Informatik III

Objektorientierte Programmierung

Hochschule Fulda
Fachbereich Elektrotechnik und Informationstechnik
Prof. Dr. Timm Grams

Datei: Inform_3 (.doc oder .pdf)
6. Juli 2009
Erstausgabe (PrograKo): 05.12.96 (Java-Version: 19.02.02)

Beschreibung der Lehrveranstaltung

Die Web-Seite zur Lehrveranstaltung ist

`http://www.hs-fulda.de/~grams/informat.htm`.

Gegenstand der Lehrveranstaltung ist die objektorientierte Software-Entwicklung.

Zu Grunde gelegt wird ein vierstufiges Vorgehensmodell der Software-Entwicklung. Es ist aus den bekannten Vorgehensmodellen abgeleitet und dem Umfang der Übungsaufgaben angepasst. Die Einbindung des Vorgehensmodells in den Übungsbetrieb steht im Dienste des *aktiven Studierens*: Rollenspiele, Exploration, Erstellen eines Arbeitsplans, Durchführung, Überprüfung der Ergebnisse. Die genaueren Festlegungen zum Übungsbetrieb und zur Testierung erfolgen in der zweiten Lektion des Kurses, nachdem das Vorgehensmodell eingeführt worden ist.

Im ersten Teil des Kurses stehen die Elemente der heutigen Software-Technik im Vordergrund. Anhand von Beispielaufgaben wird die Programmiersprache Java erkundet. Im mittleren Teil des Kurses geht es um die zentralen Merkmale der objektorientierten Programmierung. Die Beispiele werden anspruchsvoller. Im dritten Teil gewinnen dann die Anwendungen aus dem Bereich der praktischen Informatik und des Algorithmenentwurfs die Oberhand. Jetzt zahlt sich auch die mehrstufige Bearbeitung der Aufgaben richtig aus.

Die Realisierung der Programme geschieht in der Sprache *Java*. Für die Programmentwicklung steht zunächst nur ein einfacher Text-Editor zur Verfügung. Alle Programme (Editor, Übersetzer, Interpreter) werden von der Kommandozeile aus gestartet. Schon bald gehen wir auf eine einfache Entwicklungsumgebung über: BlueJ. Sie basiert auf einer simplen grafischen Darstellung von Projekten und Programmen und wurde speziell für die Lehre hergestellt. Später wird diese grafische Darstellungsmethode verfeinert zu einer Untermenge der *Modellierungssprache UML* (Unified Modeling Language).

Gegen Ende des Lehrmoduls wird als Beispiel für eine professionelle Programmierumgebung (Integrated Development Environment, IDE) *Eclipse* angesprochen.

In der heutigen Software-Technik spielen *Entwurfsmuster* (Design-Patterns) eine wichtige Rolle. Einige Beispiele führen an das Arbeiten mit Entwurfsmustern heran.

Zur Typographie: Programme und insbesondere deren Variablen werden grundsätzlich nicht kursiv geschrieben. Kursiv stehen alle Variablen und Funktionsbezeichner, die nicht Bezeichner in einem Programm sind, also insbesondere die Variablen im mathematischen Sinn oder Abkürzungen und Bezeichner für Programmteile. Kursivschrift dient im Fließtext der Hervorhebung – beispielsweise beim erstmaligen Auftreten wichtiger Stichwörter. `Schreibmaschinenschrift` für Programme wird verwendet, wenn es der Übersichtlichkeit dient (Verbesserung der Unterscheidung zwischen Programmteilen und beschreibendem Text). Die mit einem Sternchen gekennzeichneten Aufgaben und Lektionen sind optional.

Gliederung

Literaturverzeichnis

Objektorientierte Programmierung

Balzert, H.: Lehrbuch Grundlagen der Informatik. Spektrum Akademischer Verlag, Heidelberg, Berlin, Oxford 1999, 2005

Dahl, O.-J.: The Roots of Object Orientation: The Simula Language. In Broy/Denert, 2002

Grams, T.: Denkfallen beim objektorientierten Programmieren. it 34(1992)2, 102-112

Meyer, B.: Object-oriented Software Construction. Prentice Hall, New York, London, Toronto, Sydney, Tokyo 1988 (deutsche Ausgabe: Objektorientierte Softwareentwicklung. Hanser, 1990)

Vorgehen bei der Software-Konstruktion (Software-Engineering)

Beck, K.: eXtreme Programming explained. Embrace change. Addison-Wesley, Boston 2000. *Die vorgeschlagene agile Programmierung ist Alternativen zum heutige weithin gebräuchlichen Software-Lebenszyklus-Modell.*

Grams, T.: Denkfallen und Programmierfehler. Springer, Berlin, Heidelberg 1990

Hering, E.: Software Engineering. Vieweg, Braunschweig 1989. *Eine kurz gefasste Einführung.*

Holzinger, A.; Slany, W.: XP + UE → XU, Praktische Erfahrungen mit eXtreme Usability. Informatik Spektrum (April 2006) 2, 91-97. *Aktueller Erfahrungsbericht. War mir eine Anregung im Zusammenhang mit der Reduktion des Vorgehensmodells auf das Wesentliche.*

Holzinger, A.; Slany, W.: XP+UE→XU. Praktische Erfahrungen mit eXtreme Usability. Informatik Spektrum 20 (April 2006) 2, 91-97

Hütte (Hrsg.: H. Czichos): Die Grundlagen der Ingenieurwissenschaften. 29. Auflage. Springer-Verlag, Berlin, Heidelberg 1991. *Knappe und prägnante Darstellung des Software-Lebenszyklus-Modells in Kapitel J 11 (Technische Inforamtik, Softwaretechnik).*

IEC 61508: Functional Safety (deutsch: VDE 0801: Funktionale Sicherheit)

Meyer, B.: From Structured Programming to Object-Oriented Design: The Road to Eiffel. Structured Programming (1989)1, 19-39. *Spricht für eine Weiterentwicklung des Software-Lebenszyklus-Modells in Richtung Objektorientierung und Generalisierung.*

Neumann, P. G.: Computer Related Risks. The ACM Press1995

V-Modell XT Release 1.1. Bundesministerium des Innern. Bundesrepublik Deutschland 2004

Programmentwurf mit der Unified Modeling Language (UML)

Oestereich, B.: Objektorientierte Softwareentwicklung. Analyse und Design mit der Unified Modeling Language. Oldenbourg, München, Wien 1998

Oestereich, B.: Die UML-Kurzreferenz für die Praxis - kurz, bündig, ballastfrei. 2. Auflage. Oldenbourg, München, Wien 2002

Spezielle Themen der Software-Konstruktion und Anwendungen

Aho, A. V.; Hopcroft, J. E.; Ullman, J. D.: Data Structures and Algorithms. Addison-Wesley, Reading, Mass. 1983. *Algorithmen auf Graphen: Dijstra u. a.*

Aho, A. V.; Sethi, R.; Ullman, J. D.: Compilers. Principles, Techniques, and Tools. Addison-Wesley, Reading, Mass. 1986. *Das berühmte "Drachenbuch" ist ein Standardwerk der praktischen Informatik.*

Alexander, C.: The Timeless Way of Building. Oxford University Press, New York 1979

Alexander, C.; Ishikawa, S.; Silverstein, M.: A Pattern Language. Towns, Buildings, Construction. Oxford University Press, New York 1977. *Die Bücher von Christopher Alexander sind eine überzeugende Darstellung seiner Pattern Language für grundlegende Muster der Architektur (von der Landschaftsplanung bis zur Gestaltung von Türen und Fenstern). Sie lieferten die Anregung zum Buch von Gamma und anderen. Dieses Buch gehört zu den Basiswerken der objektorientierten Programmierung.*

Broy, M.; Denert, E. (Eds.): Software Pioneers. Contributions to Software Engineering. Springer-Verlag, Berlin, Heidelberg 2002. *Hier von Interesse sind die Beiträge von DeMarco (Structured Analysis: Beginnings of a New Discipline), Gamma (Design Patterns - Ten Years Later), Guttag (Abstract Data Types, Then and Now), Parnas (The Secret History of Information Hiding).*

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Mass. 1995

Knuth, D.: The Art of Computer Programming. Volume 1. Fundamental Algorithms. Addison-Wesley, Reading, Mass. 1973. *Hier findet man alles Wichtige zu Listen und Bäumen.*

Neumann, K.; Morlock, M.: Operations Research. Carl Hanser, München, Wien 1993. *Algorithmen auf Graphen: Dijstra u. a.*

Die Programmiersprache Java

Arnold, K.; Gosling, J.: The Java™ Programming Language. Addison-Wesley, Reading, Mass. 1996. *Empfohlen als Begleitmaterial zur Lehrveranstaltung.*

Gosling, J.; Joy, B.; Steele, G.: The Java™ Language Specification. Version 1.0. Addison-Wesley, Reading, Mass. First printing: August 1996. ISBN 0-201-63451-1. *Der Java-Standard.*

Li, L.: Java - Data Structures and Programming. Springer, Berlin, Heidelberg 1998

Lindholm, T.; Yellin, F.: The Java Virtual Machine Specification, Addison-Wesley, 1996

Stroustrup, B.: The Design and Evolution of C++. Addison-Wesley, Reading, Mass. 1994. *Das Buch stellt deutlich die Philosophie der objektorientierten Sprache C++ heraus und macht dabei deutlich, warum die Sprache schwer beherrschbar und für die Lehre besonders schlecht geeignet ist. (Das ist das Schicksal aller weit verbreiteten Sprachen und Java ist auf dem besten Weg, dasselbe Schicksal zu erleiden. Aber Trost ist nah: Es wird dann wieder eine neue und aufs Wesentliche reduzierte Sprache geben und die anwendungsorientierte Lehre wird nicht zögern, auf diese zu wechseln.) Da das Buch eine klare Darstellung von Implementierungsdetails objektorientierter Sprachen enthält, ist es für Leute mit C-Kenntnissen eine hilfreiche Quelle für das tiefere Verständnis der objektorientierten Programmierung.*

Links

http://java.sun.com *Die zentrale Referenz für alle Fragen im Zusammenhang mit Java. Hier finden Sie das Java Software Development Kit (JavaTM 2 SDK, Standard Edition, Documentation, Version 1.4.1). Es enthält unter anderem die Java Sprachbeschreibung (Java Language Specification) und das API. API steht für Application Pro-*

gramming Interface und bedeutet soviel wie Allgemein verwendbare Klassen und Funktionen zur Verwendung durch den Anwendungsprogrammierer.

http://www.w3.org *Die Home Page des w3-Konsortiums. Hier findet man den aktuellen HTML-bzw. XML-Standard.*

http://www.bluej.org/ *Die ballastfreie interaktive Java-Entwicklungsumgebung für die Lehre. Alles Wichtige ist da: Klassendiagramm als Ausgangspunkt aller Aktivitäten. Java-Texteditor, Testumgebung, Debugger.*

1 Die Arbeitsumgebung und das erste Java-Programm

Die plattformübergreifende Web-Sprache Java

Hintergrund

Java hieß ursprünglich Oak und war für eingebettete Systeme der Konsumelektronik gedacht. Ihr Entwickler war James Gosling. Später wurde die Sprache für das Internet eingerichtet und umbenannt. Die endgültigen Form erhielt die Sprache durch James Gosling, Bill Joy, Guy Steele und andere.

Java ist eine *generell* verwendbare *objektiorientierte* und *plattformunabhängige* Programmiersprache, die das *Klassenkonzept* und *Nebenläufigkeit* (Threads) beinhaltet. Das vorliegende Skriptum hält sich hinsichtlich der Syntaxdarstellung an die Sprach-Spezifikation von Gosling/Joy/Steele (1996).

Java ist eine anwendernahe Sprache derart, dass maschinenspezifische Eigenheiten durch die Sprache nicht zugänglich sind. Andererseits umfasst sie ein *automatisches Speichermanagement* (Garbage-Kollektor) zur Vermeidung der häufigsten Programmierfehler, die auf direkte Speicherreservierung und -freigabe durch den Programmierer („baumelnde Zeiger", Datenmüll und dergleichen) zurückzuführen sind.

In Java sind Sprachelemente vermieden worden, die sich in der Anwendung als unsicher erwiesen haben. `Java`-Programme werden vom Java Compiler (javac.exe) in ein Bytecode-Format übersetzt, die von der Java Virtual Machine (java.exe) interpretiert werden kann. Dadurch kann grundsätzlich gewährleistet werden, dass ein importiertes Programm keinen Schaden auf der Host-Maschine anrichten kann (Bild 4.1).

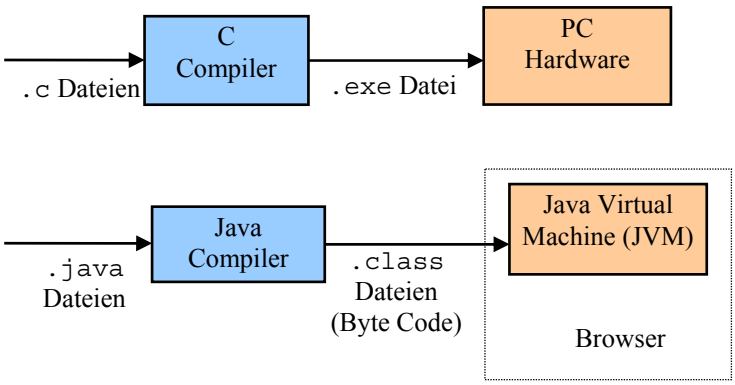


Bild 4.1 Architektur von C und Java im Vergleich

Die einfachen Datentypen werden für alle Maschinen und Implementierungen vorgeschrieben: Verschiedene Zweierkomplementdarstellungen von ganzen Zahlen, Gleitpunktzahlen in einfacher und doppelter Genauigkeit gemäß IEEE 754 Standard, ein Boolescher Typ und ein char Typ für Unicode-Zeichen.

Die benutzerdefinierten Datentypen heißen *Klassen*. *Objekte* (Exemplare[1]) sind die datenhaltigen Realisierungen solcher Datentypen. Sie werden grundsätzlich dynamisch erzeugt (wie die dynamischen Variablen in C).

---

[1] In diesem Zusammenhang wird zuweilen auch von „Instanzen" und „Instantiierung" gesprochen. Das sind schlechte Übersetzungen. Wir werden sie nicht verwenden.

Objekte werden über *Referenzen* angesprochen. Referenzen entsprechen den Zeigern bei Pascal, jedoch entfällt eine besondere Kennzeichnung der Dereferenzierung wie sie etwa in Pascal durch das Pfeilzeichen „^“ oder in C durch das Sternchen „*“ geschieht. Referenz-Typen sind die Klassen-Typen, die Interface-Typen und die Array-Typen.

Wie bei den Zeigern, können mehrere Referenzen auf ein Objekt verweisen. Die Klasse Object ist die einzige Wurzel der Klassenhierarchie von Java. Alle Klassen (einschließlich der Arrays) sind Subklassen dieser Standardklasse und enthalten folglich auch deren Methoden.

Für Folgen von Unicode-Zeichen existiert eine vordefinierte String-Klasse. Sie gehört zu den komfortabelsten Einrichtungen in Java. Nullterminierte Strings entfallen.

Zu den einfachen Datentypen wie short, int, long, float, double gibt es umhüllende Klassen (Wrapper-Classes), die für zusätzlichen Komfort beispielsweise bei der Umwandlung von Strings in Zahlen und umgekehrt sorgen: Short, Integer, Long, Float, Double.

Variablen sind typisierte Speicherbereiche. Die Variable eines einfachen Typs enthält genau den Wert eines solchen einfachen Typs. Die Variable eines Klassen-Typs kann die Null-Referenz oder die Referenz auf ein Objekt enthalten, deren Typ gleich der Klasse oder einer Unterklasse ist. Die Variable eines Array-Typs kann die Null-Referenz oder die Referenz auf ein Array enthalten. Die Variable vom Klassen-Typ Object kann die Null-Referenz oder die Referenz auf ein beliebiges Objekt enthalten, egal ob es sich um das Exemplar einer Klasse oder das eines Arrays handelt.

Die *Typumwandlung* (Type Cast) ändert den Typ einer Variablen oder eines Ausdrucks zur Compilierungszeit.

Java-Programme werden als *Packages* organisiert. Sie dienen der Abgrenzung von Namensbereichen. Der Internet Domain-Name kann für die eindeutige Kennzeichnung von Packages verwendet werden.

Ein erstes Programm

Java ist eine recht komplexe Sprache, die eine Reihe leistungsfähiger Konzepte enthält. In einem Kurzkurs wie diesem kann man Syntax und Semantik der Sprache nicht in voller Allgemeinheit einführen. Ausnahmsweise wird deshalb ein Beispielprogramm an den Anfang gestellt. Durch Exploration des Textes und Stöbern in den verwendeten Klassenbibliotheken soll ein Gefühl für die Programmierumgebung und den Programmierstil entstehen und es sollen erste Orientierungsmarken gesetzt werden. Im weiteren Verlauf des Kurses wird dann für ausgewählte Sprachelemente die allgemeine Syntax angegeben.

Jedes Java-Programm ist eine Sammlung von *Kompilierungseinheiten* (Compilation Unit). Jede Kompilierungseinheit ist ein Modul, also eine in sich geschlossene Datei, deren Inhalt vom Java Compiler separat übersetzt werden kann. Im einfachsten Fall – wie in diesem Abschnitt – wird nur eine solche Datei programmiert.

Eine Kompilierungseinheit enthält im Wesentlichen *Import-* und *Typ-Deklarationen*. Typ-Deklarationen sind zunächst einmal die Deklarationen von *Klassen*. Klassen sind die Grundstrukturelemente einer objektorientierten Programmiersprache[2].

[2] In der Programmiersprache C übernehmen die Funktionen diese Rolle.

Klassen sind Objekttypen, die sich als Fortentwicklungen der Strukturen (struct) in C auffassen lassen. Vor dem *Klassenkörper* (Class Body), der in geschweifte Klammern eingeschlossen ist, steht das Schlüsselwort class und ein vom Programmierer zu vergebender Klassenname.

Im Klassenkörper stehen die Deklarationen – das ist die Namensgebung – der *Felder* und *Methoden* dieser Klasse und der damit erzeugten Objekte[3]. Felder sind die den Objekten bzw. Klassen zugeordneten *Variablen*[4]. Sie werden in der Welt der Objektorientierung auch *Attribute* genannt. Methoden-Deklarationen beschreiben die den Objekten möglichen Aktionen. Sie entsprechen von Aufbau und Wirkungsweise weitgehend den Funktionen in C.

Eine Import-Deklaration macht Klassen anderer Kompilierungseinheiten bekannt und verfügbar.

Das folgende Programm-Beispiel beginnt mit einem Kommentar, der nur aus dem Dateinamen besteht: EinAus.java. Es folgt eine Import-Deklaration, in der die Klassen für die Ein- und Ausgabe aus dem java.io-Package verfügbar gemacht werden.

Es wird nur eine Klasse namens `EinAus` deklariert. Sie hat ein statisches Attribut namens `dataIn`. Das ist die Referenz auf ein Objekt vom Typ `StreamTokenizer`, das ein bequemes Lesen von Zeichenströmen erlaubt. Der `StreamTokenizer` wird sogleich mit einem neuen Objekt initialisiert. Gleichzeitig wird das `StreamTokenizer`-Objekt über die Parameterliste mit einem ebenfalls neuen Objekt der Klasse `InputStreamReader` versorgt. Und letzterem wird selbst wieder der Eingabe-Datenstrom `System.in`, das ist der Zeichenstrom von der Tastatur, mitgegeben[5].

Die Klasse `EinAus` besitzt zwei Methoden: `getNumber` und `main`. Eine Methode des Programms muss `main` heißen. Damit beginnt der Programmablauf.

```
/*EinAus.java*/

import java.io.*;

class EinAus {

  static StreamTokenizer dataIn= new StreamTokenizer(
    new InputStreamReader(System.in)
  );

  static int getNumber() {
    try {dataIn.nextToken(); return (int)dataIn.nval;}
    catch(IOException e) {
      System.out.println(e.toString());
      e.printStackTrace();
    }
    return 0;
```

[3] Manchem hilft es, sich eine Klasse als Plätzchenform zu denken und die Objekte als die damit erzeugten Plätzchen.

[4] in C: member

[5] Das sieht aus wie die berühmten russischen Puppen-in-Puppen. Es hat den Vorteil, dass der Anwender sich die von ihm benötigten Dienstleistungen selbst zusammenstellen kann und dass er nicht auf eine starre Ein-Ausgabe-Organisation wie bei den scanf- und printf-Funktionen von C angewiesen ist.

```
    }

    public static void main(String args[]) {
      int inch;
      System.out.print("? inch = ");
      inch= getNumber();
      System.out.println("! "+inch+" in = "+inch*2.54+" cm\n");
    }
  }
```

Methodenaufruf. Sei `o` ein Objekt der Klasse `C` und `r` eine Attribut von `o`, das selbst wieder eine Referenz auf ein weiteres Objekt ist, das eine Methode m besitzt. Die Methode lässt sich mittels Punktnotation dann folgendermaßen aufrufen: `o.r.m(…)`. Ist die Referenz `r` ein statisches Attribut der Klasse `C`, dann ist der Aufruf auch über die Klasse selbst möglich: `C.r.m(…)`. Statische Programmelemente werden durch das Schlüsselwert `static` ausgezeichnet. Die Anweisung `System.out.print("? inch = ");` bewirkt demnach einen Aufruf der `print`-Funktion des Ausgabe-Streams `out` der Klasse `System`.

Übersetzen und Starten des Programms. Der Java-Compiler ist eine ausführbare Datei mit dem Namen `javac.exe`. So wird die Übersetzung des in der Datei `EinAus.java` gespeicherten Programms aufgerufen: `javac EinAus.java`. Gestartet wird das Programm durch Aufruf der Java Virtual Machine (JVM), gefolgt vom Programmnamen: `java EinAus`. Dazu muss die JVM die Klassendatei `EinAus.class` finden können. Das setzt voraus, dass dem System auf eine der beiden folgenden Weisen der Pfad, der zur Klassendatei führt, bekannt gemacht wird:

1. Auf der Kommandozeile wird (für den Fall, dass der Pfad gleich „C:\Eigene Dateien\Java\EinfacheProgramme“ ist) der folgende Befehl eingegeben: `set classpath=C:\Eigene Dateien\Java\EinfacheProgramme`.

2. Im Betriebssystem wird die Umgebungsvariable `classpath` entsprechend gesetzt oder erweitert. Das geschieht über die folgenden Menübefehle, ausgehend vom Start-Menü: <Einstellungen> <Systemsteuerung> <System> <Erweitert> <Umgebungsvariablen>.

Gemeinsamkeiten und Unterschiede zwischen C und Java

Die Syntax von *Java-Ausdrücken* (Expression) entspricht weitgehend derjenigen von C-Ausdrücken. Die Kenntnis der letzteren wird hier vorausgesetzt. Die wesentlichen Unterschiede (Davis, 1996):

- In Java werden String-Konstanten nicht als nullterminierte Zeichen-Arrays realisiert. Stattdessen gibt es die vordefinierte Klasse String.
- Für Strings ist der Verkettungsoperator + definiert.
- Der Datentyp boolean ist kein numerischer Typ wie in C. Er hat die Werte false und true. Er kann nicht in einen Integer-Typ gewandelt werden und umgekehrt.
- Die Booleschen Operatoren && und || sind nur auf den booleschen Datentyp anwendbar. Die Operatoren wirken ansonsten genau wie in C.

- Die auf Ganzzahlen anwendbaren bitweise wirkenden logischen Operatoren (&, |, ^) sind ebenfalls auf boolesche Größen anwendbar.
- In Java werden die Felder von Klassen und Objekten initialisiert, notfalls mit einem Default-Wert. Dieser ist, je nach Typ, gleich false (bei Booleschen Variablen), '\u0000' (bei Zeichen), 0 (bei Ganzzahlen), +0.0 (bei Gleitpunktzahlen) oder null (bei Referenzen). Für lokale Variablen von Funktionen gibt es keine Default-Initialisierung.
- Zur Auszeichnung von Kommentaren gibt es zusätzlich den Doppelschrägstrich. Er leitet einen Kommentar ein, der bis zum Zeilenende reicht.

