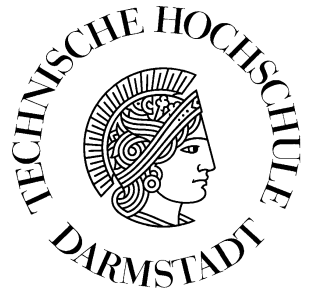
Grundlagen der Informatik I

“Programmierung”

Wintersemester 1993/94

Christoph Kreitz



Skript zur Vorlesung
Grundzüge der Informatik I

Fachgebiet Intellektik
Fachbereich Informatik
der Technischen Hochschule Darmstadt
Alexanderstr. 10, 64283 Darmstadt

Die Beiträge wurden von verschiedenen Autoren geschrieben, so daß eine “multi-personelle” Sicht des Stoffes vorliegt. Irrtümer, insbesondere aufgrund von Schreibfehlern, zeugen von der menschlichen Natur der Autoren, sind aber nicht beabsichtigt. Korrekturen werden gerne entgegengenommen. In das Skript sind ohne ausdrückliche Erwähnung Teile früherer Vorlesungs-Begleittexte aufgenommen, die von den Darmstädter Informatikprofessoren Encarnacao, H.-J. Hoffmann, Lustig, Piloty, Tzschach, Waldschmidt und Walter stammen, die diese Vorlesung in vorangegangenen Semestern gehalten haben.

Besonders danke ich Prof. W. Henhapl und M. Kremer für die Bereitstellung des entsprechenden Skriptums der Vorjahre und für anregende Diskussionen bei der Überarbeitung und Neukonzeption der Vorlesung auf dieser Grundlage.

Darmstadt, 10. Februar 1994

Inhaltsverzeichnis

Abbildungsverzeichnis

Vorwort für das Wintersemester 1993/94

Nachträglich herzliche Gückwünsche zum Abitur! Das Abitur ermöglicht es Ihnen sich zum Diplom Informatiker, Diplom Mathematiker oder Diplom Wirtschaftsinformatiker weiterzuqualifizieren. Die Betonung liegt hierbei auf *sich weiterqualifizieren*. Die Hochschule bietet Ihnen dafür den fachlichen Rahmen an, den sie nach Ihren individuellen Bedürfnissen nutzen können.

Im Gegensatz zur Schule sind Sie jedoch an der Universität für die Befriedigung Ihrer Bedürfnisse allein verantwortlich. Man könnte den Unterschied Schule – Universität mit den verschiedenen Arten von Wegen auf einem Berg vergleichen. Die Schule ist ein Wanderweg, der auf eine Alm führt, breit und gut beschildert, Auf dem Weg kamen Sie manchmal in Atemnot und der Schweiß rann in die Stirn, aber nachträglich können Sie sich wahrscheinlich an keine besonderen Schwierigkeiten mehr erinnern. Hinter der Alm kommt immer der felsige Gipfel zum Vorschein.

Die Hochschule ist ein Gewirr von Kletterpfaden zu diesem Gipfel, aus denen Sie sich einen auswählen und ihn, dann begleitet von Bergführern, erklimmen. Die Bergführer stellen Ihnen die notwendige Ausrüstung zur Verfügung. Sie werden Sie jedoch niemals hochziehen, sondern Ihnen nur die nächsten Griffe zeigen. Klettern müssen Sie selbst! Zwischendurch werden Sie sicherlich die Angst haben, daß dieser Steig nicht zum Gipfel führt oder die Ihre eigenen Kräfte übersteigt. Dann ist es Zeit, die Route mit den Begleitern (Kollegen und Bergführen) im Detail zu studieren. Vielleicht wäre eine andere Route besser, vielleicht war ein Fehler in der Wegbeschreibung, vielleicht war ein Mißverständnis bei der letzten Besprechung, vielleicht sollten Sie ein Trainingslager aufsuchen. Es gibt viele Gründe frustiert zu sein. Da hilft nur die Analyse: Wo bin ich, wohin will ich und reichen meine Fähigkeiten dafür aus?

Erreichen Sie den Gipfel, so benötigen Sie kein Gasthaus zur Befriedigung noch unerfüllter Wünsche. Ihre eigenen "Endomorphine" geben Ihnen die Befriedigung. Natürlich klettert einer besser als ein anderer. Dafür sind auch die vielen verschiedenen Routen vorhanden. Mit dem Bergführer können Sie den Ihnen angemessenen Schwierigkeitsgrad auswählen.

Dieses Bild auf das erste Semester übertragen bedeutet, daß von Seiten der Betreuung in *diesem* Semester kein Zwang ausgeübt wird, eine Minimalanforderung (Kletterschule) zu erfüllen, sondern nur Angebote gemacht, mehr als die Minimalanforderung zu tun. Das Team, das Sie im ersten Semester betreut, wird Sie in keiner Weise kontrollieren aber stets bereit sein, Ihnen Hinweise zu geben, wie Sie sich weiterentwickeln können. Ihr Studium wird Sie zufrieden stellen, wenn Sie diese Angebote wahrnehmen und immer wieder bis an Ihre Leistungsgrenze gehen. Sie werden verblüfft sein, wie dehnbar diese Grenze ist. Keine Kritik kann das Betreungsteam schwerer treffen, als der Vorwurf, Sie waren unterfordert.

Die Schwierigkeiten in den ersten Semestern des Informatikstudiums liegen nicht in der Menge des Stoffes (der ist verglichen mit Wirtschaftsfächern minimal) sondern im Einüben der *Denkformen*. Es ist nicht zu erwarten, daß Sie sich nach Durchlesen einer Grammatik und eines Wörterbuchs einer fremden Sprache sofort in dieser Sprache gut ausdrücken können. Sie müssen erst lernen, in dieser Sprache denken. Das gleiche gilt für die Informatik. Sie lernen *Logik*, *Funktionen* und *Programmiersprachen* kennen. Jede einzelne hat ein minimales Vokabular, aber die Denkweisen sind neu! Es wird Ihnen manchmal passieren, daß Sie manche Teile nicht auf Anhieb verstehen, das kann ein (didaktischer oder echter) Fehler des Vortragenden sein, es wird aber häufig auch ein Mangel an Training dieser Denkweise sein. Seien Sie dann nicht verzweifelt. Die Vorlesung und noch mehr das Skript sind redundant ausgelegt. Es wird vieles wiederholt. Verbohren Sie sich daher nicht in ein Problem, lesen Sie einfach weiter, vielleicht kommt das Verständnis beim nächsten Beispiel. Es ist aber auch zweckmäßig den alten Stoff zu wiederholen, schon allein um zu merken, welche Fortschritte Sie gemacht haben.

Sie haben sich für ein Studium an einer Universtät entschieden. Die Informatik an der Universtät hat neben dem berufsqualifizierenden Anspruch auch den Anspruch, Ihnen die wissenschaftlichen Aspekte des Faches darzustellen. Im ersten Semester steht die *Programmkonstruktion* im Mittelpunkt. Die wissenschaftliche Untersuchung der Programmkonstruktion bedeutet *Wissen über Programme zu schaffen*. Wissen über Programme

bedeutet wiederum, exakte Prognosen über das betrachtete Programm stellen zu können. Die wichtigsten Prognosen sind *Wirkung des Programms* und sein *Betriebsmittelbedarf* (Zeit und Speicher). Wir werden uns daher intensiv mit *Programmbeweisen*, d.h. dem exakten Nachweis, daß eine Vermutung über die Wirkung eines Programms auch tatsächlich eintritt, befassen und die Korrektheit von Abschätzungen über das Laufzeitverhalten nachweisen. Die beiden Themen erfordern von Ihnen mehr als die Freude am Basteln. Durch die neuen Betrachtungsweisen können Sie jedoch über das Konstruieren hinauswachsen zum Entdecker der notwendigen Zusammenhänge in einem Programm, und damit eröffnet sich Ihnen die neue Dimension der Ästethik von Programmen. Ich wünsche Ihnen viel Erfolg auf dem steilen Weg zu den Gipfeln, die Ihnen statt weite Aussichten tiefe Einsichten bieten werden.

Lehrziel

Das Lehrziel des Zyklus "Grundzüge der Informatik I - IV" ist die Einführung in die Fachsystematik und die Vermittlung der fundamentalen Methoden und Fakten der Informatik. In der Lehrveranstaltung "Grundzüge der Informatik I" steht die Vermittlung der elementaren Entwurfs- und Programmierfertigkeiten im Vordergrund (siehe Studienordnung Informatik),

Lehrinhalte

Der Lehrinhalt des Grundstudiums ist in der Studienordnung, die mehr als zehn Jahre alt ist, beschrieben:

> *Im Zentrum des Grundstudiums der Informatik stehen die Begriffe Algorithmus, Programm und Rechner. Deshalb werden Elemente der Rechnerarchitektur, des Schaltwerkentwurfs, des Übersetzerbaus, der Programmiersprachen, der Betriebssysteme und der Datenstrukturen behandelt. Anhand von Problemstellungen aus diesen Bereichen werden wichtige Eigenschaften von Algorithmen (Laufzeitverhalten, Korrektheit, u.ä.) abgeleitet. Algorithmen werden auf Rechenanlagen in problem- und maschinenorientierten Sprachen implementiert. Weiterhin soll ein Verständnis der technologischen Gegebenheiten für die Rechenanlagen vorbereitet werden. Am Abschluß des Informatikanteils stehen mathematische Modellbildungen für Rechner und eine Präzisierung des Algorithmusbegriffs.*

Die Informatik ist als Studiengang erst 30 Jahre alt und entwickelt sich noch immer stürmisch weiter. Daher ist es verständlich, daß auch die Studieninhalte weiterentwickelt werden. Die stärkste für das Grundstudium relevante Änderung ist der Übergang vom *EVA-Prinzip* (Eingabe, Verarbeitung, Ausgabe) zum Prinzip der *Objektorientierung*. Die Anpassung an das neue Paradigma wurde in den vergangenen Vorlesungen bereits vorbereitet [Hoffmann, 1990], wurde aber erst im Wintersemester 1991/92 radikal durchgeführt. In diesem Semester steht der Begriff des Objekts im Zentrum. Algorithmen und Programme sind dabei wichtige Teilaspekte. Seinen Niederschlag fand dieser Wechsel in dem Übergang von Pascal (ohne Objekte) zu Eiffel [Meyer, 1988]. In Eiffel sind die objektorientierten Sprachkonzepte einfacher und genereller als in neueren Versionen von Pascal und Modula enthalten. Das eigentliche Lehrziel der Studienordnung, die Grundlagen für den Entwurf komplexer Systeme und deren Realisierung auf Rechner(-netze)n aufzubauen, bleibt erhalten, wird aber durch diesen Wechsel wesentlich leichter erreicht.

In der Lehrveranstaltung "Grundzüge der Informatik I" liegt ein Schwerpunkt

- auf dem *Entwurf* von Systemen bescheidenen Umfangs,
- auf der *Realisierung* des Entwurfs in einer problemorientierten Sprache (Eiffel als exemplarischer Repräsentant problemorientierter Sprachen) und
- auf der *Analyse der Qualität* des Entwurfs (Strukturierung) und der Realisierung (Zuverlässigkeit, Effizienz).

Begleitend dazu wird auch auf die sozialorientierten Gütekriterien (Benutzerfreundlichkeit, Datenschutz, Computerethik, usw.) der Aufgabenstellung (*Anforderungsdefinition*) eingegangen.

Der zweite Schwerpunkt liegt im Umgang mit *formalen Systemen* (Logik, Funktionen, abstrakten Maschinen). Auf dieser Grundlage werden die von der Studienordnung geforderten Basiskonzepte der Programmiersprachen und später auch der Übersetzer vermittelt.

Kann der erste Schwerpunkt eventuell noch als Weiterführung der Informatik der Schule angesehen werden, so bringt der zweite Schwerpunkt neue *Denkweisen* ein, die in der Schule nicht geübt wurden. Hier ist auch bereits im ersten Semester der Unterschied zwischen TH- und FH-Studium zu sehen: An der TH ist das Studium stärker *grundlagenorientiert*, an der FH mehr praxisorientiert.

Die Veranstaltungen des nächsten Semesters, "Grundzüge der Informatik II" und "Rechnertechnologie" sind der Rechnerarchitektur, dem Schaltwerkentwurf und den maschinenorientierten Sprachen gewidmet. "Grundzüge der Informatik III" vertieft das Verständnis der Datenstrukturen mit ihren Operationen. In "Grundzüge der Informatik IV wird die mathematische Modellbildung für Rechner und Algorithmus vertieft.

Hinweise zur Lehrveranstaltung "Grundzüge der Informatik I"

Ideal wäre eine Lehrveranstaltung, in der eine Gruppe von 5-7 Studierenden unter Beratung eines Professors den Stoff selbst erarbeitet und einübt. Leider erfordert diese Form des Studiums bei etwa 450 Studierenden etwa 30 Professoren nur für diese eine Lehrveranstaltung. Weder die Personalsituation noch die finanzielle Ausstattung erlauben auch nur annäherungweise diesen Traum zu realisieren. Um einen Rest an individueller Gestaltung und Entfaltung ermöglichen zu können, werden Anteile, die für alle grundlegend sind, auch für alle gleichzeitig angeboten: *Vorlesung*. Die Kleingruppen von maximal sieben Studierenden, die sich nach der Idealvorstellung den Stoff gemeinsam erarbeiten, kommen dann mit zwei anderen Kleingruppen zu einem Wettbewerb der Ideen zusammen: *Übung* und *Praktikum*. Individuelle Betreuung zur Beseitigung von persönlichen Defiziten und Beratung zur über das Standardangebot hinausgehenden Studium: *Beratungsstunden*.

Vorlesung:

Vorlesung und Skript stellen das Standardangebot des Stoffes dar (das gilt nur für diese Vorlesung). In der Vorlesung wird der Stoff nur in anderen Worten, jedoch nicht mit anderen Inhalten wie im Skript dargestellt. Beide sind als *Hilfestellung* für diejenigen gedacht, die Schwierigkeiten mit der selbständigen Wissensaneignung aus Büchern haben(, was zu Beginn ganz normal ist). Der hier dargestellte Stoff ist wesentlich besser und ausführlicher in den empfohlenen Büchern zu finden. Eine Vorlesung soll Ihnen das Thema vorstellen, Ihnen die wesentlichen Fragestellungen und Denkformen nahebringen, und Sie anregen, sich *selbst* damit zu beschäftigen. Sie ist nicht gedacht als eine komplette Stoffvermittlung, auch wenn im Grundstudium die Tendenz hierzu noch etwas stärker ist. Im Hauptstudium werden sie mehr und mehr Literatur nachschlagen müssen, um sich ein geschlossenes Bild von dem vorgetragenenen Themengebiet zu machen. Es ist durchaus im Sinne dieser Veranstaltung, wenn Sie dies jetzt bereits einüben.

Damit der Lehrstoff allen angeboten werden kann, muß er für alle *ungestört* hörbar sein. Das erfordert bei dieser Massenveranstaltung eine große Solidarität mit der Kollegin oder dem Kollegen, die lieber zuhören wollen, anstatt sich die wesentlich wichtigeren Neuigkeiten aus dem Umkreis anzuhören. Bringen Sie also bitte das Opfer, auch die brennenste Information erst nach der Vorlesung an ihre Freunde weiterzugeben oder zumindest sehr leise zu reden.

Diese Vorlesung und die Praktikumsvorlesung, die ebenfalls im Rahmen der "Grundzüge der Informatik I" stattfindet, versuchen Ihnen das Thema Programmierung von verschiedenen Seiten nahezubringen. Was den

Stoff betrifft, so werden Sie eine große Redundanz bei beiden Veranstaltungen feststellen. Das ist durchaus beabsichtigt. Jedoch liegt ein Unterschied in den Schwerpunkten, die beide Veranstaltungen legen.

Die Praktikumsvorlesung wird eher den "Ingeneur"-Anteil in den Vordergrund stellen wird, bei dem es um die systematische – aber sehr konkrete – Entwicklung *lauffähiger* Programme geht. Im Gegensatz dazu stellt diese Vorlesung den *wissenschaftlichen* Aspekt stärker heraus und befaßt sich mit Konzepten und Denkformen, die nötig sind, um eine hohe Qualität bei den entwickelten Programme sicherzustellen. Notwendigerweise stehen daher das Ausprogrammieren von Details und die Feinheiten der konkreten Programmiersprache (Eiffel 3, siehe [Meyer, 1992]) im Hintergrund.[1]

Durch die Trennung von Vorlesung und Praktikumsvorlesung soll auch das Problemfeld deutlicher werden, in dem Sie sich als Informatiker bewegen werden: auf der einen Seite müssen Sie gründliche, systematisch-theoretische Entwurfsüberlegungen durchführen, um Qualität zu sichern, auf der anderen Seite steht die Anforderung, daß Ihr Programm irgendwann auch lauffähig sein muß.

Übung und Praktikum:

In der Orientierungsphase haben Sie lauter nette Kolleginnen und Kollegen kennengelernt, mit denen allen Sie gerne zusammenarbeiten würden. Aus lernpsychologischen Gründen ist es jedoch günstiger in einer kleineren Gruppe (maximal sieben) Ihre Sicht des Stoffes, die Sie sich allein erarbeitet haben, zu diskutieren. Zusammenlernen bedeutet nicht, Hilfe in Anspruch nehmen, sondern verschiedene Interpretationen prüfen und falsche erkennen. In der Informatik ist diese Form der Zusammenarbeit (Reviews) unabdingbar für das Gelingen eines Projekts. Daher nutzen Sie unentwegt die Möglichkeit Wissen und Verfahren kurz und doch vollständig darzustellen und umgekehrt Sichtweisen anderer zu verstehen und konstruktiv zu kritisieren. Die Hörsaalübungen mit etwa 20 Teilnehmern (etwa drei bis vier Kleingruppen) und die praktischen Übungen am Rechner sind aus dieser Sicht Angebote, die von der Gruppe geprüften Ergebnisse der objektiven Kontrolle (Tutor oder Rechner) zu unterziehen. Diese Kontrolle bedeutet aber nur, die Lösung ist akzeptabel. Ihre Großgruppe sollte aber darüber hinausgehen und aus den Ergebnissen der Kleingruppen die beste Lösung erarbeiten, die dann allen Teilnehmern zur Verfügung steht.

Der Tutor sorgt in der Übung nicht nur dafür, daß Ihr Verständnis des Stoffes korrekt ist, sondern er kontrolliert auch daß die Qualität der Vorlesung bzw. des Skripts Ihren Anforderungen entspricht. Stellt er in der Übung fest, daß der Stoff der Vorlesung mißverständlich oder didaktisch falsch dargestellt wurde, so gibt er diese Information in der wöchentlichen Besprechung weiter und erwirkt dadurch eine Wiederholung bzw. Klärung des kritisierten Kapitels.

Die Übung wird durch diese wechselseitige Kontrolle zum wichtigsten Teil der gesamten Lehrveranstaltung. Die Übungen sind jedoch nur Angebote, aber keine Pflicht. Ihre *aktive* Teilnahme ist für Sie von großem Vorteil, aber für die Erreichung eines Scheins nicht unbedingt notwendig. Bedenken Sie jedoch auch, wenn Sie sich alles allein erarbeiten, es anderen jedoch nicht mitteilen können, so ist Ihr Wissen und Können wertlos. Die Informatik hängt mehr als andere Technikwissenschaften von der Teamarbeit und daher von der Teamfähigkeit jedes einzelnen ab. Die Übungen haben neben der Kontrolle des Verständnisse den Sinn, durch konstruktive Kritik gemeinsam von guten Lösungen zur besten Lösung zu kommen.

Die Fähigkeit, alle vorgegebenen Aufgaben *selbständig* zu lösen (Verstehen ist zu wenig!) gibt Ihnen die Sicherheit, den Übungsschein zu erreichen. Wenn Sie die dahinterstehenden Konzepte wirklich verstehen und selbständig anwenden können, so ist für Sie auch die Vordiplomsprüfung kein Problem. Wollen Sie später

[1] Aus diesem Grunde wird es in ganz seltenen Fällen auch einmal dazu kommen, daß die in der Vorlesung benutzte Programmiersprache Eiffel 3 Abweichungen aufweist gegenüber der Sprache Eiffel 3, so wie sie der im Praktikum benutzte Compiler akzeptiert. Derartige Abweichungen müssen Sie lernen in Kauf zu nehmen, da es immer wieder geschehen wird, daß der auf Ihrem Computer installierte Compiler nicht exakt dem Standard der Sprache entspricht, wie Sie ihn kennen. Sie sollten daher frühzeitig lernen, Unterschiede – meist nur in Bezeichnungen und vordefinierten Befehlen und Klassen – aus dem Handbuch zu entnehmen.

jedoch zufrieden und nicht schamhaft auf Ihr Grundstudium zurückblicken, dann sollten Sie über das Standardangebot hinausgehen und Ihr Studium aktiv gestalten. Sind Sie noch dazu in der Lage, die Übungsgruppe in dieser Weise zu aktivieren, dann haben Sie den Sinn des Studiums begriffen.

Sprechstunde der Tutoren

Der Tutor Ihrer Übungsgruppe ist Ihr persönlicher Berater während des ersten Semesters. Insbesondere ist er der erste Ansprechspartner für Ihre individuellen Wünsche. Sei es, daß Sie Teile nicht verstanden haben, Fehler im Skript vorliegen oder auch daß Sie zusätzliche Trainingsaufgaben oder Literaturhinweise haben wollen. Bedenken Sie aber immer, daß wir Sie *fördern* wollen und damit ein Vorbeten durch den Tutor ausgeschlossen ist. Der Tutor ist verpflichtet, eine Beratung sofort abzubrechen, wenn er merkt, daß Sie sich nicht selbst um eine Lösung bemüht haben. Insbesondere wenn er sieht, daß Sie die Übung oder das Skript nicht genau gelesen haben. Er wird aber stets bereit sein, Defizite der Vorlesung und der schriftlichen Unterlagen auszugleichen.

Sprechstunde des Vortragenden

Die Sprechstunde des Vortragenden hat zwei Ziele:

1. Für die Vortragenden die direkte Rückkopplung über den Wissenstransfer: Welche Stoffinhalte wurdenzu schnell oder zu langsam, zu detailreich oder zu oberflächlich usw. dargestellt?

2. Anregung für das individuelle Studium zu finden, z.B. weiterführende Literatur besprechen oder weitere Programmieraufgaben zu definieren. Also ein individuelles Trainingsprogramm entwickeln — Doping ohne Schäden!

Es darf in keinem einzelnen Fall vorkommen, daß ein Studierender sich beklagt, daß ihm trotz Nutzung aller Angebote langweilig war weil er nicht ausreichend gefordert wurde.

Sprechstunde von Herrn Tzeras

Herr Tzeras erledigt alle organisatorische Aufgaben, wie Anmeldungen zur Klausur (für die Arbeitsplatzrechner ist jedoch die Rechner-Betriebsgruppe zuständig), Wechsel der Übungsgruppe usw. Er ist aber auch Ihr Ansprechpartner, wenn Sie persönliche Probleme mit Ihrem Tutor oder den Vortragenden haben und Sie diese Probleme nicht im direkten Gespräch bereinigen können oder wollen, wie z.B. wenn die Folien unlesbar sind, der Stoff zu langsam vorgetragen wird, das Engagement der Vortragenden und Tutoren zu wünschen übrig läßt.

Studienleistungen

Zum Abschluß Ihres Studiums werden Sie von der Hochschule eine Bescheinigung (Diplom) darüber haben wollen, daß Sie hinreichend viele Kenntnisse und Fähigkeiten erworben haben. Aus diesem Grunde müssen Prüfungen durchgeführt werden. Die Studienordnung sieht hierfür vor, daß eine Diplom- und Vordiplomsprüfung abgelegt werden muß und daß für die Zulassung zur Vordiplomsprüfung die erfolgreiche Teilnahme an diversen Grundvorlesung durch einen “Schein” nachgewiesen wird. Voraussetzung zur Zulassung zur Vordiplomsprüfung “Informatik A” sind die Übungsscheine der Lehrveranstaltungen “Grundzüge der Informatik I”. und “Grundzüge der Informatik II”. Im Gegensatz zu Vordiplomsprüfungen dürfen Sie “Scheinprüfungen” beliebig oft wiederholen (auch wenn es natürlich wünschenswert ist, nur einen Anlauf machen zu müssen).

Die Lehrveranstaltung “Grundzüge der Informatik I” umfaßt neben der Vorlesung Übungen, deren verschieden Lösungen in der Großgruppe diskutiert werden, auch Übungen, die am Rechner durchgeführt werden. Sie

bilden die Grundlage zum Erwerb des Übungsscheins für die Informatik I. Die Überprüfung des Erreichens des Lehrziels erfolgt in der Semestralklausur. Der Übungsschein wird *nur* durch die Semestralklausur erworben.

Die Prüfungsthemen von Informatik A sind (die Prozentzahlen geben das Gewicht der Themen an):

1. Entwurf 15%

   - Spezifikation von Datentypen
     – Schnittstellen
     – Semantikbeschreibung von Schnittstellen
   - Strukturierungskonzepte:
     – Parametrisierte Datentypen
     – Vererbung

2. Realisierung 50%

   Umsetzung von Schnittstellenbeschreibungen in Funktionen und Prozeduren

   - Algorithmusentwurf auf der Basis von Vor- und Nachbedingungen
   - Realisierung in höheren und maschinennahen Programmiersprachen

3. Analyse der Zuverlässigkeit 15%

   - Korrektheitsbeweise
   - Terminierungsbeweise
   - Testen

4. Analyse der Effizienz 10%

   Komplexitätsabschätzungen

5. Modelle von Programmiersprachen, 10%

   - Kontrollstrukturen
   - Speicherkonzept
     – Statische und dynamische Variablen
   - Prozedur- und Funktionskonzepte
     – Rekursion
     – Parameterübergaben

Die Prüfungsfragen betreffen die allgemeinen Konzepte für den Nachweis, daß die Grundlagen des Systementwurfs, der Systementwicklung in problemorientierten Programmier- und maschinennahen Sprachen, und der Analyse der Qualität einer Systementwicklung verstanden wurden.

Es ist in einer Klausur nicht möglich, alle Themen in gleicher Tiefe zu prüfen. Daher bilden die Prüfungsfragen nur Stichproben und wechseln von Klausur zu Klausur. Die Bearbeitung alter Klausuren ist eine sehr gute Kontrolle des eigenen Verständnisses, sind aber keine Garantie für das bestehen der nächsten Klausuren!

Für die Übungsscheine gelten die dieselben Prozentsätze, aber auf denn Stoff des jeweiligen Semesters angewandt. Insbesondere wird in den Übungsscheinen die Fähigkeit überprüft, die *gegebenen Praktikumsaufgaben weiterzuentwicklen.*

Einige empfehlenswerte Bücher

Die Vorlesung "Grundzüge der Informatik I" wird sich in vielen Teilen an dem Buch *Object-oriented Software Construction* von B. Meyer orientieren, allerdings darüber hinaus noch den Aspekt der Verifikation betonen, der in diesem Buch nicht behandelt wird. Da das Buch zudem die erste Version der Sprache Eiffel benutzt, wird als Sprachreferenz zum Nachschlagen das Buch Eiffel the Language empfohlen. Beide Bücher sind nicht billig.

Allgemein

1. H.J. Appelrath, Ludwig: Skriptum Informatik - eine konventionelle Einführung; Teubner, 1991
2. H.J. Appelrath, Ludwig, a. Spiegel: Aufgaben zum Skriptum Informatik; Teubner, 1991
3. F. L. Bauer, G. Goos: Informatik, eine einführende Übersicht; in zwei Teilen, Springer-Verlag, 3. Auflage 1990
4. F. L. Bauer, R. Gnatz, V. Hill: Informatik, Aufgaben und Lösungen; in zwei Teilen, Springer-Verlag, 1975
5. F. L. Bauer, H. Wössner: Algorithmische Sprache und Programmentwicklung; Springer-Verlag, 1981
6. O. Dahl, E. W. Dijkstra, C. A. R. Hoare: Structured Programming; Academic Press, 1972
7. H.-J. Hoffmann: Grundzüge der Informatik I + II, Interner Bericht, Kopiervorlage im LZI
8. W, Henhapl: Grundzüge der Informatik I + II, Interner Bericht, Kopiervorlage im LZI
9. G. Hommel, S. Jähnichen, C. H. A. Koster: Methodisches Programmieren; de Gruyter, 1983
10. D. Knuth: The art of computer programming; Vol 1: Fundamental algorithms; Addison-Wesley, ab 1968
    ein Klassiker, aber immer noch sehr empfehlenswert
11. B. Meyer: Object-oriented Software Construction; Prentice Hall International, 1988.
    Auch in Deutsch erhältlich, aber nicht gut übersetzt
12. H. Noltemeier, R. Laue: Informatik II; Hanser, 1984
13. H.-J. Schneider (Hrsg.): Lexikon der Informatik und Datenverarbeitung; Oldenbourg, 2. Auflage, 1986
14. E. H. Waldschmidt, H. K.-G. Walter: Grundzüge der Informatik I; BI-Wissenschaftsverlag, 1984
15. N. Wirth: Systematisches Programmieren; Teubner Studienbücher, 1972
    eine gute Einführung, inzwischen etwas veraltet
16. N. Wirth: Algorithmen und Datenstrukturen; Teubner Studienbücher, 1983

Verifikation

1. D. Gries: The Science of Programming, Springer-Verlag 1981
   Eine extrem geduldige Einführung in Systematik und Verifikation
2. E. Dijkstra: A Discipline of Programming, Prentice Hall 1976
3. B Hohlfeld, W. Struckmann: Einführung die Programmverifikation, Reihe Informatik 88, BI-Wissenschaftsv
   92

Komplexität

1. A.. V. Aho, E. Hopcroft, D. Ullman: The Design and Analysis of Computer Algorithms; Addison Wesley 1975

Programmiersprachen

1. B. Meyer: Eiffel the Language; Prentice Hall International, 1992
2. K. Jensen, N. Wirth, A. B. Michel, F. Miner: PASCAL user manual and report; Springer,

Für störungsfreie Kommunikation

1. H.-J Schneider: Lexikon der Informatik und Datenverarbeitung, Oldenbourg 91
2. Die Rechtschreibung – DUDEN Band 1; Bibliographisches Institut, 19xx
3. Ein gutes englisches Wörterbuch.

Kapitel 1

Einführung

(U. Andelfinger, W. Henhapl)

Sie haben den Studiengang Informatik, Wirtschaftsinformatik oder das Nebenfach Informatik gewählt. Ihre Wahl haben Sie wahrscheinlich nur intuitiv getroffen, da der Begriff Informatik allgemein und insbesondere aus wissenschaftlicher Sicht nicht klar definiert ist. Informatik als wissenschaftliche Disziplin ist erst 30 bis 40 Jahre alt und daher noch in stürmischer Entwicklung. Eine allgemein akzeptierte Definition gibt es bis heute nicht und wenn es sie gäbe, wäre sie bei Ihrem Studienabschluß sicherlich überholt. Trotz dieser Schwierigkeiten, genau anzugeben, was denn Informatik sei, lassen sich folgende Eingrenzungen angeben, zwischen denen die Aufgaben und Inhalte der Informatik sich bewegen:

Der Kern der Informatik ist es, Verfahren zur Problemlösung für verschiedenste Aufgabenbereiche zu entwicklen. Insbesondere sollen dabei formale Verfahren auf logischer und mathematischer Basis angewendet werden, die eine Ausführung durch Maschinen, d.h. Computer ermöglichen. Diese Verfahren werden auch als *Algorithmen* bezeichnet und formal als *Programme* beschrieben. Mit dem Computer kann man diese formale Beschreibungen durchführen. Je mehr Probleme und Aufgaben in unserem Leben aber durch solche Computerverfahren "gelöst" werden, umso stärker muß man auch mit "Nebenwirkungen" rechnen:

- Die Ampeln auf Ihrem weg zur Hochschule werden längst von Computerprogrammen gesteuert und erlauben so einen besseren Verkehrsfluß als vorher: Morgens in die Stadt werden andere Schaltzeiten eingesetzt als abends stadtauswärts. Die bessere Steuerung hat es ermöglicht, mehr Verkehr als früher in den gleichen Zeitraum zu erlauben. Fällt nun aber einmal der Computer in der Ampelsteuerung aus, haben wir ein größeres Verkehrsproblem als jemals zuvor. Es zeigen sich typische Entzugserscheinungen.
- Oder denken denken sie an die Diskussion um den Datenschutz: Während dieses Wort heute alltäglich geworden ist, ist es erst ungefähr zwanzig Jahre her, daß es dieses Wort überhaupt gibt und Hessen das erste Datenschutzgesetz weltweit (!) erlassen hat.

Diese Beispiele zeigen, daß die Informatik zwar Probleme lösen helfen kann, daß aber zum Teil auch neue Probleme entstehen können. Daher muß die Informatik neben der technischen Lösung von Problemen auch immer darauf achten, wie sich die Lösungen in den Anwendungsbereich einfügen läßt, und welche Probleme dabei eventuell neu entstehen können. Die Informatik kann zwar diese Probleme der sogenannten Technikfolgenabschätzung und sozialverträglichen Technikgestaltung nicht alleine lösen, hier ist sie auf die Mitarbeit vieler Wissenschaften angewiesen. Dennoch ist es wichtig, sich immer vor Augen zu halten, daß technische Verfahren der Informatik, ihre Anwendungen in unserer Welt und die Folgen daraus eng miteinander zusammenhängen. Im Hauptstudium bieten sich für Sie einige Wahlmöglichkeiten, wo Sie sich mit diesen Schnittstellen zu anderen Fachgebieten intensiver vertraut machen können.

Eine eher theoretische und bis ins philosophische reichende Aufgabe der Informatik ist, sich mit der Informationsverarbeitung allgemein zu befassen. Was ist Information, wie wird sie im Menschen "verarbeitet", wie kann der Mensch Schlußfolgerungen ziehen oder bei einer Aussage feststellen, ob sie wahr oder falsch ist? Die-

ser Zweig der Informatik beschäftigt sich auch mit Fragen wie z.B.: Was ist Intelligenz? Gibt es Unterschiede zwischen menschlicher und maschineller Informationsverarbeitung? Können Maschinen lernen?

Eine etwas überspitzte Definition wird hierzu in [Baumann, 1990] gegeben: "So wie die Physik die exakte Theorie der Natur ist, so ist die Informatik die exakte Theorie und technische Nachkonstruktion des Geistes". In dieser Defintion wird nicht mehr gefragt, was der Mensch denn ist. Es wird direkt davon ausgegangen, daß es für den Geist eine exakte Theorie gibt, und daß diese technisch nachkonstruiert werden kann. Dieser Anspruch wird von vielen in der Informatik als unrealistisch und von den anderen Disziplinen (Mathematik, Psychologie, Philosophie, Theologie) als unverschämt und gefährlich bzw. unverantwortlich angesehen.

Zusammenfassend läßt sich die Informatik ungefähr zwischen folgenden Ansprüchen und Inhalten eingrenzen:

> Der Kernpunkt der Informatik ist die Entwicklung von Verfahren, deren formale Beschreibung als Programme auf einem Computer durchführbar ist. Die Reduktion des Anspruchs und Inhalts von Informatik auf eine *praktikable* Theorie und Technik der Informationsverarbeitung (daraus stammt das Kunstwort "Informatik") und Algorithmenentwicklung engt jedoch Forschungszweige wie die künstliche Intelligenz zu sehr ein. Wir können also annehmen, daß für alle die Informatik zwischen einer "Theorie und Technik der Informationsverarbeitung" als untere Grenze und der "Theorie und Technik des Geistes" als (unerreichbare und ethisch umstrittene) obere Grenze liegt.

Der Vorteil der Definition "Theorie des Geistes" ist, daß sie auch die Probleme deutlicher macht:

- So wie der menschliche Geist ambivalent ist, so sind natürlich auch seine Nachkonstruktionen ambivalent. Auch der "Ungeist" kann realisiert werden. Besonders problematisch wird dies dann, wenn sich der der technische Ungeist einmal selbständig macht, wer ist dann z.B. verantwortlich, wie kann der Ungeist wieder gestoppt werden usw.
- Nachkonstruktionen stehen immer in Konkurrenz zu dem Original. Somit sind Konflikte bei der Reorganisation von menschlichen Aufgaben mit Hilfe von Informationssystemen zwangsläufig. Problematisch wird dies nicht nur bei dem Ausfall der technischen Nachkonstruktion, sondern es werden auch Fragen nach dem Selbstwert des Menschen aufgeworfen, da seine speziellen Fähigkeiten nicht mehr gebraucht werden: Während es früher Jahre brauchte, bis ein Metallfacharbeiter genügend Gefühl und Gehör für saubere Dreharbeiten entwickelt hatte, kann dies jetzt eine computergesteuerte Drehmaschine alleine durch ein Programm, und zwar rund um die Uhr.
- Eine weitergehende Fragestellung könnte hier überspitzt lauten: Was ist das Ziel der Technik und speziell auch der Informationstechnik: den Menschen unterstützen oder den Menschen ersetzen?

Das sind Beispiele für die Fragen, zu denen die Definition von Baumann anregt, und die nur interdisziplinär diskutiert und gelöst werden können. Im Studienplan ist daher für diesen Zweck der geistes- und gesellschaftswissenschaftliche Anteil von mindestens sechs SWS vorgesehen. Diese Themen werden auch in den Fachverbänden (Gesellschaft für Informatik GI, Association for Computing Machinery, ACM, IEEE Institute of Electrical and Electronics Engineers, Vereinigung deutscher Ingenieure VDI usw.) unter dem Schlagwort "Computerethik" diskutiert. Sie sollten die Angebote nutzen, um eine eigene Meinung entwickeln und vertreten zu können. Voraussetzung ist jedoch eine fundierte Kenntnis der Grundlagen der Informatik, wie sie im Grundstudium angeboten wird.

1.1 Das Ziel: Qualitativ hochwertige Informationssysteme

Die meisten Informatikstudenten besitzen bereits vor dem Studium Vorkenntnisse über Programme und Computer, teils aus dem Informatikunterricht in der Schule, teils aus eigenen Erfahrungen im Umgang mit Personal Computern. Einige von ihnen beherrschen ihren PC besser als die meisten Mitarbeiter und Professoren der Hochschule und haben gelernt, die Feinheiten diverser Programmiersprachen optimal auszunutzen. Es ist

jedoch trügerisch, diese Vorkenntnisse mit der Berufsqualifikation eines Informatikers gleichzusetzen. Das Erlernen einer konkreten Programmiersprache ist einer der unbedeutendsten Aspekte des Informatikstudiums.[1]

Viel bedeutender ist das Erlernen neuer Denkweisen und systematischer Vorgehensweisen. Das Informatikstudium soll Sie darauf vorbereiten, sehr komplexe Informationssysteme für fremde Anwender zu entwerfen wie zum Beispiel die Steuerungsprogramme für ein Verkehrsflugzeug. Dabei stehen ganz andere Fragen im Vordergrund als bei der Programmierung von kleinen Programmen (in der Größenordnung von 500 bis 5000 Zeilen Programmtext) für einen PC, den nur Sie selbst benutzen.

Schon wegen der Größe der Aufgabenstellung müssen solche Systeme in Projekten entwickelt werden, in denen mehrere Mitarbeiter verschiedene Teilkomponenten erstellen, die später dann richtig zusammenpassen müssen. Niemand wird mehr in der Lage sein, alle Details des Gesamtprogramms zu überschauen. Aus diesem Grunde ist es wichtig, daß sich alle Beteiligten an gewisse – gemeinsam vereinbarte – Rahmenbedingungen halten und allzu geniale Tricks vermeiden.

Ihre zukünftige Aufgabe wird daher zum großen Teil gestalterischer Natur sein. Sie müssen komplexe Informationssysteme entwerfen und die für die Realisierung notwendigen Werkzeuge (wie z.B. spezialisierte Programmiersprachen) entwickeln. Durch Ihr Informatikstudium sollen Sie in die Lage versetzt werden, *qualitativ hochwertige* Systeme zu entwickeln. Was das bedeutet, kann man am besten an einer Reihe von Forderungen festmachen, die an Softwareprodukte gestellt werden:

1. Wichtigstes Qualitätskriterium ist die *Korrektheit*, also die Fähigkeit eines Programms, seine Aufgaben exakt zu erfüllen. Es mag vielleicht noch erträglich sein, daß ein Programmfehler auf Ihrem PC die Löschung der gesamten Harddisk auslöst. Auf keinen Fall aber darf es vorkommen, daß kleine Vorzeichenfehler in einem unbedeutenden Unterprogramm eines Flugzeugsteuerungsprogramms mitten im Flug plötzlich einen Umkehrschub auslöst. Wenn die Sicherung der Korrektheit von Programmen nicht oberste Priorität hat, können kleine Nachlässigkeiten bereits katastrophale Folgen haben.

   Um Korrektheit sicherzustellen, muß man natürlich eine *genaue Formulierung der Anforderungen und Spezifikationen* erstellen und dann nachweisen, daß diese auch erfüllt werden. In der Praxis ist dies nur schwer zu erreichen, da zu wenige Leute darin geschult sind, Systemanforderungen exakt zu formulieren und die Korrektheit zu *verifizieren*.

2. *Robustheit*, d.h. die Fähigkeit eines Programms, auch unter außergewöhnlichen Bedingungen zu funktionieren, ist ebenfalls sehr wichtig. Hierdurch wird z.B. vermieden, daß Fehlbedienungen durch ungeübte Anwender zu katastrophalen Ereignissen führen können.

3. Ein Softwareprodukt ist eigentlich niemals fertig, da sich im Laufe der Zeit die Ansprüche der Benutzer wandeln können. Deshalb sollten Programme *erweiterbar* sein, d.h. so geschrieben sein, daß sie leicht an geänderte Anforderungen angepaßt werden können. Dies ist bei kleinen Programmen normalerweise keine Schwierigkeit, wohl aber bei großen Softwaresystemen, die oft zusammenbrechen, wenn man nur eine kleine Komponente austauscht. Die Erfahrung zeigt, daß die *einfach strukturierte* Programme, die *dezentral* aufgebaut sind (also aus vielen *unabhängigen* Komponenten bestehen), erheblich leichter zu erweitern sind, als eng verzahnte, auf Effizienz getrimmte Programme.

4. In vielen verschiedenen Softwaresystemen werden Sie Elemente vorfinden, die immer nach dem gleichen Muster gebaut sind. So gibt es zum Beispiel unzählige Sortierprogramme in den verschiedensten Anwendungsprogrammen, die aber jedes Mal von Grund auf neu programmiert wurden. Viel sinnvoller wäre es, Programme so zu entwickeln, daß sie ganz, oder zumindest zum Teil für neue Anwendungen *wiederverwendet* werden können. Das ist nicht nur effizienter und spart Entwicklungskosten, sondern verbessert auch die Zuverlässigkeit, da Fehler bei der Programmierung von Routineprogrammen vermieden werden können.

[1] Wenn Sie die Denkformen der Informatik erst einmal erfaßt haben, werden Sie eine neue Programmiersprache in 2–3 Tagen erlernen können.

5. *Kompatibilität* ist ein Maß dafür, wie leicht ein Programm mit anderen Programmen verbunden werden kann. Diese Eigenschaft ist wichtig, wenn ein Programm Informationen von einem fremden Programm weiterverarbeiten soll. Dabei gibt es aber oft Probleme, z.B. weil – gerade in der MS-DOS Welt – eine Vielfalt unterschiedlicher Dateiformate benutzt wird, die erst mühsam konvertiert werden müssen. Der Schlüssel zur Kompatibilität liegt in der Vereinbarung von *Standards* für die Kommunikation von Programmen und in einer gewissen *Einheitlichkeit* des Entwurfs.
6. Eine grundlegende Anforderung an jedes Softwareprodukt ist natürlich auch die *Effizienz*, d.h. die ökonomische Nutzung von Hardwareressourcen sowohl bezüglich Raum als auch bezüglich der Zeit. Auch wenn diese Anforderung durch die enormen Verbesserungen der Hardware nicht mehr ganz so bedeutend ist wie vor 15 Jahren, kann auf prinzipielle *Komplexitätsbetrachtungen* nicht verzichtet werden.
7. *Portabilität* ist das Maß dafür, wie leicht ein Programm auf verschiedene Hardware- und Software-Umgebungen übertragen werden kann.
8. *Verifizierbarkeit* ist das Maß dafür, wie leicht Daten und Prozeduren zur Fehlererkennung und -verfolgung während der Betriebsphase eines Programms erzeugt werden können.
9. *Integrität* ist die Fähigkeit eines Systems, seine Komponenten und Daten gegen unberechtigten Zugriff zu schützen.
10. *Benutzerfreundlichkeit* ist das Maß dafür, wie leicht ein der Umgang mit einem Softwaresystem von ungeübten Benutzern erlernt werden kann.

Nicht alle Qualitätsfaktoren sind miteinander verträglich. So schränkt z.B. die Forderung nach Integrität die Benutzerfreundlichkeit ein und die Forderung nach Portabilität die Effizienz. Es wird Ihre Aufgabe als Informatiker sein, zwischen diesen Rahmenbedingungen abzuwägen und auf der Grundlage Ihrer Kenntnisse einen Kompromiß zu erzielen. Deshalb soll Ihnen das Grundstudium zwei Arten von Fähigkeiten vermitteln.

- Zum einen sollen Sie die *theoretisch-formalen* Grundlagen kennenlernen, die nötig sind, um informationstechnische Prozesse überhaupt beschreiben zu können. Deswegen erhalten Sie in den ersten beiden Semestern eine Einführung in *formale Systeme*, die maschinell verarbeitet werden können, wie z.B. Logik, Programmiersprachen und Schaltungen. Darüberhinaus sollen Ihnen die mathematisch-theoretischen Grundlagen von formalen Systemen und die Grenzen Ihrer Anwendbarkeit verdeutlicht werden. Das geschieht allerdings erst im vierten Semester.
- Zum anderen sollen Sie aber auch *Techniken und Methoden* erlernen, formale Systeme optimal für Ihre Aufgaben einzusetzen. Leitthemen der ersten 3 Semester sind daher das *Programmieren*, also die Anwendung der Grundkonzepte von Programmiersprachen, um technische Systeme zu beschreiben, und vor allem auch *Strukturierungsmethoden für formale Systeme*. Letztere betreffen sowohl die Organisation des Entwicklungsprozesses an sich (den sogenannte *Life Cycle* der Systementwicklung) als auch die Strukturierung des zu entwickelnden Systems (*Modularisierung*). Parallel zu den Konstruktionsmethoden werden wir immer auch über Methoden reden, mit denen Sie die Qualität der von Ihnen konstruierten Systeme sicherstellen oder überprüfen können, also Verfahren zum Testen und Messen oder zum *Beweisen* von Eigenschaften der Systeme. Dabei wird wird Korrektheit und Effizienz im Vordergrund stehen.

Um Ihnen die Bedeutung dieser Begriffe exakt zu erklären, werden wir im Endeffekt das gesamte Grundstudium benötigen. Im folgenden wird trotzdem der Versuch gemacht, eine Einführung in die Denkweisen der Informatik zu geben. Sie ist für das Verständnis der weiteren Kapitel hilfreich, aber nicht unbedingt erforderlich. Sie können es ruhig überspringen und vielleicht später (mit etwas mehr Verständnis) durcharbeiten.

1.2 Theoretische Schwerpunkte der ersten beiden Semester

Alle informationstechnischen Systeme basieren auf formalen Systemen: *Programmiersprachen, Logik, Programmbibliotheken* usw. Die Hauptaufgabe des Informatikers ist, informationstechnische Abläufe derart formal

in Form eines Programms zu beschreiben, daß sie von einer Maschine (Computer) nach festen Regeln interpretiert werden können. Aufgrund der fortgeschrittenen Technik der Hardware aber auch der Software sind Fehler in der Interpretation von Programmen selten. Unentdeckte Hardwarefehler treten inzwischen extrem selten auf und die geprüften Übersetzer sind praktisch fehlerfrei.[2]

In den Grundzügen der Informatik spielen in den ersten beiden Semestern die höheren und die maschinennahen Programmiersprachen die Hauptrolle. Fast alle Programme am Markt sind in Sprachen aus diesen beiden Bereichen programmiert. Sie sind gekennzeichnet durch Kombinationen von Anweisungen und heißen daher auch *imperative* Programmiersprachen. Hinter der Kennzeichnung imperativ steckt die Idee, daß man einer Maschine eine Folge von Befehlen gibt, die sie dann der Reihe nach abarbeitet. Aber man muß auch beschreiben können, was eigentlich ein Programm leisten soll. Dazu brauchen wir *deskriptive* Sprachen, die aus der Logik stammen. Der Ausdruck deskriptiv sagt, daß man beschreibt, *was* ein System leisten soll, ohne anzugeben, *wie* diese Leistung erreicht werden kann. Für das Verständnis, wie ein Programm einer Programmiersprache zu interpretieren ist, werden wir eine *funktionale* Sprache verwenden. Funktionale Sprachen basieren auf der Idee, daß Programme aus ihren Eingaben eine Ausgabe berechnen, also im wesentlichen aus Funktionen bestehen, wie sie in der Mathematik üblich sind. Funktionale Sprachen sind daher am leichtesten zu verstehen.

Im Prinzip kann man mit jedem dieser drei Sprachtypen dieselben Aufgaben bewältigen, jedoch mit unterschiedlicher Eleganz, da die Wahl der Programmiersprache auch die Sicht auf die Welt bestimmt, die man mit seinem Programm modelliert. Um ein Gefühl für die verschiedenen Konzepte dieser Sprachen zu bekommen, stellen wir sie am Beispiel des größten gemeinsamen Teilers einander gegenüber. Dabei werden wir für jeden Sprachtyp das Programm für den größten gemeinsamen Teiler darstellen und dann anschließend besprechen, nach welchen Regeln eine Maschine dieses Programm interpretieren könnte.

Das Wort Programm ist im allgemeinen Wortschatz mehrdeutig. In unseren Kontext bedeutet es ein formal beschriebenes Verfahren, das für beliebig vorgegebene Werte durchgeführt werden kann. Es ist also mehrfach durchführbar. Eine andere Bedeutung hat das Wort Programm im Wort "Besuchsprogramm". Ein Besuchsprogramm ist zwar auch eine Beschreibung eines Ablaufs aber eben nur für einen speziellen Gast.

Aus der Schule ist der größte gemeinsame Teiler noch als eine Funktion `ggt` bekannt, die zwei natürliche Zahlen $m,n \in \mathbb{N}$ in eine natürliche Zahl $t \in \mathbb{N}$ abbildet. Um diese Funktion zu erklären, ist der Begriff "eine Zahl teilt eine andere" notwendig. Der Begriff beschreibt eine Aussage zwischen zwei Zahlen, die entweder wahr oder falsch ist: Die Aussage "a teilt b" – wir schreiben dafür `teilt(a,b)` – ist genau dann wahr, wenn a einer Teiler von b ist. Formal ist die Aussage `teilt` eine Funktion mit zwei Argumenten aus $\mathbb{N}$ und dem Resultat aus den beiden logischen Werten **wahr** und **falsch**, d.h. aus der Menge $\mathbb{B} = \{$**wahr**,**falsch**$\}$:

`teilt`$:\mathbb{N}\times\mathbb{N}\rightarrow\mathbb{B}$ *Lesweise:* $\mathbb{N}$ *mal* $\mathbb{N}$ *wird in* $\mathbb{B}$ *abgebildet*

mit der für uns wichtigen Eigenschaft, daß eine Zahl a immer ein Teiler von sich selbst ist und jeder der beiden Faktoren von a*b Teiler von a*b ist:

für alle $\mathtt{a} \in \mathbb{N}$ gilt `teilt(a,a) =` **wahr** und `teilt(a,a*b) =` **wahr**

In der Logik hat sich bei Aussagen, die stets wahr sein sollen, eingebürgert "`=` **wahr**" einfach wegzulassen, "für alle" durch das Symbol $\forall$ abzukürzen sowie "und" durch $\wedge$.[3] Wir schreiben daher kürzer:

---

[2] Für die Praxis gilt daher: Der Computer irrt nicht! Fehlverhalten eines informationstechnischen Systems beruht i.a. an den Fehlern der Entwickler der Programme oder Bedienungsfehler der Anwender. Vermeiden Sie daher stets die falschen Ausdrucksweisen: "Der Computer hat Mist gebaut". Die korrekte Ausdrucksweise lautet: "Ich habe die Anleitung fehlinterpretiert" oder "der Entwickler hat ein fehlerhaftes Produkt hergestellt". Für die Vorlesung und das Praktikum haben wir aus didaktischen Gründen die noch verhältnismäßig junge Sprache Eiffel gewählt. Hierfür gibt es bis jetzt noch keinen geprüften Übersetzer!

[3] Weitere logische Symbole, die wir dem Prädikatenkalkül erster Ordnung entnehmen, sind $\neg$ für "nicht", $\vee$ für "oder", $\Rightarrow$ für "daraus folgt", $\Leftrightarrow$ für "genau dann, wenn" und $\exists$ für "es gibt". Da die Relation "genau dann, wenn" besagt, daß der Wahrheitswert der entsprechenden Aussagen gleich ist, verwenden die meisten Programmiersprachen anstelle von $\Leftrightarrow$ das Gleichheitssymbol `=`. Dies führt aber bei Aussagen wie `(4=5) = (6=7)` eher zu Verwirrung. Wir verwenden deshalb innerhalb von logischen Aussagen üblicherweise das Äquivalenzsymbol $\Leftrightarrow$ und schreiben `(4=5)` $\Leftrightarrow$ `(6=7)`.
Die genaue Beschreibung dies Prädikatenkalküls kommt im Kapitel 2.

$\forall a \in \mathbb{N}.\ \texttt{teilt(a,a)} \wedge \texttt{teilt(a,a*b)}$

Der größte gemeinsame Teiler ist nun eine Funktion ggt, die zwei Argumente aus $\mathbb{N}$ auf eine Zahl aus $\mathbb{N}$ abbildet und die Eigenschaft hat: ggt(m,n) teilt m und n und jede Zahl t, die m und n teilt, ist kleiner oder gleich ggt(m,n).

$\texttt{ggt}: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ mit

$\forall m,n \in \mathbb{N}.\ \texttt{teilt(ggt(m,n),m)} \wedge \texttt{teilt(ggt(m,n),n)}$
$\quad \wedge\ (\forall t \in \mathbb{N}.\ \texttt{teilt(t,m)} \wedge \texttt{teilt(t,n)} \Rightarrow t \leq \texttt{ggt(m,n)})$

Diese Form der Beschreibung des größten gemeinsame Teilers ist nicht konstruktiv, denn sie gibt keinen Hinweis, wie man für bestimmte Werte von m und n die Lösung berechnet. (Das Ausprobieren für alle t ist natürlich keine konstruktive Verfahrensweise, da dies nicht in endlich vielen Schritten erfolgen kann.). Die Schwierigkeit könnten wir beseitigen durch unser Wissen, daß Werte t, die größer als m oder n sind, weder ein Teiler von m noch von n sein können. Wir können in der dritten zweiten Forderung den Bereich $\mathbb{N}$, in dem wir die Lösung t suchen, zu dem endlichen Bereich $\{i \in \mathbb{N} |\ i \leq m \wedge i \leq n\}$ reduzieren:

$\forall t \in \{i \in \mathbb{N} |\ i \leq m \wedge i \leq n\}.\ \texttt{teilt(t,m)} \wedge \texttt{teilt(t,n)} \Rightarrow t \leq \texttt{ggt(m,n)}$

Der Berechnungsaufwand (Komplexität) ist natürlich bei größeren Werten von m und n trotzdem erheblich, aber wir stellen fest, daß bei endlichen Wertebereichen die Aufzählung eine maschinelle Auswertung erlaubt, falls das verwendete Aussage teilt, stets definiert ist und ihre Prüfung beschränkten Zeitaufwand erfordert.

Mit dem Wissen aus der Zahlentheorie können wir jedoch einen Schritt weitergehen. Seit Euklid wissen wir, daß eine Zahl t, die Teiler von m und n ist, auch m-n teilt, falls m größer als n ist, und n-m teilt, falls n größer als m ist. Falls die Zahlen m und n gleich sind, dann ist jede von ihnen der größte gemeinsame Teiler.

$\texttt{teilt(t,m)} \wedge \texttt{teilt(t,n)} \Rightarrow (\texttt{m>n} \Rightarrow \texttt{teilt(t,m-n)})$
$\texttt{teilt(t,m)} \wedge \texttt{teilt(t,n)} \Rightarrow (\texttt{m<n} \Rightarrow \texttt{teilt(t,n-m)})$
$\texttt{m=n} \Rightarrow \texttt{ggt(m,n) = m}$

Diese Regeln sind nun der Ausgangspunkt unserer weiteren Betrachtungen.

1.2.1 Logik für die Problemlösung

Wir beginnen mit der "höchsten" Sprachebene, der deskriptiven Beschreibung des größten gemeinsame Teilers. In dieser Ebene hat man den Wunsch, allgemeine Fragen an die Funktion ggt zu stellen: z.B. welchen Wert ergibt ggt(12,15), für welche Werte m gilt ggt(m,4)=2? In der Logik werden allerdings nicht Funktionen, sondern *Aussagen* behandelt. Wir betrachten daher nicht die Funktion ggt, sondern die Aussage GGT, die drei Argumente hat: m und n wie bei der Funktion ggt, aber ein drittes Argument t mit der Bedeutung:

GGT(m,n,t) ist genau dann wahr, wenn ggt(m,n)=t.

Formal ist dann $\texttt{GGT}: \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{B}$. Die obigen Anfragen sind also die Suche nach der Menge aller Werte t, für die GGT(12,15,t) wahr ist, bzw. aller Werte m für die GGT(m,4,2) erfüllt ist.

1.2.1.1 "Logisches" Programm

Die obigen Eigenschaften lassen sich in der neuen Darstellung folgendermaßen beschreiben:

1. $\texttt{m>n} \Rightarrow (\ \texttt{GGT(m,n,t)} \Leftrightarrow \texttt{GGT(m-n,n,t)}\ )$
   "GGT(m-n,n,t) ist genau dann wahr, wenn GGT(m,n,t) wahr ist, falls m größer als n ist".
2. $\texttt{m<n} \Rightarrow (\ \texttt{GGT(m,n,t)} \Leftrightarrow \texttt{GGT(m,n-m,t)}\ )$
   "GGT(n-m,n,t) ist genau dann wahr, wenn GGT(m,n,t) wahr ist, falls m kleiner als n ist".

3. `m=n ⇒ ( GGT(m,n,t) ⇔ (t=m) )`
   "GGT(m,n,t) ist genau dann wahr, wenn t=m ist, falls m gleich n ist".

Diese drei Regeln sind bereits ein Logik-Programm. Sie sind konstruktiv, denn sie erklären, daß die Überprüfung der Wahrheit von GGT(m,n,t) sich auf die Überprüfung von GGT mit kleineren Argumenten m-n bzw n-m zurückführen läßt, falls m ungleich n ist. Da m und n natürliche Zahlen sind, muß diese Reduktion auf kleinere Argument einmal mit der Aussage, nun ist m=n, abbrechen. Dann haben wir aber das Ergebnis. Die Tatsache, daß zur Erklärung des GGT's wiederum der GGT benutzt wird, wird als *Rekursion* bezeichnet. GGT ist also rekursiv definiert.

1.2.1.2 Maschinelle Interpretation des logischen Programms

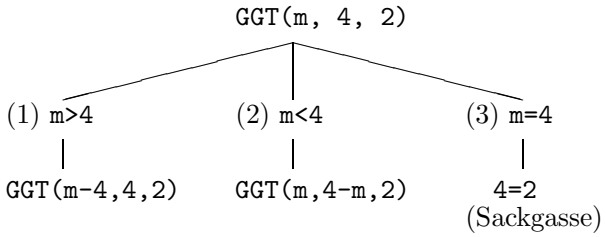
Hier setzt nun ein *maschinell durchführbares* Verfahren für die *Interpretation* auf. Man versucht für gegebene Argumente, diese Argumente mit Hilfe der Eigenschaften (Regeln) solange über die Gleichungen zu reduzieren, bis man das Ergebnis erhält.

Beispiel 1.2.1 *Gesucht ist ein t für das* `GGT(12,15,t)` *wahr ist. Das Verfahren ist hier sehr einfach: Wir prüfen, welche Regel überhaupt anwendbar ist. Da sich die Bedingungen für m und n gegenseitig ausschließen, kann jeweils nur eine Gleichung zur Reduktion verwendet werden:*

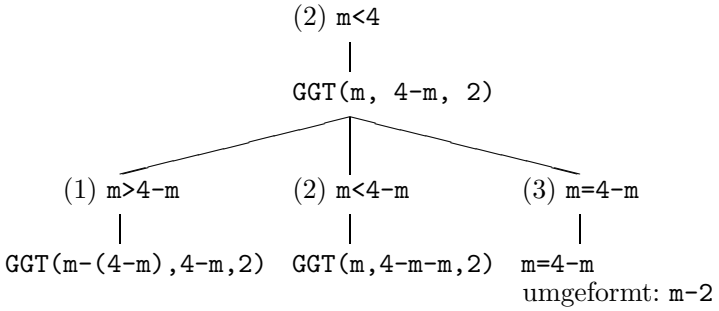
Nach (2)	gilt	`GGT(12, 15 ,t)`	⇔	`GGT(12, 3, t)`
(1)		`GGT(12, 3, t)`	⇔	`GGT(9, 3, t)`
(1)		`GGT(9, 3, t)`	⇔	`GGT(6, 3, t)`
(1)		`GGT(6, 3, t)`	⇔	`GGT(3, 3, t)`
(3)		`GGT(3, 3, t)`	⇔	`(t=3)`

Also gilt `(t=3)` ⇔ `GGT(3,3,3)` ⇔ `GGT(6,3,3)` ⇔ ... ⇔ `GGT(12,15,3)`. *Wir haben eine Lösung gefunden.*

Beispiel 1.2.2 *Komplizierter wird der Fall für* `GGT(m,4,2)`. *Hier gilt nicht mehr, daß sich die einzelnen Regeln gegenseitig ausschließen, da ja der Wert von m unbekannt ist und daher alle drei Regeln verfolgt werden müssen. Wir wenden nun stets alle drei Regeln an und notieren uns bei jeder, unter welchen Voraussetzungen die Gleichung angewandt wurde:*



Die Verbindung bedeutet die Gleichheit der logischen Werte, aber nur unter der Bedingung, daß die Voraussetzung, die an der Verbindung angeklebt ist, erfüllt ist.

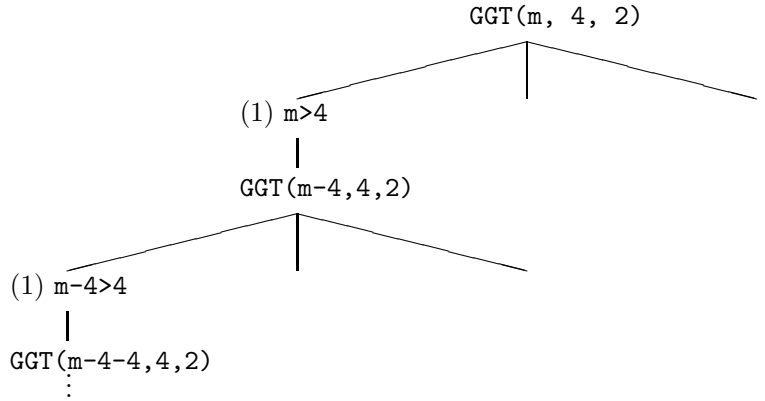


Wir haben also eine Lösung m=2 gefunden.

So zielgerichtet, wie wir hier vorgegangen sind, kann man jedoch bei einer allgemeinen Suche nach der Lösung i.a. nicht vorgehen. Es müßten alle Ersetzungen ("Suche in die Breite"), und damit eventuell unendlich viele,

durchgeführt werden, sofern man nicht in eine Sackgasse gerät. Zum Beispiel müßte auch der erste Ast weiterverfolgt werden, um weitere Lösungen zu bekommen. An diesem Beispiel zeigen sich die Probleme bei einer logischen Beschreibung. Wir können Probleme stellen, die nicht lösbar sind. Im obigen Beispiel ist die Forderung, die Lösungsmenge durch Aufzählung aller einzelnen Lösungen anzugeben. Da wir wissen, daß die Lösungsmenge alle ganzen durch 2, aber nicht durch höhere Potenzen von 2, teilbaren Zahlen enthält, ist klar, daß das Verfahren nicht aufhören (*terminieren*) kann, da diese Menge unendlich ist.

Wir können aber auch Lösungs*verfahren* wählen, die überhaupt kein Element der Lösungsmenge erbringen: Die Suche soll zuerst vollständig über (1) und dann erst über die beiden anderen Regeln erfolgen. Wendet man diese Vorgangsweise immer wieder an ("Suche in Tiefe"), so suchen wir nach einem immer größer werdenden m ohne jemals zu einer Lösung zu gelangen:



Allgemeine Verfahren zu entwickeln, die für logische Formeln mit Variablen diejenigen Variablenbelegungen finden, welche diese Formel erfüllen, ist eines der Ziele der Künstlichen Intelligenz (an der TH "Intellektik" im Sinn einer Theorie des Geistes). Die Sprache Prolog und deren Deduktionssystem ist ein Ergebnis dieser Bemühungen.

1.2.1.3 Diskussion

Der Vorteil dieser Sprache und ähnlicher deskriptiver Sprachen ist, daß man nur die Bedingung formulieren muß, *was* man vom Resultat erwartet, aber kein Verfahren (einen Algorithmus) anzugeben braucht, *wie* man zur Lösung kommt. Es gibt einen eingebauten *einheitlichen* Algorithmus zur Suche nach den Lösungen. Das Programmieren wird daher überflüssig, oder präziser fast überflüssig, da man zur Vermeidung von endlosen Auswertungen diesen allgemeinen Algorithmus etwas steuern muß.

Es ist klar, daß die Effizienz der Lösung eines Problems umso größer ist, je mehr man das Lösungsverfahren an das Problem anpassen kann. Da dies bei deskriptiven Sprachen nur geringfügig möglich ist, ist es zumeist sinnvoller, ein für das Problem spezifisches Verfahren direkt anzugeben. Dafür gibt es Sprachen, die es erlauben, diese problemspezifischen Verfahren direkt als Programm zu beschreiben. Für die Sprache ist dann wiederum ein festes Verfahren gegeben, wie Programme dieser Sprache ausgeführt werden. Im wesentlichen gibt es zwei Arten. Die eine basiert auf Funktionen im Sinne der Mathematik (z.B. Lisp, ML), die andere auf einem Maschinenbegriff, der durch von Neumann definiert wurde.

1.2.2 Funktionen für die Problemlösung

Ausgangspunkt dieser Form der Programmierung ist eine Menge von Basisfunktionen (Addition, Multiplikation, usw.) und die Möglichkeit, aus diesen Funktionen weitere zusammenzubauen. Als Kombinationsmöglichkeiten stehen mindestens die *Komposition* (Funktionszusammensetzung) und die Fallunterscheidung zur Verfügung:

1. Zwei Funktionen `f:A →B` und `g:B →C` können durch *Komposition* hintereinander ausgeführt werden. Man bildet also Ausdrücke der Art `g(f(a))` und damit eigentlich eine neue Funktion `g∘f:A →C`.

2. Die Fallunterscheidung `if..then..else` erlaubt es, Werte abhängig von einer Bedingung zu berechnen:

   `Wert = if Bedingung then Wert1 else Wert2` entspricht $\texttt{Wert} = \begin{cases} \texttt{Wert}_1 & \text{falls } \texttt{Bedingung} \text{ wahr ist} \\ \texttt{Wert}_2 & \text{sonst} \end{cases}$

1.2.2.1 Funktionales Programm

In unseren Beispiel können wir die Regeln mit Hilfe der Fallunterscheidung zu einer Funktion zusammenbauen:

```
ggt(m,n) = if m>n
              then ggt(m-n,n)
              else if m<n
                      then ggt(m,n-m)
                      else m
```

Der Wert von `ggt(m,n)` wird in diesem funktionalen Programm auf Werte von ggt mit kleineren Argumenten zurückgeführt. Damit liegt auch hier eine konstruktive Beschreibung vor. ggt ist genauso wie in der logischen Beschreibung rekursiv beschrieben.

1.2.2.2 Maschinelle Interpretation

Das allgemeine Verfahren zur Interpretation der Programme ist nun auf die Fragestellung `ggt(m,n)=?` beschränkt. Die Problemstellung `ggt(?,4)=2` ist nicht mehr zulässig. Die Auswertung erfolgt nach aus der Mathematik bekannten Verfahren, bei dem aktuelle Argumente in die Funktionsdefinition eingesetzt werden:

Beispiel 1.2.3 *Wir berechnen* `ggt(12,15)`*:*

```
ggt(12,15) = if 12>15
                then ggt(12-15,15)
                else if 12<15
                        then ggt(12,15-12)
                        else 12
```

Die Auswertung der Fallunterscheidung erfolgt durch die Berechnung der Bedingung. Ist sie wahr so ist der Wert der Funktion der Wert, der hinter dem `then` *steht, ansonsten der Wert, der hinter dem* `else` *steht. Es wird nur einer der beiden weiterberechnet! Wir können daher die Formel auf den* `else`*-Zweig reduzieren:*

```
           = if 12<15
                then ggt(12,15-12)
                else 12
```

Da `12<15`*, erfolgt die Reduktion auf den* `then`*-Zweig:*

```
           = ggt(12,15-12)
           = ggt(12, 3)
```

Nun beginnt wieder die Anwendung der Funktionsdefinition:

```
           = if 12>3
                then ggt(12-3, 3)
                else if 12<3
                        then ggt(12,3-12)
                        else 12
```

Nach der Auswertungsregel für `if`*:*

```
           = ggt(12-3,3)
           = ggt( 9, 3)
             ⋮
           = 3
```

1.2.2.3 Diskussion

Bei der deskriptiven Problembeschreibung gab es keinen Aufwand für die Programmierung, d.h. Problembeschreibung ist gleichzeitig die Problemlösung. Für funktionale Programme muß man die deskriptive Beschreibung (Regeln 1 bis 3 aus 1.2.1.1) erst in ein Programm umsetzen und diese Umsetzung auch begründen (beweisen) können. Der Aufwand ist also höher. Wozu dann das Ganze, wenn man zusätzlich keine Fragen (`ggt(?,n)=t`) stellen, sondern nur Funktionen (`ggt(m,n)=?`) beschreiben kann? Im wesentlichen liegt der Unterschied in der Effizienz. In funktionalen Programmen beschreibt man direkt das Verfahren und überläßt die Lösungssuche nicht einer allgemeinen Suchstrategie. Das Opfer, das zugunsten einer verbesserten Effizienz gebracht werden muß, ist die *Programmentwicklung* und wesentlicher noch die *Rechtfertigung* oder *Verifikation* des Programms. In unserem Fall ist die Rechtfertigung noch einfach: Die Implikationen der Regeln werden in die beiden Fallunterscheidungen überführt. Die Zulässigkeit dieser Transformation kann bewiesen werden.

Der Programmieraufwand für funktionale Programme auf der Basis einer deskriptiven Problembeschreibung ist relativ gering, wie Sie im Vergleich zu den nächsten Sprachen sehen werden. Da sie konzeptuell sehr einfach und daher leicht erlernbar sind, werden sie an anderen Universitäten als erste Programmiersprachen in dem Grundstudium eingeführt. Daß sie noch nicht allgemein verwendet werden, liegt ebenso wie bei deskriptiven Sprachen an der *Effizienz*. In diesen Sprachen steht keine Möglichkeit zur Verfügung einen Wert zu verändern. Bei der Auswertung von Formeln benötigen die neuen Werte auch neue Plätze im Speicher. Das ist besonders kritisch bei komplexen Werten wie Matrizen, bei denen sich nur eine Teilinformation ändert (Wertänderung eines Elements der Matrix). Hier wird stets eine vollständig neue Version des komplexen Wertes aufgebaut, statt nur in der alten Version die Teilinformation zu ändern. Der Effekt ist, daß viel Zeit und Platz benötigt wird beim fortwährenden Umbau von Werten (Matrizen, die sich nur in einem Wert voneinander unterscheiden). Eine Vermeidung dieses Aufwands kann man durch Kenntnis des "Platzes" der Teilinformation erreichen, um dort nur die Teilinformation aber nicht die anderen Teilkomponenten zu ändern.

Wir kommen damit zu einem Speicherbegriff, der *Plätze bereitstellt*, in denen Werte abgespeichert und dort auch abgeändert werden können. Diese Abänderungvorschriften nennt man Anweisungen. Sie bilden die Basis der *imperativen* Programmiersprachen. Ist die notwendige Kenntnis der Speicherstruktur beschränkt auf das Wissen, daß z.B. nur ganze Zahlen, reelle Zahlen, Texte usw. gespeichert werden können, so spricht man von *höheren* oder *problemorientierten* Programmiersprachen (Pascal, Modula, Ada, Fortran, Cobol, Eiffel, disziplinert verwendetes C usw.). Ist jedoch auch das Wissen über die interne Darstellung der Werteklassen als Bitmuster (Codierung der Werte in eine Folge von Nullen und Einsen) notwendig, so spricht man von *maschinennahen* Sprachen (Familie der Assembler, aber z.T. auch C). Jede dieser Programmiersprachen basiert auf einem ihr eigenen abstrakten "Maschinenmodell", das erklärt, wie die Programme durchgeführt werden. Für die höheren Programmiersprachen ist dieses Modell konzeptionell einfach, für die maschinennahen Sprachen durch seine Menge von Details unübersichtlich.

1.2.3 Problemorientierte imperative Sprachen

Im Gegensatz zu den funktionalen Sprachen, deren Interpretation auf dem bekannten Mechanismus der Funktionsauswertung aufsetzt, müssen wir für die Programmiersprachen zuerst das Grundkonzept einer *abstrakten Maschine* erklären, damit die Anweisungen an diese Maschine verständlich sind.

In ihrer einfachsten Form besteht eine abstrakte Maschine (für Programmiersprachen) aus einer Tabelle, die links die Variablennamen (Namen von Plätzen) und rechts Werte (die momentan gespeicherten Werte der Variablen) hat. Diese Tabelle wird *Speicher* genannt. Neben dem Speicher gibt es noch den *Befehlszähler*, der angibt, welcher Befehl im Programm als nächster durchgeführt wird. Die wichtigsten Befehle sind die Vorschrift, den Wert einer Variablen zu ändern (*Wertzuweisung*) und die Befehle, die angeben, welcher Befehl als nächstes ausgeführt werden soll (*Kontrollanweisungen*). Bei den letzteren sind die *Fallunterscheidung* und die *Schleife* die wichtigsten Befehle.

1.2.3.1 Imperatives Programm

Nun entwickeln wir den ggt weiter in Richtung eines Programms in einer imperativen, höheren Programmiersprache (Notation von Pascal). Wenn wir die funktionale Version nochmals anschauen, so sehen wir, daß durch die Rekursion immer wieder die beiden Fallunterscheidungen, jedoch mit veränderten Werten, durchgeführt werden. Geben wir dem ersten Argument den Speicherplatz m, dem zweiten Argument den Speicherplatz n und dem Resultat den Speicherplatz t, so sieht der Speicher vor der Ausführung folgendermaßen aus:

Speicher			
	m	12	
	n	15	
	t	?	*noch kein Wert bekannt*

Die Ausführung der Funktion besteht dann in der Ausführung der Fallunterscheidungen, aber statt des rekursiven Aufrufs mit den veränderten Argumenten werden die entsprechenden Speicherplätze überschrieben: `m:=m-n` bedeutet, der Wert des Speicherplatzes von m wird mit dem Wert vom m-n überschrieben:

```
if m>n
  then m:=m-n
  else if m<n
         then n:=n-m
         else t:=m
```

und dann das Programm wiederholt. Diese Wiederholung wird erst beendet, wenn m=n ist, oder anders ausgedrückt das Programmstück wird wiederholt (do), solange (while) die beiden Werte ungleich sind (m<>n):

```
1    while m<>n do
2         if m>n
3            then m:=m-n
4            else if m<n
5                    then n:=n-m
6                    else t:=m
```

In dieser Form ist das Programm inkorrekt, da die Wertzuweisung `t:=m` die Gleichheit von m und n voraussetzt, die aber durch die Schleifenbedingung ausgeschlossen ist. D.h. `t:=m` wird nie durchgeführt. der zugehörige else-Zweig wird nie erreicht. In der korrekten Version führen wir daher t:=m nach der Schleife durch:

```
1    while m<>n do
2         if m>n
3            then m:=m-n
4            else if m<n
5                    then n:=n-m
6                    else ;
7     t:=m
```

Damit ist wiederum das Programm gegeben und wir wenden uns nun dem allgemeinen Mechanismus der Programmausführung auf der abstrakten Maschine zu, die aus Speicher und Befehlszähler besteht.

1.2.3.2 Maschinelle Interpretation

Die Durchführung des Programms erfolgt durch Steuerung des Befehlszeigers, durch Änderung des Speichers über die Wertzuweisungen und durch die Berechnung der Ausdrücke, indem alle Variablen durch die entsprechenden Werte der Speicherplätze ersetzt werden und dann der Ausdruck entsprechend der normalen Rechenregeln ausgewertet werden. Der Ausgangszustand ist:

nächster Befehl:		1	
Speicher			
	m	12	
	n	15	
	t	?	

Da `12<>15` ist wird die erste Anweisung in der Schleife durchgeführt. Der Befehlszähler wird auf 2 gesetzt. Der Speicher bleibt unverändert.

nächster Befehl:		2	
Speicher			
	m	12	
	n	15	
	t	?	

Die Auswertung der Anweisung 2 bedeutet die Auswertung von der Bedingung `m>n` mit den aktuellen Speicherwerten. Diese ergibt **falsch**, daher wird der Befehlszähler auf die else-Anweisung 4 gesetzt.

nächster Befehl:		4	
Speicher			
	m	12	
	n	15	
	t	?	

Bei der Auswertung von 4 wird der Vergleich `m<n` ausgewertet, der nun den Wert **wahr** ergibt. Der Befehszähler wird dadurch auf die then-Anweisung 5 gesetzt.

nächster Befehl:		5	
Speicher			
	m	12	
	n	15	
	t	?	

Jetzt erfolgt die Speicheränderung durch `n:=n-m`. Die Auswertung erfolgt durch die Ersetzung der Variablenwerte der rechten Seite von `:=`, also m-n, mit den aktuellen Speicherwerten: 15, 12. Der errechnete Wert des Ausdrucks wird nun der neue Wert der Variablen auf der linken Seite von `:=`, also n, im Speicher. Damit ist Anweisung 5 beendet. Der Befehlszähler wird auf das Ende der Fallunterscheidung gesetzt. Da wir aber in einer Schleife sind, bedeutet dies, daß wir wieder bei 1 landen.

nächster Befehl:		1	
Speicher			
	m	12	
	n	3	
	t	?	

In der folgenden Darstellung werden die Befehlszählerwerte und Speicherwerte noch einmal vom Anfang an als Folge dargestellt:

nächster Befehl:		1	2	4	5	1	2	3	1	2	3	1	2	3	1	2	4	6	7	
Speicher																				
	m	12	12	12	12	12	12	12	9	9	9	6	6	6	3	3	3	3	3	3
	n	15	15	15	15	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
	t	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	3

1.2.3.3 Diskussion

Wie das Beispiel zeigt, kommt zum Algorithmus, wie er im funktionalen Beispiel schon im Groben vorgeben war, noch die Planung der Speicherplätze und die Programmierung der entsprechenden Wertzuweisungen hinzu. Auch die Umsetzung der Rekursion in eine Schleife bedeutet fehlerträchtigen Aufwand, da die Rekursion ein wesentlich einsichtigeres Konzept ist als die Schleife. Der Vorteil ist, wie bereits vorher erwähnt, der Effizienzgewinn, da bei den üblichen Rechnern die Wertzuweisung dem Funktionsaufruf an Geschwindigkeit deutlich überlegen ist. Für diese Effizienz muß man, neben dem komplexeren Programm, hauptsächlich mit einem größeren Begründungsaufwand für die Korrektheit des Algorithmus, gegenüber der deskriptiven und funktionalen Formulierung, bezahlen. Dieses Thema wird uns während des Semesters noch sehr beschäftigen.

1.2.4 Maschinennahe Sprachen

Maschinennahe Sprachen erlauben eine bessere Nutzung der Möglichkeiten eines Rechners. Maschinennahe Sprachen, die die Möglichkeiten eines Rechnerstyps X voll zur Verfügung stellen, nennt man *Assembler* des Rechnertyps X. Der besseren Nutzung steht der Nachteil gegenüber, daß nur die Instruktionen des Rechners X als Bausteine eines Programms zur Verfügung stehen. Das bedeutet, daß man die Speicherstruktur bis ins Detail kennen und sich auf den stark reduzierten Satz von Kontrollstrukturen und Wertzuweisungen beschränken muß. Ein Wechsel von einem Assemblerprogramm des Rechnertyps X auf einen Rechnertyp Y ist nicht möglich. Die Assemblerprogramme sind *nicht portabel*.

In Assemblersprachen gibt es i.a. keine Schleife, und in der Fallunterscheidung gibt es nur den Vergleich mit einer speziellen Variablen "Condition Code". Die Vergleiche müssen als Wertzuweisungen an diese Variable `cc` (in Pascalnotation `cc:= m<n`) programmiert werden, der dann durch einen bedingten Sprung (in Pascalnotation: `if cc then goto ...`) abgefragt wird. Ist der Condition Code wahr, so wird der Sprung, ansonsten der nächste Befehl durchgeführt. Komplexe Ausdrücke sind nicht erlaubt. Es gibt im wesentlichen nur zwei Arten von Wertzuweisungen:

1. links und rechts stehen nur Variable `a:= b`
2. auf der rechten Seite der Wertzuweisung steht ein Ausdruck mit nur einem Operator. Die Variable auf der linken Seite muß dann von spezieller Natur (Register r) sein und mit dem linken Operanden der rechten Seite übereinstimmen `r:=r+b`.

Unser Beispiel wird dadurch wesentlich komplizierter:

Assembler Programm

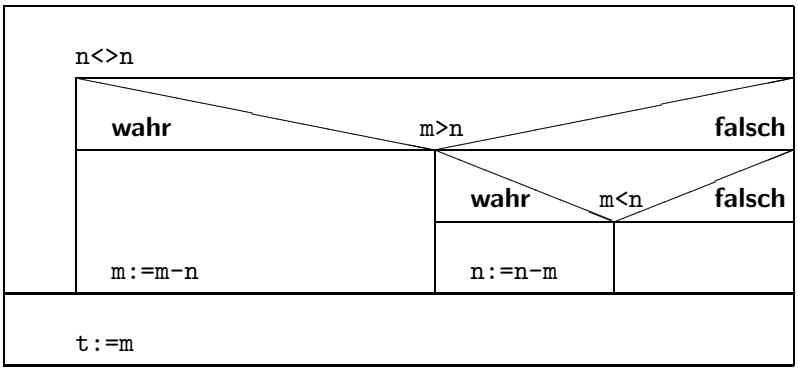
```
 1                r:=m;
 2      Anfang:   cc:=r=n;
 3                if cc then goto Ende;
 4                r:=m;
 5                cc:=r>n;
 6                if cc then goto Marke1;
 7                r:= m;
 8                cc:=r<n;
 9                if cc then goto Marke2;
10                ;
11                goto Anfang;
12      Marke1:   r:=n
13                r:=r-m;
14                goto Anfang
15      Marke2:   r:=m;
16                r:=r-n;
17                goto Anfang;
18      Ende:     t:=m;
```

Die Maschinelle Interpretation erfolgt genau so, wie oben, nur mit zusätzlichen Variablen.

Was ist aus unserem hübschen Algorithmus geworden? Natürlich könnte dieses Programm vereinfacht werden, aber zur Demonstration wurde es stur aus dem Beispiel zuvor übersetzt. *Übersetzt* bedeutet hier, daß das Programm der höheren Programmiersprache nach einem festen Verfahren in ein Programmm der maschinennahen Programmiersprache transformiert wurde. Natürlich ist der wichtigste Aspekt bei den Übersetzungen in der Informatik genauso wie bei den Übersetzungen der Dolmetscher, daß sich die Bedeutung (*Semantik*) des Programms bei der Übersetzung nicht ändert. Da die Übersetzung nach einem festen Verfahren durchgeführt wird, kann man dieses Verfahren auch einer Maschine überlassen. Das zugehörige Programm nennt man *Übersetzer* (oder *Compiler*).

Die angegebene Notation gibt es bei keinem Assembler, sie spiegelt jedoch die wesentlichen Sprachmerkmale jedes Assemblers wider. Die korrekte Notation z.B. für den Motorolarechner würde in diesem einführenden

Rahmen zu weit führen. Auffällig ist aber, daß diese Beschreibung durch die vielen Sprunganweisungen schwer verständlich ist. Daher haben sich graphischen Beschreibungen (*Struktogramme*) trotz des zusätzlichen Malaufwands bis heute gehalten:



Diskussion

Wiederum stellt sich die Frage, was ist der Vorteil des Abstiegs auf diese Ebene? Die Antwort ist einfach: Man kann auf dieser Ebene das Verfahren genau auf die Möglichkeiten der vorhandenen Hardware abstimmen. In unserem Programm könnte man zum Beispiel die Befehle 4, 7 und 15 weglassen, da der Wert von m ohnehin bereits in r geladen ist. Nur gute Übersetzer von höheren Programmiersprachen führen solche Optimierungen durch. Der andere Vorteil der maschinennahen Sprachen ist über die Angebote höherer Programmiersprachen hinauszugehen und z.B. den Anschluß anderer Geräte zu programmieren. Diesen Vorteilen stehen der wesentlich höhere und fehlerträchtige Programmieraufwand gegenüber. Daher wird die Programmierung im Assembler so weit wie möglich vermieden.

1.2.5 Schaltungen zur Problemlösung

Zum Schluß bleibt noch die Frage, wie ein Assemblerprogramm durchgeführt wird? Die fast noch lesbare Form des Assemblerprogramms wird in ein unlesbares Maschinenprogramm übersetzt (*Assemblierer*), das die gleichen Befehle wie das Assemblerprogramm hat, nur in binärcodierter Form. Dieses Programm wird dann durch Schaltungen (oder Mikroprogramme) ausgeführt. Die Schaltungen für jeden einzelnen Befehl sind im Rechner fest vorgeben, d.h. die Maschinensprache besteht genau aus den Befehlen, für deren Intrepretation eine Schaltung vorliegt.

Die Beschreibung von Schaltungen sind logische Formeln sehr einfacher Natur: Aussagenlogik. Die Schaltungen sind Thema der Vorlesung Rechnertechnologie und werden in diesem Rahmen nicht weiter behandelt.

1.2.6 Zusammenhang der Ebenen

Wahrscheinlich entstand bei der Diskussion der verschiedenen Sprachebenen der Eindruck, daß je tiefer die Ebene gewählt wird, desto effizienter wird die Ausführung. Dieser Eindruck ist richtig. Aber die Wahl der Ebene ist entscheidend für den Programmieraufwand und damit auch für die Fehlerwahrscheinlichkeit. Was nützt ein effizientes Programm, das falsche Resultate liefert. Korrekturen falscher Programme können natürlich zusätzliche Fehler einführen. Je tiefer die Ebene, desto größer der Programmier- und Korrekturaufwand. Das geht soweit, daß große komplexe Systeme heute nicht mehr in maschinennaher Sprache geschrieben werden. Um aber die konträren Ziele geringer Programmieraufwand versus Effizienz doch zu vereinen, wird bei der Systementwicklung in der möglichst höchsten Ebene begonnen und das entsprechende Programm dann per Hand oder maschinell in die effizienteren Ebenen transformiert.

Der Ausgangspunkt einer Systementwicklung ist die Beschreibung, *was* man will. Gefordert wird eine Beschreibung der erwarteten Leistungen des geplanten Systems (*Anforderungsdefinition* oder *Requirements*). Die

Beschreibung, die üblicherweise natürlichsprachlich ist, muß die einzelnen Leistungen exakt beschreiben. Eine formale Beschreibung der Anforderung in deskriptiver Form garantiert die Exaktheit, da die Formeln eine eindeutige Leseweise garantieren. In unseren Fall ist durch die Regeln 1–3 aus 1.2.1.1 eine präzise Anforderungsdefinition gegeben.

Wir könnten nun unsere formalen Anforderungen mit minimalen Änderungen in eine logische Programmiersprache (Prolog) umschreiben, bzw. wir hätten unsere Anforderungen direkt in dieser Sprache formulieren können. Da die logischen Programmiersprachen ausführbar sind, wären wir mit der Systementwicklung (GGT) fertig! Es genügt die Problembeschreibung, eine Programmentwicklung ist nicht mehr notwendig, denn wir bekommen alle Antworten durch Anfragen wie z.B. GGT(12,15,t)? Leider lassen sich nicht alle Probleme in den logischen Programmiersprachen darstellen, und weiter ist auch die Effizienz dieser Programmiersprachen in der Anwendung bei weitem nicht ausreichend. Daher muß man überlegen, wie man die geforderten Ergebnisse erreichen kann. Hier beginnt die eigentliche Programmentwicklung als intuitiver Prozess, der sich nicht durch standardisierte Verfahren durchführen läßt.

Auch wenn es kein mechanisches Verfahren gibt und nie geben kann, wie in den "Grundzügen der Informatik IV" nachgewiesen wird, so gibt es trotzdem eine *Methodik* des Programmierens. Es gibt zwei Klassen der methodischen Programmentwicklung:

1. man führt eine systematische Überführung der deskriptiven Beschreibung in ein funktionales oder imperatives Programm (*Programmsynthese*) nach bereits *bewiesenen* Regeln durch, deren geschickte Auswahl die eigentliche Programmentwicklung bedeutet, oder

2. man hat einen Einfall für ein Verfahren, von dem dann *nachträglich* geprüft werden muß, ob es der Beschreibung des Systems entspricht (*Verifikation*).

In der Praxis liegt eine Mischform vor. Man führt die Transformation nach *bewährten* aber noch nicht bewiesenen Regeln durch. Ein intuitiver Einfall ist eine Erweiterung der Regelmenge. Da die Korrektheit der Regel fehlt muß eine Verifikation des Programms durchgeführt werden. Die bewährten Regeln sind leider nur ein individueller Erfahrungsschatz. Eine brauchbare Systematik liegt noch nicht vor. Ihre Aufgabe ist es, sich dieses Regelwissen anzueignen.

Hat man ein Verfahren (Algorithmus) gefunden, das der Anforderung entspricht, und hat man dieses Verfahren in einer imperativen oder funktionalen Sprache dargestellt (Programm), dann geht der Rest weitgehend automatisch. Das funktionale oder imperative Programm wird durch ein spezielles Programm (Übersetzer oder Compiler) in ein Maschinenprogramm übersetzt, das dann über die Schaltungen interpretiert wird.

Programmentwicklung ist ein zu einen Teil intuitiver und zum anderen Teil automatisierter Prozeß, Anforderungen, die informal oder in einer höheren Logik formal beschrieben sind, in Programme überzuführen, die dann über Schaltungen einer Maschine, also durch Interpretation der Aussagenlogik, durchgeführt werden. Programmentwicklung besteht somit aus einer Reihe von Übersetzungen von einer Sprachebene in die nächst "tiefere", ohne die *Bedeutung* bei der Übersetzung zu verändern. Es ist ein zentrales Forschungsziel der Informatik, die Übersetzungsschritte immer mehr zu mechanisieren, unter vollem Bewußtsein, daß dies nie vollständig möglich sein wird.

Wir können aber auch die andere Richtung betrachten, die die Geschichte der Informatik widerspiegelt: Die Aussagenlogik, realisiert über Schaltungen, erlaubt uns die Konstruktion von Maschinen (Schickard 1623: Rechenuhr), die die arithmetischen Operationen durchführen können. Die Idee, das Verfahren der Berechnung zu codieren und auch in den Rechner abzuspeichern, führt uns zum Rechner, der in der Maschinensprache *universal* programmierbar ist.[4] Aufgrund der wenigen Ausdrucksmöglichkeiten (mehr würde die Hardwarepreise in die Höhe treiben) der Maschinensprache ist das Programmieren aufwendig und daher fehlerträchtig.

[4]Falcon 1728: Webstuhl mit Lochplatten zur Steuerung des Musters, von Jacquard 1805 verbessert, Babbage: digitaler programmgesteuerter Rechner auf mechanischer Basis, Zuse 1934 und Aiken 1944 relaisgesteuerter Rechner, Eckert und Mauchley 1964 röhrengesteuerter Rechner, Zemanek erster transistorgesteuerter Rechner in Europa

Durch das Konzept des Übersetzers (Rutishauser 1951) kann man sich von dieser frustrierenden Ebene befreien und seine Algorithmen als imperative Programme (in Fortran, Pascal, Eiffel, Modula usw.) schreiben. Doch auch hier hat man noch mit Problemen wie der Speicherverwaltung zu kämpfen. Davon kann man sich durch den Übergang zu funktionalen Programmen, die dann wiederum in imperative Sprachen übersetzt werden, befreien. Fühlt man sich sogar von der Algorithmusentwicklung belästigt, so muß man auf weitere Ergebnisse der Forschung hoffen, die die Programmsynthese automatisiert. Dann beschränkt sich die Systementwicklung nur mehr auf die Beschreibung, *was* man will. Das *Wie*, also das Programm, wird dann automatisch generiert.

Diese Schichten der formalen Systeme machen einen Teil der Faszination der Informatik aus: Durch Übersetzung von einer oberen Schicht in die nächsttiefere kann man sich von Fesseln des unteren Systems befreien und in die höhere Ebene der formalen Systeme und damit auch des Denkens aufsteigen.

Komplexere Softwaresysteme haben generell eine Schichtenstruktur. Auf der Basis einfacher Systeme baut man komplexere Systeme auf, die wiederum Basis der nächsten Entwicklung sind. Das leitet uns zum zweiten Schwerpunkt der Grundzüge der Informatik über.

1.3 Methodisch-Technische Schwerpunkte

Auf der Basis formaler Systeme können Programme entwickelt werden. Die Anwendung formaler Systeme, ist daher die wichtigste Aufgabe der Grundausbildung. Programme können von guter oder schlechter Qualität sein. Daher ist bei komplexeren Programmen wichtig, Techniken der Systementwicklung einzusetzen, die zu Programmen hoher Qualität führen. Die Technik der Systementwicklung nennt man *Software Engineering*.

1.3.1 Programmieren: Anwendung formaler Systeme

Im Zentrum der Anwendung steht das Programmieren in problemorientierten Sprachen und in maschinennahen Sprachen. Der Schwerpunkt wird hierbei auf die Programmiersprache Eiffel [Meyer, 1988, Meyer, 1992] und auf den Assembler des Motorolarechners [Kammerer, 1993] gelegt. Nach diesen beiden Semestern sollten Sie in der Lage sein, durch Selbststudium die Kenntnisse anderer Programmierprachen zu erwerben.

Die Auswahl der Sprache Eiffel hat didaktische Gründe. Als Sprache zur Erklärung von Programmierkonzepten spielt Eiffel heute die gleiche Rolle wie Pascal vor etwa 20 Jahren. Eiffel ist die einfachste und konsequenteste Programmiersprache mit objektorientierten Konzepten (siehe Abschnitt 1.3.6). Die Erfahrungen der vorangegangenen Semester zeigen, daß Eiffel sich zur Einführung in objektorientierte Sprachen besonders gut eignet. In Umfragen wurde Eiffel gegenüber C++ bevorzugt. Pascal wurde abgelehnt. Bei guter Kenntnis von Eiffel ist es ein leichtes nach C++ zu wechseln, das sich trotz softwaretechnisch schlechter Grundkonzeption langsam zum Industriestandard entwickelt. Die in der Schule übliche Sprache Pascal hat in der Schule zur Entwicklung einfacher Algorithmen ihre Berechtigung, ist für größere Systeme aber in ihrer mittlerweile veralteten Konzeption ungeeignet. Die Verbesserungsvorschläge in Richtung Modularisierung und Objektorientierung setzen in Pascal und C auf den festen Standard auf und bieten daher kein harmonisches Gesamtkonzept an.

Die Auswahl des Assemblers ist durch die jeweiligen Arbeitsplatzrechner des FB20 gegeben. Derzeit sind die Rechner mit Motorola 680x0 ausgestattet.

Bevor man programmiert, sollte man wissen, was man programmieren will. Es muß also die Anforderungsdefinition vorhanden sein. Die Programmiersprache Eiffel bietet dafür die Idee eines *Kontraktes* an: Wenn der Benutzer meines Programms die von mir vorgeschriebenen Voraussetzungen garantiert, dann garantiert mein Programm dem Benutzer die beschriebenen Ergebnisse. Die Voraussetzungen und Ergebnisse sollten nach Möglichkeit formal deskriptiv beschrieben sein, damit sie eindeutig sind und auch formal geprüft werden können. Daher werden wir zuerst einen Streifzug durch die Logik (*Aussagen-* und *Prädikatenlogik* erster Stufe) machen, bevor wir in die Programmiersprache Eiffel und in ihre Anwendung einsteigen.

Um Programmieren zu können, ist natürlich die *exakte* Kenntnis des formalen Systems, also der Programmiersprache notwendig. Manchen von Ihnen sind z.B. Syntaxdiagramme aus der Schule bekannt. Von der syntaktischen Strukturbeschreibung werden wir ausgehen und die Wirkung (*Semantik*) der einzelnen Komponenten der Programmiersprache ebenso exakt beschreiben. Dafür werden wir eine funktionale Sprache verwenden. Da wir hier eine Sprache durch eine andere erklären, haben wir einen Ebenenwechsel. Die Sprache, die eine andere Sprache (bei uns Eiffel) beschreibt, wird daher *Metasprache* genannt. (Vorwarnung: Die beiden Sprachen verwenden zum Teil das gleiche Vokabular, sind aber durch die Art des Drucks stets unterscheidbar.)

Obwohl die Sprache Eiffel auch viele Konzepte beinhaltet, die bereits in Pascal, Basic und noch einfacheren Sprachen vorhanden sind, gibt es nur wenige Gründe, die Entwicklung der exakten Beschreibung einer Programmiersprache im Rahmen einer Vorlesung der historischen Entwicklung der Programmiersprachen anzugleichen. Wir beginnen daher (erstmalig in diesem Semester) mit Klassen und Objekten und dem modularen Aufbau von Systemen auf der Basis bereits existierender Module. Wir ergänzen dies dann durch konventionellere Konzepte wie Prozeduren und algorithmische Strukturen und gehen schließlich ein wenig auf die Auswertung von Formeln und Ausdrücken sowie andere Details der Sprache ein, die für den eigentlichen Entwurf von geringer Bedeutung und eigentlich nur für das konkrete Ausprogrammieren von Details wichtig sind (, was in der Vorlesung im Hintergrund stehen soll). Die Strukturierung von Klassen über das Vererbungskonzept bildet dann den krönenden Abschluß dieser Vorlesung

Exakte Kenntnis einer Programmiersprache ist notwendig, aber in keiner Weise hinreichend für die Programmentwicklung. Notwendig ist die *Fähigkeit* des Programmierens. Leider gibt es kein allgemeines Verfahren, wie man programmiert. Daher kann das Programmieren auch nicht vermittelt werden. Die Programmierfähigkeit muß erworben werden! Analog zum Lernen einer Fremdsprache reicht Zuhören nicht aus. Sie müssen sprechen und andere müssen zuhören, um Fehler zu korrigieren. Sprachunterricht bietet Vokabel, Grammatik, Redewendungen und erfordert Training, Training, Training. Analog ist beim Erwerb der Programmierfähigkeit das Programmieren (Sprechen) unbedingt notwendig, aber wiederum nicht hinreichend! Damit Ihre Fähigkeit verbessert wird, müssen Sie Ihre Programme einer konstruktiven Kritik aussetzen: *Code Review*.

Es wird immer wieder gefordert, es soll doch gezeigt werden, wie man programmiert. Das ist aus theoretischen Gründen grundsätzlich unmöglich! Der Wunsch nach einem festen Verfahren, das immer wieder angewendet werden kann, ist aber auch eine unsinnige Forderung. Angenommen es gäbe ein Verfahren, wie Programme aus einer Problemstellung heraus entwickelt werden können, dann wäre dieses Programmsyntheseverfahren schon längst ein Programm und Programmieren somit wegrationalisiert (Ein Traum der KI!). Zum Erlernen der Programmierfähigkeit können wir also nur an Beispielen üben - kritisieren, üben - kritisieren, üben - kritisieren usw. Jeder Schritt in disem Zyklus erweitert Ihr individuelles Regelsystem!

Um Frustrationen auszuschließen: Sind Sie bei der Entwicklung eines Programms über den Aufwand verzweifelt, so können Sie sich trösten: Durch die neuen Erfahrungen wird die nächste Entwicklung sicher leichter! Grob formuliert: Ihr Aufwand für eine Neuentwicklung ähnlichen Schwierigkeitsgrades ist etwa reziprok zu der Anzahl Ihrer bisher erfolgreich abgeschlossenen Entwicklungen. Wenn Sie sich die Kurve y=k/x ins Gedächtnis rufen, dann sind nur die Werte x<k groß. Also wenn Sie über Ihre persönliche Anzahl k von erfolgreichen Entwicklungen hinauskommen, sind die Probleme vorbei. Wir helfen Ihnen dabei durch Beispiele im Skript, durch Aufgabenstellungen in den Übungen und im Paktikum.

Zur Kritik gehört natürlich auch ein Qualitätsbegriff, dessen Verletzung Anlaß zur Kritik ist. In den "Grundzügen der Informatik I bis III" stehen die Qualitätskriterien für Strukturierung, Zuverlässigkeit und Leistung des Programms im Vordergrund.

