INFORMATIK IN DER ELEKTROTECHNIK

PROGRAMM-KONSTRUKTION

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

Datei: PrograKo(.doc oder .pdf)
23. März 2006
Erstausgabe: 05.12.96 (Java-Version: 19.02.02)

Beschreibung der Lehrveranstaltung

Gegenstand der Lehrveranstaltung ist das Software Engineering, speziell das ingenieurmäßige Erstellen von Programmen.

Ausgangspunkt ist die *imperative Programmierung*, wie sie Gegenstand der Lehrveranstaltung „Einführung in die Informatik“ ist. Im Zentrum der Lehrveranstaltung steht aber die *objektorientierte Programmierung*, kurz: *ooP*. Sie ermöglicht es, die große Komplexität heutiger Software-Systeme zu meistern. Ergänzt wird diese Methode durch die *visuelle Programmierung*. Sie erlaubt es, Programme schnell und einfach mit anwenderfreundlichen Bedienoberflächen auszustatten.

Ein *Projekt* durchzieht die gesamte Lehrveranstaltung: Die Konstruktion eines blockorientierten Simulationsprogrammes für zeitdiskrete deterministische Systeme. Das Programm ist hinreichend groß, so dass wirklich schon von Programmkonstruktion gesprochen werden kann. Andererseits wird die Aufgabe in Etappen gelöst. Das entspricht dem Software-Engineering der industriellen Praxis. Es entstehen Teilaufgaben, die im Praktikum gut zu bewältigen sind.

Einen Großteil des Kurses nimmt die Einführung in die Programmiersprache Java und in die Programmierumgebung Forte für Java ein. Parallel dazu sind methodische Aspekte der Software-Erstellung zu behandeln. Diese Abschweifungen von der Sprachbeschreibung heißen hier Exkurse.

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` wird verwendet, wenn diese Schrift der Übersichtlichkeit dient: Verbesserung der Unterscheidung zwischen Programmteilen und beschreibendem Text, Verdeutlichung der Programmstruktur durch Einrücken. Die mit einem Sternchen gekennzeichneten Aufgaben und Lektionen sind optional.

Leistungsnachweis: Jeder Lektion sind Programmieraufgaben zugeordnet, die jeder Teilnehmer lösen soll. Gruppenarbeit in Gruppen aus bis zu drei Mitgliedern ist zulässig. Die Erledigung der Aufgaben (abgesehen von der abschließenden Projektdokumentation) wird durch *die Gruppe* auf *einem DIN-A4-Blatt,* auf maximal 2 Seiten also, dokumentiert. Name und Datum nicht vergessen. Alle Gruppenmitglieder nennen. Aufgabenaufteilung muss ersichtlich sein. Dieses Dokument enthält die Aufgabenstellung, die Darstellung der Methode oder des Lösungswegs und die kommentierten Ergebnisses. Bei Nachfrage sind die zugehörigen Programme vorzuführen. Die Erledigung der Aufgaben wird durch Testat auf dem Dokument bestätigt. Termin ist immer in der Vorlesungswoche nach Aufgabenstellung. Zum Testattermin sind auch sämtliche bis dahin testierten Unterlagen bereitzuhalten. Der Leistungsnachweis wird in Fachgesprächen (gruppenweise) am Ende der Lehrveranstaltung (in den Prüfungswochen) erbracht. Die Testate und die Dokumentation des Projekts sind dabei Beurteilungsgrundlage.

Gliederung

Literaturverzeichnis

Hintergrundmaterial

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.*

Gibbs, W. W.: Software: chronisch mangelhaft. Spektrum der Wissenschaft (1994) 12, 56-63

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

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

Grams, T.: Täuschwörter im Software Engineering. Informatik Spektrum 16 (1993) 3, 165-166

Grams, T.: Grundlagen des Qualitäts- und Risikomanagements - Zuverlässigkeit, Sicherheit, Bedienbarkeit. Vieweg Praxiswissen, Braunschweig, Wiesbaden 2001

Hering, E.: Software Engineering. Vieweg, Braunschweig 1989

Hütte (Hrsg.: H. Czichos): Die Grundlagen der Ingenieurwissenschaften. 29. Auflage. Springer-Verlag, Berlin, Heidelberg 1991

Informatik-Duden: Ein Sachlexikon für Studium und Praxis. Bibliographisches Institut & F.A. Brockhaus AG, Mannheim 1988

Kernighan, B. W.; Ritchie, D. M.: The C Programming Language. 2nd edition. Prentice Hall, Englewood Cliffs, New Jersey 1988

Meyer, B.: From Structured Programming to Object-Oriented Design: The Road to Eiffel. Structured Programming (1989)1, 19-39

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

Mills, H. D.: Function Semantics for Sequential Programs. Information Processing 80 (edt.: S. H. Lavington). North-Holland Publishing Company. IFIP 1980, 241-250

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

Wendt, S.: Defizite im Software Engineering. Informatik Spektrum 16 (1993) 1, 34-38

Wirth, N.: Gedanken zur Software-Explosion. Informatik Spektrum 17 (1994) 1, 5-10.

Zemanek, H.: Das geistige Umfeld der Informationstechnik. Springer, Berlin, Heidelberg 1992

Methodik der Software-Konstruktion

Balzert, H.: Lehrbuch der Software-Technik. Software-Entwicklung. Spektrum Akademischer Verlag, Heidelberg, Berlin, Oxford 1996

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

Balzert, H.: UML kompakt - mit Checklisten. Spektrum Akademischer Verlag, Heidelberg, Berlin 2001

Beck, K.: eXtreme Programming explained. Embrace change. Addison-Wesley, Boston 2000

Boehm, B. W.: Software Engineering Economics. Prentice-Hall, Englewood Cliffs (N. J.) 1982

Broy, M.; Denert, E. (Eds.): Software Pioneers. Contributions to Software Engineering. Springer-Verlag, Berlin, Heidelberg 2002

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

DeMarco, T.: Structured Analysis: Beginnings of a New Discipline. In Broy/Denert, 2002

Forbrig, P: Softwaretechnik (Kapitel 5 aus Bruns/Klimsa (Hrsg.): Informatik für Ingenieure kompakt, Vieweg, Braunschweig, Wiesbaden 2001)

Gamma, E.: Design Patterns - Ten Years Later. In Broy/Denert, 2002

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

Guttag, J. V.: Abstract Data Types, Then and Now. In Broy/Denert, 2002

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

Parnas, D. L.: The Secret History of Information Hiding. In Broy/Denert, 2002

Pomberger, G.; Blaschek, G.: Software Engineering. Hanser, München 1993

Shneiderman, B.: Designing the User Interface. Strategies for Effective Human-Computer-Interaction. 3rd Ed. Addison-Wesley, Reading, Mass. 1998

Programmiersprache und Programmierumgebung

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

Doberenz, W.: JAVA. Hanser, München 1996. *Ein sehr guter deutschsprachiger Einstieg in die Java-Programmierung. Signatur Bibiliothek FH Fulda: 31/EDV 599 Java 11*

Flanagan, D.: Java in a nutshell. A Desktop Quick Reference for Java Programmers. *Wer bereits C kennt, findet hier den passenden Einstieg in Java. Handbuch mit vielen Beispielen und Programmtexten (Wer's mag).*

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.*

Oaks, S.; Wong, H.: Java Threads. O'Reilly, Cambridge 1997. *Leichtfasslich und umfassend.*

Sun Microsystems: Forte™ for Java™, Community Edition Software. Opening the Door to Integrated Development. Sun Microsystems, Palo Alto 2000. *Kurzbeschreibung der Integrierten Entwicklungsumgebung (IDE, Integrated Development Environment) Forte für Java mit der Möglichkeit der visuellen Entwicklung von grafischen Bedienoberflächen (GUI, Graphical User Interface).*

Links

`http://java.sun.com` ist die zentrale Referenz für alle Fragen im Zusammenhang mit Java. Hier finden Sie das Java Software Development Kit (Java SDK). Es enthält unter anderem die Java Sprachbeschreibung (Java Language Specification) und das API. API steht für Application Programming Interface und bedeutet soviel wie Allgemein verwendbare Klassen und Funktionen zur Verwendung durch den Anwendungsprogrammierer.

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

1 Der Software-Lebenszyklus

Der gute Wille ist keine Entschuldigung für schlechte Arbeit.

Winston Churchill

Man sollte gleich zu Beginn das Ende des Projekts im Auge haben.

Folklore

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, 1996, S. 25 ff.):

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...100 mal 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.

Interessant ist in diesem Zusammenhang auch, dass Systemversagen zunehmend auf Software-Fehler zurückgeht. Winfried Görke berichtete auf seinem Vortrag anlässlich des Fuldaer Informatik-Kolloquiums 1992 von einer Studie, derzufolge der Anteil der Unterbrechungen aufgrund von Software-Fehlern in den letzten Jahren von 1/3 (1985) auf 2/3 (1989) angestiegen ist. Nach derselben Untersuchung sind in diesem Zeitraum die Unterbrechungen zu Lasten der Hardware-Ausfälle von anteilig etwa 30 % auf unter 20 % zurückgegangen.

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* erfolgt mit Verfahren, die intuitive und diskursive Elemente enthalten.

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)

Hier stellen diese Begriffe einen Bezugsrahmen dar. Erläuterungen dazu findet man beispielsweise im Buch von Pomberger und Blaschek (1993). Erwähnt sei eine Kritik der Begriffe „Zuverlässigkeit“ und „Wartung“ (Grams, 1993): Diese Begriffe sind der Welt der Hardware entnommen und beziehen sich dort auf anfangs intakte Systeme, die durch Änderung der körperlichen Beschaffenheit ausfallen können, beispielsweise aufgrund der Materialermüdung. Demgegenüber wird die „Software-Zuverlässigkeit“ grundsätzlich nur für von Anfang an fehlerhafte Software quantifiziert. Der Ingenieur versteht unter Wartung Maßnahmen zur Bewahrung der Funktionsfähigkeit, wohingegen der Software-Ingenieur damit die Beseitigung von Fehlern nach Inbetriebnahme meint.

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 (Gibbs, 1994; Wendt, 1993; Wirth, 1994). Fast immer sind in diesen Beiträgen auch Rezepte zu finden, wie man der Misere entgehen kann. 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.

Der Software Lebenszyklus (Software Life Cycle)

Eine Variante des *Wasserfallmodells* (Boehm, 1982):

1. Anforderungen
2. Spezifikation
3. Systementwurf
4. Programmentwurf
5. Codierung
6. Validation und Verifikation
7. Installation und Nutzung

Dieses Modell stellt mit seinen Abgrenzungen und dem strikten Nacheinander der Phasen klare Forderungen an die Software-Entwicklung. Aber es ist auch angreifbar. Die Realität sieht nämlich meist ganz anders aus. Weiterentwicklungen des Modells fehlt es aber oft an der Klarheit und Aussagekraft dieses ursprünglichen Modells. Man nehme es als Darstellung eines *Ideals*.

Eine Weiterentwicklung, die den imperativen Charakter des ursprünglichen Modells bewahrt und heute übliche Arbeitsweisen in der Softwaretechnik berücksichtigt, ist das folgende - moderate - *Clustermodell* (in Anlehnung an Meyer, 1989). Es wird Clustermodell genannt, weil im heutigen Software-Entwicklungsprozess die Tätigkeiten Entwurf und Codierung oft nicht klar getrennt werden. Sie werden demzufolge zur *Programmierung* zusammengefasst. Wir verwenden den Begriff Programmierung in diesem weit gefassten Sinn. Außerdem ist das Modell ergänzt um den Generalisierungsschritt (Siehe die Kästen „Clustermodell nach Tätigkeiten“ und „Clustermodell nach Ergebnissen“).

Clustermodell nach Tätigkeiten

1. Anforderungserfassung	←	6. Generalisierung
↓		↑
2. Spezifizierung		5. Validierung
↓		↑
3 Programmierung	→	4. Verifizierung
3.1 Entwurf		
3.2 Codierung		

Clustermodell nach Ergebnissen

1. Pflichtenheft	←	6. Bausteinbibliothek
↓		↑
2. Spezifikation		5. Validation
↓		↑
3 Programm	→	4. Verifikation
3.1 Architektur (UML-Diagramme)		
3.2 Quelltexte		

Wenn wir das Clustermodell für den Software Lebenszyklus mit den *Produktlebensphasen* aus der Einführung in die Informatik vergleichen, fällt auf, dass es in der Hardware eine klare Trennung der Aufgaben und Verantwortungsbereiche in *Entwurf*, *Implementierung* und *Realisierung* gibt. Sie entspechen in der Industrie den separaten organisatorischen Einheiten *Entwicklung*, *Konstruktion* und *Fertigung*.