Achtung: Der Modulo-Operator (%) und der Divisionsoperator für ganze Zahlen (/) sind über die „ganzzahlige Division mit Rest“ definiert. Die Gleichung (a/b)*b+a%b=a ist für alle ganzzahligen a und b erfüllt. Es ist -7%3 gleich -1.

Achtung Denkfalle: Die Operatoren sind typabhängig (in C++ nennt man so etwas Operator Overloading). Zum Beispiel ist 1/2 gleich 0 und nicht gleich 0.5, wie man es eigentlich erwarten könnte.

Übungen

Experimentieren mit vorgegebenen Programmen. Führen Sie erste Erkundungen der Programmiersprache Java durch, indem Sie mit vorgegebenen Programmen experimentieren. Halten Sie sich dabei an die folgenden Regeln:

0. Machen Sie sich anhand des **Java SDK** die Bedeutung sämtlicher Bezeichner des Programms klar.
1. Schreiben Sie das Programm ab und übersetzen Sie es. Korrigieren Sie dabei eventuelle Tippfehler. Lassen Sie das Programm noch nicht laufen.
2. Erstellen Sie *Testfälle* einschließlich *Prognose:* Notieren Sie sich dazu die von ihnen beabsichtigten Eingaben und notieren Sie zu jeder Eingabe das von ihnen erwartete *Resultat* des Programmlaufs. Schreiben Sie alles auf, was auf dem Bildschirm erscheinen wird.
3. Starten Sie das Programm, geben Sie die Daten wie notiert ein.
4. *Vergleichen* Sie ganz genau das vom Computer gelieferte Resultat mit Ihrer Prognose.
5. *Analysieren* Sie die Abweichungen und ziehen Sie Ihre Lehren daraus.
6. Variieren Sie die Eingabe und verfahren Sie wie oben: Prognose, Vergleich und Analyse.
7. Wiederholen Sie das so lange, bis Sie sicher sind, die Funktion des Programms zu kennen.
8. Variieren Sie den Programmtext und führen Sie erneut die Schritte Prognose, Vergleich und Analyse aus.

1.1 Experimentieren Sie mit dem Programm `EinAus.java` und mit dem folgenden Programm `Euklid.java`.

```
/*Euklid.C*/

import java.io.*;

class Euklid {

  public static void main(String args[]) {
    if (args.length!=2)
      System.out.println("\nEINGABE: Euklid <int p> <int q>");
    else {
      int p= Integer.parseInt(args[0]),
          q= Integer.parseInt(args[1]);
      if (p<=0||q<=0) System.out.println("FEHLER: nicht positive Zahl");
      else {
        System.out.print("! ggT("+p+", "+q+") = ");
        while(0<q){int r=p%q; p=q; q=r;}
        System.out.println(p);
      }
    }
  }
}
```

2 Elemente der Programmkonstruktion

Die *Regeln des Natürlichen Software Engineering (NSE)* habe ich im November 2003 entdeckt, als ich Studenten bei der Arbeit zusah:

1. Sobald du eine Ahnung davon hast, was zu tun ist, hacke dir dein Programm zusammen. Das nennt man *Realisierung*.

2. Fertige einen Auszug daraus. So erhältst du *Konzept* und *Entwurf*.

3. Erstelle dann die *Spezifikation.* Darin erklärst du die überraschenden Eigenschaften deines Programms zu Features.

4. Mache schließlich dem Kunden klar, dass das, was du liefern kannst, er im Grunde seines Herzens auch haben wollte. Diese anspruchsvolle Aufgabe heißt *Requirements Engineering*.

In diesem Skriptum haben die Begriffe Programmkonstruktion, Software-Technik und Software Engineering dieselbe Bedeutung.

Die Software-Krise und das Software Engineering

Schon seit Beginn des Computerzeitalters schlägt sich die Zunft der Software-Ersteller mit dem Problem herum, dass sich ihr Baumaterial schneller fortentwickelt als die Fähigkeiten und Methoden, damit nützliche Werke zu bauen. Beispiele dafür sind haufenweise dokumentiert (Neumann, 1995). Zu den Ursachen der Software-Krise zählen die folgenden (Balzert, 1999):

- Software bietet einen großen Gestaltungsspielraum.
- Die realisierbaren Funktionen sind nicht durch physikalische Gesetze begrenzt.
- Software ist leichter und schneller änderbar als Hardware.

Zu den Kosten der Software (Hering, 1984): Fehlerbeseitigung ist in der Nutzungsphase 10...100mal teuerer als in der Herstellungsphase. Grundsätzlich gilt: Je später ein Fehler gefunden wird, umso größer sind die von ihm verursachten Kosten. Mehr als die Hälfte der Fehler wird während der Nutzungsdauer gefunden. Die Herstellungskosten verhalten sich zu den Kosten während der Nutzungsdauer wie 1:2.

Die Software-Kosten gewinnen an Bedeutung, denn die Ursachen von Systemversagen liegen zunehmend in der Software. Werkstattaufenthalte von Automobilen sind heute – anders als noch vor ein, zwei Jahrzehnten – überwiegend durch Programmprobleme verursacht.

Bereits im Jahr 1969 prägte der Münchner Informatik-Professor Friedrich L. Bauer auf einer NATO-Wissenschaftstagung in Garmisch das Wort *Software Engineering*. Dahinter stecken der Anspruch und die Hoffnung, die Software-Entwicklung ingenieurmäßig zu betreiben und dadurch der Probleme Herr zu werden.

Um die Darstellung genau dieser Software-Ingenieurswissenschaft geht es hier, wobei der konstruktive Aspekt betont wird: Das Zusammenfügen von Bauelementen zu Maschinen, die gewissen Anforderungen genügen. Die *Konstruktion* ist ein kreativer Prozess auf der Grundlage formaler Methoden und bewährter Techniken. Er enthält also sowohl intuitive als auch diskursive Elemente.

Programmqualität

Qualitätsmerkmale von Software sind

- Zuverlässigkeit (Grad des Vertrauens aufgrund geringer Versagenswahrscheinlichkeit oder großer Korrektheitswahrscheinlichkeit)
- Benutzerfreundlichkeit
- Sicherheit (Zuverlässigkeit hinsichtlich der Sicherheitsspezifikation)
- Effizienz (Minimaufwand hinsichtlich Speicherplatz- und Zeitbedarf bezogen auf den tatsächlichen Aufwand)
- Wartbarkeit
- Fehlertoleranz
- Fehlerfreundlichkeit
- Übertragbarkeit (Portabilität)
- Wiederverwendbarkeit
- Erweiterbarkeit
- Änderbarkeit

Dazu kommen die mittelbaren Qualitätsmerkmale, die im Dienste der oben formulierten stehen:

- Lesbarkeit (Struktur, Verständlichkeit)
- Prüfbarkeit (Einfachheit des Zuverlässigkeits-/Korrektheitsnachweises)

Dass Software chronisch mangelhaft ist und dass die Software-Industrie immer noch nicht den Standard einer ausgereiften Ingenieurwissenschaft erreicht hat, wird in vielen Publikationen beklagt. Leider gilt nach wie vor, dass sich im Bereich der Software das Baumaterial - also die Hardware und die Sprachen - schneller fortentwickelt als unsere Fähigkeiten und Methoden, damit nützliche Werke zu bauen.

Das Vorgehensmodell: Analyse – Entwurf – Realisierung – Abnahme

Es ist eine der großen Erfahrungen, die im Laufe der so genannten Software-Krise gemacht wurde, dass der Mensch (ein geborener Jäger und Sammler!) zu voreiligen Realisierungen neigt und dass die Leichtigkeit der Software-Erstellung dieser Neigung leider entgegen kommt. Dem werden hier die Methoden der Software-Entwicklung entgegengestellt: Lebenszyklusmodelle, strukturiertes Programmieren, modulares Programmieren, objektorientierte Programmiermethodik, Extremes Programmieren.

Für diesen einführenden Kurs in die Softwaretechnik wird ein Vorgehensmodell für Projekte und Vorhaben verbindlich eingeführt[1]. Vor der Beschäftigung mit diesem Modell muss etwas über die Projektbeteiligten und deren Interessenlagen gesagt werden.

Rollenspiele. In der Praxis treffen Projektbeteiligte aufeinander, die ganz unterschiedliche Absichten, Hintergründen und Vorgehensweisen haben. Diese Vielfalt lässt sich in einem Software-Kurs nur unzureichend nachbilden. Dennoch wollen wir es versuchen. In diesem Kurs werden die Aufgaben und Übungen vorzugsweise in kleinen Gruppen aus drei Personen erledigt (Minimum: zwei, Maximum: vier). Legen Sie die Arbeit zu den Übungen als Rollenspiele an. Machen Sie vorab klar, wer die Rolle des Auftraggebers bzw. *Kunden* übernimmt und wer die des Auftragnehmers bzw. *Lieferanten*. Eine weitere sinnvolle Rollenaufteilung ist

[1] Quellen: V-Modell XT Release 1.1, IEC 61508, Holzinger/Slany (April 2006).

die zwischen dem *Konstrukteur* und dem *Tester* der Software. Teilen Sie von Anfang an die Arbeiten unter den Gruppenmitgliedern auf

Das Vorgehensmodell. Dieses Modell stellt mit seinen Abgrenzungen und dem strikten Nacheinander der *Phasen* klare Forderungen an die Projektbeteiligten. Es unterscheidet sich darin von anderen Philosophien – beispielsweise dem eXtreme Programming. Heute findet man Zwischenformen dieser Vorgehensweisen, beispielsweise das wiederholte Durchlaufen von Kurzzyklen. Die Kurzzyklen selbst richten sich dann wieder nach angemessen reduzierten Vorgehensmodellen.

Für diesen Kurs wird ein *vierstufiges Vorgehensmodell* verbindlich eingeführt. Es ist bei der Lösung der Aufgaben konsequent anzuwenden.

1. *Analyse.* Anforderungserfassung. Ergebnis ist ein *Pflichtenblatt* mit einer Beschreibung der anfallenden Aufgaben und einer *Spezifikation* der zu realisierenden Funktionen. Die Spezifikation ist die mathematisch genaue Beschreibung einer Funktion: Eingabe, Verarbeitung und Ausgabe (EVA-Prinzip). Im Pflichtenblatt enthalten sind ist die *Testspezifikation,* sie enthält die die *Testfälle.* Sie sind die Basis für die *Validierung* des Programms und für die *Projektabnahme.* Jeder Testfall besteht aus den Eingabedaten und einer detaillierten Prognose des Ergebnisses. Schon in der Analysephase werden die Vorgaben für *Verifizierung* der folgenden Phasen festgehalten[2].
2. *Entwurf.* Erstellen einer grafischen Darstellung der Programmstruktur. Darstellung zentraler Abläufe und Algorithmen mittels Pseudocode (also unter Verwendung der Strukturelemente einer Programmiersprache).
3. *Realisierung.* Programmierung in der Sprache Java. Codierung.
4. *Abnahme.* Durchführung der *Tests*. Prüfung der Dokumentation.

Hinweis für das Praktikum: Jede der Praktikumsaufgaben ist nach Maßgabe dieses Vorgehensmodells zu bearbeiten. Für die Schritte 1 und 4 machen Sie bitte vorab eine Rollenzuweisung der Teammitglieder: Die Teammitglieder spielen entweder die Kunden- oder Lieferanten-Rolle. Bevor der zweite Teilschritt begonnen wird, ist das Pflichtenblatt des ersten Teilschritts von allen Teammitgliedern und dem Kursleiter zu genehmigen und abzuzeichnen. Dasselbe gilt für den dritten Teilschritt: Die Realisierung startet erst, wenn der Entwurf vereinbart worden ist. Für Entwurfs- und Realisierungsphase werden die Rollen neu verteilt: Alle machen mit. Für die Abnahme werden die Kunden-/Lieferanten-Rollen wieder aktiviert. Alle Schritte werden in einem *Projektprotokoll* schriftlich festgehalten. Das Protokoll erfüllt ein paar Minimalforderungen an schriftliche Ausarbeitungen. Es umfasst folgende Teile: Titel, Datum, Namen der Projektbeteiligten mit Aufgabenaufteilung und Textteil. Der Textteil beginnt mit einer Einleitung und ist ansonsten entsprechend dem Vorgehensmodell zu gliedern: Analyse, Entwurf, Realisierung, Abnahme. In der Einleitung wird kurz auf das Problem, das Ziel und die verwendeten Methoden eingegangen. Das Projektprotokoll ist Grundlage des Testats der Aufgabe durch den Kursleiter. Die Testate werden individuell vergeben. Die Leistung des Einzelnen wird danach beurteilt, ob er wesentliche Teile selbst gemacht und ob er die Programmstruktur sowie die zentralen Algorithmen verstanden hat.

---

[2] Die Validierung zeigt, dass man das richtige System gebaut hat; Verifizierung ist der Beweis, dass man das System richtig gebaut hat. Ersteres weist man typischerweise durch Tests von Anwendungsfällen nach, letzteres durch Beweis der Spezifikationserfüllung.

Achtung: Die Aufgaben sind nur bei Arbeitsteilung problemlos im Zeitrahmen machbar!

Analyse: Von den Anforderungen zum Pflichtenblatt

Das Pflichtenblatt ist ein Vertrag zwischen Auftraggeber und Auftragnehmer der Software. Es ist die Basis für alle weiteren Aktivitäten des Softwareherstellers. Daraus wird dann im ersten Schritt die Spezifikation, also die mathematisch hieb- und stichfeste Beschreibung der zentralen Systemkomponenten entwickelt. Die Spezifikation beschränkt sich darauf, zu beschreiben, *was* das System leisten soll, also seine Funktion. *Wie* diese Funktion realisiert wird, ist nicht Gegenstand der Spezifikation. Das Pflichtenheft sollte möglichst umfassend und eindeutig sein. Das zu erreichen, ist Gegenstand der *Systemanalyse*. Sie stellt und beantwortet die folgenden Fragen:

- *Vollständigkeit:* Welche Funktionen werden gebraucht? Wie sehen die wichtigsten Szenarien der Anwendung aus?
- *Konsistenz:* Sind die gewünschten Funktionen miteinander verträglich? Gibt es Widersprüche?
- *Eindeutigkeit:* Gibt es nur eine Interpretation?
- *Durchführbarkeit:* Reichen die technischen Möglichkeiten aus?

Gliederung des *Pflichtenhefts* (Balzert, 1996, S. 106 ff.):

- *Ziel:* Goals und Non-Goals angeben. Qualitätsziele formulieren. Anwendungsbereich, Zielgruppen und Betriebsbedingungen angeben.
- *Produktfunktionen:* Funktionen, die zur Erfüllung der Anforderungen nötig sind, und das Produktverhalten (Performance, Leistungsfähigkeit, Antwortzeiten) festlegen. Festlegung der Schnittstellen zu der Soft- und Hardware-Umgebung.
- *Bedienoberfläche:* Auflistung der Bedieneraktionen und der Reaktionen darauf. Bedienelemente und Darstellungsarten definieren: Bildschirmlayout, Drucklayout, Tastaturbelegung, Dialogstruktur.
- *Testszenarien:* Die Szenarien der Anwendung in konkreten Testfällen niederlegen. Sie müssen möglichst umfassend sein. Sie sind so klar und eindeutig darzustellen, dass sich entscheiden lässt, ob die Spezifikation (und später das System) den Anforderungen entspricht (*Validation*).

Anhang: Brauchen wir agileres Programmieren?

In der Welt der Informatik ist die richtige Vorgehensweise und Programmiermethodik Dauerthema. Paradigmata werden aufgestellt und in Frage gestellt. Grundannahmen werden angezweifelt, Gegenpositionen formuliert. Gerade die Software-Pioniere, die am heutigen Wissensstand großen Anteil haben, geben Hinweise darauf, was zukünftig anders werden muss.

John V. Guttag (Broy/Denert, Software Pioneers, 2002), der „Erfinder“ der *abstrakten Datentypen* weist darauf hin, dass viele der alten Rezepte wirkungslos werden, wenn man an das Programmieren von *eingebetteten Systemen* (embedded systems) geht. Hier werden Beschränkungen hinsichtlich Verarbeitungskapazität und Speicherplatz wieder wirksam, wie man sie aus der Anfangszeit der Programmierung kennt. Und er sagt: „I think it’s with no doubt true that almost all software in the future will be embedded systems.”

Tom DeMarco (Broy/Denert, Software Pioneers, 2002) hält heute einen Großteil der in seinem berühmten Buch „Structured Analysis and System Specification“ von 1975 aufgestellten Prinzipien der Software-Entwicklung für nicht mehr tragfähig. Er verweist auf die Arbeit von Kent Beck „eXtreme Programming“ (2000). Dort werden Dinge empfohlen, die früher als eklatanter Verstoß gegen die Tugenden der ordentlichen Programmentwicklung gegolten haben.

Agile Programming, eXtreme Programming, XP: Kent Beck bestreitet, dass die Kosten für Fehler und Programmänderungen von Projektphase zu Projektphase enorm steigen. Änderungen sind nach seiner Auffassung auch bei fortgeschrittenem Projektverlauf leicht möglich - und das dank der heutigen Software-Technologie. Er gibt den Schlachtruf aus: „Embrace Change“. Damit widerspricht er dem vorsichtigen Ingenieur alter Schule, der nach der Regel „Change is Bad“ handelt.

Ziel des XP ist der möglichst frühzeitige produktive Einsatz eines Systems. Kent Beck gibt den Rat, zunächst mit der Realisierung eingeschränkter Funktionalität zu beginnen und die Software erst nach und nach zur vollen Reife zu bringen, wobei der Kunde und Anwender Gelegenheit bekommt, seine Wünsche ebenfalls fortzuentwickeln. System und Anforderungen bleiben im Fluss.

Zu den von ihm vorgeschlagenen Methoden gehören

1. *Testfallgetrieben Entwicklung:* Durch Testfälle wird festgelegt, was das System zu leisten hat. Die Spezifikation verliert an Bedeutung, dementsprechend auch die Verifikation. Die Validation erhält überragendes Gewicht.
2. *Paarprogrammierung:* Alle Programmteile werden von je zwei Programmierern erstellt, die sich in einen Computer teilen. Einer übernimmt eher strategische Aufgaben und der andere schreibt. Rollen und Paarbildungen innerhalb des Projekts wechseln fortwährend.
3. *Codezentrierte Kommunikation:* Teamweite Programmierregeln und die Befolgung des Grundsatzes der Einfachheit (im Sinne von Einschränkung, Einfalt) führt zu Programmen, die für alle lesbar sind.
4. *Clusterbildung:* Codieren, Testen, Anforderungen erfassen und Entwurf sind ineinander verwobene Aktivitäten und Bestandteile eines fortwährenden Prozesses.

Funktionale Sicherheit. In der Welt der Ingenieure mit ihren Normen und Sicherheitsstandards sieht die Sache anders aus. Die Grundnorm IEC 61508 ist europaweit als EN 61508 veröffentlicht. Diese Grundnorm enthält eine umfassende Definition von Sicherheitsanforderungen für Software und Hardware (Software Integrity Levels, SIL) und verpflichtet die Anwender der Norm zur Einhaltung eines Vorgehensmodells, das die folgenden Phasen umfasst: Systemanalyse, Entwurf, Modulentwurf, Codierung, SW/HW-Integration und Wartung. Präzisierungen der Norm existieren in vielen Bereichen, beispielsweise für die Software in Bahnanwendungen (DIN EN 50 128). Und in diesen Bereichen wird die Norm dann auch verbindlich vorgeschrieben.

Übungen

2.1 (Rollenspiel): Ziel ist der Aufbau eines kleinen Software-Systems für die *Verwaltung der Mitgliederdaten eines Vereins*. Teilnehmer Ihrer Gruppe übernehmen die Rolle des Auftraggebers andere die des Auftragnehmers (Entwickler) auf. Spielen Sie die Analysephase durch: Der Auftraggeber skizzieren die Anforderungen an ein solches System. In einer Teamsitzung

wird die Systemanalyse durchgeführt. Daraufhin erstellt der Auftragnehmer ein Pflichtenblatt. Der Auftraggeber prüft das Pflichtenblatt und macht gegebenenfalls Änderungsvorschläge. Der Auftragnehmer prüft seinerseits auf Realisierbarkeit. Schließlich wird das Pflichtenblatt von Auftraggeber und Auftragnehmer unterschrieben.

2.2 *Rechteck.* Schreiben Sie ein Programm, das auf der Kommandozeile ein Paar ganzer Zahlen entgegennimmt und diese als Rechteckseiten interpretiert. Auf Bildschirm ausgeben soll es

1. eine Klassifizierung der geometrischen Figur als echtes Rechteck, Quadrat, Strecke oder Punkt und
2. die Länge der Diagonalen.

Geben Sie sich insbesondere bei der Pflichtenblatterstellung große Mühe. (Der Entwurf spielt hier keine große Rolle.) Sie gehen erst an das Programmieren, wenn das Pflichtenblatt von allen – auch dem Kursleiter – abgesegnet worden ist.

Tipp zum Auffinden der mathematischen Funktionen in Java: Gehen Sie auf die Java SDK-Seite und von dort aus weiter zur Schnittstelle für die Anwendungsprogrammierung (Application Programming Interface, API). Dort finden Sie im Java.lang-Package die Math-Klasse mit den mathematischen Standardfunktion.

2.3 *Assoziativarray.* Ein Assoziativarray ist eine Array, das eine beliebige (?) Anzahl von Wertepaaren enthält. Die Wertepaare bestehen aus einem Schlüssel (Key) und einem Wert (Value). Die Schlüssel sollen hier Textfolgen (also vom Typ String) sein und die Werte ganze Zahlen (int). Das zu konstruierende Progamm `AssociativeArray` soll Zeichenfolgen entgegennehmen. Jede Zeichenfolge wird im Assoziativarray nur einmal gespeichert, auch bei mehrmaliger Eingabe derselben Folge. Der jeweils zugeordnete Wert ist gleich der Vielfachheit, mit der eine Zeichenfolge eingegeben worden ist. Nutzen Sie aus, dass für die Schlüssel eine vollständige Ordnung definiert ist (lexikalische Ordnung). Sobald ein leerer String eingegeben wird, gibt das Programm alle Zeichenfolgen in sortierter Folge zusammen mit dem jeweils zugeordneten Wert aus. Dann endet es.

Hinweis: Bevor Sie an die Aufgabe herangehen, verteilen Sie innerhalb Ihrer Gruppe die Rollen. Arbeiten Sie dann konsequent nach dem Vorgehensmodell. Jede abgeschlossene Phase wird auf dem Lösungsblatt testiert. Erst dann geht es weiter zur nächsten Stufe.

3 Das Java-Tutorial NumberList

Das Klassenkonzept: Kapselung und Vererbung und Polymorphismus

Klassen sind eine direkte Verallgemeinerung der Strukturen (C) oder Records (Pascal). Bereits bei den Strukturen in C ist es möglich, neben den normalen Membervariablen auch Pointer auf Funktionen als Member aufzunehmen. Die normalen Membervariablen heißen in der ooP *Attribute*[1] und die Member-Funktionen heißen *Methoden*.

Objekte sind die (datenhaltigen) Realisierungen der Klasse. Sie werden auf dem Heap angelegt. Es sind *dynamische Variable.*

In der objektorientierten Programmierung treten an die Stelle der Pointer auf Objekte die *Referenzen*. Fürs erste genügt es, sich die Referenzen als Pointer vorzustellen.

Klassen bieten - anders als der Datentyp Struktur - die Möglichkeit der *Typerweiterung*. Es ist nämlich möglich, eine Klasse durch Bezugnahme auf eine andere Klasse zu deklarieren mit dem Effekt, dass die neu definierte Klasse sämtliche Attribute und Methoden der ursprünglichen Klasse erbt. Die Deklaration der Unterklasse kann weitere Attribute und Methoden hinzufügen, oder auch ererbte Methoden *überschreiben*. (Beim Überschreiben bleiben Funktionsname und Argumenttypen unverändert. Nur der Funktionskörper ist neu.) Die ursprüngliche und die neu definierte Klasse stehen zueinander in einer hierarchischen Beziehung. Die ursprüngliche Klasse ist in dieser Beziehung die *Oberklasse*. Die neu definierte Klasse ist in dieser Beziehung die *Unterklasse.* Ober- und Unterklasse heißen - in enger Anlehnung an den Java-Sprachgebrauch - auch *Superklasse* bzw. *Subklasse*.