1.3.2 Strukturierungskonzepte

Die heutigen Softwaresysteme haben eine Komplexität, die es ausschließt, daß eine Person allein die vollständige Übersicht über das System in allen Details haben kann. Die Konsequenz ist "Seperation of Concerns" (Dijkstra). Diese Aufteilung der "Sorgen" kann auf zwei Arten erfolgen: Zeitlich durch eine Entwicklung in Phasen

und "örtlich" durch Zerlegung des Systems in Komponenten, die dann weitgehend unabhängig voneinander entwickelt und geprüft werden können.

1. **Phasenkonzept der Systementwicklung**

   Komplexere Softwaresysteme können nur in *Versionen* entwickelt werden. Erstens ist es unwahrscheinlich, daß vor dem Einsatz bereits alle Anforderungen dem Auftraggeber bewußt sind, und zweitens verändert der Einsatz des Systems die alten Abläufe in einem Betrieb derart, daß neue Wünsche und damit neue Anforderung an das ausgelieferte System entstehen. An eine Softwarefirma, die für eine Bibliothek ein Verwaltungsprogramm geschrieben hat, kann zum Beispiel der Wunsch herangetragen werden, dieses auf die Verwaltung von CD's zu erweitern. Sie wird versuchen, unter Wiederverwendung bereits entwickelter Programmteile ein Programm zu entwickeln, was beides gleichzeitig verwalten kann und Vorsorge für den weiteren Ausbau treffen. Dabei wird sie dann feststellen, daß sich das System grundsätzlich auch für andere hierarchischen Verwaltungen eignet, und es zu einem allgemein verwendbaren Modul generalisieren, um einen größeren Kundenkreis für ihr Produkt zu gewinnen. Diese Art der Software Entwickling nennt man *evolutionär*. Evolutionäre Entwicklung hat es stets gegeben, aber erst durch die objektorientierte Konstruktion von Systemen kann sie mit minimalen Änderungen der alten Version durchgeführt werden.

   Der Ausgangspunkt jeder Entwicklung einer Version ist eine eher vage Beschreibung eines neuen Systems oder neuer Wünsche an ein bestehendes System. Es folgt eine Analyse der Marktsituation, ob es ähnliche Angebote bereits gibt. Ist diese negativ, so erfolgt eine Präzisierung der Wünsche, um ein konkretes und präzis definiertes Entwicklungsziel zu haben. Diese Phase wird *Systemanalyse* genannt. Das Ergebnis ist ein *Pflichtenheft*, das das äußere Verhalten des geplanten Systems in Form der Anforderungsdefintion als Kontrakt genau beschreibt.

   Damit kann die anspruchvollste Phase beginnen: Der *Entwurf*. Hier wird die Grobstruktur des Systems festgelegt, die eine Zerlegung in getrennt zu entwickelnden Komponenten erlaubt. Die Realisierung der einzelnen Komponenten in einer Programmiersprache nennt man *Implementierung*. Sind die Komponenten fertig und im *Komponententest* geprüft, so werden sie zum vollständigen System integriert und in einem *Abnahmeverfahren* an den Benutzer übergeben. Gleichzeitig beginnt schon das Sammeln der zunächst noch vagen Vorschläge für die nächste Version. Diese Abfolge von Entwicklungsphasen nennt man *Life Cycle*. Wir werden in unseren Beispielen diese Phasen nur andeuten. Vertiefung dieser Thematik bietet die Lehrveranstaltung *Software Engineering* im Hauptstudium.

2. **Zerlegungskonzepte für Systeme**

   Bei den Zerlegungskonzepten geht es stets um "Separation of Concerns": Wie reduziert man ein komplexes Problem in eine Menge von Problemen geringere Komplexität? Diese Frage wollen wir in den folgenden Abschnitten genauer besprechen.

1.3.3 Schrittweise Verfeinerung

Die einfachste und häufigste Form der Strukturierung ist die Zerlegung eines Verfahrens in seine Teilschritte, die eventuell wieder in weitere Teilschritte zerlegt werden, wenn sie noch immer zu komplex sind. Ein Beispiel ist die "Burger"-Herstellung, die einem sehr genau festgelegten Verfahren folgt:

Erste Verfeinerung der Herstellung:

```
Burger-Herstellung:    Brötchen vorbereiten;
                       Brötchen füllen;
                       Brötchen verpacken
```

Zweite Verfeinerung der Herstellung:

```
Brötchen vorbereiten:  Brötchen nehmen;
                       Brötchen halbieren;
```

```
Brötchen füllen:      heißes Hacksteack auf die untere Hälfte legen;
                      Salatblatt darauflegen;
                      Tomatenscheiben darauflegen;
                      Essiggurkenscheiben darauflegen;;
                      alles unter KetchUp verstecken;
                      obere Brötchenhälfte daraufpappen
```

Diese Verfeinerung des Verfahrens wird solange fortgesetzt, bis nur mehr Verfahrensschritte vorliegen, die so elementar sind, daß sie von dem Durchführenden (Mensch oder Maschine) unmittelbar ausführbar sind.

Als Verfeinerung haben wir nur die Aufeinanderfolge von Tätigkeiten benutzt. Da die Burgerspeisekarte eine beschränkte Anzahl von Alternativen bietet, können wir das Verfahren erweitern:

```
Brötchen vorbereiten;
if Doppel-Burger verlangt
   then heißes Hacksteack hineinlegen;
        Salatblatt darauflegen;
        Tomatenscheiben darauflegen;
        Essiggurkenscheiben darauflegen;
        heißes Hacksteack hineinlegen;
        Salatblatt darauflegen;
        Tomatenscheiben darauflegen;
        Essiggurkenscheiben darauflegen;;
   else heißes Hacksteack hineinlegen;
        Salatblatt darauflegen;
        Tomatenscheiben darauflegen;
        Essiggurkenscheiben darauflegen;;
alles unter KetchUp verstecken;
obere Brötchenhälfte daraufpappen;
Brötchen verpacken
```

Wir haben nun neue Sprachelemente "if ... then ... else ..." hinzugenommen für die Verfeinerung hinzugenommen, deren Bedeutung natürlich genau definiert sein muß. Z.B. schließen sich then-Teil und else-Teil in der Durchführung aus, die Einrückung gibt wieder, welche Schritte zum then-Teil und zum else-Teil gehören.

Wir können unsere Sprache (Programmiersprache) um die Möglichkeit der Wiederholung erweitern, um die Aufgabe der Küche zu beschreiben, den Vorrat an Burgern für die Kasse immer wieder aufzufüllen.

```
while im Fach Platz ist do
      Brötchen nehmen;
      Brötchen halbieren;
      if Doppel-Burger verlangt
         then heißes Hacksteack hineinlegen;
              Salatblatt darauflegen;
              Tomatenscheiben darauflegen;
              Essiggurkenscheiben darauflegen;
              heißes Hacksteack darauflegen;
              Salatblatt darauflegen;
              Tomatenscheiben darauflegen;
              Essiggurkenscheiben darauflegen;
         else heißes Hacksteack hineinlegen;
              Salatblatt darauflegen;
              Tomatenscheiben darauflegen;
              Essiggurkenscheiben darauflegen;
      alles unter KetchUp verstecken;
      obere Brötchenhälfte daraufpappen;
      Brötchen verpacken
```

Auch hier muß das neue Sprachelement while ... do eindeutig definiert sein: Solange die Bedingung wahr ist, soll der eingerückte Teil durchgeführt werden.

1.3.4 Prozeduralisierung

Das Beispiel zeigt, daß die Strukturierung in Elementarschritten langweilig werden kann, insbesondere, wenn es dreifach, vierfach und weitere Burger gibt. Dann ist es sinnvoll, Teilverfahren einen Namen zu geben und diesen dann in der eigentlichen Verfahrensbeschreibung zu verwenden.

Namen geben:

```
Schicht einlegen: heißes Hacksteack darauflegen;
                  Salatblatt darauflegen;
                  Tomatenscheiben darauflegen;
                  Essiggurkenscheiben darauflegen;
```

Namen verwenden:

```
                  while im Fach Platz ist do
                        Brötchen nehmen;
                        Brötchen halbieren;
                        if Vierfach-Burger verlangt
                           then Schicht einlegen;
                                Schicht einlegen;
                                Schicht einlegen;
                                Schicht einlegen
                           else if Dreifach-Burger verlangt
                                   then Schicht einlegen;
                                        Schicht einlegen;
                                        Schicht einlegen;
                                   else if Doppel-Burger verlangt
                                           then Schicht einlegen;
                                                Schicht einlegen;
                                           else Schicht einlegen;
                        alles unter KetchUp verstecken;
                        obere Brötchenhälfte daraufpappen;
                        Brötchen verpacken
```

Die Möglichkeit, Teilprogrammen einen Namen zu geben und diesen dann zu verwenden, nennt man *Prozeduralisierung*. Das benannte Teilverfahren nennt man *Prozedur* und die Verwendung des Namens *Prozeduraufruf*. Das ist etwa die Ausdrucksmöglichkeit, die gerade noch durch Basic erreicht werden kann. Der Effekt ist hier erst eine Halbierung des Beschreibungsaufwands. Besser wird es, wenn wir mehr in die Prozedur stecken. Wir können sie noch informieren, wie groß die Anzahl i der Schichten werden soll:

Namen geben:

```
Schicht(i): while i>0 do
                  Schicht einlegen;
                  Zähle i um 1 herunter
```

Namen verwenden:

```
              while im Fach Platz ist do
                  Brötchen nehmen;
                  Brötchen halbieren;
                  if Vierfach-Burger verlangt
                     then Schicht (4);
                     else if Dreifach-Burger verlangt
                             then Schicht (3);
                             else if Doppel-Burger verlangt
                                     then Schicht (2);
                                     else Schicht (1);
                  alles unter KetchUp verstecken;
                  obere Brötchenhälfte daraufpappen;
                  Brötchen verpacken
```

Wir stellen der Prozedur beim Aufruf noch Information zur Verfügung, die ihren Ablauf steuert. Solche aufrufsabhängigen Angaben nennt man *Parameter*.

Wir könnten natürlich auch die Prozedur anders definieren:

```
Schicht(i): if i>1
            then Schicht einlegen;
                 Schicht (i-1)
            else Schicht einlegen
```

Hier wird die Prozedur noch einmal aufgerufen, bevor sie fertig ist. Man nennt diese Prozedur deshalb *rekursiv*. Mit den Parametern und der Rekursion haben wir die Strukturierungsmöglichkeiten von Pascal erreicht.

Mit diesen Möglichkeiten der Ablaufstrukturierung und den noch zusätzlichen Möglichkeiten der Datenstrukturierung kann man alle Verfahren beschreiben, die sich überhaupt beschreiben lassen. Die Untersuchung, welche Verfahren sich algorithmisch beschreiben lassen, ist ein Thema der theoretischen Informatik.

1.3.5 Modularisierung

Es zeigt sich nun in der Praxis, daß man weitere Strukturierungsmöglichkeiten braucht, um einen *zuverlässigen* Betrieb zu gewährleisten. Stellen Sie sich vor, der Koch darf in die Kasse greifen, obwohl nur der Kassierer für die Kasse verantwortlich ist, und der Kassierer greift ins heisse Öl, da er nicht als Koch ausgebildet ist. In unseren Burger-Betrieb hat jeder seine Aufgaben und kein anderer kann in seinen Bereich eingreifen. Daraus ergibt sich, daß der Betrieb eine Sammlung von Teilbetrieben (Küche, Kasse, Buchhaltung) ist, die sich gegenseitig Leistungen anbieten, aber jeder in seinem Bereich strikt autonom ist. Es gibt nun zwei extreme Ansätze der Kommunikation und des Materialflusses dieser Teilbetriebe:

- *Zentral:* Jeder Teilbetrieb empfängt Anweisungen und Materialien vom Chef und liefert alles an den Chef. Der Chef ordert einen einfachen Burger und stellt dafür ein Brötchen, ein tiefgekühltes Hacksteak, eine Portion Salatblätter, Tomaten, Essiggurken und KetchUp zur Verfügung. Der Koch gibt den fertigen Burger an den Chef zurück, der ihn ins Fach stellt. Eine solche zentrale Organisation der Kommunikation und des Materialflusses nennt man *Top-Down*.

- *Dezentral:* Jeder Teilbetrieb macht und verwaltet das, was er eben kann. Die Kommunikation erfolgt direkt zwischen den Teilbetrieben. Die Kasse ordert bei der Küche einen Burger. Die Küche verlangt vom Lager ein Brötchen, ein tiefgekühltes Hacksteak, eine Portion Salatblätter, Tomaten, Essiggurken und KetchUp. Das Lager liefert das Gewünschte, Die Küche macht den Burger fertig und liefert ihn an die Kasse. Ein Chef in obiger Form ist nicht notwendig. Jeder Burgerbetrieb ist nach dieser Form organisiert. Man nennt diese Organisationsform der Komunikation und des Materialflusses *objektbasiert*.

Der Unsinn einer zentralistischen Organisation ist im obigen Beispiel evident. Trotzdem wurde bis etwa 1980 nach diesem Konzept programmiert(, was aber nicht nur ein Fehler der Informatik war). Das *Subsidiaritätsprinzip*, daß jede Einheit das leistet, was sie selbst leisten und verantworten kann, und sich nicht entmündigen läßt, hat in der Informatik die Namen *Geheimnisprinzip* oder *Datenkapselung*, Dies drückt aber nur aus: "den Koch geht die Art, wie die Kasse arbeitet, nichts an und umgekehrt". Die schärfere Forderung "die Kasse macht alles, was eine Kasse eben zu tun hat" muß man sich stets dazu denken. In der Informatik heißen die Teilbetriebe *Moduln* oder *Units*. Modula, Ada und Turbo Pascal 5.0 unterstützen das objektbasierte Konzept.

1.3.6 Klassifizierung

Bis jetzt haben wir nur über eine Küche und eine Kasse gesprochen. Das ist aber unökonomisch, da eine Küche mehr produziert, als eine Kasse verkaufen kann. Daher müssen wir mehrere Kassen einrichten, die unabhängig voneinander mit der Küche zusammenarbeiten. D.h. wir haben zwar nur *eine* Beschreibung der Kasse, aber *mehrere* Kassen, die nach dieser Beschreibung arbeiten. Die Beschreibung nennen wir *Klasse*. Die einzelnen Kassen sind *Objekte* dieser Klasse, die sich genau nach dem vorgegeben Konzept der Klasse Kasse richten. Das Klassenkonzept wird von Simula67, Smalltalk, Eiffel und von Turbo Pascal 5.5 unterstützt.

Man könnte das Objektkonzept von der Modularisierung dadurch unterscheiden, daß bei der Modularisierung die Idee und die Realisierung eins sind. Es gibt ein Konzept und eben nur einen Betrieb nach diesem Konzept. Bei den Klassen gibt es eine Idee und viele Realisierungen dieser Idee. In der Wirtschaft nennt man diese Betriebsform *Franchising*. Die Klasse ist das Konzept, wie Burger-Betriebe zu betreiben sind.[5] Jeder Betrieb ist ein selbständiges Unternehmen, das aber genau nach dem Konzept betrieben wird.

1.3.6.1 Generizität

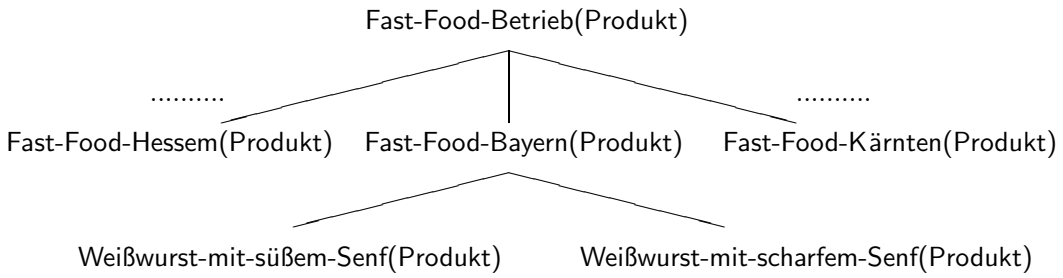
Bei der Verbreitung der Betriebe nach dem Burger-Konzept wirkt die Starrheit des Konzepts geschäftsbehindernd. Z.B. stellt es sich heraus, daß der Unterschied zwischen Burger-Betrieb und Backhähnchen-Betrieb nur in dem verkauften Produkt besteht. Organisation, Küche, Kasse usw. sind weitgehend identisch. Nichts liegt näher als sich das generelle Konzept solcher Betriebe schützen zu lassen. Erst bei der Neugründung eines Betriebs wird noch vorher festgelegt, welcher Produkttyp tatsächlich von diesem Betrieb angeboten wird. Das bedeutet, das Betriebskonzept, nennen wir es Fast-Food-Betrieb, läßt offen welcher Art das Produkt ist. Im Fast-food-Betrieb werden nur gewisse Einschränkungen für die möglichen Produkte definiert, damit z.B. die Standardküche, die nach dem allgemeinen Konzept eingerichtet ist, das Produkt auch herstellen kann.

In der Informatik nennt man diese offengelassene Information *Typparameter* (in C++ Template). Eine Klasse mit Typparameter bezeichnet man als *generische* Klasse. Wird ein Objekt einer Klasse mit einem Typparameter aufgebaut, so muß man natürlich vorher den aktuellen Typ bekannt geben. Der Typparameter verwandelt eine Klasse eigentlich in eine Klasse von Klassen. Dadurch bekommen wir eine mächtige Klassifizierungmöglichkeit: Eine Klasse ist eine spezielle Ausprägung der generischen Klasse, in welcher der Typparameter einen festen Wert hat.

1.3.6.2 Vererbung

Ein anderes Hemmnis ist die Starrheit des Angebots: Entweder nur Burger oder Backhähnchen. In Hessen ist der zusätzliche Kundenwunsch nach "Handkäs mit Musik", in Bayern nach Weißwurst, im Rheinland nach Reibekuchen mit Apfelmus, im Ruhrgebiet nach Currywurst, an der Nordsee nach Krabbenbrötchen und in Kärnten nach Kletzenudeln unüberhörbar. Nun ist jedoch das generische Konzept handelsrechtlich geschützt und jede Änderung von hohen Kosten begleitet. Außerdem ist eine generelle Änderung überhaupt nicht erwünscht. Handkäs mit Musik bedeutet in Bayern eine Zumutung für die Urbevölkerung! Daher sind *lokale Ergänzungen* des bestehenden Konzepts notwendig. Nicht das gesamte Konzept wird geändert, sondern in den verschiedenen Essregionen werden lokale Zusatzleistungen angeboten. Das Konzept der lokalen Zusatzangebote wiederholt nicht das ganze Konzept, es beruft sich nur darauf. Ein Betrieb wird also nach dem lokalen Konzept aufgebaut und "erbt" durch den Verweis auf das Grundkonzept auch dessen Konzept.

Der Mechanismus, der diese Spezialisierung oder besser Ergänzung ermöglicht, heißt in der Informatik *Vererbung* oder *Inheritance*. Es kann dabei durchaus vorkommen, daß man von verschiedenen Konzepten erbt (*Mehrfach Vererbung*): Burger-Konzept und Biergarten-Konzept. Die Vererbung erlaubt uns nun eine Klassifizierung der Klassen in Art von Stammbäumen gemäß ihres Angebots:



[5] In der Mathematik spricht man statt von Klassen eher von abstrakten Datentypen oder *algebraischen Theorien*. Generische Klassen sind *parametrisierte Theorien* und das Vererbungskonzept entspricht der *Theorieerweiterung*.

Bauen wir nun einen Betrieb nach dem Süßer-Senf(Burger)-Konzept auf, so wird dort Weißwurst mit süßen Senf zusätzlich zum normalen Burger angeboten, und nach dem Scharfer-Senf(Backhuhn)-Konzept würde Weißwürste mit scharfem Senf zusätzlich zu Backhühnern angeboten.

Erst durch das Konzept der Vererbung läßt sich die am Anfang angesprochene evolutionäre Entwicklung effizient realisieren. Die Sprachen, die Klassifizierung über Vererbung und Typparameter zulassen, sind Eiffel und C++. In den anderen werden gerade solche Erweiterungen eingebaut.

1.3.7 Zusammenfassung

Die obige Reihenfolge der Strukturierungskonzepte gibt ihre geschichtliche Entwicklung wieder. Die Strukturierung, wie sie bei der Systementwicklung durchgeführt wird, erfolgt genau umgekehrt.[6] Zuerst wird die Frage nach der Strukturierung der Daten gestellt, das System grob in Klassen zerlegt und die Beziehung der Klassen (Vererbung und Benutzung) zueinander festgelegt. Anschließend werden für jede einzelne Klasse ihre Leistungen festgelegt und jede Leistung in Form eines Kontraktes beschrieben. Ein Kontrakt ist das Versprechen des Programmierers, daß, wenn die geforderte Eingangsbedingung (*Precondition*) erfüllt ist, er garantiert, daß die Ausgangsbedingung (*Postcondition*) ebenso erfüllt ist.

Damit ist dann die Entwurfsphase beendet, und es beginnt die Implementierungsphase, in der jede Leistung einer Klasse entsprechend dem Kontrakt programmiert wird. Die Programmierung benutzt dann die schrittweise Verfeinerung im Falle, daß die Implementierungsprache eine imperative oder maschinennahe Sprache ist. Man könnte natürlich auch die einzelnen Leistungen in einer logischen oder funktionalen Sprache realisieren, unter der Beachtung der Verfeinerungkonzepte, die diese Sprachen bieten.

Die Fähigkeit zum Umgang mit formalen Systemen ist in allen diesen Phasen *unumgänglich*. Aus diesem Grunde werden wir zunächst die wichtigsten Teile der Logik und formaler Sprachbeschreibungen besprechen müssen, bevor wir die Konzepte von Programmiersprachen genauer vorstellen.

[6] Aus diesem Grunde folgen wir in dieser Vorlesung auch nicht der geschichtlichen Entwicklung sondern steigen “von oben” in die Programmierkonzepte ein.

Kapitel 2

Logik und formale Sprachbeschreibungen

(Marion Kremer)

Eine der wesentlichen Tätigkeiten, die mit Informatik verbunden sind, ist die Modellierung von Ausschnitten der realen Welt, d.h. die Repräsentation reduzierter Wirklichkeiten. Ziel dieser *Modellbildung* ist es, sowohl die zwischenmenschliche Kommunikation über die modellierten Objekte und Tätigkeiten zu ermöglichen, als auch Arbeitsanweisungen an eine Maschine zu formulieren. Letztlich geschieht die Modellbildung mit der Zielsetzung, die Lösung von Aufgaben mit Hilfe eines Rechners zu ermöglichen. Daher besteht die gesamte Entwicklung eines Software-Produktes daraus, Beschreibungen der Modelle der Tätigkeiten zu konstruieren, die von einem Rechner ausgeführt werden sollen.

Verbunden damit ergibt sich die Notwendigkeit, die Objekte, die manipuliert werden sollen, zu modellieren. Ein Eintrag in einer Datenbank ist nur das Modell der Person, die eigentlich verwaltet werden soll. In diesem Sinne ist jedes auf einer realen Maschine ablaufende Programm ein Modell. Für die Beschreibung des Modells benötigen wir zumindest eine *Sprache, die möglichst von allen Kommunikationspartnern*[1] *auf die gleiche Art verstanden wird*. Warum kann hierfür nicht die natürliche Sprache verwendet werden?

Nehmen wir an, unser Nachbar arbeitet bei der Post und ist dort für die Erstellung von Telefonbüchern zuständig. Daher möchte er wissen, wie er eine Menge von Namen sortieren kann. Um ihm dies zu erklären, müssen wir zunächst festlegen, was sortiert sein bedeutet. Na ganz einfach, sagen wir, jedes Element muß kleiner sein als sein Nachfolger. Er schüttelt den Kopf. Was ist ein Element, was bedeutet kleiner und was ist ein Nachfolger? Dieses Spiel läßt sich ziemlich lange fortsetzen und macht deutlich, daß in der natürlichen Sprache häufig Wörter und Formulierungen verwendet werden, denen keine präzise allgemeingültige Bedeutung zugewiesen werden kann. Es gibt reichlich Fälle, bei denen die Bedeutung eines Wortes von dem Kontext, in dem es verwendet wird, abhängt. Ein Beispiel hierfür ist das Wort Bank. Darüberhinaus gibt es auch Wörter, die in ein und dem selben Satz unterschiedlich interpretiert werden können. Was bedeutet beispielsweise

> Es regnet oder die Sonne scheint

Kann die Sonne nur dann scheinen, wenn es nicht regnet? Oder

> Es regnet und die Sonne scheint

Soll das heißen, daß die Sonne immer dann scheint, wenn es regnet? Entsprechendes gilt auch für Folgerungen:

> Wenn das Benzin ausgeht, so bleibt das Auto stehen.
> Das Benzin geht nicht aus, also bleibt das Auto auch nicht stehen

Diese Folgerung ist richtig, wenn mit *Wenn* 'Nur wenn' gemeint ist, sonst ist sie falsch. Um die Sache weiter zu komplizieren, gibt es Aussagen, die weder als richtig noch als falsch bewertet werden können, wie etwa

> Dieser Satz ist falsch.

[1] An dieser Stelle soll darauf hingewiesen werden, daß zwischenmenschliche Kommunikation zumindest zeitweilig Komponenten beinhaltet, die bei der Kommunikation mit einer Maschine nicht vorhanden sind.

Unter der Annahme obige Aussage sei richtig, erhalten wir einen Widerspruch, da die Aussage ihre eigene Falschheit postuliert. Nehmen wir dagegen an, daß die Aussage falsch sei, dann wäre obiger Satz richtig und wir hätten wiederum einen Widerspruch. Der Grund für dieses Problem liegt in der Selbstreferenz der Aussage.

Um in unserem Kontext (der Erstellung maschinenlesbarer Aufgabenbeschreibungen) eine Aufgabenstellung diskutieren zu können, benötigen wir also eine Beschreibungsform, die es sowohl ermöglicht, zu entscheiden, ob eine Aussage eine Interpretation hat und falls sie eine hat, welche dies ist. Dazu müssen wir in der Lage sein, einem Satz eine *eindeutige Bedeutung* zu geben und zu entscheiden, wie die Zusammensetzung mehrerer Sätze zu interpretieren ist. Ein weiterer Grund, warum wir eindeutig interpretierbare Sprachen benötigen, besteht darin, daß wir *Arbeitsanweisungen an eine Maschine* geben wollen. Diese ist jedoch nur in der Lage, eindeutige Aufträge zu verarbeiten. Computerlinguisten versuchen schon ziemlich lange, einer Maschine beizubringen, das Verstehen natürlicher Sprache zu simulieren; bislang mit mäßigem Erfolg.

Das bisher Gesagte soll nun aber nicht heißen, daß es keine Notwendigkeit für die Mehrdeutigkeiten und Inkonsistenzen der natürlichen Sprache gibt; ganz im Gegenteil. Sie ist nur für den hier angestrebten Zweck ungeeignet. Natürliche Sprache ist ein Spiegel u.a. ihrer eigenen Entstehungsgeschichte und des sozialen, kulturellen, zeitlichen, historischen und örtlichen Kontextes der Kommunikationspartner.[2] Mithin ist es durch sie auch möglich, Dinge abhängig von diesem Kontext zu diskutieren.

Was ist nun eine geeignete Sprache zur Beschreibung von Aufgabenstellungen? Die Forderung nach eindeutiger Interpretierbarkeit ist unzureichend, da in der Menge dieser Sprachen beliebig viele enthalten sind, bei denen wir auf Fragen, die für uns relevant sind, keine Antwort erhalten können. Unsere Konsequenz besteht darin, daß wir zur Beschreibung der Eigenschaften eines Programmes die *formale Logik*, speziell die in der Mathematik etablierte *Prädikatenlogik*, verwenden. Von dieser weiß man, daß sie mächtig genug ist, um die uns interessierenden Fragen zu behandeln und eine erkleckliche Anzahl Antworten zu ermöglichen. Kurz gesagt, die *Prädikatenlogik genügt den von uns gestellten Anforderungen an eine Sprache am besten.*

In diesem Kapitel werden zunächst die Grundlagen eingeführt, die wir benötigen, um Sprache zu beschreiben. Der Formalismus zur Beschreibung von Sprache überhaupt wird u.a. am Beispiel der Ausagenlogik eingeführt. Diese ist hinreichend einfach und sollte dem Inhalte nach bekannt sein, wenn auch nicht in dieser formalen Form. Die hier gewählte Darstellung der formalen Sprachbeschreibungen ist weder erschöpfend noch vollständig formal. Die eingeführten Begriffe bilden jedoch die Grundlage für die gesamte Veranstaltung.

Im Anschluß an die Darstellung formaler Sprachbeschreibungen werden wir auf die Frage eingehen, wie man von einer umgangssprachlichen Beschreibung zu einer in einer formalen Sprache gelangt. Danach werden zwei Beispiele für Spezifikationssprachen eingeführt: zum einen die bereits erwähnte Prädikatenlogik, zum anderen eine dreiwertige Logik, um für Spezifikationen und Programmbeweise eine bessere Handhabe zu haben. An dieser Stelle werden wir weitere, auch für die Aussagenlogik relevante Begriffe wie 'Tautologie' einführen. Die Logik wird dabei eingeführt, um ein Beschreibungsmittel für Eigenschaften und Aufgaben von Programmen zu haben und deren Erfüllung überprüfen zu können. Themen wie Vollständigkeit und Widerspruchsfreiheit von Ableitungssystemen, für die sich die mathematische Logik interessiert, werden daher nur am Rande gestreift. Es empfiehlt sich für Informatiker, bei anderer Gelegenheit das Thema Logik zu vertiefen.

Wir haben uns bemüht, diese Beschreibung der Grundlagen so zu gestalten, daß möglichst wenig auf Kenntnisse Bezug genommen wird, die erst in einem späteren Teil eingeführt werden. Stellenweise ist dies jedoch nicht möglich gewesen. Es empfiehlt sich daher, den Teil bis zum Kapitel Syntax nach dem Durcharbeiten des gesamten Kapitels 2 nochmals zu wiederholen. Zur Darstellung muß noch erwähnt werden, daß üblicherweise erst die formalen Definitionen angegeben werden, und im Anschluß daran die Erläuterungen folgen.

Eine Liste weiterführender Literatur findet sich am Ende dieses Kapitels. Es wird *nicht* erwartet, daß Sie diese Bücher durcharbeiten. Die dort gemachten Angaben sollen Sie aber in die Lage versetzen, bei Interesse den einen oder anderen Punkt zu vertiefen.

[2] Aus Gründen der Lesbarkeit wird auf Formen wie KommunikationspartnerInnen verzichtet. Frauen sind jedoch explizit immer mitgemeint. Wer die mangelnde Lesbarkeit solcher Ausdrücke anzweifelt, möge sich klarmachen, daß mit der Wahl der dualen Form alle Pronomina, Artikel usw. entsprechend anzupassen sind.

2.1 Formale Sprachbeschreibungen

Die Beschreibung einer Sprache gliedert sich in mehrere Teilbereiche. Diese werden in der *Semiotik*, der Lehre der Zeichen und ihrer Bedeutung zusammengefaßt.

Die *Syntax* beschreibt die Menge der formal richtig gebildeten ("syntaktisch korrekten") Sätze. Sie beschreibt ausschließlich die äußere Form, bzw. Struktur der Sätze einer Sprache. Sie ermöglicht es, zu beschreiben, daß Peter ißt einen Apfel ein korrekt geformter Satz der deutschen Sprache ist, Peter einen Apfel aber nicht.

Die *Semantik* ordnet diesen Sätzen eine Bedeutung zu. Sie gibt uns die Handhabe, darzustellen, daß wir in der deutschen Sprache den Satz Peter ißt einen Apfel für sinnvoll halten, während wir dies im Fall von Peter ist ein Auto nicht tun, obwohl er syntaktisch korrekt ist. Im Fall der Algebra gibt sie uns an, daß die Interpretation von $a * (b + c)$ gleich der von $a * b + a * c$ ist.

Ein *Ableitungssystem* gibt Regeln an, nach denen aus syntaktisch korrekten Sätzen neue syntaktisch korrekte Sätze gewonnen werden können. Mit einem Ableitungssystem kann z.B. aus den beiden Sätzen 'Genau dann, wenn Katarina beim Weitsprung 8 m weit springt, wird sie zur Olympiade zugelassen' und 'Katarina springt 8 m weit' ein neuer Satz, nämlich 'Katarina wird zur Olympiade zugelassen.' erzeugt werden.[3]

Die Syntax, die Semantik und das Ableitungssystem einer Sprache bilden zusammen ein *formales System*. Wir werden in dieser Einführung alle Teilbereiche zunächst immer in allgemeiner Form vorstellen und dann am Beispiel der Aussagenlogik vertiefen.

2.1.1 Syntax

Hier wird die formale Struktur einer Sprache dargestellt und erläutert, wie diese beschrieben werden kann. Eine formale Sprache wird in ihrem Aufbau durch die beiden Komponenten *Alphabet* und *Syntax* definiert. Diese werden auch unter dem Begriff Syntax zusammengefaßt. Das Alphabet wird im Kontext der Logik auch als *Grundmenge* bezeichnet. Das Alphabet beschreibt die Menge der erlaubten Symbole und die Syntax die Menge der zulässigen Kombinationen. Die gleichen Bestandteile haben wir auch im Fall einer natürlichen Sprache. Im Fall der deutschen Sprache beispielsweise besteht das Alphabet aus der Menge der in der deutschen Sprache zulässigen Wörter (Hund, Mann, Satz) und die Syntax schreibt vor, wie diese kombiniert werden dürfen. Die Menge der, was den Satzaufbau betrifft, erlaubten Sätze bezeichnen wir auch als Menge der *wohlgeformten* oder *syntaktisch korrekten* Sätze. Ein syntaktisch korrekter deutscher Satz ist

> Peter ißt einen Apfel.

Kein syntaktisch korrekter deutscher Satz hingegen wäre

> Peter einen Apfel.

Eine typische Regel der Syntax ist z.B. SPO (Subject, Predicat, Object) im Englischen. Hieran sehen wir, daß die Elemente eines Alphabetes in unserem Sinne keineswegs nur aus einzelnem Zeichen ('A') bestehen müssen. In der Algebra besteht das Alphabet häufig aus Variablennamen, Operationssymbolen und Klammern. In der Biologie kann die Erbinformation mittels einer Syntax beschrieben werden. Das Alphabet besteht dabei aus

[3] In der Semiotik spielen zwei weitere Begriffe eine Rolle.
Die *Pragmatik* gibt an, wie Zeichen zu verwenden sind. Sie ermöglicht es, aus dem Satz Peter ißt einen Apfel die Mitteilung zu ziehen, daß er Apfelvegetarier ist. In der Algebra wird mittels der Pragmatik beschrieben, daß die Formel $a * (b + c)$ eine Angabe für den Endpreis ist, der sich aus der Anzahl der Produkte und ihrem Einzelpreis (i.e. Nettopreis + Mehrwertsteuer) ergibt.
Die *Hermeneutik* lehrt uns die Auslegung von Zeichen und Zeichenfolgen. Dies ist immer dann notwendig, wenn Syntax und Semantik nicht ausreichen. Diese Notwendigkeit ergibt sich z.B. im Fall der Lyrik oder bei Offenbarungstexten. Auslegungsfragen spielen natürlich auch in der Informatik und dort vor allem in der Systemanalyse eine große Rolle. Sie betreffen aber auch den Studenten, der sich mit Textaufgaben herumschlagen muß, bzw. diejenigen, die diese Textaufgaben entwerfen.
Auf diese Begriffe werden wir nicht direkt eingehen, da wir für eine eindeutige Kommunikation auch eindeutige Sprachen benötigen. Sätze, die einer Auslegung bedürfen, sind hierfür ungeeignet. Um Ihnen jedoch ein Gefühl für die Probleme der Systemanalyse zu vermitteln, werden wir auch Textaufgaben stellen, für deren Bearbeitung eine Auslegung des Textes unumgänglich ist.

den Namen der auftretenden Basen, die da sind, Adenin, Cytosin, Guanin, Tymin, Uracil. Hier ist die Struktur der Sprache jedoch bislang weitgehend unbekannt.

Zur Beschreibung des Alphabetes, der Syntax und der Semantik wird eine sogenannte *Metasprache* verwendet. Die Beschreibung natürlicher Sprachen findet normalerweise in der jeweiligen Sprache selbst oder in einer anderen natürlichen Sprache statt. Wir beschreiben die Syntax der Sprache mittels einer *Grammatik*. Eine Grammatik ist eine Menge von Regeln, die angibt, wie die syntaktisch korrekten Sätze über dem Alphabet der Sprache aufzubauen sind. Die im folgenden gewählte Darstellungsform bezeichnet man als **BNF**, als *Backus-Naur Form*. Wir führen die BNF am Beispiel der arithmetischen Ausdrücke ein.

Startsymbol: Formel

Alphabet:

Grundmengen: Konstante, Name

Sonstige Symbole: $+, -, *, /, (,)$

Regeln:

Formel	::=	Term	[1]
	|	Formel + Term	[2]
	|	Formel - Term	[3]
Term	::=	Faktor	[4]
	|	Term $*$ Faktor	[5]
	|	Term / Faktor	[6]
Faktor	::=	Konstante	[7]
	|	Name	[8]
	|	(Formel)	[9]

Kontextbedingungen: keine

Abbildung 2.1: *Syntax der arithmetischen Ausdrücke in Infix-Form*

Die ganz rechts in eckigen Klammern stehende Numerierung der einzelnen Regelbestandteile ist *nicht* Teil der Grammatik oder der Syntax, sondern dient nur der hier benötigten Möglichkeit, auf einzelne Regelalternativen bezug nehmen zu können. Das gleiche gilt für die Bezeichnungen 'Startsymbol', 'Alphabet:', 'Grundmengen', 'Sonstige Symbole', 'Regeln' und 'Kontextbedingungen'.

Das Alphabet beschreiben wir durch eine Aufzählung der zulässigen Elemente. Diese gliedern sich in die Grundmengen und eine Menge sonstiger Symbole. In obigen Beispiel stützen wir uns dabei auf zwei weiter nicht erläuterte Grundmengen, nämlich Name und Konstante ab. Wir gehen einfach davon aus, daß dem Leser die Konventionen für diese Mengen bekannt sind, d.h. daß er in der Lage ist, eines ihrer Elemente auch als solches zu erkennen. Im folgenden werden wir das Alphabet nur noch dann explizit angeben, wenn dies sinnvoll erscheint.

Die Grammatik für die Syntax ist wie folgt zu interpretieren. Das Zeichen ::= trennt die linke und die rechte Seite einer Regel. Die Worte Formel, Term und Faktor sind Platzhalter oder Variable. Sie werden als *Non-Terminalsymbole* bezeichnet. Formel ist ein ausgezeichnetes Non-Terminalsymbol. Man bezeichnet es als *Startsymbol*. Häufig wird bei Grammatiken wie der obigen auf die explizite Angabe des Startsymbols verzichtet, es gilt dann die Konvention, daß das Symbol auf der linken Seite der ersten Regel das Startsymbol ist. Wir vereinbaren jedoch, daß das Startsymbol immer angegeben werden muß. Die Symbole, die nicht Non-Terminalsymbole sind, werden als *Terminalsymbole* bezeichnet. Sie dürfen nur auf der rechten Seite einer Regel vorkommen. Die Menge der Terminalsymbole entspricht dem Alphabet. Die Terminalsymbole, die nicht Bezeichner für eine Grundmenge sind, müssen in der Menge 'Sonstige Symbole' enthalten sein. In obigem Beispiel ist dies $+, -, *$, **/**, **(** und **)**.

Um nun konkrete Sätze über dieser Grammatik zu bilden, d.h. eine *Ableitung* zu erzeugen, beginnt man mit dem Startsymbol. Dieses wird durch eine der möglichen Alternativen auf der rechten Seite ersetzt. Die

möglichen Alternativen auf der rechten Seite sind mittels | getrennt. Sie werden als syntaktische Alternativen bezeichnet. Falls auf der rechten Seite einer Regel (also auch der Startregel) ein Non-Terminalsymbol auftritt, wird dieses durch eine der Alternativen auf der rechten Seite einer Regel ersetzt, bei der dieses Non-Terminalsymbol links steht. Terminalsymbole werden direkt Bestandteil des zu konstruierenden Satzes. In einem konkreten Satz erscheinen dabei konkrete Elemente der Grundmengen Namen und Kostanten. In unserem Beispiel mit Peter und dem Apfel, wären Subjekt, Prädikat und Objekt Grundmengen, die in diesem Satz verwendeten Elemente wären Peter (als Element der Menge Subjekt), ißt (als Element der Menge Prädikat) und einen Apfel (als Element der Menge Objekt). Dieser Prozeß wird solange fortgeführt, bis alle Non-Terminalsymbole durch Terminalsymbole ersetzt sind. Dabei kann jede Regel beliebig häufig (d.h. auch gar nicht) angewandt werden, vorausgesetzt, ihre linke Seite 'paßt'. Das folgende Beispiel enthält den Nachweis, daß der Satz a ∗ b + c eine syntaktisch korrekte Formel ist, d.h., daß er mit der obigen Grammatik ableitbar ist. Dabei steht ⟶ für 'wird abgeleitet zu' und die Nummer rechts bezeichnet die verwendete Regel).

Formel	⟶ Formel + Term	[2]
	⟶ Term + Term	[1]
	⟶ Term ∗ Faktor + Term	[5]
	⟶ Faktor ∗ Faktor + Term	[4]
	⟶ a ∗ Faktor + Term	[8]
	⟶ a ∗ b + Term	[8]
	⟶ a ∗ b + Faktor	[4]
	⟶ a ∗ b + c	[8]

In diesem Beispiel werden die mathematischen Ausdrücke in der sogenannten *Infix-Notation* dargestellt. Hierbei steht der Operator zwischen (in) den Operanden. a+b wäre ein typisches Beispiel für die Infix-Darstellung (oder -Notation). Bei Operatoren mit mehr als zwei Operanden ist diese Schreibweise nicht mehr gut verwendbar. Stattdessen wird in solchen Fällen die *Prefix-Darstellung*, der Operator steht *vor* den Operanden, verwendet. In der Prefix-Darstellung könnte die zwei-stellige Vergleichsoperation ≥ mit den Parametern a und b als ≥(a, b) geschrieben werden.

Wenn hier von Syntax die Rede ist, dann gehen wir immer davon aus, daß eindeutig zu identifizieren ist, welcher Art Objekt ein Wort der Sprache ist, d.h. zu welcher Grundmenge es gehört. Dies ist, was die natürliche Sprache betrifft u.U. eine unzulässige Annahme, wie das folgende Beispiel zeigt:

Peter zieht gerade Linien

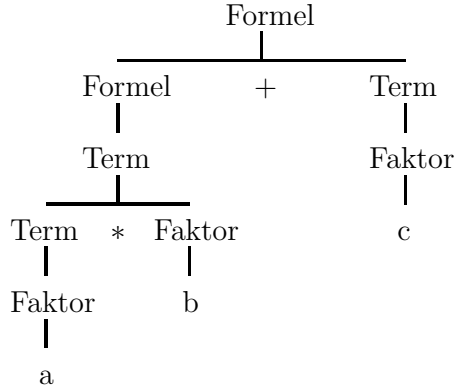
Hier ist nicht klar zu entscheiden, ob gerade ein Adjektiv oder eine zeitliche Bestimmung ist; sind die Linien, die Peter zieht, gerade, oder ist er just in diesem Augenblick dabei Linien (u.U. krumme) zu ziehen? Ein ähnliches Problem ergibt sich auch im Fall mehr formaler Sprachen, z.B. Programmiersprachen. Die Anweisung

if a ≤ b then if b ≤ c then x := a else x := c

ist mehrdeutig, da unklar ist, zu welcher if-Anweisung das else gehört. Mithin stellt sich im Fall von $a > b$ die Frage nach dem Wert von x. Hier muß die Struktur der Anweisung eindeutig interpretierbar sein. Wir gehen hier immer davon aus, daß die Elemente unserer Grundmengen disjunkt sind, d.h., daß jedem Element des Alphabetes genau eine Menge zugeordnet werden kann und daß die Struktur unserer Sätze eindeutig erkennbar ist. Letzteres bedeutet, daß die Grammatik die Möglichkeit bieten muß, den Ableitungsprozeß zu rekonstruieren, d.h. einen *Ableitungsbaum* zu erstellen.

Ein Ableitungsbaum ist eine Darstellung der Ableitung unabhängig von der Ersetzungsauswahl. Der Ableitungsbaum zeigt deutlich die Bindungsregeln, die durch die Grammatikstruktur vorgegeben sind. Falls es zur Erzeugung eines bestimmten Wortes über einer Grammatik nur eine Folge von Regelanwendungen gibt, dann bezeichnen wir die Herleitung als *eindeutig*. Eine eindeutige Herleitung erzwingt dabei auch einen eindeutigen Ableitungsbaum, während es zu einem Ableitungsbaum durchaus auch mehrere Herleitungen geben kann. Diese unterscheiden sich dann nur in der Reihenfolge ihrer Regelanwendungen. In unserem Herleitungsbeispiel hatten wir z.B. immer das am weitesten links stehende Non-Terminal-Symbol expandiert, wir hätten

natürlich auch immer das am weitesten rechts stehende verwenden oder beide Vorgehensweisen vermischen können. Wir bezeichnen die Grammatik als eindeutig, falls es zu jedem Wort über dieser Grammatik nur einen Ableitungsbaum gibt. Die folgende Abbildung zeigt den Ableitungsbaum für unser Beispiel a ∗ b + c mit obiger Grammatik.



Die folgende Grammatik erzwingt die von uns für die Punkt- und Strichrechnung intendierten Strukturen nicht, erzeugt aber die gleiche Sprache (auf den Nachweis wird hier verzichtet):

Startsymbol: Formel

Alphabet:

Grundmengen: Konstante, Name

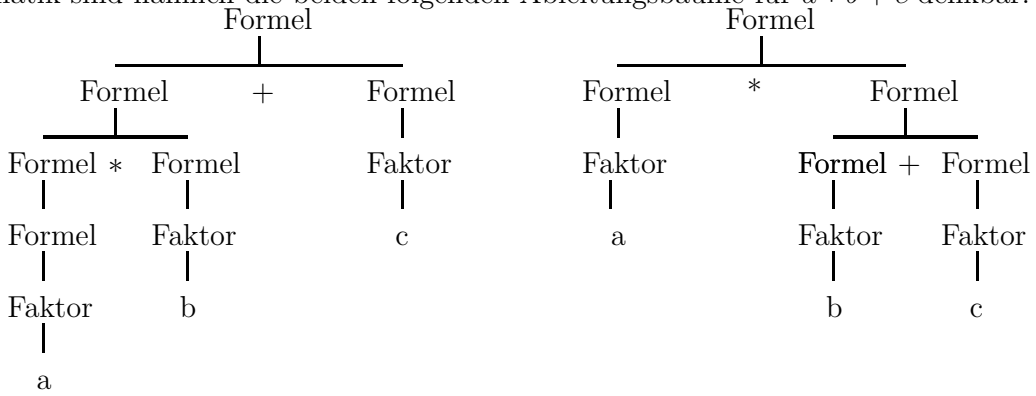
Sonstige Symbole: +, −, ∗, /, (,),

Regeln:

Formel	::=	Faktor	[1]
	|	Formel + Formel	[2]
	|	Formel - Formel	[3]
	|	Formel ∗ Formel	[4]
	|	Formel / Formel	[5]
Faktor	::=	Konstante	[6]
	|	Name	[7]
	|	(Formel)	[8]

Abbildung 2.2: *Syntax der arithmetischen Ausdrücke, mehrdeutige Version*

Mit dieser Grammatik sind nämlich die beiden folgenden Ableitungsbäume für $a * b + c$ denkbar:



Manche syntaktischen Anforderungen können nicht mit Hilfe einer BNF-Grammatik beschrieben werden. Ein Beispiel hierfür ist eine Zeichenkette, die der Form $a^n b^n c^n$ genügen soll. a^n steht dabei für eine Folge aus n-mal dem Zeichen 'a'. Ein weiteres Beispiel für eine solche Anforderung ist die Bedingung, daß ein Satz über einer Grammatik nur dann syntaktisch korrekt ist, wenn die Zahl der angegebenen Operanden für einen Operator dessen Stelligkeit entspricht. Die Vergleichs-Operation in der Arithmetik beispielsweise hat die Stelligkeit 2, da (üblicherweise) zwei Zahlen verglichen werden. Es wäre sinnlos, $\geq_2$(a, b, c) zu schreiben, da $\geq_2$ nur zwei Zahlen vergleichen kann.

Wenn wir eine Menge von Sprachen beschreiben wollen, die sich nur in ihren Grundmengen unterscheiden, und Grundmengen Operatoren verschiedener Stelligkeit enthalten, dann müssen wir *Kontextbedingungen* einführen. Seien $+_2$, $+_3$ die zweistellige bzw. die dreistellige Plus- Operation. Dann stellen die Kontextbedingungen zu folgender Grammatik sicher, daß $+_2$ nur auf 2 Operanden und $+_3$ nur auf drei Operanden angewendet werden darf.[4] $+_3(3, 4)$ wäre damit unzulässig. $+_2$, $+_3$ sind dabei Elemente aus der Grundmenge Operator.

Startsymbol:	Formel2			
Alphabet:				
Grundmengen	Name, Konstante, Operator			
Sonstige Symbole	(,), ,			
Regeln:	Formel2	::=	Konstante	[1]
			| Name	[2]
			| Funktionsanwendung	[3]
	Funktionsanwendung	::=	Operator(Operanden_Liste)	[4]
	Operanden_Liste	::=	Operand	[5]
			| Operand, Operanden_Liste	[6]
	Operand	::=	Formel2	[7]

Kontextbedingungen: Genau dann, wenn $t_1, t_2, ..., t_n$ Operanden sind und f ein n-stelliges Element aus Operator ist, ist auch $f(t_1, t_2, ..., t_n)$ eine Funktionsanwendung.

Abbildung 2.3: *Syntax der arithmetischen Ausdrücke in Prefix-Form*

Am Beispiel der Prädikatenlogik werden wir eine weitere nichtleere Menge Kontextbedingungen kennenlernen.

Zusätzlich zu der bisher gezeigten Darstellung der Syntax von Sprachen werden auch sogenannte *Syntaxdiagramme* verwendet. Wir geben nur ein Beispiel, das nach dem vorangegangenen selbsterklärend sein müsste.

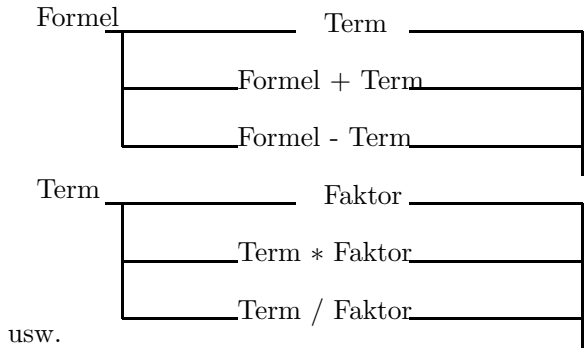


Abbildung 2.4: *Syntaxdiagramme für die arithmetischen Ausdrücke*

2.1.2 Die Syntax der Aussagenlogik

Wir verwenden im folgenden die Syntax der Aussagenlogik, um das Wissen über die Beschreibung der Syntax einer Sprache zu vertiefen. Daß dies gerade mittels der Aussagenlogik geschieht, hat zwei Gründe. Zum ersten hat die Aussagenlogik eine relativ einfache Sprache, zum anderen stellt sie die Grundlage für die später eingeführte Prädikatenlogik dar.

Die Aussagenlogik ermöglicht die Verknüpfung von Aussagen. Eine Aussage ist dabei ein Satz, der entweder wahr oder falsch sein kann. Aussagen, die nicht durch Verknüpfung entstanden sind, werden als *atomare* Aussagen bezeichnet.

[4] Man beachte, daß auch die Kontextbedingungen nichts über die Bedeutung unserer Zeichenketten sagen.

Atomare Aussagen sind zum Beispiel:

- Berlin ist die Hauptstadt der Bundesrepublik Deutschland
- Ich bin Nichtraucher
- Es regnet
- 17 ist eine Primzahl

Demgegenüber sind folgende Sätze keine Aussagen im Sinne der Aussagenlogik, weil ihnen kein Wahrheitswert zugewiesen werden kann:

- Guten Morgen
- Warum immer ich ?

Bei den Beispielen für die atomaren Aussagen zeigt sich, daß obige Definition zumindest problematisch ist, da die Korrektheit des zweiten Beispiels von demjenigen abhängt, der behauptet, er rauche nicht. Wir gehen im folgenden davon aus, daß wir atomaren Aussagen immer eindeutig einen Wahrheitswert zuordnen können.

Die Zusammensetzung der Teilaussagen erfolgt mittels *Junktoren*. Beispiele hierfür sind:

- Jan hat grüne Stiefel, und Johanna hat blaue Stiefel
- München hat am 2.6.91 1700000 Einwohner, und die Ostsee ist an ihrer tiefsten Stelle 6 Meter tief

Wir werden im folgenden Aussagen wie 'Jan hat grüne Stiefel' z.B. mit **A** benennen. Es ist nämlich zum einen sehr mühsam, immer 'Jan hat grüne Stiefel' schreiben zu müssen (**A** ist viel kürzer) und zum zweiten ist der eigentliche Inhalt der Aussage für die Aussagenlogik insoweit uninteressant, als daß wir nur wissen müssen, ob die atomare Aussage wahr oder falsch ist. Diese Entscheidung fällt aber nicht in das Aufgabengebiet der formalen Logik.

Da wir die Aussagenlogik als formales System begreifen wollen, werden wir zunächst die Syntax festlegen und im Kapitel Semantik den zusammengesetzten Aussagen eine eindeutige Bedeutung geben.

Startsymbol:	Aussage		
Alphabet:			
Grundmengen	Bezeichner		
Sonstige Symbole	$\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, (,), T, F$		
Syntax:	Aussage	::=	Atomaraussage
		|	($\neg$ Aussage)
		|	(Aussage $\wedge$ Aussage) | (Aussage $\vee$ Aussage)
		|	(Aussage $\Rightarrow$ Aussage) | (Aussage $\Leftrightarrow$ Aussage)
	Atomaraussage	::=	T | F | Bezeichner

Abbildung 2.5: *Syntax der Aussagenlogik (mit voller Klammerung)*

Für die eingeführten Operatoren wird die folgende Terminologie verwendet. $\wedge$ wird als *Konjunktion* bezeichnet, $\vee$ als *Disjunktion*, $\neg$ als *Negation*, $\Rightarrow$ als *Implikation* und $\Leftrightarrow$ als *Äquivalenz*. b $\wedge$ c wird gelesen als '*b und c*', b $\vee$ c als '*b oder c*', $\neg$ a als '*nicht a*', b $\Rightarrow$ c als '*b impliziert c*' und b $\Leftrightarrow$ c als '*b gleich c*'.[5]

Der Leser möge sich selbst das hier verwendete Alphabet klarmachen. Wir treffen hier nur in soweit eine Vereinbarung, daß Bezeichner aus einem oder einer Folge kleiner Buchstaben gewählt werden können. Jeder Bezeichner steht für eine atomare Aussage wie die bereits dargestellten. Damit ist auch klar, daß zu jeder Beschreibung eines Problems eine Zuordnung zwischen Bezeichnern und Aussagen zu treffen ist. Diese erfolgt jedoch nicht mit den sprachlichen Mitteln der formalen Logik (s. auch folgende Beispiele).

[5]In den meisten Programmiersprachen wird deshalb anstelle von $\Leftrightarrow$ das Gleichheitssymbol = verwendet.

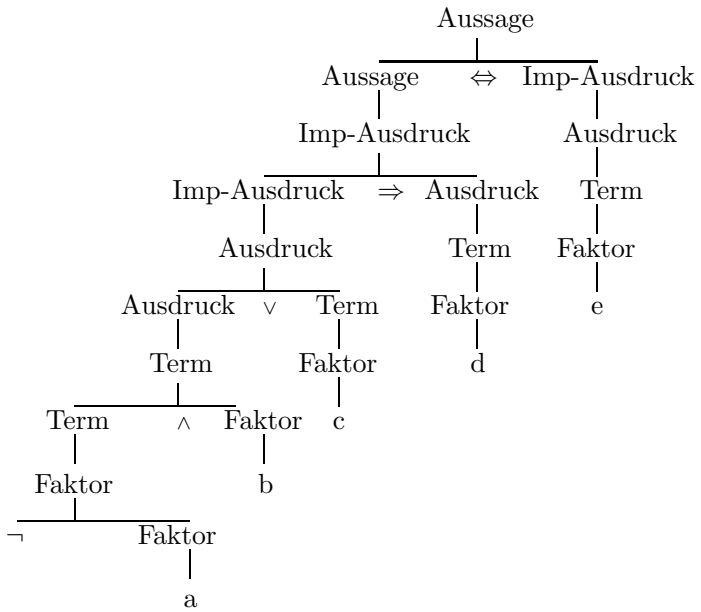
Da wir im folgenden den Begriff der Teilaussage benötigen, führen wir hier informal ein, daß *Teilaussagen* alle Aussagen sind, die Teil einer übergeordneten Aussage sind und mittels der Operatoren $\wedge, \vee, \neg, \Rightarrow$ und $\Leftrightarrow$ miteinander verknüpft werden. Die Aussage (b $\vee$ c) enthält die (atomaren) Teilaussagen b und c.

Wie man sieht, sind alle nicht atomaren Aussagen geklammert. Die folgende Grammatik (analog [Gries, 1981]) macht dies überflüssig.

Startsymbol:	Aussage		
Alphabet:			
Grundmengen	Bezeichner		
Sonstige Symbole	$\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, (,), T, F$		
Syntax:	Aussage	::=	Imp-Ausdruck | Aussage $\Leftrightarrow$ Imp-Ausdruck
	Imp-Ausdruck	::=	Ausdruck | Imp-Ausdruck $\Rightarrow$ Ausdruck
	Ausdruck	::=	Term | Ausdruck $\vee$ Term
	Term	::=	Faktor | Term $\wedge$ Faktor
	Faktor	::=	$\neg$ Faktor | (Aussage) | T | F | Bezeichner

Abbildung 2.6: *Syntax der Aussagenlogik (Klammerung nur soweit notwendig)*

In dieser Grammatik werden die folgenden *Wertigkeiten* der Operatoren realisiert: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$. Die Wertigkeit nimmt von links nach rechts ab, der ganz links stehende Operator $\neg$ bindet am stärksten. Für Folgen gleicher Operatoren ergibt sich, daß a $\wedge$ b $\wedge$ c identisch ist zu ((a $\wedge$ b) $\wedge$ c). Das soll heißen, daß zuerst der am weitesten links stehende Operator seine beiden Operanden bindet. Falls mehr als drei Operatoren auftreten, gilt natürlich dies natürlich auch und entsprechend für die anderen Operatoren. Diese Wertigkeit ist die übliche. Da die Definition der Semantik im folgenden Kapitel deutlich unübersichtlicher wird, wenn die erste Grammatikvariante Verwendung findet, benutzen wir auch an dieser Stelle die zweite. Dies macht dann insbesondere auch den Umgang mit den Semantik-Funktionen sehr viel einfacher. Man muß sich jedoch, wenn man auf die Klammern verzichtet, immer über die Priorität der Operatoren klar sein, insbesondere bei der Bestimmung bei der Bestimmung der Semantik eines Ausdruckes. Wenn im folgenden die Rede von Aussagen ist, dann sind immer solche gemeint, die obiger Syntax entsprechen. Beispiele für syntaktisch korrekte Aussagen sind T und (b $\vee$ c). Der folgende Ableitungsbaum zeigt, daß dies auch für $\neg a \wedge b \vee c \Rightarrow d \Leftrightarrow e$ gilt.



Wir möchten nochmals darauf hinweisen, daß trotz der teilweise recht suggestiven Namensgebung die Sätze über unserer Grammatik noch keine Bedeutung haben. Dies gilt insbesondere auch für die Symbole T und F.

2.1.3 Semantik

Wir werden nun zeigen, wie man die Semantik einer Sprache definieren kann. Zu diesem Zweck führen wir die 'denotationale' Semantik ein. Vorab sei jedoch gesagt, daß weitere Formen existieren. Streng genommen ist die denotationale Interpretation einer Sprache die Beschreibung des Zusammenhangs zwischen Sätzen der zu definierenden Sprache (*Quellsprache*) und einer zweiten, bei der man so tut, als wäre deren Bedeutung klar. Dies ist bespielsweise der Fall, wenn die Semantik der französischen Sprache mittels der deutschen beschrieben wird. Diese zweite Sprache bezeichnen wir als *Zielsprache*. Die Berechtigung für die Annahme, daß die Bedeutung der Zielsprache bekannt sei, ergibt sich daraus, daß für die Interpretation der Zielsprache häufig eine Struktur gewählt wird, die mathematisch gesehen, eine bekannte ist. Beispiele hierfür sind der (mathematische) Verband, Ring usw.

Der erste Schritt besteht darin, daß die Objekte (i.e. die Terminalsymbole) der Zielsprache aufgeführt werden. In unserem Fall ist das die Menge der Wahrheitswerte, **wahr** und **falsch**. Weiter benötigen wir eine Menge von Operationen auf den Objekten der Zielsprache. Im Fall der Aussagenlogik sind das die Operationen `und, oder, nicht, impl` und `gleich`. Man beachte, daß wir im vorhergegangenen Abschnitt 'Syntax der Aussagenlogik' nur eine Menge von Zeichen ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, (,)) eingeführt haben.

Den einzelnen syntaktischen Alternativen der Quellsprache wird mittels einer *Interpretationsfunktion* eine Bedeutung in der Zielsprache zugeordnet. Die Zielsprache ist eine *funktionale*, d.h. alle Ausdrücke werden als Funktionen beschrieben. Dem französischen Wort 'chien' wird das deutsche Wort 'Hund' zugeordnet.

Eine Eigenschaft der formalen Logik soll hier noch hervorgehoben werden. Üblicherweise wird bei der Definition einer formalen Logik unterschieden zwischen Symbolen, die Operationen ausdrücken (z.B. $\wedge$) und anderen, z.B. Bezeichnern. Erstere werden auch als Operatoren bezeichnet. In der Definition der Semantik wird dann jedem Operationssymbol der Logiksprache genau ein Operationssymbol der Semantiksprache zugeordnet. Dies ist für die Definition der Semantik einer formalen Sprache aber nicht zwingend. Es ist statt dessen natürlich auch möglich, alle Symbole der Quellsprache auf ein Symbol der Semantiksprache abzubilden (u.U. mit verschiedenen Parametern) oder abhängig von dem Zusammenhang in dem das quellsprachliche Symbol auftritt, verschiedene Zielsprachensymbole zu verwenden. Typische Beispiele für solche Fälle finden sich bei der Übersetzung natürlicher Sprachen ineinander. Hier kommt es sowohl vor, daß einem Verb (als Analogon zu einem Operator) mehrere Bedeutungen in der anderen Sprache zugeordnet werden oder aber mehreren Verben nur eine. Beispiel für den ersten Fall ist die Interpretation des deutschen Wortes 'können' als 'savoir' oder 'pouvoir', je nach intendierter Bedeutung. Ein Beispiel für den zweiten Fall ergibt sich aus der Umkehrung, wenn Französich in Deutsch 'übersetzt' wird. Normalerweise ist man in der Logik an den Eigenschaften formaler Systeme interessiert. Daraus ergibt sich die Notwendigkeit, diesen Systemen eine Semantik zu geben. Hierbei ist es häufig von Interesse, zu prüfen, welche Typen von Semantik für ein formales System 'Sinn machen' und welche Typen von Aussagen unabhängig von der gewählten Semantik immer 'richtig' sind. Wir benutzen die Logik als formales Beschreibungsmittel. Also müssen wir auch eine Idee davon entwickeln, welcher Typ Aussage 'wahr' ist, bzw. wie der Begriff der Wahrheit überhaupt in unserem formalen System gefaßt werden kann. Eine allgemeinere Motivation für die Einführung von Quellsprachen überhaupt ist, daß es hierdurch ermöglicht wird, die betrachtete Abstraktionsebene zu verändern, bzw. das Beschreibungswerkzeug, denn das genau ist die Sprache, den Anforderungen des zu beschreibenden Gebietes anzupassen.

In üblichen Lehrbüchern zum Thema Logik wird häufig mehr oder weniger darüber hinweggegangen, daß zunächst die Syntax und anschließend die Semantik der verwendeten formalen Sprache definiert werden müssen. Häufig werden die quell- und die metasprachlichen Operatoren einfach gleichgesetzt und der Zwischenschritt über die Semantik vermieden. Die dann angegebene Wertetafel erscheint dann eine Wertetafel für die Quellsprache zu sein. Unsere Motivation, diese Dinge möglichst sauber zu trennen, ist eine zweifache. Zum einen werden hiermit die Grundlage derjenigen Mechanismen zur Verfügung gestellt, die später für die Definition der Programmiersprache Eiffel verwendet werden. Zum anderen wird dadurch die Notwendigkeit des Kapitels über Ableitungssysteme deutlicher.

Für die Zuordnung zwischen der Objekten der Quellsprache und denen der Zielsprache benötigen wir wiederum eine Metasprache. Diese entspricht der BNF im Fall der Syntax. Deren Elemente werden wir jedoch nur insoweit einführen, als daß wir sie aktuell benötigen. Später, bei der Definition der Programmiersprache Eiffel, werden wir weitere Konstruktionselemente kennenlernen. Hier soll zunächst am Beispiel der Aussagenlogik die prinzipielle Vorgehensweise dargestellt werden.

Im ersten Schritt definieren wir die Zielsprache. Wir führen zunächst die Objekte der Zielsprache ein:

Ziel_Wahrheitswerte = { **wahr**, **falsch**}
Ziel_Operatoren = { `und, oder, nicht, impl, gleich` }

und legen weiter fest, daß für die Anwendung der Ziel_Operatoren auf die Ziel_Wahrheitswerte folgende Wertetabelle gelten soll. Dabei zeigt in der folgenden Tabelle jede Zeile für eine mögliche Wertekombination von b und c den Wert der fünf zielsprachlichen Operationen.

b	c	(`nicht` b)	(b `und` c)	(b `oder` c)	(b `impl` c)	(b `gleich` c)
falsch	**falsch**	**wahr**	**falsch**	**falsch**	**wahr**	**wahr**
falsch	**wahr**	**wahr**	**falsch**	**wahr**	**wahr**	**falsch**
wahr	**falsch**	**falsch**	**falsch**	**wahr**	**falsch**	**falsch**
wahr	**wahr**	**falsch**	**wahr**	**wahr**	**wahr**	**wahr**

Abbildung 2.7: *Wahrheitstafel für die Aussagenlogik*

Der Wert einer Aussage mit mehr als einem Operator wird dadurch bestimmt, daß obige Tabelle solange auf die Teilaussagen angewandt wird, bis die gesamte Aussage auf **wahr** oder **falsch** reduziert ist.

Beispiel: ((**wahr** `und` **falsch**) `oder` (**wahr** `und` **wahr**)) $\rightsquigarrow$ (**falsch** `oder` **wahr**) $\rightsquigarrow$ **wahr**

An dieser Stelle wird '$\rightsquigarrow$' benutzt, um die sich aus der Wahrheitstafel ergebende Gleichheit zu beschreiben. $\rightsquigarrow$ ist dabei kein Element der Sprache der Aussagenlogik. Da es aber nur endlich viele Symbole für die verschiedenen Gleichheiten gibt und meist ohnehin aus dem Kontext entscheidbar ist, welches gemeint ist, werden wir im folgenden statt $\rightsquigarrow$ immer = schreiben.

Nun sind wir in der Lage, in der Zielsprache zu rechnen. Der nächste Schritt besteht darin, den Zusammenhang zwischen Ausdrücken der Quell- und denen der Zielsprache festzulegen.

Hierzu benötigen wir zunächst den Begriff der *Funktionalität*. Die Funktionalität gibt an, welche Typen Objekte haben dürfen, auf die eine Funktion angewendet werden soll. Typen sind dabei Mengen. Ein Beispiel aus der Analysis möge dies einführen. Die Funktionalität der Wurzelfunktion kann beschrieben werden als:

$\sqrt{\ }$: Reelle-Zahl $\rightarrow$ Reelle-Zahl

Dieser Ausdruck ist so zu interpretieren, daß $\sqrt{\ }$ eine Abbildung von *Reelle-Zahl* nach *Reelle-Zahl* ist. Es wird also gefordert, daß die Funktion $\sqrt{\ }$ nur auf Elemente aus der Menge der reellen Zahlen angewendet werden darf. Ergebnis ist immer ein Element aus der Menge rechts des letzten Pfeils. Allgemeiner könnte man auch schreiben:

$\sqrt{\ }$: Reelle-Funktion $\rightarrow$ Reelle-Funktion

Hier wäre Reelle-Funktion eine Funktion, die ihrerseits eine reelle Funktion als Ergebnis liefert. Diese Verallgemeinerung ist sinnvoll, da nur so beschrieben werden kann, daß Funktionen auf Funktionen angewendet werden dürfen. Ein Beispiel hierfür ist die Funktion $\sqrt{\sqrt[3]{\ }}$, die durch Anwendung der Funktion $\sqrt{\ }$ auf die Funktion $\sqrt[3]{\ }$ entsteht. Ein weiteres Beispiel ist die wohl bekannte Multiplikationsfunktion. Üblicherweise schreiben wir: $a * b = c$, alternativ könnte man dies jedoch als $mult(a, b) = c$ schreiben. Die erste Schreibweise bezeichnet man auch als *Infix-Notation* und die zweite als *Prefix-Notation*. *mult* ist eine zweistellige Funktion, mit $mult : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$. Man mache sich deutlich, daß natürlich auch gilt: $* : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$. Durch die Infix-Schreibweise wird weniger deutlich, daß $*$, $+$ usw. natürlich auch Funktionen sind und zwar Funktionen der Stelligkeit zwei.

Für den Zusammenhang zwischen Quell- und Zielsprache definieren wir eine *Interpretationsfunktion* mit folgendender Funktionalität:

s : Syntaktische_Aussage $\times$ Zustand $\rightarrow$ Ziel_Wahrheitswert

Diese Definition besagt, daß die Funktion s ein Objekt vom Typ Syntaktische_Aussage (soll heißen einen Ableitungsbaum) und ein Objekt von Typ Zustand nimmt und ein Objekt vom Typ Ziel_Wahrheitswert (soll heißen ein Element der Zielsprache) liefert. D.h. die Funktion s ist eine Funktion mit zwei Argumenten, die einen Ausdruck der Quellsprache zusammen mit einem Zustand auf einen Wahrheitswert der Zielsprache abbildet. Aus Platzgründen werden wir im folgenden, statt jedesmal explizit den Baum anzugeben, uns auf den durch ihn ableitbaren Ausdruck beschränken.

Wozu benötigen wir einen Zustand? In dem 'Quell_Ausdruck' treten u.U. Bezeichner auf. Wir brauchen daher eine Handhabe, um zu entscheiden, welcher Wert (d.h. welches Element aus Ziel_Wahrheitswerte) einem Bezeichner zugeordnet wird. Zu diesem Zweck führen wir den Begriff des *Zustandes* ein.

Definition 2.1.1 (Zustand)

Ein Zustand ist eine Funktion, von der Menge der Bezeichner in die der Wahrheitswerte **wahr** *und* **falsch**, *also die semantischen Wahrheitswerte, d.h.*

Zustand : Bezeichner $\rightarrow$ Ziel_Wahrheitswert

In der Logik wird die Funktion Zustand häufig als Belegung *bezeichnet.*

Dieser 'Zustand', d.h. die Funktion kann z.B. über eine *Wertetabelle* festgelegt werden.

Beispiel 2.1.2

Sei Zustand *die durch* `[a` $\mapsto$ **wahr**`, bc` $\mapsto$ **falsch**`, mu` $\mapsto$ **wahr**`]` *gegebene Funktion.*
Dann bedeutet Zustand(a) *die Anwendung der Funktion* Zustand *auf den Bezeichner* a*:*
Zustand(a) = **wahr** *und entsprechend* Zustand(bc) = **falsch** *und* Zustand(mu) = **wahr**.
Man beachte, daß Zustand *keine* totale *Funktion ist, da* Zustand *nicht für alle möglichen Bezeichner definiert ist;* Zustand(Marion) *ist bspw. undefiniert. Man mache sich klar, daß auch bei der Auswertung einer Gleichung $x + 4 = 9$ der Wert von x bekannt sein muß, um den Wert der Aussage zu ermitteln.*

Wir gehen davon aus, daß jeder Bezeichner in einer Aussage in dem jeweiligen Zustand definiert ist; anderenfalls ist die gesamte Interpretation undefiniert. Dann werden die Definitionsgleichungen über die syntaktische Struktur der Aussagen rekursiv wie folgt definiert:

s (Bezeichner, state)	=	state(Bezeichner)
s (T, state)	=	**wahr**
s (F, state)	=	**falsch**
s ((Aussage), state)	=	s (Aussage, state)
s ($\neg$ Faktor, state)	=	(`nicht` s (Faktor, state))
s (Term $\wedge$ Faktor, state)	=	(s (Term, state) `und` s (Faktor, state))
s (Ausdruck $\vee$ Term, state)	=	(s (Ausdruck, state) `oder` s (Term, state))
s (Imp-Ausdruck $\Rightarrow$ Ausdruck, state)	=	(s (Imp-Ausdruck, state) `impl` s (Ausdruck, state))
s (Aussage $\Leftrightarrow$ Imp-Ausdruck, state)	=	(s (Aussage, state) `gleich` s (Imp-Ausdruck, state))

Abbildung 2.8: *Semantik aussagenlogischer Formeln*

Was auf den ersten Blick kompliziert aussieht, ist eigentlich nicht sehr schwierig. 's (Quell_Ausdruck, state)' bedeutet, daß die Semantik des quellsprachlichen Ausdrucks Quell_Ausdruck im Zustand 'state' definiert wird; genauer gesagt, daß die Funktion s auf die beiden Argumente *Quell_Ausdruck* und state angewendet wird.

Rechts von dem Gleichheitszeichen steht dann die zielsprachliche Definition für Quell_ausdruck. In dieser Definition kann die Funktion s wieder auftreten. Das bedeutet dann, daß an dieser Stelle der Parameter auf seine syntaktische Struktur hin zu prüfen ist und der Ausdruck gemäß den Definitionen ersetzt werden muß. Zu berücksichtigen ist hier, daß auch diese Anwendung der Funktion s natürlich wieder einen state erfordert.

Diese Form der Definition findet ihr Analogon in den Nicht-Terminalen der BNF. Es bleibt zu bemerken, daß zu jeder syntaktischen Alternative eine semantische Definition angegeben werden sollte.

Beispiel 2.1.3

Sei state der durch [a ↦ **wahr**, b ↦ **falsch**, c ↦ **wahr**, d ↦ **wahr**] *gegebene Zustand.*
Es soll die Semantik des Ausdrucks ((a ∧ b) ∨ (c ∧ d)) bestimmt werden:

$s((a \wedge b) \vee (c \wedge d))$, state)
= ($s((a \wedge b)$, state) oder $s((c \wedge d)$, state))
= (($s(a$, state) und $s(b$, state)) oder ($s(c$, state) und $s(d$, state)))
= ((**wahr** und **falsch**) oder (**wahr** und **wahr**))

dieser Ausdruck kann mit der Wahrheitstafel aus Abbildung 2.7 reduziert werden zu:

(**falsch** oder **wahr**)
= **wahr**

Beispiel 2.1.4

Sei state der durch [a ↦ **wahr**, b ↦ **falsch**, c ↦ **wahr**, d ↦ **falsch**, e ↦ **wahr**] *gegebene Zustand. Es soll die Semantik des Ausdrucks* $\neg a \wedge b \vee c \Rightarrow d \Leftrightarrow e$ *bestimmt werden:*

$s(\neg a \wedge b \vee c \Rightarrow d \Leftrightarrow e$, state)
= $s(\neg a \wedge b \vee c \Rightarrow d$, state) gleich $s(e$, state)
= $s(\neg a \wedge b \vee c \Rightarrow d$, state) gleich **wahr**
= ($s(\neg a \wedge b \vee c$, state) impl $s(d$, state)) gleich **wahr**
= ($s(\neg a \wedge b \vee c$, state) impl **falsch**) gleich **wahr**
= (($s(\neg a \wedge b$, state) oder $s(c$, state)) impl **falsch**) gleich **wahr**
= (($s(\neg a \wedge b$, state) oder **wahr**) impl **falsch**) gleich **wahr**
= ((($s(\neg$ a) und s(b, state)) oder **wahr**) impl **falsch**) gleich **wahr**
= ((($s(\neg$ a) und **falsch**) oder **wahr**) impl **falsch**) gleich **wahr**
= (((nicht s(a), state) und **falsch**) oder **wahr**) impl **falsch**) gleich **wahr**
= (((nicht **wahr**) und **falsch**) oder **wahr**) impl **falsch**) gleich **wahr**

Reduktion unter Anwendung der Wahrheitstafel für die Semantikfunktionen ergibt:

(((**falsch** und **falsch**) oder **wahr**) impl **falsch**) gleich **wahr**
= ((**falsch** oder **wahr**) impl **falsch**) gleich **wahr**
= (**wahr** impl **falsch**) gleich **wahr**
= (**wahr** impl **falsch**) gleich **wahr**
= **falsch** gleich **wahr**
= **falsch**

Der geneigte Leser möge sich deutlich machen, daß an Stelle der hier angegebenen Semantikfunktion auch eine möglich ist, die die Wahrheitswerte der Quellsprache auf '0' und '1', bzw. die Operatoren auf '+', '∗' und '*inv*' abbildet. Diese wird üblicherweise als *Boole'sche Algebra* bezeichnet, und es wird die folgende Wertetafel verwendet:

b	c	(inv b)	(b ∗ c)	(b + c)
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

Die Boole'sche Algebra findet bei dem Entwurf von Schaltungen vielfache Verwendung. Selbstverständlich gleichfalls möglich, wenn auch geringfügig verwirrend, wäre es, das '∧' der Quellsprache auf ein 'oder' der Zielsprache abzubilden. Damit wird für einen Ausdruck über dieser Sprache selbstverständlich eine völlig andere Semantik definiert. Diese Beispiele machen hoffentlich nochmals deutlich, daß durch die Syntax keinerlei Semantik vorgegeben ist.

Zeitweilig ist es sinnvoll, statt den Wert einer Aussage in einem konkreten Zustand zu berechnen, diesen Wert mit dem Zustand zu parametrisieren. Für die Bestimmung des Wertes in einem konkreten Zustand muß dann lediglich die in dieser Form berechnete Funktion auf den konkreten Zustand angewendet werden. Damit erhalten wir, statt der bisher verwendeten Funktion s eine Funktion (ein Funktional) S, die uns zu einer quellsprachlichen Aussage eine Berechnungsfunktion liefert:

S : Syntaktische_Aussage → (Zustand → Ziel_Wahrheitswert)

S ist ein Übersetzer, s ist ein Interpretierer. Das analoge Konzept existiert auch in der Analysis in Form des unbestimmten und des bestimmten Integrals. Die Definition erfolgt analog zu der von s über Funktionalgleichungen (das Ergebnis ist wiederum eine Funktion!) wie zum Beispiel:

S (¬ Faktor)(state) = `nicht` S(Faktor)(state)

Wir werden im weiteren immer die Interpretierer-Variante verwenden.

2.1.4 Konversion - Ableitung

Bislang haben wir nur die Möglichkeit, den Wert einer formalen Aussage dadurch zu bestimmen, daß wir die Semantik der Aussage berechnen und dann innerhalb des semantischen Modells deren Wert bestimmen. Darüber hinaus ist es häufig sinnvoll, auch über eine Möglichkeit zu verfügen, syntaktische Aussagen direkt zu manipulieren, ohne ihren Wert zu bestimmen und sei es nur, um die Lesbarkeit zu erhöhen. Die Notwendigkeit hierfür ergibt sich sowohl aus dem Wunsch, die Lesbarkeit von Aussagen zu erhöhen als auch aus dem Bestreben, den Wert einer Aussage zu bestimmen. Dies kann über das bereits eingeführte Verfahren, gerade dann wenn eine Aussage in vielen Variablen vorliegt, recht mühsam werden. Schon das Aufstellen der Wahrheitstafel erweist sich u.U. als ziemlich aufwendig (s. die im vorangegangenen Kapitel gezeigte Berechnungsvorschrift für die Zahl der Kombinationsmöglichkeiten).

Es gibt zwei Möglichkeiten zur syntaktischen Manipulation, *Konversion* und *Ableitung*. Sowohl bei der Konversion als auch bei der Ableitung kommt es ausschließlich auf die syntaktische Struktur der betrachteten Sätze an. Die Semantik spielt keine Rolle. Wichtig ist auch, daß weder in Konversions- noch in Ableitungsregeln konkrete Sätze (d.h. Aussagen im Fall der Aussagenlogik) auftauchen, sondern sogenannte *Aussagenvariablen*. Diese stehen für beliebige Sätze der betrachteten formalen Sprache. Wichtig ist nur ihre syntaktische Zusammensetzung. Ableitung ist der allgemeinere Begriff. Man kann alle Konversionsregeln auch als Ableitungsregeln schreiben. Konversionsregeln können im Gegensatz zu Ableitungsregeln in beide Richtungen gelesen werden.

2.1.4.1 Konversion

Konversion dient dem Zweck, eine Aussage in eine semantisch äquivalente umzuformen. Der Äquivalenzbegriff muß für jede Sprache eigens eingeführt werden.

Definition 2.1.5 (Äquivalenzen in der Aussagenlogik)
Zwei Aussagen A und B heißen genau dann äquivalent, wenn in allen Zuständen gilt:
s (A, state) = s (B, state)
'A ≡ B' ist dann eine Äquivalenz oder eine korrekte Konversionsregel.

Die Korrektheit einer Konversionsregel ist über ihre semantische Bedeutung natürlich nachzuweisen. Es erscheint sinnvoll, eine Vielzahl von Konversionsregeln einmal auf ihre Korrektheit zu prüfen, um das Vereinfachen von Aussagen zu erleichtern. Dabei können Konversionen auch als Optimierung über Aussagen der Quellsprache verstanden werden, bevor die Berechnung in einem konkreten Zustand stattfindet.

Diese Regeln in Abbildung 2.9 entsprechen jenen aus [Gries, 1981]. Die E_i sind Aussagenvariablen. Die Menge der Konversionsregeln sollte niemanden erschrecken. Viele haben Sie bereits schon häufig angewandt. Es ist jedoch für jede (formale) Sprache sinnvoll, sich der Regeln, deren man sich zur Manipulation von Sätzen bedient, zu vergewissern. **K4** ist benannt nach Augustus De Morgan, einem Mathematiker des neunzehnten Jahrhunderts, der zusammen mit Boole grundlegende Arbeiten auf dem Gebiet der Logik ausgeführt hat.

K1: Kommutativ-Gesetz	$E_1 \wedge E_2 \equiv E_2 \wedge E_1$
	$E_1 \vee E_2 \equiv E_2 \vee E_1$
	$E_1 \Leftrightarrow E_2 \equiv E_2 \Leftrightarrow E_1$
K2: Assoziativ-Gesetz	$E_1 \wedge (E_2 \wedge E_3) \equiv (E_1 \wedge E_2) \wedge E_3$
	$E_1 \vee (E_2 \vee E_3) \equiv (E_1 \vee E_2) \vee E_3$
K3: Distributiv-Gesetz	$E_1 \vee (E_2 \wedge E_3) \equiv (E_1 \vee E_2) \wedge (E_1 \vee E_3)$
	$E_1 \wedge (E_2 \vee E_3) \equiv (E_1 \wedge E_2) \vee (E_1 \wedge E_3)$
K4: Gesetz von De Morgan	$\neg(E_1 \wedge E_2) \equiv \neg E_1 \vee \neg E_2$
	$\neg(E_1 \vee E_2) \equiv \neg E_1 \wedge \neg E_2$
K5: Gesetz von der doppelten Negation	$\neg(\neg E_1) \equiv E_1$
K6: Gesetz vom ausgeschlossenen Dritten	$\neg E_1 \vee E_1 \equiv T$
K7: Gesetz vom Widerspruch	$\neg E_1 \wedge E_1 \equiv F$
K8: Gesetz von der Implikation	$E_1 \Rightarrow E_2 \equiv \neg E_1 \vee E_2$
	$T \Rightarrow E_2 \equiv E_2$
K9: Gesetz von der Äquivalenz	$E_1 \Leftrightarrow E_2 \equiv (E_1 \Rightarrow E_2) \wedge (E_2 \Rightarrow E_1)$
K10: Gesetz der Oder-Vereinfachung	$E_1 \vee E_1 \equiv E_1$
	$E_1 \vee T \equiv T$
	$E_1 \vee F \equiv E_1$
	$E_1 \vee (E_1 \wedge E_2) \equiv E_1$
K11: Gesetz der Und-Vereinfachung	$E_1 \wedge E_1 \equiv E_1$
	$E_1 \wedge T \equiv E_1$
	$E_1 \wedge F \equiv F$
	$E_1 \wedge (E_1 \vee E_2) \equiv E_1$
K12: Gesetz der Identität	$E_1 \equiv E_1$

Abbildung 2.9: *Konversionsregeln für die Aussagenlogik*

2.1.4.2 Ableitung

Im Gegensatz zu Konversion geht es bei der Ableitung darum, aus einer gegebenen Menge von Axiomen *syntaktisch* auf eine vorgegebene Aussage zu schließen (bottom-up) oder umgekehrt (top-down). Axiome sind dabei Voraussetzungen. In der Logik werden Aussagen, die aus Axiomen abgeleitet werden können, als *Theoreme* bezeichnet. Die Schlüsse, die dabei gezogen werden sollen, sind von der Art:

Wenn es in Tokio regnet, dann schütteln sich in Tokio die Hunde.
Es regnet in Tokio.
Schluß: Also schütteln sich in Tokio die Hunde.

oder

Wenn Katharina beim Weitsprung 8 m weit springt, dann wird sie zur Olympiade zugelassen.
Katharina wird nicht zur Olympiade zugelassen.
Schluß: Also ist Katharina nicht 8 m weit gesprungen.

Ein Ableitungssystem, auch *Kalkül* genannt, ist eine Methode des Schließens durch die Manipulation von Symbolen. Ableitungssysteme bestehen aus einer Menge von Regeln und einer Menge von Axiomen. Jede Regel besteht aus einer Menge von *Prämissen* und einer *Konklusion*. Eine Regel kann angewandt werden, wenn die Prämissen erfüllt sind. Diese Regeln werden häufig in der folgenden Art geschrieben:

$$\frac{P_1, \ldots, P_n}{K}$$

Dabei stehen die P_i für die Prämissen und K für die Konklusion. Manchmal besteht die Konklusion auch aus mehreren Aussagen. Diese werden dann durch Kommata getrennt. In unserem Beispiel wären 'Wenn es in Tokio regnet, dann schütteln sich in Tokio die Hunde.' und 'Es regnet in Tokio.' die Prämissen. Haben wir nun einen Satz, der seiner syntaktischen Struktur nach den P_i entspricht (sowohl die Prämissen als auch die Konklusion enthalten Aussagenvariablen), dann dürfen wir diesen, ohne daß sich die durch ihn ausgedrückte

Semantik ändert, durch die Konklusion ersetzen.

Hier werden Axiomenschemata angegeben. Diese unterscheiden sich von Axiomen dadurch, daß sie nur die bereits erwähnten Aussagenvariablen enthalten. Jede Aussagenvariable kann durch eine beliebige (syntaktisch korrekte) Aussage ersetzt werden. Hierbei muß jedoch darauf geachtet werden, daß gleiche Aussagenvariablen durch gleiche Aussagen zu ersetzten sind. Dies bedeutet natürlich auch, daß die Menge der Axiome unendlich viele Elemente enthält. Ein gültiges Axiom ist damit z.B.:

$$(a \vee b) \Rightarrow ((c \wedge d) \Rightarrow (a \vee b))$$

Mit der Schreibweise $\frac{P_1,\ldots,P_n}{K}$ beschreiben wir genaugenommen Regelschemata, da die Prämissen und die Konklusion, wie bereits erwähnt, nur Aussagenvariable enthalten. Die Anwendung eines Regelschemas auf konkrete Prämissen wird mit

$$A \vdash_R B$$

bezeichnet. Dabei ist A die Menge der konkreten Prämissen und B die konkrete Konklusion und R die Bezeichnung für die jeweilige Regel. $A \vdash_R B$ gilt, wenn die Aussagenvariablen der Regel R so durch Aussagen ersetzt werden können, daß die Prämissen in A enthalten sind und B die Konklusion ist.

Entsprechend dem Begriff der Äquivalenz benötigen wir hier den Begriff der Ableitbarkeit.

Definition 2.1.6 (Ableitbarkeit)

Seien S die Menge der Regeln eines Kalküls, A eine (endliche) Menge von Formeln der Logiksprache und B eine Formel der Logiksprache. Dann bezeichnen wir B als aus A ableitbar, wenn es eine Folge $R_1 \ldots R_n$ von Regeln aus S und eine eine Folge von Formeln $B_1 \ldots B_n$ mit $B_n = B$ gibt, derart, daß gilt

$$A_1 \vdash_{R_1} B_1, \ \ldots, \ A_j \vdash_{R_j} B_j, \ \ldots, \ A_n \vdash_{R_n} B$$

wobei A_j die Menge bezeichnet, welche die Axiome, die Formeln aus A und alle B_l mit $1 \leq l < k$ enthält. Falls A ausschließlich Axiome enthält, dann bezeichnen wir B als ableitbar (ohne den Zusatz 'aus A').

Damit die Semantik der betrachteten Sätze nicht in unzulässiger Weise verändert wird, muß sichergestellt werden, daß der Kalkül korrekt (*widerspruchsfrei*) ist, d.h. daß nur semantisch wahre Aussagen (Tautologien) abgeleitet werden können. Wünschenswert ist ebenfalls, daß alle Tautologien abgeleitet werden können. Ist dies der Fall, dann heißt der Kalkül *vollständig*. Der Begriff der Tautologie ist rein semantischer Natur und muß für jede Logik neu definiert werden. An dieser Stelle wollen wir ihn nur für die Aussagenlogik einführen.

Definition 2.1.7 (Tautologien in der Aussagenlogik)

Aussagen, für die in allen Zuständen gilt: s (Aussage, state) = **wahr** *heißen Tautologien oder allgemeingültige Aussagen.*

Die Frage nach dem Nachweis der Vollständigkeit und Widerspruchsfreiheit verschieben wir auf später. An dieser Stelle soll nur noch darauf hingewiesen werden, daß es nicht für jede formale Sprache vollständige und widerspruchsfreie Kalküle geben kann. Die Gründe hierfür sind theoretischer Natur und werden in Rahmen der theoretischen Informatik eingeführt. Als Beispiel für einen Kalkül soll einer für die Aussagenlogik dienen, der in Abbildung 2.10 (nach [Davis, 1989], [Gries, 1981] und [Loeckx & Sieber, 1987]) zusammengestellt ist.

Die ganz links stehenden Bezeichnungen sind die Namen, unter denen auf eine Regel Bezug genommen werden kann. 'o-I' steht für die Einführung und 'o-E' für die Elimination des Operators 'o'. Die Regel '$\Rightarrow$-E' ist auch bekannt unter dem Namen *modus ponens* (Abtrennungsregel) und wird zuweilen *mp* genannt. *Subst* bezeichnet die Regel für die Substitution. In der Substitutionsregeln ist mit $\mathcal{E}(E_1)$ gemeint, daß in der Aussage E_1 der (Teil)term p durch einen anderen ersetzt wird. Dabei ist $\mathcal{E}$ eine Funktion auf (Teil)termen von E_1. Ein Beispiel möge dies verdeutlichen:

Beispiel 2.1.8

Sei $\mathcal{E}(p) = d \vee p$, mit $E_1 = b \Rightarrow c$ und $E_2 = \neg b \vee c$.
Dann haben wir $\mathcal{E}(E_1) = d \vee (b \Rightarrow c)$ und $\mathcal{E}(E_2) = d \vee (\neg b \vee c)$
Dann erhalten wir durch Anwendung obiger Regel Subst aus der bekannten Äquivalenz $(b \Rightarrow c) \equiv (\neg b \vee c)$
die Äquivalenz $(d \vee (b \Rightarrow c)) \equiv (d \vee (\neg b \vee c))$

Axiomenschemata:

L1	$A \vee \neg A$
L2	$(A \wedge \neg A) \Rightarrow B$
L3	$(A \Rightarrow (B \wedge \neg B)) \;\Rightarrow\; \neg A$

Ableitungsregeln:

$\Rightarrow$-E	$\dfrac{E_1, E_1 \Rightarrow E_2}{E_2}$
$\Rightarrow$-I	$\dfrac{[E_1]E_2}{E_1 \Rightarrow E_2}$
$\wedge$-I	$\dfrac{E_1, ..., E_n}{E_1 \wedge ... \wedge E_n}$
$\wedge$-E	$\dfrac{E_1 \wedge ... \wedge E_n}{E_i}$
$\vee$-I	$\dfrac{E_i}{E_1 \vee ... \vee E_n}$
$\vee$-E	$\dfrac{E_1 \vee ... \vee E_n,\;\; E_1 \Rightarrow E,\;\; ...,\;\; E_n \Rightarrow E}{E}$
$\Leftrightarrow$-I	$\dfrac{E_1 \Rightarrow E2,\;\; E_2 \Rightarrow E1}{E_1 \Leftrightarrow E_2}$
$\Leftrightarrow$-E	$\dfrac{E_1 \Leftrightarrow E_2}{E_1 \Rightarrow E2,\;\; E_2 \Rightarrow E1}$
Subst	$\dfrac{E_1 \Leftrightarrow E_2}{\mathcal{E}(E_1) \Leftrightarrow \mathcal{E}(E_2),\;\; \mathcal{E}(E_2) \Leftrightarrow \mathcal{E}(E_1)}$

Abbildung 2.10: *Kalkül für die Aussagenlogik*

Mit den hier angegebenen Regeln (speziell $\vee$-I) kann z.B. aus der Annahme 'es regnet' geschlossen werden auf die Aussage 'es regnet $\vee$ die Sonne scheint'.

Die Regel '$\Rightarrow$-I' stellt eine Besonderheit in diesem Kalkül dar. Sie drückt aus, daß die Implikation $E_1 \Rightarrow E_2$ genau dem Gedanken "aus E_1 folgt E_2" entspricht. Sie verlangt als Prämisse nur, daß E_2 *unter der Annahme*, daß E_1 wahr ist – geschrieben als $[E_1]E_2$ – erfüllt sein muß. Hieraus darf man dann $E_1 \Rightarrow E_2$ schließen, ohne daß es hierfür noch irgendeiner Annahme bedarf. Da diese Regel eine gewisse formale Komplikation in den Kalkül mit hineinbringt, wird sie oft aus diesem hinausgenommen und durch eine Reihe von Axiomen ersetzt, welche Folgerungen dieser Regel sind:

L4	$A \Rightarrow A$
L5	$A \Rightarrow (B \Rightarrow A)$
L6	$((A \Rightarrow (B \Rightarrow C)) \wedge (A \Rightarrow B)) \;\Rightarrow\; (A \Rightarrow C)$
L7	$((A \Rightarrow B) \wedge (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)$
L8	$(A \Rightarrow B) \;\Rightarrow\; (\neg B \Rightarrow \neg A)$
L9	$(\neg(\neg A)) \Leftrightarrow A$
L10	$((A \vee B) \wedge (\neg A \vee C)) \;\Rightarrow\; B \vee C$

Als Beispiel zeigen wir, wie sich das Transitivitätsaxiom L7 mit $\Rightarrow$-I ohne die Axiome L4–L10 beweisen läßt:

Beispiel 2.1.9 *Wir nehmen an $(A \Rightarrow B) \wedge (B \Rightarrow C)$ und A sei erfüllt. Eine mögliche Ableitung ist:*

(1)	$(A \Rightarrow B) \wedge (B \Rightarrow C)$	*Annahme*
(2)	A	*Annahme*
(3)	$(A \Rightarrow B)$	*$\wedge$-E angewendet auf (1)*
(4)	$(B \Rightarrow C)$	*$\wedge$-E angewendet auf (1)*
(5)	B	*$\Rightarrow$-E angewendet auf (2) und (3)*
(6)	C	*$\Rightarrow$-E angewendet auf (5) und (4)*
(7)	$(A \Rightarrow C)$	*$\Rightarrow$-I angewendet auf (2) und (6) — (7) gilt ohne Annahme (2)*
(8)	$((A \Rightarrow B) \wedge (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)$	*$\Rightarrow$-I angewendet auf (1) und (7) — (8) gilt ohne jede Annahme*

Die Hinzunahme der Axiome L4–L10 läßt den Kalkül weniger systematisch erscheinen, macht ihn aber leichter handhabbar. Generell ist bei Ableitungssystemen zu berücksichtigen, daß es *den* Kalkül für eine formale Sprache nicht gibt. Regeln und Axiome können so gewählt werden, daß die gestellte Aufgabe bestmöglich

gelöst werden kann. Eine andere Aufgabenstellung kann einen anderen Kalkül sinnvoll erscheinen lassen.

Im Zusammenhang mit der Einführung der Programmiersprache Eiffel werden wir sehen, wie Programme und Aussagen über Programme als Logik aufgefaßt werden können und wie neue Aussagen abgeleitet werden können. Die Anwendung dessen ist die Verifikation von Programmen. Dort wird es dann durchaus Sinn machen, zur Abkürzung des Verfahrens die im Anhang angegebenen Regeln und Tautologien zu verwenden.

2.1.5 Zusammenhang zwischen Syntax, Ableitungssystem und Semantik

Um die zu Beginn des Kapitels erwähnte Kommunikation zu ermöglichen, müssen wir in der Lage sein, Aussagen innerhalb des syntaktischen Modells zu modifizieren, um zu entscheiden, ob zwei syntakische Aussagen äquivalent sind, oder ob eine andere Beziehung zwischen ihnen besteht. Diese Entscheidung wird bereits im Fall der Aussagenlogik sehr mühsam, wenn eine Vielzahl von Variablen in den zur Diskussion stehenden Aussagen auftritt (die Wertetabellen werden leicht sehr umfänglich). Kurzum, wir brauchen die Möglichkeit innerhalb des Modells zu rechnen. Genau diese wird durch ein *Ableitungssystem* zur Verfügung gestellt.

Das folgende Diagramm macht deutlich, daß für die Manipulation von syntaktischen Aussagen zwei Möglichkeiten bestehen. Entweder man transfomiert die syntaktische Aussage in eine semantische und 'berechnet' dann den Wert innerhalb der Semantik oder man transformiert die syntaktische Aussage zunächst via Ableitungssystem in eine vereinfachte. Anschließend wird diese vereinfachte Aussage dann in der Semantik interpretiert.

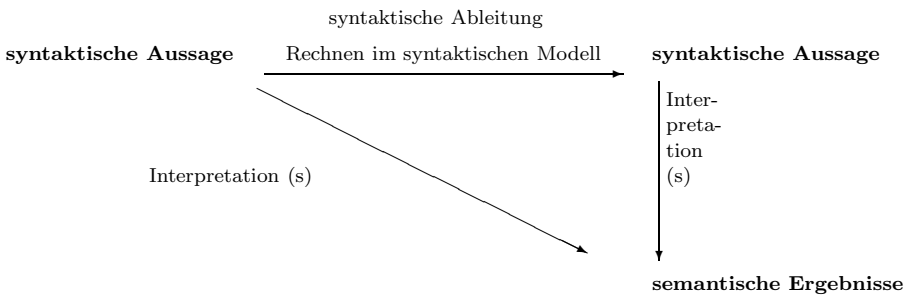


Abbildung 2.11: *Zusammenhang zwischen Syntax, Semantik und Ableitungssystem*

Bei den bislang gemachten Überlegungen zur Aussagenlogik, haben wir uns nur mit der Semantik der betrachteten logischen Operatoren ($\wedge$, $\vee$, ...) beschäftigt und gefordert, daß für die atomaren Aussagen etwas wie eine Zustandstabelle existiert. Mit der Einführung der Prädikatenlogik werden wir darüber hinaus fordern, daß für Funktionssymbole (z.B. $\sum$ in folgendem Beispiel) ein angemessenes semantisches Äquivalent existiert. Für die Bestimmung des Wahrheitswertes eines logischen Ausdrucks oder für seine Vereinfachung erweist es sich jedoch häufig als notwendig, zusätzlich zu den bisher erwähnten Informationen Wissen über die inhaltlichen Zusammenhänge zwischen Teilaussagen zu haben. Ein Beispiel möge dieses verdeutlichen. Da der erste Teil der Konjunktion den zweiten inhaltlich 'umfaßt', kann der Ausdruck

$$(x + \textstyle\sum_{i=0}^{5} i) < 100 \ \wedge \ (x + \sum_{i=3}^{5} i) < 200$$

mit dem entsprechenden Wissen aus der Arithmetik vereinfacht werden zu $x + \sum_{i=0}^{5} i < 100$.

Wir wissen nämlich, daß aus $(x + 15 < 100)$ geschlossen werden kann auf $(x + 15 < 100) \wedge (x + 12 < 200)$. Umformungen dieser Art sind in dem reinen Logikkalkül nicht mehr faßbar. An dieser Stelle kommt eine weitere 'Sorte' des Wissens (auch als *Domain-Wissen* bezeichnet) ins Spiel. Es handelt sich um Informationen über die unterliegenden Datenstrukturen, oder vielleicht besser, über die Eigenschaften der verwendeten Funktionssymbole und der atomaren Aussagen. Auch dieses Wissen kann in einem geeigneten Kalkül formalisiert werden.

In dem Kapitel über die Programmverifikation werden wir reichlich Gebrauch machen (müssen) von den durch das Domain-Wissen gerechtfertigten Transformationen. Weil diese häufig nicht durch einen expliziten syntaktischen Kalkül gerechtfertigt sind, ist es zweckmäßig, immer die verwendete 'Rechtfertigung' in Form eines geeigneten Lemmas anzugeben.

