Diplomarbeit

Softwarebibliotheken: Entwurf und Dokumentation am Beispiel von Tycoon

27. November 1995

vorgelegt von:
Tanja Jung
Saseler Chaussee 55
22391 Hamburg

Betreuer:
Prof. Dr. Joachim W. Schmidt
Dr. Ingrid Wetzel

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Datenbanken und Informationssysteme
Vogt-Kölln-Straße 30
D-22527 Hamburg

Inhaltsverzeichnis

1 Einführung **1**
- 1.1 Bibliotheken . . . 2
- 1.2 Wiederverwendung . . . 5
  - 1.2.1 Probleme der Wiederverwendung . . . 8
  - 1.2.2 Forderungen an Bibliotheken . . . 11

2 Dokumentation **18**
- 2.1 Strukturierung, Layout und Namenskonventionen . . . 20
- 2.2 Kommentare . . . 21
  - 2.2.1 Kategorisierung von Kommentaren . . . 21
  - 2.2.2 Effiziente Kommentierung . . . 22
  - 2.2.3 Techniken der Kommentierung . . . 25
- 2.3 Das Layout von Programmen . . . 35
- 2.4 Benennung und Namenswahl . . . 37
  - 2.4.1 Namen und Deklaration von Routinen . . . 37
  - 2.4.2 Routinenparameter . . . 38
  - 2.4.3 Variablennamen . . . 39
  - 2.4.4 Berechnete Werte . . . 40
  - 2.4.5 Gegensatzpaare . . . 41
  - 2.4.6 Regeln für spezifische Datentypen . . . 42
  - 2.4.7 Namenskonventionen . . . 44
  - 2.4.8 Lesbare Abkürzungen . . . 48
  - 2.4.9 Problematische Namen . . . 50
- 2.5 Anforderungen an Tycoon . . . 51

3 Konzepte zum Aufbau von Bibliotheken **52**
- 3.1 Grundlagen . . . 52
  - 3.1.1 Klassifikation von Programmiersprachen . . . 52
  - 3.1.2 Objektorientierte Programmiersprachen . . . 54
- 3.2 Vorstellung der Sprachen . . . 57
  - 3.2.1 Modula-3 . . . 57
  - 3.2.2 Eiffel . . . 58
  - 3.2.3 Smalltalk . . . 60
  - 3.2.4 C++ . . . 61
  - 3.2.5 Tycoon . . . 63
- 3.3 Spezielle Konzepte . . . 64

3.3.1 Abstraktionen für Datentypen . . . 64
3.3.2 Iteratoren . . . 65
3.3.3 Modellierung von Datum und Zeit . . . 70
3.3.4 Das Konzept der Zusicherungen . . . 74
3.4 Zusammenfassung der Anforderungen an Tycoon . . . 78

4 Bibliotheken im Vergleich 79
4.1 Modula-3 . . . 83
4.2 Eiffel . . . 85
4.2.1 Strukturierung der Bibliotheken . . . 85
4.2.2 Konvention der Methoden . . . 89
4.3 Smalltalk . . . 92
4.4 C++ . . . 93
4.5 Tycoon . . . 103
4.5.1 Programmierung im Großen . . . 103
4.5.2 Aufbau der Bibliotheken . . . 107
4.5.3 Kapselungskonzepte in TL-Bibliotheken . . . 111
4.5.4 Bibliotheken und Kapselungskonzepte . . . 112
4.6 Zusammenfassung der Anforderungen an Tycoon . . . 113

5 Überarbeiteter Entwurf der Tycoon-Bibliotheken 115
5.1 Strukturierung der Bibliotheken . . . 115
5.1.1 Spezifizierung der Bibliotheken *baseLib*, *machineLib* und *bulkLib* . . . 117
5.1.2 Schnittstellen für die Datums- und Zeitverwaltung . . . 119
5.2 Konventionen für Methoden . . . 122
5.2.1 Fundamentale Operationsnamen . . . 122
5.2.2 Klassifikation von Operationen für Massendatentypen . . . 126
5.2.3 Konvertierung von Datentypen . . . 129
5.2.4 Das div-mod-Problem . . . 132
5.2.5 Hash-Funktionen . . . 135
5.3 Zusicherungen in Tycoon . . . 136
5.4 Konzepte zur Abstraktionsbildung . . . 139
5.4.1 Iteratoren . . . 140
5.4.2 Domain . . . 141
5.5 Spezifikations- und Dokumentationsrichtlinien . . . 143
5.5.1 Dokumentenköpfe . . . 143
5.5.2 Richtlinien für den Code . . . 151

6 Zusammenfassung und Ausblick 156

Kapitel 1

Einführung

Das Tycoon[1]-System [Mathiske et al. 93] bietet eine persistente Programmierumgebung zur Entwicklung von datenintensiven Anwendungen, die im Rahmen einer Dissertation [Matthes 93] erstellt wurde. Sie ermöglicht eine flexible Definition und Integration generischer Dienste in offenen Systemumgebungen. Die polymorphe Programmiersprache TL[2] ist zur Definition von Schnittstellen, Bibliotheksmodulen und Anwendungen in das Tycoon System integriert.

Bibliotheken bilden zusätzlich zu den Datentyp-Abstraktionen ein weiteres Abstraktionsniveau über der Programmiersprache. Für Pomberger und Blaschek [Pomberger, Blaschek 93] sind sie wichtiger als die Programmiersprache selbst, weil ihre Konzepte einen größeren Einfluß auf das Denkmodell des Programmierers haben.

Eine Motivation für die Bibliotheksbildung ist die Wiederverwendung. Bibliotheken stellen Teile von geschriebenem Code zur Verfügung, die der Programmierer in sein Anwendungsprogramm einbinden kann.

Diverse Bibliotheken stehen im Tycoon System zur Verfügung. Ziel dieser Arbeit ist eine überarbeitete und erweiterte Spezifikation der Tycoon-Bibliotheken. Ein erster Schritt ist die Klassifizierung der bestehenden Schnittstellen und Module der fundamentalen Bibliotheken. Um eine Beschränkung des Themas für diese Arbeit zu erreichen, wird hier im wesentlichen nur die plattformunabhängige und grundlegende Standard-Tycoon-Bibliothek *stdenv* betrachtet. Bei einigen darüber hinausgehenden Fragen werden auch die darauf aufbauenden Bibliotheken *bulkenv* und *machineenv* mit einbezogen. Die Bibliotheken werden nach Vergleich mit Standardbibliotheken von C++, Eiffel, Modula-3 und Smalltalk neu zusammengefaßt und erweitert. Dabei werden Verbesserungsvorschläge berücksichtigt, insbesondere aus objektorientierten Bibliotheken, weil sie eine wohldefinierte Klassenhierarchie unterstützen. Die zur Zeit in Tycoon zur Verfügung stehenden Bibliotheksschnittstellen werden hinsichtlich ihrer Namen und ihrer Semantik vereinheitlicht und ergänzt. Es wird ein *Style Guide* (Anleitung mit Stilrichtlinien) erstellt, der Regeln zur Namensgebung für Datenstrukturen und darauf arbeitende Operationen enthält. Darüber hinausgehende Regeln betreffen die Namensgebung von Programmierkonstrukten und Variablen.

Als weiteres Mittel zur Unterstützung der Handhabbarkeit der Tycoon-Bibliotheken werden Richtlinien zur Dokumentation von wiederverwendbaren Modulen erarbeitet. Soweit es im Rahmen eines ganzheitlichen Konzepts sinnvoll erscheint, werden objektorientierte Ansätze in Tycoon nachgebildet. Dazu wird untersucht, ob und wie sich verschiedene Kapselungskon-

[1] **T**yped **c**ommunicating **o**bjects in **o**pen e**n**vironments
[2] **T**ycoon **L**anguage

zepte, die von der Programmiersprache TLdes Tycoon Systems unterstützt werden, in einer Anwendung gemeinsam verwenden lassen. Um die Korrektheit von Anwendungsprogrammen, die mit Hilfe der vorhandenen Bibliotheken erstellt werden, sicherzustellen, wird das Konzept der *Zusicherungen* vorgestellt und für TLneu eingeführt. Dadurch wird es ermöglicht, daß die Bausteine der Bibliothek einzeln auf Korrektheit geprüft werden können, um daraufhin dem Anwendungsprogrammierer Hilfestellung bei der Korrektheitsüberprüfung seiner Programme zu geben.

1.1 Bibliotheken

Es gibt verschiedene Vorstellungen darüber, was man unter dem Begriff *Bibliothek* versteht. Im allgemeinen Sprachgebrauch wird unter einer *Bibliothek* eine Menge von Büchern verstanden, die einem Kreis von Lesern zur Verfügung gestellt werden. Dieser Begriff ist für die Informatik im Bereich der Programmiersprachen auf Sammlungen von wiederverwendbaren Dateien, die Module oder Klassen enthalten, übertragen worden.

In der Tabelle 1.1 werden die Entsprechungen in der Semantik aus beiden Bereichen gegenübergestellt. Dies zeigt anschaulich, daß ähnliche Zugriffsarten und Verwendungszwecke zu der Bezeichnung *Bibliothek* auch in der Softwarebranche geführt haben. Lediglich die Form der Speicherung ist verschieden und wirkt sich auf die Art der Verwendung aus.

Ein Unterschied besteht in der Art der Verwaltung. Zumindest in öffentlich zugänglichen, konventionellen Bibliotheken existieren Karteien und Kataloge, die nach verschiedenen Suchkriterien einen schnellen Zugriff auf die gewünschte Menge von Büchern ermöglichen. Der Leser kann mit Hilfe der eindeutigen Bezeichnung der Bücher in den Karteien auf diese Bücher zugreifen und durch die Zusammenfassungen auf den inneren Klappenseiten weitere, differenzierte Information gewinnen, um letztendlich nur die Bücher auszuleihen, die er wirklich lesen möchte. Eine entsprechende Umgebung für einen effizienten Zugriff auf Software-Bibliotheken fehlt in den meisten Programmiersprachen. Ein Benutzer muß sich selbst einen Überblick über die angebotenen Bibliotheksbausteine schaffen, indem er alle Dateien ansieht, nach bestimmten Begriffen darin sucht und sich Listen mit den in Frage kommenden Dateien erstellt. Eine Hilfe, die aber nicht alle Bibliotheken bieten (siehe [Möller, Müller 93]), sind Dokumentationen mit kurzen inhaltlichen Zusammenfassungen. Es ist daher wünschenswert, daß auf dem Gebiet der Verwaltungswerkzeuge für Bibliotheken noch Entwicklungsarbeit geleistet wird. Eine Grundlage zur Anwendung von Software-Werkzeugen ist die Vereinheitlichung aller Dateien innerhalb einer Bibliothek. Zur Erreichung dieses Ziels leistet diese Arbeit einen grundlegenden Beitrag, indem für die Bibliotheken des Tycoon Systems vereinheitlichende Regeln erstellt werden.

In der Abbildung 1.1 wird die Umgebung von Tycoon, wie sie sich dem Anwendungsprogrammierer darlegt, schematisch wiedergegeben.

Bibliotheken fördern die Benutzung von vorhandenem Wissen in der Erstellung von neuem Wissen [Wegner 84]. In der Software-Entwicklung stellen sie "Datenbanken" mit wiederverwendbaren Software-Komponenten dar, die das Erstellen neuer Applikationen durch die Benutzung vorhandener Applikationsteile unterstützen. Beim Entwurf von Bibliotheken müssen daher Sammlungen von Software-Komponenten so organisiert werden, daß sie einfach erstellt und benutzt werden können. Darüber hinaus sollten sie auch Werkzeuge enthalten, die dieses Wissen wie Kataloge organisieren können. Im Gegensatz zu konventionellen Bibliotheken sind ihre Verwender allerdings keine Menschen, sondern Computer. Daher müssen die

	konventionell	informationstechnisch
Strukturen		
Sammlung	Eine Bibliothek ist eine Sammlung von Wissen, Information und Erfahrung in Büchern, die in einem Raum stehen.	Eine Bibliothek ist eine Sammlung von Datentypen und Objekten mit ihren Methoden auf einem Rechner.
Speicherung	Die Bücher stehen in einem genau bezeichneten Raum.	Datentypen und Objekte mit ihren Methoden werden auf einer Platte eines bestimmten Rechners gespeichert.
Objekt	Die Ordnung erfolgt in Büchern, Zeitschriften und anderen Medien.	Die Ordnung erfolgt in Modulen oder Klassen, die jeweils eine Datei bilden.
Kategorisierung	Eine Bibliothek enthält verschiedene Sachgebiete, nach denen die Bücher kategorisiert sind.	Eine Bibliothek enthält verschiedene Anwendungsgebiete, welche die Einteilung der Dateien in verschiedene Verzeichnisse vorgeben.
Dienste		
Verwendung	Der Benutzer kann einzelne Bücher durch Ausleihen ansehen und die darin enthaltene Information für sich verwerten.	Der Benutzer kann einzelne Dateien ansehen und für seine eigene Anwendung benutzen.
Auswahl	Jeder Benutzer leiht Bücher zu mehreren Themen aus.	Jeder Programmierer wählt die ihm passenden Dateien zu seinem aktuellen Anwendungsbereich.

Tabelle 1.1: Entstehung des Bibliotheksbegriffs

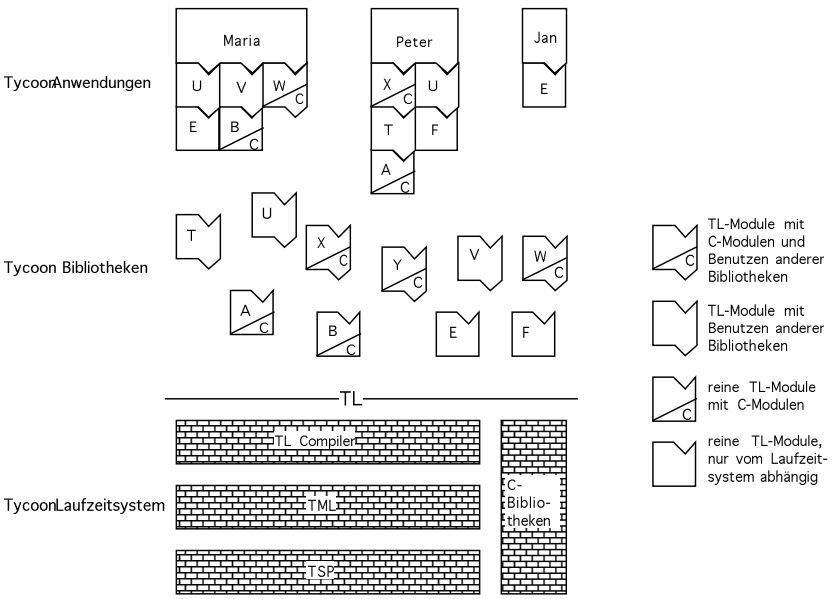


Abbildung 1.1: Das Tycoon System aus der Sicht des Anwenders

Werkzeuge auch automatisch Komponenten in Programme überführen. *Intelligente* Bibliotheken nutzen Wissen über das Anwendungsgebiet, um Programme aus den Komponenten der Software-Bibliothek zusammenzusetzen. Nach Wegner [Wegner 84] müssen beim Entwurf einer Programmbibliothek folgende Punkte berücksichtigt werden:

- Welche Arten von Komponenten enthält die Bibliothek?
- Welche Auflösung (Feinheit) und welchen Anwendungsbereich hat die Bibliothek?
- Welche Kunden (Programme, Menschen) werden die Bibliothek benutzen?
- Wie werden die Komponenten geladen, gebunden und aufgerufen?
- Wie werden die Komponenten erstellt, eingefügt, untersucht und wiedergefunden?
- Welche Beziehungen zwischen den Komponenten können bestehen und vermerkt werden?
- Welche Art von Wissen ist als Hilfe für die Programmierer zur Erstellung von Programmen aus Software-Bibliotheken nötig?

Der Designer einer solchen Bibliothek steht immer wieder vor den Problemen, ob zuviel oder zuwenig Funktionalität in einer bestimmten Klasse unterstützt wird oder ob Klassen in kleinere Einheiten gespalten werden sollen bzw. Klassen zu einer gesamten Klasse integriert werden müssen. Für diese Entwurfsprobleme gibt es bislang nur wenige Richtlinien als Entscheidungshilfe [Lea 88]. Diese Arbeit soll einem Designer helfen, eine Kategorisierung der beim Bibliotheksentwurf auftretenden Probleme vornehmen und anhand seiner individuell erstellten Anforderungen Entscheidungen im Entwurf treffen zu können.

Charakterisiert werden Bibliotheken durch die Art der enthaltenen Komponenten und durch die Art und Weise, wie Menschen und Computer diese Komponenten nutzen. Manager, Programmierer und Software-Analytiker haben unterschiedliche Anforderungen und Erwartungen. Computer haben ebenfalls verschiedene Schnittstellen zu den Software-Komponenten, z. B. Compiler-Schnittstellen zur Programmprüfung und -übersetzung oder Werkzeug-Schnittstellen zur Fehlererkennung und Versionskontrolle oder Laufzeitschnittstellen zur Ausführung von Modulen.

1.2 Wiederverwendung

Die allgemeinen Software-Qualitätskriterien *Änderbarkeit*, *Wartbarkeit* und *Portierbarkeit* werden durch Wiederverwendung unterstützt. Dabei ist die Organisation von Code in Bibliotheken eine wichtige Voraussetzung für dessen wiederholte, intensive Nutzung. Daher wird hier eine Zusammenfassung der wesentlichen Aspekte der Wiederverwendung gegeben. Anschließend werden Probleme bei der Wiederverwendung von Bibliotheksmodulen diskutiert, die durch Aufstellung eines Anforderungskatalogs an Bibliotheken weitgehend vermieden werden sollen.

In den 80er Jahren ist das Interesse an Wiederverwendung von Software ständig gestiegen. Die Bibliographie in [Börstler 89] gibt einen guten Überblick über die Veröffentlichungen in dieser Zeit. Wiederverwendung verspricht Verbesserungen in der Software-Qualität und der Produktivität bei der Software-Erstellung. Die Entwicklungskosten werden gesenkt, weil die Länge des zu dokumentierenden Programmtexts und der Umfang des Testens mit wiederverwerteter Software reduziert werden. Wenn große Teile des Codes einer Anwendung aus bereits entwickelter, getesteter Software bestehen, hat der Programmierer mehr Zeit, sich den übrigen Teilen des Produkts zu widmen. Ein System mit wiederbenutztem Code ist einfacher zu warten und zu modifizieren, weil die Programmierer mit den wiederverwendeten Blöcken vertrauter sind und den gesamten Systementwurf schneller verstehen können. Verbesserungen im wiederverwendeten Code wirken sich für alle benutzenden Applikationen gleich positiv aus.

Die Qualität der Software kann mit Hilfe der Wiederverwendung verbessert werden [Tracz 88], wenn die wiederbenutzte Software

- für Wiederverwendung entworfen,
- nach einem Standard dokumentiert,
- getestet und für Wiederverwendung ausgewiesen, sowie
- funktional verstanden und angemessen benutzt wird.

Wiederverwendung und *Wiederverwendbarkeit* von Software wird durch eine Anzahl von Methoden, Techniken und Prozessen unterstützt. Dazu gehört die Portabilität, gemeinsame Nutzung von Code in aufeinanderfolgenden Versionen, gemeinsame Subsysteme und Routinen verschiedener Applikationen, wiederholte Anwendung eines Algorithmus' und Blockbildung. Wiederverwendung wird in *interne* und *externe* Wiederverwendung unterschieden [Pomberger, Blaschek 93]. Die interne Wiederverwendung bedeutet, daß dieselbe Abstraktion innerhalb eines Projekts mehrmals benutzt wird. Unter externer Wiederverwendung versteht man das Benutzen von Software aus anderen, früheren Projekten innerhalb oder außerhalb eines

Entwicklungsteams. Die externe Wiederverwendung verspricht, desto mehr Kosten einzusparen, je häufiger die Komponenten verwendet werden, bringt aber auch eine kostenintensivere Entwicklung der wiederzuverwendenden Software mit sich.

Die Wiederverwendbarkeit besitzt drei wichtige Aspekte [Lenz et al. 87]:

Die Abstraktionsebene, auf der Programme wiederverwendet werden, ist ein Kriterium zur Klassifizierung der Wiederverwendungsmethoden. So gibt es die Ebene des *Entwurfs*, der *Spezifikation* und des *Codes*, auf der Komponenten wiederverwendet werden können. Will man den Entwurf noch einmal in einer anderen Umgebung verwenden, so kann dies durch die Einbettung in einen anderen Quellcode, die Übertragung auf eine andere Hardware-Plattform oder unter abweichenden Performanz-Bedingungen geschehen. Auf der Ebene der Spezifikationen ist Wiederverwendung auf die gleiche Weise möglich. Da eine Spezifikation die Voraussetzung für den Entwurf und den Programmcode ist, ist ein Programmierer in der Lage, aus einer Spezifikation mehrere, verschiedene Entwürfe und Programme abzuleiten.

Die Wiederverwendung von Code wird von Programmierern im Vergleich zur Wiederverwendung von der Spezifikation oder vom Entwurf am ehesten akzeptiert, weil es offensichtlich Zeit spart, indem viele Zeilen Code nicht neu geschrieben werden müssen. Eine Gefahr besteht allerdings darin, daß die Wiederverwendung von Code nur für die Kodierungsphase geplant ist. Alle vorhergehenden Phasen der Programmentwicklung können diesen Aspekt nicht mit einbeziehen, obwohl die Entscheidung zur Code-Wiederverwendung schon in der Entwurfsphase Berücksichtigung finden muß, um den Code so generell zu entwerfen, daß er für andere Anwendungen brauchbar ist. Code, der ohne Änderungen wiederverwendet werden kann, ist sehr selten zu finden. Die Programmanweisungen sind meist von ihrer Umgebung abhängig, z. B. wenn globale Variable als Kontrollblöcke eingesetzt werden, und die Details der Repräsentation werden im Code reflektiert. Daher ist es schwierig, in einem vorhandenen System die wiederverwendbaren Teile zu isolieren, wenn diese nicht von vornherein dafür vorgesehen sind. Aus diesem Grund muß die Wiederverwendung von Code auf der Spezifikations- oder Entwurfsebene vervollständigt werden. Die Möglichkeiten für die Wiederverwendung sind auf diesen Ebenen vielfältiger, weil dort keine detaillierte Entscheidung über die Repräsentation getroffen werden muß. Dadurch ist es möglich, die Entscheidung über Wiederverwendung sehr früh im Entwicklungszyklus zu treffen und in allen Phasen des Entwurfs zu berücksichtigen.

Die Methoden der Kundenorientierung sind ein weiteres Kriterium für die Klassifizierung von Ansätzen zur Wiederverwendung. Sie beschreiben, wie ein wiederverwendbarer Teil auf den Kunden zugeschnitten ist, um die neuen Anforderungen zu erfüllen. Es gibt folgende Methoden der Kundenanpassung:

- Das Programm wird überhaupt nicht an den Kunden angepaßt.
- Das Programm wird nach den speziellen Anforderungen angepaßt. Dies bedeutet, daß manuelle Änderungen der internen Teile des wiederverwendbaren Programmstücks nötig sind. Es ist somit erforderlich, daß der Programmierer detaillierte Kenntnisse dieser internen Struktur besitzt.
- Denkbar ist auch die Modifikation durch Vervollständigen von Vorlagen oder Skeletten. Diese Methode ist einfacher und sicherer als die benutzerorientierte Anpas-

sung, aber es können sich Fehler durch mangelhafte Modifikationen in die wiederverwendbaren Teile einschleichen.

- Eine generische Parametrisierung der wiederverwendbaren Programmteile garantiert die Korrektheit der modifizierten Teile, d.h. ihre Übereinstimmung mit der Spezifikation. Dies führt aber zu einem erhöhten Aufwand beim Entwurf, da von vornherein diese Modifikationen geplant werden müssen. Dieser Nachteil der höheren Kosten kann durch die mehrmalige Verwendung wieder ausgeglichen werden.

Wegen der Stellung des Tycoon Systems als Forschungsprototyp werden Methoden der Kundenorientierung in der vorliegenden Arbeit nicht weiter berücksichtigt.

Die Bedingungen der Wiederverwendbarkeit sind allgemeine Konzepte und Abstraktionen, die sichtbar werden, wenn ein Anwendungsbereich einen bestimmten Reifegrad erreicht. Diese Konzepte bilden die Grundbausteine der Wiederverwendung. Teile eines speziellen Anwendungsbereichs, die kein Konzept repräsentieren, sind nicht wiederverwendbar. Wenn die allgemeinen Konzepte nicht bekannt sind, dann können sie durch eine Analyse des Anwendungsbereichs identifiziert werden. Die Ermittlung solcher Abstraktionen ist die Voraussetzung für die Wiederverwendbarkeit.

In [Meyer 88] wird hervorgehoben, daß das objektorientierte Programmieren die Wiederverwendung von Software unterstützt (vgl. Kapitel 3). Die durch Datenkapselung kompakte Darstellung der Klassen mit ihren Methoden gibt dem Programmierer eine Übersicht über die angebotenen Dienste und unterstützt dadurch die Auswahl von Modulen zur Wiederverwendung. Zusätzlich hilft eine übersichtliche Dokumentation, die einem Leser mit wenigen Blicken erlaubt, den wesentlichen Inhalt des vorliegenden Quelltextes zu erfassen, eine effiziente Wiederverwendung durchzuführen. Dies gilt auch für Programmiersprachen, die nicht objektorientiert sind.

Wiederverwendung ist auch ein Aspekt der Vererbung in objektorientierten Sprachen. Die Eigenschaften müssen nicht für jeden Datentyp definiert werden, sondern werden von der Superklasse geerbt. Diese geerbten Eigenschaften können in der Subklasse neu deklariert werden, müssen es aber nicht. Wenn sie unverändert übernommen werden, spart der Programmierer nicht nur Zeilen im Code, sondern auch Zeit zur Entwicklung, Dokumentierung und zum Testen dieses Codes. Im Fall der Vererbung wird der entsprechende Code aus der Superklasse wiederverwendet.

In den meisten Programmiersprachen und in intensiver Form in objektorientierten Sprachen werden *kompositionsbasierte* Systeme zur Wiederverwendung angeboten. Aus vorhandenen Komponenten wird eine neue Anwendung zusammengestellt. Dies ist die häufigste Art der Wiederverwendung, die meist durch Bibliotheken realisiert wird. In objektorientierten Sprachen werden Klassen, in modularen Sprachen Module in Bibliotheken zusammengefaßt und dem Anwender zur weiteren Verwendung zur Verfügung gestellt.

In Ergänzung dazu gibt es auch *generierungsbasierte* Systeme, die eine neue Anwendung durch Generatoren erzeugen. Sie ermöglichen einen weiteren Blickwinkel auf die Wiederverwendung. An Stelle einer *Black Box*, die kompatibel zu verschiedenen Anwendungen ist und in das programmierte Produkt eingefügt wird, liegt in dieser Sicht die Antwort auf wiederverwendbare Software in der Analyse des Programmierprozesses. Horowitz macht daher in [Horowitz, Munson 84] den Vorschlag folgender Wiederverwendungsmethoden für den Programmierprozeß:

- *Very High Level Languages* (VHLLs) erlauben die Spezifikation des Anwendungsbereichs in der Syntax dieser Sprache. *Problem Oriented Languages* (POLs) sind eine Form von VHLLs für einen bestimmten Anwendungsbereich. Durch erneutes Benutzen des Compilers kann eine Spezifikation wiederverwendet werden.

- *Applikationsgeneratoren* sind Softwarewerkzeuge, die aus einer parametrisierten Spezifikation Programme erzeugen. Der Applikationsgenerator kann für jeden neuen Anwendungsbereich wiederverwendet werden.

- *Transformationssysteme* erfordern Spezifikationen auf hoher Ebene, die beschreiben, welche Aktionen von dem zu erstellenden Softwaresystem ausgeführt werden sollen (vgl. [Biggerstaff, Perlis 89]). Diese Spezifikationen werden durch eine Reihe von mustererkennenden Erweiterungen in ein Programm überführt.

Die Schlüsseltechnologie dieser drei Beispiele liegt in der automatischen Anwendung von wiederverwendbaren Komponenten. Jedes Werkzeug erkennt einige typische Muster im Anwendungsbereich, die durch Ersetzung einiger Code-Abschnitte implementiert werden können. Durch die Verwendung von transformierenden Systemen wird es für den Programmierer überflüssig, Code-Abschnitte direkt wiederzuverwenden. Diese Tätigkeit übernimmt der Transformator. Damit sind alle Richtlinien zur Vereinfachung der Wiederverwendung von Code, die in dieser Arbeit aufgestellt werden, nicht mehr anwendbar. Die Wiederverwendung durch einen menschlichen Benutzer findet nur noch auf der Ebene der Spezifikation und des Entwurfs statt.

1.2.1 Probleme der Wiederverwendung

Obwohl es für C++ viele kommerzielle Bibliotheken gibt, wird Wiederverwendung in Produktionsumgebungen selten durchgeführt [Schmid 94a]. Eine Ausnahme bilden die Bibliotheken für Benutzerschnittstellen, deren Struktur leicht durchschaubar ist und in vielen Veröffentlichungen dargestellt wird. Im folgenden sollen einige mögliche Gründe, welche die Wiederverwendung verhindern, aufgeführt werden [Tracz 88]. Hieraus werden Anforderungen an die Unterstützung der Wiederverwendung in den TL-Bibliotheken abgeleitet. Hierbei sind die Gründe aus der kognitiven Psychologie besonders hervorzuheben. Sie haben einen großen Einfluß auf die Programmierkonventionen, die in Abschnitt 5.5 für die Tycoon Bibliotheken aufgestellt werden.

Standpunkt des Programmierers

Welches mögen die Beweggründe eines *Programmierers* sein, den Code eines anderen nicht zu benutzen?

- Es bringt mehr Spaß, alles selbst zu implementieren.

- Es ist ein Zeichen der Schwäche, es nicht selbst zu können.

- Es ist nicht der eigene Code. Dies ist Teil des NIH[3]-Syndroms, mit dem sich ein Programmierer unentbehrlich macht, um den Arbeitsplatz zu sichern.

[3] NIH als Abkürzung für **N**ot **I**nvented **H**ere, engl., heißt *nicht hier erfunden.*

- Es ist einfacher, eine Komponente selbst zu schreiben als eine bestehende zu suchen, zu verstehen und herauszufinden, ob sie funktioniert. Falls sie modifiziert werden muß, ist es ebenfalls bequemer, sie neu zu schreiben.

- Es ist nicht genügend Speicherplatz frei, um die letzte Version für eine Wiederverwendung aufzuheben.

- Es existieren keine Werkzeuge, um wiederverwendbare Codeteile zu identifizieren.

- Es gibt keine Software-Entwicklungsmethoden, die die Wiederverwendung unterstützen, und so den Entwurf und die Spezifikation zusammenhanglos stehen lassen.

- In der Systemanalyse wurde versäumt, die Teile eines existierenden Systems zu bezeichnen, die wiederverwendet werden können.

- Schon in der akademischen Welt wird die Wiederverwendung von Software vernachlässigt, so daß die Studenten keine Motivation dafür erhalten.

- Der in Frage kommende Code ist nicht unterstützt, zu langsam in der Ausführung, hat falsche Datentypen als Parameter, ist zu umfangreich, in der falschen Sprache vorhanden, in der richtigen Sprache auf einem anderen Prozessor vorhanden oder nicht nach den geltenden Standards entworfen.

Standpunkt des Softwaremanagers

Der Wiederverwendung von Software in Projekten stehen aus Sicht des *Softwaremanagers* folgende Probleme entgegen:

- Wenn keine Werkzeuge zur Hilfe bei der Wiederverwendung oder wiederverwendbare Komponenten existieren, kostet es Zeit und Geld, sie zu erstellen und Erfahrung damit zu sammeln. Diese Kosten sind meist nicht im Etat eines einzelnen Projektes vorgesehen.

- Sind Werkzeuge vorhanden, so ist das Projekt von weniger Mitarbeitern zu schaffen. In einem kleinen Team ist das Ausfallrisiko durch Krankheit höher und ein kleineres Team mindert das Ansehen des Managers.

- Werden spezielle Werkzeuge genutzt, möchte der Kunde diese auch zur Programmwartung mitgeliefert haben.

- Die Eigentumsfrage von wiederverwendeten Komponenten muß geklärt werden.

- Die Verantwortlichkeiten für die auftretenden Fehler müssen zusätzlich geklärt werden.

- Wenn keine Standards vorhanden sind, um die Bibliothek zu kontrollieren, müssen diese mit zeitlichem und finanziellem Aufwand erstellt werden.

Es ist viel Erfahrung in der Wiederverwendung von Software nötig, um solche Projekte zu verwalten und zu leiten. Politische und personelle Erwägungen überdecken häufig die technischen Aspekte.

Standpunkt der kognitiven Psychologen

Auch aus der Sicht der kognitiven Psychologie gibt es Gründe, warum die Software nicht wiederverwendet wird. Das Programmieren an sich ist eine Form der Problemlösung. Das Verstehen der Verdienste der bestehenden Programmierparadigmen aus der Sicht der kognitiven Psychologie hat wertvolle Einsichten in das Behandeln von Komplexität gebracht. Es folgt eine Zusammenfassung der empirischen Ergebnisse von kognitiven Studien über wiederverwendbare Software [Tracz 88]:

- Das Kurzzeitgedächtnis beschränkt die Anzahl der Teile von Information, die gleichzeitig bewußt manipulierbar sind, auf $7 +/- 2$. Durch Modularisierung von Komponenten, d.h. Zusammenfassen von Informationseinheiten in ein semantisch sinnvolles Paket, kann diese Grenze der Komplexität eingehalten werden, obwohl die Gesamtanzahl der Einheiten sehr viel größer ist. Diese Modularisierung unterstützt auch die Paradigen der *Kapselung* und des *objektorientierten Programmierens*, welche bei der Erstellung von wiederverwendbarer Software hilfreich sind.