Sei nun *C* eine Klasse, *D* eine ihrer Unterklassen und *x* eine Referenz auf Objekte vom Typ *C*. Es ist nun möglich, der Referenz *x* auch Objekte vom Typ *D* zuzuweisen. Die Tatsache, dass eine Referenz auf Objekte verschiedenen Typs verweisen kann, wird *Polymorphismus* genannt.

Das Überschreiben in Kombination mit dem Polymorphismus stellt die wesentliche und mächtige Neuerung der ooP dar.

Das Projekt NumberList (Version 1)

Aufgabenstellung

Zur Einführung in die objektorientierte Programmierung mit Java stellen wir uns die Aufgabe, ein Programm zu schreiben, das eine Folge von ganzen Zahlen von der Tastatur entgegennimmt, diese in einer linearen Liste

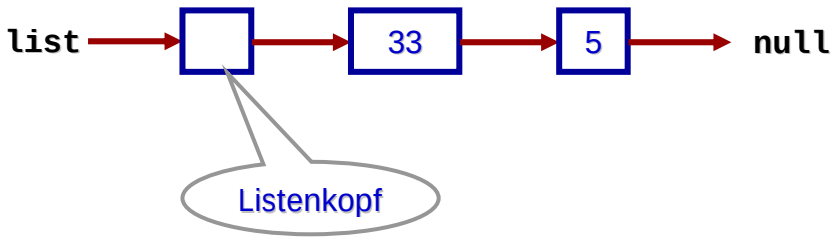


Bild 3.1 Lineare Liste ganzer Zahlen

[1] Attribute werden in Java auch Felder (field) genannt.

abspeichert und anschließend über Bildschirm ausgibt. Bild 3.1 zeigt den Fall, dass zuerst die Zahl 5, dann die Zahl 33 eingegeben worden ist. Neue Zahlen werden also unmittelbar am Listenkopf eingeführt. Die Ausgabe der Zahlen soll in der Eingabereihenfolge geschehen, also vom Ende der Liste Richtung Kopf.

Die Klasse Number

So sieht die Number-Klasse in der Programmiersprache Java aus:

```
public class Number {
  int x;
  Number next;
  Number(int x) {this.x=x;} //Default-Initialisierung fuer next
  void in(Number z) {z.next=next; next=z;}
  void write() {
    if (next!=null) next.write(); //Rekursion
    System.out.println("! "+x);
  }
}
```

Die *Klasse* enthält *Attribute*: x und next, den *Konstruktor* Number(int x) und die *Methoden* in(Number z) und write().

Konstruktoren unterscheiden sich von den Methoden dadurch, dass es keine Rückgabewerte gibt. Sie werden bei der Generierung von Objekten aufgerufen und dienen hauptsächlich der Initialisierung der Attribute des Objekts. Konstruktoren werden mit dem Klassennamen bezeichnet.

Das Attribut x enthält den Zahlenwert und next die Referenz auf das in der linearen Liste folgende Number-Objekt. Über den Konstruktor Number(x: int) wird dem erzeugten Number-Objekt die Zahl x mitgeteilt, die es repräsentiert. Die Methode in(z: Number) hat die Aufgabe, ein Objekt z der Klasse Number hinter dem aufrufenden Objekt in die lineare Liste einzufügen. Die Methode write() soll zunächst die write-Methode des folgenden Elements aufrufen und anschließend den Zahlenwert x auf dem Bildschirm ausgeben. Das bewirkt die rekursive Ausgabe der Zahlenwerte der Liste, angefangen beim Listenende bis zurück zum aufrufenden Number-Objekt.

Da in Java die Attribute von Objekten - im Unterschied zu den lokalen Variablen von Methoden - grundsätzlich eine Default-Initialisierung erhalten, ist im Konstruktor Number(int x) die Initialisierung der next-Referenz mit dem Nullobjekt (null) entbehrlich. Die Anweisung „next=null;“ kann also entfallen.

Vererbung: Zahlen und Zahlenpaare

Das Beispiel soll nun so erweitert werden, dass die Liste nicht nur einfache Zahlen, sondern außerdem Zahlenpaare enthalten kann. Wir benötigen also eine weitere Klasse für die Zahlenpaare. Wir definieren diese Klasse namens Pair als *Subklasse (Unterklasse)* von Number. Das hat die folgenden Vorteile:

1. Wir können uns bei der Definition der Pair-Klasse auf die notwendigen Ergänzungen und Modifikationen beschränken. Das ooP-Prinzip der *Vererbung* sorgt dafür, dass alle anderen Attribute und Methoden von der *Oberklasse (Superklasse)* Number übernommen werden.

2. Die Handhabung von Number und Pair wird vereinheitlicht. Das wird durch das ooP-Prinzip des *Polymorphismus* möglich. Im folgenden Unterabschnitt wird das näher erläutert.

Die write-Methode muss in der Pair-Klasse *überschrieben* werden, da jetzt ja zwei Zahlen auf Bildschirm auszugeben sind. Die in-Methode kann einfach übernommen werden: Zahlenpaare (Objekte der Klasse Pair) werden wie einzelne Zahlen (Objekte der Klasse Number) behandelt.

Der Java-Programmtext der Pair-Klasse sieht so aus:

```
public class Pair extends Number{
  int y;
  Pair(int x, int y) {super(x); this.y=y;}
  void write() {
    if (next!=null) next.write(); //Rekursion
    System.out.println("! "+x+" "+y);
  }

}
```

Mit super(x) wird der Konstruktor der Superklasse, also Number(x) aufgerufen, so dass im Pair-Konstruktor hier nur noch die Initialisierung des Attributs y nachzuholen ist.

Polymorphismus: Zahl oder Zahlenpaar?

Der Typ einer Referenz wird in der Deklaration festgelegt. Eine Referenz, deren Typ eine bestimmte Klasse ist, kann auf Objekte dieser Klasse und auf Objekte aller Unterklassen dieser Klasse verweisen. Sei beispielsweise r eine Referenz der Klasse K und L eine Unterklasse von K. Dann kann r auf Objekte der Klasse K und auf Objekte der Klasse L verweisen. Einer Number-Referenz wie next ist also nicht anzusehen, ob sie gerade auf ein Objekt der Klasse Number oder auf ein Objekt Klasse Pair zeigt.

Neben dem durch die Deklaration festgelegten statischen Typ haben Referenzen also noch einen *dynamischen Typ*. Letzterer ist gleich dem Typ desjenigen Objekts, auf das gerade verwiesen wird. Der dynamische Typ einer Referenz ist also veränderlich. Dieser *Polymorphismus* von Referenzen ist von zentraler Bedeutung für die objektorientierte Programmierung.

Der Aufruf einer überschriebenen Methode (wie etwa das write der Number- und der Pair-Klasse) richtet sich grundsätzlich nach dem dynamischen Typ der Referenz. Anders ausgedrückt: Der Methodenaufruf einer überschriebenen Methode hängt vom Typ des Objekts ab und nicht etwa vom statischen Typ der Referenz.

Wie sich das nutzen lässt, wird anhand der Hauptklasse NumberList deutlich, das die lineare Liste aus Zahlen und Paaren aufbaut.

Die Programmstruktur ist in Bild 3.2 wiedergegeben. Es handelt sich um ein *Klassendiagramm*, wie es von der interaktiven Entwicklungsumgebung BlueJ erstellt wird. In diesem Klassendiagramm werden (in Anlehnung an die *Unified Modeling Language, UML*) die Vererbungsbeziehung durch Pfeile dargestellt, deren Pfeilspitzen als offene Dreiecke gestaltet sind (Extends-Relationen). Die gestrichelten Pfeile stellen die Zugriffsbeziehungen dar

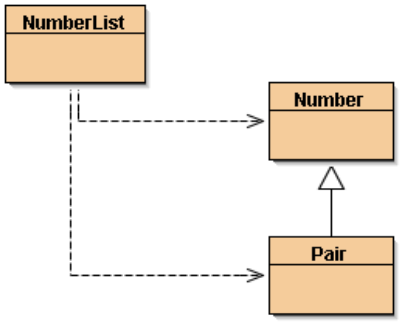


Bild 3.2 Das Klassendiagramm des Projekts NumberList

(Uses-Relationen). Der Java-Programmtext der Hauptklasse des Programms wird im folgenden Abschnitt erläutert.

Die Hauptklasse

Bis jetzt war zu sehen, wie die einzelnen Objekte unserer Zahlenliste aufgebaut sind. Wo aber ist das Programm, das sich um die Eingabe der Zahlen, das Organisieren der Liste und die Ausgabe kümmert? Diese Dinge übertragen wir einer Hauptklasse mit Namen NumberList. Sie enthält eine öffentliche statische Methode namens main. Ähnlich wie in C wird mit der main-Funktion der Programmablauf gestartet.

Der Programmtext der Hauptklasse:

```
import java.io.*;

public class NumberList {
  static Number list= new Number(0); //Head of Number List
  static StreamTokenizer dataIn= new StreamTokenizer(
    new InputStreamReader(System.in)
  );
  static char getChar() {
    try {
      dataIn.nextToken();
      return Character.toUpperCase(dataIn.sval.charAt(0));
    } catch(IOException e) {
      System.out.println(e.toString());
      e.printStackTrace();
    }
    return 0;
  }
  static int getNumber() {
    try {dataIn.nextToken(); return (int)dataIn.nval;}
    catch(IOException e) {
      System.out.println(e.toString());
      e.printStackTrace();
    }
    return 0;
  }
  public static void main(String[] args) {
    char c;
    System.out.println("Liste ganzer Zahlen. Eingabe:");
    do {
      System.out.print("? <Z>ahl, <P>aar, <S>chluss: ");
      if ((c=getChar())=='Z') {
         System.out.print("? Zahl= ");
         list.in(new Number(getNumber()));
      } else if (c=='P') {
         System.out.print("? Paar= ");
         list.in(new Pair(getNumber(), getNumber()));
      }
    } while (c!='S');
    System.out.println("Ausgabe der Zahlenliste");
    if (list.next!=null) list.next.write();
  }
}
```

Die Attribute der Klasse NumberList heißen list und dataIn. Sie sind als statisch (Schlüsselwort static) deklariert. Sie gehören zur Klasse und sind auch ohne Objekt existent und erreichbar. In *Variablenzugriffen* und *Methodenaufrufen* kann an die Stelle eines Objekt-

Namens auch der Klassenname treten. Diese statischen Methoden werden auch *Klassenmethoden* genannt.

Die Referenz list wird mit einer Zahl (Typ Number) initialisiert. Damit ist der Listenkopf kreiert. Da der Listenkopf selbst nicht zur Liste von Zahlen gehört, ist der im Konstruktor übergebene Wert 0 ohne Bedeutung.

Die Referenz dataIn dient der Verwaltung der Tastatureingabe. Hier werden zwei Klassen der Java-Bibliothek java.io verwendet. Die Definition dieser Klassen findet man im *Java Software Development Kit* (Java SDK) der Firma Sun Microsystems (http://www.javasoft.com).

Um den Zugriff auf die Klassen einer Bibliothek zu vereinfachen, werden sie in einer Import-Deklaration aufgeführt. Die Import-Deklaration „import java.io.*;" erlaubt den Zugriff auf alle Klassen der Bibliothek java.io allein über ihren jeweiligen Klassennamen.

Das Einlesen von Zeichen und Zahlen von der Tastatur geschieht mit den Methoden getChar und getNumber. Mit der try-catch-Anweisung wird eine gewisse *Fehlertoleranz* erreicht. Dieser Mechanismus wird weiter unten erläutert.

Fett geschrieben sind die Programmzeilen, in denen der Polymorphismus von Referenzen eine ausschlaggebende Rolle spielt. Beim Aufbauen der Liste werden einmal - mittels der Anweisung new Number(getNumber()) - Objekte vom Typ Number erzeugt und ein andermal - mittels der Anweisung new Pair(getNumber(), getNumber())-Objekte vom Typ Pair. Beide Arten von Objekten werden mit der Number-Methode in(...) eingefügt. Der Formalparameter dieser Methode hat den statischen Typ Number und, je nach Typ des aktuellen Parameters, den dynamischen Typ Number oder Pair. Eine ähnliche Aussage gilt für die next-Referenzen.

```
Liste ganzer Zahlen. Eingabe:
? <Z>ahl, <P>aar, <S>chluss: z
? Zahl= 1
? <Z>ahl, <P>aar, <S>chluss: p
? Paar= 2 3
? <Z>ahl, <P>aar, <S>chluss: p
? Paar= 4 5
? <Z>ahl, <P>aar, <S>chluss: z
? Zahl= 6
? <Z>ahl, <P>aar, <S>chluss: p
? Paar= 7 8
? <Z>ahl, <P>aar, <S>chluss: z
? Zahl= 9
? <Z>ahl, <P>aar, <S>chluss: s
Ausgabe der Zahlenliste
! 1
! 2 3
! 4 5
! 6
! 7 8
! 9
```

Bild 3.3 DOS Bildschirmausschnitt

Da sich Methodenaufrufe nach dem dynamischen Typ einer Referenz richten, bewirkt der Aufruf „list.next.write();", dass für jedes Objekt, ob vom Typ Number oder Pair, die jeweils passende Version der write-Methode aufgerufen wird.

In Bild 3.3 ist der Bildschirmausschnitt mit den Resultaten eines Programmlaufs wiedergegeben.

Ist- und Hat-Beziehungen

An die Vererbungspfeile (Extends-Relation) kann man sich als Relationsbezeichung „ist" oder „ist ein" angeschrieben denken. Also: Ein Pair-Objekt *ist ein* Number-Objekt. Die Ist-Relationen bilden einen Baum in Parent-Darstellung. Das ist die Repräsentation der *Klassenhierarchie*.

An den Pfeile für Uses-Beziehungen stelle man sich demgegenüber ein „hat“ vor: Ein Objekt der Klasse NumberList (oder auch die Klasse selbst) *hat* Number- und Pair-Objekte. Die Hat-Relationen bilden ein Netz, da alle möglichen Paarbildungen denkbar sind.

Hat-Beziehungen sind – im UML-Sprachgebrauch – (gerichtete) *Assoziationen*. Erfüllen diese Assoziationen zusätzliche Bedingungen, spricht man auch von *Aggregationen* (das Ganze *hat* Teile) und *Kompositionen* (die Teile sind vom Ganzen existenzabhängig) .

Entwurf: Vom Pflichtenblatt zur Architektur

Bisher haben wir der Entwurfsphase noch nicht die ihr gebührende Aufmerksamkeit zukommen lassen. Das lag am Mangel an Hilfsmitteln. Dieses Loch können wir nun - behelfsmäßig – stopfen. Folgende Software-Werkzeuge stehen nun zur Verfügung: Konsole, Editor, BlueJ, Java System Development Kit (Java SDK, auch: JDK) und Java Runtime Environment (JRE). Mit diesen Hilfsmitteln, insbesondere mit der Entwicklungsumgebung BlueJ, können wir dem Software-Entwurf (Phase 2 des Vorgehensmodells aus Abschnitt 2) eine prägnante Form geben.

2. *Entwurf.* Erstellen einer grafischen Darstellung der Programmstruktur als Klassendiagramm. Darstellung der Klassen mit den Deklarationen ihrer Attribute und Methoden. Ein Methodenrumpf enthält nur Kommentare, die die Funktion spezifizieren. Die Spezifikation beschreibt, *was* die Funktion machen soll, nicht aber *wie* sie das tut. Im Idealfall geschieht das durch Angabe der Vor- und der Nachbedingung (pre und post).

Nehmen wir als Beispiel das Projekt NumberList. Das Klassendiagramm des Entwurfs zeigt das Bild 3.2. Der Entwurf der Klasse Number hätte beispielsweise so aussehen können:

```
public class Number {
  int x;
  Number next;
  Number(int x) {this.x=x;}
  void in(Number z) {
    //pre: das Objekt z ist nicht in eine Liste eingebunden;
    //     z=Z, this.next=N
    //post: this.next=Z; Z.next=N. Ansonsten Zustand unverändert
  }
}
```

In den Bedingungen steht das Semikolon für das logische UND. Die Variabeln im mathematischen Sinn werden kursiv und groß geschrieben. Ihre Werte sind frei aber fest gewählt. Das heißt, dass die Variablen in Vor- und Nachbedingung jeweils denselben Wert haben.

Exkurs: Generische Typen

In diesem Kurs der objektorientierten Programm-Konstruktion wird ein grundlegendes Merkmal der objektorientierten Programmierung ausgiebig genutzt: Der Polymorphismus von Referenzen. In linearen Listen lassen sich so Objekte ganz unterschiedlichen Typs miteinander verketten.

Das erfordert einige Vorsicht beim Aufruf von Attributen und Methoden der verketteten Objekte. Um an die Attribute und Methoden solcherart verketteten Objekte heranzukommen, sind normalerweise explizite Typwandlungen (Type Cast) erforderlich. Manche Programmierer halten dies für eine Schwäche der objektorientierten Programmierung.

Meine Erfahrungen mit dieser Technik sind positiv: Ich kann mich an keinen Programmierfehler erinnern, der auf diese Technik der Typwandlung zurückzuführen wäre. Außerdem gibt es in Java und anderen modernen objektorientierten Sprachen die Möglichkeit, den Typ eines Objekts abzufragen und so die Typwandlung zusätzlich abzusichern.

Andere Programmierschulen legen Wert auf eine immanente Typprüfung bei der Verwendung allgemeiner Datenstrukturen wie Listen und Arrays. Dafür wurden die generischen Typen eingeführt.

Die Klasse ArrayList

Ein solcher generischer Typ ist die Klasse ArrayList. In dem folgenden Programmausschnitt habe ich diesen Datentyp einmal verwendet, um Daten von einem Hintergrundspeicher zu puffern. Da ArrayList nach Bedarf dynamisch wächst, muss der Puffer nicht von vornherein auf eine bestimmte Größe ausgelegt werden.

```
    public  RandomVariable (String name) {
      InOut.openIn(name);
      InOut.nextToken();
      a=InOut.getNumber();//System.out.print("\na= "+a);
      InOut.nextToken();
      b=InOut.getNumber();//System.out.print("\nb= "+b);
      ArrayList<Double> al= new ArrayList<Double>();
      InOut.nextToken();
      while (!InOut.endOfFile()&&al.add(new Double(InOut.getNumber())))
          InOut.nextToken();
      p=new double[n=al.size()];
      for(int i=0; i<n; i++) p[i]=((Double)al.get(i)).doubleValue();
      for (int i=1; i<n; i++) p[i]+=p[i-1];
      for (int i=0; i<n; i++) p[i]/=p[n-1];
      d=(b-a)/n;
      InOut.closeIn();
}
```

Übungen

3.1 *Arbeiten mit BlueJ:* Machen Sie sich mit der interaktiven Entwicklungsumgebung BlueJ vertraut. Legen Sie das Projekt NumberList an und experimentieren Sie mit dem Programm.

3.2 *Eine erste Version der Link-Klasse.* Die Basiselemente der Liste sind Objekte vom Typ Number. Die Klasse Number umfasst sowohl Attribute und Methoden, die allein der Verkettung dienen (next, in()) als auch solche, die eher der Anwendung zuzurechnen sind (x, write). Will man Objekte für andere Zwecke miteinander verketten, müssen die Verkettungsmechanismen neu geschrieben werden. Im Sinne der Wiederverwendung von Software lösen wir die reinen Verkettungsmechanismen aus den „Anwendungsklassen“ heraus und definieren eine neue Superklasse Link, die alles zur Verkettung nötige beinhaltet aber nichts was darüber hinausgeht.

Hinweis: Hier wird es notwendig, *Typumwandlungen* (Type Casts) durchzuführen. Das geht in Java ähnlich wie in C. Hier kommt hinzu, dass man im Falle, dass eine Referenz auf eine Unterklasse ihres statischen Typs verweist, diese Unterklasse und ihre Attribute und Methoden durch Typumwandlung „sichtbar machen“ kann. Zum Beispiel sei p eine Referenz vom Typ Link. Diese Referenz möge momentan auf ein Objekt vom Typ Number zeigen. Da die write-Methode aber erst in der Unterklasse Number deklariert ist und nicht bereits in Link, wäre der Aufruf p.write() nicht erfolgreich. Hier hilft die Typumwandlung:

((Number)p).write(). Ob eine Typumwandlung zulässig ist, können Sie vorab mit dem Operator instanceof prüfen. Einzelheiten und genaue Definitionen finden Sie in der Java Sprachbeschreibung.

3.3 Ein weiteres einführendes Beispiel in Java ist das Tutorial Lehrgangsverwaltung. Dort ist neben einer Realisierung in der Programmiersprache Java auch eine in der Programmiersprache C angegeben. Gerade in dieser Gegenüberstellung wird deutlich, was Polymorphismus heißt und wie er realisiert werden kann. Explorieren Sie die Programiersprache Java auch mit Hilfe dieses Tutorials. Führen Sie weitere Klassen und Funktionen (Methoden) ein.

3.4 Überarbeiten Sie Ihr Assoziativarray-Programm und nutzen Sie die Eigenschaften der generischen Arrays aus.

4 Module, Klassen und Objekte

Das Modulkonzept

Ein *Modul*[1] ist eine logisch und funktional überschaubare, in sich abgeschlossene und separat compilierbare Programmeinheit, die Einzelheiten der Realisierung vor den Anwendungsmoduln verbirgt und diesen eine klar definierte Schnittstelle zur Kommunikation anbietet. Die Lieferanten-Kundenbeziehung eines Programmsystems lässt sich über einen Modulbaum darstellen. Jeder Kunde sollte möglichst nur wenige Lieferanten nutzen. Auch das Hauptprogramm nutzt Module. Genau genommen macht heute ein Hauptprogramm nicht mehr, als das Zusammenspiel der Module auf der höchsten Ebene zu koordinieren.

Ein vorübersetztes Modul wollen wir auch *Lieferantenmodul* nennen. Demgegenüber schreibt der Anwender ein Kundenprogramm bzw. ein *Kundenmodul*. Dem *Geheimnisprinzip* (information hiding) folgend, verbirgt das Kundenmodul die Implementierungsdetails vor dem Anwender. Nur die Schnittstelle ist sichtbar.

Die Kompilierungseinheit (das Modul)

CompilationUnit (Kompilierungseinheit, Modul) ist das Startsymbol der Java-Syntax. Es wird durch die folgenden Produktionsregeln definiert.

CompilationUnit =
[*PackageDeclaration*] [*ImportDeclarations*] [*TypeDeclarations*]

ImportDeclarations = *ImportDeclaration* { *ImportDeclaration* }

TypeDeclarations = *TypeDeclaration* { *TypeDeclaration* }

Durch eine ImportDeclaration werden Datentypen aus anderen Packages bekannt gemacht. Sie sind dann allein durch ihren einfachen Namen ansprechbar. Die ImportDeklaration sieht so aus:

ImportDeclaration = **import** *Name* [**.***] **;**

Hierin ist Name ein vollständig qualifizierter Name. Das sind die durch Punkte voneinander getrennten Package-Namen einer Package Hierarchie. Unter DOS und Windows kann man ein Package als ein Verzeichnis (Ordner) oder eine Datei auffassen. Unter DOS und Windows wird als Trennungssymbol für Verzeichnisse und Dateien der Rückwärtsschrägstrich (Backslash) anstelle des Punkts genommen.

Als letztes ist das Package benannt, in dem der Datentyp steht. Durch einen weiteren Punkt abgetrennt folgt der einfache Name des Datentyps oder aber ein Sternchen. Letzteres sorgt dafür, dass sämtliche öffentlichen Datentypen (public) des Packages bekannt gemacht werden. In den Typ-Deklarationen steht das eigentliche Programm.

Für den nur mit imperativer Programmierung Vertrauten entsteht an diesem Punkt die Hauptschwierigkeit. Für ihn legen die Datentypen fest, welche Werte eine Variable annehmen kann und was für eine Rolle diese Variablen in Ausdrücken spielen können. Der Algorithmus eines Programms besteht aus einer Folge von Anweisungen (Operationen), die auf den Variablen

---

[1] Wir verwenden den Begriff *Modul* im Sinne von Baueinheit und sagen „das Modul“.

operieren. Datentypen und Operationen sind fein säuberlich getrennt. Variablen spielen eine passive und die Operationen (Anweisungen, Funktionen, Prozeduren) eine aktive Rolle.