2.2 Logik als Spezifikationssprache

Bisher haben wir gezeigt, wie eine Sprache in ihrer Syntax und Semantik definiert werden kann und uns dabei besonders auf die Aussagenlogik konzentriert. In diesem Abschnitt werden wir zwei weitere Logiksprachen, nämlich die der Prädikatenlogik und eine für eine dreiwertige Logik vorstellen. Mit diesen können Sachverhalte beschrieben werden, die in der Aussagenlogik nicht mehr darstellbar sind. Zuvor wird jedoch an Hand der bereits bekannteren Aussagenlogik gezeigt, wie ein Zusammenhang zwischen einer Alltagssache und einer entsprechenden Beschreibung in einer formalen Sprache hergestellt werden kann.

2.2.1 Umsetzung natürlichsprachlicher Aussagen in solche der Logik

Der Leser möge immer mal wieder versuchen, scheinbar allgemein bekannte Tatbestände in eine eindeutige sprachliche Form zu bringen. Hierbei zeigt sich nämlich immer wieder, daß bereits eine Vielzahl von Schwierigkeiten mit der sprachlichen Beschreibung von Tätigkeiten und Objekten verbunden sind. Dies liegt häufig darin begründet, daß das zu beschreibende Problem nicht vollständig erkannt ist. Ein Beispiel hierfür ist das bereits erwähnte Sortierproblem. Wie bereits erwähnt, besteht ein Problem bei der Anwendung der Logik als Beschreibungssprache in der Umsetzung der natürlichsprachlichen Ausdrücke in solche der Logik.

Wir betrachten zunächst den Satz 'Falls es regnet, wird das Picknick abgesagt'. Den ganzen Satz bezeichnen wir mit *Picknick*. Bezeichne der Bezeichner r die Aussage 'es regnet' und der Bezeichner npc die Aussage 'das Picknick wird abgesagt'. Dann kann dieser Satz als $(r \Rightarrow npc)$ geschrieben werden. Hier ist zu berücksichtigen, daß die Semantik, die üblicherweise einer umgangsprachlichen 'Falls, dann'-Aussage unterlegt wird, für den Fall, daß es nicht regnet, von der Semantik des '$\Rightarrow$' abweicht. In der Umgangssprache wird mit obigem Satz üblicherweise ausgedrückt, daß das Picknick stattfindet, wenn es nicht regnet. Wir führen also für die implizite Bedeutung 'das Picknick findet statt' den Bezeichner pc ein. Bevor wir versuchen, diesen neuen Satz zu formalisieren, überlegen wir uns, was wir gerne hätten. Wir möchten eine Aussage, die folgender Wahrheitstafel genügt:

r	pc	*Picknick*
falsch	**falsch**	**falsch**
falsch	**wahr**	**wahr**
wahr	**falsch**	**wahr**
wahr	**wahr**	**falsch**

Ein zweiter Versuch der Umsetzung in einen Ausdruck der Aussagenlogik liefert: $(r \Rightarrow npc) \wedge (\neg r \Rightarrow pc)$. Um uns zu überzeugen, daß dies auch die von uns gemeinte Bedeutung trifft, bestimmen wir die Semantik unserer rein syntaktischen Aussage.

$s\ ((r \Rightarrow npc) \wedge (\neg r \Rightarrow pc),\ \text{state})$
$= s\ ((r \Rightarrow npc),\ \text{state})\ \texttt{und}\ s\ ((\neg r \Rightarrow pc),\ \text{state})$
$= s\ (r,\ \text{state})\ \texttt{impl}\ s\ (npc,\ \text{state})\ \texttt{und}\ s\ ((\neg r),\ \text{state})\ \texttt{impl}\ s\ (pc,\ \text{state})$
$= s\ (r,\ \text{state})\ \texttt{impl}\ s\ (npc,\ \text{state})\ \texttt{und}\ (\texttt{nicht}\ s\ (r,\ \text{state}))\ \texttt{impl}\ s\ (pc,\ \text{state}\ )$

Um den Wert dieser Aussage (mit *MPicknick* benannt) zu bestimmen, benötigen wir eine Zustandstabelle:

r	pc	$MPicknick$
falsch	**falsch**	**falsch**
falsch	**wahr**	**wahr**
wahr	**falsch**	**wahr**
wahr	**wahr**	**falsch**

Der Leser möge sich überlegen, daß dies seine Interpretation des umgangssprachlichen Satzes abdeckt. Beide Wertetabellen (die angestrebte und die erreichte) sind identisch, mithin formalisiert unser Ausdruck das Gewünschte. Zum Vergleich wäre es sinnvoll, die Wertetabelle für den Ausdruck $(r \Rightarrow npc)$ aufzustellen.

An dieser Stelle sei noch auf eine prinzipielle Beschränkung aller hier vorgestellten Logiken hingewiesen. Diese besteht in der für uns ganz wesentlichen Eigenschaft der formalen Logik, dem *Extensionalitätsprinzip*. Dieses besagt, daß der Wahrheitswert einer Aussage nur von der formalen Struktur der Aussage und den Wahrheitswerten der Teilaussagen abhängt, nicht aber von inhaltlichen Bezügen zwischen den Teilaussagen. Eine Konsequenz dessen ist, daß der Satz

Lessing schrieb 'Minna von Barnhelm', während sich Preußen und Österreich im siebenjährigen Krieg bekämpften.

innerhalb der hier vorgestellten Logik nicht formulierbar ist. Denn wenn er formulierbar wäre, müßte die Beurteilung des Wahrheitsgehaltes der Gesamtenaussage nur von dem Wahrheitsgehalt der beiden Teilaussagen (Lessing schrieb 'Minna von Barnhelm' und Preußen und Österreich bekämpften sich im siebenjährigen Krieg) abhängen. Der Operator während zwingt jedoch dazu, zu prüfen, ob ein (zeitlicher) Zusammenhang zwischen dem schriftstellerischen Tun und dem siebenjährigen Krieg besteht. Ersetzt man nämlich die wahre Teilaussage Lessing schrieb 'Minna von Barnhelm' durch die ebenfalls wahre Teilaussage Lessing schrieb 'Nathan der Weise', so wird die Gesamtaussage falsch. Das Beispiel entstammt [Hermes, 1972].

2.2.2 Prädikatenlogik

Die Aussagenlogik geht von Grundaussagen aus, die entweder wahr oder falsch sein können, sagt aber nichts über die innere Struktur dieser Aussagen aus. Umfangreichere mathematische Theorien lassen sich aber nur darstellen, wenn auch die innere Struktur von Aussagen beschrieben werden kann.

Mit den Mitteln der Aussagenlogik ist es z.B. nicht möglich, die Aussage Alle durch 4 ganzzahlig teilbaren natürlichen Zahlen sind auch durch 2 teilbar weiter zu analysieren. Sie muß dort als atomare Aussage stehen bleiben. Um diese Aussage weiter zu zerlegen, würde man gemeinhin eine Konjunktion über alle Elemente angeben, für die die Aussage gelten soll. D.h. wir würden versuchen, obige Aussage umzuschreiben in 4 ist durch 4 teilbar und 4 ist durch 2 teilbar und 8 ist durch 4 teilbar 8 ist durch 2 teilbar und .... Dies entspricht unserem Verständnis von 'alle'. In diesem Fall ist dies jedoch nicht möglich. Wir haben nur Aussagen mit endlich vielen Konjunktionen eingeführt, benötigen jedoch eine unendliche Anzahl, da die Menge, über deren Elemente eine Aussage getroffen wird, unendlich viele dieser Elemente hat. Mithin enthält die Aussage, die wir durch diese Umformung erhalten, unendlich viele Konjunktionen. Wir haben also keine Möglichkeit, auf Grund der Werte der atomaren Aussagen (z.B. 4 ist durch 4 teilbar) den Wahrheitswert der gesamten Aussage zu bestimmen; wir werden nämlich niemals fertig. Dieses Problem tritt bei der Beschreibung aller Zusammenhänge auf, bei denen Aussagen über einen unendlich großen Wertebereich gemacht werden, z.B. auch bei allen mathematischen Funktionen, die über den reellen Zahlen definiert werden. Die bekannnte Schreibweise für die Fakultätsfunktion

$$\mathrm{f(x)} = \begin{cases} 1 & , falls\ x = 0\ oder\ x = 1 \\ x * f(x-1) & , sonst \end{cases}$$

ist eine implizite All-Quantifizierung über alle Werte, die x annehmen kann.

Die Prädikatenlogik führt uns hier einen Schritt weiter. Hier werden Bezeichner eingeführt (im folgenden auch als Variable benannt), die nicht nur für Wahrheitswerte stehen können, sondern auch für andere Objekte (ganze Zahlen, natürliche Zahlen, Blumensorten, Studenten usw.). Aussagen werden zweifach verallgemeinert:

1. In einer Aussage kann eine Aussagenvariable durch einen beliebigen Ausdruck ersetzt werden, der einen Wahrheitswert liefert (z.B. a $\geq$ b)

2. Es werden *Quantoren* eingeführt, nämlich ∃ und ∀. Ersterer bedeutet 'es gibt', der zweite 'für alle'. Diese werden auf Variable angewendet, die Elemente eines gewissen Bereichs bezeichnen.

Mit Quantoren wird es möglich, Aussagen über die Elemente der Grundmengen der Variablen zu treffen. Steht beispielsweise Blume für ein nicht weiter beschriebenes Element aus der Grundmenge aller Blumen, dann ist Blume eine Variable und die Aussage ∀Blume:{einjährig} . **A** bedeutet, daß für alle Elemente aus der Blumenmenge, die einjährig sind (also zum Bereich {einjährig} gehören), die Aussage **A** gilt. Diese erweiterten Aussagen werden als *Sätze* bezeichnet. Die bereits bekannte Aussagenlogik ist ein Teilbereich der Prädikatenlogik.

Im folgenden werden wir die Syntax und die Semantik der Prädikatenlogik erster Stufe und einen entsprechenden Kalkül vorstellen. Wir haben uns bemüht, Syntax und Semantik vollständig nach den bisher dargestellten Konzepten zu definieren. Damit wird die Möglichkeit geboten diese, quasi nebenbei, mitzuvertiefen.

2.2.3 Syntax der Prädikatenlogik

Die Abbildung 2.12 beschreibt die Syntax der Prädikatenlogik.

Startsymbol: Satz

Alphabet:

Grundmengen: Prädikat_Symbol, Konstantes_Prädikat, Funktions_Symbol, Konstante, Variable, Bereich

Sonstige Symbole ∧, ∨, ⇒, ⇔, ∀, ∃, . , T, F,), (, =, ,

Regeln:

Satz	::=	Atomaraussage
	|	¬ Satz | (Satz ∧ Satz) | (Satz ∨ Satz)
	|	(Satz ⇒ Satz) | (Satz ⇔ Satz)
	|	(∀ Variable : Bereich . Satz)
	|	(∃ Variable : Bereich . Satz)
Atomaraussage	::=	*T* | *F* | (Term = Term)
	|	Konstantes_Prädikat
	|	Prädikat_Symbol (Termlist)
Termlist	::=	Term | Term , Termlist
Term	::=	Konstante | Variable
	|	Funktions_Symbol (Termlist)

Kontextbedingungen:

Genau dann, wenn $t_1, t_2, ..., t_n$ Terme sind und p ein n-stelliges Element aus Prädikat_Symbol ist, ist auch $p(t_1, t_2, ..., t_n)$ ein Satz.

Genau dann wenn $t_1, t_2, ..., t_n$ Terme sind und f ein n-stelliges Element aus Funktions_Symbol ist, ist auch $f(t_1, t_2, ..., t_n)$ ein Term.

Bereich ist immer eine Menge, deren Elemente *Variable* als Werte annehmen kann. Daher ist der Unterschied zwischen Wert und Variable zu beachten, d.h. kein Element aus Variable darf als Name in dom(Bereich) vorkommen

Abbildung 2.12: *Syntax der Prädikatenlogik*

Wir vereinbaren, daß aus Gründen der Lesbarkeit *Klammern* gemäß der Wertigkeit der Operatoren *weggelassen werden können*. Dabei gilt, daß der Operator, der nach der folgenden Tabelle (aus [Loeckx & Sieber, 1987]) die geringsten Priorität hat, am schwächsten bindet:

Operator	Priorität
$\neg, \exists, \forall$	4 (höchste Priorität)
$\wedge$	3
$\vee$	2
$\Rightarrow, \Leftrightarrow$	1

Weiter vereinbaren wir, daß die *äußersten Klammern ebenfalls wegfallen können*. Mithin schreiben wir, wie bei der Aussagenlogik, $A \wedge \neg B$ statt $(A \wedge (\neg B))$. Wir verzichten darauf eine Grammatik anzugeben, die diese Priorisierung durch ihre Struktur deutlich macht, da wir für die später folgende Semantikbeschreibung gerne eine möglichst einfache Grammatik, sprich eine mit wenigen Regeln, zu Grund legen möchten. Den üblichen Konventionen entsprechend schreiben wir zweistellige Funktions- und Prädikatssymbole auch in *Infix-Notation*, d.h. a+b statt +(a,b) und a=b statt =(a,b).

Auf eine genaue Beschreibung der möglichen *Ausdrücke für den Bereich* verzichten wir hier. Üblich sind eine *Mengenbezeichnung*, die konkreten Angabe einer Menge durch *Aufzählung* oder die *Angabe eines Bereiches* wie [Rot...Blau] oder [1..7] aus einer Menge, auf der eine Ordnungsrelation 'größer' definiert ist. Wir vereinbaren, daß eine durch [anf .. ende] definierte Menge genau dann leer ist, wenn anf größer als ende ist.

Mit der hier vorgestellten Prädikatenlogik ist es auch möglich, darzustellen, daß 'Peter ist der siebte Schüler im Alphabet' gilt. Um derartige Aussagen beschreiben zu können, muß man einen Formalismus haben um den 'siebten' bezeichnen zu können, was aber mit Funktions- und Prädikatssymbolen kein Problem ist. Wir schreiben einfach: $\mathsf{Alphabet(7)} = \mathsf{Peter}$. Darüber hinaus haben wir eine Handhabe, Aussagen über alle Schüler zu machen, denn wir können schreiben: $\exists \mathsf{x} : \mathsf{1..Anz\_Schüler} \,.\, ((\mathsf{Alphabet(x)} = \mathsf{Peter}) \wedge (\mathsf{x} > \mathsf{5}))$

2.2.4 Semantik der Prädikatenlogik

Bei der Aussagenlogik hatten wir den Begriff des Zustandes eingeführt, um Belegungen von Bezeichnern mit (Warheits-)werten beschreiben zu können. Diesen benötigen wir auch hier wieder, allerdingsin einem wesentlich erweiterten Sinne, da wir neben Bezeichnern für Aussagen (in der jeztigen Terminologie konstante Prädikate) noch Variablen für Elemente einer Grundmenge, Konstanten, Funktionssymbole und Prädikatssymbole eingeführt haben. Wir brauchen daher eine Abbildung, die all diese Quelloperatoren in Zieloperatoren (Elemente der Grundmenge, Funktionen und Prädikate über der Grundmenge) abbildet. Üblicherweise wird diese Abbildung in der Logik als *Interpretation* bezeichnet.

In den meisten praktischen Anwendungen der Prädikatenlogik hat es sich eingebürgert, diese Interpretation aufzutrennen in eine Bedeutung $\mathcal{B}$ für Funktions- und Prädikatensymbole (incl. der Konstanten), die einmal fixiert wird, und eine Funktion, welche die Zustände der Variablen notiert. Wir erhalten:

Zustand	:	Variable $\rightarrow$ Grundmenge
$\mathcal{B}$	:	Prädikat_Symbol $\rightarrow$ Ziel_Prädikat
		| Konstantes_Prädikat $\rightarrow$ Ziel_Wahrheitswert
		| Funktions_Symbol $\rightarrow$ Ziel_Funktion
		| Konstante $\rightarrow$ Ziel_Konstante

Die Definitionsgleichungen für die Semantik der Prädikatenlogik sind eine Erweiterung der Semantik der Aussagenlogik. Für die Funktions- und Prädikatensymbole muß zunächst eine Bedeutung $\mathcal{B}$ festgelegt werden, bevor über den Wert von Sätzen, die solche Symbole enthalten, gesprochen werden kann. Abbildung 2.13 faßt alle Gleichungen zusammen.

Die '`if then` else'-Konstruktion ist ein Ausdruck der Metasprache und sagt uns nur, wie die Sätze der Zielsprache zu bilden sind. Aus Gründen des Verständnisses verzichten wir hier auf eine genauere Definition und gehen davon aus, daß diese Konstruktion selbsterklärend ist. Es ist jedoch zu diesem intuitiven Verständnis hinzuzufügen, daß jeweils nur der Zweig der Fallunterscheidung berücksichtigt wird, dessen Bedingung erfüllt ist. Eine Auswertung des jeweils anderen findet nicht statt.

s (T, state) = **wahr**

s (F, state) = **falsch**

s (Prädikat_Symbol (termlist), state) = $\mathcal{B}$(Prädikat_Symbol) ($s$list (termlist, state))

s (Konstantes_Prädikat, state) = $\mathcal{B}$(Konstantes_Prädikat)

s ($\text{Term}_1 = \text{Term}_2$, state) = `if` s (Term_1, state) `=` s (Term_2, state)
`then` **wahr** `else` **falsch**

s ($(\neg$ Satz), state) = (`nicht` s (Satz, state))

s ($(\text{Satz}_1 \wedge$ Satz), state) = (s (Satz_1, state) `und` s (Satz_2, state))

s ($(\text{Satz}_1 \vee$ Satz), state) = (s (Satz_1, state) `oder` s (Satz_2, state))

s ($(\text{Satz}_1 \Rightarrow$ Satz), state) = (s (Satz_1, state) `impl` s (Satz_2, state))

s ($(\text{Satz}_1 \Leftrightarrow$ Satz), state) = (s (Satz_1, state) `gleich` s (Satz_2, state))

s ($(\forall$ Variable : Bereich `.` Satz), state) = `if` Bereich $= \{\}$ `then` **wahr**
`else` `let` x $\in$ Bereich `in`
s (Satz, state + [Variable $\mapsto$ x]) `und`
s ($(\forall$ Variable : Bereich-$\{$x$\}$ `.` Satz), state)

s ($(\exists$ Variable : Bereich `.` Satz), state) = `if` Bereich $= \{\}$ `then` **falsch**
`else` `let` x $\in$ Bereich `in`
s (Satz, state + [Variable $\mapsto$ x]) `oder`
s ($(\exists$ Variable : Bereich-$\{$x$\}$ `.` Satz), state)

s (Variable, state) = state(Variable)

s (Konstante, state) = $\mathcal{B}$(Konstante)

s (Funktions_Symbol (termlist), state) = $\mathcal{B}$(Funktions_Symbol) ($s$list (termlist, state))

$s$list (termlist, state) = `if` `einelementig`(termlist)
`then` s (termlist, state)
`else` s (`head`(termlist), state) `cons`
$s$list (`tail`(termlist), state))

Abbildung 2.13: *Semantik prädikatenlogischer Formeln mit endlichen Bereichen*

Analog zu dem '`if then else`'-Konstrukt, wird hier eine weitere, die Bildung der Zielsprachensätze betreffende Konstruktion verwandt, das 'let'-Konstrukt. '`let x = Ausdruck1 in Ausdruck2`' bedeutet, daß *einmal* der Wert von Ausdruck1 bestimmt wird und dann alle Vorkommen von x in Ausdruck2 durch den Wert von Ausdruck1 ersetzt werden. Erst dann erfolgt die Bestimmung von Ausdruck2.

Eine *Liste* ist eine geordnete Reihung ihrer Elemente, in der im Gegensatz zu einer Menge Elemente auch mehrfach vorkommen dürfen. Die nicht weiter definierte Funktion `tail` angewandt auf eine Liste von Termen liefert uns die Liste der Terme, die sich aus der Ausgangsliste ohne das erste Element ergeben. Das erste Element (der "Kopf") wird von der Funktion `head` geliefert. Die Funktion `einelementig` testet, ob eine Liste nur ein Element hat und die (Infix-)Funktion `cons` hängt ein Element vor den Anfang einer Liste. Der Ausdruck state + [Variable $\mapsto$ e] bedeutet, daß sich die Funktionstabelle für diesen Ausdruck von der des Zustandes state nur an der Stelle Variable unterscheidet. Sie hat hier den Eintrag Variable $\mapsto$ e.

Die auf den ersten Blick vielleicht aufwendige Konstruktion im Fall des $\exists$ und des $\forall$ Operators dient nur dazu, die Elemente aus der Menge Bereich[6] aufzuzählen. '`.`' trennt dabei die Festlegung der durch den Quantor gebundenen Variablen und die Bereichsangabe von der eigentlichen Aussage. Man mache sich klar, daß im Fall endlicher Bereiche die Bedeutung der beiden Operatoren durch eine (unter Umständen ziemlich lange)

[6] An dieser Stelle sei noch einmal deutlich herausgehoben, daß im Rahmen dieser Vorlesung nur Quantoren mit *endlichen Bereichen* benutzt werden. Die allgemeine Logik erklärt Quantoren natürlich auch für unendliche Bereiche wie den der natürlichen Zahlen. Dabei ändert sich die Syntax nicht im geringsten, aber die obige, konstruktive Definition der Semantik muß durch eine weniger konstruktive ersetzt werden, welche Zielsprachenoperatoren "`für alle`" und "`es gibt`" benutzt. Die Bedeutung solcher Zielsprachenoperatoren kann allerdings wesentlich weniger klar angegeben werden als z.B. die von `und` und ist deshalb in der Fachwelt durchaus umstritten.

Kette von $\vee$ bzw. $\wedge$ Operationen realisiert werden kann. Erst wenn Bereich Mengen enhält, die nicht mehr endlich sind, erhalten die beiden Operatoren eine, über die Aussagenlogik hinausgehende Bedeutung. Man mache sich jedoch klar, daß obige, konstruktive Semantikbeschreibung bei unendlichen Bereichen nicht immer terminiert. D.h. im Fall des All-Operators wird die Berechnung der Semantik genau dann nicht abbrechen, wenn der Bereich unendlich ist und das Prädikat für alle Belegungen wahr liefert.

In der Aussagenlogik galt das Prinzip, daß wir eine Variable mit einem Wert belegen dürfen, solange wir alle Vorkommen dieser Variablen konsistent ersetzen. In der Prädikatenlogik gilt dieses nun nicht mehr. Das Prädikat $\mathbf{A} \Rightarrow (\exists \mathbf{x} : \mathbf{1..15} \,\mathbf{.}\, \mathbf{A})$ hat völlig unterschiedliche Bedeutungen, abhängig davon, ob **x** in **A** vorkommt. Falls x in A nicht vorkommt, dann ist dieses Prädikat immer wahr, unabhängig von der sonstigen Struktur von A (es kann dann auf das Axiom $A \Rightarrow A$ der Aussagenlogik zurückgeführt werden). Sei A jedoch $x > 17$, dann spielt die Belegung von x in A eine entscheidende Rolle, es gilt nämlich für die Belegung von x mit 13 (state = [..., $x \mapsto 13$]) , daß $13 > 17 \Rightarrow (\exists x : 1..15 \,.\, x > 17)$ mit unserer Semantik **wahr** liefert, während wir für die Belegung von x mit 18 **falsch** erhalten. Wie man deutlich sieht, wird das x in dem 'zweiten' A unseres Prädikates durch den Quantor gebunden, es ist also lediglich lokal 'bekannt'. Damit wird die Belegung für x in unserem Prädikat nicht für das x in dem zweiten A, das 'lokale' x wirksam. Falls A andererseits die Form $\exists x :$ Bereich . B hat, dann ergibt sich die Bedeutung des Gesamtprädikates wiederum unabhängig von einer Belegung von x, wir haben wiederum eine Tautologie. Genauer gesagt bedeutet dies, daß wir zwei unterschiedliche Prädikate haben, abhängig von dem Vorkommen von x in A. Derartige lokale Vorkommen haben wir natürlich auch bei Funktionssymbolen wie $\sum_{i=1}^{n}$, $\int_{i=1}^{n}$, $\prod_{i=1}^{n}$ und ähnlichen anderen. Auch hier ist i nur ein lokaler Parameter.

Die obige Semantikfunktion realisiert also die Vorstellung, daß Variable, die durch einen Quantor oder ein Funktionssymbol 'gebunden' werden, anders zu behandeln sind, als 'freie' Variable.

Sei x eine Variable und A ein Prädikat, dann wird x als *gebunden* bezeichnet, falls es in einem Unterausdruck von A der Form $\exists x : Bereich \,.\, A$ bzw. $\forall x : Bereich \,.\, A$ auftritt, sonst bezeichnen wir das Vorkommen als *frei*. Entsprechendes gilt auch für die Bezeichner, mit denen Bereiche bei Funktionssymbolen beschrieben werden (also z.B. i bei $\sum_{i=1}^{100}$). Damit ist auch klar, daß x in $x > 17 \Rightarrow (\exists x : 1..15 \,.\, x > 17)$ sowohl frei als auch gebunden vorkommt. Da in obiger Definition nicht ganz klar ist, was ein Unterausdruck ist, hier eine inhaltlich identische, aber präzisere Definition.

Definition 2.2.1 (Freies und gebundenes Vorkommen von Variablen)
Seien x eine Variable, A und B Prädikate, f ein Funktions_Symbol, p ein Prädikat_Symbol und $t_1, ..., t_n$ *Terme.*

Das freie Auftreten von x in einem Ausdruck ist durch die folgenden Bedingungen definiert.

- *jedes Vorkommen von x in einer Atomaraussage ist frei*
- *x ist frei in f($x_1, ..., x_n$), wenn f entweder ein Funktionssymbol ohne Bereichsangabe ist, oder x in der Bereichbeschreibung nicht entsprechend verwendet wird* [7]
- *x ist frei in f($t_1, ..., t_n$), falls x in mindestens einem t_i frei ist*
- *x ist frei in p($t_1, ..., t_n$) oder $t_1 = t_2$, falls x in mindestens einem t_i frei ist*
- *x ist frei in $\neg$ A, wenn x in A frei ist*
- *x ist frei in A op B, wenn x in A oder in B frei ist (op* aus $\wedge, \vee, \Rightarrow, \Leftrightarrow$ *)*
- *x ist frei in $\exists y :$* Bereich . *A bzw. $\forall y :$* Bereich . *A, wenn x in A frei vorkommt und x verschieden y ist*

$P_{x_1,...,x_n}$ zeigt an, daß P ein Prädikat mit den freien Variablen $x_1, ..., x_n$ ist.

Das gebundene Auftreten von x in einem Ausdruck ist durch die folgenden Bedingungen definiert.

- *kein Vorkommen von x in einer Atomaraussage ist gebunden*
- *x ist gebunden in f($x_1, ..., x_n$), wenn f ein Funktionssymbol mit Bereichsangabe ist und x in der Bereichbeschreibung entsprechend verwendet wird*
- *x ist gebunden in f($t_1, ..., t_n$), falls x in mindestens einem t_i gebunden ist*

[7] $\sum_{i=1}^{n}$ ist in diesem Sinne ein Funktionssymbol mit Bereichsangabe, und i ist gebunden, während n frei ist.

- *x ist gebunden in* $p(\mathrm{t}_1, ..., \mathrm{t}_n)$ *oder* $\mathrm{t}_1 = \mathrm{t}_2$*, falls x in mindestens einem* t_i *gebunden ist*
- *x ist gebunden in* $\neg$ *A, wenn x in A gebunden ist*
- *x ist gebunden in A op B, wenn x in A oder in B gebunden ist (op* aus $\wedge, \vee, \Rightarrow, \Leftrightarrow$*)*
- *x ist gebunden in* $\exists y$: Bereich **.** *A bzw.* $\forall y$: Bereich **.** *A, wenn x in A gebunden vorkommt oder x identisch mit y ist*

Definition 2.2.1 führt lediglich Namen (frei, gebunden) für einen Sachverhalt ein, der über die Semantikdefinition bereits gegeben ist. Man mache sich klar, daß die Anwendung der Definition auf ein konkretes Prädikat eine rein syntaktische Aktion ist. Mit ihr können wir nun den *Ersetzungsprozeß* von Variablen beschreiben.

Definition 2.2.2 (Ersetzung von Variablen durch Terme)
Sei x eine Variable, t ein Term und A ein Ausdruck. Dann bezeichnet $\underline{\mathrm{A}\left[\begin{smallmatrix}\mathrm{t}\\ \mathrm{x}\end{smallmatrix}\right]}$ *den Ausdruck der dadurch entsteht, daß alle* freien *Vorkommen von x durch t textuell ersetzt werden.*

Diese Definitionen ermöglichen es, einen Ableitungskalkül für die Prädikatenlogik präzise zu beschreiben.

2.2.5 Ableitungskalkül für die Prädikatenlogik

Axiomenschemata:

A1–A3	Axiome L1–L3 der Aussagenlogik
A4	$((...(x_1 = y_1) \wedge ...)...) \wedge (x_n = y_n) \;\Rightarrow\; (p(x_1, ..., x_n) \Leftrightarrow p(y_1, ..., y_n))$ für alle n-stelligen Prädikat_Symbole p ($\mathrm{n} \geq 1$)
A5	$((...(x_1 = y_1) \wedge ...)...) \wedge (x_n = y_n) \;\Rightarrow\; (f(x_1, ..., x_n) = f(y_1, ..., y_n))$ für alle n-stelligen Funktions_Symbole f ($\mathrm{n} \geq 1$)

Ableitungsregeln:

R1–R9		Regeln $\Rightarrow$-E, $\Rightarrow$-I, $\wedge$-I, $\wedge$-E, $\vee$-I, $\vee$-E, $\Leftrightarrow$-I, $\Leftrightarrow$-E, Subst der Aussagenlogik	
R10	$\exists$-I	$\dfrac{A\left[\begin{smallmatrix}w\\ x\end{smallmatrix}\right]}{\exists x : \text{Bereich} \,.\, A}$	falls w $\in$ Bereich
R11	$\exists$-E	$\dfrac{\exists x : \text{Bereich} \,.\, A_1,\ A_1 \Rightarrow A_2}{A_2}$	falls x in A_2 nicht frei vorkommt
R12	$\forall$-I	$\dfrac{A}{\forall x : \text{Bereich} \,.\, A}$	
R13	$\forall$-E	$\dfrac{\forall x : \text{Bereich} \,.\, A}{A\left[\begin{smallmatrix}w\\ x\end{smallmatrix}\right]}$	falls w $\in$ Bereich
R14		$\dfrac{(\exists x : \text{Bereich} - \{w\} \,.\, A) \vee ((x = w) \Rightarrow A)}{\exists x : \text{Bereich} \,.\, A}$	falls w $\in$ Bereich
R15		$\dfrac{\exists x : \text{Bereich} \,.\, A}{(\exists x : \text{Bereich} - \{w\} \,.\, A) \vee ((x = w) \Rightarrow A)}$	falls w $\in$ Bereich
R16		$\dfrac{(\forall x : \text{Bereich} - \{w\} \,.\, A) \wedge ((x = w) \Rightarrow A)}{\forall x : \text{Bereich} \,.\, A}$	falls w $\in$ Bereich
R17		$\dfrac{\forall x : \text{Bereich} \,.\, A}{(\forall x : \text{Bereich} - \{w\} \,.\, A) \wedge ((x = w) \Rightarrow A)}$	falls w $\in$ Bereich
R18–R19		$\dfrac{A\left[\begin{smallmatrix}w\\ x\end{smallmatrix}\right]}{(x = w) \Rightarrow A} \qquad \dfrac{(x = w) \Rightarrow A}{A\left[\begin{smallmatrix}w\\ x\end{smallmatrix}\right]}$	

Abbildung 2.14: *Ableitungskalkül für die Prädikatenlogik 1. Stufe*

Die Zusammenstellung in Abbildung 2.14 ist [Gries, 1981] entnommen und auf unsere Syntax angepaßt. Variablen werden mit kleinen Buchstaben bezeichnet, Aussagenvariablen mit großen. Alle Variablen sind implizit *allquantifiziert*, d.h. die folgenden Axiomenschemata gelten unabhängig von den verwendeten Variablennamen. Die Regeln R14–R19 sind nicht notwendig, da sie sich aus den anderen ableiten lassen. Da unser Interesse

aber nicht in einem minimalen Kalkül liegt, haben wir ihn um Regeln ergänzt, die eine Anwendung auf unsere Erfordernisse erleichtern. So ermöglicht beispielsweise R16 den Schluß von

$(\forall\ x : 1..100\ .\ x>0)$ auf

$(\forall\ x : 1..100 - \{17\}\ .\ x>0)\ \wedge\ ((x=17)\ \Rightarrow\ (x>0))$

um ein einzelnes Element (17) gesondert zu betrachten. Mit R18 erhalten wir

$(\forall\ x : 1..100 - \{17\}\ .\ x>0)\ \wedge\ (17>0)$

Da die Aussagenlogik einen Teilbereich der Prädikatenlogik darstellt, können alle Regeln und Axiome, die in dem Kalkül für die Aussagenlogik angegeben waren, auch hier verwendet werden (natürlich nur unter Berücksichtigung der Bindungsregeln für Variable).

2.2.6 Dreiwertige Logik

In der Prädikatenlogik gilt das Prinzip des '*tertium non datur*', d.h. es gibt nur zwei mögliche Wahrheitswerte. Etwas ist entweder falsch, oder aber es ist wahr. Im Zusammenhang mit der Spezifikation von Programmen gibt es mehrere Gründe, anstelle einer zweiwertigen Logik eine auf drei Wahrheitswerten basierende zu verwenden. Zum einen muß im Zusammenhang mit Programmen beschrieben werden, was 'geschieht', wenn eine Funktion einen undefinierten Wert annimmt. Zum anderen erscheint es häufig sinnvoll, in bestimmten Situationen *explizit* angeben zukönnen, daß z.B. der Wert einer Funktion in dieser Situation ohne Relevanz ist. Es lassen sich sicher weitere mögliche Interpretationen eines dritten Wahrheitswertes finden. Darüber hinaus gibt es auch Anwendungen für Logiken mit unendlich vielen Wahrheitswerten. Diese werden dann üblicherweise als Wahrscheinlichkeiten für das Eintreten bestimmter Aussagen interpretiert.

Wir werden an dieser Stelle eine dreiwertige Logik, bezeichnet mit DREIWERT, einführen, die sich sowohl an die Spezifikationssprache von Eiffel als auch die Aussagenlogik anlehnt. Geeignete analoge Erweiterungen der Prädikatenlogik sollten als Übung gemacht werden.

Startsymbol:	Aussage		
Alphabet:			
Grundmengen	Bezeichner		
Sonstige Symbole	$\neg, \wedge, \vee, \overline{\vee}, \overline{\wedge}, \Rightarrow, \Leftrightarrow, (,), T, F, U$		
Syntax:	Aussage	::=	Imp-Ausdruck | Aussage $\Leftrightarrow$ Imp-Ausdruck
	Imp-Ausdruck	::=	Ausdruck | Imp-Ausdruck $\Rightarrow$ Ausdruck
	Ausdruck	::=	Term | Ausdruck $\vee$ Term | Ausdruck $\overline{\vee}$ Term
	Term	::=	Faktor | Term $\wedge$ Faktor | Term $\overline{\wedge}$ Faktor
	Faktor	::=	$\neg$ Faktor | (Aussage) | T | F | U | Bezeichner

Abbildung 2.15: *Syntax von DREIWERT (Klammerung nur soweit notwendig)*

Um die Semantik von DREIWERT festlegen zu können, müssen wir auch unsere Semantiksprache um ein undefined erweitern. Damit erhalten wir dann die folgende Wahrheitstafel für die Metasprache. Jede Zeile zeigt für eine mögliche Wertekombination von b und c den Wert der fünf zielsprachlichen Operationen.

b	c	(`nicht` b)	(b `und` c)	(b `oder` c)	(b `impl` c)	(b `gleich` c)
falsch	**falsch**	**wahr**	**falsch**	**falsch**	**wahr**	**wahr**
falsch	**wahr**	**wahr**	**falsch**	**wahr**	**wahr**	**falsch**
falsch	**undef**	**wahr**	**falsch**	**undef**	**wahr**	**undef**
wahr	**falsch**	**falsch**	**falsch**	**wahr**	**falsch**	**falsch**
wahr	**wahr**	**falsch**	**wahr**	**wahr**	**wahr**	**wahr**
wahr	**undef**	**falsch**	**undef**	**wahr**	**undef**	**undef**
undef	**wahr**	**undef**	**undef**	**undef**	**undef**	**undef**
undef	**falsch**	**undef**	**undef**	**undef**	**undef**	**undef**
undef	**undef**	**undef**	**undef**	**undef**	**undef**	**undef**

Der Wert einer Aussage mit mehreren Operatoren wird wie gehabt dadurch bestimmt, daß obige Tabelle solange auf die Teilaussagen angewandt wird, bis die gesamte Aussage auf **wahr** **falsch** oder **undef** reduziert ist. Damit können wir nun die Semantik von DREIWERT definieren:

s (Bezeichner, state)	=	state(Bezeichner)
s (T, state)	=	**wahr**
s (F, state)	=	**falsch**
s (U, state)	=	**undef**
s ((Aussage), state)	=	s (Aussage, state)
s ($\neg$ Faktor, state)	=	(`nicht` s (Faktor, state))
s (Term $\wedge$ Faktor, state)	=	(s (Term, state) `und` s (Faktor, state))
s (Term $\overline{\wedge}$ Faktor, state)	=	let x = s (Term, state) in (`if` x `then` s (Faktor, state) `else` (`if` x = **falsch** `then` **falsch** `else` **undef**))
s (Ausdruck $\vee$ Term, state)	=	(s (Ausdruck, state) `oder` s (Term, state))
s (Ausdruck $\overline{\vee}$ Term, state)	=	let x = s (Ausdruck, state) in (`if` x = **undef** `then` **undef** `else` (`if` x = **falsch** then s (Term, state) `else` **wahr**))
s (Imp-Ausdruck $\Rightarrow$ Ausdruck, state)	=	(s (Imp-Ausdruck, state) `impl` s (Ausdruck, state))
s (Aussage $\Leftrightarrow$ Imp-Ausdruck, state)	=	(s (Aussage, state) `gleich` s (Imp-Ausdruck, state))

Abbildung 2.16: *Semantik der dreiwertigen Logik*

Damit ist dann auch die Bedeutung von $\overline{\wedge}$ und $\overline{\vee}$ klar; im Prinzip entspricht sie der von $\wedge$ und $\vee$, nur daß der zweite Operand nicht immer ausgewertet wird.

2.3 Formale Beschreibung von Programmiersprachen

Im vorigen Abschnitt haben wir die Prädikatenlogik als Beschreibungsmittel für formale Spezifikationen eingeführt. Um die Bedeutung von Programmiersprachenkonzepten einigermaßen unmißverständlich ausdrücken zu können, brauchen wir eine weitere formale Sprache, welche eine mathematische Beschreibung der *Wirkung* von Elementen der Programmiersprache ermöglicht. Diese *Sprachbeschreibungssprache* oder auch *Metasprache*) sollte auf intuitiv verständlichen und möglichst einfachen Konzepten aufgebaut werden, die nicht mehr einer ausführlichen Erklärung bedürfen.[8]

Da man im allgemeinen davon ausgehen kann, daß die *Idee einer Funktion* intuitiv klar ist, bauen wir die Metasprache aus bekannten einfachen Funktionen auf und ergänzen diese um einige einfache Strukturierungskonzepte (vgl. Abschnitt 1.2.2). Um die Konstrukte der Metasprache von eventuell gleichlautenden Konstrukten der zu beschreibenden Programmiersprache zu unterscheiden, werden wir sie unterstrichen darstellen.

2.3.1 Funktionen und zusammengesetzte Ausdrücke

Eine Funktion `f` ist eine Abbildung von Elementen eines Bereiches `A` in Elemente eines Bereiches `B`. Wir schreiben hierfür auch `f:A`$\rightarrow$`B`. `A` ist der *Definitionsbereich* von `f` und `B` der *Wertebereich* von `f`.

[8] Wir stehen hier vor dem Dilemma, daß wir eine formale Sprache als grundlegend annehmen müssen. Würde man versuchten, jede formale Sprache wieder durch mathematisch genaue Formulierungen ohne Umgangssprache zu erklären, so würde man schließlich unendlich lange nach Vereinfachungen suchen und nie zum Ziele kommen.

Funktionen, die auf allen Elementen von `A` definiert sind, heißen *total*. Ist eine Funktionen – wie die Funktion `f` mit `f(x)=1/x` – auf einigen Elementen undefiniert sein, dann heißen sie *partiell*. Die Menge der Argumente, auf denen eine Funktion `f` definiert ist, heißt der *Domain* von `f`.

Den Ausdruck "`f` ist die Funktion mit `f(x)=1/x` für alle `x`" kürzen wir manchmal mit "`f=`λ`x. 1/x`" ab.

Zwei Funktionen `f:A`$\to$`B` und `g:B`$\to$`C` können durch *Komposition* zusammengesetzt werden zu einer neuen Funktion `h:A`$\to$`C` mit `h(x)=g(f(x))` für alle `x`.

2.3.2 Bedingte Ausdrücke

Bei der Beschreibung der Prädikatenlogik wurde jeder syntaktischen Konstruktion ein Wert in der Zielsprache (der Grundmengen und Wahrheitswerte) zugeordnet. Im allgemeinen ist eine solche Zuordnung nicht ganz einfach sondern benötigt einen komplexen Ausdruck zur Beschreibung des Wertes. Bei der Semantik des Allquantors, z.B. mußten wir schreiben:

$$
\begin{aligned}
s\ ((\forall\ \text{Variable} : \text{Bereich}\ \texttt{.}\ \text{Satz}),\ \text{state}) \quad = \quad & \underline{\texttt{if}}\ \text{Bereich} = \{\}\ \ \underline{\texttt{then}}\ \ \textbf{wahr} \\
& \underline{\texttt{else}}\ \ \underline{\texttt{let}}\ \text{x} \in \text{Bereich}\ \underline{\texttt{in}} \\
& \quad s\ (\text{Satz},\ \text{state} + [\text{Variable} \mapsto \text{x}])\ \texttt{und} \\
& \quad s\ ((\forall\ \text{Variable} : \text{Bereich-}\{\text{x}\}\ \texttt{.}\ \text{Satz}),\ \text{state})
\end{aligned}
$$

Hierbei haben wir eine *Fallunterscheidung* eingesetzt, die wir – angepaßt an die meist englischsprachige Literatur mit `if..then..else` bezeichnen. Sie erlaubt es, Werte abhängig von einer Bedingung zu berechnen:

`Wert = if Bedingung then Wert`$_1$ `else Wert`$_2$ entspricht $\texttt{Wert} = \begin{cases} \texttt{Wert}_1 & \text{falls } \texttt{Bedingung} \text{ wahr ist} \\ \texttt{Wert}_2 & \text{sonst} \end{cases}$

Mit der Fallunterscheidung lassen sich z.B. die Zieloperatoren der Aussagenlogik beschreiben:

$$
\begin{aligned}
a\ \texttt{und}\ b \ &= \ \underline{\texttt{if}}\ a\ \underline{\texttt{then}}\ b\ \ \underline{\texttt{else}}\ \textbf{falsch} \\
a\ \texttt{oder}\ b \ &= \ \underline{\texttt{if}}\ a\ \underline{\texttt{then}}\ \textbf{wahr}\ \ \underline{\texttt{else}}\ b \\
&\ \ \vdots
\end{aligned}
$$

2.3.3 Abkürzungen

Um zu vermeiden, daß komplexe Teilausdrücke mehrmals explizit in einem Ausdruck genannt werden müssen, kann man abkürzende Bezeichnungen dafür einführen.

Die Konstruktion "`let x=Teilausdruck in Ausdruck`" bedeutet, daß *einmal* der Wert des Teilausdrucks bestimmt wird und dann alle Vorkommen von `x` im Term `Ausdruck` durch den Wert des Teilausdrucks ersetzt werden. Der Wert des gesamten Ausdrucks ist dann der Wert von `Ausdruck` nach dieser Ersetzung, d.h. der Wert des Terms $\texttt{Ausdruck}\left[\begin{smallmatrix}\texttt{Teilausdruck}\\ x\end{smallmatrix}\right]$

Beispiel 2.3.1
Der Wert von `let r=10 in let` π`=3.1415 in 2*`π`*r` *ist* `let` π`=3.1415 in 2*`π`*10`.
Weiteres Auswerten ergibt `2*3.1415*10 = 62.830`.

Die Konstruktion "`let x`$\in$`Bereich in Ausdruck`" bedeutet, daß zunächst die durch `Bereich` bezeichnete (endliche) Menge berechnet wird und dann ein *beliebiger* Wert aus dieser Menge gewählt wird. Dann werden, wie zuvor, alle Vorkommen von `x` im Term `Ausdruck` durch diesen Wert ersetzt. Der Wert des gesamten Ausdrucks ist dann der Wert von `Ausdruck` nach der Ersetzung. Üblicherweise kann man nicht genau sagen, was dieser Wert ist – er ist *indeterminiert*

Beispiel 2.3.2 *Der Wert von '*`let x`$\in${`1,2,3`}` in 2*x`*' ist entweder 2 oder 4 oder 6.*

Wichtig ist, daß die Auswahl des Wertes aus dem Bereich frei ist, aber dann innerhalb des Ausdruckes festliegt.

Zuweilen ist die let-Schreibweise etwas unübersichtlich, wenn der Teilausdruck so groß ist, daß er die eigentliche Funktionsdefinition in den Hintergrund drängt. In diesem Fall schreibt man

`Ausdruck where x=Teilausdruck` statt `let x=Teilausdruck in Ausdruck`
und `Ausdruck where x∈Bereich` statt `let x∈Bereich in Ausdruck`

2.3.4 Tabellen

Tabellen wurden bereits im Abschnitt 2.1.3 (siehe Seite 36) verwendet um die Belegung logischer Variablen zu beschreiben. Darüberhinaus benötigt man sie zur Erklärung der Bedeutung von Bezeichnern einer Programmiersprache, zur Charakterisierung des Speichers usw. Daher ist es sinnvoll, den Begriff der Tabelle und die hierbei verwendeten Notationen näher zu erklären. Im Prinzip ist eine Tabelle nichts anderes als eine Funktion, die nur auf *endlich* vielen Argumenten definiert ist, also einen endlichen Domain hat:

tab : Argumenttyp → Werttyp

Im Gegensatz zu Funktionen, die auf unendlich vielen Argumenten definiert sind, lassen sich endliche Funktionen durch ihren Graphen oder eine Wertetabelle beschreiben (daher der Name). Jedem gegebenen Argument `arg` ordnet die Tabelle einen Wert aus dem Wertebereich zu. Die Argumente werden links, die Werte rechts aufgelistet.

Beispiel 2.3.3
Die Funktion $\texttt{fak}_5:\mathbb{N}\to\mathbb{N}$ *sei die Funktion, welche auf den Argumenten 1..5 die Fakultät angibt:*

Definitionsbereich	*Wertebereich*
1	*1*
2	*2*
3	*6*
4	*24*
5	*120*

Den aus der Tabelle `tab` ersichtlichen Funktionswert `wert` eines Arguments `arg` beschreiben wir durch die bei Funktionen übliche Schreibweise: "`tab(arg)=wert`". So ist also z.B. $\texttt{fak}_5\texttt{(4)=24}$.

Da der Domain einer Tabelle endlich ist, können wir ihn aufzählen. Die Funktion, die den Domain einer Tabelle bestimmt, nennen wir `domain`.

Beispiel: $\texttt{domain(fak}_5\texttt{) = \{1,2,3,4,5\}}$

Der Test, ob die Tabelle `tab` für ein Argument `arg` definiert ist, entspricht der Frage, ob `arg` ein Element von `domain(tab)` ist und wird entsprechend mit `arg∈domain(tab)` ausgedrückt.

Damit sind die allgemeinen Zugriffe auf eine gegebene Tabelle beschrieben. Darüberhinaus ist es aber auch nötig, Tabellen neu aufzubauen oder zu verändern, wie z.B. bei einer Zustandstabelle für einen Speicher.

Mit `[]` wird die leere Tabelle bezeichnet, also eine Tabelle mit `domain([]) = {}`. `[]` entspricht also einer nirgends definierten Funktion. Eine Tabelle, die aus einem einzigen Wertepaar `(arg,wert)` besteht, wird (zur besseren Unterscheidung gegenüber anderen Mengen von Paaren) durch `[arg↦wert]` beschrieben. Der Pfeil soll dabei die Zuordnung symbolisieren. Diese Tabelle hat beim Argument `arg` den Wert `wert` und ist sonst undefiniert. Tabellen mit mehreren Einträgen werden als Folge von Einträgen dargestellt.

Beispiel: $\texttt{fak}_5$ `= [1↦1, 2↦2, 3↦6, 4↦24, 5↦120]`

Manchmal ist es nötig, eine Tabelle um weitere Einträge zu ergänzen, also eine Tabelle $\texttt{tab}_1$ mit einer Tabelle $\texttt{tab}_2$ zu verschmelzen. Da es aber prinzipiell möglich ist, daß beide Tabellen auf demselben Argument bereits definiert sind, aber verschiedene Werte an dieser Stelle haben, müssen wir festlegen, welcher von den beiden Einträgen nach der Verschmelzung noch gültig sein soll. Als *Konvention* hat sich bewährt, im Zweifel immer

den Wert der *zweiten* Tabelle zu nehmen, weil es dadurch sehr leicht wird, Veränderungen der Einträge in einer gegebenen Tabelle zu beschreiben. Als Symbol für die Verschmelzung nimmt man das Zeichen '+'. Es ist also `tab = tab1+tab2` genau dann, wenn gilt

`domain(tab) = domain(tab1)∪domain(tab2)`
und für alle `arg∈domain(tab)`:
`tab(arg) = if arg∈domain(tab2) then tab2(arg) else tab1(arg)`

Beispiel 2.3.4

`fak5 + [6↦720]  =  [1↦1, 2↦2, 3↦6, 4↦24, 5↦120, 6↦720]`
`fak5 + [6↦720, 5↦7]  =  [1↦1, 2↦2, 3↦6, 4↦24, 5↦7, 6↦720]`

Um aus einer Tabelle `tab` die Einträge für $\{arg_1,..,arg_n\}$ streichen, schreibt man `tab / {arg1,..,argn}`.

Beispiel: `fak5 / {1,4}  =  [2↦2, 3↦6, 5↦120]`

Wir illustrieren diese Definitionen ausführlich an einem etwas komplexeren Beispiel

Beispiel 2.3.5

Die Funktion `sum: (ℕ→ℕ)→ℕ` *soll die Summe aller Werte einer Tabelle bestimmen. Wir verwenden hierzu eine* rekursive *Beschreibung, d.h. erlauben es, in der Beschreibung von* `sum` *wieder die Funktion* `sum` *zu verwenden – allerdings mit einem "kleineren" Argument.*

```
sum(t) =  if t = []
            then 0
            else let n∈domain(t) in t(n) + sum(t/{n})
```

Wir verfolgen die Berechnung von `sum` *auf einer kleinen Tabelle* `[1↦1, 2↦2, 3↦6]`.
Durch Einsetzen erhalten wir

```
sum([1↦1,2↦2,3↦6]) =
 if [1↦1,2↦2,3↦6] = []
    then 0
    else let n∈domain([1↦1,2↦2,3↦6]) in [1↦1,2↦2,3↦6](n) + sum([1↦1,2↦2,3↦6]/{n})
```

Auswerten der Bedingung ergibt, da die Tabelle `[1↦1,2↦2,3↦6]` *nicht leer ist*

```
 if falsch = []
    then 0
    else let n∈domain([1↦1,2↦2,3↦6]) in [1↦1,2↦2,3↦6](n) + sum([1↦1,2↦2,3↦6]/{n})
```

Folglich wird der else-Zweig ausgewertet:

```
 let n∈domain([1↦1,2↦2,3↦6]) in t(n) + sum(t/{n})
```

Nach Ausrechnen der `domain`*-Funktion:*

```
 let n∈{1,2,3} in [1↦1,2↦2,3↦6](n) + sum([1↦1,2↦2,3↦6]/{n})
```

Wir wählen ein beliebiges `n∈{1,2,3}`*, z.B.* `n=2` *und erhalten:*

```
 [1↦1,2↦2,3↦6](2) + sum([1↦1,2↦2,3↦6]/{2})
```

Weiteres Auswerten ergibt:

```
 2 + sum([1↦1,3↦6]/{2})
```

Nun beginnt das ganze von vorne mit der Summe der kleineren Tabelle

```
sum([1↦1,2↦2,3↦6]) =  ...
=  2 + let n∈{1,3} in [1↦1,3↦6](n) + sum([1↦1,3↦6]/{n})
=  2 + [1↦1,3↦6](3) + sum([1↦1,3↦6]/{3})
=  2 + 6 + sum([1↦1])
=  2 + 6 + let n∈{1} in [1↦1](n) + sum([1↦1]/{n})
=  2 + 6 + [1↦1](1) + sum([1↦1]/{1})
=  2 + 6 + 1 + sum([])
=  2 + 6 + 1 + 0
=  9
```

Die Funktion `sum` war ein einfaches Beispiel für eine Funktion "höherer" Ordnung, d.h. eine Funktion, die andere Funktionen (Tabellen) als Eingabe nimmt. Zuweilen geht man noch ein wenig weiter und benutzt Funktionen, die auch Funktionen als Werte bestimmen. Ein bekanntes Beispiel hierfür ist das unbestimmte Integral.

Wir geben hier eine Funktion an, die für beliebige Funktionen auf natürlichen Zahlen die Funktionstabelle der Werte `0..n` bestimmt.

Beispiel 2.3.6

`Map(f,n) = if n = 0 then [0↦f(0)] else Map(f,n-1) + [n↦f(n)]`
Dann gilt `Map : (ℕ→ℕ)×ℕ→ (ℕ→ℕ)` *und es ist zum Beispiel*
`Map(λn.n², 5) = [0↦0, 1↦1, 2↦4, 3↦9, 4↦16, 5↦25]`

2.3.5 Listen

Bei der Erklärung der Semantik prädikatenlogischer Formeln haben wir neben dem Begriff der Tabelle auch Listen benutzt, also *geordnete* Aufzählungen von Werten, die (anders als bei Mengen) möglicherweise auch mehrfach vorkommen. Die Menge aller Listen mit Werten aus der Grundmenge A wird mit A^* bezeichnet.

Wir begrenzen Listen durch spitze Klammern. `<>` bezeichnet die leere Liste, `<1,2,3,4,3>` die Liste, die aus den Werten 1,2,3,4 und dann wieder 3 besteht.

Die Funktion `head`:$A^*\to A$ bestimmt das erste Element einer Liste, falls die Liste nicht leer ist (ansonsten ist `head` undefiniert). So ist zum Beispiel `head(<1,2,3,4,3>) = 1`

Die Funktion `tail`:$A^*\to A^*$ bestimmt den Rest der Liste nach Entfernung des ersten Elementes. Zum Beispiel ist `tail(<1,2,3,4,3>) = <2,3,4,3>`. Für die leere Liste ist `tail(<>) = <>` definiert.

Wir benötigen außerdem noch eine Operation, welche Listen zusammensetzt. Diese Funktion wird mit dem Infixsymbol `&` bezeichnet. So ist `<1,2,3,4,3> & <3,5,2,4> = <1,2,3,4,3,3,5,2,4>`

Aus diesen Grundoperatoren lassen sich eine Reihe weiterer nützlicher Funktionen bilden, die wir zuweilen verwenden werden.

Beispiel 2.3.7

1. *Die Funktion* `length`:$A^*\to\mathbb{N}$ *bestimmt die Länge einer Liste. Sie läßt sich rekursiv wie folgt beschreiben:*

   `length(l) = if l=<> then 0 else length(tail(l))+1`

   Damit berechnet sich z.B. die Länge der Liste `<1,2,3>` *wie folgt:*

   ```
   length(<1,2,3>)
    = if <1,2,3>=<>  then 0  else length(tail(<1,2,3>))+1
    = length(tail(<1,2,3>))+1
    = length(<2,3>)+1
    = (if <2,3>=<>  then 0  else length(tail(<2,3>))+1) + 1
    = (length(<3>)+1) + 1
    = (if <3>=<>  then 0  else length(tail(<3>))+1) + 1 + 1
    = (length(<>)+1) + 1 + 1
    = (if <>=<>  then 0  else length(tail(<>))+1) + 1 + 1 + 1
    = 0 + 1 + 1 + 1
    = 3
   ```

2. *Die in Abbildung 2.13 auf Seite 47 benutzte Funktion* `einelementig`:$A^*\to\mathbb{B}$ *ergibt sich durch einen Test auf die Länge 1:*

   `einelementig(l) = length(l)=1`

   Man beachte, daß bei der Definition dieser Funktion keine weitere `if`*-Abfrage nötig war, da sie selbst einen Wahrheitswert als Ergebnis liefert.*

3. *Die (Infix-)Funktion* `cons`:$A\times A^*\to A^*$ *hängt ein Element vor den Anfang einer Liste. Sie ist eigentlich nur eine Kurzschreibweise für einen Spezialfall der* `&`*-Operation:*

   `cons(a,l) = <a> & l`

4. *Die Funktion* `reverse`:$A^*\rightarrow A^*$ *dreht die Reihenfolge der Elemente einer Liste um:*

```
reverse(l) = if l=<> then <> else reverse(tail(l))&<head(l)>
```

Neben Listen werden wir auch endliche Mengen benutzen (d.h Aufzählungen von Werten, bei denen ein mehrfaches Vorkommen von Elementen nicht berücksichtigt wird). Wir gehen davon aus, das die Bedeutung von Mengennotationen bekannt ist und werden sie daher hier nicht mehr ausführlich besprechen. Abbildung 2.17 fast alle wichtigen Notationen der Metasprache zusammen.

λ`x. ausdruck`	Funktionsdefinition
`f(x)`	Funktionsanwendung
`if..then..else`	Fallunterscheidung
`let x = Teilausdruck in Ausdruck`	determinierte Abkürzung
`Ausdruck where x = Teilausdruck`	von Teilausdrücken
`let x` $\in$ `Bereich in Ausdruck`	indeterminierte Abkürzung
`Ausdruck where x` $\in$ `Bereich`	von Teilausdrücken
$\{\}$	Leere Menge
$\{arg_1,\ldots,arg_n\}$	Explizite Mengenangabe
$M_1 \cup M_2$, $M_1 \cap M_2$	Vereinigung, Durchschnitt
$M_1 - M_2$	Mengendifferenz
`[]`	Leere Tabelle (endliche Funktion)
$[arg_1 \mapsto wert_1, \ldots, arg_n \mapsto wert_n]$	Explizite Tabellenangabe
`domain(tab)`	Domain der endlichen Funktion `tab`
$tab_1 + tab_2$	Verschmelzung zweier Tabellen
`tab /` $\{arg_1,\ldots,arg_n\}$	Entfernen von Tabelleneinträgen
A^*	Menge aller Listen über Elementen aus A
`<>`	Leere Liste
$<a_1,\ldots,a_n>$	Explizite Listenangabe
`head(l)`	Erstes Element von `l`
`tail(l)`	Rest von `l`
l_1 `&` l_2	Verkettung zweier Listen

Abbildung 2.17: *Spezielles Vokabular der Metasprache*

2.4 Diskussion

Wegen der Zweideutigkeiten in der natürlichen Sprache sind formale Sprachbeschreibungen ein *notwendiges Handwerkzeug* zur präzisen Beschreibung der Programmierung. Nur damit ist es möglich, den Zusammenhang zwischen der natürlichsprachigen Beschreibung eines Problems, welches man lösen will, und der – notwendigerweise absolut formalen – Computersprache, in der eine Lösung beschrieben werden soll, zu klären.

Um Kommunikationsschwierigkeiten zu vermeiden, ist es wichtig, daß alle Beteiligten für diesen Zweck die gleiche formale Sprache, d.h. die gleichen syntaktischen Konstrukte, verwenden und mit den einzelnen Konstrukten auch die gleiche Bedeutung (Semantik) verbinden. Statt also in jedem Einzelfall lange zu diskutieren, was gemeint war, legen wir uns ein für allemal fest, was bestimmte Ausdrücke bedeuten sollen und wie man eine solche Bedeutung *ausrechnen* kann.

In diesem Kapitel haben wir Beschreibungsformen für die Syntax einer Sprache (Backus-Naur Form, Syntaxdiagramme) und ihre Semantik (Definitionsgleichungen, Interpretation in einer Metasprache) vorgestellt. Sie sollten nun in der Lage sein, die Syntax einer in dieser Art beschriebenen Sprache zu verstehen und ihre Bedeutung zu erfassen. Mit der seit langem innerhalb der Mathematik etablierte *Logik* haben wir die vielleicht

wichtigste formale Sprache zur Beschreibung erlaubter Schlußfolgerungen vorgestellt. Da die "Berechnung der Semantik" im allgemeinen ein sehr aufwendiger Prozeß ist, haben wir Methoden angegeben, *syntaktische Manipulationen* logischer Formeln durchzuführen, welche die Semantik nachweislich nicht verändern. Dies ermöglicht es, durch "stures Anwenden" syntaktischer Manipulationsregeln eines logischen Kalküls die Gültigkeit eines logischen Schlusses durch *Ableitung* zu überprüfen(, wofür man dann wieder – weil nur formale Operationen durchgeführt werden – Computerunterstützung zu Hilfe nehmen kann).

Wir haben daher nun das notwendige Handwerkzeug beisammen, um die von einem Auftraggeber gelieferte informale Aufgabenbeschreibung eines Programmierproblems zu präzisieren (also ein *Modell zu bilden*) und *systematisch* in ein korrektes Programm (also eine maschinell verarbeitbare – aber für "normale Menschen" meist schlecht lesbare – Version des Modells) zu übertragen.

2.5 Ergänzende Literatur

Für ein erfolgreiches Bestehen der Lehrveranstaltung ist es **nicht** notwendig, die im folgenden angegebene Literatur zu lesen, geschweige denn durchzuarbeiten. Diese Liste soll lediglich interessierten Lesern die Möglichkeit zur Vertiefung des behandelten Stoffes bieten. Sie erhebt keinen Anspruch auf Vollständigkeit. Die Menge allein der Bücher zum Thema Logik ist Legion. Da die Darstellungen, Sichtweisen und Schwerpunkte von Autor zu Autor wechseln, erscheint es sinnvoll, sich zunächst einige Bücher zu einem Thema anzusehen und dann erst zu entscheiden, mit welchem man seine Kenntnisse vertiefen möchte.

Die *Geschichte der Logik* wird in [Bochenski & Menne, 1983] dargestellt. Dort finden sich auch zu den verschiedensten Gebieten der Logik Angaben über weiterführende Literatur. In [Davis, 1989] wird der Zusammenhang zwischen den Begriff Wahrheit, Ableitbarkeit und Berechenbarkeit beschrieben. Das Buch ist sehr empfehlenswert, gut verständlich, aber leider teuer. Weitere Erläuterungen zur *Backus-Naur Form* und zur Aussagenlogik finden sich in [Gries, 1981]. Das Buch ist gerade für Anfänger sehr gut geeignet. [Woodcock & Loomes, 1988] bringt Anwendungsbeispiele für Logik als Spezifikationssprache.

Speziell auf für angehende Informatiker ist nach Dafürhalten der Autoren das Buch [Bauer & Wirsing, 1991] geeignet. Es bringt die Grundlagen der Aussagenlogik in einer ziemlich gründlichen Form, ohne übermäßige Voraussetzungen zu verlangen. Es werden eine ganze Reihe von Themengebieten darstellt, auf die im Rahmen dieser Lehrveranstaltung leider nicht eingegangen werden kann, so z.B. die Modallogik und immer wieder praktische Anwendungen aufgezeigt. In dem vorliegenden Band wird nicht auf die Prädikatenlogik eingegangen, es ist jedoch ein eigener Band zu diesem Thema angekündigt.

[Loeckx & Sieber, 1987] ist eine lesenswerte Einführung zum Thema Programmverifikation. Die Darstellung ist sehr klar, es werden konsistent, in einem einheitlichen Begriffsrahmen die verschiedenen Verifikationsmethoden vorgestellt. In dem Buch finden sich am Ende jedes Kapitels Bemerkungen zur Geschichte und weiterführenden Literatur. Diese Einführung dürfte jedoch eher für Studierende des Hauptstudiums mit Kenntnissen der theoretischen Informatik geeignet sein. In [Stoy, 1977] wird Beschreibung der Semantik von Programmiersprachen mittels denotationaler Semantiken dargestellt. Das Buch ist jedoch, trotz seines Grundlagencharakters, eher für Studierende des Hauptstudiums geeignet.

Kapitel 3

Klassen und Objekte

Mit der Besprechung der wichtigsten Teile formaler Sprachbeschreibungen und der formalen Logik haben wir die Grundlagen für den Umgang mit formalen Systemen gelegt. Sie sollten jetzt in der Lage sein, die Beschreibung der Syntax und der Semantik formaler Sprachen zu verstehen und umgangssprachliche Ausdrücke in der formalen Sprache der Prädikatenlogik niederzuschreiben. Dies bedeutet insbesondere, daß Sie jetzt das notwendige Handwerkzeug besitzen, um Anforderungen an ein Softwareprodukt zu präzisieren und ein Pflichtenheft in der Form einer vollständigen, logisch-formalen Spezifikation niederzuschreiben.

Ab jetzt wollen wir uns der Entwicklung von Softwaresystemen widmen und dabei versuchen, den in Kapitel 1.1 genannten Qualitätskriterien gerecht zu werden. Für diese Entwicklung und noch viel mehr für die spätere Programm*pflege*[1] ist eine Strukturierung von Softwaresystemen in verhältnismäßig kleine, überschaubare Einheiten, die weitgehend unabhängig voneinander entwickelt, geprüft und gepflegt werden können, von großer Bedeutung. Ist ein Programmsystem gut strukturiert, so kann man leichter Programmteile erkennen, die auch in andere Systeme eingebaut werden können. Dies erspart nicht nur Entwicklungskosten, sondern verbessert auch die Zuverlässigkeit, da sich die wiederverwendeten Teile bereits bewährt haben.

Schon am Ende des Einführungskapitels hatten wir darauf hingewiesen, daß es sich bei der Strukturierung komplexerer Software bewährt hat, genau in umgekehrter Reihenfolge der historischen Entwicklung der Strukturierungskonzepte vorzugehen.

> Zuerst werden die Daten strukturiert, das System grob in Klassen zerlegt und die Beziehung der Klassen (Vererbung, Benutzung) zueinander festgelegt. Danach werden für jede einzelne Klasse die notwendigen Leistungen in Form eines Kontraktes beschrieben. Anschließend wird jede Leistung einer Klasse entsprechend dem Kontrakt programmiert, wobei man sich der schrittweisen Verfeinerung als Entwurfstechnik bedient. Zum Schluß kümmert man sich um Ein- und Ausgabeformate.

Bei dieser Reihenfolge dauert es zwar länger, bis man die ersten Effekte auf dem Computer beobachten kann. Was jedoch das marktreife Gesamtsystem betrifft, kommt man erheblich schneller zum Ziel und darüber hinaus auch zu einem erheblich besseren Produkt. Dies gilt natürlich nicht für kleinere Aufgabenstellungen, für deren Lösung "konventionelle" Programmierkonzepte, die in den meisten gängigen Programmiersprachen enthalten sind, durchaus ausreichen oder gar eleganter sind. Bei größeren Aufgaben werden jedoch Programme ohne Datenstrukturierung sehr schnell unübersichtlich.

Aus diesem Grunde werden wir beim Aufbau dieser Vorlesung *nicht* der historischen Entwicklung folgen und mit den bekannteren Konzepten beginnen, sondern "von oben" in die Programmierkonzepte einsteigen und mit der Klassifizierung der Daten anfangen. Wir werden zuerst die objektorientierte Denkweise an Beispielen illustrieren und anschließend die Syntax und Semantik der entsprechenden Klassifizierungskonstrukte der Programmiersprache Eiffel vorstellen. Im nächsten Kapitel werden wir zunächst besprechen, wie man bereits auf dieser Basis die Zuverlässigkeit des Strukturierungsentwurfs *verifizieren* kann, und erst dann die vielen anderen Konstrukte vorstellen, die nötig sind um wirklich zu programmieren.

[1] Unter Pflege versteht man die Beseitigung von Fehlern und die Weiterentwicklung des Systems durch kleine Modifikationen oder Ergänzungen, die das Rahmenkonzept jedoch nicht ändern.

Wir wollen jedoch deutlich darauf hinweisen, daß die zum Zwecke der Erklärung der Programmierkonzepte gegebene Beschreibung der Programmiersprache Eiffel *keineswegs vollständig* ist, sondern nur den für die Lehrveranstaltung *notwendigen* Umfang wiedergibt. Für darüber hinausgehende Details und als Sprachreferenz hierfür empfehlen wir das Buch "Eiffel the language" [Meyer, 1992] sowie die ggf. von der Rechnerbetriebsgruppe zur Verfügung gestellten Unterlagen.

Leitbeispiel

In dieser Vorlesung werden wir öfter auf den Entwurf eines Softwaresystems zurückkommen, welches die Verwaltung mehrerer größerer Bibliotheken einschließlich eines Informationssystems für Hintergrundinformationen bereitstellen soll. An diesem Beispiel, das – trotz des völlig andersartigen Anwendungsbereichs – konzeptionell eine gewisse Ähnlichkeit zu der in den Übungen verwandten "Flugverkehrverwaltung" aufweist und sich ähnlich schnell programmieren läßt, wollen wir die Vorteile der Strukturierungsmethoden illustrieren.

> *An der TH Darmstadt gibt es eine Reihe von Bibliotheken wie zum Beispiel die Informatik-Bibliothek, die Bibliothek der Elektrotechnik, der Wirtschaftswissenschaften, die Landes- und Hochschulbibliothek, die an das Fernleiheverfahren angeschlossenen Bibliotheken usw.*
>
> *Jede dieser Bibliotheken besitzt eine große Menge von Büchern, die üblicherweise nach verschiedenen Themengebieten und innerhalb dieser nach Autoren sortiert sind. Die Bücher haben neben den üblichen bibliographischen Daten eine bibliotheksinterne Kennung, um mehrfach vorhandene Bücher voneinander zu unterscheiden. Die Buchbestände der verschiedenen Bibliotheken sind natürlich disjunkt, d.h. kein Buch gehört gleichzeitig zu mehreren Bibliotheken. Das schließt aber nicht aus, daß ein Buch mit gleichem Titel (oder gar mit der gleichen Kennung) in mehreren Bibliotheken existiert. Jedes Buch einer Bibliothek ist entweder vorhanden als entleihbares Buch, vorhanden als Präsenzexemplar, vorhanden als Semesterapparat oder ausgeliehen.*
>
> *In jeder Bibliothek gibt es verschiedene Arten von Personen, die Bücher entnehmen oder bringen können. Gewöhnliche Entleiher dürfen nur Bücher von Bibliotheken entleihen, deren Bibliotheksausweis sie besitzen. Mitarbeiter einer Bibliothek können neue Bücher dem Bestand hinzufügen oder beschädigte (oder verlorengegangene) Bücher für immer dem Bestand entnehmen. Natürlich können sie in dieser oder einer anderen Bibliothek auch ganz gewöhnliche Entleiher sein. Darüber hinaus gibt es Universitätsangestellte, die in manchen Bibliotheken ungehinderten Zugriff haben und jederzeit alle Bücher – mit Ausnahme von Büchern des Semesterapparats – entnehmen können, sich dabei aber freiwillig dem üblichen Entleihverfahren unterwerfen.*
>
> *Bücher werden nach Autor und Titel ausgeliehen. Solange das gewünschte Buch vorhanden ist und das Ausleihlimit nicht erreicht ist, kann es für jeweils maximal 4 Wochen ausgeliehen werden mit einer Verlängerungsmöglichkeit für weitere 4 Wochen. Hochschullehrer, eine spezielle Gruppe von Universitätsangestellten, dürfen Bücher für maximal ein Semester ausleihen. Beim Verleih eines Buches wird das aktuelle (letzte) Ausleihdatum und die Ausleihfrist vermerkt, bei der Rückgabe entsprechend das letzte Rückgabedatum.*
>
> *Mitarbeiter einer Bibliothek können ein konkretes Buch, gekennzeichnet durch Autor, Titel und Kennung aus dem Bestand entfernen, wenn es nicht gerade verliehen ist. Sie können neuangeschaffte Bücher hinzufügen, wobei die (eindeutige) Kennung automatisch vergeben wird.*
>
> *Die Verwaltung des Buchbestandes und aller Transaktionen soll von* einem *Computersystem übernommen werden, das hierfür folgende Informationen benötigt: aktuelles Datum, Bibliothek, Entleiher, Art der Transaktion, und die notwendigen Informationen über das Buch, das entliehen, zurückgebracht, neu hinzugefügt oder für immer entnommen wird. Das Datum wird täglich angepaßt, soll für Demonstrationszwecke aber auch von Hand (vor-)verstellbar sein. Neben der internen Verwaltung soll das Programm Verwaltungsinformationen wie z.B. Verzeichnisse der vorhandenenen Bücher und ihres Ausleihstatus, Verzeichnisse ausgeliehener Bücher, zu mahnende Entleiher, usw. bereitstellen und ebenso Informationen über die Autoren der Bücher.*
> *Weitere Aufgabenstellungen werden sich später – also während des Betriebs – ergeben.*

3.1 Objekte

Beim Entwurf eines Softwaresystems steht man vor der Aufgabe, das Zusammenspiel einer Reihe von verhältnismäßig unabhängigen Objekten und Ereignissen in einer von den späteren Anwendern gewünschten Weise verarbeiten zu müssen. Man kommt daher nicht umhin, einen oft sehr komplexen Ausschnitt dieser realen Welt zu modellieren. Aus diesem Grunde ist die erste Frage, die man sich bei der Konzeption eines großen Softwaresystems stellen sollte, *mit welchen Objekten gearbeitet werden muß*. Alles, was man dann modelliert, sollte auf der formalen Beschreibung von Objekten basieren, die unabhängig agieren können.

Die Frage nach den Objekten hat zwei Seiten. Einerseits zielt sie auf die Objekte der Wirklichkeit (sogenannte *externe Objekte*), in der das System später eingesetzt werden soll, also zum Beispiel die Bücher und Entleiher einer Bibliothek. Andererseits geht die Software natürlich nicht direkt mit diesen Objekten um, sondern verarbeitet nur *interne Repräsentationen* davon, die man der Einfachheit ebenfalls *Objekte* nennt. Beim Entwurf und der Implementierung eines Softwaresystems geht es also darum, die für die Verarbeitung wesentlichen Aspekte der externen Objekte zu analysieren und dann durch interne Objekte zu realisieren. Dieser Prozeß wird von *objekt-orientierten* Programmiersprachen wie Eiffel besonders unterstützt.

Nun stellt sich natürlich die Frage, wie man denn die internen Objekte findet, auf denen das Programm aufgebaut sein soll? Die Antwort darauf ist überraschend einfach, wenn man sich vor Augen hält, daß Programme dafür geschrieben werden, um auf bestimmte Fragen der realen Welt eine Antwort zu geben, und deswegen einen Ausschnitt der Außenwelt modellieren müssen. Die für die Verarbeitung relevanten Aspekte eines realen Objekts bestimmen damit die Komponenten des internen Objekts.

3.1.1 Einfache Objekte

Betrachten wir zum Beispiel die Bücher, die von unserer Bibliothek verwaltet werden sollen. Neben dem Text, der für den Leser, nicht aber für die Bibliotheksverwaltung relevant ist, enthält ein Buch eine ganze Reihe von bibliographischen Daten wie Name des Autors, Titel, Erscheinungsjahr, Verlag, Anzahl der Seiten, Bibliothekskennung usw. Die zugehörige interne Repräsentation eines Buches sollte also all diese Daten, die für die Verwaltung ausreichen, in einem Paket von Informationen zusammenfassen. In anderen Worten, das zugehörige interne Objekt könnte ein Verbund[2] von diversen Einzeldaten sein, etwa

```
''Bertrand Meyer''
''Objektorientierte Software-entwicklung''
1990
''Hanser''
547
''D4.1 Mey 8267''
```

Abbildung 3.1: *Ein einfaches Objekt*

Aus interner Sicht ist dies ein Objekt, welches aus 4 Zeichenketten und 2 ganzen Zahlen besteht und die Repräsentation eines tatsächlich existierenden Buches sein könnte. Diese Interpretation ist aber keineswegs zwingend, da die 6 Komponenten auch zu ganz anderen Objekten der Wirklichkeit gehören könnten.

3.1.2 Verweise

Üblicherweise enthält ein Buch wesentlich mehr Informationen über seinen Autor als nur den Namen, etwa auch Geburts- und Todesdatum, Nationalität usw. Diese Informationen gehören eigentlich nicht direkt zum

[2]Ein Verbund heißt in PASCAL "record" und in C "structure"

Buch, sondern sollten besser als ein separates Objekt zusammengefaßt werden. Diese Objekt könnte dann Teil des Objektes sein, welches ein Buch repräsentiert.

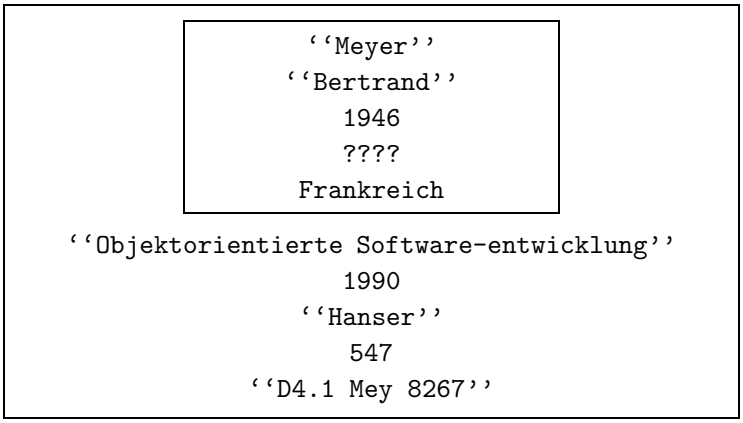


Abbildung 3.2: *Objekt in einem Objekt*

Diese Darstellungsform ist allerdings nicht sehr geschickt, da Autoren normalerweise mehr als nur ein Buch schreiben und dann Konsistenzprobleme auftreten, wenn sich eine der Komponenten des Autor-Objektes ändert. Stirbt zum Beispiel der Autor, dann muß in *allen* Buch-Objekten, die zu diesem Autor gehören, die Komponente “Todesdatum” verändert werden, weil ansonsten die Informationen widersprüchlich sind.

Es ist daher wesentlich sinnvoller, die Objekte voneinander zu trennen und *Verweise* von Buchobjekten auf Autor-Objekte zu bilden. Mehrere Bücher, die vom selben Autor geschrieben wurden, verweisen dann auf ein und dasselbe Objekt. Änderungen im Autor-Objekt sind damit automatisch in allen Buchobjekten bekannt, die hierauf verweisen.

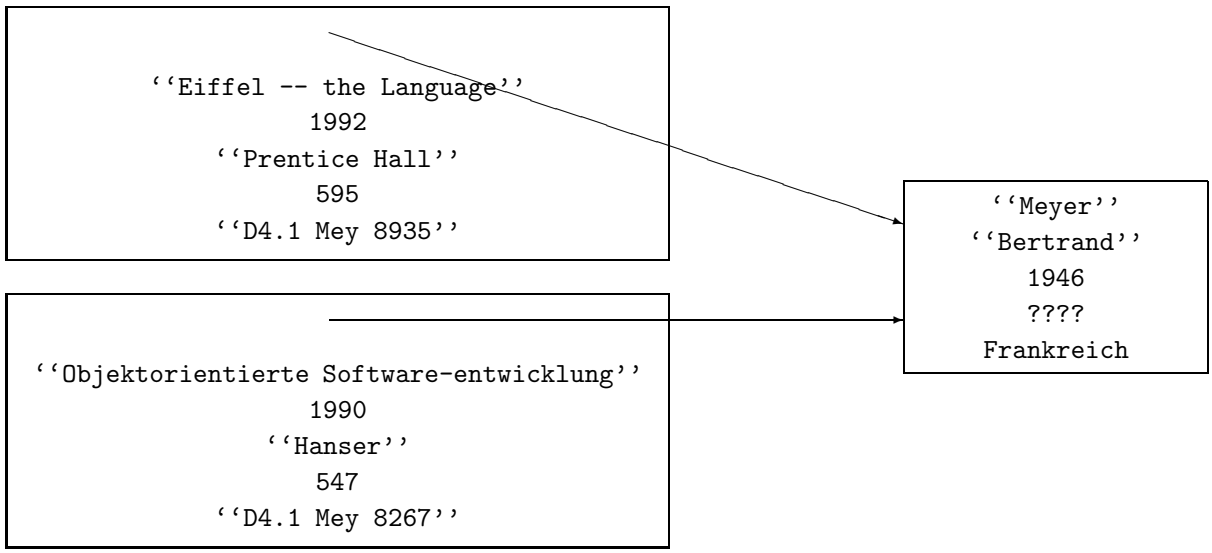


Abbildung 3.3: *Verweise*

Ein *Verweis*[3] kann also als Wert ein Objekt haben und mehrere Verweise können auf dasselbe Objekt weisen. Es ist aber durchaus möglich, daß ein Verweis *keinen* Wert hat, weil kein Objekt da ist, auf das verwiesen werden kann. In diesem Fall hat der Verweis den Wert `Void` (leer).

Die Verwendung von Verweisen ist besonders nützlich für die Darstellung komplexer Datenstrukturen wie Listen, Bäume, Warteschlangen usw. Bei der Konzeption von Eiffel hat man sich dafür entschieden, die Komponenten eines Objektes ausschließlich aus “einfachen” Objekten (ganze Zahlen, boole’sche Werte, Symbole

[3] Pascal verwendet die Bezeichnung “pointer” für einen Verweis

(character), reelle Zahlen in einfacher und doppelter Präzision) oder aus Verweisen bestehen zu lassen. Felder oder Listen als Teile von Objekten zu verwenden ist nicht möglich. Stattdessen muß aus dem Objekt heraus auf das Feld oder die Liste verwiesen werden.

Dieses Merkmal der Sprache Eiffel führt zu einer sehr dynamischen Natur von Programmen: auf Objekte angewandte Operationen führen zur Erzeugung neuer Objekte und Verweise; Verweise werden auf andere Objekte umgelenkt oder zu leeren Verweisen, usw. Dadurch entsteht ein hohes Maß von Flexibilität. Allerdings würde auch die Gefahr völlig unkontrollierbarer Programmabläufe entstehen, wenn die Verwendung von Verweisen nicht durch ein Strukturierungskonzept eingeschränkt würde. Diese Strukturierung ergibt sich automatisch aus der Tatsache, daß Objekte beschrieben werden müssen, um für das zu entwerfende System benutzbar zu sein.

3.2 Klassen

Wie beschreibt man also Objekte? Die Antwort hierauf ergibt sich aus der Anforderung, daß ein Softwaresystem nicht nur für die Verarbeitung individueller Objekte (z.B. nur für die Verarbeitung der in Abbildung 3.3 repräsentierten beiden Bücher) geschrieben wird, sondern alle Objekte verarbeiten können muß, die gewisse gemeinsame Strukturen haben wie z.B. alle Objekte, die ein Buch beschreiben. Das bedeutet, daß wir mehr an *Klassen* von Objekten interessiert sind als an Einzelobjekten. Die Beschreibung einer Klasse liefert uns somit eine Beschreibung derjenigen Eigenschaften, die alle uns interessierenden Objekte gemeinsam haben und ermöglicht es uns, die Besonderheiten eines Individuums außer Acht zu lassen.

Der Unterschied zwischen einer Klasse und einem Objekt ist derselbe wie der zwischen einer Menge und ihren Elementen (oder auch zwischen Datentypen und ihren Elementen). Die Klassen sind die abstrakten Konzepte, auf deren Beschreibung ein Programm aufgebaut wird, die Objekte sind die konkreten Ausprägungen der Konzepte, die beim tatsächlichen Ablauf des Programms eine Rolle spielen. Während des Ablaufs bleiben die Klassen und das einmal geschriebene Programm unveränderlich, während sich die individuellen Objekte durchaus ändern. Objektorientierte Sprachen wie Eiffel verwenden daher den Begriff "Klasse" um solche Datenstrukturmengen zu bezeichnen, die durch gemeinsame Eigenschaften charakterisiert sind.

Wie charakterisiert man nun eine Klasse von Objekten? Man könnte dafür irgendeine feste Darstellung wählen. So könnte man zum Beispiel die Klasse der Autoren darstellen als Zeichenketten, die an den ersten 20 Stellen den Nachnamen enthalten, die nächsten 25 Stellen die Vornamen, dann je 4 Stellen für Geburts- und Todesjahr, und dann 15 Stellen für die Nationalität. Unser Autorenobjekt in Abbildung 3.3 wäre dann

```
‘‘Meyer               Bertrand                1946????Frankreich    ’’
```

Diese Darstellung ist sehr effizient zu verarbeiten und findet daher immer noch Verwendung in vielen Softwareprodukten, die Personendaten verarbeiten müssen. Leider ist sie extrem unflexibel in Fällen, wo Personendaten nicht in das vorgesehene Schema passen wie z.B. bei

```
‘‘Kreitz              Christoph Sebastian Maxim1957????Deutschland   ’’
```

Da die Vornamen in diesem Falle mehr als 25 Zeichen einnehmen, müssen sie abgeschnitten werden, da sonst das Geburtsdatum überschrieben würde. Man könnte sich nun damit behelfen, daß man ab sofort 30 oder gar 40 Zeichen zuläßt. Dies aber ist keine gute Lösung, da nun das gesamte Programm umgeschrieben werden muß und alle bisherigen Daten entsprechend durch Einfügen von Leerzeichen angepaßt werden müssen. Bedenkt man, daß heutzutage gut ein Fünftel aller Modifikationen von Programmen aufgrund einer Änderung der Datenformate durchgeführt werden müssen, so ist eine dermaßen implementierungsabhängige Beschreibung der Klasse "Autoren" ausgesprochen ungeeignet. Sie ist *überspezifiziert*.

Auch die Beschreibung der Klasse als Verbund von drei Zeichenketten beliebiger Länge (für Name, Vorname, Nationalität) und zwei ganzen Zahlen (für Geburts- und Todesjahr) ist immer noch zu nahe an der Implementierung, da z.B. für noch lebende Autoren eine spezielle Codierung vereinbart werden muß.

Viel sinnvoller ist es, bei der *Beschreibung* einer Klasse *noch nicht an die konkrete Implementierung zu denken* sondern eine möglichst *abstrakte* Beschreibung derjenigen Informationen zu geben, die wirklich von Interesse sind. Die Klasse der Autoren, zum Beispiel, soll uns Namen, Vornamen, Nationalität, Geburts- und Todesjahr – falls vorhanden – liefern und umgekehrt zulassen, daß das Todesjahr nachgetragen wird. Alle anderen Informationen dürfen dagegen nicht verändert werden – namensrechtliche Probleme bei Heirat seien außer Acht gelassen.

Wir stehen also vor der Aufgabe, einerseits eine *vollständige, genaue und eindeutige* Beschreibung einer Klasse von Objekten zu geben, gleichzeitig aber eine Überspezifikation zu vermeiden. Die Antwort auf dieses Problem liegt in der Theorie der abstrakten Datentypen.

3.2.1 Abstrakte Datentypen

Ausgangspunkt der Theorie der abstrakten Datentypen ist die Feststellung, daß für ein Softwaresystem, welches die Objekte einer bestimmten Klasse benutzt, weniger die konkrete Realisierung relevant ist, als die Art der *Dienstleistungen*, welche es bietet. So bietet zum Beispiel die Klasse der Autoren an, Namen, Vornamen, Nationalität, Geburts- und Todesjahr, und ggf. auch das Alter eines Autors abzufragen und das Todesjahr zu verändern. *Wie* diese Dienste realisiert werden – also ob z.B. das Alter des Autors berechnet oder gespeichert wird – ist ziemlich unbedeutend. Wichtig ist dagegen, daß diese Dienstleistungen gewisse *Eigenschaften* erfüllen, auf die sich das System verlassen darf, wie zum Beispiel, daß das Alter eines Autors eine ganze Zahl ist, die größer als Null ist (wahrscheinlich wäre 8 sinnvoller, aber man kann ja nie wissen).

Eine abstrakte Datentypspezifikation beschreibt daher eine Klasse nicht durch die konkrete Realisierung sondern durch eine Liste von *Dienstleistungen*, die für die Außenwelt verfügbar sein sollen, und durch die *Eigenschaften* dieser Dienstleistungen.[4] Auf diese Art wird sichergestellt, daß die Außenwelt auf die Objekte einer Klasse nur mithilfe der bekanntgegebenen Dienstleistungen zugreifen kann und nicht etwa über die Implementierung(, welche sich ja möglicherweise ändern kann). Durch diese Form der *Datenkapselung* wird sichergestellt, daß sich jede Komponente eines Softwaresystems ausschließlich um ihre eigenen Geschäfte kümmert und in die Aufgaben der anderen nicht hineinpfuscht. Nur dieses Geheimnisprinzip garantiert die Integrität eines Moduls in einer Umwelt ständiger Veränderungen.

Für eine vollständige formale Spezifikation eines abstrakten Datentyps sind vier Fragen zu beantworten:

- Welche *Typen* werden eingeführt?
- Welche *Funktionen* (Dienstleistungen) werden eingeführt?
- Was sind die *Vorbedingungen* für die Anwendbarkeit der Funktionen?
- Welche *Axiome* (Eigenschaften) erfüllen die Funktionen?

Typen und Funktionen beschreiben die Syntax des abstrakten Datentyps, Vorbedingungen und Axiome seine Semantik. Diese Beschreibung wird in einer rein mathematischen Form gegeben, um die gewünschte Genauigkeit und Eindeutigkeit sicherzustellen.

Wir wollen die Bedeutung dieser vier Teile am Beispiel der vollständigen Spezifikation endlicher Listen (vgl. Abschnitt 2.3.5) illustrieren. Eine Liste wird als Struktur betrachtet, die folgende Dienste zur Verfügung stellt: trage ein neues Element ein (*cons*), greife auf das erste Element zu (*head*), entferne das erste Element (*tail*), prüfe ob die Liste leer ist (*empty*), erzeuge eine neue Liste (`new`), usw. Listen und ihre Dienstleistungen werden in sehr vielen Anwendungen benötigt und können auf viele verschiedene Arten implementiert werden, je nachdem, ob über die obengenannte Dienstleistungen hinaus noch weitere Dienstleistungen wie z.B. das Zusammensetzen von Listen (`&` in Abschnitt 2.3) angeboten werden sollen oder nicht.

[4] Anstelle von "Dienste" sagt man auch "*Operationen*" oder "*Merkmale*" (in Englisch "*features*").

Die vollständige Spezifikation endlicher Listen ist in Abbildung 3.4 zusammengefaßt. Sie drückt alles aus, was für den Begriff der Liste von allgemeiner Bedeutung ist und läßt alles weg, was nur für bestimmte Repräsentationen von Listen gilt.

TYPES: `List[X]`

FUNCTIONS:
```
empty: List[X] → BOOLEAN
new: → List[X]
cons: X×List[X] → List[X]
head: List[X] ↛ X
tail: List[X] → List[X]
```

PRECONDITIONS: **pre** `head(L:List[X]) = (`**not** `empty(L))`

AXIOMS:
```
∀x:X . ∀L:List[X] .
  empty(new())
  not empty(cons(x,L))
  head(cons(x,L)) = x
  tail(cons(x,L)) = L
  tail(new()) = new()
```

Abbildung 3.4: *Listen als abstrakter Datentyp*

- Die Namen der neuen Datentypen werden in einem Abschnitt **TYPES** der Spezifikation aufgelistet. Es kann sinnvoll sein, mehrere Datentypen zusammen zu spezifizieren. In unserem Fall kommt allerdings nur ein neuer Datentyp hinzu, die Liste. Dieser Typ wird eingeführt unter dem Namen "`List[X]`". Das bedeutet, das es sich um einen *parametrisierten* oder *generischen* Datentyp handelt mit einem Parameter `X`, der einen beliebigen Datentyp darstellt, über dessen Elementen die Listen gebildet werden.[5]

  Die restliche Spezifikation beschreibt nun die Eigenschaften des Typs `List[X]`, der mathematisch als *Menge von Objekten* (d.h. Listen) angesehen wird.

- In einem Abschnitt **FUNCTIONS** sind die Namen der Dienstleistungen aufgeführt, die von den Exemplaren des soeben genannten Typs zur Verfügung gestellt werden. Diese werden einheitlich als mathematische Funktionen `empty, new, cons, head, tail` beschrieben. Zu jeder Funktion wird ihre Funktionalität (vgl. Abschnitt 2.1.3 – auch "*Signatur*" genannt) angegeben, in welcher der spezifizierte neue Datentyp mindestens einmal vorkommen muß.

  So wird zum Beispiel die Dienstleistung `cons` als Funktion spezifiziert, die ein Element von `X` und eine Liste aus `List[X]` in eine neue Liste abbildet:

  ```
  cons: X×List[X] → List[X]
  ```

  Die beabsichtigte Wirkung von `cons`, nämlich daß das Element vor die Liste gehängt werden soll, gehört zu den Eigenschaften des abstrakten Datentyps und wird im Abschnitt **AXIOMS** mathematisch beschrieben. Dabei interessiert sich die mathematische Beschreibung ausschließlich für das Ergebnis der Funktion, da nur auf diese Art eine präzise und überschaubare Formulierung von Eigenschaften der Dienstleistungen möglich ist. Eventuelle Seiteneffekte der Operation `cons`, wie zum Beispiel, daß `cons` in den meisten Implementierungen einem konkreten Objekt das berechnete Ergebnis zuweist, spielen hier keine Rolle. Sie sollten erst in der letzten Phase der *Implementierung*, nicht aber in der Spezifikation eingeführt werden.

[5] Durch die Verwendung generischer Datentypen kann man sich ersparen, getrennte Spezifikationen für Listen von ganzen Zahlen, Buchstaben, oder komplexeren Datensätzen schreiben zu müssen, obwohl diese jeweils nur dieselben Dienstleistungen anbieten würden. Listen über ganzen Zahlen erhält man, indem man für `X` den Typ `INTEGER` einsetzt, also "`List[INTEGER]`" schreibt. Generische Datentypen werden wir in Abschnitt 3.6 ausführlicher besprechen.

Entsprechend ihrer beabsichtigten Bedeutung muß die Funktion `empty` eine Liste in einen booleschen Wert abbilden, die Funktion `tail` eine Liste in eine Liste und die Funktion `head` eine Liste in ein Element von `X`. Dabei ist jedoch zu berücksichtigen, daß die Funktion `head` nicht auf allen Listen definiert ist, also eine *partielle* Funktion ist. Dies wird in einem Teil der Literatur (zum Beispiel in [Meyer, 1988]) durch einen modifizierten Pfeil ($\not\rightarrow$) besonders herausgehoben, obwohl dies wegen der Nennung von Vorbedingungen eigentlich überflüssig ist.

Die Erzeugung einer neuen (leeren) Liste durch die Funktion `new` wird dadurch ausgedrückt, daß `new` keinen Argumenttyp besitzt und somit immer dasselbe Ergebnis liefert.

```
new: → List[X]
```

Aus der Sicht der Mathematik sind Funktionen ohne Argumente wie `new` (sogenannte *Konstruktor-Funktionen*) und ihr Ergebnis (die Konstante `<>`) praktisch identisch, da es von von außen betrachtet keine Rolle spielt, ob eine Dienstleistung zur Bereitstellung eines festen Wertes diesen erst berechnen muß oder auf einen bereits existierenden Wert zugreifen kann. Deshalb können *alle* Dienstleistungen als Funktionen beschrieben werden. Diese Sichtweise spiegelt sich auch in der Syntax der Sprache Eiffel wieder, welche die gleiche Philosophie verfolgt.

- Partielle Funktionen sind in fast allen Programmierproblemen eine unausweichliche Tatsache, aber auch eine mögliche Quelle für Fehler. Aus diesem Grunde müssen die Anforderungen an die Anwendbarkeit jeder partiellen Funktion klar formuliert werden. Dies ist der Zweck des Abschnitts **PRECONDITIONS**. In unserem Beispiel ist die einzige Vorbedingung, daß `head` nur auf nichtleere Listen angewandt werden darf:

  ```
  pre head(L:List[X]) = (not empty(L))
  ```

- Der Abschnitt **AXIOMS** schließlich beschreibt die semantischen Mindestanforderungen, welche sicherstellen, daß die genannten Funktionen die gewünschten Eigenschaften besitzen.

  So soll die Funktion `empty` nur für die von `new` erzeugte leere Liste wahr sein, nicht aber bei Listen, die durch `cons` erzeugt wurden. `head` und `tail` zerlegen eine durch `cons` erzeugte Liste wieder in ihre Bestandteile während auf eine leere Liste nur `tail` anwendbar ist und wieder eine leere Liste liefert.

Die abstrakte mathematische Beschreibung von Datentypen *und* Operationen in einem gemeinsamen Konzept erlaubt es, präzise Aussagen über die Effekte einer Berechnung aufzustellen und durch Verwendung der Axiome zu beweisen. Die Durchführung einer Berechnung kann nämlich als algebraische Vereinfachung ("Reduktion") betrachtet werden, in der die linken (langen) Seiten eines Axioms durch die rechten (kurzen) ersetzt werden.

So läßt sich zum Beispiel auch ohne Verwendung der anschaulich erklärten Bedeutung der Operationen `tail`, `cons` und `new` beweisen, daß das Ergebnis der Berechnung von `tail(tail(cons(4,new())))` der Wert `new()` ist. Hierzu braucht man nur das vierte und dann das fünfte Axiom anzuwenden.

Diese Aspekte abstrakter Datentypen haben sie zur Grundlage vieler Forschungen auf den Gebieten der formalen Spezifikation, der symbolischen Berechnung, der Verifikation von Programmen, dem Software-Prototyping und nicht zuletzt auch der Datenstrukturbeschreibung werden lassen. In objektorientierten Sprachen wie Eiffel wird daher jedes Programm um Klassen von Datenstrukturen herum organisiert, die auf der Grundlage abstrakter Datentypen beschrieben werden. Auf diese Art wird ein ausgewogenes Verhältnis zwischen Daten und Funktionen hergestellt. Die Struktur eines Softwaresystems beruht auf den Datenstrukturen, aber diese sind wiederum auf der Grundlage *abstrakter Funktionen* definiert:

> *Objektorientierter Entwurf ist die Entwicklung von Softwaresystemen als strukturierte Sammlung von Implementierungen abstrakter Datentypen.*

3.2.2 Klassen in Eiffel

Mit den abstrakten Datentypen haben wir eine *mathematische* Beschreibungsform gegeben, die es erlaubt, Klassen von Objekten vollständig, genau und eindeutig zu charakterisieren und dabei gleichzeitig eine Überspezifikation zu vermeiden. Wir wollen nun zeigen, wie man eine solche Beschreibungsform in einer Programmiersprache ausdrücken kann.

In der Denkweise der Programmiersprachen hat sich anstelle des Namens "abstrakter Datentyp" der Begriff "Klasse" zur Beschreibung der Gemeinsamkeiten einer Gruppe von Objekten eingebürgert. In Eiffel ist das Konzept der Klasse nahezu identisch mit dem der abstrakten Datentypen. Gegenüber der mathematischen Sichtweise ändern sich nur einige der verwendeten Begriffe und Schlüsselworte.

Die einfachste Form einer *Klassendefinition* ist etwas ähnliches wie ein Verbundtyp in Pascal, in dem nur die gemeinsame Struktur einer Gruppe von Objekten festgelegt wird:

```
class PERSON feature
     name, vornamen: STRING;
     geburtsjahr, todesjahr: INTEGER;
     nationalität: STRING
end -- class PERSON
```

Abbildung 3.5: *Eine einfache Klassendefinition*

Die Klasse in Abbildung 3.5 beschreibt die Struktur einer Menge von Objekten, die *Instanzen* oder *Exemplare* dieser Klasse genannt werden. Die Klasse erhält den Namen `PERSON` und stellt Komponenten wie `name, vorname,` etc. zur Verfügung. Die Komponenten einer Klasse werden als *Dienstleistungen* der Klasse betrachtet, die für eventuelle Benutzer dieser Klasse nach außen hin verfügbar sind. Dabei spielt es – von außen betrachtet – keine Rolle, ob diese Dienstleistungen erbracht werden, indem auf eine abgespeicherte Komponente zugegriffen wird oder indem eine Berechnung (z.B. zur Bestimmung einer eindeutigen internen Kennung) durchgeführt wird. Diese Sicht deckt sich mit der im Abschnitt 3.2.1 angesprochenen Philosophie der abstrakten Datentypen, die *alle* Dienstleistungen als Funktionen ansieht, von denen einige vielleicht keine Eingaben benötigen.

Um dieser Sichtweise Rechnung zu tragen, werden in Eiffel alle nach außen hin sichtbaren Dienstleistungen einer Klasse mit dem Oberbegriff *feature*[6] bezeichnet. Im einfachsten Fall sind Features nur *Attribute*, also Bestandteile von Klassen, die tatsächlich eine Komponente der Objekte der Klasse bezeichnen. Features können aber auch andere von der Klasse bereitgestellte Dienstleistungen sein, nämlich *Routinen*, die Operationen auf den Objekten der Klasse beschreiben, wie zum Beispiel die Bestimmung aller derzeit verfügbaren (nicht ausgeliehenen) Bücher, die von einem bestimmten Autor geschrieben wurden. Hierauf werden wir im Abschnitt 3.3 zurückkommen.

Die Schlüsselworte **class**, **feature** und **end**, gedruckt in einem anderen Schriftsatz, werden zur syntaktischen Aufteilung einer Klassendefinition in Klassendeklaration und Deklaration von Features benutzt. Features vom selben Typ können in Deklaration gruppiert werden. Für die Aneinanderreihung von Deklarationen (und später auch von anderen Instruktionen) wird als Trennzeichen das Semikolon "`;`" verwendet und für *Kommentare* der doppelte Bindestrich "`--`". Es hat sich als gute Konvention bewährt, am Ende einer Klassendefinition den Namen der Klasse im Kommentar zu wiederholen.

