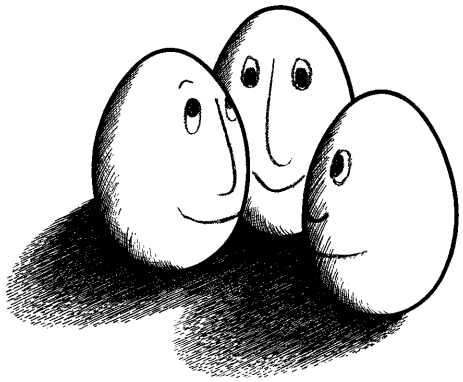
Programmierung I: JAVA
Skript zur Vorlesung
WS 1998/99

Katharina Morik

22. November 1998



Inhaltsverzeichnis

Danksagung

Bei dem Abenteuer, JAVA an der Universität Dortmund für die Erstsemestervorlesung einzuführen und eine neue Vorlesung mit Skript, Folien, Beispielprogrammen zu gestalten, haben mich viele Menschen unterstützt. Prof. Vornberger und Frank Thiesing von der Universität Osnabrück haben mir Skript, Beispiele und Aufgaben ihrer wunderbaren JAVA-Vorlesung *Algorithmen* zur Verfügung gestellt. An unserem Fachbereich hat mir insbesondere Arnulf Mester Mut gemacht und Doris Schmedding hat sorgfältig im Skript Fehler aufgespürt. Daß bei der kurzen Zeit für die Vorlesungsvorbereitung (nennen Sie die niemals "Semesterferien"!) dennoch Fehler im Skript verbleiben, versteht sich von selbst. Am Lehrstuhl 8 haben die wissenschaftlichen Mitarbeiter Ralf Klinkenberg und Stefan Haustein manche Stunde mit mir und für mich am Rechner verbracht. Sascha Lüdecke hat Implementationen, Text zum Listenabschnitt und Humor beigesteuert. Dirk Burkamp hat zur Darstellung der verteilten und nebenläufigen Programmierung, aber auch zu Diskussionen über Sinn und Zweck der Vorlesung wesentlich beigetragen. Stefan Rüping hat ein Programm zur Überführung von JAVA-Code in Text.tex geschrieben. Er ist auch der Autor des Anhangs zur Rechnerbenutzung. Andreas Greve hat die Installation der Programme vorgenommen und das Skript Korrektur gelesen.

Selbstverständlich hat die Informatikrechner Betriebsgruppe alles getan, um den Übungsbetrieb in JAVA für die Studierenden angenehm zu machen.

Gebrauchsanweisung

Dieses Skript gibt einige Definitionen und Beispiele der Vorlesung *Programmierung – JAVA* wieder und verweist auf Bücher, in denen wichtige Inhalte der Vorlesung zu finden sind. Es ersetzt weder die Vorlesungsstunden noch die Lektüre der angegebenen Literatur. Es soll aber die Vor- und Nachbereitung der Vorlesung und die Vorbereitung auf die Klausur erleichtern. Die Vorlesungsunterlagen bestehen aus:

Skript: gibt die wichtigsten Definitionen in einheitlicher Terminologie wieder; zeigt durch den Index, was Studierende wissen müssen, um erfolgreich weiterstudieren zu können.

Mitschrift: schreiben Sie selbst; verdeutlicht eigene Schwierigkeiten und deren Auflösung.

Literatur: Die angegebenen Bücher teilen sich in drei Kategorien auf:

- JAVA-Bücher zum Nachschlagen von Konstrukten der Programmiersprache mit ihren Programmbibliotheken; als Einführungs- und Nachschlagewerke zur Programmiersprache JAVA seien empfohlen:

  Flanagan, David (1998): JAVA in a Nutshell,
  deutsche Ausgabe für JAVA 1.1,
  O'Reilly
  `http://www.oreilly.com/catalog/javanut2`

  Gosling, James, **Joy**, Bill, **Steele**,Guy (1997): JAVA – Die Sprachspezifikation, deutsche Ausgabe,
  Addison-Wesley
  `http://java.sun.com/books/Series`

Bishop, Judy (1998): JAVA Gently – Programming Principles Explained, 2nd edition,
Addison-Wesley
`http://www.cs.up.ac.za/javagently`

Campione, Mary, **Walrath**, Kathy(1998): The JAVA Tutorial, 2nd edition,
Addison-Wesley
`http://java.sun.com/docs/books/tutorial-second.html`

- Lehrbücher der Informatik (Programmierung); als Informatik-Bücher zur Programmierung empfehle ich:

  Goos, Gerhard (1996): Vorlesungen über Informatik Band 2 – Objektorientiertes Programmieren und Algorithmen,
  Springer

  Dißmann, Stefan, **Doberkat**, Ernst-Erich (demnächst): Einführung in die objektorientierte Programmierung mit JAVA

- vertiefende Literatur; die vertiefende Literatur wird im Skript angeführt an den Stellen, für die sie relevant ist.

Übungsaufgaben und ihre Lösungen : ergänzen die Mitschrift. Erfahrungsgemäß fallen gerade diejenigen Studierenden durch die Klausur, die nicht die Übungsaufgaben gemacht haben. Die eigenen Erfahrungen, die bei der Lösung von Aufgaben gesammelt werden, sind durch nichts zu ersetzen!

Programme : als Hilfsmittel für die eigene Programmierung sind einige JAVA-Klassen schon für Sie vorbereitet, die Sie dann einfach in Ihren Programmen wiederverwenden können. Sie sind im Verzeichnis `~gvpr000/ProgVorlesung/Packages/` zu finden. Weiterhin als Illustration und zum Ausprobieren gibt es zu jedem Abschnitt Beispielprogramme. Diese Programme sind im Verzeichnis `~gvpr000/ProgVorlesung/Beispiele/` gespeichert. Für Ihre eigenen Programme legen Sie selbst Verzeichnisse an. (Wie, sehen Sie im Anhang.)

1 Einleitung

> Grundsätzlich ist die Informatik jedoch die Wissenschaft der *Abstraktion* – das richtige Modell für ein Problem zu entwerfen und die angemessene mechanisierbare Technik zu ersinnen, um es zu lösen. [Aho und Ullman, 1996]

Bei der Programmentwicklung finden wir gemeinsam mit einem Anwender bzw. einer Auftraggeberin eine *Aufgabenbeschreibung*. Diese abstrahiert bereits von vielen konkreten Vorgängen, die das Programm realisieren soll. Anhand der Aufgabenbeschreibung wird ein *Modell* entwickelt, das die komplexe Aufgabe in Teile aufgliedert. Je nach Programmierparadigma sind diese Teile

- Klassen und Objekte (objektorientierte Programmierung),
- Funktionen (funktionale Programmierung) oder
- logische Beziehungen zwischen Sachverhalten (logische Programmierung).

Auf der abstrakten Ebene des Modells werden die Teile mit ihren Zuständigkeiten und Kooperationen herausgearbeitet (s. Abschnitt 2.2).

Definition 1.1: Programmierung im Großen Die Konzentration auf das Zusammenwirken von Teilen nennt man Programmierung im Großen, weil hier von der inneren Gestalt der Teile abstrahiert wird. Betrachtet wird das Außenverhalten von Teilen. Der englische Terminus ist *programming in the large*.

Bei der Programmierung im Großen werden Arbeitsabläufe betrachtet, eine Architektur für die Software entworfen, die Mengen von Aufgaben zu Modulen anordnet und die Beziehungen zwischen den Modulen angibt. Die Teile sind also sehr grobe Aufgabenpakete.

Für die abstrakten Teile müssen konkrete Umsetzungen gefunden werden. Manche Umsetzungen sind inzwischen Standard geworden. So hat man beispielsweise für geordnete Mengen (z.B. Teilnehmer an einem Wettbewerb, nach ihrer Startnummer geordnet) das Modell der Liste gefunden. Das Modell der Liste kann dann durch die Datenstruktur einer verketteten Liste realisiert werden. In dem Modell wird ebenfalls angegeben, was mit der Liste gemacht werden soll. In der Teilnehmerliste wollen wir einen Teilnehmer finden – hat er sich wirklich angemeldet? Das Wesentliche an der verketteten Liste ist also, daß wir darin suchen. Eine gegebene Programmiersprache erlaubt dann eine bestimmte Formulierung der Datenstruktur.

Definition 1.2: Programmierung im Kleinen Die Ausformulierung von Teilen nennt man Programmierung im Kleinen, weil auf die Einzelheiten eines Teils geachtet wird. Betrachtet wird die interne Realisierung eines Teils. Der englische Terminus ist *programming in the small*.

Bei der Programmierung im Kleinen haben wir immer drei Fragen:

Was soll realisiert werden (eine Zahl, eine Liste, eine Menge...)?

Wie soll es realisiert werden (durch eine mit 16 Bit dargestellte ganze Zahl, durch eine verkettete Liste, ...)?

Warum ist die Realisierung vernünftig (weil sie nach nur endlich vielen Schritten bestimmt das Ergebnis ausgibt; weil sie meist nach nur 16 Sekunden das Ergebnis ausgibt)?

Wenn wir ein Modell und seine Realisierung erstellt haben, überlegen wir – wiederum abstrakt –, ob wir eine vernünftige Lösung gefunden haben. Was kann mit *vernünftig* gemeint sein?

Arbeitsablauf: Das Programm wird in einer bestimmten Arbeitssituation von bestimmten Menschen oder anderen Maschinen genutzt. Wird der Arbeitsablauf durch das Programm besser und für die Menschen angenehmer? Welche Ziele der an dem Arbeitsablauf beteiligten Menschen werden in welchen Anteilen erfüllt, welche nicht? Mit derlei Fragen beschäftigt sich das Fach *Informatik und Gesellschaft*.

Wartbarkeit und Wiederverwendbarkeit: Große Programme müssen gewartet werden. Zum einen, weil die Welt sich ändert, in der die Programme eingesetzt werden und die sie in Teilen abbilden. Zum anderen, weil man immer einen Fehler macht, etwas übersieht, wenn man ein Programm entwickelt. Deshalb ist die Wartung von Software ein wichtiges Thema der *Softwaretechnologie*. Bei wissensbasierten Systemen der *Künstlichen Intelligenz* ist die Wartung und Revidierbarkeit ein Forschungsschwerpunkt. Wiederverwendbarkeit bezeichnet die Möglichkeit, nicht immer wieder von vorn anfangen zu müssen, sondern Teile veralteter Programme in den neuen Programmen verwenden zu können. Voraussetzung dafür ist, daß möglichst unabhängige Programmteile (Module) nur durch wohldefinierte Schnittstellen mit anderen Programmteilen verbunden sind, daß Annahmen, die bei der Programmierung gemacht wurden, auch dokumentiert sind, daß man sich beim Programmieren um Allgemeinheit bemüht, statt sehr spezielle Lösungen zu programmieren.

Effektivität: Sind alle Fälle, die vorkommen können, abgedeckt? Dies ist die Frage nach der *Vollständigkeit*. Berechnen wir immer das richtige Ergebnis? Dies ist die Frage nach der *Korrektheit*. Das Spezialgebiet der Informatik, das sich mit diesen Fragen befaßt ist die *Programmverifikation*.

Effizienz: Innerhalb der *Komplexitätstheorie* wird die Schwierigkeit von Problemen untersucht. Dabei schätzen wir unter anderem den Aufwand ab, der im schlimmsten Fall von dem Programm zu erbringen ist: wie lange wird der Rechner uns im schlimmsten Falle auf ein Ergebnis warten lassen? Zusätzlich zur analytischen Aufwandsabschätzung kommt in der praktischen Informatik die empirische Überprüfung durch systematische Experimente. Wie lang braucht das Programm unter bestimmten, systematisch variierten Umständen, bis es ein Ergebnis liefert?

Im Berufsleben einer Informatikerin oder eines Informatikers spielen Modellierung und Entwicklung nach wie vor eine erhebliche Rolle. Dabei wird meist bereits vorhandene Software genutzt. Es macht die Qualität einer Informatikerin bzw. eines Informatikers aus,

1. Anwendern so gut zuzuhören, daß eine umfassende Aufgabenbeschreibung gemeinsam erarbeitet werden kann;

2. übersichtliche Modelle für komplexe Aufgaben zu erstellen;

3. Standardumsetzungen zu kennen und bei der Konkretisierung von Modellen einsetzen zu können;

4. sich anhand der Standardumsetzungen rasch in einer (auch: neuen, noch unbekannten) Programmiersprache wiederzufinden und von der Sprache in Programmbibliotheken bereitgestellte Umsetzungen zu nutzen;

5. die eigene Programmentwicklung klar zu dokumentieren und nach Effektivität und Effizienz zu bewerten.

In dieser Vorlesung werden die Punkte 2 bis 4 am Beispiel der objektorientierten Programmiersprache JAVA eingeführt. Der dritte Punkt verlangt viel Eigenarbeit anhand von Übungsaufgaben von Ihnen. Der erste Punkt wird hier noch durch Beispiele ersetzt und der letzte Punkt nur einführend angesprochen.

2 Einführung in objektorientiertes Modellieren

Objektorientiertes Modellieren ist nicht nur im Rahmen des Entwurfs eines Programms, das auf einer Rechenanlage abläuft, anwendbar. Ein objektorientiertes Modell kann Arbeitszusammenhänge, Organisationen oder die Konstruktion technischer Geräte beschreiben. Dann wird das Modell in Form von Diagrammen dargestellt, die von Menschen interpretiert werden. Setzen wir den objektorientierten Entwurf um in ein (objektorientiertes) Programm, so wird dieses Programm von der Programmiersprache interpretiert (ausgeführt).

2.1 Grundbegriffe

Wir fassen eine Aufgabe oder ein System als eine Menge miteinander kooperierender Einheiten (Teilsysteme) auf. Diese Einheiten oder Teile sind *Objekte* des Weltausschnitts, den wir modellieren. Ein Objekt ist ein Ding mit einer Tätigkeit: der Fußball, der rollt, meine Lampe, die leuchtet. Dabei ist ein Objekt immer ein ganz bestimmtes Ding (oder Wesen oder Abstraktum), also der blaue Ball von Uta, meine Schreibtischlampe. In der Philosophie spricht man von *Einzeldingen*.

> Zum Beispiel sind historische Ereignisse, materielle Objekte, Menschen und deren Schatten nach meinem wie nach den gängigsten Arten philosophischen Sprachgebrauchs sämtlich Einzeldinge; Eigenschaften, Zahlen und Gattungen dagegen nicht... [Strawson, 1959]

Objekte sind voneinander verschieden, auch wenn ihre Beschreibung es nicht deutlich macht. Selbst wenn eine Firma zwei Angestellte mit demselben Namen und demselben Geburtsdatum in derselben Abteilung hat, muß sie beiden ein Gehalt bezahlen! Zahlen sind deshalb keine Einzeldinge, weil sie stets mit sich selbst gleich sind: es gibt nur eine 1, egal wo, wofür und wie oft wir sie verwenden.

Objekte haben *Eigenschaften*, die wir angeben können: Utas Ball ist blau, meine Schreibtischlampe ist weiß. Wir unterscheiden, *daß* ein Objekt eine Eigenschaft hat – z.B. eine Farbe – davon, *welche Ausprägung* der Eigenschaft es hat – z.B. blau. Objekte können etwas tun oder auf Tätigkeiten reagieren: Utas Ball rollt, wenn sie ihn tritt, meine Lampe beleuchtet den Schreibtisch, wenn ich sie einschalte. Wir betrachten den Tritt

gegen den Ball als eine *Botschaft* an den Ball. Es wird ihm mitgeteilt, mit welcher Kraft, an welcher Stelle er getreten wird. Der Ball hat eine *Methode*, wie er auf eine Botschaft reagiert: er rollt in eine bestimmte Richtung eine bestimmte Strecke.

Objekte sind Exemplare (Beispiele, Instanzen) einer Klasse. Utas blauer Ball *ist ein* Ball, meine Schreibtischlampe *ist eine* Lampe. Alle Objekte einer Klasse haben die Eigenschaften und Methoden dieser Klasse. *Daß* ein Ball eine Farbe hat wird durch die Klasse festgelegt. Bei einigen Eigenschaften wird obendrein die Ausprägung einer Eigenschaft durch die Klasse angegeben. So habe ich die runde Form bei Utas Ball nicht angeben müssen, weil diese Ausprägung der Form für alle Objekte der Klasse gilt. Auch die Methode der Bewegung aufgrund eines Tritts muß nicht bei Utas Ball angegeben werden. Sie kann als Methode bei der Klasse beschrieben werden, so daß sie für alle Bälle gilt. Die Farbe ist allerdings eine Eigenschaft, in deren Ausprägung sich verschiedene Bälle unterscheiden. Die Ausprägung wird deshalb bei dem Objekt angegeben. Ein Objekt einer Klasse erhält die Eigenschaften und Methoden der Klasse[1]. Wenn ein neues Objekt einer Klasse erzeugt wird (Instanziierung), dann bekommt es alle Eigenschaften und eventuell einige Ausprägugen der Eigenschaften. Eine Klasse gibt ihre Eigenschaften und Methoden an ihre Objekte weiter.

Definition 2.1: Klasse Eine Klasse beschreibt die Eigenschaften und das Verhalten einer Menge gleichartiger Objekte. Die Klasse legt fest, daß die Objekte bestimmte Eigenschaften und Methoden haben. Sie kann für einige Eigenschaften auch die Ausprägung festlegen. Dann haben alle Objekte der Klasse diese Ausprägungen von Eigenschaften.

Definition 2.2: Objekt Ein Objekt ist ein Einzelding. Es erhält die Eigenschaften (ggf. auch mit Ausprägung) und Methoden seiner Klasse und kann darüberhinaus Ausprägungen für Eigenschaften haben, von denen die Klasse nur angibt, daß sie bei allen Objekten in irgendeiner Ausprägung vorhanden sind.

Es gibt Ober- und Unterklassen. So ist ein Mensch ein Säugetier und ein Säugetier ein Lebewesen. Die Oberklasse vererbt ihre Eigenschaften und Methoden an ihre Unterklassen. Wir erhalten eine Hierarchie. Ein Mann ist ein Mensch und hat damit auch alle Eigenschaften und Methoden von Säugetieren, die ja bereits alle Eigenschaften und Methoden von Lebewesen geerbt haben. Somit hat auch der Mann alle Eigenschaften und Methoden von Lebewesen (s. Abbildung 1).

Definition 2.3: Vererbung Der Begriff der Vererbung (inheritance) kann verschiedene Beziehungen zwischen Klassen ausdrücken, darunter [Goos, 1996]:

A ist ein B: Die Unterklasse A übernimmt alle Eigenschaften und Methoden der Klasse B ohne Einschränkung. Ausprägungen der Eigenschaften können für A angegeben sein, die bei B nicht festgelegt waren. Der Katalog der Eigenschaften kann bei A größer sein als bei B.

[1] Wenn ich von "Eigenschaften" spreche, ohne ihr Vorhandensein von ihrer Ausprägung zu unterscheiden, meine ich das Vorhandensein der Eigenschaft und gegebenenfalls die Ausprägung.

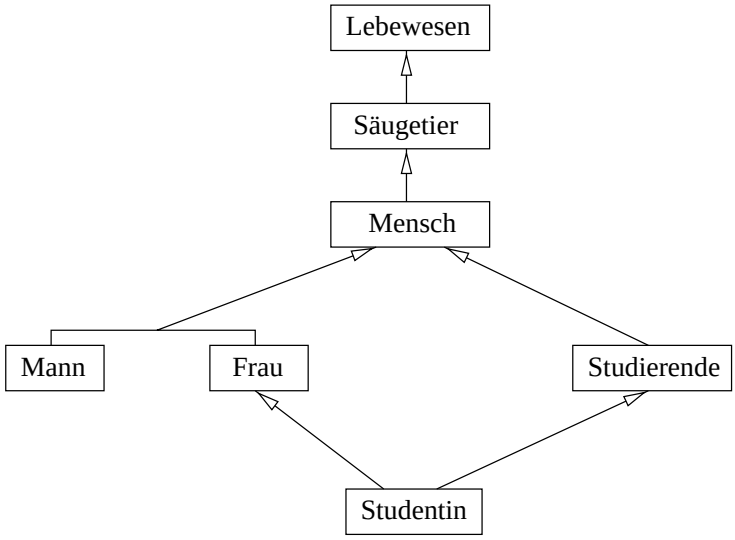


Abbildung 1: Vererbungshierarchie

A ist eine Spezialisierung von B: Ein gleichseitiges Dreieck (A) ist eine Spezialisierung eines Dreiecks (B), für das man keine drei unterschiedlichen Seitenlängen angeben kann. Hier ändert sich der Katalog der Eigenschaften von B insofern als die möglichen Ausprägungen der Eigenschaften in A eingeschränkt sind, so daß einige Eigenschaften entfallen.

A implementiert B: A realisiert die Konzepte von B. CD-Spieler realisieren digitale Abspielgeräte.

A verwendet von B: A verwendet den Code von B.

Definition 2.4: Mehrfachvererbung Erbt eine Klasse von mehreren Klassen, so spricht man von Mehrfachvererbung. Eine Studentin ist eine Frau und eine Studierende. (s. [Goos, 1996], S. 148; s. Abbildung 1).

Mit der Vererbungshierarchie und der Instanziierungsbeziehung zwischen einem Objekt und der Klasse, der es angehört, kommen wir nicht aus. So sollen beispielsweise Uta (in der Hierarchie Mensch, Säugetier, Lebewesen) und ihr blauer Ball (in der Hierarchie Ball, Kugel, unbelebtes, physikalisches Objekt) in Verbindung gebracht werden. Die Kooperation von Objekten beschreiben wir durch *Assoziationen*. Uta besitzt ihren blauen Ball. Uta tritt ihren Ball. Es wird bei Uta angegeben, daß sie einen blauen Ball besitzt. Uta ist dafür zuständig, wie sie ihren Ball tritt. Wenn auch bei dem Ball vermerkt werden soll, wem er gehört, so müssen wir eine weitere Assoziation einführen, die von dem Ball zu seiner Besitzerin führt. Der Ball ist dafür zuständig, wie er auf einen Tritt reagiert. In der Sprechweise der objektorientierten Methode: der Ball empfängt eine Botschaft von Uta und behandelt sie mit seiner Methode.

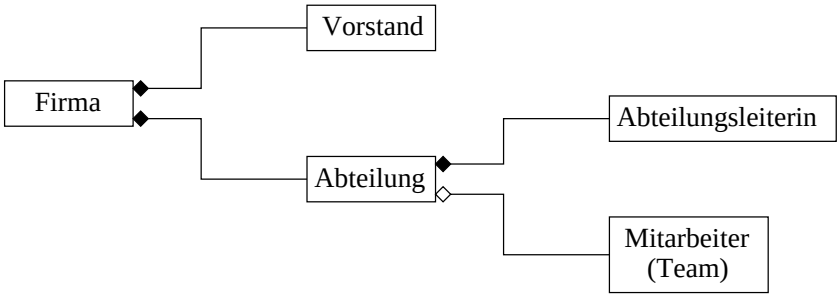


Abbildung 2: Aggregation und Komposition

Wir unterscheiden zwei Arten von Assoziationen, die eine Beziehung zwischen einem Ganzen und seinen Teilen ausdrücken: Aggregation und Komposition. Die *Aggregation* wird als *besteht aus* gelesen und mit einem leeren Rombus am Ende der Linie gezeichnet. So besteht meine Lampe aus dem Schirm, der Fassung, der Glühbirne, dem Kabel. Natürlich können wir alles in seine Bestandteile zerlegen. Es nützt aber für meinen Umgang mit der Lampe wenig, auch noch die Bestandteile der Fassung, der Birne, des Kabels zu modellieren. Ist die Lampe allerdings kaputt, so ist ein anderer als der normale Umgang erforderlich und eine feinere Modellierung wichtig. Der Zweck der Modellierung bestimmt also die Feinheit (*Granularität*) und damit, was als Objekt noch beschrieben und was schlicht ignoriert werden soll. Als Faustregel gilt: Alles, was Botschaften empfangen und mit einem anderen Objekt assoziiert werden soll, wird als Objekt aufgefaßt. Die typische Verwendung der Aggregation bezieht sich auf Organisationsstrukturen. Eine Firma besteht aus einem Vorstand und Abteilungen. Eine Abteilung besteht aus einem Abteilungsleiter oder einer Abteilungsleiterin und MitarbeiterInnen. Abteilungsleiter empfangen Berichte und Anregungen von ihren MitarbeiterInnen. Es muß nicht beschrieben werden, woraus Abteilungsleiter bestehen.

Einige Beziehungen zwischen einem Ganzen und seinen Teilen sind dergestalt, daß das Wegfallen des Ganzen zur Auflösung der Teile führt. Eine solche Assoziation wird *Komposition* genannt. Man kann eine Abteilung auflösen und hat damit immer noch die MitarbeiterInnen. Sie sind nun eben direkt der Firma zugeordnet. Es handelt sich also um eine normale Aggregation. Wenn aber die Firma aufhört zu existieren, dann fallen auch der Vorstand und die Abteilungen fort. (Natürlich gibt es die Menschen noch, aber nicht mehr in ihrer Rolle z.B. als Vorstandsmitglied.) Diese Assoziation ist eine Komposition. Die Komposition wird mit einem schwarz ausgefüllten Rombus gezeichnet. Ein Beispiel zeigt Abbildung 2.

Assoziationen können n Objekte mit m anderen Objekten in Verbindung bringen. So besucht eine Studentin mehrere Vorlesungen. Eine Professorin liest eine Vorlesung für mehrere Studierende. Hätten wir keine gerichtete Assoziation, sondern eine Relation zwischen n ProfessorInnen und m Studierenden, so wäre es eine $n : m$-Relation (wie in Abbildung 3).

Wenn wir aber eine gerichtete Assoziation haben, so geben wir nur entweder die Assoziation zwischen einer Studentin und den von ihr gehörten Vorlesungen an oder die Assoziation zwischen einem Professor und den von ihm gehaltenen Vorlesungen (s. Abbildung 4).

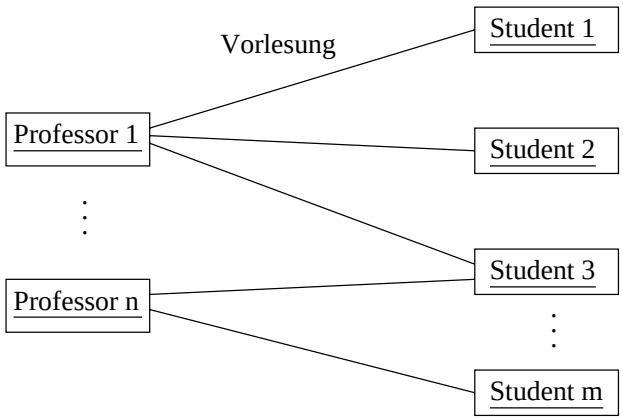


Abbildung 3: n zu m Relation

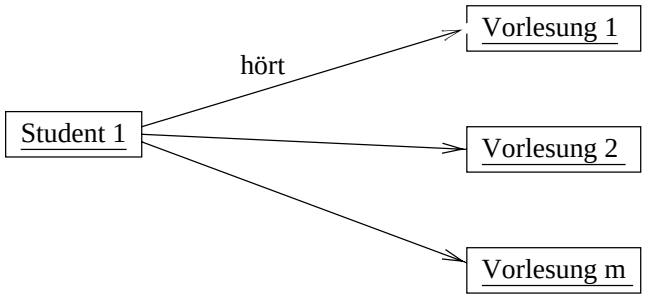


Abbildung 4: 1 zu m Assoziation

Definition 2.5: Assoziation Eine Assoziation ist eine Verbindung von einer Klasse bzw. einem Objekt zu einer anderen Klasse bzw. einem anderen Objekt. Insbesondere kann eine Assoziation von n Objekten einer Klasse auf m Objekte einer anderen Klasse gerichtet sein.

Nun wissen wir **was** wir neben der Vererbungsbeziehung noch brauchen und es stellt sich die Frage, **wie** wir es darstellen wollen. Wir können Assoziationen als Eigenschaften und Methoden darstellen.

- Wenn genau ein Objekt (z.B. meine Lampe) mit genau einem anderen Objekt (z.B. dem Lichtschalter in meinem Arbeitszimmer) assoziiert wird, so wird dies andere Objekt zur Eigenschaft des ersten Objektes (z.B. wird der Lichtschalter eine Eigenschaft der Lampe).

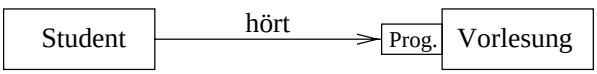


Abbildung 5: Qualifizierte Assoziation

- Wenn genau ein Objekt (z.B. eine Studentin) mit mehreren anderen Objekten (z.B. Vorlesungen) assoziiert ist, können diese vielleicht qualifiziert werden (z.B. Vorlesungen zur Mathematik, zur Programmierung), so daß die Assoziation zusammen mit der Qualifikation eindeutig wird (z.B. wird Vorlesung zur Programmierung eine Eigenschaft der Studentin). Dies illustriert Abbildung 5.

- Wenn aber notwendigerweise eine Assoziation auf mehrere andere Objekte einer Klasse gerichtet ist, so müssen wir ein Objekt einführen, das eine Kollektion darstellt. So können wir als Eigenschaft bei jedem Mitarbeiter anführen, wer für ihn zuständig ist. Es macht aber keinen Sinn, die Assoziation von der Abteilungsleiterin zu allen Mitarbeitern zu qualifizieren. Wir müßten dann im Voraus wissen, wieviele Mitarbeiter eine Abteilungsleiterin hat, um Eigenschaften der Art *mitarbeiter1* anzugeben. Stattdessen führen wir für jede Abteilung ein kollektives Objekt ein, das die Menge der Mitarbeiter darstellt. Jeder einzelne Mitarbeiter bleibt ein Objekt, das mit dem kollektiven Objekt, dem Team, verbunden ist. Eine Abteilungsleiterin kann dann entweder gezielt von einem bestimmten Mitarbeiter oder von dem Team Botschaften empfangen. So wird ein Verbesserungsvorschlag beispielsweise von einem bestimmten Mitarbeiter gemacht. Zum Geburtstag schenkt das Team etwas. Das Team ist ein Objekt einer Behälterklasse.

Definition 2.6: Assoziation, Darstellung Assoziationen werden als Eigenschaft bei der Klasse bzw. den Objekten angegeben. Bei $1:m$-Assoziationen mit $m > 1$ wird entweder eine Qualifikation vorgenommen, oder es wird ein zusammengesetztes Objekt bzw. eine Klasse von zusammengesetzten Objekten eingeführt. Eine solche Klasse, die eine Menge von Objekten ansammeln und verwalten kann, heißt auch *Behälterklasse*.

Definition 2.7: Behälterklasse Eine Behälterklasse besteht aus Objekten, die Mengen (und nicht Elemente von Mengen) sind.

Bisher waren bei Klassen nur Eigenschaften angegeben, die für jedes Objekt der Klasse gelten. Wenn bei der Klasse der Bälle angegeben ist, daß sie kugelförmig sind, so hat auch jedes Objekt der Klasse die Kugelform. Dies ist das grundsätzliche Prinzip. Wenn wir aber die Anzahl von Objekten einer Klasse zählen wollen, so ist diese Anzahl eine Eigenschaft der Klasse. Wir könnten natürlich zusätzlich zu der einer normalen Klasse **K** eine Behälterklasse **AlleK** einführen, deren einziges Objekt alle Objekte von **K** enthält. Einfacher ist die Einführung von *Klasseneigenschaften*. Eine Klasseneigenschaft ist eine Eigenschaft der Klasse. Sie gilt für die Menge ihrer Objekte, nicht für jedes einzelne Objekt der Klasse. Alle Objekte der Klasse können sich die Klasseneigenschaft ansehen und verändern.

Beispiel 2.1: Klasseneigenschaft Ein Flugbuchungssystem hat eine Klasse **Buchung**, in der der Kunde, sein Abflugort, sein Zielflugort und die Route dahin, die Fluglinie etc. zusammengeführt werden. Die Buchungsstelle möchte wissen, wieviele Buchungen sie vorgenommen hat. Dafür soll nicht eigens die Behälterklasse **AlleBuchungen** eingeführt werden, denn wir wollen ja nur die Anzahl der Objekte der Klasse **Buchung** wissen. Also wird stattdessen die Klasseneigenschaft *buchungsAnzahl* in **Buchung** eingeführt.

Wird eine Buchung durchgeführt, also ein neues Objekt der Klasse **Buchung** erzeugt (Instanziierung), dann fragt das neue Objekt den aktuellen Wert ab und erhöht ihn um 1.

2.2 Modellierung

Bei der objektorientierten Modellierung überlegen wir zuerst, was die Objekte sein sollen. Wie genau müssen wir modellieren? Wie werden die Objekte klassifiziert? Wer ist wofür zuständig? Wer kooperiert mit wem? Diagramme für diesen Entwurfsschritt sind *Klassenkarten* bzw. *Klasse-Zuständigkeit-Kooperation-Karten*. Es ist ein grober Entwurf, der im weiteren verfeinert wird.

Welche Merkmale besitzt eine Klasse? Welche Botschaften können ihre Objekte behandeln? Welche Beziehungen bestehen zwischen Objekten einer Klasse? Diagramme für diesen Entwurfsschritt heißen *Klassendiagramme* oder *Objektmodelle*.

Welche Botschaften werden versendet und behandelt? Wer kooperiert mit wem? Im *Kollaborationsdiagramm* bzw. *funktionalen Modell* werden die Kooperationen mit ihren Botschaften notiert.

Wie verändern Botschaften den Zustand von Objekten? Dies wird im *Zustandsdiagramm* bzw. dem *dynamischen Modell* aufgeschrieben.

Hier werden die einzelnen Entwurfsschritte anhand des Beispiels von [Goos, 1996] (S. 151ff) in der *Unified Modeling Language* von Grady Booch, James Rumbaugh und Ivar Jacobson [Oestereich, 1997] illustriert. Das Beispiel handelt von einem Studenten, der bei einem Rechnerhändler per Brief einen Rechner bestellt. Dieser wird ihm von der Post als Paket zugestellt.

Die *Klassenkarte* stellt die beteiligten Klassen (hier: Studenten, Rechnerhändler und Zusteller) mit ihren Zuständigkeiten und Kooperationen dar (s. Abbildung 6). Das Diagramm von Abbildung 6 ist eigentlich nicht zutreffend: bezüglich des Rechnerkaufs ist der Student einfach ein Kunde. Es geht uns nicht darum, ein bestimmtes Individuum zu modellieren, sondern eine Rolle, die es in einem Zusammenhang spielt. Daher nennen wir die Klasse mit den Zuständigkeiten bestellen, Paket annehmen besser **Kunde**. Daß der Brief und das Paket selbst fehlen, ist begründet: sie haben weder Zuständigkeiten, noch kooperieren sie mit einem der Beteiligten.

Nachdem wir nun wissen, welche Klassen es gibt, müssen wir ihre Botschaften, Handlungen und Merkmale überlegen. Wir notieren dies im *Klassendiagramm* , Abbildung 7. Die Abbildung zeigt, wer an wen Botschaften schickt und welche Eigenschaften er haben soll und welche Methoden er beherrscht. In der Klassenkarte hatten wir nur den Kunden (Studenten), den Rechnerhändler und den Zusteller. Alle diese müssen mit Adressen umgehen. Damit wir nicht bei jeder Klasse Methoden zum Verarbeiten von Adressen angeben müssen, legen wir die neue Klasse **Anschrift** fest, die Zeichenfolgen (**String**) als Absender und Empfänger interpretieren kann. Und weil wir sie nun schon als eigene Klasse haben, verallgemeinern wir sie so, daß auch die Beschriftung des Pakets und des Bestellbriefes die (hier nicht aufgeführten) Methoden des Prüfens einer Adresse und eines Namens nutzen kann. Wir sehen also, daß Klassen eigentlich durch Methoden eingeführt werden: alles, was eine bestimmte Tätigkeit ausführen kann, ist eine Klasse. Dies entspricht dem Klassenbegriff und ist obendrein praktisch:

- Verwendung *einer* Modellierung eines Vorgangs an vielen Stellen statt mehrfacher Modellierung desselben (Stichwort: Wiederverwendung von Programmteilen);

Student	
bestellen	Rechnerhändler
Paket annehmen	Zusteller

Rechnerhändler	
Bestellung annehmen	Student
Paket absenden	Zusteller

Zusteller	
Paket annehmen	Rechnerhändler
Paket abgeben	Student

Abbildung 6: Klassenkarte

- bei Veränderung des Vorgangs braucht die Modellierung (das Programm) nur an *einer* Stelle geändert zu werden und wirkt sich doch auf viele Stellen einheitlich aus. Denken Sie daran, wieviele Klassen bei der Einführung der neuen Postleitzahlen hätten geändert werden müssen, wenn wir nicht die Klasse **Anschrift** eingeführt hätten!

Die Verbindungen zwischen den Klassen sind als Assoziationen angegeben, allerdings noch nicht näher beschrieben.

Wir überlegen, was die Pfeile des Klassendiagramms eigentlich genau bedeuten sollen. Insbesondere beachten wir, welche unterschiedlichen Benutzer später mit der Software arbeiten werden und wie sich das auf die Modellierung auswirkt. Dies führt uns zum nächsten Diagramm. Das *Kollaborationsdiagramm* beschriftet die Kanten des Klassendiagramms. Es beschreibt genauer, was zwischen den Objekten kooperierender Klassen (**Kunde**, **Zusteller**, **Rechnerhändler**) ausgetauscht werden kann (Abbildung 8).

Wir haben bisher nur betrachtet, was ausgetauscht und behandelt wird. Nun wollen wir die Kollaborationen als Ereignisse betrachten, die den Zustand der Welt verändern. Für jede Klasse zeichnen wir einen *endlichen Automaten*. Ein endlicher Automat hat eine Menge von Zuständen, darunter einen ausgezeichneten Anfangszustand sowie Endzustände, eine Menge von Kanten, die Zustandsübergänge bezeichnen. Die Kanten sind also gerichtet, sie führen von einem Zustand in einen anderen. Handlungen sind typische Beispiele für Zustandsübergänge. Wenn in einem Zustand eine bestimmte Handlung ausgeführt wird oder eine bestimmte Eingabe empfangen wird, dann gelangt man in den Folgezustand. Dies muß nicht so einfach hintereinander geschehen wie es das Zustandsdiagramm zu unserem Beispiel zeigt (Abbildung 9). Eine Handlung oder Eingabe kann auch von einem Zustand in denselben überführen. Vom selben Zustand aus können durch unterschiedliche Handlungen (oder Eingaben) unterschiedliche Folgezustände erreicht werden.

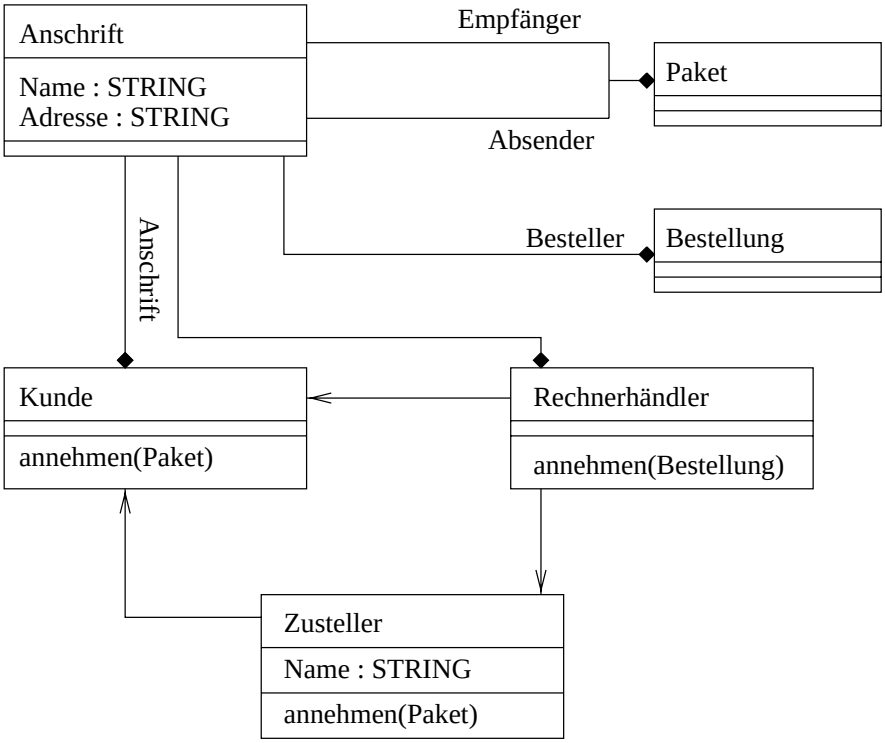


Abbildung 7: Klassendiagramm

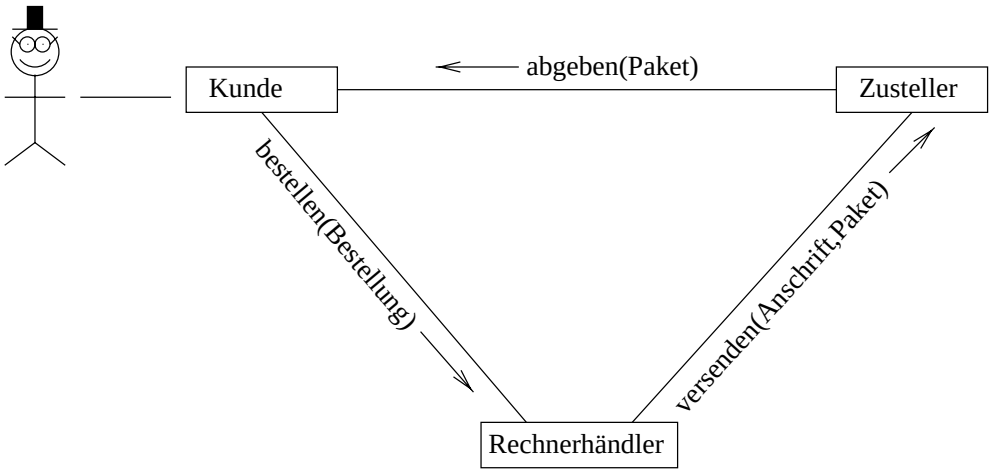


Abbildung 8: Kollaborationsdiagramm

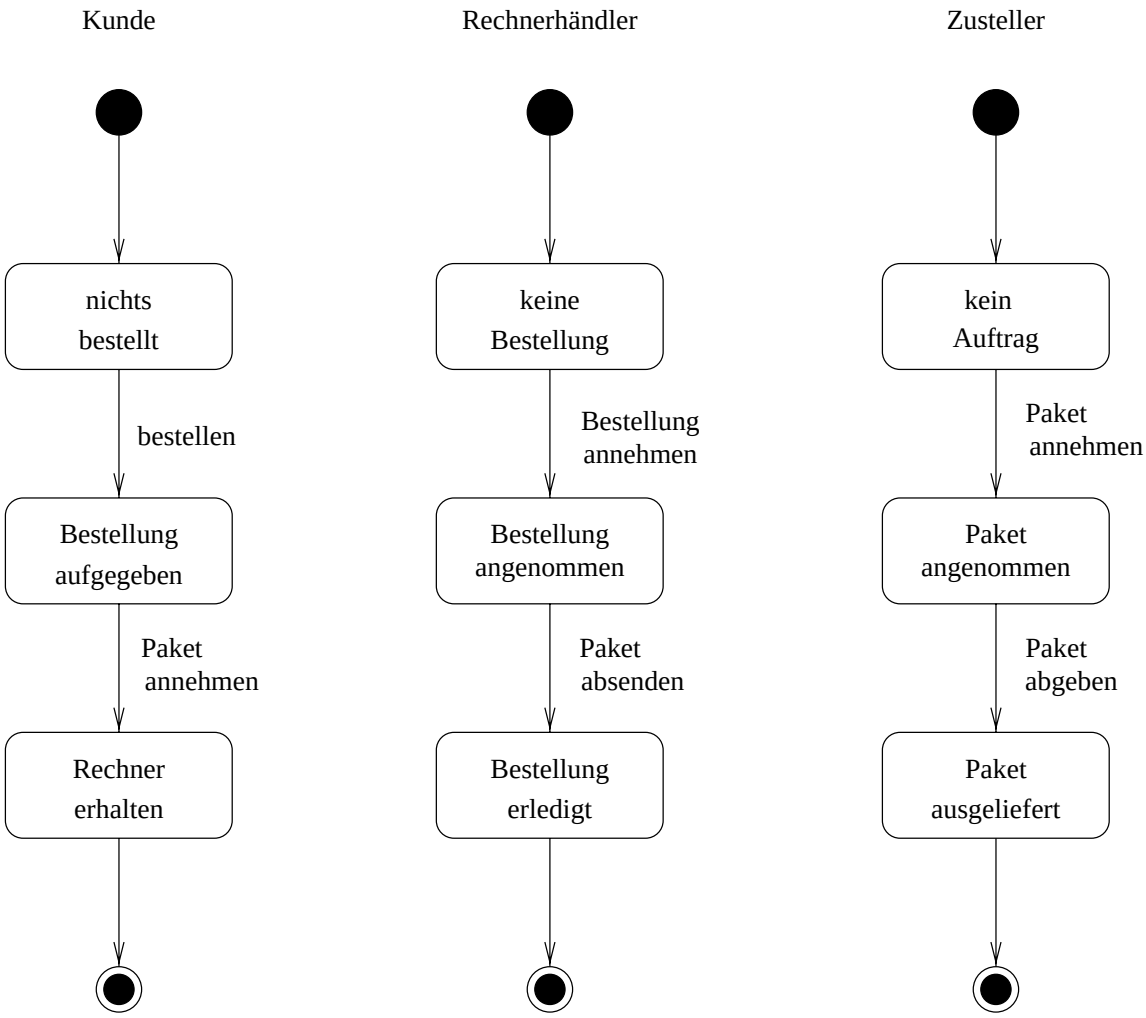


Abbildung 9: Zustandsdiagramm

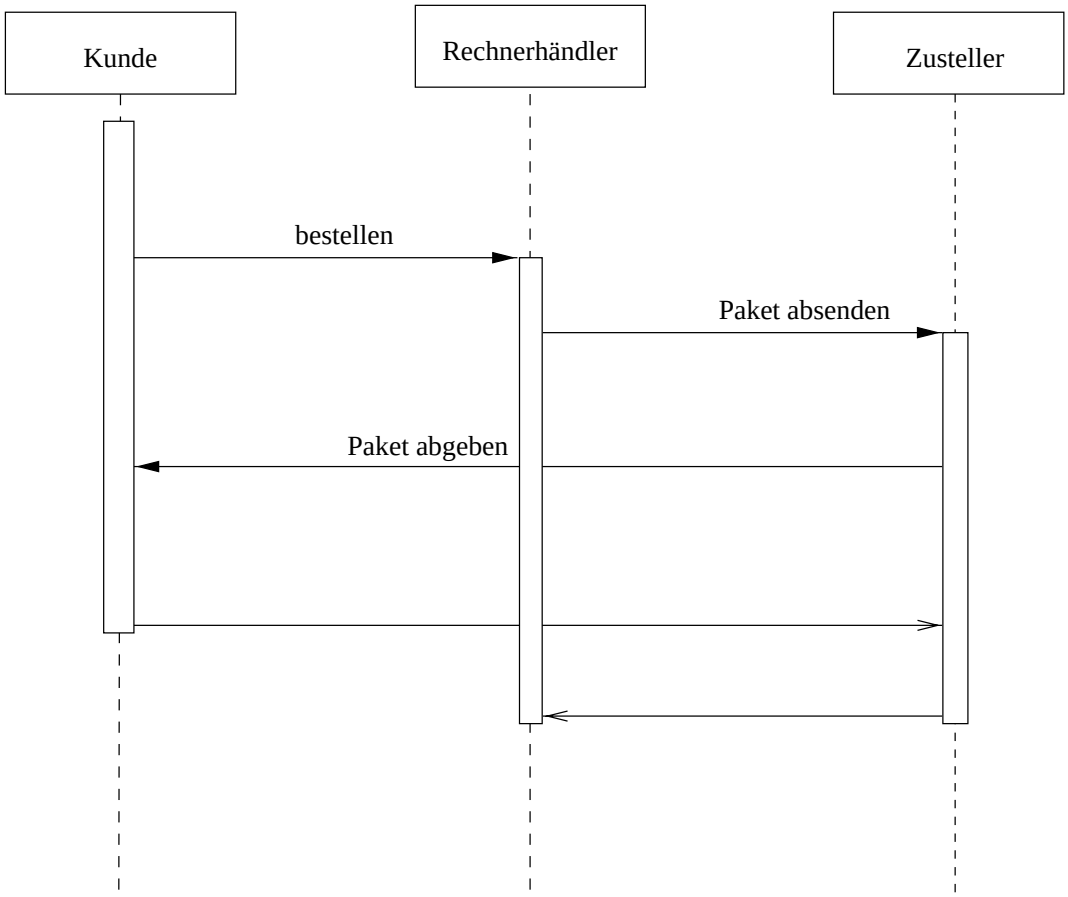


Abbildung 10: Sequenzdiagramm

Mit dem Zustandsdiagramm ist noch nichts über die zeitliche Abfolge von Handlungen zwischen Klassen ausgesagt, sondern nur für jede einzelne Klasse eine Abfolge angegeben. Das *Sequenzdiagramm* zeigt die Abfolge von Handlungen im Zusammenhang für alle beteiligten Klassen (s. Abbildung 10). Dabei sehen wir, daß der Kunde den Rechnerhändler aktiviert, der wiederum den Zusteller aktiviert. Der Kunde wartet einfach bis das Paket ankommt. Man nennt dies *Aufruf mit Warten auf Erledigung*. Erkundigt er sich, wo der Rechner denn bleibt, so nennt man dies *Aufruf mit späterer Anfrage*. Außerdem gibt es den *Aufruf mit aktiver Rüeckmeldung*. Beispielsweise kann der Kunde dem Zusteller den Empfang bescheinigen. Die Rückmeldung muß also nicht an denselben gehen, dem der Auftrag erteilt wurde. In der Abbildung 10 werden die Aufträge mit ausgefüllten Pfeilspitzen, die Rückmeldungen mit einfachen Pfeilspitzen gekennzeichnet. Der Kunde meldet an den Zusteller zurück, der wiederum an den Rechnerhändler zurückmeldet.

2.3 Was wissen Sie jetzt?

Sie sollten jetzt wissen, was eine Klasse und was ein Objekt ist. Sie sollten die Vererbungshierarchie als vertikale Beziehung und Assoziationen als horizontale Beziehung kennen.

Wie diese Begriffe eingesetzt werden, um einen Sachbereich zu modellieren, haben Sie anhand von Modellen, die unterschiedliche Aspekte betonen, sowie deren grafischer Darstellung gesehen. Gehen Sie alle Definitionen noch einmal durch. Besprechen Sie Behälterklassen und Klasseneigenschaften mit Ihren KommilitonInnen. Nehmen Sie sich irgendeinen Ausschnitt Ihres Alltags und versuchen Sie, ihn objektorientiert zu modellieren. Zeichnen Sie mindestens eine Klassenkarte und ein Klassendiagramm dazu!

3 Einführung in JAVA - erste Schritte

JAVA ist eine Programmiersprache mit den folgenden Eigenschaften:

objektorientiert , das bedeutet hier[2]:

- beim Programmieren werden insbesondere Daten und diese Daten verändernde Methoden beachtet;
- außer primitiven Datentyen sind alle Dinge in JAVA Klassen und Objekte;
- eine Klasse ist die kleinste ablauffähige Einheit in JAVA;
- alle JAVA-Programme sind Klassen.

plattformunabhängig , das bedeutet, die Sprache kann auf allen Rechnern und Betriebssystemen ausgeführt werden, weil sie in eine virtuelle Maschine übersetzt, für die die Plattformen eine Schnittstelle bereitstellen;

klares Typ-Konzept , s. z.B. Abschnitt 3.2;

verteilt , das bedeutet hier die integrierte Netzwerkunterstützung und das Laden und Ausführen von Programmen über das Internet (s. Abschnitt 9);

nebenläufig , das heißt gleichzeitige Bearbeitung mehrerer Aufgaben, wobei Prioritäten gesetzt werden können (s. Abschnitt 8).

Plattformunabängigkeit und verteiltes Arbeiten können durch das Schaubild 11 illustriert werden.

Ein bestimmter Rechner mit einer Benutzerschnittstelle erlaubt es Ihnen, einen Editor aufzurufen, in dem Sie ein JAVA-Programm schreiben [3]. Sie rufen nun den JAVA-Übersetzer auf mit dem Namen Ihres Programms und der Übersetzer zeigt Ihnen an, wo Sie die Syntax von JAVA nicht befolgt haben (s. Abschnitt 3.1).

`javac` *Programmbezeichner*`.java`

Sie ändern Ihr Programm und dann rufen Sie den Übersetzer wieder auf. Dies geht solange, bis das Programm syntaktisch korrekt ist. Dann wird es in den Bytecode von JAVAs virtueller Maschine (und eben nicht in den Maschinencode Ihres Rechners) übersetzt. Dieser Code kann auch von Ihrem Rechner verstanden werden.

Sie rufen Ihr Programm auf mit

[2] Der Gedanke der Objektorientierung wurde in den 70er Jahren am Xerox Palo Alto Research Center entwickelt. Adele Goldberg entwickelte mit einigen Kollegen die Programmiersprache *Smalltalk*, die dieser Form der Programmierung Popularität verschaffte. Heute arbeitet sie an Werkzeugen, die Teamarbeit unterstützen. Sie wurde von der Universität Dortmund mit der Ehrendoktorwürde geehrt.

[3] Genauere Hinweise zum Übersetzen und Aufrufen von Programmen oder Applets finden Sie im Anhang.

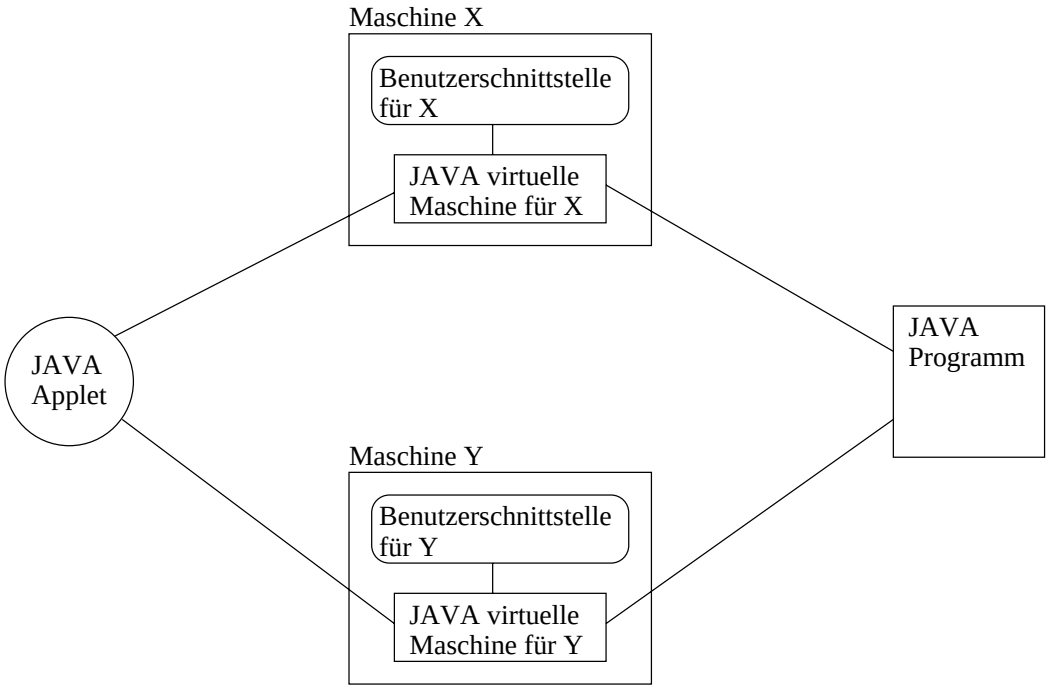


Abbildung 11: Architektur

`java` *Programmbezeichner*

und erhalten ein Ergebnis. Dies kann noch von dem abweichen, was Sie eigentlich mit dem Programm wollten. Man spricht dann von logischen Fehlern. So sieht der Prozeß aus, wenn Sie das Betriebssystem bzw. eine Fensterumgebung für das Übersetzen und Ausführen von JAVA-Programmen verwenden. Sie können aber auch Anwendungen, die andere programmiert haben, bei sich ablaufen lassen. Diese *kleinen Anwendungen*, genannt *applets*, bleiben als Programm (Quellcode) auf dem Rechner, auf dem sie bereitgestellt werden. Ein applet wird nicht auf den eigenen Rechner kopiert, sondern nur der Bytecode für JAVAs virtuelle Maschine wird von dem Entwicklungsrechner zu Ihrem Rechner geschickt. Der Bytecode verbleibt dort. Als Benutzerschnittstelle verwenden Sie dann Ihren *Browser*[4].

Einige nützliche Informationen kann man im World Wide Web (WWW) unter

`http://java.sun.com/docs/index.html` oder

`http://java.sun.com/products/jdk/1.1/docs/index.html`

finden. Auch die angeführten JAVA-Bücher sind im WWW präsent. Literatur findet man über den angegeben Bereich `java.sun.com/books` oder auch bei den Verlagen. Die meisten Dokumente brauchen Sie nicht aus den USA zu holen. Wir haben z.B. die Sprachdokumentation lokal im WWW abgelegt unter

`file:/app/unido-inf/sun4_55/jdk/1.2b4/docs/index.html`

[4]Ein *Browser* ist ein Programm, das Dokumente im Format `html` anzeigen und Klicks auf markierte Zeichenketten auswerten kann. Ist die markierte Zeichenkette eine Verbindung zu einem anderen Dokument (einer anderen Seite), so wird der Klick als Sprungbefehl zu dieser Seite behandelt.

3.1 Syntax für Klassen

Jede Sprache hat eine Syntax. Sie gibt an, in welcher Reihenfolge was für Zeichenfolgen vorkommen dürfen. Dabei wird zweistufig vorgegangen:

Lexikalische Ebene: Zunächst wird definiert, wie aus einzelnen Zeichen Wörter zusammengesetzt werden dürfen.