Das ist in der objektorientierten Programmierung anders: Die Operationen gehören zu den Datentypen. Variablen haben also grundsätzlich nicht nur passive Attribute (z.B. die Felder von Records bzw. Strukturen), sondern auch noch Methoden - das sind die Operationen, die Attribute ändern können. In Java gibt es überhaupt keine Operationen mehr, die nicht in einen Datentyp eingebettet sind. Solche benutzerdefinierten Datentypen, die Attribute und Methoden beinhalten, heißen Klassen.

TypeDeclaration = *ClassOrInterfaceDeclaration* | **;**

ClassOrInterfaceDeclaration = *ClassDeclaration* | *InterfaceDeclaration*

Uns interessiert vorerst nur die Klassen-Deklaration. Das Programm EinAus.java enthält nur eine einzige Klassendeklaration, die Hauptklasse EinAus. Details zur Grammatik und zur Bedeutung von Klassen folgen im nächsten Abschnitt.

Typ-Deklarationen

Die Syntax der Typdeklaration startet so:

TypeDeclaration = *ClassDeclaration* | *InterfaceDeclaration* | **;**

Es ist üblich, für jede Klasse oder jedes Interface eine eigene Kompilationseinheit (Modul) vorzusehen.

Eine Klassen-Deklaration spezifiziert einen neuen Referenz-Typ. Referenz-Typen haben keinen Namen für den von ihnen eingenommenen Speicherplatz; sondern es existiert nur eine Referenz auf diesen Speicherplatz.

ClassDeclaration =
[*ClassModifiers*] **class** *Identifier* [*Super*] [*Interfaces*] *ClassBody*

Wenn eine Klasse in einem bezeichneten Package mit dem vollständig qualifizierten Namen *P* deklariert worden ist, dann hat die Klasse den voll qualifizierten Namen *P*.*Identifier*. Wenn die Klasse in einem unbenannten Package enthalten ist, dann hat sie den vollständig qualifizierten Namen *Identifier*.

Im Beispiel

```
class Point { int x, y; }
```

ist die Klasse Point in einer Kompilierungseinheit ohne package-Statement enthalten; deshalb ist Point ihr vollständig qualifizierter Name. Dagegen ist im Beispiel

```
package vista;
class Point { int x, y; }
```

vista.Point der vollständig qualifzierte Name der Klasse Point.

Klassen-Modifikatoren

ClassModifiers = *ClassModifier* { *ClassModifier* }

ClassModifier = **public** | **abstract** | **final**

Der Zugangs-Modifikator public macht die Klasse auch außerhalb des Packages, in dem sie deklariert ist, zugänglich. Jede Datei darf nur eine als public deklarierte Klasse enthalten. Datei- und Klassenname müssen übereinstimmen. Ein und derselbe Modifizierer darf nur einmal in einer Klassen-Deklaration auftreten. Klassen-Modifikatoren sollten, falls überhaupt, in der obigen Reihenfolge in Klassendeklarationen auftreten.

Als abstract deklarierte Klassen sind unvollständig, d. h.: Die Klasse kann abstrakte Funktionen enthalten, das sind Funktionen, die zwar deklariert, aber noch nicht definiert sind. Die Definitionen müssen dann in einer Unterklasse der Klasse nachgeholt werden. Demgegenüber haben als final deklarierte Klassen keine Unterklassen (fertige Klassen).

Klassenkörper und Member-Deklarationen

ClassBody = **{** [*ClassBodyDeclarations*] **}**

ClassBodyDeclarations =
    ClassBodyDeclaration { *ClassBodyDeclaration* }

ClassBodyDeclaration = *ClassMemberDeclaration* | *StaticInitializer* |
    ConstructorDeclaration

ClassMemberDeclaration = *FieldDeclaration* | *MethodDeclaration* |
    ClassDeclaration | *InterfaceDeclaration*

StaticInitializer = **static** *Block*

Der Bereich eines geerbten oder deklarierten Member-Namens ist der gesamte Körper der Klassen-Deklaration.

Feld-Deklarationen

FieldDeclaration =
    [*FieldModifiers*] *Type VariableDeclarators* **;**

VariableDeclarators =
    VariableDeclarator [**,** *VariableDeclarator*]

VariableDeclarator =
    VariableDeclaratorId [**=** *VariableInitializer*]

VariableDeclaratorId = *Identifier* | *VariableDeclaratorId* **[ ]**

VariableInitializer = *expression* | *ArrayInitializer*

Der Deklaration eines zweidimensionalen Arrays reeller Zahlen mittels float x[][] ist zunächst noch kein Speicher zu geordnet. Soll mit der Deklaration gleich ein zughöriges Objekt erzeugt werden, dann geht das beispielsweise so:

```
float x[][] = new float [5][1024];
```

Übung: Prognostizieren Sie, was das folgende Programm auf dem Bildschirm ausgibt.

```
class a {
  public static void main(String v[]) {
    float x[][] = new float [5][1024];
    x[3] = new float [1111];
    x[3][1110]= (float)2.2;
    for (int i=0; i<5; i++)
      System.out.print("\n! x["+i+"].length = "+ x[i].length);
    System.out.print("\n! x[3][1109]= " + x[3][1109]);
    System.out.print("\n! x[3][1110]= " + x[3][1110]);
  }
}
```

Der Gültigkeitsbereich eines Feldnamens erstreckt sich auf den gesamten Körper der Klassendeklaration, in der er deklariert wird. Eine Methode und ein Feld dürfen denselben Namen haben. Eine Felddeklaration *verdeckt* alle Felddeklarationen mit denselben Namen in Superklassen und Superinterfaces dieser Klasse. Das gilt auch dann, wenn die Felder unterschiedliche Typen haben.

Eine Klasse *erbt* von ihrer direkten Superklasse und von ihrem direkten Superinterface alle nicht verdeckten Felder. Verdeckte Felder werden, falls sie als static deklariert wurden, über qualifizierte Namen erreicht, oder mit Hilfe des Schlüsselwortes super, oder mittels einer Typumwandlung in einen Superklassen-Typ.

Typen und Werte

Type = *PrimitiveType* | *ReferenceType*

PrimitiveType = *NumericType* | **boolean**

NumericType = *IntegralType* | *FloatingPointType*

IntegralType = **byte** | **short** | **int** | **long** | **char**

FloatingPointType = **float** | **double**

ReferenceType = *ClassOrInterfaceType* | *ArrayType*

ClassOrInterfaceType = *ClassType* | *InterfaceType*

ClassType = *TypeName*

InterfaceType = *TypeName*

ArrayType = *Type* **[ ]**

Feld-Modifikatoren

FieldModifiers = *FieldModifier* [*FieldModifier*]

FieldModifier =
public | **protected** | **private** | **final** | **static** | **...**

Regelung des Zugriffs

Für die Realisierung des Geheimnisprinzips (information hiding) mit den Feld-Modifikatoren gelten die folgenden Regeln.

Auf Member (field oder method) einer Referenz-Klasse (class, interface oder array) oder Konstruktoren eines Klassentyps kann zugegriffen werden, wenn auf den Typ zugegriffen werden kann und wenn der Zugriff auf das Member möglich ist. Der Zugriff auf Member ist folgendermaßen geregelt.

Falls das Member oder der Konstruktor als public deklariert ist, ist der Zugriff erlaubt. Alle Member eines Interfaces sind implizit public.

Wenn ein Member als protected deklariert ist, dann ist Zugriff innerhalb eines Packages und auch aus Subklassen heraus erlaubt.

Falls ein Member oder Konstruktor als private deklariert ist, dann ist der Zugriff auf die Klasse beschränkt, in der die Deklaration steht.

Ohne weitere Spezifikation des Zugriffs (Default-Zugriff) ist der Zugriff nur innerhalb des Packages, in dem die Typdefinition steht, erlaubt..

Statische Felder

Als static deklarierte Felder heißen auch Klassenvariablen. Sie sind allen Objekten der Klasse gemeinsam. Alle anderen Felder sind Objektvarariablen und werden mit dem Objekt erzeugt.

Übung: Betrachten Sie das folgende Beispielprogramm

```
class Point {
  int x, y, useCount;
  Point(int x, int y) { this.x = x; this.y = y; }
  final static Point origin = new Point(0, 0);
}

class Test {
  public static void main(String[] args) {
    Point p = new Point(1,1);
    Point q = new Point(2,2);
    p.x = 3; p.y = 3; p.useCount++; p.origin.useCount++;
    System.out.println("(" + q.x + "," + q.y + ")");
    System.out.println(q.useCount);
    System.out.println(q.origin == Point.origin);
    System.out.println(q.origin.useCount);
  }
}
```

Ein Aufruf von main druckt folgendes aus:

```
(2,2)
0
true
1
```

Erläutern Sie das Ergebnis. Auf Klassenvariablen gibt es zwei Zugriffsmöglichkeiten. Welche sind das?

Finale Felder

Konstante werden in Java als final deklariert. Solche Felder müssen initialisiert sein.

Methoden-Deklarationen

Eine *Methode* deklariert (definiert) ausführbaren Code, der aufgerufen werden kann. Beim Aufruf kann eine feste Anzahl von Werten als Argumente mitgegeben werden. Methoden entsprechen im Wesentlichen den C-Funktionen. Ein wesentlicher Unterschied besteht allerdings: In Java kann man sich darauf verlassen, dass beim Aufruf die aktuellen Parameter von links nach rechts ausgewertet werden.

MethodDeclaration = MethodHeader MethodBody

MethodHeader = [*MethodModifiers*] *ResultType MethodDeclarator*

[*Throws*]

ResultType = Type | **void**

MethodDeclarator = Identifer **(** [*FormalParameterList*] **)**

MethodBody = Block **|;**

Die Methoden-Modifizierer (*MethodModifiers*) und die *Throws*-Klausel werden in folgenden Abschnitten näher erläutert. Der Methodenkörper MethodBody besteht aus einer zusammengesetzten Anweisung (siehe C-Syntax) oder – bei abstrakten Methoden – aus einem Semikolon. Die Klasse, eine ihrer Methode und eines ihrer Feld dürfen denselben Namen haben. Aus dem Zusammenhang ist immer erkennbar, was gemeint ist.

Der Methodenname `main` ist für die Methode reserviert, mit der das Programm starten soll.

Beispiel: Die Verwendung von Typbezeichnern in Deklarationen und Ausdrücken.

```
import java.util.Random;

class MiscMath {
 int divisor;
 MiscMath(int divisor) {this.divisor = divisor;}
 float ratio(long l) {
  try {l /= divisor;}
  catch (Exception e) {
   if (e instanceof ArithmeticException)
    l = Long.MAX_VALUE;
   else l = 0;
  }
  return (float)l;
 }

 double gausser() {
  Random r = new Random();
  double[] val = new double[2];
  val[0] = r.nextGaussian();
  val[1] = r.nextGaussian();
  return (val[0] + val[1]) / 2;
 }
}
```

In diesem Beispiel werden Typangaben benutzt

- beim Import des Typs Random aus dem Package java.util
- zur Deklarationen der Feldbezeichner von Klassen. Zum Beispiel wird divisor als vom Typ int deklariert.

- zur Deklaration der Parameter von Methoden. Hier wird der Parameter l der ratio-Methode als vom Typ long deklariert.
- zur Deklaration des Resultats der ratio-Methode als vom Typ float, und des Resultats der Methode gausser als vom Typ double
- zur Deklaration des Parameters des Konstruktors für die Klasse MiscMath als vom Typ int.
- zur Deklaration der lokalen Variablen r und val der Methode gausser als vom Typ Random bzw. double[] (array of double).
- zur Deklaration des Exception Handler Parameters e der catch Clause als vom Typ Exception.

In den *Ausdrücken* kommen Typbezeichner außerdem an folgenden Stellen vor:

Die lokale Variable r der Methode gausser wird durch einen Objekterzeugungs-Ausdruck initialisiert, der den Typ Random benutzt.

Die lokale Variable val der Method gausser wird durch einen Array-Erzeugungsausdruck initialisiert, der einen Array of double der Länge 2 erzeugt.

Das return-Statement der Methode ratio benutzt eine Typumwandlung in den Typ float.

Der instanceof Operator prüft, ob e zuweisungskompatibel mit dem Typ ArithmeticException ist.

Übungen

4.0 Machen Sie sich mit einem UML-Editor-Programm vertraut, beispielsweise GO. Machen Sie sich vertraut, wie in UML die folgenden Elemente, Eigenschaften und Relationen dargestellt werden:

1. Geheimnisprinzip: private, protected, public
2. Klassenattribute und Klassenmethoden: static
3. Beziehungen zwischen Objekten: Assoziation, Aggregation und Komposition

4.1 Erstellen Sie für das Projekt NumberList ein UML-Diagramm im Design-Modus. Die Zusammenfassung von Number-Objekten in einer Liste ist eine Aggregation.

5 Vererbung

Superklassen und Subklassen

Die optionale extends-Klausel spezifiziert die direkte Superklasse (Vorgänger) der deklarierten Klasse. Die Klasse selbst ist direkte Subklasse (Nachfolger) ihrer Superklasse. Fehlt die Angabe einer Superklasse, dann ist die Klasse java.lang.Object implizit die direkte Superklasse.

Die Klassen bilden einen Baum. Die extends-Klauseln definieren unmittelbar eine Parent-Darstellung dieses Baumes. Die Object-Klasse ist Wurzel des Baums der Klassenhierarchie; sie hat keine Superklasse. B ist eine Subklasse von A, wenn man beim Durchlaufen des Klassenbaums von B in Richtung Wurzel an A vorbeikommt. A ist dementsprechend die Superklasse von B. Eine Klasse kann nicht Subklasse ihrer selbst sein.

Super = **extends** *ClassType*
ClassType = *TypeName*

Eine überschriebene Methoden *name*() kann von den Methoden der Unterklasse mittels Aufruf **super**.*name*() aufgerufen werden. Zu weiteren Verwendungen des Schlüsselworts **super** siehe Gosling/Joy/Steele (1996, S. 165, 322, 324).

Alle Klassen eines Package sind innerhalb des gesamten Packages gültig. Zu den Klassen anderer Packages ist der Zugang möglich, wenn das Host-System den Zugriff zum Package erlaubt und die Klasse als public deklariert ist.

Abstrakte Klassen

Eine abstrakte Klasse ist eine unvollständige Klasse. Nur abstrakte Klassen können abstrakte Methoden enthalten - also Methoden, die zwar deklariert aber noch nicht definiert sind. Eine Klasse enthält in den folgenden Fällen abstrakte Methoden:

- Eine abstrakte Methode wurde explizit deklariert
- Eine abstrakte Methode wurde von einer Superklasse geerbt

Im Beispiel

```
abstract class Point {
  int x = 1, y = 1;
  void move(int dx, int dy) {x += dx; y += dy; alert();}
  abstract void alert();
}
abstract class ColoredPoint extends Point {int color;}
class SimplePoint extends Point {void alert() { }}
```

muss die Klasse Point als abstract deklariert werden, weil sie die Deklaration einer abstrakten Methode namens alert enthält. Die Subklasse ColoredPoint von Point erbt die abstrakte Methode alert. Deshalb muss sie ebenfalls als abstrakt deklariert werden. In der Subklasse SimplePoint von Point wird die Methode alert definiert, deshalb muss sie nicht als abstrakt modifiziert werden.

Das Statement

```
Point p = new Point();
```

führt zu einem Fehler zur Kompilierungszeit, da ein Objekt der Klasse Point nicht erzeugt werden kann, da sie abstrakt ist. Jedoch kann die Point-Variable korrekt mit der Referenz auf eine Subklasse von Point initialisiert werden:

```
Point p = new SimplePoint();
```

Fertige Klassen

Eine Klasse kann als final modifiziert werden, wenn sie komplett ist und Subklassen nicht erwünscht sind.

Interfaces

Interfaces werden wie Klassen deklariert, nur dass sie keine variablen Attribute enthalten (höchstens konstante Felder) und dass alle Methoden implizit abstrakt sind. Das Schlüsselwort abstract entfällt. Siehe Gosling/Joy/Steele (1996), S. 186.

Interfaces sind eingeführt worden, um auf ungefährliche Art und Weise die Nachteile der *Einfachvererbung*, die ja grundlegend für Java ist, zu überwinden. Der Nachteil besteht darin, dass es unmöglich ist, eine Klasse zu definieren, die Eigenschaften und Verhalten mehrerer Superklassen erbt.

Durch diese Einschränkung wird ein berüchtigtes Problem der *Mehrfachvererbung* umgangen. Nehmen wir an, die Klassen B und C sind verschiedene Subklassen einer Klasse A. Die Klasse A möge eine Methode m enthalten, die in den Klassen B und C auf unterschiedliche Art überschrieben sind. Wenn nun eine Klasse D im Zuge der Mehrfachvererbung Subklasse von B und C ist, dann ist unklar, welche Ausprägung der Methode m sie erbt, die von B oder die von C. Dieses Problem ist in der Literatur unter dem Namen *Diamant-Vererbung* (diamond inheritance) bekannt, Bild 6.1.

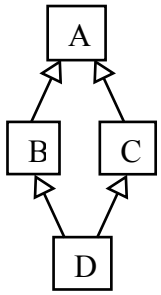


Bild 6.1 Diamant-Vererbung

Interfaces erlauben es, Zugriffsmöglichkeiten zu vererben, die nicht gleichzeitig mit Realisierungen verbunden sind (Arnold/Gosling, 1996, S. 81 f.).

Superinterfaces

Interfaces = **implements** *InterfaceTypeList*
InterfaceTypeList = *InterfaceType* { **,** *InterfaceType* }
InterfaceType = *TypeName*

Im Beispiel

```
public interface Colorable {
 void setColor(int color);
 int getColor();
}

public interface Paintable extends Colorable {
   int MATTE = 0, GLOSSY = 1;
   void setFinish(int finish);
   int getFinish();
}

class Point { int x, y; }
```

```
class ColoredPoint extends Point implements Colorable {
   int color;
   public void setColor(int color) { this.color = color; }
   public int getColor() { return color; }
}

class PaintedPoint extends ColoredPoint implements Paintable
{
   int finish;
   public void setFinish(int finish) {this.finish = finish;}
   public int getFinish() { return finish; }
}
```

bestehen folgende Beziehungen:

Das Interface Paintable ist ein Superinterface der Klasse PaintedPoint.

Das Interface Colorable ist Superinterface der Klasse ColoredPoint und der Klasse PaintedPoint.

Das Interface Paintable ist Subinterface des Interface Colorable, und Colorable ist Superinterface von Paintable.

Eine Klasse kann auf mehrerlei Weise ein Superinterface haben. Im Beispiel hat die Klasse PaintedPoint das Superinterface Colorable weil es Superinterface von ColoredPoint ist, aber auch, weil es Superinterface von Paintable ist.

Abgesehen von als abstract deklarierten Klassen müssen alle Methoden in direkten Superinterfaces in einer Deklaration oder durch eine existierende und geerbte Methode der direkten Superklasse definiert werden.

Es ist erlaubt, in einer einzigen Methodendeklaration eine Methode von mehr als einem Superinterface zu definieren. Zum Beispiel passt im folgenden Code die Methode getNumberOfScales in der Klasse Tuna zu den Methoden in den Interfaces Fish und Piano. Die Deklaration definiert beide Methoden.

```
interface Fish { int getNumberOfScales(); }
interface Piano { int getNumberOfScales(); }
class Tuna implements Fish, Piano {
   //You can tune a piano, but can you tuna fish?
  int getNumberOfScales() { return 91; }
}
```

Methoden-Modifizierer

MethodModifiers = MethodModifier {*MethodModifier*}

MethodModifier =
public | **protected** | **private** | **abstract** | **static** | **final** |**...**

Ob auf ein Package zugegriffen werden kann, wird durch das Host-System bestimmt. Die Zugriffs-Modifizierer public, protected und private regeln das Zugriffsrecht auf Klassen, Interfaces, Arrays, Felder, Methoden und Konstruktoren.

Falls eine Klasse oder ein Interface als public deklariert wurde, ist der Zugriff für jeden Java-Code möglich, der Zugriff zum Package hat, in dem es deklariert ist. Ist eine Klasse oder ein

Interface nicht public deklariert worden, dann kann es nur innerhalb des Packages verwendet werden, in dem es deklariert ist.

Auf ein Member (Feld oder Methode) eines Referenz-Typs (Klasse, Interface oder Array) oder einen Konstruktor eines Klassen-Typs kann nur dann zugegriffen werden, wenn auf den Typ zugegriffen werden kann und auch ein Zugriff auf das Member oder den Konstruktor erlaubt ist.

Der generelle Zugriff wird durch das Schlüsselwort public erlaubt; Member von Interfaces sind implizit public. Ansonsten ist der Zugriff folgendermaßen geregelt:

Falls das Member oder der Konstruktor als protected deklariert wurde, ist Zugriff nur erlaubt, wenn der Zugriff innerhalb des Packages geschieht oder innerhalb einer Subklasse.

Falls das Member oder der Konstruktor als private deklariert wurde, ist Zugriff nur innerhalb der Klasse selbst erlaubt. Auch Subklassen sind vom Zugriff ausgeschlossen.

Fehlen die Zugriffsmodifizierer, dann ist - per Default - Zugriff nur innerhalb des Packages erlaubt.

Abstrakte Methoden

Methoden können nur in abstrakten Klassen als abstract deklarierte werden. Die Deklaration einer abstrakten Methode beinhalten Namen, Anzahl und Typ der Parameter, Typ des Rückgabewerts und - gegebenenfalls - die throws-Klausel, nicht jedoch die Definition der Funktion mittels eines Funktionskörpers.

Die Defnition der Funktion muss spätestens in einer nicht abstrakten Subklasse nachgeholt werden.

Eine abstrakte Klasse kann eine abstrakte Methode durch eine weitere abstrakte Methoden-Deklaration überschreiben (override).

```
class BufferEmpty extends Exception {
   BufferEmpty() { super(); }
   BufferEmpty(String s) { super(s); }
}
class BufferError extends Exception {
   BufferError() { super(); }
   BufferError(String s) { super(s); }
}
public interface Buffer {
 char get() throws BufferEmpty, BufferError;
}
public abstract class InfiniteBuffer implements Buffer {
   abstract char get() throws BufferError;
}
```

Die überschreibende Deklaration der Methode get in der Klasse InfiniteBuffer legt fest, dass die Methode get in jeder Subklasse von InfiniteBuffer nie eine BufferEmpty-Exception auslöst (throws), vermutlich weil sie die Daten des Puffers erzeugt und ihr deshalb nie die Daten ausgehen.

Statische Methoden

Als static deklarierte Methoden heißen Klassenmethoden (class method). Klassenmethoden können sowohl über eine Referenz als auch über den Klassennamen aufgerufen werden. Nicht

als static deklarierte Methoden heißen Objektmethoden (instance method, auch: non-static-Methode). Objektmethoden werden grundsätzlich über eine Referenz, also bezüglich eines Objekts aufgerufen. Dieses Objekt wird das Bezugsobjekt, auf das sich die Schlüsselwörter this und super innerhalb des Methodenkörpers beziehen.

Fertige Methoden

Will man verhindern, dass eine Methode überschrieben oder verdeckt wird, kann man sie als fertig (final) deklarieren. Versuche, diese Methode zu überschreiben oder zu verdecken werden zur Kompilierungszeit aufgedeckt und gemeldet.

Konstruktoren

Ein Konstruktor wird bei der Erzeugung des Objektes einer Klasse benutzt.

ConstructorDeclaration = [*ConstructorModifiers*]
ConstructorDeclarator [*Throws*]
ConstructorBody

ConstructorDeclarator = *SimpleTypeName* **(** [*FormalParameterList*] **)**

ConstructorModifiers = **public** | **protected** | **private**

SimpleTypeName muss der Name der Klasse sein, in der die Konstruktordeklaration steht. Ansonsten sieht die Konstruktordeklaration wie eine Methodendeklaration ohne Resultattyp aus. Ein einfaches Beispiel ist

```
class Point {
  int x, y;
  Point(int x, int y) { this.x = x; this.y = y; }
}
```

Konstruktoren werden u. a. durch den Objekterzeugungsausdruck oder durch direkten Aufruf durch andere Konstruktoren aufgerufen. Konstruktoren sind keine Member. Sie werden nicht vererbt und können deshalb nie verdeckt oder überschrieben werden.