- Programmierexperten entwickeln Anwendungen in einem rekursiven, mentalen Prozeß, in welchem sie Teile des Anwendungsproblems in Übereinstimmung mit ihnen schon bekannten Lösungen bringen. So werden unbewußt Teile eines Entwurfs bei jeder Programmerstellung wiederverwendet.

- Die interne Konzeptualisierung der Wissensbasis mit Programmsegmenten bekommt mit wachsender Erfahrung einen einheitlichen Inhalt für alle Programmierer. Programmierer tendieren dazu, gleich zu denken und ihre Lösungen in ähnlicher Form auszudrücken.

- Programmierer können Dinge, die sie nicht verstehen, nicht wiederverwenden. Programmierexperten folgen bestimmten, expliziten Handlungsregeln, wie Namenskonventionen oder Programmierstil, die die Lesbarkeit und Verständlichkeit erhöhen. Dies bedeutet für Dinge, die wiederverwendet werden sollen, daß sie nach einem anerkannten Standard geschrieben und dokumentiert sein müssen.

Da eine unstrukturierte, lineare Folge von Befehlen für einen Menschen nur schwer zu überblicken ist, wird eine Struktur in höhere Programmiersprachen eingebracht [Wirth 72]. Die Methode des *strukturierten Programmierens* wurde in den siebziger Jahren propagiert, um den Menschen Hilfsmittel zu geben, mit ihren Ausdrucksmöglichkeiten Befehle für einen Computer zu schreiben, der auch komplizierte Prozesse nur in monotonen Befehlsfolgen versteht und verarbeitet. Dijkstra [Dahl et al. 72] fordert, daß die Struktur der Berechnung in der Struktur des Programmtexts wiederzufinden ist. Diese Methode sollte weitgehend das Programmieren mit GOTO-Anweisungen eliminieren. Ein strukturiertes Programm besteht aus einer linearen Folge von Strukturblöcken, die wiederum aus Programmschleifen und Verzweigungen bestehen. Dazu wird ein Ansatz der schrittweisen Komposition von Programmen nötig. Die zu berechnende Aufgabe wird zuerst allgemein beschrieben, um dann Schritt für Schritt verfeinert zu werden. Die Verständlichkeit und auch die Vergleichbarkeit von Programmen wird hierdurch erhöht.

Die strukturierte Programmierung allein ist aber noch nicht ausreichend, um die Lücke zwischen den technischen Möglichkeiten eines Computers und der menschlichen Wissensverarbeitung zu schließen. Die obigen Beobachtungen der kognitiven Psychologie unterstützen den Bedarf einer technisch unterstützten Programmierumgebung, um die Wiederverwendung von

Komponenten zu erleichtern und zeigen intuitiv die Gültigkeit eines Ansatzes zur Modularisierung, Bibliotheksbildung und Standardisierung. Werkzeuge zur Komplexitätsbehandlung müssen verfügbar sein und den Programmierer beim Suchen und Verstehen von vorhandenen Softwarekomponenten unterstützen.

Außer den obigen, psychologischen Wiederverwendungsproblemen, die durch Einsatz von entsprechend unterstützenden Software-Werkzeugen und Schulung der Projektmitarbeiter weitestgehend aus dem Weg geräumt werden können, gibt es in praktischen Anwendungen und Produktionsprojekten Probleme bei der Wiederverwendung von Komponenten aus Klassenbibliotheken. Einige Bibliotheken verhindern durch ihre Strukur und die internen Konzepte eine Wiederverwendung, anstatt den Programmierer auf den Weg der Wiederverwendung zu leiten [Schmid 94a]. Im folgenden Abschnitt werden diese Probleme an einem Beispiel erläutert und technische Hilfsmittel zu ihrer Beseitigung vorgestellt.

1.2.2 Forderungen an Bibliotheken

Um die Wiederverwendung von Bibliotheksmodulen zu vereinfachen, werden in [Schmid 94a] Anforderungen an Bibliotheken aus einer praktischen Anwendung abgeleitet. Diese Anwendung ist ein Beispiel zur Benutzung von kommerziellen Behälterklassen-Bibliotheken, die in der objektorientierten Programmiersprache C++ geschrieben sind. Die Erfahrungen mit diesen Bibliotheken für objektorientierte Programmiersprachen haben Gründe offen gelegt, die einen Anwendungsprogrammierer daran hindern, Teile der Bibliothek wiederzuverwenden. Die abgeleiteten Forderungen basieren auf Ideen aus [Uhl, Schmid 90]. Das hier zitierte Beispiel ist mit seinen Konsequenzen auf nicht-objektorientierte Sprachen übertragbar. Die daraus folgenden Anforderungen an Bibliotheken sollen daher bei der Neugestaltung der TL-Bibliotheken berücksichtigt werden.

Behälterklassen bestehen aus Elementen, die alle den gleichen Typ haben. In einer Art horizontaler Komposition kann eine Behälterkomponente verschiedene Behälterklassen für verschiedene Elementtypen zusammenstellen, also generisch sein. Für eine umfangreiche Wiederverwendung ist es notwendig, daß eine Behälterkomponente polymorph ist, d. h. mit allen sprachdefinierten oder benutzerdefinierten Elementtypen arbeiten kann.

Die Probleme, die bei der Zusammensetzung von Behältern aus Elementen bestehen, werden im folgenden an einem Beispiel aus [Schmid 94a] erläutert. Als Fallstudie wird eine Anwendung gewählt, welche die Häufigkeit von Worten in einem Text zählt. Der Benutzer kann auf die Worthäufigkeiten durch eine nach der Häufigkeit geordnete Sammlung zugreifen. In der ersten Phase werden die Worte gezählt, wobei das Paar *Wort* - *Anzahl* nach dem Wert des Wortes ausgewählt wird. In einer zweiten Phase werden die Worte entsprechend ihres Häufigkeitswertes ausgegeben. Daher liegt es nahe, zwei verschiedene Behälterkomponenten in den unterschiedlichen Phasen zu benutzen.

Für die entstehenden Probleme wird vorerst nur die erste Phase betrachtet. Hierfür benötigen wir als Hauptoperation eine *add*-Funktion, die folgendes ausführt: wenn das Wort nicht in dem Behälter vorhanden ist, wird es mit der Häufigkeit 1 gespeichert; wenn es schon vorhanden war, dann wird die Häufigkeit um eins erhöht. Die Struktur der Daten ist in dieser ersten Phase des Beispiels eine Abbildung von Worten zu Häufigkeiten. Dazu wird eine Behälterkomponente *map*, in einigen Bibliotheken auch *directory* oder *dictionary* genannt, mit dem Wort als Schlüsselwert gewählt.

Elementgenerizität

Bei der Realisierung konnten folgende Probleme bei der Elementunterstützung vorhandener Behälterklassen beobachtet werden:

- **Einschränkung auf Elementtypen:** Normalerweise ist der Bereich der Elementtypen eingeschränkt oder es gibt bestimmte Bedingungen für den Elementtyp, daß er beispielsweise bestimmte Funktionsnamen enthalten muß. Die Anpassung der Elementtypen an die Erfordernisse der Behälterkomponente kann in nicht-produktiven Anwendungen erfolgen, aber in Produktionsprojekten verhindert die unabhängige Entwicklung der Behälterklassen und Elementtypen durch verschiedene Teams die Wiederverwendung.

- **Fehlende Elementunterstützung:** Es sei vorausgesetzt, daß die Zeichenkette in unserer Anwendung durch den Typ *String* nach C-Konvention dargestellt werden soll. Der Behälter *map* der Klassenbibliothek, der die Elementtypen für den Schlüssel und den Wert enthält, muß nun mit dem Datentyp für das Wort als Schlüssel und dem Integertyp für den Wert zusammengestellt werden. Beispielsweise kann in der Booch Library für C++ [Booch, Vilot 90] der Typ für den Schlüssel einer *map*-Klasse ein primitiver Typ oder eine benutzerdefinierte Klasse sein. Dieses schließt den Typ für Zeichenketten, der in C++ formal mit *char** bezeichnet wird, *nicht* mit ein. Ein Wort kann also nicht als eine null-terminierte Zeichenkette dargestellt werden.

  Auch die Ersetzung des abstrakten Typs *T* durch den Typ *char** für Zeichenketten in der Definition der *map*-Klasse würde keine Lösung sein, da die Operatoren "=" und "==" dadurch auf den Zeigern statt auf den Werten arbeiten und somit statt der C-Strings Zeiger in die Anwendungsklasse eingefügt würden bzw. Worte als ungleich betrachtet würden, wenn sie an unterschiedlichen Speicherstellen stehen.

  Eine Verwendung der *map*-Komponente der Booch-Library wäre also nur für Anwendungen möglich, in denen man die Komponente für das Wort in der Worthäufigkeitstabelle ändern kann.

- **Allgemeines Basisklassenproblem:** In Bibliotheken, die in Smalltalk-ähnlicher Weise eine Baumstruktur in der Vererbungshierarchie aufweisen, ist es nötig, daß der Elementtyp vom Wurzelobjekt *Object* abgeleitet wird. Für unser Beispiel bedeutet dies, daß der Schlüssel und die Wertkomponente der *map*-Klasse von *Object* abgeleitet sein müssen. Dies schränkt den möglichen Anwendungsbereich zu sehr ein.

Generell ist die fehlende Generizität ein Kriterium, welches die Wiederverwendung verhindert. Daher wird folgende Forderung an die Behälterkomponenten einer Bibliothek gestellt:

> **Elementgenerizität**: Eine Behälterklassen-Bibliothek soll angemessene Generizität unter Beachtung der Arten von Elementtypen vorsehen, so daß alle vorkommenden Klassen von Elementtypen (sprachdefinierte Typen und benutzerdefinierte Typen mit oder ohne Konstruktoren und Destruktoren) mit den Behälterkomponenten zusammengestellt werden können.

Generizität der Repräsentation und Implementation

- **Fehlende Repräsentation:** Es sei angenommen, daß eine Liste mit Worthäufigkeiten in einem Text durch die *map*-Klasse in der Booch Library implementiert ist. Dann erhält man großen Bedarf an Speicher:
  - Ein Wort wird durch Zeiger auf ein variables Array repräsentiert.
  - Die *map*-Repräsentation benutzt ein Hash-Verkettungsschema, welches einen Zeiger auf eine verkettete Liste von Elementen in einem Hashtabellen-Eintrag speichert. Wegen der gespeicherten Verwaltungsinformation und der Listenzeiger erhält man einen zusätzlichen Speicherbedarf pro Tabelleneintrag und pro Element. Durch eine andere *map*-Repräsentation mit einer Tabellenüberlaufliste oder einem offenen Hashing wäre dieser zusätzliche Bedarf auf nahezu null Bytes zu reduzieren. Leider bieten die Bibliotheken nicht immer verschiedene Repräsentationsformen an.
- **Auswahl der Repräsentation:** Die Wahl der Repräsentation ist von der Anwendung abhängig. Daher ist für die Wiederverwendung eine größere Anzahl von verschiedenen Repräsentationen in der Bibliothek nötig.

Bei den Repräsentations- und Implementationsvarianten ist die Performanz ein entscheidendes Kriterium. Eine Lösung mit Wiederverwendung sollte die gleiche Zeit- und Speicherperformanz aufweisen wie eine kundenorientierte Anwendungsentwicklung ohne Wiederverwendung.

Als Anforderung an die Repräsentationen von Behälterkomponenten wird daher formuliert:

> **Generizität der Repräsentation und Implementation**: Eine wiederverwendbare Bibliothek soll alle grundlegenden, allgemein bekannten Repräsentationen und Implementationen für jede ihrer Komponenten zur Auswahl stellen.

Funktionale Vollständigkeit

Funktionale Vollständigkeit der Dienste einer wiederverwendbaren Klasse wird als Angebot einer minimalen Menge von einfachen, orthogonalen Diensten angesehen, aus dem jeder gewünschte Dienst zusammengesetzt werden kann. Dies reicht nicht aus, um Wiederverwendung unter allen Umständen zu gewährleisten.

Als Beispiel sei der minimale Satz von Operationen *insert*, *find*, *remove* und *get* gegeben, um in einer Liste von Worten zu navigieren. Eine *add*-Funktion würde damit in dem Fall, daß das Wort schon vorhanden ist (dieses wird mit *find* festgestellt), nacheinander ein *get*, *remove* und *insert* durchführen. Jede dieser Änderungsfunktionen durchläuft die Datenstruktur von neuem. Damit würde die Zeit zur Ausführung für ein vorhandenes Wort um 100% bis 200% über der Zeit für das Einfügen eines nicht vorhandenen Wortes liegen. Reduziert werden kann dieser Zeitbedarf nur durch Funktionen, die mehrere der gegebenen Funktionen ersetzen und somit nur einmal die Datenstruktur durchlaufen. Eine Funktion *modify* ersetzt die nacheinander ausgeführten Funktionen *remove* und *insert*.

Im Falle, daß das Wort in der *add*-Funktion nicht gefunden wird, muß anschließend beim Einfügen des Wortes mit *insert* eine zusätzliche zweite Suche durchgeführt werden. Es muß also eine Methode vorgesehen werden, die entweder den Eintrag mit dem gesuchten Schlüssel

findet und einen Cursor setzt oder zurückgibt, oder aber einen neuen Eintrag einfügt. Eine solche Operation *modifyOrInsert* durchläuft nur einmal die Datenstruktur. In Bezug auf die angebotenen Dienste wird daher gefordert:

Funktionale Vollständigkeit: Eine funktional vollständige Menge an Diensten soll nicht die minimale Menge sein, sondern auch zusammengesetzte Funktionen enthalten, um

- eine grundlegend bessere Performanz als mit einfachen Diensten zu erhalten,
- dem Benutzer die Arbeit für das mehrmalige Schreiben von gleichem Code bei jeder Verwendung zu ersparen.

Vollständigkeit der Bibliothek

Verschiedene Klassenbibliotheken enthalten auch unterschiedliche Arten von Behälterkomponenten. Diese sollten möglichst vollständig sein.

Ein geordneter Behälter kann eine Tabelle oder ein geordneter Binärbaum sein, die sich durch den Zeitbedarf beim Einfügen eines Elements unterscheiden. In den Booch und Microsoft Klassenbibliotheken für C++ findet man keinen geordneten Behälter. Benutzt man ungeordnete Behälter für geordnete Mengen in der Anwendung, dann muß man die Sortieralgorithmen neu implementieren. Als Forderung sollte eine Bibliothek möglichst alle Arten der in Tabelle 1.2 aufgeführten Behälterkomponenten enthalten.

Weitere, grundlegende Komponenten können durch Kombinieren der Basiskomponenten erstellt werden. Die am häufigsten benötigten Klassen sind in Tabelle 1.3 zusammengefaßt.

Wiederverwendung ist nur effektiv, wenn die Bausteine veränderbar sind und eine voneinander unabhängige Entwicklung erlauben. Weiterhin hat Parnas ([Parnas 72]) bereits 1972 erkannt, daß diese Bausteine für den Leser des Quelltextes verständlich sein müssen und einige Kriterien zur Zerlegung von Systemen in Module zusammengestellt. Um diese Voraussetzungen zu erfüllen, werden bezüglich der Semantik folgende Qualitätskriterien für die Wiederverwendung in [Bertram 95] aufgestellt:

- Eine wohldefinierte Anwendungssemantik bedeutet, daß jeder Baustein eine eindeutig definierte Menge von Anwendungsfakten repräsentiert.
- Eine präzise Spezifikation läßt auch den Anwendungsprogrammierer verstehen, wie und wofür der Baustein angewendet werden kann. Die wesentlichen Konzepte sind enthalten und werden auch in größeren Zusammenhängen beschrieben, wodurch die Dokumentation bestimmt wird.
- Eine minimale kognitive Distanz wird erreicht, wenn das Finden, Verstehen und Anwenden eines Bausteins geringeren Aufwand erfordert als die Neuerstellung des gewünschten Anwendungsabschnitts.
- Die präzisen Nutzungsregeln geben an, in welchem Umfang der Baustein genutzt werden kann und zeigt dessen Grenzen auf.

Vollständigkeit der Bibliothek:
Aus [Uhl, Schmid 90] und den Smalltalk-Klassen läßt sich folgender, minimaler Katalog von Basiskomponenten einer Behälterklassen-Bibliothek entwickeln:

Behälterklasse	Bedeutung	Eigenschaften
linked list oder *sequence*	verkettete Liste, Folge	Eine Liste ist linear und azyklisch. Ihr erstes und letztes Element sind ausgezeichnet. Eine Konstruktoroperation bestimmt die Struktur und die Position des einzufügenden Elements. Einfügen und Löschen ist an jeder Stelle möglich, die Relation zwischen Index und Element wird dadurch nicht verändert. Es ist keine Ordnung der Elemente vorhanden.
tabular list oder *tabular sequence*	tabellarische Liste, tabellarische Folge oder Array	Eine tabellarische Liste ist eine *linked list* mit besonderen Eigenschaften. Die Elemente werden in Bezug auf ihre Indizes verschoben, d.h. daß die Relation zwischen Index und Element durch eine Einfüge- oder Löschoperation verändert wird.
tree und *binary tree*	Baum und Binärbaum	Ein Baum besteht aus einer Vater-Sohn-Beziehung zwischen den Elementen. Jeder Sohn hat höchstens einen Vater, die Beziehung ist azyklisch. Genau ein Element, das Wurzelelement, hat keinen Vater. Einfügen und Löschen ist an jeder beliebigen Stelle des Baums möglich. Ein Binärbaum ist ein Baum, in welchem jeder Vater nur maximal zwei Söhne hat. Daher besitzt er die Eigenschaft, tabellarisch repräsentiert werden zu können. Außerdem ist nur für ihn ein Durchlaufen der Elemente für eine Infixdarstellung der Verknüpfung möglich.
sorted collection	Ordnung	Es besteht eine totale Ordnungsrelation auf dem Elementtyp. Die Position eines einzufügenden Elements ist implizit durch dessen Wert bestimmt. Ein spezieller Fall einer Ordnung entsteht, wenn die Ordnungsrelation nur auf einem Teil des Elements, dem *Schlüssel*, besteht. Der Rest des Elements wird *Info-Typ* genannt. In diesem Fall sind spezielle Operationen für den Schlüsseltyp nötig. Die Implementation einer Ordnung kann als Liste oder Baum erfolgen.
set	Menge	Eine Menge enthält keine interne Struktur. Jedes Element ist genau einmal vorhanden. Sie benötigt spezielle Mengenoperationen, wie z. B. Durchschnitt und Vereinigung.
map oder *dictionary*	Abbildung, Wörterbuch	Eine Abbildung ist eine Repräsentation von finiten Funktionen vom Schlüsseltyp zum Info-Typ. Dies entspricht einer Menge mit speziellem Schlüsseltyp.
bag	Menge mit Duplikaten	Ein Bag ist eine Menge, in welcher Duplikate erlaubt sind.

Tabelle 1.2: Hinreichende Liste der Behälterklassen

Behälterklasse	Bedeutung	Eigenschaften	Implementation
stack	Stapel	Ein Stapel ist eine Liste mit eingeschränkten Einfüge- und Löschoperationen. Der Zugriff ist jeweils nur auf das erste, oberste Element erlaubt. Dies kann durch die Einschränkung der Einfüge- und Löschoperation ausgedrückt werden.	Liste
queue	Queue, Schlange	Eine Queue ist eine Liste mit eingeschränkten Einfüge- und Löschoperationen. Eingefügt werden kann nur auf der ersten Position, gelöscht werden kann nur das letzte Element.	doppelt verkettete Struktur, meist Liste
deque	Queue mit zwei Enden	Eine Deque ist eine Liste mit eingeschränkten Einfüge- und Löschoperationen. Eingefügt und gelöscht werden kann nur auf der ersten und letzten Position.	doppelt verkettete Struktur, meist Liste
priority queue	Queue mit Prioritätsangabe	Eine Priority Queue hat dieselben Eigenschaften wie eine Queue, außer daß beim Löschen nicht das letzte Element, sondern das Element mit der höchsten Priorität entfernt wird. Dazu benötigt jedes Element die Angabe einer Priorität innerhalb der Queue.	Liste
circular queue	zirkulare Queue	Diese Queue hat kein ausgezeichnetes, letztes Element, sondern ist als Ring organisiert, um ein zirkulares Durchlaufen der Queue zu ermöglichen.	Liste, deren letztes Element auf das erste Element verweist.

Tabelle 1.3: Kombinierte Behälterklassen

- Die hinreichende Universalität eines Bausteins wird durch hinreichende Abstraktion erhalten. Dadurch wird er für möglichst viele Anwendungen nutzbar und die Anzahl der zur Verfügung zu stellenden Bausteine kann klein gehalten werden. Dies verringert den Verwaltungs- und Suchaufwand innerhalb der Bibliothek.

Bezüglich der Modellierung und Konstruktion geben folgende Kriterien den Ausschlag für die Qualität einer Bibliothek [Bertram 95]:

- Die Orthogonalität sorgt für die Kombinierbarkeit von Bausteinen, um komplexere Anwendungsprobleme modellieren zu können. Dies verringert die Anzahl der Bausteine. Die Schnittstellen zu anderen Bausteinen müssen präzise definiert sein.

- Die Kompaktheit ist eine weitergehende Forderung als die Orthogonalität, da sie nicht nur Überschneidungsfreiheit, sondern auch funktionale und nicht formale Beziehungen zwischen den Objekten und Routinen innerhalb eines Bausteins fordert. Der Baustein enthält auf diese Weise eine abgeschlossene Umgebung. Dadurch wird verhindert, daß nur Teilmengen eines Bausteins verwendet werden, die gerade die für eine Anwendung relevanten Objekte und Routinen enthält.

- Die Vollständigkeit eines Bausteins bedeutet, daß er alle korrekten und relevanten Aussagen über das abgebildete Konzept enthalten muß.

- Eine hohe Flexibilität ist die Voraussetzung dafür, daß bei Veränderungen der Anwendungswelt keine Veränderung oder Austausch der Bausteine nötig wird. Hierbei ist auch das Vorhandensein von Varianten nützlich, um änderungsfreundliche Programme erstellen zu können.

- Eine wohldefinierte Schnittstelle ist die Voraussetzung für die Austauschbarkeit von Bausteinen, da anhand der Schnittstelle genau die Unterschiede und Gemeinsamkeiten zweier Bausteine zu sehen sind. Ebenso sollte die Schnittstelle möglichst wenig Redundanz enthalten.

- Varianten von Bausteinen ermöglichen die Suche innerhalb der Bibliothek nach bestimmten Kriterien. Sie erzeugen eine Struktur der Bibliothek, die von ihrem Benutzer bei der Suche ausgenutzt werden kann. Um die Zahl der Varianten zu beschränken, werden nur die Varianten von Bausteinen in die Bibliothek aufgenommen, die nicht durch Instanziierung oder Spezialisierung vorhandener Bausteine gebildet werden können.

Kapitel 2

Dokumentation

Zur Vereinfachung des Wiederverwendungsprozesses müssen Maßnahmen ergriffen werden, um ein effizientes Lesen, Erkennen und Verstehen des Codes zu ermöglichen. In erster Linie gehören zu diesen Maßnahmen hinreichende Erklärungen des Codes, damit die geschriebenen Bibliotheksbausteine auch noch nach längerer Zeit oder von anderen Programmierern inhaltlich schnell erfaßt werden können.

Auf der Suche nach Komponenten aus einer Bibliothek benötigt der Programmierer Auskünfte über die einzelnen Komponenten. Er muß wissen, ob zwei nicht in Beziehung stehende Klassen zusammen in seiner Anwendung einsetzbar sind oder welchen Umfang die Funktionalität einer Klasse hat. Diese detaillierte Information bildet die Dokumentation, die sich von der vom Compiler benötigten Information dadurch unterscheidet, daß sie meist umgangssprachlich ohne formale Regeln formuliert ist und vom Compiler ignoriert wird. Die Dokumentation von Programmcode ist somit ein wichtiger Aspekt für Bibliotheken, weil sie die Wiederverwendung fördert. Ebenso wie das Layout des Quelltextes ist die Erstellung einer brauchbaren Dokumentation ein Zeichen von Professionalität des Programmierers und somit elementar für den Programmierer von Bibliotheken.

Die Dokumentation eines Software-Projektes besteht aus Information innerhalb und außerhalb des Quelltextes. *Externe* Dokumentation besteht aus separaten Dokumenten oder Ordnern über die Teilprojekte. Je umfangreicher ein Projekt ist und je formaler die Richtlinien zur Verwaltung des Projekts sind, desto mehr Dokumentation besteht außerhalb der Quelltexte. Tatsächlich gehen ungefähr zwei Drittel der Entwicklungsarbeit in die Erstellung der Dokumentation [Boehm, others 84].

Im Gegensatz zu externer Dokumentation steht die *interne* Dokumentation im Quelltext selbst. Es ist die detaillierteste Art der Dokumentation, da sie auf der Ebene der Anweisungen eingebracht wird. Da sie in derselben Datei wie der Programmiersprachencode steht, bleibt sie bei Modifikationen des Codes auch eher konsistent mit dem Code als die externe Dokumentation, weil der Programmierer beim Ändern des Codes gleich die dazugehörigen Erläuterungen vor Augen hat und sie anpassen kann, während er im anderen Fall erst den richtigen Ordner und dann die passende Stelle für die Modifikation suchen muß.

Externe Dokumentation besteht aus einer Sammlung von Ordnern und Dateien, welche getrennt vom Programmcode aufbewahrt werden und die Entwicklung sowie die Benutzung des Programms beschreiben.

Dazu gehören *Entwicklungseinheitenordner* (UDF = *unit development folder* bzw. SDF = *software development folder*) als informelle Dokumente, welche die Notizen eines Ent-

wicklers während der Konstruktion enthalten. Eine Einheit ist dabei entweder ein Modul oder eine Routine. Der Hauptzweck ist, eine Dokumentation der Historie der Entwurfsentscheidungen aufzubewahren. Der Standardinhalt eines UDF wird vom Projekt festgelegt. Es können Schaubilder über die Architektur des Systems sowie der zugehörigen Einheiten, z. B. in Kontext- oder Zustandsdiagrammen, zum Umfang der externen Dokumentation gehören.

Entwurfsdokumente auf unterster Ebene sind *Detailentwurfsdokumente*. Sie beschreiben die Entwurfsentscheidungen auf Modul- oder Routinenebene mit allen Alternativen und den Gründen für die endgültige Wahl. Wenn diese Information in einem formalen Dokument steht, wird der Detailentwurf als unabhängig vom Konstruktionsprozeß angesehen. In einigen Fällen besteht der Detailentwurf dagegen lediglich aus Notizen des Entwicklers in einem UDF, häufig existiert er auch allein im Quellcode und ist damit Bestandteil des entworfenen Programms.

Interne Dokumentation setzt sich aus zwei Arten von Information zusammen, die auf der Ebene der Quellcodeanweisungen existieren. Zum überwiegenden Teil besteht die Dokumentation auf der Ebene des Quellcodes aus einem guten Programmierstil. Ein guter Programmierstil vereint

- eine übersichtlich gestaltete Programmstruktur,
- Benutzen von geradlinigen und leicht verständlichen Methoden,
- inhaltlich sinnvolle und lesbare Variablennamen,
- aussagestarke Routinennamen,
- Benutzen von Namenskonstanten statt Literalen,
- übersichtliches und leicht erkennbares Layout sowie
- Minimierung des Kontrollflusses und
- Minimierung der Komplexität der Datenstrukturen.

Mit einem derartig selbstdokumentierenden Code kann man die Lesbarkeit und Verständlichkeit erhöhen.

Zusätzlich zum Programmcode im Quelltext stehende Kommentare bilden den zweiten Bestandteil der internen Dokumentation. Sie sollen die Lesbarkeit des Programms verbessern, die Verständlichkeit erhöhen und die Entwurfsentscheidungen erläutern. Vom Compiler werden diese natürlichsprachlichen Texte ignoriert.

Die externe Dokumentation wird im Rahmen dieser Arbeit nicht weiter berücksichtigt, weil sie von Analyse- und Entwicklungsmethoden sowie den Anwendungsgebieten abhängt, während diese Arbeit speziell die Unterstützung der Implementation zum Thema hat.

Im weiteren Verlauf dieser Arbeit werden die Aspekte einer hilfreichen, internen Dokumentation und eines guten Programmierstils diskutiert und anhand von Regeln genauer spezifiziert. Diese Regeln sind allgemein für Programmiersprachen höherer Ordnung formuliert. Wenn sie in einigen Fällen für TL nicht anwendbar oder überflüssig sind, dann wird dies jeweils angemerkt. Um die Kreativität des Programmierers nicht zu unterdrücken, sollen diese Regeln nur als Anleitung dienen, von welcher der Programmierer abweichen darf, um einen optimal lesbaren und verständlichen Programmierstil zu entwickeln. Die allgemeine Regel, daß einem

Projekt eine Konvention zur Namensvergabe zugrunde liegen sollte, muß allerdings von jedem Programmierer befolgt werden.

Für die Entwicklung der Richtlinien werden im Folgenden zuerst die Kategorien von Kommentaren im Quelltext ermittelt. Anhand der verschiedenen Sprachkonstrukte einer höheren Programmiersprache werden Richtlinien für eine effiziente Kommentierung aufgestellt. Danach wird kurz darauf eingegangen, wie ein schnell vom Leser zu erfassender Drucksatz (engl. *Layout*) für eine Programmiersprache aussehen sollte. Als Beispiel für ein Regelwerk zur Erstellung von Programmen mit Dokumentation wird das Buch-Paradigma von Oman und Cook kurz erläutert, da es die erstellten Richtlinien für Kommentare und Layout wirkungsvoll unterstreicht.

In einem letzten Abschnitt über das Thema Dokumentation werden Richtlinien zur Benennung von Objekten und Operationen, die in einer höheren Programmiersprache repräsentiert werden, aufgestellt. Hier finden sich die Regeln, die für eine einheitliche Bibliothekserstellung mit selbstdokumentierendem Code unerläßlich sind. Diese Regeln werden in einem späteren Abschnitt kurz zusammengefaßt und dienen als *Style Guide* (Stil-Richtlinien) für TL.

Zusammengefaßt werden in diesem Kapitel Regeln zur Erstellung von Layout, Kommentaren und Routinen-, sowie Variablennamen gegeben. Diese Aspekte eines guten Programmierstils sind elementar für die Verständlichkeit von Bibliotheksbausteinen. Die weiteren, oben angeführten Aspekte eines guten Programmierstils betreffen die Programmstruktur, den Kontrollfluß und die Komplexität von Datenstrukturen. Sie beeinflussen die Dokumentation nur indirekt, indem z. B. die Programmstruktur die Vorgabe für die Struktur der Kommentare liefert. Da sie aber nicht die Dokumentation bilden und ein Nennen der Richtlinien zur Modularisierung, Datenflußkontrolle und Bildung von Datenstrukturen im Rahmen dieser Arbeit zu umfangreich wäre, wird auf die Literatur über Software Engineering verwiesen [McConnell 93].

2.1 Strukturierung, Layout und Namenskonventionen

Einige der wichtigsten Regeln zum Erlangen eines gut lesbaren Codes sollen hier aufgeführt werden. Als Bestandteil der Dokumentation gehört zum einen die überschaubare Struktur des Kontrollflusses und der Daten, zum anderen die eindeutige Benennung der Objekte zu einem guten Programmierstil. Dabei ist es unerheblich, ob funktional, imperativ oder objektorientiert programmiert wird. Wichtig vielmehr ist, daß der entstandene Quelltext auch von anderen Programmierern mühelos verstanden wird.

Die Bedeutung von Layout und Namenskonvention wurde für das Tycoon System früh erkannt. [Matthes et al. 94] betonen: "Also adherence to common conventions makes automatic formatting tools more easier to provide and more useful." (siehe ebenso [Rovner et al. 85]). Daher ist eine Menge an Layout-Regeln und Namensrichtlinien für Tycoon bereits in den Tycoon-Berichten [Matthes, Müßig 93] und [Mathiske et al. 93] beschrieben worden. Allerdings wird in den folgenden Abschnitten anhand von Beispielen deutlich, daß es noch zusätzliche Regeln geben muß, um die Programmierung in TLeinfach, einheitlich und verständlich zu machen. Die Beschreibung dieser Richtlinien für TLerfolgt in Abschnitt 5.5.

Die Richtlinien für das Layout in TLsind in den Tycoon-Berichten sehr ausführlich dargelegt. Sie bieten eine umfassende Grundlage zur Erstellung eines guten Layouts. Daher sollen in dieser Arbeit keine weiteren Richtlinien zu diesem Thema aufgestellt werden.

In den nächsten Abschnitten werden allgemeine Richtlinien für Kommentare und Namen

aus der Literatur aufgegriffen und im Hinblick auf ihre Nützlichkeit für TL bewertet. Namenskonventionen werden in Abschnitt 2.4 diskutiert.