Entwurf, Implementierung und Realisierung verschmelzen bei der Software-Herstellung zur *Programmierung*. Damit fallen die Schnittstellen zwischen den Tätigkeitsfeldern weg; ebenso entfallen die Pflichten zur sauberen Dokumentation von Arbeitsergebnissen.

Genau diese Aufhebung der Arbeitsteilung sehen einige Kritiger heutiger Software-Produktionsprozesse als Ursache für die anhaltende Software-Krise. Als weitere Ursache wird

ausgemacht, dass die Software-Ingenieure „das Rad immer wieder neu erfinden“ und sie noch nicht in demselben Maße wie die Hardware-Bauer *Standardbausteine* herstellen und nutzen.

Das und die heute in der Industrie üblichen Vorgehensweisen zeigen, dass es doch notwendig ist, die Programmierung auf wenigstens zwei klar voneinander abgegrenzte Abschnitte aufzuteilen. Sie sind in den Kästen „Clustermodell ...“ als Unterpunkte zur Programmierung eingetragen: Entwurf und Codierung als Tätigkeiten, bzw. Architektur und Quelltexte als Ergebnisse.

Das modifizierte Clustermodell ist das diesem Kurs zugrundeliegende *Vorgehensmodell* der Software-Erstellung. Das den Kurs begleitende Projekt wird in genau diesen Stufen bearbeitet. Und in der abschließenden Projektdokumentation finden sich die Stufen des Vorgehensmodells als Gliederungspunkte wieder.

Von den Anforderungen zum Pflichtenheft

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 Spezifkation, also die mathematisch hieb- und stichfeste Beschreibung der zentralen Systemkomponenten entwickelt. Das Pflichtenheft sollte also 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*).

Aufgaben

Zu entwickeln ist ein Progamm namens `BlockSim` für die blockorientierte Simulation diskreter, deterministischer, dynamischer Systeme.

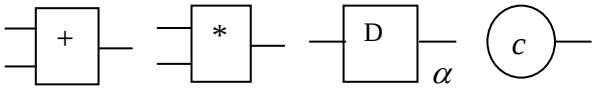


Bild 1.2 Die Bausteine von BlockSim

Der Auftraggeber hat sich folgendermaßen geäußert: „Das Programm soll sehr einfach organisiert sein; die Eingabe soll auf dem *Blockdiagramm* basieren. Für die Beschreibung der Eingabe sollen die Funktionsblöcke Addierer (+), Multiplizierer (*), Verzögerung (*D*) und Konstante (*c*) zur Verfügung stehen“. Dazu hat er Skizzen der Bausteine (Bild 1.2) angefertigt. Den Anfangswert α des Verzögerungsgliedes hat er direkt unter dessen Ausgang geschrieben. Die Dauer der Verzögerung (Delay) ist gleich der Schrittweite (Abtastintervall) *h*.

Zur Erläuterung hat der Auftraggeber ferner eine einfache Schaltung angegeben (Bild 1.3). Sie realisiert am Ausgang *y* eine mit der Zeit *t* linear ansteigende Funktion: Es ist *y* = *t* für Zeiten *t* größer null.

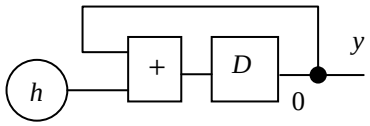


Bild 1.3 Ein Integrierer (Blocksim-Diagramm)

Die Subtraktion wird ebenfalls mit Addierern realisiert, nur dass unter dem zu subtrahierenden Eingang ein Minuszeichen notiert wird.

Mit BlockSim sollen sich einfache Systeme der Nachrichten- und Regelungstechnik simulieren lassen. Zur *Diskretisierung* solcher Systeme wird die kontinuierliche Zeit *t* auf der nichtnegativen Halbachse durch eine Folge von Zeitpunkten t_0, t_1, t_2, ... im Abstand *h* ersetzt. Es ist also: $t_k = k \cdot h$. Für unsere Zwecke reicht es aus, die durch gewöhnliche Differentialgleichungen beschreibbaren dynamischen Systeme auf äußerst einfache Weise zu diskretisieren: Wir ersetzen den Differentialquotienten (d*x*/d*t*) schlicht durch den Differenzenquotienten $(x_k - x_{k-1})/h$.

Wir bezeichnen eine aus der Zeitfunktion *z* hervorgehende Folge wieder mit demselben Buchstaben: $z = (z_0, z_1, z_2, ...)$. Die verzögerte Folge ist definiert durch $Dz = (\alpha, z_0, z_1, z_2, ...)$ mit dem vorgebbaren Anfangswert α. Fehlt eine Angabe zum Anfangswert, ist er gleich null zu setzen (Default-Wert).

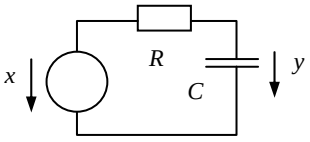


Bild 1.4 RC-Integrierglied

Beispiel: RC-Integrierglied mit anfänglich entladenem Kondensator (Bild 1.4). Die Spannungen *x* (Eingangsgröße) und *y* (Ausgangsgröße) genügen der Differentialgleichung $\tau \cdot \dot{y} + y = x$ mit der durch $\tau = RC$ gegebenen Zeitkonstanten τ. Die Diskretisierung führt auf die Differenzengleichung

$y_k = y_{k-1} + \frac{h}{\tau}(x_k - y_{k-1})$ mit dem Anfangswert $y_0 = 0$. In Folgenschreibweise: $y = Dy + \frac{h}{\tau}(x - Dy)$. Das BlockSim-Schaltbild zeigt Bild 1.5.

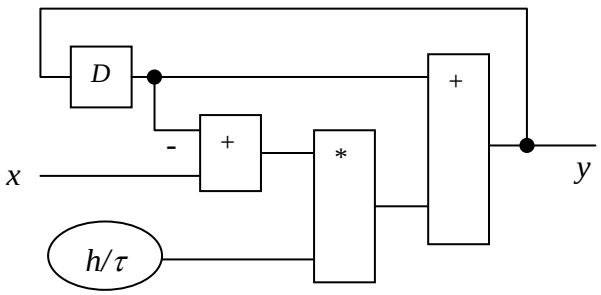


Bild 1.5 BlockSim-Diagramm des RC-Integrierglieds

1.1 Stellen Sie die *Anforderungen* an das Programm BlockSim zusammen. Welcher Art soll die Eingabe sein? Wie werden die Berechnungsergebnisse ausgegeben? Wie steht es mit der Weiterverarbeitung der Ergebnisse? Welche Schnittstellen zu anderen Programmen (beispielsweise Excel) sollen angeboten werden?

Achtung: Übernehmen Sie sich nicht. Die Sache soll im Rahmen der Lehrveranstaltung zu bewältigen sein. Das *Denken von den Grenzen her* ist angesagt. Man kann ja auch nicht einen Luxuswagen zum Preis eines Kleinwagens herstellen.

1.2 Präzisieren Sie die Anforderungen an BlockSim in einem *Pflichtenheft* für BlockSim. Gliedern Sie das Pflichtenheft entsprechend dem EVA-Prinzip (Eingabe/Verarbeitung/Ausgabe). Zu den zu beschreibenden Aufgaben gehören:

- Die Definition einer geeigneten Eingabesprache für Für die Eingabe der Modellstruktur und der Modellparameter.
- Die Festlegung der einzelnen Teilfunktionen der Verarbeitung.
- Die Festlegung der Ausgabe in tabellarischer Form.

Hier geht es nur um die Beschreibung dieser Aufgaben, nicht um die Ausführung.

1.3 Beschreiben Sie das System, das mit dem BlockSim-Modell aus Bild 1.6 simuliert werden soll. Übertragen Sie das Modell in ein Tabellenkalkulationsblatt und stellen Sie die Ausgangsgröße in Abhängigkeit von der Zeit grafisch dar.

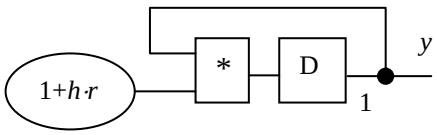


Bild 1.6 BlockSim-Diagramm eines einfachen Systems

1.4 Übertragen Sie das Modell des RC-Integrierglieds (Bild 1.5) in ein Tabellenkalkulationsblatt und führen Sie die Simulation durch. Stellen Sie das Ergebnis grafisch dar.

1.5* Eine geringfügig aufwendigere aber wesentlich genauere Methode der Diskretisierung als die oben vorgeschlagene Polygonzugmethode ist die *bilineare Transformation*. Eine elementare Einführung in die Methode ist auf meiner Web-Seite zur Simulation unter dem Stichwort „Transformationsmethoden für LTI-Systeme“ zu finden. Übertragen Sie das dort hergeleitete BlockSim-Schaltbild für das RC-Integrierglied in ein Blatt der Tabellenkalkulation und vergleichen Sie das Ergebnis mit dem der vorigen Aufgabe und mit der exakten Lösung.

2 Paradigmen 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
> Die Struktur wissenschaftlicher Revolutionen
> 1967

Die Spezifikation

Der Begriff der Spezifikation ist nicht auf die Software beschränkt. Wir betrachten im Folgenden beliebige *komplexe Systeme*, also insbesondere auch digitale Rechensysteme zusammen mit den darauf installierten Programmen. Ein komplexes System umfasst Hardware und Software.

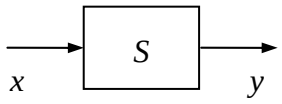


Bild 2.1 Blockschaltbild der Maschine (des Systems)

Mit *S* wird das System bezeichnet; *x* ist die Eingangsgröße und *y* die Ausgangsgröße des Systems. Die Größen *x* und *y* können Vektoren oder vektorwertige Zeitfunktionen sein, aber auch andere zusammengesetzten Datentypen und deren Werte über der Zeit sind denkbar. Umgebungsdaten sind - soweit nötig - als Komponenten in diesen Vektoren enthalten.

Das Verhalten des konkret vorliegenden Systems wird durch die Ein-Ausgaberelation *S* beschrieben: $(x, y) \in S$ heißt, dass der mögliche Eingabewert *x* bei diesem System die Ausgabe *y* zur Folge haben kann.

Der Definitionsbereich einer Relation *S* wird hier mit dom(*S*) bezeichnet:

$$\mathrm{dom}(S) = \{x \mid \text{ Es gibt ein } y, \text{ so dass } (x, y) \in S\}$$

Der Einfachheit halber setzen wir voraus, dass das betrachtete System deterministisch ist. Bei *deterministischen Systemen* gibt es zu jedem zulässigen Eingabewert *x* genau einen Ausgabewert *y*. Dann handelt es sich bei S um eine *Funktion* im mathematischen Sinn und man schreibt $y = S(x)$ anstelle von $(x, y) \in S$.

Vor- und Nachbedingung sind Boolesche Funktionen (Prädikate), also Funktionen mit dem Wertebereich {wahr, falsch}. Sie bilden die *Spezifikation* eines Systems.

Korrektheit bedeutet dasselbe wie *Funktionsfähigkeit.* Das ist die Eignung, eine geforderte Funktion unter vorgegebenen Anwendungsbedingungen zu erfüllen.

Die Vorbedingung eines Systems oder Programmabschnitts *S* bezeichnen wir mit pre(*S*) und die Nachbedingung ist post(*S*). Sei *R* die Menge aller zulässigen Ein-Ausgabepaare. Daraus lässt sich die Spezifikation in Prädikatenschreibweise ableiten: $\mathrm{pre}(S) = (x \in \mathrm{dom}(R))$ und $\mathrm{post}(S) = ((x, y) \in R)$.

Bei Programmen bezeichnet x die Wertebelegung der relevanten Variablen unmittelbar vor Eintritt in den Programmabschnitt mit der Funktion S und y ist die Wertebelegung unmittelbar nach Austritt aus dem Programmabschnitt.

Ein Programm heißt (*vollständig*) *korrekt*, wenn es diese Spezifikation erfüllt, das heißt: Unter der Bedingung, dass pre(S) vor der Programmausführung gilt, endet das Programm in einem Zustand, in dem die Bedingung post(S) erfüllt ist.

Ein Programm heißt *partiell korrekt*, wenn unter der Bedingung, dass pre(S) vor Programmausführung gilt, das Programm entweder kein Resultat liefert, oder aber ein Resultat liefert, das die Nachbedingung post(S) erfüllt. Der Begriff ist nützlich, da es den Nachweis der vollständigen Korrektheit aufzuspalten gestattet 1. in einen Beweis der partiellen Korrektheit ohne Rücksichtnahme beispielsweise darauf, ob eine Schleife endet oder nicht, und 2. in den Nachweis, dass das Programm unter der Vorbedingung terminiert.