Klassen sind die Grundkomponenten, aus denen alle Eiffel-Programme aufgebaut werden. Sie beschreiben nicht nur die Datenstrukturen, die in einem Softwareprodukt verwendet werden, sondern ebenso *alle* Module, aus denen dieses Softwareprodukt aufgebaut ist – einschließlich dem "Hauptprogramm". Diese Vorgehensweise entspricht einer konsequenten Umsetzung der objektorientierten Sichtweise. Ein Programm ist nichts

[6]Der deutsche Begriff für *feature* ist *Merkmal*. Der englische Begriff hat sich jedoch auch in der deutsprachigen Literatur eingebürgert, da er in seinem Verwendungszweck eindeutiger zu sein scheint. Wir werden dieser Tatsache Rechnung tragen und trotz des sich daraus ergebenden Mißbrauchs der deutschen Sprache und Grammatik den Begriff *feature* weiterverwenden.

Klassen sind rein statische Beschreibungen einer Menge *möglicher* Objekte. Sie beschreiben die Struktur der möglichen Objekte durch *Attribute* und mögliche Operationen auf den Objekten durch *Routinen*. Attribute und Routinen bilden zusammen die *Merkmale* (Features) einer Klasse. Im Programmtext von Softwaresystemen sind nur die Klassen sichtbar.

Objekte sind *Laufzeitelemente*, die während der *Ausführung* eines Softwaresystems erzeugt werden. Jedes Objekt ist ein *Exemplar* (*Instanz*) einer Klasse, die auch der *Typ* des Objektes genannt wird. Die Komponenten eines Objektes entsprechen den Attributen derjenigen Klasse, deren Instanz es ist. Zur Laufzeit eines Softwaresystems sind nur die Objekte sichtbar.

Abbildung 3.6: *Verhältnis von Klassen und Objekten*

anderes als ein Konzept, welches eine oder mehrere Dienstleistungen an seine Benutzer zur Verfügung stellt und genauso auf die Dienstleistungen anderer Konzepte zurückgreifen kann. Daher ist es sinnvoll, all diese Konzepte einheitlich als Klassen zu organisieren, was eine modulare Programmierung sehr stark unterstützt. Ein Software*system* ist also nichts anderes als eine Sammlung aller zugehörigen Konzepte bzw. Klassen.

Erfahrungsgemäß gibt es oft Probleme, Objekte und Klassen auseinanderzuhalten. Wir fassen daher die wichtigsten Erkenntnisse in Abbildung 3.6 zusammen.

3.2.3 Typen und Verweise

Eiffel ist eine *statisch getypte* Sprache. Das bedeutet, daß jedes Attribut (und auch jede Routine) mit einem eindeutigen Datentyp deklariert werden muß, zu dem dieses Attribut gehören soll. Die Attribute der Klasse `PERSON` in Abbildung 3.5 konnten alle mit den vordefinierten Datentypen `INTEGER` und `STRING` definiert werden. Dies ist aber normalerweise nicht der Fall, wie das Beispiel der Klasse aller Bücher zeigt:

```
class BUCH feature
      autor: PERSON;
      titel: STRING;
      erscheinungsdatum: INTEGER;
      verlag: STRING;
      anzahl_seiten, kennung: INTEGER
end -- class BUCH
```

Abbildung 3.7: *Klassendefinition mit implizitem Verweis*

Das Attribut `autor` hat als Datentyp die soeben definierte Klasse `PERSON`. Da wir, wie in Abschnitt 3.1.2 besprochen, keine Objekte innerhalb von Objekten zulassen wollen, bedeutet diese Deklaration, daß die zum Attribut `autor` zugehörige Komponente eines Buchobjekts einen *Verweis* auf ein Objekt vom Typ `PERSON` ist. Eine besondere Kennzeichnung des Datentyps für das Attribut `autor` als ein Verweistyp (wie etwa in Pascal) ist dafür *nicht* erforderlich, da sich diese Tatsache aus dem Kontext ergibt. Sie wird deshalb auch unterlassen.[7]

Attribute, deren Typ kein einfacher Datentyp ist, beschreiben Verweise auf Objekte dieses Typs

Es gibt also in Eiffel nur zwei Arten von Datentypen: *einfache Typen* (`INTEGER`, `BOOLEAN`, `CHARACTER`, `REAL`, und `DOUBLE`) und *Klassentypen*. Klassentypen müssen durch Klassendeklarationen definiert werden und gehören nicht zum Basistypsystem von Eiffel.

Von dieser Regel gibt es allerdings eine Ausnahme. Neben der eigentlichen Programmiersprache, die aus den in "Eiffel the language" [Meyer, 1992] beschriebenen Grundkonstrukten besteht, enthält die von einem Compiler zur verfügung gestellte *Programmierumgebung* der Sprache Eiffel eine Reihe von *Programmbibliotheken*

[7]Die Unterlassung dieser Kennzeichnung von Verweisen ergibt sich aus der objektorientierten Denkweise: aus der Sicht des Objekts selbst ist die entsprechende Komponente tatsächlich ein anderes Objekt und kein Verweis. Verweise sind nur die rechnerinterne Methode die Konsistenz zwischen Objekten sicherzustellen, welche dasselbe Objekt als Komponente beinhalten.

(*libraries*), die eine Reihe vordefinierter Klassen enthalten und bei Bedarf geladen werden können. Darunter befindet sich auch eine Eiffel-Basisbibliothek, die beim Aufruf der Programmierumgebung immer geladen wird. Aus diesem Grunde gibt es eine Reihe vordefinierter Klassentypen wie `STRING` und `ARRAY`, die zwar keine einfachen Typen sind, aber für die meisten Zwecke wie einfachen Typen behandelt werden können. Eine genaue Beschreibung aller vordefinierten Typen findet man üblicherweise in den Unterlagen des aktuellen Compilers.

3.2.4 Kunden, Lieferanten und Selbstreferenz

Enthält eine Klasse A eine Deklaration der Form `attribut:`B, so sagt man auch, daß die Klasse A von der Klasse B *kauft* bzw. daß B an A *verkauft*. Diese eher kaufmännische Terminologie wurde bewußt in die Konzeption von Eiffel übernommen, da das Prinzip von Liefern und Kaufen am besten die Denkweise der unabhängig operierenden Klassen ausdrückt.

Definition 3.2.1 (Kunden und Lieferanten) *Eine Klasse A heißt Kunde (Client) der Klasse B, wenn A eine Deklaration der Form* `entity:`B *enthält. B heißt in diesem Fall Lieferant (Supplier) von A.*

Der Begriff *entity* (Größe) in der obigen Definition ist das objektorientierte Gegenstück zum Begriff der Variablen in konventionellen Sprachen, umfaßt aber mehr als dieser, z.B. die Attribute einer Klasse. Eine vollständige Definition geben wir in Abschnitt 3.3.3, nachdem wir alle anderen Arten vorgestellt haben.

Wichtig ist, daß eine Klasse ihr eigener Kunde sein kann. So könnte man z.B. die Klasse `PERSON` erweitern um eine Angabe der Eltern, welche natürlich wieder Personen sind:

```
class PERSON feature
     name, vornamen: STRING;
     geburtsjahr, todesjahr: INTEGER;
     nationalität: STRING;
     vater, mutter: PERSON
end -- class PERSON
```

Abbildung 3.8: *Klassendefinition mit Selbstreferenz*

Prinzipiell wird damit die Möglichkeit gegeben, daß ein Objekt der Klasse `PERSON` auf sich selbst verweist (Selbstreferenz), also ein *Zyklus* im Graphen der Objekte vorhanden ist. Im Falle der Eltern mach dies natürlich keinen Sinn. Es sind jedoch durchaus Klassen denkbar, die eine sinnvolle zyklische Struktur im Objektgraphen erlauben, wie z.B. die folgende vereinfachte Klasse von Bankkunden, die einen Bürgen angeben müssen.

```
class BANKKUNDE feature
     name, vornamen: STRING;
     bürge: BANKKUNDE
end -- class BANKKUNDE
```

Diese Klassendefinition erlaubt z.B. die folgende durchaus realistische zyklische Objektstruktur.

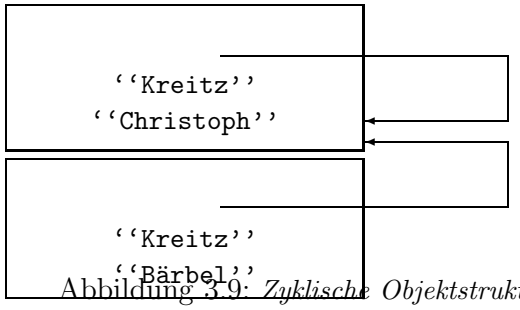


Abbildung 3.9: *Zyklische Objektstruktur*

Das Vorhandensein eines solchen Zyklus folgt nicht notwendig aus der Existenz einer sich selbst beliefernden Klasse. Allerdings ist es richtig, daß der Graph der Objekte entweder einen Zyklus haben muß oder einen

leeren Verweis (d.h. von machen Personen sind die Eltern unbekannt), da im Rechner keine unendliche Menge von Objekten verarbeitet werden können.

3.3 Routinen

In den bisher betrachteten Klassen haben wir nur Attribute definiert, welche die Struktur der Objekte erklären, die in dieser Klasse zusammengefaßt werden. Die Darstellung der Exemplare eines Typs ist aber nicht die einzige Dienstleistung, die ein abstrakter Datentyp bereitstellt. Neben der Struktur der Exemplare gehören zu einem abstrakten Datentyp auch Operationen auf diesen Exemplaren, die nach außen zur Verfügung gestellt werden. Diese Operationen werden in Programmiersprachen durch sogenannte *Routinen* beschrieben. Es gibt zwei Arten von Routinen.

- *Prozeduren* können durch Ausführung einer Aktion den *Zustand* eines Objektes *ändern.*
- *Funktionen* berechnen *Werte*, die sich aus dem Zustand der Objekte ergeben.

Dabei verstehen wir unter dem *Zustand* eines Objektes zu einem beliebigen Zeitpunkt der Ausführung eines Softwaresystems die Gesamtheit der Werte seiner Komponenten zu diesem Zeitpunkt. Ein Prozeduraufruf kann also die Werte von einem oder mehreren Komponenten eines Objektes ändern, während ein Funktionsaufruf einen Wert liefert, der aus den Werten der Komponenten eines Objektes berechnet wurde.

3.3.1 Aufruf von Routinen

Wir hatten bereits erwähnt, daß es – von außen betrachtet – keinen Unterschied macht, ob ein Wert direkt durch den Zugriff auf eine Komponente oder durch die Berechnung einer Funktion bestimmt wird. Aus diesem Grunde verwendet Eiffel für *alle* Operationen, seien es nun Zugriffe auf Komponenten, Aufrufe von Funktionen oder Aufrufe von Prozeduren die gleiche Syntax, nämlich eine *Punktnotation*.

Beispiel 3.3.1 *Nehmen wir an wir hätten für die Klasse* `PERSON` *die folgenden zusätzlichen features definiert:*

Die Funktion `anzahl_vornamen` *soll aus der Komponente* `vornamen` *die Anzahl der darin enthaltenen Vornamen bestimmen. Die Funktion* `alter` *soll bei Eingabe einer Jahreszahl das Alter der Person berechnen, sofern diese dann noch lebt. Die Prozedur* `setze_todesjahr` *soll bei Eingabe einer Jahreszahl das Todesjahr einer Person auf das angegebene Jahr setzen.*

Es bezeichne `p` *ein bereits existierendes Objekt*[8] *der Klasse* `PERSON`*. Dann liefert*

- `p.vornamen` *die aktuelle Komponente von* `p`*, welche dem Attribut* `vornamen` *entspricht,*
- `p.anzahl_vornamen` *die aktuelle Anzahl der Vornamen des Objektes* `p`*,*
- `p.alter(1993)` *das berechnete Alter der mit* `p` *bezeichneten Person im Jahre 1993,*
- `p.setze_todesjahr(1993)` *eine Veränderung des Zustandes von* `p`*: die Komponente, welche dem Attribut* `todesjahr` *entspricht wird auf 1993 gesetzt.*

Man beachte, daß in allen 4 Fällen die gleiche Notation `entity.operation(argumente)` . Sie bedeutet, daß die angegebene Operation mit den Argumenten auf das durch `entity` (*Größe*) bezeichnete Objekt angewandt wird. Es ist egal, ob diese Operation eine Funktion, eine Prozedur, oder etwa nur ein Komponentenzugriff ist – nach außen sieht es immer gleich aus. Dies entspricht einem der Grundgedanken der *Datenkapselung*: den Benutzer geht es nichts an, *wie* eine Klasse ihre Dienstleistungen ausführt – nicht einmal, ob dies die Lieferung einer gespeicherten Information oder eine Berechnung ist.

[8] Um genau zu sein: wenn eine Variable `p` ein bereits existierendes Objekt bezeichnet, dann bedeutet das gemäß der in Abschnitt 3.2.3 besprochenen Denkweise natürlich, daß sie ein *Verweis* auf dieses Objekt ist, wenn der Typ dieser Variablen kein einfacher Datentyp ist. Hierauf kommen wir später noch im Detail zu sprechen.

Entwurfsprinzip 3.3.2 (Punktnotation) *Alle* Aufrufe *von Funktionen, Prozeduren und Komponentenzugriffen auf ein Objekt werden in* Eiffel *einheitlich in der Notation* `entity.operation(argumente)` *ausgedrückt.*

Eine Operation `entity.operation(argumente)` greift auf ein Objekt über den Verweis zu, der durch `entity` bezeichnet wird. Damit dies fehlerfrei geschehen kann, muß das zugehörige Objekt natürlich existieren, d.h. der zu `entity` gehörende Verweis *darf nicht leer sein.* Dies gilt für alle Operationen, insbesondere auch für Zugriffe auf Komponenten. Der Versuch, auf ein Merkmal eines leeren Verweises zuzugreifen, ist einer der häufigsten Ursachen für Laufzeitfehler, die bei der Eiffel-Programmierung auftauchen.

In "klassischen" Programmiersprachen wie Pascal wird anstelle von `entity.operation(argumente)` die Notation `operation(objekt,argumente)` benutzt. Bei einer ersten Betrachtung scheint diese Form symmetrischer zu sein als die von Eiffel. Jedoch benutzt Pascal beim Zugriff auf Komponenten eines Verbundes ebenfalls die Punktnotation und verlangt also vom Benutzer, die Unterscheidung zwischen Zugriff und Funktion vorzunehmen. Eiffel ist da einheitlicher. Der spezielle Grund für die Punktnotation, bei der das Objekt am Anfang jedes Aufrufs genannt wird, ist wiederum die objektorientierte Denkweise: es ist das wichtigste zu sagen, auf *welchem Objekt* eine Operation ausgeführt wird. Aus diesem Grunde soll das Objekt durch die Notation besonders hervorgehoben werden.

3.3.2 Definition von Routinen

Nachdem wir gesehen haben, wie man Routinen aufruft, wollen wir nun zeigen, wie man sie definiert. Dazu muß als wichtigstes herausgehoben werden, daß Routinen prinzipiell nur als Dienstleistungen einer Klasse definiert werden können und nicht etwa als unabhängige "Unterprogramme".[9] Routinen werden daher gleichberechtigt zu den Attributen einer Klasse als Features aufgeschrieben. Abbildung 3.10 zeigt eine abstrahierte Implementierung der in Beispiel 3.3.1 benutzten Routinen innerhalb der Klasse `PERSON`.

```
class PERSON
   feature
     name, vornamen: STRING;
     geburtsjahr, todesjahr: INTEGER;
     nationalität: STRING;
     vater, mutter: PERSON;
     anzahl_vornamen: INTEGER is  -- Anzahl der Vornamen bestimmen
          do
               Result :=  Anzahl der Vornamen in  vornamen
          end; -- anzahl_vornamen
     alter(jahr:INTEGER): INTEGER is -- Alter im gegebenen Jahr bestimmen
          do
               Result := jahr - geburtsjahr
          end; -- alter
     setze_todesjahr(jahr:INTEGER) is -- todesjahr auf jahr setzen
          do
               todesjahr := jahr
          end -- setze_todesjahr
end -- class PERSON
```

Abbildung 3.10: *Klassendefinition mit Routinen*

Neben den Attributen `name, vornamen, geburtsjahr, todesjahr, nationalität, vater` und `mutter` enthält die Klasse `PERSON` drei Routinen, die man an dem Vorhandensein der Schlüsselwortfolge **is**...**do**...**end** erkennt. Diese Folge begrenzt den *Rumpf* einer Routine. Eine Routine kann *formale Argumente* in Klammern

[9] Die im Abschnitt 3.3.4 vorgestellten Operationen, die auf allen Klassen Gültigkeit haben, sind daher in Eiffel als Dienstleistungen einer Klasse `ANY` realisiert, die ihre Konzepte auf *alle* Klassen vererbt.

haben wie z.B. "(jahr:INTEGER)" in den Routinen `alter` und `setze_todesjahr`, aber das muß nicht sein. Formale Argumente werden beschrieben durch ihren Namen und ihren Typ, getrennt durch einen Doppelpunkt.

Die Routinen `alter` und `anzahl_vornamen` sind zusammen mit einem *Ergebnistyp* deklariert, der – getrennt durch einen Doppelpunkt – hinter der Argumentenliste auftritt. Dies bedeutet, daß beide Routinen als *Funktionen* deklariert werden, während die Routine `setze_todesjahr` eine *Prozedur* ohne Ergebnistyp ist.

Der Rumpf einer Routine besteht aus einer *Folge von Anweisungen*, die durch ein Semikolon voneinander getrennt sind und von den Schlüsselworten **do** und **end** begrenzt werden.[10] In Eiffel gibt es nur wenige Arten von Anweisungen. Neben der kursiv geschriebenen, nicht näher erklärten Bestimmung der Anzahl von Vornamen verwenden die Routinen von `PERSON` nur die im *Zuweisung* `:=`, die einer Größe den Wert eines Ausdrucks zuweist (siehe Abschnitt 3.3.4). Dabei wird – neben den Attributnamen – auch der besondere Name `Result` als Bezeichner für Größen benutzt.

`Result` bezeichnet das Ergebnis einer Funktion. Beim Eintritt in die Funktion wird `Result` mit einem fest vereinbarten Standardwert (siehe Tabelle 3.12) initialisiert, beim Verlassen der Funktion wird als Ergebnis der derzeitige Wert von `Result` als Funktionswert zurückgegeben. So ist z.B. der Funktionswert von `alter(1993)` genau die Differenz von 1993 und der Komponente `geburtsjahr` des angesprochenen Objektes.

Prozeduren verändern Werte gemäß der in ihnen enthaltenen Anweisungen. So setzt zum Beispiel die Prozedur `setze_todesjahr(1993)` die Komponente `todesjahr` auf 1993 (die Notwendigkeit einer solchen Prozedur diskutieren wir im Abschnitt 3.4). Prozeduren liefern keine Ergebnisse.

Routinen einer Klasse dürfen sich in ihrer Implementierung auch auf andere Routinen derselben Klasse beziehen. Man darf zum Beispiel in einer weiteren Routine der Klasse `PERSON` die Funktion `alter` aufrufen.

Den nach außen hin uninteressanten Unterschied zwischen Attributen, Funktionen und Prozeduren kann man in einer Deklaration leicht an der Syntax einer **feature**-Deklaration erkennen.

- Folgt auf den Namen des Features `f` bis zum Semikolon oder **end** nur die Typdeklaration wie in
  `f:Typ`
  so beschreibt `f` ein Attribut.
- Folgt auf den Namen des Features eine Argumentenliste, eine Typdeklaration und die Schlüsselwortfolge **is**...**do**...**end** wie in
  `f(`x_1`:`A_1`,..`x_n`:`A_n`):Typ` **is do** `...` **end**
  so beschreibt `f` eine Funktion. Das ist auch dann der Fall, wenn die Argumentenliste fehlt, wie in
  `f:Typ` **is do** `...` **end**
- Folgt auf den Namen des Features eine Argumentenliste und die Schlüsselwortfolge **is**...**do**...**end** wie in
  `f(`x_1`:`A_1`,..`x_n`:`A_n`)` **is do** `...` **end**
  so beschreibt `f` eine Prozedur. Das ist auch dann der Fall, wenn die Argumentenliste fehlt, wie in
  `f` **is do** `...` **end**
- Eine Besonderheit, welche die Verwendung des Schlüsselwortes **is** notwendig macht, haben wir noch nicht besprochen. Eine Deklaration
  `f:Typ` **is** `Wert`
  beschreibt ein *Konstanten-Attribut*, welches zusammen mit einem festen Wert deklariert wird.

3.3.3 Lokale Variablen

In vielen Routinen ist es notwendig, Zwischenwerte zu berechnen, bevor die eigentliche Aktion ausgeführt wird. So ist es z.B. sinnvoll, in der Funktion `anzahl_vornamen` zunächst die Liste der einzelnen Vornamen aus

[10] Diese Form ist nur eine Heraushebung des Verwendungszwecks einer Folge von Anweisungen. Mathematisch betrachtet ist "**do** `anweisung`$_1$`; anweisung`$_2$`; ...;anweisung`$_n$ **end**" einfach eine Liste `<anweisung`$_1$`, anweisung`$_2$`, ..., anweisung`$_n$`>`. Die Verwendung derartiger Notationen aber würde ein Programm unlesbar machen.

der Komponente `vornamen` (durch Suche nach Leerzeichen) zu extrahieren und dann deren Größe mithilfe gewöhnlicher Listenfunktionen zu berechnen. Hierzu ist es notwendig, *lokale* Variablen zu vereinbaren, also Variablen, die nur für interne Berechnungen der Funktion `anzahl_vornamen` verwendet werden. Die Verwendung lokaler Variablen entspricht der Verwendung von Schmierzetteln bei der Lösung komplizierterer Aufgaben – sie werden nur zum Finden der Lösung gebraucht, gehören aber nicht zur eigentlichen Lösung dazu. Lokale Variablen machen die Implementierung von Routinen übersichtlicher und oft effizienter.

```
anzahl_vornamen: INTEGER is
          -- Anzahl der Vornamen bestimmen
     local
          vornamen_liste : ARRAY[STRING]
     do
          vornamen_liste :=  einzelne Vornamen aus  vornamen
          Result :=  Länge der Liste  vornamen_liste
     end; -- anzahl_vornamen
```

Abbildung 3.11: *Routinendeklaration mit lokalen Variablen*

In Eiffel werden lokale Variablen durch das (optionale) Schlüsselwort **local** am Anfang eines Routinenrumpfes deklariert. In Abbildung 3.11 wird also innerhalb der Routine `anzahl_vornamen` eine lokale Variable `vornamen_lis` vom Typ `ARRAY[STRING]` definiert (über arrays sprechen wir im Abschnitt 3.6.3). Diese Variable ist nur innerhalb der Routine `anzahl_vornamen` bekannt und kann auch innerhalb der Klasse `PERSON` von keiner anderen Routine benutzt werden.

Lokale Variablen sind eine weitere Art von Größen, die innerhalb von `Eiffel` verwendet werden, um Objekte zu bezeichnen. Die folgende Definition summiert alle Formen, die eine Größe annehmen kann.

Definition 3.3.3 (Entities / Größen) *Eine Größe (entity) ist ein Bezeichner für Objekte, der folgende Formen annehmen kann*

- *ein Klassenattribut*
- *eine lokale Variable einer Routine*
- *ein formales Argument einer Routine*
- `Result`*, eine vordefinierte Größe für das Ergebnis einer Funktion.*

3.3.4 Standardoperationen für alle Klassen

Zusätzlich zu den in einer Klasse definierten Routinen gibt es in Eiffel eine Reihe vordefinierter Operationen, die in allen Klassen angewandt werden können(, was natürlich auch bedeutet, daß deren Namen nicht mehr für andere Routinen benutzt werden können). Diese werden benutzt zur *Erzeugung* neuer Objekte einer Klasse, zur *Zuweisung* von Werten, zum *Zugriff* auf Komponenten, zur *Auflösung* eines Verweises auf ein Objekt, zur *Überprüfung des Verweiszustandes*, *Duplizierung* von Objekten, zum *Kopieren* von Objektzuständen und zum *Vergleich* von Objekten.

Erzeugung von Objekten: In Abbildung 3.7 hatten wir die Klasse `BUCH` mit einem Attribut `autor` vom Typ `PERSON` definiert (`BUCH` ist also Kunde von `PERSON`). Da `PERSON` selbst ein Klassentyp ist, enthalten alle Objekte der Klasse `BUCH` an der entsprechenden Komponente einen Verweis. Solch ein Verweis kann entweder auf ein konkretes Objekt zeigen oder leer sein – also den Wert `Void` haben.

Sinnvollerweise ist ein solcher Verweis leer, solange man noch nichts anderes festgelegt hat. Um dies aber zu ändern, könnte man der entsprechenden Komponente einen Wert zuweisen (siehe unten). Aber da Objekte nicht von selbst entstehen können, muß man irgendwann damit anfangen, Objekte zu erzeugen

und diese erzeugten Objekte mit Verweisen zu verbinden. Hierzu gibt es in Eiffel eine Erzeugungsprozedur, die mit `!!` bezeichnet wird. Um zum Beispiel ein neues Objekt vom Typ `PERSON` zu erzeugen und mit der Komponente `autor` eines gegebenen Buchobjektes zu verbinden schreibt man:

```
!!autor
```

Eine Erzeugungsoperation versetzt den Verweis aus dem leeren Zustand `Void` in den Zustand **created** (*erzeugt*) und der Verweis ist genau dann in diesem Zustand, wenn er mit einem Objekt verbunden ist.

Auch das neu erzeugte Objekt muß natürlich einen Anfangszustand haben. Aus diesem Grunde ist es nötig, nicht nur für Verweise sondern auch für alle anderen Komponenten eines Objektes einen *Initialwert* festzulegen, denn es macht wenig Sinn, wenn man diesen Wert dem Zufall überläßt. In Eiffel gilt die Konvention, daß Komponenten eines Objektes gemäß dem Typ des entsprechenden Attributs nach der Tabelle in Abbildung 3.12 initialisiert werden.

Datentyp	Initialwert
INTEGER	0
BOOLEAN	false
CHARACTER	*ASCII-Null-Symbol*
REAL	0.0
DOUBLE	0.0
Klassentypen	Void

Abbildung 3.12: *Standardinitialwerte in Abhängigkeit vom Datentyp*

Es besteht auch die Möglichkeit, in einer Klassendeklaration vom Standard abweichende Initialwerte für die Erzeugung zu vereinbaren. Darauf werden wir in Abschnitt 3.3.6 zurückkommen.

Objekte existieren nicht, solange sie nicht explizit zur Laufzeit eines Systems durch eine Erzeugungsoperation erzeugt wurden. Auf diese Art wird vermieden, daß Objekte bei der Abarbeitung der Deklaration automatisch erzeugt werden müssen. Dieses würde nämlich bei Objekten einer Klasse mit Selbstreferenz, z.B. der Klasse `PERSON` in Abbildung 3.8, sofort dazu führen, daß immer wieder neue Objekte erzeugt werden müssen, da zu jeder Person ja Vater und Mutter benötigt werden. Wir wären also in einer unendlichen Schleife, bevor wir überhaupt mit der eigentlichen Verarbeitung anfangen könnten.

Zuweisung: Wie jede andere Programmiersprache kennt Eiffel das Konzept der Zuweisung von Werten an die Komponenten eines Objektes. Diese Anweisung wird mit dem Symbol `:=` ausgedrückt. Der Befehl

```
erscheinungsdatum := 1993
```

weist also der Komponente eines gegebenen Buchobjekts, die zu dem Attribut `erscheinungsdatum` gehört, den Wert 1993 zu.

Wichtig ist, daß bei der Zuweisung von *Objekten* zu einer Komponente nicht etwa das Objekt selbst, sondern nur der Verweis auf dieses Objekt zugewiesen wird. Dies entspringt wieder der in Abschnitt 3.2.3 besprochenen Denkweise, daß jede Bearbeitung nichtelementarer Objekte ausschließlich über Verweise geschieht. Bezeichnet also `person_1` ein Objekt vom Typ `Person`, dann weist der Befehl

```
autor := person_1
```

in Wirklichkeit der Komponente `autor` den gleichen Verweis zu, der auch in `person_1` eingetragen ist, denn genau besehen ist `person_1` ein Verweis auf das Objekt und nicht das Objekt selbst.

Für die Zuweisungen von Werten an die Komponenten eines Objektes gibt es allerdings in Eiffel aufgrund des Geheimnisprinzips (siehe Abschnitt 3.4) sehr starke Einschränkungen. Erlaubt ist eine Zuweisung nur *innerhalb* einer Klassendefinition. Von außen ist eine Veränderung nur über Prozeduren möglich, die durch die Klasse zur Verfügung gestellt werden.

Komponentenzugriff: Der *direkte Zugriff* auf Komponenten eines Objektes geschieht über den Namen des entsprechenden Attributs. Ist also `person_1` ein bereits bekanntes Objekt vom Typ `Person`, dann liefert

```
person_1.name
```

die Zeichenkette, die unter der zu `name` zugehörigen Komponente gespeichert ist. Bei Attributen, deren Typ eine Klasse ist, wird natürlich der Verweis geliefert und nicht etwa das Objekt selbst.

```
person_1.vater
```

liefert also einen Verweis auf das Objekt, welches den Vater von `person_1` darstellen soll.

Auch bei der Verwendung von Komponentenzugriffen gilt die obengenannte Einschränkung des Geheimnisprinzips. Ein Komponentenzugriff gilt ebenso wie jede andere Routine als eine Dienstleistung, die von der Klasse bereitgestellt werden muß. Ein Zugriff auf eine beliebige Komponente ist daher nur dann möglich, wenn dies bei der Klassendeklaration explizit vereinbart wurde. Näheres dazu besprechen wir im Abschnitt 3.4.

Auflösen eines Verweises: Manchmal kann es nötig sein, einen Verweis von dem ihm zugeordneten Objekt wieder zu lösen. Dies geschieht durch Zuweisung des Wertes `Void`. Um also die Komponente `autor` eines gegebenen Buchobjektes wieder von ihrem Objekt zu lösen, schreibt man:

```
autor := Void
```

Diese Auflösung des Verweises versetzt also Verweis aus dem Zustand **created** zurück in den leeren Zustand. Wichtig ist aber, hierdurch ausschließlich der Zustand des Verweises verändert wird und nicht etwa das Objekt selbst. Das Objekt, mit dem die Komponente `autor` verbunden war, wird nicht angerührt. Insbesondere wird es auch nicht gelöscht, sondern bleibt ggf. über andere Verweise noch zugreifbar.

Auflösung von Verweisen darf auch nicht verwechselt werden mit Anweisungen, die Speicherplatz an das Betriebssystem zurückgeben (wie etwa `dispose` in Pascal). Solche Anweisungen sind sehr gefährlich, da hierdurch eventuell Objekte zerstört werden, die noch über andere Verweise zugreifbar waren. In Eiffel (und auch z.B. in Lisp) wird die Speicherplatzverwaltung automatisch durch die Programmierumgebung durchgeführt und nicht etwa durch den Programmierer selbst.

Überprüfung des Verweiszustandes: Um den Zustand eines Verweises zu überprüfen, der mit einer Komponente verbunden ist, bietet Eiffel einen einfachen Test an. Der boolesche Ausdruck

```
autor = Void
```

liefert den Wert `true`, wenn der Verweis der Komponente `autor` leer ist, und ansonsten den Wert `false`.

Duplizierung von Objekten: Bei der Zuweisung von Objekten mittels `:=` haben wir gesehen, daß nur die Verweise kopiert werden, aber das genannte Objekt nach wie vor nur einmal vorhanden ist. Dies ist im Normalfall die sinnvollste Vorgehensweise. In manchen Fällen ist es aber nötig, eine Kopie eines bereits bekannten Objektes (und nicht nur des Verweises) anzulegen, z.B. um daraus ein Objekt zu erzeugen, das fast identisch mit dem ersten ist. Hierzu ist die Funktion `clone` da. Ist `person_1` ein bereits bekanntes Objekt vom Typ `Person`, dann liefert der Aufruf

```
clone(person_1)
```

einen Verweis auf ein *neues* Objekt, dessen Komponenten mit denen von `person_1` identisch sind. Ein solcher Aufruf macht allerdings nur Sinn im Zusammenhang mit einer Zuweisung, da sonst dieses neue Objekt nicht erreichbar wäre. Die Zuweisung

```
autor := clone(person_1)
```

hat also zwei Effekte. Sie erzeugt ein neues, mit `person_1` identisches Objekt und weist der Komponente `autor` als Wert einen *Verweis* auf dieses Objekt zu (eventuell bestehende Verweise werden also aufgelöst). `clone` und die mit `!!` zusammenhängenden Erzeugungsprozeduren sind die einzigen Operationen, welche Objekte erzeugen.

Bei der Erzeugung von Objekten mittels `clone(person_1)` ist zu berücksichtigen, daß Verweise, die im Objekt `person_1` enthalten sind, *als Verweise* identisch kopiert werden und nicht etwa zu jedem Verweis ein weiteres Objekt kopiert wird. Man sagt auch, daß `clone` eine *flache* (*shallow*) Kopie erzeugt: das neue Objekt ist wirklich in allen Komponenten identisch mit `person_1`.

In manchen Anwendungen ist es jedoch notwendig, eine komplette Kopie der gesamten Objektstruktur zu erzeugen, die an der genannten Stelle beginnt. Eine solche "tiefe" Kopie wird durch die Funktion `deep_clone` generiert. Genau besehen erzeugt also der Aufruf

```
deep_clone(person_1)
```

einen Verweis auf ein *neues* Objekt, dessen Komponenten mit einfachen Datenkomponenten mit denen von `person_1` identisch sind und dessen Verweiskomponenten wiederum auf neue Objekte verweisen, die mit den entsprechenden in `person_1` angesprochenen Objekten bis auf Verweise identisch sind usw.

Kopieren von Objektzuständen: Neben `clone`, der Erzeugung neuer Objekte, gibt es eine weitere sinnvolle Möglichkeit, Objekte inhaltlich zu kopieren, nämlich bestehenden Objekten die Zustände anderer Objekte zuzuweisen. Dies geschieht mit der Prozedur (nicht Funktion!) `copy`. Während der Befehl `autor := person_1` der Größe `autor` nur den in `person_1` enthaltenen Verweis zuweist und `autor := clone(person_1)` ein zuvor nicht vorhandenes Objekt erzeugt, wird durch

```
autor.copy(person_1)
```

ein bereits existierendes durch `autor` bezeichnetes Objekt verändert (deshalb auch die Verwendung der Punktnotation: die Operation `copy` arbeitet auf dem Objekt, auf das `autor` verweist). Dieses Objekt erhält in jeder Komponente den Wert der entsprechenden Komponente des durch `person_1` bezeichneten Objekts. Der in `autor` enthaltene *Verweis* wird dagegen nicht verändert. Wichtig ist aber, daß `autor` *keinen leeren Verweis* enthalten darf.

Genauso wie `clone` erzeugt `copy` eine flache Kopie von Objekten. Wird eine tiefe Kopie benötigt, so ist die Prozedur `deep_copy` anzuwenden, wie zum Beispiel in

```
autor.deep_copy(person_1)
```

Vergleich von Objekten: Wegen des schon öfter angesprochenen Unterschiedes zwischen einem Objekt und dem Verweis darauf gibt es zwei Möglichkeiten, Objekte zu vergleichen. Die gewöhnliche Gleichheit `=` stellt die Frage, ob die beiden Objekte identisch sind. Die Funktion `equal` testet, ob die beiden Objekte in jeder Komponente dieselben Werte haben. Das hat wieder zu tun mit der Tatsache, daß Größen, welche Objekte bezeichnen, in Wirklichkeit Verweise auf diese Objekte sind. Gleichheit der Größen bedeutet, daß die Verweise gleich sind, also daß auf *dasselbe* Objekt verwiesen wird. Der Ausdruck

```
autor = person_1
```

liefert `true`, wenn `autor` und `person_1` dasselbe Objekt bezeichnen, und sonst `false`. Das Umgekehrte, der Test auf Ungleichheit der Objekte, wird durch `autor /= person_1` ausgedrückt.

Die Funktion `equal` geht einen Schritt weiter. Statt einfach die Verweise zu vergleichen, betrachtet sie die *Zustände* der durch `autor` und `person_1` bezeichneten Objekte.

```
equal(autor,person_1)
```

liefert `true`, wenn entweder die mit `autor` und `person_1` verbundenen Verweise beide leer sind, oder wenn beide bezeichneten Objekte in jeder Komponente übereinstimmen (das bedeutet aber, daß in Komponenten enthaltene Verweise wieder auf dasselbe Objekt zeigen müssen). Wenn `autor = person_1` gilt, dann gilt auch `equal(autor,person_1)` aber nicht umgekehrt.

Auch `equal` führt einen "flachen" Vergleich durch. `equal(autor,person_1)` gilt nur, wenn Verweise in Komponenten der durch `autor` und `person_1` bezeichneten Objekte jeweils auf dasselbe Objekt zeigen. Der Vergleich, ob die bezeichneten Objekte *strukturell* identisch sind, wird aufgerufen mit

```
deep_equal(autor,person_1)
```

Gegenüber früheren Versionen von Eiffel (vor allem gegenüber dem Buch "Object-oriented Software Construction", aber auch gegenüber dem Skript von Prof. Henhapl) haben sich in der Syntax dieser Operationen einige Änderungen ergeben. Der Grund hierfür war laut [Meyer, 1992] hauptsächlich eine Vereinheitlichung der Semantik für die Punktnotation (nur anwendbar auf nichtleere Verweise), die in den alten Formen etwas unsystematisch war.[11] Die folgende Tabelle beschreibt die Zusammenhänge.

Frühere Versionen	Eiffel 3
`autor.Create`	`!!autor`
`autor.Forget`	`autor := Void`
`autor.Void`	`autor = Void`
`autor.Clone(person_1)`	`autor := clone(person_1)`
`autor.Equal(person_1)`	`equal(autor,person_1)`

Natürlich gibt es in Eiffel viele weitere vordefinierte Operationen, mit denen man in Eiffel so rechnen kann, wie in jeder anderen Programmiersprache. Da aber Arithmetik und ähnliche Operationen nichts spezifisches für die objektorientierte Denkweise sind, verschieben wir die Besprechung der "konventionellen Programmierkonzepte" auf die nächsten Kapitel.

3.3.5 Das aktuelle Exemplar

Eine der herausragenden Eigenschaften der objektorientierte Denkweise ist, daß jedes Programmkonstrukt sich auf ein Objekt bezieht. Man schreibt nie ein Programmstück, welches nur sagt "tue dies", sondern gibt allen Befehlen die Form "tue dies *an diesem Objekt*".

Um zu erkennen, auf welches Objekt sich ein Stück Eiffel-Text bezieht, muß man berücksichtigen, daß jeder Eiffel-Text Teil von genau einer Klasse ist. Ein höheres Konstrukt als Klassen gibt es in Eiffel nicht. Jede Klasse beschreibt einen Datentyp, dessen Exemplare zur Ausführungszeit die Objekte sind. Der Klassentext zeigt die Merkmale, die all diese Objekte gemeinsam haben, indem ein typisches Exemplar der Klasse, das sogenannte *aktuelle Exemplar*, beschrieben wird. Daher bezieht sich jeglicher Eiffel-Text auf genau dieses aktuelle Exemplar, d.h. jedes Vorkommen des Attributs `autor` innerhalb der Klasse `BUCH` *bezieht sich auf die Komponente des aktuellen Buch-Objektes*, welche dem Attribut `autor` entspricht. Die Zuweisung

```
erscheinungsdatum := 1993
```

in der Klasse `BUCH` bedeutet also, daß der Komponente `erscheinungsdatum` des aktuellen Exemplars von `BUCH` der Wert 1993 zugewiesen wird.

Oft aber ist es auch notwendig, auf andere Objekte zu verweisen. Hierfür verwendet man in Eiffel die bereits angesprochene Punktnotation.

Definition 3.3.4 (Qualifiziertes Vorkommen von Attributen) *Es sei* `entity` *ein Attribut vom Klassentyp K und* `a` *ein Attribut der Klasse K, dann bezeichnet* `entity.a` *die Komponente* `a` *des Objektes, welches zur Laufzeit mit* `entity` *verbunden wird.*

`entity.a` *heißt qualifiziertes Vorkommen von* `a`.

Die Punktnotation kann beliebig verschachtelt werden. Man beachte jedoch, daß selbst qualifizierte Vorkommen von `a` das Konzept des aktuellen Exemplars (nämlich `entity`) benötigen. Beispiele haben wir bereits im vorhergehenden Abschnitt gegeben.

Ein qualifiziertes Vorkommen eines Attributs wird in Eiffel immer als eine *Funktion* behandelt und nicht etwa als Name einer Komponente. Aus diesem Grunde kann man nur den Komponenten des aktuellen Exemplars

[11] Konzeptionell weist Eiffel 3 keine Änderungen gegenüber früheren Versionen auf. Änderungen beschränken sich auf eine Verbesserung der Syntax (konsistenter und deutlicher), die Klärung semantischer Unsauberkeiten, und die Hinzunahme einiger neuer sinnvoller Konstrukte als vordefinierte Bestandteile der Sprache

über den Namen eines Attributs Werte zuweisen. Wertzuweisungen von außerhalb der Klasse, die den Typ eines Objektes beschreibt, sind nicht möglich. Dies entspricht dem Geheimnisprinzip, welches wir in Abschnitt 3.4 ausführlicher diskutieren werden.

Die Bedeutung des aktuellen Exemplars erklärt die dezentralisierte Natur objektorientierter Programmierung. Es gibt *kein übergeordnetes Hauptprogramm* im Sinne der klassischen, zentralistisch denkenden Programmiersprachen. Stattdessen haben wir im Programm eine Menge von Klassen und zur Laufzeit eine Menge von Objekten, welche über Serviceleistungen der Klassen Operationen aufeinander ausführen können. Es ist klar, daß man für ein ausführbares System festlegen muß, wo dieser Ausführungsprozeß beginnen soll. Diese Festlegung aber geschieht spät und unabhängig von dem eigentlichen Programm (siehe Abschnitt 3.9).

In den meisten Fällen bleibt das aktuelle Exemplar implizit, d.h. es wird nicht direkt im Programmtext genannt. Manchmal ist es aber nötig, das aktuelle Exemplar im Programmtext explizit zu kennzeichnen – zum Beispiel um einen Verweis eines Objekts auf sich selbst zu ermöglichen. Hierzu stellt Eiffel die Funktion `Current` zur Verfügung, welche *das aktuelle Objekt der den Aufruf umschließenden Klasse* bezeichnet. Ist das aktuelle Exemplar z.B. ein Objekt der Klasse `BANKKUNDE` (vgl. Abschnitt 3.2.4), so bewirkt der Befehl

```
bürge := Current
```

daß die Komponente `bürge` des aktuellen Objektes auf sich selbst verweist, daß also der beschriebene Bankkunde für sich selbst bürgt. Man beachte jedoch, daß `Current` eine Funktion ist, die einen Wert liefert, und nicht etwa eine Größe, der man einen Wert zuweisen könnte.

3.3.6 Nicht-standardmäßiges Erzeugen

In den bisherigen Beispielen ergibt die Erzeugungsoperation jedesmal, wenn sie für irgendeine Größe einer Klasse aufgerufen wird, das gleiche (*nicht dasselb*e) Objekt. Für die Komponente `autor` eines Buchobjektes, die vom Typ `PERSON` ist, wird durch `!!autor` also immer ein Personenobjekt mit leeren Verweis auf Namen, Vornamen und Nationalität sowie Geburts- und Todesjahr 0 erzeugt. Da diese Form der Initialisierung hochgradig unflexibel ist, bietet Eiffel die Möglichkeit an, für jede Klasse eine (oder mehrere) vom Standard abweichende Initialisierungsprozedur zu deklarieren. Hierzu werden einfach einige der als features erklärten Routinen explizit als Initialisierungsprozeduren (*creators*) vereinbart. Eiffel (Version 3) benutzt hierzu das Schlüsselwort **creation**.

```
class PERSON
   creation
     init_mit_namen
   feature
     ⋮
     init_mit_namen(n,v:STRING) is  -- Initialisiere Person mit Vornamen und Namen
          do
               name := n;
               vornamen := v
          end; -- init_mit_namen
     ⋮
end -- class PERSON
```

Abbildung 3.13: *Klassendefinition mit Initialisierungsprozedur*

In Abbildung 3.13 haben wir die Klasse `PERSON` um die Initialisierungsprozedur `init_mit_namen` erweitert. Diese nimmt zwei Strings und initialisiert Namen und Vornamen eines neugeschaffenen Objektes entsprechend. Alle anderen Komponenten werden gemäß der Standardwerte initialisiert. Es ist durchaus möglich mehrere Initialisierungsprozeduren für verschiedene Zwecke zu deklarieren.

Ist in einer Klasse eine spezifische Initialisierungsprozedur `init` vereinbart, so kann sie ähnlich zur Syntax der Standardinitialisierung in der Form `!!entity.init(argumente)` aufgerufen werden. In unserem Beispiel wird also ein neues Personenobjekt, welches Bertrand Meyer darstellen soll, erzeugt durch

```
!!autor.init_mit_namen(Meyer","Bertrand")
```

Durch die Deklaration einer spezifischen Initialisierungsprozedur wird die *allgemeine Initialisierungsprozedur allerdings außer Kraft gesetzt*. Der Aufruf `!!autor` ist also nicht mehr erlaubt, wenn es für die Klasse Buch eine spezifische Initialisierungsprozedur gibt.

Durch das Konzept der Vererbung kommen weitere Möglichkeiten hinzu, die es erlauben auf die Initialisierungsprozeduren einer verwandten Klasse `ERBEN_KLASSE` ebenfalls zuzugreifen. Die Syntax des Aufrufs lautet in diesem Fall `!ERBEN_KLASSE!entity.init(argumente)` bzw., wenn es keine spezifische Initialisierungsprozedur gibt, `!ERBEN_KLASSE!entity`

Eine Prozedur, die durch **creation** als Initialisierungsprozedur deklariert wurde, darf auch wie eine gewöhnliche Prozedur – ohne `!!` – aufgerufen werden. Allerdings verhält sie sich dann auch wie eine gewöhnliche Prozedur. Ein Aufruf

```
autor.init_mit_namen(Meyer","Bertrand")
```

ist also erlaubt, erzeugt aber kein neues Objekt. Es werden lediglich Name und Vornamen enstprechend verändert.

Zum Abschluß dieses Abschnitts über Routinen fassen wir die wichtigsten Standardroutinen zusammen.

`!!entity`	Standard-Erzeugung von Objekten
`!!entity.init(argumente)`	Nicht-Standard-Erzeugung von Objekten
`entity := Ausdruck`	Zuweisung (limitiert)
`entity.attribut`	(Komponenten)Zugriff (limitiert)
`entity := Void`	Auflösen eines Verweises
`entity = Void`	Überprüfung des Verweiszustandes
`entity := clone(entity_1)`	(Flache) Duplizierung von Objekten
`entity := deep_clone(entity_1)`	Tiefe Duplizierung von Objekten
`entity.copy(entity_1)`	(Flache) Kopie von Objektinhalten
`entity.deep_copy(entity_1)`	Tiefe Kopie von Objektinhalten
`entity_1 = entity_2`	Vergleich von Objekten
`equal(entity_1,entity_2)`	(Flacher) Vergleich von Objektinhalten
`deep_equal(entity_1,entity_2)`	Tiefer Vergleich von Objekten
`Current`	Verweis auf das aktuelle Objekt

Abbildung 3.14: *Vordefinierte Operationen*

3.4 Das Geheimnisprinzip

Eine der wichtigsten Grundlagen der disziplinierten Software-Entwicklung ist das Prinzip, daß für den Benutzer eines Softwareproduktes die interne Verarbeitungsform irrelevant sein muß, ja sogar vor ihm geheim gehalten werden sollte, um einen möglichen inkompetenten Mißbrauch der Software zu vermeiden. Eine Realisierung dieses *Geheimnisprinzips* (*information hiding*) in einer Programmiersprache erleichtert die Entwicklung von Softwaresystemen, welche das Gütekriterium *Integrität* erfüllen, ohne daß dafür auf andere Kriterien verzichtet werden muß. In der Terminologie der Programmierung formuliert man dieses Prinzip wie folgt

Entwurfsprinzip 3.4.1 (Datenkapselung oder Geheimnisprinzip)

- *Die interne Darstellung der Daten muß nach außen unsichtbar sein.*

- *Die Veränderung und Auswertung der Daten darf nur über prozedurale Schnittstellen geschehen. Ein direktes Lesen und Schreiben ist nicht erlaubt, sondern nur über Funktionen und Prozeduren für Anfragen und Änderungen.*

Eiffel realisiert dieses Prinzip, indem *qualifizierte Vorkommen von Attributen als Funktionen* aufgefaßt werden, die nur einen Wert *liefern*, nicht aber eine Zuweisung von Werten zulassen. Wertzuweisungen sind nur *innerhalb* der deklarierenden Klasse, also nur über unqualifizierte Attribute des aktuellen Exemplars möglich.

Darüber hinaus müssen nicht alle Attribute nach außen hin zugänglich sein, da manche vielleicht nur für die interne Darstellung benötigt werden. So könnte es zum Beispiel sein, daß die Klasse `PERSON` die Vornamen einer Person intern als Liste einzelner Vornamen darstellt, nach außen aber nur als einen einzigen String liefert. Ob `vornamen` also ein Attribut ist oder eine Funktion, darf den *Benutzer* nicht interessieren.

Dieses Prinzip ist auch die Ursache dafür, daß der Aufruf (und die Deklaration) von Routinen und Attributen in Eiffel die gleiche Syntax haben. Es erlaubt dem Systementwickler, nachträglich die Implementierung einer Klasse zu ändern, ohne daß dies Konsequenzen für die Außenwelt hat. Kunden einer Klasse sind – was die Art der Verarbeitung angeht – von Änderungen überhaupt nicht betroffen. Es ändert sich allenfalls die Geschwindigkeit, mit der Instruktionen ausgeführt werden.

```
class PERSON
   creation
     init_mit_namen
   feature{}
     vornamen_liste: STRING;
   feature
     init_mit_namen(n,v:STRING) is ... do ... end;
     name: STRING;
     vornamen: STRING is ... do ... end;
     anzahl_vornamen: INTEGER is ... do ... end;
     geburtsjahr, todesjahr: INTEGER;
     alter(jahr:INTEGER): INTEGER is ... do ... end;
     setze_todesjahr(jahr:INTEGER) is ... do ... end;
     vater, mutter: PERSON;
     geschwister: ARRAY[PERSON] is ... do ... end;
     großeltern: ARRAY[PERSON] is ... do ... end;
     nationalität: STRING
end -- class PERSON
```

Abbildung 3.15: *Klassendefinition mit Datenkapselung*

Welche features für die Außenwelt sichtbar sein sollen, wird durch die genaue Form der **feature**-Klausel beschrieben. Features, die hinter dem Schlüsselwort **feature** {} genannt werden – in Abbildung 3.15 das feature `vornamen_liste` –, bleiben nach außen hin unsichtbar. Features, die hinter einem einfachen Schlüsselwort **feature** vorkommen (also alle anderen hier genannten features), sind für alle Kunden zugänglich.[12] Dies gilt sowohl für Attribute als auch für Routinen.

Eiffel erlaubt sogar eine feinere Unterscheidung, die auch die Verwendung der Klammern "{}" motiviert. Es kann nämlich sehr sinnvoll sein, die Zugangsberechtigung für bestimmte features abhängig zu machen von der Klasse, welche die Anfrage stellt. Dies realisiert den Gedanken, daß die Herausgabe aller verfügbarer Informationen sehr unangenehme Konsequenzen haben könnte, wenn sie in die falschen Hände geraten. Auf der anderen Seite sind diese Informationen für manche Verarbeitungen unbedingt notwendig.

Bei der Verwaltung von Studentendaten müssen zum Beispiel Namen, Matrikelnummern und Prüfungsergebnisse gesammelt werden. Das Prüfungsamt braucht alle diese Informationen, um im Endeffekt ein Diplomzeu-

[12] In früheren Versionen von Eiffel mußten diese Features explizit in einer Export-Liste genannt werden. Dies ist aber überflüssig geworden durch die verbesserte Syntax. **export** wird nur noch für die Aufschlüsselung der Sichtbarkeit vererbter features benötigt.

gnis ausstellen zu können. Auf der anderen Seite müssen zum Zwecke der schnellen Informationsverbreitung Klausurergebnisse öffentlich zur Einsicht ausgehängt werden. Es wäre Ihnen aber sicher nicht recht, wenn ein nicht so gutes Ergebnis jedem bekannt würde. Deshalb darf der Name nicht mit ausgedruckt werden, sondern nur Note und Matrikelnummer. Selektive Verbreitung von Informationen – eingeschränkt auf das unbedingt Notwendige – ist eine der wesentlichen Grundforderungen des Datenschutzes.

Deshalb können Features selektiv bereitgestellt werden, d.h. es ist möglich, zu jedem feature anzugeben, welche Klasse es benutzen darf. Diese werden in geschweiften Klammern hinter der feature-Klausel genannt.

feature {`Klasse`$_1$`,...,Klasse`$_n$} *featurelist*

bedeutet, daß die features der Aufzählung *featurelist* an die genannten Klassen `Klasse`$_1$`,...,Klasse`$_n$ (sowie an alle Klassen, die von diesen erben!) exportiert werden. Demzufolge bedeutet **feature** {} *featurelist* daß die features der Aufzählung *featurelist* überhaupt nicht weitergegeben werden. **feature** *featurelist* drückt aus, daß die features der Aufzählung *featurelist* an alle Klassen weitergegeben werden.

```
class PERSON
   creation
     init_mit_namen
   feature
     init_mit_namen(n,v:STRING) is ... do ... end;
     name: STRING;
     vornamen: STRING is ... do ... end;
     anzahl_vornamen: INTEGER is ... do ... end;
   feature{EINWOHNERMELDEAMT}
     geburtsjahr, todesjahr: INTEGER;
     alter(jahr:INTEGER): INTEGER is ... do ... end;
     vater, mutter: PERSON;
     nationalität: STRING
   feature{STANDESAMT}
     setze_todesjahr(jahr:INTEGER) is ... do ... end;
   feature{EINWOHNERMELDEAMT, PERSON}
     geschwister: ARRAY[PERSON] is ... do ... end;
   feature{PERSON}
     großeltern: ARRAY[PERSON] is ... do ... end;
   feature{}
     vornamen_liste: STRING;
end -- class PERSON
```

Abbildung 3.16: *Klassendefinition mit selektivem Export von Features*

In Abbildung 3.16 haben wir unser Beispiel um fiktive Klassen `EINWOHNERMELDEAMT` und `STANDESAMT` erweitert. Das Beispiel drückt aus, daß das Einwohnermeldeamt Geburts- und Todesjahr, Alter, Nationalität und Eltern abfragen kann, darüber hinaus – zusammen mit der Klasse `PERSON` – aber auch die Geschwister. Großeltern sind nur für Personen abfragbar. Nur das Standesamt darf das Todesjahr eintragen. Namen und Vornamen sind allgemein zugänglich.

Durch eine saubere Ausarbeitung dieser recht einfach aufgestellten Vereinbarung kann die Verbreitung von Informationen klar geregelt werden. Die Klasse `PERSON` entscheidet, wer von ihr welche Informationen bekommt. Wie diese dann von den anderen Klassen weiterverbreitet werden, fällt nicht mehr in die Verantwortung dieser Klasse, sondern unter die Zuständigkeit der Klasse, welche die bereitgestellte Information verarbeitet. Mehr zum Geheimnisprinzip findet man in [Meyer, 1988, Kapitel 2.2.5 und 9].

3.5 Copy- und Referenz-Semantik

In den vorigen Abschnitten ist bereits des öfteren darauf hingewiesen worden, daß Zuweisungen, Tests und ähnliche Operationen auf Objekten recht verschiedene Effekte haben können und daß deshalb auch mehrere Instruktionen scheinbar gleicher Natur existieren. Dies hat damit zu tun, daß Objekte eines einfachen Datentyps direkt angesprochen werden, Objekten eines Klassentyps aber nur über Verweise erreichbar sind. In diesem Abschnitt wollen wir daher den Unterschied zwischen der sogenannten *Copy-Semantik* (*Werteseman-tik*) und der *Referenzsemantik* (*Verweissemantik*) klarstellen.

3.5.1 Einfache Typen und Klassentypen

Eigentlich sollte es nur wenig Probleme bei der Beurteilung des Effektes von Zuweisungen und Vergleichen geben, da die Regeln eindeutig sind: in Eiffel enthalten alle Größen eines Klassentyps *Verweise* und nicht etwa die Objekte selbst. Standardoperationen beziehen sich daher auf Verweise – Eiffel hat eine *Referenzsemantik*.

Entwurfsprinzip 3.5.1 (Regeln der Semantik von Eiffel)

> *Eine Zuweisung der Form* `a:=b` *ist eine Wertzuweisung, wenn* `a` *und* `b` *von einem einfachen Typ (*`INTEGER`*,* `BOOLEAN`*,* `CHARACTER`*,* `REAL`*, und* `DOUBLE`*) sind. Sie ist eine Verweiszuweisung – und nicht etwa eine Objektzuweisung – wenn* `a` *und* `b` *von einem Klassentyp sind.*
>
> *Analog ist ein Test* `a=b` *bzw.* `a/=b` *ein Wertevergleich zwischen Größen einfacher Typen und ein Vergleich der Verweise zwischen Größen von Klassentypen.*

Eine Zuweisung von Objekten (d.h. eine Zuweisung der Komponenten) bzw. ein Vergleich zwischen (den Komponenten von) Objekten fällt daher *nicht* unter die Standardoperationen, die mit den Symbolen `:=`, `=` und `/=` bezeichnet werden, sondern ist eine Operation, die eine gesonderte Instruktion benötigt. Eiffel stellt hierfür die in Abschnitt 3.3.4 besprochenen Routinen `clone`, `copy` und `equal` bzw. die "tiefen" Varianten `deep_clone`, `deep_copy` und `deep_equal` zur Verfügung. Das folgende Beispiel soll die Unterschiede verdeutlichen.

Beispiel 3.5.2

> *In Abbildung 3.13 hatten wir die Klasse* `PERSON` *und die Initialisierungsprozedur* `init_mit_namen` *beschrieben. Es seien* `p_1, p_2, p_3, p_4` *Größen vom Typ* `PERSON`*. Nach Ausführung der Instruktionen*
>
> ```
> !!p_1.init_mit_namen(Meyer","Bertrand");
> p_2 := p_1;
> p_3 := clone(p_1);
> !!p_4.init_mit_namen("Kreitz",Christoph")
> ```
>
> *ist* `p_1=p_2` *wahr, nicht aber* `p_1=p_3`*. Dagegen gilt* `equal(p_1,p_2)` *und* `equal(p_1,p_3)`*.*
>
> *Die Instruktion* `p_2:=p_4` *würde dafür sorgen, daß* `p_2=p_4` *gilt,* `p_2.copy(p_4)` *dagegen nicht – auch hier gilt danach nur* `equal(p_2,p_4)`*.* `p_2.copy(p_4)` *ist übrigens nicht anwendbar, solange* `p_2` *noch keinen anderen Wert erhalten hatte.*

Die Folge der Referenzsemantik ist, daß nach einer Zuweisung `a:=b` zwischen Größen eines Klassentyps beide Größen auf dasselbe Objekt verweisen. *Änderungen des Objektes selbst betreffen somit sowohl* `a` *als auch* `b`*, Änderungen der Verweise dagegen nicht*. Auch diesen Unterschied wollen wir an einem Beispiel verdeutlichen.

Beispiel 3.5.3

> *Nehmen wir an, wir hätten in der Klasse* `PERSON` *eine weitere Routine* `setze_vater` *deklariert, welche bei Eingabe einer Größe vom Typ Person der Komponente* `vater` *eines Personenobjektes einen Verweis auf die so bezeichnete Person zuweist. Es seien* `p_1, p_2, p_3, p_4` *Größen vom Typ* `PERSON` *wie zuvor. Nach Ausführung der Instruktionen*
>
> ```
> !!p_1.init_mit_namen("Kreitz",Christoph");
> ```

```
!!p_2.init_mit_namen("Kreitz","Helmut");
p_3 := p_1;
```

gilt, wie bereits erklärt, `p_1=p_3`*. Nach Ausführung von*

```
p_4 := p_1;
p_4 := p_2;;
```

hat sich an dem durch `p_1` *bezeichneten Objekt nichts geändert und es gilt weiterhin* `p_1=p_3`*. Nach*

```
p_4 := p_1;
p_4.setze_vater(p_2);;
```

hat sich dagegen nicht nur die Vater-komponente von `p_4` *verändert, sondern auch die von* `p_1`*, da beide auf dasselbe Objekt verweisen.* `p_1=p_3` *gilt also nicht mehr.*

Der Grund hierfür ist, daß die Zuweisung `a:=b` eine dauerhafte Beziehung zwischen `a` und `b` bewirkt. Solange nicht einer der beiden *Verweise* verändert wird, zeigen beide auf dasselbe Objekt. Der Zweck dieser scheinbar komplizierten Semantik ist, daß interne Objekte dafür Objekte der realen Wirklichkeit beschreiben sollen. Änderungen dieser Objekte sollen daher auch bei allen Kunden sichtbar werden, die darauf zugreifen.

So ist z.B. die Sitzbelegungsliste für einen Flug von Frankfurt nach New York ein einzelnes Objekt, auf das alle möglichen Reisebüros auf festdefinierte Art Zugriff haben. Wird nun ein Sitz durch ein Darmstädter Reisebüro belegt, so muß er für alle anderen Reisebüros blockiert sein – ansonsten gäbe es ein heilloses Durcheinander im Flugverkehr. Aus diesem Grunde sind Verweise von dem zugreifenden Reisebüro auf die Sitzbelegungsliste wesentlich sinnvoller als etwa Kopien.

Die Tatsache, daß eine Operation auf Objekten einen weitreichenden Effekt auf andere Größen haben kann, die auf dasselbe Objekt verweisen, macht es notwendig, derartige Operationen mit großer Sorgfalt einzusetzen. Insbesondere muß man sich darüber klar sein, ob man wirklich ein Objekt verändern will und nicht etwa eine veränderte Kopie erzeugen möchte. Daher ist es notwendig, in einer Programmiersprache Operationen, die auf den *Objekten* arbeiten und nicht auf einem Verweis, als solche klar zu kennzeichnen. In Eiffel geschieht dies durch Verwendung der Punktnotation `entity.operation(argumente)`. Sie besagt, daß die Operation auf das Objekt zugreift und es ggf. verändert. Abbildung 3.17 faßt die wichtigsten Eigenschaften dieser Interpretation von Eiffel-Instruktionen zusammen.

Einfache Typen werden durch Copy-Semantik interpretiert: Zuweisung und Vergleiche beziehen sich auf Werte. Eine Zuweisung `a:=b` erzeugt in `a` eine Kopie des Wertes von `b`, die selbst keine Verbindung zu `b` mehr hat.

Klassentypen werden durch Referenz-Semantik interpretiert: Zuweisung und Vergleiche beziehen sich auf Verweise und können daher zwei Größen *binden*. Gleiche Verweise zeigen auf dasselbe Objekt. Eine Operation auf einer der beiden Größen kann über das gemeinsam genutzte Objekt die Eigenschaften der anderen verändern.

Abbildung 3.17: *Semantik von* Eiffel-*Typen*

Es sei nochmals darauf hingewiesen, daß es natürlich auch für Klassentypen Operationen gibt, die explizit eine Copy-Semantik in sich tragen. Das sind im einfachen Fall die Operationen `clone`, `copy` und `equal`, welche eine *flache* Kopie behandeln. Sie verfolgen eine Stufe eines Verweises und kopieren bzw. vergleichen alle Komponenten einzeln. Hierbei verwenden sie allerdings wieder die Referenz-Semantik. Eine hundertprozentige Copy-Semantik erhält man nur, indem man alle Verweise bis zum Ende verfolgt, also eine *tiefe* Kopie verfolgt, wie bei `deep_clone`, `deep_copy` und `deep_equal`. Diese Routinen sind jedoch sehr rechenintensiv.

3.5.2 expanded: Klassen mit Copy-Semantik

Die scharfe Trennung zwischen einfachen Typen und Klassentypen ist zuweilen lästig, da auch einfachste Datenstrukturen wie Paare und Strings, die man gar nicht so sehr als Repräsentationen realer Objekte betrachtet, ebenfalls die dynamische Natur von Objekten und eine Referenzsemantik erhalten. Aus diesem Grunde bietet Eiffel die Möglichkeit, einzelne Größen oder ganze Klassen explizit an eine Copy-Semantik zu binden.

```
expanded class INTPAIR
feature
     left, right: INTEGER
end -- class INTPAIR
```

Abbildung 3.18: *Klassendefinition mit* **expanded**

Abbildung 3.18 deklariert die Klasse `INTPAIR` als erweiterte (expanded) Grundklasse, deren Objekte direkt kopiert werden. Eine Deklaration `y:INTPAIR` hat dann den Effekt, daß die Größe `y` bei Zuweisungen nicht etwa einen Verweis auf ein Paar ganzer Zahlen erhält, sondern die beiden Zahlen selbst. Den gleichen Effekt würde man erzielen durch die Deklaration

```
y: expanded PERSON
```

auch, wenn `PERSON` gar nicht als expanded class deklariert wurde. *Größen, die als* **expanded** *deklariert wurden, verhalten sich exakt so, als ob sie von einem einfachen Typ wären.*

Natürlich möchte man auch Werte zwischen den Versionen expanded und nicht expanded einer Klasse vergleichen bzw. zuweisen (so wie man ganze Zahlen in reelle konvertieren möchte und umgekehrt). Hierfür gelten die folgenden Regeln.

`a:=b` Ist `a` nicht expanded und `b` expanded, dann ergibt sich derselbe Effekt wie bei `a:=clone(b)`: es wird ein neues Objekt erzeugt mit den gleichen Komponenten wie `b`. `a` erhält einen Verweis auf dieses Objekt.

Ist `a` expanded und `b` nicht expanded, dann ergibt sich ein Effekt wie bei `a.copy(b)`: das Objekt `a` erhält in jeder Komponente den entsprechenden Wert von `b`.

`a=b` Der Vergleich kann nur durchgeführt werden wie bei `equal(a,b)`, da es nur sinnvoll ist, die Inhalte der Objekte zu vergleichen.

expanded-Klassen verhalten sich also *genauso wie die einfachen Typen* von Eiffel. Deshalb werden die Grundtypen `INTEGER`, `BOOLEAN`, `CHARACTER`, `REAL`, und `DOUBLE` oft auch als Spezialfall der expanded-Klassen betrachtet (was eine einheitlichere Sicht auf Klassen gibt). Für expanded-Klassen darf maximal eine Initialisierungsprozedur (ohne Argumente!) deklariert werden, da Größen dieser Klassen beim Aufruf nicht mit einem Verweis, sondern als Objekt initialisiert werden.

Der Denkweise von Eiffel nach sollen erweiterte Klassen die Ausnahme bilden und erhalten aus diesem Grunde das gesonderte Schlüssselwort. Sie sind immer dann sinnvoll, wenn

- Attribute eines Objektes (wie z.B. die vier Räder eines Autos) niemals mit anderen Objekten geteilt werden dürfen,
- Basisdatentypen für Algorithmen realisiert werden sollen, die auf einer Kopiersemantik beruhen, oder
- Kommunikation mit anderen Sprachen, die auf erweiterten Klassen beruhen, stattfinden soll.

Der Normalfall in Eiffel aber sind die Klassen mit dynamischen Objekten.

3.6 Generische Klassen

Bei der Diskussion abstrakter Datentypen in Abschnitt 3.2.1 haben wir eine Möglichkeit angesprochen, die Flexibilität abstrakter Definitionen dadurch zu erweitern, daß *Typparameter* zugelassen werden. Der Bedarf hierfür wird besonders deutlich bei Klassen, die allgemeine, häufig verwandte Datenstrukturen wie Felder, Listen, Bäume und Matrizen beschreiben.

Nehmen wir zum Beispiel an, wir wollten eine Klasse definieren, die eine Liste ganzer Zahlen repräsentiert. Diese sähe dann etwa so aus

```
class INTLIST
creation  new
feature
  empty: BOOLEAN is -- Ist die Liste leer ?
         do...end;
  new is -- Erzeuge leere Liste
         do...end;
  cons(n:INTEGER) is  -- Hänge n vor die Liste
         do...end;
  head: INTEGER  is -- Erstes Element
         do...end;
  tail  is  -- Entferne erstes Element
         do...end
end -- class INTLIST
```

Wenn wir nun eine Liste reeller Zahlen benötigen, dann müssten wir das Ganze noch einmal schreiben

```
class REALLIST
creation  new
feature
  empty: BOOLEAN is -- Ist die Liste leer ?
         do...end;
  new is -- Erzeuge leere Liste
         do...end;
  cons(r:REAL) is  -- Hänge n vor die Liste
         do...end;
  head: REAL  is -- Erstes Element
         do...end;
  tail  is  -- Entferne erstes Element
         do...end
end -- class REALLIST
```

Dabei wird sich herausstellen, daß – bis auf den Namen `REAL` – die gesamte Implementierung von `REALLIST` mit der von `INTLIST` übereinstimmt. Wir haben also separate Implementierungen für Listen, obwohl diese jeweils nur dieselben Dienstleistungen anbieten. Dies macht wenig Sinn und birgt zudem die Gefahr in sich, daß spätere Erweiterungen nur in einer der beiden Klassen durchgeführt werden. Integerlisten könnten sich also plötzlich anders verhalten als Listen reeller Zahlen.

Aus diesem Grunde bietet Eiffel das Konzept der *generischen Klasse* an, um die Wiederverwendbarkeit von Modulen noch weiter zu verbessern. Eiffel ist die erste praxisrelevante Sprache, die dieses Konzept in voller Allgemeinheit aufgenommen hat.

3.6.1 Parametrisierung von Klassen

Eine generische Klasse wird mit dem Datentyp parametrisiert, welcher die Grundelemente der generische Klasse beschreibt. Abbildung 3.19 beschreibt eine generische Klasse `LIST[X]` für Listen von Objekten. Die Eiffel Syntax ähnelt dabei der in Abbildung 3.4 auf Seite 65 benutzten Schreibweise für abstrakte Datentypen.

```
class LIST[X]
creation  new
feature
  empty: BOOLEAN is -- Ist die Liste leer ?
        do...end;
  new is -- Erzeuge leere Liste
        do...end;
  cons(r:X) is  -- Hänge r vor die Liste
        do...end;
  head: X  is -- Erstes Element
        do...end;
  tail  is  -- Entferne erstes Element
        do...end
end -- class LIST[X]
```

Abbildung 3.19: *Generische Klassendefinition*

Der in eckigen Klammern angegebene Name – hier `X` – heißt *formaler generischer Parameter*. Es ist durchaus möglich, mehrere formale generische Parameter anzugeben. Diese werden dann durch Kommata von einander getrennt wie z.B. in `GENERISCH[X,Y]`.

Innerhalb der Klasse kann ein formaler generischer Parameter in allen Deklarationen wie ein normaler Datentyp eingesetzt werden, sei es als Typ von Attributen, Funktionen (wie bei `head`), Parametern in Routinen (wie bei `cons`) oder von lokalen Variablen. Nach außen hin ist der formale generische Parameter unbekannt.

Will man eine generische Klasse benutzen, um eine Größe zu deklarieren, so muß man *aktuelle generische Parameter* angeben, welche den Platz der formalen einnehmen, wie zum Beispiel bei

```
il: LIST[INTEGER]
```

Selbstverständlich muß die Anzahl der aktuellen und formalen generischen Parameter übereinstimmen. Als aktuelle generische Parameter ist praktisch alles zugelassen, was innerhalb der Kundenklasse bekannt ist, also einfache Typen, Klassentypen und ggf. sogar formale generische Parameter der Kundenklasse, falls diese ebenfalls eine generische Klasse ist.[13] Durchaus erlaubt sind also Deklarationen der Form

```
personenlisten: LIST[LIST[PERSON]]
```

Parametrisierte Klassen sind keine Typen, sondern Typschemata. Erst mit den Argumenten werden sie zu Typen. Ohne aktuelle Argumente kann man ihre Merkmale nicht ausführen und auch nicht austesten.

3.6.2 Typprüfung

Generizität hat nur in einer getypten Sprache eine Bedeutung, in der jede Größe einen bestimmten Typ haben muß. Nur dann ist es möglich zu prüfen, ob eine Operation überhaupt sinnvoll sein kann. Andernfalls gibt es keine Möglichkeit, die Typen der in eine Datenstruktur eingehenden Elemente einzuschränken, was normalerweise zu Laufzeitfehlern (im günstigen Falle) oder zu völlig unsinnigen Ergebnissen (z.B. Auslesen der internen Darstellung einer Zeichenkette als ganze Zahl) führt.

Beispiel 3.6.1

Nehmen wir einmal an, die Deklaration einer Klasse enthalte die Deklarationen

```
il: LIST[INTEGER]
pliste: LIST[PERSON]
p: PERSON
```

[13] Im Zusammenhang mit Vererbung werden wir eine weitere Form – Typen der Form **like** *Klassenausdruck* – kennenlernen, die als aktueller generischer Parameter möglich ist.

Dann sind die folgenden Anweisungen sinnvoll und gültig

```
pliste.cons(p)    -- Hänge p vor die Personenliste
il.cons(25)       -- Hänge 25 vor die Integerliste
p := pliste.head  -- Weise das erste Element der Personenliste einer Personengröße zu
```

Völlig unangebracht sind aber Anweisungen wie

```
pliste.cons(25)   -- Hänge die Zahl 25 vor die Personenliste
il.cons(p)        -- Hänge die Personengröße p vor die Integerliste
p := il.head      -- Weise das erste Element der Integerliste einer Personengröße zu
```

Eiffel ist eine *statisch getypte* Sprache, bei der alle Typprüfungen während der Übersetzung durchgeführt werden können – also nur vom Programmtext abhängen. Deshalb würden die unsinnigen Anweisungen im Beispiel bereits vom Compiler abgewiesen werden. Bei Sprachen ohne Datentypen, gibt es dagegen keine Möglichkeit, die Typen der in eine Datenstruktur eingehenden Elemente einzuschränken. In solchen Sprachen erfüllen generische Klassen keinerlei Zweck, da sowieso alle Operationen auf alles angewandt werden können. Die Last der Kontrolle liegt hier beim Programmierer.

Die Tatsache, daß innerhalb einer generischen Klasse praktisch nichts über eine Größe bekannt ist, deren Typ der formale generische Parameter ist, schränkt allerdings die Operationen auf dieser Größe sehr stark ein. Es dürfen nämlich nur Operationen benutzt werden, die auf jeden beliebigen Typ anwendbar sind. Insbesondere sind Operationen wie `!!` (Erzeugung), `clone`, `equal` oder Anwendungen von Merkmalen auf `x` *nicht* erlaubt, da nicht gewährleistet ist, daß der aktuelle generische Parameter ein Klassenyp ist. Abbildung 3.20 stellt die Regeln für die Verwendung formaler generischer Parameter zusammen.

Ist in einer Klassendeklaration `X` ein formaler generischer Parameter und `x` vom Typ `X`, dann darf `x` nur benutzt werden

- als linke Seite einer Zuweisung `x:=Ausdruck`, wobei der Ausdruck der rechten Seite ebenfalls vom Typ `X` sein muß,
- als rechte Seite einer Zuweisung `y:=x`, wobei die Größe auf der linken Seite ebenfalls vom Typ `X` sein muß,
- als aktuelle Argument im Aufruf einer Routine `f(..,x,..)`, welches einem formalen Argument vom Typ `X` entspricht (das geht nur, wenn `f` innerhalb der gleichen Klasse wie `x` deklariert wurde),
- in einem Booleschen Ausdruck der Form `x=y` oder `x/=y` (bzw. `y=x` oder `y/=x`), wobei `y` ebenfalls vom Typ `X` sein muß.

Abbildung 3.20: *Regeln für formale generische Parameter*

Wie bei gewöhnlichen Routinen übernehmen bei der Verwendung generischer Klassen die aktuellen Parameter die Rolle der formalen Parameter in der Deklaration und entsprechend die dort vereinbarten Eigenschaften. Ist also `f` ein von einer generischen Klasse `GENERISCH[X,Y]` exportiertes Attribut, dessen Typ der formaler generischer Parameter `X` ist oder eine Funktion mit entsprechendem Ergebnistyp, dann wird bei einer Kundendeklaration der Form

```
h:GENERISCH[INTEGER,PERSON]
```

der Ausdruck `h.f` (ggf. mit Parametern) innerhalb der Kundenklasse als vom Typ `INTEGER` aufgefaßt. Der Eiffel-Compiler kann daher prüfen, ob der Ausdruck korrekt benutzt wird.

Bei einer ersten Betrachtungen erscheinen die Typ-Einschränkungen an eine generische Klasse zu restriktiv. Man kann sich nämlich durchaus Anwendungsgebiete für Listen vorstellen, die aus verschiedenartigen Elementen bestehen wie z.B. Vektoren und Punkte eines zweidimensionalen Raumes. Da eine generische Klasse jedoch mit einem eindeutig festgelegten aktuellen Parameter instantiiert werden muß, sind – mit den bisherigen Mitteln – nur Listen möglich, die entweder nur aus Punkten oder nur aus Vektoren bestehen.

Dieses Problem wird durch das Konzept der Vererbung gelöst, welches wir im Abschnitt 3.8 diskutieren werden. Man deklariert einfach eine allgemeinere Klasse ZWEI_KOORD, welche die gemeinsamen Merkmale von Punkten und Vektoren charakterisiert und deklariert dann die Liste l als vom Typ LIST[ZWEI_KOORD]. Die Klassen PUNKT und VEKTOR werden als Abkömmling von ZWEI_KOORD deklariert und somit dürfen als Elemente von l sowohl Vektoren als auch Punkte eingesetzt werden.

3.6.3 Felder: Beispiele generischer Klassen

Viele Klassen der Eiffel-Basisbibliothek, die von jedem Compiler bereitgestellt wird, repräsentieren allgemeine Datenstrukturen und sind deshalb generische Klassen. Beipiele hierfür sind Kellerspeicher (Stacks), Warteschlangen (Queues), Listen, Bäume (Trees) und Felder (Arrays).

Die Klasse ARRAY der eindimensionalen Felder ist für praktische Anwendungen besonders wichtig. Im Gegensatz zu anderen Sprachen sind Felder in Eiffel *kein* vordefiniertes Sprachkonstrukt, sondern nur eine Klasse, für die ein besonders effizienter Code vorhanden ist. Daher verwenden Felder in Eiffel auch nicht die in anderen Sprachen gebräuchlichen speziellen Notationen zur Kennzeichnung von Indizes, sondern stellen für Zugriff und Zuweisung – wie immer – Routinen zur Verfügung. Dadurch kann man zuweilen bei der Realisierung komplexer indizierter Ausdrücke in Eiffel die Übersicht verlieren. Andererseits ist die Klasse ARRAY erheblich flexibler als gewöhnliche Felder. Abbildung 3.21 beschreibt die wichtigsten features der Klasse ARRAY (eine komplette Beschreibung liefert [Meyer, 1992, Kapitel 28.7]).

```
class ARRAY[X]
creation  make
feature
  lower, upper, count: INTEGER;
  make(min,max:INTEGER) is -- Erzeuge Feld mit Grenzen min und max
         do...end;
  item, infix "@" (i:INTEGER):X is -- Element mit Index i
         do...end;
  put(val:X,i:INTEGER) is -- Weise dem Element mit Index i den Wert val zu
         do...end;
  ⋮
end -- class ARRAY[X]
```

Abbildung 3.21: *Klassendefinition für Felder (mit* **infix** *Deklaration)*

Die Funktion make ermöglicht eine dynamische Felddimensionierung mit beliebigen Grenzen (allerdings ist das Feld leer, wenn max<min gilt). Die Funktion item liefert den Wert eines Feldelements und die Prozedur put ändert den Wert eines Feldelements. Das folgende Beispiel zeigt eine typische Nutzung dieser Klasse (und gibt zum Vergleich die Pascal-Notation).

Beispiel 3.6.2 *Es sei* p_array *deklariert vom Typ* ARRAY[PERSON] *und* person_1 *ein Objekt vom Typ* PERSON

```
!!p_array.make(12,24)         -- erzeuge ein Feld von Personen mit der Dimension 12..24
p_array.put(person_1,15)      -- weise dem Element mit Index 15 den Wert person_1 zu
                              -- ( Pascal-Notation p_array[15]:=person_1)
person_1 := p_array.item(17)  -- weise person_1 den Wert des Elements mit Index 17 zu
                              -- ( Pascal-Notation person_1 := p_array[17])
```

Da insbesondere die Schreibweise für Zugriffe auf Feldelemente sehr umständlich erscheint, bietet Eiffel mittlerweile die Möglichkeit an, Funktionsnamen als *Infix*-Namen zu schreiben.[14] Dies geschieht durch Angabe eines Infix-Bezeichners in Doppelhochkomma hinter dem Schlüsselwort **infix**, welches unmittelbar hinter dem eigentlichen Funktionsnamen – abgetrennt durch ein Komma – genannt werden muß.

[14]Daneben gibt es genauso die Möglichkeit, Funktionsnamen explizit als Prefix-Namen zu vereinbaren

So deklariert die Klasse `ARRAY` die Funktion `item` auch als Infix-Operator `@`. Diese Deklaration erlaubt es, das angegebene Symbol in der vertrauteren Infix Schreibweise zu verwenden. Es ist also möglich zu schreiben

```
person_1 := p_array@17
```

anstelle von `person_1 := p_array.item(17)`. Dies hebt die Beschränkungen früherer Versionen von Eiffel größtenteils auf. Eine Übernahme der Notation für Prozeduren dagegen ist nicht sinnvoll, weil diese gegen das Geheimnisprinzip und die ausschließliche Verwendung von Punktnotation bei der Veränderung von Objekten verstoßen würde. Ähnliche Klassen gibt es auch für Felder mit mehreren Dimensionen.

3.7 Verträge für Software-Zuverlässigkeit

In den bisherigen Abschnitten haben wir beschrieben, wie man Softwaremodule schreibt, die Klassen von Datenstrukturen implementieren. Wir haben dabei besonders viel Wert gelegt auf eine strukturelle Trennung der Aufgaben und Zuständigkeiten verschiedener Module. Durch Generizität haben wir die Wiederverwendbarkeit von Modulen erheblich ausgedehnt. Damit haben wir Werkzeuge zur Realisierung einiger wichtiger Qualitätsmerkmale in Softwareprodukten angesprochen: Wiederverwendbarkeit, Erweiterbarkeit und Kompatibilität.

Was uns jedoch noch fehlt sind sprachliche Hilfsmittel zur Sicherstellung von Korrektheit und Robustheit. Dies ist besonders wichtig in Anbetracht der angestrebten dezentralen Natur von Eiffel-Programmen, in der ein Kunde nicht wissen soll, *wie* der Lieferant seine Dienstleistungen realisiert, sondern sich nur darauf verlassen soll, *daß* gewisse Dienstleistungen bereitgestellt werden. Es ist offensichtlich, daß hierfür abgeklärt werden muß, welche *Garantien* ein Lieferant übernimmt und welche nicht.

Jeder, der schon einmal an einem größeren Softwareprojekt beteiligt war, in dem die Arbeit auf mehrere Phasen und ein Team von Mitarbeitern verteilt werden mußte, hat schon einmal erlebt, wie viele Diskussionen und Mißverständnisse sich um Probleme der Zuständigkeiten einzelner Module drehen: “Aber ich dachte, Du reichst mir nur normalisierte Werte durch...” oder “Warum prüfst Du das denn, da passe ich doch selber schon auf?” sind einige der häufigsten Fragen, die dabei entstehen. Dies zeigt, wie wichtig es ist, Absprachen über Anforderungen an Dienstleistungen zu treffen und *innerhalb der Sprache auch zu verankern*, damit sie nicht verloren gehen. Der Schlüsselbegriff hierbei ist das Konzept des *Programmierens durch Vertrag*. Die Beziehung zwischen Kunden und Lieferanten wird als eine formale Vereinbarung angesehen, in der die Rechte und Pflichten jeder Partei (Klasse) festgelegt sind. Nur durch eine Präzisierung der Bedürfnisse und Verantwortlichkeiten ist es möglich, das Vertrauen in die Zuverlässigkeit von Softwaresystemen zu steigern.[15]

Die Erwägungen, die zum Konzept der objektorientierten Programmierung geführt haben, hoben hervor, daß Klassen Implementierungen abstrakter Datentypen (vgl. Abschnitt 3.2.1) sein sollten. Dazu haben wir alle Serviceleistungen eines abstrakten Datentyps in der Terminologie der Programmierung beschrieben. Klassen und ihre Features können als Repräsentanten einer Spezifikation abstrakter Datentypen verstanden werden.

Abstrakte Datentypen aber sind mehr als nur eine Ansammlung von Typen und Operationen. Was uns noch fehlt, ist ein Gegenstück zu den Axiomen und Vorbedingungen, den Schlüsselkonzepten für die Charakterisierung der *semantischen Eigenschaften* eines abstrakten Datentyps.

Hierfür stellt Eiffel das Konzept der *Zusicherungen* bereit, die als *Vor-* und als *Nachbedingungen* von Attributen, Routinen und Klassen formuliert werden können. Vorbedingungen der Spezifikation werden als Vorbedingungen einzelner Routinen genannt werden, Axiome entweder als Nachbedingung einer einzelnen Routine oder eines einzelnen Attributs oder – sofern globale Eigenschaften der Klasse, z.B. das Zusammenspiel mehrerer Features betroffen sind – als *Klasseninvariante*.

[15] Natürlich ist es wünschenswert, mit Hilfe von rechnergestützten formal-logischen Methoden ein Softwareprodukt aus einer derartigen Präzisierung von Bedürfnissen und Verantwortlichkeiten zu entwickeln. Da die Forschung auf diesem Gebiet jedoch noch nicht weit genug ist, um praktisch verwendbare Werkzeuge mit absoluter Korrektheitsgarantie bereitzustellen, müssen wir uns vorerst darauf beschränken, Programme sehr sorgfältig unter Berücksichtigung derartiger Verträge zu entwickeln und ihre Einhaltung nachträglich von Hand zu verifizieren.

Eiffel	logischer Operator
and	Konjunktion $\wedge$
or	Disjunktion $\vee$
and then	Sequentielle Konjunktion $\overline{\wedge}$
or else	Sequentielle Disjunktion $\overline{\vee}$
not	Negation $\neg$
xor	Exklusive Disjunktion $\dot{\vee}$
implies	Implikation $\Rightarrow$

Abbildung 3.22: *boolesche Ausdrücke in* Eiffel

3.7.1 Zusicherungen

Eine wichtige Möglichkeit, die Gefahr von Abweichungen zwischen Software-Spezifikationen und ihren Implementierungen zu vermindern ist die Einführung von Spezifikationselementen *in die Implementierung*. D.h. man ordnet einem Element ausführbaren Codes – Klassen, Routinen, oder Anweisungen – einen Ausdruck über den Zweck dieses Elementes zu. Ein solcher Ausdruck, der angibt, was das Element eigentlich tun sollte, wird *Zusicherung* (*assertion*) genannt.

Eine Zusicherung ist eine Eigenschaft einiger Werte von Programm-Größen. Sie drückt zum Beispiel aus, daß bei der Erzeugung von Feldern der maximale Index nicht kleiner sein darf als der minimale. Mathematisch betrachtet ist eine Zusicherung etwas ähnliches wie ein Prädikat. Allerdings besitzt die in Eiffel verwendete Sprache für Zusicherungen *nur einen Teil der Mächtigkeit* der in Abschnitt 2.2.2 vorgestellten Prädikatenlogik[16] sondern entspricht in etwa nur der im Abschnitt 2.2.6 besprochenen dreiwertigen Logik. Es ist also nicht möglich, jede Aussage über die Eigenschaften von Größen präzise als Zusicherung auszudrücken. Dies ist jedoch nicht besonders problematisch, da die Zusicherungssprache in erster Linie dazu dienen soll, Verträge zwischen den *Entwicklern* von Softwaremoduln zu fixieren. Man kann daher zur Not auch Teilinformationen in der Form von Kommentaren zur Zusicherung verstecken.

Syntaktisch sind Zusicherungen boolesche Ausdrücke der Programmiersprache Eiffel (siehe Abbildung 3.22) mit einigen Erweiterungen, die nur in Zusicherungen, nicht aber im Programmtext benutzt werden können.

- Anstelle der gewöhnlichen Konjunktion **and** wird zur besseren Trennung einzelner, nicht unmittelbar zusammenhängender, Bestandteile einer Zusicherung ein Semikolon verwendet wie z.B. in

  ```
  n>0 ; not x=Void
  ```

- Bestandteile einer Zusicherung(, die durch Semikolon getrennt sind,) können mit Namen gekennzeichnet werden, die durch einen Doppelpunkt abgetrennt werden, wie z.B. in

  ```
  Positiv: n>0 ; Nichtleer: not x=Void
  ```

  Die Ähnlichkeit zur Syntax der Deklaration von Größen ist durchaus gewollt. Namen werden vom Laufzeitsystem registriert, um gegebenenfalls Meldungen zu erzeugen und eine programmierte Verarbeitung von Fehlern zu ermöglichen.

- Eine zusätzliche Erweiterung **old** werden wir im folgenden Abschnitt vorstellen.

Auf Wunsch werden Zusicherungen in der Eiffel-Umgebung zur Laufzeit überwacht, was sie zu einem mächtigen Werkzeug beim Aufspüren von Fehlern und für eine kontrollierte Behandlung von Ausnahmen macht. Diese Verwendungszwecke wollen wir jedoch erst in späteren Kapiteln aufgreifen. Unser jetziges Interesse liegt in der Anwendung von Zusicherungen als Werkzeug zur Konstruktion korrekter Systeme und zur Dokumentation, *warum* sie korrekt sind.

[16]Der Grund hierfür ist, daß zur Überprüfung von Zusicherungen Verfahren benötigt werden, die logische Beweise ausführen und hierzu den Kalkül der Zusicherungssprache und einen Suchmechanismus verwenden. Die Prädikatenlogik ist für diese Zwecke zu mächtig: es gibt kein Verfahren, das für beliebige prädikatenlogische Formeln beweisen kann, ob sie wahr oder falsch sind.

3.7.2 Vor- und Nachbedingungen

Die bedeutendste Anwendung von Zusicherungen ist die Spezifikation von Routinen, deren Aufgabe ja die Implementierung von Funktionen eines abstrakten Datentyps ist. Um diese Aufgabe präzise auszudrücken – sowohl als Hilfe für den Entwurf als auch als Teil einer Dokumentation – gibt es in Eiffel zwei Möglichkeiten, Zusicherungen einer Routine zuzuordnen: als *Vor-* und die *Nachbedingungen* (*preconditions* bzw. *requirements* und *postconditions*) der Routine. Diese zielen darauf ab, die semantischen Eigenschaften der Routine explizit zu machen.

Die Vorbedingung drückt diejenigen Eigenschaften aus, die *beim Aufruf* der Routine immer gelten müssen, damit sie ordnungsgemäß funktionieren kann. Sie gilt für alle Aufrufe, sowohl von innerhalb der deklarierenden Klasse als auch von außerhalb bei einem Aufruf durch eine Kundenklasse. In einem korrekten System sollte es der Fall sein, daß Routinen niemals in Zuständen aufgerufen werden, in denen die Vorbedingung nicht erfüllt ist. Eine Vorbedingung *kann* innerhalb einer Routinendeklarationen in der Form von Klauseln angegeben werden, die mit dem Schlüsselwort **require** eingeleitet werden. Diese Klausel muß *vor* der eigentlichen Anweisungsfolge und ggf. auch vor einem **local** erscheinen.

In der Nachbedingung wird festgelegt, welche Eigenschaften *nach Beendigung* der Routine gewährleistet sein sollen. Sie drückt eine Garantie aus, welche der Implementierer für den Fall übernimmt, daß beim Aufruf der Routine die Vorbedingungen erfüllt waren. Eine Nachbedingung *kann* innerhalb einer Routinendeklarationen in der Form von Klauseln angegeben werden, die mit dem Schlüsselwort **ensure** eingeleitet werden. Diese Klausel muß am Ende der Anweisungsfolge direkt vor dem **end** erscheinen.

Im Abschnitt 3.6 hatten wir die generische Klasse `ARRAY` mit den Features `make, lower, upper, count, item` und `put` skizziert. Als Vorbereitung für eine systematische Implementierung dieser Klasse wollen wir diese Skizze nun um die notwendigen Vor- und Nachbedingungen ergänzen, welche die semantischen Eigenschaften eindimensionaler Keller beschreiben. Dies sind zum Beispiel

- `make(min,max)` erzeugt ein Feld der Größe `max - min + 1` bzw. der Größe 0, wenn `max<min` gilt.
- `item` und `put` sind nur anwendbar, wenn der angegebene Index zwischen `lower` und `upper` liegt.
- `put` positioniert einen Wert so, daß er durch `item` mit dem gleichen Index wiedergefunden werden kann.

Abbildung 3.23 zeigt, wie diese Eigenschaften als Vor- und Nachbedingungen der Klasse `ARRAY` ausgedrückt werden können.

Zur Formulierung der Vor- und Nachbedingungen dürfen neben den normalen booleschen und arithmetischen Standardoperationen auch die *formalen Argumente der Routine* sowie *alle Attribute und Funktionen* der deklarierenden Klasse, nicht jedoch lokale Variablen, benutzt werden. In den Nachbedingungen darf zusätzlich – falls es sich um eine Funktion handelt – auch noch die Größe `Result` verwendet werden.

Eine Schreibweise, die für die Formulierung von Nachbedingungen einer Routine notwendig ist, haben wir noch nicht besprochen. Um Veränderungen im aktuellen Exemplar überprüfen zu können, ist es notwendig, den Zustand des Objektes bei Eintritt in die Routine mit dem Zustand beim Verlassen der Routine vergleichen zu können. Zur Kennzeichnung des Ursprungszustands eines Objektes wird das Schlüsselwort **old** verwendet. Allgemein gilt

> Ist `a` ein Attribut[17] der deklarierenden Klasse, dann bezeichnet **old** `a` den Wert der entsprechenden Objektkomponente beim Eintritt in die Routine.
>
> Jedes nicht von **old** angeführte Vorkommen von `a` in der Nachbedingung bezeichnet den Wert der entsprechenden Objektkomponente bei Beendigung der Routine.

[17] Im Prinzip sind nach dem Schlüsselwort **old** alle Ausdrücke erlaubt, die auch ohne dieses Schlüsselwort möglich sind. Es ist jedoch keine Schwierigkeit, diese so umzuformulieren, daß unmittelbar hinter **old** nur Attribute genannt werden, was auch zu einem verständlicheren Zusicherungsstil führt.

```
class ARRAY[X]
creation  make
feature
  lower, upper, count: INTEGER;
  make(min,max:INTEGER) is -- Erzeuge Feld mit Grenzen min und max
         do...
         ensure
             max<min implies count = 0;
             max >= min implies lower = min and upper = max;
             max >= min implies count = max - min + 1
         end; -- make
  item, infix "@" (i:INTEGER):X is -- Element mit Index i
         require
             lower <=i; i <= upper
         do...end; -- item
  put(val:X,i:INTEGER) is -- Weise dem Element mit Index i den Wert val zu
         require
             lower <=i; i <= upper
         do...
         ensure
             item(i)=val
         end; -- put
  ⋮
end -- class ARRAY[X]
```

Abbildung 3.23: *Klassendefinition mit Vor- und Nachbedingungen*

Die bisher angegebenen Routinen für die Klasse `ARRAY` benötigen die Möglichkeit des Vergleichs mit dem vorhergehenden Zustand nicht. Dies ändert sich jedoch, wenn man eine weitere Routine `resize` der Klasse `ARRAY` betrachtet, mit der man die Indexgrenzen verändern kann. Hier muß sichergestellt werden, daß vorhergehende Einträge nicht verlorengehen, d.h. daß die alten Indexgrenzen innerhalb der neuen liegen. Abbildung 3.24 zeigt, wie dies als Nachbedingung der Routine `resize` ausgedrückt werden kann.

```
resize(min,max:INTEGER) is -- Erweitere Feld auf Grenzen min und max
       do...
       ensure
           upper >= old upper;  lower <= old lower
       end; -- resize
```

Abbildung 3.24: *Nachbedingung mit* **old**

Vor- und Nachbedingungen spielen eine wichtige Rolle beim systematischen Entwurf korrekter Software, denn sie können aufgefaßt werden als ein *Vertrag* zwischen denjenigen, die eine Routine implementieren, und denen, die sie benutzen – also zwischen Lieferanten und Kunden einer Klasse. Die Klauseln von **require** und **ensure** beschreiben die jeweiligen Rechte und Pflichten.

- Die Vorbedingung bindet den Kunden, da sie die Voraussetzungen beschreibt, unter denen ein Aufruf der Routine erlaubt ist. Der Kunde verpflichtet sich, nur in den erlaubten Fällen die Routine zu benutzen und der Lieferant hat das Recht, bei Mißachtung der Voraussetzungen gar nichts zu liefern – er ist dann nicht einmal verpflichtet, daß die Dienstleistung überhaupt eine Antwort gibt.

  Im Falle der Funktion `put` bedeutet dies also, daß der Kunde sich verpflichtet, `put` nur mit Indizes aufzurufen, die zwischen `lower` und `upper` liegen, und daß andernfalls keine Leistung erbracht wird.

- Die Nachbedingung bindet die Klasse und ihre Implementierer, da sie festlegt, was nach der Ausführung

gewährleistet wird. Der Kunde kann sich darauf verlassen, daß die versprochene Leistung – bei Einhaltung der Vorbedingung – auch erfüllt wird.

Im Falle der Funktion `put` bedeutet dies also, daß der Lieferant zusagt, das gegebene Element so einzusortieren, daß es mit `item` wiedergefunden werden kann.

Die Vorteile dieses Vertrages sind gegenseitiger Natur: der Kunde bekommt bei jedem Aufruf gewisse Ergebnisse, der Implementierer weiß, daß er zu Beginn von gewissen Voraussetzungen ausgehen darf, die er nicht mehr prüfen muß. Hierdurch wird der Programmierstil erheblich vereinfacht, da die Verantwortlichkeiten auf Kunden und Lieferanten verteilt werden.

Die Frage, wie genau man Verträge schließen kann, ist hauptsächlich eine Frage des Vertrauens zwischen Kunden und Lieferanten. Je mehr Vertrauen besteht, um so genauer kann man Verträge vereinbaren und um so einfacher wird die Implementierung. Das bedeutet aber auch, daß man sich darauf verlassen können muß, daß sie eingehalten werden. Andernfalls wird jegliche Kooperation unmöglich.

Für Sie als zukünftige Softwareentwickler bedeutet dies, daß Sie sich bei der Erfüllung *Ihrer* Aufgaben keinerlei Ungenauigkeiten erlauben dürfen, sondern *garantieren* können müssen, daß Ihr Modul so arbeitet, wie Sie es versprochen haben. Aus diesem Grunde müssen Sie in der Lage sein, die Korrektheit zu *beweisen*, also zu verifizieren – natürlich unter der Annahme, daß die Leistungen, die Sie benutzen, ihrerseits korrekt sind. Auf dieses Thema, die *Verifikation* von Routinen, werden wir im nächsten Kapitel besonders eingehen.

Ist man sich über die möglichen Benutzer einer Klasse im unklaren, weil die Anwendungsgebiete sehr vielfältig sind – wie zum Beispiel bei generischen Klassen, dann lohnt es sich, diese durch ein zusätzliches *Filtermodul* zu umgeben, welches beim Aufruf einer Routine zunächst alle möglichen Fehler abfängt und die ungeschützte Version nur dann aufruft, wenn die Vorbedingungen erfüllt sind. Ein solches Modul bietet eine saubere Trennung zwischen der algorithmischen Lösung von Techniken zur Behandlung möglicher Fehleingaben. Ein Beispiel für ein Filtermodul gibt [Meyer, 1988, Abschnitt 7.3.4].

3.7.3 Klasseninvarianten

Vor und Nachbedingungen beschreiben die semantischen Eigenschaften einzelner Routinen. Darüber hinaus besteht jedoch auch die Notwendigkeit, globale Eigenschaften von Objekten zu beschreiben, die von allen Routinen eingehalten werden müßen, wie zum Beipiel die Tatsache, daß die Größe eines Feldes `count` nie negativ wird und die Abstand von `lower` und `upper beschreibt`. Dies war eine der Nachbedingungen der Erzeugungsprozedur `make` in Abbildung 3.23, wurde aber in keiner anderen Routine erwähnt, obwohl es selbstverständlich sein sollte, daß diese Bedingung immer erhalten bleibt.

Man könnte diese Bedingung nun in der Nachbedingung jeder einzelnen Routine zusätzlich erwähnen, aber dies entspricht nicht dem Gedanken, daß sie `immer` gelten soll, also eine *Invariante* (*unveränderliche Eigenschaft*) der Klasse selbst ist. Daher bietet Eiffel die Möglichkeit an, Klasseninvarianten als solche zu formulieren. Dies geschieht durch Klauseln, die mit dem Schlüsselwort **invariant** eingeleitet werden, und am Ende der Klassendeklaration genannt werden. Abbildung 3.25 zeigt die Deklaration von `ARRAY` mit Klasseninvarianten.

