus der Schlange entfernt.
- Kunden mit weniger als 3 Kaufgegenständen werden in der Schlange gesucht und dürfen sofort an die Kasse treten. Eine Methode Suchen nach einem Objekt mit bestimmter Eigenschaft ist hierzu nützlich.
- Die Schülerinnen und Schüler einer Klasse sollen sich in einer Schlange nach alphabetischer Reihenfolge aufstellen. Eine Methode *SortiertEinfuegen* löst in der Implementierung diese Aufgabe.

Bei der Umsetzung dieser Anforderungen reichen die bisherigen Methoden *EndeEinfuegen* und *AnfangEntfernen* nicht mehr aus. Es werden weitere Methoden benötigt. Dies führt weg von der Klasse WARTESCHLANGE zu der Klasse LISTE, die neben den Grundmethoden der Schlange auch noch weitere Methoden aufweist, die bei einer Anordnung von Objekten nützlich sind.

In den Kapiteln 1.1.3.2 mit 1.1.3.6 werden zuerst die Methoden der Klasse LISTE realisiert, die für eine ‚reine' Warteschlange (nur mit Einfügen und Entfernen) nötig sind. In den folgenden Kapiteln werden dann die Fähigkeiten der Klasse LISTE erweitert.

Für den weiteren Ablauf des Unterrichts muss die Lehrkraft entscheiden, ob erst eine Klasse WARTESCHLANGE implementiert wird, bei der Daten und Struktur nicht getrennt sind und die nur über die Methoden *EndeEinfuegen* und *AnfangEntfernen* verfügt. An diesem Beispiel könnte in einer einfacheren Form in die Thematik der Rekursion eingeführt werden. Eine derartige Umsetzung ist im *BlueJ-Projekt WarteschlangeEinfach* realisiert. Diese Klasse WARTESCHLANGE müsste dann zu einer Liste mit getrennten Daten und Struktur umgebaut werden.

Alternativ könnte man sofort mit der Klasse LISTE (wobei Daten und Struktur getrennt sind) beginnen und deren Fähigkeiten Schritt für Schritt, ausgehend von der Schlange, erweitern. Im Folgenden wird dieser Weg dargestellt.

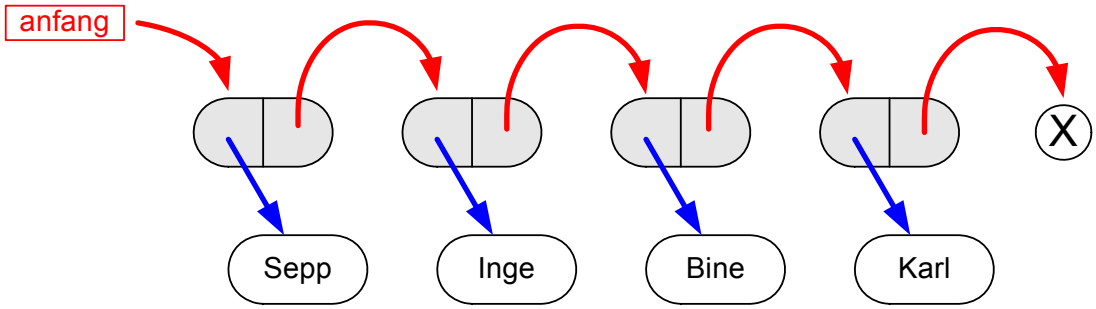
1.1.3.2 Die Klassen KNOTEN und LISTE (erste Version)

Lp: Implementierung einer einfach verketteten Liste als Klasse mittels Referenzen unter Verwendung eines geeigneten Softwaremusters (Composite); Realisierung der Methoden zum Einfügen, Suchen und Löschen.

Lp: graphische Veranschaulichung der Methoden zum Einfügen (auch an beliebiger Stelle), Suchen und Löschen.

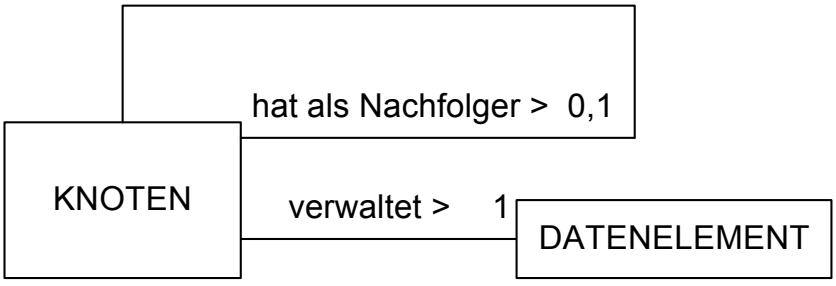
Ausgehend von den Überlegungen an einer Warteschlange an der Kasse in einem Supermarkt wird jetzt das allgemeine Prinzip einer einfach verketteten Liste entwickelt. Dabei wird der Abstrakte Datentyp (ADT) Liste konzipiert.

Die verkettete Liste ist ein typischer Vertreter einer dynamischen Datenstruktur. Die Liste besteht aus einer Menge von Knoten, die untereinander „verkettet“ sind. Jeder Knoten besteht aus einer Referenz auf das eigentlich zu speichernde Datenobjekt sowie einer Referenz auf den nächsten Knoten. Der letzte Knoten der Liste verweist auf *null*. In der Liste selbst muss nur noch eine Referenz auf den Anfangsknoten vermerkt sein, alle anderen Knoten sind über die Verkettung erreichbar. Der Anker für den Beginn der Liste wird im Folgenden mit *anfang* bezeichnet.



Knotenobjekte und Datenobjekte einer einfach verketteten Liste

In obiger Abbildung ist links der Anfang der Liste, rechts das Ende. Der letzte Knoten verweist auf *null* (durch den Kreis mit X dargestellt).



Klassendiagramm

Die Beziehungen werden durch Referenzattribute in der Klasse KNOTEN realisiert. Durch die Referenz eines Knotenobjekts auf ein anderes Knotenobjekt bilden die Knoten eine rekursive Datenstruktur („Ein Knoten ist ein Objekt, das einen Knoten als Nachfolger hat“).

Neben den üblichen Getter- und Setter-Methoden (Abfrage- und Änderungs-Methoden) benötigt die Klasse KNOTEN noch weitere Methoden. Welche erforderlich sind, lässt sich aus den Überlegungen zur Warteschlange erarbeiten.

I) Erforderliche Attribute der Klasse KNOTEN:

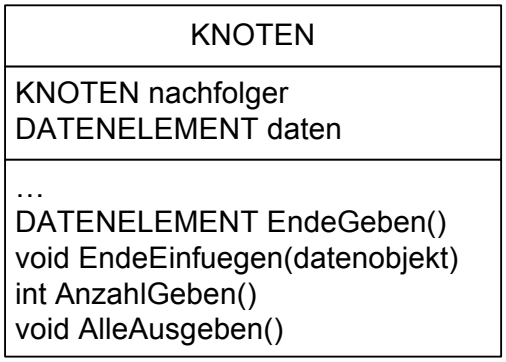
- nachfolger
  Hier wird die Referenz auf den nachfolgenden Knoten gespeichert. Das Objekt am Ende der Liste hat im Attribut *nachfolger* den Wert null.
- daten
  Hier wird die Referenz auf ein Datenobjekt gespeichert, das von dem Knoten verwaltet wird. In unserer Implementation ist es eine Referenz auf ein Objekt der Klasse DATENELEMENT.

II) Standardmethoden der Klasse KNOTEN (Konstruktor, Getter-Methoden, Setter-Methoden):

- KNOTEN()
  Anlegen eines Knoten-Objekts (Konstruktor) mit *nachfolger* = null und *daten* = null
- KNOTEN(Referenz auf Knotenobjekt, Referenz auf Datenobjekt)
  Anlegen eines Knoten-Objekts (Konstruktor) mit Referenz auf den Nachfolger dieses Knotens und Referenz auf das zu verwaltende Datenobjekt.
- NachfolgerSetzen(Referenz auf Knotenobjekt)
- NachfolgerGeben() -> Referenz auf Knotenobjekt
- DatenSetzen(Referenz auf Datenobjekt)
- DatenGeben() -> Referenz auf Datenobjekt

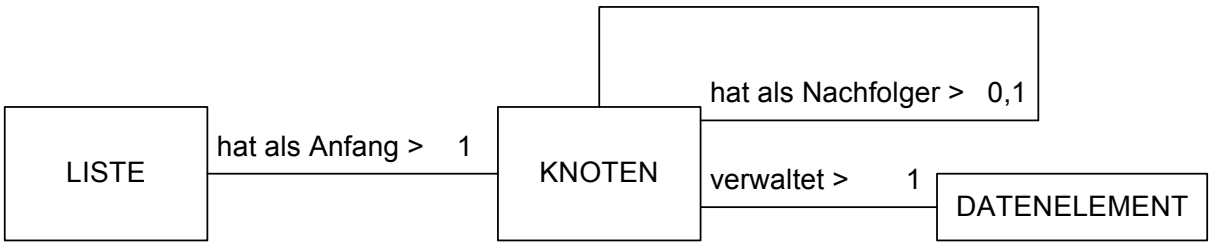
III) Weitere Methoden der Klasse KNOTEN zur Verwaltung der Daten:

- EndeGeben() -> Referenz auf Datenobjekt am Ende
  Die Funktion liefert eine Referenz auf das Datenobjekt, das vom Knotenobjekt am Ende der Liste verwaltet wird.
- EndeEinfuegen(Referenz auf ein Datenobjekt)
  Fügt am Ende der Verkettung einen neuen Knoten an, der das übergebene Datenobjekt verwaltet.
- AnzahlAbHierGeben() -> Ganzzahl
  Anzahl der Knoten, die Nachfolger des Knoten sind, einschließlich des Knoten selbst.
- AlleAusgeben()
  Diese Methode gibt für das Knotenobjekt und alle seine Nachfolger (entsprechend der Verkettungsreihenfolge) die Inhalte der verwalteten Datenobjekte aus. Sie ruft hierzu die Methode *Ausgeben*() der Datenobjekte auf, denn sie selbst kennt die Struktur der Datenobjekte nicht.



Erweitertes Klassendiagramm der Klasse KNOTEN

Die Organisation der Datenstruktur „einfach verkettete Liste“ übernimmt ein Objekt der Klasse LISTE. Dieses Objekt hat eine Referenz auf den Anfangsknoten der Liste.



Klassendiagramm

Hinzufügen, Entfernen und Abrufen der in der Datenstruktur gespeicherten Daten darf nur über die Methoden des Listenobjekts erfolgen (Datenkapselung). Die Erarbeitung der Attribute und Methoden der Klasse LISTE ergibt die folgenden Gesichtspunkte. Dabei werden im ersten Schritt hauptsächlich die Methoden überlegt, die erforderlich sind, dass die Liste als Datenstruktur Schlange verwendet werden kann.

I) Erforderliche Eigenschaften (Attribute) der Klasse LISTE:

- anfang
  Hier wird die Referenz auf den Anfangsknoten der Liste gespeichert. Auf alle weiteren Konten kann über die Verkettung zugegriffen werden.

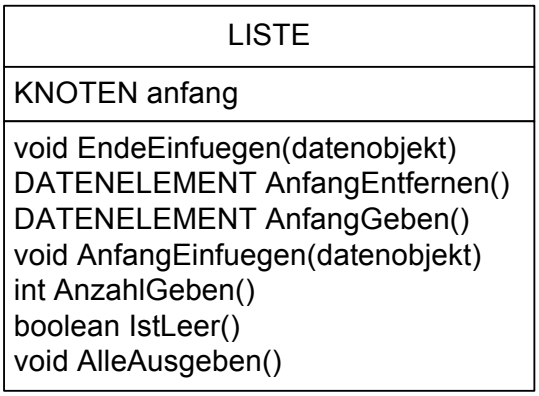
II) Standardmethoden der Klasse LISTE:

- LISTE()
  Anlegen eines Listen-Objekts (Konstruktor) mit *anfang* = null

III) Weitere Methoden der Klasse LISTE, damit sie zur Realisierung einer Schlange eingesetzt werden kann:

- IstLeer() -> true/false
  Prüft, ob die Liste leer ist.
- EndeEinfuegen(Referenz auf neues Datenobjekt)
  Fügt mittels eines Knotens das Datenobjekt am Ende der Liste ein.
  Es wird die Methode *EndeEinfuegen*() des Anfangsknotens verwendet, die sich rekursiv der anderen Knoten bedient.
- AnfangEntfernen() -> Referenz auf Datenobjekt am Anfang
  Entfernt das Knotenobjekt am Anfang der Verkettung und gibt eine Referenz auf das von diesem Knoten verwaltete Datenobjekt zurück.
- AnfangGeben() -> Referenz auf Datenobjekt am Anfang
  Gibt eine Referenz auf das vom Knotenobjekt am Anfang verwaltete Datenobjekt zurück, ohne das Knotenobjekt zu entfernen.
- AnfangEinfuegen(Referenz auf neues Datenobjekt)
  Fügt einen Knoten am Anfang der Verkettung ein, der das neue Datenobjekt verwaltet.
- AnzahlGeben() -> Ganzzahl
  Gibt die aktuelle Anzahl der Knotenobjekte in der Liste aus. Die Methode bedient sich hierzu der Methoden AnzahlAbHierGeben() der Klasse KNOTEN.
- AlleAusgeben()
  Gibt die Inhalte aller in der Liste verwalteten Datenobjekte aus. Hierbei wird zunächst

die Methode Ausgeben() der Klasse KNOTEN und letztendlich die Methode Ausgeben() der Datenobjekte aufgerufen.

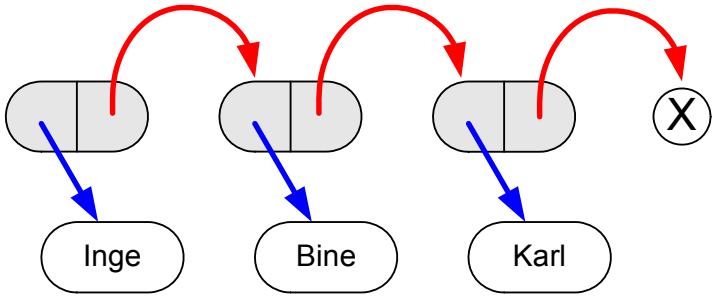


Erweitertes Klassendiagramm der Klasse LISTE

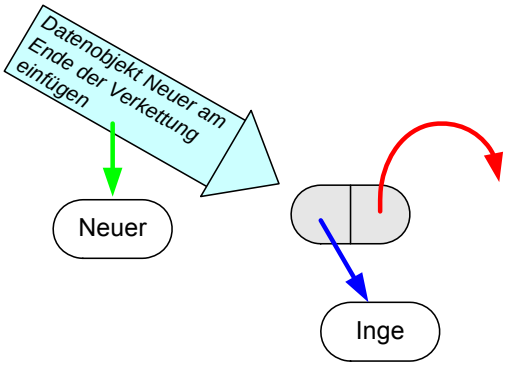
1.1.3.3 Methoden der Klasse KNOTEN

Die Getter- und Setter-Methoden sowie die Konstruktoren der Klasse KNOTEN bedürfen keiner weiteren Überlegungen. Die Methode *EndeEinfuegen*(datenobjekt) der Klasse KNOTEN muss noch näher untersucht werden.

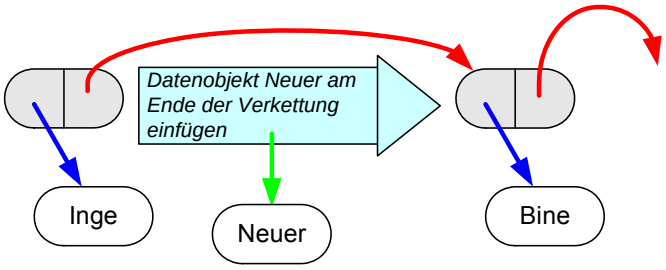
Methode EndeEinfuegen(datenobjekt) der Klasse KNOTEN



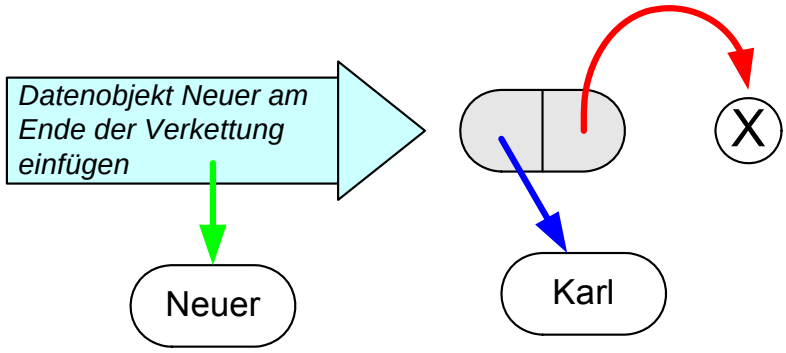
1) Ausgangssituation mit 3 Knotenobjekten



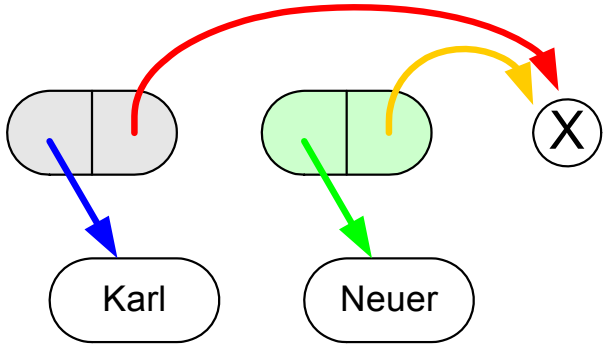
2) Durch Aufruf der Methode *EndeEinfuegen*(datenobjekt) wird eine Botschaft an ein Knotenobjekt geschickt, zusammen mit der Referenz auf das einzufügende Datenobjekt.



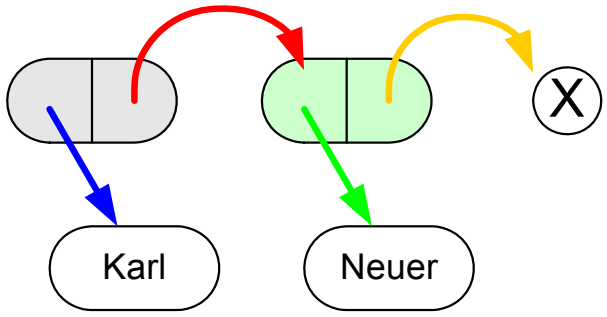
3a) Ist das Knotenobjekt nicht am Ende der Verkettung, so schickt es diese Botschaft sofort, zusammen mit der Referenz auf das einzufügende Datenobjekt, an seinen Nachfolgeknoten weiter.



3b) Das Knotenobjekt am Ende bekommt in der Methode *EndeEinfuegen*(datenobjekt) als Eingangswert eine Referenz (grüner Pfeil) auf das einzufügende Datenobjekt Neuer mitgeliefert.



4) In der Methode wird ein neues Knotenobjekt erzeugt (grüner Knoten, Knotenobjekt rechts im Bild). Dem Attribut nachfolger des neuen Kontenobjekts wird eine Referenz auf das Ende, d. h. null (oranger Pfeil), zugewiesen, dem Attribut daten die Referenz auf das einzufügende Datenobjekt.



5) In der Methode bekommt der bisherige Endknoten als Nachfolger (Attribut nachfolger) eine Referenz auf den neuen Knoten.

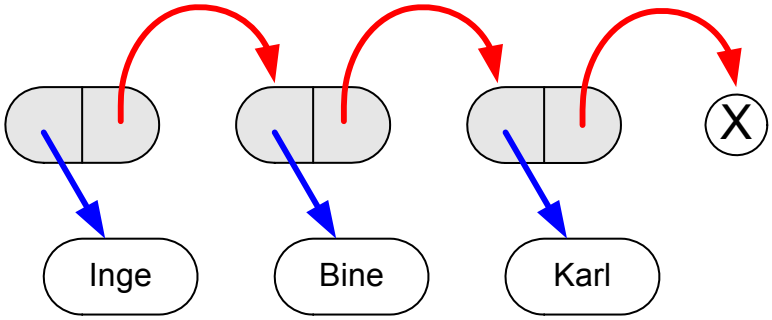
Die weiteren Methoden der Klasse KNOTEN, also *EndeGeben*(), *AnzahlAbHierGeben*() und *AlleAusgeben*() können auf ähnliche Weise veranschaulicht werden, wobei sich zwei verschiedene Darstellungsverfahren zur Veranschaulichung der rekursiven Abläufe eignen: durch Aufrufbotschaften und Rückmeldungen oder durch ein Sequenzdiagramm.

Die Vorgabe für die Methode *EndeGeben*() ist:

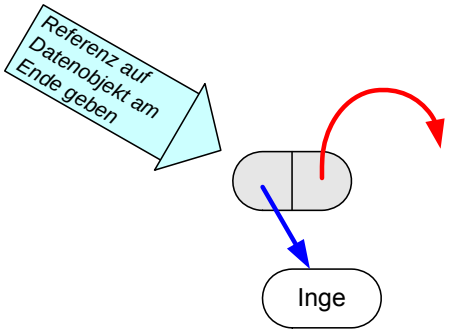
EndeGeben() -> Referenz auf Datenobjekt am Ende

Die Funktion liefert eine Referenz auf das Datenobjekt, das vom Knotenobjekt am Ende der Liste verwaltet wird.

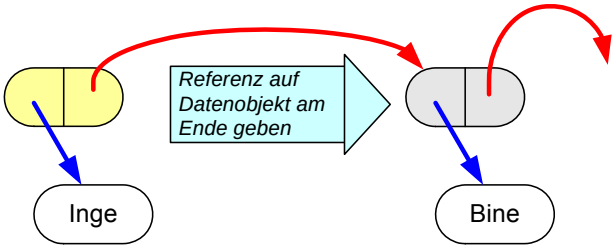
Methode EndeGeben() der Klasse KNOTEN



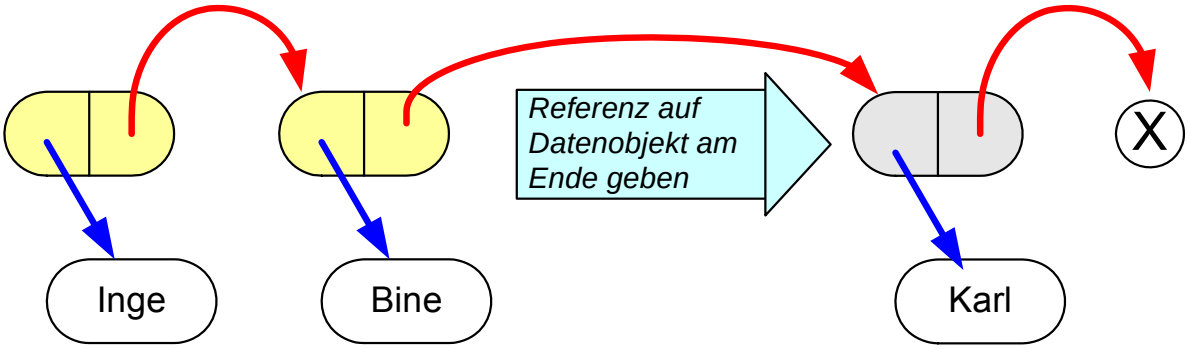
1) Ausgangssituation mit 3 Knotenobjekten



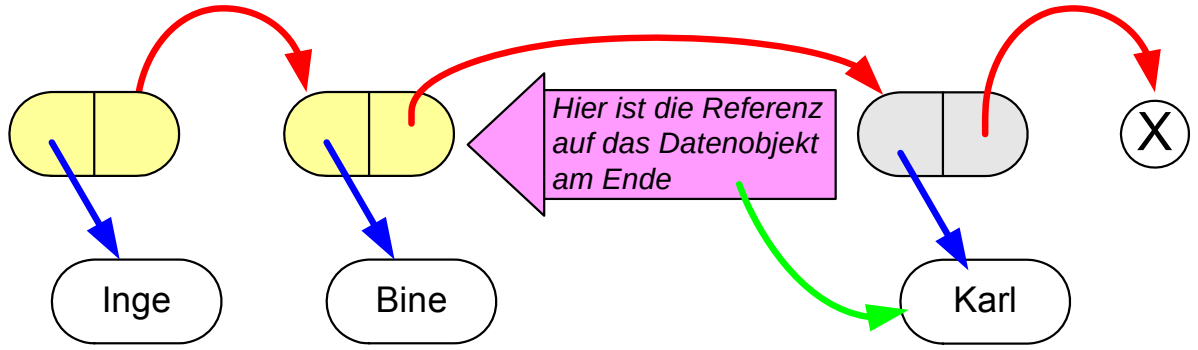
2) Durch Aufruf der Methode EndeGeben() wird eine Botschaft an ein Knotenobjekt geschickt. Es wird aufgefordert, eine Referenz auf das Datenobjekt am Ende zurückzuliefern.



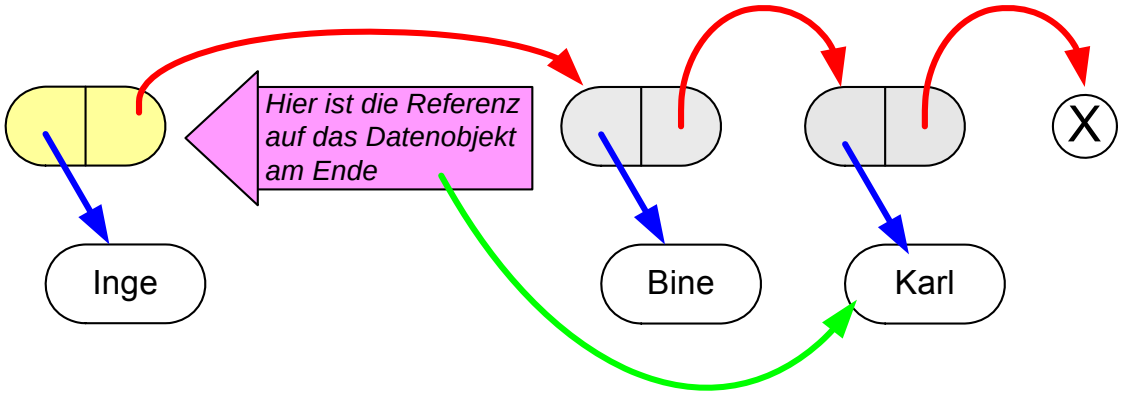
3a) Ist das Knotenobjekt nicht am Ende der Verkettung, so kann es diesen Auftrag zunächst nicht erfüllen und schickt diese Botschaft sofort an seinen Nachfolgeknoten weiter. Das Knotenobjekt (gelb, links) muss auf die Antwort warten.



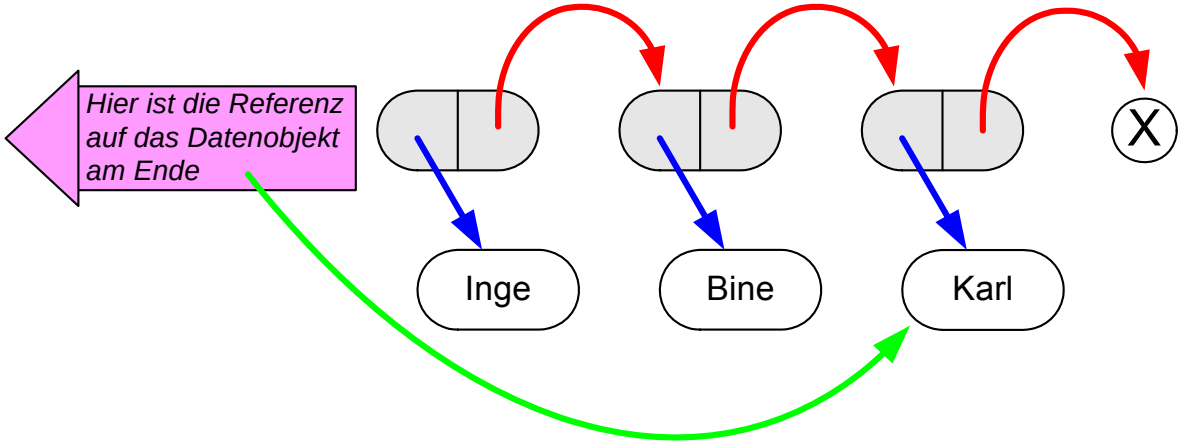
3b) Dieses Weiterschicken der Botschaft läuft bis zum Knotenobjekt am Ende. Alle Knoten davor befinden sich im Wartezustand, d. h. sie warten auf die Antwort, damit sie selbst weiterarbeiten können.



4) Das Knotenobjekt am Ende kann als Einziger diesen Auftrag sofort bearbeiten. Es liefert eine Referenz auf sein Datenobjekt an den Aufrufer zurück.



5) Der wartende Vorgänger kann (endlich) seine Arbeit fortsetzen. Wenn er mit seiner Arbeit fertig ist, gibt er den Rückgabewert an seinen Aufrufer weiter. In diesem Fall ist es die empfangene Referenz auf das Datenobjekt am Ende der Liste.



6) Das Ganze läuft solange zurück, bis der erste aufgerufene Knoten seine Arbeit erledigen und den Rückgabewert an den Aufrufer übergeben kann.

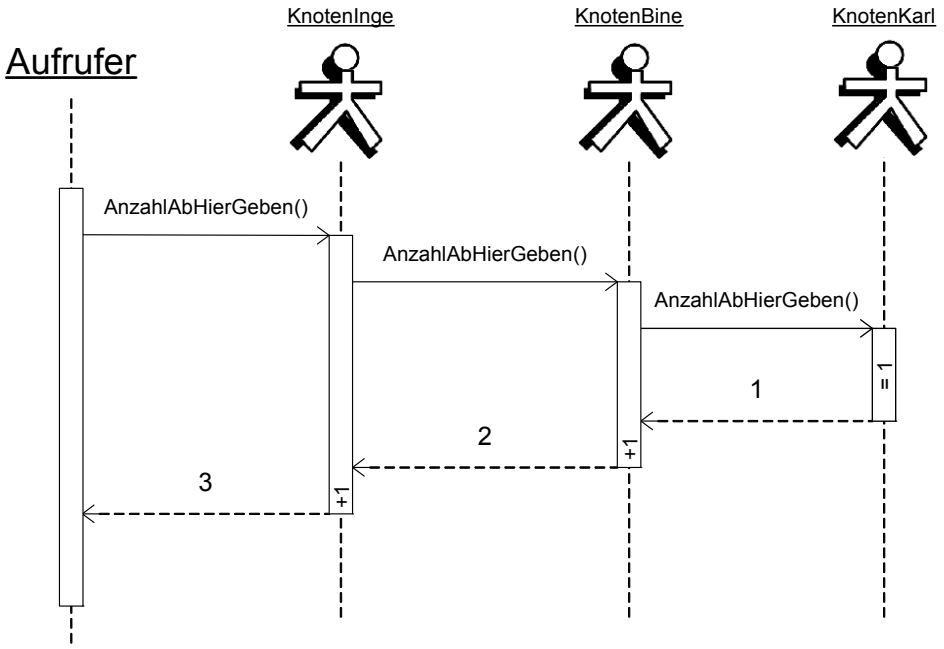
An diesem Ablauf kann man schön den rekursiven Ablauf sehen: Die Botschaft läuft bis zu dem betroffenen Objekt, die Antwort läuft wieder den Aufrufweg zurück.

Zur Erklärung der Methode *AnzahlAbHierGeben*() wird ein Sequenzdiagramm verwendet. Die Vorgabe für die Methode *AnzahlAbHierGeben*() ist:

AnzahlAbHierGeben() -> Ganzzahl

Zurückgegeben wird die Anzahl der Knoten, die Nachfolger des Knotens sind, einschließlich des aufgerufenen Knotens selbst.

Auch hier kann nur der letzte Knoten diese Frage direkt beantworten, die anderen Knoten benötigen die Hilfe des Nachfolgers.



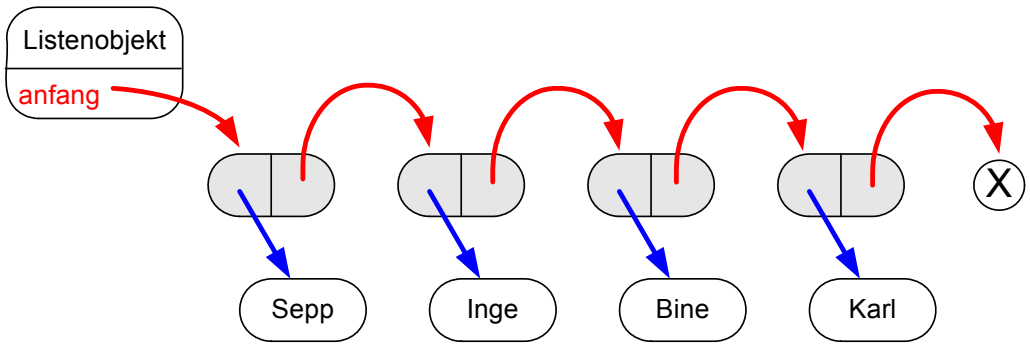
Eine Implementierung der Methoden der Klasse KNOTEN folgt im Abschnitt 1.1.3.5.

1.1.3.4 Methoden der Klasse LISTE

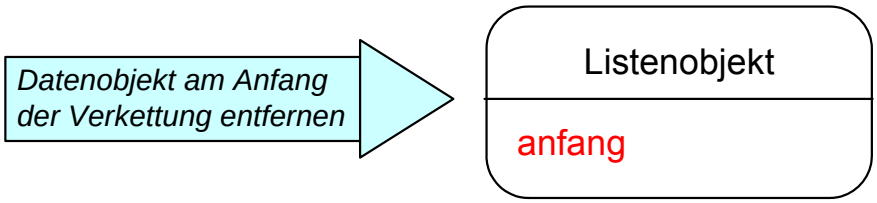
Der Konstruktor der Klasse LISTE erfordert keine weiteren Überlegungen. Die Methoden EndeEinfuegen(datenobjekt), AnzahlGeben() und AlleAusgeben() werden durch den Aufruf der entsprechenden Methoden des ersten Knotens der Liste realisiert.

Für die Methoden *AnfangEinfuegen*(datenobjekt) und *AnfangEntfernen*() benötigt ein Listenobjekt keine Hilfe der Knotenobjekte. Diese Methoden müssen aber noch näher untersucht werden.

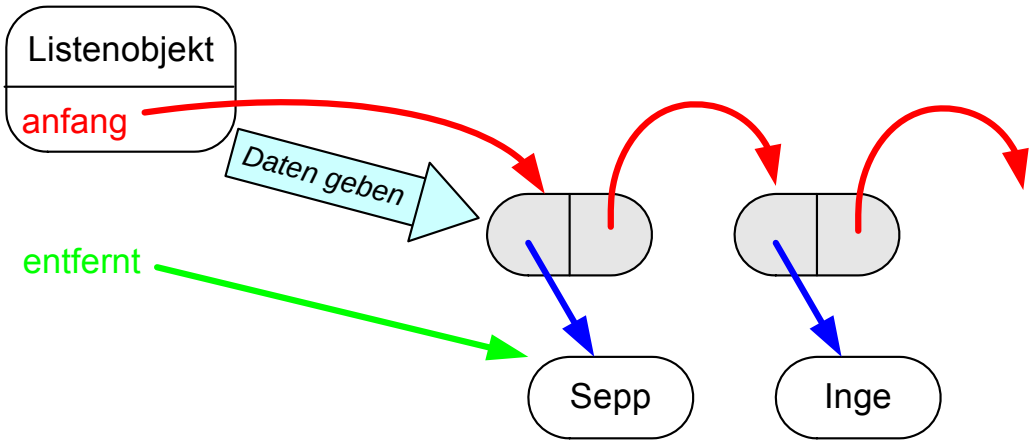
Methode AnfangEntfernen() -> Referenz auf Datenobjekt



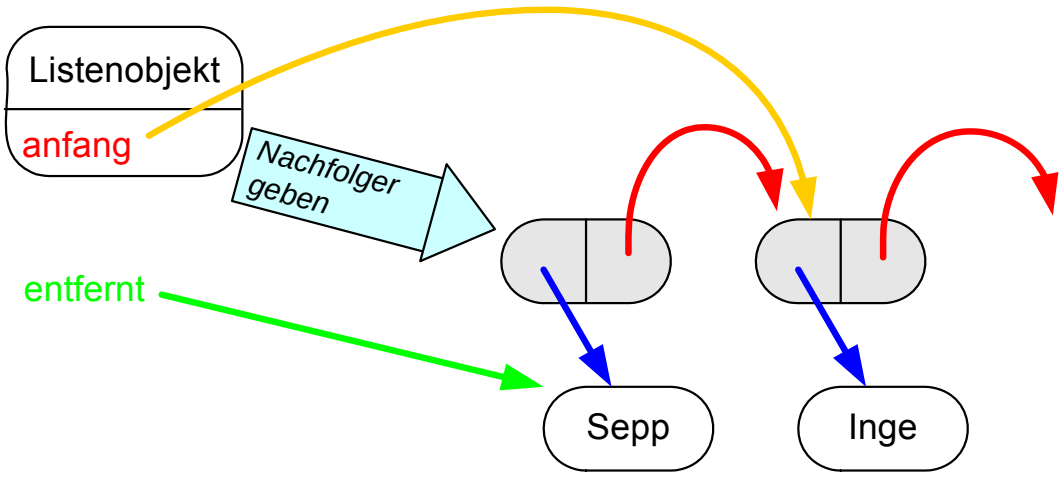
1) Ausgangssituation



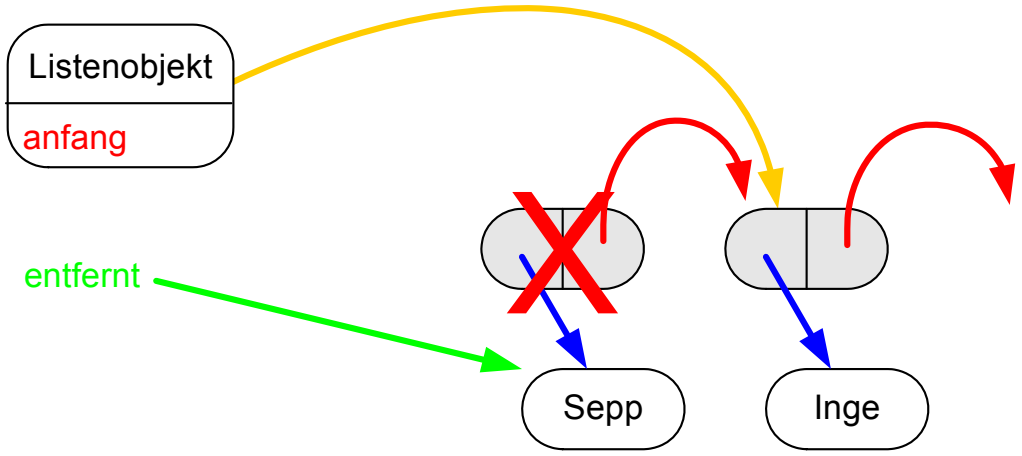
2) Durch Aufruf der Methode AnfangEntfernen() wird eine Botschaft an ein Listenobjekt geschickt.



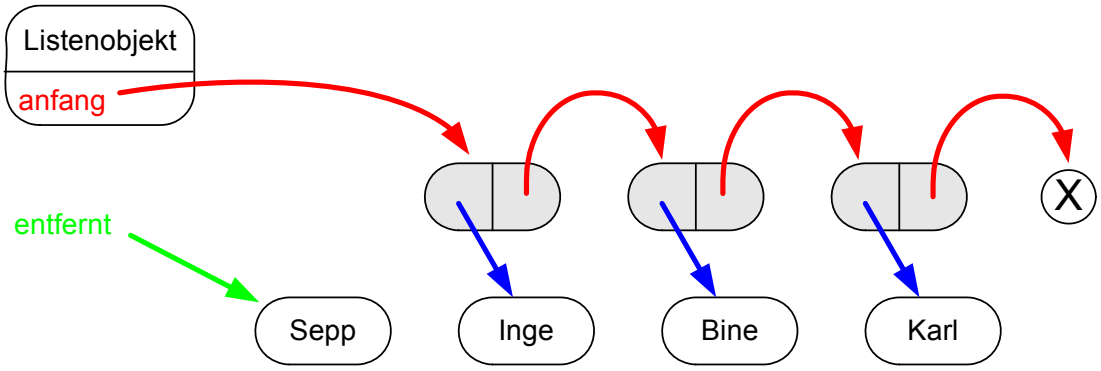
3) Das Listenobjekt reagiert und schickt die Botschaft DatenGeben() an den Anfangsknoten. Dieser gibt eine Referenz auf das von ihm verwaltete Datenobjekt zurück (Datenobjekt Sepp, grüner Pfeil). Das Listenobjekt merkt sich diese Referenz innerhalb seiner Methode AnfangEntfernen() in der Variablen *entfernt*.



4) Das Listenobjekt schickt die Botschaft NachfolgerGeben() an den Anfangsknoten. Dieser gibt eine Referenz auf sein Nachfolgeobjekt zurück. In der Methode AnfangEntfernen() wird das Referenzattribut *anfang* auf diesen Wert gesetzt.



5) Der ehemalige Anfangsknoten wird von keinem anderen Objekt gebraucht (referenziert). Beim nächsten ‚Aufräumen' (Garbage Collect) des Systems wird er deshalb aus dem Speicher entfernt (angedeutet durch das fette X).



6) Es ergibt sich die gewünschte Endsituation. Das Listenobjekt gibt die Referenz auf das Datenobjekt entfernt an den Aufrufer zurück.

Die Methode *AnfangEinfuegen*(datenobjekt) lässt sich in der analogen Art anschaulich darstellen. Eine Implementierung der Methoden erfolgt im nächsten Abschnitt.

1.1.3.5 Implementierungen von KNOTEN und LISTE

Die Implementierung sollte nicht erst am Schluss nach der Besprechung der einzelnen Methoden erfolgen, sondern immer parallel zur Erklärung. Die einzelnen Methoden können auch gleich getestet werden. Hierzu ist aber Voraussetzung, dass die Methode *AlleAusgeben*() als erstes implementiert wird.

Im Folgenden wird in Ausschnitten eine Implementierung in Java vorgestellt. Die Klasse DATENELEMENT und deren Unterklassen KUNDE, KARTE, GANZZAHL werden wie im Abschnitt 1.1.1.1 bzw. 1.1.1.3 angegeben definiert.