Falls für eine Klasse kein Konstruktor definiert wird, dann wird ein Default-Konstruktor implizit bereitgestellt. Er enthält keine Parameter und besteht nur im Aufruf des (Default-)Konstruktors der Superklasse (falls es sich nicht um die Objekt-Klasse handelt). Der Aufruf des Konstruktors einer Oberklasse geschieht mit dem Schlüsselwort **super**. Beginnt ein Konstruktor nicht mit einem expliziten Aufruf eines Konstruktors derselben Klasse oder einer Superklasse, dann erfolgt ein impliziter Aufruf eines Superklassen-Konstruktors (Gosling/Joy/Steele,1996, Abschitt 12.5, S. 180, 228).

Ein Objekterzeugungsausdruck erzeugt ein Objekt der bezeichneten Klasse und liefert einen Zeiger auf das erzeugte Objekt.

ClassInstanceCreationExpression =
new *ClassType* **(** [*ArgumentList*] **)**

ArgumentList = *expression* {**,** *expression* }

Der Klassen-Typ *(ClassType)* darf nicht abstrakt sein. Die Argumente der Argument-Liste werden dem passenden Konstruktor mitgegeben. Der Objekterzeugungsausdruck erscheint beispielsweise in der folgenden Deklaration:

```
ColoredPoint cp = new ColoredPoint();
```

Hier wird ein Objekt der Klasse ColoredPoint erzeugt und initialisiert. Der Objekterzeugungsausdruck liefert eine Referenz auf das neu erzeugte Objekt. Die Felder werden durch den zur Parameterliste passenden Konstruktur initialisiert. Sieht der Konstruktor für ein Feld keine Initialisierung vor, wird dieses auf den Default-Wert initialisiert.

Überschreiben, Verdecken, Überladen

Hier werden nur die grundlegenden Möglichkeiten für das Überschreiben, Verdecken und Überladen angesprochen.

Überschreiben (Overriding): Eine Objektmethode *m* der Klasse *C überschreibt* eine Objektmethode *n* der Klasse *A*, wenn

1. *C* Subklasse von *A* ist,
2. die Namen und Argumenttypen (Signaturen) von *m* und *n* übereinstimmen, und wenn
3. *n* als public, protected oder mit Default-Zugang im selben Package deklariert ist.

Überschriebene Methoden können mit dem Schlüsselwort `super` erreicht werden.

Verdecken (Hiding): Jedes Feld einer Klasse verdeckt gleichnamige Felder in Superklassen und Superinterfaces. Bei Methoden beschränkt sich das Verdecken auf statische Methoden gleicher Signatur. Verdeckte Methoden oder Felder sind durch den qualifizierten Namen, über das Schlüsselwort `super` oder durch Typumwandlung zugänglich.

Überladen (Overloading): Wenn zwei Methoden einer Klasse, seien sie in der Klasse erklärt oder ererbt, zwar denselben Namen haben, sich aber in der Parameterliste unterscheiden, dann nennt man den Methodennamen überladen.

Beispiel:

```
public class A {
  static int z;
  static int z() {return z;}
  int x;
  int x() {return x;}
}

public class B extends A {
  static int z;              //hiding - Verdecken
  static int z(){return z;}  //hiding - Verdecken
  int x;                     //hiding - Verdecken
  int x() {return x;}        //overriding - Ueberschreiben
  int x(boolean hidden) {    //overloading - Ueberladen
    if (hidden) return super.x; else return x;
  }
  int xx() {return super.x();}
}
```

Programmierung im Detail: Blöcke und Anweisungen

Bisher sind wir mit dem C-Wissen, was den Aufbau von Blöcken angeht, zurechtgekommen. Jetzt wird die genaue Beschreibung der Syntax der Blöcke nachgeholt. Der folgende Ausschnitt aus der Java Syntax spart markierte Anweisung, lokalen Klassen und ein paar weitere Spezialitäten aus.

Block = **{** { *BlockStatement* } **}**

BlockStatement =
LocalVariableDeclarationStatement | *Statement*

L*ocalVariableDeclarationStatement* = [**final**] *Type VariableDeclarators* **;**

Statement = *Block* | **if** *ParExpression Statement* [**else** *Statement*] |
for (*ForInitOpt* **;** [*Expression*] **;** *ForUpdateOpt* **)** *Statement* |
while *ParExpression Statement* | **do** *Statement* **while** *ParExpression* **;** |
try *Block* **catch (** *FormalParameter* **)** *Block* { **catch (** *FormalParameter* **)** *Block* } |
switch *ParExpression* **{** *SwitchBlockStatementGroups* **}** |
synchronized *ParExpression Block* |
return [*Expression*] **;** | **throw** *Expression* **; | break ; | continue ; |**
[*Expression*] **;**

SwitchBlockStatementGroups = { *SwitchBlockStatementGroup* }

SwitchBlockStatementGroup = *SwitchLabel* { *BlockStatement* }

SwitchLabel = **case** *ConstantExpression* **: | default :**

ForInit = *StatementExpression MoreStatementExpressions* |
[**final**] *Type VariableDeclarators*

ForUpdate = *StatementExpression MoreStatementExpressions*

StatementExpression = *Expression*

MoreStatementExpressions = { **,** *Expression* }

Der Ausduck (expression) in If-, While- und Do-Statements muss vom Typ boolean sein.

Programmierung im Detail: Exceptions

Die throws-Klausel wird zur Deklaration einer überprüften Ausnahmebehandlung benutzt.

Throws = **throws** *ClassTypeList*
ClassTypeList = *ClassType* {**,** *ClassType* }

ClassType muss eine Subklasse von Throwable oder Throwable selbst sein. Ist eine Exception deklariert, dann muss sie beim Methoden- oder Konstruktorenaufruf auch behandelt werden. Bildlich gesprochen, muss das die Fehlermeldung tragende „geworfene“ Objekt „aufgefangen“ werden. Die Throw-Anweisung wirft die Exception aus und das try-statement ist dafür da, die Exception aufzufangen:

ThrowStatement = **throw** *expression* **;**

Der Ausdruck im Throw-Statement muss zuweisungsverträglich mit dem Typ Throwable sein. Das einfachste Try-Statement sieht so aus

TryStatement = **try** *Block Catches*

Catches = *CatchClause* { *CatchClause* }

CatchClause = **catch (** *FormalParameter* **)** *Block*

FormalParameter = *Type VariableDeclaratorId*

Throw-Statements treten innerhalb des Blocks des Try-Statements auf. Eine Catch-Klausel hat genau einen Paramter (Exception-Parameter). Er gehört zur Klasse Throwable (oder einer Subklasse davon). Im nachfolgenden Block der Catch-Klausel kann auf den Parameter Bezug genommen werden.

Übungen

5.1 Beantworten Sie die folgenden Fragen:

- Warum können Methoden nicht zugleich statisch und abstrakt sein?
- Warum sind private Methoden implizit auch fertige Methoden (final).
- Warum können fertige Methoden (final) nicht abstrakt sein?

5.2 *Das Projekt Link2.* Erweitern Sie den Anwendungsbereich der `Link`-Klasse. Für den Test konstruieren Sie eine passende Erweiterung Klasse `NumberList`.

Das Link-Modul soll folgende Anforderungen erfüllen:

- Das Attribut `next` soll nur noch durch Methoden der Klasse `Link` verändert werden können. Dazu wird es „privatisiert“. Der Wert von `next` wird über die öffentliche Methode `Link next()` bekannt gemacht.
- Die Methode `in` wird als öffentlich deklariert.
- Es wird eine öffentliche Methode `void add(Link p)` eingeführt. Sie fügt das Element `p` hinten an die Liste an. Wenn `p` selbst Anfang einer Liste ist, wird die gesamte Liste an die von `this` angeführte Liste angefügt.
- Es werden die öffentlichen Methoden Link `out()` und Link `bisect()` eingeführt. Die `out`-Methode entfernt das auf `this` folgende Element aus der Liste, setzt dessen `next`-Referenz auf null und liefert die Referenz auf das Element. Die `bisect`-Methode schneidet den auf `this` folgenden Rest der Liste ab und liefert die Referenz auf das erste Element dieses Rests.
- Es werden die öffentlichen Methoden `int cardinal()`, `void clear()` und `boolean empty()` eingeführt; `cardinal` liefert die Anzahl der Listenelemente, `clear` setzt `next` auf null und `boolean empty` ist genau dann gleich `true`, wenn die Liste leer ist.

5.3 *Huffman-Code.* Erstellen Sie ein Programm, das eine Symbolstatistik über Tastatur entgegennimmt und das anschließend den Huffman-Code für die Symbole ausgibt. Bei den Symbo-

len können sie sich auf die Zeichen des Ascii-Codes beschränken. Nutzen Sie für die interne Darstellung des Codierbaumes die Parent-Darstellung. Arbeiten Sie streng nach dem Vorgehensmodell.

5.4 *Führungskurve (optional):* Die Punkte einer zweidimensionalen Bahnkurve sind programmintern geeignet zu repräsentieren. Die Eingabe geschieht mittels Bildschirm und Maus. Die Befehle:

Markieren eines Punktes der Bahnkurve: <Click> auf den unmarkierten Punkt

Aufheben der Markierung: <Click> auf den markierten Punkt

Eingabe eines Punktes: <Ins+Click> auf die freie Fläche

Falls vorher ein Punkt markiert war, wird der neue Punkt hinter dem markierten Punkt in die Bahnkurve eingefügt. Andernfalls gibt es zwei Möglichkeiten: Falls noch kein Punkt vorhanden war, entsteht ein Anfangspunkt, ansonsten erfolgt keinerlei Reaktion. Der Anfangspunkt wird durch einen leeren Kreis dargestellt. Die übrigen Punkte sind schwarze Kreisflächen.

6 Abstrakte Datentypen (ADT)

Benannte Packages

Eine *Package Deklaration* innerhalb einer Kompilierungseinheit definiert den Namen des Packages zu dem die Kompilierungseinheit gehört.

PackageDeclaration = **package** *PackageName* **;**

Der Package-Name in einer Package Deklaration muss der vollständig qualifizierte Name des Package sein. Falls ein Typ-Name *T* in einer Kompilierungseinheit eines Package mit dem vollständig qualifizierten Namen *P* deklariert ist, dann ist der vollständig qualifizierte Name des Typs gleich *P*.*T*. Im Beispiel

```
package wnj.points;
class Point { int x, y; }
```

ist also wnj.points.Point der vollständig qualifizierte Name der Klasse Point.

Packages dienen dazu, den Gültigkeitsbereich von Klassen-, Attribut- und Methodennamen zu beschränken. Dadurch lassen sich Namenskonflikte vermeiden.

Die Unified Modeling Language (UML)

Anhand des Klassendiagramms für das Lehrprogramm NumberList (Bild 6.1) werden einige der grafischen Elemente der Unified Modeling Language erklärt.

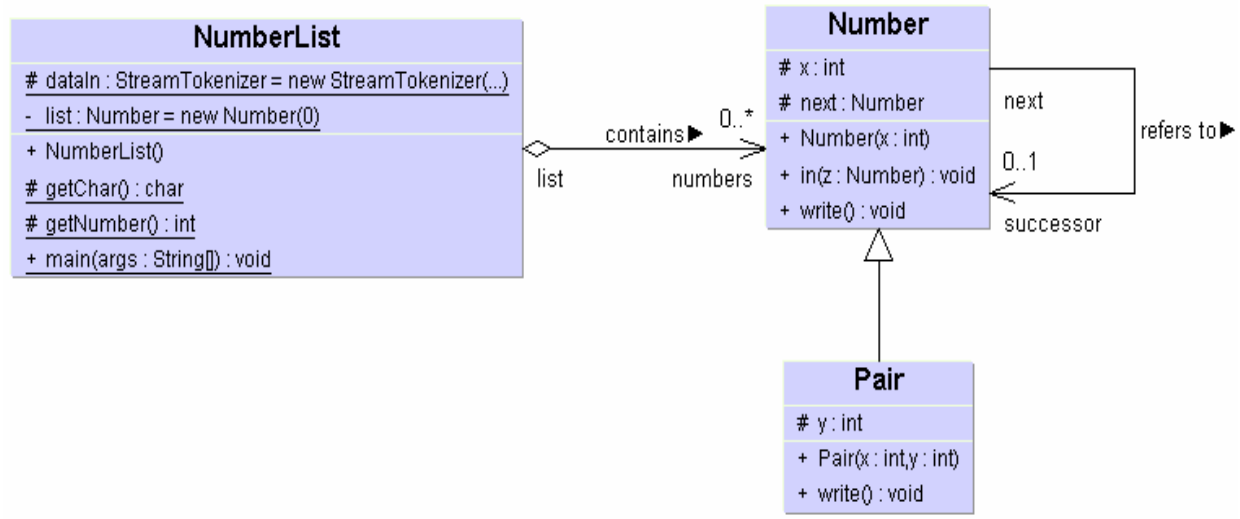


Bild 6.1 Das Modell des Programms NumberList

Klassenattribute und -methoden erscheinen mit Unterstreichung. Die Zugriffsrechte werden mit den vorangestellten Symbolen +, - und # für public, private und protected gekennzeichnet. Für die Vererbungsbeziehung steht das offene Dreieck. Pfeile kennzeichnen die Hat-Beziehung. Eine davon ist die Aggregation. Sie wird durch die offene Raute am Pfeilende gekennzeichnet. Ist die Raute ausgefüllt, handelt es sich um eine Komposition. An den Enden der Pfeile können die *Rollen* angegeben sein, die die Objekte der jeweiligen Klasse spielen und außerdem eine Angabe zur Vielfachheit der Objekte (Kardinalität). 0..* steht dafür, dass

es keines oder aber auch beliebig viele Objekte sind. In der Mitte über den Pfeilen steht der Beziehungsname.

Datenabstraktion und Generalisierung

Den Übergang von der strukturierten zur objektorientierten Programmierung kann man mit einem entsprechenden Übergang in der Mathematik vergleichen: Dort ist es der Übergang von der Arithmetik zur Algebra. Bereits in der Arithmetik gibt es eine grandiose Verallgemeinerung durch die Verwendung mathematischer Variablen. Aber noch weiter geht die Einführung allgemeiner mathematischer Strukturen, mit denen sich die Gemeinsamkeiten beispielsweise des Rechnens mit reellen Zahlen und des Rechnens mit verallgemeinerten Funktionen erfassen lassen. Beispiele für solche Strukturen sind die Gruppe, der Ring, der Körper und der Vektorraum. Sind gewisse mathematische Sätze für eine solche Struktur - unter alleiniger Berufung auf die sie definierenden Axiome - erst einmal bewiesen, hat man die Arbeit zugleich für alle mathematischen Gebilde erledigt, die genau diesen Axiomen genügen. Der Erfolg der Mathematik beruht wesentlich auf dieser Denkökonomie.

Die objektorientierte Programmierung hat ähnlich ökonomische Züge. Bei den abstrakten Datentypen - realisiert durch abstrakte Klassen beispielsweise - geht es um Algorithmen und nicht um mathematische Sätze. Die Algorithmen werden soweit möglich für abstrakte Klassen formuliert.

Dass diese Algorithmen auch in Nachkommen einer Klasse wie gewünscht funktionieren, muss dadurch sichergestellt werden, dass für die Klassen passende *Axiome* aufgestellt werden. Wie der Algebraiker hat auch der Programmierer die Gültigkeit der Axiome für seine Konkretisierungen nachzuweisen.

Als Beispiel für eine abstrakte Klasse dient uns hier die Link-Klasse: Sie ermöglicht den Aufbau einfach verketteter Listen, zwischen deren Elementen eine partielle Ordnung existiert (Bild 6.2).

Die Methoden der Klasse Link dienen der Verwaltung der linearer Listen[1]. Sie haben folgende Aufgaben: q.in(p) fügt das Objekt p unmittelbar hinter q in die lineare Liste ein; q.add(p) fügt das Objekt p ans Ende der mit q startenden Liste ein; q.out() entfernt das auf q folgende Element aus der Liste und liefert als Funktionswert die Referenz auf dieses Objekt.

Durch less wird eine (abstrakte) *partielle Ordnung*[2] festgelegt. Die Ordnungsrelation wird erst in Nachkommen von Link defi-

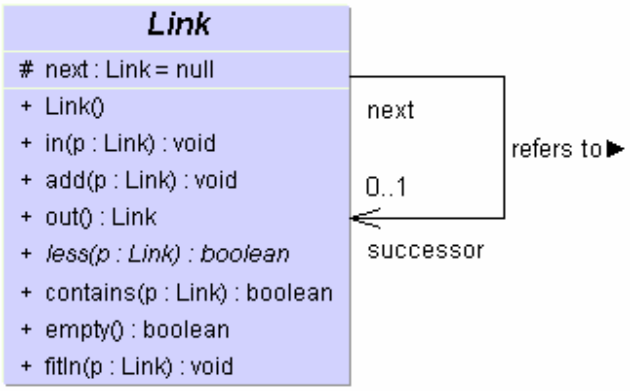


Bild 6.2 OOD-Klassendiagramm der Klasse Link

[1] Die Link-Klasse gestattet auch den Aufbau von Bäumen in Parent-Repräsentation.

[2] Eine Relation < auf einer Menge *M* heißt partielle Ordnung, wenn für alle *x, y* und *z* aus *M* gilt: 1. Aus $x<y$ und $y<z$ folgt $x<z$ und 2. Aus $x<y$ folgt $\neg(y<x)$. Das sind die Gesetze der Transitivität und Asymmetrie. Für lineare Ordnungen gilt darüber hinaus 3. Für alle *x*, *y* aus *M* ist entweder $x<y$ oder $y<x$ oder $x=y$. Anstelle von x< y steht hier x.less(y).

niert. q.empty() gilt genau dann, wenn auf das Element q kein weiteres folgt. Die Methode q.fitIn(p) fügt das Element p so in eine bereits *topologisch sortierte*[3] Liste mit dem Kopf q ein, dass die topologische Sortierung erhalten bleibt.

Da die Klasse eine abstrakte Methode enthält, ist sie selbst ebenfalls abstrakt. Abstrakte Klassen und abstrakte Methoden werden in UML durch Kursivschrift ausgezeichnet. In Java gibt es dafür das Schlüsselwort abstract[4].

Die Schritte der Datenabstraktion und Generalisierung:

1. *Bestimmung der zu generalisierenden Funktionen und Algorithmen:* Ausgewählt werden vorzugsweise Methoden, die immer wiederkehrende Probleme lösen und solche, die einen hohen schöpferischen Wert haben. Eine solche konkrete Methode ist als `public` zu deklarieren. Will man verhindern, dass die Methode in Subklassen überschrieben wird, deklariert man sie darüber hinaus als `final`. Beispiel ist die `fitIn`-Methode der `Link`-Klasse.
2. *Abstrahieren von Operatoren:* Für die Operatoren, deren Realisierung offen und anpassbar gehalten werden soll, führt man (abstrakte) Methoden ein, erzeugt also quasi abstrakte Operatoren. Diese Methoden sind in Subklassen durch Überschreibende zu konkretisieren. Beispiel für einen solchen Operator ist die `less`-Methode der `Link`-Klasse.
3. *Generalisierung der Funktionen und Algorithmen:* In allen konkreten Methoden werden die konkreten Operatoren durch Aufrufe der entsprechenden abstrakten Operatoren ersetzt.
4. *Abstrahieren der Klasse:* Es werden alle Attribute, die nicht für die Realisierung der konkreten Methoden nötig sind, gestrichen. Die verbleibenden Attribute sind vorzugsweise als `private` zu deklarieren.
5. *Formulierung der Axiome:* Identifizierung der *Strukturen* und *allgemeinen Eigenschaften* der Klasse. Formulierung der Axiome, denen die vom Anwender in Subklassen zu präzisierenden Methoden (die Operatoren) gehorchen müssen, wenn die Funktionen richtig funktionieren sollen. Im Beispiel sind die wesentlichen Axiome die der Transitivität und der Asymmetrie der partiellen Ordnung.

Die so entstehende (abstrakte) Klasse zusammen mit den Axiomen ist die Implementierung eines *abstrakten Datentyps (ADT)*. Konkrete Datentypen werden als Subklassen diese Klasse realisiert. Die abstrakte Klasse kann man auch als *Lieferantenmodul* auffassen und die daraus abgeleitet konkrete Klasse als *Kundenmodul.* Für diese Art der Kooperation zwischen Lieferanten und Kunden verwendet Bertrand Meyer (1988) den Begriff *Programming by Contract*. Der Begriff des abstrakten Datentyps wurde von John V. Guttag eingeführt (Broy, M.; Denert, E. 2002, S. 442-479).

---

[3] Eine topologische Sortierung liegt vor, wenn eine partielle Ordnung in eine lineare Ordnung eingebettet ist. Im vorliegenden Fall heißt das, dass immer dann, wenn q.less(p) gilt, das Element q in der linearen Liste vor dem Element p stehen muss.

[4] Da es für derartige abstrakte Klassen keine Realisierungen geben kann, ist uns die Möglichkeit genommen, den Kopf einer linearen Liste als Link-Objekt zu definieren. Zwei Auswege bieten sich an: Entweder man definiert speziell für Listenköpfe eine Subklasse von Link mit einer trivialen Implementierung der less-Methode. Oder man übernimmt diese triviale Implementierung in die Link-Klasse und macht dadurch Link doch wieder konkret und damit geeignet für die Bildung von Listenköpfen.

Übungen

6.1 *Invariante.* Ergänzen Sie Ihre Link-Klasse um eine Methode fitIn. Sie soll Folgendes leisten: Sei q die Referenz auf den Kopf einer topologisch sortierten oder leeren Liste. Durch den Aufruf q.fitIn(p) wird p in diese Liste unter Aufrechterhaltung der partiellen Ordnung eingefügt. Dazu wird das Element p vor dem ersten größeren Element platziert, falls ein solches existiert, und ansonsten am Ende der Liste. Weisen Sie nach, dass „Die Liste q ist topologisch sortiert“ eine *Invariante* der fitIn-Methode ist.

6.2 *Ordnungsrelation.* Formulieren Sie die Klassen Number und Pair als Nachfolger von Link. Führen Sie für die Zahlen und Zahlenpaare des Programms NumberList Ordnungsrelationen durch geeignete Konkretisierungen der less-Methode ein. Für einfache Zahlen soll es die übliche Ordnungsbeziehung sein. Für Zahlenpaare definieren Sie die Methode less im Sinne von „liegt südwestlich“. Also: q.less(p) ist genau dann wahr, wenn q.x<p.x und q.y<p.y. Weisen Sie nach, dass dadurch eine partielle Ordnung der Zahlenpaare definiert wird.

Zwischen einer Zahl und einem Zahlenpaar besteht von vornherein keine Ordnungsbeziehung. Wollen Sie in einer Liste dennoch Zahlen und Paare mischen, muss in der less-Methode herausgefunden werden, ob das Vergleichsobjekt eine Zahl oder ein Zahlenpaar ist. Dafür gibt es den Operator instanceof.

Schreiben Sie ein Testprogramm. Es soll Folgendes leisten: Eingabe einer Folge von Zahlen und Zahlenpaaren über die Tastatur. Speicherung dieser Elemente in linearen Listen so, dass die Listen stets topologisch sortiert sind. Ausgabe dieser Listen auf dem Bildschirm.

7 Datenstrukturen und Algorithmen: Graphen

Digraphen und ungerichtete Graphen

Zu den fundamentalen Datenstrukturen der Informatik gehören die *Graphen*. Eine Klasse von Graphen haben wir schon kennen gelernt: die *Bäume*. Wie alle Graphen bestehen sie aus *Knoten* und *Kanten,* in diesem Falle haben die Kanten eine Richtung (Pfeile). Die Bäume sind Sonderfälle der *gerichteten Graphen* (Directed Graph, Digraph): Die Pfeile zeigen von einem *Anfangsknoten* zu einem *Endknoten*, Bild 7.1. Der Anfangsknoten ist *Vorgänger* des Endknotens und umgekehrt bezeichnet man den Endknoten als *Nachfolger* des Anfangsknotens. Anfangs- und Endknoten einer Kante wollen wir als grundsätzlich verschieden voneinander voraussetzen; der Graph ist also schlingenfrei. Außerdem möge es zu einer gerichteten Kante grundsätzlich keine zweite mit demselben Anfangs- und Endknoten geben. Parallele Kanten sind demnach ausgeschlossen – nicht hingegen antiparallele.