Syntaktische Ebene: Dann wird definiert, wie aus den Wörtern Sätze zusammengesetzt werden dürfen.

Die Syntax wird beschrieben durch eine *Grammatik*. Die Grammatik kann für die Analyse oder Generierung von Zeichenfolgen (Wörtern oder Sätzen) verwendet werden. Bei der Analyse werden gegebene Zeichenfolgen, bestehend aus *terminalen Symbolen*, auf ihre Wohlgeformtheit bezüglich der Grammatik hin untersucht. Bei der Generierung werden Folgen von terminalen Symbolen erzeugt. Auf lexikalischer Ebene sind die terminalen Symbole die Zeichen, auf syntaktischer Ebene sind es die Wörter.

Definition 3.1: Grammatik Eine Grammatik besteht aus

- einem Startsymbol S,
- einem Alphabet A, das aus einer Menge von terminalen Symbolen T und einer Menge von nicht-terminalen Symbolen N besteht,
- einer Menge von Produktionen.

Eine *Produktion* besteht aus einer linken und einer rechten Seite. Bei kontextfreien Grammatiken steht auf der linken Seite nur ein nicht-terminales Symbol. Auf der rechten Seite stehen mehrere Symbole aus A [5]. Bei der Generierung wird das Symbol auf der linken Seite durch die Folge von Symbolen auf der rechten Seite ersetzt. Beginnend mit dem Startsymbol werden nun die Produktionen angewandt und die nicht-terminalen Symbole durch andere Symbole ersetzt. Terminale Symbole werden natürlich nicht ersetzt.

Ein einfaches Beispiel:

S : $Kopf$ $Rumpf$
$Kopf$: `Uta`
$Rumpf$: `spielt Ball`
$Rumpf$: `tritt den Ball`

Hier sind $S, Kopf, Rumpf$ nicht-terminale Symbole, `Uta`, `spielt`, `tritt`, `Ball` und `den` sind terminale Symbole. Wir generieren mit der Grammatik:

S wird zu $Kopf$ $Rumpf$,
$Kopf$ $Rumpf$ wird zu `Uta` $Rumpf$,
`Uta` $Rumpf$ wird zu `Uta spielt Ball`
oder zu `Uta tritt den Ball`.

[5] Bei *kontextsensitiven* Grammatiken wird weniger als bei kontextfreien Grammatiken gefordert. Die linke Seite muß nur kürzer sein als die rechte. Damit kann man mehr und komplexere Sprachen beschreiben. Bei *regulären* Grammatiken wird mehr als bei kontextfreien Grammatiken gefordert. Die linke Seite ist ein nicht-terminales Symbol und die rechte Seite beginnt mit genau einem nicht-terminalen Symbol, dem terminale Symbole folgen dürfen. Damit kann man weniger und einfachere Sprachen beschreiben.

Mit der Grammatik kann man also zwei Sätze generieren: `Uta spielt Ball` und `Uta tritt den Ball`.

Bei der Analyse prüfen wir, ob ein Satz wohlgeformt ist. Haben wir den Satz `Uta tritt den Ball`, ersetzen wir Symbole der rechten Seite einer Produktion durch ihre linke Seite.

$$\begin{array}{c} Kopf\ \texttt{tritt den Ball} \\ Kopf\ Rumpf \\ S \end{array}$$

Nur, wenn wir durch umgekehrte Ersetzungen zu S gelangen können, ist der Satz wohlgeformt. Diese Prüfung nimmt der Übersetzer vor. Der Satz `Uta spielt mit dem Ball` ist bei der gegebenen Grammatik nicht wohlgeformt, weil `spielt mit dem Ball` auf keiner rechten Seite einer Produktion vorkommt und deshalb $Kopf$ `spielt mit dem Ball` nicht zu $Kopf\ Rumpf$ werden kann, was zu S führen würde.

Oft wird abkürzend für die Produktionen

$S: A\ B\ C$
$S: \quad B\ C$
$S: D\ E\ F$

einfach geschrieben:

$S: A_{opt}\ B\ C$
$\quad D \quad E\ F$

Hiermit erhält der Zeilenumbruch eine Bedeutung, nämlich die der alternativen Ersetzung. Wenn nun aber die rechte Seite zu lang wird, so daß ein Zeilenumbruch *ohne* diese Bedeutung erfolgen soll, so wird rechtsbündig eingerückt.

Da JAVA vor allem Klassen behandelt, soll hier ein Ausschnitt der Grammatik für Klassen angegeben werden (Tabelle 1).

Das wichtige an einer Programmiersprache ist nicht die Anordnung von Schlüsselwörtern und nicht-terminalen Symbolen, sondern was sie bedeuten. Die Syntax einer Programmiersprache ist gerade so gestaltet, daß alle Folgen von terminalen Symbolen, die einem nicht-terminalen Symbol entsprechen, eine gemeinsame Bedeutung haben. Nicht-terminale Symbole und Schlüsselwörter haben eine eindeutige Bedeutung. Die *Semantik* einer Programmiersprache führt die Bedeutung von Programmen in der Sprache auf die Bedeutung der Schlüsselwörter und nicht-terminalen Symbole, die Programmteilen entsprechen, zurück bzw. setzt aus der Bedeutung der einzelnen Teile die Bedeutung des Programms zusammen. Dabei ist die Bedeutung eines Programms stets *operational*, d.h. sie entspricht Operationen, die auch tatsächlich ausgeführt werden. Ich beschreibe die Bedeutung hier umgangssprachlich und illustriere sie durch JAVA-Programme.

Der *Modifikator* ($Modifier$) kann *public*, *abstract* oder *final* sein[6]. Abstrakte Klassen werden in Abschnitt 3.7 behandelt. *final* bedeutet, daß die Klasse keine Unterklassen besitzen darf. *public* bedeutet, daß die Klasse von überall aus verwendet werden kann (s. Abschnitt 3.8).

[6]Leider ist es mir nicht gelungen, die Formatierung der Grammatik mit denselben Buchstabenarten zu realisieren, die im Text verwendet werden. Dort wird **Klasse**, **Methode** und *Schlüsselwort* so gesetzt wie in diesem Satz. Die nicht-terminalen Symbole und Variablen sehen *so* aus.

ClassDeclaration :
    $Modifiers_{opt}$ `class` *Identifier* $Super_{opt}$
        $Interfaces_{opt}$ *ClassBody*
Super :
    `extends` *ClassType*
Interfaces :
    `implements` *InterfaceTypeList*
InterfaceTypeList :
    InterfaceType
    InterfaceTypeList , *InterfaceType*
ClassBody :
    { $ClassBodyDeclarations_{opt}$ }
ClassBodyDeclarations :
    ClassBodyDeclaration
    ClassBodyDeclarations ClassBodyDeclaration
ClassBodyDeclaration :
    ClassMemberDeclaration
    StaticInitializer
    ConstructorDeclaration
ClassMemberDeclaration :
    FieldDeclaration
    MethodDeclaration
StaticInitializer :
    `static` *Block*

Tabelle 1: Klassendeklaration

Das Schlüsselwort *class* zeigt an, daß es sich um ein Klasse handelt. Der Name (*Identifier*) der Klasse wird eingeführt, damit man sich auf die Klasse beziehen kann. Der Name einer Klasse beginnt stets mit einem Großbuchstaben.

Durch *extends ClassType* wird die Oberklasse angegeben, von der diese Klasse erbt. Schnittstellen (*Interfaces*) lernen wir später kennen (Abschnitt 3.7).

Nach diesen Präliminarien folgt das Wichtigste einer Klasse – der Klassenrumpf (*ClassBody*). Dies ist ein Block. Ein Block beginnt mit geschweifter Klammer und endet mit geschweifter Klammer. Meistens werden hier Methoden definiert. Dazu werden zunächst die Variablen deklariert und dann die Methode selbst.

Beispiel 3.1: In der Einleitung war von Utas blauem Ball die Rede. Daß er Uta gehört, soll bei Uta vermerkt werden. Allerdings können wir eine Klasse **Besitz** definieren und Bälle sind eine Unterklasse davon. Außerdem haben Bälle immer eine Farbe. Ihre Methode besteht darin, einen Tritt in eine Ortsveränderung umzusetzen. Wir notieren den Tritt als zwei Zahlen im Koordinatensystem, die angeben, um wieviel der Ball sich auf der x- und um wieviel er sich auf der y-Achse verschieben soll.

Die Methoden erläutern wir später (Abschnitt 3.4). Hier erst einmal der Überblick über die Klasse **Ball**. Wir zeigen, daß die Klasse **Ball** gemäß der Grammatik (Tabelle 1) wohlgeformt ist.

```
class Ball extends Besitz                          //class Identifier extends ClassType
 {                                                                    // ClassBody Beginn
       float x,y;                                                      // FieldDeclaration
       String farbe;                                                   // FieldDeclaration

       public Ball (String _name,String _farbe, float _x, float _y)  {   // MethodDeclaration
             super (_name);
             farbe = _farbe;
             x = _x;
             y = _y;  }

       public void rolle (float dx,float dy) {                         // MethodDeclaration
             x += dx; y += dy;
       }

 }                                                                     // ClassBody Ende
```

Der Übersetzer von JAVA fertigt zu der Klasse **Ball** einen Syntaxbaum an. Der Syntaxbaum zeigt, welche Ersetzungen von den terminalen Zeichen des Programms zu den nicht-terminalen Zeichen der Klassendeklaration bis hin zu *ClassDeclaration* führen. Ein (unvollständiger) Syntaxbaum ist in Abbildung 12 dargestellt.

Wir sehen in diesem Beispiel die JAVA-Darstellung für

- die Vererbungshierarchie durch *extends*,
- ihre Ausnutzung durch *super*,
- eine Eigenschaft, die jedes Objekt der Klasse hat durch $farbe$,
- eine Methode, wie Objekte einer Klasse erzeugt werden (Konstruktor) durch **Ball**,
- eine Methode, die eine Botschaft von außen bearbeitet, durch **rolle**

Wie ein neues Objekt für eine Klasse angelegt wird, beschreibt die Methode *Konstruktor* (*Constructor*). Die Syntax zeigt Tabelle 2. Ein Konstruktor heißt stets wie die Klasse, zu der er ein Objekt anlegt. Der Konstruktor der Klasse **Ball** heißt also **Ball**. Er hat eine Liste von Argumenten in Klammern (*FormalParameterList*). Es folgt ein Block, der für ein neues Objekt bei der Erzeugung ausgeführt wird. Das erste darin ist der Aufruf des Konstruktors der Oberklasse durch $super(\_name)$. Da jedes Ding einen Namen hat, sorgt der Konstruktor von **Besitz** dafür, daß ein Name vergeben wird. Die Farbe, beschrieben durch irgendeine Kette von Buchstaben (**String**), und die Position, beschrieben durch x- und y-Koordinaten, sind hingegen Eigenschaften des Balls und nicht jedes Gegenstandes.

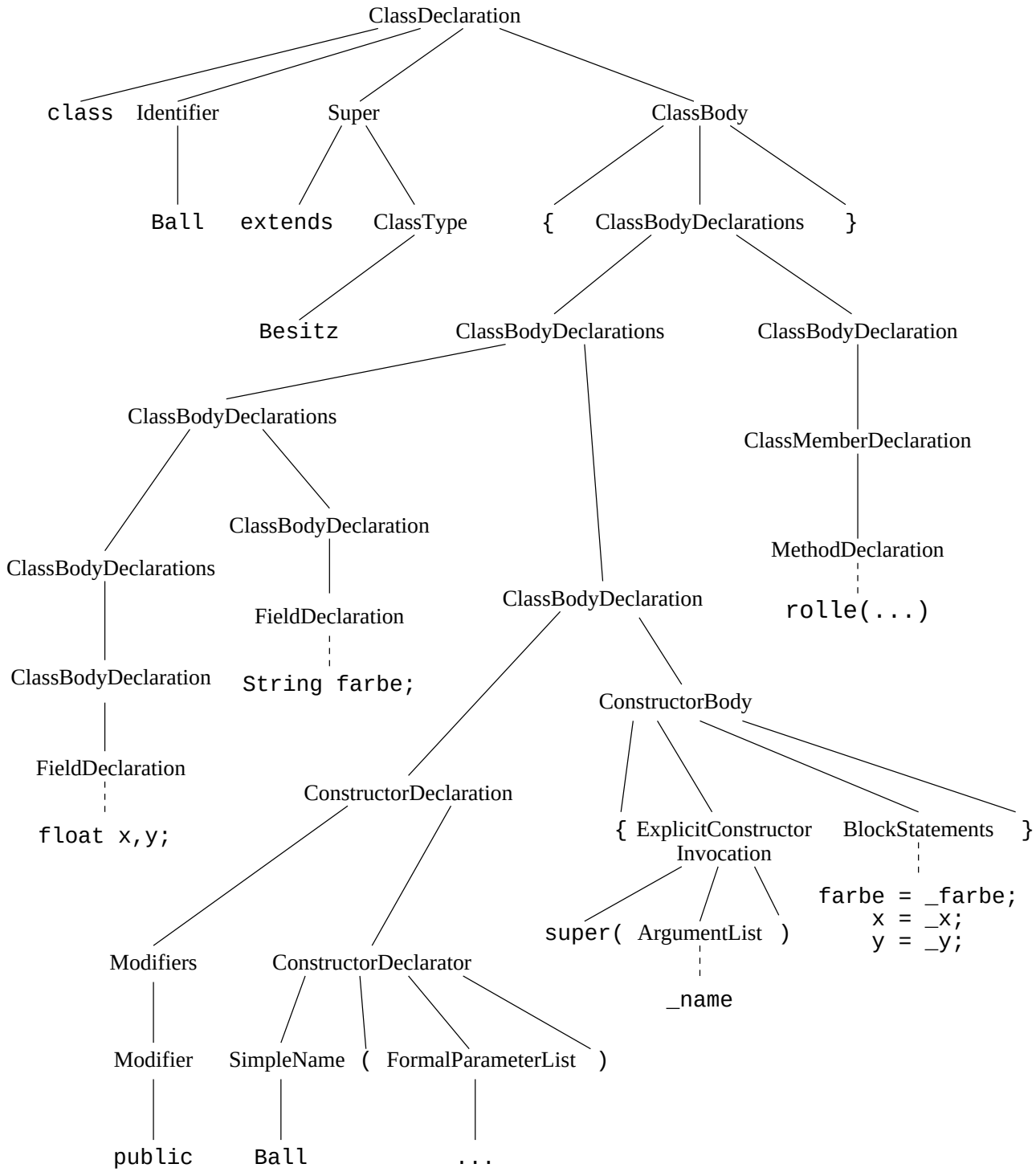


Abbildung 12: Unvollständiger Syntaxbaum für **Ball**

$ConstructorDeclaration$:
$Modifiers_{opt}$ $ConstructorDeclarator$ $Throws_{opt}$
$ConstructorBody$
$ConstructorDeclarator$:
$SimpleName$ ($FormalParameterList_{opt}$)
$ConstructorBody$:
{ $ExplicitConstructorInvocation_{opt}$ $BlockStatements_{opt}$ }
$ExplicitConstructorInvocation$:
`this` ($ArgumentList_{opt}$) ;
`super` ($ArgumentList_{opt}$) ;

Tabelle 2: Konstruktordeklaration

Die Werte der Merkmale werden dann angegeben, wenn ein neues Objekt tatsächlich erzeugt wird. Hier geht es erst einmal darum, zu deklarieren, was für die Erzeugung getan werden soll – noch nicht darum, es wirklich zu tun!

Wie sieht die Klasse **Besitz** aus?

```
class Besitz  {
        Mensch besitzer;
        String name;

        public Besitz (String _name) {
                name = _name;
        }
        public void gehoere (Mensch _besitzer)  {
                besitzer = _besitzer;  }
 }
```

Zwei Variablen werden deklariert, eine für den Besitzer und eine für den Namen (*Field-Declaration*). Der Konstruktor ist die Methode **Besitz**. Es ist keine Oberklasse angegeben. Damit wird **Besitz** direkt unter die allgemeinste Klasse von JAVA gehängt, die den leicht irreführenden Namen **Object** trägt. Alle Klassen in JAVA erben von dieser Klasse.

Wenn wir auch noch nicht die einzelnen Bestandteile einer Klassendeklaration kennen, so wissen wir jetzt immerhin, wie sie aufgebaut ist. Oben wurde gesagt, daß eine Klasse bei JAVA die kleinste ablauffähige Einheit ist. Bis jetzt wird aber noch gar nichts ausgeführt! Dazu braucht es eine Klasse mit einer Methode namens **main**. Diese Methode muß als *public* (s. Abschnitt 3.8) und *static* deklariert sein (s. Abschnitt 3.4.1). Sie muß als Parameter `String argv[]` haben (s. Abschnitt 3.2). Die Ausführung eines JAVA-Programms beginnt mit dieser Methode. Ein JAVA-Programm ist eine Menge von Klassen, von denen genau eine eine **main** Methode hat.

```
class BallBeispiel  {
        public static void main (String argv[])  {
```

```
        Ball ball;
        ball = new Ball ("ball1","blau",1,1);
    }
}
```

Die Klasse **BallBeispiel** hat eine Methode **main**, die nichts anderes tut, als ein Objekt der Klasse **Ball** zu erzeugen. Wie das geschieht, ist in der Konstruktormethode von **Ball** angegeben. Daß es geschieht, dafür sorgt

```
ball = new Ball ("ball1","blau",1,1);
```

Es gibt dann ein Objekt der Klasse **Ball** mit dem Namen `ball1`. Nach Beendigung des Programms gibt es dieses Objekt nicht mehr. Daß es vorhanden war, sieht man nicht, weil die Klassen nichts an den Benutzer melden. Das kommt noch!

3.1.1 Die Behälterklasse Vector

Sehr viele Klassen sind in JAVA bereits vordefiniert. Als Klasse, deren Objekte eine Sammlung von Teilen sind, gibt es die Felder (s. Abschnitt 3.6) und die Klasse **Vector**. Während Felder eine feste Größe haben, braucht man bei **Vector** nicht zu wissen, aus wievielen Teilen ein Objekt bestehen wird. Die Methode **addElement** fügt dem Objekt ein neues Teil hinzu. Die Methode **removeElement** entfernt ein Teil aus dem Objekt. Die Klasse **Vector** verhält sich also wie ein Behälter, in den beliebig viele Teile hineingeworfen und wieder herausgenommen werden können.

Beispiel 3.2: Jedes Objekt der Klasse **Besitz** "weiß", wem es gehört. Wenn nun ein Mensch alles, was er besitzt, versichern möchte, so bildet man die Behälterklasse **Hausrat**. Ein Objekt dieser Klasse besteht aus allen Objekten der Klasse **Besitz**, die diesem Menschen gehören. Wenn der Mensch etwas Neues bekommt, so wird diesem Objekt der Klasse **Besitz** mitgeteilt, daß es nun diesem Menschen gehört, und das Objekt der Klasse **Hausrat** wird aufgefordert, den neuen Besitz aufzunehmen.

```
import java.util.Vector;
class Hausrat extends Vector {

    public void aufnehmen (Besitz geschenk, Mensch mensch) {
        geschenk.gehoere (mensch);
        this.addElement(geschenk);
    }
}
```

Da JAVA Klassen von überall aus dem Internet laden kann, gibt es *Pakete*. Jede Klasse ist Teil eines Pakets. Das Paket ist der Ort, an dem die Klasse deklariert ist, also der Rechnerbereich, z.B. Ihr Rechnerbereich und dort das Verzeichnis, in dem Sie Ihre JAVA-Programme ablegen. In Unix (s. Anhang) wird Ihr Verzeichnis vielleicht so beschrieben `meyer/uebungen`. In JAVA heißt das Paket `meyer.uebungen`. International ist das eindeutig, wenn die Rechneradresse der Universität Dortmund noch vor das Verzeichnis mit

JAVA-Programmen gesetzt wird. Üblicherweise schreibt man das Land, die Institution, die Abteilung und dann das Verzeichnis, in dem die JAVA-Programme sind. So könnte der Student *Meyer* seine Programme in dem Verzeichnis ablegen: `de/unido/cs/ls8/meyer/uebungen/`. Die bereits von den Entwicklern der Sprache deklarierten Klassen sind in Paketen des Bereichs `java` gespeichert. Die Klasse **Vector** ist im Paket `java.util` definiert. Der Sprachkern von JAVA ist im Paket `java.lang`.

3.1.2 Was wissen Sie jetzt?

Sie haben die Schreibweise für Klassendeklarationen in Form einer Grammatik kennengelernt. Überzeugen Sie sich anhand der Grammatik, daß die Beispiele für JAVA-Klassen der JAVA-Syntax entsprechen! Versuchen Sie, anhand der Klassen-Syntax wohlgeformte Sätze der Sprache JAVA zu schreiben. Sie wissen allerdings bei den meisten Sätzen noch nicht, was sie bedeuten.

Aber einiges wissen Sie doch:

Sie haben gesehen, daß man mit *extends* die Oberklasse angeben kann und so die Vererbungshierarchie festlegt. Die Wurzel der Vererbungshierarchie ist die vordefinierte Klasse **Object**.

Eine Konstruktordeklaration legt fest, wie eine Instanz (ein Objekt) einer Klasse erzeugt wird. Es ist eine Methode mit dem Namen der Klasse. Mit *new* wird diese Methode aufgerufen und ein Objekt der Klasse erzeugt.

Enthält eine Datei mit Deklarationen von Klassen eine Klasse mit der Methode **main**, so ist es ein Programm. Die Datei heißt wie die Klasse, die die **main**-Methode enthält.

Eine Klassendeklaration legt fest, wie Objekte der Klasse aussehen. Sie kann somit zwischen solchen Objekten unterscheiden, die Instanzen von ihr sind, und solchen, die nicht Instanzen von ihr sind.

3.2 Variablen und Typen

> "The name of the song is called 'Haddock's Eyes'."
> "Oh, that's the name of the song, is it?" Alice said, trying to feel interested.
> "No, you don't understand", the Knight said, looking a little vexed. "That's what the name is called. The name really is 'The Aged Aged Man'."
> "Then I ought to have said ' That's what the song is called'?" Alice corrected herself.
> "No, you oughtn't: that's quite another thing! The song is called 'Ways and Means': but that's only what it's called, you know!"
> "Well, what is the song , then?" said Alice who was by this time comletely bewildered.
> "I was coming to that", the Knight said. "The song really is 'A-sitting On A Gate': and the tune's my own invention."
> Lewis Carroll, aus: Through the Looking-Glass, chapter 8.

JAVA hat ein klares Typ-Konzept. Dabei ist ein *Typ* nichts anderes als eine Klasse. Das klare Konzept besteht darin, daß (fast) alles in JAVA einer Klasse zugeordnet ist: jedes Objekt, jede Variable, jede Konstante ist von einem Typ, d.h. gehört zu einer Klasse. Andersherum ausgedrückt: die Klasse gibt den Wertebereich der Variablen an.

Definition 3.2: Variable Eine Variable ist ein Tripel (Name, Adresse, Wert). Der Name identifiziert die Variable. Die Adresse ist der Ort im Speicher, wo die Variable steht. Der Inhalt dieses Speicherplatzes ist der Wert der Variablen.

FieldDeclaration :
    $Modifiers_{opt}$ *Type VariableDeclarators* ;
VariableDeclarators :
    VariableDeclarator
    VariableDeclarators , *VariableDeclarator*
VariableDeclarator :
    VariableDeclaratorId
    VariableDeclaratorId = *VariableInitializer*
VariableDeclaratorId :
    Identifier
    VariableDeclaratorId `[ ]`
VariableInitializer :
    Expression
    ArrayInitializer

Tabelle 3: Variablendeklaration

Definition 3.3: Konstante Eine Konstante ist eine Variable, deren Wert unveränderlich ist.

In JAVA wird der Name geschrieben, wenn der Wert gemeint ist.

3.2.1 Variablendeklaration

Eine Variablendeklaration legt drei wichtige Dinge fest:

- Was für Werte kann die Variable annehmen? Welchen Typ hat sie? Der Typ wird meist durch eine Klasse angegeben, deren Objekte mögliche Werte der Variablen sind.
- Wessen Eigenschaften beschreibt die Variable? Die Variablendeklaration findet in der Deklaration einer Klasse statt. Meist beschreibt die Variable eine Eigenschaft, die jedes Objekt der Klasse haben soll.
- Wie heißt die Variable? Wie kann sie von allen anderen Variablen unterschieden werden?

Intern wird dem Namen einer Variable eine Adresse zugeordnet. Bei der Deklaration einer Variablen wird soviel Speicherplatz reserviert, wie es der Typ der Variablen angibt: für einen einfachen Datentyp die erforderlich Bytes, für alle anderen ein Platz, an dem die Refernz auf ein Objekt gespeichert werden kann. Dieser Speicherplatz, dessen Inhalt veränderlich ist, wird an den Namen der Variablen gebunden. Im Programm wird jeder Variablen als Typ eine Klasse zugeordnet. Dies bedeutet, daß ihr Wert ein Objekt der betreffenden Klasse sein muß. Wie sieht diese Zuordnung in JAVA aus? Sie geschieht mithilfe von Deklarationen, die syntaktisch für das nicht-terminale Symbol *FieldDeclaration* eingesetzt werden können. Die Syntax zeigt Tabelle 3.

Variablen können wie Klassen modifiziert werden. Der Modifikator (*Modifier*) kann drei Aspekte betreffen: die Sichtbarkeit wird durch *private*, *public* oder *protected* angegeben (s. Abschnitt 3.8); ob die Variable ihren Wert nicht verändern darf oder doch wird

durch *final* oder das Fehlen des Modifikators *final* ausgedrückt; ob es sich um eine normale Eigenschaft von Objekten handelt oder um eine Klasseneigenschaft (s. Abschnitt 2.1) drückt das Fehlen oder Vorhandensein des Schlüsselwortes *static* aus. Eine Klasseneigenschaft gibt es nur einmal, egal wieviele Objekte einer Klasse es gibt. Eine Objekteigenschaft bekommt jedes Objekt der Klasse. Sobald ein neues Objekt einer Klasse erzeugt wird, wird für jede Variable ohne Modifikator *static*, die in dieser Klasse oder in einer Oberklasse deklariert ist, eine neue Variable als Eigenschaft dieses Objektes erzeugt. Deklarieren wir für Menschen die Variable *hausrat* und ein neues Objekt *Uta* der Klasse **Mensch** wird erzeugt, dann wird auch eine Variable *Uta.hausrat* angelegt. Innerhalb der Klassendeklaration schreiben wir einfach *hausrat*.

Der Name einer Variablen *VariableDeclaratorId* ist ein kleingeschriebenes Wort, das _ enthalten darf, aber nicht mit einer Zahl anfangen darf. Es kann auch ein Name gefolgt von eckigen Klammern sein (s. Abschnitt 3.6). Ein Name muß eindeutig sein. Dies mag man bei dem selbst geschriebenen Programm noch garantieren können. Wenn man aber Klassen verwendet, die andere geschrieben haben, so könnten dort dieselben Namen vorkommen, die man selbst gerade verwenden möchte. Deshalb ist ein Name eigentlich viel länger als man es meist sieht. Vorangestellt wird vom JAVA-Übersetzer das Paket, in dem der Name eingeführt wird.

Der Typ *Type* einer Variable ist eine Klasse, die entweder vordefiniert oder im Programm deklariert wird.

Einige Beispiele haben wir bereits gesehen, z.B.:

```
Mensch besitzer;
String name;
```

Mensch und **String** sind Klassen. Diese Klassen müssen dem System bekannt sein, damit es prüfen kann, ob der Wert der betreffenden Variablen ein Objekt der angegebenen Klasse sein kann. Einige Klassen sind vordefiniert in JAVA, so daß wir sie direkt zur Variablendeklaration verwenden können. **String** ist so eine Klasse. **Mensch** müssen wir selbst definieren, damit die Variable `besitzer` einen Wert bekommen kann, der ein Objekt dieser Klasse ist. Wenn wir **Mensch** definieren als die Klasse derjenigen Objekte, die einen Namen, ein Geschlecht und Hausrat haben, so muß auch `besitzer` einen Namen, ein Geschlecht und Hausrat haben.

Mit dem Gleichheitszeichen kann eine Variable einen Anfangswert bekommen (zweite Produktion für *VariableDeclarator*). Dieser Anfangswert kann einfach eine bereits bekannte Variable (und das bedeutet hier: ihr Wert) sein oder eine Berechnung, die einen Wert ergibt (s. Abschnitt 3.2.3).

3.2.2 Einfache Datentypen

Zahlen, Wahrheitswerte und Buchstaben sind keine Einzeldinge. Sie sind, egal wie oft wir sie verwenden, Unikate. Daher können sie keine Klassen sein. Sie sollen aber genau wie Klassen den Wertebereich von Variablen angeben, also Datentypen sein. In JAVA heißen sie einfache Datentypen (die "komplexen" Datentypen sind die Klassen). Einfache Datentypen werden in JAVA direkt umgesetzt. Sie sind als einzige keine Klassen und werden daher auch nicht mit Großbuchstaben beginnend geschrieben.

Typ	Inhalt	Standardwert	Größe
boolean	true, false	false	1 Bit
char	Unicode-Zeichen	u0000	16 Bit
byte	Integer mit Vorzeichen	0	8 Bit
short	Integer mit Vorzeichen	0	16 Bit
int	Integer mit Vorzeichen	0	32 Bit
long	Integer mit Vorzeichen	0	64 Bit
float	Fließkommazahl	0.0	32 Bit
double	Fließkommazahl	0.0	64 Bit

String ist kein einfacher Datentyp sondern eine Klasse. Allerdings kommt es so häufig vor, daß eine vereinfachte Schreibweise eingeführt wurde. Man darf ein Objekt der Klasse **String** einfach zwischen Anführungszeichen setzen. Der JAVA-Übersetzer erzeugt automatisch ein passendes Objekt.

3.2.3 Wertzuweisungen

Die Werte von Variablen sind veränderlich. Eine Variable erhält einen (neuen) Wert durch eine Wertzuweisung. Eine Wertzuweisung kann durch einen Zuweisungsausdruck oder durch Parameterübergabe erfolgen.

Wir schreiben in JAVA einen Zuweisungsausdruck mit dem Gleichheitszeichen, das hier besser "Gleichsetzungszeichen" hieße.

Es gibt in JAVA die folgenden Zuweisungen:

- Einfache Zuweisung:
  - $v = 5$;
    bedeutet, daß v den Wert 5 bekommt.
  - $s =$ "$Zahn$";
    bedeutet, daß s den Wert "Zahn" bekommt.
- Mehrfache Zuweisung:
  - $v = w = 5$;
    bedeutet, daß w den Wert 5 bekommt und dann v den Wert von w, also 5. Zuweisungen werden immer von rechts nach links durchgeführt.
  - $s = t =$ "$Zahn$";
    bedeutet, daß t als Wert "Zahn" bekommt und dann s den Wert von t, also "Zahn".

Denkt man an das Tripel *Name*, *Adresse*, *Wert*, das eine Variable ausmacht, so bedeutet eine Wertzuweisung, daß in dem Speicherplatz, der durch die Adresse angegeben wird, ein neuer Wert steht. Dazu verwendet JAVA zwei Möglichkeiten.

Definition 3.4: Referenzzuweisung Der Wert einer Variablen ist selbst wiederum eine Adresse, in der der eigentliche Wert steht. Soll eine Variable v den Wert einer anderen Variable w erhalten, so wird die Adresse, die bei w als Wert angegeben ist, kopiert und die

Kopie der Adresse als Wert von v eingetragen. Dies macht insbesondere Sinn, wenn der Wert umfangreich ist. JAVA verwendet die Referenzzuweisung, wenn Objekte oder Felder der Wert einer Variablen sind (und nicht ein einfacher Datentyp).

Beispiel 3.3: Referenzzuweisung Nehmen wir an, eine Variable hat den Namen v, als Adresse für ihren Wert $a175$ und unter der Adresse $a175$ steht noch nichts. v soll den Wert einer anderen Variable w bekommen. Die Variable w hat als Adresse für ihren Wert $a100$. Der Wert von w sei ein Objekt, z.B. Utas blauer Ball. Dies Objekt ist im Speicher unter der Adress $a200$ zu finden. Im Speicherplatz 100 steht also "$a200$". Nun soll v den Wert von w bekommen. Dazu wird die Adresse, die unter $a100$ zu finden ist, also $a200$, kopiert und die Kopie unter der Adresse $a175$ eingetragen. Die Beschreibung von Utas blauem Ball bleibt unverändert ab $a200$ stehen. Falls sich die Beschreibung von Utas blauem Ball ändert, so auch die Werte von v und w.

Name	Adresse	Wert
	vorher:	
v	$a175$	-
w	$a100$	$a200$
	nachher:	
v	$a175$	$a200$
w	$a100$	$a200$

Definition 3.5: Wertzuweisung direkt Der Wert einer Variablen ist direkt unter der der Variablen zugeordneten Adresse eingetragen. Die Variable w übergibt direkt ihren eigentlichen Wert. JAVA verwendet die direkte Wertzuweisung bei Variablen, deren Wert von einfachem Datentyp ist.

Soll wieder die Variable v den Wert der Variablen w erhalten, wobei diesmal v und w vom einfachen Typ `double` sind, dann sieht die Wertzuweisung so aus:

Name	Adresse	Wert
	vorher:	
v	$a175$	-
w	$a100$	0,324
	nachher:	
v	$a175$	0,324
w	$a100$	0,324

Falls sich der Wert von w ändert, so bleibt der von v unverändert. Eine weitere Form, wie Variablen einen Wert erhalten sehen wir in Abschnitt 3.4.

3.2.4 Was wissen Sie jetzt?

Sie haben einen Vorteil der Sprache JAVA kennengelernt, nämlich das Typ-Konzept, das darin besteht, daß jede Variable einen vorgegebenen Wertebereich hat. Dieser Wertebereich

wird durch eine Klasse angegeben, besteht also aus allen Objekten dieser Klasse, oder durch einen einfachen Datentyp. Ein einfacher Datentyp bezeichnet Unikate: eine Zahl gibt es nur einmal, egal wie oft sie verwendet wird.

Sie wissen, was eine Variable ist und daß sie meist ihre Werte mittels einer Referenzzuweisung verändert, nur bei einfachen Datentypen mittels einer direkten Wertzuweisung.

3.3 Operatoren

Für einfache Datentypen gibt es Operatoren. Alle anderen Aktionen müssen durch Methoden ausgeführt werden. Als elementare Operationen verwenden wir hier zur Illustration die Grundrechenarten und das Aneinanderhängen von Zeichen. Grundrechenarten sind für Objekte von einem Zahlentyp (alle einfachen Datentypen bis auf `boolan` und `char`) definiert. Das Aneinanderhängen von Zeichen oder Zeichenfolgen heißt *Konkatenation* und ist für Objekte vom Typ `char` und **String** definiert. Die Konkatenation wird durch das Zeichen + angegeben. Sie kann aber nicht mit der Addition verwechselt werden, weil für Objekte vom Typ `char` oder **String** die Addition nicht vorgesehen ist. Nehmen wir an, die Variablen v, w, x seien von einem Zahlentyp und die Variablen s, t, u vom Typ **String**.

- Infixoperatoren:
  - $v = 2 + 3;$
    bedeutet, daß v den Wert 5 bekommt. Analog sind die anderen Grundrechenarten in der Infixschreibweise verwendbar ($-$, $*$, /). Bei Integer-Zahlen gibt es anstelle der normalen Division die Division mit Rest (/ liefert das ganzzahlige Ergebnis und % den Rest).
  - $v = v + 3;$
    bedeutet, daß der Wert von v um 3 erhöht wird. Eine abkürzende Schreibweise ist
    $v+ = 3;$
    Analog können auch die anderen Grundrechenarten abgekürzt geschrieben werden.
  - $s =$ "$Zahn$";
    $t =$ "rad";
    $u = s + t;$
    bedeutet, daß der Wert von t, "rad", an den Wert von s, "Zahn", gehängt wird. Das Ergebnis, "Zahn" "rad", wird der Variablen u als Wert zugewiesen.
- Präfixoperatoren:
  - $++v$ bzw. $--v$
    bedeutet, daß v um 1 erhöht bzw. vermindert wird.
- Postfixoperatoren:
  - $v++$ bzw. $v--$
    bedeutet, daß v um 1 erhöht bzw. vermindert wird. Allerdings wird im Gegensatz zu Präfixoperatoren als Wert noch der ursprüngliche Wert von v abgegeben. Sei v z.B. 4.

- $w = v++;$
  Jetzt hat w den Wert 4, v den Wert 5. Folgt nun
  $x = v;$
  so hat x den Wert 5.

Variablen vom Typ `boolean` werden meist eingesetzt, um Bedingungen zu formulieren. Eine Bedingung ist entweder wahr oder falsch, ihr Ergebnis ist folglich vom Typ `boolean`. Bedingungen werden nur für Zahlen angegeben. Es gibt aber auch logische Operatoren, die verschiedene Variablen vom Typ `boolean` verknüpfen und deren Ergebnis wiederum vom Typ `boolean` ist.

- Bedingungen:
  - == bedeutet die Gleichheit,
    sei z.B. $v = 2 + 3$ und $w = 5$, so ist bei
    $b = (v == w);$
    der Wert der Variable b vom Typ `boolean` `true`, also wahr.
  - != bedeutet die Ungleichheit,
    so ist z.B.
    $b = (v! = w);$
    der Wert von b nun `false`, also unwahr, wenn v und w den Wert 5 haben.
  - $>$ und $<$ bedeuten größer und kleiner.
    Sei b vom Typ `boolean` und *alter* vom Typ `int`:
    $b = (alter > 18);$
    b hat den Wert `true`, wenn der Wert von *alter* größer als 18 ist. Ist *alter* genau 18, so ist b `false` – natürlich ist b auch `false`, wenn *alter* kleiner als 18 ist.
  - $>=$ und $<=$ bedeuten größer oder gleich bzw. kleiner oder gleich.
    $b = (v >= 18);$
    Jetzt ist b `true`, falls *alter* 18 oder größer als 18 ist. Wir könnten b also gut *volljaehrig* nennen.

- Logische Operatoren:
  - $c\&d$ bedeutet das logische *und*, das genau dann wahr ist, wenn sowohl c als auch d den Wert `true` haben.
  - $c|d$ bedeutet das logische *oder*, das wahr ist, wenn c wahr ist, wenn c und d wahr sind, wenn d wahr ist.
  - $c\hat{}d$ bedeutet das *ausschließende oder*, das wahr ist, wenn c wahr und d falsch ist, wenn d wahr und c falsch ist.
  - $!c$ ist wahr, wenn c falsch ist.

Man schreibt die Bedeutung logischer Operatoren meist in Form von Wahrheitstafeln auf. Außen stehen die Variablen und der logische Operator, innen stehen die Belegungen der Variablen und der sich daraus ergebende Wahrheitswert als Ergebnis des Operators.

a	! a
f	t
t	f

a	b	a & b
f	f	f
f	t	f
t	f	f
t	t	t

a	b	a | b
f	f	f
f	t	t
t	f	t
t	t	t

a	b	a^b
f	f	f
f	t	t
t	f	t
t	t	f

Bedingungen und logische Operatoren können natürlich auch zusammen vorkommen. Dann wird stets zuerst die Bedingung ausgewertet, bevor die logische Operation ausgeführt wird!

```
boolean schulfrei;
int temperatur;
schulfrei = temperatur > 39 | 15 >= temperatur;
```

Hier wird der Vergleich einer Temperatur mit einem oberen und einem unteren Schwellwert mit | verknüpft. Die Variable $schulfrei$ ist `true`, wenn die Temperatur mehr als 39 oder höchstens 15 Grad beträgt. Sie wäre auch wahr, wenn die Temperatur sowohl mehr als 39 als auch weniger als 16 Grad beträgt – das kommt nur nicht vor.

Die Auswertungsreihenfolge kann auch durch gedoppelte Operatorzeichen gesteuert werden.

$c\&\&d$

bedeutet, daß d nur ausgewertet wird, wenn c bereits wahr ist. Analog wird bei

$c||d$

d nur ausgewertet, wenn c falsch ist. Bei den einfachen Zeichen werden stets beide Seiten ausgewertet.

3.3.1 Was wissen Sie jetzt?

Sie sollten nun wissen, wie man in JAVA die Grundrechenarten durchführt, wie Zeichenketten konkateniert werden und wie Variablen vom Typ `boolean` verwendet werden. Mit den Operatoren haben Sie die einfachsten Handlungen kennengelernt.

3.4 Methoden

Endlich, endlich kommen wir zum Kernstück der Programmierung, den Methoden. Die Klassen wurden ja nur unter dem Gesichtspunkt der Methoden gebildet. Die Variablen sind eigentlich nur zum Gebrauch in Methoden da. Wir haben oben bei den Variablen nur angegeben, daß Klassen ihren Wertebereich angeben. Jetzt gehen wir weiter: die Klassen geben durch ihre Methoden auch an, welche Handlungen ein Objekt – auf das eine Variable verweist – ausführen kann. Sehen wir uns also an, in welcher Form Methoden aufgeschrieben werden und wie sie die Verarbeitung von Botschaften realisieren. Schließlich begegnen wir dem Gedanken der Referenz wieder, wenn wir sehen, wie Variable ihre Werte an Methoden übertragen. Und dann sehen Sie endlich ein komplettes Programm, das Ball-Beispiel aus der Einführung, und können selbst einfache Programme in JAVA schreiben.

MethodDeclaration :
    MethodHeader MethodBody
MethodHeader :
    $Modifiers_{opt}$ *Type MethodDeclarator* $Throws_{opt}$
    $Modifiers_{opt}$ `void` *MethodDeclarator* $Throws_{opt}$
MethodDeclarator :
    Identifier ($FormalParameterList_{opt}$)
    MethodDeclarator `[]`
FormalParameterList :
    FormalParameter
    FormalParameterList , *FormalParameter*
FormalParameter :
    Type VariableDeclaratorId
Throws :
    `throws` *ClassTypeList*
ClassTypeList :
    ClassType
    ClassTypeList , *ClassType*
MethodBody :
    Block
    ;

Tabelle 4: Methodendeklaration

3.4.1 Methodendeklaration

Eine Methode beginnt mit Modifikationen, die wir schon im Abschnitt 3.1 gesehen haben, aber erst in Abschnitt 3.8 verstehen werden. *public* ist der Modifikator, den wir hier verwenden: die Methode kann von überall her gesehen werden. Auch bei Methoden – wie bei Variablen – gibt es das Schlüsselwort *static*, das angibt, daß es um eine Methode der Klasse und nicht ihrer Objekte geht. Eine als *static* bezeichnete Methode wird unabhängig von einem Objekt aufgerufen. Sie wird also nicht von einem Objekt als dessen Tätigkeit ausgeführt, sondern "einfach so".

Wenn die Methode einen Wert zurückliefert, muß natürlich klar sein, aus welchem Wertebereich dieser Wert stammen darf. Es muß also der Typ angegeben werden. In der Methode wird mit *return*(*Variablenname*) ausgesagt, wessen Wert abgeliefert werden soll. Der Wert muß vom angegebenen Typ sein. Er wird abgeliefert an das Objekt, das die Methode aufgerufen hat. Diese Abgabe eines Wertes ist gar nicht so häufig. Meist liefert eine Methode nichts zurück, sondern verändert ein Objekt oder ruft eine andere Methode auf, die ein Objekt verändert, oder gibt eine Meldung an den Benutzer aus oder zeichnet ein Bild – all dies wird als *Seiteneffekt* bezeichnet. Wenn eine Methode nur über Seiteneffekte wirksam ist, so erhält sie statt des Typs das Schlüsselwort *void* als erste notwendige (und bei i vorhandenen optionalen Modifikationen als $i+1$te) Angabe.

Eine Methode wird stets mit ihrem Namen und ihren Parametern bezeichnet. Folglich sind gleichnamige Methoden mit unterschiedlich vielen Parametern oder mit Para-

metern unterschiedlichen Typs verschiedene Methoden.[7] Eine Methode hat einen Namen (*Identifier*) und in Klammern ihre Parameter. Ein Parameter ist eine Variable. Da Variable nur Werte aus einem vorher bestimmten Wertebereich annehmen können, muß wieder der Typ der Variablen (eine Klasse oder ein einfacher Datentyp) dem Variablennamen vorangestellt werden. Wenn gar keine Parameter gebraucht werden, bleiben die Klammern dennoch stehen, woran man Methoden leicht als solche erkennt. Beispiel:

```
public void drucke () {
        System.out.println (name + "hat "+ hausrat.toString ());
}
```

Bei einigen Methoden ist absehbar, daß zur Laufzeit Fehler vorkommen können. Man kann dann dem Übersetzer im Programm mitteilen, daß man mit einem Fehler oder einer Ausnahme rechnet und erzeugt ein Objekt einer der im Paket `java.lang` definierten Fehlerklassen. Dies tut man mit dem Schlüsselwort *throws* und der Angabe der Fehlerklasse. Mit Methoden dieser Klassen lassen sich Fehlermeldungen aus dem Programm heraus konstruieren. Das Programm wird übersetzt und wenn möglich ausgeführt. Es gibt selbst seine Fehlermeldung aus, deren Erstellung Teil des Programms ist. Ausführlich besprechen wir dies in Abschnitt 3.10.

Der Code, der dann tatsächlich etwas tut, ist ein Block. Ein *Block* ist eine Folge von Anweisungen, die in geschweifte Klammern eingefaßt ist. Die Anweisungen verwenden die Parameter der Methode oder Variablen, die Eigenschaften von Objekten derjenigen Klasse bezeichnen, für die die Methode definiert wurde. Mit Methoden können wir Eigenschaften von Objekten verändern.

3.4.2 Realisierung von Assoziationen

Die einfachste Assoziation, die wir in Abschnitt 2.1 kennengelernt haben, ist die 1 : 1 Assoziation.

```
class Besitz  {
        Mensch besitzer;
        String name;

        public Besitz (String _name) {
                name = _name;
        }
        public void gehoere (Mensch _besitzer) {
                besitzer = _besitzer;
        }
}
```

Jeder Gegenstand der Klasse **Besitz** hat zwei Eigenschaften: einen Besitzer zu haben, wobei nur Menschen Besitzer sein können, und einen Namen, der eine Zeichenkette ist. Der Konstruktor **Besitz** legt für jeden Gegenstand einen Namen an. Es gibt also obligatorische Eigenschaften (hier: Name), die jedes Objekt einer Klasse hat, und fakultative

[7] Über diese eigentlich selbstverständliche Eigenschaft von JAVA wird sehr viel Aufhebens gemacht. Kümmern Sie sich nicht darum! Wenn Sie zur Identifikation einer Methode stets ihren Namen und die Parameter verwenden, können Sie nicht fehl gehen!

Eigenschaften, die nicht immer vorhanden sein müssen (hier: der Besitzer). Wir teilen dem Gegenstand den Besitzer durch die Methode **gehoere** mit. Der Gegenstand kann einen Verweis auf den Besitzer empfangen und trägt ihn bei sich ein: "ich gehoere *_besitzer*". Damit hat der Gegenstand nun auch die Eigenschaft, jemandem zu gehören; die Variable, die diese Eigenschaft ausdrückt, hat einen Wert bekommen. Die Methode **gehoere** arbeitet also mit einem Seiteneffekt, gibt keinen Wert zurück (*void*). Hier sehen wir eine Assoziation, nämlich die zwischen einem Gegenstand und seinem Besitzer. Da ein Gegenstand normalerweise genau einen Besitzer hat, kann die Assoziation durch die Methode **gehoere** leicht aufgebaut werden.

Komplizierter ist die $1 : m$-Assoziation. Ein Mensch hat viele Gegenstände. Wir fassen diese in der Behälterklasse **Hausrat** zusammen. Wenn wir so aufwendige Tätigkeiten wie Geld verdienen und Gegenstände herstellen erst einmal weglassen, dann ist die Methode recht einfach, wie ein Mensch einen neuen Gegenstand bekommt: man empfängt ihn einfach und erweitert den Hausrat.

```
class Mensch  {
        String name;
        boolean geschlecht;
        Hausrat hausrat;

        public Mensch (String _name,boolean _geschlecht)  {
                name = _name;
                geschlecht = _geschlecht;
                hausrat = new Hausrat ();
        }
        public void empfang (Besitz geschenk)  {
                hausrat.aufnehmen (geschenk,this);
        }
}
```

Der Konstruktor legt für jeden Menschen einen Namen, ein Geschlecht und ein Objekt der Behälterklasse **Hausrat** an. Alles, was dann die Methode **empfang** noch tun muß, ist, eine Nachricht an das Objekt der Klasse **Hausrat** zu schicken und damit den neuen Besitz zu bezeichnen. Dies geschieht durch den Aufruf der Methode **aufnehmen** von **Hausrat** mit dem neuen Gegenstand als Parameter. Sich selbst bezeichnet der Mensch in der Methode **aufnehmen** durch das Schlüsselwort *this*, das in einem Objekt auf sich selbst zeigt.

Was macht nun der **Hausrat**? Es ist eine Unterklasse von **Vector**. Seine Methode **aufnehmen** hat als Parameter einen Gegenstand und einen Menschen. Der Gegenstand bekommt die Nachricht, nunmehr dem Menschen zu gehören. Das Objekt, das der Wert der Variablen *geschenk* ist, ist vom Typ **Besitz** und verfügt also über die Methode **gehoere**. Diese Methode realisiert "die andere Seite" der Relation zwischen Mensch und Besitz. Wir haben die eine Seite, die Assoziation von Mensch zu Besitz bereits in der Klasse **Mensch** realisiert. Genau in der Methode von **Mensch**, die dies tut, wird auch die Assoziation von Besitz zu Mensch aufgerufen, die bei **Hausrat** realisiert wird. Obwohl wir leider die Relation in zwei Assoziationen aufteilen müssen, haben wir wenigstens sichergestellt, daß bei Einrichten der Assoziation von Mensch zu Besitz auch gleich das Einrichten der Assoziation von Besitz zu Mensch aufgerufen wird. Dies drückt den Zusammenhang zwischen

den beiden Assoziationen aus und macht das Programm leichter wartbar.

Der Hausrat hat von der Oberklasse **Vector** die Methode **addElement** geerbt und ruft diese mit dem Gegenstand als Parameter auf. Der Vorteil der Vererbung zeigt sich hir in der Kürze der Klassendeklaration. Sich selbst bezeichnet der Hausrat durch *this*.

```
class Hausrat extends Vector {

        public void aufnehmen (Besitz geschenk, Mensch mensch) {
                geschenk.gehoere (mensch);
                this.addElement(geschenk);
        }
}
```

Es unterstreicht den objektorientierten Charakter, daß das Objekt, dessen Methode ausgeführt werden soll, dem Methodennamen vorangestellt wird, wissen Sie schon, aber noch nicht, wie eindeutige Namen vergeben werden. Das kommt in Abschnitt 3.8.

Wir haben jetzt durch drei Assoziationen – nämlich den *besitzer* bei der Klasse **Besitz** und den *hausrat* bei der Klasse **Mensch** und das Element vom Typ **Besitz** bei der Klasse **Hausrat** – die Relation zwischen **Mensch** und **Besitz** ausgedrückt. Die Methoden sind so organisiert, daß der Mensch dem Hausrat eine Botschaft schickt und dieser dem Besitz, damit der neue Gegenstand im Besitz des Menschen auch sofort seinen Besitzer kennt und in der Kollektion *hausrat* verzeichnet ist.

3.4.3 Parameterübergabe

Methoden von Objekten können die Variablen verwenden, die Eigenschaften der Objekte bezeichnen. Diese brauchen nicht als Parameter übergeben zu werden. Die Eigenschaft (die Variable) ist bei jedem Objekt der betreffenden Klasse vorhanden und bekannt. Methoden können aber auch Parameter haben. Bei der Methodendeklaration wird für jeden Parameter der Typ und der Name der Variablen angegeben. Diese Variablen sind nur innerhalb der Methode bekannt. Ist die Methode abgearbeitet, sind die Variablen "vergessen".

Beim Aufruf der Methode wird als Parameter ein konkreter Wert oder eine Variable angegeben, mit deren Wert die Methode nun arbeiten soll. Diese Variable muß natürlich einen Typ haben, der dem in der Methodendeklaration für den Parameter angegebenen Typ entspricht. Wir wissen ja (Abschnitt 3.2.3), daß Variablen, deren Werte Objekte sind, ihre Werte mittels der Referenzzuweisung erhalten. Betrachten wir nun einen Parameter mit einer Klasse als Typ. Der Parameter ist eine Variable, der beim Methodenaufruf ein Wert zugewiesen werden soll. Analog zur Zuweisung geschieht dies in JAVA auf zweierlei Weise.

Definition 3.6: Referenzübergabe Sei in der Methodendeklaration ein Parameter v angegeben, dessen Typ kein einfacher Datentyp ist, sei im Methodenaufruf an entsprechender Stelle der Parameterliste eine Variable w angegeben, so wird die Adresse, die als Wert von w bekannt ist, kopiert und als Wert von v innerhalb der Methode eingetragen. Diese Parameterübergabe heißt Referenzübergabe (englisch: *call by reference*).

Beispielsweise hatten wir in der Methodendeklaration **gehoere** in der Klasse **Besitz** den Parameter *_besitzer*, der nur Werte annehmen kann, die Objekt der Klasse **Mensch**

sind. Beim Aufruf der Methode innerhalb der Methode **aufnehmen** von **Hausrat** wird als Parameter *mensch* angegeben. Der Wert von *mensch* ist die Adresse, an der beispielsweise Uta beschrieben ist. Die Adresse der Beschreibung von Uta wird nun kopiert und als Wert von *_besitzer* eingetragen. Dies ist die Referenzübergabe. Die Methode **gehoere** tut nichts anderes, als der Variablen *besitzer* eines Gegenstandes den Wert zuzuweisen, den der Parameter hat. Also wird die Adresse der Beschreibung von Uta nun auch noch als Wert von *besitzer* eingetragen. Dies ist die Referenzzuweisung.

Noch ausführlicher: Nehmen wir also an, wir hätten ab der Adresse 32 ein Objekt *ball* beschrieben und ab der Adresse 512 Uta:

Adresse	32	Adresse	512	Adresse 640
name:	"ball1"	name:	"Uta"	[]
farbe:	"blau"	geschlecht:	true	
besitzer:		hausrat:	Adr. 640	

Das Objekt, das ab Adresse 512 beschrieben ist, ruft nun die Methode **aufnehmen** auf mit $(ball, this)$. Zu diesem Zeitpunkt sei der Wert von *ball* die Adresse 32. Jetzt erhalten die Parameter von **aufnehmen** per Referenzübergabe ihren Wert: *geschenk* bekommt die Kopie der Adresse von *ball* (32) und *mensch* die Kopie der Adresse 512.

In der Methode **aufnehmen** wird nun die Methode **gehoere** des Objektes *geschenk*, also des ab Adresse 32 beschriebenen Balls, aufgerufen. Der Parameter des Aufrufs ist *mensch*. Diese Variable hat als Wert die Adresse 512. Der Parameter der Deklaration ist *_besitzer*. Der Wert von *mensch* ist die Referenz auf Uta (Adresse 512) und wird in Kopie an *_besitzer* übergeben.

Mit der Methode **gehoere** verändert ein Gegenstand seine Eigenschaft *besitzer* durch eine Referenzzuweisung. Nach so viel Durchreichen eines Wertes – die Adresse 512 wurde von *this* and *mensch* an *_besitzer* an *besitzer* gereicht – nun ein Effekt: *ball* hat als *besitzer* nun das ab Adresse 512 beschriebene Objekt. Der Wert, Adresse 512, wurde vom Objekt Uta an das Objekt *hausrat* und von da an das Objekt *ball* übergeben.

Adresse	32	Adresse	512	Adresse 640
name:	"ball1"	name:	"Uta"	[Adr. 32]
farbe:	"blau"	geschlecht:	true	
besitzer:	Adr. 512	hausrat:	Adr. 640	

Wenn Parameter keine Objekte als Wert haben, sondern von einfachem Datentyp sind, wird der Wert direkt übergeben.

Definition 3.7: Wertübergabe Sei in der Methodendeklaration ein Parameter v angegeben, dessen Typ ein einfacher Datentyp ist, sei im Methodenaufruf an entsprechender Stelle der Parameterliste eine Variable w angegeben, so wird der Wert von w kopiert und als Wert von v innerhalb der Methode eingetragen. Diese Parameterübergabe heißt Wertübergabe (englisch: *call by value*).

Da bei der Wertübergabe kein Bezug zwischen den Variablen v und w hergestellt wird, sondern nur der Wert von w als Wert von v eingetragen wird, gibt es keine Referenz von v auf w. Folglich kann w nicht durch v verändert werden. In [Dißmann und Doberkat, 1998] steht das Beispiel:

```
class Zaehler {
        public void erhoehe1 (int _x) {
                _x += 1;
                System.out.println ("waehrend "+ _x);
        }
}

class WertBeispiel {
        public static void main (String argv[]) {
                int y =3;
                Zaehler z;
                z = new Zaehler ();
                System.out.println ("vorher "+ y);
                z.erhoehe1 (y);
                System.out.println ("nachher "+ y);
        }
}
```

Es liefert natürlich die Ausgabe:

```
vorher 3
waehrend 4
nachher 3
```

y ist eine Variable, die an einem Speicherplatz (z.B. Adresse 16) steht, den Namen y hat und als Wert gleich bei der Deklaration 3 erhält. Die Methode **erhoehe1**, mit der ein Zähler eine Zahl um 1 erhöht, ist mit dem Parameter $\_x$ deklariert. Die Variable mit dem Namen $\_x$ stehe an der Adresse 128. Die Wertübergabe beim Methodenaufruf trägt als Wert von $\_x$ nun *nicht* die Adresse 16 ein, sondern 3. Nun wird der Wert von $\_x$ inkrementiert. An Adresse 16 wird nichts verändert.