Eine erste Implementierung der Schüler wird sicher Schwächen aufweisen, die im Anschluss diskutiert und behoben werden müssen. Zum Beispiel wird bei der Klasse LISTE meist übersehen, dass in allen Methoden zu überprüfen ist, ob die Liste nicht leer ist. Dieser Fall muss gesondert behandelt werden.

```
class KNOTEN
{
  private KNOTEN nachfolger;
  private DATENELEMENT daten;

  // Konstruktoren
  KNOTEN()
  {
    nachfolger = null;
    daten = null;
  }

  KNOTEN(KNOTEN naechsterKnoten, DATENELEMENT datenObjekt)
  {
    nachfolger = naechsterKnoten;
    daten = datenObjekt;
  }

  // Getter- und Setter-Methoden

  void NachfolgerSetzen(KNOTEN neuerNachfolger)
  {
    nachfolger = neuerNachfolger;
  }

  KNOTEN NachfolgerGeben()
  {
    return nachfolger;
  }

  void DatenSetzen(DATENELEMENT datenObjekt)
  {
    daten = datenObjekt;
  }

  DATENELEMENT DatenGeben()
  {
    return daten;
  }

  // weitere Methoden

  DATENELEMENT EndeGeben()
  {
    if (nachfolger == null)
      return daten;
    else
      return nachfolger.EndeGeben();
  }

  void EndeEinfuegen(DATENELEMENT datenObjekt)
  {
    if (nachfolger == null)
      nachfolger = new KNOTEN(null, datenObjekt);
    else
      nachfolger.EndeEinfuegen(datenObjekt);
  }
```

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

  //Voraussetzung: daten ungleich null!
  void AlleAusgeben()
  {
    daten.Ausgeben();
    if (nachfolger != null)
      nachfolger.AlleAusgeben();
  }

}
```

```
class LISTE
{
  private KNOTEN anfang;

  // Konstruktor
  LISTE(){
    anfang = null;
  }

  // weitere Methoden

  boolean IstLeer()
  {
    return (anfang == null);
  }

  void EndeEinfuegen(DATENELEMENT neueDaten)
  {
    if (!IstLeer())
      anfang.EndeEinfuegen(neueDaten);
    else
      AnfangEinfuegen(neueDaten);
  }

  void AnfangEinfuegen(DATENELEMENT neueDaten)
  {
    KNOTEN n = new KNOTEN(anfang, neueDaten);
    anfang = n;
  }

  DATENELEMENT AnfangEntfernen()
  {
    DATENELEMENT entfernt;

    if (!IstLeer()){
      entfernt = anfang.DatenGeben();
      anfang = anfang.NachfolgerGeben();
    }else{
      entfernt = null;
    }
    return entfernt;
  }
```

```
  DATENELEMENT AnfangGeben()
  {
    if (!IstLeer())
      return anfang.DatenGeben();
    else
      return null;
  }

  void AlleAusgeben()
  {
    if (!IstLeer())
      anfang.AlleAusgeben();
  }

  int AnzahlGeben()
  {
    if (!IstLeer())
      return anfang.AnzahlAbHierGeben();
    else
      return 0;
  }

}
```

Die Schülerinnen und Schüler testen in einem kleinen Programm (bzw. in einer Testklasse mit einer Methode *Ausfuehren*()) die Klasse LISTE als Schlange und die Richtigkeit der Implementierung. Ein möglicher Ausschnitt aus einem derartigen Programmcode könnte wie folgt aussehen (vgl. BlueJ-Projekt ListeNormal):

```
LISTE warteSchlange = new LISTE();

warteSchlange.EndeEinfuegen(new KUNDE("Huber","Eduard"));
warteSchlange.EndeEinfuegen(new KUNDE("Meier","Paul"));
warteSchlange.EndeEinfuegen(new KUNDE("Müller","Inge"));
...
warteSchlange.AlleAusgeben();
warteSchlange.AnfangEntfernen();
warteSchlange.AlleAusgeben();
...
```

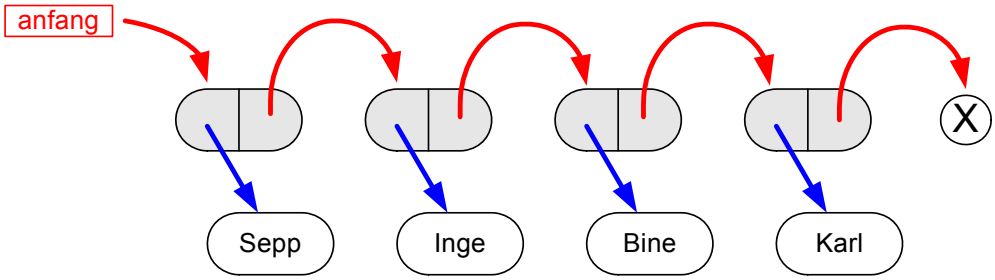
1.1.3.6 Klasse LISTENELEMENT – Entwurfsmuster Kompositum

Lp: Implementierung einer einfach verketteten Liste als Klasse mittels Referenzen unter Verwendung eines geeigneten Softwaremusters (Composite); Realisierung der Methoden zum Einfügen, Suchen und Löschen.

1.1.3.6.1 Grundlagen

Die Methoden der Klasse LISTE sind aufwendig zu implementieren, wenn man immer die Sicherheitsabfrage auf eine leere Liste durchführen muss. Beim Einfügen am Ende muss sogar anders reagiert werden, damit das Einfügen bei einer leeren Liste funktioniert. Auch die Methoden der Klasse KNOTEN verfügen immer unschöne „Abfragen“ der Art: „Bin ich das Ende der Verkettung? Wenn ja, dann muss ich anders reagieren.“

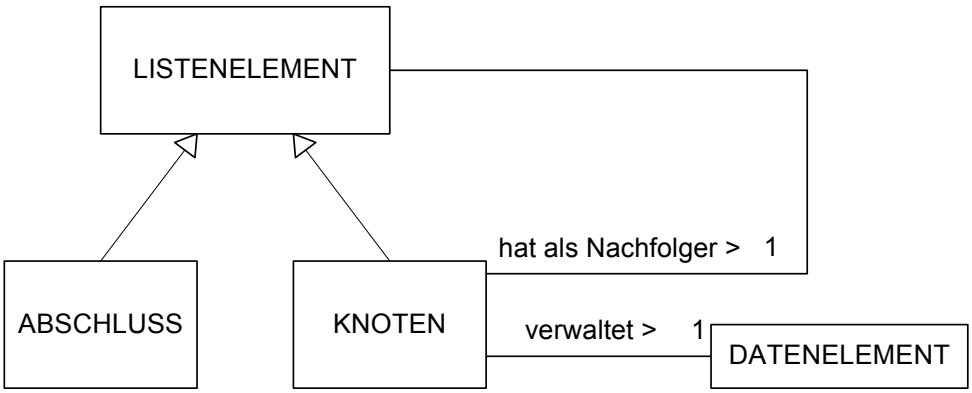
Scheinbar war die Modellierung der einfach verketteten Liste nicht optimal. Schauen wir uns deshalb die Struktur der Liste noch mal genauer an.



Knotenobjekte und Datenobjekte einer einfach verketteten Liste.

Eine besondere Rolle nimmt der Knoten am Ende der Liste ein, er hat keinen Nachfolger. Einen Knoten ohne Nachfolger bezeichnet man in der Informatik als Blatt. Jetzt könnte man auf folgende Idee kommen: Es wird eine Klasse BLATT definiert und das letzte Objekt in der Liste stammt von dieser Klasse. In obigem Beispiel gäbe es also 3 Objekte der Klasse KNOTEN und ein Objekt der Klasse BLATT. Diese Modellierung ist aber keine geeignete Lösung. Wenn das Blatt am Ende der Liste entfernt wird, dann würde der dritte Knoten von links zu einem Blatt werden. Aber: Ein Objekt kann nicht einfach seine Klasse wechseln, es gehört immer zu der Klasse, aufgrund deren Vorlage es erstellt wurde.

Eine andere bessere Lösung wäre, wenn man am Ende ein neues zusätzliches Objekt anfügt (in der Zeichnung schon als Kreis mit X erkennbar). Dieses Objekt schließt die Liste ab. Deshalb bezeichnen wir seine Klasse ABSCHLUSS. Der normale Knoten am Ende der Liste sollte gar nicht merken, dass er der Letzte ist, und deshalb wie gewohnt seinen Nachfolger referenzieren, der aber ein Objekt der Klasse ABSCHLUSS ist. Somit müssen alle Referenzen (in der Abbildung obere, rote Pfeile) gleichartige Objekte referenzieren, d. h., man benötigt eine Generalisierung (Verallgemeinerung) der Klassen KNOTEN und ABSCHLUSS, die Klasse LISTENELEMENT.



Klassendiagramm mit Vererbung

Die Klasse ABSCHLUSS hat die gleichen Methoden wie die Klasse KNOTEN, nur die interne Funktionalität der Methoden ist unterschiedlich. Somit werden die gemeinsamen Methoden in der Klasse LISTENELEMENT definiert und von den Unterklassen ABSCHLUSS und KNOTEN überdeckt. Die Klasse ABSCHLUSS benötigt keine Attribute. Da es nie Objekte der Klasse LISTENELEMENT geben wird, sind diese Klasse und alle ihre Methoden abstrakt.

Bei diesem Lösungsansatz handelt es sich um ein typisches Entwurfsmuster der objektorientierten Modellierung. Entwurfsmuster (engl. design pattern) sind bewährte Lösungsschablonen für wiederkehrende Entwurfsprobleme der Softwarearchitektur und Softwareentwicklung. Sie stellen damit eine wieder verwendbare Vorlage zur Problemlösung dar, die in einem spezifischen Kontext einsetzbar ist.

In diesem Fall handelt es sich um das Entwurfsmuster Kompositum. Durch Einführung der Klasse LISTENELEMENT können einfache Objekte der Klasse ABSCHLUSS genauso behandelt werden wie komplexe Behälterobjekte der Klasse KNOTEN (ein Knoten ‚beinhaltet' weitere Knoten (bzw. Listenelemente), die seine Nachfolger sind).

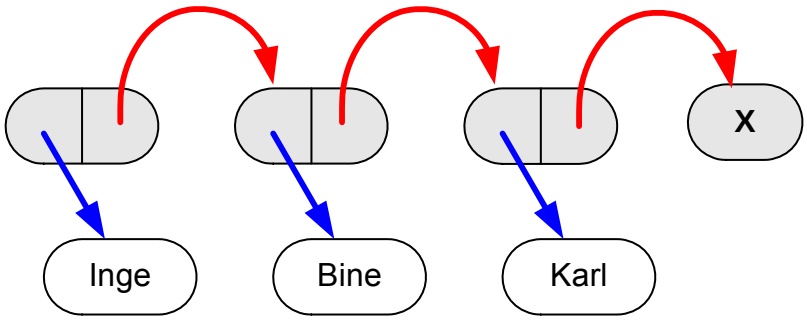
Das Kompositum (engl. Composite) ist ein Entwurfsmuster und gehört zu der Kategorie der Strukturmuster (Structural Patterns). Die Grundidee des Kompositummusters (Composite-Pattern) ist, in einer hierarchisch übergeordneten Klasse sowohl primitive Objekte als auch ihre Behälter zu repräsentieren. Das Composite-Muster erlaubt es, individuelle Objekte und Zusammensetzungen von Objekten auf gleiche Weise zu behandeln.

1.1.3.6.2 Überlegungen zu den Methoden

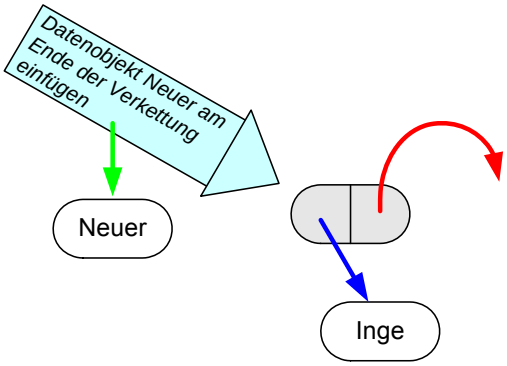
Die meisten Methoden der Klassen LISTE und KNOTEN bzw. ABSCHLUSS bleiben in ihrer Signatur (Methodenkopf) gleich, jedoch die Abläufe der Methoden vereinfachen sich wesentlich. Nur eine Methode bedarf einer näheren Betrachtung, die Methode *EndeEinfügen*(datenObjekt). Der letzte Knoten (Endknoten) der Liste unterscheidet sich jetzt nicht mehr von den anderen Knoten der Liste. Auch er gibt die Botschaft „Datenobjekt *Neuer* am Ende einfügen" ohne weitere Reaktion an den Abschluss weiter. Das eigentliche Einfügen bewerkstelligt jetzt der Abschluss. Durch das Einfügen des Datenobjekts *Neuer* ändert sich aber der Nachfolger des Endknotens. Dieser muss darüber informiert werden. Der Abschluss muss daher eine Referenz auf den neuen Knoten an den Aufrufer zurückgeben, sodass dieser seine Nachfolgerreferenz anpassen kann.

Methode EndeEinfuegen(datenObjekt) der Klasse KNOTEN

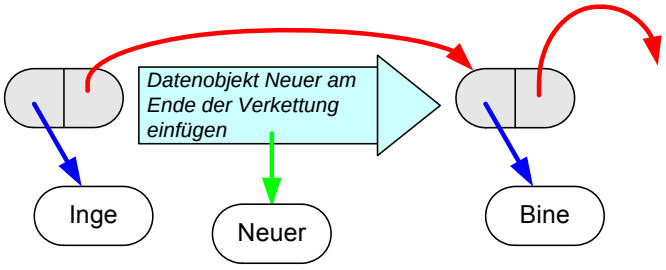
bzw. Methode EndeEinfuegen(datenObjekt) der Klasse ABSCHLUSS



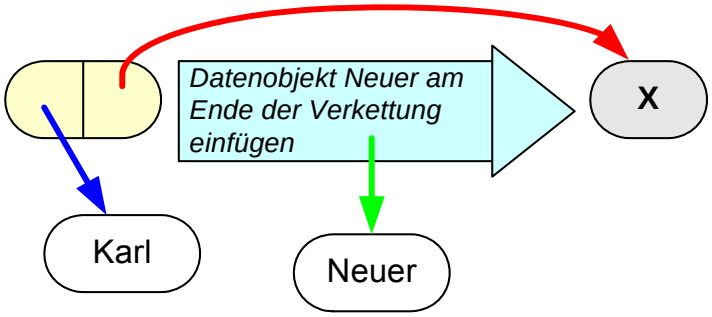
1) Ausgangssituation mit 3 Knotenobjekten und 1 Abschlussobjekt



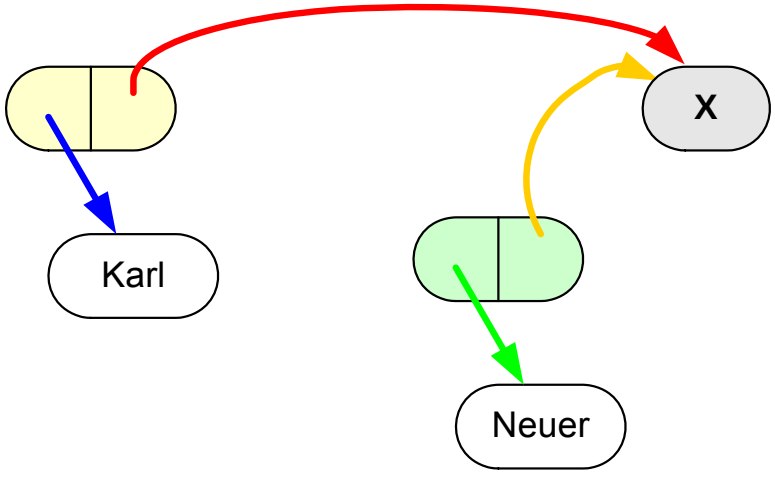
2) Durch Aufruf der Methode EndeEinfuegen(datenobjekt) wird eine Botschaft an ein Knotenobjekt geschickt, zusammen mit der Referenz auf das einzufügende Datenobjekt.



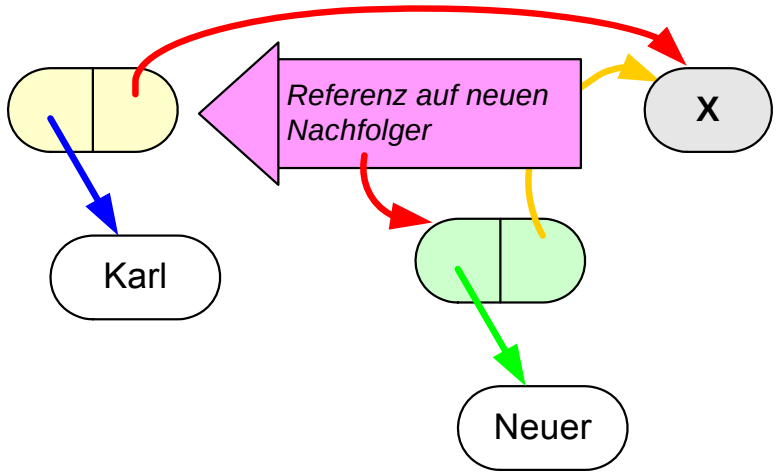
3a) Ist das Knotenobjekt nicht am Ende der Verkettung, so schickt dieses die Botschaft sofort, zusammen mit der Referenz auf das einzufügende Datenobjekt, an seinen Nachfolgeknoten weiter.



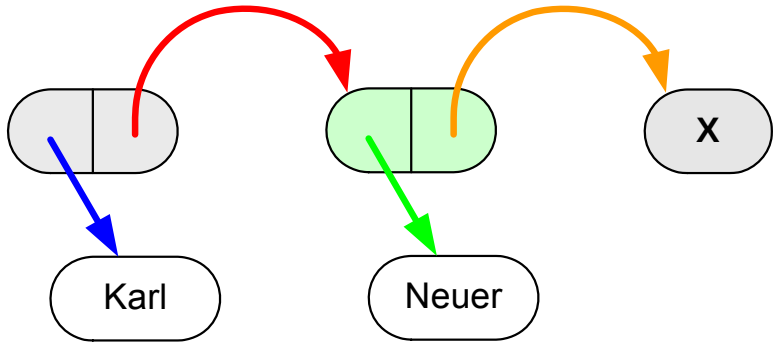
3b) Das Endeknotenobjekt verhält sich jetzt wie alle anderen Knotenobjekte, schickt die Botschaft sofort an das Abschlussobjekt weiter und wartet, dass vom Abschlussobjekt eine Referenz auf den eingefügten Knoten zurückgemeldet wird (deshalb gelbe Markierung des Knotens). Das Abschlussobjekt bekommt in der Methode EndeEinfuegen(datenobjekt) als Eingangswert eine Referenz (grüner Pfeil) auf das einzufügende Datenobjekt mitgeliefert.



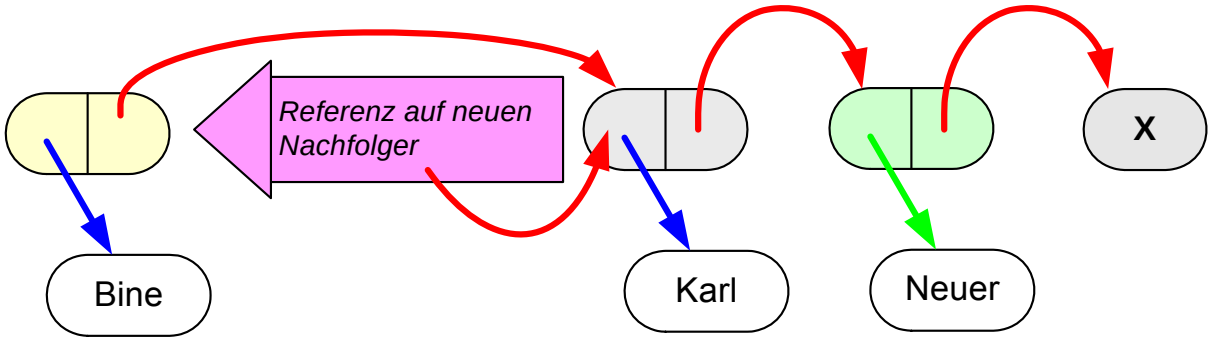
4) In der Methode EndeEinfuegen des Abschlussobjekts wird ein neues Knotenobjekt erzeugt (grüner Knoten; Knoten rechts). Dem Attribut nachfolger des neuen Kontenobjekts wird eine Referenz auf das Abschlussobjekt (oranger Pfeil; Pfeil Mitte rechts) zugewiesen, dem Attribut daten die Referenz auf das einzufügende Datenobjekt.



5) Das Abschlussobjekt gibt an seinen Aufrufer (der darauf wartet) eine Referenz auf den neu erzeugten Knoten zurück.



6) In der Methode EndeEinfuegen setzt der bisherige Endknoten als nachfolger die vom Abschlussknoten erhaltene Referenz auf den neuen Knoten. Der bisherige Endknoten kann seine Arbeit wieder fortsetzen.



7) Da die Signaturen der Methoden bei den Knotenobjekten und dem Abschlussknoten identisch sein sollen, hat auch die Methode EndeEinfuegen des Knotens einen Rückgabewert an seinen Aufrufer. Es ist auch hier eine Referenz auf den ‚neuen' Nachfolger, der aber in diesem Fall auch der ‚alte' Nachfolger ist. D. h., der Knoten gibt eine Referenz auf sich selbst zurück.

Es empfiehlt sich, dass die Schülerinnen und Schüler an dieser Stelle einen Vergleich zwischen dem ‚alten' *EndeEinfuegen*(datenObjekt) von Kapitel 1.1.3.3 und der neuen Variante durchführen.

Es ergibt sich folgende Änderung:

III) Weitere Methoden der Klasse KNOTEN:

- EndeEinfuegen(Referenz auf ein Datenobjekt) -> Referenz auf den sich evtl. geänderten Nachfolger (Objekt der Klasse KNOTEN bzw. ABSCHLUSS, d. h. allgemein der Klasse LISTENELEMENT).

  Diese Methode fügt am Ende der Verkettung einen neuen Knoten an, der das übergebene Datenobjekt verwaltet. An den Aufrufer wird eine Referenz auf seinen bei der Operation erzeugten neuen Nachfolger zurückgegeben; dies kann auch der alte Nachfolger sein, wenn sich an dieser Stelle nichts ändert.

1.1.3.6.3 Implementierung

Implementierung der Klasse LISTENELEMENT in Java:

```
abstract class LISTENELEMENT
{
   LISTENELEMENT() { }

   abstract LISTENELEMENT NachfolgerGeben();
   abstract DATENELEMENT DatenGeben();
   abstract DATENELEMENT EndeGeben();
   abstract LISTENELEMENT EndeEinfuegen(DATENELEMENT datenObjekt);
   abstract int AnzahlAbHierGeben();
   abstract void AlleAusgeben();
}
```

Für die Klasse KNOTEN ergeben sich Vereinfachungen, da jedes Mal die Überlegungen bezüglich des Endes der Liste entfallen können. Die Methode EndeEinfuegen wird entsprechend der obigen Überlegungen angepasst:

Anmerkung: Das Attribut *this* liefert eine Referenz auf das Objekt selbst.

```
class KNOTEN extends LISTENELEMENT
{
   private LISTENELEMENT nachfolger;
   private DATENELEMENT daten;

   // Konstruktoren
   KNOTEN()
   {
      super();
      nachfolger = null;
      daten = null;
   }

   KNOTEN(LISTENELEMENT naechsterKnoten, DATENELEMENT datenObjekt)
   {
      super();
      nachfolger = naechsterKnoten;
      daten = datenObjekt;
   }
```

```
  // Getter- und Setter-Methoden

  void NachfolgerSetzen(LISTENELEMENT neuerNachfolger)
  {
    nachfolger = neuerNachfolger;
  }

  LISTENELEMENT NachfolgerGeben()
  {
    return nachfolger;
  }

  void Datensetzen(DATENELEMENT datenObjekt)
  {
    daten = datenObjekt;
  }

  DATENELEMENT DatenGeben()
  {
    return daten;
  }

  // weitere Methoden

  DATENELEMENT EndeGeben()
  {
       DATENELEMENT wert;

       wert = nachfolger.EndeGeben();
         if (wert == null)
           return daten;
         else
           return wert;
  }

  LISTENELEMENT EndeEinfuegen(DATENELEMENT datenObjekt)
  {
       nachfolger = nachfolger.EndeEinfuegen(datenObjekt);
       return this;
  }

  int AnzahlAbHierGeben()
  {
       return nachfolger.AnzahlAbHierGeben() + 1;
  }

  void AlleAusgeben()
  {
       daten.Ausgeben();
       nachfolger.AlleAusgeben();
  }
}
```

```
class ABSCHLUSS extends LISTENELEMENT
{

   // KONSTRUKTOR
   ABSCHLUSS()
   {
        super();
   }
   // Getter- und Setter-Methoden

   LISTENELEMENT NachfolgerGeben()
   {
        return this;
   }

   DATENELEMENT DatenGeben()
   {
        return null;
   }


   // weitere Methoden

   DATENELEMENT EndeGeben()
   {
        return null;
   }

   LISTENELEMENT EndeEinfuegen(DATENELEMENT datenObjekt)
   {
        return new KNOTEN(this, datenObjekt);
   }

   int AnzahlAbHierGeben()
   {
        return 0;
   }

   void AlleAusgeben()
   {
   }

}
```

```
class LISTE
{
   private LISTENELEMENT anfang;

   // Konstruktor
   LISTE()
   {
        anfang = new ABSCHLUSS();
   }

   // weitere Methoden

   void EndeEinfuegen(DATENELEMENT neueDaten)
   {
        anfang = anfang.EndeEinfuegen(neueDaten);
   }
```

```
  void AnfangEinfuegen(DATENELEMENT neueDaten)
  {
       KNOTEN n = new KNOTEN(anfang, neueDaten);
       anfang = n;
  }

  DATENELEMENT AnfangEntfernen()
  {
       DATENELEMENT entfernt;

       entfernt = anfang.DatenGeben();
       anfang = anfang.NachfolgerGeben();

       return entfernt;
  }

  DATENELEMENT AnfangGeben()
  {
       return anfang.DatenGeben();
  }

  void AlleAusgeben()
  {
       anfang.AlleAusgeben();
  }

  int AnzahlGeben()
  {
       return anfang.AnzahlAbHierGeben();
  }

  boolean IstLeer()
  {
       return (anfang.AnzahlAbHierGeben() == 0);
  }

}
```

Der Test der Klasse LISTE in der neuen Form mit Abschluss ist identisch zu Kapitel 1.1.3.5. Die Schülerinnen und Schüler testen in einem kleinen Programm (bzw. in einer Testklasse mit einer Methode Ausfuehren()) die Richtigkeit der Implementierung und verwenden die Klasse LISTE als Schlange. Ausschnitt aus einem Programmcode:

```
LISTE warteSchlange = new LISTE();

warteSchlange.EndeEinfuegen(new KUNDE("Huber","Eduard"));
warteSchlange.EndeEinfuegen(new KUNDE("Meier","Paul"));
warteSchlange.EndeEinfuegen(new KUNDE("Müller","Inge"));
...
warteSchlange.AlleAusgeben();
warteSchlange.AnfangEntfernen();
warteSchlange.AlleAusgeben();
...
```

Eine Implementierung ist im BlueJ-Projekt ListeAbschluss zu finden.

1.1.3.7 Erweiterungen der Klasse ‚einfach verkettete Liste'

Lp: Realisierung der Methoden zum Einfügen, Suchen und Löschen

Lp: graphische Veranschaulichung der Methoden zum Einfügen (auch an beliebiger Stelle), Suchen und Löschen

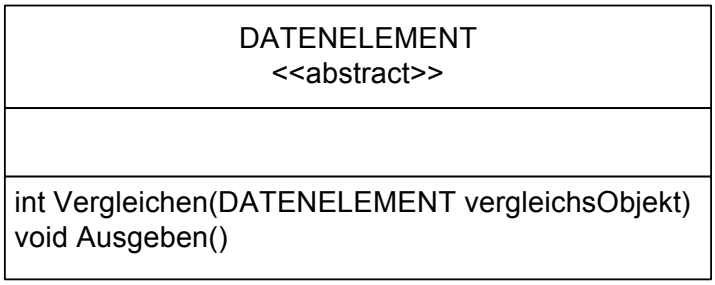
Lp: Einsatz der allgemeinen Datenstruktur Liste bei der Bearbeitung eines Beispiels aus der Praxis: Verwaltung von Elementen verschiedener Datentypen mittels Vererbung

1.1.3.7.1 Vergleich von Datenelementen

Oft ist es erforderlich, in einer Liste ein bestimmtes Datenelement aufzufinden und eine Referenz darauf auszugeben. Die Datenelemente müssen zu diesem Zweck eine Funktion bereitstellen, die es erlaubt, ein Datenelementobjekt mit einem anderen zu vergleichen. Im Hinblick auf spätere Funktionalitäten empfiehlt es sich, diesen Vergleich nicht nur für die Gleichheit bereitzustellen, sondern auch für „kleiner als" und „größer als". Welcher Attributwert bestimmt, ob ein Objekt kleiner oder größer ist als ein anderes Objekt, muss in der Klasse selbst festgelegt werden. Zum Beispiel ist bei der Klasse GANZZAHL ein Objekt mit zahl = 1 kleiner als Objekt mit zahl = 5. Bei Objekten der Klasse KUNDE kann diese Entscheidung über einen Stringvergleich der Werte im Attribut *name* erfolgen. Bei Namensgleichheit soll der Vorname entscheiden.

Das Attribut (bzw. die Attribute), das (die) die Reihenfolge bestimmt (bestimmen), wird (werden) auch als Schlüsselattribut (bzw. Schlüsselattribute) bezeichnet, da es (sie) das Objekt eindeutig kennzeichnet (kennzeichnen).

Die Klasse DATENELEMENT bzw. ihre Unterklassen werden hierzu um eine Methode Vergleichen(DATENELEMENT vergleichsObjekt) erweitert; diese liefert eine negative Zahl, wenn das aufrufende Objekt kleiner ist als das als Parameter übergebene Objekt.



Die Klasse DATENELEMENT

Ein Auszug aus einer Implementierung in Java:

```
abstract class DATENELEMENT
{
   abstract int Vergleichen(DATENELEMENT vergleichsObjekt);
   abstract void Ausgeben();
}
```

Definition der Klasse KUNDE

```
class KUNDE extends DATENELEMENT
{
   private String name;
   private String vorname;
```

```
  KUNDE(String neuerName, String neuerVorname)
  {
    super();
    name = neuerName;
    vorname = neuerVorname;
  }

  int Vergleichen(DATENELEMENT vergleichsObjekt)
  {
    KUNDE vergleichsKunde = (KUNDE) vergleichsObjekt;

    int ergebnis;
    ergebnis = name.compareTo(vergleichsKunde.name);
    if (ergebnis == 0)
      ergebnis = vorname.compareTo(vergleichsKunde.vorname);
    return ergebnis;
  }

  void Ausgeben()
  {
    System.out.println("Kunde: " + name + ", " + vorname);
  }

}
```

Zum Vergleichen erwartet ein Kundenobjekt auch wieder ein Kundenobjekt. Deshalb muss die Referenz vergleichsObjekt eine Referenz auf ein Objekt der Klasse KUNDE sein. Durch die Typanpassung (typecasting) wird erreicht, dass die Referenz auf DATENELEMENT als eine Referenz auf ein Objekt der Klasse KUNDE betrachtet wird.

Definition der Klasse KARTE

```
class KARTE extends DATENELEMENT
{
  private String rang;
  private int anzahl;
  private float preis;

  KARTE(String neuerRang, int neueAnzahl, float neuerPreis)
  {
    super();
    rang = neuerRang;
    anzahl = neueAnzahl;
    preis = neuerPreis;
  }

  int Vergleichen(DATENELEMENT vergleichsObjekt)
  {
    KARTE vergleichsKarte = (KARTE) vergleichsObjekt;
    return rang.compareTo(vergleichsKarte.rang);
  }

  void Ausgeben()
  {
    System.out.println("Karte: "+rang+", "+anzahl+", "+preis);
  }

}
```

Definition der Klasse GANZZAHL

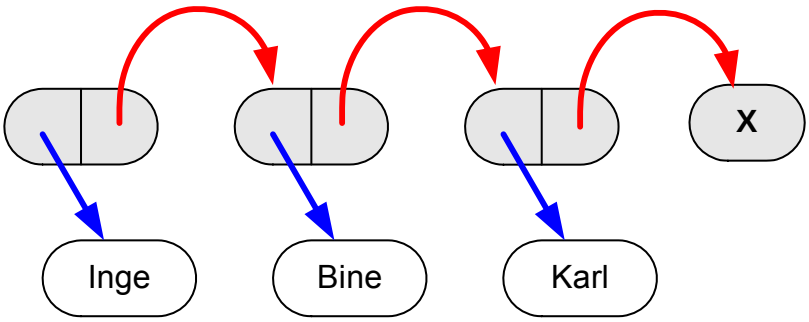
```
class GANZZAHL extends DATENELEMENT
{
  private int zahl;

  GANZZAHL(int neueZahl)
  {
    zahl = neueZahl;
  }

  int Vergleichen(DATENELEMENT vergleichsObjekt)
  {
    GANZZAHL vergleichsZahl = (GANZZAHL) vergleichsObjekt;
    int ergebnis = 0;

    if (zahl < vergleichsZahl.zahl)
      ergebnis = -1;
    if (zahl == vergleichsZahl.zahl)
      ergebnis = 0;
    if (zahl > vergleichsZahl.zahl)
      ergebnis = +1;
    return ergebnis;
  }

  void Ausgeben()
  {
    System.out.println("Zahl: " + zahl);
  }

}
```

1.1.3.7.2 Suchen

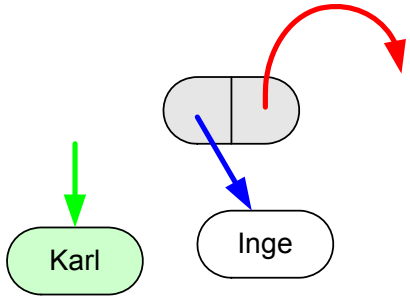
Der Verwalter der Warteschlange möchte wissen, ob eine bestimmte Person bereits in der Warteschlange ist. Hierzu verwendet er ein rekursives Vorgehen wie in Kapitel 1.1.2.2 beschrieben. Er schickt an die erste Person in der Warteschlange die Frage (Botschaft) „Sind Sie oder ein anderer in der Warteschlange die Person, die ich Ihnen auf den Zettel geschrieben habe?“ und drückt ihm einen Zettel mit den Kenndaten der gesuchten Person in die Hand. Der Erste in der Warteschlange kann die Frage nicht beantworten, schickt deshalb die Botschaft zusammen mit dem Zettel an seinen Hintermann und wartet auf die Antwort. Dieses Verfahren wird fortgeführt bis eine der Personen in der Warteschlange erkennt, dass er gesucht ist. Er kann die Frage beantworten und gibt seinem Aufrufer die Antwort „Ja, ist da“ zurück, dieser gibt die Antwort an seinen Aufrufer zurück und so weiter, bis die Antwort bei dem Verwalter der Schlange angekommen ist.

Die Methode Suchen benutzt ein rekursives Vorgehen. Die Suchanfrage läuft die verkettete Liste solange entlang, bis das gesuchte Datenelement gefunden wurde oder das Ende erreicht ist. Das Suchergebnis läuft dann wieder die Liste zurück bis zum Auslöser der Methode.

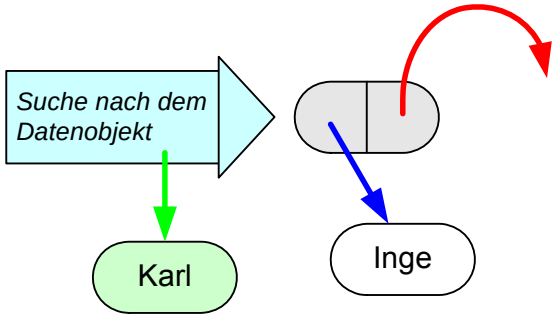
Methode Suchen(datenobjekt) der Klasse KNOTEN



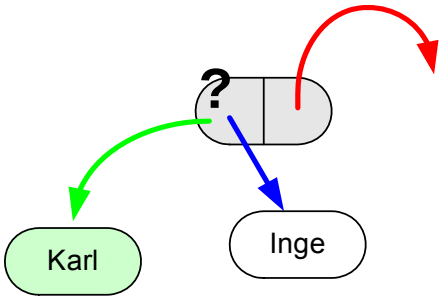
1) Ausgangssituation mit 3 Knotenobjekten und 1 Abschlussobjekt



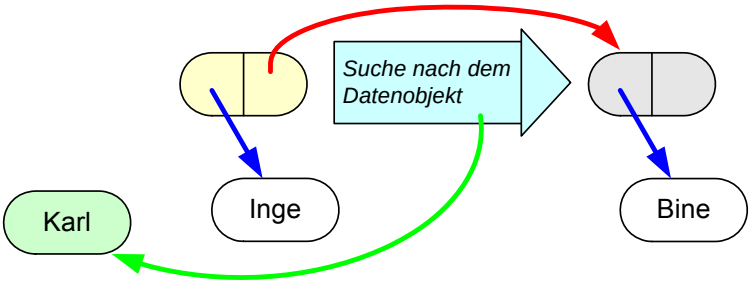
2a) Zuerst muss ein Datenobjekt von der in der Liste verwalteten Datenklasse erzeugt werden und mit den gesuchten Attributwerten belegt werden (hier grünes Objekt Karl).



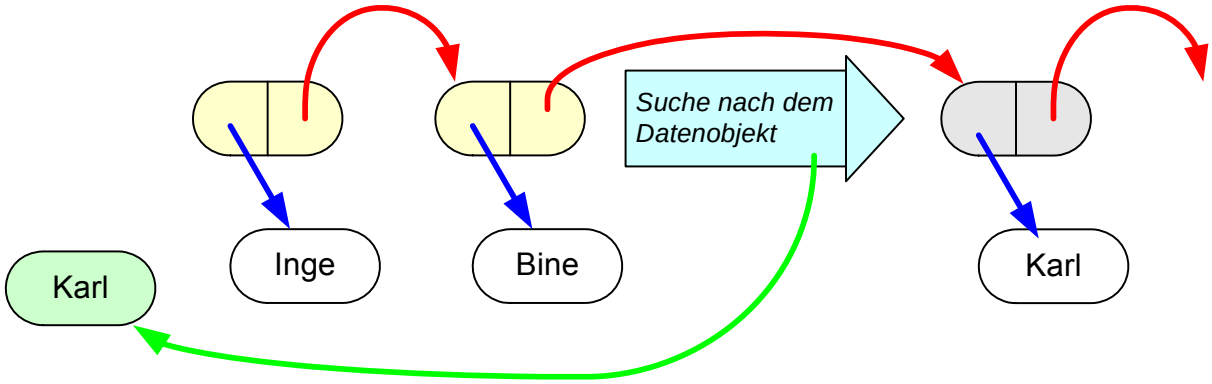
2b) Durch Aufruf der Methode Suchen(datenobjekt) wird eine Botschaft an ein Knotenobjekt geschickt, zusammen mit der Referenz auf das zu suchende Datenobjekt.



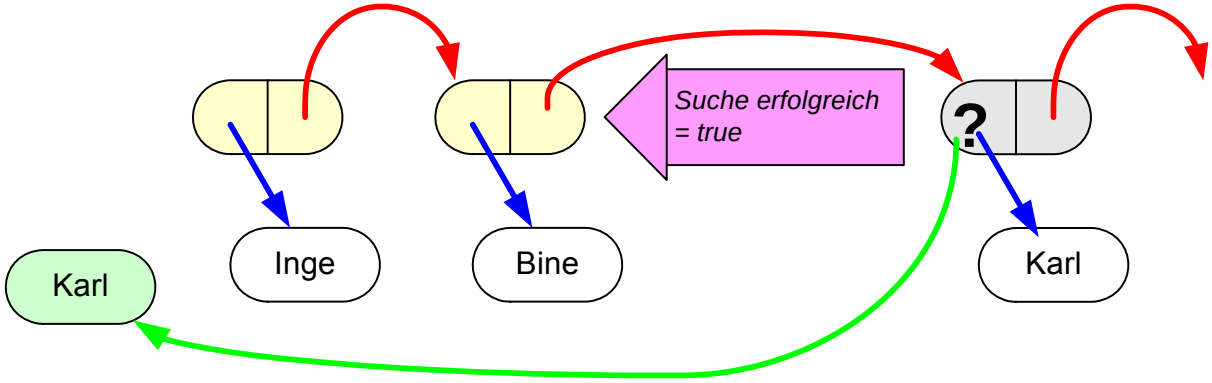
3a) Das Knotenobjekt vergleicht sein Datenelement mit dem Datenelement, das ihm als Parameter übergeben wurde. Hierzu verwendet es die Methode Vergleichen() seines Datenobjekts.



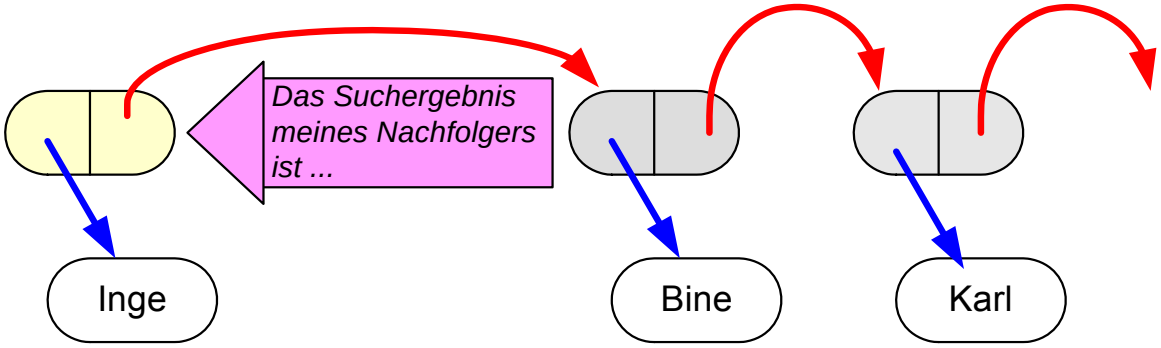
3b) Sind die Datenobjekte nicht gleich, so schickt der Knoten durch Aufruf der Methode Suchen(datenobjekt) eine Botschaft an seinen Nachfolger, zusammen mit der Referenz auf das zu suchende Datenobjekt. Der Knoten selbst muss auf die Antwort der Suchbotschaft warten (gelber Knoten, Knoten links)



4) Die Botschaft zum Suchen wird durch die verkettete Liste weiter geschickt bis Situation 5 eintritt.



5) Das Knotenobjekt (Knoten rechts, grau) vergleicht sein Datenelement mit dem Datenelement, das ihm als Parameter übergeben wurde. Hierzu verwendet es die Methode Vergleichen() seines Datenobjekts. Wenn der Vergleich als Ergebnis die Gleichheit der beiden Datenobjekte liefert, dann gibt er an seinen Aufrufer die Antwort „Suche erfolgreich“, d. h. den Wert *true*, zurück.



6) Der Knoten (hier „Bineknoten“) kann weiter arbeiten und schickt sofort die Botschaft „Das Suchergebnis meines Nachfolgers ist ...“ an seinen Aufrufer, indem er den entsprechenden Booleanwert zurückgibt. Dies geht so weiter, bis der erste Knoten das Suchergebnis an den Aufrufer des Suchprozesses zurückgeben kann.

Die Implementation des Suchens erfordert einige Erweiterungen bei den bestehenden Klassen, die im Folgenden auszugsweise angegeben sind.

Auszug aus einer Implementierung der Klasse LISTENELEMENT in Java:

```
abstract class LISTENELEMENT
{
  ...
  abstract boolean Suchen(DATENELEMENT datenObjekt);
  ...
}
```

Erweiterung der Klasse KNOTEN:

```
class KNOTEN extends LISTENELEMENT
{
  ...

  boolean Suchen(DATENELEMENT datenObjekt)
  {
  if (daten.Vergleichen(datenObjekt) == 0)
    return true;
  else
    return nachfolger.Suchen(datenObjekt);
  }

  ...
}
```

Erweiterung der Klasse ABSCHLUSS:

```
class ABSCHLUSS extends LISTENELEMENT
{
  ...

  boolean Suchen(DATENELEMENT datenObjekt)
  {
    return false;
  }

  ...
}
```

Erweiterung der Klasse LISTE:

```
class LISTE
{
  ...

  boolean Suchen(DATENELEMENT datenObjekt)
  {
    return anfang.Suchen(datenObjekt);
  }

  ...
}
```

Der Test der Klasse LISTE und der Methode Suchen verläuft identisch zu Kapitel 1.1.3.5. Die Schülerinnen und Schüler testen in einem kleinen Programm (bzw. in einer Testklasse mit einer Methode Ausfuehren()) die Implementierung und verwenden die Klasse LISTE.

Ausschnitt aus einem Programmcode:

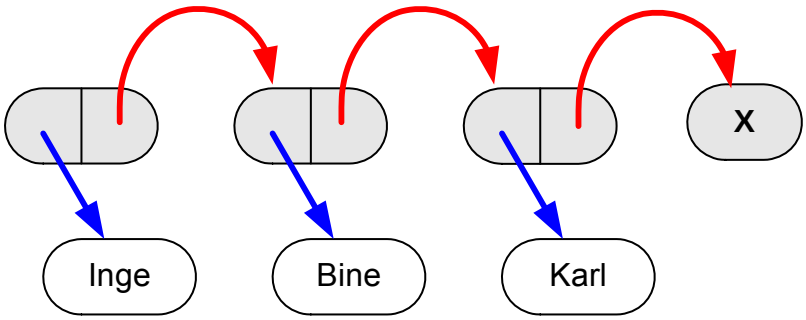
```
LISTE testListe = new LISTE();

testListe.EndeEinfuegen(new KUNDE("Huber","Eduard"));
testListe.EndeEinfuegen(new KUNDE("Meier","Paul"));
testListe.EndeEinfuegen(new KUNDE("Müller","Inge"));
...
testListe.AlleAusgeben();

KUNDE suchKunde = new KUNDE("Meier", "Paul");
System.out.println(testListe.Suchen(suchKunde));

testListe.AnfangEntfernen();
testListe.AnfangEntfernen();
testListe.AlleAusgeben();