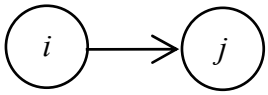


Bild 7.1 Kante mit Anfangsknoten i und Endknoten j

Sind die Kanten eines Graphen nicht gerichtet, haben wir es mit einem *ungerichteten Graphen* – oder einfach: einem Graphen – zu tun.

Zur Erinnerung: Bei Bäumen gibt es genau einen Knoten ohne Vorgänger. Das ist die Wurzel. Alle anderen Knoten haben genau einen Vorgänger, den Elter- oder Parent-Knoten. Die Nachfolger eines Elter-Knotens sind seine Kinder. Der Baum des Kurzform-Ausdrucks A=(B≤¬A) ist in Bild 7.2 wiedergegeben.

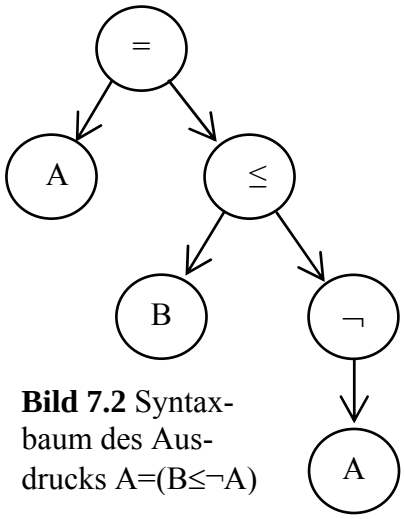


Bild 7.2 Syntaxbaum des Ausdrucks A=(B≤¬A)

Übungsaufgabe: Beweisen Sie, dass ein Baum nach dieser Definition keine Schleifen haben kann. Es ist also ausgeschlossen, von Punkt zu Punkt in Pfeilrichtung voranschreitend zum Ausgangspunkt zurückzukommen.

Achtung: Die Pfeile der Datenstruktur (Richtung der Pointer) müssen mit den Pfeilen in der Graphendarstellung nicht übereinstimmen. Das haben wir am Beispiel des Codierbaums für den Huffman-Algorithmus gesehen (Teil 2 des Skriptums). Dem dortigen Programm liegt die so genannte Parent-Darstellung des Baumes zu Grunde: Der Pointer zeigt vom Kind zum Elter.

Ein Lehrprojekt: Der PageRank-Algorithmus

Die Verlinkungsstruktur des World Wide Web ist ein Digraph. Jede Web-Seite ist ein Knoten in diesem Netz. Und Links werden durch gerichtete Kanten repräsentiert. Die Kante ist von der verlinkenden Seite auf die Zielseite gerichtet.

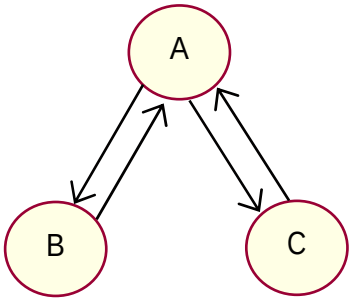


Bild 7.3 Ein Mini-Web

Das Mini-Web in Bild 7.3 besteht aus einer Hauptseite A, die auf ihre beiden Unterseiten B und C verweist. Die Unterseiten haben ihrerseits Links auf die Hauptseite.

Im World Wide Web gibt es viele Milliarden Web-Seiten. Und eines der Hauptprobleme ist, sich in diesem riesigen Informationsangebot zurechtzufinden. Suchmaschinen wie Google, Yahoo!, Lycos und AltaVista bieten da Hilfe. Auf eine Suchanfrage hin listen Suchmaschinen die dazu passenden Web-Seiten auf.

Suchmaschinen legen fest, in welcher Reihenfolge die Suchergebnisse in der Ergebnisliste erscheinen. Je weiter oben eine Seite erscheint, umso eher wird sie vom Adressaten auch tatsächlich wahrgenommen. Jeder Web-Seiten-Anbieter ist also darauf erpicht, eine möglichst gute Bewertung durch die Suchmaschine zu bekommen.

Die Bewertungsverfahren sind sehr komplex. Die Suchmaschinenbetreiber tun gut daran, sie nicht im Detail bekannt zu geben. Denn eines der Hauptprobleme ist, dass Trickser versuchen, die Eigenheiten der Berechnungsverfahren auszunutzen, um so eine möglichst hohe – wenngleich unberechtigte – Bewertung zu bekommen.

Aber einige der grundlegenden Algorithmen sind sehr wohl bekannt. Einer davon ist der PageRank-Algorithmus, benannt nicht nach den Web-Seiten (Pages) sondern nach einem seiner Erfinder, Lawrence (Larry) Page[1].

Der PageRank ist *eines* der Maße, die in die Reihenfolge der Listung von Suchergebnissen in Suchmaschinen eingehen. Je höher der PageRank, desto besser die Bewertung und desto weiter oben in der Liste der Suchergebnisse steht die Seite.

Die Berechnung des PageRanks ist Gegenstand eines Lehrprojekts. Es ist als Java-Archiv dokumentiert (PageRank.jar) und über die Web-Seite der Lehrveranstaltung frei zugänglich.

Übungen

7.1 Studieren Sie das PageRank-Programm und experimentieren Sie damit. Der Programmtext ist auf Lesbarkeit hin optimiert. Für große Netze ist das Programm zu ineffizient. Zeigen Sie die Schwachstellen im Hinblick auf die Effizienz auf und machen Sie Verbesserungsvorschläge.

[1] Zusammen mit Sergey Brin hat Lawrence Page die Firma Google gegründet.

8 Kürzeste Pfade in Graphen

Das Problem der kürzesten Pfade

Gegeben sei ein einfacher ungerichteter Graph, Bild 8.1, links. Die Verbindungen zwischen den Knoten – hier als Ecken oder Schnittpunkte gezeichnet – mögen Wegen entsprechen, deren Durchlaufen gewisse Kosten verursacht. Diese Kosten sind den Pfaden zugeordnet: Sie stehen unterhalb oder rechts von der zugehörigen Verbindung. Dadurch wird der Graph zu einem *bewerteten Graphen*, genauer: zu einem Graphen mit *nichtnegativer Kantenbewertung*.

Für jeden zusammenhängenden (schleifenfreie) Pfad in einem solchen Graphen lassen sich die Gesamtkosten als Summe der Verbindungskosten ermitteln. Im rechten Teilbild von Bild 8.1 ist mit fetter Linie ein möglicher Weg vom Start zum Ziel eingezeichnet. Die Gesamtkosten längs dieses Weges betragen 1+2+5+0+4+0+2+1+0 = 15.

Problemstellung: Gegeben sei ein Graph mit nichtnegativer Kantenbewertung. Ausgezeichnet seien ein Start- und ein Zielknoten. Unter allen Pfaden, die vom Start zum Ziel führen, ist derjenige gesucht, für den die Summe der Kantenbewertungen (Gesamtkosten) minimal wird.

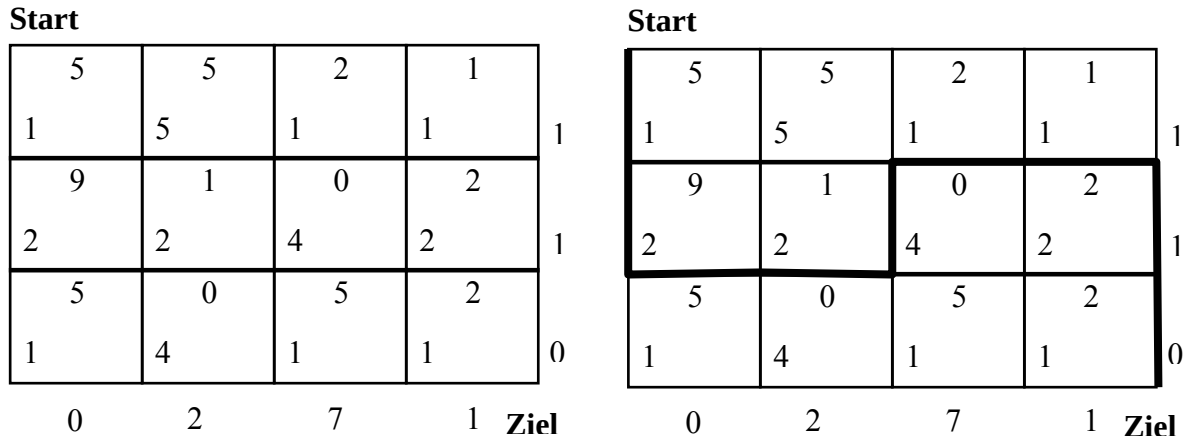


Bild 8.1 Bewerteter Graph (Wegenetz)

Das Einbettungsprinzip

Das Problem der kürzesten Pfade ist ein Musterbeispiel für die Anwendung von Problemlösungsheuristiken (Grams, 1990, S. 113 ff.). Mit der *vollständigen Enumeration*, also dem vollständigen Auflisten aller möglichen Pfade und dem direkten Vergleich der jeweils zugehörigen Wegekosten, kommt man hier nicht weit. Die Anzahl der Möglichkeiten ist schon bei diesem einfachen Beispiel ziemlich groß.

Hier hilft uns die Heuristik der *Verallgemeinerung*. Wir formulieren diese Heuristik als *Einbettungsprinzip*: Sei $P(\alpha)$ ein Problem, das für eine bestimmte konstante Größe α formuliert ist. Die Größe α möge einer Menge D angehören, für deren Elemente sich das Problem in derselben Weise stellen lässt. Anstatt das Problem $P(\alpha)$ direkt anzupacken, betten wir es in die allgemeine Problemstellung $P(x)$ für alle $x \in D$ ein. Die Lösungen für diese Probleme beschreiben wir mit zunächst noch unbestimmten und vom Problem x abhängenden Kenngrößen. Dann werden Beziehungen zwischen diesen Lösungsansätzen gesucht. Unter Ausnutzung dieser Beziehungen beginnt man mit der Lösung der leichteren Probleme und geht dann

weiter zu den schwerer lösbaren. So findet man dann möglicherweise eine allgemeine Lösung für die $P(x)$. Damit wäre dann auch das ursprüngliche Problem $P(\alpha)$ gelöst.

Die Suche nach dem minimalen Pfad können wir verallgemeinern, indem wir entweder den Start- oder den Zielpunkt variabel machen: Wir lassen nicht mehr nur einen Knoten als Start- oder Zielknoten zu, sondern alle möglichen Punkte des Graphen.

Halten wir hier einmal den Startknoten fest. Die möglichen Zielknoten bezeichnen wir mit der Variablen x. Der Optimalwert für den Knoten x – also die Kosten des kürzesten Weg bis dahin – bezeichnen wir mit $d(x)$. Das ist die *Optimalwertfunktion.* Für den Startknoten kennen wir den Wert dieser Funktion. Da keine Kante durchlaufen werden muss, ist $d(\text{Start}) = 0$.

Schritt für Schritt versuchen wir nun die Optimalwerte für die weiteren Knoten zu finden. Die Knoten, für die der Optimalwert bereits ermittelt worden ist, fassen wir zur Menge M zusammen. Die übrigen Knoten bezeichnen wir mit N. Anfangs ist also $M = \{\text{Start}\}$ und N enthält alle anderen Knoten. Nun untersuchen wir alle Kanten, die aus der Menge M herausführen.

Sei also (m, n) ein Knotenpaar mit $m \in M$ und $n \in N$ und $c(m, n)$ die Bewertung der zugehörigen Kante von m nach n. Wenn man nun vom Startknoten hin zum Knoten m einen optimalen Pfad wählt und dann den Knoten n über den ausgewählten Pfad erreicht, ergeben sich insgesamt die Kosten $d(m) + c(m, n)$. Dieser Wert wird für sämtliche solcher Knotenpaare aus $M \times N$ ermittelt. Ausgewählt wird nun derjenige Knoten n, für den sich hierbei ein minimaler Wert ergibt. Der ermittelte Wert ist der Optimalwert für diesen Knoten. Der Knoten wird aus der Menge N herausgenommen und der Menge M hinzugefügt.

Dass für den Knoten n auf diese Weise tatsächlich der Optimalwert gefunden worden ist, macht man sich so klar: Angenommen, es gäbe hin zu diesem Punkt n einen günstigeren Weg. Irgendwo müsste es auf diesem Weg ein Punktpaar (m', n') geben, so dass $m' \in M$ und $n' \in N$. Die Gesamtkosten des Wegs können also nicht kleiner als $d(m') + c(m', n')$ sein. Nachdem aber $d(m) + c(m, n) \leq d(m') + c(m', n')$ ist, sind auch die Gesamtkosten nicht günstiger als die des gewählten Pfades.

Aus der Optimalwertfunktion lässt sich der Minimalpfad mittels *Rückwärtsrechnung* ermitteln: Ausgehend vom Zielknoten fragt man sich Schritt für Schritt bis hin zum Startknoten durch. Für jeden Knoten n des Weges wählt man einen Vorgängerknoten m, für den die Beziehung $d(m) + c(m, n) = d(n)$ gilt.

Übung: Finden Sie mit dem Einbettungsprinzip den Minimalpfad im Wegenetz aus Bild 8.1. Gehen Sei einmal vom Startknoten und ein andermal vom Zielknoten aus.

Der Algorithmus von Dijkstra

Für die Knoten des Graphen wird die Klasse `Node` definiert. Die Klasse enthält wenigstens zwei Attribute: Den Namen und den Kopf einer linearen Liste für die wegführenden Kanten. Ein weiteres Attribut ist für die Knotenbewertung d vorzusehen.

Für die Kanten wird die Klasse `Edge` definiert. Mit $S(i)$ wird die Menge der Endknoten aller vom Knoten i abgehenden Kanten bezeichnet.

`Node` wird als Subklasse von `Link` definiert. Auf diese Weise lassen sich die Knoten des Graphen in Listen einfügen. So erhält man programmtechnische Realisierungen von Knotenmengen. Die Grundmenge der Knoten wird mit V bezeichnet. Knoten, die erstmals bewertet worden sind, werden aus der Grundmenge V herausgenommen und in die Menge Q eingefügt.

Schleifeninvariante der folgenden Iteration: Knoten mit endlicher Bewertung, die nicht in der Menge Q stehen, haben ihren Optimalwert. Die Menge Q enthält Knoten mit Bewertung. Der Knoten mit der kleinsten Bewertung hat seinen Optimalwert.

Initialisierung: Der Start-Knoten wird mit dem Wert $d = 0$ initialisiert. Alle anderen Knoten bleiben unbewertet ($d = \infty$). $Q = \{\text{Start}\}$. V enthält die restlichen Knoten.

Schleifenbedingung: $Q \neq \Phi$

Schleifenanweisung: Entferne minimales i aus Q und füge es in V ein. Führe für jedes j aus $S(i)$ die folgende Schritte durch: Setze $d(j) = d(i) + c(i, j)$, falls der alte Wert $d(j)$ ungünstiger ist. Falls der Knoten j unbewertet war, entferne ihn aus der Menge V und füge ihn in Q ein.

Diese Version des Algorithmus beschränkt sich auf die Berechnung der Optimalwertfunktion. Für die schnellere Wegermittlung sollte man je Knoten zusätzlich den optimalen Vorgängerknoten abspeichern (*Politikfunktion*).

Weitere Versionen und programmtechnische Hinweise bieten die Bücher von Neumann/Morlock (1993, S. 213 ff.) und Li (1998, S. 374 ff.).

Dateieingabe und Dateiausgabe

Im *Projekt „Kürzeste Verbindungen in einem Straßennetz“* ist die Ein- und Ausgabe der Daten und die Verwendung von Dateien einige Überlegungen wert. Das Hauptproblem stellt sich bei der Eingabe: Erstens wissen wir nicht von vornherein, wie umfangreich die Daten des Netzes sind und damit haben wir auch keinen Anhaltspunkt über den für diese Daten bereitzustellenden Puffer.

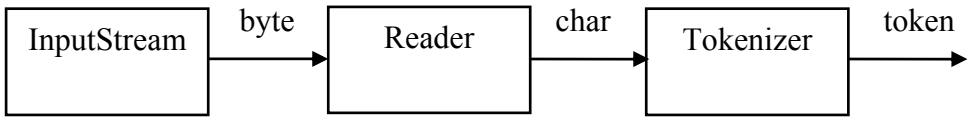


Bild 10.1 Scanner für eine Eingabedatei

Andererseits müssen wir von den Daten eine interne Repräsentation des Netzes ableiten, in dem dann die Eingabe-Informationen Gestalt annehmen. Nahe liegend ist, die Daten der Eingabedatei Stück für Stück einzulesen und die interne Repräsentation Schritt haltend aufzubauen.

Genau das sind die Aufgaben der ersten Verarbeitungsstufen eines Compilers: *Scanner* und *Parser*.

Der Scanner gruppiert die Eingabedaten zu einer Folge von *Token*, Bild 10.1. Die Token sind die bedeutungstragenden Zeichen der Eingabesprache: Namen, Operatoren, Trennzeichen, Zahlen.

Die Aufgabe des Parsers ist dann, aus dieser Tokenfolge die interne Darstellung des Wirkungsgefüges abzuleiten. Bei einer Programmiersprache ist das der *Syntaxbaum*. Im Falle des Straßennetzes ist es eine Darstellung des Netzes als gerichteter Graph.

Hier konzentrieren wir uns auf das Einlesen einer Tokenfolge aus einer *Textdatei*, die mit irgendeinem Texteditor erstellt worden ist. Am Schluss des Abschnitts ist eine einfache Java-Klasse für die Ein-/Ausgabe abgedruckt.

Der *Tokenizer* ist ein Objekt dataIn, das folgendermaßen erzeugt wird:

```
dataIn = new StreamTokenizer(
  in=new InputStreamReader(new FileInputStream(fileName))
);
```

Die Eingabe erfolgt in mehreren Stufen und für jede der Stufen wird ein eigenes Objekt erzeugt. Der FileInputStream macht nichts anderes, als die Bytes der Datei mit dem Namen fileName nacheinander dem InputStreamReader Verfügung zu stellen. Der macht daraus Zeichen des char-Datentyps (Unicode Standard). Der StreamTokenizer schließlich gruppiert diese Zeichen zu den bedeutungstragenden Symbolen der Eingabesprache. Diese Klassen gehören zum java.io-Package.

Wenn jemand einen Tokenizer braucht, dann muss er sich den aus diesen Teilen zusammenstellen. Ist das nicht zu umständlich?

Nein, ist es nicht. Denn so einfach wie hier geht das nur in ganz einfachen Fällen. In komplizierteren Fällen sind auf den einzelnen Stufen Anpassungen erforderlich.

Wenn wir uns auf die Zeichen der Ascii-Codetabelle beschränken, kommt man mit den einfachen Objekterzeugungsausdrücken des obigen Programmausschnitts aus. Der InputStreamReader verwendet, solange man nichts anderes verlangt, den Zeichensatz des Betriebssystems. Und der umfasst jedenfalls die Zeichen des Ascii-Codes.

Codierte Zeichenvorräte

Schwierig wird die Sache in dem Moment, in dem man besondere Zeichen einlesen und verarbeiten will. Beispielsweise benötigt man zur Darstellung von Kurzform-Ausdrücken Zeichen wie ¬ oder ≤.

Im Computer werden Schriftzeichen als bloße Bitfolgen dargestellt. Im einfachsten Fall steht je ein Byte (8-Bit-Wörter) für ein Zeichen. Heute ist der Unicode allgemein gebräuchlich, mit dem sich die gebräuchlichsten Schriftzeichen der Welt identifizieren lassen. Der Unicode Standard benutzt zwei Bytes zur Codierung der Zeichen (UCS-2). Das ist auch die dem Java-Datentyp char zu Grunde liegende Darstellung.

Üblich ist die Benennung der Zeichen mit einem Unicode-Escape. Es beginnt mit den Zeichen \u. Angehängt ist die Codenummer im Hexadezimalsystem. In diesem System steht beispielsweise \u0020 für das Leerzeichen. Die Zeichen von \u0000 bis \u007F, also die ersten 128 Zeichen der Codetabelle, stimmen mit den Zeichen der ASCII-Code-Tabelle überein.

Das Zeichen ¬ gehört nicht zum ASCII-Code. Es ist das Unicode-Zeichen \u00AC. Dieses lässt grundsätzlich zwar noch als ein Byte darstellen. Allerdings kommt es bei Codenummer oberhalb \007F darauf an, welcher Zeichensatz geladen ist.

Spätestens beim Zeichen ≤ verlässt uns die Hoffnung auf die Ein-Byte-Darstellung. Es ist das Unicode-Zeichen \u2264.

Aufgabe des InputStreamReader ist es, aus einem Byte-Strom einen Unicode-Zeichen-Strom zu machen. Er liest die Bytes und dekodiert diese in Zeichen vom Typ char. Dabei wird ein bestimmter codierter Zeichenvorrat (charset) zu Grunde gelegt.

Ein *codierter Zeichenvorrat* (*coded character set*) ist eine durchnummerierte Menge von abstrakten Zeichen. Die ASCII-Codetabelle stellt einen solchen codierten Zeichenvorrat dar. Ein anderer ist der volle Unicode.

Der codierte Zeichenvorrat kann bei der Erzeugung des InputStreamReader-Objekts namentlich gegeben sein. Falls nicht, wird der Default-Zeichenvorrat des Betriebssystems genommen. Im folgenden Programmbeispiel wird ein InputStreamReader erzeugt, der den Zeichenvorrat UTF-8 benutzt.

```
FileInputStream in=new FileInputStream(fileName);
InputStreamReader reader= new InputStreamReader(in, "UTF-8");
```

UTF-8 steht für *Eight-bit UCS Transformation Format* und ist ein Codierungsschema für den vollen Unicode. Dieses Schema hat den Vorteil, dass für die Zeichen der ASCII-Tabelle tatsächlich nur ein Byte aufgewendet wird. Für die anderen Zeichen werden dann zwei oder gar drei Bytes fällig.

Mittels UTF-8-Dateien lässt sich die Kommunikation zwischen den selbst konstruierten Java-Programmen und den Standard-Tools (Editoren und Bürosoftware) weitgehend einschränkungsfrei gestalten. Die Schnittstelle ist für sämtliche Unicode-Zeichen durchlässig.

Formatierte Zahlenausgabe

Grundsätzlich kann man die Ausgabe über die Ausgabe von Strings organisieren: Die wesentlichen Formatierungsangaben wie Zeilenwechsel und Tabulatoren sind Bestandteile des Strings. In Java bietet der Verkettungsoperator + und die automatische Wandlung von Zahlen in Strings die Möglichkeit, auch Zahlen auf einfache Weise einzubinden. So lassen sich Tabellen zusammenstellen, die mit Tabellenkalkulationsprogrammen weiterverarbeitet werden können.

Das Abspeichern eines solchen Strings in einer Hintergrunddatei ist mit der Methode putTextToFile(…) des vorigen Abschnitts problemlos möglich. Dabei werden die Zahlen in einer Standardnotation und weitgehend ohne Genauigkeitsverlust übermittelt. Die Formatierung der Zahlen ist in diesen Fällen Aufgabe des Empfängerprogramms, beispielsweise eines Tabellenkalkulationsprogramms.

Was aber, wenn die Zahlen formatiert übermittelt werden sollen? Ein solcher Wunsch entsteht beispielsweise, wenn man eine Tabelle direkt und ohne weitere Aufbereitung in ein Word-Dokument aufnehmen will.

Hier hilft die Klasse NumberFormat des java.text-Packages weiter. Die folgenden Programmzeilen sorgen dafür, dass die Mantissen von Gleitpunktzahlen auf drei Stellen nach dem Dezimalpunkt gerundet in den Ausgabestring übernommen werden.

```
NumberFormat nf= NumberFormat.getInstance(Locale.US);
nf.setMaximumFractionDigits(3);
…
System.out.print(nf.format(m));
System.out.print(" +- ");
System.out.println(nf.format(2.0*Math.sqrt(m*(1-m)/(n-1))));
```

Übungen

8.1 *Projekt „Kürzeste Verbindungen in einem Straßennetz“.* Erstellen Sie ein Programm, das Ihnen die Eingabe eines Straßennetzes mit Knotenpunkten erlaubt. Für die Verbindungen (Straßen) sollen sich nichtnegative Bewertungen (die Entfernungen) vorgeben lassen. Ein Start- und Zielknoten können vorgegeben werden. Das Programm soll die folgenden Ergebnisse liefern: Die Optimalwertfunktion für sämtliche Knoten des Netzes und einen optimalen Weg (Knotenfolge) vom Startknoten zum Zielknoten.