Klasseninvarianten sind also Zusicherungen, die allgemeine semantische Bedingungen formulieren, welche – im Gegensatz zu den Vor- und Nachbedingungen *einzelner* Routinen – für jedes Klassenelement als ganzes gelten. Sie können neben den Beziehungen zwischen Attributen auch semantische Beziehungen zwischen Funktionen oder zwischen Funktionen und Attributen ausdrücken. Nur Prozeduren sind in Zusicherungen nicht erlaubt, da sie bei einer Überprüfung der Zusicherung Objekte verändern würden.

Klasseninvarianten müssen natürlich nur in stabilen Zuständen, also vor und nach dem Aufruf einer von außen verwendbaren Routine, nicht aber *während* ihrer Abarbeitung gelten. “Interne” Routinen, also solche, die nicht von außen aufgerufen werden können, sind daher von der Überprüfung ausgenommen, da sie nach außen ohne direkten Effekt sind.

```
class ARRAY[X]
creation  make
feature
  lower, upper, count: INTEGER;
  make(min,max:INTEGER) is -- Erzeuge Feld mit Grenzen min und max
         do...
         ensure
             max<min implies count = 0;
             max >= min implies lower = min and upper = max
         end; -- make
  item, infix "@" (i:INTEGER):X is -- Element mit Index i
         require
             lower <=i;  i <= upper
         do...end; -- item
  put(val:X,i:INTEGER) is -- Weise dem Element mit Index i den Wert val zu
         require
             lower <=i;  i <= upper
         do...
         ensure
             item(i)=value
         end; -- put
  resize(min,max:INTEGER) is -- Erweitere Feld auf Grenzen min und max
         do...
         ensure
             upper >= old upper;  lower <= old lower
         end; -- resize
  ⋮
invariant
  nonnegative_size: count >= 0;
  consistent_size:  count = upper - lower + 1
end -- class ARRAY[X]
```

Abbildung 3.25: *Klassendefinition mit Klasseninvarianten*

Definition 3.7.1 (Invariantenregel)

Eine Zusicherung I ist genau dann eine korrekte Klasseninvariante einer Klasse C, wenn eine der beiden folgenden Bedinungen erfüllt ist

- *I gilt nach jedem Aufruf einer Erzeugungsprozedur von C, die auf Argumente angewandt wurde, welche die Vorbedingungen erfüllen.*
- *I gilt nach jedem Aufruf einer exportierten Routine, die auf Objekte angewandt wurde, welche I erfüllen und auf Argumente, welche die Vorbedingungen der Routine erfüllen.*

Dies bedeutet, daß die Klasseninvariante implizit zu den Vor- und Nachbedingungen jeder exportierten Routine hinzugefügt wird, was natürlich Konsequenzen den Vertrag zwischen Benutzer und Implementierer einer Klasse hat. Die Invariante bindet beide, sowohl den Kunden als auch den Lieferanten.

Nicht alle Zusicherungen in den Invarianten einer Klasse haben ein unmittelbares Gegenstück in der Spezifikation des zugehörigen abstrakten Datentyps, denn es gibt Eigenschaften von Attributen, die für die konkrete Implementierung, nicht aber für die abstrakte Spezifikation eine Rolle spielen. So ist zum Beipiel die Bedingung, daß in der Klasse `ARRAY` der Abstand zwischen oberer und unterer Feldgrenze nicht negativ werden darf, keine Eigenschaft, die essentiell für die Natur von Feldern ist. Allerdings ist sie sehr wichtig, um eine sinnvolle *Darstellung* von Feldern zu beschreiben, denn aus den Zusicherungen an die anderen Routinen ergibt sich, daß Objekte, welche die Invariante

```
count = upper - lower + 1; count >= 0
```

```
class LIST[X]
creation  new
feature
  empty: BOOLEAN  is -- Ist die Liste leer ?
         do...end;
  new is -- Erzeuge leere Liste
         do...
         ensure
             empty
         end; -- new
  cons(r:X) is  -- Hänge r vor die Liste
         do...
         ensure
             not empty; head = r
              -- tail würde jetzt old liefern
         end; -- cons
  head: X is -- Erstes Element
         require
             not empty
         do...end; -- head
  tail  is  -- Entferne erstes Element
         do...
         ensure
             old empty implies empty
         end -- tail
end -- class LIST[X]
```

Abbildung 3.26: *Klassendefinition mit unvollständigen Zusicherungen*

verletzen, keine gültige Darstellung von Feldern sein können. Aus diesem Grunde nennt man derartige Zusicherungen auch *Darstellungsinvarianten*. Darstellungsinvarianten beziehen sich also nicht auf einen abstrakten Datentyp, sondern auf die Objekte, welche bei einer einmal gewählten Implementierungsstruktur die Elemente dieses Datentyps darstellen können.

3.7.4 Grenzen der Anwendbarkeit von Zusicherungen

Zusicherungen in Eiffel decken nicht alles ab, was man mit Mitteln der Logik als Spezifikation von Klassen formulieren könnte. Nicht alle Axiome eines abstrakten Datentyps lassen sich auch als Zusicherung ausdrücken. So kann man zum Beispiel bei der Spezifikation endlicher Listen (Abbildung 3.4 auf Seite 65) das Axiom

```
tail(cons(x,L)) = L
```

nicht direkt in eine Nachbedingung der Routinen `tail` oder `cons` umsetzen. Dies liegt daran, daß `tail` oder `cons` Prozeduren sind, welche ein Objekt verändern, sobald sie aufgerufen werden. Damit sind sie für Zusicherungen unbrauchbar.

Prinzipiell wäre es natürlich möglich, die Zusicherungssprache von Eiffel dahingehend zu erweitern, daß sie genauso mächtig ist wie die Mathematik der abstrakten Datentypen. Dafür müßte man sie allerdings um Mengen, Folgen, Funktionen, Relationen, Prädikate und Quantoren erweitern. Solche Sprachen gibt es bereits[18], aber sie sind zur Zeit noch schwer zu implementieren und ineffizient zu testen, und vor allem auch für Nicht-Logiker schwer zu erlernen – und damit würde sie niemand benutzen.

Eiffel bietet daher einen pragmatischen Kompromiß an. Zusicherungen werden eingeschränkt auf boolesche Ausdrücke mit wenigen Erweiterungen. Dies erlaubt, die wichtigsten Eigenschaften von Klassen genau zu

[18] Als Beispiel einer solchen Sprache sei die (intuitionistische) Typentheorie genannt.

formulieren und zu überwachen. Eigenschaften die sich so nicht formulieren lassen, sollte man aber als *Kommentar* in die Zusicherungen mit aufnehmen. Dies läßt sich zwar nicht überprüfen, wird aber einem Benutzer, der mit dem Befehl **short** die Kurzdokumentation der Klasse sehen möchte, mit angezeigt. Abbildung 3.26 zeigt eine Eiffel-Spezifikation der Klasse `LIST[X]`, die auf dieses Hilfsmittel zuückgreift.

Zusicherungen dienen also zur Kontrolle semantischer Eigenschaften, als Dokumentation und Hilfe zum Verständnis, und als Strukturierungsmittel für die Verifikation. Diesen Aspekt werden wir im nächsten Kapitel hervorheben, wenn es um die systematische Inplementierung vertraglich vereinbarter Leistungen geht.

Beim objektorientierten Programmieren spielen Zusicherungen eine weitere Rolle, die über das bisher gesagte hinausgeht. Mit der Einführung in die Vererbung im nächsten Abschnitt werden wir sehen, daß Zusicherungen wesentlich dafür sind, die semantische Unversehrtheit von Klassen und Routinen im Zusammenhang mit den Mechanismen für *Polymorphismus* und *Redefinition* zu bewahren.

3.8 Vererbung

Einer der wesentlichen Aspekte der objektorientierten Denkweise ist es, erweiterbare und wiederverwendbare Softwaremodule zu schreiben. Klassen, so wie wir sie bisher beschrieben haben, besitzten bereits viele der Qualitäten, die man von wiederverwendbaren Softwarebausteinen erwartet. Sie sind in sich geschlossene Moduln. Sie wahren das Geheminisprinzip, indem sie Schnittstellen nach außen (d.h. bereitgestellte Dienstleistungen) klar von ihrer Implementierung trennen. Sie können durch das Zusicherungskonzept exakt spezifiziert werden.

Für die angestrebten Ziele Erweiterbarkeit und Wiederverwendbarkeit ist jedoch noch mehr erforderlich. Will man zum Beispiel die Klasse der Entleiher einer Bibliothek beschreiben, so benötigt man fast dieselben Dienstleistungen, wie die Klasse `PERSON` bereits zur Verfügung stellt. Es wäre also sinnlos, die Klasse der Entleiher völlig neu zu entwickeln, denn man würde praktisch den selben Code nochmal schreiben und dabei Gefahr laufen, Inkonsistenzen oder gar Implementierungsfehler zu erzeugen. Wir benötigen daher Techniken, die offensichtlichen Gemeinsamkeiten in Klassen ähnlicher Struktur zu erfassen und gleichzeitig Unterschiede in einzelnen Details zuzulassen.

Ein weiteres Problem hatten wir am Ende des Abschnitts 3.6.2 bereits angesprochen. Das Typsystem von Eiffel garantiert die Konsistenz eines Programms zum Zeitpunkt der Compilation, verbietet aber zum Beispiel die Bildung von Listen, die aus Elementen verschiedener Klassen bestehen, auch in Fällen, in denen dieses sinnvoll sein könnte, wie zum Beispiel bei Listen geometrischer Objekte wie Punkte, Vektoren, Segmente etc.

Das Konzept der *Vererbung* bietet die Möglichkeit, aus den konzeptionellen Beziehungen zwischen Klassen einen Nutzen zu ziehen. Eine Klasse kann eine Spezialisierung (*Erbe / Descendant*), Erweiterung (*Vorfahr / Ancestor*) oder eine Kombination (*Mehrfachvererbung / multiple inheritance*) anderer Klassen sein. Die Methode, derartige Beziehungen festzuhalten und als Programmierkonzept zu nutzen, wollen wir im folgenden näher betrachten.

3.8.1 Erben und Vorfahren

In Abbildung 3.5 haben wir eine einfache Deklaration der Klasse `PERSON` beschrieben, die wir in dieser Form zur Charakterisierung der Autoren eines Buchs gebrauchen könnten. Für unser Bibliothekenverwaltungssystem brauchen wir jedoch noch mehr Informationen über Personen, zum Beispiel dann, wenn sie als Entleiher einer Bibliothek auftreten. Viele relevante Merkmale von Entleihern sind identisch mit denen einer Person. Man benötigt Name, Vornamen und für manche Zwecke auch das Geburtsjahr. Darüber hinaus werden aber auch die Adresse und die Nummer des Bibliotheksausweises benötigt.

Wir können aus diesen Gemeinsamkeiten Nutzen ziehen, indem wir die Klasse `ENTLEIHER` als *Erben* (*heir, descendant*) der Klasse `PERSON` definieren, die umgekehrt zum *Elternteil* (*parent*) von `ENTLEIHER` wird. Das

bedeutet, daß alle features von PERSON auch in ENTLEIHER gelten, aber auch neue features dazukommen. In Eiffel wird das dadurch realisiert, daß am Anfang der Deklaration von ENTLEIHER eine Erbklausel erscheint, die mit dem Schlüsselwort **inherit** beginnt und danach die Elternklassen nennt.

```
class ENTLEIHER
inherit PERSON
feature
     straße, hausnummer, stadt: STRING;
     postleitzahl : INTEGER;
     ausweisnummer: INTEGER
end -- class ENTLEIHER
```

Abbildung 3.27: *Klassendefinition mit Vererbung*

Die features der Elternklasse brauchen in der feature-Klausel *nicht* erneut genannt werden, da sie automatisch verfügbar sind, *ohne daß qualifiziert werden muß*[19]. Lediglich die für die Klasse ENTLEIHER spezifischen Merkmale straße, hausnummer, stadt, postleitzahl und ausweisnummer müssen explizit angegeben werden.

Vererbung ist ein transitiver Vorgang. Jede Klasse, die ENTLEIHER beerbt, wird automatisch auch die features von PERSON übernehmen. Aus diesem Grunde spricht man nicht nur von *Eltern* sondern ganz allgemein von *Vorfahren* und *Nachkommen* einer Klasse.

Definition 3.8.1 (Nachkommen und Vorfahren)

Eine Klasse, die direkt durch eine inherit*-Klausel oder indirekt von einer Klasse A erbt, heißt* Nachkomme *von A. Eine Klasse A gilt als ihr eigener Nachkomme. Alle Nachkommen von A, die nicht mit A identisch sind, heißen* echte Nachkommen *von A.*

Ist B (echter) Nachkomme der Klasse A, so heißt A (echter) Vorfahre *von B.*

Die sprachliche Regelung schließt generische Klassen mit ein. So ist zum Beispiel die generische Klasse TWO_WAY_LIST[X] der Eiffel Bibliothek als Nachkomme der Klasse LINKED_LIST[X] deklariert. Diese Eigenschaft gilt für alle Datentypen, die man für den formalen generischen Parameter einsetzt. Die Klasse TWO_WAY_LIST[INTEGER] ist also Nachkomme von LINKED_LIST[INTEGER]. TWO_WAY_LIST[ENTLEIHER] ist Nachkomme von TWO_WAY_LIST[PERSON], weil ENTLEIHER Nachkomme von PERSON ist, und damit auch von LINKED_LIST[PERSON].

Bei einer graphischen Veranschaulichung der Klassenstruktur eines Systems werden wir Vererbung durch einen aufwärts zeigenden Pfeil darstellen. Dieser Pfeil soll die Relation "erbt von" charakterisieren

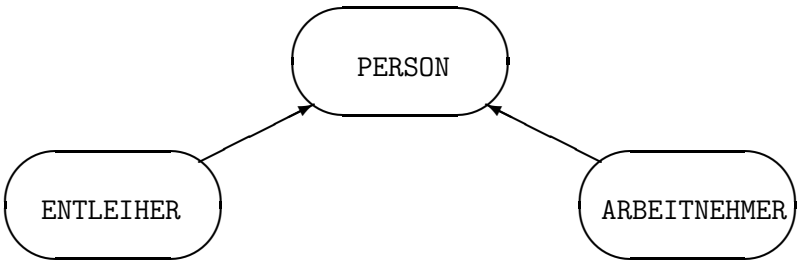


Abbildung 3.28: *Vererbung als Diagramm:* ENTLEIHER *und* ARBEITNEHMER *erben von* PERSON

[19]Dies unterscheidet Erben von Kunden einer Klasse. Kundenklassen müssen features der Lieferanten als *Dienstleistungen* betrachten, die durch einen qualifizierten Aufruf "eingekauft" werden müssen. Erbenklassen bekommen alle features umsonst. Sie dürfen sie (unqualifiziert) benutzen, als ob es ihre eigenen wären.

3.8.2 Export geerbter Features

Normalerweise werden alle features einer Elternklasse unverändert übernommen. Es gibt jedoch Gründe, Ausnahmen von dieser Regel zu vereinbaren. Manche Merkmale der Elternklasse `PERSON` machen im Zusammenhang mit der Verwaltung von Entleihern nur wenig Sinn: weder das Todesdatum noch die Nationalität sind von irgendeinem Interesse. Daher sollten diese features von der Klasse `ENTLEIHER` nicht wieder als Dienstleistungen bereitgestellt werden. Da sie nicht innerhalb der Klasse `ENTLEIHER` definiert, sondern geerbt wurden, kann man ihre Verbreitung auch nicht durch die Form der feature-Klausel kontrollieren, sondern benötigt eine explizite *Export-Vereinbarung*. In Eiffel wird dies durch Angabe einer export-Klausel innerhalb der Beschreibung der Vererbungseigenschaften ermöglicht.

```
class ENTLEIHER
inherit PERSON
      export
          name, vornamen, geburtsdatum
      end -- feature-Anpassung von PERSON
feature
      straße, hausnummer, stadt: STRING;
      postleitzahl : INTEGER;
      ausweisnummer: INTEGER
end -- class ENTLEIHER
```

Abbildung 3.29: *Klassendefinition mit Export geerbter features*

Die Angabe des Schlüsselwortes **export** besagt, daß die dahinter aufgezählten von `PERSON` geerbten features zusammen mit den neuen features als Dienstleistungen der Klasse `ENTLEIHER` bereitstehen. Die nicht genannten features `todesjahr` und `nationalität` sind nicht mehr benutzbar. Fehlt die export-Klausel, so sind *alle* von `PERSON` geerbten features erneut verfügbar. Das Schlüsselwort **end** beendet Modifikationen der geerbten features.

Auch beim Export geerbter features kann es sinnvoll sein, die Weitergabe von Informationen von der Kundenklasse abhängig zu machen. So könnte es zum Beispiel sein, daß die Nationalität eines Arbeitnehmers – ebenfalls ein Spezialfall von Personen – für das Finanzamt aus steuerrechtlichen Gründen von Bedeutung ist, nicht aber für den Arbeitgeber. Aus diesem Grunde erlaubt Eiffel einen selektiven Export geerbter features. Die Syntax ist vergleichbar mit der in Abschnitt 3.4 angegebenen Form für feature-Klauseln:

export
{ *Klassenliste*$_1$} *featurelist*$_1$;
⋮
{ *Klassenliste*$_n$} *featurelist*$_n$

Die Klasse `ARBEITNEHMER` in Abbildung 3.30 exportiert an die Klasse `ARBEITGEBER` – *und an alle Erben dieser Klasse*– die geerbten features `name, vornamen, geburtsjahr` sowie alle neu vereinbarten features[20], an die Klasse `FINANZAMT` jedoch alle geerbten (und die neuen) features. Zur Vereinfachung der Schreibweise gibt es hierfür in Eiffel (Version 3) das Schlüsselwort **all**. Es besagt, daß alle features der Elternklasse an die in der Klassenliste aufgezählten Klassen geliefert werden.

Fehlt die export-Klausel, so bleibt ein geerbtes geheimes feature geheim und ein exportiertes features wird weiter exportiert. Dieses Verhalten kann jedoch in der export-Klauseln nach Belieben verändert werden, da die Erben einer Klasse – im Gegensatz zu den Kunden – alle Rechte (und Pflichten) ihrer Ahnen besitzen und damit umgehen dürfen, wie sie es für richtig halten. Sie dürfen daher für *ihre* Kunden ein geheimes feature wieder exportieren bzw. ein exportiertes feature verstecken[21].

[20] Wir haben hier der Einfachheit halber die vier Komponenten `straße, hausnummer, stadt, postleitzahl` einer Adresse in einem Objekt zusammengefaßt.

[21] Hier gibt es allerdings feine Grenzen, da sonst die Korrektheit des Gesamtsystems verletzt werden könnte.

```
class ARBEITNEHMER
inherit PERSON
      export
          {ARBEITGEBER} name, vornamen, geburtsjahr;
          {FINANZAMT} all
      end -- feature-Anpassung von PERSON
feature
      adresse: ADRESSEN;
      gehaltsklasse: STRING;
      gehalt: REAL is -- Gehalt berechnen
           do Result := Gehalt aus Tabelle nach gehaltsklasse  end
end -- class ARBEITNEHMER
```

Abbildung 3.30: *Klassendefinition mit selektivem Export geerbter features*

Diese Koppelung der Vererbung mit dem in Abschnitt 3.4 diskutierten Geheimnisprinzip hat eine Reihe interessanter Anwendungen. Sie erlaubt es, durch verschiedene Erbenklassen auch verschiedene Sichten auf eine Datenstruktur zu ermöglichen und hierüber die unterschiedlichen Zugriffsrechte zu regeln. So gibt es zum Beispiel bei einem Bankkonto die Vorgänge `eröffnen, abheben, einzahlen, geheimzahl_eingeben, geheimzahl_prüfen, geheimzahl_ändern, löschen, gebühren_einziehen, konto_sperren`, usw. Nicht jeder dieser Vorgränge darf vom Kunden selbst ausgeführt werden (für die Eröffnung des Kontos sind z.B. einige Vorschriften zu beachten) sondern nur von den Bankangestellten am Schalter und umgekehrt. Andere Vorgänge (Kontosperrung) dürfen nur von Angestellten des Managements ausgelöst werden. Diese verschiedenen Sichten auf ein und dieselbe Datenstruktur läßt sich durch Vererbung auf eine sehr einfache und übersichtliche Art realisieren. Die in Abbildung 3.31 genannten Erben der Klasse `BANKONTO` stellen jeweils einen Teil der Dienstleistungen der Ahnenklasse bereit und darüber hinaus noch ihre eigenen speziellen features.

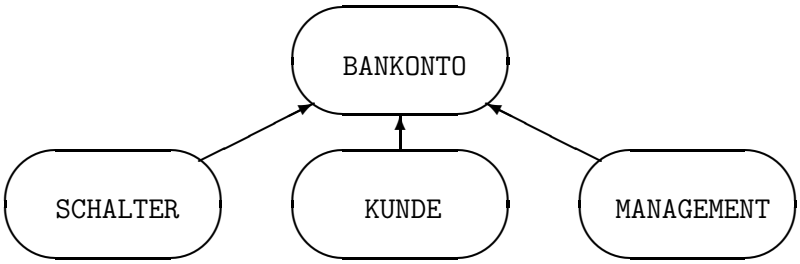


Abbildung 3.31: *Vererbung und Datenkapselung: Verschiedene Sichten auf eine Datenstruktur*

Eine gewisse Ausnahme von der Regel, daß features einer Elternklasse für gewöhnlich übernommen werden, betrifft Initialisierungsprozeduren. Meist besitzen die Erben einer Klasse mehr Attribute als ihre Vorfahren und auch die Eigenschaften der geerbten features sind oft spezieller als die der Elternklasse[22]. Aus diesem Grunde müssen spezifische Initialisierungsprozeduren einer Nachkommenklasse oft völlig anders aussehen als die der Eltern. Man kann also nicht einfach die Initialisierungsprozedur der Elternklasse erneut zur Erzeugung von Objekten der Nachkommenklasse verwenden. Ist dies dennoch gewünscht, so muß diese Prozedur erneut über eine creation-Klausel als Initialisierungsprozedur der Nachkommenklasse deklariert werden. Ansonsten ist sie nicht mehr als eine gewöhnliche geerbte Routine, die bestehende Objekte ändert, nicht aber neue erzeugt.

Vererbung ermöglicht es, Objekte einer Nachkommenklasse auch als Objekte der Vorfahrenklasse zu betrachten. Arbeitnehmer und Entleiher sind insbesondere auch Personen – man besitzt nur einige zusätzliche Informationen über Arbeitnehmer und Entleiher, die man von Personen nicht unbedingt erhalten kann.

[22]So muß zum Beispiel ein Arbeitnehmer in Deutschland immer älter als 14 Jahre sein, während für Personen eine solche Einschränkung nicht sinnvoll ist.

Ist also `p` ein Personenobjekt und `a` ein Arbeitnehmerobjekt, so ist der Aufruf `a.name, a.vornamen` genauso erlaubt wie `p.name, p.vornamen`. Darüber hinaus kann man allerdings auch `a.adresse` und `a.gehalt` aufrufen, *nicht* jedoch `p.adresse` oder `p.gehalt`

Dies gilt natürlich nur dann, wenn die geerbtes features nicht durch export-Klauseln von der Weiterverwendung ausgeschlossen werden.

Diese Eigenschaft läßt sich als eine erste wichtige Regel des Typsystems von Eiffel beschreiben.

Entwurfsprinzip 3.8.2 (Regel der Featureanwendung)

Ist eine Größe `entity` *vom Klassentyp K, so ist die Anwendung* `entity.f` *eines feature* `f` *nur dann zulässig, wenn* `f` *in einem Vorfahren von K definiert wurde.*

Man beachte, daß diese Regel "nur dann" und nicht etwa "genau dann" besagt. Die Voraussetzung, daß ein feature in einem Vorfahren von K (d.h. eventuell auch in K selbst) definiert wurde, ist notwendig für eine Anwendung, reicht aber nicht immer aus – zum Beispiel, weil das feature durch die Klasse K nicht oder nur selektiv exportiert wurde.

Die Interpretation des selektiven Exports von features wird allerdings durch die Vererbung etwas aufgeweicht. Da Erben alle Rechte ihrer Vorfahren besitzten, haben sie natürlich auch alle Zugriffsrechte. Das bedeutet, daß ein selektiv an eine Klasse exportiertes feature auch innerhalb aller Erbenklassen zur Verfügung steht – selbst, wenn diese nicht explizit in der export- bzw. feature-Klausel genannt sind.

Entwurfsprinzip 3.8.3 (Regel des selektiven Exports)

Ein an eine Klasse K selektiv exportiertes feature `f` *darf in allen Nachkommen von K benutzt werden.*

In den folgenden Abschnitten werden wir weitere Regeln des Typsystems von Eiffel vorstellen. All diese Regeln sind statischer Natur, d.h. sie können ausschließlich auf der Basis des Programmtextes überprüft werden[23]. Der Compiler wird Klassen abweisen, die eine dieser Regeln verletzen. Auf diese Art wird sichergestellt, daß es während der Laufzeit eines Systems nicht mehr zu Typfehlern kommen kann.

3.8.3 Redefinition

So wie es Gründe dafür gibt, geerbte features selektiv an verschiedene Klassen zu exportieren, gibt es auch Gründe, ihre Implementierung zu *verändern*. So wird zum Beispiel bei Universitätsangestellten das Gehalt nicht alleine aufgrund der Gehaltsklasse bestimmt, sondern hängt zusätzlich vom tatsächlichen Alter der Angestellten ab ab. Das bedeutet, daß die Funktion `gehalt` bei Universitätsangestellten nach einem anderen Verfahren berechnet werden sollte als im Allgemeinfall. Um dies zu ermöglichen, erlaubt Eiffel, geerbte features neu zu definieren. Hierzu wird in einer redefine Klausel angegeben, welche features verändert werden. Natürlich muß jetzt jedes dieser features erneut als feature der Klasse deklariert und implementiert werden.

Abbildung 3.32 enthält eine Deklaration der Klasse aller Universitätsangestellten, in der das feature `gehalt` neudefiniert wurde. Ohne die redefine-Klausel wäre die erneute Deklaration von `gehalt` als Merkmal von `UNI-ANGESTELLTE` ein Fehler, da `gehalt` bereits aus der Klasse `ARBEITNEHMER` übernommen wird.

Die redefine-Klausel teilt mit, welche features einer Elternklasse in einer Erbenklasse neudefiniert werden[24]. Hierdurch wird gewährleistet, daß *derselbe* Name sich auf verschiedene aktuelle features beziehen kann. Abhängig vom Typ des Objektes, auf das es angewandt wird, erhält man eine andere *Implementierung*.

[23] Für fast alle Regeln gibt es im Referenzbuch [Meyer, 1992] verfeinerte Formen. Diese behandeln jedoch auch alle Spezialfälle, die sich aus Kombinationen diverser Möglichkeiten ergeben und sind zur Erklärung des eigentlichen Prinzip ungeeignet.

[24] Der Compiler benötigt die redefine-Klausel eigentlich nicht, weil jede erneute Deklaration eines ererbten features ja nur eine Redefinition sein kann, sofern kein Fehler des Programmierers vorliegt. Da aber irrtümliche Redefinitionen bei umfangreichen Ahnenklassen nicht unbedingt ausgeschlossen werden können, muß Redefinition ausdrücklich als solche gekennzeichnet werden. Dies erhöht die Sicherheit und Lesbarkeit der Programme.

Es sei an dieser Stelle auch erwähnt, daß ein feature durch das Schlüsselwort **frozen** gegen Redefinition sperren kann, wenn man darauf Wert legt, allen Benutzen eine eine unveränderliche Implementierung bereitzustellen.

```
class UNI-ANGESTELLTE
inherit ARBEITNEHMER
      redefine
          gehalt
      export
          {ARBEITGEBER} name, vorname, geburtsjahr, adresse, gehaltsklasse;
          {FINANZAMT} all
      end -- feature-Anpassung von ARBEITNEHMER
feature
      universität: STRING;
      fachbereich, raum, telephon: INTEGER;
      gehalt: REAL is -- Gehalt berechnen
           do Result :=  Gehalt nach Alter und gehaltsklasse end
end -- class UNI-ANGESTELLTE
```

Abbildung 3.32: *Vererbung mit Redefinition*

Natürlich kann man diese nicht beliebig abändern. Von außen betrachtet muß ein feature im wesentlichen unverändert, d.h. auf die gleiche Art aufrufbar und veränderbar bleiben. Kurzum, die Vereinbarungen der Elternklasse müssen weiterhin Gültigkeit haben – ansonsten sollte man besser ein neues feature deklarieren.

Zu den Vereinbarungen, die von der Erbenklasse übernommen werden müssen, gehören insbesondere die Anzahl der Argumente und die Typen der features. Routinen müssen Routinen bleiben und Attribute Attribute. Die einzigen erlaubten Ausnahmen sind ein Wechsel von parameterlosen Funktionen zu Attributen – weil hier der Kunde den Unterschied nicht sieht – und ein Übergang zu Typen, die zum ursprünglichen Datentyp passen ("konform sind").

Entwurfsprinzip 3.8.4 (Regel der Redefinition)

> *Ein in einer Klasse deklariertes Attribut, Funktionsergebnis oder formales Routinenargument darf in einer Erbenklasse mit einem neuen Typ redeklariert werden, wenn der neue Typ konform zum ursprünglichen ist. Das Attribut oder die zugehörige Routine gilt als redefiniert.*
>
> *Der Rumpf einer Routine kann redefiniert werden, solange die obige Typeinschränkung nicht verletzt wird. Parameterlose Funktionen dürfen dabei als Attribute redefiniert werden.*
>
> *Redefinierte features, die nicht ursprünglich* deferred *(siehe Abschnitt 3.8.6) waren, müssen in einer entsprechenden* redefine *Klausel aufgeführt werden.*

Die Idee hinter dieser Regelung ist, daß eine Klasse immer eine speziellere Version eines in einer Ahnenklasse deklarierten Elementes anbieten kann. Deshalb ist auch eine *Redefinition eines Attributes als parameterlose Funktion nicht erlaubt*, weil hierdurch die Zuweisung von Werten an das Attribut unmöglich gemacht wird[25].

Die Redefinitionsregel verwendet den Begriff der *Konformität* zwischen zwei Datentypen. "B ist konform zu A" bedeutet in erster Näherung, daß B ein Nachkomme von A sein muß. Dies gilt in ähnlicher Form auch für den Fall, daß A und B generische Parameter enthalten. Alle in B vorkommenden aktuellen generischen Parameter müssen Nachkommen der entsprechenden Parameter in A sein. So kann zum Beispiel B die Klasse `TWO_WAY_LIST[ENTLEIHER]` sein und A die Klasse `LINKED_LIST[PERSON]`, denn die generische Klasse `TWO_WAY_LIST` ist Nachkomme von `LINKED_LIST` und der aktuelle Parameter `ENTLEIHER` ein Nachkomme von `PERSON`.

Die präzise Definition ist etwas komplizierter, da B auch in der Vererbungsklausel eine Klasse mit generischen Parametern nennen kann, die A als Vorfahren besitzt, das Konzept der Deklaration durch Assoziation hinzukommt, das wir erst in Abschnitt 3.8.5 besprechen werden, und Konformität transitiv ist.

[25] Bei einer Redefinition eines Attributes als parameterlose Funktion dürfte auch die Ahnenklasse keine Zuweisung an das Attribut mehr enthalten, da Größen dieser Klasse ja auch Objekte einer Nachkommenklasse bezeichnen können – Personengrößen dürfen zum Beispiel auch auf spezielle Personen, nämlich Entleiher zeigen. Damit würde die Redefinition aber einen Eingriff in die Elternklasse mit sich bringen, was völlig gegen den Sinn der Strukturierung eines Systems in unabhängige Klassen ist.

Definition 3.8.5 (Konformität)

Ein Datentyp B ist genau dann konform zu einem Datentyp A ("B conforms to A"), wenn eine der folgenden Bedingungen erfüllt ist.

1. *A und B sind identisch,*
2. *A ist von der Form* **expanded** *B oder B ist von der Form* **expanded** *A,*
3. *A ist* `REAL` *oder* `DOUBLE` *und B ist* `INTEGER` *oder* `REAL`*,*
4. *A ist eine Klasse ohne generische Parameter und B ist eine Klasse, die A in der* inherit*-Klausel aufführt,*
5. *A ist von der Form $G[A_1, ..A_n]$ und B führt $G[B_1, ..B_n]$ in der* inherit*-Klausel auf, wobei G eine generische Klasse ist und jedes der B_i zu A_i konform ist,*
6. *B ist von der Form* **like** `anchor` *und der Typ von* `anchor` *ist konform zu A,*
7. *Es gibt einen Typen C mit der Eigenschaft, daß B konform ist zu C und C konform zu A.*

Eine ausführliche Diskussion von Konformität und weitere Verfeinerungen der Definition findet man in [Meyer, 1992, Kapitel 13].

Vererbung und Redefinition begünstigen einen Softwareentwicklungsstil, der sich vollkommen von den bisher üblichen Vorgehensweisen unterscheidet. Statt jedes Problem ganz von Neuem zu lösen, kann man auf Lösungen ähnlicher Probleme aufbauen, ihre Dienstleistungen erweitern und gegebenenfalls durch Redefinition für die spezielle Fragestellung optimieren. Diese Form der Wiederverwendung bestehender Softwaremodule ist hochgradig ökonomisch und bescheiden zugleich: sie erkennt die Leistungen anderer Softwareentwickler an, anstatt sie durch eine völlige Neuentwicklung zu ignorieren.

Die Notwendigkeit, hierfür die Gemeinsamkeiten zwischen Gruppen verwandter Daten zu berücksichtigen, findet ebenfalls eine Antwort: in Netzwerken von Klassen, die durch Vererbung miteinander verbunden sind, können wir die logischen Beziehungen zwischen diesen Gruppen ausdrücken. Wiederverwendbarkeit wird dadurch gefördert, daß die Deklaration eines features soweit wie möglich nach oben verschoben wird, damit dieses feature in einer großen Anzahl von Nachkommen zur Verfügung steht.

3.8.4 Polymorphismus

Wir haben gesehen, daß durch Vererbung die Wiederverwendbarkeit von Softwaremodulen erheblich gesteigert werden kann. Darüber hinaus hat Vererbung aber noch einen weiteren wichtigen Aspekt, nämlich *Polymorphismus*. Dieser Begriff bezeichnet im allgemeinen die Fähigkeit, verschiedene Formen anzunehmen. Im Rahmen einer typisierten Sprache wie Eiffel bedeutet dies, daß Größen eines Programmtextes zur Laufzeit auf Exemplare verschiedener Klassen verweisen dürfen.

So ist zum Beispiel vom intuitiven Verständnis her jeder Arbeitnehmer auch eine Person – und nicht etwa etwas, was erst durch eine Konvertierung zu einer Person werden kann. Aufgrund des Vererbungskonzepts findet dieser intuitive Zusammenhang in Eiffel ein natürliches Gegenstück: eine Größe `p` vom Typ `PERSON` darf zur Laufzeit durchaus auf ein Arbeitnehmer- oder ein Universitätsangestellten-Objekt bezeichnen. Bei Programmiersprachen mit einem starren Typkonzept wäre hierfür eine explizite Konvertierung notwendig[26].

Polymorphismus ist in einer Sprache wie Eiffel leicht zu realisieren, da ja alle Größen eines Klassentyps in Wirklichkeit Verweise auf Objekte sind. Polymorphismus in Eiffel hat daher nichts zu tun mit Objekten, die ihre *Form* während der Laufzeit *ändern*, sondern bedeutet nur, daß eine gegebene Größe auf Objekte verschiedener Art verweisen kann. Durch Vererbung wird Polymorphismus also prinzipiell ermöglicht, aber gleichzeitig auch auf ein sinnvolles Maß begrenzt: eine Größe `a` vom Typ `ARBEITNEHMER` darf nicht ein einfaches Personenobjekt

[26]Eine einzige Ausnahme bilden Zahlen der Typen `INTEGER` und `REAL`, die in fast allen Sprachen automatisch ineinander umgewandelt werden, ohne daß es hierzu eines gesonderten Konvertierungsbefehls bedarf.

bezeichnen, denn nicht jede Person ist auch ein Arbeitnehmer. Vererbung ist also ein Kontrollmechanismus für Polymorphismus, der einerseits eine vollständige Klassifizierung aller Objekte ermöglicht und dennoch eine gewisse – der Realität entsprechende Freiheit – zuläßt.

Vererbung liefert auch eine Lösung für die am Ende des Abschnitts 3.6.2 angesprochene Problematik, daß Listen, die aus Elementen verschiedener Klassen bestehen, bisher nicht möglich waren. Ohne Vererbung wäre es ausgesprochen schwierig, in einem Bibliothekenverwaltungsystem die unterschiedlichen Arten von Benutzern – Entleiher, Universitätsangestellte, Professoren und Mitarbeiter der Bibliothek – in einer gemeinsamen Liste zu verwalten. Diese Form von Polymorphismus kann nun sehr leicht realisiert werden: man versucht einfach, eine gemeinsame *Oberklasse* `BENUTZER` zu definieren, deren Erben die Klassen `ENTLEIHER`, `UNI-ANGESTELLTE`, `PROFESSOR` und `MITARBEITER` sind, und deklariert die gewünschte Liste als

```
benutzerliste:LIST[BENUTZER].
```

Die in `benutzerliste` eingetragenen Verweise dürfen dann, wie in Abbildung 3.33 gezeigt, auf Exemplare aller Erbenklassen von `BENUTZER` zeigen.

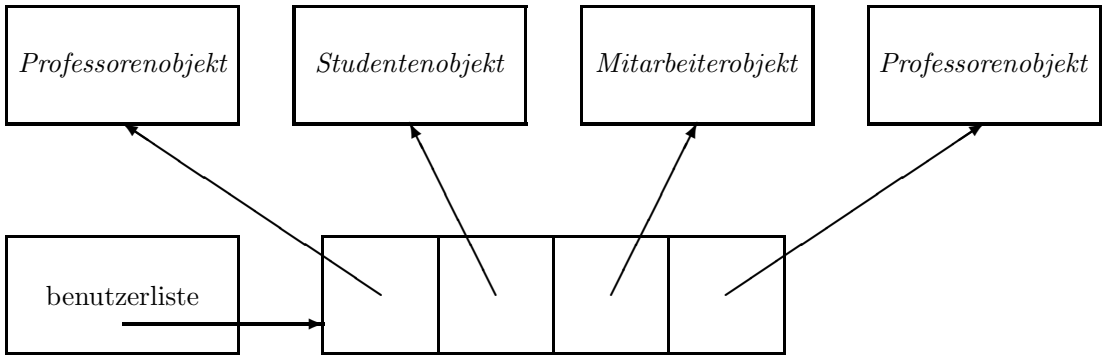


Abbildung 3.33: *Eine polymorphe Liste*

Ein unbeschränkter Polymorphismus ist in Eiffel jedoch nicht möglich, da dies dem Gedanken der Klassifizierung von Objekten in Gruppen mit gemeinsamen Merkmalen widersprechen würde. Eine Zuweisung

```
p:=a
```

ist deshalb durchaus möglich, wenn `p` vom Typ `PERSON` und `a` vom Typ `ARBEITNEHMER` ist, nicht jedoch

```
a:=p.
```

Im ersten Fall ist nämlich immer sichergestellt, daß `p` auf ein Objekt der Klasse `PERSON` bzw. ihrer Nachkommen zeigt, während im zweiten Fall `a` nach der Zuweisung durchaus auf ein Objekt zeigen könnte, welches die in der Klasse `ARBEITNEHMER` vereinbarten Merkmale überhaupt nicht kennt[27]. Hinter dieser Einschränkung steht eine weitere grundsätzliche Regel des Typsystems von Eiffel.

Entwurfsprinzip 3.8.6 (Regel der Typverträglichkeit)

Eine Zuweisung der Form `x:=y`*, wobei* `x` *vom Typ A und* `y` *vom Typ B ist, ist nur dann zulässig, wenn B konform zu A ist.*

Gleiches gilt für den Aufruf einer Routine mit formalem Parameter `x` *und aktuellem Parameter* `y`*, wobei* `x` *vom Typ A und* `y` *vom Typ B ist. Der Aufruf ist nur dann zulässig, wenn B konform zu A ist.*

[27]Die Zuweisung `a:=p` ist selbst dann verboten, wenn unmittelbar zuvor der Befehl `p:=a` ausgeführt wurde und `p` somit tatsächlich auf ein Objekt der Klasse `ARBEITNEHMER` verweist. Will man einen Verweis an ein Objekt binden, so muß sichergestellt sein, daß dessen Typ mit dem des Verweises verträglich ist.

Jede andere Regelung würde nur einen schlechten Programmierstil fördern. Wenn von vorneherein *beabsichtigt* ist, daß `p` nur ein Arbeitnehmerobjekt enthalten soll, dann sollte man `p` entsprechend deklarieren.

Um die Beschreibung des Typsystems von Eiffel etwas zu erleichtern, ist es hilfreich, den Typ eines Verweises aufzuschlüsseln in einen *statischen* und einen *dynamischen* Typ.

Definition 3.8.7 (Typ eines Verweises)

> *Der statische Typ eines Verweises ist der Typ, welcher bei der Deklaration der zugehörigen Größe verwandt wurde. Der dynamische Typ eines Verweises ist der Typ des Objektes, mit welchem der Verweis zur Laufzeit des Systems verbunden ist.*

Die Regel der Typverträglichkeit besagt also, daß der dynamische Typ eines Verweises stets ein Nachkomme des statischen sein muß. Man beachte, daß diese Unterscheidung für Größen und Objekte keinen Sinn macht, da der Typ einer Größe (deklariert im Programmtext) und eines Objektes (erzeugt als Exemplar einer bestimmten Klasse) sich niemals ändern kann. Nur Verweise können polymorph sein, da sie gleichzeitig Laufzeitdarstellungen von Größen sind und Zeiger auf Objekte.

Im vorhergehenden Abschnitt haben wir über die Möglichkeit gesprochen, Routinen in den Nachkommen einer Klasse neu zu definieren. Das Gehalt in der Klasse `ARBEITNEHMER` wird anders berechnet als in der Klasse `UNI-ANGESTELLTE`. Andererseits ist es durch den Polymorphismus möglich, ein Objekt der Klasse `UNI-ANGESTELLTE` an eine Größe `e` vom Typ `ARBEITNEHMER` zu binden. Was geschieht nun bei einem Aufruf

> `e.gehalt`?

Die Regelung in Eiffel ist einfach (und leicht zu implementieren!): der aktuelle Typ des Objektes, auf den die Größe `e` verweist, entscheidet über die Version von `gehalt`, welche zur Laufzeit ausgeführt wird. Diese Regel nennt man *dynamisches Binden*.

Entwurfsprinzip 3.8.8 (Dynamisches Binden)

> *Ist* `x` *vom Klassentyp A und* `r` *eine in A deklarierte Routine, dann wird ein Routinenaufruf der Form* `x.r` *an die Implementierung gebunden, welche innerhalb der Klasse definiert wurde, zu der das mit* `x` *zur Laufzeit verbundene Objekt gehört.*

Polymorphismus und dynamisches Binden ermöglichen eine *dezentrale Softwarearchitektur*. Jede Variante einer Operation wird in demjenigen Modul definiert, in dem auch der entsprechende Datentyp als Klasse deklariert wurde[28]. Änderungen einer bestimmten Implementierung betreffen also nur die Klasse, in der diese definiert war und die Hinzunahme weiterer Klassen mit neuen Varianten dieser Operation lassen alle anderen Klassen meist völlig unberührt – Klassen sind für ihre eigenen Implementierungen zuständig und mischen sich nicht etwa in die Angelegenheiten der anderen ein.

Durch dynamisches Binden können wir auch die Forderung nach einer *Darstellungsunabhängigkeit* von Softwaremodulen erfüllen: wir können Operationen anfordern, von der es mehr als eine Variante gibt, ohne wissen zu müssen, welche Variante nun im konkreten Fall benutzt wird. Dies ist besonders interessant bei einer Verwaltung großer Datenmengen wie zum Beispiel die Menge aller Bücher einer Bibliothek. Diese könnte intern

[28] In Programmiersprachen wie Pascal ist das nicht möglich. Hier muß stattdessen eine übergeordnete Routine geschrieben werden, die alle Varianten berücksichtigt. So muß zum Beispiel die Routine `gehalt` berücksichtigen, daß sie auf Objekt vom Typ `ARBEITNEHMER` und auf `UNI-ANGESTELLTE` angewandt werden kann. Kommt nun eine weitere Klasse mit wieder einer anderen Art der Gehaltsberechnung dazu – wie z.B. die Klasse der Professoren, bei denen das Dienstalter zählt und nicht das tatsächliche Alter –, dann muß die Routine `gehalt` umgeschrieben werden.

Der große Nachteil dieser scheinbar unbedeutenden Komplikation ist, daß Routinen wir `gehalt` jederzeit Kenntnis über das *gesamte* Softwaresystem haben müssen. Jede muß wissen, welche Arten von Arbeitnehmern im System erlaubt sind. Kleine Änderungen des Systems, wie z.B. die Hinzunahme einer weiteren Unterklasse, haben daher große Auswirkungen, da *alle* Routinen daraufhin überprüft werden müssen, ob sie von dieser Änderung betroffen sind. Dadurch werden aber Änderungen einer einmal vorgenommenen Implementierung ausgesprochen schwierig. Bewährte und geprüfte Module müssen jedesmal wieder geöffnet werden und die Gefahr, daß sich Fehler bei einer solchen Änderung einschleichen, ist immens. Dies erklärt vielleicht die mangelnde Flexibilität herkömmlicher Software.

Man sagt auch, daß ein derartiges System den sogenannten *Stetigkeitsgrundsatz* verletzt, daß kleine Modifikationen in der Konzeption auch nur kleine Auswirkungen auf das System, nämlich nur lokale Änderungen, zur Folge haben sollten.

als lineare Liste, als Feld, als alphabetischer Suchbaum, als Hashtabelle usw. dargestellt sein. Dynamisches Binden erlaubt nun, die Suche nach einem Buch eines bestimmten Autors unabhängig von dieser Darstellung durch einen Aufruf der Form

```
bücher.suche(‘‘Meyer’’)
```

zu beschreiben (wobei die Größe `bücher` die Menge aller Bücher bezeichnet). Zur Laufzeit des Systems würde dann die geeignete Version der Routine `suche` bestimmt werden, die zu der tatsächlichen Darstellung der Büchermenge paßt. Damit kann sich ein Kunde auf die Anforderung der allgemeinen Dienstleistung (die Suche nach einem Buch) beschränken und es dem Sprachsystem überlassen, automatisch die geeignete Implementierung zu finden.

Die Fähigkeit von Operationen, sich automatisch den Objekten anzupassen, auf die sie angewandt werden, ist eine der wichtigsten Eigenschaften objektorientierter Systeme. Natürlich birgt diese auch gewisse Risiken in sich, denn es wäre ja prinzipiell auch möglich, bei der Redefinition des Gehalts in der Klasse `UNI-ANGESTELLTE` etwas völlig anderes zu berechnen – etwa das Alter im Jahre 1783. Einen derartigen Unsinn kann man nur durch Zusicherungen vermeiden, da Zusicherungen an eine Routine einer Klasse auch für alle Nachkommen gelten müssen – auch dann wenn die Routine umdefiniert wurde. Hierauf kommen wir im Abschnitt 3.8.10 genauer zu sprechen.

3.8.5 Deklaration durch Assoziation

Normalerweise gibt es bei der Vererbung von Routinen wenig Komplikationen. Die meisten Routinen beziehen sich auf features des aktuellen Objektes und damit ist klar, daß sich geerbte Routinen auf das aktuelle Objekt der Erbenklasse beziehen. Die Typen der Objekte, auf denen konkret gearbeitet werden soll, liegen also jederzeit eindeutig fest. Dies wird jedoch anders, wenn lokale Variablen oder formale Parameter benutzt werden, deren Typ die deklarierende Klasse oder – noch schlimmer – eine Ahnenklasse der deklarierenden Klasse ist. In diesem Fall ist nicht klar, welchen Typ diese Parameter in einer Erbenklasse bekommen sollen.

In Abbildung 3.15 auf Seite 80 hatten wir in der Klasse `PERSON` das feature `geschwister` vom Typ `ARRAY[PERSON]` deklariert. Dieses feature vererbt sich auf die Klassen `ARBEITNEHMER` und `ENTLEIHER`, aber es ist klar, daß der Typ nach wie vor `ARRAY[PERSON]` sein muß, denn die Geschwister eines Arbeitnehmers sind nicht notwendigerweise auch Arbeitnehmer.

Anders sieht dies aus, wenn wir die Klasse `ARBEITNEHMER` erweitern um ein feature `kollegen` vom Typ `ARRAY[ARBEITNEHMER]`. Dieses feature vererbt sich auch auf die Klasse `UNI-ANGESTELLTE` und es ist klar, daß die Kollegen eines Universitätsangestellten immer Universitätsangestellte sind (gleiches gilt auch für Bankangestellte, Steuerbeamte, Bauarbeiter usw.). Eine Zuweisung der Form

```
u1 := u2.kollegen@5,
```

wobei `u1,u2` vom Typ `UNI-ANGESTELLTE` sind, ist also vom intuitiven Verständnis her immer korrekt, würde aber dennoch vom Eiffel-Compiler abgewiesen, da sie die Regel der Typverträglichkeit (Seite 103) verletzt. `u2.kollegen@5` wäre vom Typ `ARBEITNEHMER` und könnte somit der Größe `u1` nicht zugewiesen werden.

Um dieses Problem zu lösen, bietet Eiffel die *Deklaration durch Assoziation* an. Anstatt den Typ von features, lokalen Variablen oder formalen Parametern genau zu fixieren, kann man ihn an den Typ einer anderen Größe – den sogenannten *Anker* (*anchor*) – binden. Die syntaktische Form einer solchen Deklaration ist

```
entity: like anchor.
```

Dabei muß der Typ X von `anchor` ein Klassentyp und nicht von der Form **like** `other_anchor` sein. `anchor` kann ein Attribut, eine Funktion, ein formales Argument einer Routine (nur für Deklarationen innerhalb dieser Routine) oder die Größe `Current` sein.

Die Deklaration durch Assoziation bedeutet, daß `entity` immer denselben Typ haben wird wie das Objekt, auf das sich `anchor` bezieht. Innerhalb der deklarierenden Klasse ist der Typ von `entity` also X. In den

Nachkommenklassen aber folgt `entity` automatisch jeder Veränderung des Typs von `anchor`, ohne daß es hierzu expliziert redefiniert werden müßte.

Unser obiges Problem mit dem feature `kollegen` könnten wir also dadurch lösen, daß wir deklarieren

```
kollegen: ARRAY[like Current].
```

Hierdurch wird sichergestellt, daß `u2.kollegen@5` immer vom gleichen Typ ist wie `u2`.

Ein weiterer und vielleicht noch wichtigerer Vorteil dieser Technik besteht darin, bei einem Bündel von Klassenelementen (Attribute, Funktionsergebnisse, formale Argumente, lokale Variablen) sicherzustellen, daß bei einer Redefinition eines Elements der Typ der anderen automatisch mit angepaßt wird, ohne daß dies explizit deklariert werden muß. Auf diese Art vermeidet man Inkonsistenzen durch Flüchtigkeitsfehler. Man bindet daher alle Elemente durch **like** an den Typ eines einzigen und braucht nur das erste zu ändern.

In generischen Klassen der Eiffel-Bibliothek wird diese Technik sehr häufig eingesetzt. So wird zum Beispiel in der Klasse `ARRAY[X]` die Funktion `item` als Anker benutzt, deren Standardtyp der generische Parameter `X` ist. Die Prozedur `put` bindet ihr formales Argument an diesen Anker, so daß sichergestellt ist, daß nur Werte in ein Feld eingefügt werden, die zum Typ des Funktionsergebnisses von `item` passen – auch dann, wenn es nötig sein sollte, in einer Nachkommenklasse den Ergebnistyp von `item` abzuändern.

```
class ARRAY[X]
  ⋮
  put(val:like item,i:INTEGER) is -- Weise dem Element mit Index i den Wert val zu
          require
              lower <=i; i <= upper
          do...
          ensure
              item(i)=val
          end; -- put
  ⋮
end -- class ARRAY[X]
```

Abbildung 3.34: *Typdeklaration durch Assoziation*

Eine weitere wichtige Anwendung findet man in den Basisoperationen, die auf allen Klassen agieren sollen wie `copy`, `clone` und `equal`. Um beim Aufruf von `equal(some,other)` zu ermöglichen, daß der Typ der Parameter `some` und `other` gleich sein muß, sonst aber keinen Einschränkungen unterliegt, deklariert man

```
equal(some:ANY,other:like some):BOOLEAN
```

Die Klasse `ANY` ist dabei die Oberklasse aller Eiffel-Datentypen (und ohne weitere Deklaration Ahne jeder Klasse). `some` hat also einen beliebigen Typ und `other` denselben. Ist dies nicht der Fall, so kann bereits der Compiler einen Typfehler entdecken. Ohne die Deklaration durch Assoziation müßte jede Typkombination zum Vergleich zugelassen werden, was nicht sehr sinnvoll ist.

Deklaration durch Assoziation ist ein rein statischer Mechanismus, der im wesentlichen dazu dient, viele fehleranfällige Redeklarationen zu vermeiden. Prinzipiell könnte man auch ohne diese Technik auskommen. Dies würde aber die Einheit von Wiederverwendbarkeit und dem Typkonzept von Eiffel erheblich stören.

3.8.6 Deferred Classes: Abstrakte Datentypen in Eiffel

Eine der interessantesten Anwendungen von Vererbung und dynamischem Binden besteht in der Möglichkeit, in einer Klassendeklaration die Implementierung einer Routine überhaupt nicht anzugeben, sondern auf die Erbenklassen *aufzuschieben*. Das bedeutet, daß eine Routine außer ihrer Deklaration einschließlich der Vor- und Nachbedingungen keinerlei Routinenrumpf enthält, sondern nur einen Vermerk darüber, daß es den

Nachkommenklassen obliegt, konkrete Anweisungsteile bereitzustellen. Eiffel verwendet hierfür das Schlüsselwort **deferred** und sieht auch die gesamte Klasse als deferred, also aufgeschoben an. Selbstverständlich sind aufgeschobene Klassen als solche nicht "ausführbar".

Definition 3.8.9 (Aufgeschobene Klassen)

Eine Routine heißt aufgeschoben (deferred), wenn ihr Rumpf anstelle des Anweisungsteils das Schlüsselwort **deferred** *enthält.*

Eine Klasse heißt aufgeschoben, wenn sie eine aufgeschobene Routine enthält oder eine aufgeschobene Routine geerbt und nicht ausführbar redefiniert hat. Sie muß in ihrer Deklaration als **deferred class** *gekennzeichnet werden.*

Aufgeschobene Klassen sind das Eiffel-Gegenstück zu der axiomatischen Spezifikation abstrakter Datentypen, da sie wie diese nur die Dienstleistungen einer Klasse und deren Eigenschaften beschreiben, nicht aber die Implementierung. Damit können sie benutzt werden, um *abstrakte Schnittstellen innerhalb von ausführbaren Programmen* zur Verfügung zu stellen. Das bedeutet, daß bereits während der Entwicklungsphase eine Kundenklasse der aufgeschobenen Klasse übersetzt und teilweise getestet werden kann, ohne daß eine konkrete Realisierung der Lieferantenklasse bereits vorliegt. Vererbung erlaubt es dann, beliebige konkrete Realisierungen und Verfeinerungen der aufgeschobenen Klasse anzugeben.

Diese Technik unterstützt einen Entwurf großer Softwaresysteme auf einer hohen Abstraktionsebene. Es ist möglich, innerhalb ein und derselben Sprache zunächst eine Systemarchitektur zu entwerfen, jedes Modul abstrakt zu beschreiben und Implementierungsdetails in späteren Verfeinerungen auszuführen[29]. Da auch unimplementierte Routinen präzise durch Vor- und Nachbedingungen charakterisiert werden können, verschwindet ein Großteil der konzeptionellen Lücke zwischen Entwurf und Implementierung.

Ein weiterer Vorteil aufgeschobener Klassen ist die Möglichkeit einer gleitenden Realisierung von Softwaresystemen ("*partielle Implementierung*"). In aufgeschobenen Klassen können auch ausführbare Routinen vorkommen, die dann für alle Erben zur Verfügung stehen. Auf diese Weise erhält man eine Klassifikation der Implementierungen von der abstrakten Form in die verschiedenen Formen ihrer konkreten Realisierung.

Darüber hinaus bieten deferred classes einen Ersatz für Prozedurparameter in Routinen, die in Eiffel – im Gegensatz zu den meisten höheren Programmiersprachen – *nicht* erlaubt sind. Solche Parameter aber sind zum Beispiel sinnvoll, wenn man in einer mathematischen Programmpaket Integration und ähnliche Verfahren anbieten will, natürlich ohne dabei bereits die Funktion festzulegen, deren Integral berechnet werden soll. In Eiffel läßt sich das so simulieren, daß man eine deferred Routine anstelle des Prozedurparameters definiert, die dann in den Erben dieser Klasse durch die jeweilig gewünschte aktuelle Routine ersetzt wird.

Eines der wichtigsten Anwendungsgebiete aber ist die polymorphe Anwendung von gleichartigen (und gleichnamigen) Routinen aus verschiedenen Anwendungsbereichen. So kann man zum Beispiel für graphische Objekte wie Punkte, Vektoren, Segmente, Rechtecke, Polygone usw. jeweils unabhängig Routinen zum verschieben, drehen, spiegeln usw. deklarieren und implementieren. Jede dieser Routinen muß in jeder Klasse völlig unabhängig realisiert werden, führt aber – intuitiv betrachtet – jeweils dasselbe aus. Was liegt also näher, als diese als Routinen einer Klasse `GRAPHISCHE_OBJEKTE` anzusehen, die einen einheitlichen Namen haben und in den jeweiligen Spezialisierungen ihre besondere Ausprägung finden? Diese gemeinsame Oberklasse *muß* eine aufgeschobene Klasse sein, denn es gibt keine feste Anweisungsfolge, die sagt, wie man graphische Objekte generell verschiebt. Man kann nur in den Vor- und Nachbedingungen festlegen, was die Effekte sein sollen. Aufgeschobene Klassen bilden also den *Vereinigungstyp* aller Erbenklassen. Dies erlaubt es den Kunden einer deferred Klasse wie `GRAPHISCHE_OBJEKTE`, abstrakte Programme zu schreiben, die für jeden konkreten Spezialfall – also Punkte, Vektoren, Segmente, Rechtecke, Polygone usw. – funktionieren, da die Erbenklassen eine Implementierung bereitstellen.

[29] Normalerweise muß man hierfür mindestens zwei formale Sprachen verwenden – eine Spezifikations- und eine Implementierungssprache. Dies bringt jedoch eine Reihe von Problemen mit sich, wenn man von der einen auf die andere Sprache übergehen will und dabei sicherstellen möchte, daß die Lösung in der Implementierungssprache der abstrakten Beschreibung in der Spezifikationssprache entspricht.

```
deferred class LIST[X]
feature
  length: INTEGER;
  empty: BOOLEAN  is -- Ist die Liste leer ?
         do Result := length=0 end;
  new is -- Erzeuge leere Liste
         deferred
         ensure
             empty
         end; -- new
  cons(r:X) is  -- Hänge r vor die Liste
         deferred
         ensure
             not empty; head = r
              -- tail würde jetzt old liefern
         end; -- cons
  head: X is -- Erstes Element
         require
             not empty
         deferred end; -- head
  tail  is  -- Entferne erstes Element
         deferred
         ensure
             old empty implies empty
         end -- tail
invariant
  nonnegative_size: size >= 0
end -- class LIST[X]
```

Abbildung 3.35: *Aufgeschobene Klasse*

Wir wollen aufgeschobene Klassen am Beispiel der generischen Klasse aller Listen illustrieren[30], die wir bereits in Abbildung 3.4 als abstrakten Datentyp beschrieben und in Abbildung 3.26 als generische Klasse `LIST` skizziert hatten. Diese Klasse stellt Namen von Grundoperationen zur Verfügung, welche eine Art Mindestkollektion von Operationen auf Listen beschreiben.

Eine aufgeschobene Klasse muß mit den Schlüsselworten **deferred class** beginnen, um klarzustellen, daß die Klasse nicht vollständig implementiert ist, da sie mindestens eine aufgeschobene Routine enthält. Dabei ist es durchaus erlaubt, daß einige Routinen einen vollständigen Anweisungsteil enthalten. Wir haben hierzu in Abbildung 3.35 die Klasse `LIST` um das feature `length` erweitert, welches eine einfache (und in allen Erben gültige) Implementierung der Funktion `empty` ermöglicht, die *nicht* aufgeschoben ist. Andere Routinen sind aufgeschoben, was man daran erkennt, daß anstelle des sonst üblichen mit **do** beginnenden Anweisungsteils das Schlüsselwort **deferred** steht. *Alle anderen Bestandteile einer Routine – Kopf, Vor- und Nachbedingungen – bleiben erhalten .*

Man beachte, daß die aufgeschobene Klasse keine creation-Klausel hat, auch wenn die Routine `new`, die eigentlich dafür vorgesehen ist, deklariert wird. Die Begründung hierfür ist einfach: da Dienstleistungen einer aufgeschobenen Klasse eventuell nicht ausführbar sind, macht es wenig Sinn, Exemplare dieser Klasse zu erzeugen.

[30] An dieser Stelle sei darauf hingewiesen, daß die in diesem Skript benutzte Klasse `LIST` dem Listenkonzept entspricht, das in der Mathematik und in funktionalen Programmiersprachen benutzt wird, aber *nicht* der gleichnamigen Klasse der Eiffel-Basisbibliothek (das habe ich leider erst entdeckt, nachdem die Hälfte des Skripts bereits geschrieben war). Die Eiffel-Klasse `STACK` kommt in ihrem Verwendungszweck den im Skript benutzten Listen am nächsten, verwendet jedoch andere Routinennamen.

Für die Erklärung der Konzepte spielt dieser Unterschied keine Rolle, denn man hätte in der Eiffel-Basisbibliothek durchaus die hier verwandten Bezeichnungen benutzen können. Bitte berücksichtigen Sie jedoch bei Implementierungsarbeiten, daß der Compiler die Routinen `new, head, tail`, und `cons` in der Eiffel-Klasse `LIST` nicht finden wird.

Entwurfsprinzip 3.8.10 (Regel der Nicht-Erzeugung aufgeschobener Klassen)

Auf Größen, deren Typ eine aufgeschobene Klasse ist, darf keine Initialisierungsprozedur angewandt werden.

Beim ersten Hinsehen erscheint diese Regel sehr restriktiv und das wäre sie auch ohne Polymorphismus und dynamisches Binden. Hält man sich jedoch die oben genannten Anwendungen aufgeschobener Klassen vor Augen, so besagt diese Regel, daß Größen, deren Typ eine aufgeschobene Klasse ist, als Oberbegriff für Objekte der Erbenklasse zu verstehen sind. Die Größe ist *polymorph*, kann also Objekte verschiedener Klassen kennzeichnen und hierauf alle Routinen der Oberklasse anwenden.

Sie ist allerdings kein Bezeichner für eine eigene Art von Objekten. Um also Objekte zu erzeugen, die mit dieser Größe angesprochen werden sollen, muß man festlegen, welche Art von Objekten erzeugt werden sollen und hierfür die Initialisierungsprozedur der entsprechenden Erbenklasse anwenden. Dies geschieht durch den bereits in Abschnitt 3.3.6 angesprochenen Aufruf

`!ERBEN_KLASSE!entity` bzw. `!ERBEN_KLASSE!entity.init(argumente)`

je nachdem ob die Erbenklasse eine spezifische Initialisierungsprozedur bereitstellt oder nicht. Da die Deklaration einer Routine als Initialisierungsprozedur sich aber nicht vererbt sondern lokal in jeder Klasse stattfinden muß (vgl. Abschnitt 3.8.1), braucht man in aufgeschobenen Klassen auch keine Initialisierungsprozedur zu deklarieren.

Nachkommen einer aufgeschobenen Klasse können nun effektive Versionen einer aufgeschobenen Routine anbieten[31]. Da für diese Routine bisher keine Anweisungsfolge bekannt war, gilt sie auch nicht als redefiniert. Die Erbenklasse muß sie also nicht in einer redefine-Klausel aufführen (vergleiche die Regel 3.8.4 der Redefinition). Natürlich muß die Implementierung die Vor- und Nachbedingungen einhalten, die in der Ahnenklasse vereinbart wurden.

3.8.7 Mehrfachvererbung

In den bisherigen Beispielen hatten Erben immer nur ein Elternteil. Dies muß aber nicht immer der Fall sein. So sind zum Beispiel studentische Hilfskräfte gleichzeitig Studenten einer bestimmten Universität und Arbeitnehmer. Die Klasse `HILFSKRAFT` erbt also Eigenschaften von zwei Klassen, nämlich `STUDENT` und `ARBEITNEHMER`.

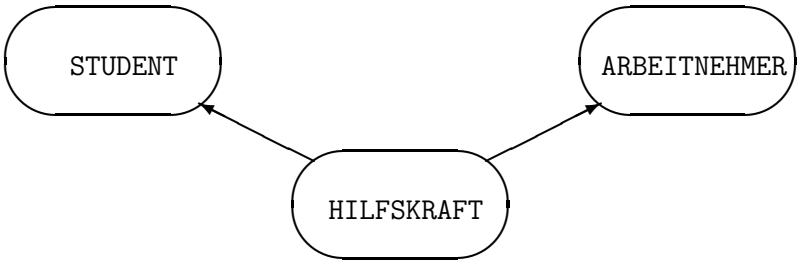


Abbildung 3.36: *Mehrfachvererbung als Diagramm:* `HILFSKRAFT` *erbt von* `STUDENT` *und* `ARBEITNEHMER`

Mehrfachvererbung läßt sich in Eiffel sehr leicht ausdrücken. In der Vererbungsklausel werden einfach mehrere Elternklassen zusammen mit den eventuell notwendigen Modifikationen geerbter features angegeben. Die Möglichkeit mehrere Elternklassen anzugeben erklärt auch die Verwendung des Schlüsselwortes **end** am Ende der feature-Anpassung einer Elternklasse. Abbildung 3.37 zeigt die Deklaration einer Klasse `HILFSKRAFT`, die als direkter Nachkomme der Klassen `STUDENT` und `ARBEITNEHMER` erklärt ist.

[31]Der Vollständigkeit halber sei auch erwähnt, daß man umgekehrt auch eine effektive Version einer Routine in eine aufgeschobene Routine umwandeln kann. Dies geschieht durch Verwendung einer Unterklausel in der Erbklausel, die mit dem Schlüsselwort **undefine** beginnt. Details findet man in [Meyer, 1992, Kapitel 10.16]

```
class STUDENT
feature
     universität: STRING;
     fachbereich: INTEGER;
     matrikelnummer: INTEGER
end; -- class STUDENT

class HILFSKRAFT
inherit
     ARBEITNEHMER
       redefine
           gehalt
       export
           name, vornamen, adresse, gehalt
       end; -- feature-Anpassung von ARBEITNEHMER
     STUDENT
       export
           universität, fachbereich
       end -- feature-Anpassung von STUDENT
feature
    monatsstunden: INTEGER;
    gehalt: REAL is -- Gehalt berechnen
         do Result :=  Festgehalt nach  monatsstunden end
end -- class HILFSKRAFT
```

Abbildung 3.37: *Mehrfachvererbung*

Eine häufige Anwendung von Mehrfachvererbung ist die Implementierung eines abstrakten Datentyps durch Routinen eines anderen, also zum Beispiel die Simulation von Listen durch Felder. Da sowohl Listen als auch Felder bereits durch Klassen beschrieben waren, ist der sinnvollste Weg, dies zu tun, eine spezielle Implementierung `FIXED_LIST` anzugeben, welche die Klasse aller durch Felder realisierten Listen beschreibt. Die Objekte dieser Implementierungsklasse sind also sowohl Listen als auch spezielle Felder. Deshalb sollte `FIXED_LIST` Erbe sowohl von der Klasse deferred class `LIST` als auch von `ARRAY` sein. Abbildung 3.38 zeigt eine mögliche Realisierung dieser Klasse.

`FIXED_LIST` bietet denselben Funktionsumfang wie `LIST` und stellt hierzu effektive Versionen der aufgeschobenen Routinen bereit, die in Begriffen von `ARRAY`-Operationen implementiert wurden. Deshalb werden alle features von `LIST` wieder exportiert, aber keines von `ARRAY`. Die features `length` und `empty` waren bereits effektiv deklariert worden und müssen nicht erneut aufgeführt werden. `new` muß als Initialisierungsprozedur vereinbart werden. Auf die erneute Angabe der Vor- und Nachbedingungen kann verzichtet werden, da sie sich nicht ändern (siehe Abschnitt 3.8.10)

Die Implementierung beschreibt eine Liste als Feld, welches die Listenelemente in umgekehrter Reihenfolge auflistet (das ist am einfachsten zu implementieren) und die Größe `length` als Zeiger auf das erste Element benutzt. `new` erzeugt ein Feld der Länge 0, `cons` erweitert die Länge um 1, vergrößert das Feld mit `resize`, wenn (**if**) dies hierfür erforderlich ist, und trägt das genannte Listenelement an der durch `length` bezeichneten "ersten" Stelle ein. `head` greift einfach auf diese "erste" Stelle zu und `tail` verschiebt den Zeiger `length` um 1 zurück, wenn dies möglich ist.

Das Beispiel ist typisch für eine verbreitete Art von Mehrfachvererbung, die man als *Vernunftehe* bezeichnen könnte. Eine Klasse bringt den Funktionsumfang, die andere die Implementierungswerkzeuge. Beide zusammen liefern eine effiziente Implementierung der gewünschten Routinen.

[Meyer, 1988, Kapitel 10.4.2-10.4.4.] diskutiert weitere verbreitete Anwendungsformen für Mehrfachvererbung. Unter diesen ist besonders die Möglichkeit einer einheitlichen *Testmethodik* für Klassen hervorzuheben. Jeder Test basiert auf gewissen Grundmechanismen wie Einlesen und Abspeichern von Benutzereingaben, Ausgeben

```
class FIXED_LIST[X]
inherit
     LIST[X];
     ARRAY[X]
       export {} all
creation new
feature
  -- Bereits effektiv deklariert
  --   length: INTEGER
  --   empty: BOOLEAN
  new is -- Erzeuge leere Liste
         do
             make(1,0);  length:=0
         end; -- new
  cons(r:X) is  -- Hänge r vor die Liste
         do
             if length = upper then resize(1,length+1) end;
             length := length+1;
             put(r,length)
         end; -- cons
  head: X is -- Erstes Element
         do
             Result := item(length)
         end; -- head
  tail  is  -- Entferne erstes Element
         do
             if length/=0 then length:=length-1 end
         end -- tail
end -- class LIST[X]
```

Abbildung 3.38: *Mehrfachvererbung: Realisierung einer* deferred class *durch Routinen einer anderen*

von Ergebnissen etc. Derartige allgemeine Mechanismen sollten der Wiederverwendbarkeit wegen innerhalb einer Klasse `TEST` gesammelt werden. Will man nun eine Klasse K testen, so braucht man einfach nur eine Klasse K`_TEST` als Erbe von K und `TEST` zu deklarieren. Auf diese Art erspart man sich, für jede Klasse die grundlegenden Testmechanismen erneut schreiben zu müssen.

3.8.8 Umbenennung

Erbt eine Klasse von mehreren Klassen, so hat sie direkten Zugriff auf alle features der Elternklassen, ohne daß qualifiziert werden muß. Dies kann in manchen Fällen zu Namenskonflikten führen. Es wäre zum Beispiel sinnvoll, die in Abbildung 3.37 angegebene Klasse der studentischen Hilfskräfte besser als Erben der Klassen `STUDENT` und `UNI-ANGESTELLTE` zu deklarieren, da eine studentische Hilfskraft ja nur bei einer Universität angestellt werden kann.

```
class HILFSKRAFT
inherit
     UNI-ANGESTELLTE
       ⋮
       end; -- feature-Anpassung von UNI-ANGESTELLTE
     STUDENT
       ⋮
       end; -- feature-Anpassung von STUDENT
feature
   ⋮
end -- class HILFSKRAFT
```

Das Problem ist nun, daß beide Elternklassen, `STUDENT` und `UNI-ANGESTELLTE` das feature `fachbereich` benutzen. Bei Aufruf dieses features ist also nicht klar, welches der beiden gemeint sein soll. Solche Konflikte müssen in einer Programmiersprache verboten werden, da die Wahl des geerbten features ja nicht zufällig getroffen werden kann und es unwahrscheinlich ist, daß in beiden Fällen dasselbe gemeint ist. So bezeichnet zum Beispiel das feature `fachbereich` bei Studenten das Hauptfach, das sie studieren, und bei Universitätsangestellten den Fachbereich, an dem sie arbeiten. Daher gilt

In Eiffel *sind Mehrfachvererbungen mit Namenskonflikten verboten.*

Die Klasse `HILFSKRAFT`, so wie sie oben beschrieben ist, würde also vom Compiler abgelehnt werden.[32]

Wie kann man sich nun behelfen? Es wäre wenig sinnvoll zu verlangen, daß die Namen aller im System deklarierten Routinen verschieden sein müssen, da dies eine praktische Weiterentwicklung einer Eiffel-Bibliothek erheblich erschweren würde. Man kann die Eltern nicht dafür verantwortlich machen, daß in einer Erbenklasse Namenskonflikte auftreten. Es ist eher das Problem der Erben, welche diese Kollision durch die Wahl der Elternklassen erzeugt haben. Deshalb muß das Problem auch in der Erbenklasse gelöst werden.

Die einfachste Möglichkeit hierfür ist eine *Umbenennung* von features, d.h. ein Mechanismus, der es ermöglicht, features unter anderen Namen anzusprechen[33]. Hierzu gibt man in der Erbklausel eine rename-Unterklausel an, welche beschreibt, unter welchen Namen geerbte features weiterbenutzt werden sollen. In unserem Beispiel würde man also das feature `fachbereich` der Klasse `STUDENT` in `studienfachbereich` umbenennen. Die Eiffel Syntax für Umbenennungen lautet

rename
feature$_1$ **as** *new_feature*$_1$ **,**
⋮
feature$_n$ **as** *new_feature*$_n$

Man achte darauf, daß mehrere Umbenenungen durch Komma getrennt sind und nicht durch Semikolon.

Umbenennung hat, wie das Beispiel in Abbildung 3.39 zeigt, auch einen Einfluß auf die features, die in den export- und redefine-Klauseln genannt werden. Allgemein gilt, daß nach einer Umbenennung nur noch der neue Name bekannt ist. Daher muß die rename-Unterklausel vor allen anderen Unterklauseln einer Erbklausel genannt werden.

Entwurfsprinzip 3.8.11 (Regel der Feature-Umbenennung)

Features, die in einer rename*-Unterklausel umbenannt wurden, können in* export- *und* redefine*-Klauseln nur unter ihrem neuen Namen angesprochen werden.*

Neben der Beseitigung von Namenskonflikten gibt es auch weitere sinnvolle Anwendungen für Umbenennungen. So ist oft der Name, unter denen eine Klasse ein feature erbt, für den speziellen Verwendungszweck nicht aussagekräftig genug. Die Routine `gehalt` hat sich bisher zum Beispiel nur auf das Bruttogehalt bezogen, für das Arbeitgeber und Finanzamt sich interessieren. Für die Familie zählt aber, was nach Steuern und Sozialabgaben übrigbleibt – das Nettogehalt. In Erbenklassen von `ARBEITNEHMER` kann es also durchaus ratsam sein, das feature `gehalt` in `bruttogehalt` umzubenennen.

Umbenennung ist ein syntaktischer Mechanismus, der sich auf die Namensgebung für ein feature innerhalb einer Klasse und ihrer Erben auswirkt. Wichtig ist jedoch, daß Umbenennung *keinen Effekt auf die Vorfahrenklassen* hat. Das feature ist für Größen, deren Typ eine Ahnenklasse der umbenennenden Klasse ist, nach wie

[32] Diese Regel gilt nicht, wenn die Klassen, in denen die kollidierenden Namen deklariert wurden, deferred sind. In diesem Falle besteht ja keine Schwierigkeit festzustellen, welche Implementierung bei einem Aufruf des features gefragt ist – es gibt ja keine. Die Regelung für diesen Fall besagt, daß beide aufgeschobenen Routinen zu einer neuen aufgeschobenen Routine vereinigt werden, die ggf. auch dann *eine* effektive Version erhalten kann. Mehr zu diesem **join** Mechanismus und den Möglichkeiten, diesen durch **rename** und **undefine** auch auf effektive features auszudehnen, findet man in [Meyer, 1992, Kapitel 10.17/18].

[33] Damit ist Umbenennung komplementär zur Redefinition, die mit demselben Namen unterschiedliche Implementierungen eines features anspricht – abhängig vom Typ des Objektes. Redefinition ist also ein *semantischer* Mechanismus, Umbenennung ist ein rein *syntaktischer*Mechanismus, der gewährleistet, daß dieselbe Implementierung eines features – abhängig vom Typ des Objektes – unter unterschiedlichen Namen angesprochen werden kann.

```
class HILFSKRAFT
inherit
      UNI-ANGESTELLTE
        redefine
            gehalt
        export
            name, vornamen, adresse, gehalt
        end; -- feature-Anpassung von UNI-ANGESTELLTE
      STUDENT
        rename
            fachbereich as studienfachbereich
        export
            universität, studienfachbereich
        end -- feature-Anpassung von STUDENT
feature
    monatsstunden: INTEGER;
    gehalt: REAL is -- Gehalt berechnen
         do Result :=  Festgehalt nach  monatsstunden end
end -- class HILFSKRAFT
```

Abbildung 3.39: *Vererbung mit Umbenennung*

vor unter dem *alten* Namen verfügbar (ansonsten würde jede Umbenennung in einer Nachfolgerklasse massive Konsequenzen für die gesamte Eiffel-Bibliothek haben). Die Verwaltung der Namen durch den Programmierer kann also lokal geschehen. Die globale Verwaltung wird von der Eiffel-Programmierumgebung gehandhabt.

3.8.9 Wiederholtes Erben

Die Möglichkeit der Mehrfachvererbung bringt neben der Gefahr von Namenskollisionen noch ein weiteres Problem mit sich: was passiert, wenn eine Klasse mehrmals ein Nachkomme ein und derselben Klasse ist? Dieses Problem tritt notwendigerweise irgendwann einmal auf, wenn man mehrfaches Erben zuläßt. Man spricht in diesem Fall von *wiederholtem Erben*.

In Abbildung 3.39 hatten wir die Klasse `HILFSKRAFT` als Erben von `STUDENT` und `UNI-ANGESTELLTE` deklariert. Dies führte dort zu keinerlei Problemen, weil `STUDENT` bisher nur über `universität, fachbereich` und `matrikelnummer` erklärt war, was keinesfalls der Realität entspricht. Im Normalfall wird man die Klasse der Studenten genauso als Erben von `PERSON` deklarieren wie die Klasse der Universitätsangestellten. Dann aber stehen wir vor dem Problem, daß `HILFSKRAFT` auf zwei Wegen die features der Klasse `PERSON` erbt.

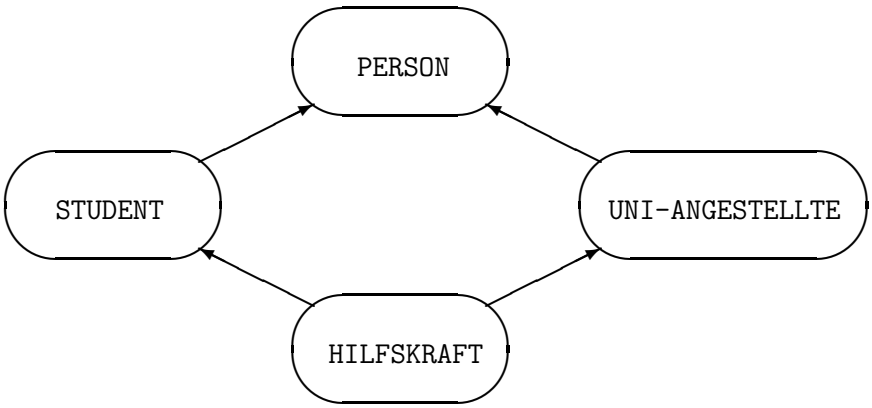


Abbildung 3.40: *Wiederholtes Erben*

Prinzipiell wäre es sogar möglich, daß eine Klasse K dieselbe Elternklasse *mehrfach* in ihrer Erbklausel auflistet, um ggf. ein feature unter mehreren Namen anzubieten. Da man diese Konstellation nicht prinzipiell ausschließen kann, muß sie sehr sauber geregelt werden.

Man könnte nun verlangen, daß alle wiederholt auftretenden features durch Umbenennung getrennt werden müssen, aber das Beispiel der Hilfskräfte zeigt, daß dies nicht gerade sinnvoll ist. Die doppelt von `PERSON` geerbten features sind ja in Wirklichkeit absolut identisch, da sie weder umbenannt noch redefiniert wurden. Deshalb möchte man sie ohne Komplikationen wieder als einfach vorkommende features benutzen können.

Entwurfsprinzip 3.8.12 (Regel des wiederholten Erbens)

> *Bei wiederholtem Erben wird jedes feature des gemeinsamen Vorfahren als gemeinsam genutzt (shared) angesehen, wenn es auf dem gesamten Vererbungspfad nicht umbenannt wurde. Features, die auf mindestens einem Wege umbenannt wurden, werden als vervielfältigt (replicated) angesehen.*

Die Regel besagt also, daß alleine der Name eines wiederholt ererbten features festlegt, wie oft es in einem Erben vorkommen muß. Normalerweise erzeugt ein wiederholt geerbtes feature *keine Kollision*, sondern wird wieder zu einem einzelnen feature vereinigt.

Jede Umbenennung eines features auf dem Vererbungpfad von der *erstmalig deklarierenden Ahnenklasse* zum Erben erzeugt jedoch eine weitere Kopie dieses Merkmals. Hierfür gibt es durchaus eine Reihe von Anwendungen. So könnten zum Beispiel `STUDENT` und `UNI-ANGESTELLTE` von der Klasse `PERSON` ein weiteres feature `adresse` geerbt haben – im ersten Fall als Wohnadresse, im zweiten als Adresse des Arbeitsplatzes. In `HILFSKRAFT` kommen nun beide Versionen vor, so daß dieses feature beim Erben umbenannt werden sollte und somit vervielfältigt wird. Die Umbenennung kann in der Klasse `HILFSKRAFT` stattfinden, aber sinnvoller wäre es, dies bereits vorher zu tun.

Die Regel des wiederholten Erbens gilt gleichermaßen für Routinen und Attribute. Nicht erlaubt ist daher, daß eine gemeinsam genutzte Routine (in der definierenden Klasse) Verweise auf Attribute oder Routinen enthält, die auf dem Vererbungspfad umbenannt wurden. Wegen der Vervielfältigung wären diese in der Erbenklasse nicht mehr eindeutig zu identifizieren. Derartige Routinen müssen auf dem Vererbungsweg ebenfalls an passender Stelle umbenannt werden. (Tut man dies zu spät, so muß man sogar umbenennen und redefinieren!)

So, wie sie bisher formuliert wurde, hinterläßt die Regel des wiederholten Erbens noch ungeklärte Probleme bei features, die an formale generische Parameter gebunden sind. In der folgenden Situation, die in komplexerer Form auch bei indirektem wiederholten Erben auftreten kann, wird ein feature `f` auf dem Vererbungsweg nicht umbenannt, erhält aber beim Erben verschiedene Typen.

```
class GENERISCH[X] feature
    f:X; ....
end -- GENERISCH

class ERBE inherit
    GENERISCH[INTEGER]; GENERISCH[PERSON]
 ⋮
end -- ERBE
```

Bei einer gemeinsamen Nutzung von `f` wäre unklar, ob `f` vom Typ `INTEGER` oder vom Typ `PERSON` ist. Ähnliche Mehrdeutigkeiten träten auf, wenn `f` als eine Routine mit einem formalen Argument vom Typ `X` deklariert worden wäre. Diese Mehrdeutigkeit wird durch die folgende Regel eliminiert.

Entwurfsprinzip 3.8.13 (Regel der Generizität bei wiederholtem Erben)

> *Der Typ eines beim wiederholten Erben gemeinsam genutzten features darf in der Elternklasse kein generischer Parameter sein. Ebenso darf eine gemeinsam genutzte Routine in der Elternklasse keine formalen Argumente enthalten, deren Typ ein generischer Parameter ist.*

Eine Konsequenz dieser Regel ist, daß die oben beschriebene Situation als Namenskonflikt gilt, der durch Umbenennung beim Erben aufgelöst werden muß. Dies führt dann zur Vervielfältigung des features gemäß der Regel des wiederholten Erbens.

Unter Berücksichtigung des wiederholten Erbens können wir jetzt das in Abschnitt 3.8.8 angesprochene Verbot von Namenskonflikten präzisieren.

Entwurfsprinzip 3.8.14 (Umbenennungsregel)

> *In einer Klasse K tritt ein Namenskonflikt auf, wenn zwei Eltern E_1 und E_2 von K ein feature mit demselben Namen* `f` *enthalten.*
>
> *Ein Namenskonflikt ist erlaubt, wenn* `f` *in einem gemeinsamen Vorfahren V von E_1 und E_2 erstmalig definiert wurde, auf keinem Vererbungspfad von V nach E_1 und nach E_2 umbenannt wurde und in seiner Typdeklaration an keine generischen Parameter von V gebunden ist.*
>
> *Nicht erlaubte Namenskonflikte müssen durch Umbenennung aufgelöst werden.*

Weitere Komplikationen bei wiederholtem Erben können durch Redefinition entstehen. Es kann sein, daß features auf verschiedenen Wegen zum gleichen Erben auch verschiedene Bedeutungen erlangt haben können. In diesem Fall liegt zwar keine Namenskollision im eigentlichen Sinne vor, aber die gemeinsame Nutzung würde dennoch zu Problemen führen, da dieses feature nun in mehreren Versionen geerbt wird.

Bei einem Aufruf des features durch eine Größe, deren Typ die Ahnenklasse ist, kann nun keine eindeutige dynamische Bindung zwischen dem Originalnamen des Aufrufs und der tatsächlich zu verwendenen Implementierung mehr stattfinden. Auch Umbenennung löst dieses Problem nicht, da die Ahnenklasse die Namen nicht kennen kann, die eine Erbenklasse neu vergibt (die würde ja einen Eingriff in die Ahnenklasse beim Programmieren der Erben verlangen).

In diesem Fall muß daher die Erbenklasse auswählen, welche Version des features dynamisch an frühere Versionen gebunden wird und welche nicht. Dies geschieht in Eiffel durch eine select-Unterklausel der entsprechenden Erbklausel. Eine solche Unterklausel beginnt mit dem Schlüsselwort **select** und listet danach die *endgültigen* Namen derjenigen features, welche dynamisch an frühere Versionen gebunden werden sollen. Nicht selektierte Versionen sind nur von den Nachfolgerklassen, nicht aber von den Ahnen her erreichbar.

[Meyer, 1992, Kapitel 11] liefert eine detaillierte Betrachtung des wiederholten Erbens und die genaue Festlegung weiterer Sonderfälle. Wir wollen im Rahmen dieser Veranstaltung nicht weiter darauf eingehen. Wiederholtes Erben wird in der Praxis nicht sehr häufig gebraucht. Eine Systemkonfiguration muß schon verhältnismäßig komplex sein, bevor diese Einrichtung wirklich benötigt wird.

In unkomplizierten Fällen ensteht wiederholtes Erben eher durch einen Anfängerfehler, bei dem die Transitivität der Vererbung ignoriert wird. Oft wird zum Beispiel die Klasse `STD_FILES` explizit mit in die Erbklausel aufgenommen, um sicherzustellen, daß Ein- und Ausgabekommandos unqualifiziert genutzt werden können. Dabei wird vergessen, daß diese Klasse bereits Vorfahre anderer Klassen ist, die man ebenfalls beerbt. Die Aufzählung von `STD_FILES` wird also überflüssig. Aufgrund der Regel des Wiederholten Erbens ist dies aber nur ein uneleganter Programmierstil, der nicht zu einem Fehler führt.

3.8.10 Vererbung und Zusicherungen

Am Ende des Abschnitts 3.8.4 hatten wir bereits darauf hingewiesen, daß Redefinition prinzipiell die Möglichkeit einer kompletten Veränderung der Semantik ererbter features ermöglicht. Dies birgt natürlich das Risiko eines Mißbrauchs in sich, der gerade durch das dynamische Binden sehr gefährlich werden könnte. Dies kann nur durch ein enges Zusammenspiel von Zusicherungen und Vererbung verhindert werden. Die Grundregel hierfür haben wir bereits öfter angedeutet:

Was die Ahnen versprechen, müssen die Erben erfüllen!

Das bedeutet also, daß die Erbenklassen nicht nur die Rechte, sondern auch die Pflichten ihrer Ahnenklassen besitzen: die neue Klasse muß die Kontrakte der ererbten Routinen und ihre Klasseninvarianten einhalten. Dies wird in Eiffel durch die folgenden Regeln sichergestellt.

Entwurfsprinzip 3.8.15 (Regel der Elterninvarianten)

Die Invarianten aller Eltern einer Klasse gelten für die Klasse selbst.

Dies bedeutet, daß die Elterninvarianten *automatisch* den Invarianten der Erbenklasse hinzugefügt werden (im Sinne von **and** bzw. ;), *ohne daß sie explizit genannt werden müssen.*

Entwurfsprinzip 3.8.16 (Regel der Zusicherungsredefinition)

Ist `r` *eine Routine der Klasse K und* `s` *eine Redefinition von* `r` *in einem Nachkommen von K, so müssen die Vorbedingungen von* `s` *schwächer sein als die von* `r` *und die Nachbedingungen stärker.*

Dies bedeutet, daß die redefinierte Version alle korrekten Eingaben der ursprünglichen Form akzeptieren und alle Nachbedingungen einhalten muß. Man darf also die Anforderungen an redefinierte Versionen verschärfen, aber niemals abschwächen. "Stärker" und "schwächer" gilt im logischen Sinne: aus dem Stärkeren muß das Schwächere durch Implikation logisch folgern. In diesem Sinne ist `x>=5` stärker als `x>=0`, `x>=0` **and then** `y<=4` stärker als `x>=0` und `x>=0` stärker als `x>=0` **or else** `y<=4`.

Eiffel 3 unterstützt diese Regel mittlerweile direkt durch das folgende Prinzip

Entwurfsprinzip 3.8.17

- *Fehlt in einer redefinierten Routine die Vor- oder Nachbedingung, so wird die Vor- bzw. Nachbedingung der ursprünglichen Version übernommen.*
- *Vorbedingungen können nur durch Verwendung einer* require else*-Klausel abgeschwächt werden.*
- *Nachbedingungen können nur durch Verwendung einer* ensure then*-Klausel verstärkt werden.*

Es ist also nicht nötig, die alten Vor- und Nachbedingungen zu wiederholen. Sie gelten automatisch, solange sie nicht verändert werden, und um sicherzustellen, daß die Regel der Zusicherungsredefinition eingehalten wird, kann man nur noch bestehende Zusicherungen an eine Routine verändern, *nicht aber völlig neue Vor- und Nachbedingungen festlegen* wie dies bei neu deklarierten features der Fall ist.

Die Modifikation einer Vorbedingung wird durch die Schlüsselworte **require else** gekennzeichnet. Sind die Vor- und Nachbedingungen der ursprünglichen Routine `precondition` und `postcondition`, so werden durch

require else `alternate_precondition`

die Vorbedingungen zu `alternate_precondition` **or else** `precondition` und durch

ensure then `additional_postcondition`

die Nachbedingungen zu `additional_postcondition` **and then** `postcondition`. Hat die ursprüngliche Routine keine Vor- bzw. Nachbedingungen, so wird entsprechend **false** bzw. **true** eingesetzt.

Die Regeln der Zusicherungsredefinition machen unter Berücksichtigung der in Abschnitt 3.7.2 angesprochenen Vertragsmetapher klar, daß die eigentliche Natur von Vererbung sehr viel mit Unterauftragsvergabe zu tun hat. Hat eine Klasse einen Auftrag angenommen, dann muß sie ihn noch lange nicht selbst ausführen, sondern kann Erbenklassen aufrufen, die diesen Auftrag – aufgrund klarer umrissener Rahmenbedingungen – einfacher und vielleicht sogar besser ausführen können. Dieser Unterauftrag wird dann durch dynamisches Binden automatisch ausgeführt, wenn der Kunde die versprochene Leistung sehen möchte.

Um die Einhaltung des ursprünglichen Vertrags garantieren zu können, muß dieser Unterauftrag so vergeben werden, daß jeder Subunternehmer mindestens genausoviel liefert, wie der Hauptauftrag verlangt. Dies geht nur, wenn schwächere Vorbedingungen und stärkere Nachbedingungen eingehalten werden, wobei man sich natürlich an dem praktisch Machbaren orientieren muß. Nur durch Einhaltung dieser Regeln kann eine systematische Entwicklung korrekter (objektorientierter) Software gewährleistet werden.

3.8.11 Kaufen oder Erben?

Wenn Sie bei einer Klassendeklaration features einer anderen Klasse K benötigen, dann werden Sie vor der Frage stehen, welche Beziehungsart gewählt werden soll: soll K vererben oder Lieferant sein?

Die meisten Programmierer, die noch nicht mit der objektorientierten Denkweise vertraut sind, werden im allgemeinen die Vererbung vorziehen, da man hier auf die qualifizierten Aufrufe verzichten kann. Dies bedeutet aber einen Mißbrauch der Vererbung, da hierdurch die durch die das Klassenkonzept geförderte Strukturierung von Softwaresystemen in unabhängige Komponenten wieder verlorengeht. Wer Vererbung nur der leichteren Schreibweise wegen bevorzugt, läuft Gefahr, in Eiffel nach wie vor einen Pascal-Stil zu verwenden.

Sinnvoller ist es, sich Gedanken darüber zu machen, welche konzeptionelle Beziehung zwischen den Objekten der realen Welt besteht, die durch die Klassen beschrieben werden. Vererbung drückt aus, daß die Objekte der Erbenklasse auch Objekte der Elternklasse *sind*, also Attribute der Elternklasse besitzen. Die Kunden-Lieferant Beziehung besagt, daß ein Kundenobjekt das Lieferantenobjekt als Bestandteil *hat* (und daß möglicherweise mehrere Kundenobjekte dasselbe Lieferantenobjekt haben).

So kann zum Beispiel die Klasse `BUCH` niemals sinnvoll ein Erbe der Klasse `PERSON` sein, da Bücher nun einmal keine Personen sind. Bei Studenten, Entleihern und Arbeitnehmern sieht das ganz anders aus.

Natürlich gibt es Gründe, von dieser Vorgehensweise abzuweichen. Bei der Implementierung der Klasse `FIXED_LIST` (Abbildung 3.38 auf Seite 111) haben wir die Klasse `ARRAY` geerbt, obwohl Felder eindeutig keine Listen *sind*. Hier stand die Effizienz und Einfachheit der Schreibweise im Vordergrund. Auch mag die Flexibilität der Redefinition ein Kriterium für Vererbung anstelle des Kaufens sein.

Die größere Flexibilität der Vererbung erkauft man sich jedoch durch eine stärkere Bindung an die Elternklasse. Kunden und Lieferanten kommunizieren über sauber definierte Schnittstellen und Verträge. Der Kunde wird von Änderungen der Implementierungen bei den Dienstleistungen nicht beeinflußt. Zwischen Eltern und Nachkommen gibt es keine derartigen Sicherheiten. Eine globale Entscheidung für oder wider die Vererbung gibt es also nicht. Sie sollte im Einzelfall durch eine Bewertung dieser Kriterien getroffen werden.

3.9 Arbeiten mit Eiffel

In den bisherigen Unterkapiteln haben wir alle wesentlichen Konzepte für die Strukturierung von Daten angesprochen und an vielen Beispielen illustriert. Damit haben wir alle Komponenten beisammen, die wir für einen systematischen Entwurf der Architektur von Softwaresystemen benötigen. Sie sind in nun der Lage, Klassen zu entwerfen, die Beziehungen der Klassen untereinander durch von Vererbung und Benutzung festzulegen und die notwendigen Leistungen einer Klasse in der Form von Kontrakten genau zu spezifizieren.

Auf Programmierkonstrukte für eine systematische Implementierung der einzelnen Leistungen einer Klasse (also Schleifen, Prozeduren, Ausdrücke) werden wir erst im nächsten Kapitel eingehen. Sie sind aber bereits prinzipiell in der Lage, die Dienstleistungen bereits existierender Klassen in ihren eigenen zu benutzen und sich somit das große Spektrum vordefinierter Softwarestücke aus der Eiffel-Bibliothek zunutze zu machen.

Bisher haben wir aber erst wenig dazu gesagt, wie man aus dieser losen Ansammlung von Klassen ein ausführbares Eiffel-Softwarepaket erzeugt. Der Grund hierfür ist, daß es einer der Grundgedanken der objektorientierten Programmierung ist, die konkrete Implementierung und die Bestimmung eines "Hauptprogramms" so lange wie möglich zu verschieben und den eigentlichen Entwurf – die Strukturierung von Daten und Dienstleistungen – in den Vordergrund zu stellen. Anders als in der "konventionellen" Programmierung wird ein also Softwaresystem nicht um eine zentrale Hauptfunktion herum entworfen, sondern als Ansammlung unabhängig agierender Einzelbetriebe, die in einer *Bibliothek* (*library*) gesammelt werden. Auf diese Art wird die Entwicklung wiederverwendbarer Softwarebausteine betont, die als Implementierungen von Klassen gebaut werden.

Die eigentliche *Montage* eines Systems, also der Prozeß des Zusammenbindens einer Menge von Klassen zu einem Software*system* zur Lösung einer bestimmten Aufgabe, muß der letzte Schritt im Softwareentwicklungsprozeß bleiben. Systeme (also Hauptprogramme) sind daher kein Sprachkonzept von Eiffel, sondern nur auf der Ebene des Betriebssystems bekannt und mit ausführbaren Prozessen verbunden. Ein System wird nur dann erzeugt, wenn man aus der Ansammlung von Klassen einen einzeln ausführbaren Prozeß benötigt.

Ein konkretes System ist gekennzeichnet durch eine sogenannte *Wurzel* (*root-class*). Das ist diejenige Klasse, welche den gewünschten Prozeß initiieren soll. Dies geschieht durch die Ausführung der Initialisierungsprozedur der Wurzelklasse, die normalerweise direkt oder indirekt Objekte aus ihrer eigenen und aus anderen Klassen erzeugt und weitere Routinen (ggf. mit Ein- und Ausgabe) anstößt, die dann ihrerseits Objekte erzeugen und weitere Routinen aufrufen usw. . In der konventionellen Denkweise entspricht also die Initialisierungsprozedur der Wurzelklasse dem klassischen Hauptprogramm. Im Unterschied zu diesem wird sie jedoch erst zum Zeitpunkt der Montage festgelegt.

Wie alle anderen Prozeduren, darf auch die Initialisierungsprozedur der Wurzelklasse formale Argumente haben. Die zugehörigen aktuellen Werte müssen dann bei der Ausführung des Systems angegeben werden. Da das System als Prozeß des Betriebssystems – also von außerhalb von Eiffel – aufgerufen wird, müssen alle Argumente dieser Initialisierungsprozedur von einem einfachen Datentyp oder vom Typ `STRING` sein.

Um ein System tatsächlich zu montieren und zu aktivieren, muß man zunächst angeben, in welcher Bibliothek das System liegt und welche Klasse als Wurzelklasse gelten soll. Beide Beschreibungen werden in einem *System Descriction File* (kurz *SDF*) angegeben. Die genaue Syntax des SDF hängt ab von dem tatsächlich verwendeten Compiler. Die Variationen sind jedoch gering. Ein typisches System Descriction File beginnt mit

`ROOT:` *Name der root class (Kleinbuchstaben!!)*
`UNIVERSE:` *Name der Bibliothek (directory name)*
⋮ *sonstige Compiler-Optionen*

Das Montagekommando des Betriebssystems, welches das SDF verarbeitet, heißt in Unix üblicherweise **es** (für **E**iffel **S**ystem)[34]. Es erwartet, das SDF unter einem bestimmten Namen (z.B. `.eiffel`) im aktuellen directory vorzufinden. Bei Ausführung sucht es die für die Montage des Systems notwendigen Klassen aus der Bibliothek heraus, übersetzt sie (genauer gesagt nur diejenigen, deren Übersetzung nicht mehr aktuell ist) und bindet sie zusammen. Das fertig montierte System wird dann als ausführbares Unix-Kommando unter dem Namen der Wurzelklasse im aktuellen directory abgelegt. Dieses Kommando kann dann zusammen mit den aktuellen Parametern für die Initialisierungsprozedur aufgerufen und ausgeführt werden.

Beispiel 3.9.1

Für das Bibliotheken-Verwaltungssystem wäre es sinnvoll, Klassen aufzustellen, welche Bücher, Bibliotheken, die verschiedenen Entleiher, die Autoreninformationen, die einzelnden Transaktionen, die Datumsverwaltung, Ein- und Ausgaben (Menüs), Lesen und Schreiben von Dateien, welche den derzeitigen Bibliothekenbestand und andere bekannte Verwaltungsdaten enthalten, usw. verarbeiten.

Um diese zu einem System zusammenzusetzen, könnte man dann eine Wurzelklasse mit dem Namen `BIB_VERWALT` *hinzufügen, deren Initialisierungsprozedur ein Datum in Form von Tag, Monat und Jahr (jeweils vom Typ* `INTEGER`*) erwartet, das Lesen diverser Dateien veranlaßt, und dann in ein Menü überwechselt, über das dann die Transaktionen und ggf. sonstige Veränderungen ausgelöst werden. Wichtig ist dabei, daß in* `BIB_VERWALT` *nur Aktionen anderer Klassen* angestoßen *werden und nicht etwa eigene Routinen geschrieben werden. Für die Montage gibt man im SDF also an*

`ROOT: bib_verwalt`
⋮ *sonstige Compiler-Optionen*

[34]Es mag Gründe geben, das Kommando in Unix anders zu benennen, zum Beispiel, wenn mehrere Versionen des Eiffel Compilers vorhanden sind. Dies ist dann jedoch nur eine Umbenennung im Rahmen des Betriebssystems, die keinen Einfluß auf das sonstige Verhalten des Compilers hat. Für Details dieser Art sollte man unbedingt die lokalen Handbücher konsultieren.