System.out.println(testListe.Suchen(suchKunde));
...
```

1.1.3.7.3 Entfernen

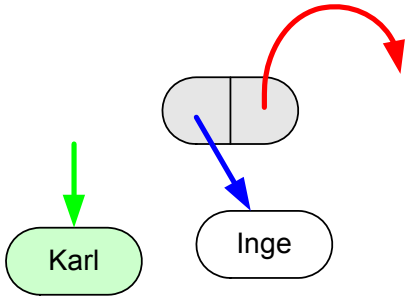
Im nächsten Schritt soll die bisherige Implementierung der Liste um die Fähigkeit erweitert werden, einen Knoten (der auf ein bestimmtes Datenelement verweist) aus der Verkettung zu entfernen.

Die Methode Entfernen hat sehr viel Ähnlichkeit mit der Methode Suchen. Einzig, wenn der Knoten mit dem Datenelement gefunden wurde, dann muss sich dieser aus der verketteten Liste „ausklinken“. Die Methode Entfernen benutzt ebenfalls ein rekursives Vorgehen. Die Suchanfrage nach dem zu entfernenden Datenelement läuft die verkettete Liste solange entlang, bis das gesuchte Datenelement gefunden wurde oder das Ende erreicht ist. Wenn es gefunden wurde, hängt sich der zugehörige Knoten aus der Verkettung aus, in dem er seinem Vorgänger seine eigene Nachfolgerreferenz als neue Nachfolgerreferenz übergibt, er wird damit in der Verkettung übersprungen.

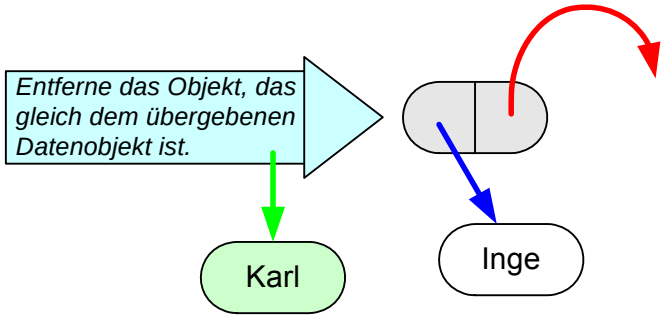
Methode Entfernen(datenobjekt) der Klasse KNOTEN



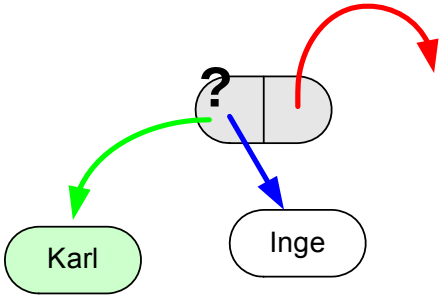
1) Ausgangssituation mit 3 Knotenobjekten und 1 Abschlussobjekt



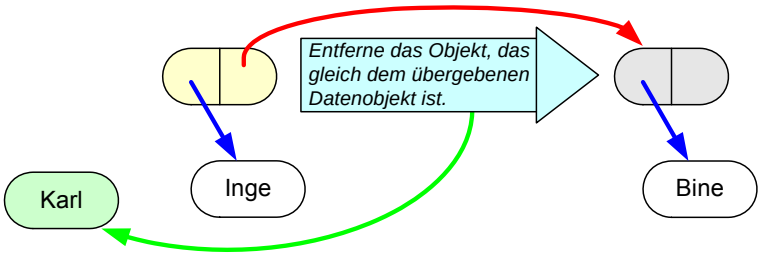
2a) Zuerst muss ein Datenobjekt von der in der Liste verwalteten Datenklasse erzeugt werden und mit den gesuchten Attributwerten belegt werden (hier Objekt Karl, grün). Das dazu „gleiche“ Datenobjekt soll dann in der Liste entfernt werden.



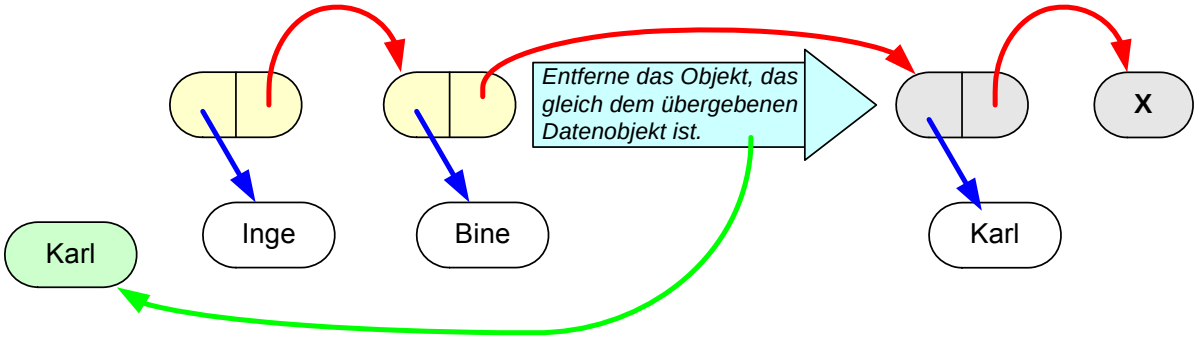
2b) Durch Aufruf der Methode Entfernen(datenobjekt) wird die Botschaft „Entferne das Objekt aus der Liste, das gleich dem übergebenen Datenobjekt ist.“ an ein Knotenobjekt geschickt, zusammen mit der Referenz auf das zu suchende/entfernende Datenobjekt.



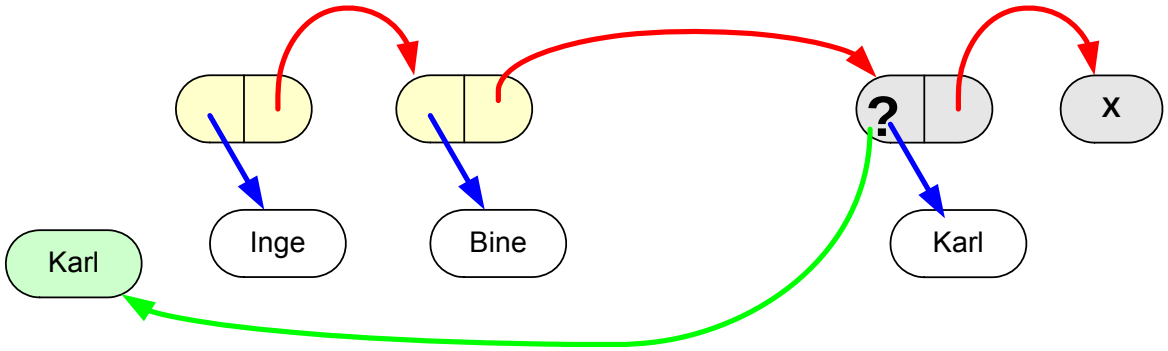
3a) Das Knotenobjekt vergleicht sein Datenelement mit dem Datenelement, das ihm als Parameter übergeben wurde. Hierzu verwendet es die Methode Vergleichen() seines Datenobjekts.



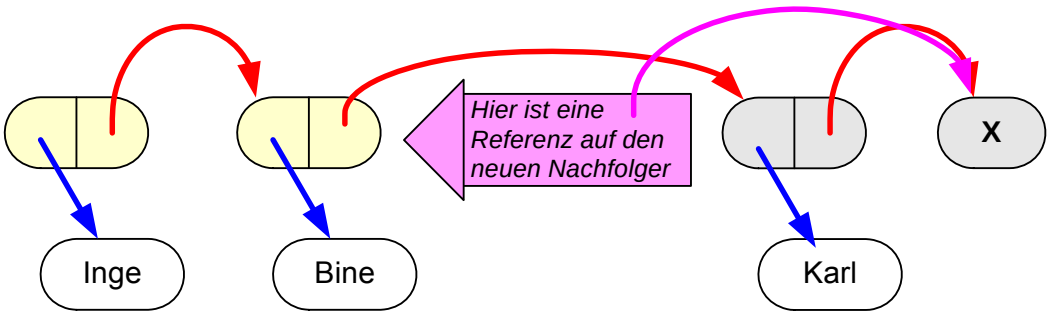
3b) Sind die Datenobjekte nicht gleich, schickt der Knoten durch Aufruf der Methode Entfernen(datenobjekt) eine Botschaft an seinen Nachfolger, zusammen mit der Referenz auf das zu suchende/entfernende Datenobjekt. Der Knoten selbst muss auf die Antwort der Entfernensbotschaft warten (gelber Knoten, Knoten links).



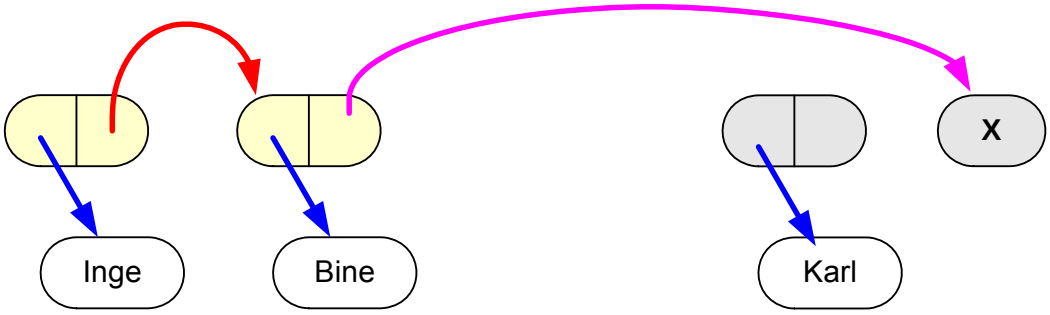
4) Die Botschaft zum Entfernen wird durch die verkettete Liste weiter geschickt, bis Situation 5 eintritt oder das Abschlusselement erreicht ist.



5) Das Knotenobjekt (hier Karl grau) vergleicht sein Datenelement mit dem Datenelement, das im als Parameter übergeben wurde. Wenn der Vergleich als Ergebnis die Gleichheit der beiden Datenobjekte liefert, dann erkennt der Knoten, dass er sich entfernen muss.

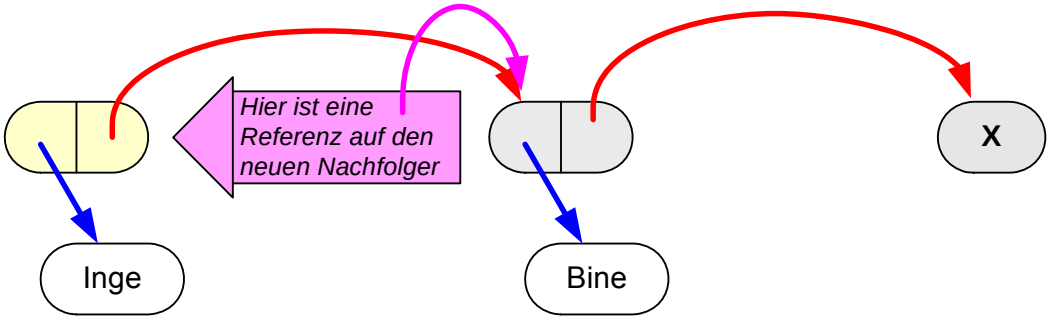


6) Zum Entfernen gibt er an seinen Aufrufer die Antwort „Hier ist eine Referenz auf deinen neuen Nachfolger“ zurück, zusammen mit einer Referenz (lila Pfeil) auf den Nachfolger des zu entfernenden Knotens, also auf sich selbst.



7) Der Empfänger der Antwort setzt sein Attribut nachfolger auf die empfangene Referenz. Dadurch ist das Knotenobjekt für Karl aus der Liste ausgeklinkt.

Noch ist der Knoten mit den Daten ‚Karl' nicht ganz gelöscht. Das besorgt in Java ein Systemprozess, die Garbage Collection (Mülleinsammlung). Der Knoten wird von keinem anderen Objekt gebraucht (referenziert). Beim nächsten ‚Aufräumen' des Systems wird er deshalb automatisch aus dem Speicher entfernt.



8) Der Knoten (hier mit Daten Bine) kann weiter arbeiten und schickt sofort die Antwort „Hier ist eine Referenz auf deinen neuen Nachfolger" zusammen mit einer Referenz (lila Pfeil) auf sich selbst zurück. Der empfangende Knoten der Antwort setzt sein Attribut nachfolger auf die empfangene Referenz. Auch wenn sich in diesem Fall nichts ändert, so wird dasselbe Verfahren angewendet (vgl. auch 1.1.3.6.2).

Dieses Zurücklaufen geht weiter, bis der erste Knoten seine Referenz an den Aufrufer zurückgeben kann.

Die Implementation des Entfernens erfordert einige Erweiterungen bei den bestehenden Klassen. Auszüge daraus:

Implementierung der Klasse LISTENELEMENT in Java:

```
abstract class LISTENELEMENT
{
   ...
   abstract LISTENELEMENT Entfernen(DATENELEMENT datenObjekt);
   ...
}
```

Erweiterung der Klasse KNOTEN:

```
class KNOTEN extends LISTENELEMENT
{
   ...

   LISTENELEMENT Entfernen(DATENELEMENT datenObjekt)
   {
      if (daten.Vergleichen(datenObjekt) == 0)
         return nachfolger;
      else
         {
            nachfolger = nachfolger.Entfernen(datenObjekt);
            return this;
         }
     }
   ...
}
```

Erweiterung der Klasse ABSCHLUSS:

```
class ABSCHLUSS extends LISTENELEMENT
{
  ...

  LISTENELEMENT Entfernen(DATENELEMENT datenObjekt)
  {
    return this;
  }
  ...
}
```

Erweiterung der Klasse LISTE:

```
class LISTE
{
  ...

  void Entfernen(DATENELEMENT datenObjekt)
  {
    anfang = anfang.Entfernen(datenObjekt);
  }
  ...
}
```

Der Test der Klasse LISTE und der Methode Entfernen verläuft identisch zu Kapitel 1.1.3.5. Die Schülerinnen und Schüler testen in einem kleinen Programm (bzw. in einer Testklasse mit einer Methode Ausfuehren()) die Richtigkeit der Implementierung. Ausschnitt aus einem Programmcode:

```
LISTE testListe = new LISTE();

testListe.EndeEinfuegen(new KUNDE("Huber","Eduard"));
testListe.EndeEinfuegen(new KUNDE("Meier","Paul"));
testListe.EndeEinfuegen(new KUNDE("Müller","Inge"));
testListe.EndeEinfuegen(new KUNDE("Neulinger","Werner"));
testListe.EndeEinfuegen(new KUNDE("Paulinger","Hilde"));

System.out.println("Alle");
testListe.AlleAusgeben();
System.out.println(testListe.AnzahlGeben());

KUNDE entfernKunde = new KUNDE("Neulinger", "Werner");
testListe.Entfernen(entfernKunde);
System.out.println("Alle nach Entfernen");
testListe.AlleAusgeben();
System.out.println(testListe.AnzahlGeben());

...
```

1.1.3.7.4 Sortiert Einfügen

Bisher können die Objekte nur am Ende oder am Anfang der Liste hinzugefügt werden. Die Reihenfolge der Objekte in der Liste entspricht der Reihenfolge, in der die Objekte erzeugt werden. Von Interesse wäre sicher ein Einfügen entsprechend eines Sortierkriteriums. Benutzt man dieses „sortierte Einfügen“ für alle Einfügevorgänge, ausgehend von einer leeren Liste,

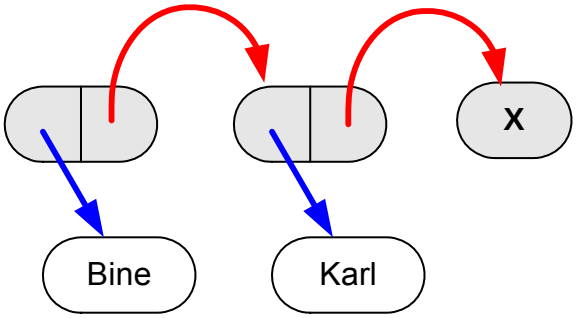
so erhält man eine sortierte Liste. Das Sortierkriterium „kleiner“, „größer“ bzw. „gleich“ legen die verwalteten Datenobjekte selbst fest (siehe hierzu Abschnitt 1.1.3.7.1). Im Folgenden wird in aufsteigender Reihenfolge einsortiert.

Auch hier kann rekursiv vorgegangen werden. Man übergibt einem Knoten ein Datenobjekt, das eingefügt werden soll. Der Knoten prüft, ob das neue Datenobjekt größer ist als sein Datenobjekt. Wenn das der Fall ist, gibt er das neue Datenobjekt an seinen Nachfolger zur Überprüfung. Stellt ein Knoten fest, dass das neue Datenobjekt kleiner ist als sein verwaltetes Datenobjekt, dann fügt er vor sich einen neuen Knoten ein, der das neue Datenobjekt referenziert.

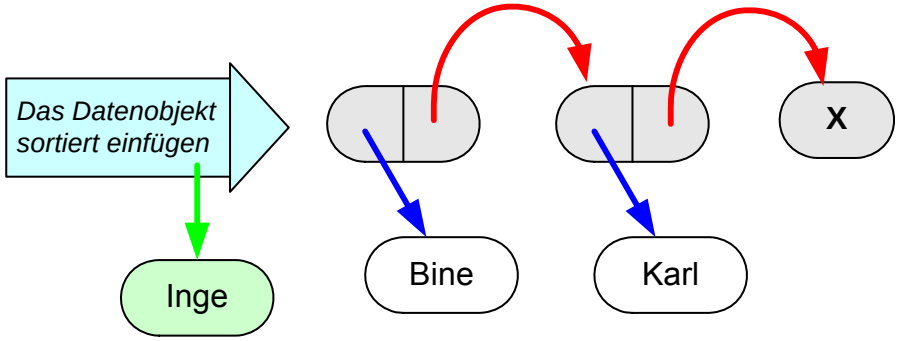
Die Schülerinnen und Schüler können diesen Algorithmus gut in einem Spiel analysieren und erproben. Eine Schülerin oder ein Schüler verwaltet das Attribut *anfang* der Liste und eine Schülerin oder ein Schüler ist ein Objekt der Klasse ABSCHLUSS, das nach Instanziierung der Liste immer vorhanden ist. Jetzt wird Schüler(in) für Schüler(in) sortiert eingefügt. Hierzu übergibt man den(die) Schüler(in) an das Objekt, das vom *anfang* referenziert wird. Der (Die) Schüler(in) wird in der Verkettung weitergereicht, bis er(sie) passend einsortiert werden kann. Die Verkettung der Liste wird durch das Auflegen der Hand auf die Schulter des jeweiligen Nachfolgers realisiert.

Die Erklärung der Methode SortiertEinfuegen kann jetzt knapper gefasst werden, denn sie hat große Ähnlichkeit mit den anderen Abläufen, vor allem mit EndeEinfuegen von Kapitel 1.1.3.6.2.

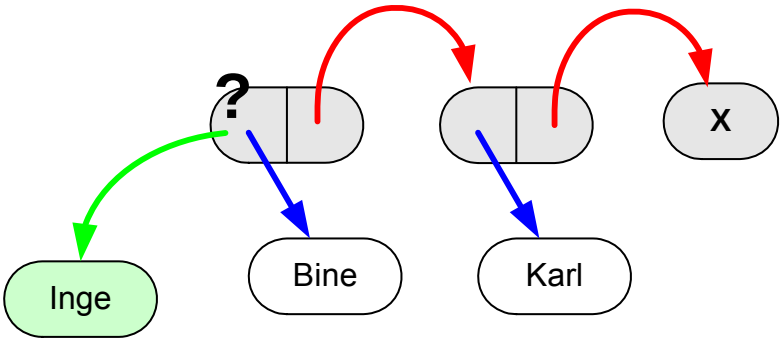
Methode SortiertEinfuegen(datenObjekt) der Klasse KNOTEN



1) Ausgangssituation mit 2 Knotenobjekten und 1 Abschlussobjekt

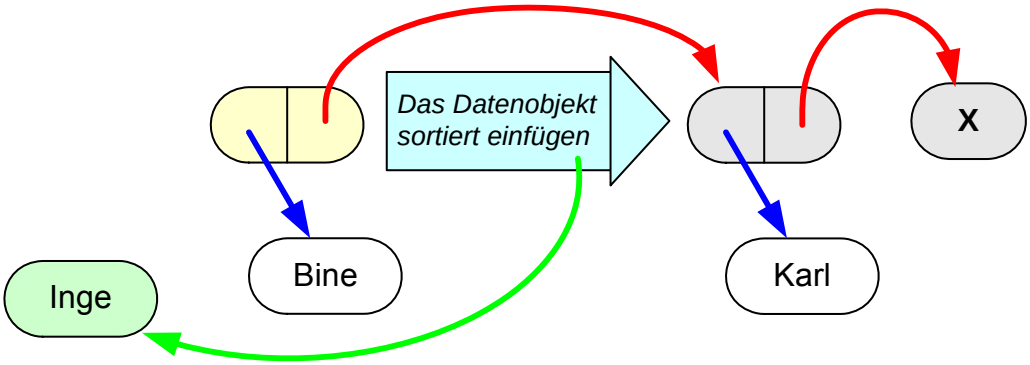


2) Durch Aufruf der Methode SortiertEinfuegen(datenObjekt) wird eine Botschaft an ein Knotenobjekt geschickt, zusammen mit der Referenz auf das einzufügende Datenobjekt.

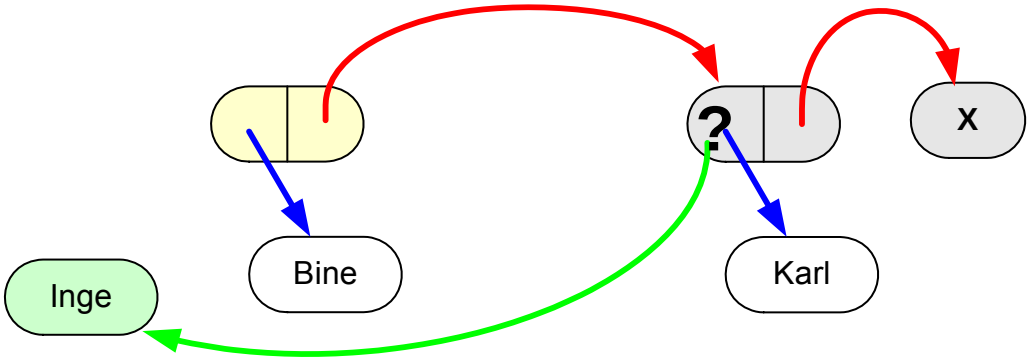


3) Das angesprochene Knotenobjekt vergleicht sein Datenelement mit dem Datenelement, das ihm als Parameter übergeben wurde. Hierzu verwendet es die Methode Vergleichen() seines Datenobjekts.

Wenn der Vergleich ergibt, dass sein Datenelement kleiner ist, als das im Parameter übergebene, dann wird fortgefahren bei 4 („gehört irgendwo hinter mir eingefügt“), sonst wird fortgefahren bei 6 („gehört vor mir einsortiert“). In diesem Beispiel weiter bei 4.

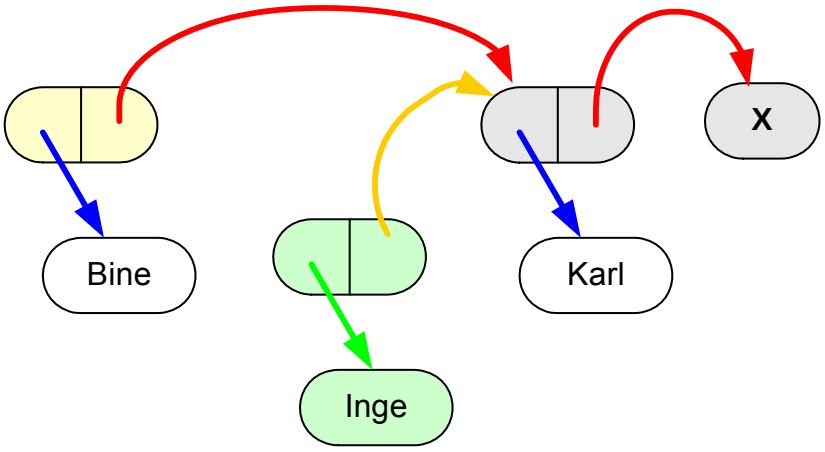


4) Durch Aufruf der Methode SortiertEinfuegen seines Nachfolgers schickt der Knoten die Botschaft sofort an das nächste Knotenobjekt weiter und wartet, dass von diesem eine Referenz auf den eingefügten Knoten zurückgemeldet wird (gelber Knoten, Knoten links). Der Nachfolgeknoten bekommt beim Aufruf der Methode SortiertEinfuegen eine Referenz (grüner Pfeil, Pfeil Mitte links) auf das einzufügende Datenobjekt mitgeliefert.

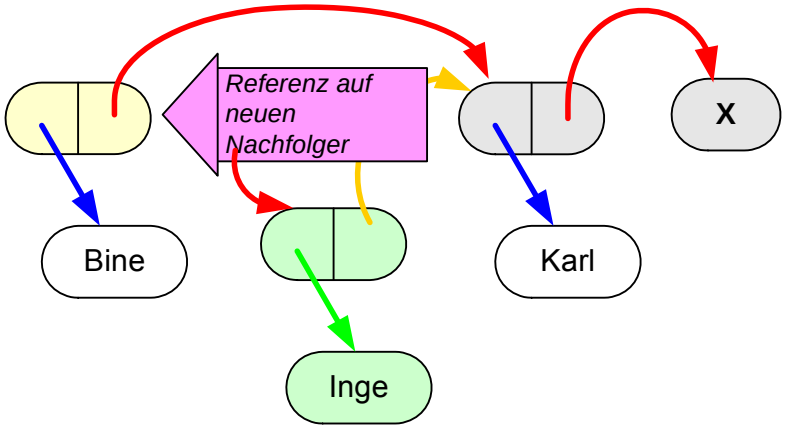


5) Das angesprochene Knotenobjekt vergleicht sein Datenelement mit dem Datenelement, das ihm als Parameter übergeben wurde.

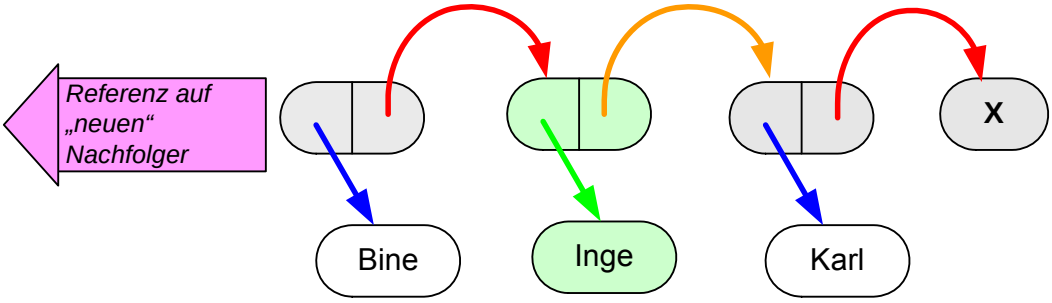
Wenn der Vergleich ergibt, dass sein Datenelement kleiner ist, als das im Parameter übergebene, dann weiter bei 4 („gehört irgendwo dahinter eingefügt“), sonst weiter bei 6 („gehört vor mir einsortiert“). In diesem Beispiel weiter bei 6.



6) In der Methode SortiertEinfuegen des aktiven Knotenobjekts (von Karl) wird ein neues Knotenobjekt erzeugt (grüner Knoten). Dem Attribut *nachfolger* des neuen Kontenobjekts wird eine Referenz auf das aktive Knotenobjekt (oranger Pfeil) zugewiesen, dem Attribut *daten* die Referenz auf das einzufügende Datenobjekt.



7) Das aktive Knotenobjekt gibt an seinen Aufrufer, der darauf wartet, eine Referenz auf den neu erzeugten Knoten zurück.



8) In der Methode SortiertEinfuegen setzt das Knotenobjekt (von Bine) als *nachfolger* die erhaltene Referenz auf den neuen Knoten.

Der Knoten von Bine sendet dann an seinen Aufrufer eine Referenz auf sich, sodass er weiterhin sein Nachfolger bleibt.

Die Implementation des sortierten Einfügens erfordert einige Erweiterungen bei den bestehenden Klassen.

Entsprechende Auszüge:

Implementierung der Klasse LISTENELEMENT in Java:

```
abstract class LISTENELEMENT
{
   ...
   abstract LISTENELEMENT SortiertEinfuegen(DATENELEMENT datenObjekt);
   ...
}
```

Erweiterung der Klasse KNOTEN:

```
class KNOTEN extends LISTENELEMENT
{
   ...
   LISTENELEMENT SortiertEinfuegen(DATENELEMENT datenObjekt)
   {
      // meine Daten sind kleiner
      if (daten.Vergleichen(datenObjekt) < 0)
      {
         nachfolger = nachfolger.SortiertEinfuegen(datenObjekt);
         return this;
      }
      // meine Daten sind größer oder gleich
      else
         return new KNOTEN(this, datenObjekt);
   }
   ...
}
```

Erweiterung der Klasse ABSCHLUSS:

```
class ABSCHLUSS extends LISTENELEMENT
{
   ...

   LISTENELEMENT SortiertEinfuegen(DATENELEMENT datenObjekt)
   {
      return new KNOTEN(this, datenObjekt);
   }
   ...
}
```

Erweiterung der Klasse LISTE:

```
class LISTE
{
   ...

   void SortiertEinfuegen(DATENELEMENT neueDaten)
   {
      anfang = anfang.SortiertEinfuegen(neueDaten);
   }
   ...
}
```

Der Test der Klasse LISTE und der Methode SortiertEinfuegen verläuft identisch zu Kapitel 1.1.3.5. Die Schülerinnen und Schüler testen in einem kleinen Programm (bzw. in einer Testklasse mit einer Methode Ausfuehren()) die Richtigkeit der Implementierung.

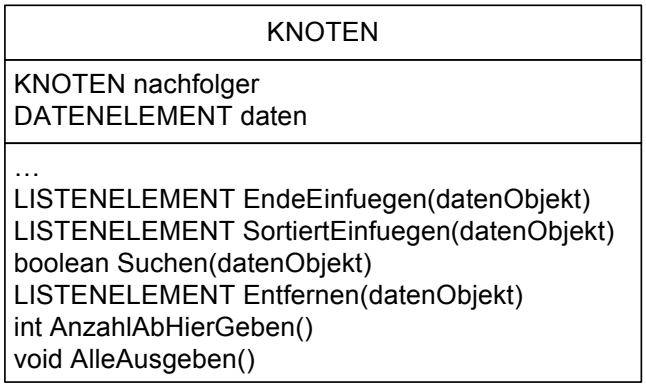
Ausschnitt aus einem Programmcode:

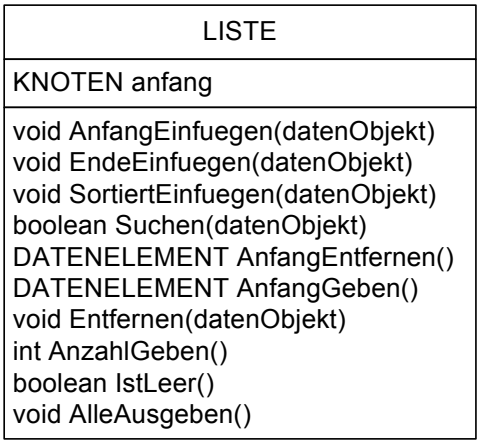
```
LISTE testListe = new LISTE();

testListe.SortiertEinfuegen(new KUNDE("Paulinger","Hilde"));
testListe.SortiertEinfuegen(new KUNDE("Müller","Inge"));
testListe.SortiertEinfuegen(new KUNDE("Huber","Eduard"));
testListe.SortiertEinfuegen(new KUNDE("Meier","Paul"));
testListe.SortiertEinfuegen(new KUNDE("Neulinger","Werner"));
testListe.SortiertEinfuegen(new KUNDE("Adam","Eva"));

System.out.println("Alle");
testListe.AlleAusgeben();
System.out.println(testListe.AnzahlGeben());

testListe.SortiertEinfuegen(new KUNDE("Zander","Xaxer"));
testListe.SortiertEinfuegen(new KUNDE("Boblinger","Isolde"));

System.out.println("Alle");
testListe.AlleAusgeben();
System.out.println(testListe.AnzahlGeben());

...
```

Eine Implementierung ist im BlueJ-Projekt ListeAbschlussErweiterung zu finden.

1.1.3.7.5 Definition und Zusammenfassung

Für die Klassen KNOTEN und LISTE wurden in den vorherigen Kapiteln einige Methoden entwickelt. Die Klasse ABSCHLUSS hat dieselben Methoden wie die Klasse KNOTEN jedoch keine Attribute. Es ergeben sich in der Zusammenschau folgende Klassendiagramme:





Definition:

Verkettete Listen gehören zu den dynamischen Datenstrukturen, die eine Speicherung von einer im Vorhinein nicht bestimmten Anzahl von miteinander in Beziehung stehenden Werten einfacher oder zusammengesetzter Datentypen erlauben. Sie werden durch Referenzen auf die jeweils folgende(n) Knoten realisiert.

Ein **Knoten** (engl. node) bezeichnet ein Element der Liste, welches eine Referenz auf die Daten und eine Referenz auf seinen Nachfolger enthält.

Die einfach verkettete Liste ist die fundamentale Form der verketteten Liste, da sie neben den einzelnen Knoten nur eine Referenz auf den ersten Knoten der Liste besitzt.

Eine Reihe weiterer nützlicher Methoden können durch die Schülerinnen und Schüler bei Bedarf ergänzt werden: Leeren der Liste; Ersetzen eines Datenelements durch ein anderes; Referenz auf das n-te Datenelement in der Liste geben; Position eines Datenelements suchen; usw.

Die meisten Programmiersprachen bieten schon fertige Implementierungen von Listen in Code-Bibliotheken an. Interessierte Schülerinnen und Schüler können sich, anhand der Dokumentation, an die Verwendung der fertigen Klassen wagen. In Java ist dies AbstractList bzw. Interface List.

1.1.4 Stapel und Schlange als spezielle Formen der Liste

Lp: Stapel und Schlange als spezielle Formen der allgemeinen Datenstruktur Liste

1.1.4.1 Stapel

Ein anschauliches Beispiel für einen Stapel ist ein Tellerstapel bei einem Buffet. Die Bedienungen bringen immer wieder frische Teller und legen diese oben auf den Stapel. Die Speisenden entnehmen von oben einen Teller. Das heißt, der als letzter auf dem Tellerstapel abgelegte Teller wird als erster wieder entnommen.

Container in einem Containerschiff werden ebenfalls in Stapeln gelagert. Zugriff hat man nur auf den obersten Container und neue Container können nur oben auf den Stapel gelegt werden. Läuft ein Containerschiff mit seiner Ladung mehrere Auslieferhäfen an, so ist es wichtig, dass die Container in der richtigen Reihenfolge im Stapel abgelegt sind. Ein Zugriff auf einen, aus Versehen zu tief eingeordneten Container erfordert die Entnahme aller darüber abgelegten Container.

Ein Stapel (Keller; engl. stack) ist eine dynamische Datenstruktur, die das Einfügen und Entfernen von Elementen nach dem LIFO-Prinzip (last-in, first-out) unterstützt. Das zuletzt hinzugefügte Element wird als erstes entnommen. Das Einfügen und die Entnahme von Elementen finden immer nur an einem Ende statt. Ein Zugriff auf innere Elemente ist nicht möglich. Obwohl ein Stapel nur eingeschränkten Zugriff auf die gespeicherten Daten ermöglicht, jedoch ist diese Art des Zugriffs für viele Vorgänge ausreichend und in machen Fällen sogar das schnellste Vorgehen.

In der Klasse STAPEL sind die Methoden Einfügen und Entfernen erforderlich.

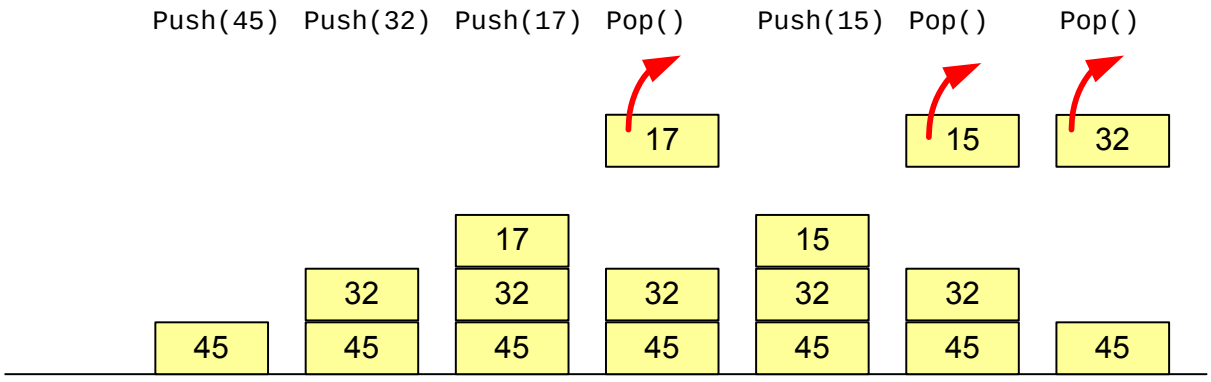
I) Erforderliche Methoden der Klasse STAPEL:

- STAPEL()
  Anlegen eines Stapel-Objekts (Konstruktor)
- Einfuegen(Referenz auf Datenelement)
  Legt die Referenz auf das Datenelement auf dem Stapel ab (engl. push)
- Entfernen() -> Referenz auf oberste Datenelement
  Das oberste (zuletzt abgelegt) Datenelement wird vom Stapel entfernt; die Funktion gibt eine Referenz auf dieses Datenelement zurück (engl. pop)

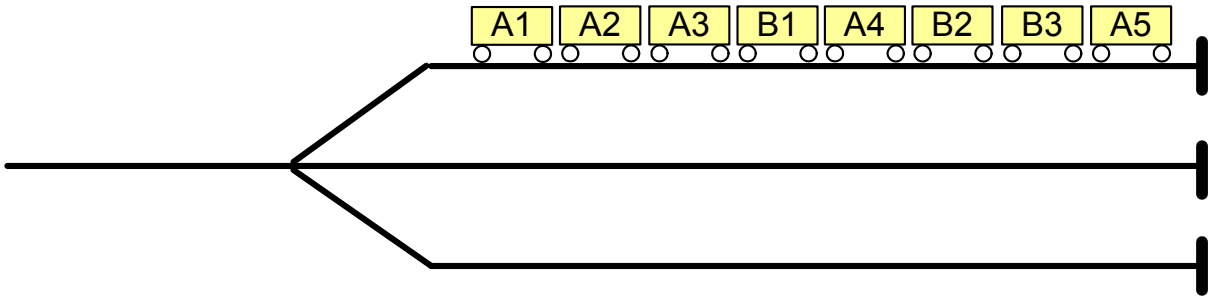
II) Weitere nützliche Methoden der Klasse STAPEL:

- NaechstenGeben() -> Referenz auf das oberste Datenelement
  Die Funktion liefert eine Referenz auf das oberste Datenelement, ohne dieses aus dem Stapel zu entfernen (engl. top oder peek).
- IstLeer()
  Diese Methode prüft, ob auf dem Stapel irgendein Datenelement (bzw. eine Referenz auf ein Datenelement) abgelegt ist.

Eine kleine Skizze hilft den Schülerinnen und Schülern, ein Gefühl für das Verhalten eines Stapels zu entwickeln.



Zur weiteren Vertiefung eignet sich das folgende Beispiel. Ein Rangierbahnhof hat mehrere Abstellgleise, die jeweils nach dem Stapelprinzip arbeiten. Aus verschiedenen Gegenden kommen die Güterzüge in einem Rangierbahnhof an. Dort werden die Waggons, je nach Zielbahnhof zu neuen Zügen zusammengestellt. Je weniger Rangiervorgänge hierzu erforderlich sind, desto effizienter arbeiten die Rangierer.



Ein Zug mit diversen Waggons (A1, A2, A3, B1, A4 …) soll mit drei Abstellgleisen G1, G2, G3 umrangiert werden, er steht im Gleis G1. Es kann immer nur ein Waggon bewegt werden und am Ende sollen alle A-Waggons in einem und alle B-Waggons im anderen Gleis stehen. Eine Lösung sollen die Schülerinnen und Schüler in der Schreibweise w = G1.Pop(); G2.Push(w); w = G1.Pop(); G3.Push(); ... festhalten.

Stapel finden gerade in der EDV eine große Verbreitung. Sie sind einfach zu implementieren und erfüllen wichtige Aufgaben. Einige Beispiele:

- Rücksprungadressen bei Aufruf von Methoden: Wird eine Methode B aus einer Methode A heraus aufgerufen, dann muss das System wissen, wo es nach Abschluss der Methode B wieder weiterarbeiten muss (in Methode A an welcher Stelle). Diese Stellen nennt man Rücksprungadressen. Da Methodenaufrufe mehrfach geschachtelt sein können, muss sich das System mehrere Rücksprungadressen merken können. Sie werden meist mit einem Stapel verwaltet, denn sie müssen der Reihe nach wieder abgearbeitet werden, die zuletzt gemerkte als Erste.
- Als Zwischenspeicher bei Termauswertungen. Bei Termen (einfacher Term z. B. 3+ 4*5 + 6) sind die üblichen Rechenregeln bezüglich der Operatoren einzuhalten (z. B. „Punkt vor Strich“). Es müssen deshalb Zwischenergebnisse gespeichert werden, bevor damit wieder weitergearbeitet werden darf. Diese Zwischenergebnisse werden in einem Stapel abgelegt, wobei man sich zweier Stapel bedient, einer für die Operatoren und einer für die Operanden.
- Organisation der Rekursion: Beim rekursiven Aufruf einer Methode wird in einer Methode dieselbe erneut aufgerufen. Die aufrufende Methode wartet auf die Antwort des Aufrufs. Dieser Zustand muss konserviert werden, damit nach dem „Zurücklaufen der Rekursion“ die aufrufende Methode wieder in dem Zustand fortgesetzt werden kann, in dem sie sich beim Aufruf befand. Diese Zustände werden auf einem Stapel zwischengespeichert.

Im Vergleich zum Stapel ist eine Schlange eine dynamische Datenstruktur, die das Einfügen und Entfernen von Elementen nach dem FIFO-Prinzip (first-in, first-out) unterstützt. Elemente werden am Ende der Schlange eingefügt und am Anfang der Schlange herausgenommen. Ein Zugriff auf innere Elemente ist nicht möglich.

1.1.4.2 Implementierung der Klasse STAPEL

Ein Stapel ist eine sehr einfache, aber vielfältig einsetzbare dynamische Datenstruktur. Es liegt nahe, diese Datenstruktur direkt zu implementieren. Im Lehrplan ist aber vorgesehen, dass die bisher entwickelte einfach verkettete Liste hierzu verwendet wird. Dieser Abschnitt sollte noch mal zur Festigung der Kenntnisse über Listen dienen.

Für den Stapel werden nur einige Fähigkeiten der allgemeinen einfach verketteten Liste benötigt. Eine Definition der Klasse STAPEL als Unterklasse LISTE wäre nicht sachgerecht,

denn die Klasse STAPEL ist keine Verfeinerung (Spezifizierung) der Klasse LISTE, sondern im Gegenteil eine Einschränkung. Diese Implementierung ist ein Beispiel für die Anwendung des so genannten Adaptermusters (siehe Kapitel 2.1.3 Softwaretechnik, Entwurfsmuster).

```
class STAPEL
{
  private LISTE liste;

  // Konstruktor
  STAPEL()
  {
    liste = new LISTE();
  }

  // Methoden
  void Einfuegen(DATENELEMENT neueDaten)
  {
    liste.AnfangEinfuegen(neueDaten);
  }

  DATENELEMENT Entfernen()
  {
    return liste.AnfangEntfernen();
  }

  DATENELEMENT NaechstenGeben()
  {
    return liste.AnfangGeben();
  }

  boolean IstLeer()
  {
    return liste.IstLeer();
  }
}
```

Bei dieser Implementierung werden die Klassen LISTE, KNOTEN und DATENELEMENT wie in den vorherigen Kapiteln definiert verwendet.

1.1.4.3 Implementierung der Klasse SCHLANGE

Auch die Klasse SCHLANGE lässt sich unter Zuhilfenahme der Klasse LISTE schnell implementieren. Auch für eine Schlange sind abgesehen vom Konstruktor nur die zwei Methoden erforderlich: Einfügen und Entfernen. Im Gegensatz zum Stapel erfolgt das Einfügen am Ende.