Nehmen wir hingegen ein Objekt, das die Zahl als eine Eigenschaft besitzt, dann nutzen wir die Referenzübergabe aus und haben daher einen Seiteneffekt[8].

```
class ZaehlerO {
        public void erhoehe1 (Geld _x) {
                _x.betrag += 1;
                _x.drucke("waehrend ");
        }
}

class Geld {
        int betrag;
        String waehrung;
        public Geld (int _betrag, String _waehrung) {
                betrag = _betrag;
                waehrung = _waehrung;
        }
        public void drucke (String txt) {
```

[8] Bitte, beachten Sie in diesem Beispiel noch nicht die Ausgabe auf den Bildschirm, die einfach Zahlen als Zeichenketten ausgibt, was in JAVA möglich ist, aber unschön.

```
            System.out.println (txt + betrag + waehrung);
        }
}

class WertBeispielK  {
        public static void main (String argv[])  {
            ZaehlerO z;
            Geld y;
            y = new Geld (3,"Euro");
            z = new ZaehlerO ();
            y.drucke ("vorher ");
            z.erhoehe1 (y);
            y.drucke ("nachher ");
        }
}
```

Dies Programm liefert natürlich die Ausgabe:

```
vorher 3Euro
waehrend 4Euro
nachher 4Euro
```

3.4.4 Das vollständige Ballbeispiel

Die im Verzeichnis ~`gvpr000/ProgVorlesung/Packages/ballbeispiel/` stehende Datei `BallBeispiel.java`, die das in der Einführung verwendete Beispiel von Uta und dem Ball in JAVA darstellt, sieht nun so aus:

```
package ballbeispiel;                                                    // 1
import java.util.*;                                                      // 2
import AlgoTools.IO;                                                     // 3


class Mensch  {                                                          // 4
        String name;                                                     // 5
        String geschlecht;                                               // 6
        Hausrat hausrat;                                                 // 7

        public Mensch ()  {                                              // 8
            name = IO.readString ("Bitte einen Vornamen eingeben:");     // 9
            geschlecht = IO.readString ("Geschlecht? (w, m) ");          // 10
            hausrat = new Hausrat ();                                    // 11
        }                                                                // 12
        public void tritt (Ball ball)  {                                 // 13
            float dx, dy;                                                // 14
            dx = IO.readFloat ("Wie soll "+this.name+"den Ball treten? DX");  // 15
            dy = IO.readFloat ("Wie soll "+this.name+"den Ball treten? DY");  // 16
            ball.rolle (dx,dy);                                          // 17
        }                                                                // 18
        public void empfang ()  {                                        // 19
            Besitz geschenk;                                             // 20
            String geschenkN;                                            // 21
```

```
            geschenkN = IO.readString ("Was bekommt "+this.name+"jetzt geschenkt?");  // 22
            geschenk = new Besitz (geschenkN);                                        // 23
            hausrat.aufnehmen (geschenk,this);                                        // 24
        }                                                                             // 25
        public void empfang (Besitz besitz) {                                         // 26
            hausrat.aufnehmen (besitz,this);                                          // 27
        }                                                                             // 28
        public void drucke () {                                                       // 29
            System.out.println (name + "hat "+ hausrat.toString ());                  // 30
        }                                                                             // 31
}                                                                                     // 32


class Hausrat extends Vector {                                                        // 33

        public void aufnehmen (Besitz geschenk, Mensch mensch) {                      // 34
            geschenk.gehoere (mensch);                                                // 35
            this .addElement(geschenk);                                               // 36
        }                                                                             // 37
}                                                                                     // 38

class Besitz  {                                                                       // 39
        Mensch besitzer;                                                              // 40
        String name;                                                                  // 41

        public Besitz (String _name) {                                                // 42
            name = _name;                                                             // 43
        }                                                                             // 44
        public void gehoere (Mensch _besitzer)  {                                     // 45
            besitzer = _besitzer;                                                     // 46
        }                                                                             // 47
        public String toString () {                                                   // 48
            return (name);                                                            // 49
        }                                                                             // 50
}                                                                                     // 51

class Ball extends Besitz  {                                                          // 52
        float x,y;                                                                    // 53
        String farbe;                                                                 // 54

        public Ball (String _name)  {                                                 // 55
            super (_name);                                                            // 56
            farbe= IO.readString ("Welche Farbe soll der Ball haben? ");              // 57
            x = IO.readFloat ("Wo ist er auf der X-Achse? ");                         // 58
            y = IO.readFloat ("Wo ist er auf der Y-Achse? ");                         // 59
        }                                                                             // 60

        public void rolle (float dx,float dy) {                                       // 61
            x += dx; y += dy;                                                         // 62
        }                                                                             // 63

        public void drucke ()  {                                                      // 64
            System.out.println (name+", "+farbe+"ist jetzt in Position:"+ x + ""+y);  // 65
```

```
    }                                                                                    // 66
}                                                                                        // 67

class BallBeispiel {                                                                     // 68
    public static void main (String argv[]) {                                           // 69
        Mensch mensch;                                                                   // 70
        Ball ball;                                                                       // 71
        mensch = new Mensch ();                                                          // 72
        System.out.println ("Und jetzt bekommt "+mensch.name+"einen Ball!");            // 73
        ball = new Ball ("ball");                                                        // 74
        mensch.empfang (ball);                                                           // 75
        while (IO.readString ("Soll "+mensch.name+"den Ball treten?").equals("ja")) {    // 76
            mensch.tritt (ball);                                                         // 77
            ball.drucke ();                                                              // 78
        }                                                                                // 79
        while (IO.readString ("Soll "+mensch.name+                                       //80
                              " ein Geschenk bekommen? (ja, nein) ").equals("ja")) {     // 81
            mensch.empfang ();                                                           // 82
            mensch.drucke ();                                                            // 83
        }                                                                                // 84
    }                                                                                    // 85
}                                                                                        // 86
```

Die Klasse **Mensch** haben wir mit ihren Methoden **Mensch** (Konstruktor) und **empfang** schon kennengelernt. Allerdings sieht sie nun doch anders aus, weil wir den Konstruktor nicht mit Parametern aufrufen. Wir wollen der Benutzerin die Möglichkeit geben, einen Namen und ein Geschlecht für ein neues Objekt der Klasse **Mensch** anzugeben. Wir wollen diese Ausprägungen von Eigenschaften also nicht innerhalb des Programms festlegen, sondern von außen erhalten. Dazu verwenden wir die Klasse **IO** aus dem Paket `AlgoTools` von [Vornberger und Thiesing, 1998]. Später werden wir sehen, wie die Methoden für das Lesen von Benutzereingaben funktionieren (Abschnitt 7.1). Im Moment nehmen wir einfach hin, daß es zwei Methoden gibt, die eine Zeichenkette (String) auf den Bildschirm schreiben und dann etwas, was die Benutzerin tippt, als Wert zurückliefert. Die Methode **readString** liest eine Zeichenkette ein. So erhält in Zeile 9 die Variable *name* den Wert, den die Benutzerin angegeben hat. Die Eigenschaft des neuen Objektes der Klasse **Mensch** erhält so ihre Ausprägung. Analog wird in Zeile 10 das Geschlecht angegeben. Das Anlegen des Hausrats erfordert keine Eingabe durch die Benutzerin. Es wird ein Objekt der Behälterklasse **Hausrat** erzeugt, das noch kein Element enthält. Die Methode **readFloat** liest eine Zahl vom Typ `float` ein. So werden in Zeile 15 und 16 Zahlen, die die Benutzerin angegeben hat, den Variablen dx und dy zugewiesen. Analog erhalten in Zeile 58 und 59 die Variablen x und y durch die Eingaben vom Bildschirm ihre Werte.

Die Methode **tritt** realisiert eine Botschaft an den Ball. Wir stellen uns vereinfachend vor, daß sich der Ball in einem Koordinatensystem an einer Position befindet. Der Tritt wird als eine Verschiebung der Position des Balles modelliert. Ein Objekt der Klasse **Mensch** teilt dem Ball mit, um wieviel er sich in Richtung der X-Achse (dx) und um wieviel er sich in Richtung der Y-Achse (dy) verschieben soll (Zeile 17).

Es gibt nun zwei Methoden mit dem Namen **empfang**. Da sie unterschiedliche Parameter haben, nämlich einmal keinen und einmal einen Besitz, sind es zwei Methoden.

Die Methode ohne Parameter verwenden wir, wenn der neue Besitz von der Benutzerin angegeben wird. Es wird ein neues Objekt der Klasse **Besitz** erzeugt (Zeile 23), das den Namen (*geschenkN*) hat, den die Benutzerin eingetippt hat. Danach wird die Methode **aufnehmen(Besitz b, Mensch m)** aufgerufen. Die Methode mit Parameter setzt voraus, daß das Objekt vom Typ **Besitz** bereits erzeugt ist und nun die Referenz auf dies Objekt übergeben wird. Dabei reicht **empfang(Besitz b)** lediglich diese Referenz an **aufnehmen(Besitz b, Mensch m)** von **Hausrat** weiter. Wir sehen hier zwei Beispiele für die Referenzübergabe hintereinander.

Jede Klasse hat eine Methode, wie ihre Objekte sich drucken. Diese Methode besteht in einem Aufruf der von JAVA bereitgestellten Methode **println** (print line), deren Parameter ein **String** ist. In Zeile 30 ist die Methode, sich zu drucken, für Menschen angegeben. Die Variable *name* kann einfach ausgedruckt werden, weil sie ja vom Typ **String** ist. Das Wort *hat* wird durch Anführungszeichen zu einem Objekt der Klasse **String** gemacht. Der Hausrat hingegen ist ja nicht von diesem Typ und muß daher erst in ein Objekt des Typs **String** umgewandelt werden. Alle in JAVA definierten Klassen sollten eine Methode **toString** haben, die angibt, wie ein Objekt der Klasse als Zeichenkette dargestellt wird. Da **Hausrat** eine Unterklasse von **Vector** ist und diese von JAVA definierte Behälterklasse eine solche Methode besitzt, verfügt auch das Objekt *hausrat* über diese Methode. Allerdings verlangt **Vector**, daß alle Objekte im Behälter auch eine Darstellung als **String** haben, die mit der Methode **toString()** erreicht wird. Deshalb wird in Zeile 48 und 49 für den Besitz eine solche Methode definiert: ein Besitz wird als Zeichenkette durch seinen Namen (der ohnehin eine Zeichenkette ist) dargestellt.

Die Klasse **Hausrat** ist so geblieben, wie bereits vorgestellt. Ihre Methode **aufnehmen** benötigt keine Eingaben vom Bildschirm und wird nur von der Methode **empfang** von **Mensch** aufgerufen. Die Klasse erbt von **Vector** sowohl den Konstruktor als auch das Hinzufügen von Elementen in das Objekt. Die Vererbung ist im Sinne von **Hausrat ist ein Vector** (vgl. Abschnitt 2.1).

Die Klasse **Besitz** hat lediglich die Methode **toString** dazubekommen. Das haben wir gerade besprochen.

Die Klasse **Ball** ist eine Unterklasse von **Besitz**. Die Vererbung ist im Sinne von **Ball ist ein Besitz**. **Ball** hat folglich die Eigenschaft, einen Namen zu haben. Bei der Konstruktion eines neuen Balls wird der Konstruktor der Oberklasse (**Besitz**) aufgerufen (Zeile 56), der ja als einzigen Parameter *_name* hat. Der Parameter von **Besitz(String _name)** erhält nun den Wert, der bei Aufruf des Konstruktors **Ball(String _name)** an erster Stelle der Parameterliste übergeben wird. Obendrein bekommt ein neuer Ball eine Position und eine Farbe (Zeilen 57 - 59). Den Aufruf des Konstruktors sieht man in Zeile 74.

Die Methode **rolle** von **Ball** empfängt die Nachricht einer Positionsänderung und verschiebt durch zwei einfache Additionen die Koordinaten. In der Nachricht stehen nicht die gegenwärtigen Koordinaten, sondern lediglich die Verschiebungen. Die gegenwärtige Position ist durch die Eigenschaften des Objekts x, y angegeben und innerhalb des Objekts stets zugreifbar. Da in der Methode diese Eigenschaften verändert werden, ist die neue Position auch nach Verlassen der Methode noch zu sehen. In Zeile 77 tritt das Objekt, auf das die Variable *mensch* zeigt, den Ball, wodurch die Methode **rolle** aufgerufen wird. In Zeile 78 druckt sich der Ball aus und wir können auf dem Bildschirm die Positionsänderung sehen. Wie schon **tritt** ist auch **rolle** grob vereinfacht. Wir müssen in der Programmdokumentation festhalten, daß wir nur eine zweidimensionale Fläche modellieren, auf der

der Ball aufliegt und daß der Tritt nur als Schub auf dieser Fläche, ohne Bezug zum Ball dargestellt ist. Auch müssen wir uns merken, daß diese Methoden so weltfremd sind, daß wir sie nicht zur Simulation oder gar tatsächlichem Kicken (von Robotern, z.B. im RoboCup) einsetzen können. Die Granularität, die wir hier gewählt haben, macht gerade die Realisierung von Assoziationen in JAVA und die Verwendung der Vererbung deutlich.

Die Klasse mit der *main*-Methode ist **BallBeispiel**. Hier wird als erstes, in Zeile 72, der Konstruktor **Mensch()** aufgerufen. Die Variable *mensch* zeigt auf das neue Objekt der Klasse **Mensch**. Der Konstruktor fragt die Benutzerin nach einem Vornamen und nach dem Geschlecht des neuen Objektes. Deshalb kann in Zeile 73 der Name dieses Menschen ausgedruckt werden. Innerhalb der Klasse **Mensch** haben wir den Namen kurz als *name* geschrieben. Hier schreiben wir *mensch.name*, denn **BallBeispiel** hat ja nicht die Eigenschaft Name und außerhalb von **BallBeispiel** haben mehrere Klassen die Variable *name*. Die nächste wichtige Handlung ist die Konstruktion eines Balles in Zeile 74. Wir legen den Namen vom Programm aus fest, weil eine Frage nach seinem Namen die Benutzerin irritieren könnte ("mein Ball heißt Willi"?). Dieses Objekt *ball* bekommt *mensch* durch die Methode **mensch.empfang(ball)** (Zeile 75). Jetzt folgen zwei Schleifen. Die Bedeutung von *while* ist in Abschnitt 3.5 beschrieben. Hier fasse ich das Verhalten zusammen. Die erste Schleife von Zeile 76 - 79 ruft die Methode **mensch.tritt(ball)** auf, solange die Benutzerin auf die Frage, ob *mensch* den Ball treten soll, mit "ja" antwortet. Wenn der Ball getreten wurde und gerollt ist, druckt er sich aus. Die zweite Schleife von Zeile 80 - 84 ruft die Methoden **mensch.empfang()** und **mensch.drucke()** auf, solange die Benutzerin auf die Frage, ob *mensch* ein Geschenk bekommen soll, mit "ja" antwortet.

Wir rufen unser Programm auf mit

```
java ballbeispiel.Ballbeispiel
```

und sehen auf dem Bildschirm

```
Bitte einen Vornamen eingeben:
```

Sagen wir ruhig: *Uta*

```
Geschlecht: (w,m)
```

w

```
Und jetzt bekommt Uta einen Ball!
Welche Farbe soll der Ball haben?
```

blau

```
Wo ist er auf der X-Achse?
```

1.0

```
Wo ist er auf der Y-Achse?
```

1.2

```
Soll Uta den Ball treten?
```

ja

```
Wie soll Uta den Ball treten? DX
```

3.0

```
Wie soll Uta den Ball treten? DY
```

0.8

```
ball, blau ist jetzt in Position:4.0 2.0
Soll Uta den Ball treten?
```

nein

```
Soll Uta ein Geschenk bekommen? (ja, nein)
```

nein

Natürlich können Sie durch eigene, andere Eingaben ein anderes Verhalten des Programms erzielen.

3.4.5 Programmzustände

Im Abschnitt 2.1 wurden Uta und ihr Ball eingeführt. Das war es, *was* wir modellieren wollten. Nun haben wir Sprachkonstrukte von JAVA angewandt und damit festgelegt, *wie* wir die Sachverhalte programmieren. Die Frage *warum* soll hier informell im Sinne der Effektivität beantwortet werden. Dazu betrachten wir die Objekte und Variablen, die in dem Programm vorkommen. Objekte sind: *mensch*, *besitz*, *ball* und *hausrat*. Ihre Variablen sind: In **Mensch**: *mensch.name*, *mensch.geschlecht*, *mensch.hausrat*; In **Besitz**: *besitz.besitzer*,*besitz.name*; In **Ball**: *ball.x*, *ball.y*, *ball.farbe*; In **BallBeispiel**: *mensch*, *ball*. Jede dieser Variablen hat einen Typ, also einen Wertebereich. Der (theoretische) *Zustandsraum* des Programms besteht aus allen Kombinationen von Werten aller Variablen.

Definition 3.8: Programmzustand Ein Programmzustand besteht aus der Belegung aller Variablen mit einem Wert.

Beispielsweise ist z_3 ein Zustand unseres Programms:

$z_3 = mensch.name : \text{“}Uta''$, $mensch.geschlecht : w$, $mensch.hausrat : [ball]$,
$besitz.besitzer : Uta$, $besitz.name : \text{“}ball''$, $ball.x : 1.0$, $ball.y : 1.2$, $ball.farbe : blau$.

Dieser Programmzustand besteht nach Zeile 75. Wir können die *main*-Methode als die Folge von Anweisungen betrachten, die von einem Anfangszustand zu einem Endzustand führt. Wir können die Zustände Schritt um Schritt verfolgen und uns so die Arbeitsweise des Programms auch ohne Interpreter klar machen. Allerdings kennen wir die Zustände nicht genau, da sie von Eingaben der Benutzerin zur Laufzeit abhängen. Immerhin können wir durch das Typ-Konzept die Wertebereiche der Variablen angeben. Manchmal können wir aber auch noch mehr aussagen. So, wie die Methode **tritt** deklariert ist, kann die Position in allen vier Feldern eines zweidimensionalen Koordinatensystems liegen. Hätten wir die Methode nur für positive Zahlen definiert, käme der Ball nie zurück. Wir könnten dann über die Klasse der Variablen $ball.x$ und $ball.y$ hinaus die Zusicherung machen, daß für jeden Anfangswert i dieser Variablen gilt: $i \leq ball.x$ bzw. $i \leq ball.y$.

Aussagen über Programmzustände heißen *Zusicherungen* (engl.: assertion). Sie werden als logische Formeln mit dem Zustand (den Variablen) als Argument geschrieben: $P(z)$. Verschiedene Zusicherungen können in logischen Beziehungen stehen. So impliziert z.B. die Aussage, $P(j) = j > 5$ die Aussage, $Q(j) = j > 4$, geschrieben als $P \rightarrow Q$. Wir können für wertverändernde Operationen (Wertzuweisungen, Operatoren, Methoden inklusive der Konstruktion eines Objektes) die Zusicherungen vor und nach Ausführung der Operation angeben. Beispielsweise sieht für die direkte Wertzuweisung

$k = 7$;

die Vorbedingung $P(k)$ so aus: k *beliebig*. Die Nachbedingung $Q(k)$ sieht so aus: $k = 7$. So fein muß die Modellierung nicht sein. Man kann auch Blöcke oder sogar eine gesamte **main**-Methode als die Operation behandeln, die eine Vor- und eine Nachbedingung hat.

Die Zusicherungen interessieren uns aus zwei Gründen (s. [Goos, 1996], S. 34f):

Zustandsverfolgung: Welche Zusicherungen $Q(z_n)$ gelten über den Zustand z_n, wenn wir wissen, daß $P(z_0)$ gilt? Wie verändert also unser Programm den Ausgangszu-

stand z_0 in n Schritten? Oder, anders herum, bei welchem Anfangszustand, beschrieben durch $P(z_0)$ ist garantiert, daß nach n Schritten $Q(z_n)$ gilt?

Verifikation: Haben wir $P(z_0)$ als Charakterisierung des Anfangszustands und $Q(z_n)$ als Charakterisierung des Zielzustands, dann ist P, Q eine *Spezifikation*. Hat das Programm, um dessen Zustände es geht, eine Folge von n Schritten, so daß $P(z_0)$ und $Q(z_n)$ gelten, und das Programm terminiert im Zustand z_n, dann ist das Programm spezifikationstreu oder *korrekt*. Die Nachprüfung der Korrektheit heißt *Verifikation*.

3.4.6 Was wissen Sie jetzt?

Sie wissen nun, wie man Methoden deklariert und aufruft. Insbesondere haben Sie dabei festgestellt, daß das Ergebnis, das eine Methode beim Aufruf an die aufrufende Stelle des Programms zurückliefert, von einem Typ sein muß, der bei der Methodendeklaration angegeben wird. Die meisten Methoden erbringen Resultate jedoch indirekt, indem sie Objekte verändern. Dann wird das Schlüsselwort *void* bei der Methodendeklaration angegeben.

Sie können nun eine Assoziation, die Sie sich bei der objektorientierten Modellierung ausgedacht haben, in JAVA-Anweisungen umsetzen: die 1:1-Assoziation als Eigenschaft eines Objektes, notiert durch eine Variable; die $1 : m$-Assoziation mithilfe einer Behälterklasse. Überlegen Sie, wie beim vollständigen Ballbeispiel die Diagramme ausgesehen haben. Damit trainieren Sie Ihre Fähigkeit, zu modellieren.

Beim Methodenaufruf wurde die Referenz- und die Wertübergabe besprochen. Daß dem Methodennamen ein Punkt und davor die Variable für das Objekt, das die Methode beherrscht, vorangestellt wird.

Probieren Sie, die Datei `BallBeispiel.java` so zu verstehen, als wären Sie der Übersetzer, also `javac`. Damit überprüfen Sie Ihr syntaktisches Verständnis von JAVA.

Probieren Sie, die Bindung der Variablen im Verlaufe des Programms bei verschiedenen Eingaben nachzuvollziehen. Dies ist die erste Annäherung an die (operationale) Semantik des Programms. Überlegen Sie sich Zustände, die das Programm bei seiner Ausführung einnimmt. Beschreiben Sie die Zustände durch Zusicherungen. Beschreiben Sie Operationen durch Vor- und Nachbedingungen.

3.5 Kontrollstrukturen

Kontrollstrukturen regeln den dynamischen Ablauf der Anweisungen eines Programms. Gerade die Einführung von Schleifen wird ja der *Mutter der Informatik*, Lady Ada Lovelace, zugute gehalten[9].

Bei einer Schleife kann immer dieselbe Folge von Anweisungen nacheinander für eine Menge von Objekten oder einfachen Daten ausgeführt werden. Wir brauchen dazu

[9] Lady Ada Lovelace (1815 - 1852) hatte als Hauslehrer den Cambridge-Professor William Fend, so daß sie eine fundierte Ausbildung in Mathematik und Astronomie erhielt. 1833 lernte sie Charles Babbage kennen und war fasziniert von seiner mechanischen Rechenmaschine. Sie übersetzte die Arbeit eines italienischen Militäringenieurs über eine Rechenmaschine und schrieb einen drei Mal so langen Kommentar dazu. In diesem Kommentar, den sie mit Babbage und de Morgan diskutierte, entwickelte sie die Idee der Programmierung sowie erste Programmierkonzepte wie Schleifen. Als einzige war Lady Lovelace in der damaligen Zeit kühn genug, Einsatzmöglichkeiten der Rechenmaschinen zu sehen, die heute selbstverständlich sind, neben Berechnungen von Prim- oder Bernouilli-Zahlen etwa auch das Erzeugen von Graphiken. Insofern kann Babbage als Vater der Hardware, Lady Lovelace als Mutter der Software betrachtet werden.

- einen Anfang, meist durch den Anfangswert einer Laufvariable (d.i. ein Zähler) gegeben,
- eine Abbruchsbedingung,
- den nächsten Wert der Laufvariable.

In JAVA werden Schleifen durch die Schlüsselwörter *for* und *while* angezeigt. *for* benötigt einen Zähler, dessen Anfangwert anzugeben ist. Die Abbruchsbedingung wird ebenfalls durch den Zähler ausgedrückt. Das nächste zu bearbeitende Objekt (oder die nächste Zahl, Buchstabe...) bekommt man ebenfalls über den Zähler.

```
for (i=1; 10 > i; i++)
        System.out.println(i*i);
```

In dem kleinen Beispiel ist i die Laufvariable, die um 1 inkrementiert wird, solange sie kleiner 10 ist. Die Ausgabe von i^2 ist der Block, der 9 mal durchgeführt wird, jeweils für einen neuen Wert von i. Wir könnten auch schreiben:

```
while (10 > i) {
        System.out.println(i*i);
        i++;
}
```

Die Abbruchbedingung ist eine logische Bedingung wie in Abschnitt 3.3 beschrieben. Ihr Wert ist vom Typ `boolean`. Der Wert von i wird nun nicht von 1 ausgehend hochgezählt, sondern außerhalb der Schleife bestimmt. Man kann mit beliebigen Werten (des richtigen Typs) in die *while*-Schleife kommen. Wenn und solange der Wert von i kleiner als 10 ist, wird das Quadrat gebildet und i inkrementiert.

Im Ballbeispiel haben wir zwei *while*-Schleifen gesehen. Die Abbruchbedingungen waren Eingaben, die nicht gleich dem **String** "ja" sind. Die Gleichheit von zwei Zeichenketten wird von der Methode **equals** geprüft (Zeilen 75 und 79). Diese Methode ist für alle Objekte der Klasse **String** vorhanden. **readString** liefert ein Objekt vom Typ **String**. Dies Objekt wird mit dem Parameter von **equals** verglichen. Sind beide Zeichenketten gleich, gibt **equals** den `boolean` Wert `true` zurück, sonst `false`. Nach der Abbruchsbedingung folgt der Block, der ausgeführt wird, solange die Bedingung wahr ist.

Hier wird die Abbruchbedingung geprüft, bevor der Block ausgeführt wird. Möchte man sicherstellen, daß der Block mindestens einmal ausgeführt wird, dann kann man die dritte Schleifenform von JAVA verwenden. Auch sie gibt mit dem Schlüsselwort *while* eine Abbruchbedingung an. Die Bedingung wird aber nach dem Block geprüft. Damit der Übersetzer erkennen kann, daß eine *while*-Schleife mit Abbruchsbedingung am Ende kommt, wird das Schlüsselwort *do* vor den Block gesetzt:

```
do {
        System.out.println(i*i);
        i++;
} while (10 > i);
```

Schleifen wiederholen Anweisungen. Wir können aber auch zu verschiedenen Anweisungen oder Blöcken verzweigen. Die Schlüsselwörter *if*, *else*, *case* und *switch* erlauben dies.

```
class SchulfreiMeldung {                          //Klasse fuer Durchsagen in einer Schule

    public static void main (String argv[]) {
        String meldung;
        boolean schulfrei;
        int temperatur;

        try {                                                    // Von try und catch wird die
            temperatur = Integer.parseInt (argv[0]);          //Umwandlung des Parameters
                                                    // vom Typ String in den Basistyp integer
                                              // umhuellt, um Fehler bei der Eingabe abzufangen.
            schulfrei = temperatur > 39 | 15 >= temperatur;                    //Bedingung
            if (schulfrei)                                                //bedingte Anweisung
                meldung = "ihr duerft nach Hause gehen";
            else
                meldung = "halt, hiergeblieben!";

            System.out.println (meldung);
        }
        catch (Exception e) {
            System.out.println ("Ungueltige Temperaturangabe");
        }
    }
}
```

Man kann nun mit `java SchulfreiMeldung 40` die schöne Aufforderung auf dem Bildschirm sehen, nach Hause gehen zu dürfen. Die Bedingung ist vor der bedingten Anweisung erfolgt und ihr Wert ist in der Variablen *schulfrei* vom Typ `boolean` gespeichert. Die Fehlerbehandlung, die bei Benutzereingaben immer angemessen ist, sehen wir noch später (Abschnitt 3.10). Wenn man will, ist dies auch eine Verzweigung: im Falle einer ungültigen Eingabe wird eine Fehlermeldung ausgegeben. Vielleicht ist es auch ganz interessant, einmal in der **main**-Methode die Argumente verwendet zu sehen. Die Parameter müssen vom Typ **String** sein und werden in einem Feld (`argv[]`, s. Abschnitt 3.6) untergebracht. Da 40 kein String ist, wird eine Methode zum Überführen einer Zahl in einen String angewandt,`Integer.parseInt (argv[0])`. `Integer` ist eine Klasse, die zum Zwecke der Ein- und Ausgabe einmal so tut, als wären Zahlen Objekte.

Schließlich gibt es die Schlüsselwörter *switch* und *case*. *switch* greift eine Variable heraus, deren Werte die Verzweigungen des Programms angeben. Werte der Variablen werden mit *case:* angegeben. Es folgt, was zu tun ist. Ein einfaches Beispiel ist das folgende.

```
import AlgoTools.IO;

class TageProMonat {

    public static void main (String argv[]) {

        int monat = IO.readInt ("Bitte Monatsnummer [1..12] eingeben: ");
        int jahr = IO.readInt ("Bitte Jahreszahl (vierstellig) eingeben: ");
        int tage = 0;
```

```
        boolean fehler = false;

        switch (monat)  {
                case 1:                                                 // Wenn Januar,
                case 3:                                                         // Maerz,
                case 5:                                                           // Mai,
                case 7:                                                           // Juli,
                case 8:                                                         // August,
                case 10:                                                        // Oktober,
                case 12:                                                      // Dezember,
                        tage = 31;                                      // dann 31 Tage.
                        break ;
                case 4:                                                       // Wenn April,
                case 6:                                                             // Juni,
                case 9:                                                     // September,
                case 11:                                                     // November,
                        tage = 30;                                      // dann 30 Tage.
                        break ;
                case 2:                                 // Spezialfall: Februar mit Schaltjahren
                        if ( ((jahr % 4 == 0) && !(jahr % 100 == 0)) || (jahr % 400 == 0) )
                        tage = 29;                                  // Schaltjahr, dann 29 Tage
                        else
                        tage = 28;                          // Kein Schaltjahr, dann 28 Tage
                        break ;
                default:                            // Monatsnummer nicht im Interval [1..12]
                        System.out.println ("Kein gueltiger Monat!");
                        fehler=true;
                        break ;
         }

        if (!fehler)  {
                System.out.println ("Dieser Monat hat "+tage+"Tage.");
        }
    }
}
```

3.6 Felder

Mehrere Daten desselben Datentyps können zu einem Feld (engl.: array) zusammengefaßt werden. Die Felder sind der Reihe nach nummeriert, beginnend bei 0. Ein Feld wird deklariert durch den Datentyp seiner Elemente und eckige Klammern.

```
int[] feldInt;
char[] feldChar;
boolean[] feldBoolean;
```

Die Länge eines Feldes wird bei der Konstruktion eines neuen Feldes durch eine Zahl in den eckigen Klammern angegeben.

```
feldInt = new int[8];
```

Hier wird ein Feld von 8 Elementen, die alle vom Typ `int` sind, erzeugt.

Um auf ein Element eines Feldes zuzugreifen, gibt man die Position des Elementes in den eckigen Klammern an.

```
feldInt[2] = 6;
 feldBoolean[0] = regnet | !regnet;
```

3.7 Abstrakte Klassen, Schnittstellen

Nehmen wir an, wir wollten – was andere schon längst getan haben – einige Klassen deklarieren, die geometrische Figuren behandeln können. Eine Klasse **Kreis** hätte einen Radius und eine Position und könnte seinen Umfang und seine Fläche angeben. Eine Klasse **Viereck** hätte zwei Kantenlängen und könnte seinen Umfang und seine Fläche angeben. Eine Klasse **Dreieck** hätte Kantenlängen und Winkel und könnte seinen Umfang und seine Fläche angeben. Wir sehen, daß wir ständig eine Methode zum Umfangberechnen und eine zur Flächenberechnung benötigen. Es liegt also nahe, eine Oberklasse **Form** einzuführen, die diese beiden Methoden festlegt. Leider geht das nicht, da jede Form ein anderes Berechnungsverfahren braucht (was soll π beim Viereck?). Trotzdem macht es große Programmpakete übersichtlicher, wenn wir bei einer Oberklasse wissen, daß alle Unterklassen bestimmte Methoden haben und welche Parameter diese haben. Deshalb gibt es abstrakte Klassen und Methoden in JAVA. Wenn Sie sich Pakete wie z.B. `java.util` ansehen, finden Sie darin viele abstrakte Klassen, z.B. für Kalender oder Wörterbücher. Die Beschreibung der Klassen besteht darin, daß abstrakte Methoden angegeben werden. Das sind Methoden mit Namen und Parametern, aber ohne einen Rumpf. Die abstrakte Klasse sagt uns, was Unterklassen können sollen und legt Bezeichnungen fest. Eine abstrakte Klasse oder Methode wird durch das Schlüsselwort *abstract* als solche ausgewiesen.

Abstrakte Klassen: Klassen, die keine Objekte haben (keine Instanzen erzeugen) und vielleicht eine abstrakte Methode, d.h. eine Methode ohne Rumpf.

- Jede Klasse mit einer abstrakten Methode ist selbst abstrakt und muß auch als Modifikator das Schlüsselwort *abstract* haben.
- Man kann Klassen als *abstract* deklarieren, ohne daß sie eine abstrakte Methode haben.
- Sollte man versuchen, ein Objekt einer abstrakten Klasse zu konstruieren, gibt es eine Fehlermeldung.
- Eine Unterklasse einer abstrakten Klasse ist selbst abstrakt, wenn sie nicht alle Methoden der abstrakten Klasse implementiert.
- Eine Unterklasse einer abstrakten Klasse, die jede Methode der abstrakten Klasse vollständig (also: mit Rumpf) definiert, kann Objekte haben. Dies ist der eigentliche Sinn einer abstrakten (Ober-)Klasse. Natürlich kann die Unterklasse auch noch zusätzliche Methoden haben.

Als wir beim Ball-Beispiel sagten, daß alle JAVA-Klassen eine Methode **toString()** haben sollten, die für ein Objekt der Klasse eine Zeichenkette anfertigt, haben wir auf eine Methode der Klasse **Object** verwiesen. Die Methode ist tatsächlich realisiert, d.h. sie hat einen Rumpf. Jede Unterklasse von **Object**, also jede Klasse in JAVA, kann diese Methode einfach übernehmen, oder für sich neu definieren. Die JAVA-Entwickler brauchten keine abstrakte Klasse, die **toString()** als abstrakte Methode hat, weil sie einen Rumpf für **toString()** schreiben konnten.

Wenn wir uns ein realistisch großes Projekt zur Entwicklung von Programmen vorstellen, dann sehen wir die Schwierigkeit, in einer abstrakten Klasse alle "Pflichten" zu notieren. Die Gruppe, die die graphische Oberfläche für die Benutzer schreibt, betrachtet die Darstellungsmöglichkeiten von Objekten. Sie möchte nicht nur **toString()** vorgeben, sondern auch abstrakte Methoden wie z.B. **fillColor(Color c)**, **draw(Drawwindow dw)**, **setPosition(double x, double y)** und dergleichen. Hingegen ist die Gruppe, die die Buchhaltungsprogramme schreibt, nicht an der graphischen Darstellung der Geschäftsbilanz, sondern an der Vollständigkeit der Angaben eines Vorgangs interessiert. Die Marketing-Gruppe, die den Versand von Werbematerial an potentielle Kunden unterstützt, hat wieder eine andere Sicht auf die Daten. Würden wir nun vorhaben, alles in eine (abstrakte) Klasse zu stopfen, hätten wir die Vorteile objektorientierter Programmierung aufgegeben und einen monolithischen Block geschaffen, der schwierig zu ändern ist. Warum sollte ein Objekt der Klasse **Kunde** nicht von allen Aspekten her gesehen werden? Als jemand, dessen Daten in bestimmter Weise auf dem Bildschirm angezeigt und geändert werden, als jemand, der seine Rechnung per Kreditkarte bezahlt hat, als jemand, der bereits die Ankündigung des Weihnachtssonderangebots erhalten hat? Der Grund ist einfach: Mehrfachvererbung gibt es in JAVA nicht! [10] Realistischerweise glaubt man nicht, daß die EntwicklerInnen alle Implikationen des logischen *und* bedenken. Wenn ein Objekt zu mehreren Klassen gehört, dann hat es die Eigenschaften der einen *und* der nächsten *und... und* der nächsten Klasse. Vielleicht widersprechen sich einige Eigenschaften? JAVA bietet einen anderen Ausweg an: die Schnittstelle (engl.: *interface*).

Schnittstelle: Eine Klasse, die mit dem Schlüsselwort *interface* anstelle von *class* ausgezeichnet ist.

- Eine Schnittstelle ist eine Klasse, die ausschließlich abstrakte Methoden hat.
- Eine Schnittstelle kann keine Objekte haben.
- Eine Schnittstelle kann von anderen Schnittstellen abstrakte Methoden erben. Das Schlüsselwort ist wie bei Klassen *extends*. Dieses Schlüsselwort hat eine eindeutige Semantik: es bedeutet die Vererbung im Sinne von **A ist ein B**.
- Eine Klasse kann Unterklasse von mehreren Schnittstellen sein. Dies wird durch das Schlüsselwort *implements* angegeben. Dann implementiert sie die abstrakten Methoden all dieser Schnittstellen. Die mehrfache Schnittstellenvererbung unterstützt die Modellierung nach verschiedenen Aspekten, ohne daß tatsächlich Programmcode vererbt wird. Das Schlüsselwort *implements* hat die Bedeutung **A implementiert B**, so daß auch diese Vererbung in JAVA vorkommt.

[10] Eigentlich muß es *Mehrfacherbung* heißen.

Eine Variable kann als Typ eine Schnittstelle haben. Das heißt, daß sie als Wert ein Objekt haben kann, das zu einer Klasse gehört, die mit *implements* als Realisierung dieser Schnittstelle deklariert wurde.

Ein einfaches Beispiel, das die Schnittstellen illustriert, seien irgendwelche Einstellungen von Bildschirmen. Die Schnittstelle für Farbsysteme sorgt für die Einstellung von Farben oder schwarz-weißer Darstellung. Die Schnittstelle für Bedienelemente sorgt für die Justierung von Helligkeit und Kontrast. In den Schnittstellen sind die Variablen lediglich Konstante, d.h. sie müssen einen Wert haben und dieser kann in der Schnittstelle nicht verändert werden. Die Methoden haben nur Modifikatoren und einen Namen – nach den Klammern für die Parameterliste (hier: leer) kommt schon das Anweisungsende.

```
interface Farbe {
        int SchwarzWeiss=0, Bunt=1;
        public void faerbe ();
}

interface Bedienelemente {
        int Hell=3, Kontrast=3;
        public void einstellen ();

}
```

Fernseher und Rechnermonitor implementieren beide Schnittstellen. Sie mischen also die Farbgebung und die Einstellung zusammen, als würden sie sowohl von **Farbe** als auch von **Bedienelemente** erben. Es handelt sich aber nicht um eine Mehrfachvererbung, denn sie erhalten keine tatsächlichen Eigenschaften oder Handlungen von den Schnittstellen. Die abstrakten Methoden müssen in den Klassen **Fernseher** und **Rechnermonitor** implementiert werden, indem für die Methodenbezeichnungen tatsächliche (in diesem Beispiel sehr reduzierte) Handlungen angegeben werden. Dabei sind die Handlungen verschieden. Zum selben Bezeichner (**faerbe**, **einstellen**) werden im Methodenrumpf jeweils unterschiedliche Handlungen angegeben (hier: den Eigenschaften Default, Helligkeit und Kontrast verschiedene Werte zugewiesen). Der Aufruf `java InterfaceTest` liefert für ein Objekt der Klasse **Fernseher** und ein Objekt der Klasse **Rechnermonitor** die Einstellungen der Farbe und Bedienelemente.

```
class Fernseher implements Farbe, Bedienelemente {
        int Default,Hell,Kontrast;
        public void faerbe () {
                Default=Bunt;
        }

        public void einstellen () {
                Hell=1;
                Kontrast=1;

        }
        public void drucke () {
```

```
            System.out.println ("TV-Farbe: "+Default);
            System.out.println ("TV-Helligkeit: "+Hell);
            System.out.println ("TV-Kontrast: "+Kontrast);
        }
}

class Rechnermonitor implements Farbe, Bedienelemente {
        int Default,Hell,Kontrast;

        public void faerbe () {
            Default=SchwarzWeiss;
        }
        public void einstellen () {
            Hell=2;
            Kontrast=2;
        }
        public void drucke () {
            System.out.println ("Monitor-Farbe: "+Default);
            System.out.println ("Monitor-Helligkeit: "+Hell);
            System.out.println ("Monitor-Kontrast: "+Kontrast);
        }
}

class InterfaceTest  {

        public static void main (String[] argv) {
            Fernseher tv=new Fernseher ();
            tv.faerbe ();
            tv.einstellen ();
            tv.drucke ();
            Rechnermonitor monitor=new Rechnermonitor ();
            monitor.faerbe ();
            monitor.einstellen ();
            monitor.drucke ();
        }
}
```

3.8 Sichtbarkeit

Die große Menge von JAVA-Klassen, die weltweit zur Vefügung steht, muß organisiert werden, damit

- der JAVA-Übersetzer die Deklarationen findet, die von dem Programm verwendet werden, das er gerade bearbeitet,
- der Zugriff auf Klassen, Methoden und Variablen auch verboten werden kann (so daß nicht jeder meinen Kontostand erfährt, wenn die Kontoführung in JAVA realisiert ist),
- Namenskonflikte vermieden werden.

In diesem Abschnitt sollen die wichtigsten Konzepte zu diesen Punkten vorgestellt werden.

3.8.1 Pakete und Sichtbarkeit

Damit der JAVA-Übersetzer die Klassen und ihre Methoden findet, die von dem Programm, das er gerade übersetzt, verwendet werden, muß es klare Richtlinien geben, wo nach Namen von Klassen, Methoden und Variablen zu suchen ist. Dazu gibt es in JAVA die *Übersetzungseinheit*, die aus mindestens einer der folgenden Deklarationen besteht:

die Paketdeklaration , die einen Namen für eine Menge von Klassendeklarationen festlegt,

die Importdeklaration , die Deklarationen aus einem anderen Paket verfügbar macht, und

die Klassen- und Schnittstellendeklarationen , die Klassen oder Schnittstellen angibt.

Die im Paket `java.lang` festgelegten Deklarationen von Klassen mit ihren Methoden gelten für jeden JAVA-Code. Andere Deklarationen müssen importiert werden, damit sie von einem JAVA-Programm aus erreichbar sind. So haben wir im Ballbeispiel mit *import* `java.util.*` alle Klassen und Methoden des Pakets `java.util` erreichbar gemacht.

- Deklarationen des Pakets `java.lang` gelten in jedem JAVA-Code.
- Importierte Deklarationen gelten in der Übersetzungseinheit, in der die *import*-Anweisung steht.

Im allgemeinen ist jede Klassendeklaration eine Datei mit dem Namen *Klassennamen*`.java`. Manchmal enthält eine Datei mehrere Klassen, davon (höchstens) eine mit einer *main*-Methode. So eine Datei ist ein unbenanntes Paket. Zu einem Zeitpunkt soll es nur ein unbenanntes Paket geben. Der Übersetzer verbindet dieses unbenannte Paket mit dem aktuellen Arbeitsverzeichnis. Wenn nun in diesem Verzeichnis auch noch benannte Pakete existieren, so kann eine Klasse des unbenannten Pakets auch von den benannten Paketen verwendet werden. Dies ist abhängig von der Plattform (Rechner und Betriebssystem, die die virtuelle JAVA-Maschine realisieren). Dies kann zu unschönen Effekten führen: Sie verschieben das unbenannte Paket in ein anderes Verzeichnis und plötzlich erhalten Sie andere Ergebnisse bei Ihren benannten Paketen! Zum Glück können wir angeben, zu welchem Paket ein Programm gehören soll. Wenn wir zu einer Menge von Klassendeklarationen eine Paketdeklaration schreiben, so gehören diese Klassen und ihre Methoden zu dem angegebenen Paket. Die Paketdeklaration besteht aus dem Schlüsselwort *package* und einem Paketnamen. So haben wir in unserem Ballbeispiel als erste Zeile einen Paketnamen festgelegt:

```
package ballbeispiel;
import AlgoTools.IO;
class Mensch {
        ...
}
...
```

- Die Dateien heißen wie die Klassen, deren Deklaration in der Datei abgelegt ist, z.B. heißt die Datei mit der *main* enthaltenden Klasse **BallBeispiel** `BallBeispiel.java`.

- Das Paket heißt wie das Verzeichnis, in dem das Programm mit der Deklaration *package* liegt. Die Verzeichnisse, die international zur Verfügung stehen sollen, werden in Anlehnung an die URL (eindeutige Kennung für Rechnerbereiche) formuliert. Während die URL an letzter Stelle den Staat bezeichnet (`de` für Deutschland) und an erster die speziellste Angabe, ist die JAVA-Konvention, daß Pakete in einem Unterverzeichnis mit dem Namen der speziellsten Angabe abgelegt werden. Unsere JAVA-Pakete müßten dem entsprechend in einem Verzeichnis `/de/unido/cs/ls8/` liegen. Allerdings haben wir die leichte Erreichbarkeit unserer Programme durch die Studierenden vor die internationale Konvention gestellt und haben sie unter `~gvpr000/ProgVorlesung/Packages/` abgelegt.

- Der Aufruf eines Programms aus einem Paket erfolgt mit dem Pfad ab dem aktuellen Arbeitspfad bzw. mit dem Pfad ab dem Endpunkt der Pfade, die in der Rechnervariable `CLASSPATH` gespeichert sind. Ist das aktuelle Verzeichnis `~gvpr000/ProgVorlesung/Packages`, so erfolgt der Aufruf des Programms `BallBeispiel` im Paket `ballbeispiel` mit

  ```
  java ballbeispiel.BallBeispiel
  ```

Pakete können Unterpakete haben. So hat das Standardpaket `java` die Unterpakete `awt, applet`, `io, lang`, `net, util`. Diese Unterpakete enthalten erst die Klassen- und Schnittstellendeklarationen, nicht das Paket `java`. Die Hierarchie der Pakete wird so verwendet, daß

- der vollständige Name beim Namen des obersten Pakets beginnt, an den mit Punkt getrennt der Name des Unterpakets gehängt wird und so fort (Beispiel: `java.awt.image`);

- die Deklarationen der Unterpakete von einem Paket aus sichtbar sind, d.h. ein Paket umfaßt seine Unterpakete.

Ein Klassen- oder Schnittstellenname ist in dem Paket bekannt, in dem er eingeführt wurde. Genauer:

- Eine Klasse oder Schnittstelle ist bekannt in allen Übersetzungseinheiten des Pakets, in dem sie deklariert wurde.

Jetzt wissen wir, wo der Übersetzer nach dem Code für eine Klasse sucht, wenn er gerade eine Einheit übersetzt: im Paket `java.lang`, in importierten Paketen und in allen Übersetzungseinheiten, die zu demselben Paket gehören wie die gerade zu übersetzende Einheit.

3.8.2 Zugriffskontrolle

Der Zugriff bzw. das Verbergen von Klassen und Eigenschaften geschieht über die Modifikatoren, die bisher nur am Rande erläutert wurden. Das Schlüsselwort *public* ist schon verschiedentlich vorgekommen. Wenn eine Klasse oder Schnittstelle *public* ist, so kann jeder Code, der Zugriff auf das Paket hat, in dem die Klasse oder Schnittstelle deklariert

wurde, auch auf die Klasse zugreifen. Dies gilt weltweit – man sollte also nicht ganz so großzügig mit diesem Schlüsselwort umgehen, wie wir es bisher getan haben.

Ist die Klasse oder Schnittstelle nicht mit dem Schlüsselwort *public* modifiziert, so kann auf sie nur von innerhalb des Pakets, in dem sie deklariert ist, zugegriffen werden.

Generell gilt:

- Eine Variable kann nur verwendet werden, wenn ihr Typ (die Klasse, die ihren Wertebereich angibt) zugreifbar ist und sie selbst zugreifbar ist.
- Eine Methode kann nur verwendet werden, wenn sie selbst zugreifbar ist und die Klasse, für deren Objekte die Methode Handlungen bereitstellt.
- Ebenso kann ein Konstruktor nur verwendet werden, wenn er selbst und die Klasse, für die er Objekte erzeugt, zugreifbar ist.

Ist die Variable, die Methode oder die Konstruktormethode als *public* angegeben, so ist sie zugreifbar. Ist sie gar nicht modifiziert, so ist sie von dem Paket aus zugreifbar, in dem die betreffende Klasse deklariert ist.

Wird eine Eigenschaft oder eine Konstruktionsmethode mit *private* modifiziert, so kann auf sie nur von innerhalb der Klasse, in der sie deklariert sind, zugegriffen werden. Es handelt sich dann um eine Variable bzw. Methode, die nur für eine Klasse reserviert ist. Insbesondere werden mit *private* modifizierte Variablen oder Konstruktionsmethoden nicht vererbt.

Der Modifikator *protected* verbietet den weltweiten Zugriff. Innerhalb des Paketes, in dem die Eigenschaft oder die Konstruktionsmethode mit dem Modifikator *protected* deklariert wurde, darf auf die Eigenschaft oder den Konstruktor zugegriffen werden. Nehmen wir an, in der Klasse **K** mit den beiden Unterklassen **K1** und **K2** wäre die Variable k als *protected* eingeführt worden. Die Klasse **K2** sei außerhalb des Paketes, in dem **K** und **K1** stehen, deklariert. Von außerhalb des Paketes darf nur aus **K2** heraus auf k zugegriffen werden. Ein Konstruktor kann nicht einmal von einer Unterklasse in einem anderen Paket aufgerufen werden.

3.8.3 Das Konturmodell

Die Sichtbarkeit von Variablen ist gar nicht so einfach. Erinnern wir uns: Eine Variable kann

- eine Klasseneigenschaft – geschrieben mit dem Schlüsselwort *static*,
- eine Objekteigenschaft – deklariert am Anfang von *ClassBody* (ohne *static*),
- ein Unikat – eine Variable von einem einfachen Datentyp,
- eine Hilfsgröße, die wir gerade mal (z.B. in einer Methode oder in einer Schleife) benötigen

ausdrücken. Eine Klasseneigenschaft ist überall sichtbar, wo die Klasse sichtbar ist. Eine Objekteigenschaft ist ebenfalls überall sichtbar, wo die Klasse sichtbar ist, deren Objekte diese Eigenschaft haben. Ob die Klasse sichtbar (zugreifbar) ist, ergibt sich daraus, in welchem Paket und mit welchem (oder keinem) Modifikator sie deklariert wurde. Das haben wir gerade gesehen.

Die Hilfgrößen werden *lokale Variable* genannt. In unserem Ballbeispiel waren z.B. *dx*, *dy*, *geschenk*, *geschenkN*, *besitz* lokale Variablen. Sie gelten nur innerhalb des Blocks, in dem sie stehen. Eine *for*-Anweisung wird wie ein Block behandelt. Ansonsten wird der Block, in dem eine lokale Variable deklariert ist, angegeben durch die nächsten geschweiften Klammern, die sie umgeben. Die nächsten Klammern ermittelt man so: von der Variable gehen Sie mit dem Finger solange nach links, bis Sie auf eine öffnende geschweifte Klammer treffen. Diese und die passende schließende Klammer umfassen den Geltungsbereich der lokalen Variablen. Innerhalb dieses Geltungsbereichs darf der Name der lokalen Variablen nicht noch einmal auftreten. Beispielsweise darf in dem Block, in dem die lokale Variable deklariert ist, nicht noch eine *for*-Schleife mit einer Variable gleichen Namens vorkommen. Außerhalb des Blocks, in dem die lokale Variable steht, darf der Name doch vorkommen. Man darf dann nur nicht glauben, daß dieselbe Variable damit gemeint sei![11] Im inneren Block ist nur die lokale Variable sichtbar, sie *verdeckt* die gleichnamige Variable außerhalb des Blocks. Will man aber die gleichnamige Variable von außerhalb verwenden, schreibt man *this.* davor.

Um nun die Sichtbarkeit von Variablen in verschiedenen Blöcken einer Klassendeklaration deutlich zu machen, gibt es das *Konturmodell* (engl. box model). Es werden Geltungsbereiche von Variablen, ihre Sichtbarkeit, durch Konturen (Schachteln) gezeichnet. Eine Schachtel gibt die Sichtbarkeit der Variablen an, die in ihr sind. Das bedeutet, daß die Variablen der äußeren Schachtel in allen inneren Schachteln sichtbar sind. Fatal ist diese Interpretation bei den lokalen Variablen, die denselben Namen haben wie Variablen in einer umgebenden Schachtel. Ohne *this* davor, ist es nicht die sichtbare Variable aus der äußeren Schachtel!

Das folgende Beispiel soll die Sichtbarkeit mit einer Klasse, ihrer Unterklasse und einer *for*-Schleife verdeutlichen. Die Klasse ist der bereits bekannte **Mensch** aus dem Ballbeispiel. Wir importieren das Paket *ballbeispiel*. Die Unterklasse ist **Studierend**. Sie erbt von **Mensch** die Eigenschaften *name*, *geschlecht* und *hausrat*. Sie erweitert den Katalog von Eigenschafte aber um *semester*, *monat* und *jahr*. Ein Objekt der Klasse **Studierend** hat nun sechs Eigenschaften, die natürlich immer dort sichtbar sind, wo das Objekt sichtbar ist. In diesem Beispiel also überall im Programm `Studi.java`. In der Methode **studieren**, die hier einfach nur das Vergehen der Monate, Semester und Jahre beschreibt und nach dem 9. Semester ein Diplom ausgibt, haben wir eine *for*-Schleife mit der lokalen Variable *i*. Probieren Sie einmal aus, was passiert, wenn Sie statt *i* den Variablennamen *monat* verwenden! So, wie das Beispiel hier steht, läßt sich die Sichtbarkeit der Variablen gut im Konturmodell darstellen: *name*, *geschlecht* und *hausrat* sind überall sichtbar; *semester*, *monat*, und *jahr* sind der Klasse **Mensch** nicht bekannt, aber in **Studierend** und **Studi** sichtbar. In der Schleife sind sie sichtbar, zur Sicherheit aber mit *this* deutlich als Eigenschaft eines bestimmten Objektes der Klasse **Studierend** gekennzeichnet, auf das der Variablenname *stud* referenziert. *i* ist nur innerhalb der Schleife sichtbar.

[11] Aus diesem Grunde empfehle ich, die lokalen Variablen mit "_" beginnen zu lassen.

```
import ballbeispiel.*; import AlgoTools.IO;                                    // 1

class Mensch  {  String name, geschlecht; Hausrat hausrat;  } ;                // 2
  class Studierend extends Mensch {                                            // 3
        int semester,monat,jahr;                                               // 4

        public Studierend () {                                                 // 5
              super ();                                                        // 6
              semester = IO.readInt ("Im wievielten Semester ist Stud? ");     // 7
              monat = IO.readInt ("Der wievielte Monat des Jahres ist jetzt?"); // 8
              jahr = IO.readInt ("In welchem Jahr? ");                         // 9
         }                                                                     // 10
        public void studieren () {                                             // 11
              for (int i=this.monat; 13>i; i++) {          //Monate zaehlen    // 12
                    if ((i!=this.monat) && (i==4 | i==10)) {                   // 13
                                                  //Semester zaehlen           // 14
                          this .semester++;                                    // 15
                          System.out.println (this .name+"ist "+this.jahr      // 16
                          +"im "+semester+". Semester");   }                   // 17
               }                                                               // 18
              this .jahr++;                                  //Jahre zaehlen  // 19
              this .monat=1;                                                   // 20
              if (9>semester)                  //Studienende noch nicht erreicht? // 21
                    studieren ();                          //dann weiterstudieren // 22
              else System.out.println ("Und jetzt das Diplom!");   //sonst Diplom // 23
         }                                                                     // 24
  }                                                                            // 25

  class Studi  {                                                               // 26
        private static void main (String argv[])  {                            // 27
              Studierend stud;                                                 // 28
              stud = new Studierend ();                                        // 29
              stud.studieren ();                                               // 30
              System.out.println (stud.name+"bekommt das Diplom "+stud.jahr);  // 31
        }
  }
```

3.9 Eingebettete Klassen

Wahrscheinlich sind Sie nun sattelfest genug, um eine kleine und nicht so sehr häufig vorkommende Komplikation zu überstehen: die eingebetteten Klassen. Klassendeklarationen haben bisher nur Eigenschaften und Methoden für ihre Objekte festgelegt. Für jede Eigenschaft wurde bei der Deklaration ein Typ angegeben. Die Eigenschaften können als Ausprägungen Objekte der angegebenen Klasse annehmen. Diese Klasse, die den Typ angibt, gibt es unabhängig von der Klasse, deren Eigenschaft nur Objekte dieses Typs annehmen kann. Eine eingebettete Klasse ist nun ausschließlich dazu da, den Wertebereich einer Eigenschaft, die die Objekte der sie umgebenden Klasse haben, darzustellen. Sie wird in der Klassendeklaration der sie umgebenden Klasse die den Deklarationen der Eigenschaften deklariert. Es gibt vier Arten eingebetteter Klassen:

- Wenn die eingebettete Klasse den Typ einer Klasseneigenschaft darstellen soll, muß sie auch mit *static* modifiziert sein.
- Wenn sie den Typ einer Objekteigenschaft darstellt, hat sie natürlich nicht den Modifikator *static*. Ihr Sinn ist, daß sie auch auf *private* Eigenschaften und Methoden der umgebenden Klasse zugreifen kann. Jedes Objekt der eingebetteten Klasse ist mit einem Objekt der einbettenden Klasse assoziiert.
- Eine *lokale Klasse* ist in einem Block deklariert und nur dort sichtbar. Sie verhält sich wie eine lokale Variable.
- Eine *anonyme Klasse* ist wie eine lokale, nur daß sie keinen Namen hat. Statt erste eine lokale Klasse zu deklarieren und sie dann zu instanziieren, wird bei der anonymen Klasse beides in einem Schritt gemacht. Das bedeutet, daß ihre Deklaration in einer Zuweisung oder in einem Methodenaufruf als Parameter vorkommen darf. Ihre Deklaration hat die Form eines Konstruktors. Es gibt also keine Möglichkeit, *extends* zu verwenden – eine anonyme Klasse ist immer eine Unterklasse von **Object** und wird vom JAVA-System intern mit dem Klassennamen der umgebenden Klasse, dem Zeichen $ und einer Zahl benannt.

Beispiele für eingebettete Klassen folgen in anderen Abschnitten, z.B. 4.5. In [Flanagan, 1998] finden sich viele Beispiele im Kapitel 5.