In sämtlichen Abschnitten über die Dokumentation wird allgemein vorausgesetzt, daß die Programmtexte mit einem üblichen, zeilenorientierten Textverarbeitungswerkzeug bearbeitet werden sollen, welches die Eingabe von links nach rechts erlaubt. Es wird nicht davon ausgegangen, daß das Textverarbeitungswerkzeug syntaktisch unterstützte Navigation erlaubt. Aus diesem Grund sind alle aufgestellten Richtlinien daran orientiert, einem Programmierer die Änderungen im Programmtext so einfach wie möglich zu machen. Das bedeutet, daß die Syntax und Semantik des Programms klar erkennbar sein und die Zahl der Bewegungen des Cursors möglichst gering gehalten werden sollen.

2.2 Kommentare

Kommentare sind nutzlos, wenn sie unverständlich geschrieben oder veraltet sind. In einem solchen Fall ist es besser, auf Kommentare zu verzichten. Aus diesem Grund gibt es hitzige Debatten über die Mächtigkeit der Kommentare. Um beim Kommentieren nicht die Verständlichkeit des Quelltextes zu zerstören, sollte man grundlegende Regeln berücksichtigen. In den folgenden Abschnitten werden einige Richtlinien zur Erstellung von effektiven Kommentaren angegeben.

2.2.1 Kategorisierung von Kommentaren

Kommentare können in fünf Kategorien eingeteilt werden [McConnell 93]:

Wiederholungen des Codes mit anderen Worten geben dem Leser zwar mehr zu lesen, stellen aber keine zusätzliche Information bereit.

Erläuterungen des Codes sind typisch, um komplizierte, trickreiche oder fehleranfällige Codeteile zu erklären. In komplizierten Codeabschnitten sind solche Kommentare nützlich, weil der Code allein verwirrend ist. Wenn der Code zu kompliziert ist, sollte er allerdings besser verändert werden, statt Kommentare hinzuzufügen.

Markierungen des Codes sind nicht dazu gedacht, im Code zu bleiben. Sie sollen nur die Stellen bezeichnen, an denen der Entwickler noch arbeiten muß. Diese Art der Kommentierung kann absichtlich aus einem syntaktischen Fehler in der Programmiersprache bestehen (z. B. ein Kommentarzeichen "(*****" ohne Beendigung des Kommentars), so daß der Compiler einen Fehler anzeigt und automatisch an die noch fehlende Änderung erinnert, oder spezielle Zeichen in einem Kommentar enthalten, nach denen der Entwickler einfach suchen kann, ohne die Übersetzung zu verhindern.

Zusammenfassungen des Codes von mehreren Zeilen in ein oder zwei Sätze sind wertvoller als die bloße Wiederholung. Besonders für Modifikationen sind sie dem Entwickler nützlich, wenn er den Code nicht selbst entworfen hat.

Beschreibungen der Absicht des Codes erklären den Zweck eines Code-Abschnitts. Ein derartiger Kommentar operiert mehr auf der Ebene des Problems als auf der Ebene der Lösung.

Das Hauptproblem der Programmierer, die ein Programm warten, ist das Verstehen der Absicht des ursprünglichen Programmierers. Gerade zur Klärung dieses Problems tragen die Zusammenfassungs- und Absichtskommentare bei. Zusammenfassungs- und Absichtskommentare sind nicht immer deutlich zu unterscheiden, welches in den meisten Fällen allerdings auch kein zusätzlicher Informationsgewinn ist. Es wird daher empfohlen, zur Kommentierung eines Programmfragments nur Zusammenfassungen des Inhalts und Beschreibungen der Absicht dieses Programmteils zu benutzen.

2.2.2 Effiziente Kommentierung

Effiziente Kommentierung ist nicht unbedingt so zeitaufwendig wie meist angenommen wird. Zuwenig Kommentierung in einem Programmtext ist genauso schlecht wie zuviel, aber eine dem jeweiligen Problem angemessene Kommentierung ist mit ökonomisch vertretbarem Aufwand zu erreichen [McConnell 93]. Im folgenden sollen ein paar Anleitungen zur effizienten Kommentarerstellung gegeben werden.

Ein Grund für einen hohen Zeitaufwand beim Kommentieren ist ein ineffektiver Stil. Sind Änderungen des Kommentars nur mit hohem Aufwand durchzuführen, unterbleibt das Kommentieren meist völlig. Dann wird zwar der Programmcode modifiziert, aber der Kommentar wird nicht angepaßt, so daß er ungenau und irreführend wird. Ein solcher Zustand sollte auf jeden Fall vermieden werden, da die Zeit, die ein Leser zum Verstehen des Programmfragments benötigt, länger ist als für ein Programmstück, welches nicht kommentiert ist.

Ein zweiter Grund für einen hohen Zeitaufwand ist, daß einem die Worte zur Beschreibung des Programms nicht sofort einfallen. Dies ist ein Zeichen dafür, daß das Programm nicht richtig verstanden wird. Die Zeit, die in diesem Fall zum Kommentieren verwendet wird, ist eigentlich Zeit zum Verstehen des Programms, die auch benötigt würde, wenn keine Kommentare geschrieben werden. Dieses Problem rührt somit vom Entwurfsprozeß des Programms her und kann nicht der Kommentierung angelastet werden.

Um änderungsfreundliche und aussagekräftige Kommentare zu erstellen, sollte man folgende Anweisungen beachten [McConnell 93]:

- Benutze Stile, die Änderungen nicht entmutigen oder verhindern. Jeder zu spielerische Stil ist langweilig und aufwendig zu pflegen. Dazu gehören Punktlinien, wie sie teilweise in Inhaltsverzeichnissen benutzt werden, und die Einrahmung eines Kommentars in Sterne (vgl. Abb. 2.1). Ebenso sollten Zeilen zur Unterstreichung von Kommentaren vermieden werden, die mit einem anderen Zeichen eingeschlossen werden, z. B. "+——+". In allen diesen Beispielen führt eine Änderung des Kommentartextes unweigerlich auch zu einer Änderung der beschriebenen Kommentarzeichen, weil in allen Fällen die Zeilenlänge an den Text angepaßt werden muß. Kommentare, deren einzelne Zeilen auch in der Länge geändert werden können, ohne das Gesamtbild zu zerstören, sind diesen vorzuziehen (vgl. Abb. 2.2).

- Nutze den Entwurfsprozeß von der Spezifikation zum Code, um schon im ersten Entwurfsschritt Kommentare als Beschreibung der Aufgabe in einer Art von Pseudo-Code zu entwerfen, die dann Schritt für Schritt in Anweisungen überführt werden.

  Wirth [Wirth 72] hat schon 1972 in seinem Ansatz über *systematisches Programmieren* erkannt, daß die Niederschrift eines Algorithmus in einer bestimmten Notation trivial ist. Die Hauptarbeit besteht vielmehr in seiner Konzeption und Verifizierung. In einem

```
(*************************************************************************
 * module: GIGATRON.C                                                    *
 *                                                                       *
 * author: Dwight K. Coder                                               *
 * date:   July 4, 2014                                                  *
 *                                                                       *
 * Routines to control the code evaluation tool.                         *
 *⋮                                                                      *
 *************************************************************************)

(* Table of short names for constants
   +====================+
   XPOS.....X Coordinate Position
   YPOS.....Y Coordinate Position
   PTTGT.....Point Grand Total                                          *)
⋮
```

Abbildung 2.1: Problematische Kommentarstile

```
(*************************************************************************
  module: GIGATRON.C

  author: Dwight K. Coder
  date:   July 4, 2014

  Routines to control the code evaluation tool.
  ⋮
 *************************************************************************)
```

Abbildung 2.2: Änderungsfreundlicher Kommentarstil

langwierigen Vorgang wird der Algorithmus seiner Lösung schrittweise angenähert. In jedem Schritt werden Einzelheiten genauer spezifiziert und die noch offenen Einzelheiten können vom ersten Schritt an als Kommentare grob beschrieben werden. Die Notation der Zwischenschritte muß nicht maschinenlesbar sein, sie kann aus Flußdiagrammen oder mathematischen Formeln bestehen. Allerdings sollte die Programmiersprache für die endgültige Version wegleitend sein. Daher bietet sich eine Pseudoprogrammiersprache für die Verfeinerungschritte an.

Entwurfssprachen wie PDL[1](*programming design language*) helfen, die Kommentierzeiten zu reduzieren. Die Aufgaben werden zuerst in einer dem Englischen ähnlichen Sprache beschrieben. Danach wird der Programmcode daraus gebildet, wobei die ursprüngliche Beschreibung zum Kommentar gewandelt wird. Werden im ersten Beschreibungsschritt schon die weiteren, hier angegebenen Richtlinien für Kommentare befolgt, entstehen vollständige und sinnvolle Kommentare nach Beenden des Kodierens.

- Kommentiere gleich beim Erstellen des Codes. Die Alternative, erst zum Ende des Projektes zu kommentieren, hat viele Nachteile. Dann erscheint die Kommentierung auf einmal sehr viel mehr Arbeit zu sein, als wenn sie Stück für Stück erledigt wurde. Am Ende des Projektes müssen außerdem die Code-Bedeutung und verwendete Algorithmen nachgeschlagen oder Zusammenhänge zwischen Codeteilen gesucht werden, weil sie vom Programmierer schon vergessen wurden, und dies benötigt zusätzlich Zeit. Zudem wird der Kommentar ungenau, weil einige Randbedingungen schon vergessen sind und im Kommentar nicht mehr erwähnt werden.

  Ein Argument gegen die begleitende Kommentarerstellung ist, daß während der Codeerstellung das Nachdenken über den Code nicht unterbrochen werden sollte. Dagegen ist einzuwenden, daß es zur Erstellung eines durchdachten Programms gehört, zuerst in einer Entwurfssprache wie z. B. PDL oder anderen Techniken zu entwerfen. Entsprechend dem Prinzip der systematischen Programmierung wird zuerst der Entwurf vereinfacht, bevor dann gleichzeitig Code und Kommentar geschrieben werden. Durch diese Methode wird der Kommentar zum Code nicht vernachlässigt, sondern schon in den ersten Entwurfsschritten mit berücksichtigt. Der Programmentwurf wirkt somit selbstdokumentierend. Die Dokumentation wird auf diese Weise sorgfältig erstellt und erübrigt eine nachträgliche Dokumentierung, die aus Zeitgründen meist oberflächlich geschieht [Schulz 78].

Anzahl der Kommentare Eine optimale Anzahl von Kommentaren, z. B. ein Kommentar pro fünf Zeilen Code, kann nicht allgemeingültig angegeben werden [McConnell 93]. Eine Beschränkung der Zahl kann nur als Anleitung zur besseren Kommentierung für Programmierer dienen, die nicht allgemein verständlich programmieren. Es ist sehr viel entscheidender für die Lesbarkeit und Wiederverwendung des Codes, ob jeder geschriebene Kommentar auch effizient ist. Zur Beurteilung der Effizienz werden die vorgenannten Kriterien benutzt. Wenn alle Kriterien erfüllt werden, sind ausreichend Kommentare vorhanden.

[1] PDL wurde 1975 von der Firma Caine, Faerber & Gordon publiziert.

2.2.3 Techniken der Kommentierung

In Abhängigkeit der Ebene und des programmiersprachlichen Konstrukts, welches kommentiert werden soll, müssen verschiedene Techniken eingesetzt werden [McConnell 93].

Einzelzeilen bedürfen selten eines Kommentars. Eine einzelne Zeile sollte nur dann kommentiert werden, wenn sie einen komplizierten Ausdruck enthält oder in ihr ein Fehler auftrat, der für nachfolgende Änderungen aufgezeichnet werden soll. Als Richtlinien gelten:

- Vermeide Kommentare mit übertragenem Sinn, für deren Entschlüsselung man bestimmtes Hintergrundwissen benötigt, welches nicht aus dem Programm oder der übrigen Dokumentation hervorgeht.

  Beispiel:

```
let newCount = oldCount + 723h    (* R.I.P.L.V.B. *)
```

  R.I.P.L.V.B. steht für *Rest in peace, Ludwig van Beethoven.* Dazu muß der Leser dieses Codes wissen, daß Beethoven im Jahr 1827 starb, welches hexadezimal 723 ist. Der Kommentar hat nach dieser Entschlüsselung gar nichts mit dem Zusammenhang zu tun, daß hier gerade 723h addiert werden müssen.

- Vermeide es, Kommentare, die für mehrere Zeilen gelten sollen, ans Ende einer Zeile zu schreiben. Es wird nicht deutlich, für welche Zeilen der Kommentar gelten soll.
- Vermeide Kommentare am Ende der Zeile für Einzelzeilen. Exemplarisch sehen derartige Endzeilenkommentare folgendermaßen aus:

```
let memToInit = memoryAvailable      (* get amount of memory available *)
let pointer = getMem(memToInit)      (* get a ptr to the avail. mem *)
zeroMem(pointer, memToInit)          (* get a ptr to the avail. mem *)
⋮
freeMem(pointer)                     (* Free memory allocated *)
```

  Die Probleme solcher Kommentare sind:

  – Sinnvolle Kommentare, der nicht in die Kategorie *Wiederholung des Codes* fallen, sind für eine Zeile schwer zu finden.
  – Sie müssen am rechten Rand ausgerichtet werden und sollten eine eigene Spalte im Layout ergeben. Überschneiden sie sich mit dem Code, sieht der gesamte Quelltext unordentlich und unübersichtlich aus.
  – Sie sind schwer zu formatieren, da mehrere Endzeilenkommentare untereinander in einer Spalte ausgerichtet werden sollten.
  – Sie sind schwer zu pflegen. Bei jeder Codeänderung in dieser Zeile muß auch der Kommentar geändert werden, weil er sonst über den Rand hinausgeschoben wird und die spaltenweise Ausrichtung mehrerer Endzeilenkommentare nicht erhalten bleibt.

– Sie sind häufig kryptisch, da am rechten Rand meist wenig Platz bleibt für den Kommentar, der darum abgekürzt wird. Unter der Kürzung des Kommentars leidet dessen Verständlichkeit.

Ausnahmen von der Regel zur Vermeidung von Kommentaren am Ende einer Zeile bestehen in drei Fällen:

- Für Datendeklarationen kann der Kommentar am Ende der Zeile stehen, da dort in Programmiersprachen wie C++ oder Pascal etwas mehr Platz ist als bei Anweisungen und diese Deklarationen weniger Änderungen unterworfen sind. Weil Tycoon keine rein imperative Sprache ist und daher nicht die für imperative Programmierung üblichen Datendeklarationen am Beginn eines Moduls oder einer Funktion erfordert, wird diese Regel nicht auf Tycoon angewendet. Auch die Typ- und Wertdeklarationen mit ***Let*** bzw. ***let*** sind in Tycoon so lang, daß sie keinen Endzeilenkommentar zulassen.
- Für Anmerkungen während der Wartung kann der Kommentar am Ende der Zeile nützlich sein, wenn sie sich direkt auf diese eine Zeile beziehen. Änderungen werden dann typischerweise mit Datum und Autorenkürzel benannt, wenn kein Versionskontrollsystem vorhanden ist.
- Um das Ende eines großen Blocks, beispielsweise das Ende einer *while*-Schleife anzuzeigen, kann der Endzeilenkommentar benutzt werden.

Absätze von Code werden in ein oder zwei Sätzen zusammengefaßt, ohne den Code selbst zu wiederholen. Dabei sind folgende Regeln zu beachten:

- Beschreibe den Code auf der Ebene seiner Absicht und seines Zweckes. Dies entspricht dem Routinennamen, den der Programmierer diesem Codeabschnitt geben würde. Ebenfalls sollte der Code selbst so geschrieben sein, daß er selbstdokumentierend ist.

  Ein Beispiel:

```
(* Swap the roots *)
let oldRoot = root[0]
let root[0] = root[1]
let root[1] = oldRoot
```

  In diesem Beispiel muß bei einem Fehler im Vertauschen der Felder der Kommentar nicht verändert werden, weil er den Zweck dieses Blocks und nicht die Ausführung beschreibt.

- Beziehe die Absatzkommentare mehr auf das *Warum* als auf das *Wie*. Wenn beschrieben wird, wie etwas durchgeführt wird, dann kann nicht gleichzeitig die Absicht des Codes erklärt werden. Das *Wie* beschreibt die Vorgänge auf der Ebene der Programmiersprache und liefert dem Leser keine neue Information:

```
(* if allocating flag is zero *)
if allocFlag == 0 ...
```

  Das *Warum* beschreibt die Vorgänge im Zusammenhang mit der Problembeschreibung und ist daher eine Bereicherung des Codes:

```
(* if allocating new member *)
if allocFlag == 0 ...
```

- Erkläre mit Hilfe der Kommentare, was im folgenden geschehen soll. Nur anhand der Kommentare müssen der Zweck des Codes erkannt und spezielle Aktivitäten vom Leser lokalisiert werden können. Daß der Kommentar vor dem Code steht, ist inzwischen etablierte Konvention in der kommerziellen Praxis [McConnell 93].
- Mache jeden Kommentar wichtig. Zuviele Kommentare vernebeln den Code. Besser ist es, den Code lesbarer zu machen, um ihm mehr Gewicht zu verleihen.
- Dokumentiere Überraschungen. Alles, was nicht aus dem Code direkt hervorgeht, muß in einem Kommentar festgehalten werden. Wird statt der üblichen Technik ein Trick zur Verbesserung der Performanz angewendet, so muß der Zweck und der Gewinn durch diesen Einsatz kommentiert werden. Ein Beispiel ist das Ersetzen der Division durch Zwei bei ganzen Zahlen mit der Right-Shift-Operation.
- Vermeide Abkürzungen. Kommentare müssen eindeutig und lesbar sein, ohne ein Rätseln über die Bedeutung von Abkürzungen zu erfordern. Abkürzungen sollten nur bei Endzeilenkommentaren Verwendung finden.
- Differenziere wichtige und weniger wichtige Kommentare. In Einzelfällen ist ein Kommentar ein detaillierter Zusatz zu einem umfassenderen Kommentar. Dies kann auf verschiedene Weise hervorgehoben werden:
  - Der umfassendere Kommentar wird unterstrichen. Nachteil: Es müssen mehr Kommentare unterstrichen werden als eigentlich beabsichtigt, denn alle folgenden, nicht unterstrichenen Kommentare sind diesem untergeordnet und meist gibt es auf der Ebene des übergeordneten Kommentars weitere Kommentare. Ein Beispiel:

```
(* copy the string portion of the table, delete omitting strings *)
(*————————————————————————————————————————*)

(* determine number of strings in the table *)

(* mark the strings to be deleted *)
```

  - Alle untergeordneten Kommentare beginnen mit Fortsetzungszeichen "...". Ein Beispiel:

```
(* copy the string portion of the table, delete omitting strings *)

(* ... determine number of strings in the table *)

(* ... mark the strings to be deleted *)
```

  - Eine weitere Lösung ist, die Anweisungen des umfassenden Kommentars in eine eigene Routine zu schreiben. Innerhalb einer Routine sollten alle Aktivitäten derselben Ebene angehören, d. h. die Routine ist *flach* (engl. *flat*).

  Für eingerückten Code bei konditionalen und Schleifenkonstrukten gilt dies nicht, da der Kommentar in diesen Fällen ebenfalls der Ebene der Einrückung folgen

sollte. Dadurch werden hier die untergeordneten und übergeordneten Kommentare durch die Einrückung unterschieden.

- Dokumentiere alles, was mit einem Fehler oder einer nicht dokumentierten Eigenschaft in einer Sprache oder einer Umgebung zu tun hat.
- Dokumentiere alle Verstöße gegen die in diesem Kapitel aufgestellten Regeln für einen guten Programmierstil (vgl. Abschnitt 2). Die Gründe für einen Verstoß gegen die Regeln müssen erklärt und gerechtfertigt werden, um einen anderen Programmierer davon abzuhalten, den Code zu verbessern und damit die wohlbegründete eigene Absicht zu verwerfen.
- Versuche, trickreichen Code zu vereinfachen. Programmierer neigen dazu, trickreiche oder fehleranfällige Abschnitte des Codes mit besonders viel Kommentar zu versehen. Kommentare können die Verständlichkeit von schwierigem Code nicht bessern. Der Code sollte neu geschrieben werden, um dann mit Kommentaren nach den hier angegebenen Regeln versehen zu werden [Kernighan, Plauger 74].

Datendeklarationen werden mit Aspekten der Variablen kommentiert, die vom Namen nicht beschrieben werden können. Laut einer Studie in [Glass 82] sind Anmerkungen zu Datendeklarationen sehr viel wichtiger als Anmerkungen zu dem Code, in welchem diese Daten benutzt werden. Obwohl in TL-Programmen keine Codeabschnitte für die Deklaration von Variablen existieren, können die folgenden Richtlinien zur Dokumentation von Funktionsparametern verwendet werden:

- Beschreibe die Einheit einer numerischen Größe. Es ist wichtig zu unterscheiden, ob *Meter* oder *Kilometer* gemeint sind und ob die Bildschirmkoordinate die Höhe oder die Breite angibt. Da die Einheiten sich auch ändern können, sind sie nicht für jeden Leser offensichtlich.
- Beschreibe den Bereich der erlaubten numerischen Werte. Erwartet man bestimmte Werte für eine Variable, so werden diese auch kommentiert, insbesondere dann, wenn die Sprache, wie TL, keine Unterbereichstypen erlaubt.
- Kommentiere die kodierte Bedeutung. Werden keine Aufzählungstypen von der Sprache unterstützt, so werden Konstantennamen für jeden Wert benutzt und dessen Bedeutung in einem Kommentar festgehalten.
- Beschreibe Einschränkungen der Eingabedaten. Kommentare sind eine Möglichkeit festzuhalten, daß eine Routine bestimmte Daten nie erhalten wird. Eine weitere Möglichkeit, die nur in einigen Sprachen (z. B. Eiffel, Alphard, Euclid) geboten wird, sind Zusicherungen (siehe hierzu auch Abschnitt 3.3.4), mit denen die Einhaltung der Restriktionen geprüft werden können.
- Beschreibe Variable, die als Bitfeld benutzt werden, auch auf der Bitebene. Jedes Bit muß mit seiner Bedeutung festgehalten werden.
- Bezeichne Kommentare zu einer Variablen mit dem Variablennamen. Um sicherzugehen, daß bei einer Änderung dieser Variablen die Kommentare mitgeändert werden, kann durch eine Suche nach diesem Namen der zu ändernde Kommentar gefunden werden.
- Beschreibe globale Daten mit dem Zweck dieser Daten und nenne den Grund für ihre Globalität. An jedem Punkt, an dem sie benutzt werden, muß deutlich gemacht

werden, daß sie global sind. Wenn dieses nicht durch Namenskonvention erfolgt (siehe Abschnitt 2.4.1), muß dafür die Kommentierung sorgen.

Vorbildliche Beispiele zu den Regeln zur Kommentierung von Datendeklarationen sind:

```
CursorX :Int (* horizontal cursor position ranges from 1 ... MaxCols *)
SignalStrength :Int (* strength of signal in kilowatts: range is >= 1 *)
CharAttrib :Int (* 0=plain, 1=italic, 2=bold, 3=boldItalic *)
```

Kontrollstrukturen werden an ihrem Beginn und ihrem Ende kommentiert.

- Stelle vor jeden Block einer Kontrollanweisung einen Kommentar, der den Zweck beschreibt, die Auswahl begründet sowie das Ergebnis des Blocks zusammenfaßt.
- Kommentiere das Ende einer Kontrollstruktur, um zu zeigen, was hier endet. Das Schlüsselwort end kann in vielen Sprachen (z. B. Pascal, C, Tycoon) verschiedene Kontrollstrukturen beenden. Eine Beschreibung der Zugehörigkeit des Wortes end ist besonders hilfreich bei langen, verschachtelten Schleifen. Teilweise können die Schleifen auch benannt werden, wie z. B. in Ada.

Routinen werden von allen hier aufgezählten Strukturen in der Computerliteratur (u.a. [DIN 85]) am häufigsten mit Ratschlägen zur Kommentierung bedacht. Die meisten dieser Autoren empfehlen, zu Beginn jeder Routine als Prolog eine Menge an Information aufzulisten, ungeachtet der Größe oder Komplexität dieser Information. Ein derartiger Dokumentenkopf für Routinen ist besonders für triviale Routinen, die nur wenige Zeilen Code enthalten, zu aufwendig und steht in keiner Relation zu der Größe der Routine (vgl. Abbildung 2.3). Wird zuviel Information für jede Routine gefordert, dann ist das der Anlaß für ungenaue Kommentare und Fehler bei der Wartung. Information unter der Überschrift "Zweck" oder "Algorithmus" für eine einfache Routine anzugeben ist schwierig, weil die Ebene der Detaillierung zwischen der Wiederholung des Routinennamens und des Codes nicht deutlich ist. Andere geforderte Information kann gar nicht angegeben werden, weil sie nicht anwendbar ist.

Wird andererseits zu wenig Information verlangt, dann kann es sein, daß der Programmierer wichtige Information nicht nennt, die dadurch nicht an den Anwender dieser Routine gelangt, wodurch die Wartung und Wiederverwendung erschwert wird. Eine umgangssprachliche Beschreibung des Zweckes und der Randbedingungen einer Routine ist wichtige Information zu ihrer Benutzung, die für den Wiederverwender vollständig vorhanden sein sollte. Es muß daher eine Technik zur Kommentierung angegeben werden, welche die Vorteile beider vorgestellten Techniken vereint.

Richtlinien zur Erstellung effizienter Kommentare für Routinen sind:

- Halte den Kommentar nahe dem Code, den er beschreibt. Das geht nicht mit einem großen Dokumentationskopf vor jeder Routine, der den Bereich zwischen Code und Beginn des Kommentars der Routine vergrößert. Vom Code entfernte Kommentare werden nicht mitgepflegt, so daß der Kommentar im Dokumentationskopf mit dem Code nicht mehr übereinstimmt und somit wertlos wird. Das Prinzip der Nähe von Kommentar und Code ist Voraussetzung dafür, daß die Kommentare auch mitgeändert werden.

```
(*****************************************************************
  Name :copyString

  Purpose:  This routine copies a string from the source string
            to the target string.

  Algorithm: It gets the length of source and then copies each character,
            one at a time, into target. It uses the loop index as an array
            index into both source and target and increments the loop/array
            index after each character is copied.

  Inputs:   input              string to be copied
  Outputs:  output               string to receive the copy of input

  Interface Assumptions: None

  Modification History: None

  Author:       Dwight K. Coder
  Date Created: 10/1/92
  Phone:        (222) 555-2255
  SSN:          111-22-3333
  Eye Color:    Green
  Maiden Name:  None
*****************************************************************)
```

Abbildung 2.3: Beispiel für einen zu umfangreichen Routinenkopf

Verschiedene Komponenten von Routinenköpfen werden in den folgenden Aufzählungspunkten beschrieben und sollten dort verwendet werden, wo sie nötig sind. Als Vorlage kann ein leerer Dokumentationskopf erstellt werden. Für jeden Einsatz dieser Vorlage werden nur die Felder ausgefüllt, die auf die jeweilige Routine zutreffen, der Rest der Vorlage wird nicht verwendet.

- Beschreibe jede Routine in ein oder zwei Sätzen zu Beginn dieser Routine. Wenn es nicht gelingt, eine Kurzbeschreibung zu finden, dann sollte über das Problem noch einmal nachgedacht werden, weil der Entwurf nicht optimiert ist. Eine kurze Zusammenfassung sollte in allen Routinen vorhanden sein.
- Beschreibe die Eingabe- und Ausgabevariable dort, wo sie deklariert werden. Werden keine globalen Daten genutzt, sollte diese Beschreibung im Rahmen der Parameterdeklaration erfolgen. Dabei kann als Ausnahmefall ein Endzeilenkommentar verwendet werden, weil die Parameternamen und Kommentare im allgemeinen kurz sind und somit in eine Zeile passen. Bei einer aussagekräftigen Namensgebung können diese Kommentare auch entfallen.

  Globale Daten sollten vermieden werden. Sie werden im Kopf der Routine dokumentiert, welches sehr viel Aufwand in der Wartung bedeutet, weil der Code in einiger Entfernung zu diesem Kommentar steht. In der Praxis entfällt diese Dokumentation daher meist, obwohl gerade globale Daten besonders sorgfältig und ausgiebig dokumentiert werden müßten, um eine fehlerlose Verwendung zu garantieren, da sie Seiteneffekten unterliegen und ihre Benutzung dadurch fehleranfällig ist.
- Differenziere zwischen Ein- und Ausgabedaten. Welche Parameter als Eingabe und welche als Ausgabe der Routine dienen, ist für den Aufruf einer Routine wichtig zu wissen. Wird diese Unterscheidung nicht von der Sprache unterstützt, wie z. B. in Pascal durch das var-Schlüsselwort, muß diese Eigenschaft explizit in Kommentare gefaßt werden. Da Tycoon diese Unterscheidung von der Sprache her nicht konsequent unterstützt, da Teile von var-Parametern geändert werden können, sollte ein Kommentar, der angibt, ob die Variable innerhalb der Routine verändert wird, für jede Routine vorgesehen werden.
- Beschreibe die Annahmen für die Schnittstelle. Es werden legale und illegale Werte, Felder, die geordnet sein müssen, Datenstrukturen, die initialisiert werden oder nur passende Daten enthalten dürfen, festgehalten. Diese Annahmen über den Zustand der Variablen werden entweder im Routinenkopf oder dort, wo die Daten deklariert sind, beschrieben. Eine derartige Dokumentation muß in jeder exportierten Routine vorhanden sein. Jede Annahme sollte gleich bei ihrer Feststellung aufgeschrieben werden. Auch benutzte globale Variable gehören zur Schnittstelle und sind wegen ihrer Fehleranfälligkeit besonders sorgfältig zu dokumentieren.
- Bewahre die Änderungshistorie einer Routine auf. Änderungen werden insbesondere nach Fehlern durchgeführt und Fehler neigen dazu, sich in einigen besonders fehleranfälligen Routinen zu konzentrieren. Daher deuten Fehler in einer Routine darauf, daß sich dort noch mehr Fehler befinden können. Nach einer bestimmten Anzahl von Fehlern sollte die Routine neu entworfen werden.
- Beschreibe die Grenzen der Routine. Die Genauigkeit des Ergebnisses oder die Undefiniertheit der Berechnung unter bestimmten Voraussetzungen sind Beispiele

für solche Grenzen. Auch das Verhalten im Fehlerfall, die vorausgesetzte Größe eines Arrays oder Änderungen in der Routine, die einen Fehler hervorrufen würden, gehören dazu.

- Dokumentiere die Seiteneffekte der Routine. Das Ändern von globalen Daten muß genau erfaßt werden, da ihre Änderung gefährlich für die korrekte Ausführung werden kann. Die nichtdokumentierte Änderung einer globalen Variablen kann in einer anderen Routine einen Fehler auslösen, weil dort dieser neue Wert der globalen Variablen nicht erwartet wird.
- Dokumentiere die Quelle der benutzten Algorithmen. Wird ein Algorithmus aus einer Zeitschrift oder einem Buch verwendet, so muß die Literaturquelle genau angegeben werden. Bei einer eigenen Entwicklung sollte eine knappe Erläuterung dieses Algorithmus' erfolgen.
- Markiere Teile des Programms mit einem speziellen Kommentar, um sie auf einfache Weise mit Hilfe eines Textverarbeitungswerkzeugs wiederzufinden. Wird z. B. jeder Anfang einer Routine mit der Zeichenfolge "(*%&" markiert, so kann von Routine zu Routine gesprungen werden, indem nach dieser Zeichenfolge gesucht wird. Ebenso können auch verschiedene Arten von Kommentaren durch unterschiedliche Zeichenfolgen markiert werden. Das erlaubt ein Herausfiltern dieser Kommentare aus dem Quelltext mit Hilfe von Werkzeugen.

  Für Tycoon wird aus diesem Grund eine Richtlinie entwickelt, die ein Bearbeiten der verschiedenen Arten von Kommentaren ermöglichen (siehe Abschnitt 5.5.1). Mit geeigneten Werkzeugen, die für Tycoon-Programme noch erstellt werden müssen, kann automatisch eine Übersicht über alle oder ausgewählte Routinen eines Moduls oder die Module einer Bibliothek erstellt werden.

Dateien, Module und Programme werden dadurch charakterisiert, daß sie mehrere Routinen enthalten. Die Aufgabe der Dokumentation ist es, eine sinnvolle Übersicht auf allgemeiner Ebene über den Inhalt der Datei, des Moduls oder des Programms festzuschreiben. Die folgenden Richtlinien gelten für alle drei erwähnten Konstrukte. Im erläuternden Text wird zur Vereinfachung stellvertretend das Wort *Datei* auch für Module und Programme benutzt. In der diese Richtlinien befolgenden Sprache TLIST es vorgeschrieben, daß eine Schnittstelle, deren Modulimplementation und ein Programm jeweils in einer eigenen Datei gespeichert werden. Die Unterscheidung in *Datei*, welche aus der Sicht des Betriebssystems existiert, und Modul bzw. Schnittstelle bzw. Programm, welche aus der Sicht der Programmiersprache existieren, erübrigt sich damit.