```
class SCHLANGE
{
  private LISTE liste;

  // Konstruktor
  SCHLANGE()
  {
    liste = new LISTE();
  }

// Methoden
```

```
  void Einfuegen(DATENELEMENT neueDaten)
  {
    liste.EndeEinfuegen(neueDaten);
  }
  DATENELEMENT Entfernen()
  {
    return liste.AnfangEntfernen();
  }
  DATENELEMENT NaechstenGeben()
  {
    return liste.AnfangGeben();
  }

  boolean IstLeer()
  {
    return liste.IstLeer();
  }

}
```

1.2 Bäume (ca. 16 Std.)

Lp: In Erweiterung ihrer Grundkenntnisse über das hierarchische Ordnungsprinzip lernen die Schüler den Baum als effiziente dynamische Datenstruktur kennen. Sie stellen fest, dass damit viele Strukturen aus anderen Gebieten abgebildet werden können. Am Beispiel der Suche in umfangreichen Datenbeständen wird den Jugendlichen deutlich, dass sich auch hier sehr oft Baumstrukturen einsetzen lassen, um die Effizienz der Informationsverarbeitung zu steigern. Bei der sprachlichen bzw. graphischen Darstellung und insbesondere bei der Implementierung dieser Datenstruktur vertiefen sie ihr Verständnis für das Prinzip der Rekursion.

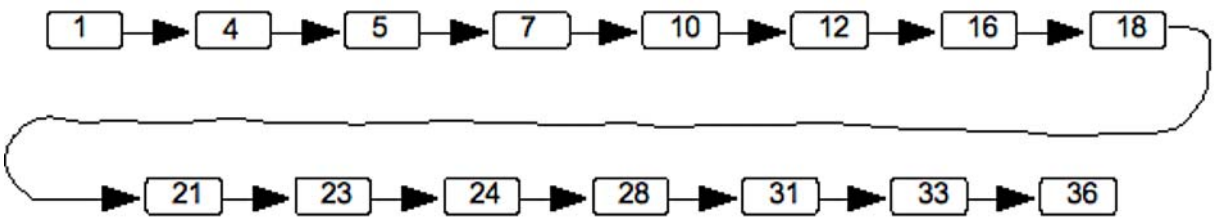
1.2.1 Effizienzsteigerung durch bessere Struktur

Lp: allgemeines Prinzip und Struktur eines Baums (insbesondere Wurzel, Knoten, Kanten, Blatt) und des Spezialfalls geordneter Binärbaum

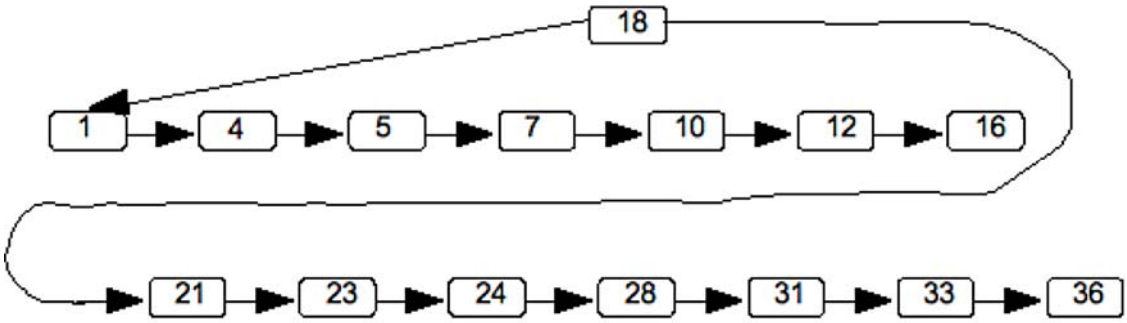
(Lp: Veranschaulichung und Implementierung der Methoden zum Einfügen und Suchen von Elementen in einem geordneten Binärbaum unter Verwendung der Rekursion)

1.2.1.1 Der Weg zum Baum

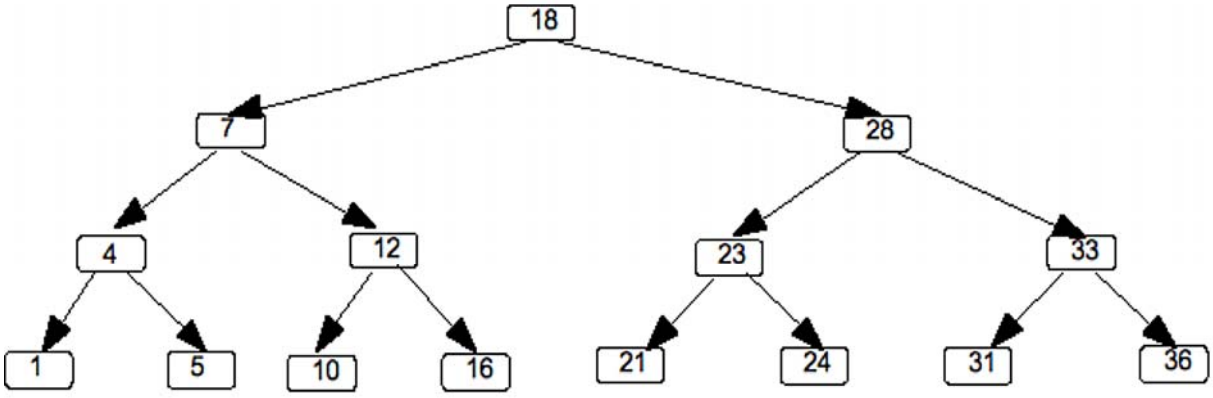
Bei der Suche in umfangreicheren sortierten Listen stellen die Schülerinnen und Schüler fest, dass die Sortierung den Suchvorgang insbesondere beim Erkennen nicht vorhandener Werte abkürzt, dass aber trotzdem die durchschnittliche Anzahl notwendiger Vergleiche linear mit der Länge der Liste zunimmt. So wird in der nachfolgend dargestellten Liste der Wert 8 nach fünf Vergleichen als „nicht vorhanden“ erkannt im Gegensatz zu 15 notwendigen Vergleichen bei einer nicht sortierten Liste. Für den Wert 33 sind aber immerhin 14 Vergleiche nötig. Im Schnitt benötigt man bei dieser Liste 8 Vergleiche, um einen vorhandenen Wert zu finden. Für eine Liste mit n Elementen sind im Durchschnitt (n+1)/2 Vergleiche nötig, um einen in der Liste vorhandenen Wert zu finden. Bei sehr großen Listen sind also durchschnittlich auch sehr viele Vergleiche notwendig.



Die Schülerinnen und Schüler stellen sehr schnell fest, dass man wesentlich eher zum Ziel kommt, wenn man die Liste in der Mitte teilt und nach dem ersten Vergleich des gesuchten Wertes mit dem Wert des mittleren Knotens gleich in der richtigen Hälfte linear, wie oben sucht.



Für die Suche nach vorhandenen Werten liegt die durchschnittliche notwendige Anzahl von Vergleichen jetzt bei knapp unter fünf, da für jeden Wert in einer der Teillisten ein Vergleich benötigt wird, um die richtige Teilliste zu finden und durchschnittlich vier Vergleiche, um den Wert in der Teilliste zu finden. Für die Suche nach dem Wert 18 ist aber nur ein Vergleich nötig, sodass der Gesamtdurchschnitt etwas unter fünf liegt. Die durchschnittliche Zahl der Vergleiche sinkt weiter, wenn man diese Halbierung der Liste konsequent fortsetzt. Damit kommt man schließlich zu folgender Struktur:



Hier sind im Schnitt nur noch knapp vier (bei n Elementen etwa: $\log_2(n)$) Vergleiche nötig, um einen Wert zu erkennen oder zu falsifizieren. Die Jugendlichen erkennen ohne Probleme die ihnen schon aus Jahrgangsstufe 6 bekannte Baumstruktur, eine ausführliche Auffrischung aller notwendigen Begriffe (Knoten, Blatt, Kante, Ast, Wurzel) ist aber in der Regel nötig.

Didaktische Anmerkungen:

- Im gesamten bisherigen Informatikunterricht ist das die erste Stelle, bei der eine wesentliche Neuerung nicht aufgrund einer neuen Aufgabenstellung, sondern allein aus internen Gründen – hier der Effizienzsteigerung – erfolgt ist.
- Das obige Verfahren führt direkt zum binären Suchbaum: Jeder Knoten hat maximal 2 Nachfolger; für jeden Knoten des Baums gilt: die Werte in allen Knoten des linken Teilbaums sind kleiner als der Wert dieses Knotens, die Werte in allen Knoten des rechten Teilbaums sind größer als der Wert dieses Knotens.

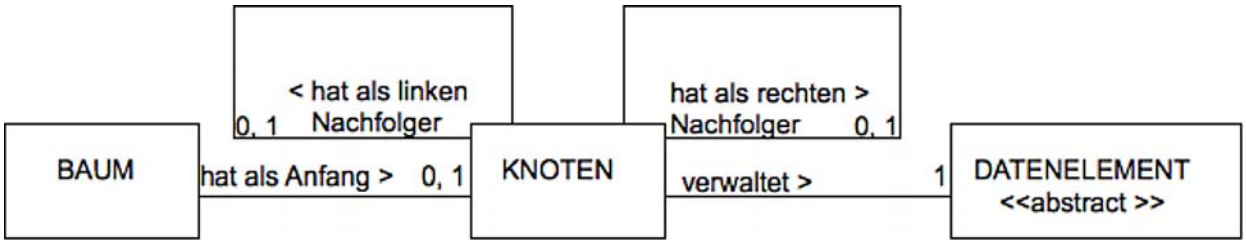
1.2.1.2 Struktur von Bäumen

Mithilfe der Kenntnisse aus dem Bereich der Listen ist es den Schülerinnen und Schülern sehr leicht möglich, in Analogie die zentralen Klassen KNOTEN und BAUM zu identifizieren und deren notwendige Attribute zu benennen. Auch die Trennung von Struktur und Daten kann direkt übernommen werden. Als didaktische Frage ist hier offen, ob das Entwurfsmuster Kompositum ebenfalls mit übernommen wird oder ob zunächst mit leeren Referenzen gearbeitet wird.

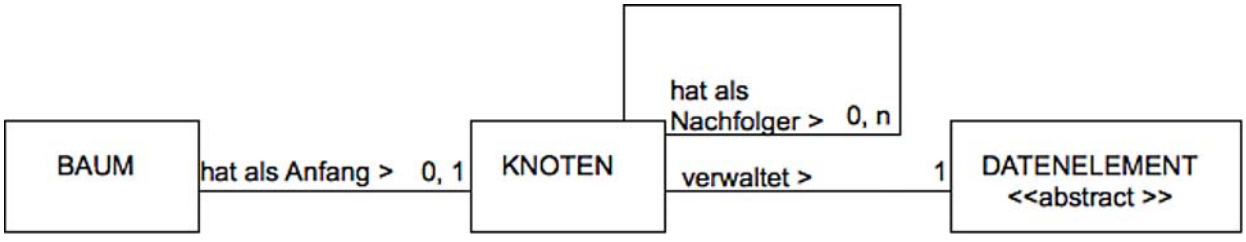
- Für die sofortige Verwendung von Kompositum spricht, dass man nicht nachbessern muss, sondern gleich den „eleganten“ Weg, der den Schülerinnen und Schülern mittlerweile gut bekannt ist, weitergehen kann.
- Für das vorläufige Arbeiten mit leeren Referenzen spricht, dass die Abbruchbedingung für die rekursiven Methoden im Quellcode explizit werden und somit das Prinzip der Rekursion nochmals „pur“ angewendet werden kann. Ebenfalls spricht für diese Variante, dass die Schülerinnen und Schüler zu Beginn nicht wissen können, ob das Entwurfsmuster Kompositum hier gewinnbringend eingesetzt werden kann. Die Jugendlichen erkennen erst im Lauf der Entwicklung, dass genau die Strukturen der Listen wieder auftreten, die zur Verwendung des Kompositums geführt haben.

Die Wahl des Weges hängt auch von der Leistungsfähigkeit der Schülerinnen und Schüler ab. Bei einem schwächeren Kurs bietet sich die zweite Variante an. Für einen starken Kurs mit guter Abstraktionsfähigkeit kommt die erste Variante infrage; die dabei gesparte Zeit kann dann für Vertiefungen verwendet werden.

Die folgenden Ausführungen beschreiten den zweiten Weg. Das erste Klassendiagramm (für den Binärbaum) besteht als aus den Klassen BAUM, KNOTEN und DATENELEMENT. Die Beziehungen für den linken bzw. rechten Nachfolger sind hier getrennt angegeben, da sie wegen der Sortierung semantische Bedeutung haben.



Der allgemeine Baum lässt sich aus diesem Klassendiagramm direkt herleiten.



Beispiele für allgemeine Bäume (wie der Ordnerbaum des Dateisystems) sind den Schülerinnen und Schülern hinreichend bekannt.

Didaktische Anmerkungen

Eine Implementierung allgemeiner Bäume ist im Lehrplan nicht vorgesehen.

Zu implementieren sind die Methoden *Suchen* und *Einfuegen* in einem geordneten Binärbaum. Entfernen eines Knotens ist als Transferaufgabe möglich, muss aber nicht thematisiert werden. Der in diesem Zusammenhang eigentlich nötige Längenausgleich der Äste (realisiert z. B. in ausgeglichenen Bäumen nach Adelson-Velskii und Landis, so genannte AVL-Bäume) sprengt sicher den Rahmen der zeitlichen Möglichkeiten.

1.2.1.3 Suchen im geordneten Binärbaum

Bei den Vorüberlegungen zum Entstehen der Datenstruktur Baum entsteht ein geordneter Binärbaum, also genau der Spezialfall des allgemeinen Baums, der im Folgenden weiter behandelt wird. Auch ergibt sich die Funktionsweise der Methode *Suchen* der Klasse KNOTEN direkt aus diesen Vorüberlegungen. Die Methode *Suchen* der Klasse BAUM stützt sich einfach nur auf diese Methode ab.

An dieser Stelle muss auch über die Signatur der Methode *Suchen* entschieden werden. Für die Erkundung der Funktionsweise der Methode *Suchen*, d. h. für das sich Eindenken in die Funktionsweise, genügt ein logischer Rückgabewert (gefunden oder nicht gefunden). Aus den Anwendungen der Klasse LISTE wissen die Schülerinnen und Schüler aber bereits, dass in vielen Fällen das gesuchte (und gefundene) Datenelement weiter verwendet werden soll; der Rückgabewert der Methode *Suchen* ist dann eine Referenz auf das gefundene Datenelement oder die leere Referenz, wenn es kein Datenelement mit dem gegebenen Schlüssel gibt. Hier wird von Beginn an mit einer Referenz als Rückgabewert gearbeitet. Es könnte aber auch für die „Testimplementierung“ ein Wahrheitswert zurückgegeben werden; beim Umbau auf das Entwurfsmuster Kompositum könnte dann auch die Methode *Suchen* erweitert werden.

```
// Methode der Klasse KNOTEN
// Suchen
Methode Suchen(wert) -> Referenz auf Datenelement oder leere Referenz
    wenn das eigene Datenelement den gesuchten Wert wert hat
        // gefunden,
        Rückgabewert = Referenz auf das Datenelement
    sonst   // beim richtigen Nachfolger weiterfragen
        wenn wert kleiner ist als der Wert des Datenelements // links gehts weiter
            wenn es einen linken Nachfolger gibt
                Rückgabewert = LinkerNachfolger.Suchen(wert)
            sonst //gibts nicht, also nichts gefunden
                Rückgabewert = leere Referenz
            endeWenn
        sonst //rechts gehts weiter
            wenn es einen rechten Nachfolger gibt
                Rückgabewert = RechterNachfolger.Suchen(wert)
            sonst //gibts nicht, also nichts gefunden
                Rückgabewert = leere Referenz
            endeWenn
        endeWenn
    endeWenn
EndeMethode
```

```
// Methode der Klasse BAUM
// Suchen
Methode Suchen(wert) -> Wahrheitswert
    // das Wurzelobjekt fragen
    wenn es ein Wurzelobjekt gibt
        Rückgabewert  = Wurzelobjekt.Suchen(wert)
    sonst
        Rückgabewert = leere Referenz
    endeWenn
EndeMethode
```

Auf dieser Ebene lässt sich die Funktionsweise des Baums mit vorhandenen Visualisierungswerkzeugen nachvollziehen, z. B. dem BaumVisualisierungsTool (Freeware, zu finden unter www.bvt.ct-labs.de).

Eine geeignete Veranschaulichung der Methoden zum Einfügen und Suchen von Elementen in einem geordneten Binärbaum soll in das didaktische Konzept einbezogen werden.

1.2.1.4 Erste Implementierung des geordneten Binärbaums

Aus Sicht der Softwareentwicklung ist eine Implementierung in diesem frühen Stadium völlig verfehlt. Auch lassen sich aufgrund der fehlenden Methode *Einfuegen* noch keine Bäume dynamisch aufbauen. Hier muss aber die pädagogische Situation berücksichtigt werden: Die Schülerinnen und Schüler erkunden die neue Datenstruktur Baum. Zu dieser Erkundung gehören auch die Implementierung und das Arbeiten mit dieser Implementierung. Diese Implementierung kann daher auch als eine Art „rapid prototyping“ gesehen werden, die nicht auf Verwendung, sondern auf Erkundung ausgelegt ist.

In dieser Phase ist der einfachste Weg zur Erstellung von Testbäumen, einen Konstruktor mit Nachfolgerangabe zur Verfügung zu stellen und mithilfe von Testumgebungen, wie sie z. B. in BlueJ zur Verfügung stehen, geeignete Testvarianten aufzubauen. Als Beispiel sind Baum1 (Beispiellösung) und Baum1Test (Beispiellösung mit Test) vorhanden.

Der Quelltext der zentralen Klasse KNOTEN ist nachfolgend abgedruckt:

```
class KNOTEN
{
    private KNOTEN linkerNachfolger;
    private KNOTEN rechterNachfolger;
    private DATENELEMENT daten;

    KNOTEN(DATENELEMENT d)
    {
        linkerNachfolger = null;
        rechterNachfolger = null;
        daten = d;
    }

    KNOTEN(DATENELEMENT d, KNOTEN lnf, KNOTEN rnf)
    {
        linkerNachfolger = lnf;
        rechterNachfolger = rnf;
        daten = d;
    }
```

```
    DATENELEMENT Suchen (DATENELEMENT wert)
    {
        // Die Methode Vergleichen ist in
        // DATENELEMENT beschrieben
        if (daten.Vergleichen(wert) == 0)
        {
            return daten;
        }
        else
        {
            if (daten.Vergleichen(wert) < 0)
            {
                if (rechterNachfolger == null)
                {
                    return null;
                }
                else
                {
                    return rechterNachfolger.Suchen(wert);
                }
            }
            else
            {
                if (linkerNachfolger == null)
                {
                    return null;
                }
                else
                {
                    return linkerNachfolger.Suchen(wert);
                }
            }
        }
    }
}
```

1.2.2 Schrittweise Implementierung eines typischen Baums

Lp: Veranschaulichung und Implementierung der Methoden zum Einfügen und Suchen von Elementen in einem geordneten Binärbaum unter Verwendung der Rekursion

1.2.2.1 Die Methode Einfuegen

Die Grundidee für die Methode *Einfuegen* ist, dass trotz Sortierung ein neues Element nie „im Innern“ eingefügt wird; es kann immer ein Knoten ohne Nachfolger gefunden werden, an den das Element korrekt sortiert angefügt werden kann.

Eine an dieser Stelle prinzipiell notwendige Entscheidung ist die Antwort auf die Frage, wie beim Einfügen von bereits im Baum vorhandenen Werten verfahren wird. Im Folgenden wird die Vereinbarung des Suchbaums zugrunde gelegt, die keine doppelten Einträge zulässt. Damit ist jeder Wert nur genau ein Mal erlaubt; er stellt damit auch einen Schlüssel für die über diesen Wert identifizierbaren Datenelemente dar. Duplikate dürfen daher nicht eingefügt werden.

Die Grundidee des Einfügens ist identisch mit der Idee zur Suche. Es wird nach dem gleichen Muster wie bei der Suche rekursiv im Baum so lange abgestiegen, bis ein freier Platz gefunden ist, d. h. bis es an der untersuchten Stelle keinen Nachfolger gibt. Da doppelte Werte nicht

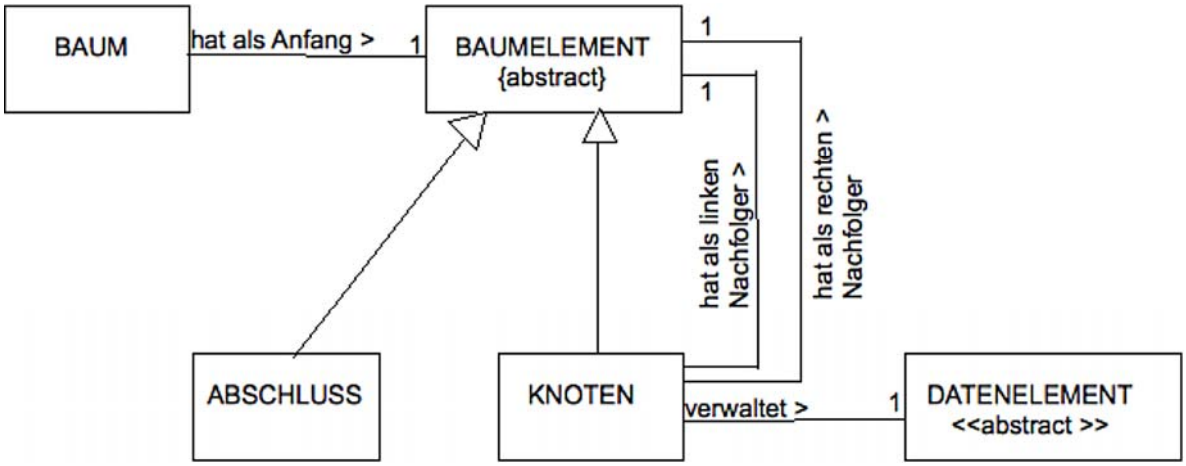
erlaubt sind, wird das Einfügen mit einer Fehlermeldung abgebrochen, wenn der einzufügende Wert im Baum gefunden wurde.

```
// Methode der Klasse KNOTEN
// Einfuegen
Methode Einfuegen(wert)
    wenn das Datenelement den gesuchten Wert hat
        // gefunden; unzulässig, da hier ein Suchbaum entsteht
        Fehlermeldung ausgeben
    sonst   // beim richtigen Nachfolger weiterfragen
        wenn wert kleiner ist als der Wert des Datenelements // links gehts weiter
            wenn es einen linken Nachfolger gibt
                LinkerNachfolger.Einfuegen(wert)
            sonst // gibts nicht, also Platz zum Einfügen des neuen Elements
                LinkerNachfolger = neuer KNOTEN(wert)
            endeWenn
        sonst // rechts gehts weiter
            wenn es einen rechten Nachfolger gibt
                RechterNachfolger.Einfuegen(wert)
            sonst // gibts nicht, also Platz zum Einfügen des neuen Elements
                RechterNachfolger = neuer KNOTEN(wert)
            endeWenn
        endeWenn
    endeWenn
EndeMethode

// Methode der Klasse BAUM
Methode Einfuegen(wert)          // Einfuegen
    wenn es ein Wurzelobjekt gibt                    // das Wurzelobjekt fragen
        Wurzelobjekt.Einfuegen(wert)
    sonst
        Wurzelobjekt = neuer KNOTEN(wert)
    endeWenn
EndeMethode
```

Nun können in den Baum Daten in verschiedener Reihenfolge eingefügt werden; das Ergebnis kann anschließend z. B. mit dem Objectinspector von BlueJ verifiziert werden. Baum2 (Beispiellösung) und Baum2Test (Beispiellösung mit Test) geben diesen Stand wieder.

1.2.2.2 Baum und Kompositum

Die Schülerinnen und Schüler besitzen nun genügend Erfahrung mit der Datenstruktur Baum, um anhand der vielen Terminierungsbedingungen in den Methoden *Suchen* und *Einfuegen* abschätzen zu können, dass das Entwurfsmuster Kompositum auch hier sehr gewinnbringend angewendet werden kann. Die Ergänzung des Klassendiagramms ist nahezu Reproduktion (bis Reorganisation).



Der Umbau der Methode *Suchen* ist trivial, da wie gehabt ein Objekt der Klasse ABSCHLUSS einfach „nicht gefunden“, d. h. die leere Referenz zurückmelden muss. Beim Einfügen muss dagegen wie auch schon bei der Liste die Methode *Einfuegen* jetzt den (potenziell neuen) Nachfolger für die Nachfolgerreferenz zurückmelden. Die übrigen Detailveränderungen sind analog, sodass die Anwendung des Musters Kompositum auf den Baum im Schwierigkeitsbereich Reproduktion bis Transfer anzusiedeln ist.

Ein direkter Vergleich des Quelltextes macht die Vereinfachungen deutlich sichtbar. Die Implementierung der Methoden in der Klasse ABSCHLUSS ist kurz.

Ohne Kompositum

```
DATENELEMENT Suchen(DATENELEMENT wert)
{
if (daten.Vergleichen(wert)==0)
{
    return daten;
}
else
{
    if (daten.Vergleichen(wert) < 0)
    {
        if (rechterNachfolger == null)
        {
            return null;
        }
        else
        {
            return
            rechterNachfolger.Suchen(wert);
        }
    }
    else
    {
        if (linkerNachfolger == null)
        {
            return null;
        }
        else
        {
            return
            linkerNachfolger.Suchen(wert);
        }
     }
}
}
```

Mit Kompositum

```
//in der Klasse KNOTEN:
DATENELEMENT Suchen(DATENELEMENT wert)
{
if (daten.Vergleichen(wert)==0)
{
    return daten;
}
else
{
    if (daten.Vergleichen(wert) < 0)
    {
        return
        rechterNachfolger.Suchen(wert);
    }
    else
    {
        return
        linkerNachfolger.Suchen(wert);
    }
}
}
```

```
//in der Klasse ABSCHLUSS:
DATENELEMENT Suchen(DATENELEMENT wert)
{
    return null;
}
```

```
void Einfuegen (DATENELEMENT wert)
{
if (daten.Vergleichen(wert) == 0)
{
    System. out. print ("Das Datenelement ist
                         schon vorhanden: ");
    wert.Ausgeben();
}
else
{
    if (daten.Vergleichen(wert) < 0)
    {
        if (rechterNachfolger == null)
        {
            rechterNachfolger =
                           new KNOTEN(wert);
        }
        else
        {
            rechterNachfolger.
                           Einfuegen(wert);
        }
     }
     else
     {
        if (linkerNachfolger == null)
        {
            linkerNachfolger =
                          new KNOTEN (wert);
        }
        else
        {
            linkerNachfolger.
                          Einfuegen(wert);
            }
        }
    }
}
```

```
//in der Klasse KNOTEN:
BAUMELEMENT Einfuegen(DATENELEMENT wert)
{
if (daten.Vergleichen (wert) == 0)
{
    System. out. print ("Das Datenelement ist
                          schon vorhanden: ");
        wert.Ausgeben();
}
else
{
    if (daten.Vergleichen(wert) < 0)
    {
        rechterNachfolger =
        rechterNachfolger.Einfuegen(wert);
    }
    else
    {
        linkerNachfolger =
        linkerNachfolger.Einfuegen(wert);
    }
}
return this;
}
```

```
//in der Klasse ABSCHLUSS:
BAUMELEMENT Einfuegen
    (DATENELEMENT wert)
{
    return new KNOTEN(wert);
}
```

Auch die Implementierung der Klasse BAUM vereinfacht sich. Das vollständige Programm ist in Baum3 (Beispiellösung) und Baum3Test (Beispiellösung mit Test) vorhanden.

1.2.2.3 Baumdurchlauf

Lp: Verfahren zur Auflistung aller Elemente eines geordneten Binärbaums: Preorder-, Inorder- und Postorder-Durchlauf; Realisierung der Methode zum Ausgeben mithilfe eines dieser Verfahren

Der Baumdurchlauf dient dazu, alle Elemente des Baums systematisch zu besuchen, z. B. um den Inhalt des Baums auszugeben, Operationen auf den Datenelementen durchzuführen, aggregierte Informationen zu gewinnen. Insbesondere bei der Ausgabe der Datenelemente ist die Reihenfolge der Bearbeitung wichtig. Den drei möglichen Durchlaufstrategien lassen sich typische Anwendungssituationen zuordnen.

Name	Strategie	Typische Anwendung
preorder	Das Datenelement des jeweilig aktuellen Knotens wird vor den Datenelementen des linken und des rechten Nachfolgers behandelt.	Ein so ausgegebener Baum wird beim Einlesen der Elemente und Einfügen unter Verwendung von *Einfuegen* identisch wieder aufgebaut.
inorder	Das Datenelement des jeweilig aktuellen Knotens wird nach dem linken, aber vor dem rechten Nachfolger behandelt.	So können die Elemente sortiert ausgegeben werden.

postorder	Das Datenelement des jeweilig aktuellen Knotens wird nach den Datenelementen des linken und des rechten Nachfolgers behandelt.	Bei der Abarbeitung von Rechenbäumen müssen die Werte aus den Nachfolgern bekannt sein, bevor der Knotenwert berechnet werden kann.

Die Struktur der drei Durchlaufmethoden ist fast identisch. Die Terminierung der Rekursion wird jeweils in den Objekten der Klasse ABSCHLUSS erzwungen.

Bei der Klasse KNOTEN:

```
Methode PreOrder()
    <Arbeit auf den Daten, z. B. Ausgabe der Daten>
    LinkerNachfolger.PreOrder()
    RechterNachfolger.Preorder ()
EndeMethode

Methode InOrder()
    LinkerNachfolger.InOrder()
    <Arbeit auf den Daten, z. B. Ausgabe der Daten>
    RechterNachfolger.InOrder()
EndeMethode

Methode PostOrder()
    LinkerNachfolger.PostOrder()
    RechterNachfolger.PostOrder()
    <Arbeit auf den Daten, z. B. Ausgabe der Daten>
EndeMethode
```

Bei der Klasse ABSCHLUSS:

```
Methode PreOrder()
EndeMethode

Methode InOrder()
EndeMethode

Methode PostOrder()
EndeMethode
```

Die Umsetzung kann in Baum4 (Beispiellösung) und Baum4Test (Beispiellösung mit Test) nachvollzogen werden.

1.2.3 Anwendungen und Vertiefungen

Die folgenden Beispiele geben mögliche Anwendungen und Vertiefungen an.

1.2.3.1 Beispiel Wörterbuch

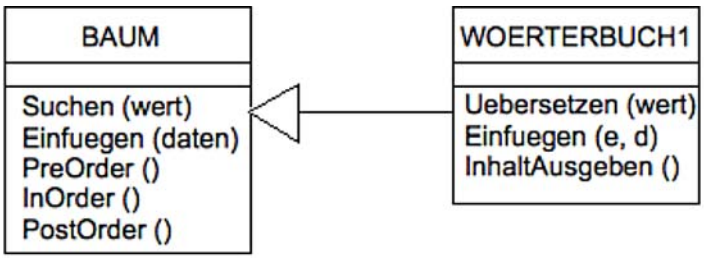
> Lp: Implementierung der Klasse geordneter Binärbaum mit einer geeigneten Programmiersprache; Verwenden und Testen der Methoden an einem Anwendungsbeispiel (z. B. erweiterbares Wörterbuch als Suchbaum).

Das Wörterbuchbeispiel (oder ein ähnliches) kann als Leitbeispiel für die Erkundung der Struktur Baum dienen (themenzentrierter Unterricht). Hier ist es als Anwendungsbeispiel dargestellt, bei dem aufgezeigt wird, auf welche Arten eine vorhandene Klasse BAUM zur Umsetzung von Anwendungen genutzt werden kann.

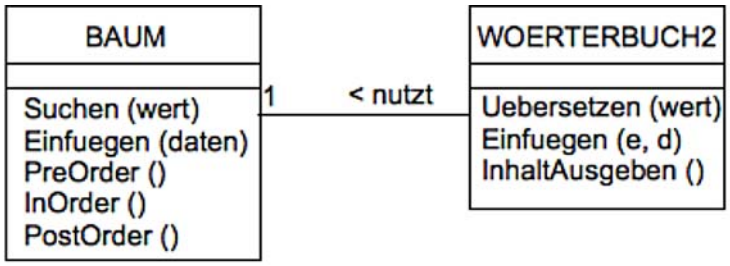
Die Methoden der Klasse BAUM werden beim Wörterbuch (aus der Sicht des Anwenders) leicht modifiziert genutzt, da beim Suchen sofort die deutsche Übersetzung des englischen Begriffs gewünscht wird; zum Teil werden die Methoden umbenannt.

Beim Einbau in ein Rahmenprogramm sind die Namen und Aufrufe für den Benutzer verborgen. Soll aber eine Klasse WOERTERBUCH zur Verwendung in beliebigen Programmen erstellt werden, muss diese Klasse Methoden mit genau der benötigten Funktionalität zur Verfügung stellen. Dabei sind zwei Umsetzungen möglich, nämlich die Bildung einer Unterklasse oder die Verwendung des Adaptermusters (vgl. Abschnitt 1.1.4.2).

Bei der Umsetzung mit Unterklasse erbt die Klasse WOERTERBUCH von BAUM und ergänzt die neu benötigten Methoden.



Bei der Verwendung des Adaptermusters (vgl. Abschnitt 1.1.4.2) nutzt eine neue Klasse WOERTERBUCH ein Objekt der Klasse BAUM, um die benötigten Methoden zur Verfügung zu stellen.



Die Klassendiagramme ähneln sich stark, erst in der Implementierung werden die Unterschiede deutlich. Die Unterschiede sind hervorgehoben.

```
class WOERTERBUCH1 extends BAUM
{


    WOERTERBUCH1()
    {
        super ();
    }
```

```
class WOERTERBUCH2
{
    BAUM baum;

    WOERTERBUCH2()
    {
        baum = new BAUM ();
    }
```

```
    void Uebersetzen (String wort)
    {
        WORT daten;
        daten = (WORT)
                Suchen (new WORT (wort,""));
        if (daten != null)
        {
            daten.Ausgeben();
        }
        else
        {
            System.out.println("Unbekanntes
                            Wort: " + wort);
        }
    }

    void InhaltAusgeben ()
    {
        InOrder ();
    }

    void Einfuegen (String englisch,
                    String deutsch)
    {
        Einfuegen(new
                  WORT (englisch, deutsch));
    }
}
```

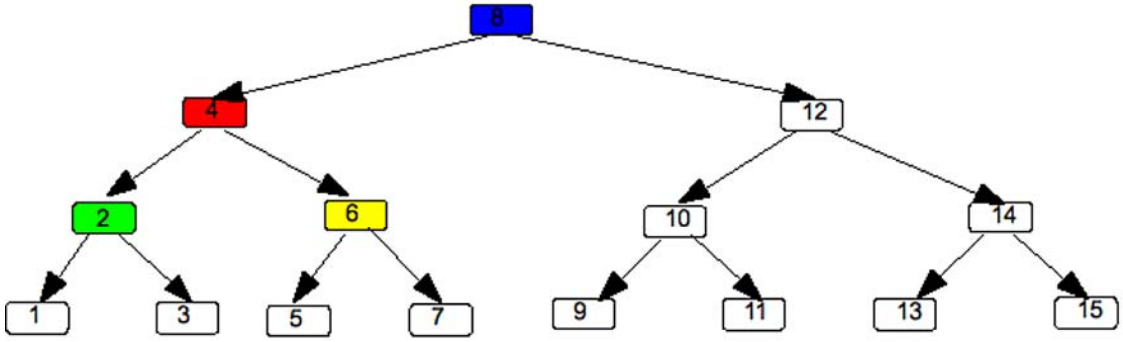
```
    void Uebersetzen (String wort)
    {
        WORT daten;
        daten = (WORT)
              baum.Suchen(new WORT(wort,""));
        if (daten != null)
        {
            daten.Ausgeben();
        }
        else
        {
            System.out.println("Unbekanntes
                            Wort: " + wort);
        }
    }

    void InhaltAusgeben ()
    {
        baum.InOrder();
    }

    void Einfuegen (String englisch,
                    String deutsch)
    {
        baum.Einfuegen(new
                   WORT (englisch, deutsch));
    }
}
```

Das Projekt Woerterbuch zeigt eine vollständige Beispiellösung.

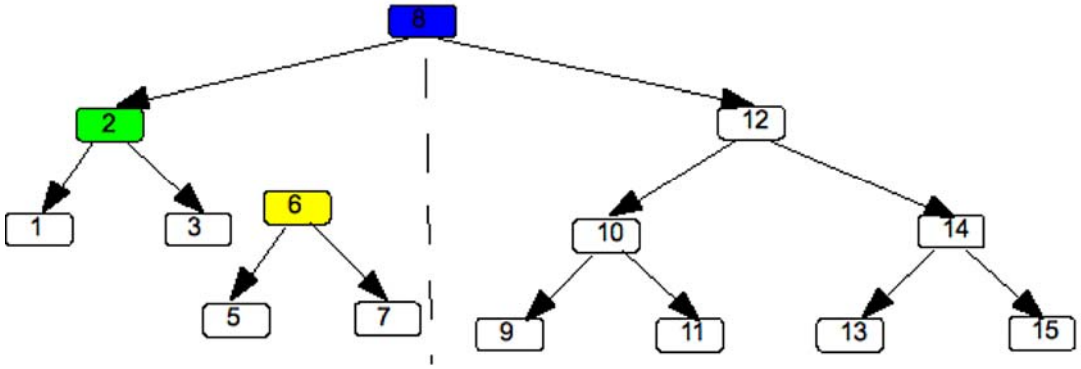
1.2.3.2 Entfernen von Knoten

Das Entfernen (gelöscht werden sie ja nicht) von Knoten eines Binärbaums ist im Lehrplan nicht explizit vorgeschrieben. Es ist hier angeführt, weil es ein gutes Beispiel der Vertiefung sowohl für den Umgang mit der Datenstruktur Baum als auch für rekursives Modellieren ist. Es sollte dabei aber erwähnt werden, dass durch Entfernen von Knoten die Äste des Baumes sehr schnell deutlich unterschiedliche Längen bekommen können, was die Vorteile des Baums beim Suchen mindert. Für praktische Anwendungen ist es daher nützlich, hier zusätzlich Ausgleichsmechanismen einzubauen. Darauf muss – wie in Abschnitt 1.2.1.2 bereits angesprochen – hier verzichtet werden, da die Behandlung dieser Ausgleichsmechanismen wie z. B. der Balancierung in AVL-Bäume den zur Verfügung stehenden Zeitrahmen überschreitet.

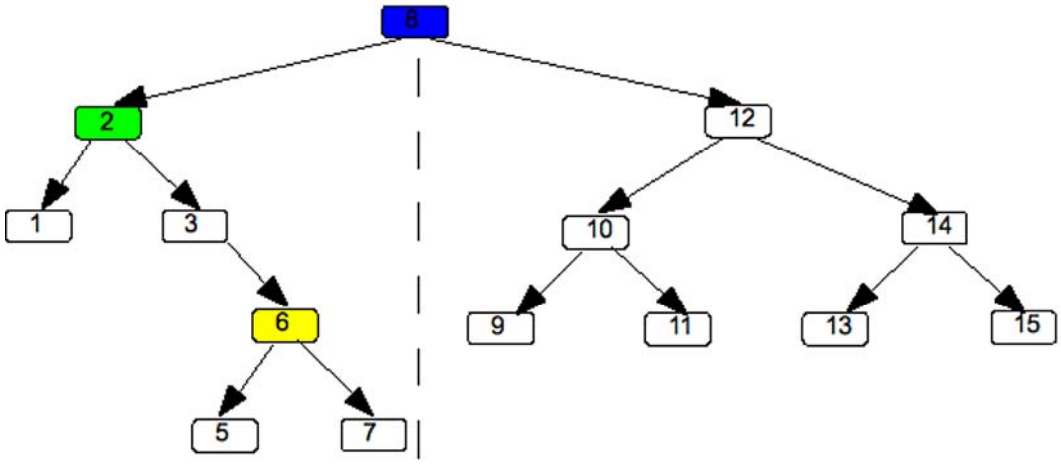
Wird ein Knoten entfernt, so hat dieser in der Regel zwei Nachfolger, von denen nur einer der Nachfolger des übergeordneten Knotens werden kann. Wird im Bild der rote (Wert 4) Knoten entfernt, so kann entweder der grüne (Wert 2) oder der gelbe (Wert 6) Knoten als Nachfolger an den blauen (Wert 8) Knoten angefügt werden, für den anderen Knoten ist hier kein Platz.



Wird der grüne (Wert 2) Knoten angehängt, so ist sicher: alle Werte des gelben (Wert 6) Knotens und seiner Nachfolger sind kleiner als die Werte des blauen (Wert 8) Knotens. Also muss dieser Teilbaum links des blauen (Wert 8) Knotens wieder eingefügt werden.



Alle Werte des grünen (Wert 2) Teilbaums sind sicher kleiner als die Werte des gelben (Wert 6) Teilbaums. Also müssen sie and die „am weitesten rechts liegende“ Stelle des grünen (Wert 2) Teilbaums angefügt werden, d. h. dort, wo das erste Mal kein Rechtsnachfolger existiert, im Beispiel an den Knoten mit dem Wert 3.



Die Methode *Entfernen* selbst sucht rekursiv (wie gehabt) nach dem zu entfernenden Knoten und gibt den eventuell neuen Nachfolger an den Aufrufer zurück.

Zum Anfügen des rechten Teilbaums verwendet sie eine Hilfsmethode *AnfügenRechts*, die bis zum ersten „freien“ Rechtsnachfolger absteigt und den Teilbaum dort anfügt. Durch die Verwendung des Kompositummusters wird auch hier die Terminierung in die Objekte der Klasse ABSCHLUSS verlagert, was die Methoden sehr kurz macht.

```
//Entfernen in der Klasse KNOTEN
Entfernen(wert) -> Referenz auf Baumelement
        wenn das Datenelement den gesuchten Wert wert hat
                // gefunden,
                Rückgabewert = LinkerNachfolger.AnfügenRechts(RechterNachfolger)
        sonst   // beim richtigen Nachfolger weiterfragen
                wenn wert kleiner ist als der Wert des Datenelements
                        //links gehts weiter
                        LinkerNachfolger = LinkerNachfolger.Entfernen(wert)
                        Rückgabewert = selbst
                sonst //rechts gehts weiter
                        RechterNachfolger = RechterNachfolger.Entfernen(wert)
                        Rückgabewert = selbst
                endeWenn
        endeWenn
EndeMethode

//Entfernen in der Klasse ABSCHLUSS
//Der zu entfernende Wert wurde nicht gefunden, eventuell Fehlermeldung
Entfernen (wert) -> Referenz auf Baumelement
        Rückgabewert = selbst
EndeMethode

//AnfügenRechts in der Klasse KNOTEN
AnfügenRechts(teilbaum)  -> Referenz auf Baumelement
        NachfolgerRechts = NachfolgerRechts.Anfügen(teilbaum)
        Rückgabewert = selbst
EndeMethode