3.10 Fehlerbehandlung

Die wörtlich gemeinte Fehlerbehandlung muß natürlich die Prorgammiererin selbst vornehmen, indem sie das Programm so lange ändert bis der Fehler nicht mehr auftritt. Mit "Fehlerbehandlung" wird aber auch die bereits im Programm vorbereitete Behandlung von Situationen bezeichnet, in denen etwas schief ging. Bei dem Beispiel zu Kontrollstrukturen mußte ich bereits so eine Fehlerbehandlung einführen, weil bei Eingaben von Benutzern sehr leicht etwas schief gehen kann und ich dies nicht durch Ändern des Programms verhindern kann.

Für Fehlerbehandlungen stellt JAVA die folgenden Anweisungen zur Verfügung: [12]

try{ } Der durch gescheifte Klammern gegebene Block ist der, in dem etwas Unvorhergesehenes passieren kann. Vielleicht erzeugt das JAVA-System zur Laufzeit des Programms ein Objekt einer Unterklasse von **Exception** oder von **Error**. Auch wenn der Block über *break*, *continue* oder *return* verlassen wird, erzeugt JAVA ein Objekt einer Unterklasse von **Exception**.

catch(*SomeException e*)**{ }** behandelt das Fehlerobjekt, das in dem nächsthöheren Block oder dem nächst zurückliegenden Aufruf erzeugt wurde.

throw löst einen Fehler eines angegebenen Typs aus. Dabei ist darauf zu achten, daß bei der Methode, die den Fehler auslösen kann, nach den Parametern und vor dem Rumpf das Schlüsselwort *throws* und der Fehlertyp steht.

[12] In LISP, der Programmiersprache, die in der Künstlichen Intelligenz schon vor etwa 40 Jahren entwickelt wurde, gab es bereits *catch* und *throw*. Das Grundkonzept der Fehlerbehandlung ist also mindestens 40 Jahre alt! Sie können davon ausgehen, daß das Konzept der Fehlerbehandlung Ihnen auch unabhängig von JAVA immer wieder begegnen wird.

```
public void methode() throws MeineException {
        ...
        throw new MeineException("Mein Ausnahmefall ist eingetreten! ");
        ...
}
```

finally{ } Der Code in dem auf *finally* folgenden Block wird immer ausgeführt, nachdem der *try*-Block verlassen wurde – egal ob der Fehlerfall aufgetreten ist oder nicht.

Alle Fehlertypen sind Unter(unter...)klassen von `java.lang.Throwable`. Sie haben immer eine Eigenschaft vom Typ **String**, die Fehlermeldungen enthält, z.B. den Text, den der Benutzer im Fehlerfalle auf dem Schirm sieht. **Throwable** hat die Unterklassen **Error** und **Exception**. Eine viel verwendete Unterklasse von **Exception** ist **ArrayAccessOutOfBounds**, die den Zugriff auf das $a.length() + k$te Element des Feldes a anzeigt.

```
public class throwtest {
    public static void main (String argv[]) {
        int i;
        try  i=Integer.parseInt(args[0]);          //1. Element soll in int umgewandelt werden
        catch(ArrayIndexOutOfBoundsException e) {          //1.Element gibt s nicht
            System.out.println("Feldelement nicht vorhanden! "); return;
        }
        i++;
        finally {
            System.out.println("gruesse ich hiermit alle meine Freunde ");
        }
    }
}
```

3.11 Was wissen Sie jetzt?

Sie können nun in JAVA programmieren. Sie wissen, daß Klassen in zweierlei Hinsicht genutzt werden: erstens beschreiben sie Objekte, die der eigentliche Gegenstand der Modellierung sind; zweitens werden ihre Objekte als als mögliche Werte von Variablen (Typen) genutzt. Manchmal sind Klassen auch einfach Merkzettel für die Methoden, die jede Unterklasse irgendwie realisieren soll. Dies sind dann abstrakte Klassen oder Schnittstellen.

Variablen (Eigenschaften) realisieren die Assoziationen, die bei der objektorientierten Modellierung eines Problems festgelegt wurden. Sie wissen, wie Variable ihren Wert bekommen und wie Werte an Methoden weitergereicht werden. Referenzzuweisung und Referenzübergabe auf der einen Seite und Wertzuweisung und Wertübergabe auf der anderen Seite sind Ihnen völlig klar. Das Konturmodell für die Sichtbarkeit von Variablen zeigt, wo Variablen verwendet werden können und wo sie unbekannt sind. Sie wissen, was ein Programmzustand ist. Vielleicht schreiben Sie sich ein kleines Programm und drucken nach jeder Wertzuweisung den Wert der Variablen aus. Vielleicht wollen Sie es etwas gröber betrachten und drucken nach Abarbeiten einer Methode oder eines Blocks den Wert der Variablen aus. Sie sehen so Programmzustände in unterschiedlicher Feinheit.

Methoden versenden und empfangen Botschaften. Sie führen Handlungen aus und verändern so den Programmzustand. Die *static* Methode **main** ist das eigentliche Programm.

Sie haben Schleifen und Bedingungen gesehen. Felder sind Ihnen vielleicht noch etwas abstrakt geblieben. Das macht nichts, denn ein ausführliches Beispiel folgt im nächsten Abschnitt.

4 Sequenzen und Sortierung

Nachdem die Grundzüge von JAVA bekannt sind, können wir uns der Programmierung mit ihren drei Fragen zuwenden: was, wie, warum? Aus der Einleitung wissen wir schon, daß es Standardmodelle in der Informatik gibt, für die jede Programmiersprache Realisierungen anbietet. Diese Standardmodelle heißen abstrakte Datentypen. Wir lernen einige kennen und sehen, wie man sie selbst in JAVA realisieren kann. In der JAVA-Bibliothek `java.util` sind sie professionell und umfangreich realisiert. Wenn Sie die "Lehreversion" verstanden haben, ist die JAVA-Bibliothek leicht zu lesen. Bleibt nur die Frage: warum? Wir lernen die drei wichtigsten Verfahren kennen, diese Frage zu beantworten, nämlich den Induktionsbeweis, die Komplexitätsabschätzung und den Performanztest. Sie werden diese Themen im weiteren Studienverlauf noch gründlicher bearbeiten. In diesem Semester geht es nicht darum, beweisen zu lernen. Hier ist wichtig, den Zusammenhang von Programmierung und Aussagen über Programme zu begreifen. Sie sollen nicht *entweder* programmieren *oder* nachdenken (reflektieren), sondern immer beides als eine Einheit beherrschen.

4.1 Selektionssortierung

Da wir nun Felder und Kontrollstrukturen kennen, können wir an einem anspruchsvolleren Beispiel ihre Verwendung betrachten. [13] Dabei folgen wir den Programmierungsschritten *was*, *wie*, *warum*.

4.1.1 Ein Modell für das Sortieren

Wir wollen ein Feld so sortieren, daß das kleinste Element zuerst steht, dann das nächst größere, und so weiter, bis das größte Element am Ende des Feldes steht, d.h. wir sortieren so, daß ein höherer Index immer auch einen höheren Wert bezeichnet. Es ist also eine Sortierung, wie wir sie auch im Alltag ständig durchführen.

Definition 4.1: Sortierung Allgemein können wir das Sortieren definieren als einen Prozeß, der eine ungeordnete Menge in eine geordnete Menge überführt. Was wir dazu brauchen ist eine Ordnungsrelation, die uns für zwei Elemente der Menge entscheidet, ob sie den gleichen Rang haben oder das eine Element einen höheren Rang hat als das andere.

[13] Das Beispiel illustriert Felder und die *imperative*, also *nicht* objektorientierte Programmierung. Natürlich wäre es schöner, wenn Sie in der ganzen Vorlesung nur objektorientierte Programmierung sehen würden. Um dieses Beispiel so zu schreiben, bräuchten wir allerdings Schnittstellen. Eine Schnittstelle würde den Vergleich angeben. Für die Klasse von Objekten, die wir sortieren wollen, müssen wir dann die Schnittstelle implementieren. Wir erhalten ein Sortierprogramm, das beliebige Objekte sortieren kann. Die spezielle Implementierung hier hat aber zwei didaktische Vorteile: erstens ist der Schritt vom imperativen Programmieren zum Induktionsbeweis kleiner. Und da der Beweis ohnehin schon schwierig ist, halte ich es für besser, wenn zwischen Programm und Beweis nur ein ganz kleiner Spalt ist. Zweitens erlaubt die Sortierung von Zahlen die leichte Ausführung von Performanztests, etwas, was unbedingt gelehrt werden muß.

Das Schöne an Zahlen ist, daß sie eine Ordnung haben. Aber auch bei Buchstaben haben wir durch das Alphabet eine Ordnung. Bei Wörtern wenden wir diese Ordnung auf jeden Buchstaben nacheinander an, so daß bezüglich des i-ten Buchstabens noch gleichrangige Wörter bezüglich des $i+1$-ten Buchstabens einen unterschiedlichen Rang bekommen. Zahlen, Buchstaben und Wörter haben eine *totale Ordnung*: es gibt nicht zwei verschiedene gleichrangige Elemente, jedes Element ist verglichen mit jedem anderen Element entweder größer oder kleiner, aber nicht gleichrangig. Ordnen wir hingegen Aussagen bezüglich ihres Wahrheitswertes, so erhalten wir alle wahren Aussagen, die gleichrangig sind, und alle falschen Aussagen, die einen anderen Rang haben. Es gibt unendlich viele wahre Aussagen[14]. Die Ordnung bezüglich des Wahrheitswertes ist also eine *partielle Ordnung*, bei der mehrere Elemente gleichrangig sind.

Problemstellung: Sagen wir nun, wir wollen eine Sortierung herstellen mithilfe einer Ordnungrelation, die bezüglich der zu sortierenden Elemente total ist. Weil es am leichtesten ist, nehmen wir hier eine endliche Menge von Zahlen.

Nun überlegen wir uns ein Vorgehen. Wir haben einen schon sortierten Teil und einen unsortierten Teil der Menge. Am Anfang ist der sortierte Teil leer, am Ende ist der unsortierte Teil leer. Dazwischen sind beide Teile nicht leer: alle Positionen kleiner i sind sortiert. Wir wollen so vorgehen, daß wir niemals den bereits sortierten Teil wieder bearbeiten müssen. Also müssen wir im unsortierten Teil (i und aufwärts) das kleinste Element auswählen und an die i-te Position stellen. Wenn wir das geschafft haben sind i Positionen sortiert und wir betrachten nur noch die Positionen $i+1$ und aufwärts. Das machen wir, bis es keinen unsortierten Teil mehr gibt.

Jetzt müssen wir nur noch im unsortierten Teil das kleinste Element suchen. Dafür nehmen wir mal an, das erste Element des unsortierten Teils sei schon das kleinste. Wir nennen es k (für "kleinstes"). Dann sehen wir weiter. Ist das nächste Element größer, sind wir bestätigt und nehmen das nächste. Wenn wir ein kleineres Element als unser k finden, merken wir uns seine Position und sehen noch alle weiteren Elemente an, um festzustellen, ob es ein noch kleineres Element gibt. Das kleinste Element, das kleiner ist als unser k – nennen wir es einfach j – wählen wir aus. Wir vertauschen die Positionen von j und k. Dieses Vorgehen ist die Sortierung durch Auswählen, englisch *selection sort*.

Was wollen wir implementieren? Die Sortierung einer n-elementigen Menge durch Auswählen: anfangs gibt es nur einen unsortierten Teil, am Ende nur einen sortierten. Dawischen haben wir auf den Positionen 0 bis $i-1$ alles sortiert. Wir wählen aus den Positionen i bis n das kleinste Element, stellen es an die i-te Position und inkrementieren i um 1.

Dies Vorgehen hat die Eigenschaft:

- Zu jedem Zeitpunkt gibt es einen Teil, der schon fertig ist, sich nicht mehr verändern wird. Wenn man also schon vor Beendigung des Programms zuverlässige Angaben über immerhin einen Teil der Aufgabe braucht, ist die Verfahren geeignet.

[14] Sie können sich leicht eine Menge von unendlich vielen wahren Aussagen konstruieren: nehmen Sie einfach die Aussage "*0 ist kleiner als n*" und setzen Sie für n nacheinander alle natürlichen Zahlen ab 1 ein.

4.1.2 Realisierung in JAVA

Wir haben in der Problemstellung schon die Ordnungsrelation und die Elemente der zu sortierenden Menge festgelegt. Jetzt haben wir uns ein Modell der Problemlösung überlegt. Wenn wir dies Vorgehen programmieren wollen, dann fragen wir uns:

Wie sollen wir dies Modell der Problemlösung implementieren?

- Welchen Typ sollen die sortierten und unsortierten Teile haben?
- Wie soll die Erweiterung des sortierten gegenüber des unsortierten Teils erfolgen?
- Wie suche ich im unsortierten Teil nach dem kleinsten Element?
- Wie vertausche ich die Elemente?

Es gibt mehrere Möglichkeiten, diese Fragen gut zu beantworten. Wir wollten ja nun die Felder illustrieren und nehmen deshalb ein Feld von Zahlen als Typ, wobei der aktuelle Index i die erste Position im unsortierten Teil angibt. Die Erweiterung des sortierten Teils ist dann einfach das Vorrücken von i. Einen zweiten Laufindex, j, verwenden wir für die Suche im unsortierten Teil: an jeder Position wird das Element mit dem gerade kleinsten Element des unsortierten Teils verglichen. Die Position des kleineren Elementes, das wir vor Ende des Feldes gefunden haben, merken wir uns. Für das Vertauschen von Positionen der Elemente brauchen wir einen Zwischenspeicher.

[Vornberger und Thiesing, 1998] entnehmen wir die folgende Realisierung des Verfahrens in JAVA:

```
public class SelectionSort {                                  // Klasse SelectionSort

    public static void sort (int[] a) {                       // statische Methode sort
        for (int i=0; i<a.length-1; i++) {                    // durchlaufe Feld
            int k = i;                                        // Index des bisher kleinsten
            int x = a[i];                                     // Wert des bisher kleinsten
            for (int j=i+1; j<a.length; j++)                  // durchlaufe Rest des Felds
            if (a[j] < x) {                                   // falls kleineres gefunden,
                k = j;                                        // merke Index
                x = a[j];                                     // merke Position
            }
            a[k] = a[i];                                      // speichere bisher kleinstes um
            a[i] = x;                                         // neues kleinstes nach vorne
        }
    }
}
```

Wir brauchen dann noch eine Klasse, die vom Benutzer eine Zahlenfolge anfordert und das sortierte Ergebnis ausgibt ([Vornberger und Thiesing, 1998]):

```
import AlgoTools.IO;

/** testet Sortierverfahren
*/
```

```
public class SelectionSortTest {

    public static void main (String argv[]) {

        int[] a;                                                    // Feld fuer Zahlenfolge

        a = IO.readInts ("Bitte eine Zahlenfolge: ");               // Folge einlesen
        SelectionSort.sort (a);                                     // SelectionSort aufrufen
        IO.print ("sortiert mit SelectionSort: ");
        for (int i=0; i<a.length; i++) IO.print (""+a[i]);          // Ergebnis ausgeben
        IO.println ();
        IO.println ();
    }
}
```

Rufen wir dies Programm auf, sehen wir:

```
Bitte eine Zahlenfolge: 2 15 10 30 1
sortiert mit SelectionSort: 1 2 10 15 30
```

Die Teile waren dabei nacheinander:[15]

bei $i = 0$ sortiert $\emptyset$, unsortiert `2 15 20 30 1`
bei $i = 1$ sortiert `1`, unsortiert `15 10 30 2`
bei $i = 2$ sortiert `1 2`, unsortiert `10 30 15`
bei $i = 3$ sortiert `1 2 10`, unsortiert `30 15`
bei $i = 4$ sortiert `1 2 10 15`, unsortiert `30`

Damit endet das Verfahren, das letzte Element muß nicht betrachtet werden. Die Zahl 2 wird zweimal verschoben. Bei $i = 1$ wird zunächst k auf 2 gesetzt, weil 10 kleiner als 15 ist. 15 wird in der Variablen x festgehalten. Beim Vergleich des j-ten Elementes, $j = 4$, mit x wird das noch kleinere Element gefunden.

Nun wissen wir also, wie wir das Problem in JAVA lösen können: durch zwei Schleifen, eine äußere und eine innere. Jetzt wollen wir diese Schleifen genauer untersuchen. Dafür müssen wir ein Prinzip kennenlernen, das wir dazu verwenden wollen, Aussagen über ein Programm zu beweisen, bevor wir dieses Prinzip auf unsere Schleifen von der Selektionssortierung anwenden.

4.1.3 Induktionsbeweis

Die *mathematische Induktion* ist ein Verfahren, das eine Aussage $S(n)$ für nichtnegative Zahlen n beweist. Das Verfahren besteht aus zwei Schritten:

Induktionsanfang: Meist ist die Aussage $S(0)$ der Induktionsanfang. Gilt die Aussage für $n = 0$? Es kann aber statt 0 irgendeine Zahl b sein, so daß $S(n)$ nur für $n \geq b$ bewiesen wird.

Induktionsschritt: Zu beweisen ist, daß aus $S(n)$ logisch folgt, daß $S(n+1)$ gilt. Wir nehmen also an, daß $S(n)$ wahr ist. Dies ist die *Induktionsannahme*. Dann zeigen wir, daß dann auch $S(n+1)$ wahr ist. Gilt $S(n)$ nicht, ist die Aussage ohnehin wahr.

[15] Der Index des ersten Elements eines Feldes ist 0, nicht 1.

Um den letzten Satz zu verstehen, hier noch einmal zur Erinnerung die Implikation.

Die logische Implikation – aus A folgt B – ist wahr, wenn

- A falsch ist oder
- A und B beide wahr sind.

In einem Induktionsbeweis zeigen wir also, daß $S(0)$ wahr ist. Dann zeigen wir, daß, falls $S(n)$ wahr ist, auch $S(n+1)$ wahr ist. Wie kann denn dieser Induktionsschritt funktionieren? Die Argumentation geht auf zwei alternative Arten.

- Wir wollen wissen, ob $S(a)$ für irgendein a gilt. Wenn $a = 0$, dann haben wir beim Induktionsanfang den Beweis schon geführt. Wenn $a > 0$, dann gelangen wir durch eine Kette dahin: $S(0)$ impliziert $S(1)$, $S(1)$ impliziert $S(2)$ und so weiter bis $S(a)$. Egal, welchen Wert a hat, irgendwann erreichen wir ihn.
- Wir können auch mit einem Gegenbeispiel argumentieren, warum der Induktionsschritt Sinn macht. Nehmen wir mal an, a wäre die kleinste Zahl, bei der $S(n)$ nicht gilt. Dann ist also $S(a-1)$ noch wahr, aber $S(a)$ nicht. Dies ist ein Widerspruch! (Die Implikation "aus A folgt B" ist falsch, wenn A wahr ist und B ist falsch.) Unsere Annahme, es gäbe ein a, so daß für $n > a$ gilt, daß $S(n)$ falsch ist, führt zu einem Widerspruch. Na, dann ist unsere Annahme falsch. Es gibt also kein a, ab dem $S(n)$ falsch ist, wenn alles davor wahr ist.

Bleibt man im Rahmen der logischen Argumentation, dann gelingt der Induktionsbeweis und macht auch Sinn. Jetzt wollen wir sehen, wie wir ihn anwenden, um ein Programm zu begründen.

4.1.4 Induktionsbeweis am Beispiel der Selektionssortierung

Wir haben für die Sortierung das Modell der Selektionssortierung erarbeitet und es in JAVA implementiert. Jetzt wollen wir auch unsere dritte Frage beantworten:

Warum funktioniert unser Programm? Wir haben zwei Schleifen ineinander geschachtelt. Können wir über diese Schleifen irgendeine Aussage machen?

Nehmen wir zunächst die innere Schleife und schreiben sie Schritt für Schritt auf, d.h. wir zerlegen die kompakte *for*-Anweisung in einzelne Schritte. An den Anfang schreiben wir, wie wir in diese Schleife hineingeraten. Dann markieren wir den Platz im Programmablauf, an dem eine Aussage $S(n)$ immer wahr sein soll. Dann schreiben wir die Schleife Schritt für Schritt auf. Die Aussage $S(n)$ selbst sehen wir nach dieser Auflistung.

1. $x = a[i]$
2. $j = i + 1$
3. Aussage $S(n)$, wobei wir uns mit n auf den Zähler j der Schleife beziehen. Wir sagen deshalb nicht einfach $S(j)$, weil wir in der Argumentation manchmal j verändern, während n gleich bleibt. Hier betrachten wir den Zustand direkt vor der Abbruchbedingung der Schleife.
4. $j \geq a.length$?

5. $x \geq a[j]$?
6. $k = j; x = a[j];$
7. $j++;$

Wenn die Frage 4) mit "ja beantwortet wird, verlassen wir die Schleife. Wenn die Frage 5) mit "nein" beantwortet wird, gehen wir zu Anweisung 7).

Was ist nun eine gute Aussage, die wir beweisen wollen? Bei einer Schleife ist es gut, etwas zu beweisen, was immer an einem Punkt in der Schleife wahr ist. Dies heißt die *Schleifeninvariante*. Worum ging es bei dieser Schleife? Es sollte im noch unsortierten Rest des Feldes nach dem kleinsten Wert gesucht werden. Der unsortierte Rest ist immer von $a[i]$ bis zum Ende von $a[]$ zu durchsuchen. Und dann soll x den kleinsten Wert haben, der zu finden war. Formulieren wir dies also als Aussage $S(n)$!

Aussage S(n): Wenn wir 4) mit n als Wert von j erreichen, ist der Wert der Variablen x der kleinste Wert im Feld $a[]$ von $a[i]$ bis $a[n-1]$ und k dessen Position.

Induktionsanfang: Es gibt nun einen natürlichen Induktionsanfang, nämlich, wenn wir das erste Mal in die Schleife hineingeraten. Dann ist $x = a[i]$ und $j = i + 1$. Also müssen wir zuerst zeigen, daß für $n = i+1$ unsere Aussage gilt: $S(i+1)$. Ausformuliert heißt das: "x ist der kleinste Wert im Feld von $a[i]$ bis $a[i]$." Dies ist wahr, denn $x = a[i]$ wurde ja gerade in 2) gesetzt und später kommen wir nicht noch einmal in die Situation, daß $n = i + 1$, weil ja in 7) j inkrementiert wird. Also gilt $S(i + 1)$.

Induktionsschritt: Wenn $S(i + 1)$ gilt, dann wollen wir weiter für $n \geq i + 1$ beweisen, "$S(n)$ impliziert $S(n + 1)$".

- Wenn $n \geq a.length$, verlassen wir die Schleife. Wir werden also nicht zur Schleifeninvarianten 3) vordringen. Damit ist der erste, der "wenn wir $j \geq a.length$ erreichen" Teil unserer Aussage falsch. Damit ist die Implikation (unsere Aussage) bestimmt wahr. Für alle $n \geq a.length$ gilt: $S(n)$ impliziert $S(n + 1)$.
- Wenn n kleiner als $a.length$ ist, gilt dann "wenn $S(n)$ dann $S(n + 1)$"? Wir werden also den Test in 4) mit $n + 1$ als Wert von j erreichen. Ist x dann der kleinste Wert des Feldes von $a[i]$ bis $a[n]$? Dazu betrachten wir zwei Fälle, je nach Ausgang des Test in 5).

  Wenn $a[n]$ nicht kleiner ist als der kleinste Wert im Feld von $a[i]$ bis $a[n-1]$, dann wird in 6) der Wert von x nicht geändert. Dann ist also x der kleinste Wert auch bei $n + 1$.

  Wenn $a[n]$ kleiner ist als der bisher kleinste Wert im Feld von $a[i]$ bis $a[n-1]$, dann erhält x in 6) den Wert $a[n]$. Dann ist also x der kleinste Wert bei $n + 1$.

  j wird in 7) inkrementiert und dann erreichen wir den entscheidenden Punkt, die Schleifeninvariante. Gerade dann gilt die Aussage $S(n + 1)$.

Wir haben also gezeigt, daß $S(n + 1)$, wenn $S(n)$ unter der Annahme, daß $S(i + 1)$.

Die innere Schleife ist also gerade so, wie das Modell der Selektionssortierung es vorsah. Prüfen wir nun die äußere Schleife! Haben wir die Eigenschaft unseres Modells wirklich implementiert, daß der bereits sortierte Teil sich nicht mehr verändert? Wir haben wieder

eine Zahl, den Wert von i, die wir für den Induktionsbeweis nutzen können. Der Induktionsanfang ist einfach der Anfangszustand des Programms[16]. Dort ist $i = 0$. Die Aussage bezieht sich wieder auf den Programmzustand direkt vor der Abbruchsbedingung der *for*-Schleife.

Aussage $T(m)$: Wenn wir den Schleifentest, $i \geq a.length - 1$, mit m als dem Wert der Variablen i erreichen, dann gilt

a) Das Feld ist von $a[i]$ bis $a[m-1]$ sortiert, d.h. $a[0] \leq a[1] \leq ... \leq a[m-1]$.

b) Alle Werte von $a[m]$ bis zum Ende des Feldes sind mindestens so groß wie jeder beliebige Wert von $a[0]$ bis $a[m-1]$.

Induktionsanfang: $m = 0$ ist der Anfang, wie durch $int\ i = 0$ angegeben. $T(0)$ ist natürlich wahr, denn ausformuliert heißt dies: "das Feld ist von $a[0]$ bis $a[-1]$ sortiert." Es gibt gar keine Elemente in $a[0]$ bis $a[-1]$. Über die leere Menge kann man beliebige Aussagen treffen – sie sind alle wahr. Also sind die nicht vorhandenen Elemente sortiert und kleiner als die Elemente in $a[0]$ bis zum Feldende. $T(0)$ ist also wahr.

Induktionsschritt: Für $m > 0$ nehmen wir an, daß $T(m)$ wahr ist, und wollen zeigen, daß dann auch $T(m+1)$ wahr ist.

- Wenn wir den Schleifentest, $i \geq a.length - 1$, nicht mit $m+1$ als Wert von i erreichen, ist der erste Teil der Implikation ("Wenn wir den Schleifentest ... erreichen") falsch und damit $T(m+1)$ sowieso wahr.
- Betrachten wir also den Fall, wenn m kleiner als $a.length - 1$ und größer als 0 ist.

  Hat i den Wert m, so wird in der inneren Schleife – wie durch $S(m)$ bewiesen – der kleinste Wert in $a[m]$ bis zum Feldende gefunden. Dieses kleinste Element wird der neue Wert von x und umgespeichert auf die Position "klein", $a[k]$.

  Wir nehmen ja an, daß $T(m)$ gilt. Es ist also $a[0]$ bis $a[m-1]$ sortiert. Jetzt speichern wir das kleinste Element des Feldrestes an die richtige Stelle im sortierten Teil. Dann ist Teil a) der Aussage $T(m+1)$ wahr: das Feld ist von $a[0]$ bis $a[m]$ sortiert.

  Wir nehmen an, daß $T(m)$ gilt und betrachten Teil b) der Aussage. Für $i = m$ gilt, daß alle Elemente im unsortierten Teil $a[m]$ bis Feldende größer oder gleich groß einem jeden beliebigen Element in $a[0]$ bis $a[m-1]$ sind. Wir haben aus dem unsortierten Rest das kleinste Element herausgenommen. Kein Element im unsortierten Teil des Felds ist kleiner als dies. Verkürzen wir den unsortierten Teil $(i++)$, dann bleiben darin immer noch nur Elemente, die nicht kleiner sind als ein Element im nunmehr verlängerten sortierten Teil $a[0]$ bis $a[m]$. In anderen Worten: "alle Elemente in $a[m+1]$ bis Feldende sind mindestens so groß wie jeder beliebige Wert eines Elementes in $a[0]$ bis $a[m]$." Also ist Teil b) der Aussage $T(m+1)$ wahr.

[16] Sie sehen jetzt, warum Zustände bei Programmen schon eingeführt wurden und erinnern sich, daß ein Zustand die Werte von Variablen angibt (3.4).

- Wenn nun $m \geq a.length - 1$ ist, verlassen wir die äußere Schleife und damit das Programm. $T(m)$ gilt, zu $T(m+1)$ kommen wir nicht mehr. Da $T(m)$ gilt, ist das Feld von $a[0]$ bis $a[m-1]$ sortiert (Teil a) der Aussage). $T(m)$ ist mindestens so groß wie irgendein Element in $a[0]$ bis $a[m-1]$ (Teil b) der Aussage). Damit ist das Feld insgesamt sortiert.

Das Programm entspricht also tatsächlich dem Modell der Sortierung, das durch die Aussagen $S(n)$ und $T(m)$ charakterisiert ist.

4.1.5 Was wissen Sie jetzt?

Sie wissen, was Informatiker meinen, wenn sie "Sortierung" sagen. Sie haben ein bestimmtes Modell der Sortierung gesehen, die Sortierung durch Auswählen. Die Realisierung in JAVA illustriert den Gebrauch von Feldern und Schleifen.

Die Selektionssortierung hat die Eigenschaft, daß bereits Sortiertes nicht mehr betrachtet zu werden braucht. Hat unser Programm auch diese Eigenschaft? Davon haben wir uns mithilfe des Induktionsbeweises überzeugt. Wir haben Eigenschaften vor der Schleife, eine "Wenn, dann"-Aussage, die vor der Abbruchbedingung der Schleife gelten soll, und eine Schleife, die aus dem Vorgängerzustand den aktuellen Zustand herleitet. Die Eigenschaften vor der Schleife drückt der Induktionsanfang aus. Die Schleife entspricht dem Induktionsschritt. Aus dem Beweis des Induktionsschrittes unter der Annahme des Induktionsanfangs können wir die Aussage herleiten.

4.2 Abstrakte Datentypen

Oft muß man eine Reihe von Daten gleichen Typs verarbeiten, zum Beispiel eine Reihe von Zahlen, Namen oder Telefonbucheinträgen. Wenn wir auf diese Daten über ihre Position in der Reihe zugreifen wollen, können wir sie als Feld modellieren. Wenn wir von einem aktuellen Element aus zum Vorgänger und von da aus zum nächsten durchgehen wollen, bietet sich die Liste an. Wenn wir immer nur das vorderste Element betrachten wollen, wie der Wirt hinter der Theke, ist die Schlange das Richtige. Dort stellt man sich hinten an. *First in, first out.* Aber manchmal ist es auch anders herum: Aktenkörbe werden eben nicht von unten nach oben, sondern vom neuesten Eingang zum frühesten bearbeitet. Dann gilt: *Last in, first out.* Das passende Modell ist der Keller.

Liste, Schlange, Keller sind abstrakte Datentypen. Dies ist nun kein Begriff der Programmiersprache JAVA, sondern ein allgemeines Konzept der Informatik. Aho und Ullmann sprechen vom *Datenmodell*, das in einer Programmiersprache durch eine *Datenstruktur* realisiert wird [Aho und Ullman, 1996]. Das, was den verschiedenen Realisierungen gemeinsam ist, sind Operationen, die bestimmte Ergebnisse erbringen. Wie dies im Einzelnen geschieht, davon wird abstrahiert. Es genügt zu wissen, daß man diese Operationen anwenden kann. Die Kenntnis der abstrakten Datentypen und wie man sie zum Lösen praktischer Probleme einsetzt, macht unabhängig von den sich laufend ändernden Programmiersprachen. Deshalb wird im Folgenden besonderer Wert auf abstrakte Datentypen gelegt (Listen, Bäume, Graphen).

Definition 4.2: Abstrakter Datentyp Ein abstrakter Datentyp ist durch eine Menge von Operationen gegeben, die bezogen auf ein Datenmodell definierte Wirkungen zeigen.

Wir stellen hier stets den abstrakten Datentyp vor, realisieren ihn in JAVA und zeigen anhand von Beispielen, wie er nützlich eingesetzt werden kann. Eine Besonderheit von JAVA ist, daß es den Gedanken der Datenabstraktion bereits in die Programmiersprache aufgenommen hat. JAVA bietet mit seinen abstrakten Klassen und Schnittstellen gerade die Abstraktion an, die bei abstrakten Datentypen gemeint ist. Obendrein sind in der JAVA-Bibliothek `java.util` die wichtigsten abstrakten Datentypen realisiert.

4.3 Listen als Verkettete Listen

Eine Liste in der Informatik entspricht der natürlichsprachlichen Auffassung dieses Begriffes. Jedes Element außer dem ersten und dem letzten hat einen Vorgänger und einen Nachfolger, d.h. ein Element kommt in der Liste als nächstes. Wenn die Reihenfolge der Elemente einer Liste auch einer Ordnung entspricht (so wie $\leq$ bei Zahlen oder das Alphabet bei Zeichenketten), so redet man von einer "geordneten Liste".

Die Operationen, die für Listen definiert sind, beziehen sich auf

- das aktuelle Element,
- Nachfolger des aktuellen Elements,
- Vorgänger des aktuellen Elementes,
- die leere Liste,
- das erste und
- das letzte Element.

Die Operationen

- gehen die Liste durch, d.h. der Nachfolger wird aktuelles Element,
- geben ein Element zurück,
- löschen ein Element,
- oder fügen eines ein.

Wie wird der abstrakte Datentyp *Liste* nun realisiert? Eine Liste hat einen Anfang und ein Ende und besteht aus ein oder mehreren Elementen. Wie wir wissen hat ein Element in der Mitte einen Vorgänger und einen Nachfolger. Für viele Fälle genügt es, wenn wir die Liste nur in eine Richtung gehen, und zwar von vorne nach hinten. Uns interessiert also nur der Anfang der Liste und der Nachfolger eines Elementes. Ein Element besteht aus zwei Teilen: einem Inhalt und einem Zeiger auf den Nachfolger, wie auf Bild 13.

Man baut nun eine Liste auf, indem man mehrere Elemente aneinander hängt. Dabei hat jedes Element einen Zeiger auf das nachfolgende Element. Eine Ausnahme bildet das Ende der Liste, hier zeigt der Zeiger in ein definiertes Nichts. In Java wird dafür das Schlüsselwort *null* verwandt. Im Programm selbst muß man sich nur noch den Start der Liste merken, alle anderen Elemente kann man von dort aus ja erreichen.

Bild 14 stellt eine einfach verkettete Liste dar, deren Elemente noch nicht geordnet sind.

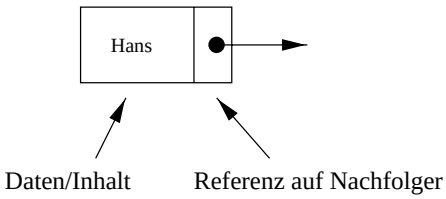


Abbildung 13: Element einer Liste

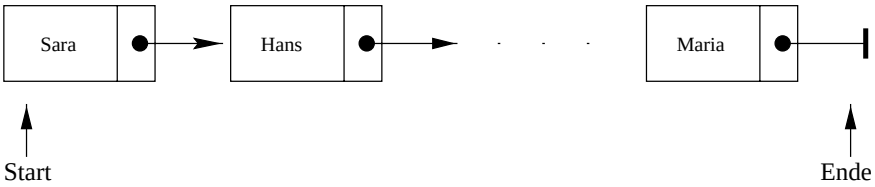


Abbildung 14: Eine einfach verkettete Liste

Als Beispiel wollen wir Andreas zwischen Sara und Hans einfügen. Dazu setzen wir erst den Nachfolger von Andreas auf Hans und dann den Nachfolger von Sara auf Andreas. Damit ist Andreas in die Liste eingefügt und die Liste sieht aus, wie auf Bild 15.

Einfacher ist es, wenn wir Andreas wieder löschen wollen. Dann müssen wir nur den Nachfolger von Sara wieder auf Hans setzen, und Andreas ist nicht mehr in der Liste.

Gelegentlich kommt es vor, das wir eine Liste in beide Richtungen abgehen müssen. Dazu erweitern wir unseren Elementtyp um eine Referenz auf den Vorgänger:

Bei einer doppelt verketteten Liste zeigt der Vorgänger des Kopfes und der Nachfolger des Endes auf *null*. Unsere Liste aus Bild 14 sieht dann so aus wie auf Bild 17.

In JAVA 1.2 heißt die Klasse der doppelt verketteten Listen `LinkedList`. Sie ist eine Unterklasse der abstrakten Klasse `AbstractSequentialList` und befindet sich im Paket `java.util`. Damit man sich vorstellen kann, wie JAVA die verkettete Liste realisiert, zeige ich hier die Deklaration eines Listenelements **Entry**, das als Eigenschaften einen Inhalt, einen Vorgänger und einen Nachfolger hat. Ein Vorgänger bzw. Nachfolger ist wieder so ein Eintrag, hat also wieder einen Inhalt, einen Vorgänger und einen Nachfolger. Der Eintrag hat die Grundmethoden für das Einfügen und Löschen. Die Liste besteht aus Einträgen. Der Konstruktor erzeugt eine leere Liste. Diese ist dadurch definiert, daß das aktuelle Element gleich seinem Vorgänger und seinem Nachfolger ist (nicht der Inhalt ist gleich, sondern das Element ist dasselbe!). Die Methoden der verketteten Liste rufen für einen Eintrag dessen Methode auf. Die Namen der deklarierten Methoden sprechen für sich:

- add(int index, Object element)
- add(Object element)
- addFirst(Object element) – fügt vorn ein
- addLast(Object element) – fügt hinten an
- get(int index)

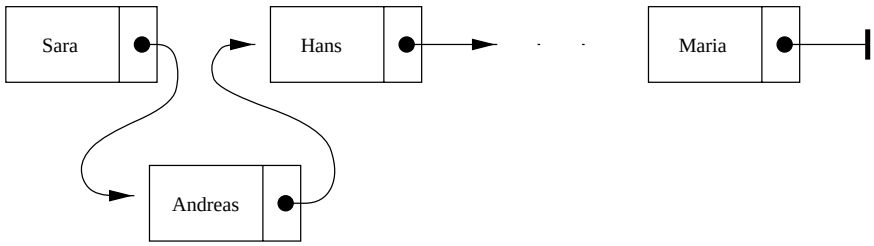


Abbildung 15: Zustand nach Einfügen von “Andreas” in die Liste

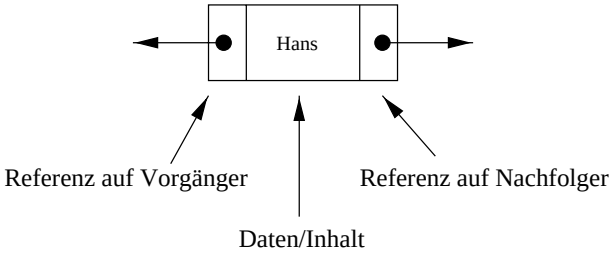


Abbildung 16: Element einer doppelt verketteten Liste

- remove(int index)
- remove(Object o) – löscht das erste Auftreten des Elements *o*.
- getFirst() – gibt das erste Element
- getLast() – gibt das letzte Element
- size() – gibt die Anzahl von Elementen an

```
private static class Entry {
    Object element;
    Entry next;
    Entry previous;

    Entry (Object element, Entry next, Entry previous) {
        this .element = element;
        this .next = next;
        this .previous = previous;
    }


    private Entry addBefore (Object o, Entry e) {
        Entry newEntry = new Entry (o, e, e.previous);
        newEntry.previous.next = newEntry;
        newEntry.next.previous = newEntry;
```

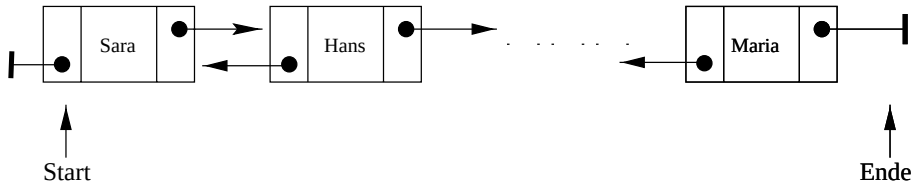


Abbildung 17: Eine doppelt verkettete Liste

```
            size++;
            modCount++;
            return newEntry;
        }
}
```

```
public class LinkedList extends AbstractSequentialList implements List, Cloneable,
java.io.Serializable {
        private transient Entry header = new Entry (null, null, null);
        private transient int size = 0;


        /**
        * Inserts the given element at the beginning of this List.
        */
        public void addFirst (Object o)  {
            addBefore (o, header.next);
        }
}
```

Außerdem gibt es in JAVA eine Klasse **ListIterator**. Ein Iterator ist ein Zeiger, der die Liste in beiden Richtungen durchgehen kann, und dabei auch Elemente einfügen, ersetzen und löschen kann. Die Methode **hasNext** liefert *true*, falls der Zeiger auf ein Element zeigt, das einen Nachfolger hat. Die Methode **next** liefert das nächste Element. In der JAVA-Implementierung ist das Durchgehen der Liste also in einer eigenen Klasse realisiert. Die Verbindung zwischen beiden Klassen wird durch Methoden realisiert. Die Methode **listIterator** der Klasse **LinkedList** erzeugt ein Objekt vom Typ **ListIterator** und hängt diesen neuen Zeiger vor das erste Element der List (wenn kein Parameter angegeben ist) bzw. das i-te Element bei **listIterator(int i)**.

Die Anwendung des abstrakten Datentyps soll nun durch ein einfaches Beispiel illustriert werden. Wir verwenden **LinkedList** als Darstellung für die Teilnehmer eines 100-Meter-Laufs. Die Klasse **Lauf100** hat als Eigenschaft eine verkettete Liste $laeufer$. Die Methode **teilnehmerEingabe** liefert eine verkettete Liste zurück, die alle Läufer enthält. Nachdem wir die Angaben zu einem Teilnehmer in bekannter Weise vom Bildschirm eingelesen haben, rufen wir die Methode **add** dieser Klasse auf. In Zeile 18 ist zu sehen, wie die Methode auf einen Teilnehmer angewandt wird, der mit dem Konstruktor der Klasse **Teilnehmer** erzeugt wird, wobei die eingelesenen Werte für seine Eigenschaften $name$ und $alter$ übergeben werden. Die Schleife nimmt so viele Teilnehmer in die Liste auf, wie

die Benutzerin will.

Die Methode **zeitenEinlesen** verwendet den Listeniterator, um die Liste sequentiell durchzugehen, solange es ein Nachfolgerelement gibt. Leider ist es nun so, daß diese Methode ein Objekt der Klasse **Object** und nicht eines der Klasse **Teilnehmer** zurückgibt. Wir müssen daher eine Typumwandlung vornehmen. Man kann in JAVA durch einen geklammerten Klassennamen angeben, daß der tatsächliche Typ eine Unterklasse von dem zurückgelieferten Typ ist. So ergibt sich Zeile 28. Das Listeniterator- Objekt i wendet seine **next**-Methode an, die ein Objekt zurückgibt, das als Objekt der Klasse **Teilnehmer** betrachtet wird und auf das t verweist.

In der **main**-Methode wird nun die verkettete Liste, die aus Objekten vom Typ **Teilnehmer** mit den Eigenschaften *name*, *alter*, *zeit* besteht, einmal durchlaufen, um die Zeiten einzulesen mit der Methode **zeitenEinlesen**. Danach wird sie noch einmal durchlaufen – wieder mit der Methode **next** von **LinkedList** – um die beste Zeit zu ermitteln. Das Ergebnis wird auf dem Bildschirm ausgegeben. Sie sehen, wie kurz der Code für das Anfügen eines Listenelementes und das Durchlaufen der Liste ist, da wir die Klassen für verkettete Listen und Listeniteratoren schon haben.

```
/**
* Lauf100.java
*
* Ein Beispielprogramm fuer den Gebrauch linearere Listen.
* Das Szenario ist ein 100m-Lauf in einem Leichtatlethik-
* wettbewerb.
*
*/

import java.lang.String;
import java.util.LinkedList;
import java.util.ListIterator;
import AlgoTools.*;

/**
* Diese Klasse fuehrt den Lauf durch und sucht dann den
* Sieger und gibt ihn aus
*/
public class Lauf100 {

    static LinkedList laeufer;                          // Die Teilnehmer am 100m-Lauf

    /**
    * Hier werden die Teilnehmer am 100m-Lauf eingelesen
    */
    static LinkedList teilnehmerEingabe () {

        LinkedList liste = new LinkedList ();
        String name;
        int alter;

        while (IO.readString ("\nWollen Sie einen Laeufer eingeben?").equals("ja")) {
```

```
            name = IO.readString ("Bitte geben Sie den Namen des Laeufers ein: ");
            alter = IO.readInt ("Bitte geben Sie das Alter des Laeufers ein: ");
            liste.add (new Teilnehmer (name, alter));                    // neuen Teilnehmer in
                                                                              // Liste aufnehmen
        }

        return liste;

}


/**
* Der Benutzer gibt hier die Zeiten der Laeufer ein
*/
static void zeitenEinlesen (LinkedList L)  {

        ListIterator i = L.listIterator ();
        Teilnehmer t;
        double zeit;

        while (i.hasNext ())  {
            t = (Teilnehmer)i.next ();
            t.zeit = IO.readDouble ("Bitte geben Sie die Zeit von "+ t.name + ", "
                                    + t.alter + "ein: ");
        }
}


/**
* Hier werden erst die Laeufer eingegeben,
* dann das Rennen gemacht und dann
* die ermittelten Zeiten aufgenommen.
*/
static void main (String[] argv)  {

        Teilnehmer sieger;                                          // Der Sieger des Laufes
        double bestZeit;                                             // Laufzeit des Siegers
        IO.println ("Hallo ! Willkommen beim 100m-Lauf !\n");

        // Eingabe der Laeufer
        laeufer = teilnehmerEingabe ();

        IO.println ("\nJetzt fuehren Sie bitte das Rennen durch und fuettern mich "
                    + "dann mit den Zeiten.");

        // Eingabe der Zeiten durch die Preisrichter
        zeitenEinlesen (laeufer);

        // Besten ermitteln
        ListIterator i = laeufer.listIterator(0);

        sieger = (Teilnehmer)i.next ();
        bestZeit = sieger.zeit;
```

```
        while (i.hasNext ())  {
                Teilnehmer t;                                   // Benoetigt fuer die Suche des Siegers
                                                                    // hier lokale Variable der Schleife
                t = (Teilnehmer)i.next ();
                if (t.zeit < bestZeit)  {
                        sieger = t;
                        bestZeit = t.zeit;
                }
        }

        // Besten ausgeben. IO.println ("\n Auswertung:");
        IO.println ("————————————————————————");
        IO.println ("Der Gewinner des diesjaehrigen 100m-Laufes ist:");
        IO.println ("Name: "+ sieger.name + ", "+ sieger.alter);
        IO.println ("Mit einer Bestzeit von "+ bestZeit);
    }

}
```

```
public class Teilnehmer  {

    public String name;
    public int alter;
    public double zeit;
    Teilnehmer (String _name, int _alter)  {
        name = _name;
        alter = _alter;
    }

}
```

4.4 Schlangen

Schlangen oder Warteschlangen sind insbesondere für die Organisation von Prozessen wichtig: wenn wir für etwas keine Zeit haben, stellen wir es in die Warteschlange. Diese arbeiten wir der Reihe nach ab und so kommt jeder (Prozeß) an die Reihe. Im Gegensatz zur Liste haben wir hier die Sicht auf das vorderste Element: dies wird bearbeitet und dann entfernt, so daß der Nachfolger dann erstes Element wird.

Definition 4.3: Schlange Eine Schlange ist ein abstrakter Datentyp, der eine Folge von Elementen mit einem sogenannten Frontelement (vorderstem Element) darstellt. Die Operationen sind das Anstellen am Ende der Schlange, das Entfernen des Frontelements, das Zurückgeben des Frontelements und das Testen, ob die Schlange leer ist.

Eine übersichtliche Implementierung dieses Datentyps ist die folgende:

```
package LS8Tools;

import AlgoTools.IO;

public class Schlange {

    private Object[] inhalt;                          // Feld fuer Schlangenelemente
    private int head;                                 // Index fuer Schlangenanfang
    private int count;                                // Anzahl Schlangenelemente

    public Schlange (int N) {                         // Konstruktor fuer leere Schlange
        inhalt = new Object[N];                       // besorge Platz fuer N Objekte
        head = 0;                                     // initialisiere Index fuer Anfang
        count = 0;                                    // initialisiere Anzahl
    }

    private boolean full () {                         // Testet, ob Schlange voll ist
        return count==inhalt.length;                  // Anzahl gleich Feldlaenge?
    }

    public boolean empty () {                         // Testet, ob Schlange leer ist
        return count==0;                              // Anzahl gleich 0?
    }

    public void enq ( Object x ) {                    // Fuegt x hinten ein
        if (full ()) IO.error ("in enq: Schlange ist voll!");
        inhalt[ (head+count)%inhalt.length]=x;        // Element einfuegen
        count++;                                      // Anzahl inkrementieren
    }

    public void deq () {                              // Entfernt vorderstes Element
        if (empty ()) IO.error ("in deq: Schlange ist leer!");
        head = (head + 1) % inhalt.length;            // Anfang-Index weiterruecken
        count--;                                      // Anzahl dekrementieren
    }

    public Object front () {                          // Liefert Element,
        if (empty ()) IO.error ("in front: Schlange ist leer!");
        return inhalt[head];                          // welches am Anfang-Index steht
    }


    public int length () {                            // Liefert die Anzahl der
        return count;                                 // Elemente in der Schlange
    }

    public String toString () {                       // Gibt Schlange in String aus
        int current;                                  // Aktueller Index
```

```
            int currentCount;                                    // Aktuelle Elementnummer
            String s;

            current = head;
            currentCount = 0;
            s = new String ("");
            while (currentCount < count)  {
                  s += ""+ inhalt[current].toString();
                  current = (current + 1) % inhalt.length;
                  currentCount++;
            }
            return (s);
      }
}
```

Beachten Sie, daß die Schlange jedes Objekt speichern kann, da sie als Feld von Objekten der Klasse **Object** realisiert ist und jede Klasse in der Vererbungshierarchie unter **Object** ist.

Als Beispiel betrachten wir zwei unterschiedliche Arten des Schlangestehens. In Deutschland gibt es meistens pro Schalter eine Schlange. Wer neu in den Schalterraum kommt, stellt sich an die kürzeste Schlange an. Möglicherweise dauert es aber gerade hier am längsten, denn man weiß ja nicht, wie lange ein Kunde in der Schlange am Schalter spricht.

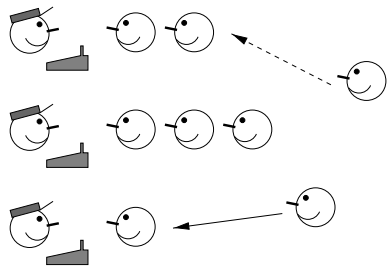


Abbildung 18: Schalterschlangen

In den Vereinigten Staaten von Amerika gibt es die Verteilerschlange. Es gibt nur eine Warteschlange, an die sich neu Hinzukommende anstellen. Wenn ein Schalter frei wird, kommt der erste der Warteschlange dran.[17]

Nachdem wir nun wissen, was wir programmieren wollen, sehen wir auch, wie es in JAVA programmiert werden kann. Die beiden Programme `SchalterTest` und `VerteilerTest` sowie die von diesen verwendeten Klassen **Mitarbeiter**, **Simulation**, **SchalterSimu** und **VerteilerSimu**. sind im Verzeichnis `~gvpr000/ProgVorlesung/Beispiele/schlange/` zu finden.

4.5 Keller

Ein Keller wird ebenso wie eine Schlange für die Verwaltung von Aufgaben oder Prozessen genutzt. Hier stellen wir aber neue Aufgaben nicht hinten an, sondern erledigen sie als erstes.

[17] Die Post in Hombruch hat diese angenehme Warteschlange übrigens auch eingeführt!

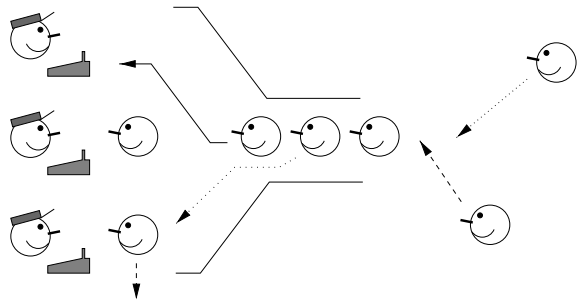


Abbildung 19: Verteilerschlange

Definition 4.4: Keller Ein Keller ist ein abstrakter Datentyp, der eine Folge von Elementen darstellt. Die Folge hat ein möglicherweise undefiniertes oberstes Element *top* oder auch *peek* genannt. Die Operationen sind das Ablegen eines Elementes auf den Keller (*push*), das Entfernen des obersten Elementes (*pop*), das Anzeigen des obersten Elementes und der Test, ob der Keller leer ist. Wird nacheinander *push* und *pop* angewandt, ist der Keller unverändert. Nach *push* mit dem Element x liefert *top* das Element x.

Eine elegante Implementierung dieses abstrakten Datentyps finden wir wieder in [Vornberger und Thiesing, 1998].

```
import AlgoTools.IO;

public class Keller {

    private class KellerEintrag {
        Object inhalt;                                  // Inhalt des KellerEintrags
        KellerEintrag next;                             // zeigt auf naechsten KellerEintrag
    }

    private KellerEintrag top;                          // zeigt auf obersten KellerEintrag

    public Keller () {                                  // legt leeren Keller an
        top = null;
    }

    public boolean empty () {                           // liefert true,
        return top == null;                             // falls Keller leer
    }

    public void push (Object x) {                       // legt Objekt x
        KellerEintrag hilf = new KellerEintrag ();      // auf den Keller
        hilf.inhalt = x;
        hilf.next = top;
        top = hilf;
    }

    public Object top () {                              // liefert oberstes
```

```
        if (empty ()) IO.error ("in top: Keller leer");          // Kellerelement
        return top.inhalt;
    }

    public void pop ()  {                                        // entfernt oberstes
        if (empty ()) IO.error ("in pop: Keller leer");          // Kellerelement
        top = top.next;
    }
}
```

Bei dieser Implementierung wird eine Klasse, **KellerEintrag**, eingebettet, so daß sie wie eine Eigenschaft der Klasse **Keller** zu betrachten ist.

Eine Klasse, die einen Keller realisiert, gibt es im Paket `java.util` unter dem Namen **Stack**. Diese Klasse ist als Unterklasse von **Vector** implementiert: die Kellereinträge sind die Elemente, die der Keller übereinander stapelt. Während bei **Keller** von der Methode **pop** nichts zurückgeliefert wird, ist der Rückgabewert der Methode **pop** bei der JAVA-Implementierung von **Stack** das entfernte Element.

Ein Beispiel zur Verwendung des Kellers ist das berühmte Problem des Affen mit der Banane. An der Decke hängt eine Banane, die der Affe nicht direkt erreichen kann. Er muß erst eine Kiste unter die Banane schieben, darauf klettern und dann die Banane holen. Dieser Plan kann als Objekt der Klasse **Stack** dargestellt werden. Dabei ist jedes Element des Kellers ein Zustand. Ein Zustand ist ein handelndes Objekt. Seine Eigenschaften sind der Name einer Handlung, die Position des Affen – hier: bei der Tür, beim Fenster und in der Mitte des Raums, die Position der Kiste und der Test, ob der Affe auf der Kiste steht. Die Position der Banane ist implizit als in der Mitte des Raumes festgelegt. Der Plan ist erfolgreich, wenn der Affe in den Zustand kommt, daß er auf der Kiste steht, die in der Mitte des Raums ist. Seine Handlung ist dann das Greifen der Banane. Dieser Zielzustand wird dann vom Keller heruntergenommen mit **pop** und die darunterliegenden Zustände werden von oben nach unten ausgedruckt und dann vom Keller genommen. Dadurch ergibt sich ein Protokoll der Zustände, die – vom Ziel zum Start – im richtigen Plan aufeinander folgen. Das Erfolgskriterium wird von der Methode **tryGrasp** realisiert.

Die eigentliche Planung erledigt **tryAll**. Ein Zustand legt sich selbst oben auf den Keller und versucht alle möglichen Handlungen, zuerst die Handlung, die den Erfolg feststellt (**tryGrasp**). Wenn alle Handlungen durchprobiert sind, entfernt sich der Zustand wieder.

Die möglichen Handlungen sind die Methoden

- **tryClimbBox** gelingt, wenn der Affe bei der Box ist und nicht auf ihr steht. Es wird ein Nachfolgezustand erzeugt, bei dem er auf der Kiste steht. Dieser neue Zustand wendet wieder seine Planungsmethode **tryAll** auf sich an.

- **tryPushBox** gelingt, wenn der Affe bei der Kiste ist und nicht auf ihr steht. Ist diese Position nicht die Mitte des Raumes, wird der neue Zustand erzeugt, daß der Affe auf der Kiste in der Mitte des Raumes ist. Wenn Affe und Kiste schon in der Mitte des Raumes sind, wird für die beiden anderen Positionen (Tür und Fenster) je ein Nachfolgezustand konstruiert. Jeder Zustand ruft für sich wieder die Planung auf.

- **tryWalk** gelingt, wenn der Affe nicht auf der Kiste steht. Ist er in der Mitte des Raums, sind die beiden Nachfolgezustände am Fenster und an der Tür. Ist er nicht

in der Mitte, ist er es im neuen Zustand. Wieder ruft jeder neue Zustand für sich die Planung auf.

Was wir dann noch brauchen, ist eine Ausgabe für die Benutzerin und die **main**-Methode, die einen neuen Zustand, den Startzustand, erzeugt und einen neuen Keller. Der Startzustand ruft für sich die Planung auf mit dem neuen, leeren Keller als Argument.