Dies erzeugt also ein Unix-Kommando `bib_verwalt`*, das dann zum Beispiel mit*

```
bib_verwalt 17 12 1993
```

aufgerufen werden kann und den Verwaltungsablauf der Bibliotheken am 17.12.1993 startet.

Die in diesem Beispiel erläuterte Vorgehensweise zeigt, daß die meisten Eiffel Klassen von den Systemen, in denen sie agieren, völlig unabhängig bleiben. Abgesehen von der verhältnismäßig kleinen Wurzelklasse können fast alle anderen Klassen in völlig verschiedenartigen Systemen wiederverwendet werden – vorausgesetzt sie sind modular und mit sauberen Schnittstellen erstellt worden. Dies trifft insbesondere natürlich für Ein- und Ausgaben, Lesen und Schreiben von Dateien, und ähnliche vielseitig verwendbare Klassen zu und führt insgesamt zu einer sehr dezentralen Systemarchitektur. Programmierern, die gewohnt sind, zentralistisch (im Sinne von Hauptprogrammen) zu denken, nötigt dies ein gewisses Umdenken ab[35]. Aus dem Text einer einzelnen Klasse kann man normalerweise *nicht* ableiten, wie sich ein System verhält, das auf dieser Klasse aufbaut. Der Schwerpunkt liegt stattdessen auf den Dienstleistungen, welche die Klasse anbietet. Die *Reihenfolge*, in der diese Dienstleistungen während der Ausführung eines Systems verlangt werden, ist ein zweitrangiges Problem.

Diese Tatsache ist der Kern des objektorientierten Denkens. Selbst dann, wenn man die vorgesehene Ausführungsreihenfolge kennt, sollte man keine ernsthaft Entwurfsentscheidung darauf gründen. Nur so kann die Flexibilität eines Entwurfs erreicht werden. Bei einer dezentralen Struktur ist es leicht, Dienstleistungen zu ändern oder hinzuzunehmen. Hatte sich aber der Entwurf an einer bestimmten Ausführungsreihenfolge orientiert, so erhält man vielleicht schneller ein lauffähiges System, aber jede Änderung der externen Anforderungen wird zu massiven Problemen bei der Anpassung des Systems führen.

3.10 Diskussion

Ziel der objektorienten Programmierung ist die *Modellierung der realen Welt*. Systeme der realen Welt kooperieren miteinander durch Leistungsangebote (exportierte Merkmale). Für den Klienten einer Leistung ist es dabei weitgehend unerheblich, wie die gewünschte Leistung zustande kommt (*Geheimnisprinzip*), außer er bekommt direkt oder indirekt mit Nebenprodukten (Speicherüberlauf, ruinierte Plattenverzeichnisse, Viren usw.) zu tun, von denen er bei der Leistungsbeschreibung (*Kontrakt*) nichts gehört hat.

[Meyer, 1988] zählt fünf Eigenschaften auf, welche jede Entwurfsmethode und die entsprechende Entwurfsbeschrei bzw. Programmiersprache gewährleisten sollen, um eine gute Modellierung zu ermöglichen:

Zerlegbarkeit in Module: Die Beschreibungsprache ermöglicht die Zerlegung in Einheiten geringerer Komplexität, die miteinander kooperieren. Obwohl der gesamte Beschreibungsaufwand anwächst, ist das Gesamtsystem durch die Zerlegung in Komponenten (*Separation of Concerns*) besser verständlich.

Die Verwaltung von Bibliotheken wird zerlegt in `BIB_VERWALT, BIBLIOTHEK, BUCH, AUTOR, ENTLEIHER, TRANSAKTION, PERSON` usw.

Zusammensetzbarkeit aus Modulen: Die Zerlegung geht natürlich nicht unendlich tief, da jedes System aus vorgegebenen Einheiten (Datentypen und Anweisungen) einer Programmiersprache zusammengesetzt werden muß. Wichtig aber ist, daß die Menge der bereits *vorgegebenen* Einheiten (*Bibliothek*) laufend ergänzt werden kann, um bei weiteren Entwicklungen, auf diese zurückgreifen zu können und unnötigen Entwicklungsaufwand zu vermeiden.

`PERSON` und `BUCH` könnte in vielen anderen Systemen eingebaut werden und die allgemeine Einsatzfähigkeit von `TRANSAKTION` und `BIBLIOTHEK` für andere Verwendungszwecke könnte durch Verallgemeinerungen (und Vererbungskonzepte) noch erhöht werden.

[35] Aus diesem Grunde haben wir in dieser Veranstaltung zunächst mit den Strukturierungskonzepten begonnen. Wir hoffen damit der Gefahr zu entgehen, daß Sie zu sehr mit dem üblichen zentralistisches Denken vertraut werden, das wir Ihnen dann mühsam wieder abgewöhnen müssten.

Verständlichkeit der Module: Die Leistungen eines Moduls sollte von außen unabhängig von seinen Verwendungen verständlich sein (*"Grob"-Spezifikation* oder *Kontrakt* des Moduls). Die Beschreibung, *wie* die Leistungen des Moduls zustande kommen, sollen aus Beschreibung der Kooperation (*"Fein"-Spezifikation* oder *Implementierung* des Moduls) einiger weniger Module (*Supplier*) und deren Grobspezifikationen ersichtlich sein.

Nur, wenn ein Modul auch unabhängig von seiner Umgebung verstanden werden kann, sind Wartungen und Erweiterungen eines Systems leicht durchzuführen.

Auf diese Punkte werden wir im nächsten Kapitel besonders eingehen. Die Grobspezifikation von `BUCH` wird einfach sein, da man mit Büchern nur wenig machen kann. Die Implementierung von `TRANSAKTION` und `BIBLIOTHEK` sollte so einfach werden, daß sie auf der Basis der Grobspezifikationen der Klassen `ARRAY, LINKED_LIST` und `BUCH` voll verständlich ist.

Stetigkeit der Module: Im Gegensatz zu "chaotischen" Systemen haben stetige Systeme die Eigenschaft, daß kleine Änderungen der Ursachen auch nur kleine Änderungen der Wirkung nach sich ziehen. Von einer Beschreibungsmethode sollte man verlangen, daß sie die Entwicklung stetiger Systeme unterstützt. Speziell sollen lokale Änderungen in einem Modul nicht die Architektur des Systems ruinieren. Anders ausgedrückt, wird ein Fehler in einem Modul repariert, so sollen die Korrekturen auf diesen Modul beschränkt bleiben.

Das bedeutet zum Beispiel, daß wir innerhalb der Klassen `PERSON` und `BUCH` tun können, was wir wollen, solange wir die Schnittstelle nicht ändern (eine Erweiterung der Schnittstelle unter Beibehaltung aller alten Verträge ist erlaubt).

Modularer Schutz: Eine Beschreibungsmethode muß es ermöglichen, daß Ausnahmesituationen (Laufzeitfehler, fehlerhafte Eingaben, Speichermangel etc.), die bei der Durchführung eines Moduls auftreten, nur lokale Effekte haben und sich nicht etwa auf andere Module fortpflanzen.

Im Vergleich zu allen anderen Sprachen, die derzeit im industriellen Einsatz sind, ist Eiffel derzeit die einzige Sprache, welche diese Prinzipien massiv unterstützt. Sie gibt also einen geeigneten Rahmen um eine adäquate Modellierung der Realität zu erreichen. Damit ist der sinnvolle Einsatz jedoch noch lange nicht gewährleistet. Viele Eiffel-Programme werden noch im Stil herkömmlicher Programmiersprachen geschrieben und verstoßen gegen die Richtlinien eines objektorientierten Entwurfs.

Welche Prinzipien sind nun zu beachten, wenn ein System methodisch korrekt entworfen werden soll? Auch hier schlägt [Meyer, 1988] fünf Kriterien vor, die eine geeignete Modularisierung gewährleisten sollen:

Moduln müssen Klassen entsprechen: Die Klassenstruktur der Zerlegungseinheiten stellt sicher, daß das System in klar definierte und voneinander abgegrenzte Einheiten *zerlegt* ist, die für sich *verständlich* sind, später in anderen Systemen *kombiniert* werden können und *Schutz* gegen Fehlfunktion gewähren.

Minimale Kommunikation zwischen Moduln: Zwischen Modulen kann auf vielfache Weise kommuniziert werden. Module können einander aufrufen (durch Typvereinbarung von Variablen), gemeinsame Datenstrukturen benutzen usw. Prinzipiell könnte jedes Modul jedes andere verwenden. *Stetigkeit* aber erreicht man nur bei geringer gegenseitiger Verwendung.

Schmale Schnittstellen: Wenn zwei Module kooperieren, so soll sich der Informationsaustausch (Parameter) auf die absolut *notwendige* Information (need to know) beschränken. Dies unterstützt *Stetigkeit* und *modularen Schutz*.

Vollständige Beschreibung der Schnittstellen: Wenn zwei Moduln kooperieren, so muß diese Kooperation in diesen beiden *vollständig* beschrieben sein. Dies fördert *Zerlegbarkeit* und *Kombinierbarkeit*, *Stetigkeit* und *Verständlichkeit*, da keine unvorhersehbaren Effekte auftreten können.

Wenn jeder Modul als Klasse definiert ist, so ist dies automatisch gewährleistet. Ist die Modularisierung über Prozeduren definiert, wie in Pascal, dann könnten die beiden über globale Variable eine Verbindung haben, die aber nicht explizit erkennbar ist. Aber nicht nur die Tatsache der Kooperation muß beschrieben, sondern auch die Wirkung der Merkmale.

Geheimnisprinzip: Die Schnittstelle ist die *einzige Information* über den Modul. Insbesondere, wie die Kontrakte realisiert sind, bleibt den Klienten verborgen. Ist die Schnittstelle minimal, so ist für die Implementierung und daher auch für Alternativimplementierungen der maximale Freiheitsgrad vorhanden. Insbesondere Änderungen, seien sie durch Fehler erzwungen oder durch Verbesserungen der Effizienz zweckmäßig, erfordern keine Änderungen im restlichen System.

In Eiffel gibt es keine globalen Variablen zwischen Klassen, daher ist das Geheimnisprinzip automatisch bei minimalen Schnittstellen gewährleistet. Bei der Modularisierung über Prozeduren mit globalen Variablen (wie C und Pascal üblich) wird das Geheimnisprinzip zwangsläufig verletzt mit der Konsequenz, daß die Wartungskosten einen hohen Anteil an den Entwicklungskosten ausmachen.

Die Klärung der Frage, wie nun ein Softwaresystem unter Beachtung dieser Kriterien entworfen und systematisch implementiert werden kann, wird das Thema des nächsten Kapitels werden.

3.11 Sprachbeschreibung

Wir wollen nun versuchen, eine präzise Sprachbeschreibung für die bisher eingeführten Konstrukte zu geben. Sie ist hilfreich zum Verständnis einer Programmiersprache, aber keineswegs notwendig. Sie dient lediglich dazu, offengebliebene Fragen der informalen Beschreibung zu beantworten.

Die Struktur dieser Sprachbeschreibung entspricht der in Kapitel 2 gegebenen Methodik bei der Einführung in die Logik: zuerst wird die *Syntax* beschrieben, anschließend die Kontextbedingungen – in der Denkweise von Programmierprachen nennt man dies *statische Semantik* – und schließlich die dynamische Semantik.

Syntax: Für die Definition der Syntax verwenden wir die bereits bekannte *Backus-Naur-Form*. Sie hat den Zweck, die *Struktur* von Programmen in eindeutiger Weise als Basis für die Beschreibung der Semantik der Programmiersprache festzulegen.

Die Syntax ist die Basis jeden Verfahrens zur Strukturanalyse (*Syntaxanalyse*) von Programmen, also der notwendigen ersten Phase einer Übersetzung von Eiffel-Programmen in die Maschinensprache.

statische Semantik: Ergänzend zur Struktur der Sprache beschreibt die statische Semantik zusätzlichen-Bedingungen für eine einheitliche Verwendung von Namen in einem sinnvollen Kontext. Hierzu gehören *Sichtbarkeitsregeln* und Verträglichkeitsbedingungen.

Sichtbarkeitsregeln definieren eine Zuordnung von Bezeichnern (Variablen) zu ihren Typen (also `REAL`, `INTEGER`, `PERSON` etc.), sowie den Geltungsbereich dieser Zuordnung im Programmtext. In der Prädikatenlogik war dies zum Beispiel der Bereich, in dem eine allquantifizierte Variable gebunden ist.

Typverträglichkeitsbedingungen beschreiben die zulässigen Kombinationen von Operatoren und Typen der Operanden. So verlangt zum Beipiel die Addition genau zwei Argumente und kann nicht auf Elemente vom Typ `CHARACTER` angewandt werden. Dies verbietet zuweilen auch Kurzschreibweisen wie `2<4<6`, da die Operation "`<`" arithmetische Werte als Argumente verlangt, während `2<4` ein Boolescher Wert ist, der nicht mit `6` verglichen werden kann.

Die statische Semantik ist die Basis der *semantischen Analyse*, ein Verfahren, welches *vor* dem Ablauf eines Programms überprüft, ob während der Durchführung eine Operation mit ungeeigneten Operanden aufgerufen werden kann. Diese Art der Analyse – die zweite Phase einer Übersetzung – macht Überprüfungen *während* der Laufzeit überflüssig und steigert somit die Effizienz des übersetzten Programms.

dynamische Semantik: Die dynamische Semantik beschreibt das Verhalten eines Programms *während* des Ablaufs. Im Rahmen dieser Vorlesung werden wir diese Semantik durch einen *Interpretierer* von Eiffel angeben, den wir wie die Prädikatenlogik durch mathematische Funktionen beschrieben. Hierzu verwenden wir die in Abschnitt 2.3 vorgestellte funktionale Metasprache[36].

Die dynamische Semantik entspricht der *Codegenerierung* beim Übersetzen.

Die Sprachbeschreibung beschränkt sich in diesem Kapitel bis auf wenige Ausnahmen auf die Syntax, denn Strukturierungskonzepte sind im wesentlichen Organisationsprinzipien.

3.11.1 Syntax

In der Beschreibung von Programmiersprachen gibt es viele Programmbestandteile, die an einer bestimmten Stelle im Programm vorkommen *können* – wie zum Beispiel eine inherit-Klausel, eine creation-Klausel, eine invariant-Klausel, usw. – aber nicht unbedingt müssen. Es wäre sehr unökonomisch und unübersichtlich, alle möglichen Kombinationen separat aufzulisten. Aus diesem Grunde erweitern wir die Notation der in Kapitel 2 (Seite 28) vorgestellten Backus-Naur-Form um die Möglichkeit, *Optionen* mit anzugeben. Die Notation

Konstrukt ::= *Teilkonstrukt* [*Option*]

steht kurz für

Konstrukt ::= *Teilkonstrukt* | *Teilkonstrukt Option*

Das bedeutet, daß die in eckigen Klammern angegebene Option an der entsprechenden Stelle vorkommen *kann*, aber nicht vorkommen muß. Die Verwendung von Optionen vereinfacht die Syntaxbeschreibung ungemein.

Im folgenden werden wir die Syntax der bisher vorgestellten Programmteile zusammenfassen[37]. Terminalsymbole (wie zum Beispiel `[ , ; ]`) werden im `typewriter`-Zeichensatz geschrieben, um sie von den Hilfssymbolen abzugrenzen. Eiffel-Schlüsselworte wie **class** werden, obwohl sie auch Worte des Terminalalphabets sind, weiterhin im Fettdruck angegeben. Non-Terminale werden als Worte in normalem Zeichensatz beschrieben. Startsymbol ist System. Fettgedruckte Konzepte bezeichnen zentrale Begriffe. Unterstrichene Konzepte werden an mehreren Stellen gebraucht, nicht unterstrichene nur in einem Block.

System	::=	Class_declaration [System]	*Ein System besteht aus Klassen*
Class_declaration	::=	[**indexing** Index_list]	
		[**deferred** | **expanded**] **class** Class_name	*Klassennamenvereinbarung*
		[`[` [Formal_generics] `]`]	*Generische Parameter*
		[**obsolete** Manifest_string]	
		[**inherit** [Parents]]	*Erbklausel*
		[**creation** Creation_clauses]	*Initialisierungsprozeduren*
		[**feature** Feature_clauses]	*Vereinbarung der Features*
		[**invariant** [Assertions]]	*Zustandkonsistenzbedingung*
		end [`--` **class** Class_name]	
Index_List	::=	*nicht besprochen – siehe [Meyer, 1992, Seite 49ff]*	
Class_name	::=	Identifier	
Identifier	::=	*Siehe Abschnitt 4.4*	
Formal_generics	::=	Formal_generic [`,` Formal_generics]	
Formal_generic	::=	Identifier [`->` Class_type]	

[36] Warnung: Die beiden Sprachen verwenden ein ähnliches Vokabular, sind aber durch die Art des Drucks stets unterscheidbar.

[37] Die Zusammenstellung entstammt [Meyer, 1992, Anhang H/I], wurde jedoch zugunsten der besseren Lesbarkeit um einige Zwischenbegriffe gekürzt und an die Syntaxdiagramme im Anhang J angepaßt. Meiner Ansicht nach enthalten die Anhänge H und I kleinere Fehler und Unstimmigkeiten zu den Syntaxdiagrammen, die hier – soweit erkennbar – korrigiert wurden.

Manifest_string ::= "Simple_string"
Simple_string ::= *Siehe Abschnitt 4.4*

Parents ::= Parent [; Parents]
Parent ::= Class_type [Feature_adaption]
Feature_adaption ::= [**rename** [Rename_pairs]] *Umbenennung geerbter features*
[**export** [New_exports]] *Export geerbter features*
[**undefine** [Features]] *Aufschieben einer geerbten Implementierung*
[**redefine** [Features]] *Redefinition einer geerbten Implementierung*
[**select** [Features]] *Auswahl eines doppelt geerbten features*
end
Rename_pairs ::= Feature_name **as** Feature_name [; Rename_pairs]
New_exports ::= [Clients] Feature_set [; New_exports]
Feature_set ::= [Features] | **all**
Features ::= Feature_name [; Features]

Feature_name ::= Identifier | Prefix | Infix
Prefix ::= **prefix** "Prefix_Operator"
Infix ::= **infix** "Infix_Operator"
Prefix_Operator ::= *Siehe Abschnitt 4.4*
Infix_Operator ::= *Siehe Abschnitt 4.4*

Clients ::= { [Classes] }
Classes ::= Class_name [, Classes]

Creation_clauses ::= Creation_clause [**creation** Creation_clauses]
Creation_clause ::= [Clients] [Comment] [Features]

Comment ::= -- [Simple_string Comment_break Comment]
Comment_break ::= New_line [Blank_or_tabs] --

Feature_clauses ::= Feature_clause [**feature** Feature_clauses]
Feature_clause ::= [Clients] [Comment] [Feature_declarations]
Feature_declarations ::= Feature_declaration [; Feature_declaration_list]
Feature_declaration ::= Feature_names
[([Entity_declarations])]
[: Type]
[**is** Feature_value] *Vereinbarung der neuen Namen*
Feature_names ::= [**frozen**] Feature_name [, Feature_names]
Feature_value ::= Routine *Definition von Prozedur oder Funktion*
| Manifest_constant *Konstantendefinition*
| **unique**

Entity_declarations ::= Identifiers : Type [; Entity_declarations]
Identifiers ::= Identifier [, Identifier_list]

Type ::= Class_Type
| **expanded** Class_type
| Identifier *formaler generischer Parameter*
| **like** Anchor *Assoziation*
| BIT Constant
| INTEGER | REAL | DOUBLE | CHARACTER | BOOLEAN
Class_type ::= Class_name [[[Types]]]
Types ::= Type [, Types]
Anchor ::= Identifier | Current

Constant	::=	Manifest_constant | Identifier	
Manifest_constant	::=	*Siehe Abschnitt 4.4*	
Routine	::=	[**obsolete** Manifest_string]	
		[Comment]	
		[**require** [**else**] [Assertions]]	*Vorbedingungen*
		[**local** [Entity_declarations]]	*Lokale Variablen*
		Routine_body	*Anweisungsteil*
		[**ensure** [**then**] [Assertions]]	*Nachbedingungen*
		[**rescue** Compound] *Ausnahmebehandlung* Siehe Abschnitt 4.3	
		end [-- Feature_name]	
Routine_body	::=	**deferred**	
	|	**do** Compound	
	|	**once** Compound	
	|	External	
External	::=	*nicht besprochen – siehe [Meyer, 1992, Seite 402ff]*	
Compound	::=	*Siehe Abschnitt 4.3*	
Expression	::=	*Siehe Abschnitt 4.4*	
Assertions	::=	Assertion_clause [; Assertions]	
Assertion_clause	::=	[Identifier :] Unlabeled_assertion_clause	
Unlabeled_assertion_clause	::=	Expression | Comment	

3.11.2 Statische und Dynamische Semantik

Eine detaillierte Besprechung der statischen und dynamischen Semantik von Klassendeklarationen einschließlich Vererbung, Benutzung und Generizität ist verhältnismäßig zeitaufwendig. In diesem Semester werden wir dieses Thema zugunsten einer ausführlicheren Diskussion von systematischer Implementierung und Verifikation von Software bis auf weiteres verschieben und ggf. auslassen.

An dieser Stelle wollen wir nur anmerken, daß Groß- und Kleinschreibung in Eiffel zwar erlaubt ist, aber nicht unterschieden wird. `INTEGER` bedeutet dasselbe wie `Integer` oder `integer`. Es hat sich als Konvention eingebürgert, Klassennamen groß zu schreiben und features klein. Natürlich dürfen Schlüsselworte wie **class**, `Current`, `Result` nicht mehr als Namen für selbstdefinierte Klassen oder features verwendet werden. Eine ausführlichere Diskussion stilistischer Standards und lexikalischer Konventionen findet man in [Meyer, 1988, Kapitel 8.1-8.2]

3.12 Ergänzende Literatur

Viele Themen dieses Kapitels werden in [Meyer, 1988, Kapitel 5, 6, 7.1-7.6, 10 und 11] in ausführlicherer Form behandelt. Man beachte jedoch, daß sich beim Übergang von der ursprünglichen Form von Eiffel zu Eiffel-3 eine Reihe von Änderungen ergeben haben. Die wesentlichen Unterschiede sind in [Meyer, 1992, Anhang E] aufgeführt und müssen dort im Detail unter dem entsprechenden Stichwort nachgeschlagen werden.

3.13 Deutsch-Englisches Begriffswörterbuch

Die meisten Begriffe der Programmierung sind unabhängig voneinander sowohl im deutschsprachigen als auch im englischsprachigem Raum geprägt worden und haben sich entsprechend in der Literatur niedergeschlagen.

Dabei deckt sich der deutsche Begriff nicht unbedingt mit der wörtlichen Übersetzung des Englischen. So entspricht z.B. dem englischen “Stack” (Stapel) der deutsche “Kellerspeicher” (kurz “Keller”). Da die Zuordnung nicht immer leicht ist, geben wir hier eine Zusammenstellung der wichtigsten Begriffe.

Deutsch	Englisch
Anker	anchor
Attribut	attribute
Aufgeschoben	deferred
Baum	tree
Beziehung	relationship
Bibliothek	library
Erbe	descendant
Erbe	heir
Exemplar	instance
Feld	array
Flach	shallow
Geheimnisprinzip	information hiding
Gemeinsam genutzt	shared
Größe	entity
Kellerspeicher	stack
Komponente	field
Konformität	conformance
Kunde	client
Lieferant	supplier
Merkmal	feature
Nachbedingung	postcondition
Nachkomme	descendant
Ordner	directory
Passen zu	conform to
Stetigkeit	continuity
Umbenennung	redefinition
Verbund	record
Vererbung	inheritance
Vervielfältigt	replicated
Verweis	pointer
Verweis	reference
Vorbedingung	precondition
Vorbedingung	requirement
Vorfahr	ancestor
Warteschlange	queue
Wiederholtes Erben	repeated inheritance
Wurzelklasse	root class
Zusicherung	assertions

Englisch	Deutsch
ancestor	Vorfahr
anchor	Anker
array	Feld
assertions	Zusicherung
attribute	Attribut
client	Kunde
conform to	Passen zu
conformance	Konformität
continuity	Stetigkeit
deferred	Aufgeschoben
descendant	Erbe
descendant	Nachkomme
directory	Ordner
entity	Größe
feature	Merkmal
field	Komponente
heir	Erbe
information hiding	Geheimnisprinzip
inheritance	Vererbung
instance	Exemplar
library	Bibliothek
pointer	Verweis
postcondition	Nachbedingung
precondition	Vorbedingung
queue	Warteschlange
record	Verbund
redefinition	Umbenennung
reference	Verweis
relationship	Beziehung
repeated inheritance	Wiederholtes Erben
replicated	Vervielfältigt
requirement	Vorbedingung
root class	Wurzelklasse
shallow	Flach
shared	Gemeinsam genutzt
stack	Kellerspeicher
supplier	Lieferant
tree	Baum

Kapitel 4

Systematische Entwicklung zuverlässiger Software

In der bisherigen Auseinandersetzung mit dem Thema "Programmierung" haben wir uns vor allem auf die spezifischen Denkweisen der objektorientierten Programmierung und die zugehörigen Konstrukte der Programmiersprache Eiffel konzentriert. Sie sollten nun verstanden haben, welche Mittel Ihnen zur Strukturierung von Softwaresystemen zur Verfügung stehen – insbesondere, wie man Daten und Dienstleistungen in Klassen zusammenfaßt, fertige Programmteile aktiviert, Wiederverwendbarkeit durch den Einsatz generischer Klassen steigert, ihre Eigenschaften in der Form von Verträgen (Zusicherungen) beschreibt, und die logischen Beziehungen zwischen Klassen durch Vererbung ausdrückt.

Wir haben bisher aber nur wenig dazu gesagt, *wie* man diese Strukturierungskonzepte einsetzen kann, um gute Softwaresysteme systematisch zu entwerfen, und mit welchen Mitteln man vertraglich vereinbarte Dienstleistungen in einer Programmiersprache implementiert und dabei sicherstellt, daß diese Implementierung tatsächlich auch die versprochenen Eigenschaften besitzt.

Diese Methoden der systematischen Entwicklung zuverlässiger Software sollen nun in diesem Kapitel besprochen werden. Wir werden zunächst kurz einige Methoden des objektorientierten Entwurfs – also Methoden der Strukturierung von Softwaresystem in Klassen, Features, Zusicherungen und Vererbungsbeziehungen – ansprechen und an unserem Leitbeispiel illustrieren. Anschließend werden wir unser Augenmerk auf die eher "konventionellen Programmierkonzepte" von Eiffel richten, die es uns erlauben, in Eiffel so zu rechnen wie in jeder anderen Programmiersprache auch. Dabei wird besonders das Qualitätskriterium *Zuverlässigkeit* (partielle und totale Korrektheit und Robustheit) im Vordergrund stehen. Aus diesem Grunde werden wir zunächst über Verifikation (Korrektheitheits*beweise*) reden und die hierzu existierenden Formalismen vorstellen, bevor wir konkret auf Programmstrukturen wie Anweisungen, Fallunterscheidungen und Schleifen zu sprechen kommen, mit denen man den Anweisungsteil von Routinen verfeinern kann. Auf die Grundbausteine jeglicher Berechnung – die elementaren Ausdrücke der Sprache Eiffel – werden wir nur kurz eingehen, da sie konzeptionell von geringer Bedeutung sind: es muß nur geklärt werden, welche Ausdrücke und Funktionen in Eiffel vordefiniert sind und was ihre Syntax ist. Mit der Besprechung von Methoden der systematischen Implementierung korrekter Software werden wir dieses Kapitel (und das erste Semester) abschließen.

4.1 Systematischer Entwurf von Softwaresystemen

Bei der Besprechung der Strukturierungskonzepte im vorhergehenden Kapitel und insbesondere durch die Diskussion der Modularisierungskriterien im Abschnitt 3.10 haben wir bereits einige Techniken angedeutet, wie man Konzepte objektorientierter Programmiersprachen einsetzen kann, um Softwaresysteme zu strukturieren *bevor* man sich an die eigentliche Implementierungsarbeit begibt. Wir wollen diese nun zusammenfassen und etwas ausführlicher erläutern.

4.1.1 Analyse und Gestaltung

Softwaresysteme werden nur selten losgelöst von einem konkreten Bezug zur realen Welt erstellt. Deshalb besteht die wesentliche Aufgabe beim Entwurf von Softwaresystemen fast immer darin, den für die Problemstellung relevanten Ausschnitt der realen Welt zu *modellieren*. Softwareentwickler stehen also vor der Aufgabe, einen Bezug zwischen der realen Welt und der Denkwelt von Computeralgorithmen herstellen zu müssen. Das verlangt zum einen, den vorgesehenen Einsatzbereich des Softwareproduktes und das zugehörige Umfeld sowie eventuelle Nebenwirkungen des Softwareeinsatzes zu verstehen, und zum anderen, einen derartigen Teil der realen Welt auf *formale Modelle* zu reduzieren, was zwangsläufig eine drastische Beschränkung der Sicht der realen Welt mit sich bringen wird.

Informatiker befinden sich daher immer in einer Doppelrolle. Einerseits sind sie *Entdecker* der Gesetzmäßigkeiten des Umfelds, das sie modellieren und durch ihr Softwareprodukt beeinflussen wollen. Anderseits sind sie aber auch *Gestalter* (oder *Erfinder*) eines Modells, das den relevanten Ausschnitt der Welt wiederspiegelt und durch das entstehende Produkt wieder eine Rückwirkung auf die reale Welt – zum Beispiel auf Arbeitsabläufe innerhalb eines Betriebs – haben wird. Sie als zukünftige Informatiker müssen sich dieser Doppelrolle immer bewußt sein. Ihre Aufgaben und Ihre Verantwortung geht weit über die reine Technik (das Codieren eines Programms) hinaus: Sie müssen die bestehende Welt und natürlich auch die Problemstellung *analysieren* und sich dafür entscheiden, wie Sie sie durch Ihr Softwareprodukt *gestalten* wollen. Erst danach können Sie an eine konkrete Implementierung denken.

Dementsprechend ist der Entwurf eines Softwaresystems in zwei prinzipiellen Schritten auszuführen:

1. Am Anfang eines Entwurfs steht die *Analyse der Leistungen*, die das Programm an seine Klienten erbringen soll. Ergebnis der Analyse ist eine Reihe von Attributen, Methoden und Funktionen, die der Anwender abrufen kann. Jede einzelne Leistung entspricht einer Routine, deren Wirkung über ihren Kontrakt beschrieben ist. Das Gesamtangebot an Leistungen des Systems wird üblicherweise als *Schnittstelle* bezeichnet.[1]Für jede einzelne Leistung wird ihr Kontrakt bestimmt und für das Gesamtprogramm die Zusicherungen, die alle Prozeduren als Nachbedingung garantieren – also die Invariante.

   Damit ist die äußere Sicht des Programms (*Grobspezifikation*) abgeschlossen und der inhaltliche Teil des Pflichtenhefts definiert: Es ist vollständig festgelegt, was das Programm leisten soll.

   Diese Aufgabe muß meist in Kooperation mit Spezialisten aus den Anwendungsbereichen geschehen, da den Informatikern meist die notwendige Kompetenz fehlt. Das befreit Informatiker allerdings nicht von der Pflicht, sich in die Denkweise der entsprechenden Dispziplinen einzuarbeiten.

2. Der nächste Schritt ist der kreative Anteil am Entwurf. Man überlegt sich, wie die einzelnen Angebote realisiert werden können. Die Beschreibung davon nennt man *Feinspezifikation*.

   Das Grundprinzip ist dabei eine *Zerlegung des Systems* in unabhängige kooperierende Komponenten (*Separation of Concerns*), also die Frage, welche Objekte benötigt werden, um die geforderten Leistungen mit geringem Aufwand zu beschreiben. Hierzu muß man versuchen, Objekte zu spezifizieren, aus deren Merkmalen man die geforderten Leistungen zusammenbauen kann.

Zur Durchführung dieser Schritte ist eine Menge Einfallsreichtum und Begabung und vor allem auch eine Menge von Erfahrung und Training erforderlich. Dennoch lassen sich – zumindest für den zweiten Schritt – ein paar allgemeine Leitlinen angeben, die ein erfolgreiches Vorgehen fördern. Hierzu wollen wir zunächst noch einmal die Grundmerkmale der objektorientierten Vorgehensweise resümieren.

[1]Ist der Klient ein Anwender (im Gegensatz zu Systemen, bei dem der Klient wiederum ein technisches Systems ist, wie z.B. bei ABS-Systemen, Waschmaschinen, autonomen Waffensystemen), dann wird er nicht direkt diese Routinen aufrufen sondern wird ein Menü sehen, das die einzelnen Routinen kennzeichnet. Die aktuellen Parameter werden dann innerhalb eines Dialogs abgefragt. In diesem Fall nennt man die Schnittstelle *Benutzerschnittstelle* (*User Interface*) und seine Darstellung *Benutzeroberfläche*.

4.1.2 Grundideen des objektorientierten Entwurfs

Bei einem reinen objektorientierten Vorgehen, wie es durch die Programmiersprache Eiffel unterstützt wird, ist zu beachten, daß Klassen der einzige Strukturierungsmechanismus für Systeme sind. Sie sind unabhängige Einheiten, obwohl sie durch Benutzung und Vererbung miteinander verbunden sein können. Aus Sicht der Programmiersprache liegen alle Klassen auf der gleichen Ebene und dürfen – genau wie Routinen – *nicht geschachtelt* werden. Diese *Eigenständigkeit von Softwareteilen* ist wesentlich für Wiederverwendbarkeit und einer der wichtigsten Unterschiede zu den gängigen blockstrukturierten Sprachen.

Eine Klasse kann als ein Lager von Dienstleistungen (exportierten features) angesehen werden, die auf den Exemplaren eines Datentyps verfügbar sind. Diese Dienste werden allen Kunden, die sie überhaupt benutzen dürfen, *gleichermaßen* zur Verfügung gestellt:

- Es gibt keine Operationen mit höherer oder niedriger Bedeutung, insbesondere keine Hauptfunktion – selbst dann, wenn bestimmte Dienste häufiger in Anspruch genommen werden als andere.
- Es gibt keine Vorschriften über die Reihenfolge, in der Dienstleistungen aufgerufen werden dürfen. Die Reihenfolge mag zwar bei der eigentlichen Ausführung eines Systems von großer Bedeutung sein. Es wäre jedoch ein Fehler, diese Reihenfolge schon bei der Deklaration der entsprechenden Dienstleistungen einzufrieren. Die Festlegung einer solchen Reihenfolge ist ausschließlich Sache der Kunden.

  Natürlich gibt es Gründe festzulegen, daß eine Operation wie das Herausnehmen von Elementen aus einer Liste nur ausgeführt werden kann, wenn zuvor eine andere Operation – das Einfügen von mindestens einem Element – stattgefunden hat. Dies ist aber eine Vorbedingung für die Ausführbarkeit der Operation und sollte auch als solche formuliert werden. Es liegt dann in der Verantwortung des Kunden, was er damit macht.

Eine Folge dieses Prinzips ist, daß es *keine Gründe gibt, die Anzahl der features einer Klasse zu beschränken.* Hat eine Operation einen logischen Bezug zu dieser Klasse, so kann man sie hinzufügen, auch wenn unklar ist, ob sie von vielen Kunden benötigt wird.[2] Das Problem, daß eine Klasse eine unüberschaubare Größe bekommen kann, ist verhältnismäßig gering, da eine zu große Anzahl von Dienstleistungen (ein zu breites Spektrum im Lager) selten stört, da man sich bei der Betrachtung von Informationen auf die features beschränken kann, die man tatsächlich braucht. Einzig die Suche nach noch unbekannten, aber eventuell brauchbaren features wird durch eine zu große Anzahl von features etwas erschwert.

Anders ist es beim *Code einer einzelnen Routine*. Dieser sollte *kurz* und überschaubar bleiben, was aber bei der objektorientierten Sichtweise von Routinen als wohldefinierte Einzeldienstleistung selten ein Problem werden wird.

Aufgrund der Hervorhebung des Wiederverwendbarkeitsaspektes eignen Klassen sich besonders für einen Entwurfsstil, der sich stark auf bereits existierende Module stützt ("*Bottom-up* Entwurf").[3] Die Stärke der objektorientierten Programmierung liegt in der Aktivierung fertiger Programmteile. Deshalb lohnt es, sich mit vordefinierten Klassen der Eiffel-Basisbibliothek und gegebenenfalls auch anderer Bibliotheken vertraut zu machen und neu zu entwickelnden Klassen *möglichst allgemein* zu gestalten, damit sie später auch in anderen Lösungen verwendet werden können.

[2] Das Problem, daß hierdurch beim Zusammensetzen eines Systems viel unbenutzter Code erzeugt wird, kann durch den Eiffel-Optimierer gelöst werden.

[3] Natürlich kann man den Entwurf nicht losgelöst von der eigentlichen Problemstellung durchführen. Ein reiner "*Top-Down* Entwurf", wie er früher favorisiert wurde, konzentriert sich aber *zu sehr* auf das Problem und man muß wichtige Entscheidungen über die Softwarestruktur in einer Phase fällen, in der das Bild noch sehr vage ist und Wesentliches von Unwesentlichem noch nicht getrennt werden kann. Dadurch – und vor allem durch Vernachlässigung der Tatsache, daß es bereits Lösungen für ähnlich gelagerte Problemstellungen geben kann – wird der Entwickler von der Komplexität des Problems geradezu erschlagen und Fehlentscheidungen werden gravierende Folgen auf das entstehende Produkt haben. Top-Down Entwicklung von Programmen bietet sich erst während der Implementierungsphase an, nachdem die Struktur des Gesamtsystems bereits so weit untergliedert ist, daß jedes Teilproblem überschaubar ist.

Um die hier genannten Vorteile ausnutzen zu können, ist es wichtig, sich die Denkweise des objektorientierten Entwurfs anzueignen und das zentralistische Denken, in dem das "Hauptprogramm" eine so wichtige Rolle spielt, beiseite zu legen, wenn man es sich schon angewöhnt hat. Bei größeren Softwaresystemen ist eine dezentrale Architektur mit überschaubaren, unabhängigen und gleichberechtigten Einheiten unverzichtbar.

4.1.3 Aufspüren der Klassen

Die Frage, die sich hierbei natürlich direkt ergibt, ist "wie finde ich die Klassen, die ich brauche?". Hierauf gibt es bisher keine allgemeingültige Antwort (und wahrscheinlich wird es auch nie eine solche geben), aber zumindest ein paar allgemeine Einsichten.

Da Lösungskritik leichter ist als Lösungen zu schaffen, sollte man zunächst *lernen, vorhandene Entwürfe zu analysieren* und ihre Stärken und Schwächen zu beurteilen, um daraus Erfahrungen für eigene Entwürfe zu gewinnen. Hierzu werden besonders die Kriterien aus Abschnitt 3.10 hilfreich sein. Es lohnt sich auch, in Teams zunächst unabhängig mehrere Lösungsansätze aufzustellen und dann zu beurteilen. Zum einen findet man hierdurch unter Umständen eine Lösung, die besser ist als jede einzelne, und zum zweiten weist die Kritik oft auf Denkfehler hin, die man beim nächsten Entwurf vermeiden kann. Aus Fehlern lernt man eine ganze Menge, aber nur wenn man zuläßt, daß diese auch aufgedeckt werden. Wer aber (sachliche) Kritik vermeidet, der wird auch selten etwas lernen.

Für das Aufspüren von Klassen, die benötigt werden, um geforderte Leistungen mit geringerem Aufwand zu beschreiben, bietet es sich an, zunächst einmal die Eiffel-*Bibliotheken nach Lösungen für ähnliche Problemstellungen zu durchforsten*. Für wichtige Datenstrukturen wie Listen, Bäume, Felder, Stacks, Queues, Dateien, Strings, Hash-Tabellen, Bäume, usw. gibt es für nahezu alle Routineaufgaben (wie Einfügen, Löschen, sortieren etc.) längst eine vordefinierte Lösung, auf die man zurückgreifen kann. Da mindestens die Hälfte aller Probleme aus derartigen Routineaufgaben besteht, erspart dieser naive, aber doch sehr wirksame Weg eine Menge Arbeit und reduziert das Risiko von Fehlern.

Ein weiterer wichtiger Hinweis ist, daß viele Klassen keine andere Aufgabe haben werden, als das Verhalten externer Objekte zu modellieren. Ihre features und Eigenschaften ergeben sich also nahezu *direkt aus der Grobspezifikation*.

Ob ein Begriff aus der realen Welt als Klasse realisiert werden soll, hängt vor allem davon ab, wie damit umgegangen werden soll. Interessiert man sich zum Beispiel in einem Bibliothekssystem nur für Namen und Vornamen des Autors eines Buches als Ordnungskriterium, führt aber keine Operationen auf Autoren durch, so sollten Autoren *nicht* als eigene Klasse deklariert werden. Stattdessen sollte die Klasse `BUCH` zwei Komponenten für Namen und Vornamen beinhalten. Anders sieht dies aus, wenn man z.B. in einem angeschlossenen Informationssystem über Buchautoren immer die aktuelle Adresse mitverwalten und ggf. ändern möchte. Hier lohnt es sich schon, Autoren in einer separaten Klasse zusammenzufassen.

Im allgemeinen kann man sagen, daß oft zu viele unnötige Klassen gebildet werden. Ein Begriff sollte genau dann zur Klasse erhoben werden, wenn er eine Menge von Objekten beschreibt, die interessante Operationen mit axiomatisch charakterisierbaren Eigenschaften besitzen.

4.1.4 Schnittstellentechniken

Ebenso wichtig wie die Strukturierung eines Systems in Klassen ist der Entwurf guter Schnittstellen, also die Fixierung derjenigen features, die nach außen hin angeboten werden. Die Qualität und Verständlichkeit dieser Schnittstellen bestimmt, in welchem Maße eine Klasse auch tatsächlich benutzt werden wird.

Eine der wesentlichen Richtlinien ist hierbei eine *strikte Trennung von Prozeduren und Funktionen*. Die Programmiersprache Eiffel läßt prinzipiell zu, daß Funktionen nicht nur Werte berechnen, sondern gleichzeitig auch "Seiteneffekte" haben, d.h. während ihrer Abarbeitung auch das aktuelle Exemplar verändern. Es ist

jedoch dringend zu empfehlen, jegliche Seiteneffekte in Funktionen zu vermeiden, da dies schnell zu sehr unübersichtlichen Programmen führen kann und besonders bei großen Softwaresystemen schnell zu unerwarteten Auswirkungen führen wird.[4] Will man gleichzeitig einen Wert berechnen und ein Objekt ändern, so sollte man diesen Effekt durch zwei getrennte Routinen – eine Funktion, die den Wert berechnet und eine Prozedur, die das Objekt ändert – auslösen. Es obliegt dann dem Kunden der Klasse zu entscheiden, ob er tatsächlich immer beide Operationen gemeinsam ausführen möchte.

Beim Schreiben von Klassen, die Datenabstraktionen wie Felder, Listen und Matrizen implementieren, ist es sinnvoll, Objekte nicht einfach als passive Sammlungen von Information anzusehen, sondern als "*aktive Datenbehälter*" mit einem internen Zustand. So wird z.B. bei Listen in der Eiffel-Bibliothek ein Cursor mitgeführt, der auf ein aktuell zu bearbeitendes Listenelement zeigt.

Eine weitere Technik wird durch Vererbung möglich gemacht. Erlaubt ein bestimmter Begriff mehrere sinnvolle Implementierungen, die je nach Verwendungszweck verschieden effizient sind, dann ist es angebracht, die Schnittstelle für diesen Begriff als allgemeine Klasse zu deklarieren, und die *verschiedenen Implementierungen* in verschiedenen Erbenklassen, die allesamt nur die in der Ahnenklasse genannte features exportieren. Meist ist die allgemeine Klasse eine aufgeschobene Klasse – Beispiele findet man hierzu in der Eiffel-Bibliothek zuhauf – manchmal aber stellt sie auch eine Standardimplementierung der vielfältig benötigten Routinen bereit, die nur in manchen Spezialfällen verändert wird.

Umgekehrt kann man aber auch *mehrere Schnittstellen für ein und denselben Begriff* anbieten, was durch die Kombination von Vererbung und Re-Export geerbter features ermöglicht wird. Diese Technik ist sinnvoll, wenn verschiedenen Kunden verschiedene Sichten auf dasselbe Objekt geboten werden sollen, wie dies z.B. im Falle der Bankkonten (vgl. Seite 99) der Fall war.

Eine tiefergehende Diskussion von Klassenschnittstellen findet man in [Meyer, 1988, Kapitel 9]. Hier werden die obengenannten Prinzipien anhand einiger Beispiele ausführlich erläutert.

4.1.5 Vererbungstechniken

Vererbung ist einer der entscheidenden Vorteile des objektorientierten Entwurfs und sollte daher gezielt, aber richtig eingesetzt werden. Wie bereits in Abschnitt 3.8.11 besprochen, bestimmt vor allem die logische Beziehung zwischen zwei Klassen, ob besser geerbt oder benutzt werden soll. Vererbung sollte genau dann genutzt werden, wenn Objekte einer Klasse Spezialfälle einer anderen Klasse sind. In allen anderen Fällen ist ein Benutzen sinnvoller. Gründe, von dieser Vorgehensweise abzuweichen, haben wir bereits besprochen.

Im allgemeinen wird man eine neu zu definiernde Klasse als Spezialfall (also Erben) einer bereits existierenden Klasse beschreiben können. Zuweilen steht man aber vor der Situation, daß die existierenden Klassen nicht allgemein genug sind, und man eine Verallgemeinerung erzeugen muß (wie z.B. bei den geometrischen Objekten, die man erst erfunden hatte, nachdem es schon Punkte, Geraden, Dreiecke und Kreise gab). Bisher gibt es keine Programmiersprache, die hierzu einen generellen Mechanismus, also ein Gegenstück zur inherit-Klausel, anbietet. Verallgemeinerungen müssen von Hand durch Editieren des Spezialfalls generiert werden. Außerdem ist es bei Verallgemeinerungen ausnahmsweise nötig, Eingriffe in eine bereits existierende Klasse vorzunehmen: in der spezielleren Klasse muß eine entsprechende Erbklausel mit Redefinitionsvereinbarungen ergänzt werden. Diese Änderungen sind jedoch geringfügig und haben *keine* Effekte auf weitere Klassen.

Redefinition ist eine der wichtigsten Techniken bei Vererbung. Sie sollte immer dann eingesetzt werden, wenn für den Spezialfall der Erben eine effizientere Implementierung als die der Elternklasse gefunden werden kann. Ein besonderer Fall ist dabei die Redefinition von Funktionen als Attribute. Dies kostet zwar Speicherplatz, spart aber zuweilen sehr viel Rechenzeit. Der Einsatz dieses Mittels erfordert aber ein sorgfältiges Abwägen.

Bei Redefinition ist immer zu beachten, daß die Semantik der redefinierten Routine erhalten bleiben muß. In Abschnitt 3.8.10 haben wir Mechanismen kennengelernt, die sicherstellen, daß sowohl die Vor- und Nach-

[4] Eine legitime Ausnahme von dieser Richtlinie wird in [Meyer, 1988, Kapitel 7.7] besprochen.

bedingungen geerbter Routinen als auch die Klasseninvarianten weiterhin Gültigkeit behalten. Da zugunsten der Effizienz eines Programms Zusicherungen jedoch nur auf ausdrücklichen Wunsch überprüft werden, liegt zuweilen die Versuchung nahe, diese Mechanismen zu umgehen, um die Implementierung zu erleichtern. Dieser Versuchung muß in jedem Falle wiederstanden werden, da ansonsten die Korrektheit des gesamten Systems gefährdet wäre.

4.1.6 Beispiel: Bibliothekenverwaltung

Wir wollen nun auf der Basis der oben angesprochenen Leitlinien die Architektur eines Softwaresystems für unser Leitbeispiel "Bibliothekenverwaltung" entwerfen.

4.1.6.1 Analyse der Problemstellung

Im Text des Leitbeispiels auf Seite 60 sind die folgenden Tatbestände und Anforderungen als wesentliche Aufgabenstellungen enthalten.

- Es gibt mehrere Bibliotheken
  - Jede Bibliotheken besitzt eine Menge von Büchern
  - Die Buchbestände der verschiedenen Bibliotheken sind disjunkt
- Es gibt viele Bücher
  - Ein Buch hat einen Autor, einen Titel, ein Themengebiet, und eine Kennung
  - Ein Buch hat Ausleihdaten: letztes Ausleihdatum, Ausleihfrist, letztes Rückgabedatum
  - Ein Buch gehört zu genau einer Bibliothek
  - Eine Bibliothek kann mehrere Exemplare eines Buchs mit gleichem Autor, Titel, und Themengebiet enthalten. Diese haben aber verschiedene Kennungen.
  - Mehrere Bibliotheken können das gleiche Buch (mit gleichem Autor, Titel, Themengebiet und Kennung) enthalten.
  - Ein Buch ist entweder entleihbar, Präsenzexemplar, Semesterapparat oder ausgeliehen. Genau einer der Fälle muß eintreten.
- Es gibt Entleiher
  - Entleiher können entleihbare Bücher aus einer Bibliothek entnehmen, deren Bibliotheksausweis sie besitzen
  - Entleiher können Bücher zu einer Bibliothek zurückbringen
- Es gibt Bibliotheksmitarbeiter
  - Bibliotheksmitarbeiter sind Entleiher
  - Bibliotheksmitarbeiter können neue Bücher dem Bestand ihrer Bibliothek hinzufügen
  - Bibliotheksmitarbeiter können Bücher aus dem Bestand ihrer Bibliothek entfernen
- Es gibt Universitätsangestellte
  - Universitätsangestellte sind Entleiher
  - Universitätsangestellte können in manchen Bibliotheken entleihbare Bücher und Präsenzexemplare entnehmen.
- Es gibt Hochschullehrer
  - Hochschullehrer sind Universitätsangestellte

- Ausleihe
  - Bücher werden nach Autor und Titel ausgeliehen
  - Bücher können an Entleiher ausgeliehen werden, wenn
    - der Entleiher eine Bibliotheksausweis der Bibliothek besitzt
    - die Bibliothek ein ausleihbares Exemplar besitzt
  - Bücher können an Universitätsangestellte ausgeliehen werden, wenn
    - der Universitätsangestellte Zugang hat
    - die Bibliothek ein ausleihbares Exemplar oder ein Präsenzexemplar besitzt.
  - Ausleihdatum und die Ausleihfrist werden vermerkt
  - Die Leihfrist beträgt 4 Wochen nach Entleihdatum
  - Die Leihfrist beträgt für Hochschullehrer ein Semester nach Entleihdatum
- Leihfrist verlängern
  - Die Leihfrist kann um 4 Wochen verlängert werden
  - Die Leihfrist kann für Hochschullehrer ein Semester verlängert werden
  - Nur der Entleiher selbst darf verlängern
- Rückgabe
  - Bücher müssen an die Bibliothek zurückgegeben werden, bei der sie ausgeliehen wurden.
  - Jeder kann ein Buch zurückbringen
  - Das Rückgabedatum wird vermerkt
- Entnahme von Exemplaren
  - Nur Mitarbeiter der Bibliothek dürfen entfernen
  - Entnahme nach Autor, Titel und Kennung
  - Das Buch darf nicht entliehen sein
- Hinzufügen neuer Exemplare
  - Nur Mitarbeiter der Bibliothek dürfen hinzufügen
  - Hinzufügen nach Autor, Titel und Themengebiet
  - Kennung wird automatisch vergeben
  - Ausleihdaten sind leer
- Verwaltung
  - Das Datum wird täglich angepaßt
  - Datum ist von Hand verstellbar
  - Transaktionen werden interaktiv angefordert (nicht direkt im Text aber sinnvoll)
- Vorgesehene Erweiterungenen:
  - Informationen über Autoren werden in einem integrierten Informationssystem aktuell gehalten.
  - Erstellung diverser Verzeichnisse
  - automatische Mahnungen

  Diese Erweiterungenen werden bis auf weiteres zurückgestellt. Der Entwurf soll jedoch die Möglichkeit der späteren Ergänzung vorsehen

4.1.6.2 Zerlegung des Systems in Dienstleistungen einzelner Klassen

Wir beschreiben nun die für eine Realisierung obiger Anforderungen nötigen Klassen und ihre Dienstleistungen. Die meisten Klassen ergeben sich unmittelbar aus der obigen Grobspezifikaktion. Bibliotheken, Bücher, Entleiher, Bibliotheksmitarbeiter, Universitätsangestellte und Hochschullehrer sind auf jeden Fall eigenständige Objekte. Autoren müssen wegen der vorgesehenen Erweiterung als eigenständige Objekte betrachtet werden. Die Verwaltung, die das Datum steuert und durch Einlesen einer gewünschten Transaktion alle Aktivitäten anstößt, kommt als Wurzelklasse dazu.

Die prinzipielle Ablaufstruktur soll der Vorgehensweise in der realen Welt entsprechen. Entleiher beantragen bei einer Bibliothek eine Transaktion – z.B. den Autor und Titel eines Buches, das sie ausleihen wollen – und erhalten je nach ihrem Status verschiedene Berechtigungen. Dies ist nötig, da z.B. bei der Ausleihe die Leihdauern je nach Entleiherklasse unterschiedlich sind. Die Bibliothek bestimmt das konkrete Buch und läßt die Ausleihdaten eintragen oder sie lehnt den Wunsch ab. Das Buch als unabhängiges Objekt trägt die entsprechenden Daten ein. Auf diese Art werden die Schnittstellen klein gehalten.

Diese Struktur wird dadurch etwas verkompliziert, daß bei der Verwaltung im Computer alle Transaktionen zentral angefordert werden müssen. Aufgrund der interaktiven Eingabe muß zunächst der Entleiher bestimmt werden, der die Transaktion beantragt. Der Typ des Entleihers hängt aber von der Art der Transaktion ab, die durchgeführt werden soll. So wird man zum Beispiel bei gewöhnlichen Entleihern die Leistung "Entnahme beantragen" sinnvollerweise überhaupt nicht vorsehen. Aus diesesem Grunde sollte man Transaktionen wie Ausleihe, Leihfrist verlängern, Rückgabe, Entnahme und Hinzufügen als unabhängig agierende Objekte betrachten, die ihrerseits eine Aktion der entsprechenden Entleihergruppe auslösen. Jede Art von Transaktionen muß in einer separaten Klasse aufgeführt werden.

Insgesamt haben wir dann folgende Verarbeitungsstruktur.

- Der Anwender wählt in einem Menü der Verwaltung die Bibliothek und eine Transaktionsart aus.
- Die Verwaltung bildet daraufhin ein Transaktionsobjekt der entsprechenden Klasse, wobei der Entleiher interaktiv bestimmt und die Bibliothek weitergereicht wird.
- Das Transaktionsobjekt veranlaßt den Entleiher, bei der Bibliothek die Transaktion zu beantragen.
- Der Entleiher beantragt bei der Bibliothek die konkrete Transaktion zu der er berechtigt ist.
- Die Bibliothek fragt nun die notwendigen Transaktionsdaten ab, bestimmt das konkrete Buch und löst dort das Eintragen der Transaktionsdaten aus.

Sinnvollerweise werden beim Start des Systems alle Informationen über bisher bekannte Objekte von einer Datei geladen und beim Verlassen wieder gesichert. Zudem bietet es sich an, die Interaktion mit dem Anwender solange aufrecht zu erhalten, bis dieser das System verlassen will.

Diese Vorüberlegungen bestimmen, welche Dienstleistungen zu welcher Klasse gehören und welche Dienstleistungen anderen Klassen zur Verfügung gestellt werden müssen.

Klasse	**Dienstleistung**	**Kunde**
BIB_VERWALT		
	bestehende Objekte laden	*Start*
	Datum anpassen	*Start*
	Ständige Benutzerinteraktion initiieren	*Start*
	Bibliothek auswählen	*Anwender*
	Transaktion auswählen und auslösen	*Anwender*
	Datum verändern	*Anwender*
	bestehende Objekte sichern	*Ende*
	Datum mitteilen	BIBLIOTHEK

Der Anwender ist kein echter Klient. Durch die Interaktion dreht sich die Beziehung um: Er wird vom Klienten zum Anbieter von Informationen![5] Die Kooperation mit dem Anwender läuft über ein hier nicht näher genanntes Ein/Ausgabeobjekt. Dabei können Nachrichten nur in Form von Zeichenketten (`STRING`, `CHARACTER`) oder Zahlen übermittelt werden. Aus diesem Grund muß oft erst einmal ein konkretes Objekt bestimmt oder erzeugt werden, das mit den Anwendereingaben identifiziert werden kann.

Bibliotheken werden von der Verwaltung durch eine nicht weiter spezifizierte Kennung bestimmt, die automatisch im Aufrufmenü eingetragen ist (ggf. könnte der Anwender dann einen Zugriff auf andere Bibliotheken durchführen, wenn er dazu berechtigt ist).

Klasse	**Dienstleistung**	**Kunde**	
TRANSAKTION			*deferred*
	Entleihername eingeben	*Anwender*	
	Transaktion durchführen	BIB_VERWALT	
AUSLEIHE			*Erbe von* TRANSAKTION
	Transaktion erzeugen mit gegebener Bibliothek	BIB_VERWALT	
	Transaktion durchführen	BIB_VERWALT	*redefiniert*
VERLÄNGERN			*Erbe von* TRANSAKTION
	Transaktion erzeugen mit gegebener Bibliothek	BIB_VERWALT	
	Transaktion durchführen	BIB_VERWALT	*redefiniert*
RÜCKGABE			*Erbe von* TRANSAKTION
	Transaktion erzeugen mit gegebener Bibliothek	BIB_VERWALT	
	Transaktion durchführen	BIB_VERWALT	*redefiniert*
ENTNAHME			*Erbe von* TRANSAKTION
	Transaktion erzeugen mit gegebener Bibliothek	BIB_VERWALT	
	Transaktion durchführen	BIB_VERWALT	*redefiniert*
HINZUFÜGEN			*Erbe von* TRANSAKTION
	Transaktion erzeugen mit gegebener Bibliothek	BIB_VERWALT	
	Transaktion durchführen	BIB_VERWALT	*redefiniert*

Bei der Erzeugung einer Transaktion wird der Entleiher durch Abfrage des Namens und Konsultation der Bibliothek bestimmt. Einer Bibliothek müssen alle Personen mit Zugangsberechtigung bekannt sein.[6]

Klasse	**Dienstleistung**	**Kunde**	
ENTLEIHER			
	Ausleihe beantragen bei Bibliothek	AUSLEIHE	
	Verlängerung beantragen bei Bibliothek	VERLÄNGERN	
MITARBEITER			*Erbe von* ENTLEIHER
	Entnahme beantragen bei Bibliothek	ENTNAHME	
	Hinzufügen beantragen bei Bibliothek	HINZUFÜGEN	
UNI_ANG			*Erbe von* ENTLEIHER
	Ausleihe beantragen bei Bibliothek	AUSLEIHE	*redefiniert*
PROFESSOR			*Erbe von* UNI_ANG
	Ausleihe beantragen bei Bibliothek	AUSLEIHE	*redefiniert*
	Verlängerung beantragen bei Bibliothek	VERLÄNGERN	*redefiniert*

Die redefinierten Ausleih- oder Verlängerungsanträge werden unterschiedliche Dienstleistungen der Bibliothek anfordern. Auf diese Art können die Zugriffsrechte leicht geregelt werden. Bei der Rückgabe ist zu beachten, daß *jeder* ein Buch zurückbringen darf. Daher wird in diesem Fall die Transaktion den Entleiher überspringen

[5] Eigentlich ist dies eine skurrile Situation. Statt seine Anweisungen zu geben, wird der Anwender meist durch Menüs "verhört". Vielleicht fühlen sich manche daher vom Computer beherrscht.

[6] Natürlich ist dies eine Vereinfachung gegenüber der wirklichen Vorgehensweise, da Namen oft nicht eindeutig genug sind und für manche Aktionen mehr Sicherheit benötigt wird. So sollte man Entleiher durch Ausweisnummern und Bibliothek identifizieren, Mitarbeiter durch Namen, Bibliothek und ein (geheimes) Passwort, und Universitätsangestellte und Professoren durch ihren Namen, Fachbereich und die Bibliothek.

und direkt mit der Bibliothek in Kontakt treten. Dies entspricht der Tatsache, daß man bei der Rückgabe das Buch auch einfach auf den Tisch legen kann.

Klasse	**Dienstleistung**	**Kunde**
BIBLIOTHEK		
	Entleihbares Buch ausleihen	ENTLEIHER
	Entleihbares Buch oder Präsenzexemplar ausleihen	UNI_ANG
	Entleihbares Buch oder Präsenzexemplar an Professoren ausleihen	PROFESSOR
	Leihfrist verlängern	ENTLEIHER
	Leihfrist für Professoren verlängern	PROFESSOR
	Buch zurücknehmen	RÜCKGABE
	Buch entnehmen	MITARBEITER
	Buch hinzufügen	MITARBEITER
	Autorname, Titel, Themengebiet oder Kennung eingeben	*Anwender*
	Entleiher nach Namen bestimmen	TRANSAKTION (*und Erben*)

Die für eine Transaktion notwendigen Daten werden von der Bibliothek selbst abgefragt, da sie erst beim Ausführen der Transaktion gebraucht werden. Für die Ausleihe sind dies Autorname und Titel, für Verlängerung, Rückgabe und Entnahme die Kennung des Buches und für das Hinzufügen Autorname, Titel und Themengebiet. Durch diese Vorgehensweise wird vermieden, daß Schnittstellen zwischen den Klassen durch Weitergabe zu früh abgefragter Daten unnötig groß werden.

Bücher sind durch ihre Kennung und die Bibliothek eindeutig festgelegt oder werden innerhalb einer Bibliothek nach Autor und Titel gesucht. Einer Bibliothek müssen daher alle eigenen Bücher bekannt sein. Autoren werden durch ihren Namen und Vornamen über das Informationssystem identifiziert.

Klasse	**Dienstleistung**	**Kunde**
BUCH		
	Autor, Titel, Kennung oder Entleihstatus mitteilen	BIBLIOTHEK
	Status auf ausgeliehen setzen mit Leihdatum, Leihfrist	BIBLIOTHEK
	Leihfrist verändern	BIBLIOTHEK
	Status auf entleihbar oder Präsenzexemplar zurücksetzen mit Rückgabedatum	BIBLIOTHEK
	Neues Buch mit Autor, Titel, Themengebiet und Kennung erzeugen	BIBLIOTHEK
INFO_SYSTEM		
	Autor nach Name und Vornamen bestimmen	BIBLIOTHEK
AUTOREN		
	Autor mit Namen und Vornamen erzeugen	INFO_SYSTEM
	Namen mitteilen	INFO_SYSTEM
	Vornamen mitteilen	INFO_SYSTEM

Bei einer späteren Ergänzung mit der Möglichkeit automatischer Mahnungen sollte beim Ausleihen natürlich der Entleiher mit in das Buch eingetragen werden. Das Zurücksetzen des Entleihstatus verlangt, daß dieser Wert beim Ausleihen im Buchobjekt separat gespeichert bleiben muß, was aber andere Objekte nichts angeht. Entnahme geschieht durch ein simples Löschen eines Verweises ohne am Buchobjekt selbst etwas zu ändern.

Damit ist die Struktur der Klassen und die Kooperation zwischen verschiedenen Objekten geregelt. Wir konnten die Schnittstellen schmal halten und vor allem auch fast ausschließlich in einer Richtung (von Verwaltung zum Buch, aber nicht zurück) organisieren.[7] Für jede Leistung müssen nun noch die Argumente beschrieben und Kontrakte gemäß den Anforderungen festgelegt werden. Darauf wollen wir in diesem Rahmen verzichten.

[7]Für eine effiziente Verwaltung ist es unter Umständen noch sinnvoll, in jedem Objekt noch eine Reihe von internen Rückreferenzen anzulegen wie zum Beispiel von Büchern auf ihre Bibliothek, von Entleihern auf die Bibliotheken, bei denen sie Zugang haben, usw. Dies aber geht zu Lasten der übersichtlichen Schnittstellenstruktur und sollte sorgfältig überlegt sein.

Es sei an dieser Stelle angemerkt, daß die genaue Typisierung mancher Argumente nach einer Datenstruktur verlangt, die üblicherweise nicht zu den vordefinierten Typen der Eiffel-Basisbibliothek gehört. Zur Speicherung der Bücher und Entleiher einer Bibliothek sowie der Autoren des Informationssystems benötigen wir eine generische Klasse, die dem mathematischen Konzept der Menge entspricht. Dies ist nötig, weil wir beliebig viele Bücher, Entleiher und Autoren zulassen wollen, Elemente hinzufügen, nach Kriterien wie Autor und Titel (und ggf. Entleihstatus) suchen, und entnehmen wollen. Indizes, wie Felder sie anbieten, werden nicht gebraucht. Stattdessen muß gesichert sein, daß Elemente nicht doppelt vorkommen. Aus diesem Grunde muß zu den obengenannten problemspezifischen Klassen eine generische Klasse `SET` hinzugefügt werden, deren Dienstleistungen sich aus den Anforderungen des Problems und den üblichen Mengenoperationen ergeben.

4.1.6.3 Vollständige Klassenstruktur

An dieser Stelle müsste nun die obige Beschreibung in eine vollständig beschriebene Struktur von Eiffel-Klassen umgesetzt werden, in der alle Dienstleistungen, die für die Modellierung des Problems relevanten Attribute (z.B. die Komponenten eines Buches) und eventuell weitere für die Verarbeitung hilfreiche features explizit genannt, typisiert und mit Vor- und Nachbedingungen versehen sind. Der selektive Export gemäß der oben genannten Kunden wäre explizit zu vereinbaren und Erbbeziehungen wären in inherit-Klauseln zu formulieren.

Es würde den Rahmen dieses Skriptes sprengen, dieses Beispiel bis ins letzte Detail auszuführen. Wir belassen es daher bei der bisher gegebenen Feinspezifikation, die mit entsprechendem Platz- und Zeitaufwand (aber ohne weitere Entwurfsüberlegungen) in eine vollständige Klassenstruktur umgesetzt werden kann.

4.1.7 Ästhetik der Programmierung

Ein letzter Aspekt, der beim Entwurf – genauso wie bei der Implementierung – eine Rolle spielen sollte, ist eher ästhetischer Natur. Es gibt viele Programme, die ihre Aufgabe erfüllen, aber unter diesen gibt es welche, deren Vorgehensweise man als "schlechten Programmierstil" bezeichnen würde.

Was ein guter und was ein schlechter Programmierstil ist, läßt sich nicht in voller Allgemeinheit sagen, da die Ansichten darüber zum Teil auseinanderdriften. Manche halten Programme für gut, wenn sie nur wenige Klassen und Routinen benötigen, kurzen Code haben, oder durch geschickte Anordnung von Anweisungen sehr effizient sind. Andere bevorzugen Programme, in denen der Code einer einzelnen Routine kurz ist.

Nach unserer Ansicht sollte ein Entwurf so gestaltet sein, daß seine Struktur für Andere durchschaubar bleibt und die wesentlichen Ideen ohne große Erklärungen erkennbar sind. Dies ist meist der Fall, wenn ein klarer Bezug zur realen Welt zu erkennen bleibt, die modelliert wird, wobei auch die Namensgebung für Klassen, features und Parameter eine nicht unwesentliche Rolle spielt. Auch bedeutet dies, daß Zusicherungen – die ja auch einen Teil der Dokumentation eines Programms darstellen – nur features benutzen, die auch exportiert werden.[8] Interessanterweise stimmen die Beurteilungen konkreter vorgegebener Programme durch verschiedene Personen meist überein[9] und deshalb ist es auch hier wichtig, Erfahrungen zu sammeln und eigene Programme der Kritik anderer zu stellen, um die eigene Beurteilungsfähigkeit zu schärfen und im Laufe der Zeit "bessere" Programme zu schreiben.

Ästhetik ist ein Kriterium, das zuweilen auf Kosten der Effizienz geht, aber bedeutend ist für das Verständnis. Dies geht Hand in Hand damit, daß man die Korrektheit eines Programms leicht einsehen und ggf. auch leicht beweisen kann. Oft versteht man ein Programm viel besser, wenn man die Gründe aufgeschrieben hat, *warum* der Entwurf und die implementierte Methode auch tatsächlich funktioniert. Dies leitet uns über zum nächsten Thema, der Implementierung von Programmen und dem Nachweis ihrer Korrektheit.

[8] Dies wird von Eiffel nicht kontrolliert und leider sind auch die Musterlösungen zuweilen von diesem Prinzip abgewichen.

[9] Sie sollten während einer Übungstunde einmal Kriterien zusammentragen, ausdiskutieren und dann eine Liste der Kriterien für einen guten Programmierstil, zusammenstellen, auf die Sie sich einigen können.

4.2 Verifikation

Beim Entwurf haben wir Vor- und Nachbedingungen an Routinen sowie Klasseninvarianten eingeführt, um "per Vertrag" die Verteilung der Aufgaben auf die einzelnen Module eines Softwaresystems klar zu regeln. Nach dem Entwurf der Systemarchitektur und der Verteilung der Einzelaufgaben ist es nun möglich, jedes Modul einzeln und unabhängig von den anderen zu implementieren. Statt einer globalen Sicht, die bisher erforderlich war, ist es nun (endlich) möglich, lokal zu arbeiten, und auf konventionellere Programmierkonzepte zurückzugreifen, zu denen seit vielen Jahren Erfahrungen vorliegen und weiter entwickelt werden.

Während dieser Implementierungsphase sind natürlich ganz andere Qualitäten gefragt als beim Entwurf, in dem Erweiterbarkeit und Wiederverwendbarkeit die wichtigste Rolle spielten. Nun, da die Verträge erstellt sind, geht es darum, daß sie auch eingehalten werden. Ansonsten wäre die Modularisierung wertlos und eine Aufteilung der Verantwortung auf mehrere Softwareentwickler unmöglich, denn man könnte sich ja nicht darauf verlassen, daß die benutzten Dienstleitungen anderer Module auch so funktionieren, wie es vereinbart war. *Zuverlässigkeit*, also Korrektheit und Robustheit stehen bei der Implementierung von Klassen und ihren einzelnen Routinen im Vordergrund. Die Frage, die sich ein verantwortungsbewußter Programmierer nun stellen sollte, lautet: "*Wie implementiere ich eine Routine, deren Zuverlässigkeit ich sicherstellen kann*?". Bei der Beantwortung dieser Frage sind zwei Aspekte zu berücksichtigen.

Zum einen muß man natürlich wissen, welche Hilfsmittel überhaupt bei der Implementierung zur Verfügung stehen, also aus welchen Programmkonstrukten und elementaren Ausdrücken man den Anweisungsteil einer Routine aufbauen kann. Hier gibt es zwischen den meisten höheren Programmiersprachen nur geringfügige Unterschiede. Fast alle bieten Konstrukte für die in Abschnitt 1.3.2 angedeuteten Strukturierungskonzepte für Implementierungen an – also für eine Verfeinerung in eine Folge kleinerer Teilschritte, Fallunterscheidung, Wiederholung, Prozeduralisierung und ggf. auch Rekursion.[10] Da diese konzeptionell nicht so schwer zu verstehen sind und Probleme allenfalls bei der syntaktischen Beschreibungsform der Programmiersprache auftreten können, kann man sich die für eine Implementierung nötigen Grundkenntnisse relativ leicht aneignen.

Zum zweiten aber – und das ist wesentlich schwieriger – muß man wissen, wie man diese Programmierkonstrukte *einsetzt*, um ein zuverlässiges Programm zu erhalten. Dabei läßt sich die Frage, wie man denn überhaupt eine lauffähige Routine erstellt, von der Frage nach der Korrektheit dieser Routine nicht trennen. Denn wer ein Programm entwickelt, der wird auch eine Vorstellung davon haben, warum dieses Programm denn so funktionieren soll, wie es vorgesehen ist. Wer also ein zuverlässiges Programm erstellen möchte, der sollte prinzipiell auch in der Lage sein, die Korrektheit zu garantieren. Warum reichen hierfür die logischen Ausdrücke in Invarianten und Vor- und Nachbedingungen von Routinen nicht aus?

Invarianten, Vorbedingungen und Nachbedingungen werden nur bei der Programm*durchführung* kontrolliert, wenn die Durchführung an dieser Prüfstelle vorbeikommt. Ist die Prüfung negativ, so bedeutet dies, daß die Vorstellungen des Programmierers vom Programmablauf und der tatsächliche Programmablauf nicht übereinstimmen: ein Fehler wurde entdeckt. Ist aber die Prüfung in allen durchgeführten Fällen positiv, dann weiß man nur, daß in diesen speziellen Fällen kein Fehler gefunden wurde. Das sagt aber überhaupt nichts darüber aus, wie sich das Programm in allen anderen Fällen verhält, die in seinem späteren Einsatz vorkommen mögen.

Man kann die Korrektheit eines Programms nicht mit Programmabläufen prüfen.

Deshalb ist es nötig, die Korrektheit von Programmen völlig unabhängig von konkreten Eingabewerten *beweisen* (*verifizieren*) zu können, und dies in einer mathematisch präzisen Form zu tun. Prinzipiell ist es sogar wünschenswert, den Beweis in einem logischen Kalkül – also einer Erweiterung des in Abbildung 2.14 auf Seite 49 vorgestellten Ableitungskalküls für die Prädikatenlogik – auszuführen, da dieser Beweis seinerseits durch

[10] Differenzen liegen nur in der syntaktischen Beschreibungsform dieser Konstrukte, dem Angebot mehrerer ähnlicher Konstrukte (z.B. `while` *und* `for`-Schleife in Pascal), und dem Umfang der vordefinierten Ausdrücke, die man nicht mehr selbst programmieren muß. Deshalb genügt es auch, diese Konzepte exemplarisch an der Sprache Eiffel zu besprechen. Hat man sie einmal verstanden, so kann man relativ leicht dasselbe Problem in einer anderen Programmiersprache implementieren, indem man im entsprechenden Manual nach der Syntax vergleichbarer Konstrukte sucht.

einen Rechner überprüft werden kann.[11] Beweisen ist derzeit jedoch noch sehr kostspielig, da es – wie für die Erzeugung von Implementierungen – keine allgemeingültigen Verfahren gibt, mit denen Beweise durchgeführt werden können (dies ist schon ist aus theoretischen Gründen grundsätzlich unmöglich). Auch mangelt es an Werkzeugen, die eine präzise Beweisführung angemessen unterstützen. Bisher erfordert es immer noch eine Menge Training und einen hohen intellektuellen Aufwand, der für manche Produkte einfach zu teuer ist. Trotzdem sollte man immer wieder versuchen, wenigstens die sicherheitsrelevanten Teile eines Systems zu beweisen (z.B. daß der Zugang zum Password nur in der geplanten Form möglich ist).

Neben der Korrektheitsgarantie gibt es aber auch noch einen anderen Grund, Programme zu beweisen. Es ist ein Spiel von hohem intellektuellen Reiz, das zudem auch die "Ästhetik" von Algorithmen begreifbar macht. Bei guten Programmen fällt es leicht, einen Korrektheitsbeweis zu führen: die wesentlichen Ideen sind leicht zu erkennen und die Struktur ist durchschaubar. Das Führen des Beweises birgt den Schlüssel zu einem Verständnis, das weit über das "es funktioniert" hinausgeht. Wer *selbst* Programme beweist, beginnt die Methodik zu erfassen, die bei der Implementierung implizit im Raume stand, und wird verstehen, in welchem Rahmen sich diese Methodik auf die Lösung anderer Probleme übertragen lassen.[12] Gewöhnt man sich an, bei der Implementierung gleichzeitig auch schon an einen möglichen Korrektheitsbeweis zu denken, so wird dies die Qualität des erstellten Programms erheblich steigern. Es empfiehlt sich fast, einen Beweis simultan zu der Implementierung zu erstellen. Auf Methoden einer derartigen Form der Programmerstellung werden wir im letzten Abschnitt dieses Kapitels ein wenig eingehen.

Wie bereits erwähnt, gibt es keine allgemeine Methodik des Beweisens. Wir können daher in diesem Skript nicht genau beschreiben, *wie* man Beweise führen kann, sondern nur einige vage Leitlinien angeben, Beweise an Beispielen demonstrieren, und versuchen, Sie dabei nicht in einer Flut von Formeln ersticken zu lassen, was wegen der Komplexität solcher Beweise leider leicht geschieht. Wichtig ist daher, daß Sie *selber* Beweise entwickeln und dies an kleinen und mittleren Beispielen trainieren. Erst hierdurch kommt die oben erwähnte Ästhetik von Programmen wirklich zutage. Als weitere Anregung sind die Bücher von Dijkstra "A Discipline of Programming" [Dijkstra, 1976] und Gries "The Science of Programming" [Gries, 1981] sehr zu empfehlen.

4.2.1 Korrektheit von Routinen und Klassen

Zum Beweis der Korrektheit einer Routine gehen wir aus von dem vereinbarten Kontrakt, also der Vorbedingung `pre`, die zu Beginn des Anweisungsteils zu finden ist, und der Nachbedingung `post`, die am Ende Gültigkeit haben soll. Um nun nachzuweisen, daß für jede akzeptable Eingabe beim Verlassen der Routine die Nachbedingung gilt, versuchen wir, Aussagen darüber zu treffen, welcher Zustand nach jeder einzelnen Anweisung erreicht ist, und *dies schrittweise zu beweisen*. Diese schreiben wir "zwischen" die einzelnen Anweisungen. Die Implementierung wird also ergänzt um logische Bestandteile, die nicht zur Programmiersprache Eiffel gehören.[13]

[11] Die Realität ist leider noch nicht soweit: Es fehlt noch an der Qualifikation der Systementwickler, Beweise mit akzeptablen Aufwand durchzuführen, an praktikablen Programmiersprachen, für die ein Beweiskalkül existiert, und schließlich an akzeptierten Beweisprüfern. Trotzdem ist das Ziel des Programmbeweisens aus gesellschaftlichen und Umwelt-Gründen notwendig.

Leider gibt ein Programmbeweis nur die Aussage, daß das Programm auf einer fiktiven mathematischen Maschine (der Semantikdefinition der Programmiersprache) korrekt läuft, aber nicht, daß sich das übersetzte Programm im Rahmen eines Betriebssystem auf einer realen Hardware sich korrekt verhält. Daher sind neben der mathematischen Methode des Programmbeweisens auch die Stichprobenverfahren der statistischen Qualitätsprüfung notwendig. Diese werden *Tests* genannt.

Ein Beweis ohne systematisches Testen gibt keine hohe Sicherheit. Systematisches Testen allein gibt nur eine schwache Wahrscheinlichkeitsaussage über die Zuverlässigkeit. Testen ist jedoch für die anderen Qualitätsmerkmale der Angemessenheit des Produkts, wie Effizienz, Benutzerfreundlichkeit usw. unbedingt erforderlich.

[12] Aus dem Versuch, Korrektheitsbeweise zu systematisieren, haben sich sogar erste Systeme ergeben, die einfache Programme automatisch aus ihren Vor- und Nachbedingungen *synthetisieren* können. Ohne eine Analyse von Programmen in Form eines Korrektheitsbeweises kann man eine Antwort auf die Frage "wie implementiere ich zuverlässige Programme" wohl kaum finden.

[13] Da der Beweis jedoch eine Aussage *über* das Programm ist, die nur für die Überprüfung, nicht aber für den Ablauf des Programms relevant ist, gehören diese Bestandteile nicht zum eigentlichen Programm und könnten, wenn man sie unbedingt in den Programmtext integrieren möchte, bestenfalls als Kommentare aufgenommen werden.

Um diese Bestandteile deutlich genug vom Programmtext abzugrenzen, hat sich eingebürgert, sie in geschweifte Klammern zu setzen. Bei einer einfachen Folge von Anweisungen sähe dann ein durch logische Bestandteile ergänztes Programm wie folgt aus

```
  ⋮
is --
require pre
do                          {pre}
   Anweisung_1;             {Aussage_1}
   Anweisung_2;             {Aussage_2}
        ⋮
   Anweisung_n              {post}
ensure  post
end
```

`Aussage`$_1$ ist dabei zugleich die Nachbedingung von `Anweisung`$_1$ als auch die Vorbedingung von `Anweisung`$_2$. Ähnliches gilt für `Aussage`$_2$, `Aussage`$_3$ usw. `pre` ist als Vorbedingung der ganzen Routine insbesondere auch Vorbedingung von `Anweisung`$_1$, `post` als Nachbedingung der ganzen Routine auch Nachbedingung von `Anweisung`$_n$. Wir haben bei dieser Vorgehensweise die Schreibweise des sogenannten *Hoare-Kalküls* verwendet.

Definition 4.2.1 (Notation des Hoare-Kalküls)

Die Schreibweise `{pre} instruction {post}`,
bezeichnet einen Satz des Kalküls für Programmbeweise.

Er ist wahr genau dann, wenn `pre` *eine Vorbedingung dafür ist, daß die Anweisung* `instruction` *terminiert und daß nach Ausführung von* `instruction` *die Aussage* `post` *gilt.*

Für die Aussagen `pre` *und* `post` *sind Ausdrücke der Prädikatenlogik erlaubt, ergänzt um die Zusicherungssprache von* Eiffel. `instruction` *muß eine korrekte* Eiffel *Anweisung sein.*

`{pre} instruction {post}` besagt also, daß die Berechnung der Anweisung `instruction` immer zu einem Ergebnis führt (terminiert) und daß im Anschluß daran die Aussage `post` gilt, vorausgesetzt, daß vor der Ausführung von `instruction` die Aussage `pre` gültig war. Ist dies nicht der Fall, so wird gemäß den üblichen Regeln der logischen Implikation überhaupt nichts gefordert. Die Anweisung braucht nicht einmal zu terminieren. Die Anweisung `instruction` darf auch eine komplexere Anweisung (ein `Compound` gemäß der Syntaxbeschreibung in Abschnitt 4.3.12.1) sein.

Beispiel 4.2.2

Ein einfacher wahrer Satz des Kalküls für Programmbeweise ist

`{x=0 ∧ y≥4} x:=y+3; y:=y-4 {x≥5 ∧ y≥0}`

An diesem Beispiel sieht man, daß Vor- und Nachbedingungen nicht unbedingt "optimal" sein müssen. Nach Ausführung von `x:=y+3` *wissen wir nämlich sogar, daß* `x≥7` *gilt. Die Nachbedingung ist also schwächer als das, was wir beweisen könnten, – oder die Vorbedingung ist stärker als das, was wir benötigen.*

Die Terminierungsbedingung in Definition 4.2.1 ist wichtig, weil es durchaus der Fall sein kann, daß die Berechnung von Anweisungen, die Schleifen enthalten, bei bestimmten Eingaben niemals endet. In diesem Fall kann man natürlich auch keine Nachbedingung mehr beweisen. Deswegen unterscheidet man auch zwei Formen von Korrektheit einer Anweisung mit Vorbedingung `pre` und Nachbedingung `post`. Verlangt man, daß `{pre} instruction {post}` wahr ist im Sinne der Definition 4.2.1, so spricht man von *totaler Korrektheit*. Schwächt man die Definition dahingehend ab, daß aus der Vorbedingung `pre` *nicht* die Terminierung der Anweisung folgen muß, und `post` nur gelten muß, wenn `pre` gilt *und* die Anweisung terminiert, dann spricht man von partieller Korrektheit.

Da aus Terminierung und partieller Korrektheit die totale Korrektheit folgt, hat es sich für die Beweisführung als zweckmäßig herausgestellt, den Beweis der Terminierung von dem der partiellen Korrektheit zu trennen.

Definition 4.2.3 (Korrektheit von Routinen)

1. *Ist r eine Routine, so bezeichnet* $\underline{\texttt{pre}_r}$ *die Vorbedingung von r,* $\underline{\texttt{post}_r}$ *die Nachbedingung von r und* $\underline{\texttt{B}_r}$ *den Anweisungsteil (Body) von r.*

   Die gültigen Argumente von r sind alle Werte, die für die formalen Argumente unter Einhaltung der Typbedingungen eingesetzt werden dürfen.

2. *Eine Routine r heißt genau dann (total) korrekt, wenn für alle gültigen Argumente x_r der Satz*

   $$\{\texttt{pre}_r(x_r)\}\ \texttt{B}_r\ \{\texttt{post}_r(x_r)\}$$

   ein wahrer Satz des Kalküls für Programmbeweise ist.

Ebenso läßt sich nun auch die Korrektheit einer ganzen Klasse präzise definieren. Eine Klasse K ist genau dann korrekt, wenn ihre durch die Anweisungsteile der Routinen gegebene Implementierung mit der durch Invariante, Vor- und Nachbedingungen gegebenen Spezifikation konsistent ist.

Definition 4.2.4 (Korrektheit von Klassen)

Eine Klasse K mit Invariante `INV` *heißt genau dann korrekt, wenn gilt*

1. *Für jede exportierte Routine r von K und alle gültigen Argumente x_r ist der Satz*

   $$\{\texttt{INV}\ \wedge\ \texttt{pre}_r(x_r)\}\ \texttt{B}_r\ \{\texttt{INV}\ \wedge\ \texttt{post}_r(x_r)\}$$

   ein wahrer Satz des Kalküls für Programmbeweise.

2. *Bezeichnet* $\texttt{Default}_K$ *die Zusicherung, daß alle Attribute von K die Initialwerte ihrer Typen tragen, so gilt für jede Initialisierungsprozedur i von K und alle gültigen Argumente x_i der Satz*

   $$\{\texttt{Default}_K\ \wedge\ \texttt{pre}_i(x_i)\}\ \texttt{B}_i\ \{\texttt{INV}\ \wedge\ \texttt{post}_i(x_i)\}$$

Gibt es keine Initialisierungsprozedur, so bedeutet die zweite Bedingung schlicht, daß $\texttt{Default}_K$ die Invariante `INV` impliziert, d.h. daß alle Initialwerte die Invariante erfüllen.

Die Notation des Hoare-Kalküls erlaubt es, schrittweise Aussagen über die Wirkungen einzelner und durch Programmkonstrukte zusammengesetzter Anweisungen als mathematischen Satz eines formalen Kalküls zu formulieren und zu beweisen. Zur Formulierung steht uns die volle Sprache der Prädikatenlogik, ergänzt um boolesche Ausdrücke von Eiffel, zur Verfügung und zum Beweis der Ableitungskalkül der Prädikatenlogik (Seite 49), ergänzt um Regeln für jedes einzelne Programmierkonstrukt. Letztere werden wir im Detail in Sektion 4.3 besprechen, wenn wir die verschiedenen Programmierkonstrukte von Eiffel vorstellen.

4.2.2 Ein Kalkül für Verifikation

Die obige Definition der Wahrheit von Sätzen des Kalküls für Programmbeweise ist für die Durchführung von Beweisen unhandlich, da sie die Auswertung der Semantik benötigt, was im allgemeinen sehr aufwendig ist. Aus diesem Grunde hat man – analog zur Prädikatenlogik – einen Ableitungskalkül entwickelt, durch den die Beweisführung auf die Anwendung formaler syntaktischer Manipulationen (Beweis*regeln*) reduziert werden kann. Die konkreten Regeln lassen sich aus der genauen Definition der Semantik beweisen, was aber nicht Thema dieser Einführungsveranstaltung sein soll.

Ein weiterer Zweck dieses Kalküls ist es, eine allgemein akzeptable Prüfung zu erlauben, ob für einen vorgegebenen Programmteil `instructions` die Korrektheit bezüglich seiner Spezifikation durch `pre` und `post` garantiert ist. Dies ist sinnvoll für Gruppen, die nicht in die Entwicklung involviert sind, wie zum Beispiel die innerbetrieblichen Qualitätskontrolle oder den TÜV, der in den kommenden Jahren immer mehr für eine außerbetriebliche Softwarekontrolle eingesetzt werden soll. Diese Gruppen interessieren sich nicht für den Entwicklungs*prozeß*, sondern nur für die Korrektheit des Ergebnisses. Für die Entwickler selbst ist die Trennung von Programmentwicklung und Verifikation unsinnig, da hierbei zweimal über dasselbe Programmstück nachgedacht werden muß.[14]

Die Grundüberlegung, die zu einem sehr vielseitigen Verifikationskalkül geführt hat, ist die folgende. Gegeben sei eine Folge von Anweisungen $\texttt{inst}_1\texttt{;...;inst}_n$ samt Vorbedingung `pre` und Nachbedingung `post`. Um zu beweisen, daß

$$\{\texttt{pre}\}\ \texttt{inst}_1\texttt{;...;inst}_n\ \{\texttt{post}\}$$

wahr ist, überlegt man sich, was die – im logischen Sinne – schwächste Vorbedingung $\texttt{pre}_n$ ist, so daß gilt

$$\{\texttt{pre}_n\}\ \texttt{inst}_n\ \{\texttt{post}\}$$

$\texttt{pre}_n$ ist dann die Nachbedingung für den Algorithmus ohne den letzten Schritt $\texttt{inst}_n$ und wir können das Verfahren wiederholen, bis wir eine Vorbedingung $\texttt{pre}_1$ für $\texttt{inst}_1$ gefunden haben, die schwächer ist als die Vorbedingung `pre`, d.h. für die $\texttt{pre} \Rightarrow \texttt{pre}_1$ gilt. Ist dies gelungen, so ist der Algorithmus als korrekt bewiesen worden.[15] Für Konstrukte wie Schleifen wird dies natürlich etwas aufwendiger, da man hier darüber nachdenken muß, was sich während eines Schleifendurchlaufs verändert und was nicht – also *invariant* bleibt.

In dem Kalkül von Dijkstra [Dijkstra, 1976] geht man sogar noch einen Schritt weiter. Dort wird zu jeder Instruktionsart eine *Berechnungsvorschrift* angegeben, *wie* man von einer vorgegebenen Nachbedingung `post` und einer Instruktion `instruction` zu derjenigen Vorbedingung `pre` kommt, welche die *geringsten* Forderungen stellt und dennoch $\{\texttt{pre}\}\ \texttt{instruction}\ \{\texttt{post}\}$ garantiert. Diese Berechnungsvorschrift **wp** (für *weakest precondition*) ist eine Prädikatentransformation

$$\textbf{wp} \ : \ \text{Instruction} \times \text{Predicate} \rightarrow \text{Predicate}$$

derart, daß für alle Anweisungen `instruction`, alle Nachbedingungen `post` und alle Vorbedingungen `pre` gilt:

$$\{\textbf{wp(instruction,post)}\}\ \texttt{instruction}\ \{\texttt{post}\}$$

und $\{\texttt{pre}\}\ \texttt{instruction}\ \{\texttt{post}\} \ \Rightarrow\ (\ \texttt{pre} \Rightarrow \textbf{wp(instruction,post)}\ )$

D.h. **wp(inst,post)** ist eine korrekte Vorbedingung und sie ist schwächer als alle anderen.

Beispiel 4.2.5 (Weakest Precondition)

Für die Nachbedingung $\texttt{x}\geq 5 \wedge \texttt{y}\geq 0$ *erhalten wir bei verschiedenen Anweisungen folgende schwächste Vorbedingungen:*[16]

$$\begin{aligned}
&\textbf{wp}(\ \texttt{x:=y+5}\ ,\ \texttt{x}\geq 5 \wedge \texttt{y}\geq 0\ ) &&\equiv\ \texttt{y}\geq 0\\
&\textbf{wp}(\ \texttt{y:=0;x:=y+5}\ ,\ \texttt{x}\geq 5 \wedge \texttt{y}\geq 0\ ) &&\equiv\ \texttt{true}\\
&\textbf{wp}(\ \texttt{x:=y+5;y:=y-4}\ ,\ \texttt{x}\geq 5 \wedge \texttt{y}\geq 0\ ) &&\equiv\ \texttt{y}\geq 4
\end{aligned}$$

Leider sind die schwächsten Vorbedingungen für Schleifen und Prozeduren sehr unhandlich, da sie Existenzquantoren einführen und man bei Verwendung der mechanisch konstruierten Vorbedingung in einem Dschungel von Quantoren stecken bleibt. Aus diesem Grunde werden wir nicht direkt im Kalkül von Dijkstra arbeiten sondern in dem etwas schwächeren von Hoare, bei dem man die Vorbedingungen selbst finden muß. Als *Vorschläge* für Vorbedingungen, die wir nicht mehr beweisen müssen, werden wir dennoch die Prädikatentransformation mitbenutzen. Diese Vorschläge werden wir jedoch meist in stärkere, aber einfachere Zusicherungen überführen müssen, die weniger Quantoren enthalten. Hierzu werden wir ein gehöriges Maß an Intuition benötigen, da die Frage nach einer schematischen, aber korrekten Vereinfachung von Vorbedingungen immer noch ein Thema der Grundlagenforschung ist.

Anstelle mit der schwächsten Vorbedingung rückwärts zu gehen, kann man übrigens auch versuchen, vorwärts zu gehen und die stärkste Nachbedingung **sp** (strongest postcondition) zu finden. Diese spielt aber für die

[14] Sinnvoller ist es, ausgehend von der Nachbedingung der Spezifikation, das Programmstück zu konstruieren und im Laufe des Konstruktionsprozesses auf eine Vorbedingung zu kommen, die weniger restriktiv ist als die Vorbedingung der Spezifikation.

[15] Diese Vorgehensweise läßt sich auch für die simultane Entwicklung von Programm und Beweis einsetzen, wenn man zu Beginn bereits eine ungefähre Idee im Kopf hat, wie der Algorithmus vorgehen soll. In diesem Fall konstruiert man in jedem Schritt die entsprechende Anweisung $\texttt{inst}_i$ gleichzeitig mit $\texttt{pre}_i$.

[16] Wir benutzen das Symbol $\equiv$ anstelle des Gleichheitssymbols `=`, um Gleichheit von Prädikaten auszudrücken und Verwechslungen mit Gleichheiten innerhalb von Zusicherungen zu vermeiden.

Programmkonstruktion keine Rolle, da die *Nachbedingung* die Wirkung eines Programmstücks festlegt und damit im Programm "zurückgerechnet" wird. Die "Vorwärtsrechnung" von der Vorbedingung (zumeist `true`) zur Nachbedingung wirkt ziellos, da die Vorbedingung keine Information über das Ziel in sich trägt.

Bei all diesen Überlegungen wurden die folgenden Regeln ohne weitere Begründung verwendet:

$$\frac{\texttt{pre} \Rightarrow \texttt{pre'},\ \ \{\texttt{pre'}\}\ \texttt{instruction}\ \{\texttt{post}\}}{\{\texttt{pre}\}\ \texttt{instruction}\ \{\texttt{post}\}}$$

Verstärkung der Vorbedingung (**VV**)

$$\frac{\{\texttt{pre}\}\ \texttt{instruction}\ \{\texttt{post'}\},\ \ \texttt{post'} \Rightarrow \texttt{post}}{\{\texttt{pre}\}\ \texttt{instruction}\ \{\texttt{post}\}}$$

Abschwächung der Nachbedingung (**AN**)

Abbildung 4.1: *Verstärkungs- und Abschwächungsregeln für Programmbeweise*

Die Verstärkung der Vorbedingung bzw. die Abschwächung der Nachbedingung erlaubt uns, einen Programmbeweis in folgender Form zu beschreiben:

Programm	**Zusicherungen**	**Prämissen**
	`{pre}`	`pre ⇒ pre'`
	`{pre'}`	
`instruction`		
	`{post'}`	`post' ⇒ post`
	`{post}`	

Vor jeder und nach jeder Anweisung `instruction` steht eine Reihe von Zusicherungen, die sich von oben nach unten jeweils abschwächen. Am rechten Rand wird zusätzlich angegeben, warum diese Abschwächung zulässig ist. Das Lesen (und meist das Erzeugen) des Programmbeweises erfolgt dann *von unten nach oben*. Folgt oberhalb einer Zusicherung wieder eine Zusicherung, so beschreiben die Anmerkungen der oberen Bedingung, warum diese Verschärfung zulässig ist. Folgt eine Instruktion, so ist deren Vorbedingung über die jeweilige Regel für diesen Instruktionstyp z.B. über die schwächste Vorbedingung nachzuweisen.

Zwei weitere Regeln sind sinnvoll, falls die Prädikate zu kompliziert werden:

$$\frac{\{\texttt{pre}\}\ \texttt{instruction}\ \{\texttt{post}\},\ \ \{\texttt{pre'}\}\ \texttt{instruction}\ \{\texttt{post}\}}{\{\texttt{pre} \vee \texttt{pre'}\}\ \texttt{instruction}\ \{\texttt{post}\}}$$

Kombination der Vorbedingungen

$$\frac{\{\texttt{pre}\}\ \texttt{instruction}\ \{\texttt{post}\},\ \ \{\texttt{pre}\}\ \texttt{instruction}\ \{\texttt{post'}\}}{\{\texttt{pre}\}\ \texttt{instruction}\ \{\texttt{post} \wedge \texttt{post'}\}}$$

Kombination der Nachbedingungen

Abbildung 4.2: *Regeln für die Kombination von Zusicherungen in Programmbeweisen*

Diese beiden Regeln erlauben es, ein Programm einzeln für Teilzusicherungen zu prüfen. Die erste Regel gibt an, daß bei mehreren alternativen Vorbedingungen (`pre ∨ pre'`), die zweite Regel, daß bei mehreren Anforderungen (`post ∧ post'`) der Beweis einzeln geführt werden kann.

Wir wollen nun im folgenden Abschnitt für alle Programmkonstrukte der Sprache Eiffel weitere formale Regeln angeben, die uns sagen, wie die Korrektheit einer komplexeren Anweisung aus Eigenschaften ihrer Bestandteile abgeleitet werden kann. Diese Regeln reichen aber noch nicht aus, um ein Programm zu beweisen. Hinzu kommen müssen die Regeln aus den jeweiligen Anwendungsgebieten. Soll etwas numerisch berechnet werden, so benötigen wir natürlich die Regeln der Arithmetik mit dem zusätzlichen Wissen über die Eigenschaften der Funktionen, die verwendet und berechnet werden sollen: z.B. `a>b ⇒ ggt(a-b,b) = ggt(a,b)`. Diese Regeln werden im folgenden als bekannt vorausgesetzt, ohne deren Kalkül explizit anzugeben.

Als Schlußbemerkung wollen wir noch anfügen, daß alle bisherigen Verifikationsmechanismen *ausschließlich für Programme mit Copy-Semantik* entwickelt wurden. Die Regeln, die wir im folgenden angeben werden, können

daher nur für Beweise von Routinen benutzt werden, die keine gemeinsam genutzten Objekte verändern, in denen also keine zwei Größen auf dasselbe Objekt verweisen (was man ggf. separat "beweist").[17] Für die volle Referenzsemantik gibt es bis heute keine brauchbare Verifikationstechnik, da sie erheblich komplizierter ist.[18]

4.3 Strukturierung von Routinen

Wir wollen in diesem Abschnitt nun endlich die Programmstrukturen besprechen, die Sie bei der Implementierung von Eiffel-Routinen benötigen. Wir werden dazu zunächst die einzelnen Sprachkonzepte zusammen mit den zugehörigen Verifikationsregeln, also das Handwerkzeug, vorstellen und erst im Anschluß daran eine genauere Sprachbeschreibung geben. Am Ende dieser Sektion wollen wir dann die Methodik der Implementierung und Verifikation von Routinen anhand von Beispielen illustrieren.

Eiffel ist eine prozedurale Sprache, in der Daten durch Anweisungen verarbeitet werden. Die Form dieser Anweisungen ist nicht anders als in anderen Programmiersprachen. Nur Schleifen sind etwas ungewöhnlich, da hier zusätzliche Zusicherungen eingefügt werden dürfen, die angeben, was sich während eines Schleifendurchlaufs verändert und was nicht. Die folgenden Anweisungsarten gehören zum Sprachumfang von Eiffel

- Zuweisung
- Routinenaufruf (qualified call)
- Zusammengesetzte Anweisungen
- Bedingte Anweisung und Fallunterscheidung (**if**, **inspect**)
- Schleife (**loop**)
- Überprüfung während eines Programmablaufs (**check**)
- Ausnahmebehandlung (**rescue** und **retry**)
- Anweisungen, die nur im DEBUG Modus mitbenutzt werden (**debug**)

4.3.1 Wertzuweisung

Die Wertzuweisung haben wir bereits mehrfach benutzt, da man ohne sie kein sinnvolles Programm schreiben kann. Ein Programm in einer imperativen Sprache beschreibt eine Folge von Zustandstransformationen (oder Speicherveränderungen). Die einzige Möglichkeit, einen Zustand direkt zu verändern, bietet die Zuweisung von Werten an einen Speicherplatz, also an eine Größe, die den Namen eines Speicherplatz kennzeichnet.

Die Wertzuweisung wird immer dann benutzt, wenn ein Wert für spätere Zwecke im aktuellen Programmablauf zwischengelagert werden soll. Der Wert wird unter einem Namen im Speicher eingetragen.Sollen Werte von einem Programmablauf zum nächsten aufgehoben werden, so müssen sie auf einem externen Speicher (Platte, Diskette oder Band) ausgelagert werden. Die Syntax der Wertzuweisung ist einfach

```
entity := Ausdruck
```

wobei `entity` eine Größe (siehe Definition 3.3.3 auf Seite 73) und `Ausdruck` ein Ausdruck (siehe Sektion 4.4) ist. Man beachte jedoch, daß Zuweisungen an formale Routinenargumente verboten sind (dies werden wir im nächsten Abschnitt ausführlicher diskutieren).