//AnfügenRechts in der Klasse ABSCHLUSS
AnfügenRechts(teilbaum)  -> Referenz auf Baumelement
        Rückgabewert = teilbaum
EndeMethode
```

Als Beispiel sind Baum5 (Beispiellösung) und Baum5Test (Beispiellösung mit Test) vorhanden.

1.3 Graphen (ca. 13 Std.)

Lp: Im Rahmen praktischer Fragestellungen, z. B. zur Planung von Verkehrsrouten, wenden die Schüler auch die Datenstruktur Graph als Erweiterung der Struktur Baum an.

- die Datenstruktur Graph als Verallgemeinerung des Baums; Eigenschaften (gerichtet/ ungerichtet, bewertet/unbewertet); Adjazenzmatrix
- Algorithmus zum Graphendurchlauf (z. B. Tiefensuche) bei einer Aufgabenstellung aus der Praxis

1.3.1 Die Datenstruktur Graph

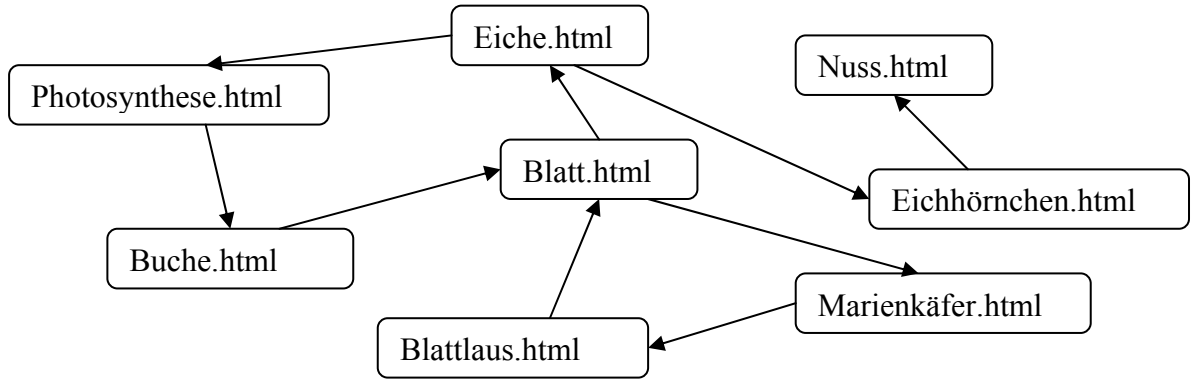
1.3.1.1 Anwendungen und Definition

Mit den bisher behandelten Datenstrukturen LISTE und BAUM können nur lineare bzw. hierarchische Strukturen modelliert werden. In den nachfolgend genannten, beispielhaften Aufgabenstellungen tritt jedoch eine netzartige Struktur auf:

- In einer auf dem Computer verarbeitbaren Form sollen für ein Gebiet alle Wege für Wanderer und / oder Radfahrer mit ihren Kreuzungspunkten erfasst werden.
- Es soll ein Navigationssystem für das deutsche Straßennetz erstellt werden.
- Basierend auf dem deutschen Streckennetz und dem Fahrplan für Schnellzüge soll nach Eingabe zweier Städte eine möglichst direkte Verbindung ausgegeben werden.
- Alle schiffbaren Flüsse und sonstigen Wasserwege sollen zur Navigation für die Schifffahrt erfasst werden.

Weitere Netze im öffentlichen Leben sind z. B. das elektrische Netz für die Stromversorgung, das Netz für Telephonverbindungen und auch das soziale Netz der Bekanntschaften zwischen Menschen.

Ein Beispiel für vernetzte Informationsstrukturen, das den Schülerinnen und Schülern schon aus Jahrgangsstufe 7 bekannt ist, sind verlinkte Dokumente, wie sie z. B. im World Wide Web vorkommen. Hierbei können die Dokumente als abgerundete Rechtecke und die Verweise zwischen zwei Dokumenten durch Pfeile dargestellt werden:



Auch für die anderen oben erwähnten Beispiele ist diese Art der Darstellung möglich: Zum Beispiel können für die Bahnverbindungen die deutschen Städte in die Umrandungen geschrieben werden; die Linien bedeuten dann eine vorhandene Schienenverbindung. Eine abstrahierende Sichtweise lässt zwei grundlegende Elemente eines Netzes erkennen: Das Netz besteht aus so genannten *Knoten* und *Kanten*, wobei jede Kante genau zwei Knoten verbindet. Folgende Tabelle zeigt eine mögliche Konkretisierung dieser abstrakten Sicht für die oben genannten Beispiele:

Anwendungszusammenhang	Knoten	Kanten
verlinkte Dokumente	einzelnes Dokument	Verweise
Karte von Wanderwegen	Kreuzungspunkte und Orte	Wanderwege
Navigationssystem für LKW	Ortschaften	Straßen
Netz für Schnellzüge	Städte	Schienenverbindungen
Schifffahrt	Kreuzungspunkte und Häfen	Wasserwege

Aus Sicht der Informatik ergeben sich für solche Anwendungen nun zwei prinzipielle Themenschwerpunkte:

a) Wie kann eine Netzstruktur modelliert werden, damit sie in einem informatischen System implementiert werden kann?

b) Welche zielgerichteten Verfahren gibt es, sich in einem Netz zu bewegen? Diese Verfahren dienen u. a. zur Beantwortung folgender exemplarischer Fragen:

- Wie ist es möglich, alle Knoten eines Graphen systematisch zu besuchen?
- Welche Wege gibt es von einem vorgegebenen Startknoten zu einem Zielknoten?
- Wie kann ein bestimmter Knoten im Netz gefunden werden?

Die abstrakte Sichtweise auf obige Beispiele unter Herausarbeitung der gemeinsamen Struktur führt auf die Definition der neuen Datenstruktur GRAPH, die für die Modellierung einer Netzstruktur geeignet ist.

Definition:

Ein GRAPH besteht aus einer endlichen Menge von **Knoten** und einer endlichen Menge von **Kanten**. Jede Kante verbindet genau zwei Knoten. Die Kanten können

a) gerichtet oder ungerichtet sein,

b) eine Bewertung haben oder nicht.

(Die Bewertung wird synonym auch als Gewichtung bezeichnet.)

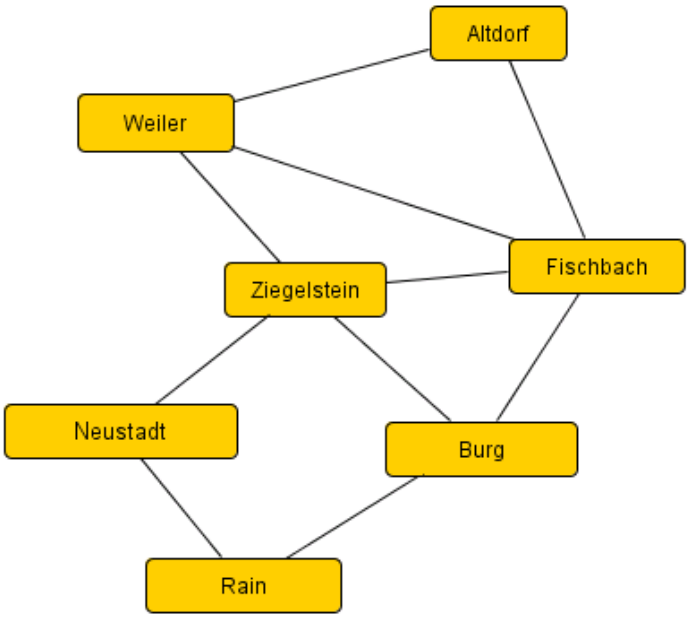
Die genannten Eigenschaften eines Graphen sollen folgende zwei Beispiele verdeutlichen:

Im eingangs erwähnten Fall der Wege für Radfahrer würden die Knoten des Graphen die Ortschaften darstellen und die Kanten des Graphen die Verbindungswege zwischen den Knoten bzw. Ortschaften. Interessieren uns nur die möglichen Verbindungen zwischen je zwei Orten, und gibt es jeweils einen Hin- und Rückweg (keine Einbahnstraßen), so kann diese Situation mit einem ungerichteten Graph ohne Gewichtung modelliert werden (Abb. 1).

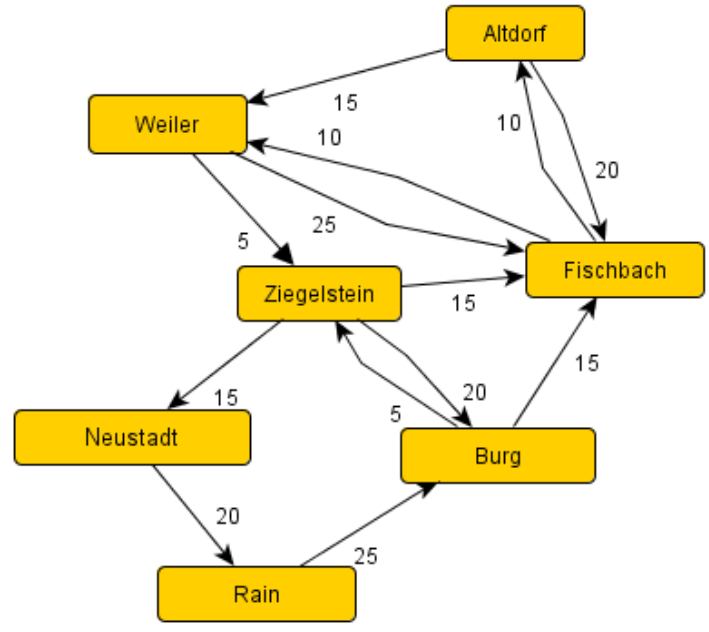
Für die Familie Müller mit kleinen Kindern ist jedoch nicht jeder Weg gleich gut zum Radfahren geeignet: Einige Strecken befährt sie nur sehr ungern aufgrund des hohen Verkehrsaufkommens oder der schlechten Fahrbahnqualität. Manchmal gibt es nur in einer Richtung einen Radweg und das Fahren in der Gegenrichtung wäre zu gefährlich. In bergigem Gelände ist die Steigung ausschlaggebend. Diese Gründe führen dazu, dass manche Wege nur in einer Richtung befahren werden bzw. die benötigte Zeit für Hin- und Rückweg sehr unterschiedlich ist. Diese Situation kann durch einen Graphen mit gerichteten und bewerteten Kanten modelliert werden. Die Bewertung soll hier die Zeit in Minuten angeben, die für das Fahren der Strecke benötigt wird (Abb. 2). So werden beispielsweise für die Fahrt von Weiler nach Fischbach über Ziegelstein 20 min Zeit veranschlagt. Dieser Graph wird im Folgenden als „Radnetz“ bezeichnet und später auch implementiert werden.

Anhand dieser beiden sehr ähnlichen Beispiele wird auch deutlich, dass der Anwendungszusammenhang die Eigenschaften des Graphen bestimmt.

Im Hinblick auf Themenschwerpunkt b) werden im Vorgriff die benötigten Begriffe zusammengestellt: Ein **Pfad** führt von einem Startknoten über mindestens eine Kante zu einem Zielknoten. Als **Länge des Pfades** bezeichnen wir in ungewichteten Graphen die Anzahl der durchlaufenen Kanten, bei gewichteten Graphen die entsprechende Summe der Kantengewichte. So hat im ungewichteten Graphen aus Abbildung 1 der Pfad Altdorf – Weiler – Ziegelstein – Neustadt die Länge 3, im Radnetz hat dieser Pfad die Länge 35. Ein Pfad heißt geschlossen oder auch **zyklisch**, wenn Start- und Zielknoten identisch sind.



ungerichteter Graph ohne Bewertung (Abb. 1)



Radnetz (Abb. 2)

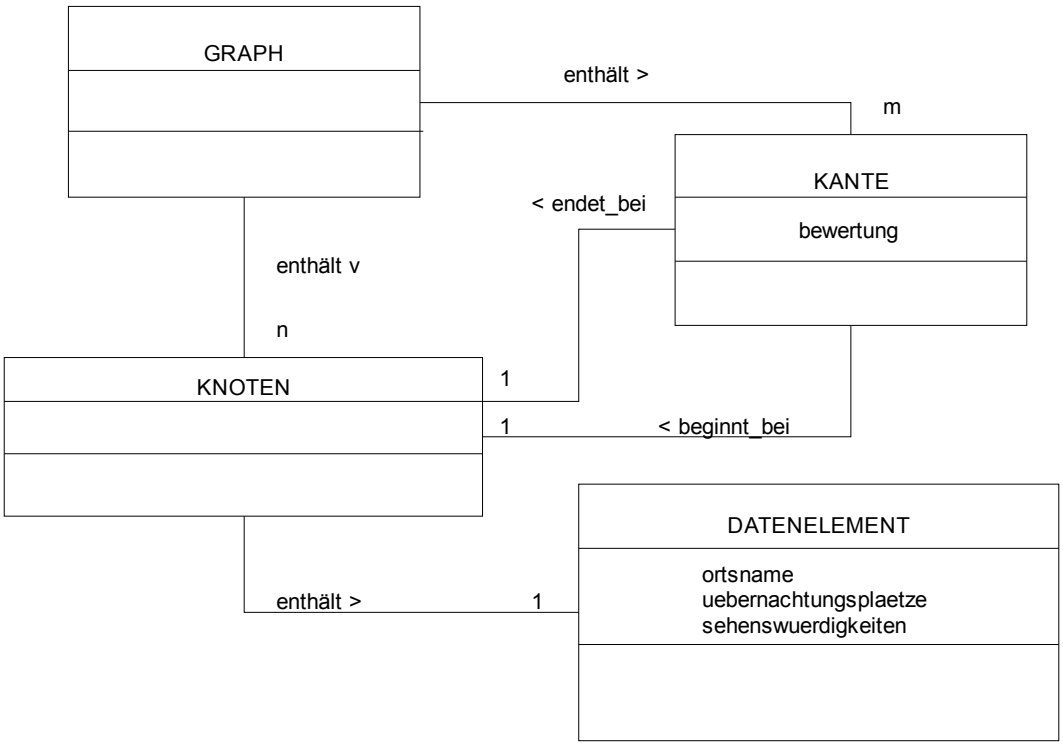
1.3.1.2 Die Struktur des Graphen als Klassendiagramm

Der erste Schritt zur Umsetzung eines Graphen in ein informatisches System ist die Entwicklung eines geeigneten Klassendiagramms. Neben der Klasse GRAPH gibt es auf jeden Fall die Klassen KNOTEN und KANTE und jeweils eine 1:n-Beziehung zwischen GRAPH und KNOTEN bzw. GRAPH und KANTE. Darüber hinaus hat die Klasse KANTE je eine 1:n-Beziehung zu ihrem Anfangs- bzw. Endknoten. Bei ungerichteten Graphen wird nicht zwischen Anfangs- und Endknoten unterschieden. Je nachdem ob der Graph bewertet oder unbewertet ist, hat die Klasse KANTE ein Attribut *bewertung* oder nicht. Das Klassendiagramm in Abbildung 3 modelliert den Graphen unmittelbar laut seiner Definition.

Darüber hinaus werden je nach Anwendung bei einem Knoten nicht nur seine Bezeichnung, sondern weitere Informationen gespeichert. So wäre es für das Beispiel Radnetz im Sinn einer touristisch interessanten Tourenplanung sinnvoll, zusätzlich zu den Namen der Ortschaften auch noch die Bettenanzahl für Übernachtungen sowie die Sehenswürdigkeiten des Ortes zu speichern. Solche Informationen sollten nicht im Knoten selbst, sondern in einem vom Knoten referenzierten Datenelement gespeichert werden, um der schon aus den Ausführungen zu den Datenstrukturen LISTE und BAUM bekannten Trennung von Struktur des Graphen und den Daten der Anwendung zu genügen. Wir benötigen also noch eine Klasse DATENELEMENT, die der Aufgabenstellung entsprechende Attribute enthält.

Dieses Klassendiagramm stellt ein Modell für die Struktur des Graphen dar und liegt noch nicht notwendig nahe an einer möglichen Implementierung. In der Schule werden für die Implementierung des Graphen weitere Vereinfachungen vorgenommen (siehe Kapitel 1.3.2). So kann von einer statischen Struktur des Graphen ausgegangen werden, d. h. die Knoten und Kanten ändern sich nach dem Graphaufbau zur Laufzeit der Anwendung nicht mehr.

Ein vollständiges Klassendiagramm, das so auch implementiert wird und alle nötigen Methoden enthält, wird im Kapitel 1.3.4 gezeigt.



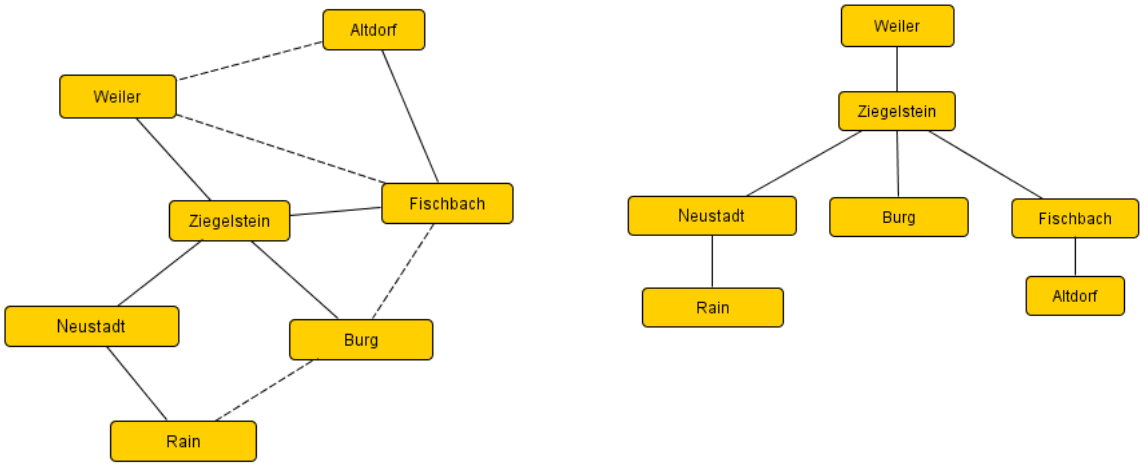
Klassendiagramm allgemeine Version (Abb. 3)

1.3.1.3 Die Datenstruktur Graph als Verallgemeinerung des Baumes

Die bereits behandelte Datenstruktur Baum kann als ein sehr spezieller Typ von Graphen aufgefasst werden. Bei einem Binärbaum beispielsweise handelt es sich um einen Graphen, bei dem jeder Knoten mit maximal drei anderen Knoten (maximal ein Vorgänger, maximal zwei Nachfolger) verbunden ist und der keine zyklischen Pfade enthält. Im Allgemeinen jedoch enthält ein Graph zyklische Pfade und kann daher nicht mithilfe eines Baumes modelliert werden.

Für das Beispiel Radnetz könnte ein sehr sparsamer Landrat in Bezug auf den Unterhalt der Radwege folgende Frage stellen: Welche Straßen brauche ich mindestens, damit die Bürger von einem beliebigen Ort zu jedem anderen Ort gelangen können? Alle „überflüssigen“ Straßen werden nicht mehr repariert. Es wäre dann nicht mehr möglich, „im Kreis“ zu fahren.

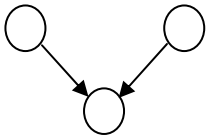
Für den ungerichteten, unbewerteten Graphen aus Abbildung 1 könnte dies dazu führen, dass beispielsweise nur noch alle in der nachfolgenden Abbildung durchgezogen gezeichneten Straßen unterhalten werden. Die jetzt entstehende Struktur ist hierarchisch: Beginnen wir mit einem beliebigen Knoten, z. B. mit Weiler, so gibt es nur mehr einen eindeutigen Weg zu jedem anderen Knoten. Es gibt keine zyklischen Pfade mehr, da sonst eine Straße überflüssig wäre. Eine solche Struktur ist uns bereits als Baum bekannt. Der ausgewählte Knoten Weiler wird zur Wurzel eines Baumes.



Baum in zwei Darstellungen (Abb. 4)

Für die Auswahl der Straßen, die weiterhin unterhalten werden (durchgezogene Linien), gibt es natürlich mehrere, verschiedene Möglichkeiten. Es muss nur sichergestellt sein, dass weiterhin von jedem Ort zu jedem anderen eine Verbindung besteht. Wählt man nun einen beliebigen Knoten als Startpunkt, so kann man stets einen Baum mit diesem Startpunkt als Wurzel zeichnen. Dieser Baum sieht dann natürlich anderes aus als der in obiger Abbildung.

Reicht auch bei einem gerichteten Graphen die Zyklenfreiheit dafür aus, dass ein Baum vorliegt? Folgendes Beispiel zeigt, dass dies nicht der Fall ist, da ein Knoten zwei Vorgänger besitzt.



Es gilt: Ein ungerichteter, unbewerteter und zusammenhängender Graph ist genau dann ein Baum, wenn er zyklenfrei ist. Ein gerichteter und zusammenhängender Graph ist dann ein Baum, wenn jeder Knoten maximal einen Vorgänger besitzt.

1.3.2 Umsetzung in eine Programmiersprache

1.3.2.1 Adjazenzmatrix

Als erstes Beispiel soll nun der gerichtete und bewertete Graph aus Abbildung 2 in eine Programmiersprache umgesetzt werden. Hierfür nennt die Fachliteratur mehrere Alternativen, wie z. B. die Verwendung von Listen für die Kanten bzw. Knoten oder die Verwendung einer sogenannten Adjazenzmatrix. Der Lehrplan sieht die Implementierung mithilfe der Adjazenzmatrix vor. Gleichzeitig ist die Verwendung von Listen jedoch eine gebräuchliche Alternative und ermöglicht eine Vertiefung und Vernetzung mit früheren Lerninhalten; diese Art der Umsetzung würde sich daher gut für eine Projektarbeit eignen.

Für die Schüler und Schülerinnen ist der Begriff „Matrix“ an dieser Stelle völlig neu. Allgemein handelt es sich hier um eine Tabelle aus (beispielsweise) Zahlen mit Zeilen und Spalten, wie sie aus Jahrgangsstufe 9 im Zusammenhang mit Rechenblättern bereits bekannt ist. Zur Beschreibung des Radnetzes wird nun die sogenannte Adjazenzmatrix erstellt. Die Zeilen und Spalten einer Matrix sind durchnummeriert. Deshalb muss man jedem Knoten eine Nummer zuordnen. Dies wird durch ein eindimensionales Feld zur Speicherung der Knoten realisiert:

Feldelement	Altdorf	Fischbach	Weiler	Ziegelstein	Burg	Neustadt	Rain
Index	0	1	2	3	4	5	6

Die Kanteninformation kann dann zweidimensional dargestellt werden:

		Altdorf	Fischbach	Weiler	Ziegelstein	Burg	Neustadt	Rain
		0	1	2	3	4	5	6
Altdorf	0		20	15				
Fischbach	1	10		10				
Weiler	2		25		5			
Ziegelstein	3		15			20	15	
Burg	4		15		5			
Neustadt	5							20
Rain	6					25		

Zur Umsetzung in eine Programmiersprache müssen noch die leeren Zellen gefüllt werden. Auf der Diagonale kann einfach eine Null eingetragen werden, da die Fahrt z. B. von Altdorf nach Altdorf keine Zeit benötigt. Gibt es zwischen zwei Ortschaften keine Verbindung, so muss dies mit einem Wert markiert werden, der im Anwendungszusammenhang nicht vorkommt, damit es zu keiner Verwechslung kommen kann. Hier kann der Wert –1 verwendet werden, da es negative Fahrzeiten nicht gibt. Genauso möglich wäre auch der Wert ∞ für eine unendlich lange Fahrzeit. Insgesamt erhalten wir folgendes zweidimensionales Feld mit 7 Zeilen und 7 Spalten, das auch als 7x7-Matrix bezeichnet wird und die Adjazenzmatrix für das Radnetz darstellt:

0	20	15	-1	-1	-1	-1
10	0	10	-1	-1	-1	-1
-1	25	0	5	-1	-1	-1
-1	15	-1	0	20	15	-1
-1	15	-1	5	0	-1	-1
-1	-1	-1	-1	-1	0	20
-1	-1	-1	-1	25	-1	0

Definition:

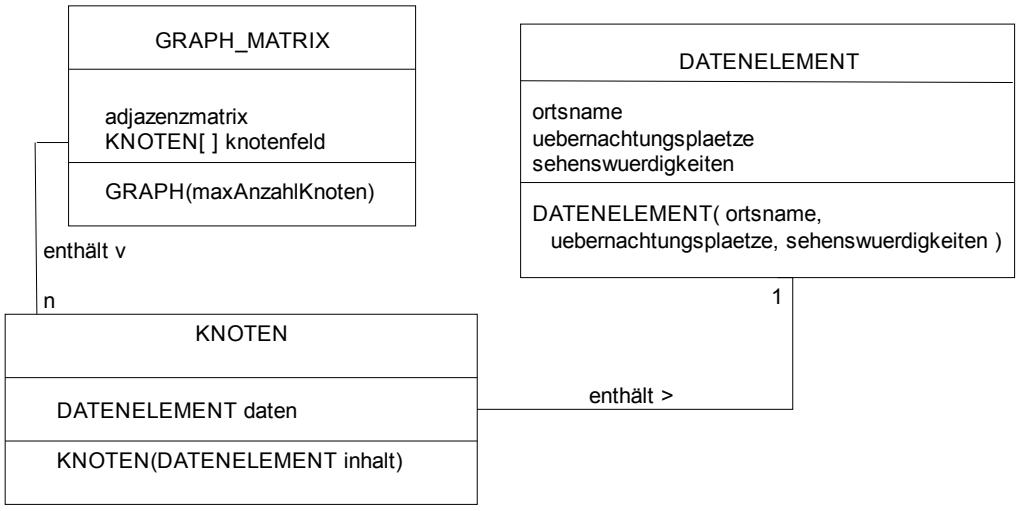
Die **Adjazenzmatrix** eines Graphen ist eine quadratische Matrix (Tabelle), deren Anzahl an Spalten bzw. Zeilen mit der Anzahl der Knoten des Graphen übereinstimmt. Die Einträge in den Zellen der Matrix geben wieder, ob zwischen zwei Knoten eine Kante existiert und welche Bewertung diese gegebenenfalls hat.

Für einen gerichteten und bewerteten Graphen wird das Matrixelement der i-ten Zeile und j-ten Spalte als die Bewertung der Kante mit dem Anfangsknoten der Nummer i (Index i-1) und dem Endknoten der Nummer j (Index j-1) interpretiert. Gibt es zwischen zwei Knoten keine Kante, so wird dies durch einen vorher vereinbarten Wert wie z. B. –1 oder ∞ gekennzeichnet. In obigem Beispiel bedeutet der Wert 15 in der 1. Zeile und 3. Spalte, dass die Kante vom Knoten mit der Nummer 1 (Index 0, Altdorf) zum Knoten mit der Nummer 3 (Index 2, Weiler) die Bewertung 15 hat. In den Anwendungszusammenhang zurückübersetzt heißt dies, dass die Fahrzeit von Altdorf nach Weiler mit dem Rad 15 min beträgt.

Eine Adjazenzmatrix hat folgende Eigenschaften:

- Ist der Graph unbewertet, so muss in der Adjazenzmatrix nur gespeichert werden, ob zwischen zwei Knoten eine Kante existiert oder nicht. Hierfür genügt ein boolescher Wert.
- Für einen ungerichteten Graphen ist die Adjazenzmatrix symmetrisch.

Die Erstellung der Adjazenzmatrix ist ein weiterer Schritt in Richtung der Implementierung eines Graphen, bei der die Kanten durch die Adjazenzmatrix repräsentiert werden. Zur Unterscheidung vom allgemeinen Klassendiagramm in Abbildung 3 wird die Klasse GRAPH nun als GRAPH_MATRIX bezeichnet, die ein Attribut *adjazenzmatrix* enthält. Im Klassendiagramm der Abbildung 5 ist ein der Implementierung angemessenes Klassendiagramm dargestellt, das nun auch mögliche Konstruktoren, aber noch keine weiteren Methoden enthält.



Klassendiagramm für die Implementierung mit Adjazenzmatrix (Abb. 5)

In Java könnte ein unveränderlicher Graph folgendermaßen „auf die Schnelle“ implementiert werden:

```
String[] knotenfeld = {"Altdorf","Fischbach","Weiler",
                       "Ziegelstein","Burg","Neustadt","Rain"};

int[][] adjazenzmatrix = {  { 0, 20, 15, -1, -1, -1, -1},
                            {10,  0, 10, -1, -1, -1, -1},
                            {-1, 25 , 0,  5, -1, -1, -1},
                            {-1, 15, -1,  0, 20, 15, -1},
                            {-1, 15, -1,  5,  0, -1, -1},
                            {-1, -1, -1, -1, -1,  0, 20},
                            {-1, -1, -1, -1, 25, -1,  0}};
```

Im Rahmen von Aufgaben auch in Prüfungssituationen stellt diese Implementierung eine Möglichkeit dar, mit geringem Zeitaufwand einen bestimmten Graphen anzugeben. Aus Sicht eines qualitativ hochwertigen Programmierstils wird in diesem Kontext vorsätzlich die Trennung von Struktur (Graph) und Daten (Ortsnamen) missachtet; es können weder Orte hinzugefügt noch gelöscht werden und für die Knoten können keine zusätzlichen Informationen (Bettenanzahl, Sehenswürdigkeiten) gespeichert werden.

1.3.2.2 Die Klassen DATENELEMENT, KNOTEN und GRAPH_MATRIX

Nach diesen Vorbereitungen können die Klassen gemäß Abbildung 5 implementiert werden.

Für das Radnetz erhalten wir nachfolgenden Programmcode für die Klasse DATENELEMENT in JAVA. Die Methode *ausgeben* sorgt für eine angemessene Art und Weise der Darstellung eines Datenelementes am Bildschirm.

```
class DATENELEMENT
{
    private String ortsname;
    private int uebernachtungsplaetze;
    private String sehenswuerdigkeiten;

    DATENELEMENT(String ortsname, int uebernachtungsplaetze,
                 String sehenswuerdigkeiten )
    {
        this.ortsname = ortsname;
        this.uebernachtungsplaetze = uebernachtungsplaetze;
        this.sehenswuerdigkeiten = sehenswuerdigkeiten;
    }

    void ausgeben()
    {
        System.out.println( " Im Ort " + ortsname + " gibt es " +
                            uebernachtungsplaetze + " Betten. ");
        System.out.println( " Sie können hier folgendes erleben: " +
                            sehenswuerdigkeiten );
    }
}
```

Die Klasse KNOTEN gehört zur Struktur des Graphen und ist unabhängig von der konkreten Ausprägung des Graphen im Anwendungszusammenhang. Die folgende Implementierung der Klasse KNOTEN müsste in einem anderen Anwendungszusammenhang (z. B. im Rahmen

der Implementierung eines Telekommunikationsnetzes) nicht geändert werden; bis auf eine etwaige Ergänzung von Methoden ist sie weiterhin einsetzbar.

```
class KNOTEN
{
    private DATENELEMENT daten;

    KNOTEN(DATENELEMENT inhalt)
    {
        daten = inhalt;
    }

    DATENELEMENT datenLiefern()
    {
        return daten;
}   }
```

Die Klasse GRAPH_MATRIX realisiert den Graphen mit seinen Knoten und Kanten. Sie enthält als Attribute ein Feld für die Knoten und ein zweidimensionales Feld für die Adjazenzmatrix. Zur einfacheren Programmierung wird die aktuell im Graphen enthaltene Anzahl an Knoten in einem weiteren Attribut gespeichert.
Im Konstruktor wird die Größe der Felder festgelegt. In der Schule genügt das Arbeiten mit statischen Graphen, d. h., der Graph wird mit einer festen Anzahl von möglichen Knoten erzeugt, die für die Laufzeit der Anwendung ausreichend ist. Zusätzlich werden im Konstruktor die Diagonalelemente der Adjazenzmatrix mit null, die restlichen Matrixelemente mit –1 vorbelegt.

```
class GRAPH_MATRIX
{
    private KNOTEN[] knotenfeld;
    private int[][] adjazenzmatrix;
    private int anzahlKnoten = 0;

    // Konstruktor
    GRAPH_MATRIX(int maxAnzahlKnoten)
    {
        knotenfeld = new KNOTEN[maxAnzahlKnoten];
        adjazenzmatrix = new int[maxAnzahlKnoten][maxAnzahlKnoten];

        //Die Adjazenzmatrix wird mit den Werten 0 auf der Diagonalen,
        //sonst mit den Werten –1 („keine Kante“) vorbelegt.
        for(int i=0; i < maxAnzahlKnoten; i=i+1)
        {
            for(int j=0; j < maxAnzahlKnoten; j=j+1)
            {
                if(i == j)
                {
                    adjazenzmatrix[i][j] = 0;
                }
                else
                {
                    adjazenzmatrix[i][j] = -1;
                }
            }
        }
    } //Ende Konstruktor
    …
}
```

Zum Aufbau eines konkreten Graphen werden nun noch Methoden zum Einfügen von Knoten und Kanten benötigt. Wird ein Knoten eingefügt, so geschieht dies auf dem ersten freien Platz im Knotenfeld; in der Adjazenzmatrix werden die entsprechenden Zellen mit Werten belegt. Der fertig konstruierte Graph braucht für die Anwendung in der Schule nicht mehr verändert zu werden, es wird auf diesem Graphen gearbeitet. Ein dynamisches Einfügen von Knoten und Kanten ist nicht vorgesehen. Das Löschen von Knoten ist im Anwendungsfall nicht erforderlich und kann weggelassen werden.

Trotzdem wurden für das Radnetz die Methoden für das Löschen von Knoten bzw. Kanten programmiert, um dem Leser einen Einblick in eine mögliche Realisierung anzubieten. Relativ leicht ist das Löschen von Kanten möglich, hier muss nur der betreffende Eintrag in der Adjazenzmatrix verändert werden. Schwieriger ist das Löschen eines Knotens: Hier muss auch die zugehörige Zeile und Spalte in der Adjazenzmatrix gelöscht werden, was das Verschieben der nachfolgenden Zeilen bzw. Spalten erfordert. Sollte es aufgrund einer Aufgabenstellung z. B. im Rahmen eines Projektes erforderlich sein, den Graphen zur Laufzeit zu verändern, so wäre eine Implementierung mithilfe der Adjazenzmatrix unangemessen. Geeigneter wäre dann z. B. die Verwendung von Adjazenzlisten.

Zum Testen des Beispiels Radnetz muss ein Objekt der Klasse GRAPH erzeugt werden. Dies erfordert, zuerst die Datenelemente und die Knoten zu erzeugen und anschließend alle Kanten einzugeben. Hierfür wurde für das obige Beispiel eine Testklasse implementiert. Manche Entwicklungsumgebungen unterstützen das automatische Aufzeichnen einer Testeinheit, so kann z. B. mit BlueJ eine so genannte „Testunit“ erzeugt werden, die das Aufrufen von Methoden als Programmtext speichert.

```
class TEST{
    //Für Testzwecke darf auf den Graphen selbst zugegriffen werden.
    public GRAPH_MATRIX radnetz;

    //Konstruktor
    TEST()
    {
       radnetz = new GRAPH_MATRIX(20);

       //Zu Testzwecken werden die ersten Knoten fest angegeben.
       //Die Rückgabewerte bezeichnen den Index des Knotens.
            int a = radnetz.knotenHinzufuegen(new DATENELEMENT("Altdorf",
                                                    100, "Marktplatz"));
            int f = radnetz.knotenHinzufuegen(new DATENELEMENT("Fischbach",
                                              80, "Fischspezialitaeten"));
            int w = radnetz.knotenHinzufuegen(new DATENELEMENT("Weiler",
                                                      20, "Naturbad"));
       //Weitere Knoten hinzufügen

       //Zu Testzwecken werden Kanten zwischen obigen Knoten festgelegt.
            radnetz.kanteHinzufuegen(a,f,20);
            radnetz.kanteHinzufuegen(a,w,15);
            radnetz.kanteHinzufuegen(f,a,10);
            radnetz.kanteHinzufuegen(f,w,10);
       //Weitere Kanten hinzufügen
    }

    void ausgeben()
    {
        radnetz.knotenAusgeben();
        radnetz.matrixAusgeben();
}   }
```

1.3.3 Verfahren für den Graphendurchlauf

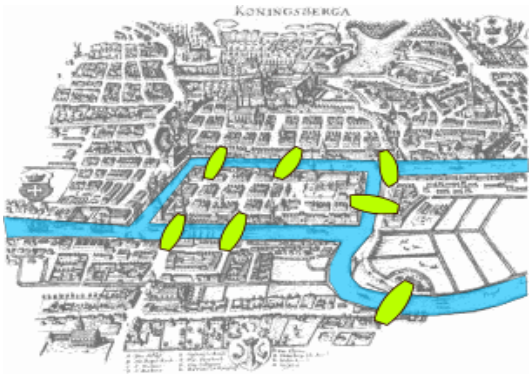
Nach der Implementierung eines Graphen wenden wir uns grundsätzlichen Fragestellungen für Graphen zu, die in mehreren Varianten auftreten können. Für das Radnetz wären dies beispielsweise:

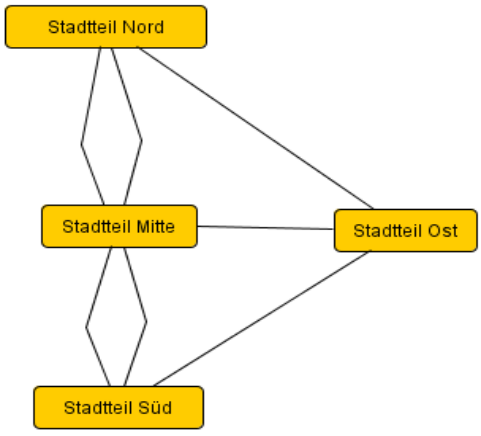
- Welchen Weg gibt es von einem Startort A zu einem Zielort B?
  - Dieser Weg soll über möglichst wenig Zwischenknoten führen. Dies wäre z. B. für Familien interessant, die auf ihrer Route aufgrund des meist hohen Verkehrs möglichst wenig Orte passieren möchten.
  - Dieser Weg soll möglichst kurz sein. Hierbei wird die Entfernung als Summe der Gewichtungen der durchlaufenen Kanten verstanden. Für das Radnetz ist daher der Weg mit der kürzesten Fahrzeit, nicht der mit der kürzesten Wegstrecke gemeint.
- Wie kann ausgehend von einem Startort A eine Rundreise über alle Orte erfolgen, die am Zielort B endet?
  - Hierbei soll jeder Ort mindestens einmal durchfahren werden.
  - Die Rundreise soll möglichst kurz sein.

In diesem Zusammenhang treten in der Literatur zwei bekannte historische Fragen auf. Die Beantwortung dieser Fragen geht weit über das in der Schule geforderte Niveau hinaus, jedoch werden sie aus Gründen der Allgemeinbildung an dieser Stelle angesprochen. Hierbei kann deutlich werden, dass Graphen schon vor der Entwicklung leistungsfähiger Computer ihren Platz in der Wissenschaft hatten.

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 genau einmal überquert, und wenn ja, ob auch ein Rundweg möglich ist, bei dem man wieder zum Ausgangspunkt gelangt.

Übertragen in die Sprache der Graphen folgt für die Lösung dieses Problems, dass der Weg jede Kante genau einmal durchlaufen muss. Allerdings treten hier bei der graphischen Darstellung zwischen zwei Knoten (Stadtviertel) mehrere Kanten (Brücken) auf, was über die Standarddefinition eines Graphen hinausgeht; hierfür würde ein sogenannter Multigraph benötigt.





Königsberger Brücken (Abb. 6)

Problem des Handlungsreisenden (engl. traveling salesman problem): Die Aufgabe besteht darin, eine Reihenfolge für den Besuch mehrerer Orte so zu wählen, dass die gesamte Reisestrecke des Handlungsreisenden nach der Rückkehr zum Ausgangsort möglichst kurz ist. Seit seiner ersten Erwähnung als mathematisches Problem im Jahre 1930 haben sich viele Forscher damit befasst und neue Optimierungsverfahren daran entwickelt und erprobt. Als Unterproblem tritt es z. B. bei der Verteilung von Waren, bei der Planung von Touren eines Kunden- oder Pannendienstes oder bei der Genom-Sequenzierung auf, dabei werden die Begriffe „Stadt“ und „Entfernung“ sinngemäß übertragen.

Eine detaillierte Ausarbeitung der oben genannten Fragestellungen würde den schulischen Rahmen sprengen, die prinzipiellen Gedanken über mögliche Lösungsverfahren sind jedoch auch für den (Informatik-)Unterricht interessant.

Bekannte Verfahren zum Graphendurchlauf sind die Tiefen- oder Breitensuche. Beide Verfahren lösen z. B. folgende Aufgabe: Ausgehend von einem Startknoten sollen alle erreichbaren Knoten des Graphen mindestens einmal besucht werden. Lediglich ein Verfahren wird vom Lehrplan gefordert. Im Folgenden konzentrieren wir uns auf die algorithmisch einfachere Tiefensuche. Dieses Verfahren ermöglicht auch eine sogenannte Suche: Soll ein Weg ausgehend von einem Startknoten zu einem bestimmten Zielknoten gefunden werden, kann der Graphendurchlauf leicht an diesem Knoten abgebrochen werden. Allerdings hat der durchlaufene Pfad keine besonderen Eigenschaften: Weder muss es der kürzeste Weg sein, noch ist abgesichert, dass alle Kanten mindestens einmal besucht werden.

Die Tiefensuche basiert auf folgender rekursiver Strategie: Ausgehend von einem Knoten S wird ein weiterer direkt erreichbarer Knoten T besucht. Bevor die Tiefensuche zu S zurückkehrt, werden erst weitere direkt erreichbare Knoten von T mit derselben Strategie besucht („zuerst in die Tiefe gehen“). Erst wenn T keine unbesuchten, direkt erreichbaren Knoten mehr hat, kehrt die Tiefensuche zu S zurück und fährt dann mit einem anderen von S aus direkt erreichbaren Knoten fort.

Bevor der Algorithmus allgemein formuliert und implementiert werden kann, wird dieses Verfahren am Beispiel Radnetz erläutert. Zur Vereinfachung werden die Ortsnamen mit ihren Anfangsbuchstaben abgekürzt und die Bewertungen der Kanten weggelassen. Die Bildfolge Abbildung 7 ff. stellt die nachfolgend beschriebenen Schritte dar. Farblich werden hierin die schon besuchten Knoten dunkel markiert.

1. Schritt: Startknoten sei der Knoten A. Damit wird dieser Knoten als besucht markiert.

2. Schritt: Ein weiterer von A aus direkt erreichbarer Knoten wird besucht. Dieser Knoten ist in der Regel nicht eindeutig bestimmt, hier kämen z. B. W oder F infrage. An dieser Stelle wird durch die in 1.3.2.1 festgelegte Adjazenzmatrix eine Reihenfolge vorgegeben: Die zum aktuellen Knoten A gehörende Zeile wird nach dem ersten positiven Eintrag durchsucht. Damit ist F der nächste von A aus direkt erreichbare Knoten. Nun wird F besucht. Die Tiefensuche kehrt dann nicht zurück zu A, sondern sucht ausgehend von F nach weiteren direkt erreichbaren, unbesuchten Knoten. Daher spricht man von „in die Tiefe gehen“.

3. Schritt: Der Knoten F hat als einzigen direkt erreichbaren Knoten, der noch nicht besucht wurde, den Knoten W. W wird besucht. Die Tiefensuche sucht nun ausgehend von W.

4. Schritt: W hat nur Z als direkt erreichbaren, unbesuchten Knoten. Z wird besucht.

5. Schritt: Z hat nun noch B und N als direkt erreichbare, unbesuchte Knoten. Laut Adjazenzmatrix wird nun B besucht.

6. Schritt: B hat keine unbesuchten, direkt erreichbaren Knoten mehr. Die Tiefensuche kehrt zurück zu Z.

7. Schritt: Z hat noch N als direkt erreichbaren, unbesuchten Knoten. N wird besucht.

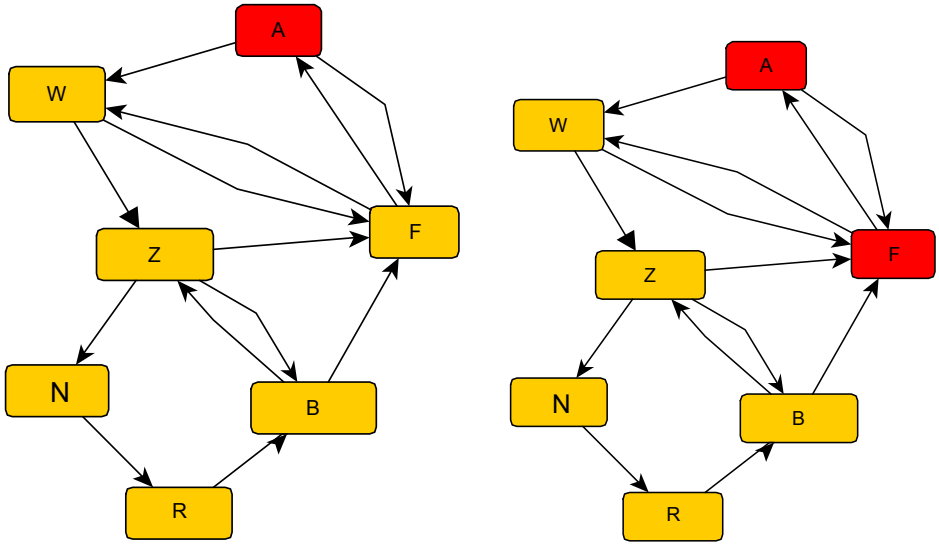
8. Schritt: N hat nur R als unbesuchten, direkt erreichbaren Knoten. R wird besucht.

Die Tiefensuche kehrt über N – Z – W – F – A zurück zum Startknoten und endet, da von keinem dieser Knoten noch unbesuchte Knoten direkt erreicht werden können.

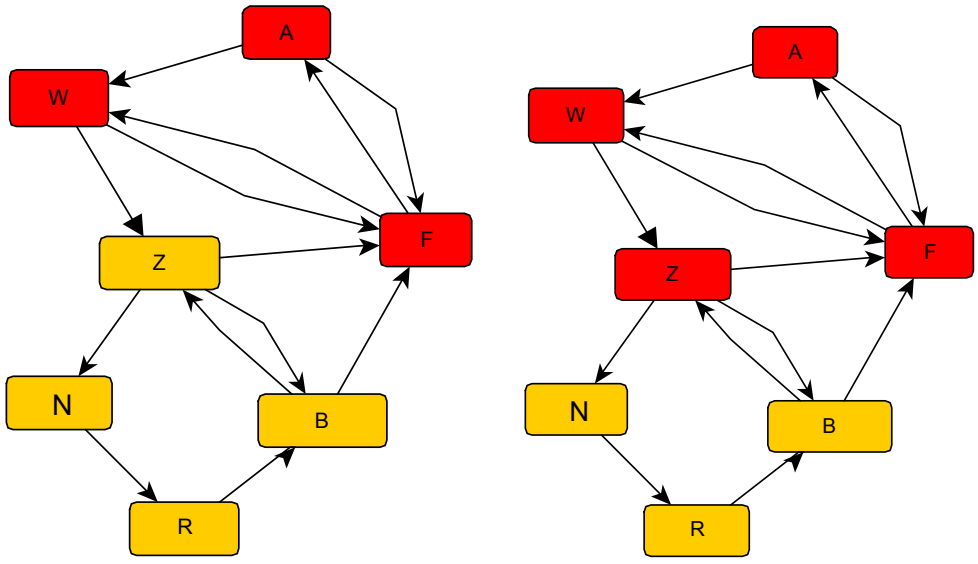
Insgesamt durchläuft die Tiefensuche folgenden Pfad:

A – F – W – Z – B – Z – N – R – N – Z – W – F – A.

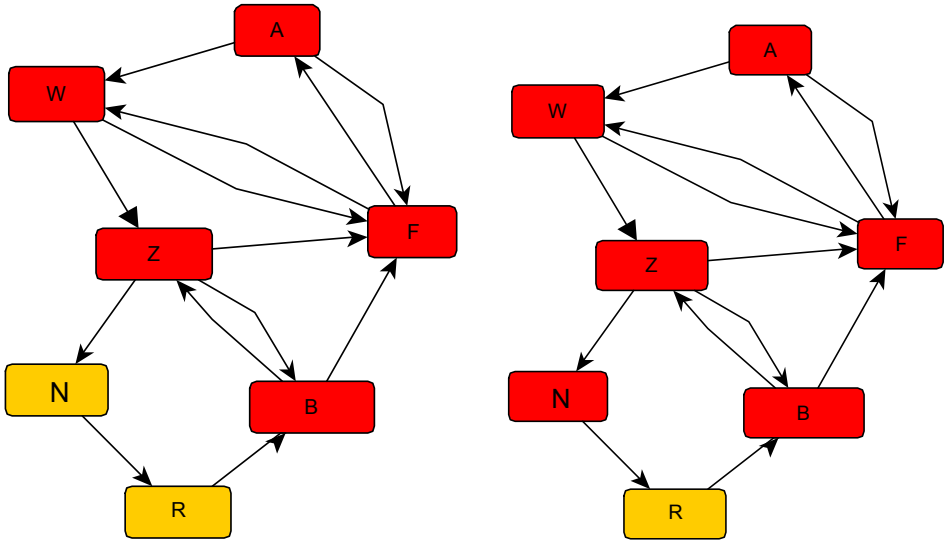
Im Beispiel Radnetz wird ein Knoten nur dann am Bildschirm ausgegeben, wenn er das erste Mal besucht wurde, d. h., die Rückkehrwege der Tiefensuche werden nicht angezeigt. So ergibt sich als Reihenfolge der besuchten Knoten A – F –W – Z – B – N – R.



Tiefensuche Schritt 1 und 2 (Abb. 7)



Tiefensuche Schritt 3 und 4 (Abb. 8)



Tiefensuche Schritt 5 und 7 (Abb. 9)

Als zweites Beispiel wird die Tiefensuche mit dem Startknoten Z durchgeführt. Dann ergibt sich beispielsweise Z – F – A – W – A – F – Z – B – Z – N – R – N – Z.

Nach diesen Vorbereitungen kann der **Algorithmus der Tiefensuche** beschrieben werden:

1. Markiere alle Knoten des Graphen als „unbesucht“.
2. Führe die Tiefensuche für den Startknoten durch gemäß Punkt 3.
3. Führe die Tiefensuche für einen Knoten K rekursiv durch:
   a. Markiere K als besucht.
   b. Solange K einen noch unbesuchten, direkt erreichbaren Knoten T hat, führe Schritt 3 der Tiefensuche für T durch.

Bemerkung: Nur bei sogenannten zusammenhängenden Graphen erreicht die Tiefensuche alle Knoten ausgehend von einem Startknoten. Die Frage, wie überprüft werden kann, ob ein Graph zusammenhängend ist oder nicht, geht über das schulische Niveau hinaus und würde weg vom Thema führen. Gegebenenfalls arbeitet die Tiefensuche nur auf dem Teilgraphen, der alle vom Startknoten aus erreichbaren Knoten enthält.

Für die Implementierung der Tiefensuche im Beispiel Radnetz sind noch folgende Schritte zu tun:

- Die Klasse KNOTEN erhält zusätzlich ein boolesches Attribut *markierung*, um den Zustand „besucht“ oder „nicht besucht“ während des Graphendurchlaufs speichern zu können. Außerdem werden Methoden benötigt, um dieses Attribut zu lesen und zu setzen.
- Die Klasse GRAPH_MATRIX wird um zwei Methoden ergänzt:
  - *tiefensucheStarten(int startKnotenNr)*
  - *tiefensucheDurchfuehren(int knotenNr)*

Der Aufruf der öffentlichen Methode *tiefensucheStarten* legt den Startknoten fest und markiert alle Knoten des Graphen als unbesucht. Dann erfolgt der Aufruf der rekursiven Methode *tiefensucheDurchfuehren*, die Schritt 3 des obigen Algorithmus implementiert.