Und so sieht dieses Vorgehen in JAVA aus:

```
import java.util.*;


class State
{
        static final int AT_DOOR = 0;
        static final int AT_WINDOW = 1;
        static final int MIDDLE = 2;

        boolean onBox;
        int position;
        int boxPosition;

        String action;

        State (String _action, int _position, int _boxPosition, boolean _onBox)  {
                action = _action;
                position = _position;
                boxPosition = _boxPosition;
                onBox = _onBox;
        }

        void tryAll (Stack plan)  {
                plan.push (this );

                tryGrasp (plan);
                tryClimbBox (plan);
                tryPushBox (plan);
                tryWalk (plan);

                plan.pop ();
        }

        void tryGrasp (Stack plan)  {
                if (position == MIDDLE && onBox)  {
                        System.out.println ("got the banana!");

                        while (!plan.empty ())
                        System.out.println (plan.pop ());

                        System.exit (0);
                }
        }
```

```
        void try ClimbBox (Stack plan) {
                if (position == boxPosition && !onBox)
                new State ("ClimbBox", position, boxPosition, true).tryAll(plan);
        }

        void tryPushBox (Stack plan) {
                if (position == boxPosition && !onBox) {
                        if (position != MIDDLE)
                        (new State ("PushBox", MIDDLE, MIDDLE, onBox)).tryAll (plan);
                        else {
                                (new State ("PushBox", AT_WINDOW, AT_WINDOW,
                                                                        onBox)).tryAll (plan);
                                (new State ("PushBox", AT_DOOR, AT_DOOR, onBox)).tryAll (plan);
                        }
                }
        }

        void tryWalk (Stack plan) {
                if (!onBox) {
                        if (position != MIDDLE)
                        (new State ("Walk", MIDDLE, boxPosition, onBox)).tryAll (plan);
                        else {
                                (new State ("Walk", AT_WINDOW, boxPosition, onBox)).tryAll (plan);
                                (new State ("Walk", AT_DOOR, boxPosition, onBox)).tryAll (plan);
                        }
                }
        }

        static public String posToString (int pos) {
                switch (pos) {
                        case AT_DOOR: return "at door";
                        case AT_WINDOW: return "at window";
                        case MIDDLE: return "in the middle";
                }
                throw new RuntimeException ("Illegal Position: "+pos);
        }


        public String toString () {
                return action + ": "+ "monkey "+ posToString (position) + ", "
                          + (onBox ? "on box; ": "not on box; ")
                          + "box "+ posToString (boxPosition);
        }
}


class Monkey {
        public static void main (String argv []) {
                new State ("Start", State.AT_DOOR, State.AT_WINDOW, false).tryAll (new Stack());
        }
}
```

Die Ausgabe des Programms ist:

```
got the banana!
ClimbBox: monkey in the middle, on box; box in the middle
PushBox: monkey in the middle, not on box; box in the middle
Walk: monkey at window, not on box; box at window
Walk: monkey in the middle, not on box; box at window
Start: monkey at door, not on box; box at window
```

Da die Startsituation fest ist, ergibt sich immer dasselbe Verhalten. Aber darum geht es hier nicht. Vielmehr soll gezeigt werden, daß Keller sehr gut zur Planung verwendet werden können. Wir können das Planungsproblem so beschreiben:

Gegeben: ein Anfangszustand, ein Zielzustand und eine Menge von Handlungen mit Vorbedingungen und einem Nachfolgezustand.

Finde: eine Folge von Handlungen, die vom Anfangs- in den Zielzustand führt.

Der Keller liefert als oberstes Element den aktuellen Zustand. Eine Handlung erzeugt einen neuen Zustand, der oben auf den Keller gelegt wird, wenn "der handelnde Zustand" die Vorbedingung erfüllt. Ansonsten bleibt alles beim alten und die nächste Handlung muß probiert werden. Es ist nun interessant zu untersuchen, ob es womöglich unendliche Zyklen von Handlungsfolgen gibt. Die Möglichkeit unendlicher Zyklen wird insbesondere durch unterscheidende Vorbedingungen ausgeschlossen. Wenn die Vorbedingung einer Handlung in keinem Nachfolgezustand dieser Handlung gilt, kann sie nicht noch einmal ausgeführt werden. Die Vorbedingungen der Handlungen gelten im Beispiel meist für Mengen von Situationen, die nichts gemeinsam haben. Nur die Vorbedingung von **tryWalk** wird auch von Situationen erfüllt, die auch die Vorbedingungen anderer Handlungen erfüllen. Daher wird **tryWalk** als letzte Methode in **tryAll** angewandt. Obendrein erfüllt der Nachfolgezustand wieder die Vorbedingung von **tryWalk**. Im Affe und Banane Beispiel ist **tryWalk** und **tryPushBox** durch den Bezug auf die Mitte so implementiert, daß der Affe nicht hin- und hergehen kann. Zur Planung gibt es in der *Künstlichen Intelligenz* eine Fülle von Ansätzen und Kriterien zu ihrer Bewertung. Wenn es einen erfolgreichen Plan gibt, findet mein Verfahren ihn und endet dann? Endet mein Verfahren auch, wenn es keinen erfolgreichen Plan gibt? Wenn es mehrere erfolgreiche Pläne gibt, findet mein Verfahren den besten? Wie genau muß ich die möglichen Zustände zur Planungszeit wissen oder kann ich sie während der Ausführungszeit verarbeiten? In dieser Vorlesung können wir uns nicht mit diesen Fragen beschäftigen.

4.6 Rekursion

Die Rekursion ist eine Art und Weise, ein Problem zu formulieren. Wir haben die Grundidee der Rekursion bereits beim Induktionsbeweis kennengelernt: ein Problem wird zerlegt in einen Anfang und einen Induktionsschritt. Der Induktionsschritt konstruiert eine Kette vom Anfang $S(0)$ zu einem beliebigen $S(a)$, wobei wir nur den Schritt von $S(n)$ nach $S(n+1)$ notieren und beweisen müssen. Das Wort "Anfang" ist dabei vielleicht irreführend. Eigentlich geht es um das Erfolgskriterium, dem wir entnehmen, ob wir die Lösung gefunden haben, den Beweis fertiggestellt haben. Wir überlegen immer zuerst, unter welchen Bedingungen wir fertig sind. Dann konstruieren wir analog zu Induktionsschritt eine Folge

von Fällen, von denen jeder von vorigen Fällen abhängt. Wir schreiben eine Methode, die einen Teil des Problems behandelt und sich selbst für den restlichen Teil des Problems wieder aufruft. Wichtig ist dabei, daß jeder neuerliche Aufruf der Methode ein kleineres Problem bewältigt und daß es eine Folge von Aufrufen derselben Methode gibt, die zu dem Zustand führt, in dem das Erfolgskriterium erfüllt ist. Das Schema rekursiver Programmierung ist:

- Was hätte ich gern? – Präzise Formulierung des Erfolgskriteriums!
- Wie reduziere ich das Problem? – Kern der Methode
- Rekursiver Aufruf mit dem reduzierten Problem.

Der rekursive Aufruf kann *direkt* sein, d.h. die Methode ruft sich selbst auf. Wir haben ein Beispiel bereits gesehen in dem Beispiel für die Sichtbarkeit lokaler Variablen in Abschnitt 3.8. Sehen wir uns die Methode **studieren** noch einmal an:

- Ich hätte gern mein Diplom. Als Erfolgskriterium wird nun einfach das Erreichen des 9.Semesters (oder eines höheren) angegeben. Ist dies Kriterium erfüllt, gibt es das Diplom und die Methode ist beendet.
- Das Problem wird durch das Verstreichen von Semestern Der Kern der Methode ist die Schleife und dann das Erhöhen des Jahres um 1.
- Mit dem neuen Jahr und dem Monat 1 ruft sich die Methode selbst auf.

Die Rekursion besteht hier in einem einzigen Aufruf der Methode innerhalb der Methode. Die Methode wird *linear-rekursiv* genannt. Obendrein erfolgt der rekursive Aufruf im letzten Schritt der Methode. Deshalb heißt diese Rekursion *Endrekursion*. Man kann linerar-rekursive Methoden ganz einfach in Schleifen umwandeln. Dann ist die umgewandelte Methode nicht mehr rekursiv, sondern *iterativ*. Wir müssen dazu nur das, was wir reduzieren – hier: die Semester – als Schleife verwenden und die Abbruchbedingung als Schleifenabbruch schreiben. Die iterative Fassung der Methode **studieren** sieht so aus:

```
public void studierenI () {
      while (9 > this.semester) {                          // Studienende noch nicht erreicht?
                                                                  //dann weiterstudieren
            for (int i=this.monat; 13>i; i++) {                    // Monate zaehlen
                  if ((i!=this.monat) && (i==4 | i==10)) {         // Semester zaehlen
                        this.semester++;
                        System.out.println (this.name+" ist "+this.jahr+" im "
                                          +semester+". Semester");
                  }
            }
            this.jahr++;                                           // Jahre zaehlen
            this.monat=1;
      }

      System.out.println ("Und jetzt das Diplom!");                // sonst Diplom
}
```

Programmiersprachen wie PROLOG oder LISP arbeiten meist mit Endrekursion, die vom Übersetzer intern in eine Schleife umgesetzt wird.

Da eine aufgerufene Methode, sobald sie fertig ist, die Kontrolle wieder an die aufrufende Stelle abgibt, erfolgt ganz von allein nach der schrittweisen Reduktion des Problems die schrittweise Konstruktion der Lösung. Im **studieren**-Beispiel war da nichts zu konstruieren. Aber erinnern Sie sich an den Affen und die Banane? Dies ist ein Beispiel für die *indirekte Rekursion*. Die Methode **tryAll** ruft andere Methoden auf, die wiederum – von einem neuen Zustand aus, also mit hoffentlich verkleinertem Problem – **tryAll** aufruft. Das Problem wird solange reduziert, bis das Erfolgskriterium **tryGrasp** erreicht ist. Es folgt dann die Konstruktion der Lösung, hier: durch Ausdrucken und Entfernen des jeweils obersten Kellerelementes. Beim rekursiven Abstieg, d.h. dem schrittweisen Verkleinern des Problems, legen wir immer neue Zustände auf den Keller. Bei rekursiven Aufstieg, d.h. dem Zusammenbau der Lösung, entfernen wir einen Zustand nach dem anderen.

Rekursion ist ein äußerst vielseitig einsetzbarer Denkstil. So kann man ein Problem in zwei Hälften aufteilen und dann jede Hälfte mit derselben Methode, die das Problem immer in zwei Hälften zerlegt, aufrufen. Nehmen wir z.B. das aus der kognitiven Psychologie bekannt gewordene Beispiel der Türme von Hanoi.

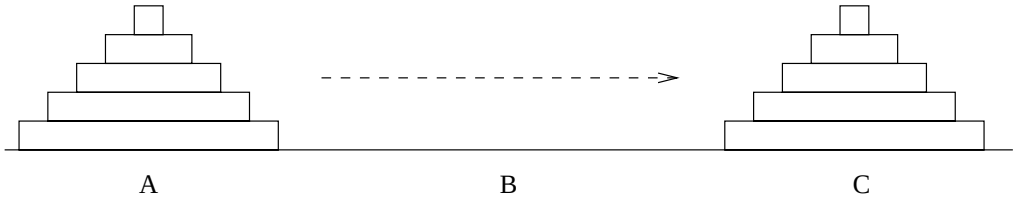


Abbildung 20: Türme von Hanoi

Die Aufgabe besteht darin, einen der Größe nach geordneten Stapel von Holzscheiben von einem Feld auf ein anderes zu bringen, wobei niemals eine größere Scheibe auf einer kleineren liegen darf, immer nur eine Scheibe auf einmal bewegt werden darf, aber ein Zwischenfeld zur Hilfe genommen werden kann. Es hat sich nun gezeigt, daß die einfache Problemlösungsstrategie, den Unterschied zwischen dem aktuellen Zustand und dem Zielzustand zu reduzieren, bei diesem Problem nicht zum Erfolg führt. Die Versuchspersonen in einem kognitionspsychologischen Experiment wandten zuerst diese fruchtlose Strategie an, bevor sie das Problem in Teilziele zerlegten und dann lösen konnten [Kotovsky et al., 1985]. Wir wollen die oberste Scheibe, nennen wir sie Scheibe 1, von Platz A nach Platz C bringen. Wenn darunter die anderen Scheiben richtig geordnet liegen, ist das der Erfolg. Wir müssen also nur noch den Stapel unter Scheibe 1 an den richtigen Platz in der richtigen Reihenfolge bringen.

Sehen wir uns hier die rekursive Lösung der Türme von Hanoi an:

- Das Problem ist gelöst, wenn wir die Scheibe 1 auf den geordneten Stapel auf Platz C legen.
- Wir reduzieren das Problem, indem wir immer kleinere Stapel von Scheiben betrachten.

- Das Problem, n Scheiben von A unter Verwendung von B nach C zu verlegen, läßt sich in zwei Probleme aufteilen: verlege $n-1$ Scheiben vom Start- zum Zwischenplatz und verlege $n-1$ Scheiben vom Zwischen- zum Zielplatz. Wir schreiben also eine Methode für n Scheiben, die sich selbst zweimal für $n-1$ Scheiben aufruft.

Da hier mehrere, nämlich zwei Aufrufe der Methode **verlege** innerhalb von **verlege** vorkommen, heißt die Rekursion hier *baumartig*.

Die Lösung der Türme von Hanoi entnehmen wir wieder [Vornberger und Thiesing, 1998].

```
import AlgoTools.IO;

/** Tuerme von Hanoi:
* n Scheiben mit abnehmender Groesse liegen auf dem Startort A.
* Sie sollen in derselben Reihenfolge auf Zielort C zu liegen kommen.
* Die Regeln fuer den Transport lauten:
* 1.) Jede Scheibe muss einzeln transportiert werden.
* 2.) Es darf nie eine groessere Scheibe auf einer kleineren liegen.
* 3.) Es darf ein Hilfsort B zum Zwischenlagern verwendet werden.
*/

public class Hanoi {

    static void verlege (                                   // drucke die Verlegeoperationen, um
    int n, // n Scheiben
    char start, // vom Startort
    char zwischen, // unter Zuhilfenahme eines Zwischenortes
    char ziel) {   // zum Ziel zu bringen
        if (n == 1)                                                     // Erfolgskriterium
        IO.println ("Scheibe 1 von "+ start + "nach "+ ziel);
        else {
            verlege (n-1,start, ziel, zwischen);                        //1. Abstieg
            IO.println ("Scheibe "+ n +"von "+ start + "nach "+ ziel);
            verlege (n-1,zwischen, start, ziel);                        //2. Abstieg
        }
    }

    public static void main (String argv[]) {
        int n;
        do {   n = IO.readInt ("Bitte Zahl der Scheiben (n>0): ");   } while (n <= 0);
        verlege (n,'A','B','C');
    }
}
```

Aufruf: java Hanoi
ergibt:

```
Bitte Zahl der Scheiben (n>0): 3
Scheibe 1 von A nach C
Scheibe 2 von A nach B
Scheibe 1 von C nach B
```

```
Scheibe 3 von A nach C
Scheibe 1 von B nach A
Scheibe 2 von B nach C
Scheibe 1 von A nach C
```

Es wird eine richtige Handlungsfolge ausgegeben. Um vielleicht besser zu sehen, wie es dazu kommt, sei hier noch einmal der zweifache rekursive Abstieg graphisch hervorgehoben. Auch soll die Parameterübergabe deutlich werden: auf einer Einrückungsebene haben die Variablen *start*, *zwischen* und *ziel* natürlich je *einen* Wert. Da beim Aufruf aber die Parameter verdreht werden, haben die Variablen auf der nächst tieferen Ebene andere Werte.

formale Aufrufe	Werte
verlege(n,Start,Zwischen,Ziel)	verlege(3,A,B,C)
? n=1	
verlege(n-1,Start,Ziel,Zwischen)	verlege(2,A,C,B)
≙ verlege(n,Start,Zwischen,Ziel)	verlege(2,A,C,B)
? n=1	
verlege(n-1,Start,Ziel,Zwischen)	verlege(1,A,B,C)
≙ verlege(n,Start,Zwischen,Ziel)	verlege(1,A,B,C)
?! n=1	
drucke 'Scheibe 1 von Start nach Ziel'	Scheibe 1 von A nach C
drucke 'Scheibe n von Start nach Ziel'	Scheibe 2 von A nach B
verlege(n-1,Zwischen,Start,Ziel)	verlege(1,C,A,B)
≙ verlege(n,Start,Zwischen,Ziel)	verlege(1,C,A,B)
?! n=1	
drucke 'Scheibe 1 von Start nach Ziel'	Scheibe 1 von C nach B
drucke 'Scheibe n von Start nach Ziel'	Scheibe 3 von A nach C
verlege(n-1,Zwischen,Start,Ziel)	verlege(2,B,A,C)
≙ verlege(n,Start,Zwischen,Ziel)	verlege(2,B,A,C)
? n=1	
verlege(n-1,Start,Ziel,Zwischen)	verlege(1,B,C,A)
≙ verlege(n,Start,Zwischen,Ziel)	verlege(1,B,C,A)
?! n=1	
drucke 'Scheibe 1 von Start nach Ziel'	Scheibe 1 von B nach A
drucke 'Scheibe n von Start nach Ziel'	Scheibe 2 von B nach C
verlege(n-1,Zwischen,Start,Ziel)	verlege(1,A,B,C)
≙ verlege(n,Start,Zwischen,Ziel)	verlege(1,A,B,C)
?! n=1	
drucke 'Scheibe 1 von Start nach Ziel'	Scheibe 1 von A nach C

Auch baumartig-rekursive Methoden können wir in iterative Methoden umwandeln. Jetzt reicht es aber nicht, einen Schleifenzähler zu verwenden, der über den rekursiven Abstieg Buch führt. Wir müssen einen Keller verwenden, auf dem wir der Reihe nach die vormaligen Aufrufe von **verlege** stapeln. Diese vormaligen Aufrufe sind nun Zustände. Im Prinzip wandeln wir das baumartig-rekursive **verlege** so um:

1. Wir beginnen mit dem ersten Aufruf, z.B. $3, A, B, C$ und legen ihn auf den Keller.

2. Wir lesen den obersten Kellereintrag, wenn es einen gibt, und treten in die Iteration ein (3).

3. (Iteration) Dann erzeugen wir die Nachfolgezustände und legen sie auch oben auf. Wir haben schon beim Kellerbeispiel vom Affen und der Banane gesehen, daß das Leerräumen des Kellers mit Ausdrucken die Zustände von hinten nach vorn ausgibt. Daher drehen wir die vormaligen Aufrufe $n-1, start, ziel, zwischen$ und $n-1, zwischen, start, ziel$ jetzt in der Reihenfolge um: erst $n-1, zwischen, start, ziel$, dann $n-1, start, ziel, zwischen$.

   Wenn wir keine Nachfolgezustände erzeugen können, weil die Abbruchbedingung $n == 1$ erfüllt ist, drucken wir diesen Zustand aus.

   Bei Beendigung der Iteration wird das bearbeitete Kellerelement $(n, start, zwischen, ziel)$ vom Keller genommen.

4. Wir nehmen das oberste Kellerelement und gehen zu (3), es sei denn, der Keller sei leer – dann sind wir fertig.

Diese Umwandlung von einem rekursiven in ein iteratives Programm baut einen Stapel auf, der die Aufrufe des rekursiven Programms, geordnet nach der Rekursionstiefe auf den Stapel legt. Im Beispiel ist
$3, A, B, C,$
$2, B, A, C,$
$print(3, A, B, C),$
$2, A, C, B$
der Keller nach der ersten Iteration. Dann wird $2, A, C, B$ bearbeitet, so daß der Keller nach der zweiten Iteration so aussieht:
$3, A, B, C,$
$2, B, A, C,$
$print(3, A, B, C),$
$1, C, A, B$
$print(2, A, C, B)$
$1, A, B, C$
Das iterative Programm für die Türme von Hanoi sieht so aus:

```
import java.util.*;
import AlgoTools.*;

class Hanoiiterativ {
        public static void main (String argv[]) {
                int n=IO.readInt("Bitte geben Sie die Anzahl der Scheiben an: ");
                new Zustand (false,n,'A','B', 'C').verlegen(new Stack ());
        }
}

class Zustand {
        int n;
        char start;
        char zwischen;
```

```
        char ziel;
        boolean drucken;

        public Zustand (boolean _drucken, int _n, char _start, char _zwischen, char _ziel) {
                n=_n;
                drucken=_drucken;
                start=_start;
                zwischen=_zwischen;
                ziel=_ziel;
        }

        void drucke () {
                System.out.println (n+"von "+start+"nach "+ziel);
        }

        public void verlegen (Stack keller) {
                keller.push (this );

                while (!keller.empty())
                ((Zustand)keller.pop ()).rekursionsersatz(keller);

                System.exit (0);
        }

        public void rekursionsersatz (Stack keller) {
                if (drucken || n == 1)
                drucke ();
                else  {
                        drucken=true;
                        keller.push (new Zustand (false, n-1, zwischen, start,ziel));
                        keller.push (this );
                        keller.push (new Zustand (false, n-1,start, ziel,zwischen));
                }
        }
}
```

Erfahrungsgemäß lernt man rekursives Denken am besten durch Übung. Deshalb wird auch im nächsten Abschnitt ein Problem rekursiv gelöst.

4.7 Sortierung durch Mischen

Die Sortierung haben wir im Abschnitt 4.1 bereits als Problemstellung kennengelernt. Dort wollten wir bereits einen fertigen Teil übergeben können, bevor alles sortiert ist. Jetzt ist uns diese Eigenschaft nicht so wichtig. Stattdessen betrachten wir das Sortierungsproblem einmal rekursiv:

- Wir möchten gern eine gemäß eines Ordnungskriteriums geordnete Folge von Objekten haben. Wir nehmen Zahlen und ihre >-Ordnungsrelation.
- Wir teilen die ungeordnete Menge in zwei Teile und rufen für jeden Teil unsere Sortiermethode auf.

- Wir mischen die beiden jede für sich geordneten Folgen, indem wir sie elementweise von links nach rechts vergleichen: bei jedem Schritt wird das kleinere Element der beiden in die Ergebnisfolge eingetragen.

In JAVA sieht das so aus:

```
public class AnimatedMergeSort  {

        public static void sort (int [] feld)  {                    // wrapper auf parameterisierten
                sort (feld, 0, feld.length);                                          // aufruf
        }

        public static void sort (int [] feld, int unten, int oben) {
                if (unten < oben - 1)  {                                        // noch was zu tun?
                        int mitte = (unten + oben) / 2;                                  // ja, split

                        sort (feld, unten, mitte);                                 // beide teilfolgen
                        sort (feld, mitte, oben);                                 // rekursiv sortieren

                        merge (feld, unten, mitte, oben);                 // sortierte teilfolgen mischen
                }
        }

        public static void merge (int[] feld, int unten, int mitte, int oben)  {

                int i = unten, j = mitte, k=0;                                          // Laufindizes
                int[] ergebnis = new int[oben - unten];            // Platz fuer Ergebnisfolge besorgen

                while ((i < mitte) && (j < oben))  {                    // mischen, bis ein Feld leer
                        if (feld[i] < feld[j])                          // jeweils das kleinere Element
                        ergebnis[k++] = feld[i++];                // wird nach ergebnis uebernommen
                        else
                        ergebnis[k++] = feld[j++];
                }

                if (i == mitte)                                                 // falls i am Ende:
                while (j<oben) ergebnis[k++] = feld[j++];          // Rest von oben uebernehmen
                else                                                            // falls j am Ende:
                while (i<mitte) ergebnis[k++] = feld[i++];         // Rest von unten uebernehmen

                for (i = 0; i < ergebnis.length; i++)              // Ergebnis in Feld zurueckkopieren
                 {
                        feld [i + unten] = ergebnis [i];
                        ShowArray.show (feld, unten, oben-1, unten+i);
                }
        }
}
```

4.8 Was wissen Sie jetzt?

Sie wissen, was ein abstrakter Datentyp ist. Im einzelnen haben Sie Listen, Schlangen und Keller kennengelernt. Sie wissen, durch welche Operationen und Bedingungen diese abstrakten Datentypen definiert sind und Sie haben gesehene, wie man sie in JAVA implementiert. Dadurch haben Sie jetzt ein gewisses Handwerkszeug für die Programmierung gewonnen: bei einer neuen Aufgabenstellung überlegen Sie, ob sie sich (in Teilen) als Liste oder Schlange oder Keller auffassen läßt? Wenn ja, nehmen Sie die genau studierte JAVA-Implementiertung als Vorlage und programmieren das neue Problem "nach Buch". Üben Sie sich im Alltag darin, verkappte Listen, Schlangen und Keller zu finden! Beobachten Sie Ihre eigene Arbeitsweise: gehen Sie mehr nach Warteschlange oder eher nach Keller vor? Wenn es so eine neuartige Mischung ist, wie könnte man sie in das Schlangen- oder Kellerschema einbauen?

Sie haben die elegante Formulierung von Problemlösungen durch Rekursion kennengelernt. Gehen Sie mit KommilitonInnen ins Cafe und diskutieren, ob Rekursion einfacher oder schwieriger ist als eine feste Folge von Handlungen. Diskutieren Sie auch, welches Verhältnis zwischen Schleifen (Iterationen) und Rekursion besteht! Bedenken Sie dabei, daß die Informatik lebendigen Stoff bietet, über den jede/r nachdenken kann. Auswendig gelernt werden müssen nur die Schritte der Rekursion wie hier angegeben. Wenn Sie aber diese Denkweise in Ihren Alltag integrieren, können Sie sie ohnehin aufsagen.

4.9 Aufwandsabschätzung

Wenn wir ein Programm geschrieben und ausprobiert haben, können wir überlegen, ob es korrekt ist. Dazu können wir Aussagen überlegen und sie mithilfe des Induktionsverfahrens beweisen (s. Abschnitt 4.1.3). Wir können uns aber auch fragen, wie lang unser Programm wohl braucht, um die Lösung zu finden? Die absolute Zeit hängt natürlich von unserem Rechner und dem Übersetzer der Proagrmmiersprache ab. Der Übersetzer produziert aus den JAVA-Anweisungen eine bestimmte Anzahl von Anweisungen der JAVA virtuellen Maschine und diese wird wiederum von einer Plattform (Rechner, Betriebssystem) in eine bestimmte Anzahl von Maschinenbefehlen übersetzt. Damit kann dasselbe JAVA-Programm auf unterschiedlichen Rechnern in unterschiedlich viele Maschinenbefehle übersetzt werden. Obendrein ist die Ausführung eines Maschinenbefehls auf verschiedenen Rechnern unterschiedlich schnell. Wir wollen, wenn wir die Laufzeit unseres Programms untersuchen, von diesen Faktoren abstrahieren. Wir untersuchen die Laufzeit in Bezug auf die Größe der zu verarbeitenden Daten. Insbesondere fragen wir uns, in welchem Verhältnis die Laufzeit mit der Größe der Eingabe wächst. Wir unterscheiden die folgenden Verhältnisse der Laufzeit $T(n)$ zur Größe n der Eingabe, wobei c und $k \geq 2$ irgendwelche Konstanten sind:

konstant: $T(n) = c$

logarithmisch: $T(n) = c \cdot \log n$

linear: $T(n) = c \cdot n$

$n \log n$**:** $T(n) = c \cdot n \log n$

quadratisch: $T(n) = c \cdot n^2$

polynomiell: $T(n) = c \cdot n^k, \quad k \geq 2$

exponentiell: $T(n) = c \cdot 2^n$

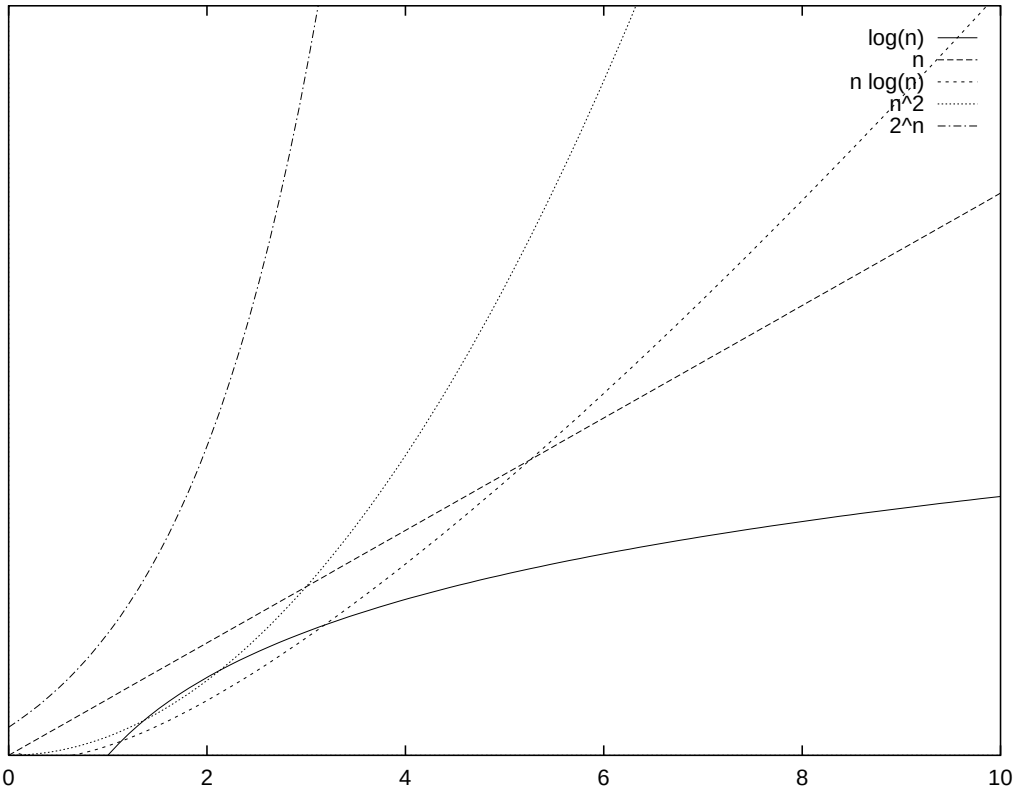


Abbildung 21: Laufzeiten verschiedener Programme

Die Abbildung 21 zeigt, wie sich die Laufzeiten verschiedener Programme in Bezug zur Größe n verhalten. Meist ist n die Anzahl der Daten in der Eingabe, aber es kann auch etwas anderes sein, z.B. die Länge der Eingabe in Bits, oder die Länge der Ausgabe. Wir sagen "Programm XYZ ist quadratisch in" und geben dann unsere Formalisierung der Größe n an. Ist bei der linearen Laufzeit die Konstante c recht groß, so ist ein Programm mit quadratischer Laufzeit bei kleinen Werten von n schneller. Ab dem Punkt, an dem sich die Linien schneiden, ist das lineare Programm schneller und der Abstand wächst. Auch die lineare und die quadratische Laufzeit unterscheiden sich deutlich erst ab einer bestimmten Größe. Achten Sie bei Diagrammen zur Laufzeit auch auf die Skalierung der Achse für n – gerade exponentielle Programme werden gern mit logarithmischer Skalierung dargestellt.

Beispiel 4.1: Nehmen wir an, Programm A hätte die Laufzeit $100 \cdot n$ und Programm B hätte die Laufzeit $2 \cdot n^2$. Dann ist B schneller als A, solange $n < 50$. Für $n = 100$ ist A schon doppelt so schnell wie B. Für $n = 1000$ ist A 20mal so schnell.

Es gibt nun in der Informatik fünf Klassen von Funktionen, die den Aufwand eines Programms (Zeit- oder Platzbedarf) nach oben und unten beschränken. Sie werden im weiteren Verlauf die Studiums alle fünf kennenlernen. Hier sehen wir uns nur die O-Notation an. Sie gibt den schlimmsten Fall an.

Definition 4.5: Aufwand O Die Laufzeit eines Programms wird durch eine Funktion $T(n)$ über der Größe n (der Eingabe) angegeben. n und $T(n)$ sind nichtnegative (meist ganze) Zahlen. $f(n)$ sei eine Funktion, die über nichtnegativen (ganzen) Zahlen definiert ist. Wir sagen, " $T(n)$ ist $O(f(n))$", wenn es eine Zahl n_0 gibt und eine Konstante $c > 0$, so daß für alle Zahlen $n \geq n_0$ gilt $T(n) \leq c \cdot f(n)$.

Wenn wir nun den Aufwand abschätzen wollen, müssen wir irgendein n_0 und irgendein c auswählen und beweisen, daß $T(n) \leq c \cdot f(n)$ für alle $n \geq n_0$ ist. [18] Nehmen wir an, $T(n) = (n+1)^2$, dann ist $T(n)$ quadratisch, d.h. $O(n^2)$, denn wir können $c = 2$ und $n_0 = 3$ auswählen oder $c = 4$ und $n_0 = 1$. Es gilt sowohl
$T(n) = (n+1)^2 \leq 2 \cdot n^2 = 2 \cdot f(n)$ als auch
$T(n) = (n+1)^2 \leq 4 \cdot n^2 = 4 \cdot f(n)$.
Man sieht dies besonders leicht, wenn man die Formel anwendet:
$(n+1)^2 = n^2 + 2 \cdot n + 1$
Dann ist nämlich $T(n)$ sichtbar kleiner als unser $c \cdot f(n)$:
$T(n) = n^2 + 2 \cdot n + 1 \leq n^2 + 2 \cdot n^2 + n^2 = 4 \cdot n^2$.
Natürlich darf n_0 nie 0 sein, weil jede Multiplikation mit n_0 dann 0 ergibt.

4.9.1 Aufwandsabschätzung für die Sortierung durch Mischen

Wie finden wir für ein bestimmtes Programm heraus, in welche Klasse sein Laufzeitverhalten und sein Bedarf an Speicherplatz fällt gemäß der O-Notation? Wir betrachten die Programmschritte ungeachtet der realen Ausführungszeit. Als Beispiel analysieren wir jetzt die Sortierung durch Mischen, eine Methode nach der anderen. Die Methode **merge** ist nicht rekursiv, sondern besteht aus drei Schleifen: die erste *while*-Schleife, die bedingte *while*-Schleife und die *for*-Schleife. Wir betrachten die Größenordnung des Aufwands in Bezug zur Länge der Felder. Wir nehmen n als Feldlänge.

- Wie beim Induktionsbeweis betrachten wir zuerst den Basisfall: wie ist der Aufwand bei $n = 1$?
    - Als erstes sehen wir in der *while*-Schleife die Abbruchsbedingung. Wenn sie falsch ist, also ein Feld leer ist, dann sind wir mit dieser Schleife fertig. Bei nur einem Element ist sicherlich eines der Felder leer. Der Aufwand für die Schleife war nur die eine Handlung, die Bedingung zu testen, also $O(1)$.
    - Es ist dann auch in der bedingten Schleife nur eines der Felder mit seinem Rest in das Ergebnis zu übernehmen. Wieviele Schritte dies sind, hängt von der Länge des Feldes ab. Im Basisfall also 1. Wir haben wieder $O(1)$.
    - Die *for*- Schleife hängt ebenfalls direkt von der Feldlänge ab, ist also ebenfalls nur einmal durchzuführen. Wir haben drei Mal den Aufwand $O(1)$ im Basisfall.

[18] Ich übernehme hier Argumentation und Beispiel von [Aho und Ullman, 1996].

Das macht insgesamt $O(1)$, denn es geht ja um die Funktionsklasse und nicht um das Zählen der tatsächlichen Schritte. $T(1)$ ist $O(1)$.

- Nehmen wir nun ein größeres n.
  - Es kann auch jetzt eines der Felder leer sein. Wenn das eine Feld immer die kleineren Elemente hatte als das andere, so ist es leer und das andere noch nicht. Wir haben dann – unabhängig von der Feldlänge – den Aufwand $O(1)$ für den (erfolgreichen) Test. Solange kein Feld leer ist, müssen wir das kleinste Element auswählen. In der Schleife ist dies ein konstanter Aufwand, der nicht von n abhängt. Wie oft wir die Schleife durchlaufen, hängt von n ab. Wir zählen den Feldindex immer um 1 hoch, so daß wir jedes Mal mit einem um 1 verkürzten Feld erneut in die Schleife eintreten bis eines von beiden leer ist. Dies ergibt also im schlimmsten Falle $n-1$ Schritte ($n=1$ hatten wir schon). Die erste Schleife ist also $O(1)+T(n-1)$.
  - Die bedingte Schleife muß nun den Rest des Feldes in das Ergebnis übernehmen. Im schlimmsten Fall muß das eine Feld ganz durchgegangen werden. Damit haben wir nochmals $n-1$ Schritte.
  - Die dritte Schleife geht auf jeden Fall das gesamte Feld, mit dem wir **merge** aufgerufen haben, durch. Auch hier haben wir wieder $n-1$ Schritte.

  Wir setzen die Abschätzungen der Schleifen für $n \geq 1$ zusammen und erhalten: $T(n) = O(1) + T(n-1)$. Also ist der Aufwand von **merge** $O(n)$.

Aber wir rufen ja **merge** von **sort** aus auf, indem wir immer wieder das Gesamtfeld in zwei Teile teilen. Jetzt wollen wir $T(m)$ mit m als der Länge des gesamten, ursprünglichen zu sortierenen Feldes betrachten – ist das immer noch linear in der Feldlänge?

- Als Basisfall nehmen wir wieder $m=1$. Dieses winzige Feld wird nur in **sort** mit dem ersten Test bearbeitet, der negativ beantwortet wird. **merge** wird nicht aufgerufen. Wir haben $T(1)$ ist $O(1)$.

- Bei $m > 1$ müssen wir nun das Aufteilen betrachten. Dies ist rekursiv. Solange nicht ein einelementiges Feld durch das Aufteilen entsteht, wird die Feldlänge immer durch 2 geteilt. Für jede der beiden Felder wird **sort** wieder aufgerufen und dann **merge**. Der Aufwand von **merge** war für die geteilten Felder $O(n)$. Nun ist $n = \frac{m}{2}$. Wir haben folglich den Aufwand $T(\frac{m}{2})$ zweimal. Weil hier die Anzahl der Schritte von der Feldlänge abhängt, müssen wir auch $2 \cdot T(\frac{m}{2})$ angeben. Das Ergebnisfeld ist wieder so lang wie das Ausgangsfeld, also $O(m)$. Wir erhalten für $m > 1$:

  $$T(m) = 2 \cdot T(\tfrac{m}{2}) + m, \qquad \text{(Rekursion)}$$

  wobei m eine Potenz von 2 ist (wir teilen ja immer durch 2). Wir merken uns dies als Beschreibung für den Rekursionsschritt.

Was für ein Verhältnis ist das? Wir sehen sofort, daß es nicht linear und nicht exponentiell ist. Quadratisch sieht es auch nicht aus. Versuchen wir einfach, zu beweisen, daß der

Aufwand logarithmisch ist! [19]

Als Beispiel, das die Vorstellung unterstützt, nehmen Sie:
Feldlänge = 8
Wir teilen das 1. Mal auf und erhalten
2· Feldlänge = $2 \cdot 4$
Wir teilen das 2. Mal auf und erhalten
4· Feldlänge = $4 \cdot 2$
Wir teilen das $\log_2 8 = 3$. Mal auf und erhalten
8· Feldlänge = $8 \cdot 1$.

Basis $T(1) = a$

Induktion $T(n) = 2 \cdot T(\frac{n}{2}) + bn$, wobei n eine Potenz von 2 ist.

Jetzt raten wir, daß $f(n) = c \cdot n \cdot log_2 n + d$, wobei c und d Konstanten sind. Und dann beweisen wir, daß $T(n) \leq f(n)$ mit vollständiger Induktion über n.

Aussage $S(n)$: Wenn n eine Potenz von 2 ist und $n > 1$ gilt, dann ist $T(n) \leq f(n)$ mit $f(n) = c \cdot n \cdot log_2 n + d$.

Induktionsanfang Wenn $n = 1$ gilt $T(1) \leq f(1)$, falls $a \leq d$ ist. $f(1) = d$ weil $c \cdot 1 \cdot log_2 1$ gleich 0 ist.

Induktionsschritt Nehmen wir an, wir hätten bis $n - 1$ die Aussage schon bewiesen. Dann müssen wir jetzt den nächsten Schritt beweisen, $S(n)$.

Wenn n keine Potenz von 2 ist, gilt der "Wenn"-Teil der Aussage nicht und damit gilt $S(n)$ ohnehin.

Wenn n eine Potenz von 2 ist, nutzen wir unsere Annahme aus, daß alle früheren Schritte bereits bewiesen sind, also auch $S(n/2)$, d.h. es gilt

$$T(n/2) \leq (c \cdot n/2) \cdot log_2(n/2) + d \qquad \text{(Induktionsannahme)}$$

Nun müssen wir zeigen, daß

$T(n) \leq c \cdot nlog_2 + d$ gilt.

Die induktive Definition von $T(n)$ sagt, daß gilt

$$T(n) = 2T(n/2) + bn.$$

In dieser Gleichung setzen wir die Induktionsannahme für $T(n/2)$ ein.

Das ergibt

$$T(n) \leq 2((c \cdot n/2) \cdot log_2(n/2) + d) + bn.$$

Mit Hilfe von

$log_2(n/2) = (log_2 n) - 1$,

Ausmultiplizieren,

$2 \cdot (n/2) = n$,

$-c \cdot n + b \cdot n = (b - c) \cdot n$ vereinfachen wir die Ungleichung so:

[19] Es gibt auch eine sehr schöne Herleitung des genauen Aufwands in [Aho und Ullman, 1996]. Es gibt dort drei Beweise für die Komplexität der Mischsortierung.

$$\begin{aligned} T(n) &\leq 2((c \cdot n/2) \cdot log_2(n/2) + d) + bn \\ &= c \cdot n \cdot (log_2 n - 1) + b \cdot n + 2d \\ &= c \cdot n \cdot log_2 n - c \cdot n \cdot 1 + b \cdot n + 2d \\ &= c \cdot n \cdot log_2 n + b \cdot n - c \cdot n + 2d \end{aligned}$$

daß wir die folgende Ungleichung erreichen:

$$T(n) \leq c \cdot n \cdot log_2 n + (b - c) \cdot n + 2d.$$

Damit unsere Aussage gilt, muß $(b - c)n + d \leq 0$ sein. $n > 1$, also muß $b - c \leq -d$ sein. Anders ausgedrückt muß gelten $c \geq b + d$.

Dies gilt bei $d = a$ und $c = a + b$, was auch zu der Beschränkung von d im Basisfall paßt, nämlich $a \leq d$.

Wir haben also durch Induktion über n gezeigt, daß für alle $n \geq 1$, die Zweierpotenzen sind, gilt:

$$T(n) \leq (a + b) \cdot n \cdot log_2 n + a.$$

Damit gehört der Aufwand für die Mischsortierung in die Klasse $O(n \log n)$.

4.10 Schnellsortierung

Die Schnellsortierung illustriert noch einmal das rekursive Programmieren in JAVA. Außerdem gibt es hier Gelegenheit, den Unterschied zwischen der Aufwandsabschätzung für den schlimmsten Fall und dem am häufigsten beobachteten Laufzeitverhalten zu diskutieren. Wie bei der Sortierung durch Mischen wird auch bei der Schnellsortierung das Feld in jeweils 2 Felder aufgeteilt, die dann wieder sortiert werden. Die Idee ist hier aber, daß diese Aufteilung nicht nur den Feldindex, sondern auch die Feldelemente berücksichtigen soll. Jedes Element der linken Hälfte des Feldes soll schon einmal kleiner sein als alle Elemente der rechten Hälfte. Dann sortiert man die linke und die rechte Hälfte gerade so wie das ganze Feld (rekursiver Aufruf der Methode **sort**). Diese Grundidee wird so präzisiert: Man nehme ein Element in der Mitte des Feldes und nenne es x (s. Abbildung 22).

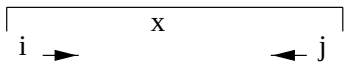


Abbildung 22: Zwei Zähler in QuickSort

Ein Zähler i wird an den Anfang des Feldes gesetzt, ein Zähler j an das Ende. Nun läuft i das Feld hinauf und vergleicht jedes Element mit x. Solange die Elemente, auf die i zeigt, kleiner sind als x, rückt der Zeiger weiter vor bis j. Ist eines größer als x, beginnt j das Feld hinunterzulaufen. Solange die Elemente, auf die j zeigt, größer sind als x, läuft j weiter bis i. Ist aber eines kleiner als x, dann werden die Inhalte, auf die i und j zeigen, vertauscht. Treffen sich i und j, wird die Sortierung mit den beiden Feldern links und rechts des Felds, an dem sich i und j getroffen haben, wieder aufgerufen. Wenn sie sich immer genau in der Mitte treffen, dann gibt es keinen Unterschied in der Laufzeitabschätzung zwischen Schnellsortierung und Sortieren durch Mischen. Der Aufwand ist in diesem Falle $O(nlogn)$ bei n Elementen. Im Gegensatz zur Mischsortierung kann es hier aber vorkommen, daß

i nur wenige Elemente das Feld aufwärts gegangen ist und schon größer ist als x. Dann wird nur ein Element in die untere Hälfte getan und j bis Feldende enthält alle anderen. Nehmen wir als Beispiel das Feld

20,30,10,24,50

Sei $x = 10$, dann bricht schon beim ersten Vergleich i seinen Lauf ab. j kommt bis x. Dann werden die Elemente getauscht. Das ergibt:

10,30,20,24,50

Nun besteht das linke Feld nur aus dem Element 10 und das rechte Feld aus allen anderen Elementen. Das rechte Feld soll schnell sortiert werden.

30,20,24,50

$x = 20$. i bricht beim ersten Vergleich ab. j kommt bis x. Die Elemente werden getauscht, so daß das Feld nun so aussieht:

20, 30, 24, 50

Wieder besteht das linke Feld nur aus einem Element. Das rechte Feld wird sortiert.

30, 24, 50

$x = 24$ und i scheitert wieder beim ersten Versuch, so daß noch einmal getauscht werden muß. j durchläuft fast das gesamte Feld fast so oft, wie es Elemente hat (s. Abbildung 23).

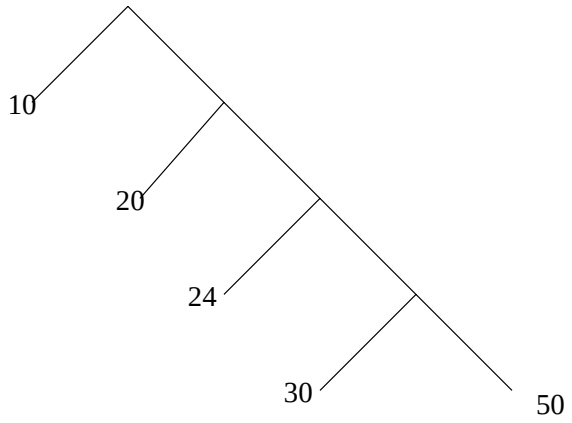


Abbildung 23: Der für QuickSort ungünstige Fall

Der schlimmste Fall der Laufzeitabschätzung ist $O(n^2)$.

Die folgende Implementierung zeigt das Verhalten auf dem Bildschirm an. Die graphische Darstellung zeigt die Elemente des Feldes als Punkt in dem Koordinatensystem, dessen x-Achse den Feldindex (0 bis Feldlänge -1) und dessen y-Achse das Feldelement (eine ganze Zahl) angibt. Die senkrechten Striche sind die Zähler i und j.

```
/** Rekursives Sortieren mit Quicksort
* Idee: partitioniere die Folge
* in eine elementweise kleinere und eine elementweise groessere Haelfte
* und sortiere diese nach demselben Verfahren
*/

public class AnimatedQuickSort {
```

```
    public static void sort (int [] a)
    {
        sort (a, 0, a.length - 1);

        ShowArray.show (a);
    }

    public static void sort (int[] a, int unten, int oben)  {
        int tmp;
        int i = unten;
        int j = oben;
        int x = a[ (unten+oben) / 2];                          // Pivotelement, willkuerlich

        do  {
            while (a[i] < x)
            {
                i++;                                           // x fungiert als Bremse
                ShowArray.show (a, i, j);
            }

            while (a[j] > x)
            {
                j--;                                           // x fungiert als Bremse
                ShowArray.show (a, i, j);
            }

            if ( i<=j )  {
                tmp = a[i];                                            // Hilfsspeicher
                a[i] = a[j];                                                 // a[i] und
                a[j] = tmp;                                        // a[j] werden getauscht
                i++;
                j--;
            }
        }  while (i <= j);
                                          // alle Elemente der linken Haelfte sind kleiner
                                                // als alle Elemente der rechten Haelfte

        if (unten < j) sort (a, unten, j);                          // sortiere linke Haelfte
        if (i < oben ) sort (a, i, oben );                         // sortiere rechte Haelfte
    }

                        // Best case: O(n * log n), wenn Partitionen immer gleich gross sind
                    // Average case: O(n * log n), um den Faktor 1.4 schlechter als best case
                    // Worst case: O(n * n), wenn Partitionen entarten: 1 Elem. + n-1 Elem.
}
```

4.11 Was wissen Sie jetzt?

Vielleicht haben Sie sich ein bißchen erschrocken, als wir die Mischsortierung besprochen haben: so ein simples Programm und so eine komplizierte Aufwandsabschätzung! Ich

möchte Sie beruhigen: Sie werden innerhalb der theoretischen Informatik noch so viele Beweise sehen und selbst machen, daß sie ihren Schrecken verlieren. Für diese Vorlesung müssen Sie sich merken:

- Was ist die O-Notation? (Die richtige Antwort ist die Definition.)
- Welche Funktionsklassen unterscheiden wir? (Die richtige Antwort listet konstant, logarithmisch, linear, $n \log n$, quadratisch, polynomiell und exponentiell mit der jeweiligen Funktion auf.)
- Wann z.B. benötigt ein Programm einen Aufwand, der $O(n \log n)$ in der Anzahl der Eingabedaten ist? (Sie könnten antworten: wenn ein rekursives Verfahren die Gesamtmenge von n Daten immer in 2 gleichgroße Teile teilt, wobei für jeden Teil das Verfahren wieder aufgerufen wird, so gibt es $\log_2 n$ Aufrufe. Wenn das Bearbeiten eines Aufrufs linearen Aufwand erfordert, so ist der Gesamtaufwand $O(n \log n)$.)
- Kennen Sie ein Sortierverfahren mit Aufwand $O(n \log n)$? (Ja, Sortierung durch Mischen.)
- Warum ist die Schnellsortierung im schlimmsten Fall nicht $O(n \log n)$ und was ist der schlimmste Fall? (Das Feld ist so sortiert, daß die kleinsten (bzw. größten) Elemente in der Mitte liegen. Dann wird immer nur 1 Element in die eine "Hälfte" gepackt, so daß das Feld nicht logarithmisch, sondern linear zerlegt und bearbeitet wird. In diesem Fall ist der Aufwand $O(n^2)$.)
- Gibt die O-Notation genau die Anzahl der Schritte eines Programms an? (Nein.) Begründen Sie, wieso nicht.

Sie sollen hier ja nur die Programmierung nach zwei Richtungen hin verankern. Einmal in Richtung theoretischer Informatik und einmal in Richtung praktischer Informatik. Die Ankerplätze selbst lernen Sie dann im weiteren Verlauf des Studiums kennen.

4.12 Performanztest

Wir haben bereits zwei Methoden gesehen, Eigenschaften eines Programms zu beweisen: den Induktionsbeweis, bei dem wir eine Behauptung, was das Programm tut, begründen und die Aufwandsabschätzung, mit der wir Zeit- und Platzbedarf eines Verfahrens angeben. Beides sind Methoden, die sich auf das Verfahren, das wir programmiert haben, beziehen. Wir wollen nun das tatsächlich realisierte Programm prüfen. Verhält es sich so, wie die Aufwandsabschätzung angibt? Wenn es exponentiell viel Zeit verbraucht, obwohl das Problem eigentlich in polynomieller Zeit lösbar sein sollte, dann ist es schlecht programmiert. Wenn es in den meisten Fällen polynomiell viel Zeit verbraucht, obwohl es im schlimmsten Falle exponentiell viel Zeit benötigt, ist das kein Widerspruch. Es ist dann sinnvoll, genau herauszufinden, wie die schnellen sich von den langsamen Fällen unterscheiden. Meist führt dies wieder zu einem theoretischen Resultat: es wird ein leichteres Problem formalisiert und bewiesen, daß dieses nur polynomiell viel Zeit benötigt. In dieser Weise sind theoretische und praktische Arbeiten stets miteinander verwoben.

Das systematische Testen eines Programms besteht aus den folgenden Schritten:

- Was wollen Sie testen?

- Welche Daten brauchen Sie, um den Test durchzuführen?
- Wie erfassen Sie die Ergebnisse?
- Durchführen und Revidieren der Testläufe
- Interpretation und Kommunikation der Ergebnisse

Leider wird die erste Frage oft mit einer existenziell quantifizierten Aussage beantwortet: "Es gibt einen Aufruf und Datensatz, bei dem mein Programm tut, was es soll." oder "Es gibt einen Aufruf und Datensatz, bei dem mein Programm schnell ist." Ein systematischer Test muß aber stets eine all-quantifizierte Aussage unterstützen: "Mein Programm tut immer, was es soll." oder "Mein Programm liefert sein Ergebnis immer innerhalb von 10 Minuten." Ebenso bedauerlich sind unpräzise Formulierungen wie das "tut, was es soll". Präzisierungen verlaufen meist nach einem der folgenden Schemata:

1. Wollen Sie testen, ob das Programm eine festgelegte Anzahl von Fällen in der ebenfalls festgelegten Weise behandelt?
2. Wollen Sie testen, ob das Programm den Zeitrahmen einhält, der für verschiedene Aufgaben vorgegeben ist? So soll ein Roboter beispielsweise bei der Hindernisvermeidung in Sekundenbruchteilen ausweichen, darf bei der Wegeplanung ruhig 2 Minuten (aber nicht mehr!) verbrauchen und muß seine Fahrt neben einem Menschen her (dem er z.B. den Koffer trägt) auf das variierende Gehtempo des Menschen abstimmen können.
3. Wollen Sie testen, ob Ihr Programm mehr Fälle bearbeiten kann als ein anderes (z.B. das bisher weltbeste Programm zur selben Aufgabe)? Achten Sie darauf, daß Sie nicht einfach nur andere Fälle bearbeiten!
4. Wollen Sie testen, ob Ihr Programm schneller ist als ein anderes?
5. Wollen Sie das Laufzeitverhalten eines Programms in Bezug zur Größe der bearbeiteten Datenmenge testen? Definieren Sie präzise, was Sie unter der Größe verstehen!

Die Fragestellung, die ein Test beantworten soll, bestimmt implizit auch das Kriterium, mit dem das Ergebnis beantwortet werden soll.

- Ein binäres Kriterium stellt einfach fest, ob das tatsächliche Verhalten mit dem festgelegten Verhalten identisch ist – ja oder nein. Wir können dann feststellen, bei wieviel Prozent der Beispiele für einen Fall, das tatsächliche Verhalten verschieden vom festgelegten war. In der Statistik gibt es Resultate darüber, wieviele Beispiele für einen Fall notwendig sind, um diese Prozentzahl als Erwartungswert für einen Fehler zu interpretieren.
- Es gibt aber auch gleitende Kriterien, bei denen die Abweichung des tatsächlichen vom festgelegten Verhalten in eine reellwertige Zahl zwischen 0 und 1 abgebildet wird.
- Sie können auch komplexere Kriterien formulieren, wie etwa, daß der Erwartungswert für einen Fehler bei den selteneren Fällen höher ist, als bei den häufigen, oder daß die Abweichung bei den häufigen Fällen stets niedriger ist als bei den selteneren.

Wenn die Frage und das Kriterium zur Beurteilung klar formuliert sind, ergibt sich meist einfach die Menge der Daten, die Sie verwenden müssen. Die Daten müssen jene Faktoren variieren, die Einfluß haben können, und Beispiele für die Fälle sein, die Sie behandeln wollen. Oft gibt es bereits vorbereitete Standarddatensätze, an denen alle ihre Programme überprüfen. So ein Standarddatensatz heißt *Benchmark*. In der Literatur findet man dann, welche Programme bereits mit welchem Ergebnis in welcher Zeit die Tests des Benchmarks absolviert haben. Daraus ergibt sich die Weltrangliste, in die Sie dann Ihr Programm einordnen können.

Ein einfaches Beispiel soll das Testen deutlich machen. Wir haben drei Sortierverfahren kennengelernt. Aus der Analyse der Verfahren wissen wir bereits, daß sie alle korrekt sind, d.h. sie sortieren die Eingabe von Zahlen tatsächlich gemäß der Ordnungsrelation $>$. Wir brauchen also kein Fehlermodell. Wir haben auch die Aufwandsabschätzung gemäß der O-Notation gesehen. Daher wissen wir, in welchem Verhältnis zur Länge der Eingabe die Laufzeit im schlimmsten Falle steht. Als Größe eines Datensatzes nehmen wir also die Anzahl der Elemente der zu sortierenden Menge. Nun können wir testen, unter welchen Umständen welches Sortierverfahren das schnellste ist. Als Umstand definieren wir uns ein Maß für die Abweichung der Eingabemenge von der sortierten Menge. Der Benutzer gibt eine Zahl zwischen 0 und 1 für den Grad der Abweichung an. 1 führt zu einer völlig ungeordneten Menge, 0 zu einer bereits sortierten. Als Schnelligkeit definieren wir zum einen die Anzahl der Schritte und zum anderen die Laufzeit. Das Messen der Laufzeit ist eigentlich nicht so einfach, wie wir es uns hier gemacht haben. Wenn ein Rechner noch andere Prozesse bedient als das Sortierverfahren, dann ist die Uhrzeit beim Start und Beenden des Programms keine zuverlässige Aussage. Ein weiteres Problem bei Zeitmessungen ist, daß das messende Programm seinerseits Zeit verbraucht. Wir haben hier deshalb die graphische Darstellung der Sortierung ausgeschlossen, wenn die Zeit gemessen werden soll.

Die Interpretation und Kommunikation der Ergebnisse ist hier einfach, da die Theorie bereits besteht und die Ergebnisse nicht von der Theorie abweichen.