Am Beginn einer Datei sollte ein Kommentar stehen, der den Inhalt der Datei nach folgenden Regeln beschreibt:

- Beschreibe den Zweck der Datei. Die Zugehörigkeit jeder Routine zu dieser Datei sollte evident sein oder begründet werden. Das ist insbesondere für den Anwendungsprogrammierer eine entscheidende Information, wenn er eine Routine zur Erfüllung einer bestimmten Aufgabenstellung für seine Anwendung in der vorhandenen Bibliothek sucht. Das Anwendungsgebiet jeder Datei muß erkennbar sein. Falls dies nicht der Fall sein sollte, wird empfohlen, eine Aufteilung der für diese Datei vorgesehenen Routinen in mehrere Module mit deutlich voneinander getrennten Anwendungsgebieten vorzunehmen.

Bei Verwendung einer Programmiersprache, die das Konzept der Modularisierung nicht vorsieht, wird die Kurzbeschreibung des Zwecks wichtiger, weil es dann für einen potentiellen Verwender der Datei keine offensichtlichen semantischen Zusammenhänge der in der Datei aufgeführten Routinen gibt. Jede Information, die hilfreich zum Auffinden einer gesuchten Routine sein kann, sollte darum in der Zweckbeschreibung stehen.

- Autorisiere die Datei oder die Routine mit Namen, gegebenenfalls einer Telefonnummer oder einer e-Mail-Adresse. Damit wird sichergestellt, daß andere Programmierer ihre Probleme und Fragen bzgl. dieser Datei an den Autor richten können. Diese umfassende Autorisierung ist besonders bei kommerziell erstellten Produkten für einen engen Kontakt mit den Kunden wichtig. In Tycoon, welches hauptsächlich internen Forschungszwecken dient, werden die meisten Bibliotheksbausteine von Studenten implementiert, die keine eigene Telefonnummer an ihrem Arbeitsplatz besitzen, so daß eine Telefonnummer meist nicht anzugeben ist. Eine e-Mail-Adresse ist ebenfalls nicht für jede Datei des Tycoon Systems als Referenz nötig. Für die hautpsächlich intern am Arbeitsbereich DBIS genutzten Systeme ist die e-Mail-Adresse der einzelnen Implementierer ohnehin bekannt und alle Implementierer sind auf direktem Weg zu erreichen, ohne das e-Mail-Netz zu nutzen. Für externe Anwender des Tycoon Systems reicht es aus, eine e-Mail-Adresse als Referenz für das gesamte System zu kennen. Würde für jede einzelne Datei eine e-Mail-Adresse weitergegeben, ergäben sich durch die Vielzahl der Autoren im System zuviele Adressen, die außerdem durch das Studienende einiger studentischer Mitarbeiter nicht mehr aktuell wären.
- Füge ein Copyright in das Programm ein. In vielen Firmen ist das üblich. Da das Tycoon System nicht kommerziell vertrieben wird, ist ein Copyright nicht von Bedeutung.
- Stelle eine Tabelle mit sämtlichen Abkürzungen zusammen. Zum besseren Verständnis des Lesers müssen alle abgekürzten Variablennamen einmal zu Beginn der Datei erklärt werden. Da für TLdie Länge der Namen nicht begrenzt ist, kann auf solche Tabellen verzichtet werden.

Die vorstehenden Regeln zur Kommentierung werden in Kapitel 5 auf ihre Anwendbarkeit in Tycoon hin überprüft und in eine Sammlung von Richtlinien aufgenommen.

Das Buch-Paradigma für Programmdokumentation Um eine möglichst umfassende Sicht auf ein gesamtes Anwendungssystem zu bieten, ist eine Kategorisierung und Modularisierung nötig. Für die Effektivität einer Dokumentation ist es entscheidend, daß verschiedene Arten des Zugriffs auf eine Datei unterstützt werden. Ein derartiger Zugriff kann das Suchen nach einer Routine sein, die eine bestimmte Aufgabe erfüllen soll. Es kann aber auch das Lesen der Datei sein, um einen Überblick über die darin zur Verfügung gestellten Methoden zu erhalten. Die aufgestellten Richtlinien zur Kommentierung von Dateien, Modulen oder Programmen erfüllen diese Anforderung nicht vollständig.

P. Oman und C. Cook haben zwei Studien über Dokumentation veröffentlicht ([Oman, Cook 90a; Oman, Cook 90b]), in denen sie nach einem Kodierungsstil suchen, der verschiedene Stile des Lesens von Programmen unterstützt. Ein Ziel war die Unterstützung von top-down, bottom-up und fokussiertem Suchen. Ein anderes Ziel war die Teilung des Codes in kleinere

Stücke, die ein Programmierer leichter erfassen und sich daran leichter erinnern kann als an eine lange Liste homogenen Codes. Der Stil sollte Hinweise auf die Organisation des Codes auf oberster, allgemeiner und auf niedrigster, detaillierter Ebene vorsehen.

Als Ergebnis hielten sie fest, daß ihre Ziele erreicht werden, wenn sie den Code als spezielle Art eines Buches ansehen und dementsprechend formatieren. In ihrem Buch-Paradigma werden der Code und seine Dokumentation in mehreren Komponenten ähnlich denen eines Buches organisiert, um dem Programmierer eine Sicht des Programms auf oberster Ebene zu ermöglichen.

Das Vorwort ist eine Gruppe von einführenden Kommentaren zu Beginn einer Datei und hat dieselbe Bedeutung wie das Vorwort eines Buches. Es gibt einen Überblick über das Programm.

Das Inhaltsverzeichnis zeigt die zugehörigen Dateien, Module und Routinen, die jeweils einem Kapitel eines Buches entsprechen. Diese Darstellung kann als Liste, als Verzeichnis oder als graphische Struktur erfolgen.

Die Abschnitte sind die Abschnitte innerhalb einer Routine, wie z. B. Routinendeklaration, Datendeklaration und ausführbare Anweisungen.

Die Querverweise sind Verweistabellen auf den Quelltext, welche auch die Zeilennummern enthalten.

Die von Oman und Cook beschriebenen Techniken auf niedrigster Ebene, welche die Ähnlichkeit zwischen einem Buch und einem Programmausdruck ausnutzen, sind ähnlich den in Abschnitt 2.3 beschriebenen Techniken von Layout und Stil.

Ein Test mit einer Gruppe von erfahrenen Programmierern, welche Wartungsaufgaben in den Sprachen Pascal und C erfüllen sollten, ergab, daß die Wartungsarbeiten des nach dem Buch-Paradigma dokumentierten Codes um 20 % schneller durchgeführt werden konnten als bei traditionell dokumentiertem Code [Oman, Cook 90b]. Die Autoren schlossen daraus, daß das Verständnis eines fremden Programmes um 10 bis 20 % erhöht werden kann, wenn die typografischen Prinzipien des Buchentwurfs bei der Dokumentation beachtet werden.

Das Buch-Paradigma macht deutlich, daß es entscheidend ist, Dokumentation auf zwei Ebenen vorzusehen, welche die Organisation des Programms auf höchster und auf niedrigster Ebene erläutert. Die Anwendung dieses Prinzips auf die Sprache TL wird in Abschnitt 5.5.1 diskutiert.

Programmtexte als Literatur Ebenso wie das Buch-Paradigma verfolgt auch D. Knuth einen Vergleich von Programmen mit der Literatur. Sein Ansatz des *Literarischen Programmierens* [Knuth 92] verbessert die *strukturierte Programmierung*[2], indem nicht nur der Programmtext, sondern auch dessen Dokumentation während des Programmentwurfs beachtet wird. Sein *WEB*-System überbrückt die Spalte zwischen dem Programmtext und der Dokumentation und erzeugt ein literarisches Produkt. Ein *WEB*-Programm beinhaltet Dokumentation und Programmcode gleichermaßen, wodurch die Dokumentation immer auf dem aktuellen Stand gehalten werden kann. Auf den Quelltext des *WEB*-Programms sind zwei Systemroutinen anwendbar. *TANGLE* erzeugt ein maschinenausführbares Programm, während

[2]Strukturierte Programmierung wurde in den siebziger Jahren durch die Einführung der Programmiersprache Pascal zu einem wesentlichen Ziel der Programmierung [Reiser, Wirth 94].

WEAVE ein Dokument erzeugt, welches das Programm deutlich beschreibt und die Programmwartung erleichtert. Das ausführbare Programm und das zugehörige Dokument sind mit dieser Methode automatisch konsistent, da sie beide aus derselben Quelle erzeugt werden.

Innerhalb eines Quelltexts können in *top-down* und *bottom-up* Technik verschiedene Module gebildet werden, die jeweils in die drei Abschnitte

- Kommentar,
- Makrodefinition und
- Programmtext

unterteilt sind. Dies unterstützt den rekursiven Entwurfsprozeß, welcher von kognitiven Psychologen gefordert wird, weil er die natürliche Denkweise des Menschen widerspiegelt (siehe Abschnitt 1.2.1). Das menschliche Verstehen wird gefördert, weil formale und nicht-formale Methoden nebeneinander im Quelltext stehen und individuell vom Programmierer bearbeitet werden können.

Dieses *WEB*-System ist portabel, da die nötigen Werkzeuge, im Protoyp ein Pascal-Compiler und das Textverarbeitungswerkzeug TEX, auf den meisten Rechnern verfügbar sind.

Die Techniken des Buch-Paradigmas und des Literarischen Programmierens zeigen, daß Verbesserungen im Hinblick auf ein besseres Verstehen von Programmtexten nötig sind und daß verschiedene Ansätze existieren. Allerdings sind sie aufwendiger als die bisherige Programmiertechnik, die Kommentare innerhalb des Quellcodes als Entgegenkommen dem Leser gegenüber versteht und sich auf die nötigsten Kommentare beschränkt. Für das Tycoon System sind derartige Ansätze in der Implementierung zu aufwendig. Aus diesem Grund beschränkt sich die vorliegende Arbeit auf die Untersuchung, wie mit den schon vorhandenen Mitteln eine Verbesserung der Dokumentation erreicht werden kann. Dazu sollen die in diesem Kapitel aufgestellten Richtlinien zur Kommentierung dienen.

2.3 Das Layout von Programmen

Auch das *Layout* (der *Drucksatz*) ist für eine effektive Wiederverwendung von Bedeutung. Das Layout bestimmt das Aussehen des Quelltextes und spiegelt seine Strukturen wider. Ein Leser dieses Quelltextes soll durch das Layout im Erkennen der Programmstruktur unterstützt werden. Ein gut strukturiertes Layout erleichtert daher die Wiederverwendung, indem es ein schnelles Erfassen der wesentlichen Strukturen und Abläufe im Quelltext ermöglicht.

Viele Programmierbücher behandeln Layout und Stil einer Sprache, aber allgemeine Richlinien über den Programmierstil sind selten. Noch seltener sind Erwägungen über das richtige Layout. Ein klassisches Buch ist 1974 von Kernighan und Plauger [Kernighan, Plauger 74] erschienen. Für die zur damaligen Zeit weniger strukturierten Programmiersprachen konnte ein gut lesbares Layout dem Programmierer viel Zeit und Mühe bei der Fehlersuche sparen.

Beim Programmieren sind die Zeiten des trickreichen Programmierens längst vorbei, da die Programmiersprachen immer komfortabler werden, indem sie höhere Abstraktionskonzepte anbieten, und nicht mehr Anweisungen auf Assemblerebene eingesetzt werden müssen, um effiziente Programme zu erhalten. Auch die Angebote an Konstrukten und Namensgebungen vermehren die Möglichkeiten, einen strukturierten und lesbaren Quelltext zu erstellen. Trotzdem müssen die Programmierer immer wieder darauf hingewiesen werden, daß ein guter Code nicht nur aus sinnvollen Routinen- und Variablennamen sowie einer durchdachten Struktur

besteht, sondern diese Struktur ebenfalls durch die Darstellung unterstützt werden muß. Nur auf diese Art können lesbare und funktionierende Programme erstellt werden, die nicht nur vom Autor, sondern gleichfalls von anderen Programmierern wartbar sind.

McConnell [McConnell 93] stimmt mit der Ansicht von Knuth [Knuth 92] überein, daß die Details des Layouts gleich während des Kodierungsprozesses Berücksichtigung finden müssen, da sie nachträglich nur sehr schwer geändert werden können. Die Autoren von [Biggerstaff, Perlis 89] legen dagegen überhaupt keinen Wert auf Layout und Kommentare im Quelltext, da sie der Ansicht sind, daß diese Dokumentation zu fehleranfällig ist, weil sie während des gesamten Prozesses immer wieder mit überarbeitet werden muß, um auf dem aktuellen Stand zu sein. Für das Tycoon Systems wird daher ein Ansatz verfolgt, der beide Ansichten berücksichtigt. Tycoon-Programme sollen so wenig wie möglich und so viel wie nötig kommentiert werden. Das Layout wird in [Matthes, Müßig 93] hinreichend festgelegt. Es soll in diesem Abschnitt verdeutlicht werden, welchen Zielen ein Layout dienen soll.

Für Routinen, Module und Programme des Tycoon Systems ist das Aussehen des Quelltextes wichtig, welches vom Layout bestimmt wird. Voraussetzung für die folgenden Ausführungen ist, daß die Darstellung von Texten mit einem zeilenorientierten Textverarbeitungswerkzeug erfolgt. Die Einteilung und Kommentierung von Absätzen innerhalb eines Moduls ist für den Neuentwurf der Tycoon-Schnittstellen und -Module ebenso von Bedeutung, da die Klassifizierung der Routinen innerhalb eines Moduls durch das Layout hervorgehoben werden soll.

Das Layout des Quelltextes ist ein ästhetischer Aspekt des Programmierens[3]. Es betrifft nichts, was außerhalb des Programms sichtbar wäre. Layout-Techniken dienen nur zum schnelleren Verständnis, zur besseren Übersicht und Revision des Programms auch noch Monate nach dessen Erstellung. Diese Techniken sollen es anderen erleichtern, das Programm zu lesen, zu verstehen und eventuell zu modifizieren.

Das Ziel eines guten Layouts ist

- eine präzise Beschreibung der logischen Struktur des Programms. Dieses grundlegende Theorem der Formatierung wird durch Einrückung und Leerzeilen erreicht.

- eine konsistente Repräsentation der logischen Struktur des Codes. Konsistent anwendbare Layout-Richtlinien sollten für möglichst alle Fälle gelten und möglichst wenig Ausnahmen enthalten, so daß ein Programmierer sich die Regeln leichter merken kann.

- eine Verbesserung der Lesbarkeit. Auch Regeln, welche die vorherigen Forderungen erfüllen, können die Lesbarkeit mindern. Als ein Beispiel gilt die Einrückung von Code bei if-Abfragen. Wird diese Einrückung unter die jeweiligen Schlüsselworte vorgenommen, dann kann bei Schachtelung von Anweisungen und entsprechend langen, bedingten Ausdrücken die Zuordnung des folgenden else-Konstrukts für den Leser schwierig auszumachen sein.

  Beispiel:

  if (soldCount > 10 and prevMonthSales > 10) then
  if (soldCount > 100 and prevMonthSales > 10) then

[3] McConnell [McConnell 93] sagt dazu: "Die visuelle und intellektuelle Freude an gut formatiertem Code können nur wenige Nicht-Programmierer verstehen. Programmierer, die stolz auf ihr Werk sind, erreichen große, künstlerische Befriedigung durch Aufarbeiten der visuellen Struktur ihres Codes."

```
if (soldCount > 1000)  then  begin
                             markDown := 0.10
                             profit := 0.05
                             end
                       else markDown := 0.05
                       endif
                                          else markDown := 0.025
                                          endif
                                     else markDown := 0.0
                                     endif
```

- eine Resistenz gegenüber Modifikationen. Die Änderung einer Codezeile sollte nicht die Änderung weiterer Zeilen zur Wiederherstellung des richtigen Layouts nach sich ziehen.

2.4 Benennung und Namenswahl

Ein Name in der Programmierung sollte sorgfältig ausgesucht werden. In den folgenden Abschnitten werden daher Richtlinien für die Namensgebung im Zusammenhang mit verschiedenen Sprachkonstrukten diskutiert.

2.4.1 Namen und Deklaration von Routinen

Ein Routinenname sollte alles beschreiben, was von der Routine ausgeführt wird. Einige Richtlinien zur Generierung effektiver Routinennamen sind:

- Ein Name für eine Prozedur wird mit einem Verb und dessen Objekt bezeichnet. Sie führt eine Operation auf einem Objekt aus, die vom Namen wiedergegeben werden sollte. *printReport()*, *repaginateDocument()* sind Beispiele dafür. In objektorientierten Sprachen sollte der Name des Objekts nicht im Prozedurnamen verwendet werden, weil das Objekt selbst im Aufruf eingebettet ist. Ein typischer Aufruf wäre z. B. *Report.print()*. Wenn die Prozedur auch in diesem Fall *printReport()* hieße, dann würde im Aufruf Redundanz entstehen: *Report.printReport()*

- Ein Name für eine Funktion beschreibt den Rückgabewert. *cos()* und *currentPrinterColor()* beschreiben präzise, welchen Wert die Funktion zurückgibt.

- Attributabfragen werden mit dem Präfix *is* vor dem beschreibenden Adjektiv versehen. Damit wird einheitlich herausgestellt, daß es sich um eine boolsche Frage handelt, die mit *Ja* oder *Nein* beantwortet werden kann. *isEmpty(stack)* kann nur *1* für *Ja* oder *0* für *Nein* zurückgeben. Eine Angabe der Größe des Stacks würde der englischen Semantik widersprechen.

- Bedeutungslose oder unklare Verben werden vermieden. Einige dehnbare Begriffe als Namen sagen nichts über die Bedeutung der Routine aus. Beispielsweise sagen *performServices()* oder *processInput()* nichts darüber aus, was in der Routine passiert, außer daß sie etwas mit Diensten bzw. Eingabe zu tun haben. In vielen dieser Fälle ist die

Routine selbst genau spezifiziert. Dann sollte eine genauere Bezeichnung als Routinenname gewählt werden, um diese Spezifikation wiederzugeben. In einigen Fällen ist aber auch die Routine ungenau spezifiziert, was durch den in diesem Fall passenden, aber unpräzisen Namen ausgedrückt wird. In diesen Fällen sollte die Routine mit allen abhängigen Routinen neu strukturiert werden, um einen genau zu beschreibenden Zweck zu implementieren.

- Namen beschreiben ausnahmslos alle Ausgaben und Seiteneffekte einer Routine, die während ihrer Ausführung auftreten. Dies kann zu adäquaten, aber viel zu langen Namen führen, wie z. B. *computeReportTotalsAndSetPrintingReadyVar()*. Abhilfe kann nur eine verfeinerte Version oder eine Aufspaltung in mehrere Routinen schaffen, die alle Seiteneffekte ausschaltet, indem die Aktionen direkt durchgeführt werden.

- Routinennamen werden so lang gewählt, wie es für eine treffende, eindeutige Aussage über die Routine nötig ist. Laut [McConnell 93] haben Forschungen ergeben, daß die optimale, durchschnittliche Länge von Variablennamen 9 bis 15 Zeichen beträgt. Da Routinen komplizierter sind, weil sie einen gesamten Abschnitt vom Code beinhalten, sind ihre Namen länger. [Rees 82] hat daher die Länge von 20 bis 35 Zeichen als optimal bezeichnet. Die tatsächlich benötigte Länge für aussagekräftige Namen liegt wahrscheinlich dazwischen.

Konventionen sind für generelle Operationen unentbehrlich. In einigen Systemen müssen verschiedene Arten von Operationen unterschieden werden. Dies kann am einfachsten durch eine Namenskonvention erreicht werden. Beispielsweise wurde in der Entwicklung des *OS/2 Presentation Managers* jede Routine mit destruktiver Eingabe durch das Präfix *get* gekennzeichnet, während jede Routine mit nicht-destruktiver Eingabe mit dem Präfix *Query* versehen wurde. Damit ist festgelegt, daß *GetInputChar()* das aktuelle Eingabezeichen liest und den Puffer löscht, während *QueryInputChar()* gleichfalls das aktuelle Zeichen liest, aber den Eingabepuffer unverändert läßt.

2.4.2 Routinenparameter

Für das Verständnis der Routinendeklarationen sind auch die Parameterdeklarationen entscheidend. Die Wahl ihrer Namen sagt dem Leser einiges über den Zweck der Routine und die Länge der Parameterliste beeinflußt die Lesbarkeit. Um die Schnittstellenprobleme zwischen Routinen abzubauen, sind eine Reihe von Regeln zu beachten:

- Es ist sicherzustellen, daß die aktuellen Parameter zu den formalen Parametern passen. Bei strikt typisierten Sprachen wie Tycoon tritt das Problem nicht auf, daß die beim Aufruf benutzten Parametertypen nicht zu den in der Schnittstelle definierten Typen der Parameter passen.

- Parameterlisten werden in der Reihenfolge Eingabe-Ändern-Ausgabe angegeben. Die Parameter, die nur eingegeben werden, stehen vor den Parametern, die zusätzlich geändert und ausgegeben werden. Am Ende stehen die Parameter, die von der Routine nur ausgegeben werden. Diese Reihenfolge spiegelt die Reihenfolge der Operationen innerhalb der Routine wider.

- Ähnliche Parameter verschiedener Routinen stehen in konsistenter Reihenfolge.

- Alle Parameter einer Routine werden auch benutzt. Unbenutzte Parameter müssen aus der Routinenschnittstelle entfernt werden, da sie häufig zu Mißverständnissen bei Programmierern, welche diese Routine verwenden, und Fehlern in der Weiterentwicklung der Routine führen.

- Zustands- und Fehlervariable werden an das Ende der Parameterliste gestellt. Sie sind nur eine Randerscheinung des Hauptziels der Routine und werden nur ausgegeben. Damit entspricht dies auch der Regel über die Reihenfolge.

- Parameter dürfen nicht als Arbeitsvariable zur Zwischenspeicherung innerhalb der Routine benutzt werden. Zu diesem Zweck werden lokale Variable deklariert.

- Annahmen über die Schnittstelle werden dokumentiert. Dies kann in einem Kommentar geschehen, ist aber als *Zusicherung* zusätzlich vom Compiler prüfbar und führt zu einem korrekteren Programm. *Zusicherungen* werden in Abschnitt 3.3.4 näher erläutert.

  Folgende Schnittstellen-Eigenschaften der Parameter sollten dokumentiert werden:

  - Modifizierbarkeit der Parameter (Eingabe-, Änderungs-, Ausgabeparameter)
  - Einheit der numerischen Parameter (Inch, Meter ...)
  - Bedeutung des Statuscodes und Fehlercodes
  - Bereich der erwarteten Werte
  - Werte, die niemals auftreten sollten

- Begrenzung der Anzahl der Parameter auf ungefähr sieben. Sieben ist eine magische Zahl in der menschlichen Wahrnehmung. Die meisten Menschen können nicht mehr als sieben Elemente auf einmal wahrnehmen und behandeln sihe auch Abschnitt 1.2.1).

- Wenn es wichtig ist, die Eingabe-, Änderungs- und Ausgabeparameter zu unterscheiden, sollte zu ihrer Identifizierung eine Namenskonvention, z. B. mit Hilfe von Präfixen, aufgestellt werden.

- Nur die Felder eines strukturierten Datentyps, die wirklich in der Routine benutzt werden, sollten übergeben werden. Eine derart präzisierte Schnittstelle vereinfacht die Wiederverwendung der Routine an anderer Stelle. Eine Ausnahme für diese Regel bilden die abstrakten Datentypen (ADT). Wenn ADTs benutzt werden, soll die innere Struktur verborgen bleiben. Zur Bewahrung der Abstraktionsebene ist es in diesem Fall nötig, die gesamte Struktur als Parameter zu übergeben.

- Der Mechanismus zur Parameterübergabe sollte nicht umgangen werden, um effizienteren Code zu erhalten. Das gesamte Programm ist dann nicht mehr portierbar.

2.4.3 Variablennamen

Ein Name sollte lesbar, leicht zu merken und treffend sein. Er beschreibt vollständig das Objekt, welches von der Variablen repräsentiert wird. Zur Namensfindung wird die Repräsentation der Variablen in Worten beschrieben und als Name gewählt. Wird dabei der Name zu lang, ist er nicht leicht zu merken. Eine Abkürzung wiederum kann den Namen leicht unlesbar machen, ein vollständig ausgeschriebener Name ist dagegen leicht zu entziffern, denn er muß

nicht dekodiert werden. In den folgenden Abschnitten werden darum Richtlinien angegeben, die der Findung eines optimalen Namens dienen sollen. Diese Richtlinien sind nicht immer gleichzeitig anwendbar. Sie dienen nur als Leitfaden zu einer sinnvollen Namensgebung. Jeder Programmierer muß sich für den jeweiligen Zweck seines Objekts die passenden Richtlinien heraussuchen.

Jede Variable sollte nur zu *einem* Zweck benutzt werden. Dies wird häufig für temporäre Variable mißachtet, die über den gesamten Quellcode verteilt für verschiedene Aktionen wiederverwendet werden.

Ein guter mnemonischer Name beschreibt das Problem und nicht den Lösungsweg. Es wird das *Was* und nicht das *Wie* ausgedrückt. Daher sollte ein Name auf keinen Aspekt des Programmierens und Berechnens eingehen. Er darf nur auf das Problem selbst verweisen. Beispielsweise bezeichnet *BitFlag* ein Bit, welches den Status eines Druckers angibt. Aus diesem Namen geht aber nur hervor, daß ein Bit bei der Implementation des Status benutzt wurde, und nicht, was diese Zustandsvariable repräsentieren soll. Daher ist es besser, das Bitfeld für den Status des Druckers *PrinterReady* zu nennen.

Die Länge des Namens ist für die Verständlichkeit des Quelltextes von Bedeutung. Zu kurze Namen übermitteln nicht genug Bedeutung und lassen auf keine Beziehungen untereinander schließen. Zu lange Namen sind umständlich zu tippen und verwischen die optische Struktur des Programms. Programme mit durchschnittlicher Namenslänge zwischen 8 und 20 Zeichen sind leicht auf Fehler zu untersuchen [McConnell 93]. Dies bedeutet nicht, daß alle Namen auf diese Länge gebracht werden müssen. Aber werden im Code sehr viele, kürzere Namen entdeckt, dann sollte geprüft werden, ob alle Namen klar und eindeutig gewählt sind.

Auch der Geltungsbereich der Variablen hat Auswirkungen auf die Namenslänge. Je kürzer der Name einer Variablen ist, desto eher deutet das darauf hin, daß diese Variable nur ein Zwischenspeicher mit begrenztem Geltungsbereich ist. Ein Programmierer setzt voraus, daß der Wert einer solchen Variablen nicht außerhalb dieser paar Zeilen Code benutzt wird. Ein häufiges Beispiel ist der Zähler einer Schleife, der mit i benannt wird. In einer Studie fand W. J. Hansen [Shneiderman 80] heraus, daß längere Namen für seltener benutzte Variable oder globale Variable benutzt werden und kürzere Namen besser geeignet sind für lokale Variable und Schleifenvariable. Da kurze Namen der Grund für viele Probleme sind, werden sie von vorsichtigen Programmierern als eine Art von defensiver, vorausschauender Programmierung vermieden.

2.4.4 Berechnete Werte

Manche Programme enthalten Variable für Gesamtsumme, Durchschnitt, Maximum etc. in verschiedenen Kontexten. Ein Name für derartige Berechnungen sollte mit entsprechenden Bezeichnern, wie z. B. *Ttl* oder *Sum* für Summen, *Avg* für Durchschnitt oder *Max* für einen maximalen Wert, modifiziert werden, der dann am Ende des Namens stehen. Dies hat zahlreiche Vorteile:

1. Der entscheidende Teil des Namens, der die Bedeutung trägt, steht am Beginn, wo er hervorsticht und als erstes gelesen wird.

2. Bei Einhaltung dieser Konvention gibt es keine Verwechslung, wenn beide möglichen Namen, *TtlRevenue* und *RevenueTtl*, in demselben Programm benutzt werden. Semantisch sind diese Namen äquivalent und die Konvention wird verhindern, daß sie verschieden benutzt werden.

Wortpaar	Bedeutung
add/remove	Hinzufügen bzw. Entfernen von Elementen in eine Menge
insert/delete	Einfügen bzw. Löschen von Elementen in eine Menge
increment/decrement	Erhöhen bzw. Verringern eines Werts um einen bestimmten Wert
lock/unlock	Sperren bzw. Freigeben eines (Speicher-)Abschnitts
old/new	alter bzw. neu erstellter Wert einer Variablen
source/destination	Quelle bzw. Ziel einer Änderungsoperation
begin/end	Anfang bzw. Ende eines Abschnitts
first/last	erstes bzw. letztes Element
put/get	Bringen bzw. Holen eines Elements
min/max	minimaler bzw. maximaler Wert
open/close	Öffnen bzw. Schließen eines Speichers
source/target	Quelle bzw. Ziel einer (Lese-/Schreib-)Operation
create/destroy	Erstellen bzw. Zerstören einer Datenstruktur
get/release	Holen bzw. Freigeben von Speicherplatz
up/down	Rollen nach oben bzw. unten in einer Liste
next/previous	nächstes bzw. vorheriges Element in einer Liste
show/hide	Anzeigen bzw. Verstecken einer Variablen
start/stop	Starten bzw. Beenden einer Aktivität

Tabelle 2.1: Gegensatzpaare

3. Eine Menge von Namen mit denselben Postfixen, wie *RevenueTtl* und *ExpenseTtl*, *RevenueAvg* und *ExpenseAvg*, hat eine ansprechende Symmetrie, die auf eine gewisse Ordnung und Beziehung der von den Variablen repräsentierten Objekte untereinander schließen läßt.

4. Die Konsistenz fördert die Lesbarkeit und Verständlichkeit. Außerdem erleichtert es die Programmpflege.

Eine Ausnahme zur Regel, daß berechnete Werte am Ende des Namens bezeichnet werden, ist die Position des Bezeichners *Num*. Zu Beginn eines Variablennamens stehend bezieht er sich auf eine Gesamtsumme, am Ende angefügt bezeichnet er einen Index. Um diese Mehrdeutigkeit zu umgehen, benutzt man als Ersatz für *Num* besser andere Bezeichner, z. B. *Count* und *Index*.

In einigen Sprachen ist auch die Benutzung von Zeigervariablen möglich, die durch das Postfix *Ptr* gekennzeichnet werden sollten. Da TLkeine Zeigerstruktur enthält, ist diese Kennzeichnung in den TL-Bibliotheken nicht nötig.

2.4.5 Gegensatzpaare

Gegensätzliche Ausdrücke, wie z. B. *insert* und *delete*, müssen ihrer Definition entsprechend benutzt werden. Die Festlegung einer Namenskonvention für Gegensatzpaare nützt der Konsistenz, welche die Lesbarkeit und Verständlichkeit erhöht. Das Wort *end* (für *Ende der Elementliste*) ist nicht so gebräuchlich als Gegensatz zum Wort *first* (für *erstes Element*) wie

last (für *letztes Element*) und führt zu Verwirrung. Verständnisschwierigkeiten treten beim Leser des Codes auf, weil er im Allgemeinen als Gegensatz zu *end* das Wort *begin* erwartet.

Für jede Anwendung sollte vor Beginn der Implementation festgelegt werden, welche Worte darin als Gegensätze benutzt werden. Die genaue Bedeutung kann allerdings noch gewählt werden, so z. B. ob das erste und letzte Wort eines Textes in der Darstellung auf dem Bildschirm mit *first/last* oder mit *begin/end* angewählt werden soll. Das Wort *get* wird häufig als Gegensatz zu *put* benutzt, aber auch in Verbindung mit *release*. Wenn eine eindeutige Zuordnung von Wort und Bedeutung benötigt wird, dann muß in einem dieser Paare das *get* durch ein anderes Wort ersetzt werden. Somit ist also nicht auszuschließen, daß sich entgegen der allgemein üblichen Paarung neue Gegensatzpaare in der individuell erstellten Konvention bilden. Im Tycoon System soll insbesondere für die Benennung von Operationen eine Liste der Gegensatzpaare erstellt werden.

2.4.6 Regeln für spezifische Datentypen

Überlegungen für die Namensgebung einiger besonderer Konstrukte, wie Schleifen oder Aufzählungen, und in den meisten Programmen vorhandene Variable zur Kontrolle des Programmflusses, wie Zustandsvariable, temporäre Variable oder Konstante, werden im Folgenden angestellt.

Schleifenindices werden sehr oft benutzt, da eine Schleife ein allgemeines Programmkonstrukt ist. Die Variablen zur Schleifenkontrolle haben wegen ihres häufigen Auftretens üblicherweise einfache, kurze Namen wie *i, j* oder *k*. Wird diese Variable außerhalb der Schleife benutzt, sollte ihr Name die Bedeutung wiedergeben, z. B. *RecordCount*. Das gilt ebenfalls, wenn die Schleife im Quellcode länger als einige Zeilen ist, da in diesem Fall schnell unklar werden kann, wozu die Variable eingeführt wurde und was sie bezeichnet. Dies tritt insbesondere dann auf, wenn der Leser den Code nicht sequentiell, sondern auf ein bestimmtes Problem hin durchliest. Erfahrene Programmierer vermeiden kurze Namen aus einem weiteren Grund. Sie wollen beim Ändern, Kopieren und Erweitern keine Probleme zu verursachen. Außerdem können auf diese Weise keine Verwechslungen der Indices auftreten. Beim Zugriff auf Felder ist der längere Name weiterhin vorteilhaft, um die Bedeutung herauszustellen. Als wichtigste Regel sollten einzelne Buchstaben, wenn überhaupt, **nur** für Schleifenvariable benutzt werden. Diese Konvention hat sich seit den Anfängen der Programmierung etabliert und bewährt.