Die Spezifikation eines Systems S, das als Eingangsgröße den Vektor $x = (x_1, x_2, ..., x_n)$ hat und daraus den Ausgabewert y als Summe der Komponenten dieses Vektors berechnen soll, sieht beispielsweise so aus:

$$\text{pre}(S) = \text{true}$$
$$\text{post}(S) = (y = \Sigma_{1 \le i \le n}\ x_i)$$

Hier haben wir es mit Variablen im mathematischen Sinn zu tun. Wenn wir auf Programmvariablen übergehen ($x \rightarrow$ x, $y \rightarrow$ y, $n \rightarrow$ n), müssen wir die Tatsache, dass der Vektor x (Programmvariable!) durch S nicht verändert wird, in die Spezifikation aufnehmen:

$$\text{pre}(S) = \text{Zustand gleich } \alpha$$
$$\text{post}(S) = (y = \Sigma_{1 \le i \le n}\ x[i];\ x=\alpha(x),\ n=\alpha(n))$$

Verifikation und Validation: Programmbeweis, Test

Ob ein System die *Anforderungen* erfüllt, wird durch die *Validation* festgestellt. Der Nachweis, dass ein System korrekt ist (die Spezifikation erfüllt), heißt *Verifikation.* Also: Die Validation ist der Nachweis, dass man *das richtige System baut* und die Verifikation ist der Nachweis, dass man *das System richtig baut*.

Ein *Test* ist die Überprüfung, ob ein (deterministisches) System S auf sämtliche Daten einer Teilmenge T der Menge der Eingabedaten X korrekt antwortet. Der Test ist genau dann bestanden (negativ), wenn $R(x, S(x))$ für alle $x \in T$ gilt. Ein Test heißt vollständig, wenn $T = \text{dom}(R)$. Die Verifikation ist nur bei deterministischen Systemen und bei sehr kleinem Eingaberaum dom(R) mit einem vollständigen *Test* möglich.

Bei *nichtdeterministischen Systemen* oder Systemen mit großem Eingaberaum muss man zu Beweistechniken greifen.

Zum *Versagen* kommt es, wenn ein System nicht korrekt ist und sich diese Unkorrektheit aufgrund einer bestimmten zulässigen Beanspruchung (Eingabe) durch nicht spezifikationsgemäßes Verhalten offenbart.

Eingabewerte außerhalb von dom(R) stellen Überbeanspruchungen oder Missbrauch dar. Über das Systemverhalten wird in solche Fällen nichts gesagt.

„Top Down“ oder „Bottom Up“?

Das reine Top-Down-Vorgehen ist gekennzeichnet durch das *Denken vom Resultat her:* Ausgehend von der Spezifikation wird die Aufgabe in Teilaufgaben zerlegt, für die wiederum Spezifikationen erstellt werden. 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
```

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.

Demgegenüber hat das Bottom-Up-Vorgehen gerade bei den zuletzt genannten Qualitätsanforderungen seine Stärken. Sowohl das Top-Down als auch das Bottom-Up gehören zum Software-Entwurf; so lassen sich die Nachteile beider Strategien vermeiden.

Die Besonderheiten einer Programmiersprache zusammen mit der zugehörigen Programmierumgebung und dem dadurch geprägten Programmierstil bezeichnet man heute auch als *Paradigma.* Gängige Paradigmen der Programmierung sind (GI-Fachseminar, 1989):

1. *Imperative Programmierung:* Algorithms + Data Structures = Programs. Sprachen: Pascal, FORTRAN, COBOL, Algol, C
2. *Funktionale Programmierung:* Programm = Kollektion von Funktionen. Typisch: Rekursion. Sprachen: Lisp, Miranda
3. *Logische Programmierung:* Programm = Fakten + Regeln. Prädikatenlogik erster Ordnung. Sprache: PROLOG
4. *Objektorientierte Programmierung:* Programm = strukturiertes Ensemble von Klassen. Klasse = Attribute + Methoden + Axiome. Sprachen: Simula, C++, Eiffel, Object Pascal, Smalltalk, Java

Die imperative Programmierung betont den Top-Down-Entwurf - die objektorientierte Programmierung ist eher Bottom-Up.

Der Algorithmenentwurf geschieht typischerweise Top-Down, und genau darum geht es in den Aufgaben dieses Abschnitts.

Neue Werkzeuge - neue Wege: eXtreme Programming

In der Welt der Informatik ist die Frage nach der richtigen Vorgehensweise und Programmiermethodik Dauerthema. Paradigmen 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 (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 (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.

Grundannahme: 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" (nicht etwa die alte Regel des vorsichtigen Ingenieurs: „Change is Bad").

Ziel ist der möglichst frühzeitige produktive Einsatz eines Systems. Er 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.

Aufgaben

2.1 Gegeben ist die C-Deklaration „`float a[n];`". Schreiben Sie die Spezifikation eines Programms, das die Elemente des Vektors (Arrays) der Größe nach sortiert. Führen Sie geeignete

mathematische Variable und Prädikate ein, die es auszudrücken gestatten, dass der resultierende Vektor a eine Permutation des anfänglichen Vektors sein soll.

2.2 Erstellen Sie für die Verarbeitung des Programms BlockSim eine *Spezifikation*. Das Herzstück des Programms ist ein Algorithmus (eine Schleife), der für jeden Zeitschritt alle Variablen des simulierten Systems aktualisiert (neu berechnet). Definieren Sie für den Verarbeitungsschritt (Schleifenkörper) dieses zentralen Algorithmus eine geeignete *Invariante*, so dass die Korrektheit der Berechnungen gewährleistet ist. Sehen Sie eine Fehlermeldung für den Fall vor, dass die Eingabedaten in sich widersprüchlich sind. Was sind die Bedingungen für die Fehlermeldung?

Zerlegen Sie den Verarbeitungsschritt in die Teilfunktionen `transmit` und `evaluate`. Die Teilfunktion `transmit` sorgt dafür, dass die Ausgangswerte der Verzögerungsglieder neu gesetzt werden: Der Wert des Eingangs wird zum neuen Ausgangswert. Die Teilfunktion `evaluate` sorgt für die Neuberechnung der Verknüpfungen. Der Initialisierungsabschnitt `init` stellt die Gültigkeit der Invarianten unmittelbar vor Schleifeneintritt sicher. Im Falle, dass das nicht funktioniert, soll eine Fehlermeldung ausgegeben werden.

Spezifizieren Sie die Teilfunktionen mittels Vor- und Nachbedingungen.

3 Konzepte der objektorientierten Programmierung

> Selbst langjährige Kenner der Materie sind immer wieder verblüfft,
> wenn ihre Programme bloß durch Hinzufügen einiger neuer Klassen
> um ursrprünglich nicht eingeplante Funktionalität bereichert werden,
> ohne dass die Programme selbst geändert werden mussten.
>
> Pomberger/Blaschek (1993)

Allgemeine Informationen über die objektorientierte Programmierung findet man heute in den Büchern über Software Engineering, beispielsweise im Buch von Pomberger und Blaschek (1993, Kapitel 5). Ansonsten halten wir uns hier an die konkreten Programmiersprachen und Programmierumgebungen für Eiffel, Object Pascal, Oberon und C++. Die objektorientierte Programmierung ist vor allem daran zu messen, inwieweit sie die Beschränkungen der strukturierten Programmierung überwindet und tatsächlich die *Wiederverwendbarkeit, Änderbarkeit und Erweiterbarkeit* von Software fördert.

Einleitung: Ein neuer Programmierstil

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 Entwicklung wiederverwendbarer 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 näherbringt?

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,* kommt beim *Zwei-Seile-Problem* von Maier zum Vorschein (Anderson, 1988): Zwei von der Decke hängenden Seile sollen miteinander verknotet werden; sie sind jedoch so weit voneinander entfernt, dass der „Problemlöser“ nicht beide Enden gleichzeitig erreichen kann. Im Raum befinden sich neben anderen Gegenständen auch ein Stuhl und eine Zange.

Stellt man diese Aufgabe einer Reihe von Versuchspersonen, so wird meist versucht, unter Einbeziehung des Stuhls, beide Seilenden gleichzeitig zu fassen zu kriegen. Die Problemlösung wird entweder gar nicht oder doch verblüffend spät gefunden: Durch Anbinden der Kombizange wird eines der Seile zu einem Pendel gemacht. Die Versuchsperson fasst nun das andere Seil und ergreift zusätzlich das hin- und herpendelnde Seil in einem günstigen Augenblick. Jetzt gelingt es ihr, die Seile miteinander zu verknoten.

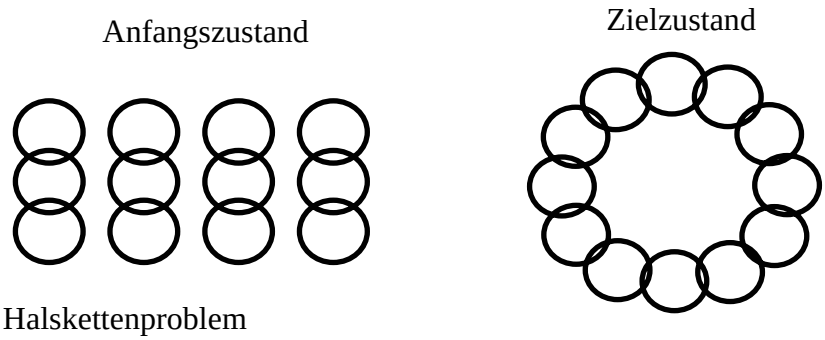


Bild 3.1 Das Halskettenproblem

Die Problemlösung wird dadurch erschwert, dass wir dazu neigen, mit den Gegenständen die damit möglichen Verrichtungen sehr eng zu assoziieren: Wir wissen eben, dass eine Zange dazu da ist, einen Gegenstand zu fassen und zu formen. Uns fällt es schwer, von dieser normalen Funktion des Gegenstands wegzudenken.

Den Effekt der funktionalen Gebundenheit macht auch das *Halsketten-Problem* deutlich (Bild 3.1): Gegeben sind vier jeweils dreigliedrige Ketten. Daraus ist eine geschlossene Halskette herzustellen. Das Öffnen eines Kettenglieds kostet 10 Pfennige und das Schließen 15 Pfennige. Die Gesamtkette soll höchstens 75 Pfennige 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.

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 naheliegend. 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.

Das objektorientierte Denken wird in einem weiteren Modell der kognitiven Psychologie nachgebildet, nämlich in den *Schemata zur Wissensrepräsentation* (Anderson, 1988, S. 120 ff.). Das *Gegenstandsschema* unseres Wissens zum Konzept „Haus“ umfasst

- Attribute wie „aus Stein“, „rechteckig“, „500 m^2“
- die Generalisierungshierarchie: „... ist ein Gebäude“
- die Hierarchie der Teile: Räume - Wände - Fenster

Daneben gibt es offenbar *Ereignisschemata:* Zu unserem Wissen über Restaurants gehört beispielsweise nicht nur das, was sich dort ereignet, sondern auch, dass es sich in der Regel in zusammenhängender Weise und in einer bestimmten Reihenfolge ereignet (Anderson, 1988, S. 128).

Beim objektorientierten Programmieren steht - ganz analog zu den Schemata - die Modellierung von Gegenständen zusammen mit ihren Funktionen und Eigenschaften im Zentrum. Für diese Schemata oder Muster sind in der Welt der Programmierung die Bezeichnungen *Klasse* der auch *Objekttyp* üblich. Die 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)*

Zu den Software-Pionieren der objektorientierten Programmierung gehören Ole-Johan Dahl, David L. Parnas, John V. Guttag und Erich Gamma.

Objektorientierte Programme sind strukturierte Ensembles von Klassen. Klassen (auch: Objekttypen) lassen sich als abstrakte Datentypen, bzw. deren programmtechnische Realisierung, auffassen. 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.*

Programmiermethodik

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“.

Etwas Ähnliches hat der Architekt Christopher Alexander (1977, 1979) für seine Zunft gefordert[1]. Seine Idee einer „Pattern Language“ wurde unter der Bezeichnung „Design Patterns“ von der Informatik aufgegriffen und abgewandelt (Gamma u. a., 1995).

---

[1] "Indeed it turns out, in the end, that what this method does is simply free us from all method" (Alexander, 1979, S. 13). "The proper answer to the question, 'how is a farmer able to make a new barn?' lies in the fact that

Die Methode von Abbot

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

Das Modulkonzept

Ein *Modul*[2] 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. Genaugenommen macht heute ein Hauptprogramm nicht mehr, als das Zusammenspiel der Module auf der höchsten Ebene zu koordinieren. Das Hauptprogramm ist die Wurzel des Modulbaums.

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

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*[3] und die Member-Funktionen heißen *Methoden*. Damit ist also die Möglichkeit einer Kapselung von Attributen und Methoden grundsätzlich bereits in der imperativen Sprache C angelegt. Die folgende C-Deklaration einer Struktur (hier einmal ohne Membervariablen) zeigt diese Technik