```
//Methode der Klasse GRAPH_MATRIX
Methode tiefensucheStarten(startKnotenNr)
    wiederhole für alle Knoten
            setze Markierung des Knotens auf den Wert für unbesucht
    endeWiederhole
    wenn startKnotenNr >=0 und startKnotenNr < anzahlKnoten dann
            tiefensucheDurchfuehren(startKnotenNr)
    endeWenn

//Methode der Klasse GRAPH_MATRIX
Methode tiefensucheDurchfuehren(knotenNr)
    Markiere den Knoten mit der Nummer knotenNr als besucht
    //Gib evtl. Informationen über den Knoten aus
    //Tiefensuche für den nächsten direkt erreichbaren unbesuchten Knoten aufrufen
    wiederhole für j = 0 bis j < anzahlKnoten
        wenn adjazenzmatrix[knotenNr][j] > 0          //direkt erreichbarer Knoten
              und der Knoten mit der Nummer j wurde noch nicht besucht
        dann
              tiefensucheDurchfuehren(j)              //rekursiver Aufruf
        endeWenn
    endeWiederhole
```

In JAVA könnte die Implementierung in der Klasse GRAPH_MATRIX wie folgt aussehen:

```
//Methode zum Starten der Tiefensuche
void tiefensucheStarten(int startKnotenNr)
{
    //Am Anfang alle Knoten auf unbesucht setzen
    for(int i=0; i < anzahlKnoten; i++)
    {
        knotenfeld[i].markierungSetzen(false);
    }

    //Beginn der Tiefensuche mit dem angegebenen Startknoten
    if (startKnotenNr >= 0 && startKnotenNr < anzahlKnoten)
    {
        tiefensucheDurchfuehren(startKnotenNr);
    }
}
```

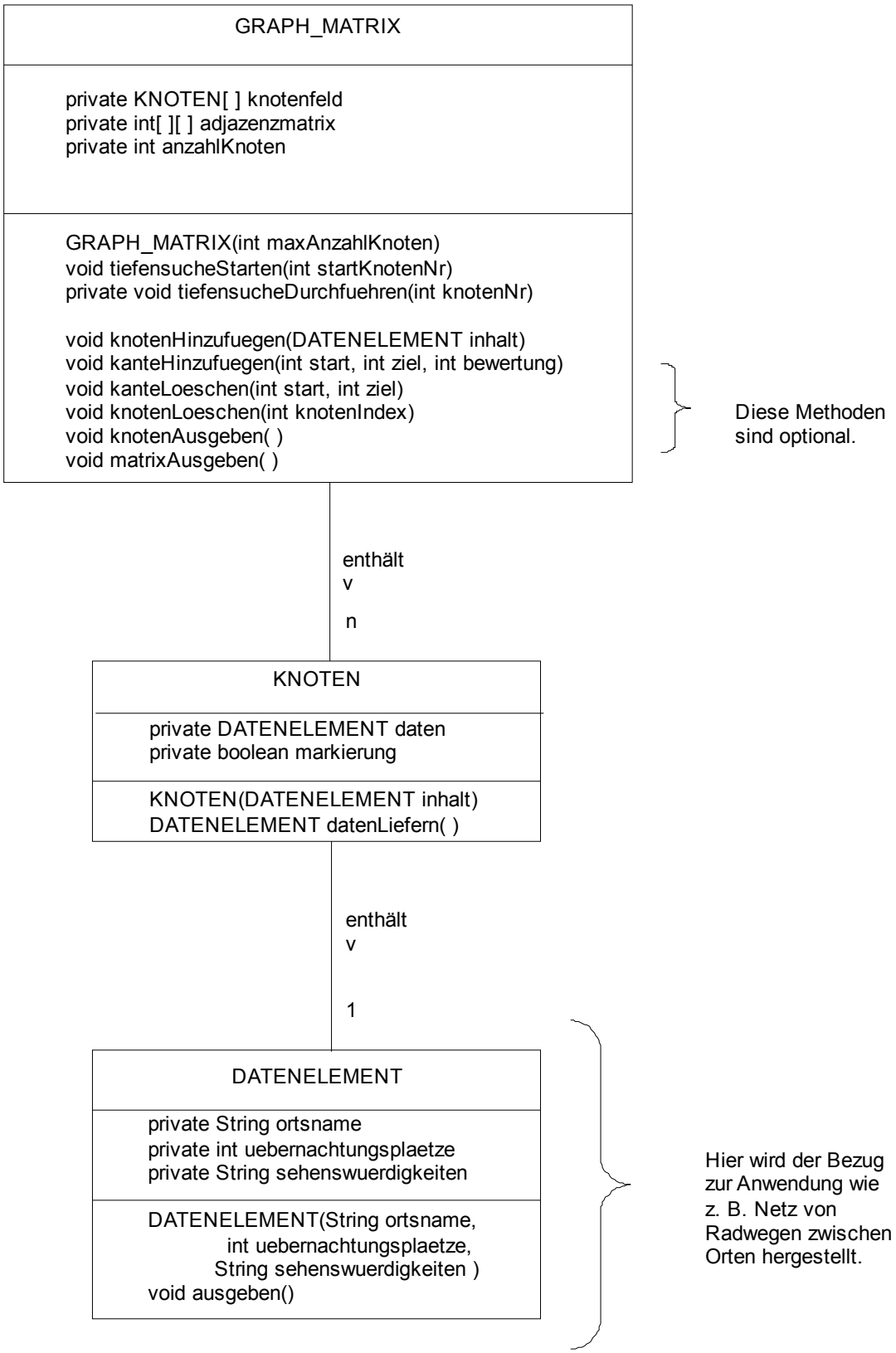
```
//Rekursive Methode zum Durchfuehren der Tiefensuche
private void tiefensucheDurchfuehren(int knotenNr)
{
   //Knoten als besucht markieren und auf der Konsole ausgeben.
   knotenfeld[knotenNr].markierungSetzen(true);
   System.out.println("Jetzt wurde der folgende Ort besucht:" );
   knotenfeld[knotenNr].datenLiefern().ausgeben();

   //Für benachbarte Knoten die Tiefensuche erneut aufrufen.
   for( int j=0; j < anzahlKnoten; j++)
   {
       if (adjazenzmatrix[knotenNr][j] > 0
           && !(knotenfeld[j].markierungGeben()))
       {
           //Rekursiver Aufruf der Tiefensuche für den naechsten
           //Knoten
           tiefensucheDurchfuehren(j);
       }
    }
}
```

Diese Implementierung ist insoweit allgemein verwendbar, als sie keine Annahmen über den Anwendungszusammenhang macht. Es werden allein die Methoden *markierungSetzen, markierungGeben* bzw. *datenLiefern* der Knoten verwendet. Die Implementierung der Methode *ausgeben* ist in die Klasse DATENELEMENT ausgelagert. Für ein anderes Anwendungsbeispiel könnten die Klassen für Graph und Knoten wieder verwendet werden.

Eine Implementierung ist im BlueJ-Projekt Graph_Radnetz zu finden.

1.3.4 Klassendiagramm der Klasse GRAPH_MATRIX

Alle nötigen Attribute und Methoden sind nun bekannt und werden in folgendem erweiterten Klassendiagramm dargestellt:



Vollständiges Klassendiagramm (Abb. 10)

1.3.5 Ausblick

Wie bereits an einigen Stellen deutlich wurde, gibt es vielfältige Möglichkeiten, das Thema Graphen zu vertiefen. Im Rahmen einer arbeitsteiligen Projektarbeit (vgl. Lehrplan Inf 11.2.2; Kapitel 2) könnten sich verschiedene Gruppen mit jeweils einem der nachfolgenden Aspekte befassen und dies am Ende den anderen Gruppen vorstellen.

a) Zur Darstellung von Graphen können auch Adjazenzlisten verwendet werden. Diese speichern für jeden Knoten die von ihm aus direkt erreichbaren Knoten in einer Liste ab. Dieser Ansatz eignet sich gut als Anwendung der Datenstruktur Liste.

b) Für den Graphendurchlauf gibt es neben der besprochenen Tiefensuche auch das Verfahren der so genannten Breitensuche. Hier werden ausgehend von einem Knoten K zuerst alle von K aus direkt erreichbaren Knoten N_1, ..., N_k besucht („in die Breite gehen“) und dann erst mit den von N_1 aus direkt erreichbaren Knoten fortgefahren, usw.

c) Außerdem können interessante Varianten des Graphendurchlaufs implementiert werden:

- Der durchlaufende Pfad wird gespeichert und kann angezeigt werden.
- Es kann ein Zielknoten gesucht werden. Sobald er gefunden ist, bricht der Graphendurchlauf ab.
- Es sollen alle Wege von einem Startknoten zu einem Zielknoten gefunden werden.
- Jede Kante soll mindestens einmal durchlaufen werden.

d) Ein besonders bekannter Algorithmus ist das **Verfahren von Dijkstra**: Es dient der Berechnung eines kürzesten Pfades zwischen einem Startknoten und einem beliebigen Knoten in einem gewichteten Graphen.

2 Softwaretechnik (ca. 26 Std.)

Lp: Das Arbeiten in Projekten ist die typische Vorgehensweise bei der Entwicklung großer Systeme. Mit den bisher erworbenen Kenntnissen und Fertigkeiten sind die Schüler nun in der Lage, größere Softwaresysteme (z. B. Geschäftsabläufe einer Bank, Autovermietung) eigenständig zu gestalten. Hierbei bauen sie auch ihre Fähigkeit zur Planung und Durchführung von Projekten aus. Die Jugendlichen übernehmen verstärkt persönliche Verantwortung und erfahren die Notwendigkeit, eigene Ansichten und Ideen vor anderen darstellen und vertreten zu können. Sie setzen alle bisher erlernten Beschreibungstechniken der Informatik ein und machen sich damit deren Zusammenwirken in einem größeren Kontext bewusst. Die Schüler erkennen, dass sie bei einigen Teilproblemen auf bereits vorgefertigte Standardlösungen in Form von Softwaremustern zurückgreifen können.

Ein Schwerpunkt des Lehrplanpunkts 11.2 liegt eindeutig auf der praktischen Projektarbeit. Das einleitende Kapitel 2.1 (Planung und Durchführung kooperativer Arbeitsabläufe) ist daher nicht als eine ausführliche theoretische Behandlung des Themenkreises Softwaretechnik gedacht. Hier sollen vor allem die bisherigen Erfahrungen der Schülerinnen und Schüler aus den Projekten der vorangegangen Jahrgangsstufen systematisiert und strukturiert werden; einige Ergänzungen (z. B. Wasserfallmodell) vervollständigen die Kenntnisse, um so ein vollständiges und tragfähiges Fundament für die Planung und Durchführung größerer Projekte zu schaffen.

Das Kapitel 2.2 (Praktische Softwareentwicklung) spricht die Verwendung geeigneter Softwaremuster an. Ausdrücklich eingefordert wird MVC (model-view-controler). Es empfiehlt sich, eine kurze Einheit mit wenigen, geeigneten Softwaremustern (z. B. Beobachtermuster als Bestandteil von MVC) in die Behandlung von 11.2.1 zu integrieren.

Um ausreichend Zeit für die Projektarbeit zu haben, sollte der Vorlaufteil nicht zu lang sein. Hier könnten z. B. 6 Stunden angesetzt werden. Damit verbleiben ca. 18 Stunden (6 Wochen) für das Projekt sowie ca. 2 Stunden für die Vorstellung des Ergebnisses und die Reflexion.

2.1 Planung und Durchführung kooperativer Arbeitsabläufe

Lp: Die Schüler systematisieren und vertiefen am konkreten Beispiel ihre Kenntnisse über die verschiedenen Schritte bei der Planung und Durchführung eines Softwareprojekts. Zur Koordinierung paralleler Arbeitsgruppen nutzen sie das Semaphorprinzip.

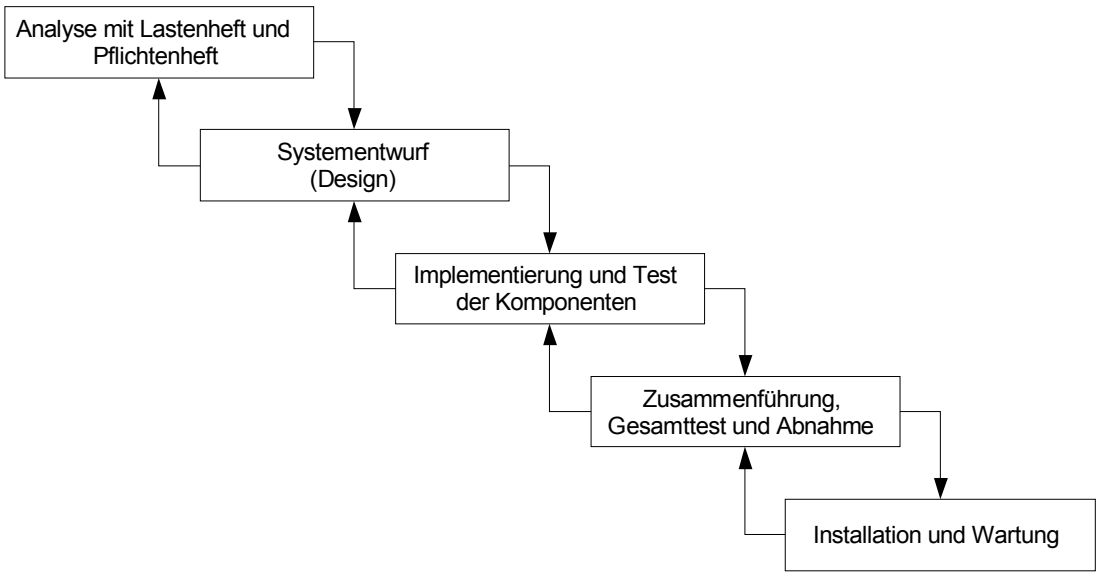
2.1.1 Planungsgrundlagen

In einer kurzen Reflexion der bisher durchgeführten Projekte machen sich die Jugendlichen bewusst, welche Mechanismen zur Planung und Durchführung von Projekten sie schon kennengelernt haben. So konnte schon in Jahrgangsstufe 7 über Schnittstellen gesprochen werden (Präsentation mit Webseiten: „Wo und unter welchem Namen findet sich eine verwandte Datei?“, Karol: „Wo steht Karol nach Beendigung einer Teilaufgabe?“). In Jahrgangsstufe 9 mussten Meilensteine gesetzt werden (z. B. Klassendiagramm erstellt, Tabellenschemata festgelegt, Datenbank angelegt, Beispieldaten eingefügt, Abfragen programmiert), um am Ende rechtzeitig fertig zu werden. In Jahrgangsstufe 10 stand die Aufteilung in parallel arbeitende Gruppen und damit die Einhaltung von Schnittstellen im Vordergrund. Dabei geht es nicht nur um die Kenntnis dieser Mechanismen, sondern auch um die Frage: „Welchen Beitrag leisten diese Mechanismen für eine erfolgreiche Durchführung des Projekts?“

Auf dieser Basis kann extrapoliert werden, wie diese Mechanismen bei einer weiteren Steigerung des Projektumfangs verflochten und eventuell ergänzt werden müssen. Eine wichtige Ergänzung der bekannten Maßnahmen ist das (erweiterte) Wasserfallmodell. Es

beschreibt die einzelnen Phasen des Projektablaufs und legt als wichtige Erkenntnis fest, dass jede Phase abgeschlossen sein muss, ehe die nächste begonnen wird. Der Grund für diese Festlegung kommt aus der Erfahrung, dass die Korrektur eines Fehlers oder einer Auslassung um so mehr Zeit und damit Geld kostet, je später das Problem erkannt wird. Allerdings hat es sich in der Praxis gezeigt, dass es trotz sorgfältiger Analyse und detailliertem Entwurf immer noch zu Nachbesserungen kommen kann. Zum einen liegt es daran, dass der (die) Auftraggeber(in) die Aufgaben nur informell beschreiben kann und daher spezielle Randbedingungen für Detailanforderungen erst während der Implementierungsphase auffallen. Zum anderen liegt es auch daran, dass der Systementwurf nicht immer vollständig formal erstellt wird („der Rest ist schon klar“), was zu zusätzlichem Bedarf an die Schnittstellen führen kann. Diese Aspekte sollten mit den Schülerinnen und Schülern auch im Hinblick auf die Sorgfalt ihrer Planungen deutlich diskutiert werden. Als Mechanismus zum Umgang mit diesen Fehlerquellen wird als Erweiterung des Wasserfallmodells zugelassen, dass bei auftretenden Problemen in den Phasen so weit zurückgegangen wird, bis der Fehler behoben werden kann; anschließend werden die betroffenen Phasen wieder vervollständigt.

In der Literatur existieren für das Wasserfallmodell vielfältige, mehr oder weniger detaillierte Varianten mit unterschiedlicher Schwerpunktsetzung. Für die Schule sollte ein nicht zu filigranes Modell gewählt werden, das die Bereiche Installation und Wartung nicht zu sehr betont. Eine mögliche Strukturierung ist:



Einfaches Wasserfallmodell

Der erste Teil umfasst die Analyse der Situation, die informelle Fixierung der Auftraggeberwünsche (Lastenheft) und die geplante Umsetzung dieser Wünsche (Pflichtenheft). In der zweiten Phase wird das zu erstellende System entworfen. Dafür werden in der Regel formale Diagramme eingesetzt (siehe Kapitel 2.1.2). Wichtig ist, dass in dieser Phase bereits die zu leistende Arbeit in Teilprojekte aufgeteilt wird und die Schnittstellen zwischen diesen Teilen exakt festgelegt werden. Die zentrale Rolle dieser Phase kann im Projekt betont werden (siehe 2.2.1).

2.1.2 Diagramme und ihre Verwendung

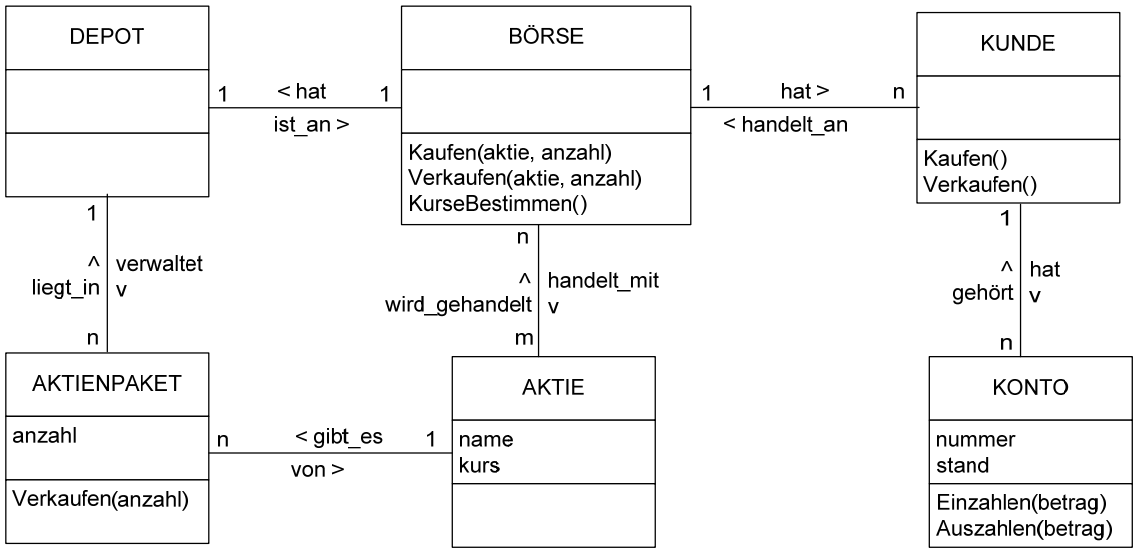
Die Schülerinnen und Schüler kennen aus der Unter- und Mittelstufe bereits alle notwendigen Diagrammarten für die Erstellung des Systementwurfs. Im Folgenden werden diese im Überblick dargestellt und deren Einsatzmöglichkeiten reflektiert. Gegebenenfalls können

nützliche Spielarten der bereits bekannten Diagrammtypen wie ein erweitertes Klassendiagramm oder eine Klassenkarte ergänzt werden.

Die gezeigten Beispieldiagramme beziehen sich auf die nachfolgend unter 2.2.2 vorgestellten Projektthemen Aktienhandel und Fahrradroutenplaner.

Klassendiagramm

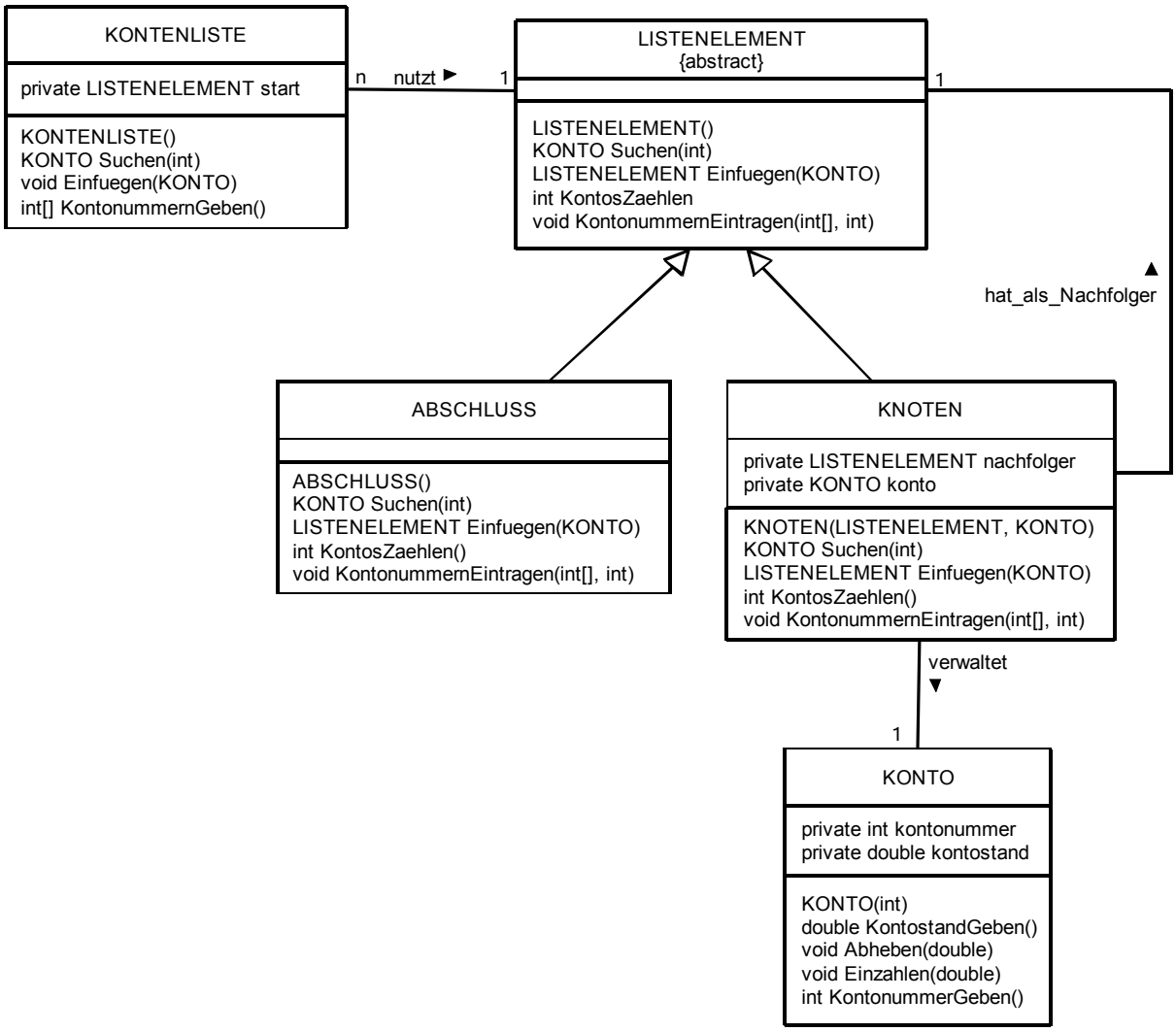
Ein Klassendiagramm beschreibt die vorhandenen Klassen mit ihren Attributen und Methoden sowie die Beziehungen zwischen den Klassen. Es gibt damit ein umfassendes Bild des Gesamtsystems wieder.



Übersicht über die Klassen eines (vereinfachten) Aktienhandels nach der Situationsanalyse

Erweitertes Klassendiagramm

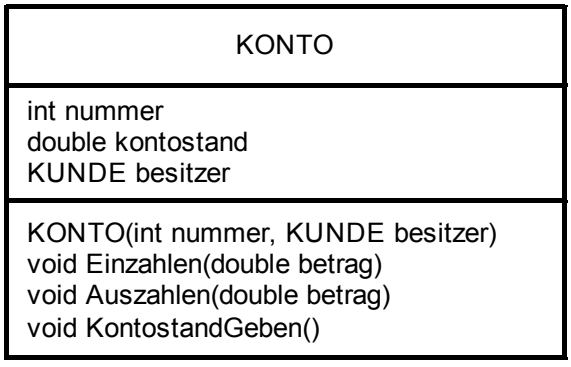
Es ergänzt das Klassendiagramm um implementationsspezifische Elemente wie Datentypen oder Referenzattribute zur Umsetzung der Beziehungen. Methoden zum Lesen und Setzen der Attribute (Geber- und Setzermethoden) werden nicht angegeben, es sei denn, dass ihnen im Modell eine besondere Rolle zukommt (z. B. Überprüfen umfangreicherer Bedingungen beim Setzen, Überprüfen von Zugriffsberechtigungen beim Lesen). Sie können mit geeigneten Namen (z. B. in der Form AttributnameGeben(), AttributnameSetzen(wert)) ad hoc verwendet werden. Die Konstruktoren können zur Dokumentation der Parameter angegeben werden, falls die verwendete Sprache Konstruktoren zur Verfügung stellt.



Kontenverwaltung des Aktienhandels

Klassenkarte

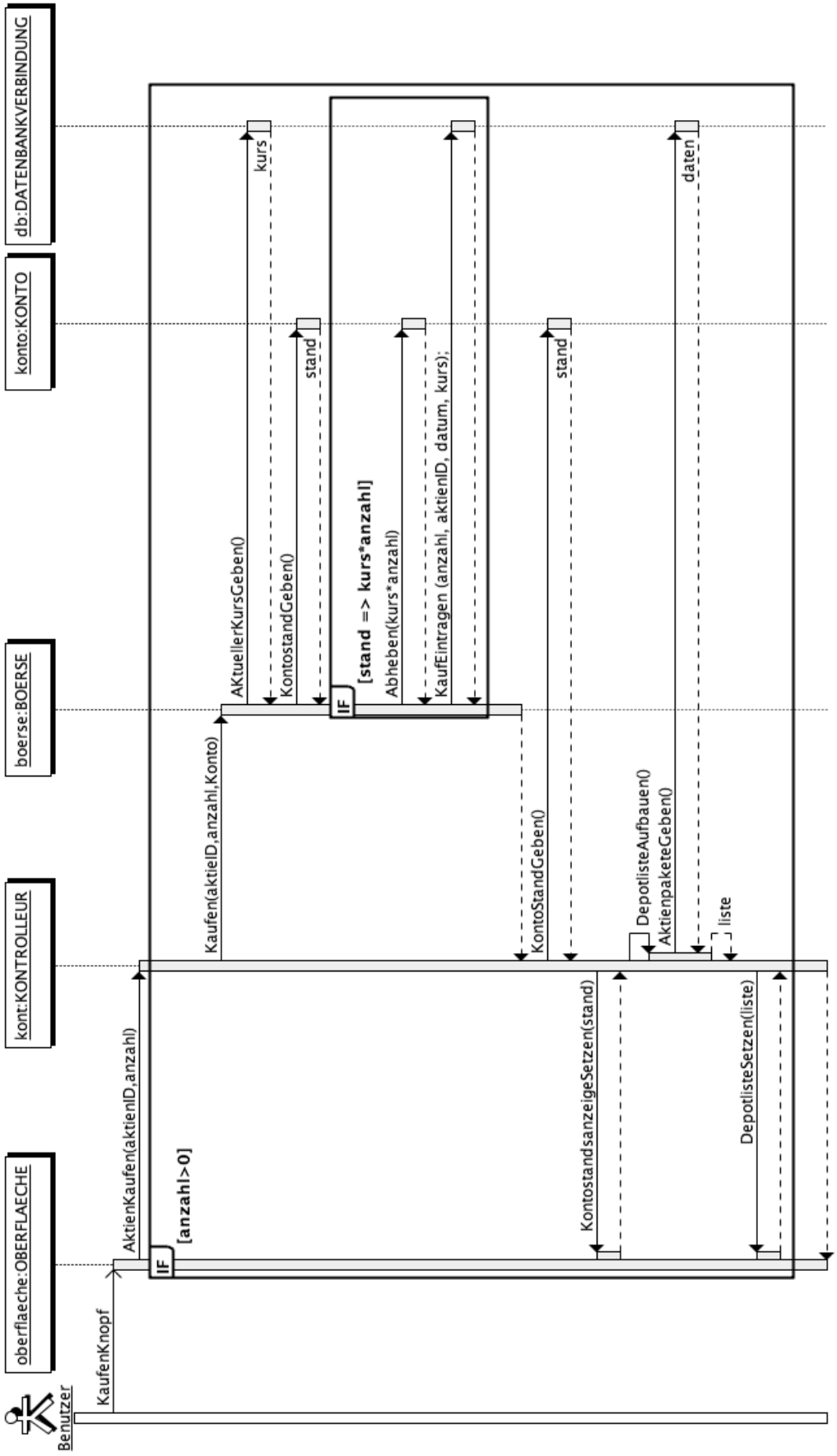
Ein nur eine Klasse beschreibendes Klassendiagramm wird aus didaktischen Gründen oft auch als Klassenkarte bezeichnet (UML kennt diesen Begriff nicht). Meistens wird in Klassenkarten die erweiterte Form des Klassendiagramms verwendet.



Klassenkarte für die Klasse KONTO in erweiterter Form

Sequenzdiagramm

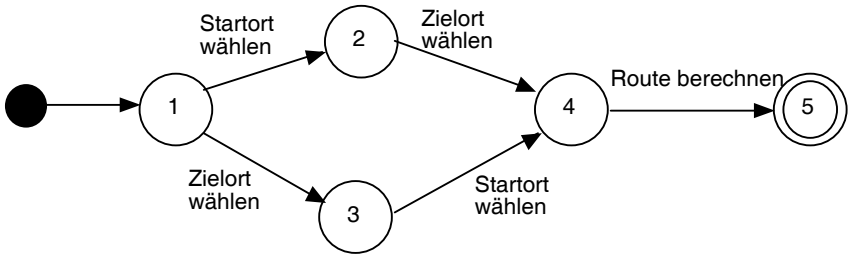
Ein Sequenzdiagramm beschreibt die Abfolge der Botschaften und Methodenaufrufe, die zur Erfüllung einer bestimmten Aufgabe benötigt werden. Auslöser ist oft ein durch einen Benutzer ausgelöstes Ereignis. Sequenzdiagramme verdeutlichen damit auch die notwendigen Leistungen, die einzelne Objekte zu erbringen haben, und geben Informationen darüber, wo Schnittstellen zwischen den Klassen benötigt werden. Auch die fließende Information (Parameter, Rückgabewerte) kann analysiert werden. Dazu wird beim Aufrufpfeil der Methodenname mit der Liste der Parameter angegeben. Beim Rückkehrpfeil kann der Rückgabewert als Kommentar angegeben werden; damit ist aber keine Wertzuweisung im Sinn eines Funktionsergebnisses verbunden.



Ablauf beim Aktienkauf

Zustandsdiagramm

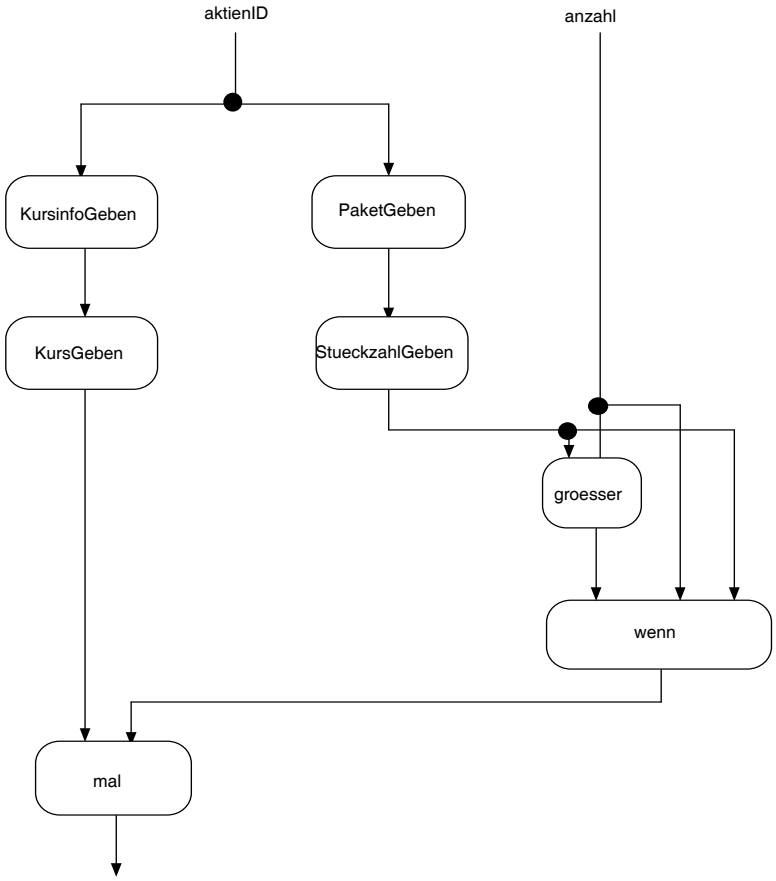
Ein Zustandsdiagramm beschreibt Abläufe aus „statischer“ Sicht, identifiziert also die Zustände, in denen sich das System oder ein Teil davon befindet und die Ereignisse, die die Übergänge zwischen diesen Zuständen auslösen. Eine typische Anwendung ist die Beschreibung der Benutzerführung. Aber auch die Arbeitsweise größerer Teilsysteme kann oft durch ein Zustandsdiagramm gut visualisiert werden.



Benutzerführung beim Fahrradroutenplaner

Datenflussdiagramm

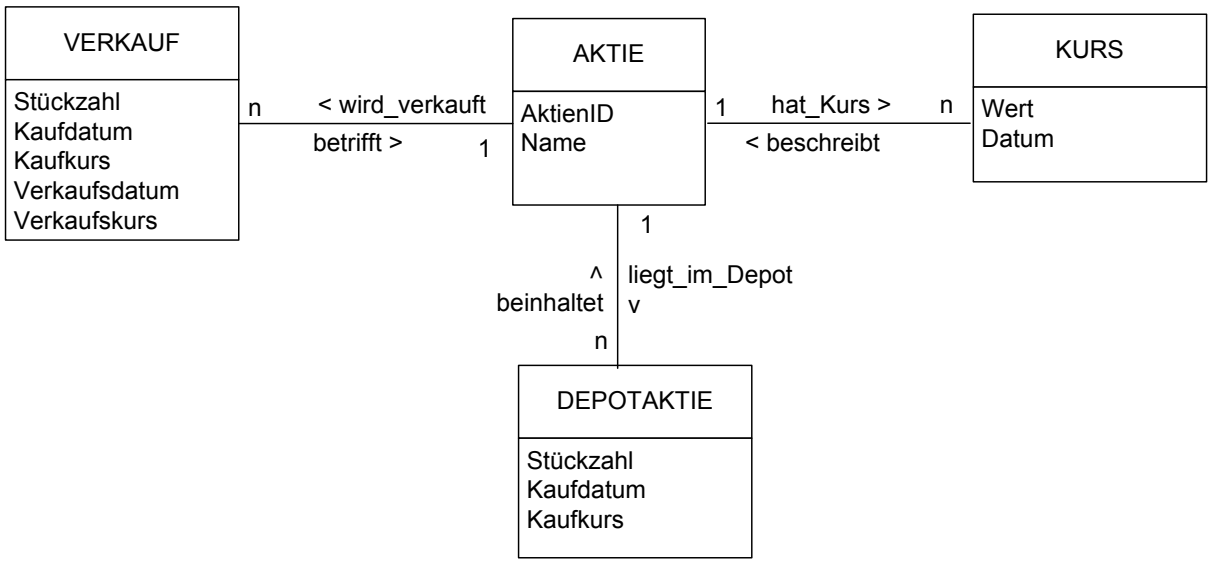
Ein Datenflussdiagramm beschreibt den Datenfluss aus funktionaler „black-box-Sicht“. Dabei wird festgehalten, welche Daten zu verarbeiten sind und was mit ihnen geschehen soll, nicht aber, wie die Leistung erbracht wird. Datenflussdiagramme können damit zur globalen Beschreibung von Teilleistungen des Gesamtsystems verwendet werden. Sie sind aber auch zur Visualisierung und Entzerrung komplexer logischer oder arithmetischer Ausdrücke geeignet.



Berechnung des Werts eines Aktienverkaufs

Datendiagramm

Ein Datendiagramm beschreibt die vorhandenen Klassen mit ihren Attributen sowie die Beziehungen zwischen den Klassen. Es wird verwendet, um die Struktur der persistenten Daten eines Informatiksystems wiederzugeben.



Datendiagramm des Aktienhandels

Zur Verfeinerung wird bei der Abbildung auf eine relationale Datenbank das Tabellenschema verwendet; es ergänzt Datentypen, Schlüssel und die zur Umsetzung der Beziehungen notwendigen Fremdschlüssel und Hilfstabellen. Das Tabellenschema spielt also hier die gleiche Rolle wie das erweiterte Klassendiagramm bei der objektorientierten Programmierung.

aktie(AktienID, name)
kurse(LfdNr, AktienID, Wert, Datum)
depotaktie(PaketNummer, AktienID, Stueckzahl, Kaufdatum, Kaufkurs)
verkaeufe(VerkaufNr, AktienID, Stueckzahl, Kaufdatum, Kaufkurs, Verkaufsdatum, Verkaufskurs)

Tabellenschema des Aktienhandels

2.1.3 Entwurfsmuster

Entwurfsmuster helfen bei der Strukturierung der Klassen in der Entwurfsphase, weil sie gut durchdachte Lösungsschemata für immer wieder vorkommende Standardsituationen anbieten. Eines der wichtigsten Grundprinzipien guten Programmentwurfs ist die strikte Trennung zwischen der eigentlichen Programmlogik (dem Modell, englisch: model) und der Benutzerführung (view) durch das Verbindungs- und Koordinationsglied (controller). Das Entwurfsmuster für diese Trennung wird entsprechend MVC (model-view-controller) genannt. Um Entwurfsfehler zu vermeiden, die gerade in diesem Bereich von unerfahrenen Entwicklern immer wieder gemacht werden und den Schülerinnen und Schülern grundlegende Strategien an die Hand zu geben, ist das Entwurfsmuster MVC ausdrücklich im Lehrplan verankert.

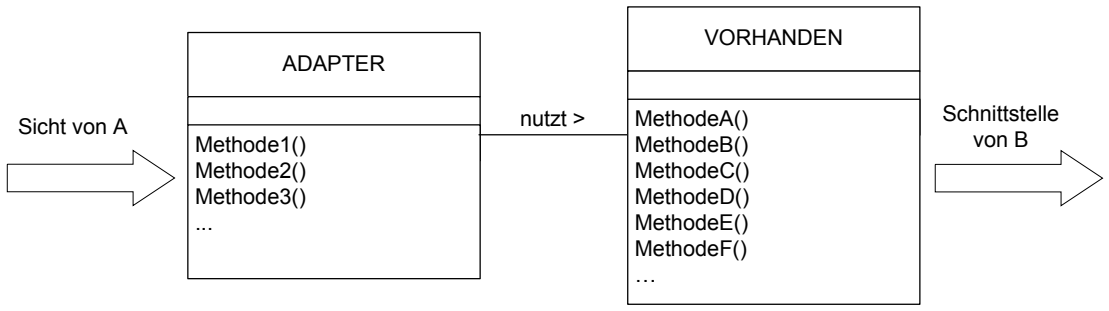
MVC ist ein so genanntes zusammengesetztes Muster (compound pattern). Es vereinigt zwei einfache Muster, das Strategiemuster (strategy pattern) mit der vereinfachten Form Adaptermuster (adapter pattern) sowie das Beobachtermuster (observer pattern). Es empfiehlt sich, hier schrittweise vorzugehen und die drei Teilmuster zuerst getrennt zu behandeln. Erst dann werden sie bei der Übertragung auf die Einbindung der Oberfläche zusammengesetzt. Zusätzlich ergibt sich für die Schülerinnen und Schüler der Vorteil, dass sie das sehr häufig anwendbare und in graphischen Oberflächen immer wieder angewandte Beobachtermuster als eigenständiges Muster kennen- und gebrauchen lernen. Beim Beobachtermuster kommt auch ein weiteres Prinzip guten Programmentwurfs sehr deutlich zur Geltung: Bei der Kommunikation zwischen Objekten verschiedener logischer Teile eines Programms sollten nicht die konkreten Klassen angesprochen werden, sondern Schnittstellenklassen, die die eigentliche Implementierung verbergen.

In diesem Zusammenhang kann auch thematisiert werden, dass das im Lehrplanpunkt 11.1 „Rekursive Datenstrukturen“ verwendete Muster Kompositum (composite) in der Literatur für zwei verschiedene Situationen verwendet wird. Diese zweite Verwendung (vom Gebrauch her die häufigere) ist als letztes Beispiel der Entwurfsmuster mit aufgeführt.

Adaptermuster

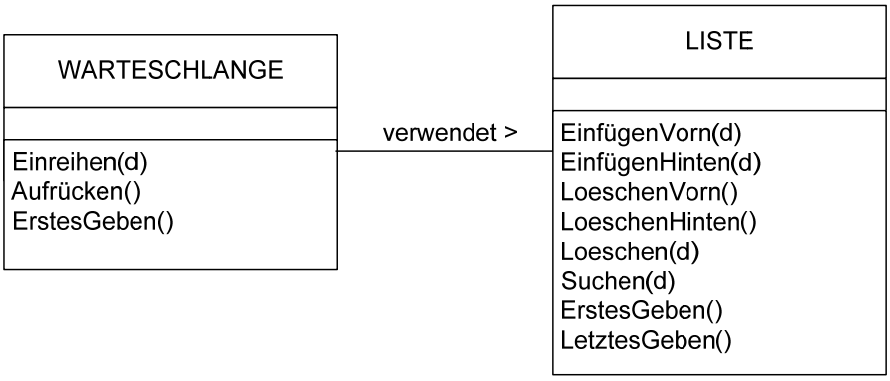
Im Urlaub kann man öfter die Erfahrung machen, dass die Steckersysteme verschiedener Länder nicht immer kompatibel sind. Um ein elektrisches Gerät an der Steckdose anschließen zu können, sind daher manchmal Zwischenstecker (Adapter) notwendig.

Auch beim Zusammenfügen einzelner Programmteile sind solche Adapter in Form eigenständiger Zwischenklassen (Adapterklassen) sinnvoll. Sie bieten dem Programmteil A die Schnittstelle, die er erwartet, und passen sie an die – eventuell auf mehrere Klassen verteilte – Schnittstelle von Programmteil B an. Das „Einziehen dieser Zwischenebene“ empfiehlt sich auch dann, wenn im ursprünglichen Entwurf die von A erwartete Schnittstelle mit der von B angebotenen Schnittstelle identisch ist, da so flexibler auf spätere Erweiterungen bzw. Korrekturen reagiert werden kann.



Adaptermuster allgemein

Ein sehr einfaches Beispiel für das Adaptermuster ist die Implementierung einer Warteschlange mithilfe einer vorhanden allgemeinen Liste. Die Klasse WARTESCHLANGE ist ein einfacher Adapter, der von der Vielzahl der Methoden der Klasse LISTE nur die für die Warteschlange benötigten unter geeignetem Namen weiterreicht.



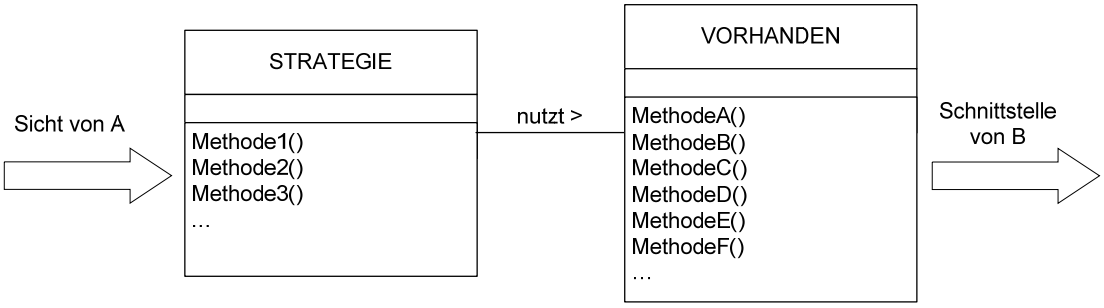
Adaptermuster bei der Warteschlange

Die Methoden der Warteschlange verwenden einfach die korrespondierenden Methoden der LISTE: Einreihen nutzt EinfuegenHinten, Aufruecken nutzt LoeschenVorn und ErstesGeben nutzt ErstesGeben. Alle für eine Warteschlange nicht anwendbaren Methoden sind nun nicht mehr sichtbar.

```
class WARTESCHLANGE
{
    private LISTE liste;

    WARTESCHLANGE ()
    {
        liste = new LISTE();
    }

    void Einreihen(DATENELEMENT d)
    {
        liste.EinfuegenHinten(d);
    }
        :
```

Strategiemuster

Dieses Muster kann man sich als Erweiterung des Adaptermusters vorstellen. Botschaften des Programmteils A werden aber nicht wie beim Adapter einfach nach B weitergereicht. Die Strategiemusterklasse filtert die Botschaften mit eigener Logik, die z. B. durch einen Automaten beschrieben werden kann. Jede ankommende Botschaft (also jeder Aufruf einer Methode der Strategieklasse) löst dann einen (bedingten) Zustandsübergang aus. Die ausgelösten Aktionen werden als Botschaften (Methodenaufrufe) an die vorhandene(n) Klasse(n) (Programmteil B) weitergegeben. Das Klassendiagramm ist damit gleich dem des Adaptermusters, die Aufgaben der Methoden der Strategieklasse sind aber umfangreicher.



In einfacheren Fällen besteht die Strategie darin, dass eine Methode des Adapters mehrere Methoden der Klasse(n), auf die sich der Adapter abstützt, verwenden muss. Im obigen

Beispiel wäre das der Fall, wenn die Liste die Methoden LoeschenVorn und LoeschenHinten nicht zur Verfügung stellen würde. Die Methode Aufruecken müsste dann ErstesGeben und Loeschen verwenden:

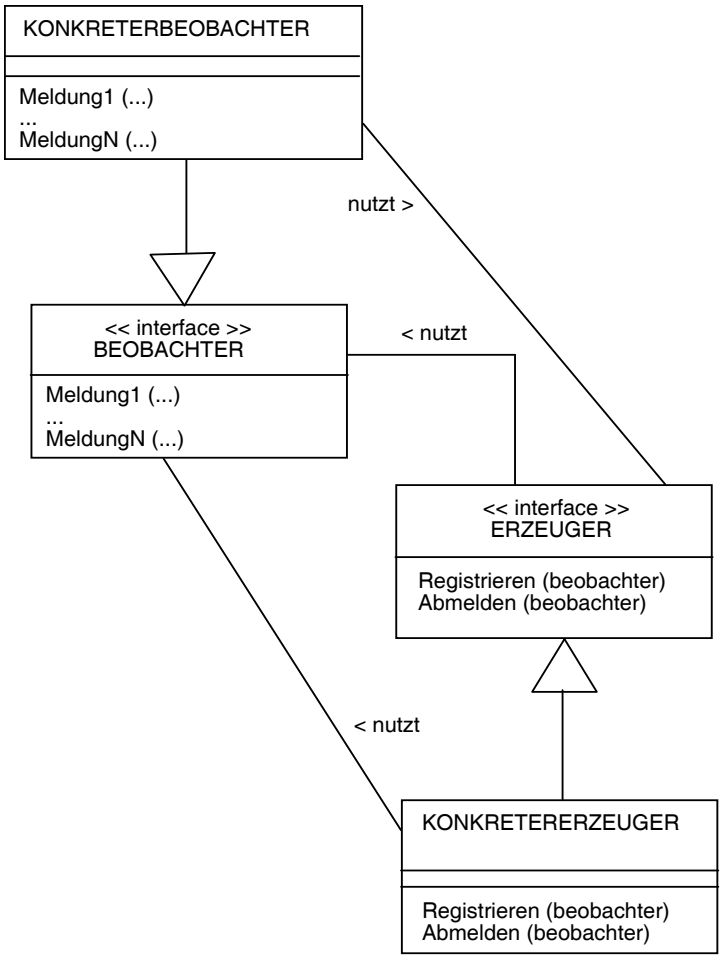
```
void Aufruecken ()
{
    Loeschen(ErstesGeben());
}
```

Beobachtermuster

Dieses Muster dient dazu, die (einseitige) Kommunikation zwischen einem Erzeuger von Information und den Interessenten an dieser Information (Beobachter) offen und flexibel zu strukturieren. Dieses Muster ist auch ein Paradebeispiel für die Trennung verschiedener Programmteile durch Schnittstellen.

Der Erzeuger der Meldungen bietet ein Interface an, das potenzielle Interessenten an den Meldungen nutzen können, um ihren Beobachtungswunsch anzumelden (Methode Registrieren). Bei Bedarf können sie sich auch wieder vom Empfang der Meldungen abmelden.

Die Interessenten müssen ihrerseits garantieren, dass sie alle Botschaften des Erzeugers verstehen, d. h., sie implementieren die geforderte Schnittstelle. Das Klassendiagramm macht deutlich, dass beide beteiligten Seiten nur Kenntnis der Schnittstellen haben; welche Klassen diese Schnittstellen implementieren, bleibt ihnen verborgen.



Beobachtermuster allgemein

Die Schnittstellenklassen beschreiben die notwendigen Methoden.