Zustandsvariable geben den Zustand des Programms wieder. Das Wort *flag* sollte nicht benutzt werden, da es nichts darüber aussagt, was diese Zustandsvariable beschreibt. Einer Zustandsvariablen sollten Werte zugewiesen werden, die dann mit Aufzählungstypen, Namenskonstanten oder globalen Variablen verglichen werden. Ausdrücke wie

$$StatusFlag \; = \; 0x80$$

sind erklärungsbedürftig, da weder die Bedeutung von *StatusFlag* erkennbar ist, noch für jeden Leser sofort klar ist, welchen Sinn der Wert $0x80$ hat. Dagegen implizieren die Zuweisungen

$$CharacterType = CONTROL_CHARACTER$$
$$DataReady = TRUE$$

bei dem Leser sofort eine Idee, worum es sich handelt.

Temporäre Variable beinhalten Zwischenresultate von Berechnungen. Die gebräuchliche Bezeichnung mit *Temp, X* oder anderen, ungenau beschreibenden Namen ist genauso nichtssagend wie die Schleifenvariable *i*. Derartige Variable werden nicht so sorgfältig behandelt wie andere, aussagekräftige Variable, wodurch die Fehlerquote steigt.

Im Prinzip existieren alle Variablen des Programms temporär, weil sie in den meisten Programmiersprachen mit der Beendigung des Programms oder schon vorher vom Programmierer gelöscht werden. Die Auszeichnung einiger Variablen als temporär zeigt, daß nicht festgelegt ist, welchem Zweck sie dienen. Auch ein Zwischenergebnis, welches nur gebildet wird, weil es in folgenden Berechnungen mehrmals wiederverwendet wird, sollte einen Namen erhalten, der seinen Inhalt genau beschreibt.

Ein Beispiel in C:

```
/* Compute roots of a quadratic equation.
This assumes that (b^ 2 - 4 * a * c) is positive. */
Temp     = sqrt(b^ 2 - 4 * a * c)
root[0]  = (-b + Temp) / (2 * a);
root[1]  = (-b - Temp) / (2 * a);
```

Ein besserer Ansatz wird allerdings wie folgt geschrieben, da hier die temporäre Variable nach ihrem Zweck benannt wird:

```
/* Compute roots of a quadratic equation.
This assumes that (b^ 2 - 4 * a * c) is positive. */
Discriminant = sqrt(b^ 2 - 4 * a * c)
root[0]      = (-b + Discriminant) / (2 * a);
root[1]      = (-b - Discriminant) / (2 * a);
```

Boolsche Variable haben einige typische Namen:

Done zeigt an, daß eine Routine, eine Schleife oder ein anderer Programmabschnitt fertig abgearbeitet ist. *Done* wird auf *False* gesetzt, bevor etwas getan wird, und bei Vollendung der Aufgabe auf *True* gesetzt.

Error bedeutet das Auftreten eines Fehlers. Sie wird zu Beginn auf *False* gesetzt und bekommt den Wert *True*, wenn ein Fehler aufgetreten ist.

Found wird benutzt, um anzugeben, ob ein Wert gefunden wurde oder nicht. Sie wird auf *False* gesetzt, wenn der Wert nicht gefunden wurde und auf *True*, wenn der Wert das erste Mal aufgetreten ist. *Found* wird benutzt, wenn ein Feld nach einem Wert, eine Datei nach einer Identifikationsnummer eines Angestellten, eine Liste nach einer bestimmten Zahlungsgröße usw. durchsucht wird.

Success gibt an, ob eine Operation erfolgreich ausgeführt wurde. Sie erhält den Wert *False*, wenn die Operation mißlang und *True*, wenn die Operation erfolgreich war. Wenn es möglich ist, sollte *Success* durch einen spezifischeren Namen ersetzt werden, der genau angibt, was "erfolgreich" bedeutet, z. B. *ProcessingComplete*. In vielen Fällen kann auch stattdessen einer der genannten, typischen Namen benutzt werden, z. B. *Found*, wenn der Erfolg eines Programmabschnitts darin besteht, einen Wert erfolgreich gefunden zu haben.

Boolsche Variable sollten einen Namen erhalten, der *True* oder *False* impliziert. *Done* oder *Success* sind daher gute Beispiele für boolsche Variablennamen, da sie nur zwei Zustände zulassen. Dagegen sind *Status*, *Error* oder *SourceFile* negative Beispiele, da sie nicht offensichtlich nur *True* oder *False* sein können. Heißt es, wenn *Status* den Wert *True* hat, daß etwas einen Status hat oder daß der Status von irgendetwas in Ordnung ist? Bessere Namen sind in diesem Fall Zusammensetzungen wie *StatusOK* oder *SourceFileAvailable*.

Als eine weitere Konvention für boolsche Variable wird empfohlen, alle booleschen Namen mit *Is* zu beginnen, so daß der Name eine Frage darstellt, deren Antwort den Zustand bzw. Wert der Variablen wiedergibt. Ein Vorteil dieser Methode ist, daß es eine Zusammensetzung mit ungenau beschreibenden Namen verhindert. Beispielsweise ergibt *IsStatus* als boolsche Variable keinen Sinn.

Als weitere Regel sollten negative Namen vermieden werden, da ihre Negierungen innerhalb des Programmtexts schwierig zu lesen sind. Es entstehen komplexe Ausdrücke mit zweifacher Verneinung, wie in: *If not NotFound then ...*

Aufzählungstypen bestehen aus Konstanten, die durch ein gemeinsames Präfix oder Suffix als zu diesem Aufzählungstypen gehörend gekennzeichnet werden. Ein typisches Beispiel ist:

type color is (colorRed, colorGreen, colorBlue);

Konstante werden mit den abstrakten Entitäten benannt, die sie repräsentieren. Es ist änderungsfeindlich, Konstante mit dem Wert zu benennen, den sie darstellen, weil sich dieser Wert ändern kann. *maxDonuts* ist daher ein änderungsfreundlicherer Name als *bakersDozen*, da es lächerlich wirkt, wenn der Wert von *bakersDozen* auf 13 geändert werden muß.

2.4.7 Namenskonventionen

Über die Benennung von Variablen und auch Routinen gibt es mehr oder weniger formale Namenskonventionen. Die Regel *Benutze aussagekräftige Namen* ist eine wenig formale Konvention. Eine sehr viel formalere Konvention ist die *Ungarische Konvention*, die am Ende dieses Abschnitts kurz vorgestellt wird. Der Grad der benötigten Formalität für ein Projekt ist abhängig von der Anzahl der Leute, die an dem Projekt arbeiten, der Größe und der erwarteten Lebenszeit der erstellten Programme.

Einige Konventionen sind zu restriktiv und werden daher ineffektiv, weil sie die Kreativität des Programmierers einschränken, worunter die Qualität der Programme leidet.

Warum benötigt man Konventionen?

- Sie erleichtern dem Programmierer die Arbeit, indem eine globale Entscheidung in der Konvention dem Programmierer viele lokale Entscheidungen abnimmt und er sich auf die wichtigeren Charakteristika des Codes konzentrieren kann.

- Konventionen reduzieren die Menge der möglichen Namen. Ohne Konvention kann man ein und dasselbe Objekt mit verschiedenen Namen benennen, z. B. *PointTtl* und *TtlPoints*. Während der Erstellung des Codes mag es noch keine Konfusion geben, aber ein anderer Programmierer, der den Code später liest, wird davon verwirrt.

- Konventionen sind auch ein Hilfsmittel, um das Wissen über Projekt- und Zeitgrenzen hinweg zu transferieren, denn die Ähnlichkeit in den Namen fördert die Verständlichkeit. Außerdem kann der Code eines neuen Projektes schneller erfaßt werden, wenn die Menge des Codes konsistent ist und der Leser keine Rücksicht auf persönliche Programmierstile nehmen muß.

- Konventionen können weiterhin Namenskonstanten und Aufzählungstypen emulieren, um programmiersprachliche Schwächen auszugleichen. Sie unterscheiden zwischen lokalen, modulinternen und globalen Daten und können Typinformationen beinhalten, die vom Compiler nicht unterstützt werden.

- Sie heben als weiteren Vorteil Beziehungen zwischen Objekten hervor. Strukturierte Daten werden vom Compiler automatisch verwaltet. In TL beispielsweise werden zusammengesetzte Namen erlaubt, um die Felder einer Struktur bzw. die Operationen aus dem Sichtbarkeitsbereich eines anderen Moduls aufzurufen: *company.employee.name* bzw. *Person.getName*. Für eine Sprache, die keine strukturierten Datentypen zuläßt, kann dies durch eine Namenskonvention, z. B. einheitliche Präfixe als Pseudostruktur, ergänzt werden.

Jede Konvention ist daher besser als keine. Sie strukturiert den Code und verringert die Menge der Probleme, um die sich ein Programmierer kümmern muß [McConnell 93], um dadurch die Anzahl möglicher Fehler beim Programmentwurf zu verringern. Außerdem ermöglicht das Einhalten von Konventionen insbesondere in der Formatierung automatische Werkzeuge [Rovner et al. 85].

Unter welchen Umständen sollte eine Namenskonvention eingeführt werden?

- Wenn mehrere Programmierer an einem Projekt arbeiten.

- Wenn ein Programm für Änderungen und Pflege an einen anderen Programmierer weitergegeben werden soll.

- Wenn das Programm von anderen Programmierern eingesehen werden soll.

- Wenn das Programm so groß ist, daß es nicht im Ganzen überschaut werden kann, sondern in Teilstücken überdacht werden muß.

- Wenn eine Menge unüblicher Ausdrücke für ein Projekt gelten und man Standardausdrücke oder Abkürzungen im Code benutzen will.

Die meisten Projekte nutzen nicht formale Namenskonventionen. Sie können sprachabhängig oder sprachunabhängig sein. Als nützlich für eine sprachunabhängige Konvention mögen die folgenden Richtlinien gelten:

- Identifiziere globale Variable mit Präfix *g_*.
- Identifiziere Modulvariable mit Präfix *m_* oder *static* außerhalb des Geltungsbereichs jeder Routine.
- Identifiziere Typdefinitionen mit Suffix *_t* oder *_T*.
- Identifiziere Namenskonstanten mit Suffix *_c* oder *_C*.
- Identifiziere Aufzählungstypen mit Suffix *_e* oder *_E*.
- Identifiziere Eingabeparameter in Sprachen, die sie nicht explizit durch Compilerkonstrukte unterstützen (wie z. B. *Var* als Kennzeichnung für einen änderbaren Parameterwert in Pascal), um ein zufälliges Überschreiben des Wertes zu verhindern. Dazu kann ein Präfix *IP* (Abk. für *input*) vor den Namen gesetzt werden. Der Programmierer sieht dann, daß eine solche Variable nicht auf der linken Seite einer Zuweisung stehen darf.
- Formatiere Namen, um die Lesbarkeit zu erhöhen. Dazu können Großbuchstaben oder Trennzeichen verwendet werden. Diese Techniken sollten allerdings nicht gemischt werden, um die Lesbarkeit durch Konsistenz zu erhalten. Die Lesbarkeit von *GYMNASTICSPOINTTOTAL* erhöht sich stark, wenn Groß- und Kleinschreibung (*GymnasticsPointTotal*) oder Trennzeichen (*gymnastics_point_total*) benutzt werden.

Diese Regeln sind für TLnicht nützlich.

Sprachabhängige Konventionen sind in die Sprachen eingebaute Richtlinien. Als Beispiel für eine sprachabhängige Konvention gelten die Regeln in Fortran, daß Integervariablennamen mit den Buchstaben *I* bis *N* beginnen, *I, J* und *K* nur für Schleifenindices benutzt werden und *X, Y* und *Z* Gleitkommazahlen repräsentieren. Diese Konvention wird vom Compiler überprüft.

Diese gesamten Namenskonventionen erscheinen auf den ersten Blick sehr kompliziert und umfangreich. Sie können aber vereinfacht werden, wenn sie an die jeweiligen Bedürfnisse des Projekts und der verwendeten Programmiersprache angepaßt werden. Dabei muß berücksichtigt werden, daß allgemein für Variablennamen drei Arten von Information in den Namen gesteckt werden:

1. Der Inhalt der Variable (z. B. was durch sie repräsentiert wird)
2. Der Datentyp (z. B. *int* oder ein benutzerdefinierter Typ)
3. Der Ort in der Struktur des Programms (z. B. der Name des Moduls oder ein Präfix zur Kennzeichnung der Globalität)

Ungarische Namenskonvention Die Ungarische Namenskonvention ist eine Menge von detaillierten Richtlinien zur Benennung von Variablen und Routinen. Sie soll hier als Beispiel für eine formale Konvention, die den Inhalt, den Datentyp und die Struktur des Datentyps im Namen wiedergibt, kurz erläutert werden. Verbreitet ist diese Konvention im Gebrauch mit der Sprache C, insbesondere in der Microsoft Windows-Programmierung. Sie wird *ungarisch* genannt, da die entstehenden Namen fremdsprachlich aussehen und weil der Erfinder, Charles Simonyi, aus Ungarn stammt.

Ungarische Namen bestehen aus drei Teilen:

Der Basistyp ist der Datentyp der Variablen. Meist ist er ein abstrakter Datentyp, der sich nicht auf die angebotenen Basisdatentypen der Sprache (*int*, *real*, *char* u.a. in C) bezieht. Bezeichner sind mnemonische Abkürzungen, die für ein Programm erzeugt und standardisiert werden. Einige Beispiele:

wn für Window, *scr* für Screen region, *fon* für Font usw.

Datentypen werden entsprechend benannt.

Ein oder mehrere Präfixe beschreiben die Benutzung der Variablen. Sie sind standardisiert. Einige Beispiele:

a für Array, *c* für Count, *e* für Element eines Arrays, *i* für Index eines Arrays, *p* für Pointer usw.

Diese Präfixe werden klein geschrieben und vor den Basistyp im Variablennamen gehängt. Sie können mit Basistypen und untereinander beliebig kombiniert werden. Beispielsweise ist *cfon* die Anzahl der Schriften.

Der Bezeichner ist der letzte Bestandteil eines ungarischen Namens. Wenn keine ungarische Konvention benutzt wird, würde dieser Teil allein den gesamten Variablennamen ausmachen. Er gibt den Inhalt der Variablen wieder und wird nach den Regeln für sinnvolle Variablennamen gebildet. Außer den vom Programmierer zu bildenden Namen gibt es Standardbezeichner, deren Bedeutung festgelegt ist:

Min für den Minimalwert, *First* für den ersten Wert, *Last* für den letzten Wert, *Lim* für den Grenzwert, *Max* für den Maximalwert einer Folge.

Bezeichner können und sollen mit Präfixen und Basistypen zusammengesetzt werden.

Beispiele für Ungarische Namen sind:

achDelete ist ein Array von Zeichen zum Löschen.
ghscrMessages ist ein globaler Handle einer Bildschirmregion für Nachrichten.

Neben den generellen Vorteilen einer Konvention haben Ungarische Namen folgende Vorteile:

- Durch viele Standardnamen muß sich der Programmierer weniger Namen für ein Programm merken.
- Sie kann über mehrere Projekte hinweg angewendet werden.
- Sie fügt Präzision den Namensbereichen zu, die sonst leicht unpräzise werden, wie z. B. die Unterscheidung von *Min* und *First*, *Last* und *Max*.
- Sie erlaubt das Überprüfen von Typen bei ADTs, die der Compiler nicht testen kann.
- Sie dient zur Dokumentation von Typen in einer schwach typisierten Sprache oder Umgebung, z. B. in der Fensterprogrammierung.
- Sie macht die Namen kompakter.

Die Nachteile der Ungarischen Konvention sind:

- Einige im Umlauf befindliche Versionen der Ungarischen Konvention ignorieren die Benutzung von ADTs als Basistypen. Stattdessen werden die sprachinternen Basistypen benutzt, so daß der Programmierer eine manuelle Typüberprüfung durchführen muß, anstatt dies vom Compiler prüfen zu lassen.

- Es wird die Bedeutung und die Repräsentation der Daten in deren Benennung kombiniert. Es wird keine Änderung des Namens gewünscht, wenn eine Variable ursprünglich als Integer deklariert wurde, und nun als Longreal gilt. Gerade dies wird aber durch die Ungarische Konvention nötig.

- Es werden wenig informative Variablennamen unterstützt, da der Bezeichner im Namen weggelassen werden kann und trotzdem eindeutige Namen entstehen. Der Programmierer muß sich dazu zwingen, diese Bezeichner für eine vollständige Benennung zu benutzen.

Für strikt typisierte Sprachen bringt eine derartig strenge Einschränkung der Namen keine zusätzlichen Vorteile. Daher sollte aus Gründen der Lesbarkeit in diesen Sprachen, zu denen auch TL zählt, auf eine derartige, formale Namenskonvention verzichtet werden.

2.4.8 Lesbare Abkürzungen

Der Wunsch nach Abkürzungen ist ein historisches Erbe aus den frühen Jahren der Computerwissenschaft. Frühe Programmiersprachen wie Basic, Fortran und Assembler haben die Variablennamen auf zwei bis acht Zeichen eingeschränkt und so die Programmierer gezwungen, Abkürzungen zu erfinden. In modernen Programmiersprachen wie C, Pascal oder Ada ist es nicht mehr nötig, sinnvolle Namen abzukürzen. Auch für TL sollten die nachfolgenden Richtlinien zur Abkürzung von Namen nicht zur Anwendung kommen.

Sollte man dennoch zu Abkürzungen durch Randbedingungen oder Projektvorgaben gezwungen sein, ist es besser, eine gute Methode dafür zu verwenden. Es gibt verschiedene Techniken, z. B. für die Bedeutung überflüssige Worte zu entfernen oder kurze Synonyme zu verwenden, die allein angewendet nicht unter allen Umständen gut sind.

Es folgt eine Aufzählung von Richtlinien für Abkürzungen, die sich teilweise widersprechen und daher nicht alle gleichzeitig angewendet werden sollten.

- Nutze Standardabkürzungen, die im Wörterbuch stehen und allgemein in Gebrauch sind.

- Entferne alle Vokale außer dem Anfangszeichen.

- Nimm den ersten oder die ersten Buchstaben eines jeden Wortes.

- Schneide nach dem ersten, zweiten oder dritten Buchstaben jedes Wortes ab.

- Nimm nur den ersten und letzten Buchstaben eines Wortes.

- Benutze jedes bezeichnende Wort im Namen bis maximal drei Worte.

- Entferne nutzlose Endungen wie *ing, ed.*

- Nimm den am meisten beachteten Laut in jeder Silbe.

- Iteriere über diese Techniken, bis jeder Variablenname auf die benötigte Anzahl von Zeichen gekürzt ist.

Es gibt phonetische Abkürzungen, die jeden Laut eines Wortes und nicht die Schreibweise durch Abkürzungen wiedergeben. Beispiele:

hilite, b4, sk8ing

für

Highlight, before, skating.

Bei vielen Abkürzungen muß man allerdings erst einmal rätseln: *TRMN8R* steht für *Terminator*!

Folgende Regeln sollten zur Vermeidung von Fallstricken berücksichtigt werden:

- Kürze nicht durch Weglassen nur eines Zeichens ab. Es bedeutet wenig zusätzlichen Aufwand, dieses Zeichen zu tippen, im Vergleich zum Verständnis beim Lesen. Ein Beispiel ist in vielen Kalendern die Abkürzung *Jun* und *Jul*.

- Führe Abkürzungen konsistent durch. Nutze immer dieselbe Abkürzung, z. B. an allen Stellen *Num* oder an allen Stellen *No*, aber nicht beide gleichzeitig. Kürze ein Wort nicht nur in einigen, sondern in allen Namen ab.

- Erzeuge Namen, die auch auszusprechen sind. *XPos* ist besser als *Xpstn*, da man häufig über den geschriebenen Code sprechen muß.

- Vermeide Kombinationen, die zu falscher Aussprache führen. Die Bezeichnung *ENDB* für das Ende eines Blocks ist besser als *BEND*, wobei diese Irreführung mit einer Trennungstechnik (wie z. B. in *b_end*) nicht passiert.

- Nutze einen Thesaurus, um Namenskollisionen zu beheben. Bei einer Begrenzung auf drei Zeichen könnten *fired* und *full revenue disbursal* zu der Abkürzung *frd* führen. Es ist besser, *fired* durch *dismissed* und *full revenue disbursal* durch *complete revenue disbursal* zu ersetzen, um die entsprechenden Abkürzungen *dsm* und *crd* unterscheidbar zu machen.

- Dokumentiere Abkürzungen in Übersetzungstabellen. Für Sprachen, die nur sehr kurze Variablennamen erlauben, sollte am Beginn des jeweiligen Blockes eine Übersetzungstabelle als Erinnerung an die jeweiligen mnemonischen Inhalte der Variablen stehen. Dieser Punkt muß bei der Kommentierung von Code in den Dokumentenköpfen (siehe Abschnitt 2.2.3) berücksichtigt werden.

- Denke daran, daß die Namen mehr den Leser als den Schreiber des Codes betreffen. Ein Test mit Code, den ein Autor lange nicht mehr gesehen hat und nun verstehen will, zeigt ihm die Fehler in seiner eigenen Abkürzungspraxis auf.

2.4.9 Problematische Namen

- Vermeide irreführende Namen oder Abkürzungen. Ein Name muß eindeutig sein. Beispiel: *FALSE* als Abkürzung für *Fig and Almond Season* führt zur Zweideutigkeit, weil *FALSE* im Allgemeinen als Gegensatz zu *TRUE* verwendet wird und nicht als abkürzende Bezeichnung.

- Vermeide Namen mit ähnlicher Bedeutung. *Input* und *InVal*, *Filenum* und *FileIdx* sollten vermieden werden, da sie semantisch leicht verwechselt werden können und somit schwer zu findende Fehler verursachen.

- Vermeide Namen mit unterschiedlicher Bedeutung, aber ähnlichen Namen. Finde einen neuen Namen oder eine andere Abkürzung, wenn *ClientRecs* und *ClientReps* auftreten. Namen sollten sich durch mindestens zwei Buchstaben unterscheiden oder sich am Anfang oder Ende des Namens unterscheiden. Technisch werden diese Unterschiede mit *psychologischer Distanz* bezeichnet.

- Vermeide Namen, die ähnlich klingen (*Homonyme*), z. B. *wrap* und *rap*. Sie sind ungünstig beim Diskutieren über den Code.

- Vermeide Numerierungen in Namen. Sollte eine Numerierung entscheidend sein, ist es besser, eine Feldvariable zu benutzen. Meist gibt es eine bessere Lösung, um *Total1* und *Total2* zu unterscheiden. Hier gilt das für die Bedeutung von kurzen, als temporär angesehenen Variablen analog.

- Vermeide falsch buchstabierte Worte in Namen. Das Benutzen von *hilite*, um drei Buchstaben zu sparen, nötigt den Leser, sich zu merken, wie *highlight* falsch geschrieben wurde, wie also die *richtige*, falsche Schreibweise ist: *Hilight*, *Hilit* oder *Highlite*?

- Vermeide Namen, die üblicherweise im Englischen falsch geschrieben werden: *acummulate, calender, prefered* und andere. Eine Liste solcher Namen findet sich in vielen englischen Handbüchern.

- Unterscheide Variablennamen nicht allein durch Großschreibung. In case-sensitiven Sprachen sollte man vermeiden, *Frd*, *FRD* und *frd* als Abkürzungen für verschiedene Objekte nebeneinander zu benutzen. Sie können leicht verwechselt werden.

- Vermeide die Namen von Standardbibliotheksroutinen und vordefinierten Variablen. Jede Sprache enthält eine Liste von reservierten und vordefinierten Namen.

- Benutze keine Namen, die in keiner Beziehung zu dem stehen, was die Variable repräsentiert. Die Namen der Freundin, der Oma oder des besten Bieres als Variablennamen sollten vermieden werden, wenn das Programm nicht von ihnen handelt. Aber auch in diesem Falle könnten sie sich ändern, so daß generische Namen wie *GirlFriend* oder *FavoriteBeer* vorzuziehen sind.

- Vermeide Namen mit Buchstaben, die schwer zu lesen sind. Das sind Zeichen, die durch ihr Aussehen in den meisten Schriften nur schwer auseinanderzuhalten sind, z. B. *O* und *0*, *l* und *1*. Sollte die einzige Unterscheidung zwischen zwei Namen in einem solchen Buchstaben bestehen, sollten andere Namen gewählt werden.

2.5 Anforderungen an Tycoon

Die Sprache TLhat gut strukturierte Kontrollanweisungen und besitzt mit dem Modul- und Bibliothekskonzept eine Voraussetzung zum Entwerfen gut strukturierter Programme. Das Layout und die Dokumentation muß zusätzlich durch Richtlinien festgelegt werden, um einheitliche Programme zu gewährleisten. Da in TLdie Länge der Variablennamen nicht beschränkt ist, kann auf Regeln zur Abkürzung von Namen verzichtet werden. Die ungarische Konvention ist nur im Zusammenhang mit abkürzenden Richtlinien nützlich. In anderen Fällen überwiegt der Nachteil ihrer schlechten Lesbarkeit, weil der Bedeutung tragende Teil des Namens nicht zu Beginn des Wortes steht.

Einige Regeln zur Vergabe von Operationsnamen und Bezeichnern müssen jedoch erstellt werden, um ihre Aussagekraft sicherzustellen. Die in diesem Kapitel festgehaltenen Erfordernisse für eine aussagefähige Dokumentation werden in Kapitel 5 in einem *Style Guide* an das Tycoon System angepaßt.

Kapitel 3

Programmiersprachliche Konzepte zum Aufbau von Bibliotheken

In diesem Kapitel werden die Sprachkonzepte eingeführt, welche die Grundlage für den Aufbau von Bibliotheken bilden. Hierzu werden im Vergleich zu Tycoon die Sprachen Eiffel, Modula-3, C++ und Smalltalk kurz vorgestellt. Um die grundsätzlichen Unterschiede dieser Programmiersprachen beschreiben zu können, wird vorweg eine Klassifikation nach verschiedenen Konzepten gegeben. Zur Unterscheidung zwischen klassischen Sprachen und objektorientierten Sprachen werden im Anschluß die Konzepte der Objektorientierung kurz vorgestellt. Nach den zusammenfassenden Einführungen in die hier verwendeten Sprachen werden weitere Konzepte erwähnt, die für eine Reorganisation der Basisbibliotheken im Tycoon System von Bedeutung sind.

3.1 Grundlagen

3.1.1 Klassifikation von Programmiersprachen

Das weite Spektrum der Programmiersprachen kann durch verschiedene Konzepte klassifiziert werden. Einige Klassifikationen werden hier aufgeführt, um eine Einordnung der in dieser Arbeit verglichenen Sprachen zu ermöglichen. Jede Sprache kann zu mehreren Kategorien gehören.

Das ***Programmiermodell***, auch Programmierstil genannt, dient dazu, den Algorithmus für die Berechnung auszudrücken.

- ***Imperative*** Programmierung beschreibt die Berechnung in Form von Anweisungen, wie Zuweisungen, Abfragen, Verzweigungen usw. Für die Ausführung werden Variable benötigt, welche die zu modifizierenden Datenwerte beinhalten. Dieses Modell ist das älteste Programmiermodell, weil es der von-Neumann-Architektur entspricht. Die Assemblersprachen, Algol, C und Ada gehören in diese Kategorie.
- ***Funktionale*** Programmierung wurde durch die Sprache Lisp verbreitet und basiert auf dem mathematischen Ansatz des Lambda-Kalküls. Es wird keine Variable benötigt, da die Berechnung als Funktion der Eingabedaten beschrieben wird, welche das Ergebnis als Ausgabedaten herausgibt. Rein funktionale Sprachen erzeugen schwierig zu pflegende Programme [Beaudouin-Lafon 94] und sind daher

selten. Aus pragmatischen Gründen wird vielfach das Konzept der Variablen in die Sprachen aufgenommen. Hierzu zählen Lisp mit allen Dialekten und ML.

- ***Logische*** Programmierung besitzt ebenso wie die funktionale Programmierung einen mathematischen Ansatz. Formale Logik wird hier durch Prädikate ausgedrückt, welche ein Programm beschreiben. Durch logische Ableitungen werden während der Laufzeit neue Formeln erzeugt, deren Wahrheitsgehalt untersucht wird. Die bekannteste logische Sprache ist Prolog.
- ***Objektorientierte*** Programmierung basiert auf der Datenkapselung. Sie hat Ähnlichkeiten mit der imperativen Programmierung, wobei der Schwerpunkt in der objektorientierten Programmierung auf den Datenstrukturen und nicht auf den Algorithmen liegt. Die Operationen des Objekts können imperativ, funktional oder nach dem logischen Programmiermodell konzipiert sein.

Das Konzept der ***Typisierung*** wird in Programmiersprachen eingeführt, um deren Sicherheit zu erhöhen, da typisierte Ausdrücke neben der Typüberprüfung zur Laufzeit mittels statischer Analyse des Programmtexts auf Typkorrektheit des Typsystems überprüft werden können.

- ***Statische*** Typisierung ist sehr einschränkend, weil sie im Rahmen der statischen Analyse während der Übersetzung jedem Ausdruck einen Typ zuordnet. Pascal ist ein typischer Vertreter dieser Art von Sprachen.
- ***Strikte*** Typisierung bedeutet, daß eine statische Analyse jeden Typfehler, der während der Ausführung auftreten wird, anzeigen muß, aber daß nicht jedem Ausdruck ein statischer Typ zugeordnet wird. Einige Typen müssen dann zur Laufzeit abgeleitet werden. Die meisten typisierten, objektorientierten Sprachen, wie z. B. Simula, Modula-3 und C++, sind strikt typisiert, ebenso wie Sprachen mit parametrischem Polymorphismus oder Generizität, wie z. B. Ada.
- ***Schwach*** typisierte Sprachen berechnen und prüfen ihre Typen erst zur Laufzeit ohne vorherige, statische Analyse. Dieses Konzept wird auch *dynamische* Typisierung genannt.
- ***Untypisierte*** Sprachen nutzen kein Typkonzept und testen die Gültigkeit in dieser Hinsicht nicht. Vom Ansatz her sind Lisp ebenso wie die objektorientierte Sprache Smalltalk untypisiert [Gall et al. 95].

Ein Typsystem erhöht die Sicherheit der Programmierung, welches insbesondere für große Software-Systeme entscheidend ist. Darüber hinaus führt es zu einer effizienten Ausführung [Beaudouin-Lafon 94].

Die ***Implementation*** einer Sprache kann auf verschiedene Weise erfolgen und ist nicht sprachabhängig.

- ***Interpreter*** einer Sprache prüfen die Semantik in einer einzigen Phase. Interaktive Interpreter führen jeden Ausdruck sofort nach seiner Eingabe in den Rechner aus. Der Benutzer kann schnell testen und ändern, aber die gesamte Ausführung wird langsamer, weil die gesamte semantische Prüfung in der Interpreterphase stattfinden muß, statt in einer einleitenden Compilerphase. Viele der objektorientierten Sprachen werden interpretiert, um den Vorteil der größeren Flexibilität hinsichtlich der Typüberprüfungen und Portabilität zu nutzen.

- ***Übersetzer*** (engl. *Compiler* erzeugen in einer einleitenden Übersetzungsphase aus dem Programmtext Code, der direkt durch die Maschine ausführbar ist und daher effizienter als interpretierter Code ist. Die Information des Typsystems kann genutzt werden. Deshalb werden die meisten typisierten Sprachen durch einen Übersetzer implementiert.
- ***Semi-kompilierte*** Sprachen sind wie interpretierte Sprachen charakterisiert. Zur Verbesserung ihrer Effizienz benutzen sie einen für den Benutzer unsichtbaren Übersetzer im Hintergrund, der das gesamte Programm, oder Teile davon, direkt in Maschinencode oder in eine Zwischensprache übersetzt, welche vom Interpreter ausgeführt wird. Viele Interpreter für objektorientierte Sprachen haben einen derartigen Compiler im Hintergrund, z. B. Smalltalk oder Self.

Die ***Modularisierung*** beschreibt die Art und Weise, in welcher die Programme vom Programmierer in Module geteilt und Daten gekapselt werden können. Module erhöhen die Sicherheit und bieten eine Basis für die Wiederverwendung von Code.

- Wird ***keine*** Modularisierung vorgesehen, liegt es in der Verantwortung des Programmierers, diese mit Hilfe von Programm- und Namenskonventionen nachzubilden. Beispielsweise verlangt die Sprache Pascal, daß alle globalen Variablen, Prozeduren und Funktionen verschiedene Namen erhalten und nur durch die Verschachtelung von Prozeduren können Definitionen verborgen werden. Die Sprache C erlaubt dagegen die Verteilung eines Programms auf mehrere Dateien mit privaten Variablen und Funktionen. Es werden aber keine verschachtelten Prozeduren oder Funktionen zugelassen.
- ***Explizite*** Modularisierung bedeutet die Einführung der Modularisierung als Konzept in die Sprache. Die Kombination mit anderen Konzepten, z. B. der Generizität in Ada, erlaubt die Definition von generischen Modulen, welche mit Typen parametrisiert sind.
- ***Implizite*** Modularisierung wird in objektorientierten Sprachen angeboten, indem es kein eigenes Konzept von Modulen gibt, sondern dieses implizit im Konzept der *Klasse* enthalten ist. Die Einheit der Modularisierung ist streng an die Datenstruktur gebunden. Diese enge Kopplung von objektorientierten Sprachen an die Modularisierung läßt diese Sprachen als besser geeignet für die Wiederverwendung erscheinen.