[17] Zum Glück tritt der Fall, daß *innerhalb* einer Routine zwei Größen auf dasselbe Objekt verweisen, recht selten auf. Meist ist es so, daß *verschiedene* Routinen Größen benutzen, die auf ein und dasselbe Objekt referenzieren, und diese Routinen hintereinander ausgeführt werden. Dann liegt aber eine klare Trennung vor und die Verifikationsmechanismen sind wieder anwendbar.

[18] Es gibt den Vorschlag, jede Referenzvereinbarung vom Typ `T` in eine Deklaration `expanded ARRAY[expanded T]` zu übersetzen. Eine Erzeugung bedeutet dann ein weiteres Element in diesem Feld und eine Referenz einen Feldindex. Wir werden diesen Weg aber nicht beschreiten, da die Komplexität dieser Beweisführung weit über den Anspruch einer Einführung hinausführt.

Eine Wertzuweisung `entity := Ausdruck` setzt den Wert der Größe `entity` auf den Wert des angegebenen Ausdrucks und läßt alles andere unverändert.[19]

Welche logischen Aussagen kann man nun über die Effekte einer solchen Wertzuweisung treffen? Wir wollen zur Beantwortung dieser Frage von der Nachbedingung ausgehen und versuchen, die notwendigen Vorbedingungen zu beschreiben. Da wir wissen, daß sich bei der Wertzuweisung `entity := Ausdruck` ausschließlich der Wert der Größe `entity` ändern wird, wollen wir die Nachbedingung etwas abstrakter beschreiben. Anstatt

`entity := Ausdruck {post}`

schreiben wir

`entity := Ausdruck {P(entity)}`

wobei wir `P` als Prädikat mit einer einzigen freien Variablen (z.B. `x`) ansehen. Die Aussage `P(entity)` drückt dann aus, daß nach Ausführung der Wertzuweisung diejenige Zusicherung gültig sein soll, die entsteht, wenn wir für `x` den *jetzigen Zustand* von `entity` einsetzen. Der Vorteil dieser Beschreibungsform ist, daß wir nun für die Angabe einer hinreichenden Vorbedingung das Prädikat `P` verwenden dürfen, ohne das dieses noch daran gebunden ist, welchen Zustand `entity` *nach* der Wertzuweisung hat.

Eine hinreichende Vorbedingung – ja sogar die schwächste Vorbedingung – läßt sich mit diesem Mittel nun sehr leicht beschreiben. Soll nach der Zuweisung `entity := Ausdruck` die Aussage `P(entity)` gelten, dann muß vorher `P(Ausdruck)` wahr gewesen sein, da ja `entity` genau den Wert des Ausdrucks zugewiesen bekommt. Anders ausgedrückt: Wenn man weiß, daß ein Prädikat `P` für das Argument `Ausdruck` (also `P(Ausdruck)`) schon vor der Wertzuweisung gilt, dann gilt das Prädikat nach der Wertzuweisung `entity := Ausdruck` für die Größe `entity` (also `P(entity)`). Dieser Zusammenhang wird genau durch die folgende Regel und den zugehörigen Prädikatentransformer beschrieben.

`{P(Ausdruck)} entity := Ausdruck {P(entity)}`

wp`(entity := Ausdruck , P(entity) )` $\equiv$ `P(Ausdruck)`

Abbildung 4.3: *Verifikationsregel und Prädikatentransformer für die Wertzuweisung*

Erfahrungsgemäß ist diese Formulierung für Ungeübte etwas gewöhnungsbedürftig. Wir wollen sie daher an einigen Beispielen illustrieren.

Beispiel 4.3.1 (Verifikation der Wertzuweisung)

Betrachten wir die Wertzuweisung `y:=y+1`*. Wenn nach ihr* `y<b` *gelten soll, dann müssen wir sicherstellen, daß* vor *ihr bereits die Bedingung* `y+1<b` *erfüllt ist:*

`{y+1<b} y:=y+1 {y<b}`

Das sollte auch verständlich sein, denn sei `a` *der Wert von* `y` *vor der Zuweisung und sei* `a+1<b` *wahr, dann hat* `y` *nach der Zuweisung den Wert* `a+1` *– es gilt* `y=a+1`*. Daher ist nach der Zuweisung* `y<b` *gleichbedeutend mit* `a+1<b` *und somit wahr.*

Man kann diesen Satz aber auch durch stures Einsetzen erhalten. Um die Nachbedingung `P(y)` $\equiv$ `y<b` *auszudrücken, wobei* `P` *als freie Variable ein* `x` *haben soll, müssen wir nur jedes Vorkommen von* `y` *durch* `x` *ersetzen. Wir haben also* `P(x)` $\equiv$ `x<b`

Nun ersetzen wir `x` *durch den ausdruck* `y+1` *und erhalten* `P(y+1)` $\equiv$ `y+1<b`

Setzen wir dies nun in die Regel `{P(y+1)} y:=y+1 {P(y)}` *ein, so ergibt sich genau das obige.*

[19] Man beachte, daß Wertzuweisungen nicht vergleichbar sind mit Definitionen oder Gleichungen. Die Wertzuweisung `a:=a+1` ist weder eine sinnvolle Definition, noch kann die Gleichung `a=a+1` jemals erfüllt werden – deshalb auch die Notation mit dem Doppelpunkt. Die Wertzuweisung ist ein *Befehl*, die Größe auf der linken Seite des `:=`-Symbols auf den Wert zu setzen, der sich bei der Berechnung der rechten Seite ergibt.

Der *häufigste Fehler*, der bei der Bestimmung der Vorbedingung einer Wertzuweisung gemacht wird, ist die Umkehr der Richtung. Wie man sich leicht am Beispiel `y=1` und `b=2` klarmachen kann, gilt *nicht*

$\{$ `y<b` $\}$ `y:=y+1` $\{$ `y+1<b` $\}$.

Wir geben ein paar arithmetische Beispiele für die schwächsten Vorbedingungen von Zuweisungen. Diese werden gebildet durch stures Einsetzen und nachträgliches Vereinfachen.

Beispiel 4.3.2 (Weakest Precondition der Wertzuweisung)

			vereinfacht
wp`(n:=n+1, x*n = n!)`	$\equiv$	`x*(n+1) = (n+1)!`	`x = n!`
wp`(x:=x*n, x = n!)`	$\equiv$	`x*n = n!`	`x = (n-1)!`
wp`(x:=x+m, x>0)`	$\equiv$	`x+m > 0`	`x > -m`
wp`(x:=x+m, a>0)`	$\equiv$	`a>0`	*keine Änderung!*
wp`(x:=l, x>m)`	$\equiv$	`l>m`	

4.3.2 Routinenaufruf

Die Verwendung von Routinen haben wir schon im Abschnitt 3.3 angesprochen. Wir wollen dies nun vertiefen und um die zugehörigen Verifikationsregeln ergänzen.

Routinen sind ein wichtiges Strukturierungsmittel bei der Implementierung von Klassen. Sie unterstützen die schrittweise Verfeinerung, da sie ermöglichen, die Programmiersprache durch selbstdefinierte Anweisungen und Ausdrücke an das jeweiligen Problem anzupassen. Dabei müssen wir unterscheiden zwischen *Prozeduren* und *Funktionen*:

- Die Definition einer Prozedur entspricht der Beschreibung einer komplexeren Anweisung. Der Aufruf einer Prozedur ist somit die Durchführung einer *selbstdefinierten Anweisung*. Prozeduraufrufe sind neben der Wertzuweisung die einzige Form einer elementaren Anweisung.[20] Alle anderen Sprachkonstrukte bieten nur die Möglichkeit, *gegebene* Anweisungen zu neuen zusammenzusetzen.

- Im Kontrast dazu beschreibt eine Funktion einen komplexeren Ausdruck im Sinne von Abschnitt 4.4 und nicht etwa eine Anweisung. Ein Funktionsaufruf führt also zur Berechnung eines *selbstdefinierten Ausdrucks*. Dies ermöglicht auch in imperativen Sprachen ein weitgehend funktionales Programmieren, wo dies von der Problemstellung her angebracht ist – wie zum Beispiel bei der Berechnung arithmetischer Funktionen wie der Fakultät oder des größten gemeinsamen Teilers.

Die Verwendung von Routinen, um Probleme zu verfeinern, ist ein grundlegendes Programmierkonzept vieler Programmiersprachen. In Pascal und ähnlichen Sprachen ist es daher erlaubt, innerhalb von Routinen weitere Routinen zu definieren, um diese noch stärker zu strukturieren. In Eiffel (und C) geht das nicht, da Eiffel Routinen als Dienstleistungen von Klassen versteht und nicht etwa als selbständiges Strukturierungskonzept.

In Eiffel *dürfen innerhalb von Routinen keine weiteren Routinen deklariert werden.*

Routinen sind im wesentlichen als Kurzbeschreibung längerer Programmteile anzusehen. Der Name der Routine ist eine Abkürzung für den Anweisungsteil, der beim Aufruf der Routine ausgeführt wird.[21] Durch die Verwendung formaler Argumente wird dieser Anweisungsteil vielseitiger anwendbar, da nun die Routine zur Abkürzung aller Programmstücke benutzt werden kann, die bis auf bestimmte Parameter identisch sind.

[20] Der Aufruf einer Initialisierungsprozedur ist in diesem Zusammenhang eine spezielle Form des Prozeduraufrufs.

[21] Es sei an dieser Stelle noch erwähnt, daß das Routinenkonzept über den Dienstleistungs- und Strukturierungscharakter hinaus noch die Einbindung *externer* Routinen ermöglicht, die in anderen Programmiersprachen implementiert wurden. Dies ermöglicht es, "alte" und zum Teil sehr effiziente Software weiterzuverwenden, anstatt sie erneut in Eiffel codieren zu müssen, und dennoch klar definierte Eiffel-Schnittstellen zur Verfügung zu haben. Eine ausführlichere Beschreibung dieser Möglichkeit findet man in [Meyer, 1992, Kapitel 24].

4.3.2.1 Die Rolle formaler Parameter

In erster Näherung kann man Routinenaufrufe als versteckte Textersetzungen im Programm auffassen: anstelle des Routinennamens wird der Anweisungsteil eingesetzt, in dem wieder jedes Vorkommen eines formalen Parameters durch den aktuell angegebenen Wert ersetzt wird. Die tatsächliche Realisierung eines Routinenaufrufs wird zwar (zugunsten von Effizienz, lokalen Variablen und der Möglichkeit von Rekursion) völlig anders gestaltet, aber diese Sichtweise erklärt die Beschränkungen von Eiffel im Umgang mit formalen Argumenten:

Entwurfsprinzip 4.3.3 (Geschützte formale Argumente)

> *Die formalen Argumente* $\texttt{y}_1,\ldots,\texttt{y}_n$ *einer durch* $\texttt{r(y}_1\texttt{:T}_1,\ldots,\texttt{y}_n\texttt{:T}_n\texttt{)}$ **is do** ... **end** *definierten Routine sind innerhalb des Anweisungsteils von* `r` *geschützt und dürfen nicht direkt verändert werden.*
>
> *Eine direkte Veränderung von* $\texttt{y}_i$ *ist dabei eine Wertzuweisung der Form* $\texttt{y}_i\texttt{:=Ausdruck}$ *oder eine Veränderung des in* $\texttt{y}_i$ *enthaltenen Verweises durch Aufruf von Initialisierungsprozeduren, falls der Typ von* $\texttt{y}_i$ *eine Klasse ist.*

Die Art der Operationen, die eine Routine auf ihren Argumenten ausführen darf, ist also stark eingeschränkt: aktuelle Parameter werden als Werte übergeben, die nicht verändert werden dürfen. Man beachte jedoch, daß dieses Verbot nur für *direkte Veränderungen* gilt. Es ist durchaus erlaubt, ein *Objekt* zu ändern, auf das ein formales Argument $\texttt{y}_i$ verweist, denn hierdurch wird der Wert von $\texttt{y}_i$ selbst ja nicht verändert. Ein Aufruf wie $\texttt{y}_i\texttt{.copy(y}_j\texttt{)}$ ist also durchaus erlaubt, nicht jedoch $\texttt{y}_i\texttt{:=y}_j$.

Die Konsequenz davon ist, daß beim Aufruf einer Routine *beliebige Ausdrücke als formale Parameter* angegeben werden dürfen, solange die Typbedingung eingehalten wird. Es gibt jedoch nur eine Möglichkeit, Berechnungsergebnisse an die aufrufende Stelle zurückzuliefern, nämlich als Resultat einer Funktion. Da dieses Resultat seinerseits von einem (ggf. expanded) Klassentyp sein darf, wird hierdurch die Möglichkeit, mehrere Werte gleichzeitig zu übergeben, nicht eingeschränkt.

Auf diese Art erzielt man eine klare Trennung von Wertberechnungen und Veränderungen von Objekten, was erheblich zur Verständlichkeit von Routinen beiträgt. Prozeduren ("O-Funktionen") sind ausschließlich zur Veränderungen von Objekten da und *können* keine Werte zurückliefern. Funktionen ("V-Funktionen") berechnen Werte (values) und *sollten* Objekte – zumindest ihre äußere Erscheinung – unverändert lassen.[22] Letzteres kann jedoch vom Compiler nicht erzwungen werden und muß daher ein methodischer Hinweis bleiben.

4.3.2.2 Lokale und qualifizierte Aufrufe

Der *Aufruf* einer Prozedur[23] `r` als Anweisung kann entweder lokal oder entfernt geschehen.

- Ein *lokaler* Aufruf bezieht sich auf das aktuelle Exemplar und ist unqualifiziert wie in
  `r` (ohne Argumente) oder
  $\texttt{r(A}_1,\ldots,\texttt{A}_n\texttt{)}$ (mit Argumenten).
- Ein *entfernter* Aufruf wird auf ein Objekt angewandt, das durch einen beliebigen Ausdruck dargestellt werden darf, und ist qualifiziert wie in
  `entity.r` (einfache Qualifizierung, ohne Argumente) oder
  `f(a).r` (Qualifizierung durch Funktion, ohne Argumente) oder
  $\texttt{g(f(a)).h(b,x).z.r(A}_1,\ldots,\texttt{A}_n\texttt{)}$ (mehrstufige geschachtelte Qualifizierung mit Argumenten).

  Ein mehrstufig qualifizierter Aufruf `u.v.r` kann als Abkürzung für `x:=u.v ; x.r` angesehen werden, bei der man sich die Zwischenvariable `x` erspart.

[22] Es ist durchaus möglich, die interne Darstellung eines Objektes zu ändern, ohne daß dies nach außen Wirkung hat. Diese Möglichkeit ist zuweilen auch innerhalb von Funktionen sinnvoll und wird ausführlicher in [Meyer, 1988, Kapitel 7.7] diskutiert.

[23] Funktionsaufrufe sind keine Anweisungen sondern nur Ausdrücke, die auf der rechten Seite einer Wertzuweisung, oder in den aktuellen Parametern und Qualifizierungen eines Prozeduraufrufs vorkommen dürfen – wobei Funktions*aufrufe* allerdings beliebig geschachtelt sein dürfen. Die hier vorgestellten Regeln gelten allerdings für Funktionen und Prozeduren gleichermaßen.

Die Routine `r` muß natürlich eine Routine derjenigen Klasse sein, zu der das durch den qualifizierenden Ausdruck beschriebene Objekt gehört, und für die aufrufende Klasse überhaupt *verfügbar* sein, also nicht durch selektiven Export von der Benutzung ausgeschlossen sein. Bei mehrstufiger Qualifizierung gilt dies für den ganzen Weg von der aufrufenden zur definierenden Klasse.

Eine entfernte Initialisierung der Art `a.!!b` oder `a.!!b.init(Ausdruck`$_1$`,...,Ausdruck`$_n$`)` ist übrigens *nicht erlaubt*. Die Initialisierung von Komponentenobjekte eines Objektes ist – wie die Zuweisung von Werten – ein Privileg, dessen allgemeine Freigabe dem Geheimnisprinzip widersprechen würde. Die Klasse, welche ein feature `b` von einem Klassentyp bereitstellt, sollte auch explizit eine Routine `init_b` bereitstellen, wenn die Initialisierung von `b` durch Kundenklassen erlaubt werden soll.

4.3.2.3 Verifikation von Routinenaufrufen

Die Verifikation von Prozeduraufrufen ist über das Kontraktmodell bereits weitgehend vorbereitet. Wir haben in Definition 4.2.3 auf Seite 141 festgelegt, daß wir eine Routine r als korrekt ansehen, wenn ihr Anweisungsteil $\mathtt{B}_r$ die vereinbarten Vor- und Nachbedingungen einhält, also wenn für alle gültigen Argumente x_r gilt

$$\{\mathtt{pre}_r(x_r)\}\ \mathtt{B}_r\ \{\mathtt{post}_r(x_r)\}$$

In den Vor- und Nachbedingungen dürfen außer den formalen Argumenten nur noch Funktionen, die von der definierenden Klasse erreichbar sind, und – bei Funktionen – die Größe `Result` genannt sein.[24] Da in der Routine die formalen Argumente und das (meist nicht explizit genannte) aktuelle Objekt der definierenden Klasse aber nur Platzhalter für die aktuellen Argumente und das tatsächlich zu bearbeitende Objekt sind, dienen diese in dem Kontrakt ebenso als Platzhalter. Bei der Verifikation eines Aufrufs müssen also nur die formalen Argumente gegen die aktuellen Argumente und das aktuelle Objekt gegen das im Aufruf angegebene Objekt ausgetauscht werden, um einen gültigen Satz über diesen Aufruf zu erhalten.

Um dies präzise auszudrücken, müssen wir die Vor- und Nachbedingungen durch Prädikate beschreiben, deren freie Variablen Platzhalter für die formalen Argumente und das aktuelle Objekt sind. Ist die Prozedur `r` durch

`r(y`$_1$`:T`$_1$`,...,y`$_n$`:T`$_n$`)` **is** do ... **end**

deklariert, so beschreiben wir ihre Vorbedingungen durch

$\mathtt{pre}_r$`(y`$_1$`,...,y`$_n$`, actual)`

und ihre Nachbedingungen durch

$\mathtt{post}_r$`(y`$_1$`,...,y`$_n$`, actual)`,

wobei `actual` die Rolle von `Current` übernimmt, also implizit vor jedem Prozeduraufruf innerhalb des Anweisungsteils von `r` steht, um das Objekt zu kennzeichnen, auf dem gerade gearbeitet wird. Für lokale oder entfernte Aufrufe von `r` gelten dann folgende Sätze:

- $\{\mathtt{pre}_r(\mathtt{A}_1,\ldots,\mathtt{A}_n,\mathtt{Current})\}\ \mathtt{r}(\mathtt{A}_1,\ldots,\mathtt{A}_n)\ \{\mathtt{post}_r(\mathtt{A}_1,\ldots,\mathtt{A}_n,\mathtt{Current})\}$

  Dabei kann `Current` durch Vereinfachungen wieder entfernt werden.
- $\{\mathtt{pre}_r(\mathtt{A}_1,\ldots,\mathtt{A}_n,\mathtt{entity})\}\ \mathtt{entity.r}(\mathtt{A}_1,\ldots,\mathtt{A}_n)\ \{\mathtt{post}_r(\mathtt{A}_1,\ldots,\mathtt{A}_n,\mathtt{entity})\}$

Der Prädikatentransformer ergibt sich entsprechend: kann bei einem Routinenaufruf `entity.r(A`$_1$`,...,A`$_n$`)` die Nachbedingung als $\mathtt{post}_r$`(A`$_1$`,...,A`$_n$`,entity)` ausgedrückt werden, so ist $\mathtt{pre}_r$`(A`$_1$`,...,A`$_n$`,entity)` die zugehörige schwächste Vorbedingung. Je nach Implementierung von `r` wäre prinzipiell eine noch schwächere Vorbedingung möglich. Da diese aber im Kontrakt nicht enthalten ist und sich die Implementierung von `r`– unter Einhaltung des Kontraktes – ändern darf, kann eine schwächere Vorbedingung *nicht garantiert* werden.

[24] Im Idealfall lassen sich die Vor- und Nachbedingungen komplett als Zusicherungen der zugehörigen require und ensure Klausel formulieren. Zuweilen ist es jedoch nötig, diese mithilfe der vollen Prädikatenlogik etwas präziser zu formulieren, als dies mit der Zusicherungssprache von Eiffel – entsprechend der Diskussion in Abschnitt 3.7.4 – möglich ist.

Für eine korrekt implementierte Prozedur r mit formalen Argumenten y_r und abstrakten Vor- und Nachbedingungen pre_r bzw. $post_r$ gilt

$\{pre_r(A_r, entity)\}\ entity.r(A_r)\ \{post_r(A_r, entity)\}$

$wp(entity.r(A_r)\ ,\ post_r(A_r, entity)) \quad \equiv \quad pre_r(A_r, entity)$

Bei unqualifizierten Aufrufen wird Current für entity eingesetzt und der Ausdruck vereinfacht.

Abbildung 4.4: *Verifikationsregeln für Prozeduraufrufe*

In einem Korrektheitsbeweis müssen wir nun versuchen, die Nachbedingung post eines Prozeduraufrufs $entity.r(A_1, \ldots, A_n)$ zu schreiben als $post \equiv post_r(A_1, \ldots, A_n, entity)$. Gelingt dies, so können wir die schwächste Vorbedingung durch Einsetzen in $pre_r(A_1, \ldots, A_n, entity)$ erstellen.

Beispiel 4.3.4 (Verifikation eines Prozeduraufrufs)

In der Klasse ARRAY[INTEGER] *sei die Routine* sort *ohne formale Argumente und ohne Vorbedingung deklariert und habe als Nachbedingung* ∀i:lower..upper-1 . item(i)<item(i+1).

Es sei a:ARRAY[INTEGER] *mit* a.lower=4 *und* a.upper=9. *Als Nachbedingung für den Routinenaufruf sei gefordert* a@4<a@6. *Dies kann durch den Aufruf* a.sort *erreicht werden, denn es gilt*

Programm	**Zusicherungen**	**Prämissen**
	{true}	
a.sort		
	{∀i:4..8. a@i<a@(i+1)}	∀-E *mit i=4*
	{a@4<a@5}	∀-E *mit i=5*
	{a@5<a@6}	a@4<a@5 ∧ a@5<a@6 ⇒ a@4<a@6 *(Arithmetik)*
	{a@4<a@6}	

Man beachte, daß item(i) = Current.item(i) *und* a.item(i)=a@i *ist.*

Bisher haben wir nur über Prozeduraufrufe gesprochen und nicht über Aufrufe von Funktionen. Der Grund hierfür ist, daß Funktionsaufrufe keine Anweisungen sondern Ausdrücke sind, die im Programmtext immer nur in Kombination mit Wertzuweisungen und Prozeduraufrufen vorkommen können. Die Vorgehensweise bei der Verifikation ist aber sehr ähnlich zu der bei Prozeduraufrufen. Allerdings benötigen wir für Vor- und Nachbedingungen einen weiteren Parameter, der für die Größe Result steht.

Wir beschreiben also die Vorbedingungen einer Funktion f mit Argumenten y_f durch $pre_f(y_f, actual)$ und ihre Nachbedingungen durch $post_f(y_f, actual, Result_f)$. Diese führen jedoch *nicht* zu einem eigentlichen Satz des Hoare-Kalküls sondern zu einer logischen Implikation:

Für eine korrekt implementierte Funktion f mit formalen Argumenten y_f und abstrakten Vor- und Nachbedingungen pre_f bzw. $post_f$ gilt

$pre_f(A_f, entity) \ \Rightarrow \ post_f(A_f, entity, entity.f(A_f))$

Bei unqualifizierten Aufrufen wird Current für entity eingesetzt und der Ausdruck vereinfacht.

Abbildung 4.5: *Verifikationsregel für Funktionsaufrufe*

Bei der Verwendung einer Funktion f innerhalb einer Anweisung kann diese Regel dann hinzugenommen werden, um Abschwächungen von Bedingungen innerhalb der Argumentationskette eines Programmbeweises zu *begründen*. Dies lohnt sich aber immer nur dann, wenn hierdurch ein Ausdruck $f(A_f)$ gänzlich eliminiert werden kann. Bei Wertzuweisungen ist dies der Fall, wenn die Nachbedingung von f stärker als die der Anweisung ist, bei Routinenaufrufen, wenn die Nachbedingung eine Gleichung über Result enthält. Im Normalfall erfordert der nutzbringende Einsatz dieser Regel jedoch ein gehöriges Maß an Einfallsreichtum.

Wir verzichten an dieser Stelle auf eine weitere Diskussion und geben stattdessen einige Beispiele.

Beispiel 4.3.5 (Funktionsaufrufe in Anweisungen)

- *Die Funktion* `sqr` *habe als formales Argument* `x`*, als Vorbedingung* `x>5` *und als Nachbedingung* `Result>25`*. Die Nachbedingung der Anweisung* `y:=sqr(y)` *sei* `y>20`*. Dann führt folgende Argumentationskette zu einer Vorbedingung:*

Programm	**Zusicherungen**	**Prämissen**
	{`y>5`}	`y>5` ⇒ `sqr(y)>25`
	{`sqr(y)>25`}	`sqr(y)>25` ⇒ `sqr(y)>20`
	{`sqr(y)>20`}	
`y:=sqr(y)`		
	{`y>20`}	

- *Die Funktion* `dbl_pos` *habe als formales Argument* `x`*, als Vorbedingung* `x>0` *und als Nachbedingung* `Result=2x`*. Die Prozedur* `r` *habe als formales Argument* `x`*, als Vorbedingung* `x>10` *und als Nachbedingung* `b>2x`*. Die Nachbedingung der Anweisung* `a.r(dbl_pos(6))` *sei* `a.b>20`*. Dann führt folgende Argumentationskette zu einer Vorbedingung:*

Programm	**Zusicherungen**	**Prämissen**
	{}	`true` ⇒ `6>0` *(Arithmetik)*
	{`6>0`}	`6>0` ⇒ `dbl_pos(6)=2*6`
	{`dbl_pos(6)=12`}	`dbl_pos(6)=12` ∧ `12>10` ⇒ `dbl_pos(6)>10` *(Arithmetik)*
	{`dbl_pos(6)>10`}	
`a.r(dbl_pos(6))`		
	{`a.b>2*dbl_pos(6)`}	`dbl_pos(6)=12` ∧ `a.b>2*dbl_pos(6)` ⇒ `a.b>24`
	{`a.b>24`}	`a.b>24` ⇒ `a.b>20`
	{`a.b>20`}	

Eine weitere Formalisierung würde versuchen, die angegebenen Prämissen, soweit sie nicht auf reiner Arithmetik beruhen, um den Namen der zugehörigen logischen Regel zu ergänzen. Auf diese Art erhält man einen Beweis, der maschinell geprüft werden kann.

4.3.3 Zusammengesetzte Anweisungen

Folgen von Anweisungen haben wir bereits ausgiebig benutzt. Sie ermöglichen, einen komplexen Ablauf in eine Serie Einzelschritte zu zerlegen, die hintereinander ausgeführt werden. Die Syntax hierfür ist einfach: die Folge wird – durch Semikolon getrennt (optional, aber empfehlenswert) – hintereinandergeschrieben.

> `Anweisung`$_1$ `;...;` `Anweisung`$_n$

Die Bedeutung dieser Folge von Anweisungen ist naheliegend. Zuerst wird `Anweisung`$_1$ ausgeführt, danach `Anweisung`$_2$ usw. bis schließlich `Anweisung`$_n$ ausgeführt ist.

Für eine mathematische Charakterisierung solcher Folgen von Anweisungen ist es hilfreich zu wissen, daß jede einzelne Anweisung wiederum beliebig komplex sein darf, insbesondere also auch eine weiterere Folge von Anweisungen. Dies erlaubt es, jede Folge von Anweisungen als *Komposition zweier Anweisungen* auszudrücken und mit einer einzigen Verifikationsregel für Anweisungsfolgen auszukommen (die bei längeren Folgen dann mehrmals angewandt werden muß). Die Regel ist verhältnismäßig einfach, da sie genau die intuitive Vorstellung des Zusammensetzens von Anweisungen beschreibt.

> Ist `pre` eine Vorbedingung von `p` für eine Instruktion `Anweisung`$_1$ (also eine Vorbedingung dafür, daß `p` nach Ausführung von `Anweisung`$_1$ gilt,) und `p` seinerseits eine Vorbedingung von `post` für `Anweisung`$_2$, dann ist `pre` eine Vorbedingung von `post` für die zusammengesetzte Instruktion.

Man beachte, daß diese Regel – im Gegensatz zu den Regeln für Wertzuweisungen und Prozeduraufrufe – zwei gültige (abgeleitete) Sätze als Prämissen benötigt, um einen neuen gültigen Satz des Hoare-Kalküls abzuleiten.

Die schwächste Vorbedingung **wp** für eine zusammengesetzte Instruktion ergibt sich entsprechend durch eine Komposition der einzelnen schwächsten Vorbedingungen.

$$\frac{\{\texttt{pre}\}\ \texttt{Anweisung}_1\ \{\texttt{p}\},\ \ \{\texttt{p}\}\ \texttt{Anweisung}_2\ \{\texttt{post}\}}{\{\texttt{pre}\}\ \texttt{Anweisung}_1\ ;\ \texttt{Anweisung}_2\ \{\texttt{post}\}}$$

$$\mathbf{wp}(\texttt{Anweisung}_1\texttt{;Anweisung}_2\ \texttt{, post}) \quad \equiv \quad \mathbf{wp}(\texttt{Anweisung}_1\ \texttt{, }\mathbf{wp}(\texttt{Anweisung}_2\texttt{, post}))$$

Abbildung 4.6: *Verifikationsregel und Prädikatentransformer für zusammengesetzte Anweisungen*

Beispiel 4.3.6 (Weakest Precondition zusammengesetzter Anweisungen)

1. $\mathbf{wp}$`(n:=n+1;x:=x*n , x=n!)` $\equiv$
   $\mathbf{wp}$`(n:=n+1 ,` $\mathbf{wp}$`(x:=x*n, x=n!) )` $\equiv$
   $\mathbf{wp}$`(n:=n+1 , x*n = n! )` $\equiv$
   `x=n!` *(Vergleiche Beispiel 4.3.2 auf Seite 146)*

   Das bedeutet, die schwächste Vorbedingung von `x=n!` *für* `n:=n+1; x:=x*n` *wieder die Bedingung* `x=n!` *ist. Diese Bedingung ist als invariant gegenüber der Ausführung der beiden Anweisungen. Diese Tatsache werden wir später bei der Verifikation einer Funktion zur Berechnung der Fakultätsfunktion (siehe Beispiel 4.3.9 auf Seite 161) ausnutzen.*

2. *Es sei* `P` *ein beliebiges Prädikat mit zwei freien Variablen. Dann gilt:*

   $\mathbf{wp}$`(hilf:=a; a:=b; b:=hilf , P(a,b))` $\equiv$
   $\mathbf{wp}$`(hilf:=a,` $\mathbf{wp}$`(a:=b; b:=hilf , P(a,b)))` $\equiv$
   $\mathbf{wp}$`(hilf:=a,` $\mathbf{wp}$`(a:=b,` $\mathbf{wp}$`(b:=hilf, P(a,b) ) ) )` $\equiv$
   $\mathbf{wp}$`(hilf:=a,` $\mathbf{wp}$`(a:=b, P(a,hilf) ) )` $\equiv$
   $\mathbf{wp}$`(hilf:=a, P(b,hilf) )` $\equiv$
   `P(b,a)`

   Da wir für `P` *alles einsetzen dürfen, haben wir allgemein bewiesen, daß* `hilf:=a; a:=b; b:=hilf` *die Werte von* `a` *und* `b` *vertauscht.*

Zum Zweck der besseren Lesbarkeit werden wir in Beispielen ab jetzt immer die Zusicherungen direkt in unser Programm einfügen und dabei das auf Seite 140 vorgestellte Schema benutzen.

$$\begin{array}{ll} & \{\texttt{pre}\} \\ \texttt{Anweisung}_1\texttt{;} & \{\texttt{p}_1\} \\ \texttt{Anweisung}_2\texttt{;} & \{\texttt{p}_2\} \\ \quad\vdots & \\ \texttt{Anweisung}_n & \{\texttt{post}\} \end{array}$$

Diese Notation hat folgende Bedeutung: stehen zwei Zusicherungen $\{\texttt{p}\}$ und $\{\texttt{q}\}$ untereinander, so sind sie nach den Regeln der Prädikatenlogik oder nach anwendungsspezifischen Regeln (z.B. der Arithmetik) ableitbar. Steht zwischen zwei Zusicherungen $\{\texttt{p}\}$ und $\{\texttt{q}\}$ eine Anweisung `Anweisung`, so bedeutet dies, daß $\{\texttt{p}\}$ `Anweisung` $\{\texttt{q}\}$ aufgrund der entsprechenden Ableitungsregel des Hoare-Kalküls gilt. Wir haben diese Notation bereits in füheren Beispielen verwendet, um Sätze des Hoare-Kalküls mit logischen Regeln zu modifizieren. Erst die Verifikationsregel für zusammengesetzte Anweisungen jedoch rechtfertigt es, auch mehrere Anweisungen hintereinander zu verwenden.

Beispiel 4.3.7 (Verifikation zusammengesetzter Anweisungen)

Programm	**Zusicherungen**	**Prämissen**
	$\{\texttt{P(b,a)}\}$	
`hilf:=a;`	$\{\texttt{P(b,hilf)}\}$	
`a:=b;`	$\{\texttt{P(a,hilf)}\}$	
`b:=hilf`	$\{\texttt{P(a,b)}\}$	

4.3.4 Bedingte Anweisung und Fallunterscheidung

Bedingte Anweisungen gehören zur Grundausstattung jeder imperativen Programmiersprache. Sie ermöglichen es, abhängig vom Zustand verschiedene Teilprogramme ausführen zu lassen, was oft schon bei ganz einfachen Problemstellungen notwendig ist.

4.3.4.1 Grundform einer bedingten Anweisung

Will man zum Beispiel den Abstand zwischen zwei Zahlen `x` und `y` berechnen, so muß man immer die Differenz zwischen der größeren Zahl und der kleineren bestimmen. Da zu Beginn aber unklar ist, welche der beiden Zahlen die größere ist, muß zunächst verglichen werden und dann *abhängig vom Ergebnis des Vergleichs* die richtige Differenz gebildet werden. Die syntaktische Beschreibung dieser Vorgehensweise entspricht fast direkt der natürlichsprachigen (englischen) Beschreibung: Wenn `x<y` ist, dann soll als Ergebnis `y-x` berechnet werden und andernfalls `x-y`.

```
  ⋮
abstand(x,y:REAL):REAL -- Abstand der Zahlen x und y
   is
     if x<y then Result := y-x
       else Result := x-y
     end
   end
```

Die Grundform einer bedingten Anweisung entspricht dem obigen Beispiel: wenn eine bestimmte Bedingung erfüllt ist, dann soll eine Anweisung ausgeführt werden, andernfalls eine andere:

if `Bedingung` **then** $\texttt{Anweisung}_1$ **else** $\texttt{Anweisung}_2$ **end**

Als Bedingung sind alle boole'schen Ausdrücke zugelassen, als Anweisung eine beliebig komplexe (auch leere) Folge von Instruktionen. Die Schlüsselworte **if**, **then**, **else** und **end** dienen zur Begrenzung der entsprechenden Programmteile. Die Bedeutung einer solchen bedingten Anweisung liegt auf der Hand:

Ergibt die Auswertung von `Bedingung` den Wert **true**, so wird $\texttt{Anweisung}_1$ ausgeführt, sonst $\texttt{Anweisung}_2$.

Der zweite Teil, einschließlich des Schlüsselwortes **else**, kann auch weggelassen werden, wenn im "else-Fall" keine Anweisungen auszuführen sind.

if `Bedingung` **then** $\texttt{Anweisung}_1$ **end**

In diesem Fall geschieht gar nichts, wenn `Bedingung` den Wert **false** ergibt.

4.3.4.2 Mehrere Alternativen

Nun gibt es auch Fälle, in denen mehr als eine Alternative zu betrachten ist, wie zum Beispiel bei der aus der Analysis bekannten *Signumfunktion*. Diese Ergibt bei Eingabe einer Zahl `x` eine `1`, wenn `x` positiv ist, eine `-1`, wenn `x` negativ ist, und eine `0`, wenn `x` Null ist.

```
  ⋮
signum(x:REAL):REAL -- Signumfunktion
   is
     if x>0 then Result := 1
       else if x<0 then Result := -1
             else Result := 0
            end
     end
   end
```

Derartig verschachtelte **if**'s sind eigentlich nicht notwendig, da wir drei verschiedene Alternativen betrachten,

die alle dieselbe Variable x betreffen. Aus diesem Grunde gibt es in Eiffel die Möglichkeit, Alternativen mit **elseif** nebeneinander zu stellen, statt sie zu schachteln.

```
  ⋮
signum(x:REAL):REAL -- Signumfunktion
   is
     if x>0 then Result := 1
       elseif x<0 then Result := -1
       else Result := 0
     end
   end
```

Abbildung 4.7 beschreibt die allgemeine syntaktische Form für bedingte Anweisungen.

```
if Bedingung1 then Anweisung1
  elseif Bedingung2 then Anweisung2
  elseif Bedingung3 then Anweisung3
    ⋮
  else Anweisungelse
end
```

Abbildung 4.7: *Syntax der bedingten Anweisung*

Die Bedeutung dieser Form liegt auf der Hand: sie ist eine Abkürzung für die folgende geschachtelte Anweisung

```
if Bedingung1 then Anweisung1
  else if Bedingung2 then Anweisung2
         else if Bedingung3 then Anweisung3
                                 ⋮
                               else Anweisungelse
                             end
                ⋮
              end
       end
end
```

Ist $\texttt{Bedingung}_1$ erfüllt, so wird $\texttt{Anweisung}_1$ ausgeführt; andernfalls wird $\texttt{Bedingung}_2$ ausgewertet und der Befehl $\texttt{Anweisung}_2$ ausgeführt, wenn sie **true** ergibt; andernfalls wird $\texttt{Bedingung}_3$ ausgewertet usw.

4.3.4.3 Fallunterscheidung

Eine weitere Form der bedingten Anweisung erweist sich als sinnvoll, wenn man das folgende Beispiel betrachtet. In dem Bibliothekenverwaltungsprogramm, das wir im Abschnitt 4.1.6 strukturiert hatten, soll ein Benutzer zu Beginn der Bearbeitung eine Transaktionsart auswählen, welche den weiteren Ablauf bestimmt. Eine mögliche Realisierung wäre dabei wie folgt:

```
-- DEKLARATIONEN
--   transakt:TRANSAKTION        (Void)
--   home_bibliothek:BIBLIOTHEK (bereits bestimmt)
--   Ausleihe, Verlängern, Rückgabe, Entnahme, Hinzufügen: CHARACTER (Konstante)
--   erfragte_transaktionsart:CHARACTER (interaktiv erfragt)
--   sitzung_beenden:BOOLEAN (false)
  ⋮
if      erfragte_transaktionsart = Ausleihe   then !AUSLEIHE!transakt.init(home_bibliothek)
  elseif erfragte_transaktionsart = Verlängern then !VERLÄNGERN!transakt.init(home_bibliothek)
  elseif erfragte_transaktionsart = Rückgabe   then !RÜCKGABE!transakt.init(home_bibliothek)
  elseif erfragte_transaktionsart = Entnahme   then !ENTNAHME!transakt.init(home_bibliothek)
  elseif erfragte_transaktionsart = Hinzufügen then !HINZUFÜGEN!transakt.init(home_bibliothek)
  else  sitzung_beenden := true
end
```

Diese Aufzählung ist durch die ständige Wiederholung von `erfragte_transaktionsart = ...` unnötig aufwendig und führt bei der Durchführung zu unnötigen Laufzeiten, da bei der Auswertung von `antwort` immer wieder in den Speicher gegriffen wird. Die meisten Sprachen einschließlich Eiffel bieten daher einen vereinfachten Mechnismus an.

```
⋮
inspect erfragte_transaktionsart
  when Ausleihe   then !AUSLEIHE!transakt.init(home_bibliothek)
  when Verlängern then !VERLÄNGERN!transakt.init(home_bibliothek)
  when Rückgabe   then !RÜCKGABE!transakt.init(home_bibliothek)
  when Entnahme   then !ENTNAHME!transakt.init(home_bibliothek)
  when Hinzufügen then !HINZUFÜGEN!transakt.init(home_bibliothek)
  else sitzung_beenden := true
end
```

Im Unterschied zur bedingten Anweisung mit **if** wird hier nicht immer wieder ein *boole'scher* Wert berechnet, der dann entscheidet, welche Anweisung durchgeführt wird, sondern einmal ein Ausdruck, der einen `INTEGER`- oder `CHARACTER`-Wert ergibt. Dieser Wert wird mit allen Konstanten der **when**-Teile verglichen. Ist der Wert ungleich allen Konstanten, dann wird der **else**-Teil ausgeführt, ansonsten die Anweisung, in deren **when**-Teil der Vergleich erfolgreich war. Abbildung 4.8 beschreibt die allgemeine Form der Fallunterscheidung.

```
inspect Ausdruck
   when Auswahl_1 then Anweisung_1
   when Auswahl_2 then Anweisung_2
   when Auswahl_3 then Anweisung_3
      ⋮
   else Anweisung_else
end
```

Abbildung 4.8: *Syntax der Fallunterscheidung*

Dabei muß `Ausdruck` vom Typ `INTEGER` oder `CHARACTER` sein (*andere Werte sind nicht gestattet*), $\texttt{Auswahl}_i$ eine Liste von Konstanten oder Intervallen wie `4..8` oder `'a'..'z'`. Die durch die Auswahlmöglichkeiten beschriebenen Bereiche dürfen sich *nicht überlappen* und müssen vom Typ des Ausdrucks sein. Wenn der Ausdruck jedoch z.B. ein Lesebefehl war, kann es durchaus sein, daß sowohl `INTEGER` als auch `CHARACTER`-Werte in den Auswahlmöglichkeiten vorkommen. Es ist zulässig, daß der **else**-Fall ausgelassen wird.

Die Bedeutung dieses Konstruktes ist die folgende:

> Zunächst wird der Wert des hinter **inspect** genannten `Ausdruck` berechnet. Aufgrund der Bedingungen an die in den Auswahlmöglichkeiten beschriebenen Bereiche gibt es dann höchstens einen Bereich, zu dem dieser Wert gehört, weil er entweder direkt aufgezählt wird oder in dem entsprechenden Intervall liegt. Ist $\texttt{Auswahl}_i$ die Beschreibung dieses Bereiches, so wird $\texttt{Anweisung}_i$ ausgeführt.
>
> Gibt es keinen derartigen Bereich, so wird $\texttt{Anweisung}_{\texttt{else}}$ ausgeführt, falls ein **else**-Fall vorhanden ist, und ansonsten eine *Ausnahme* (eine Fehlersituation, siehe Abschnitt 4.3.7) ausgelöst.

Es sei an dieser Stelle vermerkt, daß Programmierer aus der Denkwelt von Pascal dazu tendieren, die Fallunterscheidung in Situationen einsetzen zu wollen, wo es konzeptionell nicht angebracht ist. Dies ist zum Beispiel der Fall, wenn eine Operation vom konkreten Typ des Objektes abhängen soll, das gerade betrachtet wird. In diesem Fall ist eine Fallunterscheidung nach Objekttyp (in Eiffel sowieso undurchführbar) nicht der richtige Weg, da in der Fallunterscheidung Kenntnisse über das gesamte System und alle verfügbare Klassen verankert werden müssten, was spätere Erweiterungen so gut wie unmöglich macht. Vererbung und dynamisches Binden sind in diesem Fall erheblich besser geeignet, da die Operation offensichtlich eine spezielle Dienstleistung der Klasse des Objektes ist. Eine etwas tiefergehende Diskussion der sinnvollen Verwendung von Fallunterscheidungen findet man in [Meyer, 1992, Kapitel 14.6].

4.3.4.4 Verifikation

Die Verifikationsregel für bedingte Anweisungen ist in der einfachen Form ohne **elseif** noch relativ leicht zu verstehen. Um zu beweisen, daß $\{\texttt{pre}\}$ **if** Bedingung **then** $\texttt{Anweisung}_1$ **else** $\texttt{Anweisung}_2$ **end** $\{\texttt{post}\}$ gilt, muß man sich beide Anweisungen einzeln ansehen. $\texttt{Anweisung}_1$ wird ausgeführt, wenn Bedingung gilt (und natürlich auch pre), $\texttt{Anweisung}_2$ wird ausgeführt, wenn Bedingung *nicht* gilt, also $\neg$Bedingung (und pre) wahr ist. Beide Voraussetzungen müssen jeweils die Gültigkeit der Nachbedingung post zur Folge haben.

$$\frac{\{\texttt{pre} \wedge \texttt{Bedingung}\}\ \texttt{Anweisung}_1\ \{\texttt{post}\}, \quad \{\texttt{pre} \wedge \neg\texttt{Bedingung}\}\ \texttt{Anweisung}_2\ \{\texttt{post}\}}{\{\texttt{pre}\}\ \textbf{if}\ \texttt{Bedingung}\ \textbf{then}\ \texttt{Anweisung}_1\ \textbf{else}\ \texttt{Anweisung}_2\ \textbf{end}\ \{\texttt{post}\}}$$

Der Prädikatentransformer ergibt sich aus dem folgenden Zusatzargument. Die schwächste Vorbedingung für post hängt davon ab, ob vor Ausführung der Anweisung Bedingung gilt oder nicht. Im ersten Fall ist dies dann die schwächste Vorbedingung bezüglich $\texttt{Anweisung}_1$ andernfalls diejenige bezüglich $\texttt{Anweisung}_2$. Dies läßt sich durch logische Implikationen leicht ausdrücken

$$\begin{aligned}&\mathbf{wp}(\texttt{Bedingung}\ \textbf{then}\ \texttt{Anweisung}_1\ \textbf{else}\ \texttt{Anweisung}_2\ \textbf{end}, \texttt{post})\\ \equiv\ &\texttt{Bedingung} \Rightarrow \mathbf{wp}(\texttt{Anweisung}_1, \texttt{post}) \wedge \neg\texttt{Bedingung} \Rightarrow \mathbf{wp}(\texttt{Anweisung}_2, \texttt{post})\end{aligned}$$ [25]

Läßt man den **else** Zweig aus, so vereinfacht sich die Regel, da die schwächste Vorbedingung von post unter einer leeren Anweisung (bei der gar nichts geschieht) natürlich wieder post selbst ist. Abbildung 4.9 faßt die Verifikationsregeln und Prädikatentransformer bedingter Anweisungen zusammen.

$$\frac{\{\texttt{pre} \wedge \texttt{Bedingung}\}\ \texttt{Anweisung}_1\ \{\texttt{post}\}, \quad \{\texttt{pre} \wedge \neg\texttt{Bedingung}\}\ \texttt{Anweisung}_2\ \{\texttt{post}\}}{\{\texttt{pre}\}\ \textbf{if}\ \texttt{Bedingung}\ \textbf{then}\ \texttt{Anweisung}_1\ \textbf{else}\ \texttt{Anweisung}_2\ \textbf{end}\ \{\texttt{post}\}}$$

$$\frac{\{\texttt{pre} \wedge \texttt{Bedingung}\}\ \texttt{Anweisung}_1\ \{\texttt{post}\}, \quad \texttt{pre} \wedge \neg\texttt{Bedingung} \Rightarrow \texttt{post}}{\{\texttt{pre}\}\ \textbf{if}\ \texttt{Bedingung}\ \textbf{then}\ \texttt{Anweisung}_1\ \textbf{end}\ \{\texttt{post}\}}$$

$$\begin{aligned}&\mathbf{wp}(\texttt{Bedingung}\ \textbf{then}\ \texttt{Anweisung}_1\ \textbf{else}\ \texttt{Anweisung}_2\ \textbf{end}, \texttt{post})\\ \equiv\ &\texttt{Bedingung} \Rightarrow \mathbf{wp}(\texttt{Anweisung}_1, \texttt{post}) \wedge \neg\texttt{Bedingung} \Rightarrow \mathbf{wp}(\texttt{Anweisung}_2, \texttt{post})\end{aligned}$$

$$\begin{aligned}&\mathbf{wp}(\texttt{Bedingung}\ \textbf{then}\ \texttt{Anweisung}_1\ \textbf{end}, \texttt{post})\\ \equiv\ &\texttt{Bedingung} \Rightarrow \mathbf{wp}(\texttt{Anweisung}_1, \texttt{post}) \wedge \neg\texttt{Bedingung} \Rightarrow \texttt{post}\end{aligned}$$

Abbildung 4.9: *Verifikationsregeln und Prädikatentransformer für bedingte Anweisungen*

Etwas komplizierter wirkt die Verifikationsregel für die allgemeine bedingte Anweisung mit **elseif** (siehe Abbildung 4.7). Dies liegt daran, daß jedes **elseif** eine weitere Verschachtelungsstufe eröffnet und man die obige Verifikationsregel mehrfach anwenden muß. Eine Regel für die allgemeine bedingte Anweisung enthält also für jeden **elseif**-Zweig eine weitere Prämisse. Im Zweig **elseif** $\texttt{Bedingung}_i$ **then** $\texttt{Anweisung}_i$ müssen alle bisher genannten Bedingungen falsch sein und $\texttt{Bedingung}_i$ wahr. Alles andere ist genauso wie bisher.

$$\frac{\begin{array}{l}\{\texttt{pre} \wedge \texttt{Bedingung}_1\}\ \texttt{Anweisung}_1\ \{\texttt{post}\},\\ \{\texttt{pre} \wedge \neg\texttt{Bedingung}_1 \wedge \texttt{Bedingung}_2\}\ \texttt{Anweisung}_2\ \{\texttt{post}\},\\ \quad\vdots\\ \{\texttt{pre} \wedge \neg\texttt{Bedingung}_1 \wedge \ldots \wedge \neg\texttt{Bedingung}_n\}\ \texttt{Anweisung}_{\texttt{else}}\ \{\texttt{post}\}\end{array}}{\{\texttt{pre}\}\ \textbf{if}\ \texttt{Bedingung}_1\ \textbf{then}\ \texttt{Anweisung}_1\ \textbf{elseif}\ \ldots\ \textbf{else}\ \texttt{Anweisung}_{\texttt{else}}\ \textbf{end}\ \{\texttt{post}\}}$$

Die allgemeine Fallunterscheidung läßt sich, wie oben illustriert, auf die allgemeine bedingte Anweisung zurückführen. Da die sich hieraus ergebende Verifikationsregel konzeptionell nichts neues bringt, aber noch einmal etwas komplizierter wird, verzichten wir auf die Angabe einer eigenen Regel.

[25] Logische Umformungen führen auch zu einer anderen häufig benutzten Beschreibungsform. Es reicht, daß entweder Bedingung wahr ist und die Vorbedingung bezüglich $\texttt{Anweisung}_1$ oder daß $\neg$Bedingung gilt und die Vorbedingung bezüglich $\texttt{Anweisung}_2$:

$$\begin{aligned}&\mathbf{wp}(\texttt{Bedingung}\ \textbf{then}\ \texttt{Anweisung}_1\ \textbf{else}\ \texttt{Anweisung}_2\ \textbf{end}\ , \texttt{post})\\ \equiv\ &\texttt{Bedingung} \wedge \mathbf{wp}(\texttt{Anweisung}_1, \texttt{post}) \vee \neg\texttt{Bedingung} \wedge \mathbf{wp}(\texttt{Anweisung}_2, \texttt{post})\end{aligned}$$

Beispiel 4.3.8 (Verifikation bedingter Anweisungen)

Wir wollen nun gleichzeitig eine Implementierung und einen Korrektheitsbeweis für die Fakultätsfunktion entwickeln. Die Endbedingung des zu entwickelnden Programms sei `korrekt ⇒ Result=arg!`*. Um dies unkompliziert zu realisieren, setzen wir voraus, daß* `korrekt:BOOLEAN` *ein Attribut der umgebenden Klasse sei, welches nach Ausführung der Fakultätsfunktion angibt, ob der berechnete Wert benutzt werden darf.*

Die Entwicklung folgt dem Top-Down Ansatz: wir geben eine grobe Struktur vor, deren feinere Details dann separat entwickelt werden. Der erste Schritt der Programmentwicklung legt zunächst nur die Rahmenbedingungen fest, nämlich ob die Fakultät überhaupt korrekt berechnet werden kann:

Programm	**Zusicherungen**
`fakultät(arg:INTEGER):INTEGER`	
is `-- Berechnung von arg!`	
if `arg > 0`	
then	{`arg>0`}
fakultäts-berechnung	
	{`Result=arg!`}
`korrekt:=true`	
else	
`korrekt:=false`	
end	
end	{`korrekt ⇒ Result=arg!`}

Wir haben dabei für das Programmstück fakultäts-berechnung*, dessen Verfeinerung erst im Beispiel 4.3.9 auf Seite 161 erfolgen wird, nur die Spezifikation – also die Vor- und Nachbedingung – festgelegt:*

{`arg>0`} fakultäts-berechnung {`Result=arg!`}

Wir zeigen nun, daß `true` *die schwächste Vorbedingung von* `korrekt ⇒ Result=arg!` *für die Funktion* `fakultät` *ist, d.h. daß die Funktion* `fakultät` *– vorausgesetzt wir können* fakultäts-berechnung *wie gewünscht realisieren – keinerlei Vorbedingungen benötigt, also sehr robust ist.*[26]

	Programm	**Zusicherungen**	**Prämissen**
1	`fakultät(arg:INTEGER):INTEGER`		
2	**is** `-- Berechnung von arg!`	{`true`}	pre
3	**if** `arg > 0`		
4	**then**		pre∧Bedingung
5		{`true∧arg>0`}	K1,K11
6		{`arg>0`}	
7	fakultäts-berechnung		Spezifikation
8		{`Result=arg!`}	K10,K1
9		{`false ∨ Result=arg!`}	
10		{`¬true ∨ Result=arg!`}	K8
11		{`true ⇒ Result=arg!`}	
12	`korrekt:=true`		Wertzuweisung
13		{`korrekt ⇒ Result=arg!`}	post
14			
15	**else**		pre∧¬Bedingung
16		{`true∧¬(arg>0)`}	∧-E
17		{`true`}	K10,K1
18		{`true ∨ Result=arg!`}	
19		{`¬false ∨ Result=arg!`}	K8
20		{`false ⇒ Result=arg!`}	
21	`korrekt:=false`		Wertzuweisung
22		{`korrekt ⇒ Result=arg!`}	post
23	**end**		
24	**end**	{`korrekt ⇒ Result=arg!`}	post

[26] Voll robust wird das Programm erst durch Absicherung gegen Überlauf in der Multiplikation, also gegen Eigenschaften der Implementierung auf einer bestimmten Hardware.

Dieser Beweis ist in der folgenden Form zu lesen:

Man beginnt grundsätzlich mit der Nachbedingung des Programmstücks (24). Entsprechend der Verifikationsregel für bedingte Anweisungen ist diese Nachbedingung auch die Nachbedingung der einzelnen Zweige (13) und (22).

Betrachten wir nun den **else***-Zweig, dann ergibt uns die Prädikatentransformation durch* `korrekt:= false` *die Zusicherung (20). Diese ist – wegen der angegebenen Konversionsregeln aus Abbildung 2.9 (Seite 39) äquivalent zu (19), (18) und (17) und wird mit den Ableitungsregeln aus Abbildung 2.10 41) auf (16) verstärkt. (16) ist die von der* **if***-Regel verlangte Vorbedingung.*

Genauso erfolgt der Beweis des **then***-Zweigs. Über die Prädikatentransformer und Konversionsregeln erhält man aus (13) die Zusicherung (3), die on der* **if***-Regel verlangte Vorbedingung.*

Damit haben wir die Prämissen der **if***-Regel erfüllt und können daraus schließen, daß* `true` *die Vorbedingung von* `korrekt ⇒ Result=arg!` *ist.*

4.3.5 Wiederholung

Routinen, die wir mit den bisher vorgestellten Mechanismen aufbauen können, lohnen sich kaum zu programmieren, da der komplette Berechnungsablauf durch *einzelne* Anweisungen beschrieben werden muß. Ein wirklicher Gewinn entsteht erst dadurch, wenn wir Berechnungen mehrfach ausführen können, ohne sie immer wieder hinschreiben zu müssen. Diese Möglichkeit der Wiederholung von Anweisungen bietet die Schleife.[27]

Betrachten wir dazu erneut unser Bibliothekenverwaltungsprogramm. Das Menü, mit dem ein Benutzer zu Beginn der Bearbeitung eine Transaktionsart auswählen kann, sollte sinnvollerweise nach der Abarbeitung der Transaktion erneut auf dem Bildschirm erscheinen, bis der Benutzer schließlich eine Beendigung der Sitzung auslöst. Dieses Verhalten können wir (als Teil der Initialisierungsprozedur von `BIB_VERWALT`) erzielen, wenn wir unser Programmstück von Seite 154 wie folgt erweitern.

```
from sitzung_beenden := false
  until sitzung_beenden = true
  loop  -- Interaktionsmenü verarbeiten
        inspect benutzerantwort_im_interaktionsmenü
          when Ausleihe   then !AUSLEIHE!transakt.init(home_bibliothek)
          when Verlängern then !VERLÄNGERN!transakt.init(home_bibliothek)
          when Rückgabe   then !RÜCKGABE!transakt.init(home_bibliothek)
          when Entnahme   then !ENTNAHME!transakt.init(home_bibliothek)
          when Hinzufügen then !HINZUFÜGEN!transakt.init(home_bibliothek)
          else sitzung_beenden := true
        end
end
```

Dieses Programmstück besagt, daß die Verarbeitung des Interaktionsmenüs mittels Fallunterscheidung (die Anweisungen zwischen **loop** und **end**) solange wiederholt werden soll, bis `sitzung_beenden=true` erfüllt ist. Dabei wird zu Beginn der Schleife – vor dem ersten Test `sitzung_beenden=true` – die Anweisung `sitzung_beenden:=false` ausgeführt.

Diese Schleifenkonstruktion von Eiffel ist eine Mischung aus Wiederholungs- und Zählschleifen, die in anderen Programmiersprachen oft getrennt auftreten. Die Begründung hierfür ist der Wunsch, die Sprache möglichst einfach zu halten. Die allgemeine Form einer Eiffel-Schleife ist daher die folgende

```
from Initialanweisung
  until Abbruchbedingung
  loop  Schleifenanweisung
end
```

[27] Den gleichen Effekt kann man natürlich auch durch einen rekursiven Aufruf von Routinen erzielen, was eleganter, für Nichtmathematiker aber meist viel unüberschaubarer ist als eine Schleife.

Die Bedeutung einer solchen Schleife ist die folgende: zunächst werden die Initialanweisung(en) im **from**-Teil ausgeführt[28] und anschließend die Schleifenanweisung(en) zwischen **loop** und **end** solange wiederholt, bis die im **until**-Teil genannte Abbruchbedingung erfüllt ist. Ist diese bereits nach der Initialanweisung erfüllt, dann wird die Schleifenanweisung überhaupt nicht ausgeführt. Die Anweisungen dürfen dabei wie immer beliebig komplex oder auch leer sein (was aber nur im Falle der Initialanweisung Sinn macht).

Mit dem Schleifenkonstrukt sind wir nun in der Lage, das noch fehlende Programmstück aus Beispiel 4.3.8 – die Implementierung von "fakultäts-berechnung" – anzugeben

```
from Result := 1 ; n := 1
  until n>=arg
  loop
     n := n+1;
     Result := Result*n
end
```

Dabei muß die lokale Variable `n` zuvor in der Funktion `fakultät` mit **local** `n:INTEGER` vereinbart werden.

Schleifen sind (neben Routinen) bei der Implementierung das mächtigste Handwerkzeug, da sie erlauben, bestimmte Anweisungsfolgen beliebig oft durchführen zu lassen. Andererseits machen Schleifen es aber auch erheblich schwerer, das Verhalten eines Programmes präzise zu beschreiben, wenn man nicht extrem sorgfältig programmiert. Dadurch, daß die Anzahl der Wiederholungen der Schleifenanweisungen erst während der Laufzeit – nämlich nach Beendigung – der Schleife bestimmt werden kann, ist das Verhalten von Schleifen weniger geradlinig als das von zusammengesetzten oder bedingten Anweisungen. Es ist nicht einmal gesichert, ob die Schleife überhaupt jemals beendet wird. Deshalb muß man bei der Verifikation von Programmen, die Schleifen enthalten, zwei Fragen beantworten.

Bricht die Schleife jemals (geplant) ab? (*Terminierung*)
Wenn sie endet, erfüllt sie ihre Nachbedingung? (*Partielle Korrektheit*)

Für den Beweis der Terminierung hat sich folgender Gedankengang bewährt. Wenn eine Schleife nach einer Anzahl von Schritten terminiert, dann wird bei jedem Schleifendurchlauf die Anzahl der noch ausstehenden Schleifendurchläufe um eins kleiner. Im Prinzip läßt sich also die Anzahl der noch ausstehenden Schleifendurchläufe durch einen Integer-Ausdruck beschreiben, der von allen möglichen in die Schleife verwickelten Parametern abhängt und bei jedem Schleifendurchlauf geringer wird.

Den genauen Ausdruck zu finden ist jedoch im allgemeinen zu schwierig. Stattdessen behilft man sich mit einer Näherungslösung. Es genügt ja, einen anderen, etwas groberen Zusammenhang durch einen Integer-Ausdruck zu beschreiben, der – wie die Anzahl der noch ausstehenden Schleifendurchläufe – stets größer gleich Null ist, wenn die Abbruchbedingung nicht erfüllt ist, und bei jedem Schleifendurchgang mindestens um eins kleiner wird. Wenn man diesen Ausdruck – die sogenannte *Variante* der Schleife – finden und die obengenannten Eigenschaften beweisen kann, dann ist gezeigt, daß die Schleife abbrechen *muß* .[29]

Für die partielle Korrektheit gilt folgende Überlegung. Will man über das Ergebnis einer Schleife nach der Terminierung etwas aussagen, so muß man eine Programmeigenschaft formulieren, die nur von den in die Schleife verwickelten Variablen abhängt und nach jedem Schleifendurchgang gültig ist. Da aber die Anzahl der Schleifendurchgänge im voraus nicht feststeht, muß man die Gültigkeit dieser Eigenschaft durch eine Art Induktion beweisen. Dies aber geht nur, wenn man die Eigenschaft so formuliert, daß sie vor dem Beginn der eigentlichen Schleife wahr ist (Induktionsanfang) und nach einem einmaligen Schleifendurchlauf immer dann wahr ist, wenn sie vorher gültig gewesen war (Induktionsschritt). Man sagt daher, daß die Programmeigenschaft gegenüber einer Schleifendurchführung *invariant* ist, da sie sich bei dem Durchgang nicht ändert.

[28] Man hätte beim Entwurf von Eiffel natürlich auch auf den **from**-Teil verzichten können, da die Initialanweisung ja einfach *vor* der Schleife genannt werden könnte. Die Verwendung des **from** sorgt jedoch dafür, daß Anweisungen, deren Sinn ausschließlich in der Bereitstellung gewisser Anfangsbedingungen für eine Schleife liegt, auch dieser Schleife zugeordnet werden sollten.

[29] Das mathematische Argument, das man üblicherweise mit Wohlordnungen beweist ist grob gesagt das folgende: Ein endlicher Wert, der bei jedem Schleifendurchgang mindestens um eins kleiner wird, wird irgendwann einmal negativ. Dies aber ist nur möglich, wenn die Abbruchbedingung erfüllt ist, da er andernfalls nach Voraussetzung größer gleich Null sein muß.

Im allgemeinen ist es sehr schwierig, die Variante oder die Invariante einer Schleife zu *finden*. Der Beweis, daß ein bestimmter Ausdruck tatsächlich eine Variante oder Invariante der Schleife ist, ist dagegen oft sehr einfach, wenn man den Ausdruck erst einmal gefunden hat.

Sicherlich ist derjenige, der sich einen Algorithmus ausgedacht hat, am ehesten dazu in der Lage, die Invariante und Variante einer Schleife anzugeben, denn die Schleife soll ja eine bestimmte Idee zur Lösung eines Problems wiederspiegeln. Es wäre daher sinnvoll, diese Idee *während der Verfeinerung zu fixieren*, indem man Invariante und Variante aufstellt, *bevor* die Schleife ausprogrammiert wird. Diese Vorgehensweise, bei der man sich schon während der Implementierung Gedanken über die Korrektheitsgarantie macht, fördert eine fehlerfreie Entwicklung von Programmen und erleichtert den späteren Korrektheitsbeweis. Zudem erspart es dem Programmierer, die bei "konventioneller Vorgehensweise" unweigerlich auftretenden Fehler mühsam aufspüren und eliminieren zu müssen.

Die Sprache Eiffel unterstützt eine derartig systematische Programmentwicklung, indem sie erlaubt, Variante und Invariante als Zusicherungen – gekennzeichnet durch Schlüsselworte **variant** und **invariant** – mit in das Programm aufzunehmen und somit die wichtigste Programm- und Beweisidee zu dokumentieren.

Bei unserer Fakultätsberechnung zum Beispiel war die Grundüberlegung, die Variable `n` schrittweise von unten an `arg` anzunähern, bis `n=arg` ist, also `arg-n` zu Null wird. Dabei soll nach jedem Schritt die Größe `Result` den Wert `n!` enthalten, zum Schluß also den gesuchten Wert `arg!`. Formuliert man diese Idee als Variante und Invariante, so ergibt sich `arg-n` als Variante und `n<=arg` $\wedge$ `Result=n!` als Invariante. Da der arithmetische Ausdruck `n!` aber kein Bestandteil von Eiffel ist (sonst wäre ja auch das ganze Programm überflüssig) muß der Teil `Result=n!` in einen Kommentar verlagert werden. Auf diese Art wird eine wichtige Programmidee aufgehoben, auch wenn sie nicht als Eiffel-Zusicherung formulierbar ist.

```
from Result := 1 ; n := 1
  invariant n<=arg    -- und Result=n!
  variant arg-n
  until n>=arg
  loop
     n := n+1;
     Result := Result*n
end
```

Es ist einsichtig, daß `arg-n` stets kleiner, aber nie negativ wird. Da wir im Beispiel 4.3.9 detailliert beweisen werden, daß `n<=arg` $\wedge$ `Result=n!` tatsächlich eine Invariante ist, wissen wir, daß nach der Beendigung der Schleife Invariante und Abbruchbedingung gelten, also `n<=arg` $\wedge$ `Result=n!` $\wedge$ `n>=arg`. Dies vereinfacht sich zu `n=arg` $\wedge$ `Result=arg!`. Also haben wir die gewünschte Nachbedingung von "fakultäts-berechnung" gezeigt.

```
from Initialanweisung
   invariant Invariante
   variant Variante
   until Abbruchbedingung
   loop  Schleifenanweisung
 end
```

Abbildung 4.10: *Vollständige Syntax der Schleifen (mit Invarianten)*

Abbildung 4.10 beschreibt die allgemeine Form von Schleifen mit Invarianten. Die **invariant**- und **variant**-Teile sind dabei optional, dürfen also fehlen, während alle anderen Bestandteile vorkommen *müssen*.

Die Bedeutung dieser Schleife ist auch beim Vorkommen von Varianten und Invarianten dieselbe wie die bereits erklärte. Zusätzlich werden jedoch auf Wunsch – analog zu anderen Zusicherungen – die Eiffel-Bestandteile (also die Nicht-Kommentare) des **invariant**-Teils und des **variant**-Teils bei jedem Schleifendurchgang geprüft. Man kann also die Variante und die Invariante auch zum Testen verwenden. Ihre wichtigste Bedeutung ist aber ihr Kommentarcharakter, der dem Leser des Programms beim Code Review eine Hilfe zur Prüfung und zum Führen von Korrektheitsbeweisen gibt.

Rein hypothetisch könnte man bei der Verifikation einer Schleife auch ohne Invariante und Variante auskommen, wenn man beschreiben könnte, wie oft die Schleife durchlaufen wird. Wäre bekannt, daß die Schleife

from `Init` **until** `Abbruch` **loop** `Anweisung` **end**

nach genau n Durchgängen terminiert, dann könnte die Vorbedingung `pre` der Schleife aus der Nachbedingung `post` wie folgt berechnet werden:

$$\mathbf{wp}(\underbrace{\texttt{Anweisung;Anweisung; ...;Anweisung}}_{n-mal}\texttt{ , post})$$

Mathematisch kann die Unkenntnis über die konkrete Anzahl n der Durchgänge bis zur Terminierung durch einen Existenzquantor ersetzt werden und es ist tatsächlich möglich, einen Prädikatentransformer ohne Kenntnis dieser Anzahl zu formulieren. Die schwächste Vorbedingung für die Nachbedingung `post` unter der obigen Schleife ist daher eine Formulierung der folgenden Aussage:

> Es gibt eine Anzahl n, so daß gilt
> Für alle Werte $k \in \{1..n-1\}$ ist nach k-maliger Durchführung von `Anweisung` ist `Abbruch` nicht erfüllt und nach n-maliger Durchführung von `Anweisung` ist `Abbruch` erfüllt
> und es gilt die schwächste Vorbedingung von `post` bezüglich der n-fachen Ausführung von `Anweisung`.

Für eine Präzisierung dieser Aussage muß man nun noch den Begriff der n-malige Durchführung einer Anweisung durch ein Prädikat ersetzen. Wir führen zu diesem Zweck ein Prädikat $\underline{H_n}$ ein, welches folgende Bedeutung haben soll:

> H_n`(Anweisung, Abbruch, post)` ist die schwächste Vorbedingung dafür, daß nach genau n-maliger Durchführung von `Anweisung` die Bedingung `Abbruch` erfüllt ist und `post` gilt.

H_n kann durch eine rekursive logische Definition ausgedrückt werden, denn H_0 gilt genau dann, wenn die Schleife sofort abbricht und `post` gilt, und H_{n+1} genau dann, wenn die Schleife nicht abbricht und nach einmaliger Ausführung von `Anweisung` H_n gilt.

$$H_0\texttt{(Anweisung,Abbruch,post)} \equiv \texttt{Abbruch} \wedge \texttt{post}$$

$$H_{n+1}\texttt{(Anweisung,Abbruch,post)} \equiv \neg\texttt{Abbruch} \wedge \mathbf{wp}\texttt{(Anweisung, } H_n\texttt{(Anweisung,Abbruch,post))}$$

Mit H_n läßt sich die schwächste Vorbedingung für `post` bezüglich einer Schleife ausdrücken als

$$\mathbf{wp}\texttt{(Init, } \exists \texttt{n:}\mathbb{N}\texttt{ . } H_n\texttt{(Anweisung, Abbruch, post))}.$$

Durch den Existenzquantor ist diese Form für die Praxis zu umständlich. Daher verzichtet man bei Schleifen im allgemeinen auf die *schwächste* Vorbedingung und stellt eine Verifikationsregel auf, die *hinreichende* Prämissen für Terminierung und Erfüllung der Nachbedingung enthält. Für diese Schlußregel sind Informationen über die Invariante `Inv` und die Variante `Var` der Schleife unabdinglich.

Für die partielle Korrektheit reicht es aus, über die Invariante zu argumentieren. Wenn die Invariante nach der Initialisierung gilt und nach jeder Ausführung von `Anweisung`, wenn sie vorher galt, dann gilt nach Beendigung der Schleife die Invariante und natürlich auch die Abbruchbedingung.

$$\frac{\{\texttt{pre}\}\ \texttt{Init}\ \{\texttt{Inv}\}\ \texttt{,}\ \{\texttt{Inv} \wedge \neg\texttt{Abbruch}\}\ \texttt{Anweisung}\ \{\texttt{Inv}\}}{\{\texttt{pre}\}\ \textbf{from}\ \texttt{Init}\ \textbf{until}\ \texttt{Abbruch}\ \textbf{loop}\ \texttt{Anweisung}\ \textbf{end}\ \{\texttt{Inv} \wedge \texttt{Abbruch}\}}$$

Diese Regel berücksichtig noch nicht, daß auch die Terminierung eine Prämisse für die *totale* Korrektheit der Schleife sein muß. Um Terminierung zu beweisen, benötigen wir als Voraussetzung, daß die Variante nach der Initialisierung nicht negativ ist und nach jeder (legalen) Ausführung von `Anweisung` kleiner geworden ist:

$$\{\texttt{pre}\}\ \texttt{Init}\ \{\texttt{Var}\geq\texttt{0}\}\ \text{ und }\ \{\neg\texttt{Abbruch} \wedge \texttt{Var}\geq\texttt{0} \wedge \texttt{Var=x}\}\ \texttt{Anweisung}\ \{\texttt{Var}\geq\texttt{0} \wedge \texttt{Var<x}\}$$

Invariante und Variante brauchen für den Nachweis der Korrektheit nicht explizit in der Schleifenkonstruktion genannt zu sein. Es ist jedoch extrem hilfreich, sie bei der Programmierung zu formulieren, da sonst das Aufstellen von Invariante und Variante zum – fast unlösbaren – Hauptproblem der Verifikation wird. Sie sollten es sich daher angewöhnen, Schleifen immer in der vollständigen Form

from Init **invariant** Inv **variant** Var **until** Abbruch **loop** Anweisung **end**

niederzuschreiben, weil dies die Verwendung der Verifikationsregel erheblich erleichtert. Abbildung 4.11 faßt alle Komponenten dieser Regel zusammen.

$$\frac{\{\text{pre}\}\ \text{Init}\ \{\text{Inv} \wedge \text{Var}\geq 0\}\ ,\quad \{\text{Inv} \wedge \neg\text{Abbruch} \wedge \text{Var}\geq 0 \wedge \text{Var}=x\}\ \text{Anweisung}\ \{\text{Inv} \wedge \text{Var}\geq 0 \wedge \text{Var}<x\}}{\{\text{pre}\}\ \textbf{from}\ \text{Init}\ \textbf{until}\ \text{Abbruch}\ \textbf{loop}\ \text{Anweisung}\ \textbf{end}\ \{\text{Inv} \wedge \text{Abbruch}\}}$$

$$\textbf{wp}(\textbf{from}\ \text{Init}\ \textbf{until}\ \text{Abbruch}\ \textbf{loop}\ \text{Anweisung}\ \textbf{end}\ ,\ \text{post}\ ) \equiv \textbf{wp}(\text{Init},\ \exists n{:}\mathbb{N}\,.\,H_n(\text{Anweisung},\ \text{Abbruch},\ \text{post}))$$

wobei

$$H_0(\text{Anweisung},\ \text{Abbruch},\ \text{post}) \equiv \text{Abbruch} \wedge \text{post}$$

$$H_{n+1}(\text{Anweisung},\ \text{Abbruch},\ \text{post}) \equiv \neg\text{Abbruch} \wedge \textbf{wp}(\text{Anweisung},\ H_n(\text{Anweisung},\ \text{Abbruch},\ \text{post}))$$

Abbildung 4.11: *Verifikationsregel und Prädikatentransformer für Schleifen*

Um also zu zeigen, daß pre eine Vorbedingung von post bezüglich einer Schleife ist, muß man Invariante und Variante bestimmen, obige Verifikationsregel anwenden und zum Schluß unter Anwendung der Regel zur Abschwächung der Nachbedingung (**AN**) zeigen, daß $\text{Inv} \wedge \text{Abbruch} \Rightarrow \text{post}$ gilt.

Wir wollen diese Vorgehensweise wieder an der Fakultätsfunktion zeigen, für die wir nun das (bereits auf Seite 158 vorgestellte) Programmstück "fakultäts-berechnung" entwickeln und dabei auch beweisen.

Beispiel 4.3.9 (Verifikation der Fakultätsberechnung)

Im Beispiel 4.3.8 auf Seite 156 wurde für "fakultäts-berechnung" folgende Spezifikation vorgegeben:

$\{\text{arg}>0\}$ fakultäts-berechnung $\{\text{Result}=\text{arg}!\}$

Result=arg! *ist die gewünschte Nachbedingung. Sie muß nun mithilfe von Wissen über die Ordnung der natürlichen Zahlen an die Form* $\text{Inv} \wedge \text{Abbruch}$*, also die Nachbedingungsform der Schleife, angepaßt werden. Dabei geht natürlich schon eine gewisse Vorstellung über die vorgesehene Berechnung der Fakultät mit ein, nämlich daß wir die Fakultät durch eine schrittweise Annäherung der Eingabe* arg *durch eine Zahl* n von unten *bestimmen und versuchen, in jedem Durchlauf die Eigenschaft* Result=n! *zu garantieren. Für* n=arg *ist dann* Result=arg!*. Wir spalten nun noch* n=arg *auf in* $\text{n}\leq\text{arg} \wedge \text{n}\geq\text{arg}$*, um eine mögliche Abbruchbedingung für die Annäherung von unten zu beschreiben und erhalten als äquivalente Beschreibung für die gewünschte Nachbedingung*

$$\text{Result}=\text{arg}! \quad\equiv\quad \text{Result}=\text{n}! \wedge \text{n}\leq\text{arg} \wedge \text{n}\geq\text{arg}$$

Dies ist der eigentlich kreative Anteil an der Programmentwicklung. Die Abbruchsbedingung $\text{n}\geq\text{arg}$ *und die Invariante* $\text{Result}=\text{n}! \wedge \text{n}\leq\text{arg}$ *sind somit festgelegt und müssen nun bei der weiteren Entwicklung eingehalten werden. Gemäß der Verifikationsregel für Schleifen müssen wir nun noch eine Initialanweisung* Init *festlegen mit der Eigenschaft*

$$\{\text{arg}>0\}\ \text{Init}\ \{\text{Result}=\text{n}! \wedge \text{n}\leq\text{arg}\}$$

und eine Schleifenanweisung Anweisung *mit*

$$\{\text{Result}=\text{n}! \wedge \text{n}\leq\text{arg} \wedge \neg(\text{n}\geq\text{arg})\}\ \text{Anweisung}\ \{\text{Result}=\text{n}! \wedge \text{n}\leq\text{arg}\}$$

oder vereinfacht

$$\{\text{Result}=\text{n}! \wedge \text{n}<\text{arg}\}\ \text{Anweisung}\ \{\text{Result}=\text{n}! \wedge \text{n}\leq\text{arg}\}$$

Aufgrund unserer ungefähren Vorstellung über die vorgesehene Berechnung der Fakultät probieren wir für die Initialanweisung eine Belegung von n *mit* 1 *und müssen dann natürlich auch* Result:=1 *festlegen.*

```
Init   ≡   Result := 1 ; n := 1
```

Dies ist tatsächlich eine korrekte Intialisierung, denn durch Rückwärtsargumetation von der Nachbedingung erhält man folgende Argumentationskette

Programm	**Zusicherungen**	**Prämissen**
	{arg>0}	arg>0 ⇒ 1≤arg ⇒ 1=1!∧1≤arg
	{1=1! ∧ 1≤arg}	
Result := 1		
	{Result=1! ∧ 1≤arg}	
n := 1		
	{Result=n! ∧ n≤arg}	

In der Schleifenanweisung wollen wir den Wert n *um Eins erhöhen und müssen, um die Invariante wieder zu erfüllen, entsprechend das Resultat wieder mit* n *multiplizieren*

```
Anweisung   ≡   n := n+1; Result := Result*n
```

Auch die Schleifeninitialisierung ist korrekt, da wir schon aus Beispiel 4.3.6 auf Seite 151 wissen, daß die Bedingung Result=n! *invariant gegenüber* Anweisung *ist. Auch hier liefert ein Rückwärtsargumetieren den vollständigen Beweis:*

Programm	**Zusicherungen**	**Prämissen**
	{Result=n! ∧ n<arg}	n<arg ⇔ (n+1)≤arg
	{Result=n! ∧ (n+1)≤arg}	Result=n! ⇔ Result*(n+1)=(n+1)!
	{Result*(n+1)=(n+1)! ∧ (n+1)≤arg}	
n := n+1		
	{Result*n=n! ∧ n≤arg}	
Result := Result*n		
	{Result=n! ∧ n≤arg}	

An diesem Argument sieht man, wie wichtig es ist, in der Vorbedingung nicht nur die Invariante Result=n!∧n≤arg *sondern auch die Negation der Abbruchbedingung* ¬(n≥arg) *zu haben. Damit haben wir alle Bestandteile der Schleife beisammen und gezeigt, daß folgendes Programmstück für die Berechnung der Fakultät korrekt ist, falls es terminiert:*

```
from Result := 1 ; n := 1
  invariant n<=arg    -- und Result=n!
  until n>=arg
  loop  n := n+1; Result := Result*n
end
```

Es bleibt nun noch die Terminierung zu zeigen. Wir müssen einen Integer-Ausdruck finden, der stets größer gleich Null ist, nach der Initialisierung erfüllt ist, und bei jedem Schleifendurchgang mindestens um eins kleiner wird. Da n *in jedem Schritt erhöht wird, probieren wir* arg-n*, denn* arg-n≥0 *gilt immer, bis die Routine mit* n≥arg *abbricht. Zu zeigen bleibt also noch,*

{arg>0} Result := 1 ; n := 1 {arg-n≥0}

und {¬(n≥arg)∧arg-n≥0∧ arg-n=x} n := n+1; Result := Result*n {arg-n≥0∧arg-n<x}.

Die erste Bedingung ist leicht zu beweisen:

Programm	**Zusicherungen**	**Prämissen**
	{arg>0}	arg>0 ⇒ arg-1≥0
	{arg-1≥0}	
Result := 1		
	{arg-1≥0}	
n := 1		
	{arg-n≥0}	

Die zweite Bedingung wird vor dem Beweis vereinfacht zu:

{arg-n>0 ∧ arg-n=x} n := n+1; Result := Result*n {arg-n≥0 ∧ arg-n<x}

Programm	Zusicherungen	Prämissen
	{arg-n>0 ∧ arg-n=x}	arg-n=x ⇒ arg-n≤x ⇒ arg-(n+1)<x
	{arg-n>0 ∧ arg-(n+1)<x}	arg-n>0 ⇒ arg-(n+1)≥0
	{arg-(n+1)≥0 ∧ arg-(n+1)<x}	
n := n+1		
	{arg-n≥0 ∧ arg-n<x}	
Result := Result*n		
	{arg-n≥0 ∧ arg-n<x}	

Damit sind alle Voraussetzungen der Verifikationsregel erfüllt und das angegebene Programmstück ist korrekt. Wir fassen nun noch einmal den gesamten Korrektheitsbeweis in geschlossener Form zusammen:

Programm	Zusicherungen	Prämissen
	{arg>0}	
from		pre
	{arg>0}	arg>0 ⇒ 1≤arg ∧ arg-1≥0 ∧ 1=1!
	{1=1! ∧ 1≤arg ∧ arg-1≥0}	
Result := 1 ;		
	{Result=1! ∧ 1≤arg ∧ arg-1≥0}	
n := 1		
	{Result=n! ∧ n≤arg ∧ arg-n≥0}	Inv∧Var≥0
invariant n<=arg -- und Result=n!		
variant arg-n		
until n>=arg		
loop		Inv∧¬Abbruch∧Var≥0∧Var=x
	{Result=n! ∧ n≤arg ∧ ¬(n≥arg) ∧ arg-n≥0 ∧ arg-n=x}	
	n≤arg ∧ arg-n≥0 ∧ ¬(n≥arg) ⇒ n<arg ∧ arg-n>0	
	{Result=n! ∧ n<arg ∧ arg-n>0 ∧ arg-n=x}	
	arg-n=x ⇒ arg-n≤x ⇒ arg-(n+1)<x	
	arg-n>0 ⇒ arg-(n+1)≥0	
	n<arg ⇒ n+1≤arg	
	Result=n! ⇒ Result*(n+1)=(n+1)!	
	{Result*(n+1)=(n+1)! ∧ n+1≤arg ∧ arg-(n+1)≥0 ∧ arg-(n+1)<x}	
n := n+1;		
	{Result*n=n! ∧ n≤arg ∧ arg-n≥0 ∧ arg-n<x}	
Result := Result*n		
	{Result=n! ∧ n≤arg ∧ arg-n≥0 ∧ arg-n<x}	Inv∧Var≥0∧Var<x
end		Inv∧Abbruch
	{Result=n! ∧ n≤arg ∧ n≥arg}	n≤arg ∧ n≥arg ⇒ n=arg
	{Result=n! ∧ n=arg}	Result=n! ∧ n=arg ⇒ Result=arg!
	{Result=arg!}	

4.3.6 Überprüfung

Bisher haben wir Zusicherungen in der Form von Vor- und Nachbedingungen von Routinen, Klasseninvarianten, Schleifeninvarianten und -varianten kennengelernt. In all diesen Fällen dienen sie hauptsächlich dem Zweck, *während* der Entwurfs- und Implementierungsphase bestimmte Eigenschaften festzulegen, die an der angegeben Stelle – also vor oder nach einem Routinenaufruf bzw. einer Schleife – gelten sollen.

Genauso sinnvoll kann es sein, Zusicherungen innerhalb von Anweisungen zu plazieren. So muß man zum Beispiel beim (lokalen oder entfernten) Aufruf einer Routine sicherstellen, daß die Vorbedingungen dieser Routine erfüllt sind, daß also z.B. die Funktion `item(i)` der Klasse `ARRAY` niemals mit Indizes `i` aufgerufen wird, die außerhalb der Feldgrenzen liegen. Dies könnte man natürlich dadurch sicherstellen, daß man vor jedem Aufruf einen entsprechenden Test durchführt und dann den Aufruf in eine bedingte Anweisung verlagert.

Diese Vorgehensweise würde jedoch zu einer Unmenge überflüssigen Programmtextes führen, da in den meisten Fällen die Vorbedingungen der aufgerufenen Routine durch die vorhergehenden Anweisungen sichergestellt wurden. Meist ist dies offensichtlich (z.B. wenn kurz vor dem Aufruf von `item(i)` die Zuweisung `i:=lower` steht). In den Fällen, wo dieser Zusammenhang nicht so offensichtlich ist, sollte man eine Aussage über diese Eigenschaft in den Programmtext mit aufnehmen, die klarstellt, daß man sich bei der Implementierung über die Situation durchaus im klaren war *und daß jede Änderung des Programmtextes auf diese Programmeigenschaft Rücksicht zu nehmen hat.*

Eiffel bietet die Möglichkeit an, auch diese Aussagen als überprüfbare Zusicherungen innerhalb einer check-Anweisung zu formulieren. Eine solche Anweisung beginnt mit dem Schlüsselwort **check** und enthält dann eine Reihe von Zusicherungen, die mit **end** abgeschlossen werden.

```
check
  Zusicherung_1;
  Zusicherung_2;
     ⋮
  Zusicherung_n
end
```

Als Zusicherungen sind die Ausdrücke der Zusicherungssprache von Eiffel (Sektion 3.7) erlaubt. Sie werden bei Ausführung der check-Anweisung überprüft, sofern die entsprechenden Parameter im System Definition File gesetzt sind. In diesem Falle bricht das Programm ab, falls eine der Zusicherungen verletzt ist. Da die Ausdruckskraft der Zusicherungen jedoch begrenzt ist und die Überprüfung von Zusicherungen Laufzeit kostet, geht die Rolle der Check-Anweisungen nur selten über die eines sinnvoll plazierten Kommentars hinaus.

Man sollte sich daher darüber im klaren sein, daß eine Überprüfung von Zusicherungen kein Ersatz für einen Korrektheitsbeweis sein kann, sondern bestenfalls – in Verbindung mit dem Ausnahmemechanismus, den wir im folgenden Abschnitt besprechen – eine zusätzliche Absicherung gegenüber unkontrollierbarem Verhalten des Gesamtsystems, falls darin – was leider sehr wahrscheinlich ist – auch unverifizierte Klassen zum Einsatz kommen, die eventuell gewisse Abmachungen wie die Vorbedingungen eines Routinenaufrufs nicht einhalten.[30]

4.3.7 Umgang mit Fehlern: Disziplinierte Ausnahmen

In einer idealen Programmierwelt wären alle Programme als korrekt nachgewiesen und gegenüber jeglicher Form von Fehlbedienung abgesichert. Leider aber sind wir von beiden Zielen noch weit entfernt. Zum einen gibt es zu wenig Werkzeuge, welche eine Verifikation von Programmen unterstützen, und zum anderen ist es ausgesprochen aufwendig, Programme robust zu gestalten.

Wird zum Beispiel vom Benutzer eines Programms eine numerische Eingabe erwartet, so muß man damit rechnen, daß sich der Benutzer bei der Eingabe vertippt und innerhalb der eingegebenen Ziffernfolge ein Buchstabe auftritt. Die verwendete Eingaberoutine (`readreal`, siehe Abschnitt 4.3.9) kann daher ihren Kontrakt nicht erfüllen und keine Zahl an die aufrufende Stelle zurückgeben. Sie bricht also ihre Tätigkeit in einer Art "organisierter Panik" ab und hinterläßt ein Fehlersignal (das feature `error` hat einen Wert ungleich Null).

Sicherlich wäre es nicht sinnvoll, beim Auftreten eines solchen Fehlers gleich das gesamte Programm abzubrechen. Bei einer erneuten Aufforderung würde der Benutzer seine Eingabe sicherlich mit größerer Konzentration wiederholen. Man könnte also jede Eingaberoutine in eine Schleife einbetten, die solange wiederholt wird, bis kein Fehlersignal mehr auftritt. Diese Vorgehensweise hätte jedoch den Nachteil, daß das Programm unnötig durch Schleifen aufgebläht wird und der Ausnahmefall das eigentliche Programm dominiert.

Ebenso muß man damit rechnen, daß bei der Überprüfung von Zusicherungen festgestellt wird, daß eine der vorgesehenen Bedingungen – zum Beispiel die Vorbedingung einer Routine – nicht eingehalten wird. Die aufgerufene Routine kann diesen Fehler nicht beheben. Es wäre aber auch nicht sinnvoll, über einen einmal

[30] Eine ausführliche Diskussion über einen sinnvollen Einsatz des Überprüfungsmechanismus bei möglichst geringen Laufzeitverlusten findet man in [Meyer, 1988, Kapitel 7.10.1/2]

entdeckten Fehler stillschweigend hinwegzugehen. Daher muß die Verletzung von Zusicherungen dazu führen, daß die Routine abbricht und eine Fehlermeldung absetzt. Aber auch in diesem Fall wäre ein radikaler Abbruch des gesamten Programms nicht immer die beste Konsequenz.[31]

Da Ausnahmesituationen sich also nicht prinzipiell ausschließen lassen, bieten viele Programmiersprachen mittlerweile einen Mechanismus zur Behandlung von Ausnahmen (*exceptions*) an, welcher Ausnahmesituationen "abseits" von dem eigentlichen Programmablauf behandelt. Derartige Mechanismen bergen zwar die Gefahr in sich, daß Programmierer sie mißbrauchen um die strenge Struktur von Eiffel-Programmen zu umgehen. Sie werden jedoch gebraucht, um tatsächliche Ausnahmesitationen zu einem wohldefinierten Ende zu bringen anstatt einen Benutzer vor die Situation eines unkontrollierten Abbruchs zu stellen.

Zwei Vorgehensweisen sind sinnvoll, wenn eine Ausnahme – also eine abnorme Bedingung während der Ausführung eines Softwareelements – auftritt.

Wiederaufnahme (resumption):

Man versucht innerhalb der Routine, in der die Ausnahmesituation auftritt, diese durch Korrekturmaßnahmen zu beseitigen und nimmt die Durchführung der Routine wieder auf. In dieser Form ist es eventuell möglich, den Kontrakt zwischen dem Kunden und der Routine doch noch zu erfüllen.

Natürlich muß die Wirkung der Korrekturmaßnahmen für die Routine zugänglich sein. Übergebene Argumente können nicht verändert und lokale Variablen nicht neu initialisiert werden. Die Wiederholung (in Eiffel ausgelöst durch die Anweisung **retry**) muß also hinter den lokalen Deklarationen aufsetzen.

Organisierte Panik:

Läßt sich auf der Ebene der Routine die Ausnahmesituation nicht beheben, dann muß die Korrektur vom Benutzers durchgeführt werden. Dazu ist es notwendig, den eigenen Zustand in geordneter Weise abzubrechen (also z.B. vor dem Abbruch die Klasseninvariante zu erfüllen) und den Kunden von der Ausnahmesituation zu informieren. Der Kunde hat dann die Möglichkeit, seinerseits eine Fehlerbehandlung vorzunehmen und ggf. gezielte Korrekturen anzubringen.

Alle anderen Vorgehensweisen würden einen Bruch gegenüber der Vertragssituation zwischen Kunden und Lieferanten bedeuten. Insbesondere ist es nicht akzeptabel, nach der Entdeckung einer Ausnahmesituation stillschweigend die Steuerung an den Kunden zurückzugeben, der dann ahnungslos weiterarbeiten und ggf. eine katastrophale Situation auslösen würde.[32] Eine Routine muß ihren Vertrag erfüllen oder scheitern.

Der Ausnahmemechanismus von Eiffel beruht auf diesen Grundgedanken und bietet zu jeder Routine die Möglichkeit einer Ausnahmebehandlung an. Hierzu kann man *nach dem Anweisungsteil und nach der Nachbedingung* eine rescue-Klausel angeben, in der festgelegt wird, was im Falle einer Ausnahme zu tun ist.

```
r is    -- Kommentar
   require Vorbedingungen
   local   Lokale Variablen
   do       Anweisungsteil
   ensure  Nachbedingungen
   rescue  Ausnahmebehandlung
end
```

[31] Mögliche weitere Ausnahmesituationen sind Fehler in der Arithmetik (Division durch Null, negatives Argument im Logarithmusg), Speicherplatzmangel, Fehler in der Ein- und Ausgabe. usw. Die Information, welche Art von Ausnahme aufgetreten ist, ist in der jeweiligen Klassendefinition enthalten, deren Routinendurchführung die Ausnahme ausgelöst hat. Für Systemfehler und allgemeine Ausnahmesituationen, wie die Verletzung von Zusicherungen, kann in der Klasse `EXCEPTION` die entsprechende Information gefunden werden, vorausgesetzt die aktuelle Routine ist einem Erben dieser Klasse.

[32] Dies gilt übrigens nicht nur im Bereich der Programmierung. Wenn man in seinem eigenen Verantwortungsbereich einen Fehler entdeckt, dann ist es unverantwortlich, stillschweigend darüber hinwegzugehen in der Hoffnung, niemand würde es merken. Werden Fehler frühzeitig eingestanden, so mag dies vielleicht etwas peinlich sein, aber es ist wenigstens noch möglich, Gegenmaßnahmen zu ergreifen. Wartet man damit zu lange, dann wird eine Korrektur unmöglich und die Katastrophe wird unausweichlich. Beispiele hierzu liefert die Geschichte (und das Erfahrungsumfeld der meisten Menschen) zur Genüge.

Wann immer während der Ausführung des normalen Rumpfes eine Ausnahme auftritt, dann wird diese Ausführung angehalten und stattdessen die Anweisung `Ausnahmebehandlung` ausgeführt. Innerhalb dieser Klausel (und nirgendwo anders) darf die Anweisung **retry** ausgeführt werden, welche einen Neustart des Anweisungsteils der Routine auslöst. So könnte zum Beispiel die oben erwähnte fehlerhafte Eingabe so gerettet werden, daß der Benuzter auf die korrekte Syntax hingewiesen wird und dann erneut die Eingabe gelesen wird(, bis irgendwann einmal der Geduldsfaden reißt).

```
numerische_eingabe_lesen:REAL is
  do
     message("bitte eine REAL-Zahl eingeben");
     io.readreal
     Result := io.lastreal
  rescue
     message("REAL-Syntax: d.d  oder  d.dEd  oder  d.dE-d");
     retry
end
```

Zuweilen fällt die Ausnahmebehandlung aufwendiger aus als die eigentliche Routine. So könnte man in der Routine `numerische_eingabe_lesen` erst ab dem zweiten Fehlversuch die Syntax angeben und die Anzahl der Fehlversuche begrenzen. Der Anweisungsteil bleibt davon jedoch völlig unberührt und somit liegt eine klare Trennung von der eigentlich vorgesehenen Ausführung und der Behandlung von Fehlern vor – sowohl im Programmtext als auch während des Ablaufs, in dem im Normalfall die rescue-Klausel keine Rolle spielt.[33]

Nur, wenn nach einer Ausnahmesituation ein **retry** erfolgreich war, kann der Vertrag einer Routine erfüllt werden. Andernfalls *muß* sie scheitern. Daher scheitert eine Routinenausführung stets dann, wenn die rescue-Klausel bis zum Ende ausgeführt wurde, und dem Aufrufer wird das Fehlverhalten mittels einer Ausnahmesituation mitgeteilt.[34] Im schlimmsten Fall pflanzt sich die Ausnahmesituation durch alle Aufrufe zurück bis zur Wurzelklasse und beendet dort die Programmausführung.

Bezüglich Korrektheit gelten für die rescue-Klausel andere Anforderungen als für den normalen Anweisungsteil einer Routine. Zum einen braucht die rescue-Klausel *nicht* die Nachbedingung der Routine zu erfüllen, denn ihre Aufgabe ist ja nur, eine Ausnahmesituation einem geordneten Ende entgegenzuführen oder eine Wiederaufnahme des Anweisungsteils vorzubereiten. Die Nachbedingung einer rescue-Klausel besteht also nur aus der Klasseninvarianten und ggf. der Vorbedingung der Routine, falls **retry** aufgerufen wird. Zum anderen aber muß die rescue-Klausel *in jedem Falle* fehlerfrei arbeiten. Ihre Vorbedingung ist daher immer `true`.

Definition 4.3.10 (Korrektheit von Ausnahmebehandlungen)

Innerhalb einer Klasse K mit Invariante `INV` *heißt eine Routine r genau dann Ausnahme-korrekt, wenn für alle gültigen Argumente x_r und jeden Zweig b ihres* rescue*-Blocks gilt*

- $\{\texttt{true}\}\ \texttt{b}\ \{\texttt{INV} \wedge \texttt{pre}_r(x_r)\}$, *falls b mit* **retry** *endet*
- $\{\texttt{true}\}\ \texttt{b}\ \{\texttt{INV}\}$, *falls b ohne* **retry** *endet*

Die Ausnahme-Korrektheit kann mit den bisher bekannten Verifikationsregeln bewiesen werden.

4.3.8 Debugging

Während der Entwicklung und Erprobung eines Programms läßt man sich üblicherweise eine Reihe von Testdaten ausgeben, um auf Fehlern schneller auf die Spur zu kommen. Dies ist besonders hilfreich bei den unvermeidlichen Tippfehler beim Erstellen des Programmtextes, die man auch bei einem gründlichen

[33] Weitere Beispiele für einen sinnvollen Einsatz der Ausnahmebehandlung findet man in [Meyer, 1988, Kapitel 7.10.4/5]

[34] Fehlt die rescue-Klausel, so wird beim Übersetzen vom Compiler die Klausel "**rescue** `default_rescue`" eingesetzt. Die Prozedur `default_rescue` (der Klasse `ANY`) hat normalerweise keinerlei Wirkung und bewirkt einen sofortigen Abbruch. Sie kann aber redefiniert werden, falls für ein System eine bestimmte Standardausnahmenbehandlung gewünscht wird.

Gegenlesen nicht findet(, aber auch bei Denkfehlern, die entstehen, wenn das Programm nicht verifiziert wurde. In den meisten Sprachen muß man hierzu Ergänzungen im Programmtext vornehmen, die man später wieder rückgängig macht. Da dies auch wieder zu Fehlern führen kann, bietet Eiffel die Möglichkeit an, bestimmte Anweisungen in eine debug-Anweisung zu verlagern. Die Syntax

debug `Anweisung`$_1$`; ...; Anweisung`$_n$ **end**

beschreibt eine Anweisung, die nur übersetzt wird, wenn die Compiler-Option DEBUG im System Description File entsprechend gesetzt ist. In diesem Falle verhält sich die debug-Anweisung wie eine gewöhnliche Anweisung, d.h. die Folge `Anweisung`$_1$`; ...; Anweisung`$_n$ wird ausgeführt, wann immer sie an der Reihe ist. Ist die Option ausgeschaltet, so wird die debug-Anweisung einfach ignoriert, als ob sie gar nicht da wäre.

4.3.9 Einfache Ein- und Ausgabe

Im Gegensatz zu den üblichen Programmiersprachen hat Eiffel als Sprache *keine* Ein- und Ausgabe. Jedoch stellt das Eiffel System die Bibliotheksklassen `FILE` und `STANDARD_FILES` zur Verfügung, deren features alle nötigen Prozeduren und Funktionen anbieten.

- Die Klasse `FILE` bietet als Objekte sequentielle Dateien an. Eine sequentielle Datei kann man sich als ein Band vorstellen, von dem man entweder nur von vorne nach hinten stückweise lesen kann (Eingabeband), oder an dem man nur am Ende neue Werte anfügen kann (Ausgabeband).

- Die Klasse `STANDARD_FILES` bietet als Attribute drei sequentielle Dateien vom Typ `FILE` an:
  – `input` für die Eingabe,
  – `output` für die Ausgabe und
  – `error` für Fehlermeldungen

  Darüber hinaus liefert sie Routinen, um die Ein- und Ausgabe am Bildschirm realisieren zu können. Abbildungen 4.12 und 4.13 stellen die wichtigsten davon zusammen.