```
interface BEOBACHTER
{
    void Meldung1(...);
        :
    void MeldungN(...);
}
interface ERZEUGER
{
    void Registrieren(BEOBACHTER b);
    void Abmelden(BEOBACHTER b);
}
```

Jeder konkrete Beobachter muss die Meldungen empfangen. Die Registrierung beim Erzeuger kann im Konstruktor erfolgen (wenn von Beginn an beobachtet werden soll); falls nur zeitweise beobachtet wird, werden An- und Abmeldung an den entsprechenden Stellen vorgenommen.

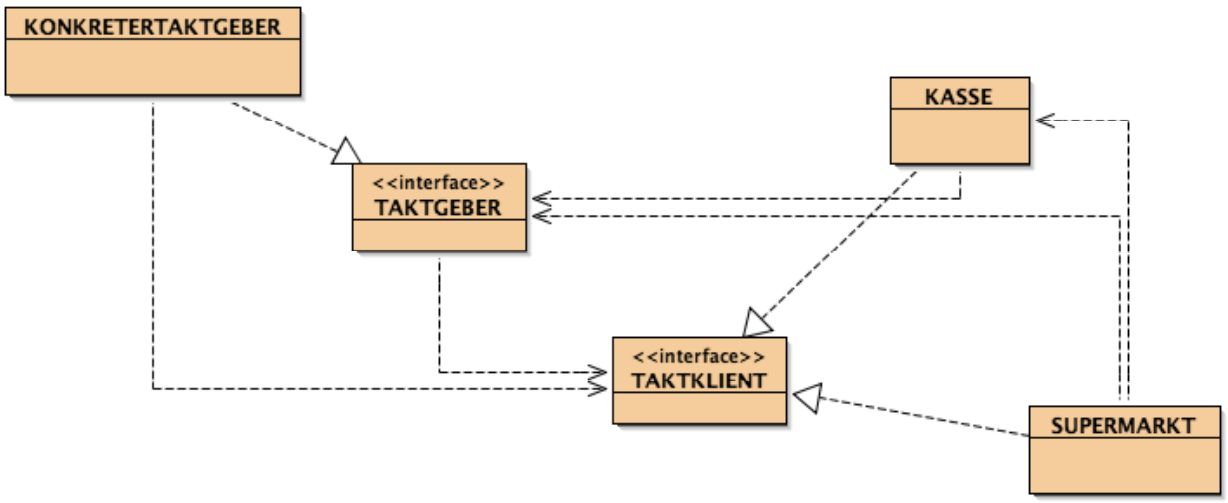
```
class KONKRETERBEOBACHTER implements BEOBACHTER
{
      :
    KONKRETERBEOBACHTER(ERZEUGER e)
    {
          :
        e.REGISTRIEREN(this);
    }

      :

    public void Meldung1(...)
    {
        <was immer auch zu tun ist>
    }

      :
```

Der konkrete Erzeuger von Meldungen muss die registrierten Beobachter verwalten. Dafür sind in allen Sprachen verfügbare Containerklassen bestens geeignet.

```
import java.util.ArrayList;
class KONKRETERERZEUGER
{
    private ArrayList<BEOBACHTER> beobachter;

    KONKRETERERZEUGER()
    {
          :
        beobachter = new ArrayList<BEOBACHTER>;
    }

    public void Registrieren(BEOBACHTER b)
    {
        beobachter.add(b);
    }

    public void Abmelden(BEOBACHTER b)
    {
        beobachter.remove(b);
    }
```

```
void Meldung1(...)
{
    for (BEOBACHTER b: beobachter)
    {
        b.Meldung1(...);
    }
}

  :
```

In einer Simulation (z. B. Supermarkt, für Details vgl. Handreichung für Jahrgangsstufe 10) erzeugt ein Taktgeber Taktimpulse für eine gemeinsame Zeitbasis der Simulation. Die Klassen der Simulation, die den Takt benötigen, um zeitabhängige Aktionen auszuführen (im Beispiel die Kassen und die Kundenerzeugung), registrieren sich beim Taktgeber, der hier die Rolle des Erzeugers hat. Sie müssen über eine Schnittstelle (im Bild TAKTKLIENT, entspricht der Schnittstelle BEOBACHTER des Musters) bekanntgeben, dass sie über eine Methode TaktImpulsEmpfangen (entspricht Meldung) verfügen, die der Taktgeber zur rechten Zeit aufrufen kann. Auch der Taktgeber (im Bild die Klasse KONKRETERTAKTGEBER, entsprechend KONKRETERERZEUGER) gibt über eine Schnittstelle (TAKTGEBER, entsprechend ERZEUGER) preis, dass er über die Methoden Registrieren und Abmelden verfügt, die alle Klassen nutzen können, die die Taktimpulse empfangen wollen und dafür die Schnittstelle TAKTKLIENT implementieren.



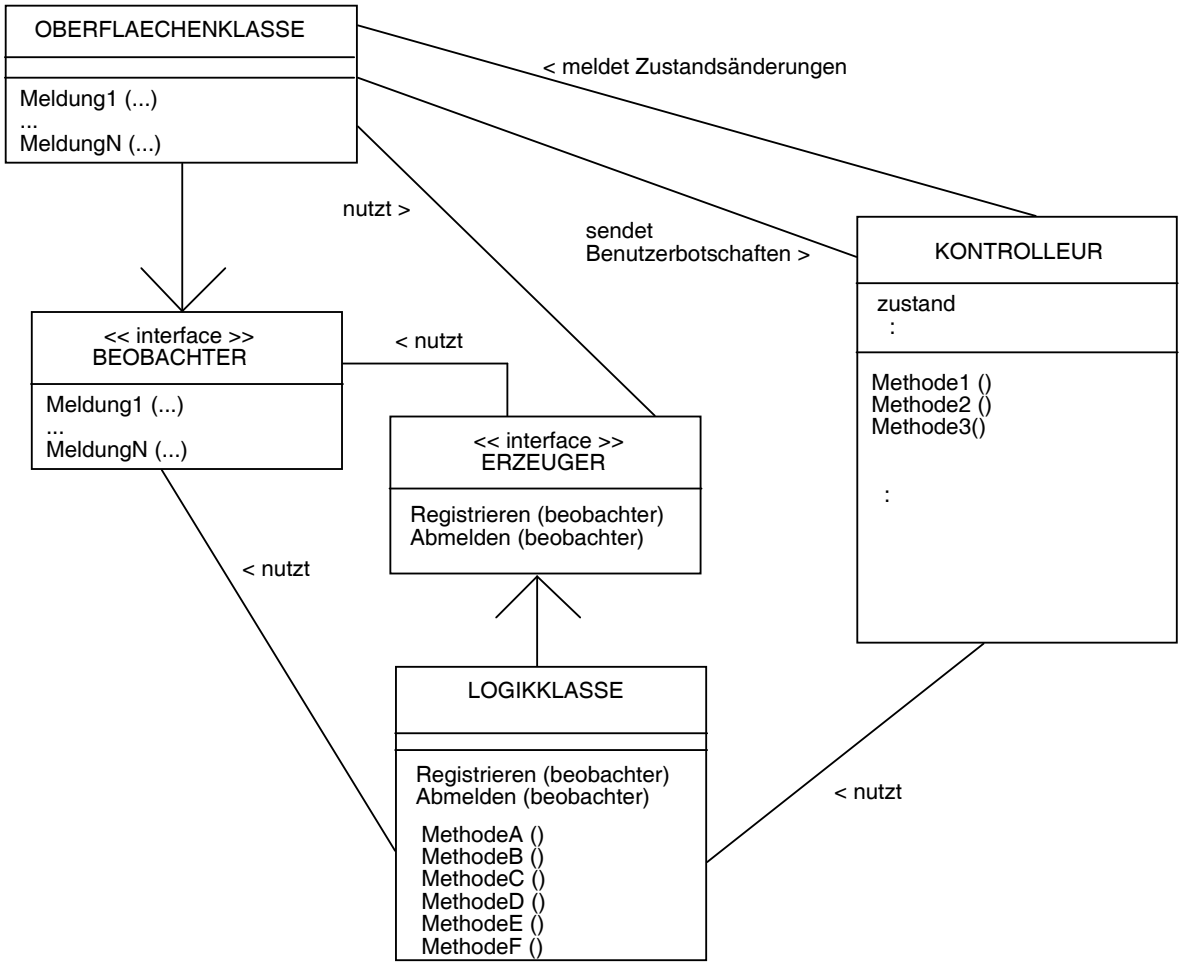
Beobachtermuster beim Supermarkt

Falls es die verwendete Sprache erlaubt (wie z. B. Java), können die Schnittstellen statt in Form abstrakter Klassen durch Interfaceklassen beschrieben werden.

MVC

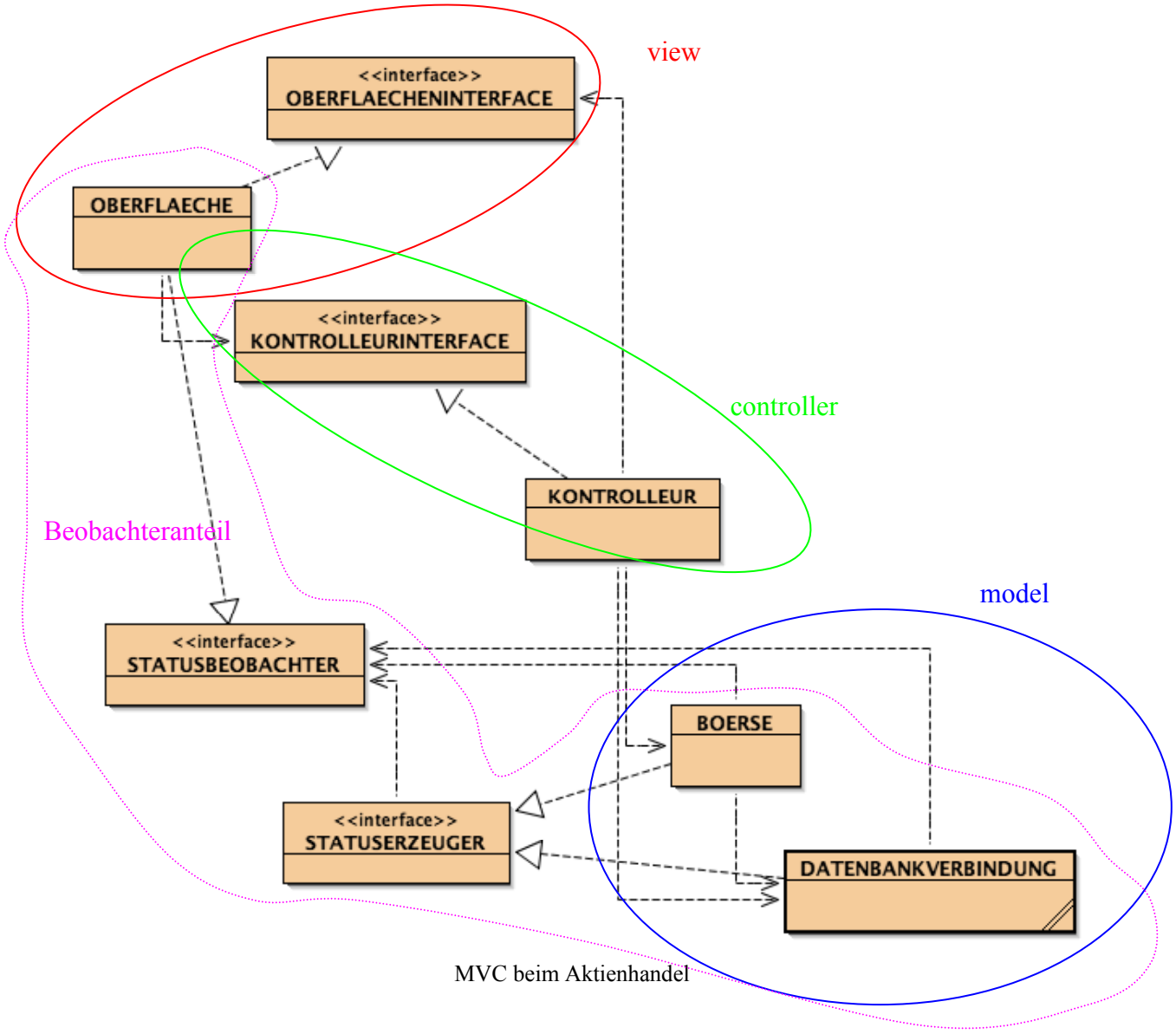
Das Muster MVC vereinigt die beiden oben genannten Entwurfsmuster. Programmteil A im Sinn des Strategiemusters ist hier die Benutzeroberfläche (view), Programmteil B ist die Programmlogik (model). Der Kontrolleur (controller) empfängt Botschaften der Oberfläche und gibt sie an geeignete Methoden der Programmlogik weiter. Insoweit spielt er die Rolle des Adapters. Normalerweise wird im Kontrolleur aber auch die Benutzerführung abgebildet, d. h., der Kontrolleur bildet den Bedienzustand des Benutzers ab. Daher implementiert der Kontrolleur das Strategiemuster. Zusätzlich meldet er den aktuellen Bedienzustand an die Oberfläche zurück (dadurch werden Knöpfe verfügbar oder nicht verfügbar etc.).

Um direkt Information aus der Programmlogik anzeigen zu können (nicht alle Information kann über Rückgabewerte von Methoden des Kontrolleurs zurückgemeldet werden), registriert sich die Oberfläche als Beobachter bei den entsprechenden, Information produzierenden Teilen der Programmlogik.



MVC allgemein

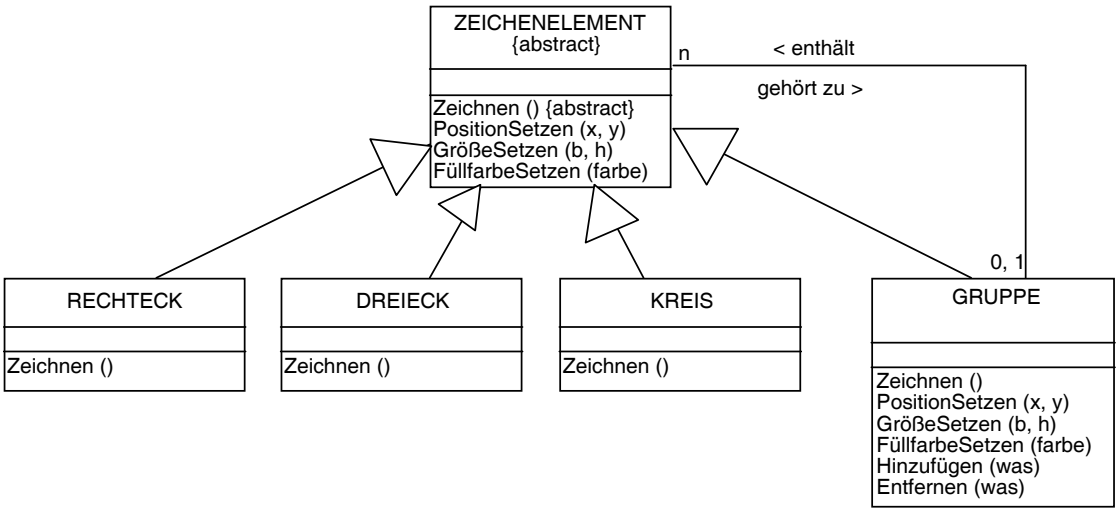
Im Ausschnitt aus dem Klassendiagramm des BlueJ-Projekts sind für den Aktienhandel die Gliederungen klar erkennbar: Im unteren Teil die Programmlogik (model), repräsentiert durch die Klassen BOERSE und DATENBANKVERBINDUNG, im oberen Teil die Oberfläche (view), in der Mitte die Klasse KONTROLLEUR (controller). Botschaften gehen von der Oberfläche zum Kontrolleur und von dort aus weiter an die Programmlogik. Der Kontrolleur gibt Statusmeldungen an die Oberfläche zurück. Dabei sind Oberfläche und Kontrolleur durch Interfaceklassen voneinander getrennt (lose Kopplung); die konkreten Implementierungen „kennen sich nicht“. Das Beobachtermuster, über das die Oberfläche Statusänderungen aus der Programmlogik erfährt, ist auf der linken Seite erkennbar, realisiert in den Interfaceklassen STATUSERZEUGER und STATUSBEOBACHTER.



MVC beim Aktienhandel

Kompositum

Häufiger als bei den rekursiven Datenstrukturen wird das Entwurfsmuster Kompositum dafür verwendet, dass Gruppen von Objekten das gleiche Verhalten zeigen wie einzelne Objekte. Einfachstes Beispiel sind Graphikdokumente. Die einzelnen Klassen wie LINIE, RECHTECK oder KREIS haben unter anderem die Methoden Zeichnen() oder Verschieben(). In Graphikdokumenten gibt es auch die Klasse GRUPPE, mithilfe derer mehrere Einzelobjekte zu einer (temporären) Einheit verbunden werden können. Die Klasse GRUPPE muss die Methoden Zeichnen() und Verschieben() ebenfalls zur Verfügung stellen. Beim Aufruf dieser Methoden muss sie die Botschaft an alle Objekte der Gruppe weiterleiten. Um Objekte der Klasse GRUPPE wie Objekte aller anderen Graphikklassen verwenden zu können, verwendet man folgendes Muster:



Kompositum bei Graphikobjekten

Statt der Klasse ABSCHLUSS (vgl. Kompositum bei Liste/Baum), die „nur“ die Abbruchbedingung für die rekursiven Methoden implementiert sind hier mehrere Klassen (RECHTECK, DREIECK, ...) vorhanden, die neben dem Umsetzen der Abbruchbedingung auch noch die eigentliche Arbeit leisten, nämlich das Darstellen der Zeichenobjekte. Dafür verwaltet die Klasse KNOTEN (vgl. Kompositum bei Liste/Baum) die eigentlichen Datenobjekte, während die Klasse GRUPPE nur Informationen weitergibt. Entscheidend für das Muster Kompositum ist die (rekursive) Beziehung *enthält* zwischen GRUPPE und GRAPHIKOBJEKT. Sie identifiziert dieses Muster als Kompositum, auch wenn die Eigenschaften der Klasse GRUPPE sich von denen der Klasse KNOTEN deutlich unterscheiden.

2.2 Praktische Softwareentwicklung

Lp: Bisher haben die Schüler verschiedene Modellierungstechniken der Informatik einzeln angewandt. Nun erkennen sie, dass eine angemessene Beschreibung größerer Systeme nur durch die kombinierte Verwendung aller bisher erlernten Modellierungstechniken möglich ist. Bei der Implementierung ihrer Modelle setzen sie bekannte Datenstrukturen situationsgerecht ein und achten bei der Gestaltung der Bedieneroberfläche insbesondere auf Benutzungsfreundlichkeit.

2.2.1 Durchführung des Projekts

Das Wasserfallmodell legt die grundlegenden Schritte der Durchführung fest. Damit definiert es auch die Marken für Meilensteine. Hier kann den Schülerinnen und Schülern durch entsprechende Festlegung der Zeiten nochmals klar werden, wie wesentlich die Entwurfsphase ist. Es empfiehlt sich, das Pflichtenheft zu Beginn des Systementwurfs und die beim Systementwurf erstellten Unterlagen zum Meilenstein „Beginn der Implementierung“ einzufordern, zusammen mit dem Hinweis, dass diese Unterlagen in die Beurteilung des Projekts mit einfließen.

Dabei sollte auch angesprochen werden, dass bereits während des Systementwurfs die Untergruppen gebildet werden und ihre parallele Arbeit aufnehmen. Die Rolle des Projektleiters muss bereits zu Beginn der Arbeit besetzt werden, die Untergruppenleiter werden in dieser Phase bestimmt.

Stärker als in bisherigen Projekten müssen die Teilgruppen den Zugriff auf gemeinsame Ressourcen (Schnittstellenbeschreibungen, zentrale Klassen etc.) koordinieren. Hier kommt die im Lehrpan geforderte Idee der Semaphore zur Koordination des Zugriffs zum Tragen.

Für jede gemeinsam genutzte Ressource wird eine Markierung (Fähnchen oder Ähnliches) an einem zentralen Ort aufgestellt. Jeder, der die zugeordnete Ressource bearbeiten will, holt das Fähnchen, erledigt seine Arbeit und stellt anschließend das Fähnchen zurück. Ist das Fähnchen nicht da, wird die Ressource gerade von einer anderen Gruppe bearbeitet; man muss warten.

Theoretisch kann es dabei natürlich zu Verklemmungen (deadlocks) kommen. Gruppe 1 reserviert Ressource A und stellt im Lauf der Arbeit fest, dass sie auch Ressource B braucht. Gruppe 2 hat mit Ressource B begonnen und benötigt nun auch Ressource A. Sollte dieser Fall eintreten, müssen die Schülerinnen und Schüler Intelligenz zeigen. Das Problem kann dann bei der Projektreflexion analysiert werden. Ein allzu allgemeines (theoretisches) Eingehen auf diese Problematik ist vom Lehrplan an dieser Stelle nicht gefordert; Organisation paralleler Prozesse und Verklemmungen sind Thema in Jahrgangsstufe 12.

Unterrichtsmethodisch kann bezüglich der Projektarbeit wie folgt vorgegangen werden: Es kann ein kleineres, unterrichtsbegleitendes Projekt zum Erwerb der erforderlichen Kompetenzen durch die Schülerinnen und Schüler durchgeführt werden. Im Anschluss daran kann dann ein eigenständiges, arbeitsteiliges Softwareprojekt durch die Jugendlichen durchgeführt werden.

2.2.2 Anforderungen an Projektthemen

Für die Arbeit am größeren Projekt steht ein Zeitraum von ca. 6 Wochen zu je drei Wochenstunden zur Verfügung. Da es auch möglich ist, Planungsteile zu Hause erstellen zu lassen, können bereits relativ komplexe Aufgabenstellungen angegangen werden. Die Möglichkeiten sollten dazu ausgereizt werden, für die Schülerinnen und Schüler interessante Themen aufzugreifen.

Direkte Konsequenz aus dem Lehrplanauftrag, das Entwurfsmuster MVC zu verwenden, ist, dass das zu erstellende Informatiksystem eine graphische Bedienoberfläche besitzt. Das Erfüllen dieser Forderung hat mehrere positive Effekte. Für die Jugendlichen ist die Entwicklung eines solchen Systems wesentlich spannender als z. B. die Entwicklung eines Servertools, von dessen Arbeit nichts direkt sichtbar wird. Auch ergibt sich daraus bereits eine sehr eigenständige Untergruppe mit guten Möglichkeiten zur Binnendifferenzierung (die Grundkonzeption der Oberfläche benötigt Teammitglieder mit Überblick; die Detailausgestaltung enthält viel Routinearbeit). Im Unterricht können auch geeignete, von der Lehrkraft zur Verfügung gestellte, einfacher zu verwendende Frameworks für die Erstellung graphischer Oberflächen Einsatz finden. So könnte beispielsweise der Einsatz des Frameworks JGUIToolbox von Hans Witt – derzeit zu finden unter www.ovtg.de/3_arbeit/informatik/JGUIWeb/JGUIWebstart.html – hilfreich sein. Auch der Java-Editor (javaeditor.org/wiki/index.php/Java-Editor/de) kann für die Erstellung graphischer Oberflächen nützlich sein.

Die Verwendung rekursiver Datentypen (Liste oder Baum) ergibt sich normalerweise von selbst. Hier ist es eine Designentscheidung des Projektteams, ob für das konkrete Projekt dann universelle Strukturen mit der Einbindung der Datenelemente über eine Schnittstelle zum Tragen kommen oder ob die Listen bzw. Bäume Referenzen auf konkrete Klassen besitzen, weil sie Operationen zur Verfügung stellen müssen, die nur mit Objekten genau dieser Klasse möglich sind. Der erste Fall ist ein typischer Griff in den Bausteinkasten. Im zweiten Fall (Listen oder Bäume mit speziellen, von den jeweiligen Daten abhängigen Methoden) wird nach bekannten Vorlagen (Kompositum) ein neuer Baustein erstellt. Beide Entscheidungen erfüllen den Lehrplan.

Eine weitere Anforderung an Projektthemen entsteht aus der Lehrplanvorgabe, dass das „Zusammenspiel der verschiedenen Beschreibungstechniken beim Systementwurf: Datenmodellierung – Ablaufmodellierung – funktionale Modellierung – Objektmodellierung“ beinhaltet sein muss. Dass Objektmodellierung verwendet wird, gilt für jedes Thema. Ablaufmodellierungen werden für die Beschreibung von Methoden (Algorithmik) oder z. B. für die Beschreibung der Oberfläche (Zustandsmodell) ebenfalls in so gut wie jedem Thema benötigt. Um die Datenmodellierung sinnvoll anwenden zu können, ist es nötig, dass das zu erstellende Informatiksystem persistente Daten benötigt, die in einer Datenbank gespeichert werden. Hier ergibt sich für die formale Beschreibung der Bedeutung größerer Abfragen auch ein Anwendungsbereich funktionaler Modellierung.

Hier ist ein breites Feld für Schwerpunktsetzungen möglich. So kann ein Projekt den Schwerpunkt in der Algorithmik legen: Es soll ein Auskunftssystem für Radwege erstellt werden. Das Kartenmaterial (Orte, Wege) wird in einer Datenbank gespeichert. Die Datenbankanbindung ist hier relativ einfach (nur Abfrage), während der Dijkstra-Algorithmus zur Berechnung der kürzesten Verbindung höhere Ansprüche an eine Untergruppe stellt.

Umgekehrt stellt eine Aktiensimulation größere Anforderungen an die Datenbankanbindung (Datenpflege, Aggregation und Gruppenbildung bei der Abfrage), während keine komplexeren Algorithmen benötigt werden. Dieses Thema bietet auch Raum für fächerübergreifende Arbeit, hier mit Wirtschafts- und Rechtslehre.

2.2.3 Datenbankanbindung

Bisher haben die Schülerinnen und Schüler zwar die Verwendung einer Datenbank über SQL-Anweisungen kennengelernt; diese Anweisungen wurden aber in der Regel immer über eine graphische Benutzeroberfläche eingegeben oder sogar von dieser erzeugt. Neu für die Jugendlichen ist, dass die Ansteuerung der Datenbank von einem selbst erstellten Programm aus erfolgen muss. Für die hier vorgestellten Projekte ist allerdings nur die Datenmanipulation und -abfrage von Interesse, die Datendefinition (Erzeugen, Warten und Löschen von Tabellen) kann weiterhin mit den bekannten Werkzeugen durchgeführt werden. Im Projekt kann jedoch natürlich mehr gefordert werden.

Zweckmäßigerweise wird die Ansteuerung der Datenbank in einer eigenen Klasse zusammengefasst. Nur die Gruppe, die diese Klasse erstellt, muss sich daher mit den Details und der genauen Implementierung der Datenbankanbindung beschäftigen. Trotzdem ist für alle Mitglieder des Teams wichtig, die prinzipielle Struktur der heute üblichen Klassenbibliotheken für die Datenbankansteuerung zu kennen. Nur so können sie die Möglichkeiten für die Leistungen der projekteigenen Klasse zur Datenbankanbindung einschätzen.

Für das Modell der Ansteuerung der Datenbank genügen zwei Klassen. Objekte der Klasse DATENBANKZUGRIFF verwalten die Kommunikation mit der Datenbank. Die Klasse DATENBANKZUGRIFF hat vier zentrale Methoden.

VerbindungAufbauen(serveradresse, datenbank, benutzer, passwort)
Diese Methode stellt die Verbindung zwischen Programm und Datenbank her. Parameter sind die für den Zugriff auf die Datenbank wichtigen Informationen wie Internetadresse des Datenbankservers oder Dateipfad zur Datenbankdatei, gegebenenfalls die Datenbank, falls der Server mehrere Datenbanken verwaltet und Benutzer und Passwort, wenn das Datenbank-management eine Benutzerkontrolle durchführt.

AnweisungAbsetzen(SQL-Anweisung)
Damit werden Datenmanipulationsanweisungen (Einfügen, Ändern und Löschen von Datensätzen) ausgeführt. Die Anweisungen werden als Zeichenkette übergeben, die die gewünschte SQL-Anweisung enthält.

AbfrageAbsetzen(SQL-Abfrage)
Damit werden Abfrageanweisungen ausgeführt. Die Abfragen werden als Zeichenkette übergeben, die die gewünschte SQL-Abfrage enthält. Das Ergebnis ist ein Objekt der Klasse ERGEBNISTABELLE, welches die Ergebnistabelle der Abfrage verwaltet. Es hängt vom Datenbankmanagementsystem und der Klassenbibliothek ab, ob dieses Objekt nur so lange verwendet werden kann, bis die nächste Abfrage abgesetzt wird oder ob die Ergebnistabellen mehrerer Abfragen gleichzeitig bearbeitet werden können (Regelfall).

VerbindungSchließen()
Diese Methode beendet die Verbindung zur Datenbank und gibt eventuell Ressourcen wie noch nicht vollständig bearbeitete Ergebnistabellen frei. Diese Methode sollte immer aufgerufen werden, wenn die Verbindung nicht mehr benötigt wird, insbesondere beim Ende des Programms.

Eine zweite Klasse ERGEBNISTABELLE beschreibt die Objekte zur Verwaltung der Ergebnistabelle von Abfragen. Die Ergebnistabelle wird grundsätzlich Datensatz für Datensatz bearbeitet. Die Felder des gerade aktiven Datensatzes können mit einer weiteren Methode gelesen werden.

NaechstenDatensatzPositionieren()
Diese Methode macht den Nachfolger des gerade aktiven Datensatzes zum aktiven Datensatz. Als Rückgabe liefert die Methode den Wert wahr, wenn noch ein Datensatz existiert, sonst falsch. Nach Ausführen einer Abfrage ist noch kein Datensatz aktiv, sodass die Methode *NaechstenDatensatzPositionieren* ausgeführt werden muss, bevor der erste Datensatz gelesen werden kann. Damit lassen sich Datensätze in einer sehr einfachen Wiederholung bearbeiten:

```
wiederhole solange NaechstenDatensatzPositionieren()
     <Bearbeiten des Datensatzes>
endewiederhole
```

FeldwertGeben (name)
Liefert den Wert des angegebenen Feldes aus dem aktiven Datensatz. Manche Implementierungen haben die Einschänkung, dass die Felder nicht benannt sind, sondern nur über Nummern (ab 0 oder ab 1) angesprochen werden können.

Der folgende Programmtextausschnitt zeigt die Umsetzung in Java für die Ansteuerung einer MySQL-Datenbank oder alternativ (nur zwei unterschiedliche Zeilen) einer ODBC-Quelle (z. B. Access-Datenbank) aus dem Beispiel Aktienhandel. Alle nicht zur Datenbankansteuerung notwendigen Teile (wie der Erzeuger-Teil des Beobachtermusters) sind weggelassen; von den Methoden sind nur jeweils ein Beispiel für das Auswerten einer Anfrage und für eine Datenmanipulation angegeben.

```
//Klassen für das JDBC-Paket
import java. sql. *;

class DATENBANKVERBINDUNG implements STATUSERZEUGER
{
    //Treiber für MySQL
    private String treiber = "com.mysql.jdbc.Driver";
```

```
    //Treiber für ODBC
    //private String treiber =
        "sun.jdbc.odbc.JdbcOdbcDriver";
    // URL der MySQL-Datenbank
    private String pfad =
        "jdbc:mysql://localhost/aktienhandel";
    //URL der ODBC-Datenquelle
    //String pfad ="jdbc:odbc:aktienhandel";
    private String user = "aktien";
    private String pWort = "handel";
    // Das Verbindungsobjekt
    private Connection conn;
    // Hält die Referenz auf das einzige existierende
    // Objekt der Klasse DATENBANKVERBINDUNG
    private static DATENBANKVERBINDUNG verbindung =
        new DATENBANKVERBINDUNG ();

    /**
     * Der Konstruktor öffnet die Datenbankverbindung
     */
    private DATENBANKVERBINDUNG()
    {
        // Datenbankverbindung öffnen
        // Darf nur ein mal ausgeführt werden, daher wird
        // das Muster Singleton verwendet.
        // ODBC bzw. JDBC Treiber laden
        // (Seit Java 5 würde es automtisch gemacht werden,
        // ist so aber sicherer.)
        try
        {
            Class.forName(treiber);
        }
        catch(Exception e)
        {
            System.out.println
              ("Start_Fehler beim Laden des Treibers\n"+ e);
            System.exit(-1);
        }

        //Öffnen der Datenbank
        try
        {
            conn = DriverManager.getConnection
                      (pfad, user, pWort);
        }
        catch(Exception e)
        {
            System.out.println
               ("Start_Fehler bei Datenbankzugriff\n"+ e);
            System.exit(-1);
        }
    }

    /**
     * Beendet die Verbindung zur Datenbank,
     * alle Resourcen werden freigegeben.
     */
    void Schliessen()
    {
        try
        {
```

```
            conn. close();
            verbindung = null;
        }
        catch(Exception e)
        {
            Fehlermeldung
      ("Fehler beim Beenden der Datenbankverbindung\n"+ e);
        }
    }

    /**
     * Liste der Aktien im Depot herausgeben
     * @return Feld mit den IDs und Namen der Aktien.
     */
    String [] AktienNamenHolen ()
    {
        ResultSet sqlResult;
        Statement stmt;
        String[] liste;
        try
        {
            //Objekt der Klasse Statement erzeugen
            stmt = conn.createStatement();
            // Mit diesem Statement-Objekt eine Anfrage
            // absetzen; die Ergebnistablle wird über
            // ein Objekt der Klasse ResultSet bearbeitet
            sqlResult = stmt.executeQuery
                ("SELECT AktienID, Name FROM aktie");
            // Um die Anzahl der Ergebniszeilen zu ermitteln,
            // holt man sich am Besten die Zeilenenummer
            // der letzten Zeile
            sqlResult.afterLast();
            sqlResult.previous();
            // Damit wird das Ergebnisfeld angelegt
            liste = new String [sqlResult. getRow ()];
            // Wieder auf den Anfang positionieren …
            sqlResult.beforeFirst();
            int i = 0;
            // … und die Ergebnistabelle zeilenweise
            // abarbeiten
            while (sqlResult.next())
            {
                liste [i] =
                    sqlResult. getString ("AktienID") + " " +
                    sqlResult. getString ("Name");
                i += 1;
            }
            // Ergebnistabelle auf dem Server freigeben
            sqlResult. close ();
            // Restliche Ressourcen freigeben
            stmt.close();
            return liste;
        }
        catch(Exception e)
        {
            Fehlermeldung
              ("AktienHolen: Fehler bei SQL Abfrage\n" + e);
            return null;
        }
    }
```