3.1.2 Objektorientierte Programmiersprachen

In modularen Programmiersprachen existieren neben der Typisierung der zu repräsentierenden Objekte Module, um die Eigenschaften dieser Objekte zu kapseln. Diese Konzepte von Modulen und Typen werden in objektorientierten Sprachen in dem einzigen Konstrukt der Klasse zusammengefaßt. Durch die Benutzung von Klassen als Modul und gleichzeitig als Typ werden Ableitungen von Klassen (sogenannte *Subklassen*), dynamisches Binden, Polymorphismus und insbesondere Vererbung möglich [Meyer 88].

Eine Sprache mit den Konzepten von *Objekten*, *Klassen* und *Vererbung* wird als objektorientiert bezeichnet [Gall et al. 95; Wegner 87]. Zusätzlich zu diesen Basiskonzepten enthalten die meisten Programmiersprachen die weiteren objektorientierten Konzepte *Polymorphismus* und *dynamisches Binden*.

Klassen und Objekte "Ein *Objekt* ist als Abstraktion eines 'realen Dinges' zu verstehen, welches einen inneren Zustand besitzt, der nur mittels der Methoden des Objekts manipuliert werden kann" [Gall et al. 95, S. 195]. Der Zustand des Objekts wird durch Felder repräsentiert. Die Methoden legen fest, welche Nachrichten oder Aufrufe an dieses Objekt geschickt werden dürfen. Durch den Austausch von Nachrichten zwischen Objekten wird der Kontrollfluß realisiert.

Eine *Klasse* beschreibt eine Menge von Objekten, die in bezug auf ihre Struktur ähnlich sind. Ein Objekt ist eine Instanz einer Klasse. Ebenso wie Objekte besitzen auch Klassen interne Klassenvariablen und Methoden, z. B. zum Erzeugen oder Löschen eines Objekts.

Da eine Klasse eine Schablone zur Erzeugung eines abstrakten Datenobjekts und eine Beschreibung ihrer Veränderung ist, kann sie als ein abstrakter Datentyp angesehen werden. Im Unterschied zur gewohnten prozedurorientierten Schreibweise fehlt in der Schreibweise der Methodenschnittstelle der erste Parameter, der den Typ des zu ändernden Objekts angibt. Dieser Typ ist in objektorientierten Sprachen systemimmanent bekannt.

Polymorphismus und dynamisches Binden "Unter Polymorphismus ist die Fähigkeit einer Größe zu verstehen, zur Laufzeit unterschiedliche Ausprägungen annehmen zu können" [Gall et al. 95, S. 196]. Polymorphe Sprachen erlauben mehr als einen Typ für ihre Funktionsparameter. Prinzipiell wird zwischen *Ad-hoc* und *universellem* Polymorphismus unterschieden (siehe [Cardelli, Wegner 85]).

Ad-hoc -Polymorphismus besteht, wenn eine Funktion auf einer endlichen Menge von Typen operieren kann. Unterschieden wird dabei

Überladen (engl. *overloading*), welches syntaktisch gleiche Namen für semantisch unterschiedliche Konzepte erlaubt, und

automatische Typanpassung (engl. *coercion*), welche als semantisches Konzept die Umwandlung des Funktionsarguments in den angegebenen Typ durchführt.

Universeller (engl. *universal*) Polymorphismus liegt vor, wenn Funktionen auf einer unbeschränkten Anzahl von Typen arbeiten. Im Unterschied zum *Ad-hoc*-Polymorphismus wird in diesem Fall der gleiche Code für verschiedene Argumenttypen ausgeführt. Hier wird unterschieden in

Inklusions- (engl. *inclusion*) Polymorphismus, welcher das Konzept der Vererbung modelliert, indem ein Objekt der Subklasse ebenso zur Superklasse zugehörig ist und in dessen Kontext verwendet werden kann, und

parametrischen (engl. *parametric*) Polymorphismus, welcher Generizität ermöglicht. Eine parametrisch polymorphe Funktion besitzt Typparameter, welche die Typen der Argumente für eine konkrete Verwendung bestimmen.

Vererbung Im Zusammenhang mit objektorientierten Sprachen und für die Strukturierung von Bibliotheken bildet die Vererbung ein grundlegendes Merkmal. Sie ist die Basis für die Wiederverwendung und Erweiterbarkeit der einzelnen Bibliotheksbausteine. Als zentrales Konzept der Objektorientierung erlaubt sie die Ableitung neuer Klassen von bereits vorhandenen Klassen. Dies kann eine Erweiterung, Spezialisierung oder Kombination von bereits vorhandenen Klassen sein.

Die Aussage "Klasse B erbt von Klasse A" bedeutet, daß die Instanzen von B dieselben Felder wie die Instanzen von A enthalten und daß die Methoden von A in B verfügbar sind. Zusätzlich kann die Subklasse B neue Felder zu den ererbten Feldern von A hinzufügen und neue Methoden definieren. Die von A ererbten Methoden können für B modifiziert werden. Weiterhin hat jede in B definierte oder redefinierte Methode Zugriff auf die Felder und Methoden von A und B.

Als Konsequenz aus dieser Vererbungsbeziehung folgt, daß B nicht modifiziert werden muß, wenn für A neue Methoden oder Felder hinzugefügt werden. Diejenigen Felder oder Methoden von A, die B nicht benutzt, können in A ebenso gelöscht werden.

Werden die Methoden von A in B nur redefiniert, so wird A *spezialisiert*. Werden bei der Definition der Subklasse B neue Methoden hinzugefügt, so wird die Klasse A *angereichert*. Die Vererbung erreicht daher zwei Effekte, die Spezialisierung und die Anreicherung.

Die Vererbungsbeziehung erzeugt einen Wald von Klassenbäumen. Jede Klasse ist ein Knoten im Wald. Die Wurzeln der einzelnen Bäume sind diejenigen Klassen, die von keiner anderen Klasse erben. Die Nachfahren im Baum sind die Subklassen. In den Sprachen, in denen nur ein einziges Objekt als Wurzel dient, entsteht nur ein einzelner Baum als Vererbungshierarchie.

Sprachen mit Mehrfachvererbung, d. h. daß jede Klasse direkt von mehreren Klassen erben kann, lassen den Klassenwald zu einem Graphen werden. Zur Vermeidung von zirkularen Definitionen ist in diesem Fall nicht erlaubt, Zyklen im Graphen zu haben, d. h. daß eine Klasse weder direkt, noch indirekt durch andere Superklassen, Subklasse von sich selbst sein darf. Die Einführung von Mehrfachvererbung kann zu einigen Problemen führen, welche insbesondere Namenskonflikte bei ererbten Feldern und Methoden betreffen. Da die Vererbung und insbesondere die Mehrfachvererbung keine Konzepte des Tycoon Systems sind, weil es nicht objektorientiert ist, sind die angesprochenen Probleme im Zusammenhang mit dieser Arbeit nicht von Bedeutung. Eine weiterführende Erläuterung der auftretenden Konflikte bei Mehrfachvererbung findet sich in [Beaudouin-Lafon 94].

Das Konzept der Vererbung erzeugt in sich geschlossene Objekte, welche ebenfalls als Modul betrachtet werden können. Ein Modul bietet seiner Umgebung eine Menge von Operationen an. Ohne Vererbung müßte jedes Modul sämtliche Operationen selbst deklarieren. Die Implementation dieser Operationen kann allerdings mit Hilfe schon deklarierter Operationen erfolgen. Die Vererbung bietet dafür die komfortable Möglichkeit, auf der Grundlage vorhandener Module neue Module durch einfaches Hinzufügen neuer Operationen zu definieren. Die vorhandenen Operationen müssen kein zweites Mal deklariert werden. [Meyer 88] unterscheidet diese verschiedenen Funktionen eines Moduls durch die Begriffe *Kunde* und *Ableitung*. In objektorientierten Sprachen ist eine Klasse

Kunde einer anderen Klasse, wenn sie deren Operationen benutzt, die in der Schnittstelle definiert sind. Beispielsweise kann eine Stack-Implementation auf einem Array beruhen und ist damit dessen *Kunde*.

Ableitung einer anderen Klasse, wenn sie als Erweiterung oder Spezialisierung dieser Klasse entworfen ist. Dies umfaßt Einfach- und Mehrfachvererbung.

Neben diesem Aspekt der Modularität hat die Vererbung noch einen weiteren Aspekt, der die Erweiterbarkeit betrifft. Das Konzept des Polymorphismus beschreibt für objektorientierte Sprachen die Möglichkeit einer Entität, sich zur Laufzeit auf verschiedene Instanzen zu beziehen. Z. B. kann die Berechnung der Länge einer Kante für ein Polygon oder die spezielle

Form eines Rechtecks durchgeführt werden. Welches Objekt wirklich berechnet wird, kann sich erst zur Laufzeit entscheiden.

3.2 Vorstellung der Sprachen

In diesem Abschnitt werden die Programmiersprachen Modula-3, Eiffel, Smalltalk, C++ und Tycoon kurz vorgestellt. Dabei werden im Besonderen diejenigen Konzepte betrachtet, welche die Grundlage für den Aufbau von Bibliotheken bilden.

3.2.1 Modula-3

Die Programmiersprache Modula ist imperativ und dient zur modularen Programmentwicklung. Modula-3 ist 1986 als Erweiterung von Modula-2 [Wirth 88] entwickelt worden, welches auf den von N. Wirth entwickelten Sprachen Pascal und Modula beruht. Dabei sind problematische Konzepte, wie variante Records und vorzeichenlose numerische Datentypen, weggefallen [Nelson 91]. Grundlegende Konzepte sind Schnittstellen (*interfaces*), Objekte, generische Module, Nebenläufigkeit (*threads*), Kapselung unsicheren Codes und automatische Speicherbereinigung (*garbage collection*). Die Ausnahmebehandlung (*exception handling*) sorgt dafür, daß bei Auftreten eines Fehlers der Programmablauf eventuell mehrere Sichtbarkeitsbereiche verläßt.

Ein Programm besteht aus mehreren Modulen. Jedes Modul wird getrennt vom Compiler übersetzt, wobei über Modulgrenzen hinweg dieselben Konsistenzprüfungen wie innerhalb des Moduls ermöglicht werden [Reiser, Wirth 94]. Die Schnittstellen werden explizit in einer eigenen Datei deklariert und in einer zugehörigen Datei als Modul implementiert. Ein abstrakter Datentyp wird repräsentiert durch ein Schnittstellenmodul, welches einen Datentyp und mehrere, *Methoden* genannte Prozeduren zu dessen Verarbeitung enthält. Die Verwendung von Modulen beschränkt allgemein den Sichtbarkeitsbereich von Variablen und verbirgt die Information über die Repräsentation des Datentyps (Geheimnisprinzip, engl. *information hiding*). Im Schnittstellenmodul werden Namen und Eigenschaften eines Typs spezifiziert, die für andere, importierende Module relevant sind. Das Implementationsmodul enthält lokale und damit private Variable und Operationen, welche die Schnittstelle nicht zur Verfügung stellen muß.

Diese Abstraktion in Form von Modulen verkörpert eine Erweiterbarkeit im Bereich der Prozeduren. Die für erfolgreiche Programmentwürfe wichtige Erweiterbarkeit im Bereich der Datendefinitionen wird von Modula-3 nicht unterstützt, weil Typen nicht sowohl erweitert als auch kompatibel sein können [Reiser, Wirth 94]. Diese Möglichkeit der Erweiterbarkeit von Typen eröffnen die objektorientierten Sprachen, indem alle Operationen, die für Variable der Superklasse definiert sind, auch auf Variable der Subklasse angewendet werden können.

In die Sprache eingebaut sind die fundamentalen Datentypen INTEGER, CARDINAL, BOOLEAN, CHAR, REAL und LONGREAL. Einige weitere Standardschnittstellen sind in jeder Sprachimplementation von Modula-3 vorgesehen. Diese Schnittstellen können als mitgelieferte *Bibliotheken* angesehen werden. Darüber hinaus wurden von Anwendern umfangreiche Bibliotheken entwickelt, wie die in [Horning et al. 93] beschriebenen Schnittstellen, welche in Abschnitt 4.1 vorgestellt werden.

In Modula-3 gibt es Objekte, welche aus Referenzen auf ein Datenfeld und zugehörige Methoden bestehen. Ein Objekt repräsentiert den Wert eines ADTs , dessen Operationen durch eine Methodenliste implementiert werden. Nach einem strengen Typkonzept wird für

jedes Objekt ein bestimmter Objekttyp definiert. Es kann nur auf Attribute oder Methoden eines Objekts referenziert werden, nicht auf den Objekttyp als solchen. Die Definition der Attribute und Methoden findet sich in einer Objekttypdefinition. Die Implementation der Methoden erfolgt außerhalb der Objekttypdefinition als Prozeduren.

Mittels einfacher Vererbung erben Subtypen von Objekttypen sämtliche Attribute und Methoden entlang der Vererbungshierarchie. Modifikation und zusätzliche Spezifikation der Objekttypen wird dabei ermöglicht. Eine allgemeine Subtyprelation $<:$ kann auf verschiedene Typen und ebenso auf Objekte angewendet werden. Ist S ein Subtyp von T ($S <: T$), dann ist jeder Wert vom Typ S ebenso vom Typ T und kann in jedem Zusammenhang verwendet werden, in dem ein Typ T verlangt wird. Zur Unterstützung dieser fundamentalen Subtypisierung führt der Compiler eine strikte Typüberprüfung durch.

Zusammen mit struktureller Äquivalenz und Subtypisierung bietet Modula-3 Inklusionspolymorphismus. Die Redefinition von ererbten Methoden stellt ein Überschreiben (*overriding*) von Methoden dar und ein Überladen (*overloading*) von Operatoren kennt Modula-3 ebenso. Des weiteren bietet Modula-3 parametrischen Polymorphismus durch die Möglichkeit, generische Schnittstellen zu bilden.

3.2.2 Eiffel

Die Sprache Eiffel wurde speziell für die objektorientierte Programmierung entwickelt. Sie ist für verschiedene Unix-Plattformen verfügbar. Da der Compiler als Zwischensprache während der Übersetzung C benutzt, kann Eiffel auf weitere Systeme, die einen C-Compiler umfassen, portiert werden. Die Sprachbeschreibung findet sich in [Meyer 90a].

Der Compiler erlaubt statische Typüberprüfung und dynamisches Binden. Mehrfachvererbung bildet das grundlegende Konzept zum Aufbau der Bibliotheken. Durch die Konstrukte ***inherit*** (engl., *beerben*) und ***deferred*** (engl., *aufgeschoben*) wird die Vererbungsbeziehung zwischen Klassen implementiert. Das Vererbungskonzept für Klassen bedeutet implizit eine Modularisierungsmöglichkeit für Eiffel-Programme.

Das Bilden generischer Unterprogramme oder Subklassen kann mit Hilfe der Vererbung simuliert werden, birgt jedoch den Nachteil, daß Code in den Klassen teilweise kopiert werden muß [Meyer 86]. Aus diesem Grund wurde in Eiffel zusätzlich zur Vererbung der Mechanismus der Generizität eingeführt. Generische Klassen mit formalen, generischen Parametern realisieren parametrischen Polymorphismus in Eiffel. Jede dieser Klassen beschreibt ein sogenanntes *type template*, von dem man durch Übergabe konkreter Typen eine direkt verwendbare Klasse ableiten kann. Die Formen des Ad-hoc-Polymorphismus werden von Eiffel nicht unterstützt.

Das Konzept der Zusicherungen (siehe Abschnitt 3.3.4) ist grundlegend in Eiffel und dient der Erhaltung der semantischen Konsistenz von Methoden bei ihrer Redefinition.

Eine Klasse beschreibt eine Menge von Laufzeit-Objekten, die durch ihre Attribute und Methoden charakterisiert werden. Ihre Sichtbarkeit für alle, ausgewählte oder keine anderen Klassen kann explizit bei der Klassendefinition festgelegt werden, um eine Art der Kapselung (engl. *information hiding*) zu realisieren. Objekte werden als Instanzen von Klassen zur Laufzeit explizit erzeugt.

Die Information über die Schnittstellen der Bibliotheken wird in Ada oder Modula-3 in Form einer eigenen Schnittstellendefinition aufbereitet. Dies bedeutet für den Implementierer dieser Bibliotheken, daß er viele Eigenschaften zweimal, in der Implementation und der Schnittstelle, deklarieren muß. Eiffel nutzt dagegen die in den Klassen vorhandene Information automatisch, um dem Bibliotheksbenutzer die nötige Klassenbeschreibung zu geben. Auf

diese Weise wird dem Implementierer die Arbeit einer separat zu erstellenden Dokumentation erspart.

Durch die Definition einer *flat-short form* (die Kurzform einer Klassendefinition mit Kommentaren und Methodenbeschreibungen) ist eine Dokumentation der Klassen gleichzeitig zur Entwicklung vorhanden. Das Werkzeug *short* stellt zu diesem Zweck direkt aus der Definition einer Klasse eine kurze Beschreibung her. Diese Kurzbeschreibung dient als Grundlage für den möglichen Wiederverwender, der diese Dokumentationsform als Informationsquelle über die Methoden und deren Zusicherungen nutzen kann.

Die *flat-short form* ist vergleichbar mit der Schnittstelle einer Klasse in modularen Sprachen. Implementationsdetails werden nicht dargestellt. Vererbende Klassen und exportierte Methoden sind darin enthalten. Die Darstellung einer Methode reduziert sich auf die Angabe der Parameter, des Kommentars zur Methode und deren Zusicherungen. Diese Information reicht für den Anwendungsprogrammierer aus, um entscheiden zu können, ob diese Methode in die gewünschte Anwendung eingebracht werden kann. Als Beispiel wird ein Ausschnitt der Schnittstelle eines Stacks gezeigt:

deferred class interface STACK[G]
feature specification
has(v: G): BOOLEAN
– Does structure include v?
ensure
not_found_in_empty: Result implies not empty

extendible: BOOLEAN
– May new items be added?
...
push(v: G)
– Push v onto top of stack.
require
not full
ensure
not empty; top = x

Die objektorientierte Sprache Eiffel benutzt die Vererbung zur Bildung der verschiedenen Klassenbibliotheken. Diese Vererbungshierarchie ist sorgfältig nach generellen Entwurfsprinzipien entworfen worden. Meyer sagt dazu in seinem Artikel [Meyer 90d]: "This article presents the concerted efforts which have been made to advance the cause of component-based software development through the construction of the Basic Eiffel Libraries."

Es existiert eine allgemeine, abstrakte Klasse, von der alle Bibliotheks- und Anwendungsklassen erben. Diese Klasse *ANY* enthält allgemeine Konzepte für beliebige Datenstrukturen. In Abbildung 3.1 sind die fundamentalen Klassen der Eiffel-Bibliotheken in ihrer Beziehung zu den gesamten Anwendungsklassen dargestellt.

In Eiffel gibt es kein sprachimmanentes Konstrukt zur Bildung von Bibliotheken. Alle zur Verfügung stehenden Datenstrukturen sind eine Sammlung von Klassen, die durch das Dateiverwaltungssystem geordnet sind. In der Beschreibung von Eiffel werden Klassen desselben Verzeichnisses mit einem Bibliotheksnamen, z. B. *Graphics Library*, versehen. Nur diejenigen Klassen, die für eine Anwendung benötigt werden, werden mittels des Vererbungskonstrukts

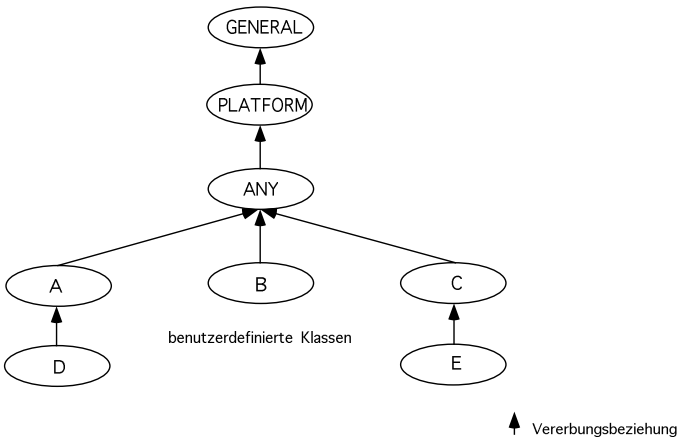


Abbildung 3.1: Fundamentale Hierarchie in Eiffel

in das Anwendungsprogramm integriert. Die Klassen eines Programms ergeben sich damit als Blätter des Vererbungsbaumes. Dieses Verfahren der Bibliotheksanbindung ergibt sich in objektorientierten Sprachen direkt aus dem Sprachentwurf. Es wird ein kleiner Sprachkern gebildet und alle wiederzuverwendenden Strukturen sowie abstrakte Datentypen werden als Klassen modelliert.

3.2.3 Smalltalk

Smalltalk-80 [Goldberg, Robson 83] ist eine objektorientierte, untypisierte Programmiersprache, die von der Xerox Palo Alto Research Center Learning Research Group in den siebziger Jahren als eine Interpretersprache entwickelt wurde. Im Hintergrund wird ein Compiler benutzt, so daß die Sprache als semi-kompiliert bezeichnet werden kann. Beeinflußt wurde sie vor allem vom Klassenkonzept in Simula67. Daher verwendet Smalltalk auch die Metapher von *Objekten*, die *Botschaften* miteinander austauschen. Es wurde erstmals versucht, ein einheitlich objektorientiertes System zu erstellen. Smalltalk stellt aus diesem Grund ebenfalls eine Arbeitsumgebung dar, die auf einem Arbeitsplatzrechner mit Maus und grafischem Bildschirm implementiert ist. Die fensterorientierte Bedienerschnittstellengestaltung ist mittlerweile eine Art Standard für Arbeitsplatzrechner geworden.

Ein Smalltalk-Programm besteht ausschließlich aus Objekten, welche einen internen Zustand annehmen können, auf den Operationen anwendbar sind. Smalltalk verfügt über einen allgemeinen Objektbegriff, so daß ein Objekt eine Datenstruktur, ein Prozeß oder eine Beschreibung eines Objekts (*Metaklasse*) sein kann. Nur durch Botschaften (engl. *messages*), welche den Operationsaufrufen in anderen Sprachen entsprechen, werden Operationen des Objekts angesprochen. Der Empfänger der Botschaft bestimmt, auf welche Weise die Operation ausgeführt wird. So wird das Prinzip de Kapselung (*information hiding*) gewahrt, indem der Sender der Botschaft nicht weiß, wie die gewünschte Operation durchgeführt wird. Die Menge der Botschaften, auf die ein Objekt reagieren kann, bildet seine Schnittstelle. Bei Auftreten eines Fehlers wird keine Ausnahmebehandlung durchgeführt, sondern der Fehler lediglich ausgegeben.

Eine Klasse in Smalltalk beschreibt die Implementation einer Menge von Objekten, die

alle eine ähnliche Art von Systemkomponente repräsentieren. Jedes einzelne Objekt ist eine Instanz seiner Klasse. Sie definiert die Schnittstelle ihrer Objekte, die in Gruppen von Methoden zur Veränderung, Initialisierung oder Anfrage unterteilt ist. Ein Klassenrumpf enthält die Implementation der Methoden.

Im Gegensatz zu anderen objektorientierten Sprachen haben die Variablen in Smalltalk keinen Typ. Daher führt der Interpreter keine Typüberprüfung durch, welches Smalltalk flexibler macht als andere Sprachen mit typisierten Modulen [Marcotty, Ledgard 87].

Grundsätzlich sind in Smalltalk alle Objekte polymorph, da die Methoden zur Laufzeit dynamisch ausgewählt werden (*Inklusionspolymorphismus*). Zur Erhaltung der Klassenkonsistenz wird eine automatische Typanpassung (*coercion*) durch Konvertierung von numerischen Werten in den jeweils allgemeineren Typ implementiert.

Ein wichtiges Konzept für die Bildung von Klassenhierarchien ist die Vererbung. Jede Klasse, mit Ausnahme der Klasse Object, ist Unterklasse einer anderen Klasse. Da jede Klasse höchstens eine Oberklasse besitzt, entsteht eine Baumstruktur von Klassen mit der Klasse Object als Wurzel. Eine Unterklasse erbt alle Eigenschaften der jeweiligen Oberklasse, welche sie auch neu definieren kann. Ebenso kann sie neue Eigenschaften hinzufügen. Damit ist die Unterklasse eine Spezialisierung der Oberklasse. Da für eine Klasse auch Methoden ihrer Oberklasse anwendbar sind und abhängig vom Beginn der Suche im Vererbungspfad einmal die spezielle und einmal die übergeordnete Methode aufgefunden wird, existieren die Mechanismen *self* und *super*. Mit ihrer Hilfe kann explizit angegeben werden, aus welcher Klasse die anzuwendende Methode stammt.

Smalltalk ist nicht nur eine Programmiersprache, sondern auch eine Arbeitsumgebung [Hoffmann 87]. Als Hilfsmittel während der Implementierungsphase steht dem Anwendungsprogrammierer ein *System Browser* als Werkzeug zur Verfügung. Der *System Browser* bietet die Möglichkeit, sich interaktiv über die im System implementierten Klassen zu informieren, sie zu ändern und weitere Klassen hinzuzufügen. Dadurch erhält der Anwendungsprogrammierer Information über die Aufgabe einer Klasse, die Stellung einer Klasse innerhalb der Klassenhierarchie, Sinn und Zweck der Variablen und Funktionsweise der Methoden. Er kann sich Übersichten über all diejenigen Klassen erstellen lassen, welche eine bestimmte Methode implementieren oder verwenden.

Mit diesem Werkzeug kann der Implementierer einer Anwendung erkunden, welche der bereits vorhandenen Klassen direkt oder als Vererbungsklasse für neu zu implementierende Klassen benutzt werden können. Er kann dann mit dem *System Browser* Klassen ändern oder hinzufügen.

Die Benutzerschnittstelle von Smalltalk-80 ist ein klassisches Beispiel für direkte Manipulation der Objekte, d.h. der Benutzer kontrolliert sein Modell durch Zeigen auf graphische Objekte. Es unterstützt objektorientiertes Programmieren, welches die direkte Manipulation begünstigt. Die Nachteile von Smalltalk sind zum einen, daß es als interpretative Sprache für zeitkritische Anwendungen ineffizient ist, zum anderen, daß es auf wenigen Betriebssystemen verfügbar ist und keine Anbindung von Bibliotheken in anderen Sprachen erlaubt [Gorlen 87].

3.2.4 C++

Die Sprache C++ ist eine Erweiterung von C mit dem Konzept der Objektorientierung. Ein Programm in C++ kann rein imperativ oder objektorientiert entworfen sein. Daher wird C++ eine *hybride* Sprache genannt. Diese Konstellation ermöglicht effiziente, systemnahe Programme und gleichzeitig die Weiterverwendung existierender C-Programme [Gall et al.

95]. In [Ellis, Stroustrup 90] wird der Sprachstandard für C++ beschrieben. Es existieren verschiedene Compiler, wie der GNU C++ Compiler *gcc* für Unix. C++ ist strikt typisiert und ermöglicht durch die Bildung von Klassen eine implizite Modularisierung.

Da C++ nicht, wie z. B. Smalltalk, vollständig objektorientiert ist, enthält es außer dem Sprachkern, welcher die Sprache C umfaßt, Werkzeuge, welche nach Wunsch eine objektorientierte Programmierung erlauben. Daher gibt es einige Einschränkungen im Vergleich zu anderen objektorientierten Sprachen:

- Nicht alle Datentypen werden einheitlich behandelt.

  Neben den Basistypen, die in C existieren, gibt es abstrakte Datentypen, die sich ähnlich wie eingebaute Typen verhalten. Die Basistypen sind dabei *keine* Klassen und können darum keine Funktionalität vererben. Ihre eingebauten Funktionen können ebenfalls nicht redefiniert werden.

- Es existiert *keine* Standardklassenbibliothek.

  Daher kann der Benutzer die objektorientierten Eigenschaften nicht voll ausschöpfen. Er muß jede höhere Klasse selbst definieren und kann sie dann wie die eingebauten Klassen verwenden.

  Da C++ weit verbreitet ist, werden zum Ausfüllen dieser Lücke zahlreiche Bibliotheken für spezielle Anwendungen angeboten. Es gibt inzwischen eine große Anzahl kommerzieller Software-Bibliotheken, die zum Teil in Abschnitt 4.4 kurz vorgestellt werden.

C++ bietet keine Trennung von Schnittstelle und Modul, wie Modula-3 oder TL. Der Vorteil einer Trennung liegt in der Flexibilität, Erweiterbarkeit, Portierbarkeit und Modularität der Software. Da eine Klasse B, die von der Klasse A erbt, die Schnittstelle mit öffentlichen Funktionen und die Implementation mit privaten Variablen erweitern darf, entsteht für die Schnittstellen und die Implementationen dieselbe Vererbungshierarchie. Um zu ermöglichen, daß eine abgeleitete Klasse nur die Schnittstelle erbt und auf einer anderen Implementation beruhen kann, werden die Konzepte der Mehrfachvererbung, virtuellen Funktionen und virtuellen Basisklassen benötigt. Ein Gitter von Schnittstellenklassen kann auf diese Weise durch mehrere Gitter von Implementationen unterstützt werden. Das Verfahren wird in [Martin 91] beschrieben und ist insbesondere für verteilte Umgebungen nützlich, weil die Schnittstellen global definiert werden, während ihre Implementationen lokal verteilt sind.

Durch dieses Verfahren der Trennung erhält C++ die Basis zur Vererbung von Implementationen, welches in herkömmlichen, modularen Sprachen mit Trennung von Schnittstelle und Implementation, wie z. B. Ada und Modula-3, nicht möglich ist. Die Möglichkeit der Trennung von Schnittstelle und Implementation wird in den in Abschnitt 4.4 verglichenen Bibliotheken, mit Ausnahme der Booch Components, nicht ausgenutzt, weil sie auf den klassischen Compilern beruhen, die keine Mehrfachvererbung oder virtuelle Funktionen anbieten.

Die vorgestellten Formen des Polymorphismus werden alle von C++ realisiert. Der parametrische Polymorphismus wird über den Mechanismus von Schablonen (engl. *templates*) unterstützt. Das Überladen von Operationen und eine benutzerdefinierte, implizite Typanpassung wird ermöglicht.

Das strikte Konzept der Datenkapselung kann durch spezielle Deklarationen aufgeweicht werden [Gall et al. 95]. Wird eine Klasse oder Methode als *friend* einer Klasse deklariert, so hat sie Zugriff auf den privaten Teil der Klasse, in welcher die Definition steht. Dies ist eine Umgehung der Methodenschnittstelle, welche zur Erhöhung der Laufzeiteffizienz führt, aber das Konzept der Kapselung verletzt.

3.2.5 Tycoon

Das Tycoon System [Matthes 93] [Mathiske et al. 93] ist eine persistente Programmierumgebung zur Entwicklung von datenintensiven Anwendungen. Es ermöglicht eine flexible Definition und Integration generischer Dienste in offenen Systemumgebungen. Das System verhält sich wie ein interaktiver Interpreter. Der Compiler erlaubt gleichfalls die Übersetzung einer gesamten Datei. Zur Definition neuer, sowie zur Anbindung existierender externer Dienste wurde die Programmiersprache TL entwickelt. Die algorithmisch vollständige, strikt typisierte und polymorphe Programmiersprache TL stellt generische Benennungs-, Bindungs- und Typisierungskonzepte zur Verfügung.

Ausgehend von der funktionalen Programmiersprache Quest [Cardelli 89] [Cardelli 90] über P-Quest [Matthes 91] [Niederée et al. 92], welches die Konzepte von Quest aufnimmt und um das Konzept der Persistenz erweitert, wurde TL entwickelt. Neben dem Ziel, einen abstrakten Sprachkern mit generischen Benennungs-, Bindungs- und Typisierungskonzepten für vordefinierte semantische Objekte zu entwickeln, wird TL hauptsächlich geprägt durch die Möglichkeit, diesen Sprachkern typsicher um Objekte und Dienste in einem *Add-On* Ansatz [Matthes, Schmidt 91] zu erweitern. Das Konzept der Subtypisierung mit dem Subtypoperator <: wurde aus Quest übernommen und ähnelt dem Subtypoperator in Modula-3.

Der funktionale Sprachkern von Tycoon enthält nach [Matthes 93; Münnix 94] die folgenden, grundlegenden Sprachkonzepte (vergleiche auch [Wetzel 94]):

- algorithmische Vollständigkeit,
- strikte und explizite Typisierung,
- Polymorphismus,
- Funktionen höherer Ordnung,
- strukturelle Subtypisierung für alle Typkonstruktoren,
- abstrakte Datentypen,
- Modularisierung und generische Bibliotheken,
- orthogonale Persistenz,
- typsichere Anbindung externer Funktionalitäten.

Tycoon ist auf die Formen des universellen Polymorphismus beschränkt. Es bietet keinen Ad-hoc-Polymorphismus, weil keine automatische Typanpassung und kein Überladen von symbolischen und alphanumerischen Bezeichnern [Matthes, Müßig 93, Abschnitt 2.5] möglich ist.

Im Rahmen dieser Arbeit ist von den vorgenannten Konzepten das der Modularisierung und Bildung von Bibliotheken von Bedeutung. Um größere Programme übersichtlich zu gestalten und laden zu können, werden sie in Module, Schnittstellen und Bibliotheken eingeteilt. Diese werden dann separat übersetzt, gebunden und ausgeführt. Dieses Konzept der Modularisierung bildet für den Aufbau und die Benutzung von generischen Bibliotheken die Grundlage und stellt ein Hilfsmittel zur Programmierung im Großen dar.