Sehen Sie sich einfach einmal an, was das folgende Testprogramm macht, indem Sie `java TestSort` im Verzeichnis `~gvpr000/ProgVorlesung/Beispiele/sortieren/` aufrufen!

```
import AlgoTools.IO;

class TestSort {
    static public void main (String [] argumente) {
        int verfahren;
        int anzahl;
        double pv;

        IO.println ("1: SelectionSort");
        IO.println ("2: QuickSort");
        IO.println ("3: MergeSort");

        do {
            verfahren = IO.readInt ("[1..3]? ");
        } while (verfahren < 1 || verfahren > 3);

        do {
```

```
            anzahl = IO.readInt ("Anzahl zu sortierender Elemente (>1)? ");
        } while (anzahl < 1);

        do {
            pv = IO.readDouble ("Pseudo-Varianz [0..1]? ");
        } while (pv < 0 || pv > 1);


        int [] testArray = new int [anzahl];

        for (int i = 0; i < testArray.length; i++) {
            testArray [i] = ((int) (i + pv * (Math.random () - 0.5) * testArray.length)
            + testArray.length) % testArray.length;
        }

        ShowArray.display =
             IO.readBoolean ("Animation anzeigen (=> keine Zeitmessung)? ");

        IO.println ("Bitte warten...");

        double startTime = System.currentTimeMillis ();

        switch (verfahren) {
            case 1: AnimatedSelectionSort.sort (testArray); break ;
            case 2: AnimatedQuickSort.sort (testArray); break ;
            case 3: AnimatedMergeSort.sort (testArray); break ;
        }

        if (ShowArray.display) {
            ShowArray.showArray.repaint ();              // Anzeige Endzustand erzwingen
        }
        else {                                          // Zeit nur anzeigen, wenn nicht animiert
            IO.println ("Laufzeit: "+ (System.currentTimeMillis () - startTime) / 1000
                        + " Sek.");
        }

        IO.println ("Schritte: "+ ShowArray.counter);
    }
}
```

5 Bäume, Graphen und Suche

Bäume und Graphen gehören zu den wichtigsten Strukturen der Informatik. Fast alle Probleme können als Baum oder Graph dargestellt werden und die Lösung des Problems dann als ein Pfad in dem Baum oder Graph oder als das Ziel des Pfades. Die Graphentheorie geht auf Euler zurück, der wissen wollte, ob es einen Weg gibt, der genau einmal jede der sieben Brücken von Königsberg überquert und dann wieder am Ausgangspunkt

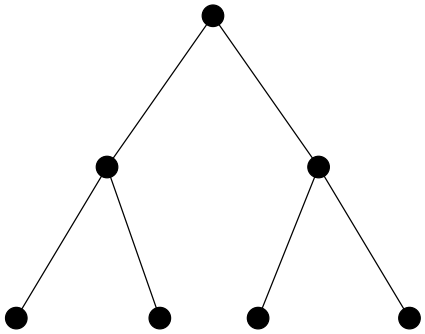


Abbildung 24: Ein binärer Baum

ankommt.[20] Dieses Problem ist nicht lösbar, aber die dafür entwickelte Notation hat viele andere Probleme lösen helfen. Bäume sind eingeschränkte Graphen, weswegen manche auch bei ungerichteten Graphen von einem *Wald* sprechen. Wir fangen hier mit den Bäumen an und sprechen dann von Graphen. Als Tätigkeit, die wir in Bäumen und Graphen ausführen, behandeln wir die Suche. Graphentheorie und Suche werden Sie Ihr ganzes Studium hindurch begleiten.

5.1 Binäre Bäume

Ein Baum besteht aus Knoten und Kanten. Eine Kante verbindet zwei Knoten in einer Richtung. Der Knoten, von dem Kanten ausgehen, zu dem aber keine Kanten hinführen, heißt *Wurzel*. Ein Baum hat immer nur eine Wurzel. Im Gegensatz zur Natur wird bei Bäumen der Informatik die Wurzel immer oben hingezeichnet. Ein Knoten, zu dem eine Kante hinführt, von dem aber keine Kante abgeht, heißt *Blatt*. Blätter werden unten hingezeichnet. Bei einem Baum hat jeder Knoten nur eine hinführende Kante.

Definition 5.1: Binärer Baum Der abstrakte Datentyp *binärer Baum* ist entweder leer, oder besteht aus einem Knoten, dem ein linker und ein rechter binärer Baum zugeordnet ist. Die Operationen sind der Test, ob der (Teil-)baum leer ist, die Rückgabe des linken und die Rückgabe des rechten Teilbaums. Außerdem gibt es eine Operation, die die Wurzel des Baums liefert.

Die Definition ist rekursiv: jeder Unterbaum hat wieder eine Wurzel, an der ein rechter und ein linker Baum hängt. Blätter sind also Bäume, deren rechter und linker Teilbaum leer sind. Dadurch, daß wir den linken und den rechten Unterbaum unterscheiden, ist der Baum *geordnet*.

Die Darstellung in JAVA ist sehr einfach und folgt der rekursiven Definition genau [Vornberger und Thiesing, 1998].

```
package LS8Tools;

import AlgoTools.IO;
```

[20] Leonhard Euler, 1707 – 1783, schweizer Mathematiker.

```
public class Baum {

    Object inhalt;                                              // Inhalt
    Baum links, rechts;                                  // linker, rechter Teilbaum

    public final static Baum LEER = new Baum ();        // leerer Baum als Klassenkonst.

    public Baum () {                                    // konstruiert einen leeren Baum
        inhalt = null;                                                // kein Inhalt
        links = null;                                                     // keine
        rechts = null;                                                  // Kinder
    }

    public Baum (Object x) {                                  // konstruiert ein Blatt
        this (LEER, x, LEER);                                        // mit Objekt x
    }

    public Baum (Baum l, Object x, Baum r) {                 // konstruiert einen Baum
        inhalt = x;                                         // aus einem Objekt x und
        links = l;                                           // einem linken Teilbaum
        rechts = r;                                    // und einem rechten Teilbaum
    }

    public boolean empty () {                                          // liefert true,
        return (inhalt == null);                                  // falls Baum leer ist
    }

    public Baum left () {                                     // liefert linken Teilbaum
        if (empty ()) IO.error ("in left: leerer Baum");
        return links;
    }

    public Baum right () {                                   // liefert rechten Teilbaum
        if (empty ()) IO.error ("in right: leerer Baum");
        return rechts;
    }

    public Object value () {                              // liefert Objekt in der Wurzel
        if (empty ()) IO.error ("in value: leerer Baum");
        return inhalt;
    }
}
```

Ein binärer Baum wird aufgebaut, indem jeweils ein linker und ein rechter Teilbaum zu einer Wurzel angegeben wird. Er wird also von den Blättern her bis zur Wurzel aufgebaut. In der JAVA-Realisierung nutzen wir aus, daß die Referenzzuweisung als Wert einer Variablen auf ein Objekt verweist. Die Variable *links* erhält als Wert gerade die Referenz auf den Baum, der links unter dem aktuellen Knoten hängt. [21] Wir bauen einen neuen Baum

[21] Ist die Sprache der Informatik nicht phantastisch: nach Schlangen und Kellern nun (an der Wurzel) hängende Bäume! Falls Sie einen neuen Datentyp erfinden, zögern Sie nicht, ihn Garten zu nennen und

auf, indem wir den Konstruktor zunächst für zwei Blätter und deren Mutter aufrufen. Das erzeugte Objekt vom Typ **Baum** wird dann der Wert der Variablen *links* in dem Baum, dessen linken Unterbaum wir gerade erzeugt haben...

Wir verwenden den abstrakten Datentyp *binärer Baum* meist zur Suche. Wir unterscheiden

- die erschöpfende Suche, bei der alle Knoten abgelaufen werden, bis das Ziel erreicht ist;
- die heuristische Suche, bei der eine Heuristik die Kanten auswählt, die wir begehen, wobei irgendeine Bewertung (Heuristik) genutzt wird;
- die gezielte Suche, die besser *Finden* hieße, weil wir genau wissen, welche Kante wir entlanggehen müssen.

Außerdem gibt es verschiedene Reihenfolgen, in denen wir die Knoten eines Baumes besuchen. Der wichtigste Unterschied ist, ob wir erst alle Nachfolger eines Knoten betrachten (Breitensuche), oder ob wir einen Nachfolger auswählen und dessen Nachfolger betrachten, von denen wir einen auswählen etc. (Tiefensuche).

5.1.1 Tiefen- und Breitensuche

Zur Illustration der uninformierten Suche (d.h. wir haben kein Vorwissen, das uns befähigt abzuschätzen, wo ungefähr das Ziel liegt) eignet sich das Labyrinth. Wir kennen den Weg nicht, aber wir kennen das, was wir suchen. Abbildung 25 zeigt ein einfaches Labyrinth mit dem Eingang S und dem Ziel Z. Die Entscheidungspunkte sind mit kleinen Buchstaben bezeichnet. Da wir gerade binäre Bäume besprechen, gibt es an jedem Entscheidungspunkt nur die Frage, ob wir rechts oder links gehen wollen.

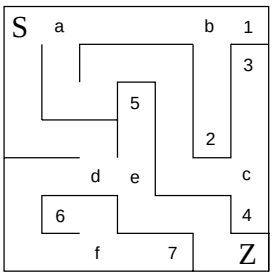


Abbildung 25: Labyrinth

Das Labyrinth kann als Baum gezeichnet werden, bei dem jeder Knoten ein Entscheidungspunkt ist und der nachfolgende Weg eine wegführende Kante. Die Blätter sind Sackgassen oder das Ziel.

Bei der Suche in diesem Baum können wir die Nachfolger eines Knotens auf zwei verschiedene Arten betrachten: wir fügen sie wie beim Keller vorn an die bereits gesammelten Nachfolger an (Tiefensuche) oder wie bei der Schlange hinten an die gesammelten Nachfolger an (Breitensuche). Die Tiefensuche eignet sich hervorragend zur Rekursion, weil wir

ebenfalls hängen zu lassen.

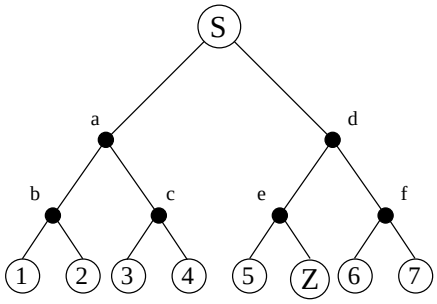


Abbildung 26: Baumdarstellung des Labyrinthes

bei jedem Baum von dessen Wurzel aus den jeweiligen linken Unterbaum betrachten, bis schließlich ein linker Unterbaum leer ist oder seine Wurzel das Ziel. Wenn der linke Unterbaum das Ziel nicht enthält, betrachten wir den rechten Unterbaum. Natürlich betrachten wir innerhalb dieses Baums dann wieder zuerst den linken Unterbaum. Die Tiefensuche sieht in JAVA so aus:

```
public static boolean tiefensuche (Baum b)  {
      boolean amZiel = false;                          // Variable, ob wir das Ziel gefunden haben

      if (b.empty ()) return false;                                  // Leerer Teilbaum ? Raus.

      if (tiefensuche (b.left ())) amZiel = true;                                  // Ist Ziel links ?
      if (!amZiel && tiefensuche (b.right ())) amZiel = true;                       // oder rechts ?

      if (!amZiel)  {                                       // Ziel nicht in Teilbaeumen gefunden
            IO.println ("Knoten: "+ b.value ());                          // Akt. Knoten ausgeben

            if (b.value ().equals("Ziel"))  {                          // Test, ist das Ziel hier ?
                  IO.println ("Ziel erreicht !");
                  amZiel = true;
            }
      }
      return amZiel;
}
```

Die Tiefensuche betrachtet in unserem Beispiel also $a, b, 1, 2, c,\ 3, 4,\ d, e, 5,\ Z$ – in dieser Reihenfolge. Sie merkt sich den Weg nicht, sondern vermeldet nur `true`. Auch ist hier das Ziel ein für allemal festgeschrieben. Eine allgemeinere Fassung verwendet eine Vergleichsmethode, die die Gleichheit des aktuellen Knoten und eines Ziels prüft. In JAVA realisieren wir das mithilfe eines Interface, das für alle Arten von Zielen (Buchstaben, Wörter, Zahlen, Bedingungen, ...) implementiert werden muß.

Die Breitensuche verwendet eine Schlange zum Speichern der Knoten, die noch nicht besucht wurden. Nachfolger werden hinten an die Schlange gehängt. Das Frontelement der Schlange wird betrachtet, ob es vielleicht das Ziel ist.

```
public static void breitensuche (Baum wurzel)  {
      Baum b;                                                    // Hilfsvariable
      boolean amZiel = false;                                    // Wert, ob wir schon im Ziel sind

      Schlange zuBesuchen = new Schlange (20);                   // Schlange der Knoten
                                                                 // die noch nicht besucht wurden
      if (!wurzel.empty()) zuBesuchen.enq (wurzel);              // Mit Wurzel starten

      while (!zuBesuchen.empty() && !amZiel)  {                  // Solange noch Knoten
                                                                 // vorhanden und Ziel nicht erreicht,
            b = (Baum)zuBesuchen.front ();                       // obersten Knoten aus Schlange
            zuBesuchen.deq ();                                   // nehmen und loeschen

            IO.println ("Knoten: "+ b.value ());                 // und ausgeben.

            if (b.value ().equals("Ziel"))  {                    // Sind wir hier am Ziel ?
                  amZiel = true;
                  break ;                                        // Ziel gefunden, while-Schleife verlassen
            }

                                                // Eventuelle Nachfolger hinten an Schlange haengen
            if (!b.left().empty()) zuBesuchen.enq (b.left ());
            if (!b.right().empty()) zuBesuchen.enq (b.right ());
      }

      if (amZiel)
      IO.println ("Ziel erreicht !");
      else
      IO.println ("Habe mich verlaufen !");
}
```

Die Breitensuche besucht die Knoten in der folgenden Reihenfolge: a, d, b, c, e, f, 1, 2, 3, 4, 5, Z. Beide Suchverfahren sind erschöpfend.

Die Klasse **Traversierung** im Verzeichnis

`~gvpr000/ProgVorlesung/Beispiele/baum/`

enthält eine *main*-Methode, die einen Baum aufbaut und für diesen Tiefensuche und Breitensuche aufruft.

5.2 Bäume mit angeordneten Knoten

Die Ordnungsrelation beim binären Baum trifft nur die Unterscheidung zwischen links und rechts. Wir können Vorwissen durch eine Ordnungsrelation über den Nachfolgern (Unterbäumen) ausdrücken, um gezielt zu suchen. Wenn wir eine Wegbeschreibung für das binäre Labyrinth haben, so wissen wir an jedem Entscheidungspunkt, welchen Unterbaum wir wählen sollen. Im Beispiel hieße die Wegbeschreibung "rechts, links, rechts". Die Wegbeschreibung verwendet also die Ordnungsrelation des Baumes. Nun gibt es nicht so furchtbar viele Probleme, deren Problemstellung bereits eine Folge von links/rechts-Entscheidungen ausdrückt, so daß die Problembeschreibung gleichzeitig die Lösung ist. Deshalb verallgemeinern wir den binären Baum zu einem Baum, dessen Unterbäume gemäß einer Ordnungsrelation angeordnet sind – egal, wieviele es sind. Ein häufig ge-

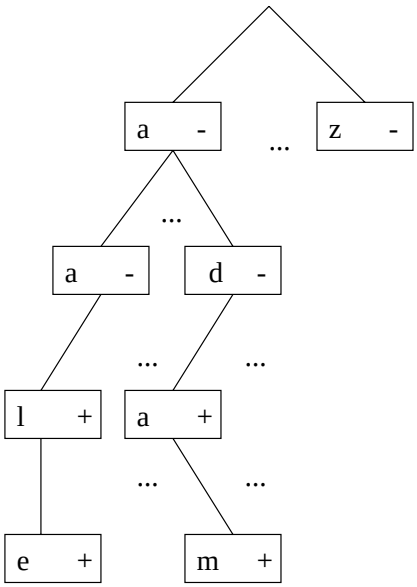


Abbildung 27: Unvollständiger LTree deutscher Wörter

brauchtes Exemplar eines solchen Baums mit angeordneten Knoten ist der *lexical retrieval tree*, auch *ltree* oder *trie* genannt. Er nutzt als Ordnungsrelation das Alphabet und wird zum Speichern von Lexika verwendet.

Definition 5.2: LTree ist ein Baum, bei dem jeder Knoten ein Zeichen aus einem geordneten Alphabet darstellt und eine Markierung. Ein Wort ergibt sich aus der Konkatenation der Zeichen, die von der Wurzel zu dem Knoten mit positiver Markierung führen.

Mit diesem Baum können wir gezielt suchen, weil jedes Wort, das wir suchen, schon seine Wegbeschreibung ist. Nicht alle erreichbaren Knoten stellen Wörter dar. Es sind dann negativ markierte Knoten. Nicht alle Wörter sind Blätter. Manchmal liegen auf dem Weg zu einem Wort viele andere Wörter. Die Wörter auf dem Weg zu einem Wort sind dessen *Präfixe*. So sind *ab* und *ablauf* Präfixe von *ablaufen*. In der Abbildung 27 ist *Aal* ein Präfix von *Aale* und die Programmiersprache *ada* ein Präfix von *Adam*. Würden wir die Wörter alle einzeln speichern, hätten wir mehr Speicher verbraucht, nämlich so viel mehr wie es Präfixe gibt. Der LTree verbraucht nur so viel Speicher wie eine Matrix mit der Alphabetlänge als Breite und der Länge des längsten Wortes als Tiefe. Er ist also bestens geeignet, eine riesige Menge von Wörtern aufzunehmen. Je häufiger es zu einem Wort Präfixe gibt, umso besser schneidet der LTree im Vergleich mit anderen Speicherarten ab.

Bäume können in vielfältiger Weise dargestellt werden. Wir haben die binären Bäume als Objekte mit den drei Eigenschaften *links* vom Typ binärer Baum, *rechts* vom Typ binärer Baum und *inhalt* vom Typ **Object** realisiert. Wir nutzten dabei also die Referenzzuweisung von JAVA aus. Jetzt realisieren wir den Baum mithilfe eines Feldes von Nachfolgern. Da das Alphabet eine Ordnung hat, wissen wir, an wievielter Position im Feld wir einen bestimmten Buchstaben finden. Der Feldindex kann also bestimmt werden. Wir nutzen aus, daß in JAVA jeder Buchstabe einen Zahlenwert hat.

Die Klasse **LTree** hat die folgenden Methoden:

LTree(char c) nimmt einen Buchstaben, markiert ihn und erzeugt das Feld seiner Nachfolger.

insert(String s) fügt ein neues Wort in den Baum ein, indem für jeden Buchstaben der Reihe nach der Konstruktor aufgerufen wird.

remove(String s) entfernt ein Wort aus dem Baum.

hasNext() prüft, ob ein Knoten noch Nachfolger hat.

getIndex(char c) bestimmt den Zahlenwert des Buchstabens.

dump() gibt den Baum aus.

dump(String s) gibt den Baum rekursiv aus, von der Wurzel bis s und dann die Nachfolger von s.

Sie finden das Programm mit einem Beispiel im Verzeichnis
`~gvpr000/ProgVorlesung/Beispiele/baum`.

Spielen Sie mal damit! Wenn Sie es genügend verstanden haben, können Sie sich ja einmal andere Alphabete ausdenken und eine Kopie des Programms entsprechend ändern! Überlegen Sie auch, wie die Blätter dahingehend erweitert werden können, daß sie z.B. Übersetzungen der gefundenen Wörter enthalten.

5.3 Was wissen Sie jetzt?

Sie müssen unbedingt wissen, was ein binärer Baum ist. Die rekursive Definition "ist entweder leer oder besteht aus einem Knoten mit einem linken und einem rechten binären Baum als Nachfolger" muß ihnen unter allen Umständen flüssig über die Lippen gehen.

Die Ordnungsrelation ist ausführlich besprochen worden. Wenn sie beim binären Baum mit links und rechts auch etwas mager ausfällt, so ist sie doch auch dort nicht zu unterschätzen. Schließlich sind Wegbeschreibungen bei binären Systemen genau auf diese Ordnungsrelation zu übertragen. Der LTree hat eine beliebige aber feste Anzahl von Nachfolgern je Knoten und nutzt die Ordnung seines Alphabets aus.

5.4 Graphen

Graphen können die verschiedensten Beziehungen darstellen. Erfunden wurden sie, um räumliche Beziehungen abstrakt darzustellen: es gibt einen Weg von a nach b. Dazu reichten Bäume nicht aus, denn es gibt mehrere Wege zu einem Ort (viele Wege führen nach Rom). Genausogut können wir aber zeitliche Beziehungen darstellen. Wir interpretieren die Knoten als Zustände und lesen die Kanten "und dann". Auf diese Weise können wir einfache Entwicklungen darstellen wie etwa den Zyklus der Jahreszeiten, das Knospen, Blühen, Frucht tragen und Abfallen bei Obstbäumen. Planung kann auch mithilfe von Graphen formuliert werden. Dann lesen wir eine Kante zwischen a und b als "a muß vor b geschehen". Kausale Beziehungen, bei denen die Kanten als "verursacht" gelesen werden, lassen sich ebenfalls gut durch Graphen darstellen. Die Problemlösung, die wir mit Graphen anstellen, ist wie bei den Bäumen die Suche. Wir suchen einen Weg von a nach

b, wir suchen den Nachfolger eines Knotens, um vorherzusagen, was kommt, wir suchen eine Erklärung für eine Beobachtung (wodurch wird sie verursacht?). Es lohnt sich also, genauer zu wissen, was Graphen sind und wie sie in JAVA formuliert werden können.

Wir beginnen mit gerichteten Graphen und den Suchmethoden Tiefen- und Breitensuche. Dann besprechen wir ungerichtete Graphen, die Spezialisierungen des gerichteten Graphen sind. Graphen sind eine anschauliche Darstellung endlicher zweistelliger Relationen. Zum Beispiel können wir die Relation {(a,b), (a,c), (a,d), (b,c), (d,b), (d,c)} als Graph zeichnen wie in Abbildung 28 zu sehen.

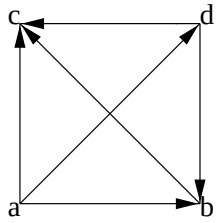


Abbildung 28: Graph zur Relation

Wir sehen, daß es wie beim Baum Knoten und Kanten gibt, aber es muß keinen Knoten ohne Vorgänger geben und ein Knoten kann mehrere Vorgänger haben.

Definition 5.3: Graph, gerichteter Gegeben sei eine Grundmenge von Knoten V (engl. vertex, vertices). Ein gerichteter Graph $G = (V, E)$ besteht aus V und Kanten $E \subseteq V \times V$ (engl. edge(s)). Ist V endlich, so ist auch der Graph G endlich.

Das kartesische Produkt $V \times V$ setzt jedes Element in V mit jedem Element in V in Beziehung. Sei $V = \{a, b, c, d\}$, so ist $V \times V$:

{(a,a), (a,b), (a,c), (a,d), (b,a), (b,b), (b,c), (b,d), (c,a), (c,b), (c,c), (c,d), (d,a), (d,b), (d,c), (d,d)}. Aus diesem kartesischen Produkt greift E eine Teilmenge heraus, z.B. die oben angeführte. Unsere Definition umfaßt aber auch etwa diese Teilmenge:

{(a,b), (b,a), (a,a), (c,d), (d,c), (c,c)}.

Dieser eine Graph besteht aus zwei Teilen, die nicht miteinander verbunden sind. Wie man in Abbildung 29 gut sieht, ist jeder Teil für sich betrachtet zusammenhängend. Da der Graph aber mehr als einen zusammenhängenden Teil hat, ist er insgesamt unzusammenhängend.

Definition 5.4: Graph, stark zusammenhängend Ein gerichteter Graph heißt stark zusammenhängend, wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist. Erreichbar ist ein Knoten v_i von einem anderen Knoten v_1 aus, wenn es eine Kante (v_1, v_i), zwei Kanten $(v_1, v_2), (v_2, v_i)$ oder eine Folge von Kanten gibt $(v_1, v_2), (v_2, v_3), ..., (v_{i-1}, v_i)$.

Definition 5.5: Graph, schwach zusammenhängend Ein gerichteter Graph heißt schwach zusammenhängend, wenn es zwischen zwei Knoten immer einen *Semiweg* gibt. Ein Semiweg zwischen zwei Knoten ist eine Folge von Kanten, die die Knoten verbindet, wobei man von der Richtung der Kanten absehen darf.

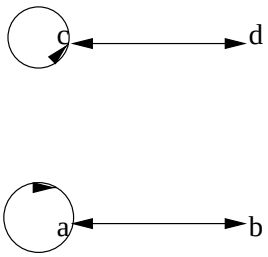


Abbildung 29: Unzusammenhängender Graph mit zwei Zusammenhangskomponenten

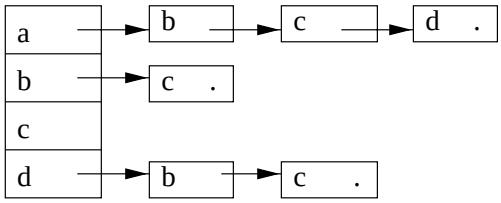


Abbildung 30: Adjazenzlisten für einen Graphen

Definition 5.6: Zusammenhangskomponente Ein Teilgraph heißt Zusammenhangskomponente, wenn er bezüglich der Zusammenhangseigenschaft (stark oder schwach) maximal ist, d.h. der Teilgraph kann nicht durch einen weiteren Knoten einer eine weitere Kante des Graphen erweitert werden, ohne eben diese Eigenschaft zu verlieren.

Ein Graph mit mehr als einer Zusammenhangskomponente heißt *unzusammenhängend*. Der Graph in Abbildung 28 ist schwach zusammenhängend. Der Graph in Abbildung 29 ist unzusammenhängend.

Wie kann man nun einen Graphen darstellen? In der Literatur werden zwei Varianten beschrieben: die Adjazenzliste und die Adjazenzmatrix. Nimmt man statt *Adjazenz*, ein aus dem englischen direkt übernommenes Wort, das deutsche Wort *Nachfolger*, so ist klar, daß wir zum einen eine Liste von Nachfolgern für jeden Knoten angeben können, zum anderen eine Matrix, oben und seitlich mit allen Knoten beschriftet und in den Kästchen steht, ob es eine Kante gibt oder nicht.

Wenn der Graph gerichtet ist, ist die Nachfolgerliste praktisch. Ebenso, wenn zwischen vielen Knoten keine Kante besteht. Wenn es zwischen fast allen Knoten eine Kante gibt, ist die Nachfolgermatrix praktisch. Ebenso, wenn direkt auf einen Knoten gesprungen werden soll. Abbildung 30 zeigt die Listen-bzw. Felddarstellung für den in Abbildung 28 gezeichneten Graphen. Tabelle 5 zeigt die Matrixdarstellung für denselben Graphen.

Wir programmieren einen gerichteten Graphen in zwei Schritten. Zunächst schreiben wir die Klasse **Vertex** mit den Eigenschaften eines Knoten:

- contents: Inhalt des Knoten – ein Objekt;
- successors: Nachfolger – eine verkettete Liste;
- alreadyVisited: Markierung, ob der Knoten schon besucht wurde.

und den Methoden:

		Nachfolger			
		a	b	c	d
Knoten	a	0	1	1	1
	b	0	0	1	0
	c	0	0	0	0
	d	0	1	1	0

Tabelle 5: Adjazenzmatrix

- **Vertex(Object c)** wobei einem Knoten die Referenz auf irgendein Objekt zugewiesen wird, eine verkettete Liste für die Nachfolger erzeugt wird und die Markierung, ob der Knoten schon besucht wurde, mit `false` initialisiert wird.
- die Methoden **getContents**, **getSuccessors** und **visited** liefern die Ausprägung der jeweiligen Eigenschaft eines Knotens zurück.
- **addSuccessor** trägt in die verkettete Liste der Nachfolger eines Knotens einen neuen Knoten ein. Dazu wid die Methode der verketteten Listen **add** aufgerufen.
- **setVisited** weist der Eigenschaft $alreadyVisited$ einen neuen Wert durch die Parameterübergabe *Wertübergabe* zu.
- **toString** gehört zum guten Ton einer JAVA-Klasse!
- depthFirstSearch: Tiefensuche von einem bestimmten Knoten aus, wobei jeder Knoten, der bei der Tiefensuche besucht wird, markiert wird mit $alreadyVisited$ =`true`. Auf diese Weise wird ein Zyklus erkannt, wenn bei der Verfolgung der Nachfolger ein bereits besuchter Knoten als Nachfolger vorkommt.
- breadthFirstSearch: Breitensuche von einem bestimmten Knoten aus.

Die Realisierung von Ralf Klinkenberg in JAVA ist übersichtlich:

```
package LS8Tools;

import java.util.LinkedList;                    // ADT Liste implementiert als verzeigerte Liste
import java.util.ListIterator;                          // Iterator fuer verzeigerte Listen
import AlgoTools.IO;                          // Vornbergers AlgoTools-I/O-Klasse


public class Vertex  {

    protected Object contents;                            // Inhalt des Knotens (z.B. Name)
    protected LinkedList successors;                     // Liste der Nachfolger des Knotens
    protected boolean alreadyVisited;                    // Markierung, ob der Knoten schon
                                                        // beim Graphdurchlauf besucht wurde

    public Vertex (Object _contents)  {
        contents = _contents;
```

```
        successors = new LinkedList ();
        alreadyVisited = false;
}


public Object getContents () {   return (contents);   }
public LinkedList getSuccessors () {   return (successors);   }
public boolean visited () {   return (alreadyVisited);   }

public void setVisited (boolean visited) {   alreadyVisited = visited;   }

public void addSuccessor (Vertex v) {   successors.add (v);   }


public String toString () {                    // Inhalt (z.B. Name) & Liste der Nachfolger
        String s;                             // fuer den Knoten erzeugte Ausgabezeichenkette
        ListIterator i;                          // Iterator fuer die Nachfolger des Knotens
        Vertex v;                                  // aktuell betrachteter Nachfolgerknoten

        s = new String (contents.toString () + ": ");
        i = successors.listIterator ();
        while (i.hasNext ()) {
                v = (Vertex) i.next ();
                s += (v.getContents ()).toString();
                if (i.hasNext ()) s += ", ";
        }
        return (s);
}


public Vertex depthFirstSearch (Object vertexContents) {
                                    //// Tiefensuche mit Zyklenerkennung (ab diesem Knoten);
                    //// Mit '/****/' startende Zeilen sind optional (nur zur Visualisierung).
        ListIterator i;                                  // Iterator fuer den aktuellen Nachfolger
        Vertex v;                                        // aktuell betrachteter Nachfolgerknoten
        Vertex r;                          // von Tiefensuche in tieferer Ebene gefundener Knoten

        /****/ IO.print (" "+ this.contents);
        this.setVisited (true);                          // aktuellen Knoten als besucht markieren

        if (this.contents.equals (vertexContents)) {
                /****/ IO.println (" (Suche erfolgreich)");
                return (this);
        }

        i = (this.successors).listIterator();
        while (i.hasNext ()) {
                v = (Vertex) i.next ();
                if (!(v.visited ())) {
                        r = v.depthFirstSearch (vertexContents);
                        if (r != null) return (r);
                }
        }
```

```
        return (null);                                  // Suche von diesem Knoten aus erfolglos
    }

    public Vertex breadthFirstSearch (Object vertexContents)  {
                                        //// Breitensuche mit Zyklenerkennung (ab diesem Knoten)
                            //// Mit '/****/' startende Zeilen sind optional (nur zur Visualisierung).
        ListIterator i;                                 // Iterator fuer den aktuellen Knoten
        Vertex v;                                       // aktuell betrachteter Knoten
        Vertex successor;                               // aktuell betrachteter Nachfolgerknoten
        Schlange openVertices                           // bereits besuchte Knoten mit noch zu
            = new Schlange (100);                       // besuchenden Nachfolgern
        openVertices.enq (this );
        this.setVisited (true);

        while (!(openVertices.empty ()))  {
            v = (Vertex) openVertices.front ();
            openVertices.deq ();

            /****/ IO.print (" "+ v.getContents());
            if (vertexContents.equals (v.getContents()))  {
                /****/ IO.println(" (Suche erfolgreich)");
                return (v);
            }

            i = (v.getSuccessors()).listIterator();
            while (i.hasNext())  {
                successor = (Vertex) i.next();
                if (!(successor.visited()))  {
                    successor.setVisited (true);
                    openVertices.enq (successor);
                }
            }
        }

        /****/ IO.println (" (Suche erfolglos)");
        return (null);
    }
}
```

Bei dieser Implementierung ist die Tiefensuche (Breitensuche) nur von einem bestimmten Knoten aus möglich. Die Suche ist seine Methode. In einem zweiten Schritt legen wir um die Klasse der Knoten noch eine Schicht herum. Dies ist der eigentliche Graph. Die Objekte dieser Klasse können nun alle ihre Knoten als unbesucht markieren. Diese globale Initialisierung, die für wiederholtes Suchen in einem Graphen erforderlich ist, ist innerhalb der Klasse **Vertex** nicht möglich. In der Klasse Graph kann die Tiefensuche (Breitensuche) dann für den gesamten Graphen und auch wiederholt durchgeführt werden. Die Klasse **Graph** steht in den Paketen für die Vorlesung:

`~gvpr000/ProgVorlesung/Packages/LS8Tools/`

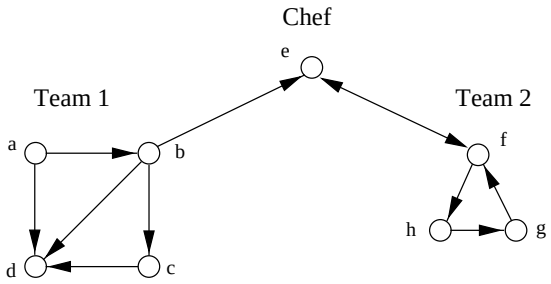


Abbildung 31: Witze erzählen in einer Firma

Hier nun ein Beispiel für die Verwendung eines gerichteten Graphen. Wir beschreiben den Kommunikationsfluß in einer Firma. Beim Witze erzählen muß man ja darauf achten, daß man ihn nicht der Quelle, d.h. demjenigen, der ihn in Umlauf gebracht hat, wieder erzählt. Die Tiefensuche mit Zyklenerkennung warnt uns davor, einen schon besuchten Knoten nicht noch einmal zu besuchen, d.h. einem, der den Witz schon kennt, ihn noch einmal zu erzählen.

Die Firma mit dem Kommunikationsfluß ihrer Witze sieht man in Abbildung 31. Das ausführbare Programm ist in unserem Verzeichnis

`~gvpr000/ProgVorlesung/Beispiele/graph/`

Sie sehen, daß das Team 2 mit dem Chef stark zusammenhängend ist. Insgesamt ist der Graph aber nur schwach zusammenhängend.

Ungerichtete Graphen können wir als Spezialisierung gerichteter Graphen betrachten: jede Kante zwischen zwei Knoten geht in beide Richtungen. Der einzige Unterschied zwischen **Vertex** und seiner Unterklasse **UndirectedVertex** besteht in der Methode, einen Knoten – nennen wir ihn v_1 – als neuen Nachfolger zu einem anderen Knoten – nennen wir ihn v_2 – hinzuzufügen: man muß nachsehen, ob es die Kante zwischen v_1 und v_2 schon gibt. Dabei muß man sowohl die Nachfolger von v_1 als auch die Nachfolger von v_2 prüfen.

```
/*****************************************************************************
** Datei: LS8Tools/UndirectedVertex.java
** Datum: 1998/09/21
**
** Instanzen der Klasse 'LS8Tools.UndirectedVertex' sind Knoten in einem
** ungerichteten Graphen. Da Instanzen dieser Klasse auch gleichzeitig
** Instanzen der Oberklasse 'LS8Tools.Vertex' sind, kann eine Instanz der
** Klasse 'LS8Tools.Graph' auch aus Knoten des Typs
** 'LS8Tools.UndirectedVertex' bestehen.
*/

package LS8Tools;

import java.util.LinkedList;                    // ADT Liste implementiert als verzeigerte Liste
import java.util.ListIterator;                          // Iterator fuer verzeigerte Listen
import AlgoTools.IO;                                 // Vornbergers AlgoTools-I/O-Klasse
import LS8Tools.Vertex;                        // Klasse fuer Knoten in gerichteten Graphen
```

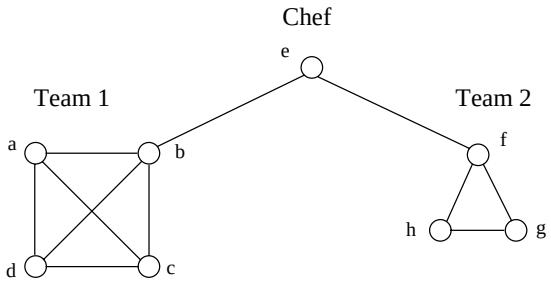


Abbildung 32: Die Weitergabe von Tips in einer Firma

```
public class UndirectedVertex extends Vertex {

    public UndirectedVertex (Object _contents) { super (_contents); }

    public void addSuccessor (Vertex v) {
        if (!(successors.contains (v)))
            successors.add (v);
        if (!((v.getSuccessors ()).contains(this)))
            (v.getSuccessors ()).add (this);
    }

}
```

Als Beispiel für einen ungerichteten Graphen können wir wieder unsere Firma betrachten, wobei aber diesmal nicht das Erzählen von Witzen sondern die Weitergabe von JAVA-Tips dargestellt wird. Da es sich bei allen in der Firma um engagierte JAVA-Anfänger handelt, werden Tips immer in beide Richtungen ausgetauscht. Während kein Witz von c nach g gelangte, kommen JAVA-Tips nun von jeder Person zu jeder anderen Person. Dennoch sind die Teams untereinander stärker verbunden als miteinander. Team 1 ist eine *Clique*, d.h. jeder Knoten ist mit jedem anderen verbunden. Ebenso ist Team 2 eine Clique, weil auch hier alle miteinander verbunden sind. Beide Cliquen sind maximal, weil wir den Chef nicht mit hinzunehmen können: er ist nicht mit jedem Mitglied des Teams verbunden. Abbildung 32 zeigt die Firma bezüglich des Austauschs von Tips zur Programmierung.

Der Graph kann mit Breiten- und Tiefensuche durchlaufen werden, um festzustellen, ob ein Knoten von einem anderen Tips erhält, ob z.B. der Chef von as Weisheit profitiert (ja). Das Beispielprogramm ist natürlich wieder in unserem Verzeichnis zu finden und ausführbar.

6 Darstellung von Mengen

Wir haben bereits auf verschiedene Arten Mengen von Objekten dargestellt: als Feld von Objekten, in einer verketteten Liste, in einem Keller oder einer Schlange. Dort haben wir die Elemente sortiert und dann gemäß der Sortierung auf sie zugegriffen. Wir konnten die Elemente auch als Knoten in einem Baum oder einem Graphen speichern. Wir suchen ein Element und liefern es dann zurück. So ganz befriedigend ist das nicht. In beiden Fällen müssen wir uns elementweise zu dem gewünschten Element bewegen. Das geht aber auch

direkter! Es gibt noch zwei Möglichkeiten, Mengen darzustellen, die in diesem Abschnitt besprochen werden sollen.

6.1 Charakteristische Vektoren

Eine Darstellung für Mengen ist die des *charakteristischen Vektors*. Wenn wir eine endliche Menge haben, die nicht erweitert wird, z.B. Spielkarten oder Waren eines Geschäfts, das sein Sortiment nicht ändert, dann vergeben wir für jede Karte bzw. jede Ware eine Position in einem Feld von `boolean` Elementen. Die Abbildung von dem Element auf den Index ist beliebig, aber festgelegt. Wir können z.B. für die Darstellung eines Einkaufs an allen Positionen, die sich auf einen Artikel beziehen, den ein Kunde gekauft hat, den Wert `true` eintragen. Wir haben dann nicht dargestellt, wieviel von einer Sorte der Kunde gekauft hat. Wir können aber z.B. – wie es bei der *Entdeckung von Assoziationsregeln*, einem Verfahren des maschinellen Lernens, geschieht – herausfinden, welche Waren meist zusammen gekauft werden, um sie dann an entfernten Ecken des Ladens auszustellen, so daß der Kunde möglichst viel der Ladenfläche sehen muß.

Nun wissen wir, *was* wir darstellen wollen. In JAVA können wir das mit einem Feld von `boolean` Elementen tun. Wenn wir ein Element hinzufügen wollen, das als i-te Position im Feld codiert wird, brauchen wir nur eine Zuweisung `a[i] = true;` zu schreiben. Jeder Einkauf z.B. wäre ein Objekt vom Typ `boolean[]`.

Die Frage *warum* nach den Vor- und Nachteilen ist schnell beantwortet. Ein Vorteil eines charakteristischen Vektors ist, daß das Einfügen oder Löschen eines Elementes nur $O(1)$ benötigt. Der Zeitaufwand für das Bilden von Schnittmengen zweier charakteristischer Vektoren, ihre Vereinigung oder ihre Differenz ist proportional zur Kardinalität der Ausgangsmenge (z.B. *aller* Waren) und nicht proportional zur Kardinalität einer bestimmten Menge (z.B. eines Einkaufs). Wenn fast immer fast alles gekauft wird, die zu behandelnde Menge also meist fast so groß wie die Ausgangsmenge ist, ist der Aufwand linear in der Anzahl von Elementen. Wenn wir aber den realistischen Fall betrachten, wieviele verschiedene Waren ein Laden führt, wird unser Vektor arg lang und ist nur dünn besetzt, d.h. die zu behandelnde Menge ist viel kleiner als die Ausgangsmenge. Und wenn wir alle Einkäufe eines Vierteljahres speichern wollen, so müssen wir uns ernste Gedanken über das Speichern all dieser Daten machen. Der Nachteil der charakteristischen Vektoren ist ihr Platzbedarf. Sie sind also nur bei eher kleinen Ausgangsmengen angemessen.

6.2 Hashing

Das Speichern großer Mengen bringt uns zu der zweiten Darstellung von Mengen, dem Zerhacken von Feldern, englisch *hashing*. Nehmen wir an, jedes Element hätte einen eindeutigen Bezeichner. Man nennt so einen Bezeichner *Schlüssel* (engl.: key). Wir können dann eine Abbildung zwischen dem Schlüssel und dem Speicherplatz definieren, an dem das Element mit diesem Schlüssel liegt. Die Funktion

$f(x) \to \mathbb{N}$

liefert für einen Schlüssel x eine natürliche Zahl, die die Speicheradresse bezeichnet. Wir wenden diese Funktion an, um ein Element in den Speicher einzutragen, und um direkt mit dem Schlüssel auf es zuzugreifen. Wir nennen diese Funktion *Hash-Funktion*. Nun fragt sich natürlich, wie wir für eine Menge von Objekten die Funktion $f(x)$ definieren sollen. Unsere erste Idee ist vielleicht, den charakteristischen Vektor als Zahl aufzufassen,

so daß er den Speicherplatz bezeichnet. Der Schlüssel wäre der charakteristische Vektor und seine Interpretation als Zahl wäre die Hash-Funktion. Diese Hash- Funktion ist eineindeutig. Aber gerade die eineindeutigen Funktionen haben ja den Nachteil, daß sie so viel Platz ver(sch)wenden! Wir wählen also eine Funktion, mit der wir zwar eindeutig den Speicherplatz erreichen, aber nicht vom Speicherplatz zurück auf den Schlüssel kommen? Das ist eine *perfekte Hash-Funktion*. Manchmal findet man sie. Zum Beispiel wird die Menge $\{braun, rot, blau, violett, türkis\}$ zufällig durch die Funktion $f(x) = Wortlänge - 3$ auf 5 aufeinander folgende Positionen eines Feldes abgebildet. Bei sehr großen Mengen findet man sie aber meistens nicht. Wir überlegen: wenn Elemente der darzustellenden Menge eher selten vorkommen, können wir eigentlich ruhig ein- und denselben Speicherplatz für mehrere Elemente vorsehen. Wir müssen dann insgesamt weniger Speicherplatz reservieren und meistens geht es gut. Die Leitidee ist: Wir wollen möglichst wenig Speicherplatz verbrauchen und es soll möglichst nur ein Objekt auf einen Speicherplatz abgebildet werden. Werden verschiedene Objekte auf dieselbe Adresse abgebildet, spricht man von einer *Kollision*. Wir nehmen Kollisionen in Kauf. Dann darf die Menge auch wachsen, ohne daß wir die Darstellung verändern müssen (wie bei charakteristischen Vektoren). Die Menge darf sogar unendlich sein.

Definition 6.1: Hash-Funktion Eine Hash-Funktion bildet mögliche Elemente einer Menge auf eine feste Anzahl von Adressen ab.

Beispiel 6.1: Hash-Funktion für Wörter Ein Wort ist eine Folge von Buchstaben $w = c_0, c_1, ...c_{n-1}$. Jedem Buchstaben ist Zahlenwert zugeordnet. Dann ist die Summe dieser Zahlenwerte *modulo* der Anzahl der Adressen B, also der Rest, der übrig bleibt, wenn die Summe durch B geteilt wird, die Adresse. Sie liegt gewiß im Intervall von 0 bis B.

$f(w) = \left(\sum_{i=0}^{n-1} c_i\right)$ modulo B

Wenn nun zwei Elemente der Menge auf denselben Speicherplatz abgebildet werden (Kollision), gibt es zwei Verfahren: das offene und das geschlossene Hashing. Wir besprechen hier nur das offene Hashing. Das offene Hashing faßt jeden der ursprünglichen Speicherplätze als Anfang einer verketteten Liste auf. Die Listen heißen englisch *buckets* (Eimer). Die Hash-Tabelle ist ein Feld von verketteten Listen. Abbildung 33 zeigt das Schema einer Hash-Tabelle beim offenen Hashing. Um ein Element x der Menge einzutragen, wird $f(x)$ berechnet und gibt die Liste an, in die x eingetragen werden soll. Um ein Element x der Menge zu finden, wird $f(x)$ berechnet und damit die Liste gefunden, in der nach x zu suchen ist. Wir haben B Listen, die nur etwa $(1/B) \cdot M$ lang sind, wobei M die Kardinalität der Menge ist. Wenn $B = M$ ist, so enthält jede Liste der Hash-Tabelle im Durchschnitt nur 1 Element. Dies ist der Idealfall für die Hash-Tabelle.

Definition 6.2: Lastfaktor Der Lastfaktor einer Hash-Tabelle ist bei der Anzahl B von Adressen und der Kardinalität M der darzustellenden Menge das Verhältnis M/B.

Wenn M/B etwa 1 ist, ist der Aufwand, ein Element einzufügen oder zu finden $O(1)$. Diesen Idealfall können wir mit einer Schwelle von beispielsweise 75% annähern, indem

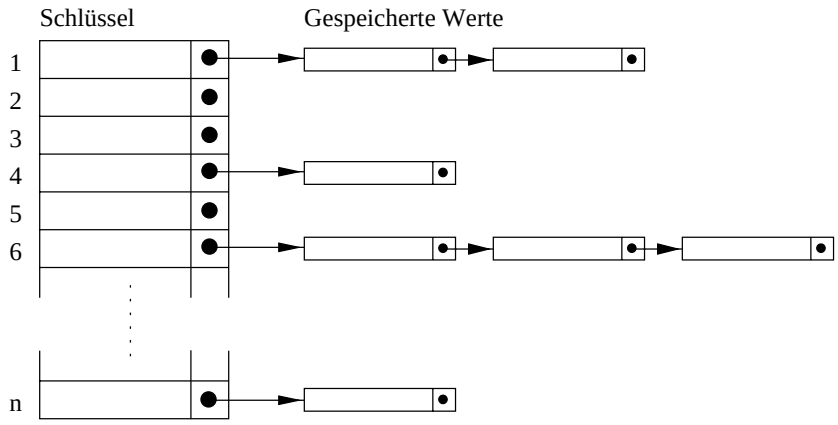


Abbildung 33: Hash-Tabelle – Offenes Hashing

wir fordern $M/B \leq 1,33$.

Ich habe gesagt, daß die Menge beliebig groß werden darf und spreche nun von ihrer Kardinalität. Das sieht aus wie ein Widerspruch. Tatsächlich haben wir in der Informatik zu jedem Zeitpunkt eine bestimmte Menge. Sie ist vielleicht die Teilmenge der eigentlich darzustellenden Menge. Dann werden nach und nach neue Elemente hinzukommen. Aber auch dann haben wir zu jedem Zeitpunkt eine bestimmte Menge, deren Kardinalität wir kennen.

Beispiel 6.2: Wir sehen vielleicht anfangs für die erwartete Kardinalität der Menge $M = 1.000.000$ einen Lastfaktor von 1 vor. Dann wächst die Menge immer mehr. Bei 1.080.000 Elementen haben wir einen Lastfaktor von 1,35. Wir beschließen, eine neue Hash-Tabelle mit dem Lastfaktor 1 anzulegen, sehen also 1.080.000 Speicherplätze vor. $B =$ 1.000.000 und dem Lastfaktor 1,08 anzulegen. Das Anlegen der neuen Tabelle erfordert $O(M)$ Aufwand. Aber das Auffinden eines Elements in der neuen Tabelle ist jetzt wieder nahe $O(1)$.

In JAVA gibt es im Paket `java.util` die Klasse **Hashtable**. JAVA verwendet das offene Hashing mit einem Schwellwert für den Lastfaktor, der auf 75% voreingestellt, vom Programmierer aber verändert werden darf. Auf Anforderung oder wenn der Lastfaktor überschritten wird, legt JAVA eine neue Hash-Tabelle an (**rehash**). Die Methoden der Klasse **HashTable**, die eine Unterklasse von **Dictionary** ist, will ich hier nicht alle aufführen. Sie können sie in der Dokumentation nachlesen. Sogar das JAVA- Programm selbst müßten Sie vor dem Hintergrund dieses Abschnitts halbwegs verstehen können.[22] Die wichtigsten Methoden sind **put** und **get**, mit denen man Elemente in die Tabelle einträgt, bzw. ihr findet. Die Methode **keySet** gibt die Menge der vergebenen Schlüssel zurück.

Ein Beispiel zeigt die Verwendung von Hash-Tabellen für das Speichern und den Zugriff auf Wörter mit ihrer Bedeutung. Die Wörter selbst sind der Schlüssel. Das Objekt, das diesem Schlüssel zugeordnet ist, ist hier die Bedeutung des Wortes. Die Hash-Funktion

[22] Die Operation *modulo* wird in JAVA % geschrieben.

muß der Programmierer nicht angeben. Stellen Sie sich die oben angeführte Summe der Zahlenwerte *modulo* der vielleicht 12 Speicherplätze vor, die für unser kleines Beispiel nötig sind. Das Beispiel veranstaltet ein Wörterraten mit dem Benutzer.

```
import AlgoTools.IO;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Random;

class HashBeispiel {

    public static void main (String[] argv) {
        Hashtable lexikon;                                          // Unser Lexikon
        Object[] words;                                  // Ein Feld mit allen Begriffen
        Random zufall = new Random ();                    // Ein Zufallszahlengenerator
        int i;                                                      // Eine Zufallszahl
        String begriff, geraten, nochmal;

        lexikon = new Hashtable ();                              // Erzeugen des Lexikons

        lexikon.put ("Auto", "Hat 4 Raeder, produziert Abgas");          // Ein paar Begriffe
        lexikon.put ("Fahrrad", "Hat 2 Raeder, produziert Schweiss");
        lexikon.put ("Flugzeug", "Fliegt schnell und manchmal zuverlaessig.");
        lexikon.put ("Rakete", "Fliegt schnell und transportiert Satelliten");
        lexikon.put ("Skates", "Hat 8 Raeder, produziert Spass");
        lexikon.put ("Vogel", "Fliegt langsam und legt Eier");

        IO.println ("Hallo ! Willkommen beim Begrifferaten !");
        IO.println ("Das Lexikon enthaelt "+ lexikon.size () + "Begriffe.");

        words = lexikon.keySet ().toArray();                  // Menge der Woerter bestimmen

        do {                  // Hauptschleife. Gibt Bedeutung und fragt Benutzer nach Begriff
            i = zufall.nextInt (lexikon.size ());                 // Einen Begriff auswaehlen
            begriff = words[i].toString();

            IO.println ();                                                      // Abfragen
            IO.println (lexikon.get (begriff).toString());
            geraten = IO.readString ("Was ist das ? ");

            if (geraten.equals (begriff))
            IO.println ("Hurra ! Richtig !!");
            else
            IO.println ("Leider falsch. Die richtige Antwort lautet: "+ begriff);

            nochmal = IO.readString ("Nocheinmal (ja/nein) ? ");
        } while (nochmal.equals ("ja"));

        IO.println ("Auf Wiedersehen beim Begrifferaten.");
    }
}
```

6.3 Was wissen Sie jetzt?

Machen Sie sich mal ein Liste, welche Implementierungen von Mengen Sie schon kennengelernt haben! Welche kommt der tatsächlichen Menge am nächsten?

Jedenfalls kennen Sie eine Hash-Funktion und wissen, was Hash-Funktionen im allgemeinen sind. Sie wissen, was eine Kollision ist und wie das offene Hashing damit umgeht.

Sie kennen auch gerichtete und ungerichtete Graphen, deren Nachfolger als verkettete Liste gespeichert werden. Schreiben Sie sich auch umgekehrt einmal auf, was wir alles mit verketteten Listen realisiert haben. Bedenken Sie, eine verkettete Liste ist *kein* abstrakter Datentyp, sondern eine Art, *wie* wir ihn realisieren.

Sie haben die Breiten- und Tiefensuche in Bäumen und in Graphen gesehen – was ist eigentlich der Unterschied?

7 Ereignisbehandlung und graphische Oberflächen

Wir haben bisher Zeichenketten eingelesen und auf den Bildschirm geschrieben. Dazu haben wir das Paket `AlgoTools` von Vornberger und Thiesing verwendet. Wie funktioniert es? In diesem Abschnitt wird zunächst das Geheimnis der zeilenweisen Benutzereingaben gelüftet. Wir stellen ein kleines Programm vor, das das Prinzip illustriert, das auch den `AlgoTools` zugrunde liegt.

Tatsächlich werden aber für Systeme, die einem Benutzer angenehm sein sollen, graphische Oberflächen programmiert. Bei der Systementwicklung rechnet man etwa 75 % der Entwicklungszeit für die Erstellung der Oberfläche. Hier soll ein Überblick über die Mittel gegeben werden, die JAVA bereitstellt. Dabei werden wir nicht die Bibliotheken im Detail betrachten, sondern die Konzepte kennenlernen, die den Klassen in den Bibliotheken zugrunde liegen. Mit dieser Orientierung sind Sie dann bei Bedarf in der Lage, sich geeignete Objekte und Methoden zusammenzusuchen. Es ist sogar wahrscheinlich, daß Sie die Konzepte auch in anderen Programmiersprachen wiederfinden. Ihr Wissen veraltet also nicht.

In diesem Abschnitt beschäftigen wir uns mit der Entwicklung von graphischen Oberflächen (*Graphical User Interfaces*, kurz: GUI). JAVA stellt ein Paket zur Verfügung, genannt *abstract windowing toolkit*, abgekürzt *awt*, das Klassen zum Zeichnen, Komponenten, Layout-Klassen, Ereignisbehandlung und Bildbearbeitung enthält. [Bishop, 1998] stellt dieses Paket sehr klar und ausführlich dar.

7.1 Textzeilen als Benutzereingabe

Ein Grundkonzept für die Interaktion eines Programms mit einem Drucker, einem Bildschirm, einem anderen Prozeß ist das des *Stroms*. Gemeint ist damit ein Strom von Daten, der zum Beispiel vom Programm auf den Drucker oder den Bildschirm geschickt wird, oder von dem Bildschirm zu dem Programm. So ein Strom kann also zum Programm hinführen. Dafür sind Unterklassen des Interface **InputStream** oder **Reader** zuständig. Das Interface **InputStream** ist recht maschinennah. Beispielsweise liest die Klasse **FileInputStream**, die es implementiert, Bytes aus einer Datei, deren Name (oder ein anderes

Objekt der Klasse **FileDescriptor**) angegeben wurde. Die Klasse **ObjectInputStream** kann Objekte, die von einem **ObjectOutputStream** zu einem Strom von Daten gemacht wurden, wieder in getrennte Objekte umwandeln. Meist verwendet man aber Implementierungen von **Reader**. Die Klasse **BufferedReader** liest nicht zeichenweise ein, sondern verwendet einen Puffer. Der Strom "ergießt" sich in den Puffer, bis dieser voll ist. Das Programm liest immer einen vollen Puffer. Die Methode **readLine()** liest eine Textzeile und gibt sie als **String** zurück. Sie wirft eine Fehlermeldung, weshalb sie in einem Programm innerhalb eines *try*-Blocks verwendet werden sollte. Das **System** hat als vorgegebenen Ort, von dem ein Programm liest, *in*. Dies bezieht sich kurz gesagt auf den Bildschirm, von dem aus das Programm aufgerufen wurde. Damit die Methode aufgerufen werden kann, muß es natürlich ein Objekt geben vom Typ **BufferedReader**. In dem kleinen Beispiel unten sehen Sie in Zeile 4 die Deklaration der Variablen *reader*. In Zeile 10 wird ein Objekt erzeugt, auf das dann *reader* referenziert. Dieses Objekt wendet in Zeile 14, eingeklammert in einen *try*-Block (Zeilen 12 - 16), die Methode **readLine()** an.

Die Ausgabe eines Programms verläuft ganz analog: es gibt die Interfaces **OutputStream** und **Writer**. **System**.*out* bezeichnet den Ort, zu dem der Strom vom Programm fließen soll, hier einfach den Bildschirm, von dem aus das Programm aufgerufen wurde. Auch für das Schreiben gibt es die Verwendung von Puffern. In dem kleinen Beispiel unten wird die Basismethode für das Schreiben verwendet statt eines Objektes vom Typ **Writer** mit seinen Methoden. Die Methode **println** von **System** gibt einen String aus.

Die Klasse **Screen**, die hier zur Illustration programmiert wurde, definiert eine Methode **readLine(String s)**. Es wird der Text *s* an den Benutzer ausgegeben (Zeile 6) und dafür gesorgt, daß er nicht im Ausgabestrom hängenbleibt (Zeile 7). Ein Objekt vom Typ **BufferedReader** wird erzeugt, das eine Zeile vom Bildschirm liest und als Objekt vom Typ **String** zurückgibt.