```
#define Zustand struct Zustand
Zustand {Zustand *(*Transition)(char e);};
```

---

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).

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

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

Die dynamischen Variablen einer Klasse heißen *Objekte*. Das sind die datenhaltigen Realisierungen der Klasse. Sie werden - wie die dynamischen Variablen der Strukturen - auf dem Heap abgelegt.

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*. 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.

Demonstrationsbeispiel NumberList

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 abspeichert und anschließend über Bildschirm ausgibt. Bild 3.2 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.

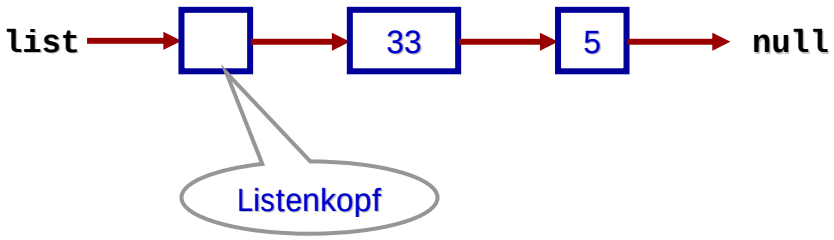


Bild 3.2 Lineare Liste ganzer Zahlen

Die Klasse Number

Die Ergebnisse der Analyse und des Designs sind Objekt- und Klassendiagramme, die nach den Regeln der Unified Modeling Language (UML) gestaltet werden. Die Diagramme werden mit dem Werkzeug GO, das dem Lehrbuch von Balzert beigefügt ist, erstellt[4]. Für dieses ein-

[4] Früher war es einmal Mode, selbst einfache Programmstrukturen grafisch darzustellen, beispielsweise mit Nassi-Shneiderman-Diagrammen. Aber gerade einfache Strukturen brauchen so etwas nicht. Hier schließe ich

fache Beispiel beschränken wir uns auf die Darstellung der *Klassendiagramme* im Designmodus.

Eine Klasse wird in UML als dreigeteiltes Rechteck gezeichnet. Darin stehen

1. der Name der *Klasse*,
2. die *Attribute* (Variablen) und
3. die *Methoden* (Funktionen) der Objekte dieser Klasse.

Bild 3.3 zeigt als Beispiel das Klassendiagramm der Klasse Number.

Das *Klassendiagramm* enthält die folgenden Präzisierungen:

1. Die Typen der Attribute und deren Initialisierung: x und next.
2. Die Typen der Parameter und Rückgabewerte der Methoden: in(...) und write().
3. Die Festlegungen gemäß *Geheimnisprinzip*: Private Attribute und Methoden sind mit einem Minuszeichen markiert. (Sie kommen im Beispiel nicht vor.) Private Elemente sind nur innerhalb einer Klasse und innerhalb deren Objekte sichtbar. Öffentliche (public) Elemente erhalten ein Pluszeichen. Die öffentlichen Elemente bilden die Schnittstelle nach außen. Außerdem gibt es noch die Kennzeichnung mit dem Nummernsymbol #. Es steht für die Geheimnisstufe protected. Attribute und Methoden mit dieser Kennzeichnung sind nicht nur innerhalb der Klasse, sondern auch in allen Subklassen dieser Klasse sichtbar. Der Grad der Öffentlichkeit wächst also von - über # nach +.

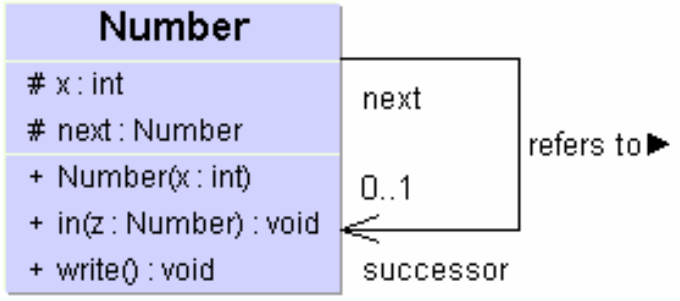


Bild 3.3 Die Klasse Number

4. *Konstruktoren:* Sie 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.
5. Graphische Darstellung der Beziehungen zwischen Objekten: In Bild 3.3 ist eine *Assoziation* zwischen den Objekten der Klasse Number dargestellt: Jedes Objekt der Klasse referenziert über das Attribut next seinen Nachfolger in der linearen Liste. Die Assoziation ist demzufolge ein Pfeil, der von der Klasse Number ausgeht und wieder zurück auf dieselbe Klasse verweist. Durch die Angabe „0..1“ wird hier gesagt, dass es keinen oder einen (direkten) Nachfolger geben kann. Das Ende der Liste wird durch eine Referenz auf das Nullobjekt (null) markiert, das heißt: Es gibt keinen Nachfolger.

Erläuterungen zum Beispiel: 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:

---

mich David L. Parnas an, der meint: „Programme soll man schreiben, nicht zeichnen“ (2002). Die Ausnahme von der Regel: Für die objektorientierte Modellierung empfinde ich die grafische Darstellung UML als hilfreich.

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.

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);
  }
}
```

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.

Bild 3.4 zeigt das ergänzte UML-Diagramm.

Verbindungen mit einem offenen Dreieck stehen für eine Vererbungsbeziehung. Das Dreieck zeigt auf die Oberklasse. Unterklassen *erben* die Attribute und Methoden der Oberklasse und fügen ihnen eventuell weitere hinzu. Methoden der Unterklassen *überschreiben* Methoden der Oberklasse, vorausgesetzt sie haben dieselbe Signatur (identischen Namen und dieselben Parametertypen) und denselben Typ des Rückgabewertes.

Die write-Methode muss durch die 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.

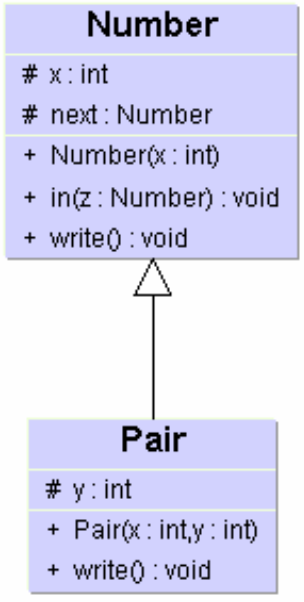


Bild 3.4 Die Klasse Pair als Subklasse von Number

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

```
public class Pair extends Number{
  int y;
  protected 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.

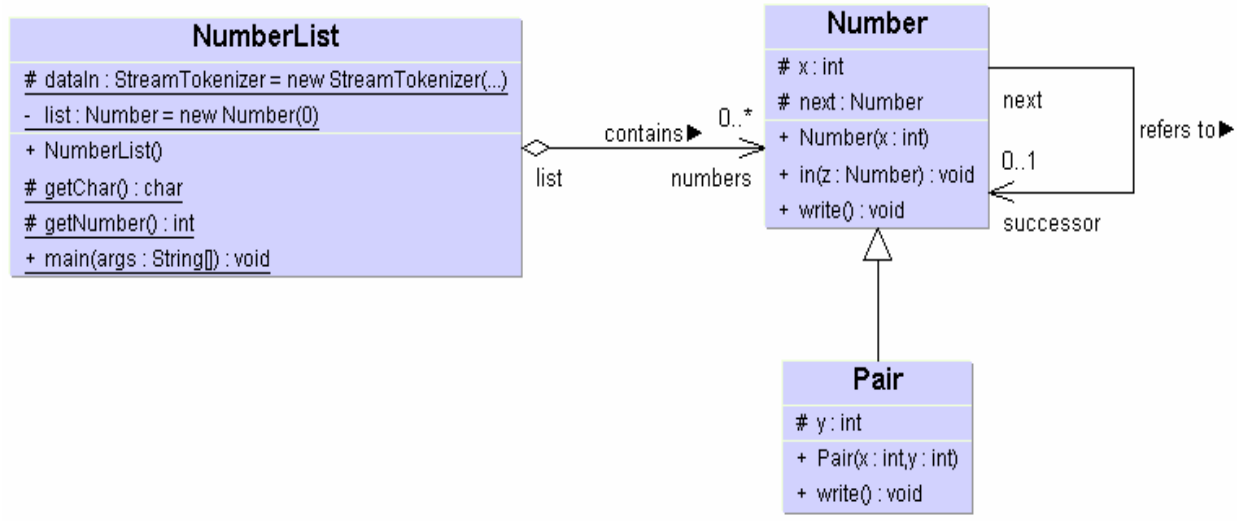


Bild 3.5 Das Modell des Programms NumberList

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 ist veränderlich. Dieser *Polymorphismus* von Referenzen ist von zentraler Bedeutung für die objektorientierte Programmierung.

Der Aufruf einer überschriebenen Methoden (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 am Programm NumberList deutlich, das die lineare Liste aus Zahlen und Paaren aufbaut. Die Programmstruktur ist in Bild 3.5 wiedergegeben. 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 private 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. Statische Attribute und Methoden sind im UML-Diagramm an der Un-

terstreichung zu erkennen. 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.

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.

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 3.6 ist der Bildschirmausschnitt mit den Resultaten eines Programmlaufs wiedergegeben.

```
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 3.6 DOS Bildschirmausschnitt

Assoziationen und Aggregationen

Im Klassendiagramm werden auch gewisse Beziehungen zwischen den Objekten sichtbar. An den Verbindungslinien - soweit sie nicht die Klassenhierarchie betreffen - stehen Angaben über die mögliche Anzahl der Objekte und die Art der Verbindung. Eine offene Raute zeigt eine *Aggregation* an; sie ist auf der Seite derjenigen Klasse angebracht, bei der die „Verantwortung für das Ganze" liegt. Eine Aggregation ist eine Sonderform der *Assoziation* (Bezie-

hung zwischen Klassen). Sie beschreibt, wie sich etwas Ganzes aus seinen Teilen zusammensetzt (Oestereich, 1998 und 2002).

Dass über list sämtliche Elemente der linearen Liste zumindest indirekt referenzierbar sind, wird im UML-Diagramm (Bild 3.5) als *Aggregation* dargestellt. Es wird gezeigt, wie sich das Ganze (hier eine lineare Liste) - aus seinen Teilen (hier den Number-Objekten) zusammensetzt. Durch Angaben wie „0..1“ und „0.. *“ wird gesagt, wie viele Objekte der jeweiligen Klasse an der Assoziation beteiligt sein können. „0.. *“ steht für „beliebig viele“.

Ein weiteres einführendes Beispiel zur objektorientierten Programmierung ist im Web unter

`http://www.fh-fulda.de/~grams/SimMaterial/Lehrgangsverwaltung.ZIP`

zu finden. 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.

Aufgabe

3.0 *Datenabstraktion:* Für einfach verkettete Listen, zwischen deren Elemente eine partielle Ordnung existiert, führen wir die abstrakte Klasse Link als Kopplungselement ein (Bild 3.7). Die Elemente solcher Listen sind Objekte von Subklassen der Link-Klasse. Die Methoden der Klasse Link dienen der Verwaltung der linearer Listen[5]. 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*[6] festgelegt. Die Ordnungsrelation wird erst in Nachkommen von Link definiert. 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*[7] 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.

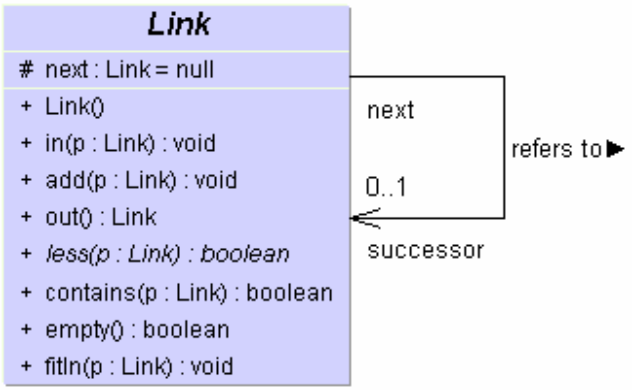


Bild 3.7 OOD-Klassendiagramm der Klasse

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

[6] 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).

[7] 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.

In Java gibt es dafür das Schlüsselwort abstract.

Formulieren Sie die Klassen Number und Pair als Nachfolger von Link. Da die Ordnungsrelation nicht benötigt wird, können Sie sich mit einer trivialen Realisierung begnügen. Auch die Methode fitIn müssen Sie jetzt noch nicht fertig machen. Das Thema wird später wieder aufgegriffen.

Hinweis: Hier wird es notwendig, Typwandlungen (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 Typwandlung „sichtbar machen“ kann. Ob eine Typumwandlung zulässig ist, können Sie vorab mit dem Operator instanceof prüfen. Einzelheiten und genaue Definitionen finden Sie in der Java Sprachbeschreibung.