Benutzen Sie den Algorithmus von Dijkstra und halten Sie sich an die Hinweise aus dieser Lektion.

Erstellen Sie ein *digraph-Package,* in dem Sie alle wichtigen Klassen für die Erstellung und Verwaltung von Digraphen und ungerichteten Graphen unterbringen. Führen Sie alle Entwurfsschritte gründlich aus und notieren Sie die jeweiligen Ergebnisse im *Projektprotokoll*. Beginnen Sie in dieser Lektion mit dem Pflichtenblatt und dem Entwurf.

Hinweis: Die Eingabe der Daten direkt über die Tastatur ist diesmal – allein wegen des Umfangs des Datenmaterials – nicht angebracht. Die Netzstruktur und die Bewertungen sollten über eine Datei eingelesen werden können. So können Sie für die Formulierung der Daten, für Korrekturen und Erweiterungen einen gewöhnlichen Texteditor nutzen. Nehmen Sie in das Pflichtenheft auf, dass eine geeignete Eingabesprache zu definieren ist. Denken Sie daran, wie man so etwas bei Programmiersprachen macht.

9 Entwurfsmuster: Algorithmen auf Bäumen

Repräsentation von Bäumen: Das multiLinkedStruct-*Package*

Diesem und den folgenden Lektionen liegt die Software des Lehrprojekts KurzForm zu Grunde. Die Web-Page zur Lehrveranstaltung bietet den Zugriff auf das Java-Archiv des Projekts.

Als Beispiele für die Datenstruktur *Baum* nehmen wir die Syntaxbäume von Kurzformausdrücken (Informatik I), zum Beispiel den Baum aus Bild 7.2.

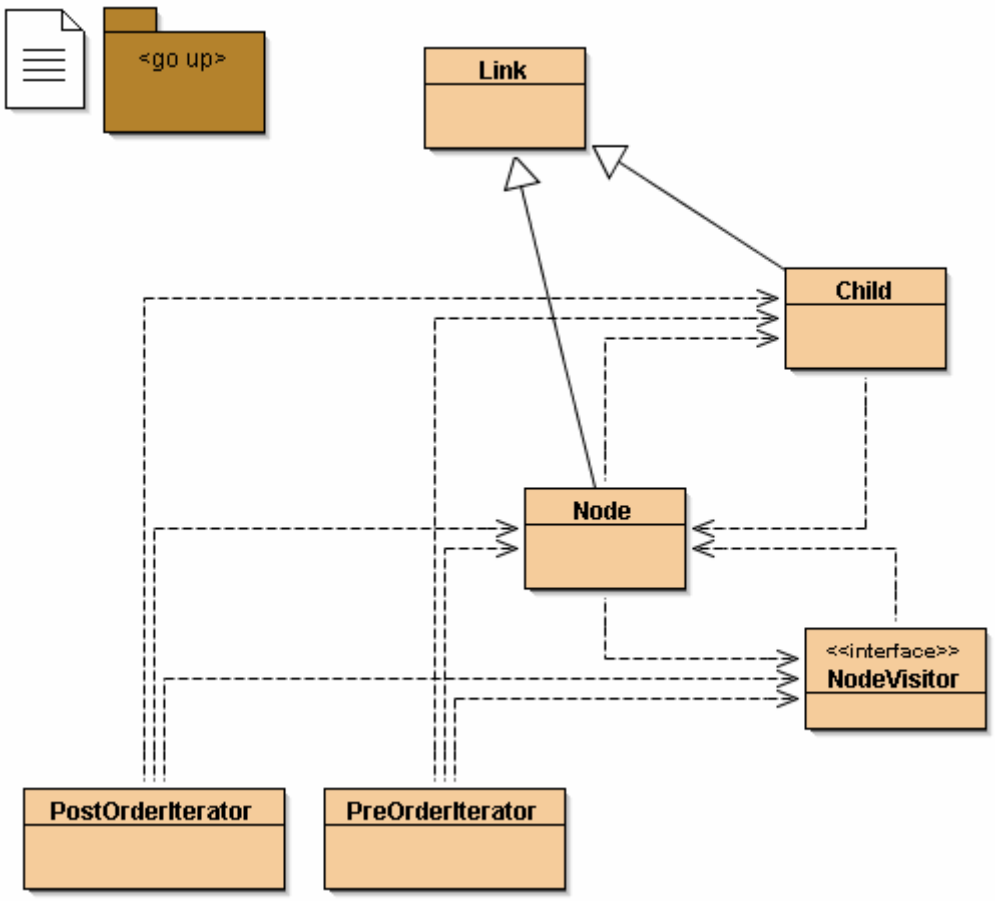


Bild 9.1 Das multiLinkedStruct-Package

Das multiLinkedTree-Package stellt die grundlegende Datenstruktur zur Verfügung. Die Klasse Node dient der Parent-Darstellung von Bäumen. Zur Erleichterung der Baumdurchläufe (Tree Traversal) mittels Iteratoren wird jeder Knoten (Node) mit einer linearen Liste seiner Kinder (Child) ausgestattet.

Die Iteratoren sind Realisierungen des Iterator-Entwurfsmusters[1]. Es sind *interne* Iteratoren. Bei Aufruf ihrer traverse-Methode durchlaufen sie den durch seine Baumwurzel benannten Baum selbständig und rufen für jeden Knoten die visit-Methode des NodeVisitors auf. Zu-

[1] Die Generalisierungsmethode der Entwurfsmuster wurde von Erich Gamma u. a. beschrieben. Sie basiert auf den Ideen des Architekten Christopher Alexander (s. Quellen).

sammen mit den konkreten Visitoren (Implementierungen der NodeVisitor-Schnittstelle) bilden sie die *Generatoren* des „Compilers“.

Die Philosophie der Entwurfsmuster

Immer wieder hat die Informatik Anregungen von der Architektur erhalten. Begriffe wie „Software-Architektur“ zeugen davon. Von großem Einfluss sind in neuerer Zeit die Ideen des Architekten Christopher Alexander (1977, 1979). Seine Idee einer „Pattern Language“ wurde unter der Bezeichnung *Design Patterns* (Entwurfsmuster) von der Informatik aufgegriffen und abgewandelt (Gamma u. a., 1995).

Ein paar Aussagen Christopher Alexanders sollen das Wesen der Design Patterns deutlich machen: “Indeed it turns out, in the end, that what this method does is simply free us from all method” (Alexander, 1979, “The Timeless Way of Building”, S. 13). “The proper answer to the question, ‘how is a farmer able to make a new barn?’ lies in the fact that every barn is made of patterns ...These patterns in our minds are, more or less, mental images of the patterns in the world: they are abstract representations of the very morphological rules which define the patterns in the world ... the system of patterns forms a language” (Alexander, 1979, S. 178, 181, 183).

Design Patterns in der objektorientierten Software-Konstruktion sind Beschreibungen eleganter Lösungen für spezifische Probleme: „They reflect untold redesign and recoding as developers have struggled for greater reuse and flexibility in their software” (Gamma, Helm, Johnson, Vlissides, 1995, xi). “The design patterns in this book are *descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context”* (Gamma u.a., 1995, S. 3).

Ein Beispiel für das Iterator- und das Visitor-Muster

Unter den 23 Design Patterns, die Gamma, Helm, Johnson und Vlissides in ihrem Buch aufführen findet man auch das *Iterator*- und das *Visitor*-Muster.

Will man aus dem Syntaxbaum eines Kurzform-Ausdrucks die Postorder- oder irgendeine andere Darstellung gewinnen, muss man den Baum Knoten für Knoten durchlaufen, beispielsweise nach den Regeln des *Postorder-Traversals*.

Und für jeden Knoten sind dann spezifische Operationen durchzuführen. Die beiden Funktionen sollte man sauber voneinander trennen: Tree-Traversal einerseits und die je Knoten auszuführende Operation andererseits.

Das Tree-Traversal wird mit dem Iterator-Muster und die Operation durch das Visitor-Muster realisiert. Konkrete Visitoren müssen die Visitor-Schnittstelle implementieren:

```
public interface NodeVisitor {void visit(Node p);}
```

Es werden Iteratoren für die verschiedenen Durchlaufordnungen (Pre- und Postorder) benötigt. Die internen Iteratorn sind unmittelbar der Datenstruktur zugeordnet und Bestandteil des multiLinkedStruct-Packages. Der Postorder-Iterator sieht so aus:

```
public class PostOrderIterator {
  NodeVisitor p;
  public PostOrderIterator(NodeVisitor p) {this.p=p;}

  /*traverse(n) durchlaeuft den Baum mit der Wurzel n und ruft fuer jeden
```

```
  besuchten Knoten den Visitor auf. Der Durchlauf geschieht in Postorder.
  **********************/
  public void traverse (Node n) {
    for (Child c= n.children.next(); c!=null; c=c.next())
      traverse(c.child);
    p.visit(n);
  }
}
```

Der Baum-Durchlauf eines Baumes mit dem Wurzelknoten n wird durch Aufruf traverse(n) bewirkt. Die eigentlichen Aktionen, die je Knoten auszuführen sind, sind Angelegenheit der konkreten NodeVisitor-Objekte. Die konkreten NodeVisitor-Klassen müssen die Schnittstelle NodeVisitor implementieren. Die je Knoten auszuführenden Aktionen werden mit der visit-Methode realisiert.

Das multiLinkedStuct-Package ist ein abstrakter Datentyp (ADT): Die Strukturelemente und Iteratoren gehören zum „Angebot“ des Datentyps. Abstrakt bleiben die Visitoren, also das, was je Knoten zu tun ist. Die Knoten eines Baumes und ihre Visitoren sind vom Anwender zu konkretisieren (Programming by Contract).

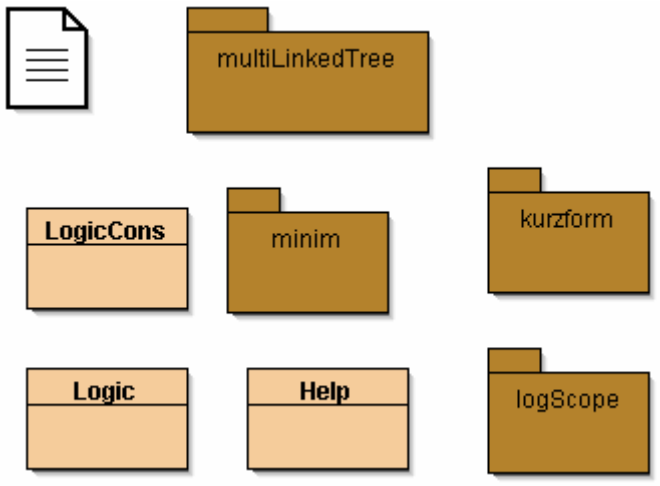
Übungen

9.1 *Projekt „Kurzform-Parser“.* Studieren Sie die Dokumentation des Projekts und bereiten Sie sich so auf die folgende Aufgabe vor. Experimentieren Sie mit dem Programm.

9.2 *Projekt „Zweipole“, Teil 1.* Es soll ein Programm entwickelt werden, das die textuelle Beschreibung von Zweipolen entgegennimmt und den Widerstandwert ausgibt. Schreiben Sie das Pflichtenblatt des Projekts. Definieren Sie eine Sprache zur Beschreibung von Zweipolen. Jeder Zweig kann eine Parallelschaltung oder eine Serienschaltung von Zweipolen sein. Es handelt sich also um eine rekursive Struktur. Sie dürfen hier voraussetzen, dass es sich um ein reines Widerstandsnetzwerk handelt.

10 Compiler und Interpreter für die Kurzform-Logik

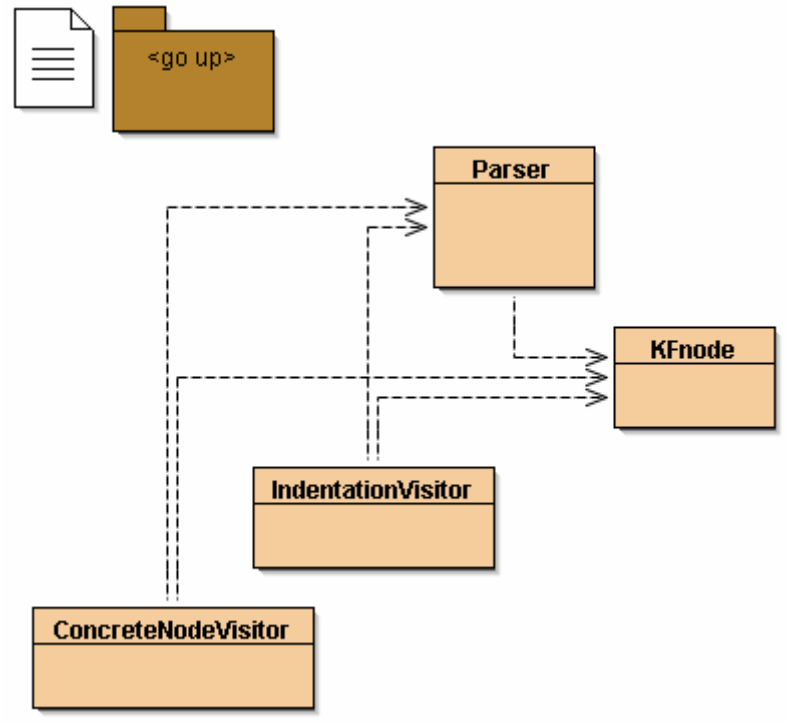
Gesamtprojekt: Die Hauptklassen



LogicCons ist die Hauptklasse der Konsolenversion. Logic bietet demgegenüber eine einfache grafische Bedienoberfläche mit einem aufklappbaren Hilfefenster (Help).

Das multiLinkedTree-Package stellt die grundlegende Datenstruktur zur Verfügung. Das kurzform-Package enthält die anwendungsspezifische Konkretisierung der Datenstruktur, den Parser und die Methoden zur Erzeugung der verschiedenen Baumdarstellungen (Postorder, Preorder, Einzugstechnik). Das logScope-Package sorgt für die Erzeugung der Erfüllungsmenge und das minim-Package liefert die minimierte disjunktive Normalform.

Das kurzform-Package: Parser



KFnode-Elemente sind Node- und damit auch Link-Elemente. Aus Elementen dieser Klasse wird der Syntaxbaum aufgebaut. Über die Link-Verkettung wird die Parent-Darstellung des Syntaxbaumes realisiert. Jeder Knoten referenziert über sein next-Attribut seinen Elter-Knoten. Darüber hinaus hat KFnode zwei Attribute. Eins für die Funktion des Knotens und eins für die Liste seiner Kinder. Die Funktion eines Knotens ist gleich dem Verknüpfungsoperator, der auf seine Kinder anzuwenden ist. Der Verknüpfungsoperator ergibt sich aus dem nichtterminalen Symbol, zu dem der Knoten gehört und (im Falle der „= ≤"–Mehrdeutigkeit) aus der folgenden Verzweigung.

Es wird ein äußerst einfacher *Top-Down-Parser* realisiert. Jeder rechten Seite α einer Produktionsregel (bzw. einem noch nicht verbrauchten Teil davon) und damit jeder Verzweigungsmöglichkeit im Syntaxbaum kann eine Symbolmenge FIRST(α) zugeordnet werden (siehe Tabelle). Diese Menge sagt uns nämlich, dass aus dieser Verzweigung nur ein Pfad hervorgehen kann, der mit einem Symbol aus dieser Menge endet. Zu jedem Zeitpunkt gibt es immer nur Verzweigungsmöglichkeiten, deren zulässige Symbolmengen disjunkt sind. Das heißt, dass mit Hilfe dieser Symbolmengen der Syntaxbaum sukzessive aus der Folge der Eingabesymbole Symbol für Symbol entwickelt werden kann. Jedes gelesene Symbol der Folge sagt uns eindeutig, wie die Entwicklung des Syntaxbaums weitergeht (Predictive Parsing).

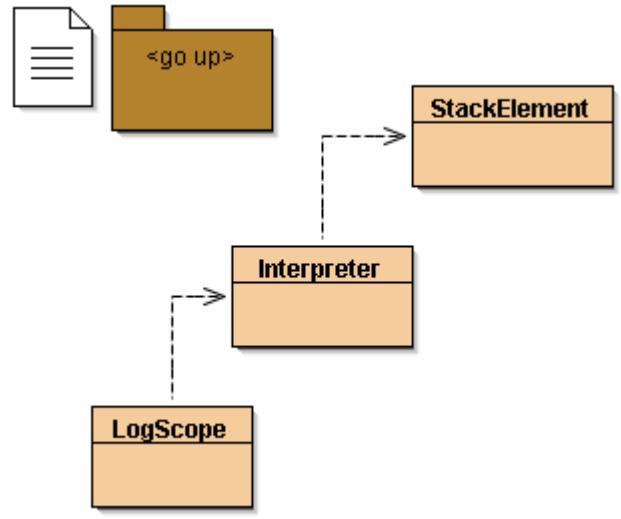
α	**FIRST(α)**
Expression	¬, (, 0, 1, A, B, …, Z
SimpleExpression	¬, (, 0, 1, A, B, …, Z
ComparativeOperator SimpleExpression	=, ≤
Term	¬, (, 0, 1, A, B, …, Z
+Term	+
Factor	¬, (, 0, 1, A, B, …, Z
¬Factor	¬
(*Expression*)	(
Element	0, 1, A, B, …, Z
Constant	0, 1
Variable	A, B, …, Z

Am Anfang besteht der Baum nur aus der Wurzel, die zum nichtterminalen Symbol *Expression* gehört. Weitere Verzweigungsmöglichkeiten gibt es nicht. Jetzt wird der Baum linear „von oben nach unten" entwickelt, bis das erste Eingabesymbol untergebracht werden kann. Dabei hat sich eine Reihe von Verzweigungsmöglichkeiten aufgetan, deren FIRST-Symbolmengen disjunkt sind. Jetzt ist der Syntaxbaum längs des aktuellen Pfades nach oben nach derjenigen Verzweigungsmöglichkeit abzusuchen, deren FIRST-Symbolmenge das zweite Eingabesymbol enthält. Diese Verzweigung wird aufgebaut und linear von oben nach unten entwickelt, bis auch das zweite Eingabesymbol verbraucht ist. Und so weiter.

Das Programm dieses Parsers soll diese Vorgehensweise transparent machen und mit möglichst einfachen Sprachmitteln realisieren (Goto-Programm realisiert mittels while-Schleife und switch-Anweisung).

Die *Visitoren* implementieren die NodeVisitor-Schnittstelle des multiLinkedTree-Packages. Die konkreten Visitoren (ConcreteNodeVisitor, IndentationVisitor) legen in der visit-Methode fest, was zu passieren hat, wenn der Iterator einen Knoten ausgewählt hat und für diesen Knoten die visit-Methode aufruft. Der ConcreteNodeVisitor hat die Aufgabe, je nach gewähltem Iterator, die Postorder- oder die Preorder-Darstellung des Kurzformausdrucks zu erzeugen. Und der IndentationVisitor erzeugt die Einzugsdarstellung des Syntaxbaums mit Hilfe des Preorder-Iterators.

Das logScope-Package: Interpreter



Die *Interpreter-Klasse* realisiert den Auswertungsalgorithmus für Kurzformausdrücke, die in Postorder-Darstellung gegeben sind. *LogScope* liefert den logischen Spielraum eines in Postorder-Darstellung gegebenen Kurzformausdrucks.

Übung

10.1 *Projekt „Zweipole", Teil 2.* Schreiben Sie einen Compiler, der Ausdrücke der Zweipol-Sprache in Postorder-Darstellung umwandelt. Schreiben Sie einen Interpreter für den Fall, dass alle Elementarzweipole reine Widerstände sind. Eingabe in den Interpreter sind 1. die Postorderdarstellung des Zweipols und 2. die Wertebelegung der Widerstandsvariablen. Als Ergebnis soll der Widerstandswert des Zweipols geliefert werden. Bearbeiten Sie diese Aufgaben im Rahmen eines Projekts.

11 Texte

Physikalische und Logische Fonts

Ein Symbol oder Zeichen beschreibt einen Buchstaben, eine Ziffer oder ein Satzzeichen auf abstrakte Weise. Ein 4solches Symbol ist eine Äquivalenzklasse von Signalen. Diese Äquivalenzklassen nennen wir *abstrakte Zeichen*. Sie bilden die Elemente der codierten Zeichenvorräte. Die Ausprägung der Zeichen auf der Signalebene nennen wir *konkrete Zeichen*.

Ein *Font* oder *Schriftschnitt* bildet einen codierten Zeichenvorrat auf die konkreten Zeichen ab.

Die Java-2-Platform unterscheidet *physikalische* und *logische Fonts*.

Physikalische Fonts sind Bibliotheken mit den bildnerischen Elementen der Zeichen (Glyphs), mit denen die abstrakten Zeichenfolgen in die konkreten Zeichen umgesetzt werden. Dabei werden Technologien wie TrueType und PostScript Type 1 benutzt.

Physikalische Fonts haben Namen wie Helvetica, Palatino, HonMincho. Ein Font deckt nicht sämtliche Zeichen des Unicode ab. Typischerweise unterstützt ein Font nur eine begrenzte Menge von Schreibsystemen, beispielsweise nur die lateinischen Zeichen oder nur die japanischen. Welche Fonts verfügbar sind, hängt von der Konfiguration des Systems ab.

Es gibt fünf *logische Fonts*, die in jeder Java-Laufzeitumgebung verfügbar sind: Serif, SansSerif, Monospaced, Dialog, and DialogInput. Diese logischen Fonts sind keine realen FontBibliotheken. Erst durch die Laufzeitumgebung werden die logischen Fonts auf physikalische Fonts abgebildet. Diese Abbildung ist von der lokalen Konfiguration abhängig. Es kann zu Zeichenmischungen zwischen verschiedenen Fonts kommen. Diese Zeichenmischung wird akzeptiert, um einen möglichst großen Zeichenvorrat abdecken zu können. Durch die logischen Fonts wird unter anderem der Schriftduktus festgelegt[1]. Auf diese Weise wird trotz der Mischung der Schriften ein harmonisches Erscheinungsbild erreicht.

Weitere Elemente einer Schrift sind die *Strichstärke* und die *Schriftlage*. Beispiele sind Plain, Bold und Italic. Der *Schriftgrad* ist eine Größenangabe und diese bezieht sich auf den Buchstaben H. Maßeinheit ist der *typographische Punkt* (3/8 mm).

Vom Byte zum Character

Die Sonderzeichen ¬ und ≤ sollen sowohl im Hilfe-Fenster als auch im Ein- und Ausgabefenster korrekt dargestellt werden.

Der System-Font kommt damit normalerweise nicht zurecht. Gerade wenn man die üblichen Texteditoren nutzen will, muss man sehen, wie diese Zeichen vom Editor über die Hintergrunddatei in das Java-Programm gelangen.

Zuallererst muss man sehen, ob der benutzte Editor überhaupt ermöglicht, die Sonderzeichen auf den Hintergrundspeicher zu bringen und wenn ja, mit welcher Codierung er das tut. Üblich ist die Utf8-Codierung. Die folgende Eingabemethode liest Daten von einer Textdatei, die im Utf8-Format codiert sind.

---

[1] Gulbins, J.; Kahrmann, C.: Mut zur Typographie. Springer-Verlag, Berlin, Heidelberg 1992

```
static String getTextFromFile(String fileName) {
    int maxBuf=10000;
    char[] buf = new char[maxBuf];
    String t=null;
    try {
        FileInputStream in=new FileInputStream(fileName);
        InputStreamReader reader= new InputStreamReader(in, "UTF-8");
        reader.skip(1);
        reader.read(buf, 0, maxBuf);
        t= String.valueOf(buf);
        in.close();
    } catch(IOException e){
        System.out.print("File cannot be read\n"+e.toString());
    }
    return t; //String.valueOf(buf);
}
```

Dokumentation

Regeln zur Aufbau und zur inhaltlichen Gestaltung

Auch für die Dokumentation gilt als oberste Regel: Strebe nach *Einfachheit* und *Direktheit der Kommunikation*.