Für den Fall, daß eine Zahl gelesen werden soll, wird die Methode **readFloat(String s)** definiert. Sie macht nur eine Typumwandlung. Dazu verwendet sie die Klasse **Float**, die JAVA dafür vorgesehen hat, aus dem einfachen Unikat einer reellen Zahl ein Objekt zu machen. (Analog gibt es **Integer**.) Der Konstruktor von **Float** hat als Parameter ein Objekt vom Typ **String**. Wir rufen **readLine(prompt)** auf. Damit gibt es nun ein Objekt vom Typ **Float**, das eine String-Darstellung für eine Zahl darstellt. Es ist gerade die Eingabe des Benutzers. Das Programm, das **readFloat** aufruft, erwartet aber eine reelle Zahl. Also müssen wir mit der Methode **floatValue**, die jedes Objekt vom Typ **Float** beherrscht, die Darstellung als Unikat vom einfachen Datentyp `float` wiedergewinnen. *return* gibt diese reelle Zahl vom Typ `float` zurück. Die eine Zeile 24 des Programms hat es in sich!

```
import java.io.*;                                                    // 1

class Screen                                                         // 2
{                                                                    // 3
    static BufferedReader reader = null;                             // 4

    static public String readLine (String prompt)  {                // 5
        System.out.println (prompt);                                 // 6
        System.out.flush ();                                         // 7
        if (reader == null)                                          // 8
```

```
            {                                                                    // 9
                reader = new BufferedReader (new InputStreamReader (System.in)); // 10
            }                                                                    // 11

            try                                                                  // 12
            {                                                                    // 13
                return reader.readLine ();                                       // 14
            }                                                                    // 15
            catch (Exception e)                                                  // 16
            {                                                                    // 17
                System.err.println ("Fehler in Screen.readLine: "+ e.toString ()); // 18
            }                                                                    // 19

            return null;                                                         // 20
        }                                                                        // 21
        static public float readFloat (String prompt)                            // 22
        {                                                                        // 23
            return ((new Float (readLine (prompt))).floatValue());               // 24
        }                                                                        // 25
}                                                                                // 26
```

7.2 Komponenten

Eine Graphische Benutzeroberfläche besteht aus einem oder mehren Fenstern, die Elemente wie Knöpfe, Menüleisten, oder Eingabezeilen enthalten. Für diese Elemente stellt Java fertige Klassen zur Verfügung, die Aufgaben wie ihre eigene Anzeige und die Annahme von Ereignissen übernehmen. Der Programmierer muß diese Objekte "nur noch" an der gewünschten Stelle plazieren und sich darum kümmern, entsprechende Programmfunktionalität mit den Elementen zu verbinden.

Alle Elemente der Benutzeroberfläche sind in JAVA Nachfahren der abstrakten Klasse **Component**. Die Klasse **Component** bietet abstrakte Methoden, mit denen die Größe von Elementen gesetzt oder abgefragt werden können und mit denen das System veranlassen kann, daß sich ein Element der Benutzeroberfläche darstellt. Die Erben von **Component** wie **Label** zur Anzeige von Texten oder interaktive Elemente wie **Button** oder **InputLine** füllen diese abstrakten Methoden dann mit konkreter Funktionalität.

7.2.1 Container

Ein besonderer Nachfahre von **Component** ist **Container**. Dieses Elementklasse kann eine Menge von weiteren Elemente – also auch weitere Container – enthalten. Ein Beispiel für einen Container ist ein Fenster (**Frame**), es gibt aber auch andere Container. So wird beispielsweise die Klasse **Panel** benutzt, um Elemente in einer bestimmten Weise zu gruppieren. Die Container leisten eine Abbildung der von **Component** geerbten Methoden auf ihre einzelenen untergeordeten Elemente. Ein Aufruf der Methode zum Neuzeichnen eines Containers veranlaßt beispielsweise, daß alle enthalten Elemente neu gezeichnet werden. Ferner besitzen Container zusätzliche Methoden, um Elemente zu verwalten, insbesondere um neue Elemente hinzuzufügen.

Ein besonderer Container ist **Frame**. **Frame** entspricht einem Bildschirmfenster und enthält in dem Inhaltsbereich alle anderen Container sowie in dem Rahmen einige fertige

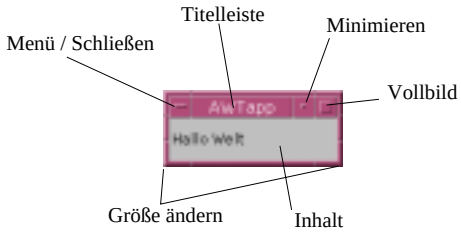


Abbildung 34: Ein Fenster

Bedienelemente, mit denen das Fenster minimiert, maximiert, geschlossen oder in der Größe geändert werden kann (siehe Abbildung 34).

Beispiel 7.1: Mit diesem Wissensstand können wir bereits unsere erste Fenster-basierte Applikation in Java entwickeln – das obligatorische Hallo-Welt-Programm:

```
import java.awt.*;                                                    // 1

class HalloWelt  {                                                    // 2
      public static void main (String [] args)  {                    // 3
            Frame frame = new Frame ();              // Neues Fenster erzeugen // 4
            frame.add (new Label ("Hallo Welt"));          // Label einfuegen // 5
            frame.pack ();                             // Layouten lassen // 6
            frame.show ();                            // Fenster anzeigen // 7
      }                                                               // 8
}                                                                     // 9
```

Das **add** in Zeile 5 fügt dabei das **Label** "Hallo Welt" in das darüber erzeugte Fenster ein. Der folgende Aufruf der Methode **pack** veranlaßt, daß das Fenster seine Größe anhand der enthaltenen Elemente berechnet. Der Aufruf von **show** schließlich veranlaßt die Darstellung des Fensters auf dem Bildschirm.

7.3 Ereignisse

Leider ist das vorangehende Beispiel etwas statisch – so statisch, daß es nicht möglich ist, das Fenster zu schließen. Dies ist also eine gute Gelegenheit, um einmal das im Anhang beschriebene Unix-Kommando "kill" auszuprobieren.

Da die Beendung eines Programmes mit "kill" nicht sehr anwenderfreundlich ist, entspricht dies nicht sonderlich dem "Geist" von graphischen Benutzeroberflächen, die ja gerade den Anwendern die Bedienung unserer Programme erleichtern sollen.

Aber wie kommen wir nun an die Information, daß das Fenster geschlossen werden soll?

Das Schöne an JAVA – oder besser an der ereignisorientierten Programmierung – ist nun, daß wir nicht in irgendeiner Schleife abfragen müssen, ob dieser oder jener Knopf gedrückt worden ist, sondern wir uns einfach automatisch benachrichtigen lassen können,

wenn irgendetwas "passiert", an dem wir interessiert sind. Dazu stellt Java "Zuhörer"-Interfaces (**Listener**) für alle möglichen Ereignisse bereit.

Die Lösung besteht darin, das entsprechende Interface – in unserem Fall **WindowListener** – zu implementieren und JAVA mitzuteilen, daß unser **WindowListener** die Ereignisse "abonnieren" möchte, die unser Fenster betreffen.

Beispiel 7.2: Ereignisse Da ein **WindowListener** sehr viele Methoden für diverse Ereignisse wie Maximierung und Minimierung des Fensters besitzt, die wir alle überschreiben müssten, können wir besser statt dessen von der Klasse **WindowAdapter** erben: WindowAdapter implementiert **WindowListener** mit leeren Methoden, wir müssen nur noch diejenigen überschreiben, an denen wir wirklich interessiert sind.

```
import java.awt.event.*;                                                    // 1

class Closer extends WindowAdapter {                                        // 2
      public void windowClosing (WindowEvent e) {          // Ereignis windowClosing // 3
            System.exit (0);                                 // Programm verlassen // 4
      }                                                                      // 5
}                                                                            // 6
```

Unsere Ereignisbehandlung muß nun noch an das Fenster gekoppelt werden:

```
import java.awt.*;                                                          // 1

class HalloWelt2 {                                                          // 2
      public static void main (String [] args) {                           // 3
            Frame frame = new Frame ();                                     // 4
            frame.add (new Label ("Hallo Welt"));                           // 5
            frame.addWindowListener (new Closer ());     // Eventhandler registrieren // 6
            frame.pack ();                                                  // 7
            frame.show ();                                                  // 8
      }                                                                     // 9
}                                                                           // 10
```

In Zeile 6 wird eine Instanz unseres "Zuhörers" für Fensterereignisse erzeugt. Diese wird durch Aufruf von **addWindowListener** als Ereignis-Empfänger für unser Fenster eingetragen.

Betätigen wir nun bei unserem "Hallo Welt"- Programm das Benutzerelement zum Schließen den Fensters, terminiert unser Programm nun ordentlich und das Fenster wird tatsächlich geschlossen.

7.4 Container und Layout

Da unser bisheriges Programm nur aus einem Element besteht, mußten wir uns noch keine Gedanken um die Anordnung der Elemente machen.

Sobald wir mehr als ein Element in unser Programm aufnehmen, stellt sich dieses Problem jedoch. Den Elementen dabei einfach feste Koordinaten zuzuordenen ist nur auf den ersten Blick eine Lösung für das Problem.

Feste Koordinaten sind nicht auflösungsunabhängig, natürlich möchte man aber, daß das Programm auf einem Bildschirm mit 640x480 Pixeln genauso gut läuft wie auf einem Bildschirm mit 1600x1200 Pixeln. Desweiteren sind feste Koordinaten auch nicht Plattform-unabängig: Während eine spartanische Oberfläche vielleicht um eine Button einen einfachen Rahmen zieht, kann es sein, daß Windows 2000 zusätzlichen Platz für einen dreidimensional animierten Schatten benötigt.

Glücklicherweise bietet JAVA auch für dieses Problem einen einfachen und leistungsfähigen Mechanismus: Jedem Container kann ein Layout-Manager zugeteilt werden, der die "Feinarbeit" der Platzverteilung vornimmt. Der Programmierer muß die grobe Richtlinie durch die Wahl des Layoutmanagers treffen.

Grundlage für die Funktion der Layoutmanager ist, daß in JAVA jede Komponente eine minimale, eine bevorzugte und eine maximale Größe besitzt, die über die Methoden **getMinimumSize**, **getPreferredSize** und **getMaximumSize** abfragbar sind. Diese Methoden sind in **Component** abstrakt und werden von den Erben mit sinnvollen Werten gefüllt.

Die Layout-Manager leisten mit diesen Methoden ihrer untergeordneten Elements zwei Dinge: Zum einen können sie daraus diese drei Werte für sich selbst berechnen, zum anderen verteilen sie den Verfügbaren Platz anhand bestimmter Regeln an die Elemente.

GridLayout beispielsweise berechnet die größte minimale Höhe und Breite aller untergeordneten Elemente und erzeugt eine Tabelle aus Zellen dieser Größe. Die Anzahl der Zeilen und Spalten wird vom Programmierer vorgegeben.

Der minimale Platzbedarf des Containers ergibt sich dann aus der Anzahl der Zeilen bzw. Spalten multipliziert mit der Zellenhöhe bzw. -breite. Überschüssiger Platz – wenn etwa der Anwender das Fenster größer zieht – wird an alle Zellen gleichmäßig verteilt, so daß alle Zellen immer die gleiche Höhe und Breite haben.

Das Border-Layout (Abbildung 35) dagegen teilt den Container in fünf Bereiche. Dabei sind die Mitte und der obere, untere, rechte und linke Rand des Containers jeweils ein Bereich. Die Berechnung des minimalen Platzbedarfes ist dabei erwartungsgemäß so, daß alle Elemente in ihrer minimalen Ausdehnung dargestellt werden können. Interessanter bei dem Border-Layout ist die Verteilung des überschüssigen Platzes: Der gesamte überschüssige Platz geht an die Mitte; alle anderen Bereiche bekommen immer nur Platz für ihre minimale Ausdehnung – zumindest in die Richtung, die sonst auf Kosten der Mitte gehen würde. Beim Einfügen von Elementen in einen Container mit Border-Layout muß der Name des gewünschten Bereiches mit angegeben werden. Jeder Bereich kann dabei nur ein Element enthalten. Da ein geschachteltes **Panel** auch ein Element ist, ist dies keine wirkliche Beschränkung.

CardLayout schließlich stellt den Inhalt auf verschiedenen Seiten dar, die mit einem Bedienelement umgeschaltet werden können.

Voreingestelltes Layout bei dem Container **Panel** ist die Klasse **FlowLayout**, die alle Elemente des Panels einfach nebeneinander anordnet, bis kein Platz mehr in der aktuellen Zeile ist; dann wird einfach in die nächste Zeile umgebrochen. Bei Frames ist als Default **BorderLayout** eingestellt. Wird ein anderes als das default-Layout gewünscht, kann dieses dem Konstruktor von **Panel** übergeben werden.

Durch Schachtelung von Panels mit verschiedenen Layout-Managern kann das gesamte

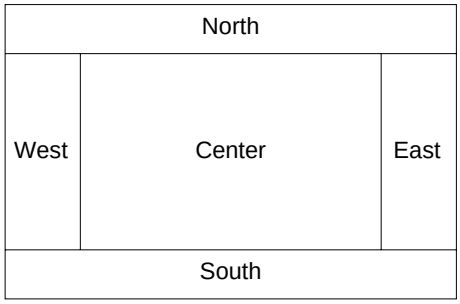


Abbildung 35: BorderLayout

Layout beliebig kombiniert werden.

Beispiel 7.3: Layout-Manager Zur Verdeutlichung soll ein praktisches Beispiel aus dem Programmieralltag dienen. Eine berühmte Therorie besagt, daß – wenn genügend Affen zufällig auf Schreibmaschinen tippen – irgendwann dabei auch die gesamte Weltliteratur herauskommt (*British Museum Algorithm*).

Nun möchten wir diese These empirisch untersuchen (zu dem entsprechenden Beweis kommen Sie in einer weiterführenden Vorlesung). Wir sind natürlich gegen Tierversuche und wollen daher lieber den Computer als Simulator für die Affen nutzen. Der Simulator soll zufällig Sätze erzeugen und anzeigen. Dabei soll ein Knopf "Verwerfen" den aktuellen Satz verwerfen und einen neuen generieren, ein Knopf "Speichern" soll den Satz in eine angezeigte Liste "würdiger" Ergebnisse aufnehmen.

Das Layout soll dabei so aussehen, daß die Knöpfe nur den nötigen Platz einnehmen, der gesamte überschüssige Platz im Fenster soll an die Ergebnisliste gehen.

Wir wählen also die Klasse **BorderLayout**, deren "Center" wir mit der Liste (Klasse **List**) füllen. Den unteren Bereich füllen wir mit einem **Panel**, das die beiden Knöpfe (Klasse **Button**) enthält. Für die Anordnung der beiden Knöpfe im Panel verwenden wir ein **FlowLayout**-Objekt (Zeile 15).

```
import java.awt.*;                                                    // 1
import java.awt.event.*;                                              // 2

class GhostWriter extends Frame implements ActionListener {          // 3

    List log;                                  // Listen-GUI-Element // 4
    int count = 1;                                                    // 5

    GhostWriter () {                                                  // 6
        super ("Die Affen tippen...");          // Fenstertitel setzen // 7
        setLayout (new BorderLayout ());                              // 8

        Button buttonAccept = new Button ("speichern");   // Button erzeugen // 9
        buttonAccept.setActionCommand ("speichern");      // Event zuordnen // 10
        buttonAccept.addActionListener (this );          // Eventhandler setzen // 11
```

```
        Button buttonReject = new Button ("verwerfen");          // noch ein Button // 12
        buttonReject.setActionCommand ("verwerfen");                                  // 13
        buttonReject.addActionListener (this );                                       // 14

        Panel buttonPanel = new Panel (new FlowLayout ());          // Panel fuer // 15
        buttonPanel.add (buttonAccept);                               // Buttons // 16
        buttonPanel.add (buttonReject);                                         // 17

        log = new List ();                                    // Liste initialisieren // 18

        add ("Center", log);                                // Liste und ButtonPanel // 19
        add ("South", buttonPanel);                           // ins Fenster einfuegen // 20
        addWindowListener (new Closer ());                  // Eventhandler fuer Close // 21

        generateNewSentence ();                           // Einen neuen Satz erzeugen // 22
        pack ();                                                        // layouten // 23
        show ();                                                 // Fenster anzeigen // 24
    }                                                                           // 25

    public void actionPerformed (ActionEvent event) {                           // 26
        if (event.getActionCommand ().equals ("verwerfen")) log.remove (0);     // 27
        generateNewSentence ();                                                 // 28
    }                                                                           // 29

    void generateNewSentence () {                      // Zufaelligen Satz erzeugen // 30
        String sentence = "";                                                   // 31

        do {                                                                    // 32
            char c = (char) (Math.random () * 30 + (int) 'a');                  // 33
            sentence = sentence + (c > 'z' ? ' ' : c);                          // 34
        } while (Math.random () < 0.95);                                        // 35

        if (sentence.trim ().equals ("")) {                  // Leerstring abfangen // 36
            sentence = "\143\157\147\151\164\157\040";                          // 37
            sentence += "\145\162\147\157\040\163\165\155";                     // 38
        }                                                                       // 39

        log.add ("#"+ (count++) + ": "+ sentence.trim (), 0);                   // 40
    }                                                                           // 41

    public static void main (String [] argv) {                                  // 42
        new GhostWriter ();                                                     // 43
    }                                                                           // 44
}                                                                               // 45
```

Mit dem parameterisierten Aufruf des Konstruktors der Superklasse in Zeile 7 wird der Titel des Fensters gesetzt.

Mit **setActionCommand** in den Zeilen 10 und 13 wird den Knöpfen jeweils eine Zeichenkette zugeordnet, mit der sie im entsprechenden Listener (Interface **ActionListener**, Methode **ActionPerformed**) einfach identifiziert werden können. Die Verbindung zwischen den Knöpfen und der Ereignisbehandlung wird mit **addActionListener** in den

Zeilen 11 und 14 vorgenommen.

In Zeile 18 wird ein graphisches Element vom Typ **List** angelegt, das eine Liste von Objekten speichern und diese mittels der **toString**-Methoden anzeigen kann. Mit **add** fügen wir in Zeile 40 ein neues Element in die Liste ein. Falls der aktuelle Eintrag verworfen werden soll, wird er in Zeile 27 wieder aus der Liste gelöscht.

Die Klasse List bietet noch mehr Möglichkeiten wie etwa die Verwaltung von Selektionen, die hier nicht genutzt werden. Eine ausführliche Beschreibung dieser Klasse und auch der anderen GUI-Elemente findet sich in der API-Referenz.

Wenn wir mit unserem Programm eine größere Menge an klugen Sätzen gesammelt haben, wird irgendwann der Platz in dem Fenster nicht mehr ausreichen, um die gesamte Liste darzustellen. In diesem Fall wird in die Liste – ohne daß wir uns darum kümmern müssten – eine Bildlaufleiste eingeblendet, mit der wir den sichtbaren Bereich in der Liste verschieben können.

Bei unserem Hallo Welt-Beispiel können wir dagegen das Fenster so klein machen, daß ein Teil des Inhaltes verdeckt wird, ohne daß eine Bildlaufleiste erscheint. Wünschen wir ein ähnliches Verhalten wie bei der Liste, können wir Elemente ohne eigene Bildlaufleiste in eine **ScrollPane** einfügen. Die **ScrollPane** ist ein spezieller Container, der automatisch Bildlaufleisten erhält, wenn der verfügbare Platz zur vollständigen Darstellung des Inhalts nicht ausreicht.

7.5 Selbst malen

Bisher haben wir uns darauf beschränkt, unser Programm aus "vorgefertigten" Komponenten zusammenzustellen. Natürlich kann es vorkommen, daß wir etwas anzeigen möchten, für das es keine vorgefertigte Komponente gibt. In diesem Fall müssen wir einen Erben der Klasse **Canvas** implementieren. Die Methode **paint** dieser Klasse überschreiben wir mit unserer eigenen Routine, in der wir den Inhalt des Elementes nach unseren Vorstellungen zeichnen.

Die Methode **paint** erhält dazu einen Grafikkontext (**Graphics**) übergeben, der verschiedene Zeichenfunktionen zur Verfügung stellt.

Da JAVA natürlich nicht wissen kann, wieviel Platz unser Objekt auf dem Bildschirm bennötigt, sollten wir zusätzlich die Methode **getPreferredSize** überschreiben.

Beispiel 7.4: Für unser Beispiel möchten wir ein Anzeigeobjekt entwickeln, das den Zustand eines von uns erschaffenen künstlichen Lebewesens, dem Javagochi, visualisiert.

```
import java.awt.*;                                                    // 1
import java.awt.event.*;                                              // 2

class JavagochiLight extends Canvas implements ActionListener {      // 3
    double gewicht;                                                   // 4
    static final int NORMALGEWICHT = 10;                              // 5

    JavagochiLight () {                                               // 6
        gewicht = NORMALGEWICHT;                                      // 7
        setBackground (Color.white);          // weisser Hintegrund   // 8
```

```
        Button buttonFuettern = new Button ("Füttern");          // Button erzeugen // 9
        buttonFuettern.setActionCommand ("Fuettern");  // ActionCommand zuordnen // 10
        buttonFuettern.addActionListener (this );              // Eventhandler festlegen // 11

        Button buttonVerdauen = new Button ("Sport");          // noch ein Button // 12
        buttonVerdauen.setActionCommand ("Sport");                                 // 13
        buttonVerdauen.addActionListener (this );                                  // 14

        Panel buttonPanel = new Panel ();                     // Panel fuer Buttons // 15
        buttonPanel.add (buttonFuettern);                                          // 16
        buttonPanel.add (buttonVerdauen);                                          // 17

        Frame frame = new Frame ("JavagochiLight");          // ein Fenster erzeugen // 18
        frame.add ("Center", this );                                 // ins Fenster // 19
        frame.add ("South", buttonPanel);                                          // 20
        frame.pack ();                                                             // 21
        frame.show ();                                                             // 22
        frame.addWindowListener (new Closer ());          // aus dem letzten Beispiel // 23
    }                                                                              // 24

    public void actionPerformed (ActionEvent actionEvent) {          // Eventhandler // 25
        String command = actionEvent.getActionCommand ();                          // 26

        if (command.equals ("Sport")) gewicht--;                      // abnehmen? // 27
        else if (command.equals ("Fuettern")) gewicht++;              // zunehmen? // 28

        repaint ();                                          // neuzeichnen veranlassen // 29
    }                                                                              // 30

    public Dimension getPreferredSize () {                                         // 31
        return new Dimension (200, 200);                                           // 32
    }                                                                              // 33

    public void paint (Graphics graphics) {                                        // 34
        int canvasHeight = getSize ().height;                        // etwas rechnen // 35
        int canvasWidth = getSize ().width;                                        // 36
        int height = Math.min (canvasHeight, canvasWidth) / 2;                     // 37
        int width = (int) (height * gewicht / NORMALGEWICHT);                      // 38
        int x0 = (canvasWidth - width) / 2;                                        // 39
        int y0 = (canvasHeight - height) / 2;                                      // 40

        graphics.drawOval (x0, y0, width, height);                     // zeichnen // 41
    }                                                                              // 42

    public static void main (String [] argv) {                                    // 43
        new JavagochiLight ();                                                     // 44
    }                                                                              // 45
}                                                                                  // 46
```

Das wesentlich Neue in diesem Beispiel steckt erwartungsgemäß in der **paint**-Methode (Zeilen 34 bis 42), wo die Größe des Zeichenbereiches ermittelt (**getDimension**, Zeilen 35 und 36) und anschließend eine Ellipse – zusätzlich abhängig von dem aktuellen Gewicht unseres Lebewesens – gezeichnet wird (**drawOval** [23], Zeile 41).

Bei **getPreferredSize** haben wir es uns etwas einfach gemacht und geben ein festes Werte-Paar zurück. In realen Anwendungen lassen sich sicher sinnvollere Belegungen – wie etwa ein fester Prozentsatz der verfügbaren Bildschirmfläche – finden.

Zugegebenermaßen ist etwas Phantasie erforderlich, um sich unter der sehr abstrakten Darstellung durch eine Ellipse ein Lebewesen vorzusellen. Immerhin findet sich auf der CD zur Vorlesung eine Version des Javagochi, die ein Gesicht besitzt, in dem man sogar ablesen kann, wie glücklich das Javagochi gerade ist.

Ein wichtiger Punkt in dem Beispielprogramm ist, daß wir die Methode **paint** überschreiben, aber **repaint** aufrufen. Warum wird **paint** nicht direkt aufgerufen?

repaint sagt dem System, daß das entsprechende Element neu gezeichnet werden muß. Das Neuzeichnen selbst wird von JAVA später durch Aufruf der Methode **paint** veranlaßt. So kann das System unter Umständen mehrere Aufrufe von **repaint** zusammenfassen. Auch wenn es möglich und in einigen Büchern sogar beschieben ist, sollte in keinem Fall **paint** direkt aufgerufen werden.

7.6 AWT und Swing

Bisher haben wir nur Komponenten des *Abstract Window Toolkit* (AWT) von JAVA beschrieben. Das AWT legt stellt für Bedienelemente der verschiedenen Oberflächen, unter denen JAVA läuft, ein einheitliches Interface zur Verfügung. Diese Oberflächen unterscheiden sich jedoch sehr, so daß oft Elemente, die bei einer Oberfläche automatisch vorhanden sind, für eine andere Oberfläche komplett nachprogrammiert werden müssen, will man sich nicht auf die Schnittmenge beschränken. Außerdem steigt der Gesamtaufwand für die Anpassungen der Schnittstellen mit der Komplexität der Elemente und der Anzahl der Unterstützten Plattformen.

Also hat Sun mit Swing einen neuen Weg eingeschlagen: Statt alle Komponenten für x Systeme anzupassen und dann auf einem Teil der Systeme doch selbst darstellen zu müssen, verwaltet Swing direkt alle Elemente selbst und kümmert sich auch immer selbst um die Darstellung. Da für Swing-Elemente keine besonderen Systemressourcen beansprucht werden, werden diese auch Leichtgewicht-Komponenten (Lightweight-Components) genannt.

Die Swing-Klassen heißen – falls es ein Äquivalent gibt – genauso wie die AWT-Klassen, nur daß dem Klassennamen ein "J" vorangestellt ist. Einen größeren Unterschied gibt es bei dem **JFrame**, in das untergeordnete Komponente nicht mehr direkt eingehängt werden können. Details dazu finden sich in der API-Beschreibung.

Ein weiterer essentieller Unterschied zwischen dem AWT und Swing ist, daß bei Komponenten wie Tabellen und Listen das Anzeigeelement und der Inhalt getrennt werden. Es können sich also zwei Listen eine Datenstruktur für den Inhalt teilen. Auch eine Datenbank kann problemlos als Inhaltsmodell für eine Liste herangezogen werden, wenn das entsprechende Interface bei der Datenbank bereitgestellt wird. Dagegen verwaltet die Liste aus dem GhostWriter-Beispiel alle Einträge selbstständig im Hauptspeicher. Es macht aber wenig Sinn, etwa eine große Datenbank in eine Liste umzukopieren – spätestens wenn

[23] Ja, ich kenne den Unterschied. **drawOval** zeichnet wirklich eine Ellipse.

auch der virtuelle Speicher erschöpft ist, gibt es mit diesem Modell ein Problem.

7.7 Was wissen Sie jetzt?

Sie sollten nun in der Lage sein, einfache Programme mit einer graphischen Benutzeroberfläche zu schreiben. Insbesondere haben Sie das Konzept der ereignisorientierten Programmierung verstanden. Sie wissen, wie Container und Layoutmanager zu benutzen sind, um Elemente in einem Fenster anzuorden. Sie sind in der Lage, eigene Elemente durch Programmierung eines Erben von **Canvas** zu entwickeln.

8 Nebenläufige Programmierung

Nebenläufige Programmierung bedeutet, daß in unserem Programm verschiedene Pfade (Threads) gleichzeitig verfolgt werden. Eigentlich kann ein "normaler" Rechner mit einer CPU immer nur ein Programm gleichzeitig ausführen, die Parallelausführung wird dann durch häufiges Umschalten zwischen den einzelnen Pfaden simuliert. Dieses Umschalten kostet sogar zusätzliche Rechenzeit und durch Parallelisierung können neuartige Probleme in Programmen auftreten, auf die später noch weiter eingegangen wird. Wozu dient also dieser ganze Aufwand?

Ein gutes Beispiel für den Sinn von Parallelverarbeitung ist ein Web-Browser: Während parallel noch Bilder aus dem Netz geladen werden, kann die unvollständige Seite oft schon gelesen werden. Parallelverarbeitung macht immer dann Sinn, wenn mit einer bestimmten Frequenz Aufgaben immer wieder auszuführen sind, auf Ereignisse reagiert werden soll oder Aufgaben auszuführen sind, bei denen die CPU die meiste Zeit auf Daten wartet statt effektiv zu arbeiten.

8.1 Threads

Für die Parallelverarbeitung stellt JAVA eine spezielle Klasse zur Verfügung: Erben der Klasse **Thread** können die Methode **run** mit parallel zu verarbeitenden Schritten füllen. Durch Aufruf der **Thread**-Methode **start** wird die **run**-Methode im Hintergrund parallel ausgeführt.

Beispiel 8.1: Thread Als Beispiel soll wieder unser künstliches Lebewesen aus dem letzten Kapitel dienen: Es soll einen Verdauungs-Thread erhalten, der das Gewicht alle fünf Sekunden um einen Zähler vermindert.

Die Methode **sleep** legt dabei den aktuellen Thread eine einstellbare Anzahl von Millisekunden schlafen. Da Der Thread von außen mit einer Unterbrechung "aus dem Schlaf gerissen" werden kann, muß die entsprechende Meldung (Exception) abgefangen werden. In einer "richtigen" Simulation könnte die Zeitdifferenz mit **System.getCurrentTimeMillis** genauer bestimmt und als Faktor bei der Gewichtsreduktion eingerechnet werden.

```
class Verdauung extends Thread  {                                    // 1
    JavagochiLight javagochi;                                        // 2

    Verdauung (JavagochiLight _javagochi)  {                         // 3
```

```
        javagochi = _javagochi;                                                    // 4
    }                                                                              // 5

    public void run () {                                                           // 6
        while (true) {                                                             // 7
            try {                                                                  // 8
                sleep (5000);                                  // 5 sek warten // 9
            } catch (InterruptedException e) {    }                            // 10
            javagochi.gewicht--;                                 // abnehmen // 11
            javagochi.repaint ();                               // neu zeichnen // 12
        }                                                                          // 13
    }                                                                              // 14

    static void main (String [] argv) {                                            // 15
        new Verdauung (new JavagochiLight ()).start ();                            // 16
    }                                                                              // 17
}                                                                                  // 18
```

8.2 Synchronisation

Mit der Nebenläufigkeit handeln wir uns auch neue Probleme ein: Haben zwei Threads parallel Zugriff auf eine gemeinsame Datenstruktur, kann dies zu erheblichen Problemen führen. Zur Verdeutlichung soll uns wieder ein Beispiel dienen.

Beispiel 8.2: Mit unseren neuen Kenntnissen sind wir bereit für neue Herausforderungen. Die Nexus GmbH hat den neuen Roboter-Typ "Marvin" entwickelt. Durch seine einzigartigen "reflection"-Fähigkeiten ist er in der Lage, sein eigenes Betriebssystem zu analysieren. Dabei verfällt er jedoch immer wieder in Depressionen, da das Betriebssystem aus Kompatibilitätsgründen nicht verbessert werden kann. Unsere Aufgabe ist nun, ein Zusatzmodul zu programmieren, das den Zustand des Roboters überwacht und mit Glückshormonen versorgt, falls der Depressionswert zu nahe an die kritische Marke 100 kommt, bei der wegen übermäßiger Depression die Schaltkreise irreparabel beschädigt würden.

Die strengen Vorschriften der Herstellerfirma besagen, daß aus Sicherheitsgründen alle Komponenten doppelt ausgeführt sein müssen.

Wir starten also unseren Überwachungsthread einfach doppelt – falls einer mal abstürzt, kann sich der andere immer noch um die Hormone kümmern.

```
class Marvin extends Thread {                                                      // 1
    boolean hormone = false;                                                       // 2
    boolean defekt = false;                                                        // 3
    double depression = 25;                                                        // 4

    class Watchdog extends Thread {                  // Ueberwachungs-Thread // 5

        public void run () {                                                       // 6
```

```
            while (true) {                                                    // 7
                if (depression > 75 && !hormone) setHormone (true);           // 8
                else if (depression < 50 && hormone) setHormone (false);      // 9

                try {                                                         // 10
                    sleep (2000);                          // 2 Sek. schlafen // 11
                } catch (InterruptedException e) {   } ;                      // 12
            }                                                                 // 13
        }                                                                     // 14
}                                                                             // 15

Marvin () {                                              // Roboter-Thread   // 16
        new Watchdog ().start ();                                 // sicher ist // 17
        new Watchdog ().start ();                                   // sicher // 18
}                                                                             // 19

public void run () {                                                          // 20
        while (true) {                                                        // 21
            System.out.println ("Depressionen: "+ depression);                // 22
            try {                                                             // 23
                Thread.sleep (1000);                          // Depressionen // 24
            } catch (InterruptedException e) {   }       // aendert sich 1x pro Sek. // 25

            depression += hormone ? -5 : 5;                    // um 5 Prozent // 26

            if (depression > 99) {                           // total deprimiert? // 27
                System.out.println ("Kurzschluss");              // Kurzschluss // 28
                System.exit (0);                                              // 29
            }                                                                 // 30
        }                                                                     // 31
}                                                                             // 32

void setHormone (boolean einaus) {                  // Hormonpumpe ein / aus // 33
        if (einaus != hormone && !defekt) {                                   // 34
            System.out.println ("Schalte Hormonpumpe "+ (einaus ? "ein": "aus")); // 35

            try {                                                             // 36
                Thread.sleep (1000);                 // Operation dauert etwas // 37
            } catch (InterruptedException e) {   }                            // 38

            if (hormone == einaus) {                  // sollte nicht vorkommen, // 39
                hormone = false;                     // da im if oben abgefangen // 40
                defekt = true;                                                // 41
                System.out.println ("Doppelschaltung - Hormonpumpe zerstoert!");// 42
            } else {                                                          // 43
                hormone = einaus;                                             // 44
                String msg = einaus ? "ein": "aus";                           // 45
                System.out.println ("Hormonpumpe "+ msg + "geschaltet");      // 46
            }                                                                 // 47
        }                                                                     // 48
}                                                                             // 49

public static void main (String [] argv) {                                    // 50
```

```
            new Marvin ().start ();                                    // 51
        }                                                              // 52
}                                                                      // 53
```

Starten wir das Beispiel, führt es trotz unserer Sicherheitsmechanismen zum Kurzschluß. Mit nur einem Thread funktioniert der Hormonregler dagegen einwandfrei, wovon wir uns durch Auskommentieren von Zeile 18 leicht überzeugen können. Das Problem sollte also wahrscheinlich etwas mit der Nebenläufigkeit zu tun haben. Tatsächlich ist die Ursache für die Fehlfunktion, daß beide Überwachungsthreads bei Erreichen der Depressionsgrenze in die Methode **setHormone** springen. Diese Methode ist jedoch nicht *reintrant*, das heißt, sie ist nicht dafür ausgelegt, daß sie von zwei Threads "gleichzeitig" abgearbeitet wird:

Das Flag **hormone** wird am Anfang der Methode **setHormone** abgefragt, um das Einschalten der Hormonpumpe zu vermeiden, wenn diese bereits eingeschaltet ist. Bevor jedoch der erste Thread dieses Flag in Zeile 44 setzen kann, gelangt der zweite Thread in die Methode, und liest noch den alten Wert. Die Pumpe wird also doppelt eingeschaltet, was zum Ausfall führt.

Auch eine Verlegung der Abfrage, ob die Hormonpumpe bereits aktiviert ist, an den Anfang der Methode beseitigt das Problem nicht prinzipiell – ein Problemfall wird nur unwahrscheinlicher. Zwischen der Abfrage und dem Umsetzen der Variable kann immer ein anderer Thread noch den veralteten Wert lesen.

Diese Programmstelle darf also nicht parallel ausgeführt werden. Solche Bereiche eines Programmes heißen "kritische Sektionen".

Glücklicherweise besitzt Java zur Lösung dieses Problemes wieder einen einfachen Mechanismus: Wird die Methode **setHormone** als *synchronized* deklariert, stellt JAVA sicher, daß sie immer nur von einem Thread gleichzeitig "betreten" werden kann. Allgemein werden alle synchronisierten Methoden eines Objektes vor parallelem Zugriff geschützt, alle Aufrufe werden automatisch serialisiert.

8.3 Deadlocks

Aber selbst der Einsatz der Synchronisation kann uns nicht automatisch vor allen Problemen schützen, die aus der Parallelverarbeitung resultieren.

Beispiel 8.3: Deadlocks Das Studentenwerk plant als besondere Attraktion für die Mensa eine chinesische Woche. Um die Preise nicht erhöhen zu müssen, kann jedoch nur ein Stäbchen an jeden Studenten ausgegeben werden. Das Problem soll dadurch gelöst werden, daß sich jeweils zwei benachbarte Studenten ein Stäbchen teilen. Zur Vereinfachung wird angenommen, daß alle Studenten nebeneinander an einem runden Tisch sitzen. Um dieses Modell vorab zu untersuchen, sind wir beauftragt, das Eßverhalten der Studenten zu simulieren.

Die Studenten sollen dabei jeweils durch einen Thread simuliert werden, der zuerst das rechte Stäbchen nimmt – falls verfügbar –, dann das linke Stäbchen, dann etwas ißt, und dann beide Stäbchen wieder ablegt. Die Verfügbarkeit von Stäbchen modellieren wir dabei durch ein Boolesches Feld. Beim Greifen soll die Simulation zuerst prüfen, ob

das entsprechende Stäbchen verfügbar ist und im Erfolgsfall als nicht mehr verfügbar markieren.

Zwischen Abfrage und Setzen des Flags könnte ein anderer Thread das Flag auch abfragen, erhält "frei" – beide Threads besäßen das gleiche Stäbchen. Diese kritische Sektion (**getStick**) deklarieren wir also direkt als *synchronized*, um die Probleme aus dem letzten Beispiel zu vermeiden.

```
class Mensa  {                                                                    // 1
    static boolean [] stickAvailable;                                             // 2

    class Student extends Thread  {                                               // 3
        int id;                                                                   // 4

        Student (int _id)  {                                                      // 5
            id = _id;                                                             // 6
        }                                                                         // 7

        void action (String description)  {                                       // 8
            System.out.println ("Student "+id+"is "+description);                 // 9
            try  {                                                                // 10
                sleep ((long) (Math.random () * 2000));                           // 11
            }  catch (InterruptedException e)  {    } ;                           // 12
        }                                                                         // 13

        public void run ()  {                                                     // 14
            while (true)  {                                                       // 15
                action ("thinking");                                              // 16

                while (!getStick (id))  {    }  // versuchen, Staebchen aufzunehmen // 17

                while (!getStick ((id+1) % stickAvailable.length))  {             // 18
                    action ("trying to get 2nd Stick");                           // 19
                }                                                                 // 20

                action ("eating");                                  // njam njam // 21

                putStick (id);                                // fertig, beide Staebchen // 22
                putStick ((id + 1) % stickAvailable.length);        // zuruecklegen // 23
            }                                                                     // 24
        }                                                                         // 25
    }                                                                 // class Student // 26

    Mensa (int count)  {                                                          // 27
        stickAvailable = new boolean [count];                                     // 28

        for (int i = 0; i < count; i++) stickAvailable [i] = true;   // Staebchen verteilen // 29
        for (int i = 0; i < count; i++) new Student (i).start ();    // Studenten starten // 30
    }                                                                             // 31

    void putStick (int i)  {                                                      // 32
        System.out.println ("Student releases stick "+ i);                       // 33
```

```
        stickAvailable [i] = true;                                          // 34
    }                                                                       // 35

    synchronized boolean getStick (int i) {                                 // 36
        if (stickAvailable [i]) {                                           // 37
            stickAvailable [i] = false;                          // got stick i // 38
            System.out.println ("Student took stick "+i);                   // 39
            return true;                                                    // 40
        } else {                                                            // 41
            return false;                                  // nicht verfuegbar // 42
        }                                                                   // 43
    }                                                                       // 44

    public static void main (String[] argv) {                               // 45
        new Mensa (5);                    // Mensa mit 5 Studenten erzeugen // 46
    }                                                                       // 47
}                                                                           // 48
```

Starten wir das Programm, sehen wir, daß nach einer Weile kein Student mehr ißt, sondern alle auf ein Stäbchen warten. Was ist passiert?

Alle Studenten haben das rechte Stäbchen ergriffen, halten also ein Stäbchen. Da kein Student sein Stäbchen wieder hergibt, kann auch kein Student das fehlende linke Stäbchen bekommen.

Eine solche Situation, in denen Threads wechselseitig auf die Freigabe einer Ressource warten, um weiterarbeiten zu können, wird *Deadlock* genannt.

8.4 Schlafen und aufwecken

Natürlich läßt sich auch dieses Problem lösen: Dürfen Studenten in der kritischen Sektion nur beide Stäbchen nehmen oder keines, kann ein Deadlock nicht mehr auftreten.

Es gibt aber noch einen weiteren Punkt, der in dem Beispiel nicht besonders elegant gelöst ist: Die Studenten warten "aktiv" in einer Schleife darauf, daß ein Stäbchen verfügbar wird, verbrauchen also unnötig wertvolle Rechenzeit in unserer Simulation. Für kritische Sektionen gibt es eine spezielle Methode **wait**, die die Sperre für andere Threads, eine synchronisierte Methode dieses Objektes zu betreten, aufhebt. Der Thread, der **wait** aufruft, wird dabei solange "schlafen gelegt", bis von einem anderen Thread schlafende Threads durch Aufruf von **notify** oder **notifyAll** aus einem synchronisierten Bereich "geweckt" werden. Besitzer der Sperre wird dabei der geweckte Thread. Werden mehrere Threads geweckt, werden ihre kritischen Sektionen sequentiell abgearbeitet, es ist also auch beim Wecken mehrerer Threads sichergestellt, daß immer nur einer sich in einer kritischen Sektion befindet.

Beispiel 8.4: Schlafen und Aufwecken Als Beispiel für Schlafen und Aufwecken verbessern wir die Mensa-Simulation mit unserem neuen Wissen. **getSticks** bekommt nun beide Stäbchen bzw. legt sich schlafen, bis beide verfügbar sind, **putSticks** weckt schlafende Studenten durch Aufruf von **notifyAll**: Sobald jemand seine Stäbchen ablegt, besteht die Möglichkeit, daß ein bisher "schlafender" Student beide Stäbchen bekommen kann.

```
class Mensa2                                                                    // 1
{                                                                               // 2
    boolean [] stickAvailable;                                                  // 3

    class Student extends Thread {                                              // 4
        int id;                                                                 // 5

        Student (int _id) {                                                     // 6
            id = _id;                                                           // 7
        }                                                                       // 8

        void action (String description) {                                      // 9
            System.out.println ("Student "+id+"is "+description);               // 10

            try {                                                               // 11
                sleep ((long) (Math.random () * 2000));                         // 12
            } catch (InterruptedException e) { } ;                              // 13
        }                                                                       // 14

        public void run () {                                                    // 15
            while (true) {                                                      // 16
                action ("thinking");                                            // 17
                getSticks (id);                                                 // 18
                action ("eating");                                              // 19
                putSticks (id);                                                 // 20
            }                                                                   // 21
        }                                                                       // 22
    }                                                                           // 23


    Mensa2 (int count) {                                                        // 24
        stickAvailable = new boolean [count];                                   // 25

        for (int i = 0; i < count; i++)                                         // 26
        stickAvailable [i] = true;                                              // 27

        for (int i = 0; i < count; i++)                                         // 28
        new Student (i).start ();                                               // 29
    }                                                                           // 30

    synchronized void putSticks (int i) {                                       // 31
        System.out.println ("Student "+i+"releases sticks");                   // 32
        stickAvailable [i] = true;                                              // 33
        stickAvailable [(i + 1) % stickAvailable.length] = true;                // 34
        notifyAll ();                                                           // 35
    }                                                                           // 36

    synchronized void getSticks (int i) {                                       // 37
        while (!(stickAvailable [i] && stickAvailable [(i + 1) % stickAvailable.length])) {  // 38
            System.out.println ("Student "+i+"is waiting");                     // 39
            try {                                                               // 40
                wait ();                                                        // 41
```

```
                } catch (InterruptedException e) {    }                    // 42
            }                                                              // 43
            stickAvailable [i] = false;                                    // 44
            stickAvailable [(i+1) % stickAvailable.length] = false;        // 45
        }                                                                  // 46


        public static void main (String[] argv) {                          // 47
            new Mensa2 (5);                                                // 48
        }                                                                  // 49
}                                                                          // 50
```

8.5 Was wissen Sie jetzt?

Sie sollten nun in der Lage sein, einfache nebenläufige Programme zu schreiben und die damit verbundenen Gefahren kennen. Sie wissen, wie kritische Bereiche geschützt werden können und wie aktives Warten mit **wait** und **notify** vermieden werden kann.

9 Netzwerkintegration und verteilte Programmierung

Wie schon in Abschnitt 3 erwähnt, ist bei JAVA die Unterstützung verteilter Programmierung, das Nutzen von anderswo deklarierten Klassen Teil des Konzepts der Sprache. JAVA paßt deshalb gut in die vernetzte Rechnerwelt.

9.1 Client – Server

Die zunehmende Vernetzung von Computern (und den von ihnen verwalteten Informationen) führt zu einer steigenden Netzlast. Das Schlagwort *Client – Server* ist deshalb so beliebt, weil durch verteiltes Rechnen versucht wird, die Netzlast zu senken.

Definition 9.1: Server Eine Soft- oder Hardware, die anderen Soft- oder Hardwarekomponenten Dienste über eine eine Kommunikationsschnittstelle anbietet.

Definition 9.2: Client Eine Soft- oder Hardware, die von Servern angebotene Dienste nutzt.

Ein Beispiel für eine Anwendung dieser Technologie wäre ein Client-Programm, das die lokale Eingabe von komplexen Rechenanweisungen ermöglicht und Teilaufgaben an einen Rechner (Server) überträgt, der auf das Lösen von bestimmten Rechenanweisungen spezialisiert ist. Ein aktuelles Beispiel sind Programme, die WWW-Server durchsuchen (sog. Web-Spider) und im Verlauf dieser Aktivität alle für sie zugänglichen Seiten eines WWW-Servers lokal herunterladen, dort verarbeiten und das Ergebnis in einer Datenbank speichern. Diese enorme Netzlast ließe sich durch den Einsatz der Client-Server Technologie drastisch verringern: auf dem WWW-Server wäre ein Programm, das auf eine gezielte Anfrage des Web-Spiders antworten würde. Der Web-Spider muß sich darauf verlassen können, daß seine Anfragen verstanden und beantwortet werden können. Dafür kann er

darauf verzichten, all die WWW-Seiten bei sich zu speichern. Häufig scheitert der Einsatz dieser Technik an der Heterogenität der Netzwerkumgebungen, den Sicherheitsproblemen und an der Komplexität der Protokollspezifikation. Das *Remote Method Invocation* (RMI) Konzept von JAVA löst zumindest zwei der drei Probleme:

- **Heterogenität**: da JAVA nahezu maschinenunabhängig ausführbar und für zahlreiche Plattformen verfügbar ist.
- **Komplexität**: da sich mit RMI mit minmalem Aufwand eine Client-Server Anwendung entwickeln läßt.
- **Sicherheit**: dieses Problem löst JAVA nicht [24] .

9.2 Remote Methode Invocation

Die RMI-Schnittstelle[25] ermöglicht die Kommunikation von Objekten, die sich auf verschiedenen Rechnern befinden.

Definition 9.3: Remote-Objekt Ein Remote-Objekt ist ein Objekt, das zur Laufzeit durch die JAVA Virtual Machine A gehalten wird und von Objekten, die sich in einer Maschine B befinden, angesprochen werden kann. Das Remote-Objekt kann also auch als Server-Objekt aufgefaßt werden.

Unter der Oberfläche von RMI verbergen sich Konzepte wie Serialisierung, Sockets und Ströme, die jedoch den Rahmen dieses Skriptes sprengen würden. Hier sehen wir nur anhand eines kleinen Beispiels, wie das RMI verwendet wird.

9.2.1 Stümpfe und Skelette

Unter dieser ein wenig martialisch wirkenden Überschrift verbirgt sich die Frage nach dem "Wie?".

Wie kann ein Client-Objekt ein Remote-Objekt ansprechen, obwohl das Remote-Objekt lokal nicht existiert und dessen Struktur damit auch nicht bekannt ist?

Definition 9.4: Stumpf Ein Stumpf (engl. Stub) ist eine kleine Schnittstellen-Beschreibung einer Klasse, die die **UnicastRemoteObject**-Klasse extendiert. Der Stumpf wird zur Beschreibung eines Remote-Objektes an den Client-Rechner übertragen. Stümpfe werden über den `rmic`-Befehl durch den Programmierer erzeugt und der Programmierer hat keine Möglichkeit, diese abzuändern.

[24] An der Universität Dortmund werden Sicherheitsaspekte wie Verschlüsselung und Privatsphäre in der Spezialvorlesung Sicherheit behandelt

[25] RMI ist die Fortführung des Remote Procedure Call-Konzeptes (RPC) von SUN, das jedoch nicht für Objektorientierung ausgelegt war.

Definition 9.5: Skelett Ein Skelett (engl. Skeleton) ist dem Stumpf ähnlich, verbleibt jedoch auf dem Server.

Für die Klasse eines Remote-Objektes werden Stümpfe erzeugt. Da die Stümpfe nur die Beschreibung einer Klasse und nicht deren gesamte Funktionalität enthalten, können diese kostengünstig über das Netz übertragen werden.

Definition 9.6: Registratur Eine Registratur (engl.: registry) registriert und verwaltet Instanzen von Remote-Objekten, die in einer Maschine gehalten werden. Die Registratur stellt Informationen zu diesen Objekten bereit und kann üeber eine URL angesprochen werden.

Wie erhält eine Anwendung den Stumpf? Die Anwendung öffnet einen Kommunikationskanal zu einer Registratur. Die Anwendung fordert dann anhand eines Namens bei der Registratur ein Objekt an. Der Stumpf des angeforderten Objektes wird nun übertragen und kann in der Client-Anwendung angesprochen werden. Die Verbindung mit diesem virtuellen Objekt verdeckt die Kommunikationsvorgänge im Hintergrund. Die Client-Anwendung benötigt lediglich den Server-Namen und die Port-Nummer[26] der Registratur sowie den Namen des angeforderten Objektes.

Wie wird ein Objekt zu einem Remote-Objekt? Wie Abbildung 37 zeigt, beschreibt eine Schnittstelle **Server**, die die Schnittstelle **Remote** erweitert, ein Remote-Objekt und jede Klasse, die **Server** implementiert, muß somit alle von **Remote** angebotenen Methoden implementieren. Eine solche Klasse zur Beschreibung von Remote-Objekten ist die Klasse **ServerImpl**[27], die die **UnicastRemoteObject**-Klasse erweitert und die Schnittstelle **Server** implementiert. Die Stümpfe and Skelette müssen dann über den `rmic`-Befehl erzeugt werden (hier: **rmic ServerImpl**). Eine Instanz der Remote-Objekt-Klasse **ServerImpl** wird in der Applikation **ServerDaemon** erzeugt und in einer Registratur registriert. Dieses Objekt kann nun durch eine Applikation **ServerClient** angesprochen werden.

9.2.2 Ein Beispiel für einen Server

Ein kleines Beispiel für einen RMI-Server und einen RMI-Client demonstriert eindrucksvoll die simple Handhabung der RMI-Technologie. Die Schnittstelle **GameServer** erweitert die Schnittstelle **Remote** und beschreibt die vom Server für Clients zur Verfügung gestellten Methoden. Es sind Methoden zur Verwaltung einer Spielstandstabelle, deren Rangfolge anhand der absoluten Spielstände berechnet wird.

```
package RMIBeispiel;

import java.rmi.*;
import RMIBeispiel.DatabaseException;
```

[26] Eine Portnummer ist eine Art Telefonnummer, auf der ein anderer Rechner einen bestimmten Dienst erreichen kann. Z.B. ist Port 80 traditionell der Port eines HTTP-Servers (WWW-Server) und der Standard-Port einer Registratur ist 1099.

[27] Die Schreibweise `...Impl` ist natürlich nicht zwingend, erleichtert jedoch das Arbeiten mit den Schnittstellen

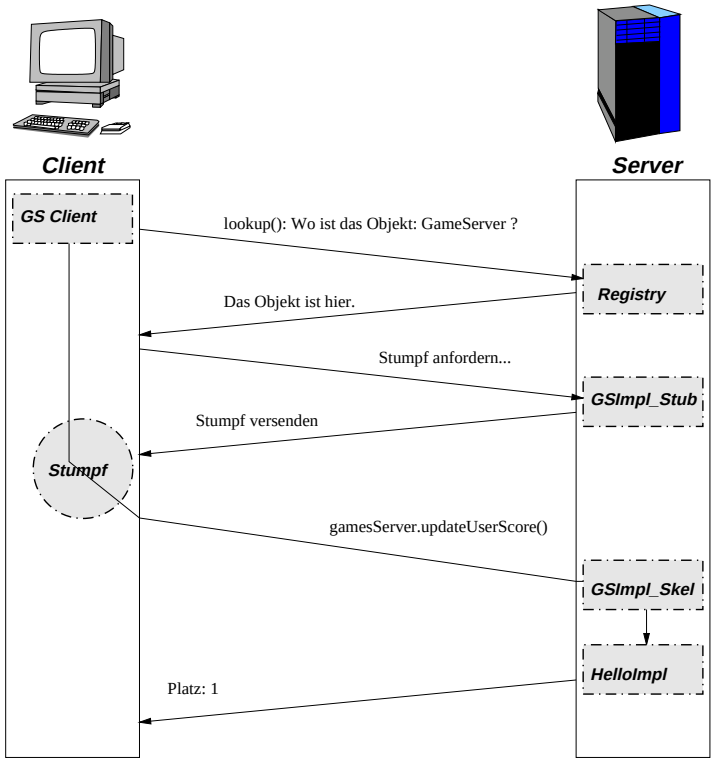


Abbildung 36: Kommunikation zwischen Client und Server für den Aufruf einer Methode eines Remote-Objektes durch ein Client-Objekt (Quelle: [Harold, 1997])

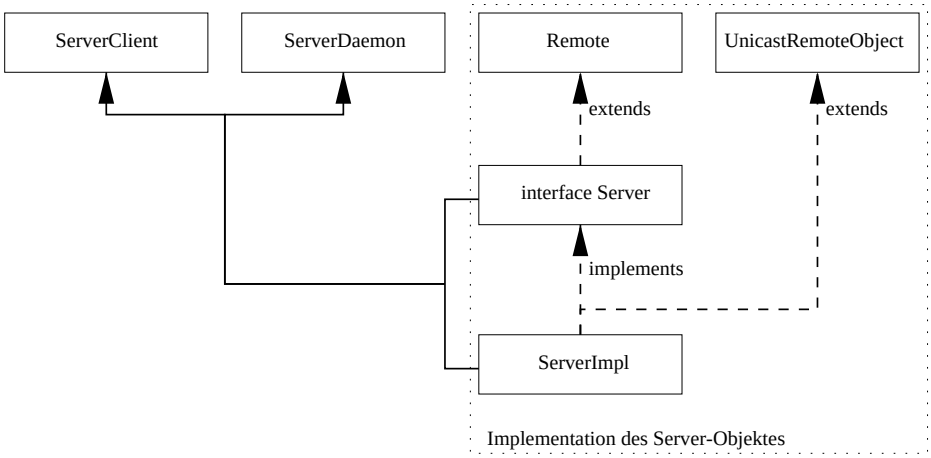


Abbildung 37: Übersicht der zentralen Schnittstellen und Klassen für ein RMI-Client/Server-System

```
public interface GameServer extends Remote {

    /**
    * Schreibt einen Spielstand in den Server ein.
    * Ist unter der Benutzerkennung bereits ein Spielstand vorhanden
    * wird dieser ueberschrieben.
    *
    * @param <code>Benutzername</code> Die eindeutige Benutzerkennung
    * des Spielers
    * @param <code>score</code> Der aktuelle Spielstand des Spielers.
    *
    * @return Den aktuellen Tabellenplatz in der High-Score-Liste
    *
    * @exception java.rmi.RemoteException
    * RMI-Fehler. Siehe Java-Dokumentation
    * @exception DatabaseException
    * Falls Probleme mit der Datenbank des Game-Repositorys
    * auftraten
    */
    public int updateScore(String userID, int score)
    throws RemoteException,
    DatabaseException;

    /**
    *
    * Gibt den Spielernamen und Score des nten Tabellenplatzes zurueck.
    * (name und score werden durch ein Sonderzeichen getrennt)
    *
    * @exception DatabaseException
    * Falls Probleme mit dem Game-Repository auftraten.
    *
    * @exception java.rmi.RemoteException
    * RMI-Fehler. Siehe Java-Dokumentation
    */
    public String getUserScore ( int placement )
    throws DatabaseException, RemoteException;

}                                                          //GameServer
```

Jede der in der **GameServer**-Schnittstelle zur Verfügung gestellten Methoden wird in der **GameServerImpl**-Klasse implementiert. Tatsächlich implementiert diese Klasse noch weitere Methoden sowie einen Konstruktor. Die **GameServerImpl**-Klasse muß die Klasse **UnicastRemoteObjekt** erweitern.

```
package RMIBeispiel;

import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
```

```
import java.net.*;

import RMIBeispiel.DatabaseException;
import RMIBeispiel.Entry;

import java.util.*;

public class GameServerImpl
extends UnicastRemoteObject implements GameServer  {
        ...
        /**
        * Der Konstruktor der Game-Server Klasse.
        * In diesem wird ein <code>Game-Repository</code> erzeugt.
        *
        * @exception DatabaseException
        * Falls Probleme mit dem Game-Repository auftraten
        *
        * @exception java.rmi.RemoteException
        * RMI-Fehler. Siehe Java-Dokumentation
        */
        public GameServerImpl (String filename)
        throws RemoteException, DatabaseException  {
                ...
        }


        public int updateScore (String userID, int score)
        throws RemoteException, DatabaseException  {
                ...
        } ;


        public String getUserScore ( int placement )
        throws DatabaseException, RemoteException  {
                ...
        } ;

}
```

Die RMI-Registratur muß erzeugt und verwaltet werden. Zusätzlich muß zumindest eine Instanz der Klasse **GameServerImpl** erzeugt und in der Registratur registriert werden. Diesen Part übernimmt die Applikationsklasse **GameServerDaemon**[28].

```
package RMIBeispiel;

import RMIBeispiel.DatabaseException;
import RMIBeispiel.GameServerImpl;
import RMIBeispiel.GameServer;
```

[28]Der Name Daemon kommt aus dem Bereich der Betriebssysteme. Als Dämon wird ein Prozeß bezeichnet der ständig im Speicher gehalten wird und ewig (quasi untot) weiterarbeitet. Z.B. ein Server

```
import java.io.*;
import java.rmi.*;
import java.rmi.registry.*;
import java.net.MalformedURLException;

public class GameServerDaemon
 {
      private GameServer gameServer;                                          //1

      public GameServerDaemon ()                                              //2
       {
            System.runFinalizersOnExit (true);                          // deprecated

            if (System.getSecurityManager ()==null)  {
                  System.setSecurityManager (new RMISecurityManager ());    //3
            }

            try  {
                  LocateRegistry.createRegistry (2060);                      //4
                  gameServer = new GameServerImpl ("spielstand.dat");        //5
                  Naming.bind ( "//kiew.cs.uni-dortmund.de"
                              + ":2060/GameServer", gameServer);             //6
            }  catch (Exception e)  {
                  e.printStackTrace () ;                  // Catch auf alle Exceptions //7
            }
      }

      public static void main (String[] args)
       {
            final GameServerDaemon daemon = new GameServerDaemon ();          //8
      }
}
```

In Zeile 1 wird ein **GameServer**-Objekt deklariert. Die Applikation instanziiert sich selbst (8) und ruft den Konstruktor (2) auf. Falls kein Sicherheitsmanager vorhanden ist, wird eine neue Instanz (3) dieser Klasse erzeugt. Die eigentlichen RMI-Aufrufe folgen nun: In Zeile 4 wird eine Registratur auf Port 2060 erzeugt und die in (5) erzeugte Instanz des über RMI zugreifbaren Objektes an den Namen "GameServer" gebunden (6). Die Methode **Naming.bind()** benötigt eine URL, die mit dem gewünschten Objektnamen abgeschlossen wird. Daß alle Ausnahmen (7) undifferenziert abgefangen werden, dient nur der Vereinfachung des Beispiels und ist nicht zur Nachahmung empfohlen.