3.1 Führen Sie einen ersten Entwurf des Programms BlockSim aus. Erstellen Sie mit dem Programm GO ein Objektdiagramm und das OOA-Klassendiagramm.

Modellieren Sie die Klassen für die Funktionsblöcke (Addierer, Multiplizierer, Konstante, Verzögerung) als Nachkommen der Klasse `Link`.

Führen Sie auch für die Verzögerungsbausteine eine eigene lineare Liste ein und eine weitere für die Verknüpfungsbausteine. Zu den Verknüpfungsbausteinen rechnen wir hier auch die Konstanten. Definieren Sie geeignete Zwischenklassen, beispielsweise die Klasse `Operation` als direkte Oberklasse der Klassen `Add`, `Mult` und `Const`. Die Listen können den Klassen `Operation` und `Delay` dann als statische Attribute zugeordnet werden.

Verallgemeinerungen machen zuweilen eine Sache einfacher! Es lohnt sich, den abstrakten „Universalbaustein“ für Verknüpfungen und Delays gleich so zu formulieren, dass er eine beliebige Anzahl von Eingangsvariablen unterstützt: Konstante haben keinen, Verzögerungen haben einen, Addierer und Multiplizierer haben zwei Eingänge. Beim weiteren Ausbau des Systems wird man Addierer, Multiplizierer und weitere Funktionen mit einer beliebigen Zahl von Eingängen einführen. Dann kann man auf diesen Universalbaustein zurückgreifen und seine allgemein verwendbaren Funktionen nutzen.

4* 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 normalerweise in ein Bytecode-Format übersetzt, die von der Java Virtual Machine interpretiert werden kann (*The Java Virtual Machine Specification*, Addison-Wesley, 1996). Dadurch kann grundsätzlich gewährleistet werden, dass eine importiertes Programm keinen Schaden auf der Host-Maschine anrichten kann (Bild 4.1).

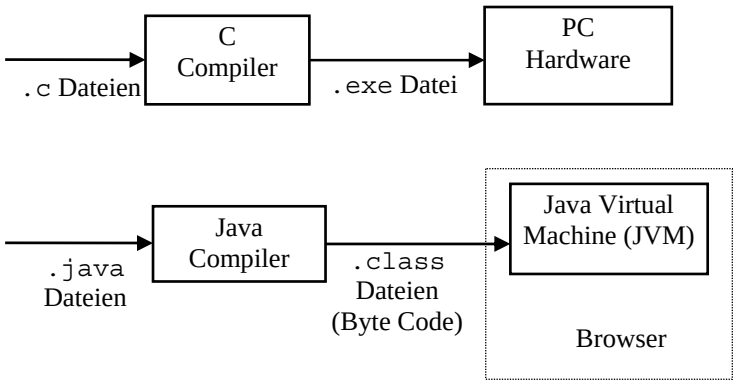


Bild 4.1 Architektur von C und Java im Vergleich

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 werden grundsätzlich 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.

Objekte (Exemplare, Instanzen) von Referenztypen werden grundsätzlich dynamisch erzeugt. Wie bei den Zeigern, könnnen 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. Standard-Klassen existieren auch als Rahmen für einfache Datentypen.

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.

Typ-Umwandlungen (Type Cast) ändern den Typ einer Variablen oder eines Ausdrucks zur Compilierungszeit.

Java-Programme werden als Packages organisiert, ähnlich den Modulen von Modula. Der Internet Domain-Name kann für die eindeutige Kennzeichnung von Packages verwendet werden.

Ein einfaches Applets: Compilation Unit, Hauptklasse

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 verwendeteten 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.

Als Einstiegsbeispiel wird ein einfaches *Applet* gewählt. Ein Applet ist eine spezielle Java-Klasse, die unter der JVM eines Browsers ablaufen kann. Das Programm `ERIK_app` ist eine Übertragung und Erweiterung des Erik-Programms aus der Lehrveranstaltung „Einführung in die Informatik“. Dieses einfache Applet ist über die HTML-Befehle (Tags)

```
<applet code=ERIK_app.class id=ERIK_app width=320 height=100 ></applet>
```

in meine Informatik-Seite eingebettet.

```
import java.io.*;
import java.applet.*;
import java.awt.*;

//Hauptklasse (main class) des Applets ERIK_app
public class ERIK_app extends Applet
{
 private static String s_b, s_z;
 private static int z, b;
 private final static int idle=0, in_z=1, in_b=2, run=3;
 private static int state;

 public void reset() {s_b=""; s_z=""; b=0; z=0;}

 //Die Methode init() wird vom Abstract Window Toolkit (AWT)
 //aufgerufen, wenn ein Applet geladen wird.
 public void init(){resize(320, 100);}

 //Die Methode Start wird nach der init-Methode aufgerufen und
 //immer dann, wenn das Applet erneut besucht wird.
 public void start(){state=idle;}
```

```
 //Die Methode paint wird von der Java Virtual Machine (JVM) immer
 //dann aufgerufen, wenn das Applet angezeigt werden soll.
 public void paint(Graphics g){
 if (state==idle) {
   g.drawString("Hier klicken ...", 10, 10);
   g.drawString("... oder da", 250, 30);
   g.drawString("... oder da", 90, 60);
   g.drawString("... oder da", 170, 90);
  } else {
   g.drawString("Das Programm ERIK", 10, 10);
   g.drawString("wandelt eine Dezimalzahl z ...", 10, 30);
   g.drawString("(Zahleneingabe und Neustart: ...)", 10, 45);
   g.drawString("? z = " + s_z + ((state==in_z)?"_":""), 10, 65);
   g.drawString("? b = " + s_b + ((state==in_b)?"_":""), 10, 80);
   if (state==run) {
    try {
     b=Integer.valueOf(s_b).intValue();
     z=Integer.valueOf(s_z).intValue();
    }
    catch(NumberFormatException e){state=idle;}

    g.drawString((state==run)?
      (b>1)?
        "! " + code(z)
      :"Eingabefehler: Basis muss größer als eins sein"
    :"Eingabefehler: Jede Zahl besteht aus wenigstens einer...",
    10, 95);
   } else /*NULL*/;
  }
 }

 //Rekursive Prozedur zur Wandlung einer Dezimalzahl in einen
 //String, der die Darstellung im Stellenwertsystem zur Basis b
 //enthält.
 static String code(int z)
 {return(z/b!=0?code(z/b):"")+(char)(z%b>9?z%b-10+'A':z%b+'0');}

 //Die Methode mouseDown() wird immer dann aufgerufen, wenn die
 //Maustaste gedrückt wird, während der Mauszeiger sich über dem
 //Applet-Fenster befindet.
 public boolean mouseDown(Event evt, int x, int y){
  if (state==idle) {state++; reset(); repaint();}
  return true;
 }

public boolean keyDown(Event evt, int nKey) {
  if (state==idle)
   if ((nKey=='\r')|(nKey=='\n')) {state++; repaint();}
   else /*NULL*/;
  else {
   if (('0'<=nKey) & (nKey<='9'))
    if (state==in_z) s_z+=(char)nKey;
    else if (state==in_b) s_b+=(char)nKey; else /*NULL*/;
   else if ((nKey=='\r')|(nKey=='\n')) {
    state++; state%=4;
    if (state==idle){state++; reset();}
   } else if (nKey=='\u001B' /*Esc*/) state=idle;
   repaint();
  }
  return true;
 }
}
```

Gemeinsamkeiten und Unterschiede zwischen C und Java

Die Syntax von Java-Ausdrücken entspricht weitgehend derjenigen von C-Ausdrücken. Es genügt, auf die wesentlichen Unterschiede hinzuweisen (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 Auszeichnugn 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 Definition 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.

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* }

Die Komplierungseinheit Erik_app steht in einem unbenannten Package. Deshalb fehlt die PackageDeclaration. Die Quelltextdatei hat den Namen Erik_app.java. Das Übersetzungsergebnis heißt Erik_app.class.

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 ensteht 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 Programmes besteht aus einer Folge von Anweisungen (Operationen), die auf den Variablen 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 Erik-Applet enthält nur eine einzige Klassendeklaration, die Hauptklasse ERIK_app. Details zur Grammatik und zur Bedeutung von Klassen folgen im nächsten Abschnitt.

Aufgaben

4.1 Schreiben Sie den Quelltext des Erik-Applets ab. Übersetzen Sie den Text und lassen Sie die Java-Klasse im Rahmen eines eigenen HTML-Dokuments laufen.

4.2 Explorieren Sie das Programm des Erik-Applets. Versuchen Sie Antworten auf die folgenden Fragen zu finden:

Welche Klassen werden verwendet? (Schauen Sie sich insbesondere die Applet-Klasse genauer an.)

- Wie heißen die Variablen des Programms?
- Wie heißen die Methoden?
- Was leisten die Methoden?
- Welche Methoden sind statisch, welche nicht?
- Geben Sie an, wo das Überschreiben und der Polymorphismus wirksam werden.

Erläutern Sie den Programm-Code Zeile für Zeile. (Nehmen Sie sich für diese Aufgabe ausreichend viel Zeit, auch über die reine Praktikumsstunden hinaus. Geben Sie sich erst zufrieden, wenn Ihnen der Programmtext nicht mehr fremd vorkommt und wenn Sie sich im SDK gut zurechtfinden.)

4.3 Untersuchen Sie die Ausdrücke des Applets und stellen Sie die Gemeinsamkeiten und Unterschiede zu C-Ausdrücken fest. Warum darf in der Methodendeklaration

```
static String code(int z)
 {return(z/b!=0?code(z/b):"")+(char)(z%b>9?z%b-10+'A':z%b+'0');}
```

nicht „z/b“ anstelle von „z/b!=0“ stehen?

5 Klassen und Objekte

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*

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];
```

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 Typwandlung 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

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 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 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.

Aufgaben

5.1 Formulieren Sie für die Zahlen und Zahlenpaare des Programms NumberList Ordnungsrelationen durch geeignete Konkretisierungen der Link-Methode less. 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 tatsächlich eine partielle Ordnung der Punkte definiert wird.

Zwischen einer Zahl und einem Zahlenpaar besteht keine Ordnungsbeziehung. In der less-Methode muss also herausgefunden werden, ob das Vergleichsobjekt eine Zahl oder ein Zahlenpaar ist, um im Falle inkompatibler Typen den Wert false zurückgeben zu können. Das kann mit dem Operator instanceof geschehen.

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

Hinweise. Falls noch nicht geschehen, schreiben Sie jetzt für ihr Link-Klasse eine funktionsfähige Variante der 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 fitIn die *Invariante* „Die Liste q ist topologisch sortiert“ erhält.

5.2 Ändern Sie das Testprogramm so, dass die Ein- und Ausgabe der Punktfolgen über Textdateien geschieht.

6 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 A in Richtung Wurzel an B 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 Package 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

ein direktes Superinterface deklariert oder erbt eine - nicht notwendigerweise abstrakte - Methode und eine Definition dieser Methode wird weder in der Klasse selbst vorgenommen noch 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. Sie enthalten keinen Methodenkörper (siehe weiter unten). Das Schlüsselwort abstract entfällt. Siehe Gosling/Joy/Steele (1996), S. 186.

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 werden stets ohne Referenz auf ein Objekt aufgerufen. Nicht als static deklarierte Methoden heißen Objektmethoden (instance method, auch: non-static-Methode). Objektmethoden werden grundsätzlich 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.

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 all 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 Axiomen aufgestellt werden. Wie der Algebraiker hat auch der Programmierer die Gültigkeit der Axiome für seine Konkretisierungen nachzuweisen.

In der Klasse Link ist das Axiomensystem durch dasjenige der partiellen Ordnung gegeben. Die Funktion `fitIn` liefert immer dann das gewünschte Ergebnis, wenn die Methode `less` allen Objekten der Subklasse tatsächlich eine partielle Ordnung aufprägt.

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 als 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. Beispiel: Die Axiome der partiellen Ordnung, denen die Methode `less` der `Link`-Klasse gehorchen muss.

Die so entstehende (abstrakte) Klasse zusammen mit den Axiomen ist die Implementierung eines *abstrakten Datentyps.* 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* (Meyer, 1988).

Aufgaben

6.1 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?

7 Blöcke und Anweisungen

Die Syntax

Bisher sind wir mit dem C-Wissen, was den Aufbau von Blöcken angeht, zurechtgekommen. Jetzt wird die genaue Beschreibung der Syntax der Blöcke nachgeholt:

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

BlockStatement =
LocalVariableDeclarationStatement | *Statement*

Statement =
StatementWithoutTrailingSubstatement | *LabeledStatement* | *IfThenStatement* |
IfThenElseStatement | *WhileStatement* | *ForStatement*

StatementWithoutTrailingSubstatement =
Block | *EmptyStatement* | *expressionStatement* | *SwitchStatement* | *DoStatement* |
BreakStatement | *ContinueStatement* | *ReturnStatement* | *SynchronizedStatement* |
ThrowStatement | *TryStatement*

IfThenStatement = **if (** *expression* **)** *Statement*

IfThenElseStatement =
if (*expression* **)** *StatementNoShortIf* **else** *Statement*

StatementNoShortIf =
StatementWithoutTrailingSubstatement |
LabeledStatementNoShortIf |
IfThenElseStatementNoShortIf |
WhileStatementNoShortIf |
ForStatementNoShortIf

Anweisungen zerfallen also in zwei Kategorien: solche, die mit einem if-Statement enden, die keine else-Klausel („short if statement“) haben, und solche, bei denen das nicht so ist. Nur Anweisungen (Statements), die nicht mit einem „short if statement“ enden, können als Unteranweisungen vor dem Schlüsselwort else erscheinen[1].

Es folgt eine Auswahl der Anweisungen. Throw- und Try-Statement werden in einem späteren Abschnitt behandelt.

---

[1] Hier sieht man einmal, wie schwer sich manche syntaktischen Dinge formalisieren lassen. Eine einfachere Möglichkeit, die If-then-else-Mehrdeutigkeit aufzulösen, ist bei der Programmiersprache C vorgeführt worden. Im Buch von Kernighan und Ritchie (1988, S. 56) wird die Mehrdeutigkeit mit Hilfe einer Zusatzinformation zur Backus-Naur-Notation aufgelöst: „Jedes else gehört zum letzten vorhergehenden if, das noch keinen else-Teil hat“.

EmptyStatement = **;**

ExpressionStatement = *StatementExpression* **;**

StatementExpression = *Assignment* | *PreIncrementExpression* |
PreDecrementExpression | *PostIncrementExpression* |
PostDecrementExpression | *MethodInvocation* |
ClassInstanceCreationExpression

IfThenStatement = **if (** *expression* **)** *Statement*

IfThenElseStatement =
if (*expression* **)** *StatementNoShortIf* **else** *Statement*

IfThenElseStatementNoShortIf =
if (*expression* **)** *StatementNoShortIf* **else** *StatementNoShortIf*

WhileStatement = **while (** *expression* **)** *Statement*

WhileStatementNoShortIf =
while (expression **)** *StatementNoShortIf*

DoStatement = **do** *Statement* **while (** *expression* **) ;**

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

Beispiel 1: Durch die folgenden Deklarationen