Die Zeit des Lesers ist knapp. Er hat nicht Zeit, sich durch einen schlecht gegliederten Wust weitschweifiger Texte zu wühlen. Die Aufmerksamkeit des Auftraggebers oder Chefs zu verspielen ist kein Erfolgsrezept!

Die Projektdokumentation muss für die Zielgruppe bzw. für einen unvorbereiteten, aber sachkundigen Leser leicht verständlich abgefasst sein.

Aufbau der Projektdokumentation:

TITELEI

- *Titel (Wer, Wann, Was, Wo?):* Projektbezeichnung mit Untertiteln. Nennung des Autors oder der Autoren. Datum. Ort.
- *Vorwort (Was, Warum, Wie?):* Nennt die Absicht, die mit dem Projekt und der Dokumentation verfolgt wird und Besonderheiten in inhaltlicher oder gestalterischer Hinsicht.
- Zu*sammenfassung:* Fasst den Textteil speziell unter den Aspekten Problem, Ziel, Methode und Ergebnis der Arbeit zusammen. Hier steht nichts, was über den Textteil hinausgeht. Die Zusammenfassung muss *allgemeinverständlich und prägnant* formuliert sein. Man nehme sich vor, sie auf maximal einer Seite, besser noch auf eine halben, unterzubringen. Die Zusammenfassung soll dem Leser die Möglichkeit bieten, schnell die Relevanz der Dokumentation und des Projekts für sich und seine Arbeit festzustellen.
- *Inhaltsverzeichnis:* Hier sind alle Punkte des Lebenszyklus (Clustermodell) abzuhandeln. Dadurch ist die oberste Ebene der Gliederung des Textteils nahezu vollständig bestimmt.

TEXTTEIL

- *Einleitung (optional):* Hier werden das zu lösende Problem, das Ziel des Projekts und die wesentlichen Methoden genannt. Das ist auch die Stelle, an der das gesamte technische Umfeld, die Trends und vergleichbare Produkte angesprochen werden.

- *Hauptteil:* Die Gliederung des Hauptteils folgt dem Lebenszyklusmodell.

ANHANGTEIL

- *Literaturverzeichnis:* Hier stehen nur die Werke, die im Textteil auch tatsächlich zitiert werden. Auf Korrektheit der Zitate achten. §1 Urheberrechtsgesetz (UrhR) besagt: „Die Urheber von Werken der Literatur, Wissenschaft und Kunst genießen für ihre Werke Schutz nach Maßgabe dieses Gesetzes." § 2 UrhR stellt unter Anderem „Sprachwerke, wie Schriftwerke und Reden, sowie Programme für die Datenverarbeitung" unter Schutz. § 51 UrhR regelt die Zulässigkeit von Zitaten „in einem durch den Zweck gebotenen Umfang" und § 63 UrhR legt die Pflicht fest, „stets die Quelle deutlich anzugeben".
- *Index (optional)*

Regeln zur Typographie und weitere Tipps

Für *Layout* und *Formelsatz* sind die Normen verbindlich. Man orientiere sich an sorgfältig gestalteten Büchern (z. B.: „Technische Informatik" von Schiffmann/Schmitz).

Wie beim Web-Design gilt es auch hier den Wahrnehmungsapparat des Lesers durch schlichte und durchgängige Seitengestaltung zu entlasten. Ein paar Grundregeln:

- Form follows function
- Gestaltungselemente nach festen Regeln verwenden
- Farbcode vorab festlegen
- Beschränkung auf Bilder, die zum Thema passen und die die Darstellung unterstützen
- Verzicht auf Schnickschnack und Höflichkeitsfloskeln

Wichtige Hinweise zur professionellen Textverarbeitung und zum Desktop Publishing (DTP) findet man im Buch von Jürgen Gulbins und Christine Kahrmann „Mut zur Typographie" (Springer, Berlin, Heidelberg 1992). Unter anderem findet man dort die „Zehn typischen Sünden beim DTP" (S. 251 ff.). Die folgenden Punkte sind daran angelehnt.

1. *Falsche Formate:* Etwa 60 Buchstaben je Zeile fördern die Lesbarkeit. Das sollte man bei der Wahl der Schriftgröße und der Spaltenbreite berücksichtigen.
2. *Zu viele Schrifttypen und falsche Schriftauszeichnung:* Gehen Sie mit Unterscheidungsmerkmalen möglichst sparsam um. Im Text genügt meist die Kursivschrift zur Hervorhebung. Fette Schrift im Fließtext nur verwenden, wenn **Aufdringlichkeit beabsichtigt** ist. Genausowenig braucht man im Allgemeinen Unterstreichungen, Schattierungen und Versalien. Die Unterschiede in Schrifttyp und Schriftauszeichnung sollten weder zu groß noch zu klein sein. Machen Sie sich von vornherein klar, was sie mit den unterschiedlichen Schriften sagen wollen: Legen Sie die Regeln zur Wahl von Schriftbild, Schriftgröße, und Schriftauszeichnung (kursiv, fett usw.) vorab nieder. Haben diese Regeln für das Verständnis des Textes größere Bedeutung, dann sollten diese Regeln in das Dokument aufgenommen werden.
3. *Unpassende Schriften:* Verwenden Sie für umfangreiche Texte vorzugsweise Schriften mit Serifen (wie Times New Roman in dem vorliegenden Text). Kursiv sollte man als Schriftauszeichnung verwenden und nicht als Grundschriftschnitt. Ausgefallene Schriften können belebend sein, bei Dauergebrauch wirken sie ermüdend. Die Standardschriftarten

Helvetica und Times mögen „lang gefahrene Reifen“ und schon ein wenig abgenutzt sein, aber es sind saubere und gut lesbare Schriften. Für technische Dokumentationen, die keinem überhöhten ästhetischen Anspruch genügen müssen, sind sie allemal gut.

4. *Formatieren mit Leerzeichen und Zeilenumbruch:* Texte werden mit Absatzeinstellungen und eventuell zusätzlich mit Tabulatoren formatiert und nicht, indem man den Abstand zwischen Absätzen mit Leerzeilen oder die Einrückung mit Leerzeichen setzt.

5. *Falsche Satzzeichen:* Die Begrenzung der Schreibmaschinentastatur bestehen heute nicht mehr. Auch wenn man sich an die Vewendung falsche Satzzeichen gewöhnt hat: es geht besser. Im Deutschen sind beispielsweise nur die Anführungszeichen „“ korrekt. Die Ellipse besteht aus drei Punkten. So geht es also nicht: 1, 2, 3, ... Das Ausrufezeichen sparsam verwenden. Hier gilt dasselbe wie für die Schriftauszeichnung. Höchstens ein Ausrufezeichen in Folge, alles andere ist schlechter Stil!!!

6. *Schlecht gegliederte Texte:* Gestalten Sie Überschriften so, dass sie die Suche unterstützen und das Auffinden erleichtern. Durch unterschiedliche Abstände nach oben und nach unten sollen sie zeigen, zu welchem Abschnitt sie gehören. Vermeiden Sie zu viele Gliederungsstufen. Der *Gliederung* ist vor der eigentlichen Schreibarbeit mit allerhöchster Sorgfalt zu erstellen. Gehen Sie mit der Zeit ihres Lesers sorgfältig um. Aussagestarke Überschriften wählen. Stichwortlisten vorzugsweise linksbündig ausrichten, das kommt den Lesegewohnheiten entgegen und erleichtert das Durchmustern.

7. *Zu viele Stile:* Nicht nur Schriftarten, Schriftschnitt, Schriftgrade oder die Satzausrichtung sind typographische Stilelemente, sondern ebenso die Strichstärke der Linien, die zwischen Textspalten, als Abgrenzung zwischen Abschnitten oder in Tabellen benutzt werden. Auch die Art, wie Strichzeichnungen angelegt, wie Abbildungen und Tabellen im Gestaltungsraster platziert werden, sind *Stilelemente*. Gehen Sie mit diesen so sparsam um, wie mit den Schriftstilen.

8. *Zu wenig Rand und zu wenig weißer Raum:* Zu volle Seiten bieten ein unharmonisches Bild. Denken Sie auch an den Leser, der sich Randnotizen machen will. Es ist ein Kompromiss zu finden zwischen dem Anspruch der harmonischen Seitengestaltung und dem Bestreben, zusammengehörige Information möglichst auf einer Seite unterzubringen.

9. *Falsche und hässliche Trennungen:* Auf korrekte Trennung achten. Hier machen automatische Trennungsprogramme noch viele Fehler. Notfalls zur weichen Silbentrennung (in Word mit der Tastenkombination „<Cntrl> -“) übergehen. Zu viele Trennungen innerhalb eines Abschnittes vermeiden (maximal drei). Zu große Lücken wirken wie eine optische Trennung. Zwischen zwei Wörtern und hinter Satzzeichen steht genau ein Leerzeichen. Vor den Satzzeichen Doppelpunkt, Komma, Semikolon und Punkt sind Leerzeichen fehl am Platze.

10. *Falsche Sicherheit:* Die „Sinnsuche des Wahrnehmungsapparats“ bewegt uns dazu, auch im Unsinn noch Sinn zu erkennen. Durch Gedrucktes wird diese *Denkfalle* noch gefährlicher. Ein einigermaßen korrekt gestaltetes und auf dem Laserdrucker ausgegebenes Dokument beeindruckt fast jeden. Der optische Eindruck „wie gedruckt“ wiegt einen sehr leicht in der falschen Sicherheit, das Dokument sei bereits perfekt. Auch ein eindrucksvolles Papier kann noch viele typographische, sachliche und orthographische Fehler enthalten.

12 Paradigmata der Programmierung

> Diese *Werke* dienten eine Zeitlang dazu, für nachfolgende Generationen von Fachleuten die anerkannten Probleme und Methoden eines Forschungszweigs zu bestimmen. Sie vermochten dies, da sie zwei wesentliche Eigenschaften gemeinsam hatten. Ihre Leistung war *neuartig* genug, um eine beständige Gruppe von Anhängern anzuziehen, die ihre Wissenschaft bisher auf andere Art betrieben hatten, und gleichzeitig war sie auch noch *offen* genug, um der neuen Gruppe von Fachleuten alle möglichen ungelösten Probleme zu stellen. Leistungen mit diesen beiden Merkmalen werde ich von nun an als „Paradigmata“ bezeichnen.
>
> Thomas S. Kuhn, 1967
> Die Struktur wissenschaftlicher Revolutionen

„Top Down“ oder „Bottom Up“?

Das reine Top-Down-Vorgehen ist gekennzeichnet durch das *Denken vom Resultat her:* Ausgehend von einer Spezifikation mittels Vorbedingung (precondition) und Nachbedingung (postcondition) wird die Aufgabe in Teilaufgaben zerlegt, für die wiederum Spezifikationen in derselben Weise erstellt werden.

Ein Programm oder Programmabschnitt erfüllt die Spezifikation, wenn es für die Variablen Werte liefert, die die Nachbedingung erfüllen, vorausgesetzt, unmittelbar vor Eintritt in das Programm oder den Programmabschnitt haben die Variablen die Vorbedingung erfüllt.

Durch diese *schrittweise Verfeinerung* erhält man schließlich Aufgaben, die sich durch elementare Programmbausteine lösen lassen.

Als Darstellungsmittel für Programme auf allen Stufen des Prozesses der schrittweisen Verfeinerung verwenden wir einen aus C abgeleiteten Pseudocode nach dem Muster

```
init;
while (!ende) rechne;
```

Wobei die kursiv bezeichneten Programmteile weiter zu spezifizieren sind. Ist beispielsweise *I* eine Invariante der obigen Schleife und soll init die Invariante unter allen Umständen, also für jeden vorher bestehenden Zustand, herstellen, dann lauten die Spezifikationen der Programmteile *init* und *rechne* folgendermaßen:

```
pre(init) = true
post(init) = I

pre(rechne) = !ende && I
post(rechne) = I
```

Nach Durchlaufen der Schleife gilt als Nachbedingung für den gesamten Programmabschnitt *ende && I.*

Das Top-Down-Vorgehen steht im Einklang mit den Qualitätsanforderungen Zuverlässigkeit, Benutzerfreundlichkeit, Sicherheit und Effizienz. Aber sie steht im Widerspruch zu den Qualitätsanforderungen Wiederverwendbarkeit, Erweiterbarkeit und Änderbarkeit.

Beim Bottom-Up-Vorgehen wird mit den Teilen begonnen. Erst wenn die fertig sind, wird die nächst höhere Integrationsebene ins Auge gefasst. So wird das Ganze aus den Teilen zusammengesetzt.

Das geht schlecht, wenn alle Teile Sonderanfertigungen sind. Mit Standardbausteinen hingegen klappt das ganz gut (siehe Lego). Das Bottom-Up-Vorgehen hat gerade bei den zuletzt genannten Qualitätsanforderungen seine Stärken. Sowohl das Top-Down- als auch das Bottom-Up-Vorgehen gehören zum Software-Entwurf. So lassen sich die Nachteile beider Strategien weitgehend vermeiden.

Die Besonderheiten einer Programmiersprache zusammen mit der zugehörigen Programmierumgebung und dem dadurch geprägten Programmierstil bezeichnet man heute auch als *Paradigma.* Die zwei im vorliegenden Informatikkurs relevanten Programmier-Paradigmata sind

2. *Imperative Programmierung:* Algorithms + Data Structures = Programs. Sprachen: Pascal, FORTRAN, COBOL, Algol, C
2. *Objektorientierte Programmierung:* Programm = strukturiertes Ensemble von Klassen. Klasse = Attribute + Methoden + Axiome. Sprachen: Simula, C++, Eiffel, Object Pascal, Smalltalk, Java

Die imperative Programmierung betont das Top-Down-Vorgehen - die objektorientierte Programmierung das Bottom-Up-Vorgehen.

Konzepte der imperativen und der objektorientierten Programmierung

Die heute dominierenden *imperativen Programmiersprachen* hat Niklaus Wirth in einem seiner Buchtitel treffend charakterisiert:

Algorithms + Data Structures = Programs

Die Trennung von Algorithmen (Funktionen, Prozeduren) und Datentypen ist vorherrschend. Der Entwurf beschäftigt sich vornehmlich mit den Algorithmen. Die Datenstrukturen werden zwar parallel mitentwickelt, spielen aber eher eine Nebenrolle. Diese Vorgehensweise ist außerordentlich erfolgreich vor allem in solchen Fällen, wo man es mit nur wenigen und einfachen Datentypen zu tun hat, die wenig Arbeit machen.

In technischen Systemen herrschen einfache Schnittstellenbeziehungen mit Variablen vom ganzzahligen oder vom reellen Typ vor. Die Variablenmengen sind konstant und die Schnittstellenbeziehungen starr; kurz: die Struktur ist statisch. Dafür sind die Funktionen meist recht kompliziert. Für die Simulation solcher Systeme eignen sich imperative Programmiersprachen wie Algol, Pascal und Modula-2 besonders gut.

Für die Programmierung mit imperativen Programmiersprachen hat sich ein eigener Programmierstil entwickelt: die *strukturierte Programmierung*.

Die strukturierte Programmierung geht von konkreten Anforderungen an ein gewünschtes System aus, und in einem ersten Schritt werden in einer möglichst formalen und vollständigen Spezifikation genau diese Anforderungen präzisiert. Aufgabe ist nun, ein Programm (allgemeiner: ein System) zu bauen, das genau diesen Anforderungen genügt. Nach den Regeln der strukturierten Programmierung geschieht das in einem Prozess der *schrittweisen Verfeinerung*. Dabei wird die Gesamtaufgabe in Teilaufgaben solange immer weiter heruntergebrochen, bis diese sich durch überschaubare Module lösen lassen.

Die Entwicklung des Systems ist dabei fest auf das zu erreichende Ziel ausgerichtet. Der Blick auf künftige Anwendungen der so entstehenden Programm-Module würde den Entwicklungsprozess stören. Der bei dieser Methode vorherrschende *Top-Down-Entwurf* ist also grundsätzlich steif gegenüber Änderungen und Erweiterungen, und er fördert nicht die Ent-

wicklung wieder verwendbarer Software. Diesen Nachteil der strukturierten Programmierung hat Meyer (1989) sehr pointiert herausgestellt.

Das strukturierte Programmieren lässt sich als ein Versuch auffassen, die Kluft zwischen unserer Art zu denken und den Erfordernissen der Programmiersprachen zu überbrücken. Lässt sich diese Kluft vielleicht dadurch verringern, dass man die Programmiersprache dem natürlichen Denken näher bringt?

Zunächst einmal ist herauszufinden, in welchen Punkten die imperative Programmierung unserem Denken widerstrebt. Hinweise darauf gibt uns die Gestaltpsychologie: Beim Suchen nach neuen Lösungen wird unser Denken von den sogenannten *Einstellungen* geleitet. Das ist eine Ausrichtung des Denkens, die in vielen Fällen hilfreich ist, die aber zuweilen das Problemlösen auch erschwert.

Ein bestimmter *Einstellungseffekt,* die sogenannte *funktionale Gebundenheit,* wird im *Halsketten-Problem* deutlich (Bild 12.1): Gegeben sind vier jeweils dreigliedrige Ketten. Daraus ist eine geschlossene Halskette herzustellen. Das Öffnen eines Kettenglieds kostet 10 Cent und das Schließen 15 Cent. Die Gesamtkette soll höchstens 75 Cent kosten.

Wenn Sie das Problem selbst lösen wollen, unterbrechen Sie die Lektüre dieses Aufsatzes und tun sie das jetzt.

Die Lösung - nämlich eine Kette vollständig zu zerlegen und mit den drei Gliedern die anderen drei Ketten zu verbinden - fällt vielen entweder überhaupt nicht oder verblüffend spät ein. Meist schlägt man sich lange mit dem untauglichen Lösungsversuch herum, indem man ein Endglied jeder Kette öffnet und die nächste Kette dort einhängt.

Die Denkoperationen zur Veränderung des Problemzustands scheinen mit dem Gegenstand verbunden zu sein und sie bilden zusammen mit dem Gegenstand das *Objekt* der Betrachtung. Die Kettenstücke des Problems bilden solche Objekte, die im Wesentlichen durch das *Attribut* „dreigliedrig“ und die *Methode* „Verknüpfen“ charakterisiert sind.

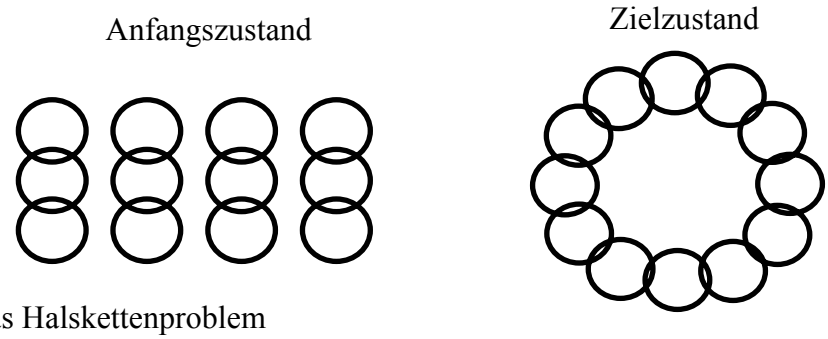


Bild 12.1 Das Halskettenproblem

Objektorientiert wollen wir eine Art zu denken nennen, bei der die Denkoperationen (Methoden) zur Veränderung des Problemzustands mit den Gegenständen verbunden sind. Werkzeuge zusammen mit den gelernten Gebrauchsanweisungen sind Objekte in diesem Sinne (Grams, 1992).

Dieses objektorientierte Denken, bei dem zunächst die mit dem betrachteten Objekt verbundenen Methoden aktiviert werden, ist ein sehr nützlicher Mechanismus. Er ermöglicht in alltäglichen Problemlagen das schnelle Auffinden von Lösungen.

Dass ein solcher Mechanismus hin und wieder versagt, ist nahe liegend. Er tut es vornehmlich dann, wenn wir ein nicht alltägliches Problem haben. Genau ein solches nichtalltägliches Problem war es ja auch, das uns das Aufdecken des Mechanismus überhaupt erst ermöglicht hat.

Beim objektorientierten Programmieren steht die Modellierung von Gegenständen zusammen mit ihren Funktionen und Eigenschaften im Zentrum. Ein solches Modellierungsschema heißt in der Welt der Programmierung *Klasse* oder auch *Objekttyp*. Die Gegenstände oder Objekte selbst sind die konkreten Realisierungen der jeweiligen Klasse. Statt Objekt sagt man deshalb auch Exemplar (englisch: Instance).

Die fundamentalen Konzepte der objektorientierten Programmierung sind

- *Modularisierung*
- *Klassenkonzept* (Kapselung von Attributen und Methoden)
- *Vererbung*
- *Polymorphismus*
- *Geheimnisprinzip (Information Hiding)*

Objektorientierte Programme sind strukturierte Ensembles von Klassen. Klassen (Objekttypen). Die grundlegenden Bestandteile einer Klasse sind ihre *Attribute* und ihre *Methoden.* Später wird es sich noch als sinnvoll erweisen, wenn wir zur Klasse auch noch die Gesetzmäßigkeiten (Axiome) rechnen, die für den Umgang mit den Objekten der Klasse gelten:

Klasse = Attribute + Methoden + Axiome

Die Verschmelzung von Algorithmen und Datentypen zu einem Objekttyp heißt *Kapselung.*

Anmerkungen zur Programmiermethodik: Die Methode von Abbot

Die einschneidendste Veränderung beim Übergang von der imperativen zur objektorientierten Programmierung erfährt die *Programmiermethodik*: Es gibt neue Antworten auf die Fragen, wie große Projekt zu strukturieren und wie umfangreiche Programmieraufgaben anzupacken sind.

Die strukturierte Programmierung wird durch die neue Methodik nicht obsolet. Für das Programmieren im Kleinen, für das Erstellen von Methoden, stellt die strukturierte Programmierung die unerlässliche Grundlage dar.

Aber es wird heute nicht mehr versucht, diese Methodik auf das Programmieren im Großen zu übertragen. Etwas anderes von dem, was wir bei den bisherigen Übungen gesehen haben, lässt sich aber sehr wohl übertragen: *Das Lernen von Vorbildern und das Bestreben, sie zu übertreffen.*

Durch die verbesserte Wiederverwendbarkeit von Software muss heute nicht mehr alles von Grund auf entwickelt werden. Heute ist die vorrangige Tätigkeit des Programmierers nicht mehr das Schreiben von Texten, sondern das Lesen: Er durchstöbert Klassenbibliotheken nach verwendbaren Klassen und Modulen. Dann fügt er diese - mit möglichst wenigen Erweiterungen und Ergänzungen - so zusammen, dass das gestellt Problem gelöst wird.

Dazu muss der Programmierer die folgenden Fähigkeiten erwerben:

- Grundmuster (Pattern) vorhandener Lösungen erkennen
- den Anwendungsbereich der Lösungen erweitern (Generalisierung)
- die verallgemeinerten Lösungen auf vorgegebene Probleme zu übertragen (Spezialisierung)

Genau genommen ist das der Verzicht auf eine besondere Programmiermethodik. Denn das, was hier gefordert wird, macht jede schöpferische Tätigkeit aus. Das neue Paradigma bringt die Programmierung unserer normalen Art zu Denken wieder näher. Es genügt, die alten Rezepte anzuwenden. Das ist ein Trend „Zurück zur Natur".

Abbot hat einige Faustregeln für die Strukturierung objektorientierter Programme angegeben (Pomberger/Blaschek 1993, S. 120 ff.). Ausgangspunkt ist eine Beschreibung der Anforderungen. Diese Aufgabenbeschreibung wird in den folgenden Schritten analysiert.

- Herausfiltern von Hauptwörtern: Gattungsnamen wie Mensch, Auto sind Kandidaten für eine Klasse
- Hauptwörtlich gebrachte Zeitwörter deuten auf Aktionen hin. Sie werden wie die Zeitwörter behandelt
- Relevante Zeitwörter liefern Hinweise auf Aktionen, die durch Methoden realisiert werden können
- Suche nach Gemeinsamkeiten liefert Hinweise auf mögliche Klassenhierarchien

Sachverzeichnis

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z