Routine	Vorbedingung	Nachbedingung
`readchar`		Das erste Zeichen des Eingabebands wurde entfernt und in den Puffer eingelesen
`lastchar`	`readchar` wurde durchgeführt	`Result` = das im Puffer gespeicherte Zeichen
`readint`	Am Eingabeband steht als nächstes eine Ziffernfolge	Die Zifferfolge wurde gelesen, vom Band entfernt, als `INTEGER`-Wert interpretiert und im Puffer gespeichert
`lastint`	`readint` wurde durchgeführt	`Result` = der Integerwert im Puffer
`readreal`	Am Eingabeband steht eine Zeichenfolge, die eine `REAL`-Zahl darstellt	Die Zifferfolge wurde gelesen, vom Band entfernt, als `REAL`-Wert interpretiert und im Puffer gespeichert
`lastreal`	`readreal` wurde durchgeführt	`Result` = der Realwert im Puffer
`readline`		Alle Zeichen bis zum Return-Zeichen wurden eingelesen und im `STRING`-Puffer gespeichert
`laststring`	`readline` wurde durchgeführt	`Result` = Verweis auf den Stringpuffer[35]
`next_line`		Ignoriert den Rest der Zeile. Der nächste Wert wird in der nächsten Zeile erwartet

Abbildung 4.12: *Die wichtigsten Routinen für die Eingabe*

[35]Mit `laststring.duplicate` erhält man eine Kopie des gelesenen Strings statt des Verweises auf den Puffer

Prozeduraufruf	Effekt
`putint(n)`	Ausgabe des Wertes von n auf die Ausgabedatei
`putreal(r)`	Ausgabe des Wertes von r auf die Ausgabedatei
`putchar(c)`	Ausgabe des Wertes von c auf die Ausgabedatei
`putstring(s)`	Ausgabe des Wertes von s auf die Ausgabedatei
`new_line`	nächste Ausgabe erfolgt am Beginn der nächsten Zeile

`n` muß ein Integerausdruck, `r` ein Integer- oder Realausdruck,
`c` ein Characterausdruck und `s` ein Stringausdruck sein.

Abbildung 4.13: *Die wichtigsten Routinen für die Ausgabe*

Um also eine Ein- oder Ausgabe ausführen zu können, muß man ein feature vom Typ `STANDARD_FILES` deklarieren und dann über qualifizierte Aufrufe die entsprechenden Routinen anstoßen. Um dies zu vereinfachen, wurde in der Klasse `ANY`, von der jede Klasse automatisch erbt, ein feature `io:STANDARD_FILES` *vordefiniert*. Daraus ergibt sich die folgende Schreibweise für Ein- und Ausgaben.

```
io.putint(43);
io.new_line;
io.putstring("   Dies war eine 43");
io.new_line;
io.putstring("...................")
```

erzeugt als Ausgabe

```
43
   Dies war eine 43
...................
```

Um eine Integerzahl einzulesen und an eine Variable `n` zu übergeben, benötigt man folgende Anweisungen:

```
io.readint; n := io.lastint
```

Auf den ersten Blick erscheint diese Form der Ein- und Ausgabe etwas seltsam und umständlich. Der qualifizierte Aufruf mit `io` wirkt unnötig kompliziert und die Trennung des Einlesens in eine Einleseprozedur mit anschließendem Funktionsaufruf noch viel mehr. Schließlich erscheint es lästig, daß man bei Lese- und Schreiboperationen immer den Typ des Wertes mit angeben muß. Dennoch gibt es für alles gute Gründe.

Der qualifizierte Aufruf macht klar, daß es sich bei Ein- und Ausgabe um Dienstleistungen handelt und nicht um Bestandteile der Sprache. Den Klienten dieser Dienstleistung interessiert ja eigentlich nur, *daß* er bestimmte Werte als Eingaben bekommt bzw. an den Benutzer ausgeben kann. *Wie* die Ein- und Ausgabe realisiert ist – z.B. ob mit oder ohne Window-Unterstützung – geht ihn nichts an. Der Vorteil dieser Sicht ist, daß die einfachen Dienstleistungen durch verbesserte Versionen ersetzt werden können (mit Maus- und Menüsteuerung oder besserer Fehlerbehandlung), ohne das deshalb das Programm geändert werden muß.[36]

Die etwas umständlich wirkende Eingabe ergibt sich aus der strikte Trennung von Prozeduren und Funktionen. Einlesen und einen Eingabewert zu bestimmen sind, genau besehen, zwei Operationen. Die erste nimmt eine Veränderung des Objektes `input` vor (der Lesekopf des Eingabebandes wird verschoben und ein Pufferobjekt beschrieben) die zweite bestimmt einen Wert, der zugewiesen werden kann. Es wäre nicht sinnvoll, ausgerechnet bei Ein- und Ausgabe von der Eiffel-Philosophie Abstand zu nehmen und die Eingabe als Funktion mit Seiteneffekten oder als Prozedur mit einer Veränderung übergebener Parameter zu realisieren.

Die Unterscheidung des Typs bei Lese- und Schreibroutinen entspricht dem strengen Typkonzept von Eiffel, von dem man auch bei Ein- und Ausgabe nicht so einfach abweichen sollte. Der Vorteil dieser Vorgehensweise ist, daß fehlerhafte Eingaben und Ausgaben dort entdeckt und behandelt werden können, wo sie entstehen (nämlich bei Ein- und Ausgabe) und nicht erst nachträglich bei der Verarbeitung zu Laufzeitfehlern führen.

Nichtsdestotrotz ist die Eiffel-Schreibweise gewöhnungsbedürftig, vor allem, wenn man bereits mit anderen Programmiersprachen gearbeitet hat, in denen allgemeinere Lese- und Schreibroutinen erlauben, gleichzeitig mehrere Werte verschiedenen Typs einzulesen bzw. in einen definierbaren Muster auszugeben. Das Fehlen derartiger Routinen macht die Programmierung *einfacher* Ein- und Ausgabe in Eiffel zu einer aufwendigen Tätigkeit. Erst bei der Gestaltung *komplexer* Benutzeroberflächen kann Eiffel seine Vorteile wieder ausspielen.

[36] Wer dennoch den qualifizierten Aufruf umgehen will, der kann die Klasse `STANDARD_FILES` in der Erbklausel aufführen.

4.3.10 Die Verifikation rekursiver Routinen

In den bisherigen Abschnitten haben wir alle Arten von Anweisungen besprochen und Regeln aufgestellt, wie ihre Korrektheit zu beweisen ist. Bei diesen Betrachtungen haben wir jedoch einen Sonderfall außer Acht gelassen, der bei der Verifikation noch Schwierigkeiten erzeugen kann, nämlich Routinen die sich selbst direkt oder indirekt wieder aufrufen. Unser Programmstück zur Berechnung der Fakultät (siehe Beispiel 4.3.9 auf Seite 161) hätten wir zum Beispiel auch durch folgende *rekursive* Funktion ausdrücken können.

```
fak_berechnung(n:INTEGER):INTEGER is
  require n>0
  do
    if n=1 then Result := 1
           else  Result := n * fak_berechnung(n-1)
    end
end
```

Dies spiegelt einen Gedankengang der Mathematik wieder, die Fakultätsfunktion statt durch eine Interation durch zwei Gleichungen `1!=1` $\wedge$ `n!=n*(n-1)!` auszudrücken, und ist um einiges eleganter (und leichter zu verifizieren) als die Schleife. Dennoch tauchen *rekursive* Routinen tauchen in der Praxis recht selten auf, da der gleiche Effekt mit Schleifen oft effizienter zu erreichen ist. In manchen Fällen ist dies allerdings recht schwierig und deshalb kann man auf die Möglichkeit rekursiver Routinenaufrufe nicht sinnvoll verzichten.

So gibt es zum Beispiel Sortierprogramme für Felder, die zunächst einen "Mittelwert" bestimmen, das Feld in zwei Teile zerlegen, die nur größere bzw. nur kleinere Werte enthalten, und dann in jeden Teil wieder nach dem gleichen Verfahren sortieren, bis es nichts mehr zu teilen gibt. Eine rekursive Beschreibung dieses sehr effizienten Verfahrens ("Quicksort") ist auf Grundlage dieser Beschreibung sehr leicht zu geben. Ein Programm, welches dasselbe nur durch Einsatz von Schleifen erreicht, ist dagegen nicht so leicht zu finden.

Darüber hinaus gibt es arithmetische Funktionen, die nur durch rekursive Gleichungen spezifiziert werden können. Ein beliebtes Beispiel hierfür sind die Fibonaccizahlen, die wie folgt beschrieben sind.

> Die erste Fibonaccizahl ist 1, die zweite ebenfalls.
> Alle größeren Fibonaccizahlen ergeben sich aus der Summe ihrer beiden Vorgänger.

Bezeichnet man die n-te Fibonaccizahl mit `fib(n)`, so kann man dies durch zwei Gleichungen ausdrücken:

`fib(1)=fib(2)=1, fib(n+2)=fib(n+1)+fib(n)`.

Ein rekursives Programm für die Fibonaccizahlen ist leicht zu schreiben. Wir müssen nur jede Eingabe einheitlich durch den Parameter `n` beschreiben, die Fälle `n=1, n=2` und `n>2` unterscheiden und die zweite Gleichung durch eine Indextransformation `n` $\rightarrow$ `n-2` umschreiben.

```
fib(n:INTEGER):INTEGER is
  require n>0
  do
     if n<3 then Result := 1
            else Result := fib(n-1)+fib(n-2)
     end
  ensure fib(1)=1; fib(2)=1  -- fib(n+2)=fib(n+1)+fib(n)
end
```

Die Programmierung der Fibonaccizahlen durch Schleifen (siehe Abbildung 4.14) bedarf dagegen schon eines Tricks – der Zwischenspeicherung von Ergebnissen innerhalb einer Schleife –, welcher bei komplizierteren Rekursionen aber nicht mehr anwendbar ist.

Die Verifikation eines rekursiven Programms erscheint sehr einfach. So lassen sich die Gleichungen in der Nachbedingung des Programms für Fibonaccizahlen sehr leicht beweisen. Es gibt dabei jedoch eine Schwierigkeit, die an dieser Stelle sehr gerne übersehen wird: rekursive Aufrufe von Routinen können – genauso wie Schleifen – zu nichtterminierenden Programmen führen, wie folgendes sehr einfache Beispiel zeigt

```
fib(n:INTEGER):INTEGER is
  require n>0
  local i, fib_1, fib_2 : INTEGER
  do
     if n<3 then Result := 1
            else
              from fib_1 := 1; fib_2 := 1; Result := fib_1+fib_2; i := 3
                until   i = n
                invariant Result=fib(i); fib_1=fib(i-1); fib_2=fib(i-2)
                variant n-i
                loop   i := i+1;  fib_2 := fib_1;  fib_1 := Result;  Result := fib_1+fib_2
              end
     end
  ensure fib(1)=1; fib(2)=1   -- fib(n+2)=fib(n+1)+fib(n)
end
```

Abbildung 4.14: *Berechnung der Fibonaccizahlen durch Schleifen*

```
fakultät_niemals_stop(n:INTEGER):INTEGER is
  do Result := n * fakultät_niemals_stop(n-1)  end
```

In diesem Programm wird versucht, die Fakultät nach dem üblichen Muster zu berechnen, aber vergessen, einen Initialwert festzulegen. Die Funktion wird sich selbst solange aufrufen, bis das Betriebssystem die rekursiven Aufrufe nicht mehr verwalten kann und mit Speicherüberlauf abbricht (ansonsten würde sie endlos laufen).

Deshalb muß man bei der Verifikation rekursiver Programmen ebenfalls die *partielle Korrektheit und die Terminierung* betrachten und eine Invariante (was gilt bei jedem rekursiven Aufruf) und eine Variante (was wird bei jedem Aufruf kleiner) aufstellen. Die grundsätzliche Vorgehensweise ist also ähnlich zu derjenigen bei der Verifikation von Schleifen. Es wird jedoch manches leichter und anderes komplizierter. Insbesondere läßt sich keine formale Verifikationsregel aufstellen sondern nur eine "halb-formale" Beweisvorschrift.[37]

- Die *Invariante* einer rekursiven Routine entspricht dem Vertrag dieser Routine, den sie insbesondere *auch mit sich selbst* schließt. Bei jedem rekursiven Aufruf der Routine wendet man also einfach die Verifikationsregel für Routinenaufrufe (siehe Abschnitt 4.3.2.3) an und geht dabei davon aus, daß dieser Aufruf seinen Vertrag erfüllt. Mit den restlichen Verifikationsregeln läßt sich dann – wie bei nicht-rekursiven Routinen – die (partielle) Korrektheit der Implementierung beweisen, wie im folgenden Beispiel:

```
fak_berechnung(n:INTEGER):INTEGER is
  do                          {n>0}                                          pre
    if n=1
      then                    {n>0∧n=1}                          n>0∧n=1 ⇒ 1=n!
                              {1=n!}
        Result := 1
                              {Result=n!}
      else                    {n>0∧n≠1}                  n>0∧n≠1 ⇒ n-1>0 ∧ n*(n-1)!=n!
                              {n-1>0 ∧ n*(n-1)!=n!}      n-1>0⇒fak_berechnung(n-1)=(n-1)!
                              {n*fak_berechnung(n-1)=n!}
        Result := n * fak_berechnung(n-1)
                              {Result=n!}
    end                       {Result=n!}                                    post
end
```

Dieser Beweis wäre natürlich völlig wertlos ohne den Nachweis der Terminierung, also der Aussage, daß alle Aufrufe irgendwann einmal ohne einen weiteren rekursiven Aufruf ein Ergebnis liefern müssen. Die Korrektheit von Ergebnissen, die – wie `fak_berechnung(1)` – ohne Selbstreferenz bestimmt werden, ist natürlich schon im Beweis der partiellen Korrektheit enthalten.

[37] Es ist zwar durchaus möglich, die in Abbildung 4.15 aufgestellte Beweisvorschrift in Form einer formalen Verifikationsregel niederzuschreiben. Dies aber würde einen erheblich aufwendigeren Formalismus erfordern und zu einer für die Intuition kaum noch verständlichen Regel führen.

- Die *Variante* einer rekursiven Routine muß ausdrücken, daß die Routine nur endlich oft aufgerufen werden kann. Wie bei den Schleifen benötigt man hierzu einen Integer-Ausdruck, der stets größer gleich Null ist und bei jedem rekursiven Aufruf mindestens um eins kleiner wird. Anders als bei der Schleife muß diese aber durch einen Ausdruck beschrieben werden, der nichts über die inneren Eigenschaften der Routine sagt, sondern nur von den äußeren Merkmalen abhängt – also den formalen Parametern, dem untersuchten Objekt und ggf. dem Resultat. Wir müssen die Variante also im Extremfall durch einen Integer-Ausdruck `Var(`$\mathtt{y}_r$`,actual)` beschreiben, um alle möglichen Abhängigkeiten zu berücksichtigen.

  In unserem Beispiel `fak_berechnung` ist die Variante ein sehr einfacher Ausdruck, nämlich der Parameter `n` selbst. Aufgrund der Vorbedingung `n>0` ist die Variante stets größer gleich Null. Bei jedem rekursiven Aufruf wird sie geringer, da `fak_berechnung(n-1)` der einzige rekursive Aufruf ist. In den meisten rekursiven Routinen ist die Variante zum Glück ähnlich einfach und offensichtlich wie hier.

Eine Komplikation, welche das Aufstellen einer formalen Verifikationsregel erheblich erschwert, ist die Tatsache, daß eine Routine beliebig viele rekursive Aufrufe haben darf und diese beliebig tief in anderen Programmkonstrukten eingebaut sein dürfen. Da Rekursion aber als eine Art Gegenstück zu Schleifen anzusehen ist, macht es wenig Sinn, rekursive Aufrufe innerhalb einer Schleife zu plazieren. Auch würde ein Korrektheitsbeweis dadurch nahezu unmöglich. Daher darf man davon ausgehen, daß rekursive Aufrufe – wie in unserem Beispiel – *nur innerhalb von bedingten Anweisungen oder Fallunterscheidungen* vorkommen. Somit kann man zu jedem rekursiven Aufruf eine *Bedingung* angeben, die besagt, wann dieser Aufruf überhaupt ausgeführt werden soll. Die Existenz einer solchen Bedingung ist wichtig für einen Terminierungsbeweis, da ohne diese Bedingung die Routine sich unendlich oft aufrufen würde.

Die in Abbildung 4.15 angegebene Verifikations-"regel" für rekursive Routinen beschreibt eine Beweismethodik, welche das oben Gesagte etwas stärker präzisiert und sich an der Schleifenregel (Abbildung 4.11) orientiert.

Es sei `r` eine Routine mit formalen Argumenten $\mathtt{y}_r$ und abstrakten Vor- und Nachbedingungen $\mathtt{pre}_r$ bzw. $\mathtt{post}_r$. Der Anweisungsteil $\mathtt{B}_r$ enthalte insgesamt `m` rekursive Aufrufe der Form $\mathtt{entity}_i\mathtt{.r(A}_i\mathtt{(y}_r\mathtt{))}$, wobei jeweils $\mathtt{entity}_i$ ein Objekt und $\mathtt{A}_i\mathtt{(y}_r\mathtt{)}$ Ausdrücke für aktuelle Argumente beschreibt. $\mathtt{Bed}_i\mathtt{(y}_r\mathtt{)}$ beschreibe die Bedingung für den i-ten rekursiven Aufruf.

Die Routine r ist genau dann total korrekt, wenn gilt

Partielle Korrektheit: *Unter der Annahme*, daß alle rekursiven Aufrufe die Spezifikation von `r` erfüllen, kann mithilfe der Verifikationsregeln[38]gezeigt werden

$$\{\,\mathtt{pre}_r(y_r)\,\}\ \ \mathtt{B}_r\ \ \{\,\mathtt{post}_r(y_r)\,\}$$

Terminierung: Es gibt einen Integer-Ausdruck `Var`, der nur von den Argumenten $\mathtt{y}_r$ und dem aktuellen Objekt abhängt und für den gezeigt werden kann

$$\mathtt{pre}_r(y_r)\ \Rightarrow\ \mathtt{Var}(y_r\mathtt{,Current}) \geq 0$$

und für *jeden* rekursiven Aufruf $\mathtt{entity}_i\mathtt{.r(A}_i\mathtt{(y}_r\mathtt{))}$

$$\mathtt{pre}_r(y_r) \wedge \mathtt{Bed}_i\mathtt{(y}_r\mathtt{)}\ \Rightarrow\ \mathtt{pre}_r\mathtt{(A}_i\mathtt{(y}_r\mathtt{))} \wedge\ \mathtt{Var(A}_i\mathtt{(y}_r\mathtt{),entity}_i\mathtt{)<Var}(y_r\mathtt{,Current})$$

Abbildung 4.15: *Verifikationsregel für rekursive Routinen*

Für die partielle Korrektheit darf man also die Korrektheit aller Aufrufe voraussetzen. Im Terminierungsbeweis ist zu zeigen, daß die Variante bei jedem vertragsgemäßen Beginn nicht negativ ist (entspricht der Initialisierung von Schleifen) und bei jedem rekursiven Aufruf kleiner wird (entspricht der Schleifenanweisung). Dabei muß aus den Bedingungen des Aufrufs die entsprechende Vorbedingung folgen (entspricht der Annahme, daß die Abbruchbedingung nicht erfüllt ist).

[38]Dabei kommen insbesondere natürlich die Verifikationsregeln für Routinenaufrufe (Abbildungen 4.4 und 4.5 zum Tragen.

Beispiel 4.3.11 (Verifikation rekursiver Aufrufe)

1. *In unserem Fakultätsberechnungsprogramm* `fak_berechnung(n)` *gibt es einen rekursiven Aufruf, der nur dann stattfindet, wenn* `n≠1` *gilt. In jedem Schritt wird* `n` *um Eins kleiner.*

   Wir haben damit $\texttt{Bed}_1\texttt{(n)} \equiv \texttt{n}\neq\texttt{1}$ *und* `Var(n,Current)=n`.

   Es folgt:
   $\texttt{n>0} \Rightarrow \texttt{n}\geq\texttt{0}$
   also $\texttt{pre(n)} \Rightarrow \texttt{Var(n,Current)}\geq\texttt{0}$
   und
   $\texttt{n>0} \wedge \texttt{n}\neq\texttt{1} \Rightarrow \texttt{n-1>0} \wedge \texttt{n-1<n}$
   also $\texttt{pre(n)} \wedge \texttt{Bed}_1\texttt{(n)} \Rightarrow \texttt{pre(n-1)} \wedge \texttt{Var(n-1,Current)<Var(n,Current)}$

   Die partielle Korrektheit hatten wir bereits bewiesen.

2. *Im Programm* `fib(n)` *zur Berechnung der Fibonaccizahlen gibt es zwei rekursive Aufrufe, die nur dann stattfindet, wenn* $\texttt{n}\not<\texttt{3}$ *gilt. Bei jedem Aufruf wird* `n` *um Eins bzw. Zwei kleiner.*

   Wir haben damit $\texttt{Bed}_1\texttt{(n)} \equiv \texttt{Bed}_2\texttt{(n)} \equiv \texttt{n}\geq\texttt{3}$ *und* `Var(n,Current)=n` .

   Mit diesen Angaben ist die Terminierung leicht zu beweisen. Die partielle Korrektheit folgt unmittelbar aus der Anweisung `Result := fib(n-1)+fib(n-2)` *, da dies genau der Spezifikation entspricht.*

Über die bisher betrachteten Möglichkeiten hinaus kann man rekursive Aufrufe noch weiter verkomplizieren. So gibt es zum Beispiel Anwenungen, in der zwei Prozeduren sich gegenseitig immer wieder aufrufen. Dies kommt aber verhältnismäßig selten vor und soll hier nicht weiter vertieft werden.

4.3.11 Diskussion

Eiffel-Anweisungen sind nach der Methodik der strukturierten Programmierung gestaltet und zielen auf eine bestmögliche Unterstützung einer systematischen und verifizierbaren Implementierung von Routinen.

- Anweisungen haben stets nur einen Eingang und einen Ausgang – ein Verlassen von Schleifen oder bedingten Anweisungen an einer anderen Stelle als der mit **end** bezeichneten ist nicht möglich.

- Es besteht eine strikte Trennung zwischen Funktionen (ohne Seiteneffekte) und Prozeduren (ohne Wertrückgabe). Mischformen sind unerwünscht, da sie die Übersichtlichkeit von Programmen zerstören.

- Bei Routinenaufrufen und Zuweisungen gilt ein strenges Typkonzept, das nur durch Konformitätsregeln (Definition 3.8.5 auf Seite 102) aufgeweicht werden kann.

- In Schleifen, Routinen und an jeder Stelle einer Anweisungsfolge können Zusicherungen plaziert werden, die Aussagen über die an dieser Stelle zu erwartenden Programmeigenschaften formulieren und – im Rahmen der Zusicherungssprache – über[prüfbar sind.

- Die Behandlung von Ausnahmen wird in kontrollierter Form außerhalb des eigentlichen Routinenrumpfes vorgenommen und erlaubt kein stillschweigendes Ignorieren von Fehlern.

- Anweisungen, die nur für Kontrollzwecke und zum Aufspüren von Fehlern relevant sind, können als separate debug-Anweisungen formuliert werden.

- Ein- und Ausgaberoutinen sind Dienstleistungen einer speziellen Klasse und keine vordefinierten Programmiersprachenelemente.

Die Art der Anweisungen ist nicht anders als in anderen imperativen Programmiersprachen. Es gibt Zuweisungen, Prozedur- und Funktionsaufrufe, bedingte Anweisungen und Schleifen. Im Vergleich zu anderen Programmiersprachen wirken manche Anweisungsformen zuweilen weniger elegant oder sogar etwas umständlich. Der große Vorteil dieser Konzeption ist jedoch, daß die scheinbaren Einschränkungen einen Programmierer zwingen, sich über Struktur, Korrektheit und Robustheit Gedanken zu machen, bevor der Programmtext niedergeschrieben wird. Ein "Hacken" von Programmen, was von Sprachen wie C, Fortran, Basic (und z.T. auch Pascal) geradezu gefördert wird, wird zwar nicht unmöglich gemacht, aber doch erschwert.

Darüber hinaus können stilistische Standards wesentlich zum Verständnis eines Programmtextes beitragen. Sinnvoll plazierte Kommentare, die das erfassen, was sich in Eiffel-Zusicherungen nicht formulieren läßt, ein übersichtlich gestaltetes Layout des Textes, eine konsistente Verwendung von Groß- und Kleinschreibung (Eiffel unterscheidet dies nicht) und eine sinnvolle Namenswahl bei Klassen und features sind von großer Bedeutung. Einige wertvolle Hinweise hierzu gibt [Meyer, 1988, Kapitel 8.1/2].

Zu jeder Anweisung gibt es Verifikationsregel, die besagt, wie mann die Korrektheit einer Anweisung beweist, wenn man die Korrektheit ihrer Bestandteile – soweit vorhanden – schon bewiesen hat (oder als bewiesen annimmt). Diese rein syntaktischen Regeln sind formal genug, um mit Rechnerunterstützung angewandt werden zu können. Dies erleichtert eine computerisierte *Überprüfung* eines fertigen Korrektheitsbeweises. Das Finden eines solchen Beweises kann jedoch noch nicht automatisiert werden, wenn Schleifen oder Rekursion im Anweisungsteil einer Routine vorkommen. Daher erfordert ein Korrektheitsbeweis neben dem sturen Anwenden von Regeln immer ein gewisses Maß von Einfallsreichtum.

4.3.12 Sprachbeschreibung

Zur Präzisierung der bisher gegebene Beschreibung von Anweisungen wollen wir nun die Sprachbeschreibung aus Sektion 3.11 um die Syntax der Anweisungen ergänzen.[39] Diese wird dann in der nächsten Sektion durch die Syntax der Ausdrücke vervollständigt werden.

4.3.12.1 Syntax

Compound	::=	[Compounds]	Siehe Abschnitt 3.11.1
Compounds	::=	Instruction [`;` Compound]	*Folgen von Anweisungen*
Instruction	::=	Creation	*Initialisierung*
	|	Call	*Routinenaufruf*
	|	Entity `:=` Expression	*Wertzuweisung*
	|	Entity `?=` Expression	*Zuweisung*
	|	Conditional	*Bedingte Anweisung*
	|	Multi_branch	*Fallunterscheidung*
	|	Loop	*Schleife*
	|	Debug	*Anweisung im DEBUG-mode*
	|	**check** [Assertions] **end**	*Prüfung einer Zusicherung*
	|	**retry**	
Entity	::=	Identifier | `Result`	
Creation	::=	`!`[Type]`!` Writable [`.`Unqualified_call]	

[39] Die präzise Beschreibung der Semantik wird in diesem Semester zugunsten von Systematik und Verifikation ausgelassen.

Unqualified_call	::=	Identifier [`(` [Actuals] `)`]
Actuals	::=	Actual [`,` Actuals]
Actual	::=	Expression | `$` Identifier
Call	::=	[`(` Expression `)` `.`] Call_chain
Call_chain	::=	Unqualified_call [`.` Call_chain]
Conditional	::=	**if** Then_part_list [Else_part] **end**
Then_part_list	::=	Then_part [**elseif** Then_part_list]
Then_part	::=	Expression **then** Compound
Else_part	::=	**else** Compound
Multi_branch	::=	**inspect** Expression [When_part_list] [Else_part] **end**
When_part_list	::=	**when** Whenpart [When_part_list]
When_part	::=	Choices **then** Compound
Choices	::=	Choice [`,` Choices]
Choice	::=	Manifest_constant
	|	Identifier
	|	Integer_constant `..` Integer_constant
	|	Character_constant `..` Character_constant
Loop	::=	**from** Compound [**invariant** [Assertions]] [**variant** [Identifier `:`] Integer_expression] **until** Expression **loop** Compound **end**
Debug	::=	**debug** Compound **end**

4.4 Ausdrücke: Grundbausteine von Eiffel-Programmen

In jeder modernen Programmiersprache wird die Berechnung von numerischen Werten unterstützt und die Auswertung einfacher (Boolescher) logischer Formeln bereitgestellt. Da das eigentliche Rechnen fast ausschließlich in der Auswertung arithmetischer und Boolescher Ausdrücke besteht, bilden Ausdrücke (*expressions*) die Grundbausteine aller Programmierung.[40] Im wesentlichen wird für die Bildung von Ausdrücken auf allgemein übliche Notationen zurückgegriffen und die meisten Ausdrücke sind eigentlich selbsterklärend. Die einzige Einschränkung besteht darin, daß man sich auf den Zeichensatz, der auf der Computertastatur zur Verfügung steht, beschränken muß und daß jeder Ausdruck in einer (beliebig langen) Zeile niedergeschrieben werden können muß.

Daraus ergibt sich, daß boolesche Junktoren wie $\wedge$, $\vee$, $\Rightarrow$ und $\neg$ durch Schlüsselworte **and**, **or**, **implies** und **not** ersetzt werden müssen, Vergleichsoperatoren wie $\geq$, $\leq$, $\neq$ durch Doppelsymbole `<=`, `>=` und `/=` und Funktionen wie $\sqrt{\ }$ durch einen Namen (`sqrt`. Da Bruchstriche nicht erkannt werden, muß eine Division `Fracab` durch einen Schrägstrich getrennt nebeneinander geschrieben werden: `a/b`. Ähnliches gilt für Exponenten. Statt x^3 müssen wir `x^3` schreiben. Damit dies nicht zur Verwirrung führt, spielen Klammern und Prioritätenregelungen ("Punktrechnung geht vor Strichrechnung" etc.) eine große Rolle.

[40] Funktionale Programmiersprachen benutzen sogar fast ausschließlich Ausdrücke, um Algorithmen zu beschreiben. Anstelle von Schleifen werden rekursive Funktionsaufrufe benutzt, bedingte Anweisungen haben als Gegenstück bedingte Ausdrücke und Folgen von Anweisungen die Komposition von Funktionen.

Die Formulierung von Ausdrücken muß präzise sein. Abkürzungen wie `a<b<c`, die in der Mathematik durchaus üblich sind, müssen verboten werden, da die unmittelbare Interpretation `a<(b<c)` oder `(a<b)<c` wäre. In beiden Fällen würde ein boolescher Ausdruck mit einem numerischen Wert verglichen, was aufgrund des Typkonzeptes von Eiffel nicht erlaubt werden kann. Gemeint ist `a<b` $\wedge$ `b<c` und so ist es auch zu schreiben.

Ausdrücke werden in Zuweisungen, in Zusicherungen, als Argumente von Prozeduraufrufen und zur Qualifizierung von Aufrufen benötigt. Sie bestehen im wesentlichen aus den folgenden Bestandteilen: *Konstanten, Größen,* `Current`*, Funktionsaufrufe und Ausdrücke mit Operatoren.*

Wir werden im folgenden die wichtigsten Merkmale von Ausdrücken kurz zusammenfassen. Genauere Informationen über erlaubte Ausdrücke findet man in [Meyer, 1992, Kapitel 23], Informationen über Bezeichnung und Bedeutung bereits vordefinierter Funktionen in den Klassenbeschreibungen der Eiffel-Basisbibliothek.

4.4.1 Konstanten

Zu jedem Basistyp von Eiffel gibt es vordefinierte Konstanten

- Es gibt zwei Boolesche Konstanten **true** und **false**.
- `INTEGER`-konstanten sind Ziffernfolgen der üblichen Form, die `+` und `-` als Vorzeichen haben dürfen wie in `23, -457, +12`.
- `REAL`- und `DOUBLE`-Konstanten dürfen zusätzlich einen Dezimalpunkt benuzten und eine Angabe einer ganzzahligen Potenz von 10, welche durch ein `e` getrennt hinter der eigentlichen Zahl genannt wird.

  Beispiele sind `23.4, -23, +45.03, .985, -99., 2.345e5 (=234500), 23.456e-3(=0.023456)`
- `CHARACTER`-Konstanten bestehen aus einem einzelnen Symbol, welches in einfache Hochkommata eingeschlossen ist wie `'a', 'z', '3', '>', '/'`. Spezielle Zeichen wie ein "Return"-Symbol werden – beginnend mit einem Backslash-Symbol "\" –codiert (siehe [Meyer, 1992, Kapitel 25.15]).
- Zeichenkettenkonstanten sind Elemente der Klasse `STRING`. Wegen ihrer großen praktischen Bedeutung gelten Zeichenkettenkonstanten jedoch als Teil der vordefinierten Sprachsyntax. Eine Zeichenkette wird in Anführungszeichen eingeschlossen geschrieben wie in

  ```
  "Dies ist eine Zeichenkette"
  ```

  Soll in dieser Zeichenkette ein Anführungszeichen vorkommen, so muß zur Abgrenzung vom Ende der Zeichenkette, ein Backslash-Symbol direkt davor gestellt werden. Nichtdruckbare Zeichen werden gemäß der Konventionen über `CHARACTER`-Konstanten codiert.

4.4.2 Größen

Größen haben wir bereits in Definition 3.3.3 beschrieben. Erlaubt sind Klassenattribute, lokale Variablen und formale Argumente einer Routine, sowie die vordefinierte Größe `Result`, die für das Ergebnis einer Funktion steht. Größen werden benutzt, um Exemplare einer Klasse zu bezeichnen.

4.4.3 `Current`

Das reservierte Wort `Current` bezeichnet das aktuelle Exemplar einer Klasse, ist aber keine Größe.[41] Eine Konvention der Sprache Eiffel ist, daß in qualifizierten Aufrufen von Merkmalen die explizite Benennung von `Current` entfallen darf. Statt `Current.feature` darf man einfach `feature` schreiben. Dennoch ist die Benutzung von `Current` in in folgenden Fällen notwendig:

[41] Deshalb sind Zuweisungen and `Current` *nicht* erlaubt.

- um eine Kopie des aktuellen Exemplars zu erzeugen (`x:=clone(Current)`),
- um einem Routinenargument einen Verweis auf das aktuelle Exemplar zu übergeben (`x.r(Current)`),
- um zu prüfen, ob eine Größe auf das aktuelle Exemplar verweist (`x=Current`), oder
- um den Typ einer Größe bei Vererbung an das aktuelle Exemplar zu binden (`x:`**like** `Current`).

4.4.4 Funktionsaufrufe

Bisher haben wir die elementaren Grundformen von Ausdrücken vorgestellt. Die Möglichkeit, existierende Ausdrücke zu neuen Ausdrücken zusammenzustellen, wird durch Funktionsaufrufe und Operatoren geboten. Die äußere Form von Funktionsaufrufen haben wir bereits zusammen mit Prozeduraufrufen im Abschnitt 4.3.2 besprochen. Gültige Aufrufe (und damit syntaktisch korrekte Ausdrücke) sind zum Beispiel

```
empty, fib(n), a.fib(23), a.f(g(x,23,z)), a.f(g(x,23,z)).fib(2)
```

Man beachte aber die Bemerkungen über die Rolle formaler Parameter und die Einschränkungen bei entfernten Funktionsaufrufen.

4.4.5 Ausdrücke mit Operatoren

Mit Operatoren können Ausdrücke in anderer Form zusammengesetzt werden als durch Funktionsaufrufe mit Klammerung der Argumente. Es gibt *einstellige* Operatoren, die *vor* einen Ausdruck gesetzt werden können (*prefix*) und zweistellige Operatoren, die *zwischen* zwei Ausdrücke gesetzt werden können (*infix*).

- Die wichtigsten vordefinierten einstellige Operatoren sind `+` und `-`, die vor `INTEGER`- und `REAL` Ausdrücken stehen dürfen, und **not**, was auf Boolesche Ausdrücke anwendbar ist. Darüber hinaus dürfen beliebige (nullstellige) Funktionen als Prefix-Operatoren deklariert wie zum Beispiel in der Deklaration

  prefix `"+":NUMERIC`

  Eine solche Deklaration vereinbart das Zeichen `+` als freien Operator, der auf ein Objekt vom Typ `NUMERIC` angewandt werden darf.

- Die wichtigsten zweistelligen Operatoren sind die Vergleichsoperatoren `=, /=, <, >, <=` und `>=`, deren Operanden vom Typ `INTEGER`, `REAL` oder `CHARACTER` sein dürfen. Ausdrücke mit diesen Operatoren bilden einen Booleschen Ausdruck. Aus diesem Grunde akzeptieren diese Operatoren – ebenso wie der Exponentialoperator ^ – *genau* 2 Operanden.

  Anders ist dies bei den numerischen Operanden `+,-,*,/` und den Booleschen Junktoren **and**, **or**, **implies**, **xor**, **and then** und **or else**.[42] Diese dürfen auch in mehrstelligen Ausdrücken benutzt werden wie in

  `4+3*15/5` und `4=5` **and** `3+4=7` **or** `7<12`.

  Als Infixoperator dürfen auch beliebige einstellige Funktionen deklariert werden wie zum Beispiel in

  `item,` **infix** `"@"(i:INTEGER):X`,

  die innerhalb der Klasse `ARRAY[X]` das Symbol `@` als Infixoperator vereinbart, der zwischen ein Feld und einen Index gestellt werden kann, wie in `a@i`.

Für die Klasse `ARRAY` wurde mittlerweile ebenfalls eine Kurzschreibweise als Sprachbestandteil vereinbart.

```
<<23, n+4, 36, n-2>>
```

bezeichnet ein Feld mit 4 Komponenten und Indexgrenzen 1 und 4. Auf diese Art erspart man sich, bei einfachen Feldern erst eine Initialisierung durchzuführen und dann mühsam alle Komponenten mit `put` aufzufüllen.

[42] Man beachte, daß **and then** und **or else** im Gegensatz zu **and** und **or** nicht *kommutativ* sind, also die Reihenfolge bei der Auswertung eine Rolle spielt.

4.4.6 Sprachbeschreibung

Die nun folgende Syntax von Eiffel-Ausdrücken schließt die Sprachbeschreibung von Eiffel ab. Die hier und in den Abschnitten 3.11.1 und 4.3.12.1 angegebene Grammatik beschreibt jedoch nur die grundsätzliche Form von Eiffel-Programmen. Aufgrund der bisher besprochenen Typeinschränkungen und anderer Restriktionen mögen bestimmte Programmstücke zwar syntaktisch korrekt, aber dennoch nicht akzeptabel sein.

4.4.6.1 Syntax

Identifier	::=	Letter [Letters_or_digits]	Siehe Abschnitt 3.11.1
Letters_or_digits	::=	Letter [Letters_or_digits]	
	|	Digit [Letters_or_digits]	
	|	_ [Letters_or_digits]	
Expression	::=		Siehe Abschnitt 3.11.1
		Call	*Funktionsaufruf*
	|	Expression Comparison Expression	*Vergleich*
	|	Constant	
	|	`Result`	
	|	`Current`	
	|	`<<` [Expressions] `>>`	*Explizite Feldangabe*
	|	**strip** `(` [Attributes] `)`	
	|	**old** Expression	
	|	Unary Expression	
	|	Expression Binary Expression	
	|	`(` Expression `)`	
Comparison	::=	`=` | `/=`	
Expressions	::=	Expression [`,` Expressions]	
Attributes	::=	Identifier [`,` Attributes]	
Prefix_Operator	::=	Unary | Free_Operator	Siehe Abschnitt 3.11.1
Infix_Operator	::=	Binary | Free_Operator	Siehe Abschnitt 3.11.1
Unary	::=	**not** | `+` | `-`	
Binary	::=	`+` | `-` | `*` | `/` | `//` | `\\` | `^` | `<` | `>` | `<=` | `>=`	
	|	**and** | **or** | **xor** | **and then** | **or else** | **implies**	
Manifest_constant	::=	Boolean_constant	Siehe Abschnitt 3.11.1
	|	Character_constant	
	|	Integer_constant	
	|	Real_constant	
	|	Bit_constant	
	|	Manifest_string	
Boolean_constant	::=	`true` | `false`	
Character_constant	::=	`'` Character `'`	
Integer_constant	::=	[Sign] Digits	
Sign	::=	`+` | `-`	
Real_constant	::=	[Sign] Mantisse [`e` [Sign] Digits]	
Mantisse	::=	Digits `.`	
	|	`.` Digits	
	|	Digits `.` Digits	
Bit_constant	::=	Bit_sequence	

Simple_string	::=	" [Nonempty_String]"	Siehe Abschnitt 3.11.1
Nonempty_String	::=	ASCII [Nonempty_String]	
Digits	::=	Digit [Digits]	
ASCII	::=	*Alle ASCII Symbole*	
Letter	::=	A | ... | Z | a | ... | z	
Digit	::=	0 | ... | 9	

4.4.7 Diskussion

Mit den Ausdrücken sind wir auf dem niedrigsten Niveau der Sprache Eiffel angelangt. Unter diesem gibt es nur noch das Niveau von Sprachen, denen nicht einmal die meisten arithmetischen Operationen bekannt sind, und die noch näher an den wirklichen Fähigkeiten der Prozessoren orientiert sind. Diese Assemblersprachen sind einer der Schwerpunkte des zweiten Semesters.

4.5 Systematische Implementierung von Routinen

Bisher haben wir uns im wesentlichen mit den Sprachkonzepten, ihrem Verwendungszweck und den zugehörigen Verifikationsregeln befaßt und an Beispielen illustriert, wie man Programme nachträglich verifizieren kann. In den Beispielen 4.3.8 und 4.3.9 auf Seite 156 bzw. 161 haben wir versucht, die Implementierung einer einfachen Routine mit den besprochenen Hilfsmitteln systematisch zu entwickeln, um so den Korrektheitsbeweis zu erleichtern.

Wir haben in diesen Beispielen bereits implizit eine Reihe von Prinzipien und Methoden angewandt, die auf Grundgedanken der Programmiermethodologie zurückgehen – allerdings ohne sie besonders hervorzuheben. Wir wollen dies zum Abschluß dieses Semesters nachholen und kurz einige der wichtigsten Grundsätze der systematischen Programmierung[43] ansprechen. Ausführlichere Abhandlungen zu diesem Thema findet man in [Dijkstra, 1976, Gries, 1981]. Für Interessierte besonders zu empfehlen ist [Gries, 1981, Kapitel 13-18], wo man neben einer exzellenten Erklärung auch eine Vielfalt von Beispielen finden kann.

4.5.1 Allgemeine Prinzipien

Eines der größten Probleme bei der Programmierung ist, daß viele Programmierer keine klare Vorstellung davon haben, was Korrektheit wirklich bedeutet und wie man *beweisen* kann, daß ein Programm tatsächlich korrekt ist. Das Wort "Beweis" hat für die meisten einen unangenehmen Beigeschmack, bedeutet aber zunächst einmal nicht mehr als *ein Argument, das den Leser von der Wahrheit eines Sachverhalts überzeugt*. Dies verlangt im Prinzip weder einen Formalismus noch eine mathematischen Vorgehensweise. Daß die derzeit vorherrschende Methode, sich von der Korrektheit eines Programms zu überzeugen, jedoch unzureichend ist, zeigt die Tatsache, daß Programmierer einen Großteil ihrer Zeit damit zu verbringen, Fehler aus ihren Programmen zu eliminieren – und dies, obwohl sie von der Richtigkeit ihrer Ideen überzeugt waren.

Ein Teil dieses Problems liegt darin begründet, daß Spezifikationen von Programmen oft zu wenig präzisiert werden und deshalb der Korrektheitsbegriff unklar bleibt. Ein zweiter Grund ist, daß Programmierer zu wenig mit Methoden vertraut sind, die – wie der in Abschnitt 4.2 und 4.3 vorgestellte Kalkül – eine präzise Beweisführung unterstützen und weder formale noch informale mathematische Beweise führen können.

Gute und korrekte Programme können jedoch nur auf der Basis eines durch Vor- und Nachbedingungen geschlossenen Vertrages implementiert werden, wobei die Entwicklung der Implementierung niemals losgelöst von der Formulierung einer Beweisidee geschehen darf. Es ist einfach zu schwer, ein bereits existierendes

[43] Wenn im folgenden von *Programmierung* die Rede ist, dann ist mehr die *Implementierung* von Routinen gemeint und weniger der Entwurf der Klassenstruktur, den wir in Sektion 4.1 besprochen hatten.

Programm nachträglich als korrekt nachzuweisen. Auch ist es effizienter, die Einsichten, die sich aus den Beweisideen ergeben, in die Implementierung einfließen zu lassen. Als Hilfsmittel zum Aufbau von Beweis und Programm können die schwächsten Vorbedingungen gelten, die wir zu jedem Programmkonstrukt formuliert haben. Dies wollen wir als erstes wichtiges Prinzip der Programmierung formulieren:

Entwurfsprinzip 4.5.1 (Entwicklung durch Beweisführung)

Ein Programm und sein Korrektheitsbeweis sollten immer gleichzeitig entwickelt werden, wobei der Beweis *die Vorgehensweise bestimmt.*

Auf den ersten Blick scheint dieses Prinzip eine Behinderung der Kreativität zu sein. Daß genau das Gegenteil richtig ist, wollen wir ausführlich an einem Beispiel illustrieren, auf das wir öfter noch zurückkommen werden.

Beispiel 4.5.2 (Maximale Segmentsumme)

Betrachten wir einmal das folgende einfache, aber doch realistische Programmierproblem

> *Zu einer gegebenen Folge $a_1, a_2, \ldots, a_n$ von n ganzen Zahlen soll die Summe $m = \sum_{i=p}^{q} a_i$ einer zusammenhängenden Teilfolge bestimmt werden, die maximal ist im Bezug auf alle möglichen Summen zusammenhängender Teilfolgen $a_,a_{j+1} \ldots, a_k$.*

Derartige zusammenhängende Teilfolgen heißen Segmente und das Problem ist deshalb als das Problem der maximalen Segmentsumme bekanntgeworden. Für die Folge `-3, 2,-5, 3,-1, 2` *ist zum Beispiel die maximale Segmentsumme die Zahl* `4` *und wird erreicht durch das Segment* `3,-1, 2`*.*

Natürlich könnte das Problem dadurch gelöst werden, daß man einfach alle möglichen Segmente und ihre Summen bestimmt und davon die größte auswählt. Die Realisierung dieser Idee in Eiffel *(mit einem vorgegebenen Feld als aktuellem Objekt) wird in Abbildung 4.16 beschrieben.*

```
maxseg:INTEGER
  is
  local p, q, i, sum :INTEGER
  do
    from p := lower             -- variiere untere Grenze p
         Result := item(lower); -- Initialwert zum Vergleich darf nicht Null sein
      until p >= upper
      loop
        from q := p             -- variiere obere Grenze q
          until q > upper
          loop
            from i := p ;       -- Berechne ∑_{i=p}^{q} a_i
                 sum := item(i) -- Initialwert der Summe
              until i = q
              loop
                   i := i+1;
                   sum := sum+item(i)
            end -- sum = ∑_{i=p}^{q} a_i
            if sum > Result
              then Result := sum
            end
            q := q+1
        end;
        p := p+1
    end
end
```

Abbildung 4.16: *Berechnung der maximalen Segmentsumme: direkte Lösung*

Diese Lösung ist jedoch weder elegant noch effizient, da die Anzahl der Rechenschritte für Folgen der Länge n in der Größenordnung von n^3 liegt – also 1 000 Schritte für Folgen der Länge 10 und schon 1 000 000 für Folgen der Länge 100. Es lohnt sich daher, das Problem systematisch anzugehen. Dazu formulieren es zunächst einmal in mathematischer Notation.

Zu einer Folge a der Länge n soll $\underline{M_n} = \texttt{max}(\{\sum_{i=p}^{q} a_i \mid 1 \leq \texttt{p} \leq \texttt{q} \leq \texttt{n}\})$ *berechnet werden.*

Da eine Lösung ohne Schleifen nicht möglich sein wird, bietet es sich an, nach einer induktiven Lösung zu suchen, bei der die Länge der Folge (wie in der direkten Lösung) natürlich eine Rolle spielen wird. Für einelementige Folgen ist nämlich die Lösung trivial – die maximale Segmentsumme ist der Wert des einzigen Elementes – und es besteht eine gewisse Hoffnung, daß wir das Problem einheitlich lösen können, wenn wir die (betrachtete) Folge um ein weiteres Element ergänzen.[44]

Nehmen wir also an, wir hätten für eine Folge der Länge n die maximale Segmentsumme M_n bereits bestimmt. Wenn wir nun ein neues Element a_{n+1} hinzufügen, dann ist die neue maximale Segmentsumme M_{n+1} entweder die Summe eines Segmentes, welches a_{n+1} enthält oder die Summe eines Segmentes, welches a_{n+1} nicht enthält.

Im ersten Fall müssen wir wissen, was die maximale Summe eines Segments ist, welches das letzte Element a_{n+1} enthält. Wir nennen diese Summe L_{n+1} und untersuchen sie später. Im zweiten Fall ist die Lösung einfach, da das letzte Element keine Rolle spielt – sie ist identisch mit M_n. Insgesamt wissen wir also, daß M_{n+1} das Maximum von L_{n+1} und M_n ist.

Nun müssen wir noch L_{n+1} bestimmen, also die maximale Summe einer Teilfolge $a_j, a_{i+1} \ldots, a_{n+1}$. Da wir induktiv vorgehen, können wir davon ausgehen, daß L_n bereits aus dem vorhergehenden Schritt bekannt ist. Ist L_n negativ, dann ist die maximale Summe einer Folge, welche a_{n+1} enthält, die Summe der einelementigen Folge a_{n+1} (jede länge Folge hätte nur eine kleinere Summe). Andernfalls können wir "gewinnen", wenn wir die Folge a_{n+1} um das Segment ergänzen, dessen Summe L_n war, und erhalten L_{n+1} als Summe von L_n und a_{n+1}. Da L_1 offensichtlich a_1 ist, haben wir die Induktion auch verankert.

Dieses Argument[45] *sagt uns genau, wie wir eine Lösung berechnen und als korrekt nachweisen können. Abbildung 4.17 zeigt die Realisierung des Algorithmus samt seiner Invarianten in* Eiffel*, wobei wieder von einem vorgegebenen Feld a als aktuellem Objekt ausgegangen wird.*

```
maxseg:INTEGER
  is
  local n, L_n :INTEGER
  do
    from n := lower;
         Result := item(n);
         L_n := item(n)
      until n >= upper
      invariant n<=upper                  -- ∧ Result=M_n ∧ L_n=L_n
      variant  upper - n
      loop
         if L_n > 0
            then L_n := L_n + item(n+1)
            else L_n := item(n+1)
         end;                         --  L_n = L_{n+1}
         if L_n > Result
            then  Result := L_n
         end;                         -- Result = M_{n+1}
         n := n+1
      end
end
```

Abbildung 4.17: *Berechnung der maximalen Segmentsumme: systematisch erzeugte Lösung*

[44] Wir haben daher den Maximalwert M_n mit einem Index n versehen, der auf die Abhängigkeit von der Länge hinweist.

[45] Die mathematischen Gleichungen, die dieses Argument unterstützen, sind die folgenden.

$M_1 = \max(\{\sum_{i=p}^{q} a_i \mid 1 \le p \le q \le 1\}) = \sum_{i=1}^{1} a_i = a_1$

$M_{n+1} = \max(\{\sum_{i=p}^{q} a_i \mid 1 \le p \le q \le n+1\}) = \max(\{\sum_{i=p}^{q} a_i \mid 1 \le p \le q \le n\} \cup \{\sum_{i=p}^{q} a_i \mid 1 \le p \le q = n\})$
$= \max(\max(\{\sum_{i=p}^{q} a_i \mid 1 \le p \le q \le n\}), \max(\{\sum_{i=p}^{n} a_i \mid 1 \le p \le n\})) = \max(M_n, L_{n+1})$

$L_1 = \max(\{\sum_{i=p}^{1} a_i \mid 1 \le p \le 1\}) = \sum_{i=1}^{1} a_i = a_1$

$L_{n+1} = \max(\{\sum_{i=p}^{n+1} a_i \mid 1 \le p \le n+1\}) = \max(\{\sum_{i=p}^{n+1} a_i \mid 1 \le p \le n\} \cup \{\sum_{i=p}^{n+1} a_i \mid 1 \le p = n+1\})$
$= \max(\{\sum_{i=p}^{n} a_i + a_{n+1} \mid 1 \le p \le n\} \cup \{\sum_{i=n+1}^{n+1} a_i\}) = \max(\max(\{\sum_{i=p}^{n} a_i + a_{n+1} \mid 1 \le p \le n\}), \max(\{a_{n+1}\}))$
$= \max(\max(\{\sum_{i=p}^{n} a_i \mid 1 \le p \le n\}) + a_{n+1}, a_{n+1}) = \max(L_n + a_{n+1}, a_{n+1})$

Der vollständige Korrektheitsbeweis ergibt sich direkt aus einer Umsetzung des obigen Argumentes und ist daher nicht mehr sehr schwer. Darüber hinaus ist der Algorithmus auch viel effizienter. Die Anzahl der Rechenschritte für Folgen der Länge n liegt nun in der Größenordnung von n – also nur etwa 300 für Folgen der Länge 100.

Das obige Beispiel zeigt, daß man durch das Prinzip, den Beweis als Leitlinie zur Implementierung zu verwenden, zu erheblich effizienteren und auch leichter zu verifizierenden Lösungen kommt.[46] Am folgenden Beispiel wollen wir nun demonstrieren, daß manche Probleme überhaupt nicht gelöst werden können, wenn man sich nicht zuvor eine Beweisidee zurechtlegt.

Beispiel 4.5.3 (Das Kaffebohnen Problem)

In einer Kaffedose befinden sich helle und dunkle Kaffebohnen. Aus dieser greife man sich zufällig zwei Bohnen heraus. Sind sie gleich, so werden beide herausgenommen und eine dunkle wieder hineingetan (es sind genügend dunkle Bohnen vorhanden). Sind sie unterschiedlich, so wird die helle zurückgelegt und die dunkle herausgenommen. Das Ganze wird wiederholt, bis nur noch eine Bohne in der Dose ist.

Die Frage lautet nun, wie die Anzahl der hellen und dunklen Bohnen damit zusammenhängt, welche Farbe die letze Bohne hat. Was bleibt z.B. übrig, wenn ursprünglich 43 helle und 32 dunkle Bohnen in der Dose waren?

Nach einigen Versuchen stellt sich heraus, daß Experimente bei der Beantwortung dieser Frage überhaupt nicht weiterhelfen. Es gibt Leute, die Stunden damit vergeudet haben, die verschiedensten Kombinationen durchzuprobieren, ohne zu einem tragfähigen Ergebnis zu kommen.[47]

Das einzige, was hier weiter hilft, ist die Überlegung, daß es irgendeine Eigenschaft geben muß, die durch das Herausnehmen der Bohnen nicht *verändert wird, also invariant bleibt. Diese Eigenschaft, zusammen mit der Tatsache, daß nur eine Bohne übrigbleibt, kann dann die Antwort geben.*

Nun, in jedem Zug verschwinden entweder zwei helle Bohnen aus der Dose oder keine und entweder verschwindet eine dunkle Bohne oder es kommt eine hinzu. Nach einigem Nachdenken fällt auf, daß Zwei und Null jeweils gerade Zahlen sind. Das bedeutet, daß bei einer geraden Anzahl von hellen Bohnen nach einem Zug wieder eine geraden Anzahl von hellen Bohnen in der Dose liegt und ähnliches für eine ungerade Anzahl gilt. Die Eigenschaft, ob eine gerade oder ungerade Anzahl heller Bohnen vorhanden ist, bleibt also bei jedem Zug unveränderlich.

Als Konsequenz wissen wir nun, daß zum Schluß genau dann eine dunkle Bohne übrigbleibt, wenn zu Beginn eine gerade Anzahl von hellen Bohnen in der Dose war.

Beide Beispiele zeigen, wie wichtig es ist, die Eigenschaften derjenigen Objekte zu kennen, mit denen das Programm zu tun haben soll. Je mehr Zusammenhänge man formulieren kann, um so größer ist die Chance, ein gutes und effizientes Programm zu schreiben.

Entwurfsprinzip 4.5.4

Mache Dir die Eigenschaften der Objekte klar, die von einem Programm manipuliert werden sollen.

Auch wenn wir die Bedeutung von Korrektheitsbeweisen besonders hervorheben, sei dennoch darauf hingewiesen, daß eine vollständige Formalisierung der Beweise in einem Kalkül meist weder notwendig noch erstrebenswert ist, solange keine Werkzeuge zur Unterstützung des Kalküls bereitstehen. Zu viel Formalismus führt dazu, daß die wesentlichen Ideen in einem Übermaß an unverständlichen Details verlorengehen.

[46] Man mag hier natürlich einwenden, daß man eventuell auch auf anderen Wegen zu einer ähnlich guten Lösung gekommen wäre. Ich persönlich halte dies aber für sehr unwahrscheinlich.

[47] Dies sagt auch etwas aus über die Aussagekraft von Tests bei der Analyse von Programmen. Man kann Tausende von Testläufen durchführen ohne dadurch eine sichere Aussage über die Korrektheit des Programms zu gewinnen. Man kann mit Tests oft nicht einmal den Gründen für die Fehler auf die Spur kommen, die man am äußeren Verhalten bereits entdeckt hat.

Andererseits darf man sich aber auch nicht zu sehr auf seine Intuition und den gesundem Menschenverstand verlassen, da hierdurch zu viele schlechte Entwürfe und Fehler entstehen.

Angestrebt werden sollte stattdessen ein ausgewogenes Gleichgewicht zwischen den beiden. Offensichtliches[48] sollte unerwähnt bleiben, wichtige Punkte hervorgehoben werden und soviel Details hinzugefügt werden, wie für ein Verständnis nötig sind. Hierzu muß eine weniger formale, aber konsistente Notation eingeführt werden. Nur beim Auftreten von Schwierigkeiten sollte auf den genauen Formalismus zurückgegriffen werden.

Dieses Gleichgewicht, das schon in der Mathematik von großer Bedeutung ist, ist für die Programmierung besonders wichtig. Programme enthalten so viele Details, die *absolut* korrekt sein müssen. Auf der anderen Seite sind sie oft so groß, daß ein Einzelner sie kaum noch lesen kann.

Entwurfsprinzip 4.5.5 (Formalismus und gesunder Menschenverstand)

> *Benutze die Theorie, um Erkenntnisse zu erhalten; verwende Intuition, wo es angebracht ist; greife auf Formalismen als Hilfsmittel zurück, sobald Schwierigkeiten oder komplizierte Details auftreten.*

Dies erfordert natürlich Erfahrung – sowohl im Umgang mit der formalen Theorie als auch mit dem Einsatz von Intuition bei Begründungen. Es lohnt, sich Erfahrung mit Formalismen dadurch anzueignen, indem man die vielen kleinen Routineprogramme, die man zu schreiben hat, sehr sorgfältig implementiert und verifiziert. Der Nebeneffekt dieser Vorgehensweise ist, daß eine der größten Fehlerquellen von Softwaresystemen frühzeitig eliminiert wird.[49]

Erfahrung kann im Endeffekt durch nichts ersetzt werden. Man kann sich noch so viel Wissen durch Lesen und Zuhören aneignen – um wirklich zu lernen, muß man selbst aktiv werden und die Prinzipien *anwenden*. Die Ideen mögen naheliegend und leicht einzusehen sein, aber Ihre Anwendung kann ohne Training sehr mühsam sein. Ein Prinzip zu akzeptieren und es umzusetzen sind zwei verschiedene Dinge.

Entwurfsprinzip 4.5.6 (Bewußtes Anwenden von Prinzipien)

> *Ein leicht einzusehendes Prinzip darf niemals als "offensichtlich" verworfen werden, da nur ein* bewußtes Anwenden *von Prinzipien zum Erfolg führt.*

4.5.2 Programmierung als zielgerichtete Tätigkeit

In unseren Beispielen haben wir die Bedeutung des zielgerichteten Vorgehens in der Programmierung hervorgehoben. Die Zielsetzung, die ein Programm erfüllen soll, also die Nachbedingung, ist erheblich wichtiger als die Situation, von der man ausgeht (die Vorbedingung). Natürlich spielt die Vorbedingung eine Rolle, aber die Nachbedingung liefert erheblich mehr Erkenntnisse. Die Entwicklung eines Fakultätsprogramms ohne das Ziel, die Fakultät zu berechnen, wäre geradezu aberwitzig. Auch die angestrebte Verwendung vorgefertigter Softwaremodule, die in Eiffel besonders unterstützt wird, hat nur dann einen Sinn, wenn man sich die Bestandteile danach zusammensucht, was man eigentlich lösen will.

Entwurfsprinzip 4.5.7 (Zielorientierung)

> *Jede Programmentwicklung muß sich an dem angestrebten Resultat orientieren.*

---

[48] Was "offensichtlich" ist, beruht natürlich ebenfalls auf Erfahrung und dem Umfeld, in dem ein Beweis präsentiert wird. Empfehlenswert ist daher, zunächst extrem detailliert zu arbeiten, und erst später dazu überzugehen, Details wieder zu entfernen. Nach einer Weile wird sich dann herausstellen, mit welchen Argumenten man fehlerfrei umgehen kann ohne alles zu überprüfen.

[49] Wie wichtig dies ist, zeigt folgendes Rechenexempel. 500 einfache Routinen in einem Softwaresystem sind keine Seltenheit. Wenn jede dieser Routinen sich nur in einem von 1000 Fällen falsch verhält, dann tritt im Gesamtsystem schon in jedem zweiten Fall ein Fehler auf. Bei unverifizierten Routinen liegt die Fehlerwahrscheinlichkeit meist sogar höher, nämlich zwischen einem und 5 Prozent. In diesem Fall mach das Gesamtsystem praktisch immer einen Fehler (d.h. in 99,4% bzw. 99,999% der Fälle).

Dies ist die Grundidee der sogenannten *Verfeinerung*. Ausgehend von dem bereits abgeschlossenen Vertrag einer Routine sollte man versuchen, diesen in sinnvolle und überschaubare Teilaufträge zu zerteilen und sich erst dann nach bereits existierenden Teillösungen umschauen. Die Zielorientierung ist auch der Grund, bei der Entwicklung die schwächsten Vorbedingungen und nicht etwa die stärksten Nachbedingungen einzusetzen.

Bevor man zielorientiert vorgehen kann, muß man sich natürlich erst einmal *vollständig* und *unzweideutig* klarmachen, was das zu lösende Problem genau ist. Diese Aussage erscheint trivial, wird aber doch sehr häufig vernachlässigt.

Entwurfsprinzip 4.5.8 (Präzise Problemstellung)

Vor der Programmsentwicklung müssen Vor- und Nachbedingungen präzisiert und verfeinert werden.

Die Einhaltung dieses Prinzips wird durch die Vertragsmetapher von Eiffel und die in Sektion 4.1 angesprochene Entwurfsmethodik sehr stark unterstützt. Die größte Schwierigkeit liegt jedoch darin, eine Spezifikation gleichzeitig einfach und präzise zu gestalten. Gelingt dies, so ist die nachfolgende Implementierung meist eine leichte Aufgabe.

Die Verwendung natürlicher Sprache oder einer mathematischen Notation für die Spezifikation ist normalerweise ein zu hohes Niveau. Es birgt die Gefahr in sich, daß Begriffe vorkommen, die für den Programmierer nicht eindeutig oder ihm sogar unbekannt sind. Da diese Abstraktionsform jedoch notwendig ist, um die wesentlichen Absichten, *was* das Programm tun soll, zu vermitteln, empfiehlt es sich, eine durch mathematische Notationen ergänzte Sprache zu verwenden, in der zu jedem "Nichtstandard"-Begriff eine genaue Definition gegeben wird, auf die man im Zweifelsfall zurückgreifen kann. Dadurch werden Spezifikationen übersichtlich und bleiben dennoch präzise.

Beispiel 4.5.9 (Maximum)

Bei der Berechnung der maximalen Segmentsumme in Beispiel 4.5.2 haben wir uns eines Programmstücks bedient, welches das Maximum zweier Zahlen berechnet, und sind implizit davon ausgegangen, daß es klar ist, was "Maximum" bedeutet. Wir wollen nun eine Definition dieses Begriffs nachholen und das zugehörige Programmstück systematisch entwickeln.

Eine Zahl z ist das Maximum zweier Zahlen x und y, wenn z die größere der beiden Zahlen ist. Ausgedrückt mit Mitteln der Logik heißt dies zum Beispiel:

`z = max(x,y)` $\equiv$ `z`$\geq$`x` $\wedge$ `z`$\geq$`y` $\wedge$ `(z=x` $\vee$ `z=y)`[50]

Mit dieser Definition können wir nun ohne weitere Bedenken ein Programmstück zur Berechnung des Maximums in einfacher, aber doch verständlicher Weise wie folgt spezifizieren.

{true} `maximum` **{**`Result=max(x,y)`**}**

Wie können wir nun vorgehen, um anhand der Nachbedingung das Programmstück im Detail zu implementieren? Da keine weiteren Bedingungen angegeben sind als `Result=max(x,y)` *und bisher keine Routine existiert, die diese Spezifikation erfüllt, müssen wir auf die Definition zurückgreifen und ausgehend von den einfachsten Bestandteilen der Nachbedingung unsere Anweisungen festlegen.*

Da die Spezifikation die Bedingung `Result=x` *enthält, wählen wir als ersten Kandidaten für eine Teillösung die Zuweisung* `Result := x` *. Wir bestimmen nun die schwächste Vorbedingung dieser Anweisung für die gegebene Nachbedingung:*

wp`(Result:=x, Result=max(x,y))`
$\equiv$ `x=max(x,y)`
$\equiv$ `x`$\geq$`x` $\wedge$ `x`$\geq$`y` $\wedge$ `(x=x` $\vee$ `x=y)`
$\equiv$ **true** $\wedge$ `x`$\geq$`y` $\wedge$ `(`**true** $\vee$ `x=y)`
$\equiv$ `x`$\geq$`y`

[50] Denkbar wäre auch `z = max(x,y)` $\equiv$ `(x`$\leq$`y` $\Rightarrow$ `z=y)` $\wedge$ `(x>y` $\Rightarrow$ `z=x)`

Wir wissen daher, daß `Result := x` *auf jeden Fall die Nachbedingung erfüllt, wenn* `x≥y` *ist. Es liegt daher nahe, diese Zuweisung in eine bedingte Anweisung einzubetten.*

if `x>=y` **then** `Result := x` **else** `??` **end**

Uns fehlt nun nur noch das Programmstück für den durch `??` *gekennzeichneten Teil. Für diesen kennen wir nun die Vorbedingung* `¬(x≥y)`*. Ein Blick auf die noch nicht betrachtete Teilbedingung* `Result=y` *in der Spezifikation läßt uns hierfür als Kandidaten* `Result := y` *wählen. Analog zu oben ergibt sich hierfür als schwächste Vorbedingung*

wp`(Result:=y, Result=max(x,y))` $\equiv$ `y≥x`

Da diese Vorbedingung von `¬(x≥y)` *impliziert wird, können wir für* `??` *die Zuweisung* `Result := y` *wählen und sind fertig.*

if `x>=y` **then** `Result := x` **else** `Result := y` **end**

ist das gesuchte Programmstück, welches das Maximum der Zahlen `x` *und* `y` *berechnet.*

Die Entwicklung der Lösung in diesem Beispiel illustriert eine einfache Methode, um einfache (bedingte) Anweisungen aus einer gegebenen Nachbedingung zu entwickeln.

Entwurfsprinzip 4.5.10 (Strategie zur Erzeugung bedingter Anweisungen)

Ein Programmstück sei spezifiziert durch eine Vorbedingung `pre` *und die Nachbedingung* `post`*. Eine Verfeinerung dieses Programmstücks zu einer bedingten Anweisung kann durch folgende Strategie erzielt werden.*

1. *Suche eine Instruktion* `Anweisung`*, deren Ausführung die Nachbedingung in einigen Fällen erfüllt.*
2. *Bestimme eine Bedingung* `B` *für die gilt* $\{$`pre`$\wedge$`B`$\}$ `Anweisung` $\{$`post`$\}$*.*
3. *Erzeuge das Programmstück* **if** `B` **then** `Anweisung` **else** `??` **end** *bzw., wenn* `B` *den Wert* **true** *hat, das Programmstück* `Anweisung`*.*
4. *Für die noch zu bestimmende Anweisung* `??` *setze* `pre'` $\equiv$ `pre`$\wedge\neg$`B` *und wiederhole die Strategie, bis die in (2) gefundene Bedingung* **true** *wird.*

Bei der Anwendung dieser Strategie kann es durchaus sein, daß einige der Anweisungen mittels der unten besprochenen Strategie zur Erzeugung von Schleifen erzeugt werden müssen. Beide Methoden können sehr oft erfolgreich eingesetzt werden, obwohl sie verhältnismäßig einfach sind.

4.5.3 Entwurf von Schleifen

In unseren Beispielen haben wir gesehen, wie wichtig es ist, die Entwicklung von Schleifen auf eine Formulierung der Grundidee in Form von Varianten und Invarianten zu stützen.

Beispiel 4.5.11 (Summe einer Folge)

Bei der direkten Lösung zur Berechnung der maximalen Segmentsumme in Beispiel 4.5.2 haben wir ein Programmstück verwendet, welches die Summe der Elemente einer Folge a von ganzen Zahlen berechnet. Auch dieses Programmstück wollen wir nun systematisch entwickeln. Die Spezifikation dieses Programmstücks lautet:

$\{$`p≤q`$\}$ `summe` $\{$`sum=`$\sum_{i=p}^{q} a_i\}$

Grundidee der zu erzeugenden Schleife ist es, den Index `i` *von* `p` *bis* `q` *laufen zu lassen und dabei jeweils die Teilsumme* $\sum_{k=p}^{i} a_k$ *berechnet zu haben. Dies liefert*

variant `q-i`
invariant `p≤i≤q` $\wedge$ `sum=`$\sum_{k=p}^{i} a_k$

Als Initalanweisung genügt es, `i` *mit der unteren Feldgrenze zu belegen und die Summe* `sum` *entsprechend auf* `item(i)` *zu setzen. Danach gilt die Invariante und die Variante ist nicht negativ.*

`Init` $\equiv$ `i:=p ; sum:=item(i)`.

Als nächstes bestimmen wir die Abbruchbedingung der Schleife. Da `Inv` $\wedge$ `Abbruch` $\Rightarrow$ `post` *gelten muß, vergleichen wir die Invariante mit der Nachbedingung und kommen zu dem Schluß, daß* `i=q` *(oder auch* `i`$\geq$`q`*) für diesen Zweck genügt.*

`Abbruch` $\equiv$ `i=q`.

Zum Schluß bestimmen wir die Schleifenanweisung. Diese muß die Variante verringern, um Terminierung zu erreichen, und die Invariante erhalten, um Korrektheit zu garantieren. Um die Variante zu verringern, müssen wir `i` *erhöhen, weil wir* `p` *und* `q` *nicht ändern können. Der Schritt* `i:=i+1` *alleine würde allerdings die Invariante zerstören. Um diese wieder herzustellen, müssen wir den Summenwert anpassen. Wegen* $\sum_{k=p}^{i+1} a_k$ `=` $\sum_{k=p}^{i} a_k$ `+` a_{i+1} *reicht es,* a_{i+1} *aufzuaddieren. Wir erhalten*

`Anweisung` $\equiv$ `i := i+1; sum := sum+item(i)`

und haben damit alle Komponenten der Schleife beisammen. Insgesamt ergibt sich also folgendes Programmstück zur Berechnung der Summe der Elemente einer Folge a von ganzen Zahlen.

```
from i := p ;
     sum := item(i)
 until i = q
 loop
     i := i+1;
     sum := sum+item(i)
end
```

In diesem Beispiel wird das angestrebte Gleichgewicht zwischen Formalismus und Intuition besonders deutlich. Vor- und Nachbedingung sowie Variante und Invariante wurden formal präzise beschrieben. Die Entwicklung der einzelnen Programmteile geschah dagegen weniger formal, orientierte sich aber an den durch die Verifikationsregel für Schleifen (Abbildung 4.11 auf Seite 161) vorgegebenen Randbedingungen.

Darüber hinaus illustriert das Beispiel aber auch eine einfache Methode, um Schleifen zu entwickeln: die Abbruchbedingung sollte vor der Schleifenanweisung festgelegt werden und innerhalb der Schleifenanweisung sollte zunächst die Verringerung der Variante angestrebt werden. Diese Methodik haben wir auch schon bei der Berechnung der Fakultät mittels einer Schleife im Beispiel 4.3.9 auf Seite 156 angewandt.

Entwurfsprinzip 4.5.12 (Strategie zur Erzeugung von Schleifen)

Ein Programmstück sei spezifiziert durch eine Vorbedingung `pre` *und die Nachbedingung* `post`*. Eine Verfeinerung dieses Programmstücks zu einer Schleife kann durch folgende Strategie erzielt werden.*

1. *Ausgehend von einer groben Beweisidee lege die Invariante* `Inv` *und die Variante* `Var` *fest.*
2. *Bestimme eine Initialanweisung* `Init` *, welche die Invariante erfüllt.*
3. *Wähle eine Abbruchbedingung* `Abbruch`*, für die gilt* `Inv` $\wedge$ `Abbruch` $\Rightarrow$ `post`*.*
4. *Bestimme eine Schleifenanweisung* `Anweisung`*, welche die Variante verringert und die Invariante erhält. Die Verringerung der Variante sollte als erstes verfolgt werden.*
5. *Erzeuge das Programm* **from** `Init` **until** `Abbruch` **loop** `Anweisung` **end**

Offen ist allerdings noch die Frage, wie denn die Invariante und die Variante aus einer vorgegebenen Problemstellung abgeleitet werden kann. Auch hier lohnt sich ein zielorientiertes Vorgehen.

Da wir wissen, daß die Invariante zusammen mit der Abbruchbedingung die Nachbedingung implizieren muß, können wir die Invariante als eine *Abschwächung der Nachbedingung* ansehen. Wir werden also versuchen, die Nachbedingung so abzuschwächen, daß wir eine Invariante erhalten können. Hierzu gibt es eine Reihe von Möglichkeiten.

Entwurfsprinzip 4.5.13 (Invarianten als Abschwächung der Nachbedingung)

Eine Invariante kann durch Abschwächung der Nachbedingung `post` *auf folgende Arten erzeugt werden.*

Entfernen eines Konjunktionsgliedes: *Wenn* `post` *die Form* $\texttt{A} \wedge \texttt{B} \wedge \texttt{C}$ *hat, dann kann* `B` *entfernt werden und* $\texttt{A} \wedge \texttt{B}$ *als Invariante gewählt werden.* `B` *ist dann in Kandidat für die Abbruchbedingung.*

Ersetzen einer Konstanten durch eine Variable:
In der Nachbedingung $\texttt{sum=}\sum_{k=p}^{q} a_k$ *kann zum Beispiel die Konstante* `q` *durch eine neue Variable* `i` *ersetzt werden, deren Wertebereich natürlich beschränkt werden muß.*

$\texttt{sum=}\sum_{k=p}^{q} a_k \quad \equiv \quad \texttt{sum=}\sum_{k=p}^{i} a_k \ \wedge \ \texttt{i=q}$

Erweiterung des Wertebereichs einer Variablen:
In der obigen Bedingung kann `i=q` *erweitert werden zu* $\texttt{p} \leq \texttt{i} \leq \texttt{q}$.

Erweiterung durch Disjunktion: *Als Invariante kann* $\texttt{post} \vee \texttt{A}$ *gewählt werden, wobei* `A` *eine beliebige Bedingung ist.*

Die ersten drei Methoden sind ziemlich sinnvoll, während die letzte nur relativ selten eingesetzt wird. Meist tritt eine Kombination der Methoden auf. In Beispiel 4.5.11 haben wir z.B. erst eine Konstante durch eine Variable ersetzt und dann den Wertebereich dieser Variablen ausgedehnt. Eine Reihe weiterer Beispiele und ergänzender Hinweise findet man in [Gries, 1981, Kapitel 16].

Die Variante einer Schleife erfüllt zwei Aufgaben. Einerseits soll sie sicherstellen, daß die Schleife überhaupt terminiert. Zum anderen liefert sie auch eine obere Schranke für die Anzahl der Schritte, die bis zum Abbruch durchgeführt werden. Für ein und dasselbe Problem gibt es daher sehr verschiedene Varianten – je nachdem ob das Interesse eines Programmierers nur in der Terminierung oder auch in einer effizienten Lösung liegt.

Auch wenn die Variante formal nur ein Integer-Ausdruck ist, beschreibt sie dennoch eine Eigenschaft des zu erzeugenden Programms. So ist z.B. bei unserem Programm zur Summierung von Folgenelementen die Variante die Anzahl der noch nicht aufsummierten Elemente, die in jedem Schritt geringer werden soll. Die Tatsache, daß diese Anzahl stets positiv ist, ergibt sich aus der Invarianten. Es lohnt sich daher, diese Eigenschaft zunächst informal zu beschreiben und dann mithilfe der bisher formulierten Eigenschaften in einen Integer-Ausdruck umzuwandeln.

Entwurfsprinzip 4.5.14 (Strategie zum Erzeugen der Variante)

Beschreibe die Variante zunächst informal als eine Eigenschaft, die sich aus der Invariante und der Spezifikation ergibt. Formalisiere sie dann als Integer-Ausdruck.

4.5.4 Sinnvoller Einsatz von Rekursion

Wir haben im Abschnitt 4.3.10 gesehen, daß der Effekt von Schleifen genauso gut durch die Deklaration rekursiver Routinen erreicht werden kann. Rekursion ist ein nützliches Hilfsmittel, das unbedingt zum Repertoire eines guten Programmierers gehören sollte, da manche Probleme sich auf diese Art sehr viel eleganter lösen lassen als durch Schleifen.

Es gibt zwei wichtige Programmierstrategien, bei denen Rekursion besonders häufig auftritt. Man kann eine Aufgabenstellung dadurch lösen, daß man einen Teil davon auf ein bekanntes Problem zurückführt und den Rest als einfachere Variante der ursprünglichen Aufgabenstellung ansieht. Man kann aber auch das Problem so aufteilen, daß beide Teile einfachere Varianten der ursprünglichen Aufgabenstellung sind. Diese Strategie ist unter dem Namen *Divide&Conquer* (Teilen und Erobern) bekannt geworden. Wir wollen beide kurz anhand von Beispielen erläutern.

Beispiel 4.5.15 (Vertauschen von Sektionen in Feldern)

In einem Array a *mit Grenzen* lower *und* upper *sollen zwei Sektionen* $\mathtt{S}_l \equiv \mathtt{a}_{lower}, \mathtt{a}_{lower+1}, \ldots, \mathtt{a}_{p-1}$ *und* $\mathtt{S}_r \equiv \mathtt{a}_p, \mathtt{a}_{p+1}, \ldots, \mathtt{a}_{upper}$ *miteinander vertauscht werden. D.h. aus dem Feld*

$\mathtt{a}_{lower}, \mathtt{a}_{lower+1}, \ldots, \mathtt{a}_{p-1}$	$\mathtt{a}_p, \mathtt{a}_{p+1}, \ldots, \mathtt{a}_{upper}$

soll das Feld

$\mathtt{a}_p, \mathtt{a}_{p+1}, \ldots, \mathtt{a}_{upper}$	$\mathtt{a}_{lower}, \mathtt{a}_{lower+1}, \ldots, \mathtt{a}_{p-1}$

entstehen. Da das ursprüngliche Feld sehr groß sein kann, kommt als zusätzliche Einschränkung hinzu, daß der zusätzlich verbrauchte Speicherplatz nur eine konstante Größe haben darf, also nicht etwa ein zweites Feld für Kopierzwecke zur Verfügung steht.

Diese Aufgabe wäre sehr einfach zu lösen, wenn die beiden Sektionen gleich groß *wären. In diesem Fall könnte man nämlich einfach* $\mathtt{a}_{lower}$ *mit* $\mathtt{a}_p$ *tauschen,* $\mathtt{a}_{lower+1}$ *mit* $\mathtt{a}_{p+1}$*, usw. Es ist also nicht sehr schwer, eine Routine* swapequals *zu schreiben, welche Sektionen gleicher Größe tauscht.*

Die Tatsache, daß ein Spezialfall unseres Problems leicht zu lösen ist, wollen wir uns bei der Konstruktion eines Algorithmus zunutze machen. Nehmen wir einmal an, die linke Sektion $\mathtt{S}_l$[51] *wäre größer als die rechte* $\mathtt{S}_r$ *(andernfalls läßt sich das Argument umkehren). Dann können wir sie aufspalten in zwei Teile* $\mathtt{S}_{l1}$ *und* $\mathtt{S}_{l2}$*, von denen der erste genauso groß ist wie* $\mathtt{S}_r$*:*

$\mathtt{S}_{l1}$	$\mathtt{S}_{l2}$	$\mathtt{S}_r$

Wir tauschen nun $\mathtt{S}_{l1}$ *und* $\mathtt{S}_r$ *mithilfe der Routine* swapequals *und erhalten*

$\mathtt{S}_r$	$\mathtt{S}_{l2}$	$\mathtt{S}_{l1}$

.

Da $\mathtt{S}_r$ *nun an der richtigen Stelle steht, brauchen wir nur noch das* $\mathtt{S}_{l2}$ *und* $\mathtt{S}_{l1}$ *zu tauschen. Dies ist das selbe Problem wie das ursprüngliche, wobei jedoch die zu tauschenden Sektionen kleiner geworden sind. Wir können also das gleiche Verfahren erneut aufrufen und tun dies solange, bis die Sektionen gleich lang sind, was spätestens dann eintritt, wenn die Sektionen die Länge Eins erreicht haben.*

Damit ist die wesentliche Idee des Algorithmus – der übrigens eine sehr starke Verwandtschaft zu der Berechnung des größten gemeinsamen Teilers zweier Zahlen hat – klar. Beim vollständigen Ausprogrammieren muß man nur noch berücksichtigen, daß sich bei jedem Schritt die Feldgrenzen ändern und ebenso der Index, an dem die beiden Sektionen aneinanderstoßen.

Das Beispiel illustriert die Grundidee der folgenden Strategie

Entwurfsprinzip 4.5.16 (Entwurf durch Reduktion auf leichtere Probleme)

Versuche, ein Problem so aufzuspalten, daß zunächst die Lösung eines bereits bekannten einfacheren Problems angewandt werden kann und der Rest eine (kleinere) Variante des ursprünglichen Problems beschreibt.

Je nach Problemstellung kann "einfacher" dabei sehr verschiedene Bedeutungen haben. Im obigen Beispiel handelte es sich um einen Spezialfall, d.h. daß Problem wurde zusätzlich auf Sektionen gleicher Länge beschränkt. Im Beispiel 4.5.2 der maximalen Segmentsumme (Seite 179) hat es sich gelohnt, das Problem dahingehend zu erweitern, daß gleichzeitig die maximale Sume von Segmenten berechnet wurde, die bis zum rechten Ende des Feldes reichen – das Problem wurde dadurch leichter. Ein Problem mag einfacher werden, wenn man es zunächst verallgemeinert (einige Beschränkungen aufhebt) und aus der allgemeinen Lösung die Gesuchte als Spezialfall ansieht. Dies ist zum Beispiel der Fall, wenn man die Reihenfolge der Elemente eines Feldes $\mathtt{a}_1, \ldots, \mathtt{a}_n$ Elementen umdrehen möchte: man schreibt zunächst eine Prozedur, welche beliebige Segmente mit Grenzen links und rechts umdreht, und wendet diese dann an auf die Grenzen 1 und n.[52]

[51] Bei kompliziert zu beschreibenden Sachverhalten lohnt es sich immer, Namen zur Abkürzung einzuführen und auf die genaue Definition erst dann zuzugreifen, wenn Detailfragen gelöst werden müssen. Die Beschreibung der Grundidee des Algorithmus wäre erheblich undurchsichtiger, wenn wir die Indizes die ganze Zeit mitschleppen würden.

[52] Ein immer noch sehr empfehlenswertes Buch über allgemeine Methoden zum Lösen von Problemen ist [Polya, 1945].

Beispiel 4.5.17 (Quicksort)

In einem Array `a` *sollen die Elemente in aufsteigender Reihenfolge sortiert werden.*

Das Problem ist sehr leicht zu lösen, wenn `a` *maximal zwei Elemente hat. In diesem Fall genügt ein Vergleich der beiden Elemente, die dann ggf. getauscht werden müssen. Wir versuchen daher, das Sortierproblem schrittweise auf das Sortieren zweielementiger Felder zurückzuführen. Dazu bestimmen wir zunächst ein "mittleres" Feldelement* `x`*. Anschließend teilen wir das Feld in drei Bereiche auf*

$\mathtt{a}_{\leq x}$	x	$\mathtt{a}_{>x}$

Dabei enthält $\mathtt{a}_{\leq x}$ *alle Feldelemente, die kleiner oder gleich* `x` *sind (mit Ausnahme von* `x` *selbst) und* $\mathtt{a}_{>x}$ *alle Feldelemente, die größer als* `x` *sind.*[53] *Um das gesamte Feld in einen sortierten Zustand zu bringen, genügt es nun, die beiden Bereiche* $\mathtt{a}_{\leq x}$ *und* $\mathtt{a}_{>x}$ *unabhängig voneinander zu sortieren. Da beide Bereiche kleiner sind als das ursprüngliche Feld, kann man hierzu einfach das gleiche Verfahren wieder anwenden, bis die Bereiche nur noch aus zwei Elementen bestehen. In diesem Fall führt man das einfache Sortierverfahren durch.*

Auch hier muß man beim vollständigen Ausprogrammieren nur noch die Veränderungen in den Bereichsgrenzen berücksichtigen. Der resultierende Algorithmus wird Quicksort *genannt, weil er im Mittel extrem schnell ist: bei Feldern der Länge* n *ist man nach nur* $\log_2 n$ *Aufteilungen am Ziel und jedes Aufteilen benötigt genau* n *Schritte. Insgesamt liegt damit die Anzahl der Schritte in der Größenordnung von* $n \log_2 n$*, was von keinem anderen bekannten Algorithmus unterboten wird. Wegen der Problematik, nicht immer einen guten Mittelwert bestimmen zu können, kann allerdings die Anzahl der Schritte im schlimmsten Fall auf* n^2 *anwachsen.*

Die Entwicklung des Quicksort-Algorithmus illustriert die Grundprinzipien der Divide&Conquer Strategie.

Entwurfsprinzip 4.5.18 (Divide&Conquer Strategie)

Teile das Problem in zwei "gleiche" Hälften auf und behandle jede Hälfte mit dem gleichen Verfahren, bis die Teile klein genug sind, um eine direkte Lösung zu erlauben.

Diese Strategie liefert oft sehr effiziente Algorithmen. Wie die Strategie 4.5.16 führt sie eine Reduktion auf einfachere Probleme durch. Der Unterschied zwischen den beiden Strategien liegt im wesentlichen nur in der Zielrichtung: die einfache Strategie fragt danach, welche *bereits bekannten* Probleme mit Erfolg verwandt werden können, während die Divide&Conquer-Strategie das Problem zunächst in kleinere Teile *aufteilt* und dann fragt, wie die Lösungen der kleineren Teile zur Lösung des Gesamtproblems beitragen.

Beide Strategien führen normalerweise zu sehr eleganten und effizienten *rekursiven* Algorithmen. Da rekursive Aufrufe im Normalfall jedoch unnötig viel Speicherplatz verschwenden, ist im Allgemeinfall einem Algorithmus der Vorzug zu geben, welcher Schleifen anstelle von Rekursion enthält. Es lohnt sich daher fast immer, rekursive Aufrufe in Schleifen umzuwandeln.

Entwurfsprinzip 4.5.19

Ersetze Rekursion durch Schleifen, wann immer dies einfach durchzuführen ist.

Ein weiterer Grund für die Verwendung von Schleifen ist, daß Rekursion nicht der natürlichen Denkweise der meisten Programmierer entspricht und daher oft schwer zu verstehen ist. Die *Wiederholung* einer Anweisung liegt dem menschlichen Denken oft näher als die rekursive mathematische Beschreibung. Deshalb sollte sich auch ein mathematisch trainierter Programmierer bei der Ausformulierung einer Lösung direkt auf eine iterative Denkweise einlassen. Dies macht die Entwicklung von Schleifen und Invarianten erheblich leichter und führt zu Programmen, die auch von anderen verstanden werden können.[54]

[53] Das Verfahren hierzu ist ein separat zu erzeugender Standardalgorithmus, der wie folgt vorgeht: Man suche simultan von links das erste Element, das größer als `x` ist, und von rechts das erste Element, das kleiner oder gleich `x` ist. Diese werden vertauscht und dann weitergesucht, bis sich die beiden Suchindizes treffen.

[54] Die angesprochenen Lösungen beider Beispiele dieses Abschnitts können ohne größeren Aufwand als iterative Programme ausformuliert werden. Details hierzu liefert [Gries, 1981, Kapitel 18].

Mit einer wichtigen strategischen Anmerkung wollen wir das Thema der systematischen Implementierung von Routinen abschließen. In allen Beispielen haben wir unsere Algorithmen zunächst auf einer mathematischen Ebene entwickelt und erst zum Schluß in konkrete Programme umgesetzt. Diese Vorgehensweise verhindert, daß man sich zu früh an die speziellen Eigenschaften einer Programmiersprache und die darin vorgegebenen Operationen auf einer Datenstruktur (wie `ARRAY`, `LINKED_LIST` etc.) bindet und dadurch den Blick für effizientere und elegantere Lösungen verliert. Meist ergibt sich die sinnvollste Datenstruktur erst *nachdem* das Problem bereits gelöst ist.

Entwurfsprinzip 4.5.20

Programmiere nicht innerhalb einer Programmiersprache sondern in die Sprache hinein.

Dies entspricht auch der grundlegenden Philosophie von Eiffel. Was zählt, ist der zu erfüllenden Vertrag und die darin versprochenen Dienstleistungen. *Wie* diese Dienstleistungen realisiert werden, ist erst von nachrangiger Bedeutung. Deshalb stelle man bei der Entwicklung des Algorithmus erst eine Wunschliste auf, welche "abstrakten" Dienstleistungen gebraucht werden, und entscheide sich erst im Nachhinein für die konkrete Realisierung – am besten dadurch, daß man die gewünschte Datenstruktur samt ihrer Dienstleistungen in einer separaten Klasse realisiert.

4.6 Ethik und Verantwortung

In der bisherigen Vorlesung haben wir uns – bis auf wenige Anmerkungen – im wesentlichen mit technischen und methodischen Fragen der Programmierung befaßt und uns besonders mit Qualitätskriterien wie Strukturierung, Wiederverwendbarkeit, Zuverlässigkeit und Effizienz auseinandergesetzt. Entwicklung guter Softwaresysteme bedeutet aber weit mehr als eine Berücksichtigung solcher "technischer" Kriterien, da Softwareprodukte die Umgebung, in der sie eingesetzt werden, zuweilen drastisch verändern kann.

Innerhalb eines Betriebes können sich manche Arbeitsabläufe verändern und Angestellte zu einer vollständigen Umstellung ihrer gewohnten Arbeitsweise zwingen, was besonders (aber nicht nur) für ältere Menschen eine unzumutbare Belastung mit sich bringt.

Rechner werden in sicherheitsrelevanten Bereichen wie in Flugzeugen (Autopilot), Autos (ABS Systeme), Kraftwerken (automatische Abschaltung) oder medizinischen Geräten eingesetzt, in denen sie weitgehend autonom und ohne Kontrolle durch den Menschen arbeiten und durch unerwünschtes Verhalten oder Fehlfunktionen Menschen verletzen oder gar töten können.

Rechner verarbeiten in immer größerer Menge personenbezogene Daten und können leicht dazu mißbraucht werden, über das unbedingt notwendige Maß in die Privatsphäre von Menschen einzudringen. Der zunehmende Einsatz von Rechnern in allen Bereichen des Alltagslebens hat dazu geführt, daß "Computerexperten" anfällig sind für neue Formen krimineller Handlungen, deren Tragweite sie sich meistens gar nicht bewußt sind. Es ist möglich, Menschen zu belästigen (Erzeugung unerwünschter Bilder auf dem Bildschirm oder ähnliche Störungen), ihr (geistiges) Eigentum zu stehlen (unerlaubtes Kopieren von Software), ihnen Schaden zuzufügen (Viren), oder auch größere Katastrophen auszulösen (Eindringen in militärische Rechnersysteme).

Man könnte die Liste der Auswirkungen des Computereinsatzes beliebig weiterführen. Schon diese wenigen Beispiele zeigen jedoch, daß Informatiker sich mit mehr beschäftigen müssen als mit den rein fachliches Aspekten ihres Berufes. Die Frage nach der *Verantwortung* des eigenen Handelns muß frühzeitig gestellt und so gut wie möglich beantwortet werden. Dabei kann man verschiedene Arten von Verantwortung unterscheiden, deren Be- oder Mißachtung unterschiedliche Konsequenzen haben wird.

Es gibt Dinge, für die Sie vom Gesetz her verantwortlich sind, sowohl als Privatperson als auch im Rahmen der Ausübung ihres Berufes. Gesetze regeln, wofür Sie *haftbar* gemacht werden können, sei es nun, weil Sie etwas getan haben, für dessen Folgen Sie Verantwortung tragen, oder weil Sie etwas unterlassen haben, was Sie eigentlich hätten tun müssen. So müssen Sie damit rechnen, bestraft zu werden, wenn Sie von einem Freund

lizenzpflichtige Software kopieren, für die Sie in einem Geschäft normalerweise über 100 Mark bezahlen müssten (*Raubkopie*). Das gleiche geschieht, wenn Sie nachts an einem Unfallort vorbeifahren ohne Hilfe zu leisten, wenn nicht schon jemand anderes da ist (*unterlassene Hilfeleistung*). Für von Ihnen produzierte und verkaufte Hard- und Software müssen Sie *Garantien* übernehmen und ggf. entstandene Schäden ersetzen. Liefern Sie ein größeres Softwarepaket nicht innerhallb der vertraglich vereinbarten Zeit, so zahlen Sie Konventionalstrafen, usw.

Innerhalb Ihres Studiums und mehr noch an ihrem späteren Arbeitsplatz haben Sie Verantwortungsbereiche, für die Sie zur Rechenschaft gezogen werden können. Als Arbeitnehmer sind Sie verpflichtet, gewisse Aufgaben zu übernehmen und diese gewissenhaft auszuführen. Mangelnde Loyalität gegenüber Arbeitgeber und Kollegen zeigt, daß Sie nicht in die Firma passen und besser entlassen werden sollten. Mangelnde Ehrlichkeit im Studium (Abschreiben in Prüfungen) – wenn auch selten kontrolliert – zeigt, daß Ihnen die nötige sittliche Reife für das Studium fehlt, und kann bei (Vor-)Diplomprüfungen die Exmatrikulation zur Folge haben.

Neben diesen Verantwortungsbereichen, bei denen ein Verstoß normalerweise sanktioniert wird, gibt es aber auch den Bereich der allgemeinen "moralischen" Verantwortung, die Sie gegenüber anderen Menschen, der Gesellschaft, der Umwelt, und – sofern Sie dies akzeptieren – gegenüber Gott besitzen. Es würde den Rahmen einer Informatik-Grundvorlesung sprengen, dieses Thema in aller Ausführlichkeit zu behandeln. Nichtsdestotrotz möchte Ich Sie durch einige Anmerkungen zum Nachdenken und ggf. zu einer Änderung Ihrer bisherigen Einstellung anregen.[55]

In Ihrem Arbeitsumfeld aber auch in Ihrem privaten Umgang mit Computern werden Sie ständig der Versuchung begegnen, sich Vorteile zu beschaffen, zu denen Sie eigentlich nicht berechtigt sind. Sie können sich lizenzpflichtige oder von Kollegen erarbeitete Software kopieren, weil dies billiger ist als sie zu kaufen oder weil Sie dadurch das geistige Eigentum anderer als eigene Arbeit verkaufen können[56] Sie können in den privaten Dateien von Kommilitonen herumsuchen, weil Sie neugierig sind. Sie können mit einigem Aufwand in Rechnersysteme eindringen, ohne daß Sie ein eingetragener Benutzer sind. Es gibt viele Dinge, die Sie aufgrund Ihrer Fähigkeit tun *könnten* und bei denen Sie nicht erkennen, warum Sie sie nicht tun *sollten*. Versuchen Sie, diesem Anreiz von Anfang an zu widerstehen. Es ist schwer, sich eine Unsitte wieder abzugewöhnen, sobald sie einmal zur Gewohnheit georden ist. Zudem verschwindet auf die Dauer das Unrechtsbewußtsein und Sie werden irgendwann vor der Versuchung stehen, Handlungen zu begehen, die tatsächlich kriminell sind.

Sie sollten sich darüber im klaren sein, daß Ihre Arbeit weitreichende Auswirkungen auf die Allgemeinheit haben wird. Setzen Sie sich zum Ziel, die Sicherheit, Gesundheit und das Wohlergehen der Allgemeinheit zu fördern, und versuchen Sie, die Konsequenzen Ihrer Tätigkeit im Hinblick auf diese Aspekte abzuschätzen.

Dies verpflichtet Sie natürlich dazu, sich so weit wie möglich kompetent zu machen, sowohl was Ihre Fachkompentenz und die Kenntnis der Gesetzeslage betrifft als auch die ein Verständnis des Bereichs, in dem ihre Arbeit Auswirkungen zeigen wird. Letztere verlangt Ihre ständige *Bereitschaft*, die Anliegen und Interessen der verschiedenen Betroffenen zu verstehen und angemessen zu berücksichtigen. Die Weiterentwicklung der eigenen Gesprächs- und Urteilbereitschaft ist ein Prozeß, der niemals abgeschlossen sein darf.

Auch wenn Sie sich darum bemühen, Ihre Fähigkeiten ständig auszubauen, ist eine realistische Kenntnis der eigenen Grenzen sehr wichtig. Wenn Sie Aufgaben übernehmen, für die Sie nicht im geringsten kompetent sind, können Sie großen Schaden anrichten. Zeigen Sie genügend Größe, indem Sie auch einmal zugeben, daß Sie etwas *überhaupt* nicht können. Allerdings sollten Sie durchaus auch Aufgaben annehmen, die Ihre derzeitigen Fähigkeiten übersteigen, wenn es möglich ist, daß Sie sich die nötigen Kenntnisse während der Bearbeitungszeit aneignen.

---

[55] Die Anmerkungen entstammen weitgehend dem Ethikcodex der amerikanischen Association for Computing Machinery (ACM) und den vorgeschlagenen ethischen Leitlinien der deutschen Gesellschaft für Informatik (GI) und wurden durch meine persönlichen Ansichten ergänzt. Es ist mir bewußt, daß sie subjektiv und unvollständig sind, aber ich hoffe, daß Sie dennoch zu einem verantwortungsvolleren Handeln beitragen können.

[56] Dies spricht nicht gegen Kopien von sogenannter "öffentlich zugänglicher" (public domain) Software oder Kopien mit Zustimmung des anderen.

Seien Sie sich bewußt, daß gerade im Softwarebereich eine Perfektion unmöglich ist. Auch bei sorgfältigstem Vorgehen wird es Ihnen nicht gelingen, ein großes Softwarepaket absolut fehlerfrei oder gar optimal zu gestalten. Selbst wenn dies der Fall wäre, können Hardwarefehler dies wieder zerstören. Es ist unverantwortlich, einen Auftraggeber durch überzogene Versprechungen, die Sie später nicht einhalten können, in falscher Sicherheit zu wiegen – besonders in sicherheitsrelevanten Anwendungen. Hohe Qualität aber dürfen Sie durchaus versprechen, wenn Sie bereit sind, sie zu liefern.

Wenn Ihnen Fehler unterlaufen, oder Sie zugesagte Abmachungen nicht einhalten können, sollten Sie ehrlich genug sein, dies auch zuzugeben, um größere Folgeschäden, die sich vielleicht erst Jahre später einstellen, abzuwenden.[57] Dies erfordert Mut und zuweilen die Bereitschaft, Konsequenzen auf sich zu nehmen, denen Sie sich vielleicht durch Stillschweigen (Feigheit?) hätten entziehen können.

Sie müssen damit rechnen, ab und zu in Situationen zu geraten, in denen Ihre Pflichten gegenüber Arbeitgeber oder Kunden im Konflikt mit Ihrer Verantwortung gegenüber den Betroffenen oder der Gesellschaft im Allgemeinen stehen. Dieser Verantwortung gerecht zu werden, erfordert die Bereitschaft, einen persönlichen Schaden in Kauf zu nehmen um einen *möglichen* großen Schaden von der Gesellschaft abzuwenden, die sich wahrscheinlich nicht einmal dankbar erweist.

Eine derartige *Zivilcourage* aufzubringen ist wirklich nicht leicht, aber ohne die Bereitschaft von Leuten, in ihrem Einflußbereich für höhere Werte einzutreten, auchwenn dies mit persönlichen Verlusten verbunden ist, würde unsere Gesellschaftsordnung in kürzester Zeit zugrunde gehen.

[57] Natürlich darf man nicht bei jeder kleinen Abweichung in Panik geraten und Leute durch "übertriebene Ehrlichkeit" verunsichern. Was wirklich ein Fehler oder ein Nichteinhalten einer Abmachung ist, ist eine Ermessensfrage, die man wieder nur mit der entsprechenden Kompetenz verantwortungsvoll beantworten kann.

Literaturverzeichnis

[Bauer & Wirsing, 1991] Friedrich L. Bauer and Martin Wirsing. *Elementare Aussagenlogik*. Springer Verlag, 1991.

[Baumann, 1990] R. Baumann. *Didaktik der Informatik*. Klett Schulbuchverlag, 1990.

[Bochenski & Menne, 1983] I. M. Bochenski and Albert Menne. *Grundriß der formalen Logik*. Number 59 in Uni Taschenbücher. F. Schöningh Verlag, 5 edition, 1983.

[Davis, 1989] Ruth E. Davis. *Truth, Deduction, and Computation*. Computer Science Press, New York, 1989.

[Dijkstra, 1976] Edsger W. Dijkstra. *A discipline of Programming*. Prentice Hall, 1976.

[Gries, 1981] David Gries. *The science of programming*. Springer Verlag, 1981.

[Hermes, 1972] H. Hermes. *Einführung in die mathematische Logik*. , Stuttgart, 3 edition, 1972.

[Hoffmann, 1990] H.-J. Hoffmann. Grundzüge der Informatik I + II. Skriptum, TH Darmstadt, 1990.

[Kammerer, 1993] Kammerer. Grundzüge der Informatik II. Skriptum, TH Darmstadt, 1993.

[Loeckx & Sieber, 1987] Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification*. B.G. Teubner, Stuttgart, 1987.

[Meyer, 1988] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[Meyer, 1992] Bertrand Meyer. *Eiffel – the Language*. Prentice Hall, 1992.

[Polya, 1945] G. Polya. *How to solve it*. Princeton University Press, Princeton, New Jersey, 1945.

[Stoy, 1977] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, 1977.

[Woodcock & Loomes, 1988] J. C. P. Woodcock and M. Loomes. *Software engineering mathematics*. Pitman, London, 1988.