Aufgrund der funktionalen Sprachgrundlage sind in TL alle Operationen als Funktionen deklariert. Eine Funktion ist mit einer Prozedur vergleichbar, wenn sie den Supertyp *Ok* zurückgibt.

3.3 Spezielle Konzepte

Besondere Sprachkonzepte, die sich in den verglichenen Sprachen finden ließen und im Zusammenhang mit den Basisbibliotheken des Tycoon Systems eine Bedeutung haben, werden kurz vorgestellt und auf ihre Verwendung in Tycoon hin überprüft.

3.3.1 Abstraktionen für Datentypen

Wertabstraktion Operationen zum Kopieren von Elementen eines Behälterdatentyps (*copy*) und Vergleichen von Ausprägungen eines Behälterdatentyps (*equal*) werden in allen Datentypen, welche eine Sammlung von Werten darstellen, benötigt. Daher sollen derartige Operationen in einer eigenen Schnittstelle und Modulimplementation zusammengefaßt werden, um dann für jede Datenstruktur importiert werden zu können. Dies spart zahlreiche Deklarationen und Implementationen derselben Operation. In einer derartigen Schnittstelle für Wertabstraktionen muß vorgesehen werden, daß die Behälterdatentypen verschiedene Eigenschaften haben. Einige Wertausprägungen sind gleich, wenn ihre einzelnen Werte gleich sind, andere sind nur gleich, wenn sämtliche Verweise innerhalb der Datenstruktur auch gleich sind. Dies wird in einer neuen Schnittstelle *Value* für das Tycoon System berücksichtigt (siehe Abschnitt 5.2.5).

Numerische Abstraktion Eine Abstraktion der numerischen Eigenschaften für alle numerischen Basisdatentypen erscheint sinnvoll, da arithmetische Basisoperationen von allen Typen benötigt werden. Eine zentrale Schnittstelle zur Deklaration erspart die wiederholte Eingabe derselben Deklaration und sichert die Übereinstimmung in allen Basisdatentypen. Ein derartiges Konzept ist mit der Klasse *NUMERIC* in Eiffel realisiert worden. In Abbildung 3.2 wird dargestellt, in welcher Weise die Basisklassen von abstrakten Klassen der Standardbibliothek erben. Die zahlenwertigen Basisklassen (z. B. *INTEGER* und *REAL* erben sämtliche Basisoperationen (z. B. + und ∗) aus der abstrakten Klasse *NUMERIC*.

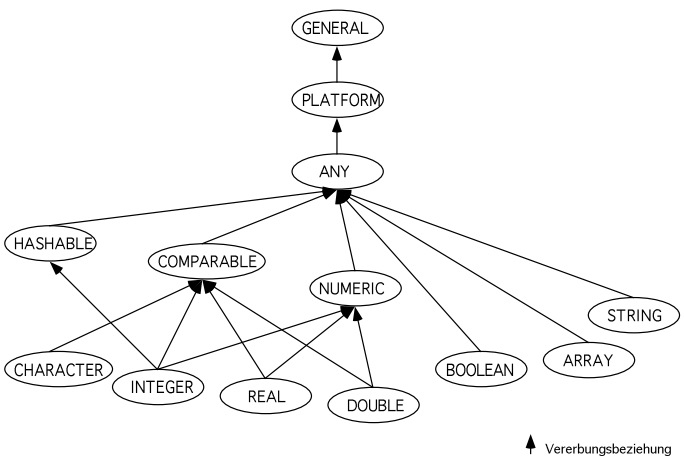


Abbildung 3.2: Hierarchie der Basisklassen in Eiffel

3.3.2 Iteratoren

Als Iteration wird das wiederholte Durchlaufen von Anweisungen oder die wiederholte Anwendung von Funktionen auf gleichartige Objekte bezeichnet [Engesser 93]. Ein Iterator ist der Mechanismus zur Durchführung einer Iteration [Traub 95]. Behälter bieten durch ihre homogene Struktur besonders gut die Möglichkeit der Anwendung und den Bedarf an Iteratoren. Da sie in den meisten Basisklassenbibliotheken enthalten sind, ist auch das Konzept der Iteratoren grundlegend für Basisklassenbibliotheken [Möller, Müller 93].

In den 50er Jahren wurden *Generatoren* eingeführt. Generatoren sind Subroutinen, die wie eine Funktion aufgerufen werden, aber bei jeder Beendigung einen neuen Wert zurückgeben. Ein neueres Konzept sind die *Iteratoren* [Baker 93]. Sie können als wiederholter Aufruf eines Generators angesehen werden, wobei auf die vom Generator erzeugten Werte Funktionen angewendet werden können.

In den 70er Jahren wurden in Sprachen wie Alphard und CLU spezielle Konstrukte zum Abkürzen von Iteratorkonstrukten, wie z. B. Schleifen über alle Elemente eines Behälters, eingeführt. Auch in Sprachen ohne diese Konstrukte können Iteratoren mit konventionellen Prozeduren implementiert werden [Lamb 87]. Seither wird das Konzept auch in objektorientierten Sprachen wie Eiffel und C++ verwendet.

"Im Kontext der funktionalen Programmiersprachen werden unter Iteratoren Funktionen höherer Ordnung verstanden. Im Gegensatz dazu versteht man in der Welt der objektorientierten Programmiersprachen Iteratoren oft als Objekte einer Iteratorklasse, welche die zur Iteration nötige Funktionalität anbietet."[Traub 95]

Es gibt damit verschiedene Sichten auf das Konzept des Iterators. In Tycoon wird ein Iterator als eine homogene Sammlung von Elementen aufgefaßt, auf die nur mit wenigen Methoden zugegriffen werden kann. Dies wird durch die abstrakte Schnittstelle *IterRep* repräsentiert.

Iteratoren als Konzepte

Ein *Iterator* ist ein Mechanismus, der für eine Datenstruktur beschreibt, wie eine beliebige Operation auf jedes Element eines Exemplars der Datenstruktur angewendet wird. Er bezeichnet also einen Traversierungs- und Positionierungsmechanismus für Datenstrukturen, wobei der Traversierungsmechanismus das Navigieren innerhalb der Operation festlegt.

[Möller, Müller 93] unterscheiden folgende Konzepte von Iteratoren:

Passive* Iteratoren** erhalten vom Benutzer eine Botschaft, die vom Iterator auf alle Elemente angewendet wird. Dies wird sinnvoll, wenn an alle Elemente eines Behälters die gleiche Botschaft gesendet werden soll. Passive Iteratoren bauen auf aktiven Iteratoren auf und stellen nach außen die Schnittstelle für den Benutzer zur Verfügung. Sie stellen das funktionale Modell eines Iterators dar. Sie werden in [Traub 95] auch ***interne Iteratoren genannt.

Ein passiver Iterator stellt einen einzigen Service (*iterate*), der eine Operation für alle Elemente des Containers ausführt, zur Verfügung. Dieses Konzept wird in den Ada-Bibliotheken für ADTs verwendet. [Uhl, Schmid 90]

Die Vorteile passiver Iteratoren bestehen in der Sicherheit vor fehlerhafter Benutzung und unterbrechender Ausführung anderer Operationen auf dem Container.

Ihr Nachteil besteht in der fehlenden Granularität. Alle Elemente müssen vom Iterator besucht sein, bevor ein weiterer Zugriff auf eines der Elemente möglich ist. Kontroll-

fluß und Sequentialisierung wird auf diese Weise mehr vom System als vom Benutzer kontrolliert.

Aktive* Iteratoren** sind eigene Klassenobjekte. Dem Benutzer werden Zugriffsrechte für die Aufzählung von Elementen innerhalb eines Behälters zur Verfügung gestellt. Die Zugriffsmethoden können vom Benutzer direkt aufgerufen werden. Sie werden in [Traub 95] auch ***externe Iteratoren genannt.

Aktive Iteratoren stellen elementare Schleifenkontrollstrukturen zur Verfügung, die für eine Besucherfolge oder generelle Schleife über die Elemente zusammengesetzt werden können. Diese Art von Iteratoren ist in C++-Klassenbibliotheken weit verbreitet. Die Kontrollstrukturen sind:

1. "Zugriff auf aktuelles Element", z. B. *get*
2. "Vorgehen zum nächsten Element", z. B. *next*
3. "prüfe, ob weitere Elemente vorhanden sind", z. B. *isValid*

Der Vorteil aktiver Iteratoren ist ihre größere Feinheit und Flexibilität. Dabei muß man ihre Unsicherheit bei fehlerhafter Benutzung in Kauf nehmen. Eine falsche Zusammensetzung der Iterationsschleifenkontrollstruktur, sowie Zustandsänderungsoperationen, die mit Ausführung einer Iterationsschleife aussetzen, können die Abbildung des Behälterzustands inkonsistent werden lassen.

Jeder Benutzer einer Bibliothek mit Iteratoren sollte diese möglichen Sicherheitsprobleme kennen, um sie zu vermeiden.

Die Schleifenkontrollstrukturen werden für Iteratoren durch einfache Operationen, wie *isEmpty, get* und *rest*, realisiert.

Es gibt mehrere *Implementationsansätze* für Iteratoren. Aus konzeptioneller Sicht sollten Iteratoren über eine 'benutzt'-Beziehung implementiert werden. Hierdurch wird die Möglichkeit geschaffen, mehrere Iteratoren pro Exemplar zu benutzen. Wird die Speicherung der aktuellen Position der Iteratoren hingegen nicht vom Behälter getrennt, so kann pro Exemplar nur ein Iterator auf dem Behälter aktiv sein.

In objektorientierten Sprachen stellt die Klasse *Iterator* Methoden zur Navigation in Behälterklassen zur Verfügung. In jeder Behälterklasse existiert eine Methode, die der Erzeugung eines Iterators aus der abstrakten Iteratorklasse dient.

Wird der Iterator als Markierung aufgefaßt, so kann diese

- zwischen zwei Elementen des Behälters stehen.

  Dabei treten Probleme der Positionierung auf, wenn der Behälter keine Elemente enthält oder ein Element neben der Markierung entfernt wird.

- direkt auf einem Element des Behälters stehen.

Ebenfalls kann die Richtung der Navigation eines Iterators kann in beide Richtungen erlaubt oder pro Exemplar eingeschränkt sein. Dies zieht Probleme bei Einfügungs- oder Entfernungsmethoden nach sich. *Stabile Iteratoren* lösen dieses Problem. Sie ermöglichen die Modifizierung einer Datenstruktur auch dann, wenn dieser aktive Iteratoren zugeordnet sind. Die Modifikationsmethoden kennzeichnen den Status ihrer Ausführung mit einem Rückgabewert. Mit dieser Hilfe werden die Zustände der aktiven und stabilen Iteratoren verwaltet. ET++ ist die einzige Bibliothek, die stabile Iteratoren verwendet.

Stabile Iteratoren

Ein allgemein bekanntes Problem der Iteratoren entsteht, wenn sich der Zustand des zugrundeliegenden Behälters während der Iteration ändert. Man kann dabei drei Problemklassen unterscheiden:

1. Entfernen des aktuellen Elements

   Dieses Problem tritt auf, wenn innerhalb einer Iteratorschleife das für den Iterator aktuelle Element mit den Methoden des Behälters entfernt wird. Welches ist das aktuelle und welches das nächste Element, um die *get*- und *next*-Operation ausführen zu können?

   Dieses Problem basiert auf einer logischen Ebene. Es ist daher unabhängig von der Funktionalität, Repräsentation und Implementation in jeder Containerklasse zu finden.

2. Implizites Shiften des aktuellen Elements

   Dieses zweite Problem tritt nur bei bestimmten Repräsentationen des Behälters auf. Es wird durch das Beispiel des Einfügens im Behälter nach einem bestimmten Element repräsentiert. In diesem Fall müssen alle Elemente vor diesem Element, und damit auch das aktuelle Element selbst, einen Eintrag vorgeschoben werden, um Platz für das einzufügende Element zu machen. Danach würde das aktuelle Iterator-Element aber nicht mehr auf das ursprüngliche Element zeigen. So wird entweder ein Element von der Iteratorschleife übersprungen oder doppelt behandelt.

3. Ungültigkeit des Iterators

   Bei einigen Datenstrukturen, z. B. Bäumen, ist es sinnvoll, den gesamten Pfad von der Wurzel bis zum aktuellen Element im Iterator zu speichern. Ein solcher Iterator wird *spanned* genannt, da er nicht nur den Zeiger auf das aktuelle Element, sondern auch auf anderer Elemente enthält. Es treten ähnliche Probleme wie im vorigen Fall auf, wenn der Baum nach dem Einfügen oder Entfernen eines Elements reorganisiert werden muß. Die Referenz auf das aktuelle Element bleibt zwar korrekt, aber der Pfad von der Wurzel zu diesem Element ist nicht mehr richtig. Es kann dann nicht mehr mit Hilfe des Pfades auf das nächste Element zugegriffen werden, sodaß Elemente zweimal oder gar nicht vom Iterator besucht werden.

Das korrekte Verhalten von sicheren, aktiven Iteratoren bei derartigen unterbrechenden Änderungsoperationen kann definiert werden. Eine Realisierung und Implementation wird in [Schmid 94b] auf der Grundlage von C++ gezeigt. Dazu wird ein Konzept der **stabilen** (auch *sicher* genannten) Iteratoren eingesetzt.

Iteratoren in Eiffel

Die Iteration ist eine systematische Traversierung von Strukturen [Meyer 90b]. Die Iteratorklassen kapseln Kontrollstrukturen für die Traversierung. Mit Iteratoren werden genau die Traversierungseigenschaften von Containerklassen abstrahiert.

Ein Konzept der Iteration wird zum Durchlaufen von Klassen (Listen, Bäume) in vorbestimmter Reihenfolge benötigt. Dabei wird eine bestimmte Aktion auf alle oder ausgewählte Elemente der Klasse angewendet.

Eine Iteratorklasse umfaßt daher:

1. Eine Datenstruktur eines bekannten Typs mit spezieller Ordnung.
2. Eine Operation *item_action*, die auf ausgewählte Items angewendet wird.
3. Eine boolsche Anfrage *item_test*, anwendbar auf alle in Frage kommenden Items.
4. Eine Politik oder Richtlinie, basierend auf *item_test*, um zu bestimmen, auf welche Elemente *item_action* angewendet wird.

Mit der Iteratorklasse werden Strategien für alle Standardkontrollstrukturen angeboten, wie *for ... all*, *upto ... first* etc. Damit werden diese Schleifenkonstrukte und Quantoren durch Iteratoren ersetzt, um auf diese Weise häufig benutzte Strukturen zu kapseln.

Damit ist das Schreiben von Exit-Bedingungen oder Invarianten überflüssig und die Weiterverwendung in abgeleiteten Klassen möglich.

Iteratoren kapseln nicht die Eigenschaften der Datenstruktur, sondern der Struktur-Clients. Daher werden Iteratoren NICHT in den Datenstrukturklassen wie *Tree* oder *Set* mit aufgenommen, sondern als eigene Klasse modelliert. Würden beispielsweise die Iterator-Operationen in *LIST* modelliert und soll in *TEXT_PROZESSOR* über alle Paragraphen einer Liste iteriert werden, dann wird damit gefordert, daß jedes Element ein Listenelement ist. Somit müßte *TEXT_PROZESSOR* ein Erbe von *LIST* sein. Dies ist aber nicht die Bedeutung von *TEXT_PROZESSOR*. Mit Iteratoren als abstrakte Klassen entsteht dieses Problem der zweifachen Bedeutung nicht.

Abbildung 3.3 zeigt die Hierarchie der Iteratorklassen. Diese sind nach den Klassifizierungskriterien in verschiedene, abstrakte Iteratorklassen eingeteilt, um damit die Eigenschaften verschiedener Behälterklassen nachvollziehen zu können.

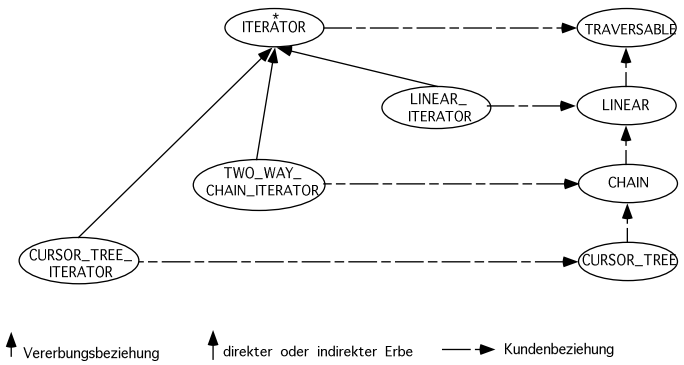


Abbildung 3.3: Iteratorklassen in Eiffel

Iteratoren in Tycoon

Abstrakte Schnittstelle. Ein Iterator kann als die Verallgemeinerung verschiedener Behälter angesehen werden. Spezielle Eigenschaften, wie das Vorhandensein einer Ordnungsrelation oder der Frage, ob Duplikate von Objekten enthalten sind, werden gekapselt. Das Verbergen der Implementation hat zur Folge, daß Iteratoren zur Transformation von der einen Darstellung (z. B. *list*) einer Objektmenge in eine andere Darstellung (z. B. *set*) benutzt werden

können. Zu diesem Zweck sind in den meisten Schnittstellen und Modulen der Massendatenbibliothek *bulkLib* die Operationen *create* und *elements* eingefügt. *elements* erzeugt einen Iterator mit den angegebenen Elementen, während *create* mit den Elementen eines Iterators einen Behälter des jeweiligen Datentyps erstellt.

Grundlage des Iterators ist die abstrakte Schnittstelle IterRep. Sie enthält als Typdefinition des Iterators die elementaren Operationen

- empty() :Bool
- get() :E
- rest() :T(E)

als Tupeltyp. Für die Ausführung von *get* und *rest* muß *not(empty())* erfüllt sein. Diese Operationen sind mit den elementaren Iteratordefinition *Get, Next* und *IsValid* in [Schmid 94b] zu vergleichbar.

In Tycoon wird der Iterator als eine Sammlung von Elementen betrachtet, deren erstes Element mit *get()* zugreifbar ist. Mit *rest()* wird das erste Element, das gerade behandelt wurde, "herausgeschnitten" und die übrigen Elemente werden als neuer Iterator (d. h. einer Menge von Elementen) zurückgeliefert.

Hierbei bezeichnet "erstes Element" immer das erste für die auf dem Iterator definierte Folge. In Tycoon sind im Moment nur sequentielle Iteratoren implementiert. Grundsätzlich sind Iteratoren aber nur als homogene Folge von Elementen definiert, deren Ordnung unbestimmt ist. Mit den Iterator-Operationen muß nur sichergestellt werden, daß jedes Element genau einmal durchlaufen wird. Diese Abfolge muß nicht dieselbe Reihenfolge haben wie die Ordnung im Behälter.

Da in Tycoon der Iterator als ein Datentyp modelliert wird, ist er unabhängig vom Behälterdatentyp. Es muß nicht für jeden Typ von Massendatenstruktur eine eigene Iteratorstruktur implementiert werden.

Benutzerschnittstelle. Auf der abstrakten Schnittstelle aufbauend werden Funktionen höherer Ordnung in der Schnittstelle *Iter* zur Verfügung gestellt, die eine systematische Traversierung der Objekte erlauben. Die Traversierungseigenschaften von Behälterdatenstrukturen werden gekapselt. Damit können die Operationen des Iterators zur Traversierung auf den Elementen eines Behälters benutzt werden. Die Iterationsmethoden sind dabei für alle Behälter gleich definiert, ihre Implementierung hängt allerdings von deren Eigenschaften ab.

Für den Anwendungsprogrammierer ist die abstrakte Schnittstelle eines Iterators verborgen. Er muss nur die in *Iter* deklarierten Operationen kennen. Der Programmierer von Bibliotheken muß dagegen bei der Implementation von Massendatentypen sowie des Iterators die Schnittstelle *IterRep* benutzen.

Das Iterator-Modul enthält alle möglichen Traversierungsoperationen (u. a. *forDo*, *forEach*). Dies entspricht den Operationen, die in der Objektorientierung gefordert bzw. üblich sind (siehe [Meyer 90b]).

Weiterhin enthält die Iterator-Schnittstelle Operationen aus verschiedenen, möglichen Iterator-Implementationen, wie z. B. *order, reverse, first, last, theOnly* für das Modul *sequence* oder *any, member* für das Modul *set*. Einige der Operationen werden also eher für ungeordnete Datentypen verwendet, anderer eher für geordnete. Dies wird von der Sicht des Benutzers bestimmt. Es ist nötig, alle diese Operationen in einer Iterator-Schnittstelle zur

Verfügung zu stellen, da der Iterator allgemein anwendbar ist und nicht speziell für jeden Datentyp erstellt wird.

Weitere Operationen werden für die Datenbankapplikation eingesetzt, z. B. *map, zip* oder *fold*. Diese Operationen dienen der Schnittstelle von Tycoon zu Datenbanken und sind wegen der besonderen Eigenschaft von Tycoon als System zur Anbindung von Datenbanken für die Bibliothek wichtig.

3.3.3 Modellierung von Datum und Zeit

Module zur Datums- und Zeitangabe sind ein wichtiges Element für den Anwendungsprogrammierer. Unter anderem ist es in vielen Anwendungen sinnvoll, die Dauer eines Ereignisses angeben zu können. Dies sollte in unterschiedlicher Darstellung, z. B. als Anzahl von Tagen oder als Anzahl von Monaten, möglich sein. Für erzeugte Dokumente wird vom Anwendungsprogrammierer beispielsweise die Angabe von Datum und Zeit gewünscht, um nicht nur die Version, sondern auch das Alter und damit auch die Gültigkeit des Dokuments auf lange Sicht feststellen zu können. Der Benutzer eines Programms wünscht vielleicht die Darstellung des aktuellen Datums auf jedem Eingabebildschirm. Es ist daher nötig, eine Vielzahl von Attributen und Routinen für die Datums- und Zeitverwaltung auf unterer Ebene bereitzustellen. Es bietet sich an, diese Verwaltungsroutinen in einer Basisbibliothek zur allgemeinen Verfügung zu stellen. In den folgenden zwei Abschnitten wird die Modellierung der Datums- und Zeitangaben in der Datenbankabfragesprache *SQL* und der höheren, modularen Programmiersprache *Modula-3* beispielhaft dargestellt.

Modellierung in SQL

Die Möglichkeiten der Darstellung von Datum und Zeit sind in SQL sehr vielfältig [Date, White 89]. Die vielen zur Unterstützung des Programmierers angebotenen Funktionen erschweren allerdings auch den Überblick und die Verwendung der Datums- und Zeitdatentypen.

In SQL werden Datum und Zeit in drei Datentypen zur Verfügung gestellt, die in Abhängigkeit zu der Zeitzone UTC (*universal time coordinated*, früher GMT = *Greenwich mean time*) definiert werden. SQL schreibt nur die Anzahl der darzustellenden Positionen vor und läßt die Form der Implementation offen.

DATE speichert Werte für das Jahr, den Monat und den Tag eines Datums in einer 8-stelligen Zeichenkette *yyyymmdd*, welche den Tag ebenso wie den Monat zweistellig und das Jahr vierstellig abbildet. Somit können Datumswerte vom 1.1.0000 bis zum 31.12.9999 repräsentiert werden. Daten vor unserer Zeitrechnung (*B.C.* = Before Christ) können nicht verarbeitet werden. Hierfür fehlt als Grundlage eine generelle Vereinbarung, um z. B. das Jahr 0 auf verschiedenen Rechnern einheitlich zu behandeln.

TIME speichert Stunde, Minute und Sekunde einer Zeitangabe in einer mindestens 6-stelligen Zeichenkette *hhmmss*. Erlaubte Werte decken den Bereich von 00:00:00 bis 23:59:61 ab. Implementationsabhängig kann die Genauigkeit der Sekunden auch in Bruchteilen angegeben werden. Ein Datentyp wird entweder durch TIME oder durch TIME(p) spezifiziert, wobei p die Anzahl der Nachkommastellen angibt. Um die Synchronisation mit der realen Weltzeit zu ermöglichen, werden 61 Sekunden erlaubt, wenn bei einem Ausgleich zum Jahreswechsel ein oder zwei Sekunden eingefügt werden müssen.

TIMESTAMP wird durch die Zusammensetzung der Typen DATE und TIME gebildet, mit einer Genauigkeit zur nächsten Microsekunde. Die Einschränkungen der beiden oben beschriebenen Typen gelten hier ebenfalls. Wie bei TIME kann die Angabe des Bruchteils durch die Spezifikation TIMESTAMP(p) angegeben werden.

Im neuen Standard von SQL-92 [Melton, Simon 93] werden weitere Datentypen aus TIME und TIMESTAMP gebildet, die zusätzliche Information über die Abweichung der repräsentierten Zeit zur UTC enthalten.

TIME WITH TIME ZONE enthält zusätzlich zu den Angaben in TIME Information über den Offset zu UTC als ein INTERVAL HOUR TO MINUTE der Länge 6.

TIMESTAMP WITH TIME ZONE enthält zusätzlich zu den Angaben in TIMESTAMP Offset-Information zu UTC der Länge 6.

Intervalle sind in SQL-92 die Differenz zweier Daten oder Zeiten zur Angabe der Dauer eines Ereignisses. SQL-92 definiert zwei Arten von Intervallen, die nicht miteinander verknüpft werden können: die *Jahr-Monat-Intervalle* und die *Tag-Zeit-Intervalle*. Mit einem Parameter kann die Anzahl der Nachkommastellen bei der Intervallspezifikation angegeben werden.

Jahr-Monat-Intervalle können nur den Wert eines Jahres, eines Monats oder beides enthalten, z. B. INTERVAL YEAR, INTERVAL MONTH(p), INTERVAL YEAR(p) TO MONTH.

Tag-Zeit-Intervalle enthalten Werte für Tage, Stunden, Minuten oder Sekunden, z. B. INTERVAL DAY(6) TO MINUTE, INTERVAL MINUTE(3) TO SECOND(4).

Intervalle können mit den Datums- und Zeitangaben innerhalb eines Ausdrucks manipuliert werden. Ein Intervall kann beispielsweise von einer Tageszeitangabe subtrahiert werden, um wieder eine Tageszeit zu erhalten.

In SQL-92 erlaubt ein *CAST*-Ausdruck die Konvertierung eines Datentyps in einen anderen Datentyp. Bei Datums- und Zeittypen kann dies nur mit folgenden Einschränkungen durchgeführt werden:

- Jeder numerische Wert kann in ein Intervall mit einer Komponente umgewandelt werden.
- Ein String kann in einen Datums- oder Zeittyp verwandelt werden, wenn der Inhalt der Zeichen für diesen Typ sinnvoll ist.
- DATE kann zu einer Zeichenkette oder zu TIMESTAMP (TIME-Anteil wird mit 0 aufgefüllt) konvertiert werden.
- TIME kann zu einer Zeichenkette oder zu TIME mit anderen Bruchangaben für Sekunden oder zu TIMESTAMP (DATE-Anteil wird mit 0 aufgefüllt) konvertiert werden.
- TIMESTAMP kann zu einer Zeichenkette, zu DATE, zu TIME oder zu TIMESTAMP mit anderen Nachkommastellen konvertiert werden.
- Ein Jahr-Monat-Intervall bzw. Tag-Zeit-Intervall kann zu einer Zeichenkette, einer Zahl oder zu einem anderen Jahr-Monat-Intervall bzw. Tag-Zeit-Intervall mit anderer Feldpräzision konvertiert werden.

Modellierung in Modula-3

Die Schnittstellen *Date*, *Time* und *Tick* [Horning et al. 93] ermöglichen den für alle Plattformen einheitlichen Zugriff auf die Zeitdarstellung des jeweiligen Betriebssystems. In der aktuellen Implementierung der Modula-3 Bibliothek[1] werden diese drei Datentypen in Abhängigkeit vom Betriebssystem deklariert.

Time ist die Darstellung der Zeit als LONGREAL. Die fragliche Zeit wird in Sekunden seit einem festgelegten Bezugszeitpunkt dargestellt. Dazu wird die Funktion *Now() :T* zur Ermittlung der aktuellen Zeit deklariert. Die Variable *Grain* ist ebenfalls als LONGREAL deklariert und liegt im Intervall (0...1]. Sie gibt die Auflösung der Zeitdarstellung an und ist nie größer als 1/60 Sekunde. Nach der Anweisungsfolge
t0 := Time.Now(); t1 := Time.Now()
ist entweder *t1 = t0* oder *t1 >= t0 + Time.Grain*.

Date ist die komplexe Darstellung des Datums. Es wird als Recordtyp

```
TYPE T = RECORD
  year: CARDINAL;
  month: Month;          {Monate: Jan, Feb,...}
  day: [1..31];
  hour: [0..23];
  minute: [0..59];
  second: [0..59];
  offset: INTEGER;       {Differenz von UTC zur lokalen Zeitangabe}
  zone :TEXT;            {Angabe der Zeitzone zur Interpretation der Darstellung}
  weekDay: WeekDay;      {Wochentage: Sun, Mon,...}
END;
```

deklariert und kann jeden Zeitpunkt seit dem 01. Januar des Jahres 00 nach dem gregorianischen Kalender darstellen.

Zu diesem Datentyp werden nur zwei Funktionen deklariert:

FromTime konvertiert eine Zeitangabe vom Typ *Time* in den Typ *Date*.

ToTime konvertiert ein Datum in eine Darstellung vom Typ *Time*.

Tick ist der Wert einer Zeit mit Auflösung unterhalb einer Sekunde. Dieser Wert ist implementationsabhängig und wird in der genannten Bibliothek als Typ *WORD* implementiert. Für den Typ *Tick* werden drei Funktionen deklariert, die dem Benutzer den Zugriff ermöglichen:

Now() gibt den aktuellen Zeitpunkt in *Tick*.

ToSeconds(t) gibt die Ticks *t* in der Darstellung von Sekunden wieder.

[1] Im WorldWideWeb ist die aktuelle Implementation von DEC SRC unter *http://www.research.digital.com/SRC/m3sources/html/INDEX.html* erreichbar.

FromSeconds(s) stellt die Sekunden *s* als *Tick* dar. Bei der Konvertierung ist eine Rundung nötig.

Tick.Now und *Time.Now* werden über lange Zeiten nicht synchronisiert. *Ticks* messen kleine Intervalle korrekt, um z. B. Antwortzeiten eines Prozesses zu messen. Dagegen gibt *Time* die übliche Uhrzeit an, welches eine Darstellung der 24 Stunden eines Tages in Sekunden erfordert.

Modellierung in Tycoon

In diesem Abschnitt wird erläutert, welches Konzept in Tycoon bisher für die Realisierung von Datumsangaben verwendet wurde. In Tycoon gibt es in der Version 1.0 der TL-Bibliotheken nur einen Datentyp für *Date* und *Time* mit den folgenden Feldern und deren erlaubten bzw. sinnvollen Werten:

year	*(* 0001 .. 9999 *)*
month	*(* [jan .. dec] *)*
day	*(* [1 .. 31] *)*
hour	*(* [0 .. 23] *)*
minute	*(* [0 .. 59] *)*
second	*(* [0 .. 59] *)*
offset	*(* Zahl *)*
zone	*(* Zeichenkette *)*
weekDay	*(* sun ... sat *)*
yearDay	*(* [0 .. 366] *)*
dst	*(* [0 .. 1] *)*

Dies stellt einen Zeitpunkt einer bestimmten Zeitzone nach dem gregorianischen Kalender dar. Die Einschränkungen der Werte werden in der Implementation berücksichtigt.

Das Feld *offset* spezifiziert die Differenz zwischen der lokalen Zeit und der Zeit in der Zone UTC. Diese Differenz wird in Sekunden angegeben mit positiven Werten für die westlich von UTC liegenden Zeitzonen.

Das Feld *zone* gibt den Namen für die Zeitzone, in der dieses Datum gilt, meist in einer Abkürzung an. Beispielsweise steht "PDT" für *Pacific Daylight Time*.

weekDay spezifiziert den Wochentag durch eine Zahl 0 ... 6, welche die Tage *Sonntag* ... *Samstag* repräsentieren.

yearDay spezifiziert den Tag des Jahres bei einer Durchnumerierung von 0 ... 366.

Das *dst*-Feld gibt die Anzahl der Stunden an, welche die Sommerzeit von der ursprünglichen Zeit abweicht.

Zusätzlich zu den Operationen zum Kreieren und Konvertieren eines Datums werden zur Zeit folgende Operationen angeboten:

localZone, UTCZone geben die Information über die Zeitzone des lokalen Hosts und von UTC.

dayOfWeek, dayOfYear geben den Wochentag bzw. den Tag im Jahr an.

leapYear gibt an, ob das Datum in einem Schaltjahr liegt.

addDays addiert eine Anzahl Tage (auch negativ) zu einem Datum und gibt das so errechnete Datum aus.

daysBetween gibt die Differenz (positiv oder negativ) in Tagen zwischen zwei Daten aus.