```
class Point { int[] metrics; }
interface Move { void move(int deltax, int deltay); }
```

wird eine Klasse Point und ein Interface Typ Move deklariert. Es verwendet einen Array-Typ `int[]` zur Deklaration eines Feldes `metrics` der Klasse `Point`.

Beispiel 2: Typen werden in Deklarationen oder in gewissen Ausdrücken benutzt.

```
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 Typwandlung in den Typ float.

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

Aufgaben

7.1 Definieren Sie für das Programm BlockSim eine Eingabesprache in der erweiterte Backus-Naur-Form (EBNF).

7.2 Schreiben Sie den Eingabeteil für BlockSim mit folgendem Funktionsumfang:

1. Einlesen der Eingabedaten aus einer Textdatei
2. Erzeugen der Objekte des Modells gemäß Eingabesprache (Parser)
3. Erfassung der Objekte in linearen Listen
4. Verkettung der Objekte: Repräsentation der Verknüpfungen durch Referenzen (Verweis vom Ziel auf die Quelle, entgegen dem Datenfluss zwischen den Blöcken)

Zu Testzwecken wird das Programm um eine Bildschirmausgabe ergänzt. Auf dem Bildschirm sollen die Bausteine getrennt nach Speichern und Verknüpfungen erscheinen (alles in Textform, also ohne grafische Bedienoberfläche).

8 Exceptions, Threads und Packages

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. (In der Catch-Klausel des Erik-Applets wird darauf verzichtet.)

Threads

Threads gestatten es unter anderem, die Ausführung von Applets für bestimmte Zeit zu unterbrechen. Mit Threads lassen sich auch Verarbeitungsprozesse von den Ein-Ausgabe-Vorgängen entkoppeln. Diese Technik wird beispielsweise in der Ereignisorientierten Simulation eingesetzt.

Zu Aufbau und Wirkungsweise von Threads siehe Arnold/Gosling (1996, Kapitel 9, S. 159 ff.). Einen guten und leicht fasslichen Einstieg in die Java-Thread-Programmierung hab ich im Buch von Oaks und Wong (1997) gefunden.

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.

Exkurs: Allgemeine Regeln für die Gestaltung bedienbarer Maschinen

In meinem Buch Grundlagen des Qualitäts- und Risikomanagements (2001) habe ich die allgemeine Regeln für dei Entwicklung bedienbarer Maschinen zusammengestellt. Hier ein Auszug daraus (Quellenangaben findet man im Buch).

Regel 1. Strebe nach *Einfachheit.*

Einfach ist ein System, wenn seine Funktion leicht *überprüfbar* und *verständlich* ist. Das einfache System enthält nur die notwendigsten der geforderten Eigenschaften und Fähigkeiten und ein *Minimum an Komponenten*, und diese sind möglichst *schwach miteinander verkoppelt* Kurz: es ist ein schlankes System.

Regel 2. Sorge für *Direktheit* der Manipulation und Kommunikation. Schaffe eine *selbsterklärende Bedienoberfläche*. Mache die Funktion *sichtbar*.

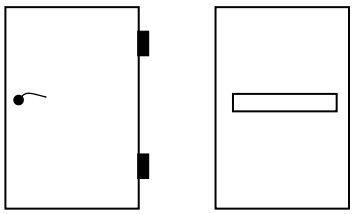


Bild 7.1 Klassische Tür (links) und moderne Tür (rechts)

Die Bedienoberfläche sollte Signale geben, so dass ein adäquates mentales Modell des Artefakts induziert und die korrekten Bedienhandlungen angeregt werden. Ein gutes Beispiel ist „die klassische Tür“: Aufgesetztes Türblatt und Anbringung der Klinke sagen, was zu tun ist. Die „moderne Tür“ hingegen verheimlicht ihre Funktion (Bild 7.1). Das schlechte Design kommt deswegen meist mit einer Bedienungsanleitung daher. Es wird eigens draufgeschrieben, ob man ziehen oder drücken muss. Solche Erläuterungen stellen kommunikative Umwege dar; sie widersprechen dem Prinzip der Direktheit der Manipulation und Kommunikation.

Ein Musterbeispiel für direkte Manipulation aus dem Bereich der Software stellen die Programme der Tabellekalkulation dar (VisiCalc, Excel).

Regel 3. *Bedenke die Gewohnheiten* des Bedieners und Anwenders, nutze seine *Routine* und beachte *Konventionen*.

Um der Leseroutine entgegenzukommen, sollten Stichwortlisten auf Buchseiten, Web-Pages oder Bedienoberflächen eines Computerprogramms grundsätzlich linksbündig ausgerichtet werden.

Zu den Konventionen gehören die *allgemein akzeptierten Codes* für Darstellungs- und Bedienungselemente. Beispielsweise steht im „Farbcode“

- Rot für halt, heiß oder Gefahr,
- Grün für weiter oder sicher,

- Gelb für Vorsicht und
- Blau für kalt.

Wenn Maschinen den Menschen verunsichern, indem sie ihn darüber im Unklaren lassen, was gerade passiert (Fall 5), dann ist das ein Verstoß gegen die

Regel 4. Sorge für die *Rückmeldung* des Erfolgs oder Misserfolgs von Aktionen und für Informationen über den Maschinenzustand.

Regel 5. Achte auf eine zweckmäßige *Aufteilung der Funktionen* auf Mensch und Maschine unter Berücksichtigung der jeweiligen Fähigkeitsprofile.

Die Regel verlangt, das zu automatisieren, was die Maschine besser kann als der Mensch und auf Automatisierung dort zu verzichten, wo dem Menschen durch Automatisierung Tätigkeiten aufgebürdet werden, in denen er nicht gut ist.

Einen ersten Anhaltspunkt für eine solche Aufteilung bietet die Liste der Fähigkeitsprofile von Fitts (Kasten „Liste von Fitts“). Sie ist zu ergänzen, insbesondere um den Hinweis, dass der *Mensch beim Überwachen nicht besonders gut* ist. Eine weitgehende Automatisierung drängt den Menschen aber gerade in die Rolle des Überwachers. Beispiel dafür ist das Autokorrekturprogramm von Word.

Von Lisanne Bainbridge stammt der Begriff von der *Ironie der Automatisierung*: Ein weitgehend automatisiertes System enthält dem Bediener die Gelegenheiten zum Einüben der im Ernstfall wichtigen Fertigkeiten vor. Letztlich wird der Gewinn, den die Automatisierung verspricht, durch das zusätzlich erforderliche Operateur-Training teilweise wieder aufgefressen.

Nancy Leveson stellt folgenden Zusammenhang her: Wenn vom Operateur nur geringe Mitwirkung gefordert wird, kann das zu einer geringeren Wachsamkeit und zu einem übermäßigen Vertrauen in das automatisierte System führen. Und Mica R. Endsley stößt ins gleiche Horn, wenn sie konstatiert, dass das Situationsbewusstsein verloren gehen kann, wenn die Rolle des Bedieners auf passive Kontrollfunktionen reduziert wird.

Felix von Cube sieht die Langeweile als Ursache des Übels: „Dadurch, dass der Unterforderte seine Aufmerksamkeit nicht oder nur zum geringen Teil für seine Arbeit einzusetzen braucht, richtet er sie auf andere Bereiche. So wird sie unter Umständen ganz von der Arbeit abgezogen, es kommt zu gefährlichen Situationen.“

Der richtigen Funktionsaufteilung gilt das Hauptaugenmerk bei der Cockpitgestaltung von Flugzeugen. Entsprechend gut ist dieses Thema untersucht.

Regel 6. Nutze die natürlichen Ordnungsprinzipien und die Gestaltgesetze, insbesondere das Gesetz der Gruppierung und der Nähe. Ordne die Funktionselemente und deren Bedienungselemente abbildungstreu einander zu.

Wenn das alles nicht ausreicht, oder wenn das Learning by Doing und die Exploration des Systems angeregt werden sollen, tue ein Übriges entsprechend der

Regel 7. Schaffe die verzeihende Bedienoberfläche. Erlaube das Rückgängigmachen von Fehleingaben.

Liste von Fitts	
Menschen sind heutigen Maschinen überlegen hinsichtlich der Fähigkeiten	Heutige Maschinen sind Menschen überlegen hinsichtlich der Fähigkeiten
• geringe Mengen visueller oder akustischer Energie aufzuspüren, • optische oder akustische Muster wahrzunehmen, • zu improvisieren und flexibel zu reagieren, • große Mengen Information über lange Zeit zu speichern und sich zur passenden Zeit an die relevanten Fakten zu erinnern, • der Induktion, also Erweiterungsschlüsse zu machen, • Entscheidungen zu fällen.	• schnell auf Steuerungssignale zu reagieren und große Kraft präzise einzusetzen, • Routineaufgaben durchzuführen, • Information kurzfristig zu speichern und auf Aufforderung hin vollständig zu löschen, • logische und arithmetische Kalkulationen durchzuführen, • hochgradig komplexe Operationen durchzuführen, das heißt, viele verschiedene Dinge auf einmal zu tun.

Aufgaben

8.1 Definieren Sie durch Konkretisierung der `less`-Methode eine partielle Ordnung für die Verknüpfungsbausteine, so dass nach einer topologischen Sortierung der Verknüpfungsbausteine deren Berechnung in der Reihenfolge der Auflistung durchgeführt werden kann. Unter der Vorbedingung, dass die Verknüpfungsbausteine topologisch sortiert sind, findet man eine sehr einfache und effiziente Realisierung für das Programmteil *`evaluate`*. Geben Sie diese Realisierung an.

8.2 Eine Sortierung der Verknüpfungsbausteine analog zur Sortierung der Punkte der `Number`-Klasse unter Zuhilfenahme der `less`-Methode lässt sich nur mit großem programmiertechnischem Aufwand erreichen. Stellen Sie fest, auf welche Schwierigkeiten die Programmierung der `less`-Funktion stößt.