```
/**
 * Verkauf aus einem Paket eintragen
 * @param paketID Schl&uuml;ssel des Pakets
 * @param aktienID Schl&uuml;ssel der AKtie
 * @param anzahl Anzahl der zu verkaufenden Aktien
 * @param restAnzahl Anzahl der restlichen Aktien im
                     Paket
 * @param kaufdat Datum des Kaufs
 * @param kaufkurs Kurs beim Einkauf
 * @param verkaufdat Datum des Verkaufs
 * @param verkaufkurs Kurs beim Verkauf
 */
void VerkaufEintragen
    (int paketID, int aktienID, int anzahl,
     int restAnzahl, String kaufdat, double kaufkurs,
     String verkaufdat, double verkaufkurs)
{
    Statement stmt;
    try
    {
        //Objekt der Klasse Statement erzeugen
        stmt = conn.createStatement ();
        // Damit die erste Anweisung absetzen
        stmt.executeUpdate
            ("INSERT INTO verkaeufe (AktienID,
              Stueckzahl, KDatum, KKurs, VDatum,
              VKurs) "+
             " VALUES("+ aktienID +","+ anzahl + ", '"+
              kaufdat +"',"+ kaufkurs +", '"+ verkaufdat
              + "'," + verkaufkurs +");");
        // Weitere Anweisungen absetzen
        if (restAnzahl > 0)
        {
            stmt.executeUpdate
                ("UPDATE depotaktie SET Stueckzahl = " +
                 restAnzahl + " WHERE PaketNummer = " +
                 paketID);
        }
        else
        {
            stmt.executeUpdate
             ("DELETE FROM depotaktie WHERE PaketNummer = " +
              paketID);
        }
        // Ressourcen frei geben
        stmt.close ();
    }
    catch(Exception e)
    {
        Fehlermeldung
    ("VerkaufEintragen: Fehler bei SQL Abfrage\n"+ e);
    }
}
```

3 Anwendungsorientierte Aufgaben

Im Informatiklehrplan der Jahrgangsstufe 11 wird insbesondere der Einsatz der vermittelten rekursiven Datenstrukturen bei der Bearbeitung von Beispielen aus der Praxis verlangt. Die folgenden fünf Aufgabenvorschläge sollen einen Eindruck vermitteln, wie solche Praxisbeispiele ausschauen könnten. Dabei erstreckt sich die Bandbreite von typischen praktischen Anwendungen, beispielsweise die Aufgabensammlung „IT-Kompetent“, über Simulationen, beispielsweise die Sammlung „Verschiebebahnhof“ bis hin zu klassischen Anwendungsaufgaben, beispielsweise die Aufgaben zur Postfix-Notation.

Allen Vorschlägen liegen folgende Grundintentionen zugrunde:

- Nutzen und Anwenden aller im Unterricht erarbeiteten rekursiven Datenstrukturen
- Sinnvolles Anpassen und Erweitern der Grundfunktionalitäten dieser Datenstrukturen, falls die Anwendung dies erfordert
- Vertiefen des Umgangs mit Algorithmen

Grundsätzliche Idee der Aufgaben ist, dass Datenstrukturen nicht mehr neu implementiert werden, sondern die im Unterricht entwickelten Klassen verwendet werden. Damit wird den Schülerinnen und Schülern klar, dass die auf ein allgemeines Konzept aufbauenden Datenstrukturen in vielerlei Anwendungen ohne großen Aufwand eingesetzt und genutzt werden können.

Natürlich lassen sich die Aufgabenstellungen auch so modifizieren, dass die notwendigen Datenstrukturen zur Wiederholung teilweise oder auch im Gesamten von den Schülerinnen und Schülern neu implementiert werden.

Zu allen Aufgaben gibt es Lösungshinweise und Lösungsvorschläge. Die Beispielimplementationen wurden mit BlueJ erstellt, d. h., die verwendete Programmiersprache ist Java. Bei den Lösungsvorschlägen war es nicht vorrangig das Ziel, besonders elegante und kurze Lösungen zu entwickeln, sondern in nachvollziehbaren Schritten zu arbeiten. Klar sollte sein, dass insbesondere bei Implementierungen viele Wege zum Ziel führen.

In vielen Fällen können die Aufgaben insbesondere in algorithmischer Hinsicht leicht erweitert werden, beispielsweise bei Einschränkungen in den Aufgabenformulierungen, die meist aus Gründen der einfacheren Umsetzung gemacht wurden (siehe z. B. Aufgabe „IT-Kompetent – Verwaltung der Kundendaten“: Bei der Implementierung der Methode *KundeLoeschen()* Berücksichtigung des Falles, dass die zu löschenden Kundendaten gar nicht existieren). Dies kann insbesondere zur Wiederholung und Vertiefung programmiertechnischer Kompetenz wie auch zur Differenzierung, insbesondere bei Schülern mit guten Programmierkenntnissen, gut ausgenutzt werden.

3.1. IT-Kompetent

3.1.1 Aufgabenstellung

Hinweise: Dieser Aufgabenvorschlag kann im Unterricht parallel zur Abarbeitung von Lehrplanpunkt Inf 11.1.1 (Listen) und Inf 11.1.2 (speziell geordneter Binärbaum) eingesetzt werden, da er aus aufeinander aufbauenden Aufgabenteilen besteht.
Es ist aber auch möglich, nur Einzelaufgaben zu bearbeiten. Dabei kann durchaus auf die Lösungsvorschläge der Aufgaben zurückgegriffen werden, auf die die ausgewählte Aufgabe aufsetzt.

Stoffbezug: 3.1.1.1:
- Schlange

3.1.1.2
- Ungeordnete Liste

3.1.1.3
- Geordnete Liste

3.1.1.4
- Geordneter Binärbaum
- Baumdurchlauf

3.1.1.1 IT-Kompetent – Grundaufgabe

Die sehr stark frequentierte telephonische Hotline der Firma IT-Kompetent, die Verwaltungssoftware für Schulen entwickelt, funktioniert nach folgendem Prinzip:

Der Anrufer, der Kunde der Firma ist, gibt seine Kundennummer und eine Kurzbeschreibung des Problems an. Nach Abgabe der Daten wird der Anrufer in der Regel in eine Warteschleife geführt. Ein freier Berater kann den am längsten wartenden Anrufer abrufen und aufgrund der eingegebenen Daten, die dem Berater angezeigt werden, kompetent beraten.

Dieses System soll nun durch eine Klasse IT_HOTLINE simuliert werden. Die Dateneingabe des Anrufes und der Abruf des am längsten wartenden Anrufes sollen dabei über eine geeignete Methode der Klasse IT_HOTLINE erfolgen. Außerdem kann zu jeder Zeit die Anzahl der in der Warteschleife befindlichen Anrufer abgefragt werden.

a) Entwerfen Sie ein Klassendiagramm.
b) Implementieren Sie das System.

3.1.1.2 IT-Kompetent – Verwaltung der Kundendaten (Erweiterung von 3.1.1.1)

Nun wird das Hotline-System der Firma IT-Kompetent verfeinert. Die Kundendaten (Kundennummer, Schulname, eingesetzte Software) sind im System in einer einfach verketteten Liste gespeichert. Vereinfachend wird davon ausgegangen, dass jeder Kunde nur eine Software von IT-Kompetent einsetzt.

Für die Datenpflege sollen folgende Methoden zur Verfügung stehen:

- *NeuenKundenAufnehmen(...)*

  Dabei sollen alle Kundendaten aufgenommen und der Kunde im System „abgespeichert“ werden.

- *KundenLoeschen(…)*

  Der Kunde mit der übergebenen Kundennummer wird aus dem System gelöscht. Es wird davon ausgegangen, dass die entsprechenden Daten im System gespeichert sind.

- *KundendatenAusgeben(kundennummer)*

  Die Daten des Kunden mit der gewünschten Kundennummer werden auf der Konsole ausgegeben. Dazu wird das entsprechende KUNDE-Objekt in der Liste (rekursiv) gesucht und, falls die Suche erfolgreich war, zurückgegeben. Anschließend werden die Daten ausgegeben.

Die zur Suche notwendigen Methoden in der Datenstruktur Liste sind ggf. noch zu implementieren.

- *SchulnameAendern(kundennummer, neuerSchulname)*

  Der Schulname des Kunden mit der übergebenen Kundennummer soll geändert werden können. Dabei wird davon ausgegangen, dass der Kunde existiert.

Zudem werden dem Berater beim Abruf eines Anrufers gleichzeitig die gespeicherten Daten des Kunden auf der Konsole angezeigt.

Implementieren Sie in der Klasse IT_HOTLINE die oben geforderten Methoden zur Datenpflege sowie die Methode zum Abruf des nächsten in der Warteschleife befindlichen Anrufers mit entsprechender Anzeige der Daten.

3.1.1.3 IT-Kompetent – Überblick durch Sortierung (Erweiterung von 3.1.1.2)

Die Abteilungsleiterin Frau Meier, die für die Hotline von IT-Kompetent verantwortlich ist, möchte sich schnell einen Überblick verschaffen können, welche Kundendaten derzeit im System gespeichert sind. Dabei braucht sie diese Angaben nach Kundennummer sortiert.

Der hausinterne Programmierer entscheidet sich für folgendes (sicher nicht optimales) Verfahren:

- Es wird eine neue, anfangs leere Liste erzeugt.
- In diese Liste werden die Kundendatenobjekte der ungeordneten Liste einzeln an der richtigen Stelle einsortiert.

Setzen Sie folgende Methoden in der Klasse IT_HOTLINE um:

- *GesamteKundendatenGeordnetAusgeben():*

  Die Kundendaten werden, sortiert nach der Kundennummer, auf der Konsole ausgegeben. Dabei wird die Methode *KundendatenSortieren* genutzt.

- *KundendatenSortieren()*:

  Diese privat deklarierte Methode erzeugt nach dem oben beschriebenen Verfahren aus der vorliegenden ungeordneten Kundenliste eine sortierte Liste und gibt diese zurück.

Bemerkung: Der in der obigen Aufgabe verwendete Algorithmus zum Sortieren von Elementen wird in der Fachliteratur Insertion Sort genannt.

3.1.1.4 IT-Kompetent – jetzt effektiver speichern (Erweiterung von 3.1.1.2 oder 3.1.1.3)

Die Anzahl der im System gespeicherten Kunden mit ihren Daten wird immer umfangreicher. Deshalb entschließt sich IT-Kompetent, diese Daten nun in einem geordneten Binärbaum abzuspeichern. Dazu soll das derzeitige System, das als Datenstruktur noch eine Liste verwendet, „upgedatet“ werden.

a) Passen Sie das vorliegende Projekt diesbezüglich an.

   Denken Sie dabei auch an die Aktualisierung der Methoden. Dabei können Sie auf eine Umsetzung der Methode *KundenLoeschen(kundennummer)* verzichten, da der dahinterstehende Algorithmus nicht ganz einfach ist.

b) Implementieren Sie zusätzlich eine Methode *GesamteKundendatenGeordnetAusgeben(),* die die Kundendaten alphabetisch geordnet auf der Konsole ausgibt.

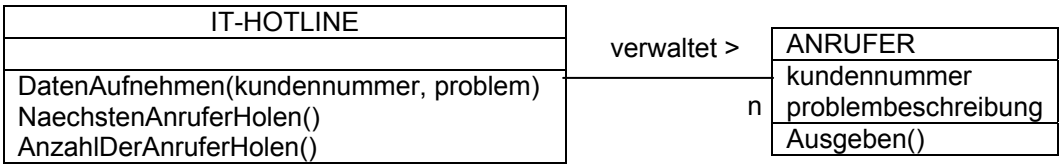
Hinweis:

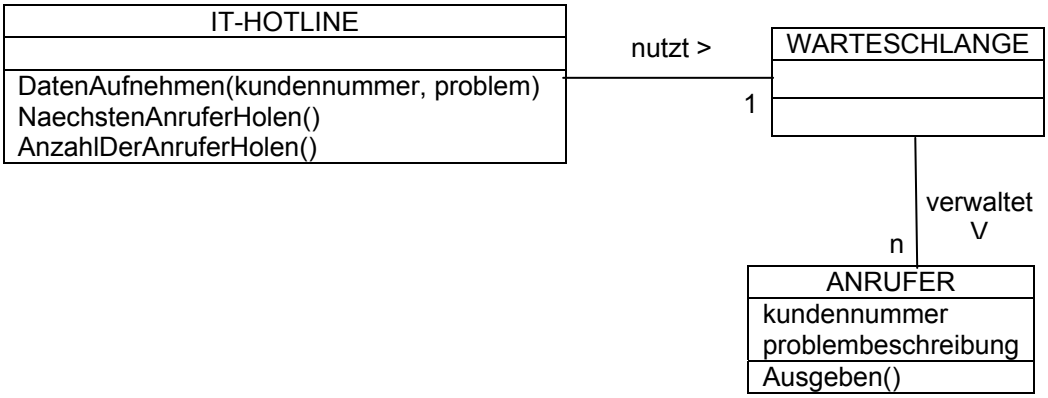
- Falls Sie einer der Aufgaben „IT-Kompetent – Verwaltung der Kundendaten“ bzw. „IT-Kompetent – Überblick durch Sortierung“ gelöst haben, können Sie eine dieser Lösung als „derzeitiges System“ verwenden. Ansonsten wird Ihnen das „derzeitige System“ von Ihrer Lehrkraft zur Verfügung gestellt.
- Man könnte weiterhin noch eine Methode *KundendatenUeberspielen()* implementieren, die die Daten aus der bisherigen Liste ausliest und diese dann im geordneten Binärbaum abspeichert.

3.1.2 Lösungshinweise und Lösungsvorschläge

3.1.2.1 Lösungshinweise: IT-Kompetent – Grundaufgabe

a) In der allgemeinsten Form könnte man sich folgende Klassendiagramme vorstellen. Je nach Umsetzung der Warteschlange können zusätzliche Klassen notwendig werden.





b) Die Umsetzung der Warteschlange kann je nach Unterrichtsstand erfolgen durch

- ein Feld: siehe BlueJ-Projekt *IT-Kompetent-Grundaufgabe-Feld*
- eine verkettete Liste ohne Verwendung des Kompositums:
  siehe BlueJ-Projekt *IT-Kompetent-Grundaufgabe-ohneKompositum*
- eine verkettete Liste unter Verwendung des Kompositums:
  siehe BlueJ-Projekt *IT-Kompetent-Grundaufgabe-mitKompositum*

3.1.2.2 Lösungshinweise: IT-Kompetent – Verwaltung der Kundendaten

Lösungsvorschlag: siehe BlueJ-Projekt *IT-Kompetent-VerwaltungDerKundendaten*

Anmerkungen zu den Lösungsideen:

- Es wird nun zwischen Anrufern (Klasse ANRUFER) und Kunden (Klasse KUNDE) unterschieden. KUNDE „verwaltet“ die im System abgelegten Kundendaten.
- *NeuenKundenAufnehmen* und *KundenLoeschen(kundennummer)* sind einfach umzusetzen; sie nutzen die bereits in der Implementation der Handreichung vorgegebenen Listen-Methoden *Entfernen* und *EndeEinfuegen.*
- *KundendatenAusgeben(kundennummer)* erfordert die Erweiterung der Methoden der Datenstruktur Liste
  - LISTE: DatenSuchen(DATENELEMENT vergleichsdaten) sucht in der Liste den Knoten, der die den Vergleichsdaten (Parameter vergleichsdaten) entsprechenden Daten verwaltet
  - LISTENELEMENT, KNOTEN und ABSCHLUSS benötigen entsprechende Methoden *DatenAbHierSuchen*

Als Schlüssel wird die Kundennummer verwendet. Die Methode *Vergleichen* in KUNDE wird entsprechend angepasst.

- *SchulnameAendern(kundennummer, neuerSchulname)* nutzt die gerade angesprochenen Methoden zum Finden eines bestimmten Datenobjekts.
- *NaechstenAnruferHolen():* Die Kundennummer wird aus dem ANRUFER-Objekt „ausgelesen“, dazu ist eine Methode *KundennummerGeben()* in ANRUFER notwendig. Mit der ausgelesenen Kundennummer kann dann die Methode *KundendatenAusgeben* genutzt werden.

3.1.2.3 Lösungshinweise: IT-Kompetent – Überblick durch Sortierung

Lösungsvorschlag: siehe BlueJ-Projekt *IT-Kompetent-UeberblickDurchSortierung*

Anmerkungen zu den Lösungsideen:

- Es werden die bereits in der Implementation der Handreichung vorgegebenen Listen-Methoden *AnfangGeben(), SortiertEinfuegen()* und *AlleAusgeben()* genutzt.
- Bei der Erzeugung der geordneten Liste in *KundendatenSortieren()* wird die ungeordnete Liste „abgebaut“. Um den Datenbestand nicht zu verlieren, wird die Kundenliste deshalb in eine temporäre Liste kopiert.

Zum Testen wird beim Aufruf des Konstruktors von IT_HOTLINE bereits ein kleiner Kundenstamm angelegt:

Kundenummer	Schulname	Software
1	Gymnasium Passau	Windols
78	Gymnasium München	Linus
4	Gymnasium Nürnberg	Linus II
19	Gymnasium Vilshofen	Windols

3.1.2.4 Lösungshinweise: IT-Kompetent – jetzt effektiver speichern

Lösungsvorschlag: siehe BlueJ-Projekt *IT-Kompetent-JetztEffektiverSpeichern*

a) Es ist eigentlich nicht viel zu tun!

   Es werden nun gleichzeitig die Datenstrukturen Liste und Baum benötigt. In den in der Handreichung vorgestellten Implementierungen nutzen beide Strukturen die Klassen KNOTEN und ABSCHLUSS, die aber abhängig von der Datenstruktur unterschiedliche Implementierungen enthalten. Aus diesem Grund werden die entsprechenden Klassennamen mit dem Zusatz _LISTE bzw. _BAUM versehen. Grundsätzlich gäbe es sicher elegantere Möglichkeiten, beispielsweise das Verpacken der Strukturen in eigene Pakete. Der damit verbundene (Erklärungs-)Aufwand ist aber hinsichtlich des Verständnisses der Aufgaben nicht notwendig, weshalb darauf verzichtet wird.

b) Eine sortierte Ausgabe ist bei einem geordneten Binärbaum mithilfe eines Inorder-Durchlaufes möglich. Bei der Implementierung wird deshalb einfach auf die Methode *InOrder()* der Klasse BAUM zurückgegriffen.

3.2. Rangierbahnhof

3.2.1 Aufgabenstellung

Hinweise: Im Kapitel 1.1.4.1 der Handreichung wird im Zusammenhang mit der rekursiven Datenstruktur Stapel das Beispiel eines Rangierbahnhofs angedeutet. Dieses Beispiel soll nun simuliert und vertieft werden.

Stoffbezug:
- Stapel
- Umgang mit Algorithmen

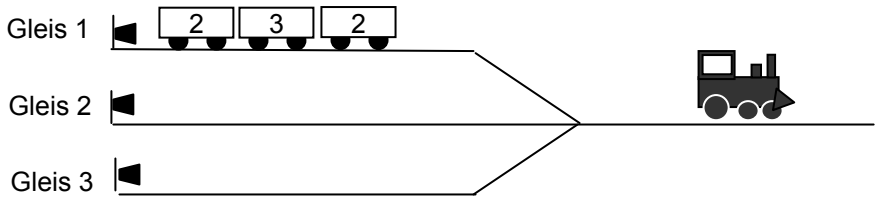
Ein Rangierbahnhof hat mehrere Abstellgleise. Güterzüge kommen in diesem Bahnhof an und werden zu neuen Zügen zusammengestellt.

In unserem Fall hat der Rangierbahnhof drei Gleise. Auf Gleis 1 stehen die Waggons, die umsortiert werden sollen. Dazu steht eine Lokomotive zur Verfügung, die aber immer nur einen Waggon transportieren kann.

Die Waggons haben eine eindeutige Waggonnummer und sind einem bestimmten Waggontyp (E = offener Güterwaggon; F = geschlossener Güterwaggon; I = Kühlwaggon) zugeordnet.

3.2.1.1 Rangierbahnhof – Situation aus dem Kapitel Listen

Die Waggons sollen auf Gleis 2 und Gleis 3 umrangiert werden. Dazu hat der Rangiermeister vorher die Waggons mit einem Zettel markiert, auf dem das Zielgleis angegeben ist.



a) Entwickeln Sie ein Programm, das den Rangiervorgang simuliert. Die Abstellgleise sollen dabei mithilfe der Datenstruktur STAPEL umgesetzt werden. Die zentrale Klasse RANGIERBAHNHOF soll dabei mindestens folgende Methoden haben:

- *NeuenWaggonAufGleis1Stellen*

  Damit kann Gleis 1 nacheinander mit Waggons unter Berücksichtigung der jeweiligen Attribute bestückt werden.

- *AktuelleGleisbelegungGeben*

  Auf der Konsole wird die aktuelle Belegung aller drei Gleise ausgegeben.

- *Simulieren*
  Bei Aufruf dieser Methode wird der Rangiervorgang durchgeführt.

b) Testen Sie Ihr Programm mit verschiedenen Belegungen von Gleis 1.

c) Implementieren Sie nun folgende Methoden:

i) *AnzahlWaggonGeben(gleisnummer)*

Es soll die Anzahl der Waggons auf einem bestimmten Gleis bestimmt und auf der Konsole ausgegeben werden.

ii) *WaggonSuchen(waggonnummer)*

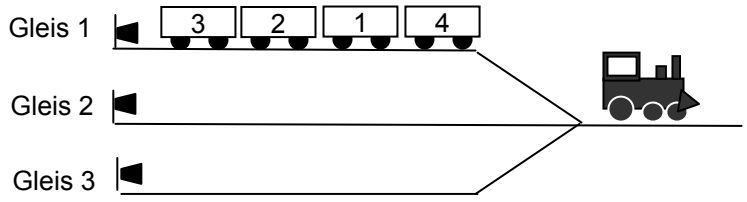
Es soll der Waggon mit einer bestimmten (eindeutigen) Waggonnummer gesucht werden. Auf der Konsole wird die Nummer des Gleises ausgegeben, auf der sich der Waggon befindet.

iii)*AnzahlMitWaggontypGeben(gleisnummer, typ)*

Es soll die Anzahl der Waggons eines bestimmten Waggontyps auf dem gewünschten Gleis bestimmt und auf der Konsole ausgegeben werden.

3.2.1.2 Rangierbahnhof – Abgeänderte Situation

Die Reihenfolge der Waggons, die auf Gleis 1 stehen, soll nun geändert werden. Dazu hat der Rangiermeister die Waggons mit Nummern markiert. Die Waggons sollen nun so rangiert werden, dass sie nach dem Rangiervorgang am Gleis 3 nach der vom Rangiermeister vergebenen Nummer geordnet sind. Dabei soll der Waggon mit der kleinsten Nummer ganz vorne, d. h. bei der Lokomotive, stehen.



Folgender Algorithmus löst das Problem:

1. Verschiebe die Waggons von Gleis 1 auf 3, solange die richtige Reihenfolge auf Gleis 3 gewährleistet ist.

2. Wenn die Nummer des (nächsten) Waggons auf Gleis 1 größer als die Nummer des Waggons auf Gleis 3 ist, verschiebe solange Waggons von Gleis 3 nach Gleis 2, bis der Waggon von Gleis 1 an der richtigen Stelle auf Gleis 3 eingefügt werden kann.
3. Wiederhole Schritt 1 und 2, bis Gleis 1 leer ist.
4. Ist das Gleis 2 nicht leer, so verschiebe alle Waggons von Gleis 2 nach Gleis 1 und beginne bei Schritt 1.

a) Vollziehen Sie die Strategie mit der obigen Gleisbelegung auf einem Blatt Papier nach.

b) Setzen Sie die Simulation mit einem Programm um. Der Rangiervorgang soll dabei wieder in der Methode *Rangieren()* umgesetzt werden. Dabei soll die Anzahl der Rangierschritte mitprotokolliert und auf der Konsole ausgegeben werden.

Es stellt sich nun die Frage, ob man den Rangiervorgang in der Regel schneller, d. h. in weniger Schritten, durchführen kann.

Beim obigen Algorithmus kann man dazu beispielsweise bei Schritt 4 ansetzen:

Gleis 2 hat beim ersten Durchlauf der Schritte 1 bis 3 die Funktion eines Hilfsgleises. Man kann nun auf das Umrangieren von Gleis 2 auf Gleis 1 verzichten, wenn man bei einem erneut notwendigen Durchlauf der Schritte 1 bis 3 das Gleis1 als Hilfsgleis verwendet. Ist nach dem Arbeiten der Schritte 1 bis 3 das Hilfsgleis 1 noch mit Waggons belegt, so übernimmt wiederum Gleis 2 die Funktion des Hilfsgleises.

c) Vollziehen Sie die geänderte Strategie mit obiger Gleisbelegung wiederum auf Papier nach und zeigen Sie, dass mit dieser Verbesserung das Umrangieren bei der obigen Situation in acht Schritten möglich ist.

d) Setzen Sie den verbesserten Algorithmus in einer Methode *RangierenBesser()* um und testen Sie die Methoden für verschiedene Gleisbelegungen.

e) Ist der verbesserte Algorithmus in jedem Fall schneller als der ursprüngliche? Begründen Sie kurz!

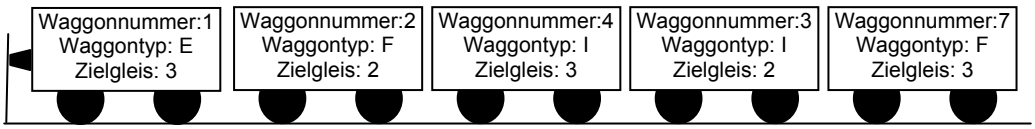
3.2.2 Lösungshinweise und Lösungsvorschläge

3.2.2.1 Lösungshinweise: Rangierbahnhof – Situation aus dem Kapitel Listen

Für die Umsetzung beider Lösungsvorschläge wurde die in Kapitel 1.1.4.2 vorgestellte Implementierung (Anwendung des Adaptermusters) verwendet.

a) Lösungsvorschlag: BlueJ-Projekt *Rangierbahnhof_Handreichung*

Beim Erzeugen eines Objekts der Klasse RANGIERBAHNHOF wird Gleis 1 ist zum Testen bereits mit 5 Waggons belegt:



Bemerkungen zur Verwaltung der Waggons:

Zur Umsetzung der Simulation benötigt man eine Klasse WAGGON, die die waggonspezifischen Attribute (hier die eindeutige Waggonnummer, die Waggonklasse und das

Zielgleis) sowie speziell auf die vorgegebene Situation abgestimmte Methoden (beispielsweise zum Abrufen des Zielgleises) hat.

Insbesondere bei den speziellen Methoden ist es nicht sinnvoll, sie in der Oberklasse DATENELEMENT zu schreiben, da sie ja in keinem unmittelbaren Zusammenhang mit der Listen- bzw. Stapelstruktur stehen.

Wenn man nun in der Klasse RANGIERBAHNHOF zum Umgang mit den WAGGON-Objekten den Datentyp DATENELEMENT verwendet, kommt es beim Aufruf der WAGGON-spezifischen Methoden zu Problemen. Dieses Problem kann nur gelöst werden, wenn in RANGIERBAHNHOF an den betroffenen Stellen der Datentyp WAGGON statt DATENELEMENT verwendet wird. Da aber die Methoden der (in der Handreichung vorgestellten) Listenmethoden auf den Datentyp DATENELEMENT ausgerichtet sind, sind weitere Anpassungen notwendig. Zwei Ansätze sind denkbar:

(1) Es wird ganz auf die Klasse DATENELEMENT verzichtet. Das erfordert aber die Anpassung sämtlicher Klassen an das spezielle Datenobjekt WAGGON. Damit ist aber die Datenstruktur Stapel bzw. Liste nicht mehr allgemein verwendbar.

(2) Die Klasse DATENELEMENT wird beibehalten. An den kritischen Stellen der Klasse RANGIERBAHNHOF wird Typkonvertierung angewendet.

In der Lösung wird Ansatz 2 realisiert, da er unter Ausnutzung der bereits vorliegenden Implementierungsvorschläge aus Kapitel 1.1.4.2 am wenigsten aufwendig ist.

Falls keine Implementierung der Liste bzw. des Stapels vorliegt, wäre sicherlich auch Ansatz 1 überlegenswert.

Die Schülerinnen und Schüler sollten jedenfalls beide Varianten umsetzen können.

b) Die Bedeutung des Testens sollte den Schülerinnen und Schülern immer wieder nahegebracht werden.

c)

i) *AnzahlWaggonGeben(gleisnummer):* relativ einfache Aufgabe

Es kann die bereits in der Klasse LISTE implementierte Methode *AnzahlGeben* verwendet werden, die über eine entsprechende, vom Schüler zu programmierende Methode *AnzahlGeben* in der Klasse STAPEL genutzt werden kann.

ii) *WaggonSuchen(waggonnummer):* etwas schwierigere Aufgabe

- Es kann wiederum die bereits in der Klasse LISTE implementierte Methode *Suchen* genutzt werden.
- In *WaggonSuchen* muss dann noch gewährleistet werden, dass alle drei Gleise durchsucht werden.
- Die Methode *Vergleichen* in der Klasse WAGGON vergleicht dabei die Waggonnummer, die das Schlüsselattribut des Waggons ist.

iii) *AnzahlMitWaggontypGeben(gleisnummer, typ):* schwierigere Aufgabe

Nachdem die Suche nach Datenobjekten mit bestimmten Eigenschaften doch auch im allgemeinen Fall von Datenobjekten öfter benötigt wird, wird hier die Entscheidung getroffen, die Liste um eine allgemeine Methode *AnzahlMitEigenschaftGeben* zu erweitern. Dies erfordert eine Anpassung folgender Klassen durch Hinzufügen entsprechender Methoden:

- in LISTE, STAPEL: *AnzahlMitEigenschaftGeben* ähnlich der Methode *AnzahlGeben*
- in ABSCHLUSS, KNOTEN: *AnzahlMitEigenschaftAbHierGeben* ähnlich der Methode *AnzahlAbHierGeben*
- in DATENELEMENT: *EigenschaftVergleichen*, evtl. abstrakt
- in WAGGON: *EigenschaftVergleichen*, wobei die zu vergleichende Eigenschaft der Waggontyp ist

3.2.2.2 Lösungshinweise: Rangierbahnhof – Abgeänderte Situation

a) Lösung in 13 Schritten

Rangierschritt	Gleis 1	Gleis 2	Gleis 3
	3 2 1 4		
G1 → G3	3 2 1		4
G1 → G3	3 2		4 1
G3 → G2	3 2	1	4
G1 → G3	3	1	4 2
G3 → G2	3	1 2	4
G1 → G3		1 2	4 3
G2 → G1	2	1	4 3
G2 → G1	2 1		4 3
G1 → G3	2		4 3 1
G3 → G2	2	1	4 3
G1 → G3		1	4 3 2
G2 → G1	1		4 3 2
G1 → G3			4 3 2 1

b) Lösungsvorschlag: BlueJ-Projekt *Rangierbahnhof_AbgeaenderteSituation02*

Beim Erzeugen eines Objekts der Klasse RANGIERBAHNHOF wird Gleis 1 bereits mit der in der Aufgabenstellung gegebenen Waggonkonstellation belegt.

(Bei dem Lösungsvorschlag wurden nicht benötigte Methoden der Klassen LISTEN, KNOTEN und ABSCHLUSS aus Übersichtlichkeitsgründen weggelassen. Auch auf die Attribute Waggonnummer und Waggontyp wurde verzichtet!)

- Die Umsetzung des Algorithmus in *Rangieren()* ist nicht so einfach, da beim Vergleichen von Waggonnummer insbesondere beachtet werden muss, dass das betreffende Gleis nicht unbesetzt sein darf.
- Das Zählen der Rangierschritte ist beispielsweise über eine lokale Variable *anzahlschritte*, die bei jedem Rangierschritt um 1 erhöht wird, in *Rangieren()* möglich.

c) Lösung in 8 Schritten

Rangierschritt	Gleis 1	Gleis 2	Gleis 3	Bemerkung
	3 2 1 4			Gleis 2 ist Hilfsgleis
G1 → G3	3 2 1		4	
G1 → G3	3 2		4 1	
G3 → G2	3 2	1	4	
G1 → G3	3	1	4 2	
G3 → G2	3	1 2	4	
G1 → G3		1 2	4 3	Gleis 1 ist Hilfsgleis
G2 → G3		1	4 3 2	
G3 → G2			4 3 2 1	

d) Die entscheidende Idee ist die Einführung von zwei Hilfsvariablen *Startgleis* und *Hilfsgleis*, in denen das jeweilige Start- bzw. Hilfsgleis abgespeichert wird. Der Wechsel von Start- und Hilfsgleis ist das in der Jahrgangsstufe 10 normalerweise diskutiertes Problem des Tausches zweier Variablenwerte mithilfe einer Hilfsvariablen.

e) Nein, der optimierte Algorithmus ist nicht immer schneller. Ist die Reihenfolge der Waggons auf Gleis 1 bereits (von der Lokomotive gesehen) absteigend sortiert, so benötigen beide Algorithmen die gleiche Anzahl von Schritten. In diesem Fall wird nämlich Schritt 4 des ursprünglichen Algorithmus gar nicht durchgeführt.

3.3 Postfixnotation

3.3.1 Aufgabenstellung

Hinweise:

- Die Vertiefungen 1 und 2 sind zwar generell nicht sehr schwierig, aber aufgrund der erforderlichen Beachtung einiger Sonderfälle (Zugriff auf leeren Stapel etc.) „liegt der Teufel ein bisschen im Implementierungsdetail“.
- Vertiefung 3 behandelt den aus der Mathematik bekannten Rechenbaum. Dieser kann als Binärbaum, jedoch nicht als geordneter Binärbaum dargestellt werden, da ein Operand mehrfach vorkommen kann und es keine Ordnungsrelation zwischen Operatoren und Operanden gibt. Das Beispiel zeigt aber, wie man einen Baum aus Teilbäumen zusammensetzen kann, was aber laut Lehrplan in der Implementierung nicht umgesetzt werden muss. Als Vertiefung oder Ausblick ist die Aufgabe sicher interessant.

  Die Aufgabe eignet sich aber auch zum Üben der Durchlauf-Methoden, da diese unabhängig vom geordneten bzw. nicht geordneten Binärbaum sind. Man könnte den Lösungsvorschlag ohne Durchlauf-Methoden angeben und nur die Implementierung der Methoden *TermInInorderAusgeben(), TermInPostorderAusgeben()* und *TermInPreorderAusgeben()* verlangen.

Stoffbezug: 3.3.1.1

- Stapel
- Umsetzung von Algorithmen

3.3.1.2 Vertiefung 1

- Umsetzung von Algorithmen

3.3.1.3 Vertiefung 2

- Umsetzung von Algorithmen

3.3.1.4 Vertiefung 3

- Stapel
- Binärbaum
- Baumdurchlauf
- Umsetzung von Algorithmen

Die allgemein gebräuchliche und auch in der Schule verwendete Schreibweise von Termen, beispielsweise 4 + 3 oder (a - b) * d, wird als Infixnotation bezeichnet, da sie die Operatoren, (also die Rechenzeichen) zwischen die Operanden (also die Zahlen bzw. Variablen) setzt. Bei der Postfixnotation werden dagegen zuerst die (zwei) Operanden und anschließend der Operator aufgeschrieben. Obige Beispiele lauten in Postfixnotation also 4 3 + bzw. a b – d *. Beim zweiten Beispiel fällt auf, dass keine Klammern mehr benötigt werden.

Hinweis: Wir betrachten hier nur so genannte zweistellige Operatoren, also Operatoren, die zwei Operanden benötigen. Dies sind bei uns die typischen Rechenoperatoren +, -, / und *. Weiterhin gibt es beispielsweise auch einstellige Operatoren, wie beispielsweise das Fakultätszeichen ! oder die Potenz hoch 2.

Die Postfixnotation ist bei der Termberechnung durch Rechner vorteilhaft, da damit eine Termauswertung unter Verwendung eines Stapels einfach möglich ist.

a) Informieren Sie sich per Internetrecherche genauer über die Infix- und Postfixnotation.

3.3.1.1 Überführung der Infix- in die Postfixnotation

Die Überführung eines Terms von der Infix- in die Postfixnotation gelingt mit nachfolgendem Algorithmus.

Hinweis: Dabei wird von folgenden Vereinfachungen ausgegangen:

- Die Reihenfolge der Rechenoperationen beim Term in Infixnotation wird durch Klammern vorgegeben, d. h., die Regeln „Punkt vor Strich“ und „Rechnen bei gleichwertigen Rechenzeichen von links nach rechts, falls keine Klammern vorhanden sind“ gelten nicht.

  Beispiele: a + (b * c) statt a + b * c

  (a + b) + c statt a + b + c

- Als Operatoren stehen nur die vier Grundrechenoperatoren zur Verfügung: +, - , *, /
- Die Terme enthalten nur einziffrige Zahlen. Ebenso bestehen Variablennamen nur aus einem Zeichen.

```
term = Term in Infixschreibweise;
termInPostfixnotation = leer;
zeichenposition = 1;
WIEDERHOLE BIS alle Zeichen von term abgearbeitet sind:
        Untersuche das Zeichen, das im Term an zeichenposition-ter Stelle steht
                und erhöhe den Wert von zeichenposition um 1
```

```
WENN dieses Zeichen eine öffnende Klammer ist
DANN  lege dieses auf dem Stapel ab
SONST WENN dieses Zeichen eine (einziffrige) Zahl oder eine Variable ist
        DANN hänge dieses Zeichen an TermInPostfixnotation
        SONST WENN dieses Zeichen eine schließende Klammer
                DANN nimm solange das oberste Zeichen vom Stapel und hänge es
                an termInPostfixnotation, bis das oberste Stapelelement eine öffnende
                Klammer ist. Lösche die öffnende Klammer vom Stapel.
                SONST WENN dieses Zeichen ein Rechenzeichen ist
                        DANN lege das Zeichen auf dem Stapel ab
                        ENDE WENN
                ENDE WENN
        ENDE WENN
ENDE WENN
ENDE WIEDERHOLE
Hänge noch auf dem Stapel liegende Zeichen nacheinander an termInPostfixnotation
```

b) Vollziehen Sie den Algorithmus auf einem Blatt Papier für folgende Terme nach:
   i. (2 + c) – d
   ii. 3 * (a + b) – 4.

c) Entwickeln Sie ein Programm, das einen Term von der Infix- in die Postfix-Notation überführt.

   Der notwendige Stapel soll dazu Objekte der Klasse ZEICHEN verwalten, deren Objekte für jeweils ein Zeichen zuständig sind.

   Die zentrale Klasse UMWANDLER soll dabei folgende Methoden enthalten:

   - *TermEingeben(String term)*

     Eingabe des Terms in Infixschreibweise, bei Verwendung von BlueJ beispielsweise durch den entsprechenden Methodenaufruf.

   - *InInfixnotationUmwandeln(String term)*

     Überführung des in Infixnotation vorliegenden Terms in Postfixschreibweise nach obigem Algorithmus.

   - *ErgebnisAusgeben()*

     Ausgabe des Terms in beiden Notationen auf der Konsole. Die Methode wird am Ende des Umrechungsvorgangs von der Methode *InInfixnotationUmwandeln* genutzt.

     Hilfestellung:

     In Java stellt beispielsweise die Klasse String nützliche Methoden zum Umgang mit Zeichenketten bereit, u. a.:
     char charAt(int index): Liefert das Zeichen aus dem String, das an der Position *index* steht. Achtung: Das erste Zeichen hat den Index 0.

Hinweis für „Java-Profis“:

Java stellt für einfache Datentypen, wie z. B. *int* oder *char*, sogenannte Wrapper-Klassen, wie z. B. *Integer* oder *Charakter*, zur Verfügung. Diese Wrapper-Klassen kann man nun

bei unseren Listenimplementierungen gut einsetzen, da diese Listen Objekte verwalten müssen. Man erspart sich bei der obigen Aufgabe die Programmierung einer eigenen Klasse ZEICHEN. Die Wrapper-Klassen stellen bereits einige Methoden zur Konvertierung von bzw. in Strings zur Verfügung. Informieren Sie sich im WWW ggf. genauer über diese Wrapper-Klassen.

3.3.1.2 Vertiefung 1 – Berücksichtigung klammerloser Ausdrücke, also z. B. Beachtung von der „Punkt vor Strich“-Regel

Man kann die „Vorfahrtsregeln“, die in der Aufgabe „Überführung der Infix- in die Postfix-Notation“ mithilfe von Klammern realisiert wurden, auch direkt im Algorithmus umsetzen. Dazu muss die in der Aufgabe „Überführung der Infix- in die Postfixnotation“ definierte „Regel“

```
WENN dieses Zeichen ein Operator  ist
        DANN lege das Zeichen auf dem Stapel ab
ENDE WENN
```

durch folgende Regeln ersetzt werden:

- Ist das Zeichen ein Operator und ist der Stapel leer, so wird das Zeichen auf dem Stapel abgelegt.
- Ist das Zeichen ein Operator mit höherem Rang als das oben auf dem Stapel befindliche, so wird das Zeichen auf dem Stapel abgelegt.
- Ist das Zeichen ein Operator mit gleichem oder niedrigerem Rang, als das oben auf dem Stapel befindliche, dann wird, solange dies der Fall ist, das obere Zeichen vom Stapel genommen und an termInPostfixnotation angehängt. Anschließend wird das Zeichen auf den Stapel gelegt.

a) Testen Sie den Algorithmus für folgende Terme auf einem Blatt Papier:

   i) a + b * 2

   ii) a / b * c

b) Erweitern Sie Ihr Programm aus der Aufgabe „Überführung der Infix- in die Postfix-Notation“, d. h., passen Sie die Methode *InInfixnotationUmwandeln* entsprechend an!

   Hilfreich ist es, einige Hilfsmethoden zu entwickeln, die *InInfixnotationUmwandeln* nutzen kann, beispielsweise

   - *OperatorrangIstHoeher(char zeichen)*

     Die Methode prüft, ob der Operator *zeichen* einen höheren Rang als das oberste Zeichen auf dem Stapel hat, d. h., es wird die Regel „Punkt vor Strich“ umgesetzt. In diesem Fall liefert die Methode wahr zurück, in allen anderen Fällen, beispielsweise wenn das oberste Zeichen auf dem Stapel gar kein Rechenzeichen ist, falsch.

   - *ObersteszeichenIstOperator()*

     Es wird geprüft, ob das oben auf dem Stapel liegende Zeichen ein Operator ist und entsprechend des Ergebnisses der Überprüfung wird ein boolescher Wert zurückgegeben.

c) Testen Sie Ihr Programm für folgende Terme:

   1 + 2; (1+ 2); 1 + 2 * 3; (1+ 2) *3; 1 + 2 * 3 – 4 / 5; 1 + 2 * (3 – 4) / 5

3.3.1.3 Vertiefung 2 – Berechnung von Termen

Ein (variablenfreier) Term in Postfix-Notation kann mithilfe eines Stapels sehr einfach ausgewertet werden.

a) Entwickeln Sie am Beispiel 3 4 5 + * [= 3 * (4 + 5)] einen Algorithmus für diese Auswertung und formulieren Sie diesen in Pseudocode.

Alternative zu Aufgabe a)

Der Auswertungsalgorithmus lautet folgendermaßen:

```
WIEDERHOLE BIS alle Zeichen von TermInPostfixnotation abgearbeitet sind:
    Spalte das erste Zeichen des Terms ab
    WENN dieses Zeichen eine (einziffrige) Zahl ist
        DANN lege dieses auf dem Stapel ab
    ENDE WENN
    WENN dieses Zeichen ein Rechenzeichen ist
        DANN
        Hole die oberste Zahl vom Stapel, sie ist der 2. Operand
        Hole die oberste Zahl vom Stapel, sie ist der 1. Operand
        Führe die Rechenoperation aus
        Lege das Ergebnis auf dem Stapel ab
    ENDE WENN
ENDE WIEDERHOLE
```

Vollziehen Sie den Algorithmus am Beispiel 3 4 5 + * [= 3 * (4 + 5)] nach.

b) Setzen Sie den in Teilaufgabe a) entwickelten Algorithmus in einer Methode *TermAusrechnen()* um.

Der dazu benötigte Stapel soll Objekte der Klasse ZAHL verwalten. Achten Sie dabei auf folgende Sondersituation bei der Konvertierung von *Character* in *Integer*:

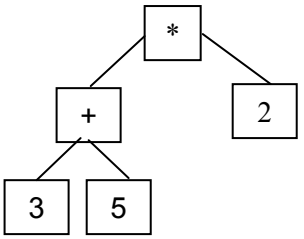
Die Typkonvertierung *(int) zeichen* liefert den entsprechenden Unicode-Wert des in *zeichen* abgespeicherten Char-Wertes. (Unicode ist ein internationaler Standard, in dem langfristig für jedes sinntragende Schriftzeichen oder Textelement aller bekannten Schriftkulturen und Zeichensysteme ein digitaler Code festgelegt wurde.)

Im Falle von *char zeichen = 3; int zahl = (int) zeichen* ist der Wert von *zahl* nach der Zuweisung 51, da der Unicode-Wert des Zeichens 3 eben 51 ist. Um nun auf den tatsächlichen Wert zu kommen, kann man beispielsweise von *zahl* den Unicode-Wert des Zeichens *0*, der *48* ist, abziehen. Durch diesen Trick wird die Typkonvertierung richtig durchgeführt: int zahl = (int) zeichen – (int) ’0’, der Wert von zahl beträgt dann 51 – 48 = 3.

Hinweis:
- Der obige „Konvertierungsvorschlag“ ist einfach, er ist aber zugegebenermaßen nicht besonders elegant. In der Klassenbibliothek von Java werden auch „echte“ Konvertierungsmethoden zur Verfügung gestellt. Diese können von Interessierten natürlich ebenfalls verwendet werden.
- Beachten Sie, dass die Berechnung von Quotienten bei Verwendung des

Datentyps int auf ein ganzzahliges Ergebnis führt, d. h., der Divisionsrest wird ignoriert. Zum Testen empfiehlt sich deshalb die Verwendung von Zahlen, die ohne Rest teilbar sind.

3.3.1.4 Vertiefung 3 – Rechenbaum



Aufgabe: Wie Sie bereits seit der 5. Jahrgangsstufe wissen, kann die Struktur eines Terms durch einen Rechenbaum dargestellt werden. Nebenstehend sehen Sie einen solchen Baum, der den Term

(3 + 5) * 2

repräsentiert.

a) Handelt es sich bei dem Baum um einen

- Binärbaum?
- geordneten Binärbaum?

Begründen Sie jeweils Ihre Antwort.

Falls der Term nun in der Postfix-Notation, in unserem Beispiel also 3 5 + 2 *, vorliegt, ist eine Umwandlung in einen Rechenbaum sehr leicht möglich, indem man unter Verwendung eines Stapels folgenden Algorithmus anwendet:

(1) Führe nachfolgende Schritte nacheinander für alle Zeichen des Terms (in Postfix-Notation) durch:

- Wenn dieses Zeichen ein Operand ist, dann erzeuge einen Baum, der diesen Operanden in einem Datenobjekt verwaltet, und lege den Baum auf den Stapel.
- Ist das Zeichen ein Operator, also +, -, * oder /, so erzeuge ebenfalls einen Baum *neuerBaum*, der diesen Operator in einem Datenobjekt verwaltet.

  Nimm den obersten Baum vom Stapel und mache die Wurzel dieses Baumes zum rechten Nachfolger der Wurzel von *neuerBaum.*

  Nimm einen weiteren Baum vom Stapel und mache die Wurzel dieses Baumes zum linken Nachfolger der Wurzel von *neuerBaum.*

Lege den neuen Baum auf den Stapel.

(2)Auf dem Stapel liegt nun ein Knoten, der Wurzel des Rechenbaums ist.

b) Vollziehen Sie den Algorithmus für das obige Beispiel auf einem Blatt Papier nach.

c) Entwickeln Sie ein Programm, mit dem aus einem Term in Postfixnotation der entsprechende Rechenbaum aufgebaut werden kann. Beachten Sie folgende Hinweise:

- Die Operanden im Term sollen dabei aus Einfachheitsgründen einstellig sein.
- Die zentrale Klasse RECHENBAUM soll folgende Methoden implementieren:
  - *TermInPostfixnationEingeben(String term)*

    Eingabe eines in Postfixnotation eingegebenen Terms. Dieser Term soll als Attributwert eines geeigneten Attributs gespeichert werden.
  - *InBaumUmwandeln()*

Der in Postfixnotation abgespeicherte Term wird in einen Rechenbaum umgewandelt und ebenso als Attributwert eines geeigneten Attributs abgespeichert.

- Die Datenstruktur BAUM benötigt u. a. die Methoden *LinksEinfuegen (BAUM b)* bzw. *RechtsEinfuegen(Baum b),* mit der der übergebene Baum bei der Wurzel als linker bzw. rechter Teilbaum „eingehängt" werden kann. Die Idee ist dabei, im Wurzelknoten mit dem Attribut *linkerNachfolger* bzw. *rechterNachfolger* die Wurzel des übergebenen Baumes zu referenzieren.
- Der Stapel verwaltet Elemente vom Typ BAUM.

d) Um aus dem Rechenbaum den entsprechenden Term zurückzugewinnen, muss man die Baumelemente mithilfe des Preorder-, Inorder- oder Postorder-Durchlaufs ausgeben. Ergänzen Sie die Klasse RECHENBAUM um entsprechende Methoden *TermInPreorderAusgeben(), TermInPostorderAusgeben()* und *TermInPostorder-Ausgeben().*

e) Testen Sie die in d) entwickelten Methoden. Was fällt Ihnen beim Ergebnis der Methoden, insbesondere beim Inorder- und Postorderdurchlauf, auf? Können Probleme bei der Interpretation des Ergebnisses des Inorderdurchlaufs auftreten?

3.3.3 Lösungshinweise und Lösungsvorschläge

3.3.3.1 Lösungshinweise: Überführung der Infix- in die Postfix-Notation

b)

i)

Term in Infix-Notation	Stapel	Term in Postfix-Notation
(2 + c) – d		
2 + c) – d	(	
+ c) – d	(	2
c) – d	+ (	2
) – d	+ (	2 c
– d		2 c +
d	–	2 c +
	–	2 c + d
		2 c + d –

ii)

Term in Infix-Notation	Stapel	Term in Postfix-Notation
(3 * (a + b)) – 4		
3 * (a + b)) – 4	(	
* (a + b)) – 4	(	3
(a + b)) – 4	* (	3

a + b)) – 4	(* (	3
+ b)) – 4	(* (	3 a
b)) – 4	+ (* (	3 a
)) – 4	+ (* (	3 a b
) – 4	* (	3 a b +
– 4	–	3 a b + *
4	–	3 a b + *
	–	3 a b + * 4
		3 a b + * 4 –

c) Lösungsvorschlag: siehe BlueJ-Projekt *InfixInPostfix*

Zur Umsetzung des Algorithmus wurde die switch-Anweisung verwendet, weil es damit übersichtlich programmiert werden kann. Natürlich ist auch die Programmierung einer verschachtelten bedingten Anweisung möglich.

3.3.1.2 Lösungshinweise zu Vertiefung 1

a) Testen Sie den Algorithmus für folgende Terme auf einem Blatt Papier:

i)

Term in Infix-Notation	Stapel	Term in Postfix-Notation
a + b * 2		
+ b * 2		a
b * 2	+	a
* 2	+	a b
2	* +	a b
	* +	a b 2
	+	a b 2 *
		a b 2 * +

ii) analog

b) Lösungsvorschlag: siehe BlueJ-Projekt *InfixInPostfixVertiefungI*

Hinweise:

- In manchen Situationen muss nun ein Zugriff auf das oberste Zeichen des Stapels möglich sein, ohne das KNOTEN-Objekt löschen zu müssen.
- Es wäre natürlich auch möglich, dass man den obersten Knoten vom Stapel entfernt, den notwendigen Datenwert ausliest und dann den Knoten wieder auf den Stapel legt. Dann wäre keine Erweiterung der Methoden notwendig.
- Die Anpassung der Methode *InInfixnotationUmwandeln* ist gar nicht so einfach, da beim Vergleich mit dem obersten Zeichen auf dem Stapel viele Fälle zu beachten sind, u. a. ob die Liste leer ist und ob das oberste Zeichen ein Rechenzeichen ist. Bei der vorliegenden Implementierung wurden die Fälle sukzessive abgearbeitet. Man könnte die Methode sicher insgesamt kürzer und eleganter schreiben.
- Die Methode *RechenzeichenrangIstHoeher* stellt eine gute Übung für den Umgang mit logischen Operatoren da.

3.3.1.3 Lösungshinweise zu Vertiefung 2

a) Algorithmus:

```
WIEDERHOLE BIS alle Zeichen von TermInPostfixnotation abgearbeitet sind:
        Spalte das erste Zeichen des Terms ab
        WENN dieses Zeichen eine (einziffrige) Zahl ist
                DANN  lege dieses auf dem Stapel ab.
        ENDE WENN
        WENN dieses Zeichen ein Rechenzeichen ist
                DANN
                        Hole die beiden obersten Zahlen vom Stapel
                        Führe die Rechenoperation aus
                        Lege das Ergebnis auf dem Stapel ab
        ENDE WENN
ENDE WIEDERHOLE
```

b) Lösungsvorschlag: siehe BlueJ-Projekt *InfixInPostfixVertiefungII*

3.3.1.4 Lösungshinweise zu Vertiefung 3

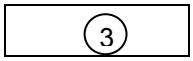
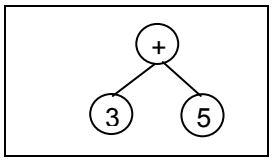
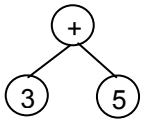
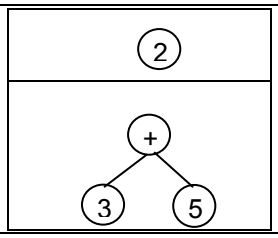
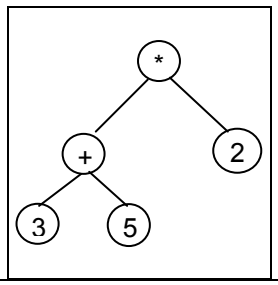
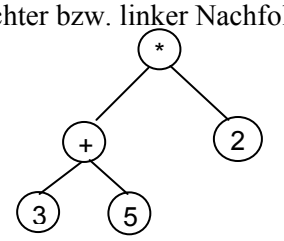
- Es handelt sich um einen Binärbaum, da jeder Knoten maximal zwei Nachfolger besitzt.

  Die inneren Knoten verwalten die Operatoren +, -, * und /. Diese sind zweistellige Operatoren, d. h., sie benötigen zwei Operanden, die jeweils in den Kindknoten bzw. im Teilbaum, dessen Wurzel der Kindsknoten ist, abgespeichert sind.

- Es handelt sich nicht um einen geordneten Binärbaum.

  Die Abspeicherung der Termzeichen erfolgt nicht über eine Ordnungsrelation, wie beispielsweise *größer als* (>) oder *kleiner als* (<). Operatoren und Operanden können in unserem Fall nicht bzgl. einer für die Auswertung des Terms vernünftigen Ordnungsrelation verglichen werden.

a) Term in Postfixnotation: 3 5 + 2 *

Betrachtetes Zeichen	Stapel	Aktion
3		Baum erzeugen, dessen Wurzel das Zeichen 3 verwaltet Ablage des Baumes im Stapel
5		Baum erzeugen, dessen Wurzel das Zeichen 5 verwaltet Ablage des Baumes im Stapel
+		Baum erzeugen, dessen Wurzel das Zeichen + verwaltet. Die obersten zwei Elemente des Stapels holen und als rechter bzw. linker Nachfolger anfügen  Ablage diese Baumes im Stapel
2		Baum erzeugen, dessen Wurzel das Zeichen 2 verwaltet Ablage des Baumes im Stapel
*		Baum erzeugen, dessen Wurzel das Zeichen * verwaltet. Die obersten zwei Elemente des Stapels holen und als rechter bzw. linker Nachfolger anfügen.  Ablage diese Baumes im Stapel
		Holen des vollständigen Baumes aus dem Stapel

b) Lösungsvorschlag: siehe BlueJ-Projekt *Rechenbaum*

c) Lösungsvorschlag: siehe BlueJ-Projekt *Rechenbaum*

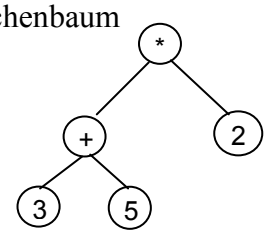
d)

- *TermInInorderAusgeben()* liefert den Term in Infix-Notation, also in der uns geläufigen Notation;
- *TermInPostorderAusgeben()* liefert den Term in Postfix-Notation;

  Problem bei der Rückverwandlung in die Inorder-Notation ist, dass notwendige Klammern nicht gesetzt werden, die die richtige Rechenreihenfolge erzwingen.

  Beispiel:

  Der Term (3 + 5) * 2 hat in der Postorder-Notation die Form 3 5 + 2 *. Der Term wird damit in den Rechenbaum



  umgewandelt. *TermInInorderAusgeben()* liefert: *3 + 5 * 2*

  Aber (3 + 5) * 2 ≠ 3 + 5 * 2

  Es sind also noch Überlegungen erforderlich, wie Klammern richtig zu setzen sind.

- Bei der *TermInPreorderAusgeben()* folgen die Operatoren vor den Operanden. Man spricht von der Prefix-Notation bzw. Präfix-Notation. Diese Notation hat bei Termen keine besondere Bedeutung.