Ein Server ist ohne Client sinnlos. Einen Beispiel-Client für unseren oben erstellten Server stellt die Klasse **GameServerClient** dar. In dieser Klasse wird eine Verbindung zur Registratur hergestellt und eine virtuelle Kopie des registrierten GameServer-Objektes erzeugt. Nach dem Erzeugen dieser virtuellen Kopie kann auf das Objekt wie üblich zugegriffen und dessen Methoden genutzt werden.

```
package RMIBeispiel;

import RMIBeispiel.DatabaseException;
import RMIBeispiel.GameServerImpl;
import RMIBeispiel.GameServer;

import java.rmi.*;
import java.rmi.registry.*;
import java.net.MalformedURLException;

public class GameServerClient
 {
      private GameServer gameServer;

      public GameServerClient ()
       {
            System.runFinalizersOnExit (true);                                  // deprecated
            System.setSecurityManager ( new RMISecurityManager ());             // 1

            try  {
                  gameServer = (GameServer)Naming.lookup ( "//kiew.cs.uni-dortmund.de"
                                                          + ":2060/GameServer");  // 2
            }  catch (Exception e)  {   e.printStackTrace ()   }

            try  {
                  for (int i=1;i<=1000;i++)  {
                        System.out.println ("Platz "+i+": "+gameServer.getUserScore(i)) ;
                  }
            }  catch (DatabaseException e)  {    }                              // 3
            catch (RemoteException e)  {   e.printStackTrace () ;   }           // 4

            try  {
                  gameServer.updateScore ("peter",1);                           // 5
                  gameServer.updateScore ("paul",100000);                       // 6
                  gameServer.updateScore ("mary",50);                           // 7
            }  catch (Exception e)  {   e.printStackTrace () ;   }              // 8

            try  {
                  for (int i=1;i<=1000;i++)  {
                        System.out.println ("Platz "+i+": "+gameServer.getUserScore(i)) ;
                  }
            }  catch (DatabaseException e)  {    }                              // 9
            catch (RemoteException e)  {   e.printStackTrace () ;   }           // 10
      }

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

Nachdem wieder ein Sicherheitsmanager benötigt wurde (1), findet eigentlich nur ein einziger RMI-Aufruf unter (2) statt. In diesem Aufruf verbindet sich der Client mittels der Methode **Naming.Lookup()** mit der in der URL angegebenen Maschine und fordert das unter dem Namen (hier GameServer) registrierte Objekt von der Registratur an. Das zurürckgelieferte Objekt wird über eine Typumwandlung zu einem **GameServer**-Objekt typisiert. In den Zeilen (3),(4),(8) wird eine vereinfachte Fehlerbehandlung durchgeführt. In den Zeilen (5),(6),(7) wird die **updateScore**-Methode des **GameServer**-Objektes angesprochen und es werden drei Benutzer mit Spielständen eingetragen. Danach wird die aktuelle Tabelle angezeigt.

9.2.3 Start des Servers und des Clients

Die Quelltexte zu den Klassen **GameServer**, **GameServerDaemon** und **GameServer-Client** liegen in einem der Übungsverzeichnisse. Der Server wird während der gesamten Vorlesung auf der entsprechenden Maschine laufen, so daß Sie nur den Clienten starten müssen. Zu diesem Zweck wechseln Sie bitte in das entsprechende Verzeichnis und führen Sie das Skript *startclient* aus. In diesem werden alle relevanten Optionen gesetzt. Sollten Sie sich die Quellen zum Experimentieren in Ihr eigenes Verzeichnis kopieren wollen, müssen Sie nicht nur den javac-Compiler starten, sondern auch noch das Skript entsprechend anpassen. Die Stümpfe und Skelette erzeugte ich mit dem *rmic* Befehl des JDK. Also sollten Sie einen eigenen Server programmieren wollen, müssen Sie auch diese erzeugen...

```
rmic RMIBeispiel.GameServerImpl
```

Vorsicht: In der JDK Version 1.2.x wurde das Sicherheitskonzept gegenüber der Version 1.1.x erweitert, so daß Sie zur Ausführung des Beispiels unter dem JDK 1.1.x keine Kommandozeilen-Optionen angeben bzw. System-Properties setzen müssen. Sehen Sie sich das kleine Skript `startserver` an und Sie werden den Unterschied schnell erkennen.

A Einführung in die Rechnerbenutzung

Wenn Sie im Arbeitsraum für Studierende mit einem Rechner arbeiten wollen, sehen Sie einen Bildschirm, eine Tastatur und eine Maus. Die Rechner sollten eingeschaltet sein.

Merke: Niemals den Rechner ausschalten!

Falls nichts zu sehen ist auf dem Bildschirm, betätigen Sie irgendeine Taste. Jetzt sollten Sie eigentlich etwas sehen. Sie sind mit dem Betriebssystem und einem Fensterverwaltungssystem konfrontiert. Das Betriebssystem bei den Rechnern der Firma SUN ist Solaris, die Fensterverwaltung heißt Common Desktop Environment. Solaris ist das SUN Operating System (OS) 5 mit einigen zusätzlichen Werkzeugen. SUN OS5 basiert auf UNIX. Die Rechner sind bereits für Sie vorbereitet, so daß Sie keine Programme installieren müssen. Im Rahmen dieser Vorlesung haben Sie ohnehin nur mit der Fensterverwaltung, dem Betriebssystem, einem Texteditor und der Entwicklungsumgebung für JAVA zu tun. Da es hier um den Einstieg um Grundkonzepte der Programmierung geht, sollen von diesen ausdrucksstarken und vielseitigen Programmen nur einige wenige Bedienungsmöglichkeiten beachtet werden[29].

A.1 Unix

Sie müssen sich beim Betriebssystem als berechtigter Benutzer anmelden. Bei dem Übungsgruppenleiter oder der Übungsgruppenleiterin erhalten Sie einen Namen mit Schlüsselwort. Unter Ihrem Namen finden Sie nach der Anmeldung stets Ihre Dateien vor – egal, an welchem Rechner Sie sich anmelden. Geben Sie mittels der Tastatur an der Stelle, wo ein Strich (cursor) hinter `login:` ist, Ihren Namen ein und drücken Sie die return-Taste. Geben Sie dann Ihr Schlüsselwort ein. Dies ist nicht zu sehen. Drücken Sie die return-Taste. Jetzt haben Sie Zugang zum Betriebssystem.

Wenn Sie sich zum ersten Mal beim Betriebssystem anmelden sollten Sie in Ihrem eigenen Interesse Ihr Schlüsselwort ändern. Benutzen Sie dazu den Befehl **passwd**, der in Abschnitt A.1.1 erklärt wird. Seien Sie bitte vorsichtig im Umgang mit Ihrem Schlüsselwort, denn Sie sind für alles verantwortlich, was unter Ihrem Namen auf dem Rechner geschieht. Geben Sie daher niemals Ihr Schlüsselwort an andere Leute weiter.

Sie können mithilfe des Betriebssystems unter anderem die folgenden Dinge tun.

- Verzeichnisse und Dateien verwalten,
- Programme (z.B. das Programm für elektronische Post, den Browser für das Internet oder den Texteditor) aufrufen und benutzen,
- Programme (z.B. in JAVA) übersetzen und ausführen lassen,

Die Programme, mit denen Sie es hier zu tun haben, werden in den Abschnitten A.2.1, A.2.2,A.3 und A.4 beschrieben. Alle anderen Programme schreiben Sie selbst.

Sie können alle Programme aus der Menüleiste der Arbeitsumgebung (s. Abb 38) starten. Die einzelnen Piktogramme [30] haben folgende Bedeutung: In der Mitte der Menüleiste befinden sich vier Tasten, mit denen sie auf vier verschiedene Bildschirme wechseln

[29] Das soll Sie nicht hindern, eigene Entdeckungen zu machen!

[30] Wer sich für den Unterschied von Piktogrammen, Icons und Symbolen interessiert kann gerne bei Prof. Morik einen Kurzvortrag über Semiotik anfordern!

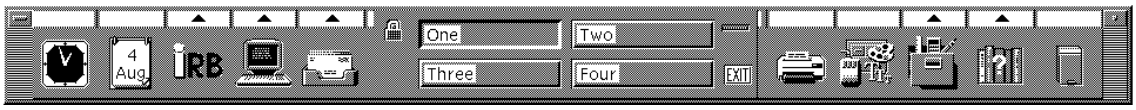


Abbildung 38: Die Menüleiste

können. Links davon befinden sich fünf große Piktogramme, eine Uhr, ein Kalender, das IRB–Symbol, ein Terminal–Piktogramm und ein Email–Piktogramm. Wenn Sie auf den kleinen Pfeil über den Piktogrammen klicken öffnet sich weitere Menüs, die weiter unten beschrieben werden. Direkt neben den Bildschirm–Tasten in der Mitte des Menüs befinden sich drei weitere Piktogramme. Mit dem **Exit**–Piktogramm können Sie die Arbeitsumgebung wieder verlassen, die Anzeige darüber blinkt, wenn gerade ein Menübefehl ausgeführt wird und mit dem kleinen Schloß rechts können Sie Ihren Bildschirm abschließen, so daß sie erst Ihr Schlüsselwort eingeben müßen um weiterarbeiten zu können. Bitte benutzen Sie diesen Befehl nicht, denn so verhindern Sie, daß Ihre Kommilitonen an dem Rechner arbeiten können. Auf der rechten Seite der Menüleiste befinden sich noch fünf weitere Piktogramme, die wichtigste davon ist das Bücher–Piktogramm mit dem Fragezeichen. Mit diesem Menüpunkt können Sie sich weitere Hilfetexte ansehen.

Das IRB–Menü: Im IRB–Menü (s. Abb 39) können Sie die Hilfe der Informatikrechner–Betriebsgruppe in Anspruch nehmen, insbesondere erhalten Sie hier eine Einführung in die Rechnerbenutzung. Bitte benutzen Sie den Menüpunkt „Ändern der login shell“ nur, wenn Sie wirklich wissen was sie tun.

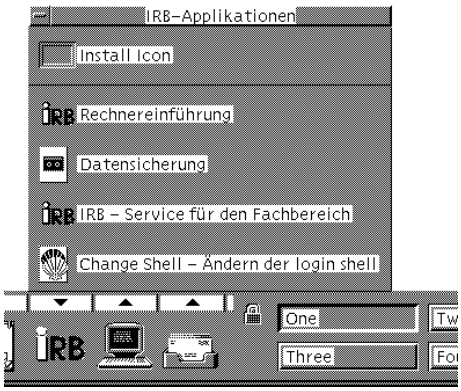


Abbildung 39: Das IRB-Menü

Das Anwendungs–Menü: Dieses Menü ist das wichtigste Menü. Sie können ein Terminal öffnen, in dem Sie Befehle direkt eingeben können, Sie können den Emacs starten, Sie können Informationen zur Vorlesung abrufen und die Newsgruppe zur Vorlesung besuchen.

Das Kommunikations–Menü: Im Kommunikationsmenü könne Sie die Programme für die elektronische Post, das World Wide Web und die Newsgruppen starten. Diese

Abbildung 40: Das Anwendungs-Menü

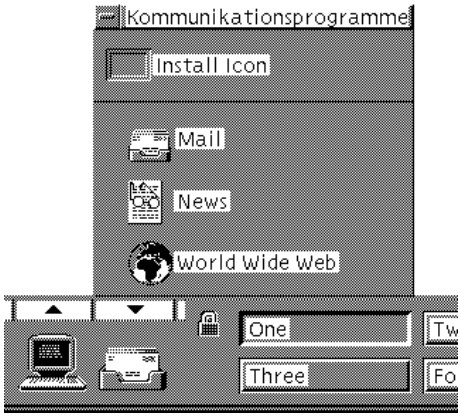


Abbildung 41: Das Kommunikations-Menü

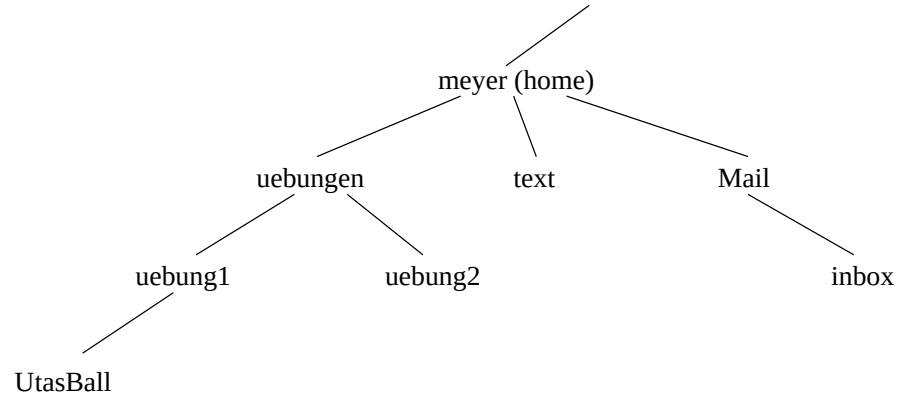


Abbildung 42: Verzeichnisbaum

Programme werden im Abschnitt A.2 erklärt.

Prozesse: Prozesse, die Sie gestartet haben, bekommen Sie angezeigt, wenn Sie **ps** eingeben. Manchmal wundert man sich, was man so alles in Gang gesetzt hat! Und manchmal kann man den Prozeß nicht mehr auf die vorgesehen Art oder mit **C-c** beenden. Dann hilft nur noch das Eintippen von **kill** und der Prozeßnummer, die von **ps** angezeigt wurde.

Wenn Sie wissen wollen, was ein Befehl bedeutet, können Sie das on-line Handbuch aufrufen **man** *Befehl*. Wenn Sie ein Stichwort wissen und den passenden Befehl nicht, so erhalten Sie durch **apropos** *Stichwort* eine Liste mit Befehlen, die etwas mit dem Stichwort zu tun haben. Leider sind Stichworte stets englisch, so daß Sie **delete** und nicht **loeschen** eingeben müssen.

A.1.1 Dateiverwaltung

Dateien werden in Verzeichnissen gespeichert. Verzeichnisse bilden eine Baumstruktur (s. Abbildung 42).

Sie können in einem aktiven Fenster durch die Eingabe von Befehlen mittels der Tastatur oder durch Mausklicks auf Piktogramme der graphischen Menüleiste unten auf dem Bildschirm die Dateien verwalten. In der Hoffnung, daß die grafischen Piktogramme für sich selbst sprechen, gebe ich hier kurz die textuellen Befehle an.

mkdir *Verzeichnisname* Dieser Befehl legt als Unterverzeichnis zu dem, in dem Sie gerade sind (wie das heißt, erfährt man durch **pwd**), ein neues Verzeichnis mit dem Namen *Verzeichnisname* an.

Beispiel: Sie sind gerade im Verzeichnis **meyer**. Sie geben ein **mkdir uebungen** und erhalten so ein Unterverzeichnis mit dem Namen **uebungen**.

cd *Verzeichnisname* Das angegebene Verzeichnis wird zum aktuellen Verzeichnis. Wenn es kein direktes Unterverzeichnis ist, so enthält *Verzeichnisname* den gesamten Pfad vom aktuellen Verzeichnis bis zum Zielverzeichnis.

Beispiel: Sie sind im Verzeichnis **meyer** und möchten zum Verzeichnis **uebung1**. Sie geben ein: **cd uebungen/uebung1**.

Wenn es im Verzeichnisbaum über dem aktuellen Verzeichnis ist, so geht man zunächst mit **cd** (ohne Angabe eines Verzeichnisses!) zum obersten eigenen Verzeichnis (home) und verfährt dann wie gehabt.

Beispiel: Sie sind im Verzeichnis **UtasBall**. Sie wollen in das Verzeichnis **inbox**. Sie geben erst ein **cd** und dann **cd Mail/inbox**.

ls oder **ls** *Verzeichnisname* Die Dateien und Unterverzeichnisse des aktuellen oder mit **Verzeichnisname** bezeichneten Verzeichnisses werden angezeigt. Denselben Effekt erreichen Sie, wenn Sie bei der unteren Menüleiste auf das Verzeichnispiktogramm drücken und in dem dann erscheinenden Fenster auf das Ordnerpiktogramm, unter dem der Verzeichnisname steht, mit der Maus klicken.

cp *Datei Dateikopie* Die Datei *Datei* wird kopiert. Der Name der angelegten Datei ist an zweiter Stelle engegeben (*Dateikopie*). Der Name der kopierten Datei darf ein Unterverzeichnis mitangeben.

Beispiel: Sie sind im Verzeichnis **meyer**, in dem die Datei **text** abgelegt ist. Sie geben ein **cp text uebungen/uebung2/musterloesung**. Jetzt ist **musterloesung** eine Kopie von **text** und im Verzeichnis **uebung2** abgelegt.

mv *Dateiname Verzeichnisname* verschiebt eine Datei in ein Verzeichnis. **mv** *Dateiname1 Dateiname2* benennt die Datei um.

Beispiel: Ihre Lösung der ersten Übungsaufgabe wird als beste Lösung anerkannt und zur Musterlösung für alle gemacht. Sie wollen **UtasBall** in **musterloesung** umbenennen. Sie sind im Verzeichnis **uebung1** und geben ein: **mv UtasBall musterloesung** . Ein Namenskonflikt mit der Musterlösung einer anderen Übungsaufgabe entsteht nicht, denn implizit enthält jeder Dateiname den Pfad vom Heimatverzeichnis zur Datei.

lpr *Dateiname* druckt eine Datei aus. Mit **lpq** können Sie sich anzeigen lassen, welche Dateien in der Warteschlange für den Drucker sind und welche gerade ausgedruckt wird.

rm *Dateiname* löscht die Datei mit dem Namen *Dateiname*.

rmdir *Verzeichnisname* löscht das Verzeichnis *Verzeichnisname*, falls es leer ist.

chmod *Dateirechte Dateiname*. Zu jeder Datei gehören in Unix ein Eigentümer, eine Gruppe und gewisse Rechte, die der Eigentümer (**u**), die Gruppe (**g**) oder alle anderen Benutzer des Rechners (**o**) ausüben können. Die möglichen Rechte sind Schreibrechte (**w**), Leserechte (**r**) und Ausführungsrechte (**x**). Die Rechte einer Datei oder eines Verzeichnisses werden angezeigt, wenn sie den Befehl **ls -l** (s.o.) eingeben.

Beispiel: Der Befehl **chmod g+r text** erlaubt allen Benutzern aus Ihrer Gruppe die Datei **text** zu lesen, **chmod g-r text** entzieht diese Rechte wieder. Mit **chmod go= text** geben Sie an, daß niemand außer Ihnen Rechte an der Datei **text** hat.

passwd Mit diesem Befehl können Sie Ihr Schlüsselwort, mit dem Sie sich beim Betriebssystem anmelden, ändern. Wenn Sie **passwd** eingeben werden Sie zuerst aufgefordert, Ihr altes Schlüsselwort einzugeben. Danach müssen Sie ein neues Schlüsselwort eingeben und zur Sicherheit das neue Schlüsselwort wiederholen. Keine dieser Eingaben erscheint auf dem Bildschirm.

Einige Bemerkungen zum Schlüsselwort: Da Sie für alles, was unter Ihrem Namen am Rechner geschieht, verantwortlich sind lohnt es sich, mit dem Schlüsselwort besonders sorgfältig zu sein. Benutzen Sie keine Schlüsselwörter die leicht zu erraten sind, wie etwa Ihren Vornamen, Ihr Haustier oder Namen aus Büchern oder Filmen. Benutzen Sie stattdessen ein Wort, das nicht im Lexikon vorkommt und das sowohl Großbuchstaben, Kleinbuchstaben als auch Sonderzeichen (z.B. Ziffern) enthält und benutzen Sie ein Schlüsselwort, das mehr als sechs Zeichen lang ist. Andererseits sollte Ihr Schlüsselwort auch nicht zu kompliziert sein, damit Sie es sich merken können ohne es aufschreiben zu müssen. Ein guter Trick ist, sich einen Satz auszudenken und die Anfangsbuchstaben als Schlüsselwort zu nehmen. So wird z.B. aus „Vier Studenten gehen in die Mensa“das Schlüsselwort „4SgidM.“.

& Dieses Zeichen ist kein Kommando an sich, sondern es weist Unix an, nach dem Start eines Kommandos nicht auf die Beendigung des Programms zu warten, sondern sofort weitere Kommandos entgegenzunehmen. Das Kommando wird dann parallel zu Ihrer restlichen Arbeit ausgeführt. Das **&** muß immer hinter dem zugehörigen Kommando stehen. So ruft z.B. **emacs&** den Texteditor auf und erwartet weitere Kommandos.

A.2 Arbeiten im Netz

Alle Rechner des Fachbereichs Informatik sind in einem Netzwerk miteinander verbunden, das wiederum mit dem Internet verbunden ist. Somit können Sie alle Resourcen benutzen die das Internet bietet, darunter die elektronische Post (*email*), das World Wide Web (*WWW*) und die Newsgroups. Für die Email ist für Sie das Programm **dtmail** installiert und mit dem **Netscape Communicator** können Sie das WWW und die Newsgroups benutzen. **dtmail** starten Sie durch einen Klick auf das Email–Piktogramm in der Menüleiste, **Netscape** starten Sie durch einen Klick auf das WWW- oder NewsPiktogramm in der Menüleiste oder durch das Kommando **netscape&** starten. Der **Netscape Communicator** besteht eigentlich aus mehreren Programmen, darunter dem **Navigator** für das WWW, **Collabra Discussions** für News und **Messenger Mailbox**, womit Sie ebenfalls Emails bearbeiten können. Die Programme werden über den Menüpunkt *Communicator* (Abb. 43) oder die Piktogrammleiste unten links (Abb. 44) gestartet; diese vier Piktogramme entsprechen von links nach rechts dem**Navigator**, der **Messenger Mailbox**, **Collabra Discussions** und einem Editor für WWW–Seiten, der hier nicht besprochen wird Das wichtigste Kommando gleich vorweg: Ganz rechts in der Menüleiste befindet sich der Menüpunkt **Help**. Hier erhalten Sie Informationen zu allen Befehlen von Netscape, auf die hier nicht eingegangen wird (und natürlich auch weitere Informationen zu den Befehlen, auf die hier eingegangen wird).

Bevor Sie loslegen aber einige Warnungen:

- Beachten Sie, daß die Kapazität der Netzwerkverbindung begrenzt ist. Da Ihre Rechner keine eigenen Festplatten haben, wird auch jeder Zugriff auf eine Datei über das

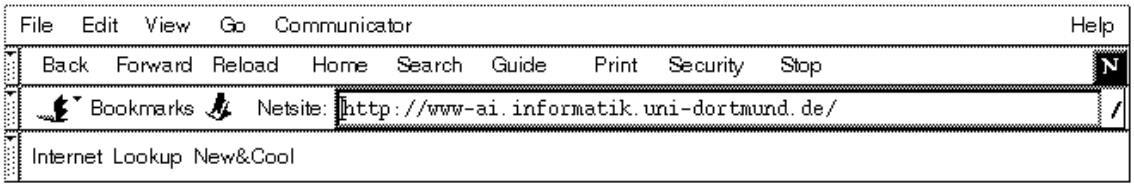


Abbildung 43: Menü des Netscape Navigator

Abbildung 44: Netscapes Piktogramme

Netzwerk erledigt. Wundern Sie sich also nicht, daß Ihr Java–Compiler so langsam ist wenn sich Ihr Nachbar gerade ein Video aus dem Internet lädt. Fordern Sie ihn auf, das zu lassen.

- Denken Sie bitte daran, daß die Rechner hauptsächlich dem Studium dienen. Selbstverständlich hat niemand etwas dagegen, wenn Sie auch einmal im Internet herumstöbern, aber bitte überlassen Sie Ihren Kommilitonen den Rechner, wenn diese ihre Übungsaufgaben lösen wollen.
- Für den richtigen Umgang im Internet gibt es einige ungeschriebene Gesetze, die sogenannte *Netiquette*. So gilt es zum Beispiel als höflich, beim Schreiben von Emails oder in Newsgruppen seinen richtigen Namen anzugeben. Insbesondere wenn Sie in Newsgruppen schreiben wollen sollten Sie sich zuerst die Informationen in der Gruppe **de.newusers.infos** ansehen.

A.2.1 Elektronische Post

Die elektronische Post ist sicherlich das wichtigste Handwerkszeug im Netz. Zunächst die Theorie: Eine Email-Adresse besteht aus dem Benutzernamen und dem Namen des Rechners, getrennt von dem Zeichen @ („at “), also etwa **gvpr000@diavolo.informatik.uni-dortmund.de**. Ist der Rechnername nicht vollständig wird automatisch zum vollen Namen ergänzt, sie können also auch nur **gvpr000@diavolo** oder **gvpr000** eingeben. Voraussetzung dafür ist allerdings, daß im ersten Fall die Adresse Ihres Rechners auch auf **.informatik.uni-dortmund.de** endet (Man spricht in diesem Fall davon, daß die Rechner der gleichen *Domain* angehören) bzw. daß im zweiten Fall der Benutzer **gvpr000** am gleichen Rechner wie Sie existiert.

Ein weiterer wichtiger Begriff ist das *subject* einer Mail (deutsch: der Betreff). Hier gibt man kurz an, worum es in der Mail geht; der Empfänger sieht in seinem Email–Programm Ihren Absender und die *subject*–Zeile und kann seine Emails entsprechend sortieren.

Nun die Praxis: Nach dem Start von **dtmail** erscheint ein Fenster,in dessen oberer Hälfte man eine Liste seiner Emails sieht (s. Abb. 45), klickt man auf eine Mail, so erscheint der Text in der unteren Hälfte des Fensters. Will man eine Mail schreiben, wählt man im Menüpunkt **compose** den Unterpunkt **New Message**, will man auf eine Mail, die man

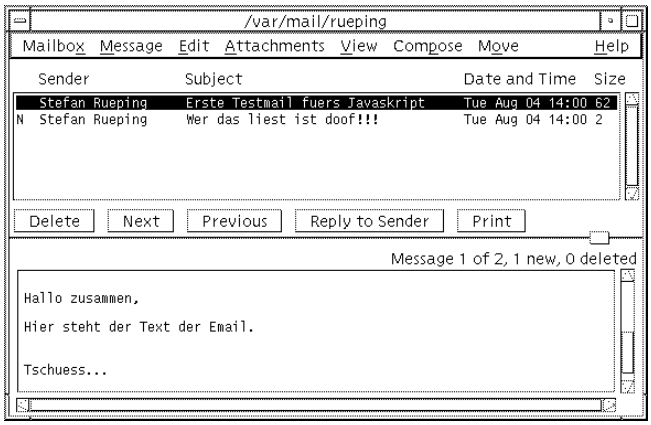


Abbildung 45: Das Hauptfenster von dtmail

bekommen hat antworten, so klickt man diese Mail an und wählt Menüpunkt **compose** den Unterpunkt **Reply to Sender**.

In beiden Fällen erscheint ein neues Fenster (s. Abb. 46), in dessen oberer Hälfte man den Empfänger einträgt (**To**), den Betreff (**Subject**) und optional wer eine Kopie dieser Email erhalten soll (**CC**). Im mittleren Teil des Fensters gibt man den Text ein und im unteren Teil erscheinen die Attachments der Mail, das sind Dateien, die man mit der Mail mitschicken möchte. Diese fügt man mit dem Menüpunkt **Attachments** hinzu. Klickt man auf den Knopf **Send** ganz unten links wird die Email abgesandt, mit dem Knopf **Close** beendet man das Fenster ohne die Email abzusenden.

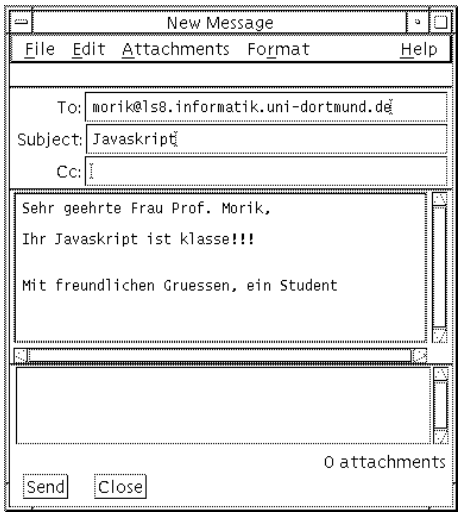


Abbildung 46: Das Emailfenster von dtmail

A.2.2 World Wide Web

Das World Wide Web (WWW) ist die „graphische Seite“ des Internet, hier kann man Texte, Bilder, Musik und vieles mehr abrufen. Dabei hat jedes Dokument eine eindeutige Adresse, die URL (Uniform Resource Locator). Diese hat etwa die Form **http://www-ai.informatik.uni-dortmund.de/index.html**, sie besteht aus dem verwendeten Protokoll (standardmäßig **http**, HyperText Transport Protocol), der Adresse des Rechners (**www-ai.informatik.uni-dortmund.de**, der Webserver des Lehrstuhls 8) und der Datei, die man ansehen möchte (**index.html**). Für lokale Dateien hat die URL die Form **file:***Voller Dateiname.*

Das Navigieren im WWW ist einfach: Innerhalb eines Dokumentes sind Verweise auf andere Dokumente (*links*) unterstrichen oder farblich hervorgehoben. Klickt man darauf, gelangt man zu diesem Dokument. Die Dokumente sind durch diese Verweise weltweit vernetzt.

Um zu einem Dokument zu gelangen, gibt es drei Wege:

1. Man gelangt über Links von einem bekannten Dokument dorthin.
2. Man kennt die URL und gibt diese in der Zeile unterhalb des Menüs oder im Menüpunkt **File / Open Page** ein.
3. Man findet die Adresse über eine Suchmaschine. Dies ist eine Datenbank, die die Adressen vieler WWW–Dokumente gespeichert hat und in der man nach Stichworten suchen kann. Bekannte Suchmaschinen sind z.B. Yahoo (**www.yahoo.com**) oder HotBot (**www.hotbot.com**).

A.2.3 News

Eine Newsgroup ist die Computer–Version eines schwarzen Bretts. Man kann Nachrichten aufhängen oder auf andere Nachrichten antworten und so mit anderen Benutzern in Kontakt treten. Eine Newsgroup ist stets einem bestimmten Thema gewidmet, so ist die bereits oben erwähnte Gruppe **de.newusers.infos** zur Hilfe für neue Benutzer von Newsgruppen gedacht und die Gruppe **unido.informatik.lehre.gvpr** dient zur Diskussion über die Vorlesung und die Übungen. Insbesondere können hier Fragen zum Umgang mit dem Rechner oder zum Skript gestellt werden.

Um mit Newsgruppen zu arbeiten gibt es in Netscape **Collabra Discussions**. Hier erscheint ein Fenster (Abb. 47), in dem unter anderem ein Piktogramm **fbi-news** erscheinen sollte. Dies ist der Newsserver des Fachbereichs Informatik. Mit einem Klick der rechten Maustaste auf das Piktogramm sehen sie ein kleines Menü, aus dem Sie den Punkt **Add Discussion Group** wählen. Nun erscheint ein weiteres Fenster, das alle vorhandenen Newsgruppen anzeigt. Klickt man auf den kleinen Punkt rechts neben dem Namen, so erscheint ein kleines Häkchen, dieses zeigt an daß man die Diskussion in dieser Gruppe verfolgen möchte. Wenn man dieses Fenster mit **OK** verläßt erscheint der Name der Newsgruppe unterhalt des Piktogramms **fbi-news**. Wenn man auf dieses Piktogramm doppelklickt erscheint nun das Fenster des Netscape–Emailprogramms **Messenger Mailbox**, das auch zum Lesen und Schreiben der von Newsgruppen benutzt wird.

Im oberen Teil des Fensters von **Messenger Mailbox** sieht man eine Liste der Nachrichten in der Newsgruppe , klickt man auf eine Nachricht, so erscheint der Text in der unteren Hälfte des Fensters. Will man einen neuen Nachricht schreiben, benutzt man das

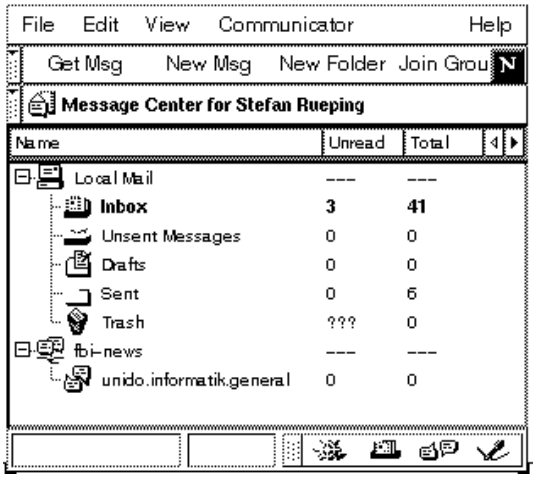


Abbildung 47: Netscape News

Piktogramm **New Msg**, will man auf einen alte Nachricht antworten kann man auch **Reply** wählen, man muß dann das Subject nicht nochmal eingeben und die neue Nachricht wird in der Newsgruppe automatisch zur alten sortiert.

In dem neuen Fenster trägt man im oberen Teil das Subject der Nachricht ein und im unteren Teil den Text der Nachricht. Mit einem Klick auf **Send Now** wird die Nachricht dann abgeschickt. Auf dieselbe Art werden in Netscape übrigens auch Emails bearbeitet.

A.3 Emacs

Der Texteditor Emacs ist der meistbenutzte Editor unter Unix. Der Grund dafür ist seine große Flexibilität, man kann ihn an beinahe jede Aufgabe anpassen und sogar selbst Erweiterungen programmieren. So gibt es zum Beispiel für viele Programmiersprachen, unter anderem für Java, einen eigenen Modus, der die Programmierarbeit unterstützt.

Der Nachteil dabei ist allerdings, daß der Emacs auch ein sehr komplexes Programm ist. Selbst ausgewachsene Emacs–Spezialisten entdecken immer noch neue Befehle oder Tastenkombinationen, von denen sie noch nie zuvor gehört haben. Dieser Abschnitt soll Ihnen einen ersten Einblick in die wunderbare Welt des Emacs geben.

Der Emacs wird entweder über die Menüleiste gestartet oder mit dem Befehl **emacs** bzw. **emacs** *Dateiname*. Enthält *Dateiname* eine Extension (z.B. *.java* oder *.tex*) wechselt der Emacs automatisch in den entsprechenden Modus. Dies bedeutet, daß sich der Emacs zum Beispiel auf die Syntax der Sprache einstellt, d.h. der Text wird automatisch eingerückt und im Menü erscheinen sprachspezifische Menüpunkte. Der Emacs ist also kein reiner Texteditor, sondern er orientiert sich an der Struktur der verwendeten Sprache.

Ein wichtiges Konzept des Emacs sind die Puffer. Ein Puffer ist der Teil, der den Text aus einer Datei enthält. Man kann mehrere Puffer gleichzeitig öffnen, man kann also mehrere Dateien (oder die gleiche Datei mehrmals) editieren. Eine Liste aller offenen Puffer erscheint unter dem Menüpunkt **Buffers**.

Wichtige Informationen enthält die Statuszeile (die schwarze Zeile unten, Abb. 48). Hier erscheint unter anderem der Name des Puffers (**EinTest.txt**), der Name des aktuellen

Modus (**Text**), die Zeile, in der sich gerade der Cursor befindet (**L8**) und eine Angabe darüber, wo im Text sich der im Fenster sichtbare Teil befindet (in Prozent des gesamten Textes oder **ALL**, wenn der ganze Text zu sehen ist). Ganz links erscheinen zwei Sterne, wenn der Puffer seit dem letzten Speichern verändert wurde. Unter der Statuszeile befindet sich ein Minipuffer, in dem Meldungen angezeigt oder Eingaben erwartet werden.

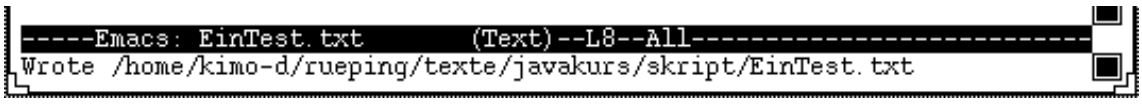


Abbildung 48: Die Emacs–Statuszeile

A.3.1 Wichtige Befehle und Tastenkombinationen

Hier sind die wichtigsten Tastenkombinationen zur Bedienung des Emacs. Die Schreibweise **C-<Taste>** bedeutet, daß man die Control–Taste gedrückt hält während man die andere Taste drückt. **M-<taste>** bezeichnet entsprechend die META–Taste (oder EDIT oder ALT–Taste, je nach Rechner).

C-x C-f dient zum Öffnen einer Datei. In der untersten Zeile des Fensters wird man nach dem Dateinamen gefragt. Der Emacs komplettiert übrigens unvollständige Dateinamen selbstständig, wenn man die Tab–Taste drückt, ist der Name nicht eindeutig erhält man durch einen weiteren Druck auf die Tab–Taste eine Liste der möglichen Endungen.

C-x C-s speichert den Puffer wieder in die Datei.

C-x C-w speichert den Puffer in eine neue Datei. Der Cursor springt in den Minipuffer, dort muß man den neuen Dateinamen angeben.

C-x C-c beendet den Emacs.

C-g dient dazu, Eingaben oder Berechnungen des Emacs abzubrechen.

C-_ macht die letzte Änderung rückgängig.

C-<Leerzeichen> C-w Diese Tastenkombination dient zum Ausschneiden eines Textstücks aus dem Text. Man positioniert den Cursor auf den Anfang des Textstücks und gibt **C-<Leerzeichen>** ein, dann bewegt man den Cursor auf das Ende des Textstücks und gibt **C-w** ein. Das Textstück wird aus dem Puffer gelöscht und intern im sogenannten **Kill-Ring** gespeichert. Der **Kill-Ring** ist ein interner Puffer des Emacs, in dem alle ausgeschnittenen Textstücke gespeichert werden.

C-<Leerzeichen> ESC-w Diese Tastenkombination dient zum Kopieren eines Textstücks in den **Kill-Ring**, sie wird genau wie das Ausschneiden benutzt.

C-y und **ESC y**. Die Tastenkombination **C-y** kopiert den ersten Eintrag im Kill-Ring an die Stelle des Cursors in den Puffer. Gibt man direkt im Anschluß daran **ESC y** ein, so wird der erste Eintrag durch den zweiten Eintrag im Kill-Ring ersetzt,

weitere Eingaben von **ESC y** ersetzen den Eintrag wiederum durch den jeweils Vorhergehenden, der letzte Eintrag im Kill-Ring wird dann wieder durch den ersten Eintrag ersetzt (und deshalb heißt der Kill-Ring auch Ring und nicht Kill-Liste!).

ESC d löscht die Zeichen vom Cursor bis zum Ende des Wortes und fügt sie in den Kill-Ring ein.

C-k löscht die Zeichen vom Cursor bis zum Ende der Zeile und fügt sie in den Kill-Ring ein.

C-s dient zur Suche nach einem bestimmten Wort im Text. Man gibt in der untersten Bildschirmzeile das gesuchte Wort ein und während der Eingabe springt der Emacs bereits zur ersten Fundstelle des Wortes. Um nach weiteren Vorkommen des Wortes zu suchen betätigt man **C-s** nochmal. Die Suche beginnt an der Cursorposition und endet am Ende des Dokuments.

C-x n mit $n = 1,2,3$ zeigt einen bis drei Puffer gleichzeitig im Emacs–Fenster an.

ESC x bringt einen in die Kommandozeile. Hier kann man weitere Emacs–Kommandos eingeben.

ESC x goto-line läßt den Cursor in eine bestimmte Zeile springen, man wird nach der Zeilennummer gefragt.

ESC x query-replace Mit diesem Befehl kann man Worte im Text suchen und ersetzen. Man gibt den Suchbegriff und den zu ersetzenden Begriff ein und der Emacs stoppt bei jedem Vorkommen des Wortes im Text und fragt, ob das Wort ersetzt werden soll. **y** eingeben führt zum Ersetzen, **n** nicht.

ESC x text-mode Im Text-Modus wird der Inhalt der Datei als natürlichsprachlicher Text gesetzt.

A.3.2 Java–Modus

Der Emacs bietet einige Hilfmittel, um die Programmierung mit Java zu vereinfachen. So wird der Quelltext zum Beispiel automatisch eingerückt und die Java–Schlüsselwörter werden farblich markiert, was die Lesbarkeit deutlich verbessert. Schließt man eine Klammer, so wird die zugehörige öffnende Klammer kurz angezeigt; man kann so gut überprüfen, ob man eine Klammer vergessen hat und damit eine häufige Fehlerquelle ausschließen.

Im Menü erscheint ein neuer Menüpunkt **Java**, mit dem man z.B. den Cursor im Text besser fortbewegen kann. Probieren Sie einfach aus was passiert. Um ein Programm zu kompilieren können Sie im Menü **Tools** den Menüpunk **Compile** auswählen. Sie müßen dann im Minipuffer unten den Befehl **javac** *Dateiname* eingeben, bei allen weiteren Kompiliervorgängen merkt sich der Emacs diesen Befehl. Das Fenster mit dem Programmtext teilt sich und es erscheint die Ausgabe des Compilers. Wenn sich im Laufe des Kompilierens ein Fehler ergibt können Sie durch einen Klick mit der mittleren Maustaste auf die Fehlermeldung direkt zur entsprechenden Stelle im Quelltext springen.

A.3.3 Weitere Hilfe zum Emacs

Weitere Hilfe zum Emacs findet man im Menüpunkt **Help**. Hier werden unter anderem alle Tastenkombinationen und Modi erklärt. Zum Beginn ist das Emacs Tutorial empfehlenswert, hier werden noch einmal die wichtigsten Befehle angesprochen, die man auch direkt ausprobieren kann.

A.4 Java Entwicklungsumgebung

Die Java Entwicklungsumgebung (Java Development Kid, JDK) ist eine Sammlung von Programmen zur Programmierung mit Java. Die wichtigsten darunter sind **javac**, **java**, **appletviewer**, **jar** und **jdb**.

Dies sind noch nicht alle Programme aus der Java Entwicklungumgebung und es werden auch nicht alle Optionen für diese Programme angegeben. Weitere Informationen erhalten Sie in der Online–Dokumentation.

A.4.1 Der Java–Compiler

Der Java Compiler **javac** compiliert Java Quellcode in Java Bytecode, er wird mit dem Befehl **javac** *[Optionen] Dateien*[31] aufgerufen. Man kann **javac** mit beliebig vielen Quelldateien aufrufen, die alle auf **.java** enden müssen und er erzeugt für jede Klasse (!) in den Quelldateien eine **.class**–Datei. Der Compiler verlangt, daß in einer Quelldatei nur eine **public**–deklarierte Klasse existiert und daß diese denselben Namen wie die Quelldatei hat. So würden beim Compilieren folgender Datei (mit dem Namen **HalloWelt.java**) ...

```
public class HalloWelt {
        public static void main (String argv[]) {
                Begruessung B = new Begruessung ();
                System.out.println (B.deutsch ());
        }
}

class Begruessung {
        public String deutsch () { return "Hallo Welt!"; }
        public String english () { return "Hello World!"; }
}
```

...die Dateien **HalloWelt.class** und **Begruessung.class** erzeugt. Wollte man **Begruessung** auch als **public** deklarieren, müßte man eine eigene Datei **Begruessung.java** anlegen.

Normalerweise schreibt der Java Compiler alle **class**–Dateien in dasselbe Verzeichnis wie die **java**–Dateien. Will man, daß die Dateien in ein anderes Verzeichnis geschrieben werden muß man dieses mit der Option **-d** *Verzeichnis* angeben. Mit der Option **-nowrite** liest der Compiler die Datei wie gewohnt ein, schreibt aber keine Ausgabedatei; dies ist nützlich wenn man die Syntax der Datei überprüfen will, ohne sie tatsächlich komplett zu übersetzen. Mit der Option **-verbose** gibt der Compiler während seiner Arbeit zusätzliche

[31] Die eckigen Klammern bedeuten, daß dieser Teil nicht unbedingt angegeben werden muß.

Meldungen aus und die Option **-g** dient dazu im Bytecode Informationen für den Java–Debugger (s. Abschnitt A.4.4) anzulegen.

A.4.2 Der Java–Interpreter

Der Java Interpreter wird mit dem Kommando **java** *[Interpreter Optionen] Klassenname [Programm Argumente]* gestartet. Der Java Interpreter lädt die dem Klassennamen entsprechende **.class**–Datei und startet die darin enthaltene Methode **public static void main(String argv[])** , wobei die Programm–Argumente im Feld **argv**[] übergeben werden.

Die wichtigsten Interpreter–Optionen sind **-help**, wodurch ein kurzer Hilfetext ausgegeben wird und **-verbose**, wodurch der Java Compiler beim Laden jeder neuen Klasse eine Meldung ausgibt.

A.4.3 Das Java Archivwerkzeug

Das Java–Archivierungsprogramm **jar** dient dazu, mehrere Dateien die zu einem Programm gehören zu einem komprimierten Archiv zusammenzupacken. **jar** schreibt alle Dateien, die man angibt zusammen mit einer Hilfsdatei (genannt **manifest**–Datei) in eine neue Archivdatei und komprimiert diese. **jar** wird wie folgt benutzt:

jar cf *Archivdatei Dateien* legt ein neues Archiv mit dem Namen *Archivdatei* an und speichert die angegeben Dateien darin. Man kann auch ganze Verzeichnisse angeben.

jar tf *Archivdatei* listet den Inhalt des Archives am Bildschirm auf.

jar xf *Archivdatei [Dateien]* holt die angegeben Dateien oder Verzeichnisse aus dem Archiv und schreibt sie in das aktuelle Verzeichnis. Werden keine Datei- oder Verzeichnisnamen angegeben, so wird das ganze Archiv extrahiert.

A.4.4 Der Java–Debugger

Der Java–Debugger **jdb** ist eine Möglichkeit Javaprogramme auszutesten - allerdings eine sehr unkomfortable. Um den Java–Debugger zu benutzen sollte man den Java–Compiler mit der Option **-g** aufrufen, damit dieser Informationen für den Debugger im Bytecode anlegt. Der Debugger wird mit dem Kommando **jdb** *Klassendatei* aufgerufen, wobei die Endung **.class** weggelassen wird. Der Debugger startet und man kann eines der folgenden Kommandos eingeben:

help zeigt eine Liste der verfügbaren Kommandos mit einer kurzen Erklärung.

run *[Klasse] [Argumente]* startet die **main()**–Methode der angegebenen Klasse mit den Argumenten. Wird keine Klasse angegeben wird die Klasse, die beim Start von **jdb** angegeben wurde gestartet.

exit beendet **jdb**.

stop definiert einen Stoppunkt. Starte man das Programm wird die Ausführung am Stoppunkt unterbrochen, so daß man sich zum Beispiel die Variablenbelegung anschauen kann. Den **stop**–Befehl gibt es in zwei Varianten: **stop in** *Klasse.Methode* stoppt

beim Anfang der Methode, **stop at** *Klasse:Zeile* stoppt beim Erreichen der angegebenen Zeile in der Quelldatei. Der Befehl **stop** ohne weitere Parameter gibt eine Liste aller definierten Stoppunkte aus.

clear *Stoppunkt* löscht den Stoppunkt wieder, der Stoppunkt wird wie beim Setzen in der Form *Klasse.Methode* oder *Klasse:Zeile* angegeben.

step führt die nächste Programmzeile aus und stoppt dann wieder.

cont setzt die Ausführung des Programmes nach einem Stoppunkt fort.

print *Id* gibt das Objekt, die Klasse, das Feld oder die lokale Variable mit der Bezeichnung *Id* aus.

dump *Id* gibt die Werte alle Felder des Objekts aus. Gibt man den Namen einer Klasse an, so werden alle statischen Methoden und Variablen angezeigt, sowie die Oberklasse.

locals zeigt eine Liste der lokalen Variablen an.

methods *Klasse* zeigt eine Liste aller Methoden der Klasse an.

A.4.5 Der Appletviewer

Der Appletviewer dient zum Starten von Applets, das sind in HTML–Dokumente eingebettete Javaprogramme (s. Abschnitt A.5). Der Appletviewer wird mit dem Befehl **appletviewer** [**-J***Javaoption*] **datei**|**url** gestartet und er führt alle in der Datei enthaltenen Applets aus. Dabei werden die angegebenen Optionen dem Java Interpreter übergeben.

Man kann Applets auch in einem Java–fähigen WWW–Browser, etwa dem Netscape Navigator, ausführen, wenn man das entsprechende HTML–Dokument lädt.

A.5 Applets

Ein Applet ist ein kleines Java–Programm, das von einem WWW-Browser aus dem Netzwerk geladen werden und von diesem ausgeführt werden kann. Alternativ kann ein Applet auch in einem Appletviewer (s. A.4.5) gestartet werden. Applets unterscheiden sich in mehreren Punkten von einem normalen Java–Programm, hauptsächlich aus zwei Gründen: Zum einen verlangt die Einbindung in einen Browser eine andere Art der Programmkontrolle, zum anderen sollte man Programmcode, den man von einer unbekannten Rechner bezogen hat grundsätzlich als nicht vertrauenswürdig ansehen.

Um dem Sicherheitsaspekt Rechnung zu tragen ist ein Applet einigen Einschränkungen unterworfen. So kann ein Applet zum Beispiel nicht auf das lokale Dateisystem zugreifen, andere Netzwerkverbindungen als die zum Ursprungsrechner aufbauen oder auf bestimmte Systemresourcen zugreifen. Weiter wird der geladene Java–Bytecode einer Sicherheitüberprüfung unterworfen, die sicherstellen soll, daß der Code den Javainterpreter nicht beschädigen kann.

Aus der Sicht des Programierers unterscheidet sich ein Applet von einem Programm vor allem dadurch, daß es eine Unterklasse der Klasse **Applet** ist und als solche einige Standardmethoden implementiert. Das Applet wird dann nicht über eine **main()**–Methode gestartet sondern die Standardmethoden werden vom Browser aufgerufen. Das Applet kontrolliert also den Thread, in dem es ausgeführt wird nicht sondern es reagiert nur auf

Befehle des Browsers. Um den Browser nicht zu blockieren dürfen die Methoden des Applets auch keine lange Berechnungen ausführen, sondern dafür muß ein eigener Thread gestartet werden.

Hier ist eine Liste der absolut wichtigsten Methoden der Klasse **Applet**, eine vollständige Liste ist in der Java–Dokumentation zu finden.

init() Diese Methode wird aufgerufen, wenn das Applet zum ersten Mal in den Browser geladen wird. Die Initialisierung des Applets sollte hier und nicht in einem Konstruktor geschehen, da der Browser keine Argumente an einen Konstruktor übergibt und erst hier sichergestellt werden kann, daß der Browser dem Applet alle nötigen Informationen zur Initialisierung, wie z.B. Parameter, zur Verfügung stellen kann.

destroy() Diese Methode wird aufgerufen, wenn das Applet aus dem Browser gelöscht werden soll. Hier sollten alle Resourcen, die das Applet hält freigegeben werden.

paint() Diese Methode wird vom Browser aufgerufen, wenn das Applet sich neu auf dem Bildschirm darstellen soll. Als Argument bekommt diese Methode ein **Graphics**–Objekt (definiert in **java.awt.Graphics**), das zum Darstellen benutzt wird.

getParameter() Diese Methode liefert den Wert eines Parameters, der in der zum Applet gehörigen HTML-Datei definiert ist. Als Argument erwartet die Methode den Namen des Parameters als String und sie liefert den Wert des Parameters ebenfalls als String zurück.

Mit diesen Methoden kann man bereits ein einfaches Applet erstellen:

```
import java.applet.Applet;
import java.awt.Graphics;

public class HalloApplet extends Applet  {
      String derText;
      public void init ()  {
            derText = "Hallo "+ getParameter ("UserName") + "!";
      } ;
      public void paint (Graphics g)  {
            g.drawString (derText, 50, 25);
      }
}
```

Wenn ein Browser ein Applet darstellen soll muß es in einem WWW–Dokument eingebunden werden. Dazu dienen die HTML-Befehle <**APPLET**> und <**PARAM**>. Ein Dokument, das das Hallo–Applet aufruft sieht z.B. so aus:

```
<HTML>
   <HEAD>
      <TITLE> Das Hallo-Applet </TITLE>
   </HEAD>
   <BODY>
      Hier ist das Applet:
     <APPLET CODE="HalloApplet.class" WIDTH=250 HEIGHT=30>
        <PARAM name="UserName" value="kleiner Programmierer">
```

```
    </APPLET>
  </BODY>
</HTML>
```

Dieses Dokument definiert eine HTMl–Seite mit der Kopfzeile „Das Hallo-Applet“, im Rumpf des Dokuments wird das Hallo–Applet aufgerufen. Dazu dient der **APPLET=**–Befehl, in dem der Klassenname des Applets und die Höhe und Breite des Applets angegeben werden. Mit dem **PARAM=**–Befehl werden die Namen und die Werte der Parameter des Applets definiert. Man kann zusätzlich mit dem Befehl **ARCHIV=** ein ganzes **jar**–Archiv angeben, in dem sich das Applet befindet. Das hat den Vorteil, daß direkt das ganze Archiv geladen wird, damit man weitere nötige Dateien nicht später nachladen muß. Betrachtet man das HTML–Dokument mit Netscape sieht man Abbildung 49.

Abbildung 49: Das Hallo-Applet

A.6 Weitere Informationen

Zu den angesprochenen Themen gibt es eine Unmenge an Büchern für Einsteiger und Fortgeschrittene, ein Besuch in der Bibliothek ist sehr zu empfehlen. Auch im Internet gibt es viele Informationen zu den angesprochenen Programmen und dem Umgang mit Rechnern. Zu empfehlen ist die Rechnereinführung der Informatikrechner–Betriebsgruppe (IRB), zu erreichen im IRB–Menü oder über **http://irb-www.informatik.uni-dortmund.de/** im Untermenü **Anleitungen**. Ein weiterer Startpunkt für die Suche im WWW ist die Startseite des Fachbereichs Informatik (**http://www.informatik.uni-dortmund.de/**).

Das Hochschulrechenzentrum bietet eine Broschüre zum Thema Unix mit dem Titel „UNIX - Eine Einführung“ an. Sie umfasst 139 Seiten und ist für 6,80 DM beim Kundenbüro des HRZ (GB V / R. 108) zu kaufen.

Literatur

[Aho und Ullman, 1996] Aho, A. V. und Ullman, J. D. (1996). *Informatik - Datenstrukturen und Konzepte der Abstraktion.* Thomson Publishing.

[Bishop, 1998] Bishop, J. (1998). *JAVA Gently – Programming Principles Explained.* Addison-Wesley.

[Dißmann und Doberkat, 1998] Dißmann, S. und Doberkat, E.-E. (1998). *Einführung in die objektorientierte Programmierung mit JAVA.* Oldenbourg?

[Flanagan, 1998] Flanagan, D. (1998). *Java in a Nutshell,deutsche Ausgabe.* O'Reilly.

[Goos, 1996] Goos, G. (1996). *Vorlesungen über Informatik 2: Objektorientiertes Programmieren und Algorithmen.* Springer.

[Harold, 1997] Harold, E. R. (1997). *Java Network Programming.* O'Reilly, 1. Auflage.

[Kotovsky et al., 1985] Kotovsky, K., Hayes, J., und Simon, H. (1985). Why are some problems hard? Evidence from Tower of Hanoi. *Cognitive Psychology*, 17:248 – 294.

[Oestereich, 1997] Oestereich, B. (1997). *Objektorientierte Softwareentwicklung mit der Unified Modeling Language.* Oldenbourg.

[Strawson, 1959] Strawson, P. F. (1959). *Einzelding und logisches Subjekt (Individuals).* Reclam.

[Vornberger und Thiesing, 1998] Vornberger, O. und Thiesing, F. (1998). Vorlesung: Algorithmen. WS 1997/98.

Index