8.3 Entwickeln Sie für die Verarbeitungsbausteine eine Alternative zur topologischen Sortierung mittels less. *Hinweise:* Die Liste der Bausteine wird von vorn nach hinten aufgebaut, indem man aus der Menge der noch nicht eingefügten Bausteine einen auswählt, dessen Eingganggrößen entweder durch Eingabedaten bzw. Verzögerungsbausteine definiert sind oder durch bereits in die Liste eingefügte Bausteine geliefert werden. Auf diese Weise kann man zunächst alle Konstanten in der Liste unterbringen. Führen Sie für die Verknüpfungsbausteine eine boolesche Methode `predContainedIn` ein. Sie hat als Eingabeparameter eine Liste von Verknüpfungsbausteinen und liefert genau dann den Werte `true`, wenn alle Verknüpfungsbausteine, deren Ausgangsgrößen von dem aufrufenden Baustein benötigt werden, in der Liste enthalten sind. Definieren Sie `topologicalSort` als boolesche Funktion, so dass eine Signalisierung des Scheiterns der Sortierung möglich ist.

8.4 Beweisen Sie, dass die `topologicalSort`-Methode genau dann erfolgreich sortiert, wenn das zu simulierende System selbst in sich widerspruchsfrei ist und beispielsweise keine

Schleifen aus Verknüpfungsbausteinen enthält. Sehen Sie eine geeignete Reaktion bei erfolglosem Abbruch der Sortierung vor. *Hinweis:* Führen Sie einen indirekten Beweis.

8.5 Für die Liste der Verzögerungsbausteine gibt es nicht die Möglichkeit einer topologischen Sortierung. Warum nicht? Wie lässt sich die korrekte Funktion des Programmteils `transmit` sicherstellen? Denken Sie an den Trick, der beim Master-Slave-Flip-Flop angewendet worden ist, um Eingänge und Ausgänge voneinander zu entkoppeln.

8.6 Überarbeiten Sie die Spezifikation Ihres Programms unter Berücksichtigung der topologischen Sortierung.

8.7 Überarbeiten Sie den Entwurf des Programms BlockSim (OOA-Diagramm). Erstellen Sie das OOD-Diagramm und erstellen Sie daraus den Quellcode.

9 Die Integrierte Entwicklungsumgebung (IDE)

Heute gibt es weit entwickelte integrierte Programmierumgebungen (Integrated Development Environment, IDE). Sie ermöglichen einen einheitlichen Zugang zu den Entwicklungshilfsmitteln (Tools) einer Sprache:

1. Text-Editor, möglichst mit Syntax-Prüfung bereits während der Eingabe,
2. Editor für die Gestaltung von Bedienoberflächen (Graphical User Interface, GUI) durch direkte Manipulation der Grafikelemente (Drag-and-Drop-Technik),
3. Compiler,
4. Debugger,
5. Projekt- und Versionenverwaltung,
6. Laufzeitumgebung.

Solche Werkzeuge sind für eine leistungsfähige Software-Produktion heute unentbehrlich.

Forte für Java

Für Java gibt es inzwischen eine IDE, deren Basisversion frei erhältlich ist: die „Forte for Java, Community Edition“. Sie ist leicht erweiterbar. Die nächsthöhere Stufe ist die „Internet Edition“ für den Web-basierten E-Business und darüber liegt die „Enterprise Edition“ für die verteilte kooperative Entwicklung großer Software-Projekte (Sun Microsystems, 2000). Wenn hier von Forte die Rede ist, dann ist immer „Forte for Java, Community Edition“ gemeint.

Forte bietet für den Entwurf netzwerkzentrierter *Anwendungen* (Programme) und *Applets* folgende Unterstützung:

1. Ein Fenster *Visual Interface Design* und die *Component Palette* für die *visuelle Programmierung* grafischer Bedienoberflächen (GUI) auf der Basis von Swing-Klassen.
2. Ein Fenster *Code Generation and Editing* mit Zugang zu einem Programmtext-Editor mit dynamischer Quellcode-Ergänzung.
3. Projektmanagement
4. Debugger
5. Object Browser
6. Fenster für die Anzeige der Attribute und Eigenschaften von Klassen

Exkurs: Verifikation und Validation

Verifikation

Die Verifikation ist der Nachweis dafür, dass ein Programm *S* die durch pre(*S*) und post(*S*) gegebene Spezifikation erfüllt. Das heißt: Erfüllen die Variablen des Programms unmittelbar vor Durchlaufen von *S* die Vorbedingung pre(*S*) dann kommt das Programm zu einem Ende und die Variablen erfüllen anschließend die Bedingung post(*S*). Nehmen wir als Beispiel für *S* die Anweisungsfolge der swap-Funktion mit der folgenden Spezifikation:

`pre(„t = x; x = y; y = t;“): Zustand ist gleich` α

```
post(„t = x; x = y; y = t;“): x=α(y) ∧ y=α(x)
```

Der Beweis geschieht nach der Methode der symbolischen Programmausführung: Anfänglich ist der Wert von x gleich α(x), kurz x=α(x). Dementsprechend ist y=α(y) und t=α(t). Nach der Zuweisung „t=x;“ ist t=α(x), die Werte der anderen Variablen sind unverändert. Nach der Zuweisung „x = y;“ ist x=α(y), sonst ändert sich nichts. Nach der Zuweisung „y = t;“ ist y=α(x), sonst ändert sich nichts. Damit ist der Beweis erbracht: Schließlich gilt x=α(y) $\wedge$ y=α(x).

Beachten Sie, dass die Vor- und Nachbedingungen mathematisch/logische Ausdrücke sind und = die Bedeutung des Gleichheitszeichens hat. In den Programmtexten selbst ist = das Zuweisungszeichen.

Validation

Die Validation soll zeigen, dass das Programm die an Anforderungen erfüllt. Hier geht es also darum, zu zeigen, dass bestimmte Eingabedaten tatsächlich zum zu erwartenden Ergebnis führen. Die Validation geschieht also durch *Tests*. Test ist aber nicht gleich Test.

Man kann an die Sache herangehen, indem man sich den einen oder anderen Eingabedatensatz, der einem eher zufällig in den Sinn kommt, aufschreibt und dem Programm zur Verarbeitung übergibt. Dann schaut man sich das Ergebnis auf seine Plausibilität hin an. Schließlich vergisst man die Sache und den Datensatz. Dieses Vorgehen ist pure Zeitverschwendung und - schlimmer noch - es weckt Illusionen über die Fähigkeiten des Programms.

Stattdessen sind Tests sorgfältig zu entwerfen. Und da einiger Aufwand in ihnen steckt, hebt man die Testfälle auf. Man wird die Tests später wieder dringend benötigen, wenn das Programm weiterentwickelt oder überarbeitet worden ist. Solche Wiederholungen von Tests heißen *Regressionstests*.

Die *Sinnsuche des Wahrnehmungsapparats*, unser Bestreben, Gesetzmäßigkeiten auch in irgendwelche eher unzusammenhängende Sachverhalte hineinzusehen, macht auch vor Testergebnissen nicht Halt. Fehler werden leicht übersehen, wenn man nicht bereits eine Vorstellung vom korrekten Ergebnis hat. Also gilt die Regel, dass vor dem eigentlichen Test sämtliche Testfälle einschließlich der erwarteten Ergebnisse formuliert und dokumentiert sein müssen. Die sorgfältig erstellte *Prognose* ist für die Wirksamkeit des Tests unerlässlich.

Die Fähigkeit, gute Testfälle zu erfinden, wächst mit der Erfahrung. Zur Unterstützung des Gedächtnisses empfiehlt es sich, einen *Regelkatalog für Testfälle* anzulegen und fortzuschreiben. Die folgenden Regeln haben sich allgemein bewährt:

1. Gültige und ungültige Eingabedaten wählen.
2. Sonderfälle und Entartungen berücksichtigen.
3. Aus jeder wichtigen *Äquivalenzklasse* wenigstens einen Eingabedatensatz vorsehen (äquivalent sind Datensätze dann, wenn das Programm vermutlich gleichartig darauf reagiert, wenn also zu erwarten ist, dass entweder bei jedem der Datensätze ein Fehler auftritt oder bei keinem).
4. *Grenzwerte* testen. Also, vorzugsweise Werte vorgeben, die an den Intervallgrenzen - diesseits und jenseits - und an den Wertebereichsgrenzen von Variablen liegen. Elemente mit besonderen Eigenschaften wie 0 oder 1 berücksichtigen.

Aufgaben

9.1 Öffnen Sie *Forte* und arbeiten Sie sich durch das *Getting Started Tutorial*. Notieren Sie Fragen - sowohl zur IDE als auch zum Java-Code. Anschließend wollen wir gemeinsam die Antworten dazu erarbeiten.

9.2 *Programmierung:* Stellen Sie das Programm BlockSim fertig, so dass es folgendes leistet: Nach dem Start von der Konsole aus fragt das Programm zunächst den Namen der Eingabedatei ab. Dann liest es die Eingabedatei und startet die Verarbeitung. Das Ergebnis wird in tabellarischer Form als Textdatei ausgegeben.

9.3 Simulieren Sie mit diesem Programm die Schaltungen aus Bild 1.3 und Bild 1.4. Stellen Sie die Ergebnisse unter Zuhilfenahme eines Tabellenkalkulationsprogramms (beispielsweise Excel) grafisch dar.

9.4* Simulieren Sie weitere Systeme aus dem ersten Abschnitt der Lehrveranstaltung Simulation (Populationsdynamik, Modell der Volkswirtschaft). Aufgabenstellungen:

`http://www.fh-fulda.de/~grams/SimMaterial/SimAufgabenWeb.htm`

9.5 *Verifizierung:* Zeigen Sie die Korrektheit des zentralen Algorithmus Ihres Programms. Beweisen Sie, dass die Spezifikation (Aufgaben 2.2 und 8.6) auch tatsächlich eingehalten wird. Beweisen Sie vor allem auch, dass der Algorithmus nur dann nicht erfolgreich arbeitet, wenn er es prinzipiell nicht kann, d. h.: wenn die Eingabedaten widersprüchlich sind.

9.6 *Validierung:* Zeigen Sie, dass das Programm die formulierten Anforderungen auch tatsächlich erfüllt (Aufgabe 1.1). Geben Sie verschiedene Blockschaltbilder ein. Variieren Sie die Reihenfolge der Bausteine. Überprüfen Sie das Ergebnis einiger Schaltungen mittes Tabellenkalkulation. Geben Sie auch wenigstens ein in sich widersprüchliches Blockdiagramm ein.

10 Entwurf und Programmierung der Bedienoberfläche (GUI)

Daueraufgabe des Ingenieurs ist die Konstruktion „bedienbarer Maschinen“. In dieser Lektion wird das Thema vertieft. Dabei konzentrieren wir uns speziell auf die Gestaltung von grafischen Bedienoberflächen von Programmen (GUI). Weiterführende Informationen dazu sind im Buch „Designing the User Interface“ von Ben Shneiderman (1998) zu finden.

Die sieben Stufen der Mensch-Computer-Interaktion

Donald Norman nähert sich in seinem Klassiker „The Design of Everyday Things“ der Analyse der Mensch-Maschine-Interaktion über ein Sieben-Stufen-Modell. Der Bediener (oder Anwender) durchläuft die folgenden mehr oder weniger voneinander abgegrenzten Schritte:

1. Bildung einer Zielvorstellung (Forming the goal)
2. Fassen der Absicht (Forming the intention)
3. Festlegen einer Aktion (Specifying an action)
4. Durchführen der Aktion (Executing the action)
5. Wahrnehmung des Zustands „der Welt“ (Perceiving the state of the world)
6. Deutung des Zustands „der Welt“ (Interpreting the state of the world)
7. Bewertung des Ergebnisses (Evaluating the outcome)

Fehlerquellen

Es gibt eine Kluft zwischen dem *mentalen Modell*, das der Bediener von der Maschine hat, und dem tatsächlichen Zustand der Maschine. Sie geht - nach Norman - hauptsächlich auf zwei Ursachen zurück:

- Absicht und erlaubte Aktionen passen nicht zusammen: The *gulf of execution*.
- Wahrnehmung und tatsächlicher Systemzustand passen nicht zusammen: The *gulf of evaluation*.

Es folgen einige Beispiele für die Kluft zwischen mentalem Modell und Realität. Ich habe sie selbst erlebt oder sie wurden mir von Bekannten berichtet. Mein Buch Qualitäts- und Risikomanagement enthält mehr davon.

Fehlende Programmeigenschaften (Features). Ich will mir von einem Simulationsprogramm (FaultTree+) den Verlauf einer bestimmten Kenngröße (die Systemausfallrate) errechnen lassen. Aber genau das hat der Programmierer des Simulationsprogramms nicht vorgesehen.

Wasserhähne - zwar ohne GUI, dennoch lehrreich. In der Toilette einer Gaststätte wollte ich die Hände waschen. Ein Becken war an der Wand, darüber eine elegant geschwungenes Stück Metall. Das muss der Wasserhahn sein, denke ich. Aber wie geht er an? Ich versuche dies und das, drehe ein wenig am Ende des Metallteils herum. Plötzlich schießt Wasser aus dem Ding, aber an einer Stelle, wo ich es nicht vermutet habe. Nur der Anbringungsort und die Abwesenheit konkurrierender Gebilde sagt, um was es sich handeln könnte. Stimulierende Signale fehlen. Und wo keine Signale sind, kann nichts wahrgenommen werden. Das mentale Modell des Objekts bleibt vage.

Typing fast. Dem Computer Risks Forum, Volume 20, Issue 64 (Thursday 4 November 1999) entnehme ich eine Meldung folgenden Inhalts: Bei der Arbeit mit dem Tabellenkalkulationsprogramm Excel verabschiedet sich plötzlich und ohne Vorwarnung das Programm. Nach Neustart ist festzustellen, dass die Arbeit einer ganzen Stunde verloren ist. Als Ursache stellt sich heraus, dass die Panne bei Eingabe des Texts „// cannot be tested in test harness“ passiert sein muss. Und das ging so:

- Excel 97 behandelt den Schrägstrich wie das Drücken der Alt-Taste, was den Speicher-Auswahlknopf in der Menü-Zeile aktiviert.
- Der zweite Schrägstrich wird ignoriert.
- Das Drücken der Leertaste öffnet das Drop-Down-Menü.
- Das „c“ wählt die „close“-Option des Menüs.
- Darauf öffnet sich die Dialog-Box mit der Auswahl „Do you want to save? Yes / No / Cancel“.
- Das „a“ wurde ignoriert.
- Das „n“ aktivierte die „No“-Option der Dialog-Box.
- Das war's dann. Excel verabschiedet sich ohne Sicherung der Datei.

Hier kommen wenigstens drei Dinge zusammen.

1. Der Geübte geht zunehmend zu automatisierten Abläufen über und reduziert dementsprechend die Überwachung.
2. Es gibt eine starke Verkopplung der verschiedenen Betriebsmodi (hier: Eingabemodus und Befehlsmodus) durch eine Vielzahl von Möglichkeiten, von einem Modus in den anderen zu wechseln. Die Umschaltung geht mit Tastenkombinationen in mehreren Varianten, über eine tastaturgesteuerte Menüauswahl und mittels mausgesteuerter Menüauswahl. Diese an sich als benutzerfreundlich gedachte Eigenschaft heutiger Bürosoftware erhöht die verdeckte Komplexität und führt im Alltag dazu, dass man zuweilen versehentlich den Modus wechselt.
3. Die Rückmeldung über den Moduswechsel und den erreichten Programmzustand ist schwach. Es kommt zu Wahrnehmungsfehlern. Man erfährt nicht, wo man gelandet ist. Die Kontrolle geht verloren, das mentale Modell kann nicht nachgeführt werden.

Das Aufeinandertreffen von Routine und ungenügender Signalisierung sind im Alltag häufig Quellen von Ärgernissen. Ein paar Beispiele:

Textverarbeitung. Ich bringe einen Teil des Textes, den ich gerade bearbeite in eine andere Datei. Anschließend arbeite ich weiter und vergesse diesen Transport. Am Ende der Sitzung kommen die üblichen Fragen: Änderungen speichern? ja/nein/abbrechen. Bei der Datei, die ich gerade bearbeite, lasse ich speichern. Bei den anderen ist mir das Abspeichern suspekt und ich verneine. Damit habe ich die Arbeit mehrerer Stunden - zunächst unbemerkt - versehentlich vernichtet.

Allgemeine Richtlinien für den Entwurf

Aus dem Sieben-Stufen-Modell lassen sich folgende *allgemeinen Richtlinien* für den Entwurf einer Bedienoberfläche herleiten: Sorge dafür, dass der Bediener

- ein angemessenes Ziel formulieren kann,
- die passenden Bedienelemente leicht findet (Anordnung und Beschriftung),
- die nötigen Bedienaktionen erkennt, und
- eine passende und nicht irreführende Rückmeldung erhält.

Anders ausgedrückt: Sag', was das Programm leistet und wie man an die Leistungen herankommt - und das möglichst direkt, *ohne separate Bedienungsanleitung!* Und mache unmissverständlich klar, welche Leistung auf Anforderung erbracht worden ist.

Kurz: Mache die Funktion sichtbar!

Auf meiner Web-Seite

```
http://www.fh-fulda.de/~grams/WebDesignMain.htm
```

habe ich ein paar Informationen zu den Themen

- Anordnung und Gruppierung von Information,
- Farbgestaltung und
- optische Reize

zusammengestellt. Dort sind auch einige Links zu finden, insbesondere zu den wegweisenden „Web Pages That Suck“.

Die Schritte der GUI-Erstellung

Die folgende Reihenfolge der Schritte beim Erstellen eines GUI hilft, Irr- und Umwegen zu vermeiden:

- Anforderungserfassung und Spezifikation der Bedienoberfläche
- Erfassung der verfügbaren Werkzeuge (IDE, JDK, Editoren, Laufzeitumgebung)
- Studium der Komponentenpalette (Frames, Buttons, Textfelder, usw.)
- Entwurf der GUI als Bleistiftskizze
- Zuordnung der Komponenten zu den Elementen der Skizze
- GUI-Programmierung mit Texteditor oder visuell
- Test der Grundfunktionen des GUI
- Integration der Bedienoberfläche in das Gesamtprogramm
- Systemtest

Aufgaben

10.1 Entwerfen und Programmieren Sie eine einfache und generell verwendbare grafische Bedienoberfläche für Systeme mit Datenein- und -ausgabe über Textdateien auf der Basis eines JFrames. Die Bedienoberfläche soll ein Fenster zum Editieren der Eingabedatei besitzen. Die Eingabedatei soll entsprechend aktualisiert werden können. Integrieren Sie die Bedienoberfläche in das Programm BlockSim.

10.2 Die Eingabe in das Programm BlockSim basiert weitgehend auf textueller Darstellung der Information und auf der direkten Manipulation dieses Textes in Texteditoren. Wie lässt sich das ändern? Machen Sie Vorschläge und diskutieren Sie diese im Hinblick auf die Flexibilität und den weiteren Ausbau des Systems.

10.3 Welche Generalisierungen wurden gemacht und können noch gemacht werden. Definieren Sie eine generell verwendbare abstrakte `LinkA`-Klasse, die auf der `Link`-Klasse basiert und die einen Methode der topologischen Sortierung anbietet, die der bei BlockSim realisierten entspricht. Benennen Sie die universell verwendbaren Klassen und stellen Sie eine kleine *Bausteinbibliothek* zusammen.

11 Simulation einer Füllstandsregelung

Anwendungsbeispiel für BlockSim: Füllstandsregelung

Größen der Füllstandsregelung	
x	Füllstand (Regelgröße)
z	Zufluss
w	Sollwert des Füllstands
x_w	Regelabweichung
y	Abfluss (Stellgröße)
Parameter	
a	P-Anteil des Reglers
b	I-Anteil des Reglers
Systemgleichungen	
$x_w = x - w$	
$\mathrm{d}x/\mathrm{d}t = z - y$	
$y = a \cdot x_w + b \cdot \int x_w(t)\mathrm{d}t$	

Für das Ausregeln kleiner Schwankungen des Flüssigkeitsstands in einem Gefäß mit Zufluss und regelbarem Abfluss hat man das folgende Modell: Der Flüssigkeitsstand x (Regelgröße) in einem zylindrischen Gefäß soll auf einem konstanten Wert w = 120 cm (Sollwert, Führungsgröße) gehalten werden. Die Regelabweichung ist $x_w = x - w$. Zu- und Abfluss werden gemessen in cm/s und geben die jeweilige Änderung des Flüssigkeitsstandes je Zeiteinheit an. Der Abfluss y ist die Stellgröße. Ihr Wert ist nach unten durch null begrenzt. Der Zufluss ist konstant gleich z = 10 cm/s.

Die letzte der Systemgleichungen beschreibt das Verhalten eines proportional-integralen Reglers (PI-Regler). Die Parameter a und b bestimmen den P- bzw. den I-Anteil.

Nichtlineares Ausströmgesetz: Die Aufgabenstellung ist insofern etwas vereinfacht worden, als direkt der Abfluss zur Regelgröße gemacht wurde. Aber eigentlich ist nicht der Abfluss, sondern nur die Ventilöffnung direkt einstellbar. Wegen des Ausströmgesetzes von Torricelli ist der tatsächliche Abfluss proportional zur Ventilöffnung multipliziert mit der Wurzel aus dem Füllstand - letzterer gemessen über dem Abfluss. Wir normieren die Stellgröße so, dass bei einem Füllstand von x_0=120 cm der Ausfluss genau gleich der Stellgröße y ist. Wir können also das Torricellische Ausströmgesetz dadurch berücksichtigen, dass wir die Formel $\mathrm{d}x/\mathrm{d}t=z-y$ ersetzen durch $\mathrm{d}x/\mathrm{d}t = z - y \cdot \sqrt{x/x_0}$.

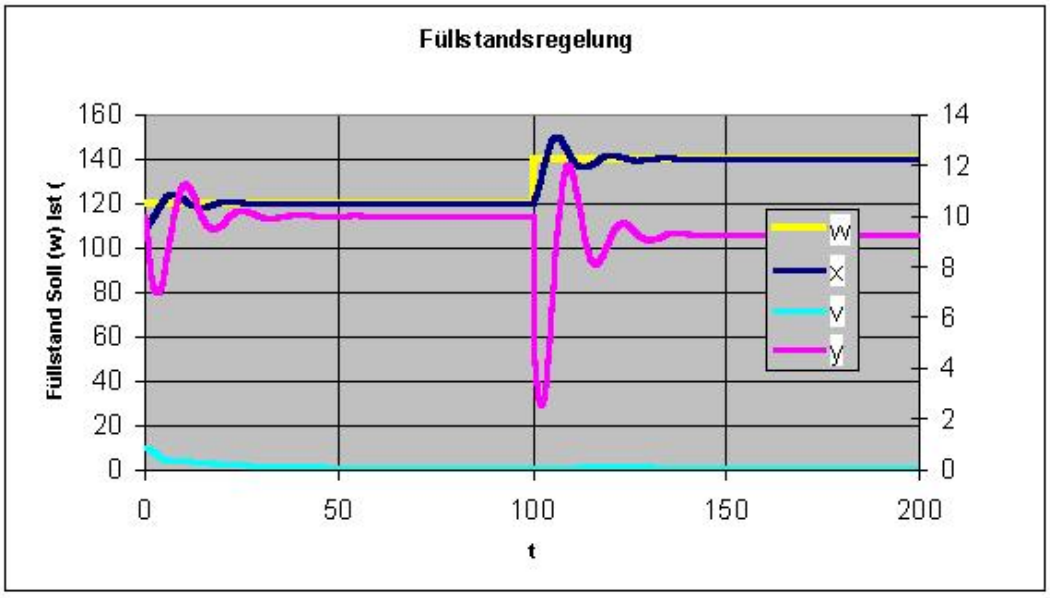
Aufgaben

11.1 Diskretisieren Sie die Systemgleichungen unter Verwendung der Integrationsformel von Euler-Cauchy (Polygonzugverfahren).

11.2 Erstellen Sie das Blockdiagramm für den linearen Fall unter ausschießlicher Verwendung der BlockSim-Bausteine.

11.3 Simulieren Sie das System für einen Sollwertänderung von w = 120 cm auf 150 cm. Der Zufluss ist dabei konstant: z = 10 cm/s. Die Parameter des Reglers sind eingestellt auf die Werte a = 0,2/s und b = 0,2/s^2.

11.4 Stellen Sie die Ergebnisse mit einem Tabellenkalkulationsblatt grafisch dar wie beispielsweise im folgenden Bild.



11.5* Ergänzen Sie BlockSim um einen Baustein für die Wurzelfunktion. Berücksichtigen Sie im Simulationsmodell das Torricellische Ausströmgesetz.

12 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:

- *Titel:* Projektbezeichnung mit Untertiteln und Nennung des Autors oder der Autoren.
- *Vorwort:* Nennt die Absicht, die mit dem Projekt und der Dokumentation verfolgt wird und Besonderheiten in inhaltlicher oder gestalterischer Hinsicht.
- Zus*ammenfassung:* Fasst den Textteil speziell unter den Aspekten Problem, Ziel, Methode und Ergebnis der Arbeit zusammen. Hier steht nichts Neues. 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 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.* Dem Textteil kann eine *Einleitung* vorangestellt sein. 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. Anschließend kommt der Hauptteil, dessen Gliederung dem Lebenszyklusmodell folgt.
- *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“.

Regeln zur Typographie und weitere Tips

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.

Aufgaben

12.1 Erstellen Sie eine Gesamtdokumentation des Projekts BlockSim unter Verwendung der bereits bei Erledigung der vorangegangenen Aufgaben erstellten Texte und Bilder.

Sachverzeichnis

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z