Im Vergleich zu den vorherigen Abschnitten ergibt sich daraus eine Anzahl von Forderungen an eine neue Bibliothek für Tycoon. Die Trennung der maschinenabhängigen Implementation von den portablen Schnittstellen wie in Modula-3 ist nicht vorhanden. Ebenso ist diese Realisierung in TLnicht universell zur Darstellung von Zeitpunkten und gleichfalls Zeitintervallen geeignet, welches in SQL-92 implementiert ist. Dabei müssen die Konvertierungsmöglichkeiten von einem Datumstyp zu einem anderen festgehalten werden. Eine Neuorganisation dieser Schnittstellen und Implementationsmodule wird in Kapitel 5.1.2 vorgestellt.

3.3.4 Das Konzept der Zusicherungen

Qualitätsfaktoren für Software sind Wiederverwendbarkeit, Erweiterbarkeit und Kompatibilität. Sie dürfen nicht auf Kosten der Korrektheit und Robustheit von Programmen erreicht werden. Die Korrektheit eines Programmes besagt, wie weit dieses Programm seine Spezifikation erfüllt. Obwohl es kein vollständiges Konzept zum Erreichen von Korrektheit gibt, sollte ein Programmierer versuchen, die Diskrepanz zwischen seiner Software-Implementation und der Spezifikation so weit wie möglich zu reduzieren. Zu diesem Zweck kann zu einem Programmfragment ein Ausdruck assoziiert werden, der angibt, unter welchen Annahmen dieses Fragment korrekt arbeitet und welche Aussagen nach Abarbeitung des Fragments gelten. Hierdurch wird ein Spezifikationskonstrukt in die Implementation eingeschlossen. In Eiffel werden derartige Konstrukte *Zusicherungen* genannt.

Der Begriff der *Zusicherung* (engl. *assertion*) stammt aus den klassischen Werken von Floyd [Floyd 67], Hoare [Hoare 69] und Dijkstra [Dijkstra 76], welche Pionierarbeit zum Thema *Korrektheit von Programmen* geleistet haben. Bisher haben nur wenige Sprachen dieses Konzept aufgenommen, u.a. Alphard [Shaw 81] und Euclid [Lampson et al. 77], die speziell zur Entwicklung von nachweisbar korrekten Programmen entworfen wurden. Der Begriff der Klasseninvarianten stammt aus dem Werk von Hoare über Datentypinvarianten [Hoare 72]. Als eine der ersten objektorientierten Sprachen hat Eiffel das Konzept von Zusicherungen in seinen Sprachumfang aufgenommen [Meyer 88].

Korrektheit von Programmen

Da es kein vollständig korrektes Programm gibt, kann die Korrektheit nur in Beziehung auf eine bestimmte Spezifikation erreicht werden. In [Meyer 90c] werden zwei Arten der Korrektheit unterschieden:

Vollständige Korrektheit Ein Programmfragment a ist vollständig korrekt für die Vorbedingung P und die Nachbedingung Q, genau dann, wenn gilt: Wenn a in einem Zustand, der P erfüllt, ausgeführt wird, dann terminiert die Ausführung und der Zustand nach der Ausführung erfüllt Q.

Partielle Korrektheit Ein Programmfragment a ist partiell korrekt für die Vorbedingung P und die Nachbedingung Q, genau dann, wenn gilt: Wenn a in einem Zustand, der P

erfüllt, ausgeführt wird und diese Ausführung terminiert, dann erfüllt der Zustand nach der Ausführung Q.

Die partielle Korrektheit ist eine bedingte Korrektheit, die für ein Programm nur geprüft werden muß, wenn das Programm terminiert. Im folgenden wird daher unter *Korrektheit* eines Programmfragments dessen vollständige Korrektheit hinsichtlich der Zusicherungen verstanden.

Strikte statische Typisierung dient der Korrektheit, aber sie reicht für eine vollständige Überprüfung der Korrektheit nicht aus. Klassen in Eiffel sind Implementationen von abstrakten Datentypen. Sie sind Ansammlungen von Attributen und Routinen, welche die Funktionen der Spezifikation des abstrakten Datentyps (ADT) repräsentieren. Gleichfalls müssen aber auch die semantischen Eigenschaften, die durch Axiome und Vorbedingungen ausgedrückt werden, als Grundlage für die Eigenschaften der Instanzen des Typs berücksichtigt werden. Im allgemeinen Klassenkonzept hat man diesen Aspekt des Konzepts der ADTs aus den Augen verloren [Meyer 88]. In Eiffel realisieren Zusicherungen und die damit verbundenen Konzepte die Implementation der semantischen Eigenschaften von ADTs. Damit kann zwar eine vollständige Korrektheit nicht erreicht werden, aber die Überprüfung der Zusicherungen erreicht mehr Sicherheit der Anwendungsprogramme als die statische Typüberprüfung allein.

Zusicherungen stellen dem Programmierer grundlegende Werkzeuge bereit, um die Argumente für Korrektheit ausdrücken und validieren zu können. Diese Werkzeuge sind eine Hilfe beim Testen und zur Fehlerbeseitigung. Sind die Programme auf diese Art erst einmal als korrekt bewiesen, wird die Neigung des Anwendungsprogrammierers unterstützt, diese Programme wiederzuverwenden. Zusicherungen vermindern das Risiko der unerwarteten und unnormalen Ausführung einer Routine, wodurch das Vertrauen der Programmierer auch in umfangreiche Programmsysteme verstärkt wird.

Programmieren durch Vertrag

Ein Schlüsselbegriff für Zusicherungen in Eiffel ist das **Programmieren durch Vertrag** (engl. *programming by contract*). Dabei wird die Beziehung zwischen einer Klasse und dem Benutzer dieser Klasse als formelle Vereinbarung angesehen, welche die Rechte und Pflichten jedes Partners festhält. Für dieses Vertragsmodell ist die Klasse einer Bibliothek der Lieferant von Diensten und das Anwendungsprogramm, welches diese Klasse benutzt, der Kunde. Nur durch die genaue Definition der Ansprüche und Verantwortlichkeiten jedes Moduls werden die Zuständigkeiten in großen Software-Systemen übersichtlich gehalten und das Vertrauen auf ihre Funktionstüchtigkeit erworben. In diesem Zusammenhang wird in Eiffel auch die Ausnahmebehandlung bei Laufzeitfehlern genannt.

Die Zusicherungen werden zur Spezifikation von Routinen benötigt. Als Hilfe im Entwurf und zum späteren Verständnis muß die Aufgabe einer Routine klar umrissen werden. Dazu können *Vorbedingungen* und *Nachbedingungen* für jede Routine aufgestellt werden. Eine Vorbedingung drückt die Eigenschaften aus, die beim Aufruf erfüllt sein müssen. Sie gibt die Bedingungen an, unter denen ein Aufruf der Routine korrekt ist. Eine Nachbedingung beschreibt diejenigen Eigenschaften, die von der Routine nach ihrer Beendigung garantiert werden. Hierdurch wird die Situation nach der erfolgreichen Ausführung charakterisiert. Beide Bedingungen beziehen sich im allgemeinen auf eine Routine, im Falle von Eiffel ist dies die Methode einer Klasse.

Zusätzlich können für jede Klasse *Invarianten* aufgestellt werden. Dies sind Bedingungen, die zu stabilen Zeitpunkten immer für diese Klasse gelten. Sie müssen vor und nach

Ausführung jeder Methode dieser Klasse garantiert werden. Während der Ausführung einer Methode darf eine Zusicherung kurzzeitig verletzt sein. Die Prüfzeitpunkte liegen jeweils vor Beginn und nach Ende einer Methodenausführung.

Nach dem Vertragsmodell verpflichtet die Vorbedingung die Umgebung, ihre Annahme sicherzustellen. Die Nachbedingung verpflichtet die Routine selbst, diese Bedingung nach Ausführung zu erfüllen. Vorbedingungen und Invarianten werden vor Beginn einer Routine geprüft, während Nachbedingungen und Invarianten nach Beendigung einer Routine geprüft werden. Die Verletzung einer Zusicherung zu Beginn oder am Ende der Routine löst eine Ausnahme aus.

Für jede Routine können strenge semantische Eigenschaften festgestellt werden, die unabhängig von einer speziellen Repräsentation sind. Beispiele sind die Routinen *pop, push* und *top* einer generischen Klasse *Stack*. *pop* und *top* können nur ausgeführt werden, wenn schon mindestens ein Element im Stack vorhanden ist. Nach Ausführung von *push* ist die Anzahl der Elemente um eins erhöht worden und nach *pop* ist die Elementanzahl um eins geringer als vor ihrer Ausführung. In der Tabelle 3.3.4 werden die Verpflichtungen und Nutzen der Partner des Vertrages am Beispiel der Operation *push* aufgeführt.

	Verpflichtung	Nutzen
Kundenprogrammierer	*push(x)* darf nur für einen nicht-vollen Stack aufgerufen werden.	x wird als neuestes, oberstes Element des Stacks geführt, d.h. nach Beendigung gibt *top()* das Element x zurück. Die Variable *nb_elements* wird um 1 erhöht.
Modulimplementierer	Der Implementierer versichert, daß x an die oberste Stelle des Stacks geschoben wird. Der Stack ist nach Beendigung von *push(x)* nicht leer.	Er muß keine Fälle behandeln, in denen der Stack schon voll ist.

Tabelle 3.3.4: Rechte und Pflichten von Kunde und Lieferant im Vertragsmodell

Vor- und Nachbedingungen dienen dazu, die semantischen Eigenschaften explizit zu machen. Sie werden als Klauseln in die Routinendeklarationen eingeführt. In Eiffel sind die Schlüsselworte ***require*** und ***ensure*** zu diesem Zweck definiert. ***require*** bezeichnet die Vorbedingung und steht *vor* der Implementation des Routinenkörpers. Mit ***ensure*** werden die Nachbedingungen eingeleitet, die *nach* dem Routinenkörper folgen und vor dem ***end*** der Routine stehen. Das Schlüsselwort ***old*** verweist auf den Zustand der nachfolgenden Variable *vor* Beginn dieser Routine. Invarianten, Vor- oder Nachbedingungen enthalten einen boolschen Ausdruck pro Zeile. Enthält eine Zusicherungsklausel mehrere Bedingungen, so werden sie während der Übersetzung konjunktiv verknüpft.

```
class STACK
...
push(x :T) is
        -- Add x on top
```

```
    require
      not full
        -- i.e. nb_elements < array_size in this representation
    do
      nb_elements := nb_elements + 1;
      implementation.enter(nb_elements, x)
    ensure
      not empty;
      top = x;
      nb_elements = old nb_elements + 1
    end;
...
  invariant
    empty = (nb_elements = 0)
end;
```

Dokumentation

Für die Organisation der Schnittstellen bei umfangreichen Software-Systemen ist die Benutzung der Zusicherungen grundlegend. Die Vor- und Nachbedingungen sowie die Invarianten stellen eine kompakte und präzise Form dar, mit der mögliche Benutzer einer Bibliothek Information über die in den Modulen angebotenen Methoden und deren Verwendung erhalten. Zusätzlich kann das Dokumentationswerkzeug *short* von Eiffel (siehe Abschnitt 3.2.2) die nötige Information zur Wiederverwendung aus den Klassen der Bibliothek extrahieren.

Fehlererkennung

Zur Laufzeit können Zusicherungen als Werkzeug zur Fehlererkennung nützlich sein. Für den Eiffel-Compiler existieren verschiedene Optionen zur Prüfung von Zusicherungen, die der Programmierer für seine Übersetzung schalten kann. Es können

- keine Zusicherungen,
- nur Vorbedingungen oder
- alle Zusicherungen inklusive der Invarianten

geprüft werden. Dabei ist es sinnvoll, sämtliche Bedingungen einschließlich der Nachbedingungen für Bibliotheksmodule nur solange zu prüfen, bis die Bibliothek validiert ist, da die vollständige Prüfung sehr zeitintensiv ist. Der Programmierer einer Anwendung muß nach dem *Programmieren durch Vertrag* nur dafür Sorge tragen, daß er die Vorbedingungen der benutzten Bibliotheksmodule erfüllt. Daher reicht es für das Testen der Korrektheit seines Programms aus, wenn nur die Prüfung der Vorbedingungen eingeschaltet ist.

Dieser Schalter zur Prüfung von Zusicherungen ist eine Voraussetzung für die Erstellung fehlertoleranter Systeme und zur Fehlererholung. Die Eiffel-Klauseln ***rescue*** und ***retry*** ermöglichen eine Toleranz gegenüber den auftretenden Fehlern, indem das Programm die fehlerhaften Anweisungen durch ***retry*** wiederholt ausführt, nachdem die mit ***rescue*** spezifizierten Rettungsmaßnahmen durchgeführt wurden. Auf diese Konzepte wird hier nicht näher

eingegangen, weil sie im Hinblick auf die Tycoon-Bibliotheken keine interessanten Erkenntnisse bringen. Eine ausführliche Beschreibung dieser Eiffel-Konzepte findet sich in [Meyer 88] und [Mandrioli, Meyer 92].

3.4 Zusammenfassung der Anforderungen an Tycoon

Für die Typen des Tycoon Systems sollen möglichst viele abstrakte Datentypen gebildet werden, um eine einheitliche Definition der Typen zu erreichen. Der Nachteil, daß Vererbung nur bedingt in Tycoon nachzubilden ist, wird auf diesem Weg ausgeglichen. Wünschenswert wäre eine Abstraktion der numerischen Operatoren in einem eigenen Modul, vergleichbar mit der abstrakten Klasse *NUMERIC* in Eiffel. Da die numerischen Basisdatentypen wie z. B. *Int* jedoch mit ihren grundlegenden Operationen fest in TL eingebaut sind, ist es nur auf Umwegen erreichbar, eine weitere Ebene mit numerischen Operationen zwischen diesen Typen und der Sprache einzuführen. Um den Sprachkern möglichst einfach strukutriert zu halten, wird in dieser Arbeit kein Realisierungskonzept für eine numerische Abstraktion vorgeschlagen.

Die in Tycoon vorhandenen Konzepte der Modularisierung und Subtypisierung sind für die Bibliotheken grundlegend. Die strikte Typisierung ermöglicht dem Anwendungsprogrammierer eine strenge Kontrolle über die Beziehungen innerhlab seiner Applikation.

Nicht zu verbessern sind die ebenfalls in anderen Sprachen realisierten Konzepte von Iteratoren in der umfangreichen Schnittstelle *Iter* in TL.

Die Forderung bezüglich der Datums- und Zeitdarstellung in TL betrifft den Komfort der Datentypen. Die Realisierung von Zeitpunkten und gleichfalls Zeitintervallen, sowie die Separierung des maschinenabhängigen Implementationsteils muß durch eine Neuorganisation dieser Schnittstellen durchgeführt werden. In Modula-3 wird im Gegensatz zu SQL, welches die Anzahl der darstellbaren Stellen einer Zeitangabe von der Plattform abhängig macht, eine maschinenunabhängige Darstellung verwendet, so daß die Repräsentation von Modula-3 als Vorbild für die reorganisierten Tycoon-Bibliotheken verwendet werden soll.

Im Rahmen des *Programmierens durch Vertrag* werden in Eiffel die Konzepte von Vor- und Nachbedingungen, Invarianten, ***old***-Klausel, Ausnahmebehandlung und Compiler-Optionen zum individuellen Prüfen der Zusicherungen eingesetzt.

In Tycoon werden Zusicherungen als ein neues Konzept eingeführt, um die Prüfbarkeit der Bibliotheksmodule zu erhöhen. Die Vorteile von Zusicherungen liegen in den vier Anwendungen [Meyer 88]:

- Hilfe beim Erstellen korrekter Software
- Hilfe bei der Dokumenation
- Werkzeug zur Fehlererkennung und -beseitigung
- Unterstützung für fehlertolerante Software

Mit getesteten Modulen wird das Vertrauen der Programmierer in die Bibliothek höher und dadurch die Anzahl der Wiederverwendungen gesteigert. In Abschnitt 5.3 wird erläutert, welche dieser von Eiffel bekannten Konzepte in Tycoon implementiert werden können.

Kapitel 4

Bibliotheken im Vergleich

Bibliotheken werden eingeführt, um Programmteile, welche meist in Form von Modulen abgelegt sind, wiederverwenden zu können. Dabei wird die interne von der externen Wiederverwendung unterschieden. In objektorientierten Sprachen entsprechen die wiederverwendbaren Module den Klassen.

Die Module werden nach verschiedenen Kriterien in getrennte Bibliotheken sortiert. Als Klassifizierungskriterien gelten nach [Pomberger, Blaschek 93]:

- Herkunft (Projekt, Firma, Programmiersprache, Programmierer)
- Entwicklungsstand (Delta, Alpha, Beta)
- Verwendungszweck wie Standardmodule, Datenstrukturen, Betriebssystemmodule, Benutzerschnittstellenmodule, grafische Module, anwendungsspezifische Module (Mathematik, Simulationen, kaufmännische Aufgaben)

In den meisten Programmiersprachen werden die Module nach ihrem Verwendungszweck in Bibliotheken gruppiert. Welche Kriterien zur Bildung einer derartigen Bibliothek herangezogen werden sollten, ist Gegenstand einiger Publikationen, z. B. [Möller, Müller 93; Coggins 90; Cohen et al. 92]. Innerhalb einer Bibliothek sind die Module somit meist stark voneinander abhängig. Alle Module, die demselben Verwendungszweck dienen, stellen jeweils einen funktionalen Block dar. Die Standardmodule und Datenstrukturmodule gehören zu den grundlegenden Modulen, auf denen andere Module aufbauen. Daraus ergeben sich die Nachteile, daß Teile einer Bibliothek nicht allein verwendet werden können. In Sprachen mit globalem Namensraum kann die Benutzung von zwei verschiedenen Bibliotheken für ein Anwendungsprogramm schnell zu Namenskollisionen führen, wenn beide Bibliotheken dieselbe Klasse enthalten. Derartige Mehrdeutigkeiten können für diese Art Sprachen gelöst werden, wenn jede Bibliothek Präfixe für globale Namen oder *name spaces* wie in C++ benutzen würde.

Für die Auswahl einer Bibliothek sind eine Reihe von Designkriterien von wesentlicher Bedeutung, da sie sich unmittelbar auf den Entwurf und die Struktur des Anwendungsprogramms auswirken. Das geeignete Bibliotheksdesign richtet sich nach deren Einsatzbereich und den dort geltenden Randbedingungen. So steht bei numerischen Anwendungen die Laufzeiteffizienz im Vordergrund, während für grafische Benutzeroberflächen die Portierbarkeit von Bedeutung ist. Ein generelles Design, welches allen Anforderungen entspricht, gibt es nicht [Stal, Berger 93a].

Daher muß jeder Bibliotheksbenutzer seine Randbedingungen bestimmen und danach die geeignete Bibliothek auswählen. Es muß beachtet werden, daß umfassende, festgefügte Hierarchien als Bibliotheken zusätzliche Zeiten beim Laden, Übersetzen, Binden und Ausführen des Anwendungsprogramms verursachen. Außerdem wird eine Hierarchie mit vielen, großen Klassen für den Anwender unübersichtlich. Eine Bibliothek mit kleinen, übersichtlichen Klassen unterstützt das Bottom-Up-Design und erleichtert den Entwurf auf der Anwendungsebene. Der Bibliotheksbenutzer kann die Klassenhierarchie in diesem Fall besser seinen Bedürfnissen anpassen und die Funktionalität ausschöpfen. Den unterschiedlichen Wünschen der Benutzer nach Optimierung des Speicherbedarfs für Code und Daten, der Geschwindigkeit und der Fehlererholung muß Rechnung getragen werden, auch wenn diese Ziele nicht immer gleichzeitig erreicht werden können.

Für den Entwurf von Bibliotheken muß der mögliche Anwendungsbereich auch miteinbezogen werden, um die erwünschten Randbedingungen zu erfassen. Ebenso muß der Entwickler einer Bibliothek die Konzepte der Programmiersprache, für welche die Bibliothek entworfen werden soll, berücksichtigen. Die wichtigsten Prinzipien bei der Erstellung von Bibliotheken betreffen darüber hinaus die folgenden Fragen:

- Was macht Bibliotheken speziell im Vergleich zu Anwendungsprogrammen?
- Wie bildet man korrekte und robuste Bibliotheken und nutzt ihre angebotenen Schnittstellen entsprechend der angegebenen Regeln?
- Wie dokumentiert man Klassen bzw. Module?
- Wie groß sollte eine Klasse bzw. ein Modul sein?
- Wie benennt man Operationen und Klassen bzw. Module?
- Welche weiteren Vereinbarungen werden getroffen, z. B. die Wahl von Kategorien für Operationen und Richtlinien für das Indizieren von Strukturen?
- Wie wird eine durchschaubare Vererbungsstruktur entworfen?

Im folgenden soll an Beispielen untersucht werden, welche Konzepte und Methoden in vorhandenen Programmiersprachen angewendet werden, um die vorgestellten Prinzipien zu unterstützen.

In der objektorientierten Programmierung sind Klassen ein grundlegender Bestandteil von Programmiersprachen. Alle Anwendungsprogramme setzen sich aus Klassen zusammen. Die Konzepte von Klassenbibliotheken haben einen Einfluß auf das Denkmodell des Programmierers [Pomberger, Blaschek 93] und bilden ein weiteres Abstraktionsniveau über der Programmiersprache.

Das Konzept der Vererbung für objektorientierte Sprachen läßt die Klassen einer Bibliothek in einer verzweigten Struktur erscheinen. Es werden dabei im wesentlichen zwei verschiedene Architekturen unterschieden. Wenn alle Klassen direkt oder indirekt eine gemeinsame Basisklasse besitzen, entsteht eine komplexe Baumhierarchie. Das Gegenteil dazu ist ein Klassenwald, in welchem mehrere Wurzelklassen existieren und somit voneinander unabhängige Klassen enthalten sein können. Ein Klassenwald ist gerade für die fundamentalen Klassen effizienter, da in vielen Anwendungen nur eine begrenzte Auswahl an Klassen benötigt wird. Wiederum andere Anwendungen, z. B. grafische Benutzeroberflächen, beinhalten einen abgegrenzten Aufgabenbereich und lassen einen Klassenbaum günstig erscheinen. Aus diesem

Grund bestehen einige Bibliotheken in ihrer Gesamtheit aus einem Wald mit lose gekoppelten Baumhierarchien. Einige Gründe für die Strukturierung einer Klassenbibliothek als Baum oder als Wald werden in Tabelle 4.1 [Lea 88] angegeben.

Die genannten Gründe für und wider der Baum- und Waldhierarchie sind auch in nichtobjektorientierten Sprachen anwendbar, indem mit Hilfe der Subtypisierung und Wiederverwendung von Code Bibliothekshierarchien erstellt werden. Ein Baum hat die Vorteile, daß der Programmcode vererbt wird und damit nicht für jede Datentypstruktur neu geschrieben werden muß. Es wird damit eine automatische Namenskonvention bezüglich der Operationen erreicht. Ein Wald bringt dem Bibliotheksimplementierer größere Freiheit, indem sich eine neu implementierte Datenstruktur auf eine ausgewählte andere Datenstruktur beziehen kann, ohne notwendigerweise eine gesamte Subtyphierarchie von einer Wurzel her zu importieren. Teilbäume sind wünschenswertes Ziel der Implementierung, da sie überschaubarer für den Anwendungsprogrammierer sind.

Da Tycoon keine Vererbung enthält, muß versucht werden, die Bibliotheken mit Hilfe von Subtypisierung und Namenskonventionen zu strukturieren. Die Modula-3-Bibliothek liefert eine durchdachte Struktur auf der Grundlage einer modularen Sprache und ist in diesem Bereich mit Tycoon vergleichbar. Die Smalltalk-Bibliothek wird als eine der ersten hierarchischen Strukturen als Beispiel aufgeführt. C++ als hybride Sprache liefert die Grundlage für verschieden aufgebaute Bibliotheken, die teilweise mit umfassenden Konzepten die Sprache ergänzen. Wegen der großen Verbreitung von C++ und dessen Bibliotheken ist die Namenskonvention vielen Anwendern geläufig und sollte auch für Tycoon berücksichtigt werden. Die ebenfalls objektorientierte Sprache Eiffel wird besonders wegen der klar durchdachten Bibliotheksstruktur in ihrer Eignung für eine Übertragung auf Tycoon geprüft.

In den nächsten Abschnitten erfolgt die Vorstellung dieser Bibliotheken. Ein Hauptaugenmerk wird dabei auf die Struktur der Bibliothek gelegt. Soweit es aus der vorhandenen Literatur hervorgeht, wird dargestellt, in welchem Rahmen die jeweilige Bibliothek hierarchisch gegliedert, erweiterbar und portierbar ist. Ihr Umfang und der Bereich von möglichen Anwendungen wird untersucht. Auch die Generizität ist ein entscheidendes Merkmal, welches angeführt wird. Der Aspekt der Dokumentation wird betrachtet, um eine Hinweis auf Namenskonventionen zu erhalten. Die Möglichkeiten einer Fehlerbehandlung spielen eine besondere Rolle in Bibliotheken. Der Benutzer von Bibliotheksmodulen muß die Gewissheit haben, daß die aufgerufene Operation nicht unerwartet abbricht und er die Möglichkeit erhält, seine Anwendung trotz eines Fehlers ordnungsgemäß zu beenden.

Die Art der benutzten Vererbung ist insbesondere für objektorientierte Sprachen von Bedeutung. Je nach Zweck der Bibliothek wird weiterhin geprüft, welche Basisdatentypen vorhanden sind und in welcher Weise sie in die Sprache eingebunden werden. Im Hinblick auf die Problemstellung dieser Arbeit werden möglichst die Implementationen der Basisdatentypen für einen Vergleich herangezogen. Sie sollen als Beispiel für die Auswahl der implementierten Methoden und deren Namenskonvention in den überarbeiteten Bibliotheken für das Tycoon System dienen.

Nicht alle genannten Aspekte sind für sämtliche Bibliotheken bewertbar, da bei einigen die diesbezügliche Information fehlt oder die Sprache vom Umfang her das entsprechende Konzept nicht anbietet. Soweit es möglich ist, wird zum Abschluß ein vergleichender Überblick über alle erwähnten Bibliotheken gegeben.

	Klassenbaum	Klassenwald
Konsistenz und Flexibilität	+/− Ein konsistentes Programmiersystem, welches Entscheidungen über Klassenstruktur, Interkommunikation, Vererbungskonventionen etc. getroffen hat, ist einfacher zu verwenden, aber bietet weniger Flexibilität.	+ Die Bibliothek ist flexibler, weil die Integration von neuen Klassen, die von verschiedenen Programmierern entworfen wurden, direkt geschehen kann.
	+ Mechanismen wie z. B. die Ausnahmebehandlung werden auf einfache Weise durch Vererbung für alle Klassen eingebunden, indem die Basiskonzepte in der Wurzel des Baumes vereinbart werden.	− Die Einbindung eines einheitlichen Konzeptes kann lediglich durch Einhaltung von Vorgaben erreicht werden.
	+ Inkonsistenzen, zum Beispiel durch Namensgleichheit, werden nahezu ausgeschlossen.	− Die Möglichkeit von Inkonsistenzen zwischen den Klassen ist wegen ihrer uneinheitlichen Darstellung in der Bibliothek größer.
	+/− Eine konsistente Hierarchie strukturiert die Klassen in einer Weise, die nicht für alle Anwendungen deren Abhängigkeiten geeignet reflektiert. Beim Einfügen einer neuen Klasse muß der Bibliotheksprogrammierer entscheiden, an welche Stelle im Baum sie am besten paßt. Eine Baumstruktur eignet sich am besten für Systeme, in denen jede erbende Klasse Funktionalität zufügt oder eine Spezialisierung darstellt.	+ Ohne festgelegte Hierarchie können Abhängigkeiten in Anwendungen besser nachgebildet werden. Der Entwurf der Bibliothek sollte dabei vorsehen, daß unabhängige Klassen vom Anwendungsprogrammierer in größere Bäume oder Wälder eingebunden werden können.
Effizienz und Redundanz	− Für eine Klasse unnötige Methoden müssen in der Hierarchie weitergegeben werden, um die Konventionen der gesamten Hierarchie zu enthalten. Dies führt zu geringerer Effizienz, weil die Übersetzungs-, Lade- und Laufzeiten durch die größere Menge Code erhöht werden [Coggins 90].	+ Klassen werden nach fundamentalen Prinzipien entworfen und können daher effizienter und redundanter sein.
	+/− Für eine Baumhierarchie müssen virtuelle Funktionen eingefügt werden, welche benutzerdefinierte Subklassen erlauben. Dies verringert die Laufzeiteffizienz.	+/− Die Wiederverwendung ist eingeschränkt, wodurch wiederum Redundanz erzeugt wird. Der Bibliotheksprogrammierer muß entscheiden, ob Methoden virtuell benutzt werden, damit eine benutzerdefinierte Ableitung dieser Klasse erstellt werden kann.
Sprachabhängige Einschränkungen	− Sprachen mit eingebauten Datentypen unterstützen den Klassenbaum nicht, weil diese Typen keine Klassen sind und damit aus der Hierarchie herausfallen.	+/− Ein Klassenwald stellt weitere Datentypen auf die gleiche Ebene wie die eingebauten Basistypen.
	+/− Parametrisierung für Subklassenbildung kann durch Definition einer allgemeinen Klasse, z. B. Stack, auf den Elementen der Wurzelklasse *Object* erzeugt werden.	− In einem Klassenwald muß bei fehlendem parametrisierenden Klassengenerierungsmechanismus auf pseudogenerische Methoden zum Klassenentwurf zurückgegriffen werden. Diese sind teilweise ineffizient und können einige Klassenbeziehungen nicht ausdrücken.

Tabelle 4.1: Vor- und Nachteile einer Bibliothek in Baum- oder Waldstruktur

4.1 Modula-3

Modula-3 trennt als grundlegendes Konzept die Schnittstelle von der Implementation. Hierbei wird, ebenso wie in Ada, nicht die Vererbung als Mechanismus zur Bildung von Implementationen benutzt. Die hier zum Vergleich herangezogene Menge von Schnittstellen hat sich aufgrund der mehrjährigen Erfahrung von Programmierern als nützlich erwiesen. Sie decken alle grundlegenden Datenstrukturen ab und sind daher für alle Anwendungen nützlich. Der Umfang der Schnittstellen wird in [Horning et al. 93] beschrieben. Darüber hinaus werden die Schnittstellen und darin definierte Operationen im Quelltext dokumentiert.

Die Menge an Schnittstellen kann als Sammlung von Bibliotheken angesehen werden und enthält u. a. einige generische Schnittstellen. Diese Bibliotheken können vom Anwendungsprogrammierer beliebig um weitere Schnittstellen und Module erweitert werden. In den Vergleich werden insbesondere die Standardschnittstellen einbezogen, welche einige grundlegende Datentypen enthalten, die nicht in der Sprache implementiert sind, z. B. *Real* und *Word*. In [Horning et al. 93] sind nur leichte Modifikationen der in [Nelson 91] beschriebenen Standardschnittstellen vorgenommen worden.

Da Modula von der ersten Version an maschinenunabhängig entworfen wurde [Wirth 88], indem alle maschinenabhängigen Eigenschaften in eigenen Modulen gehalten werden, sind auch die Schnittstellenbibliotheken mit wenig Aufwand portierbar.

Die Einteilung der Schnittstellen erfolgt in sechs Gruppen, die als eigene Bibliotheken aufgefaßt werden können. Diese Gruppen beinhalten jeweils Schnittstellen zu einem bestimmten Verwendungszweck. Dieses sind:

Standard Interfaces: Standardschnittstellen

Text	Operationen auf Zeichenketten
Thread	Synchronisationsprimitive für vielfache Kontrollthreads
Word	Operationen auf Word (ohne Vorzeichen)
Real, *LongReal*, *Extended*	Eigenschaften dieser Floating Point-Typen
RealFloat, *LongRealFloat*, *ExtendedFloat*	numerische Operationen für Floating Point
FloatMode	Operationen zum Testen des Verhaltens der Implementation in Bezug auf numerische Bedingungen
Lex	Parsen von Zahlen und anderen Daten von einem Eingabestrom
Fmt	Textformatierung von Zahlen und anderen Daten in den Typ *Word*

Die Schnittstellen *Lex* zum Lesen vom Eingabe-Stream und *Fmt* zur Formatierung aller Typen sind nicht unbedingt für jede Anwendung nötig, stehen aber in enger Beziehung zu diesen Standardschnittstellen.

Data Structures: Behälterdatentypen

Sequence	generische Schnittstelle von Folgen, Zufügen ist an beiden Enden möglich
Atom	atomares Symbol wie in Lisp
List, *ListSort*	Operationen auf verketteten Listen und dazu passenden Sortieralgorithmen, generisch
Sx	symbolischer Ausdruck (*symbolic expression*), der als rekursive, verkettete Liste mit Operationen zum Lesen, Schreiben und Manipulieren repräsentiert wird
Table	partielle Abbildung, die Änderung und Iteration unterstützt, generisch
SortedTable	generische Schnittstelle mit partieller Abbildung auf eine total geordnete Menge

Algorithms: Nützliche Algorithmen

ArraySort	Sortieren eines Arrays, generisch
Random	Pseudo-Zufallszahlengenerator
Fingerprint	64-Bit-Prüfsumme zum *fingerprint* von Zeichenketten und generelleren Datenstrukturen

Fingerprint bietet eine sichere Prüfsumme, die von Instanzen des Typs *Text* oder *Array* gebildet wird. Sie können zu einem neuen *Fingerprint* kombiniert werden. Ein Hash-Code kann ebenfalls von einem *Fingerprint* gebildet werden.

I/O Streams: *IO*, *Wr*, *Rd*, *Stdio* und andere, spezielle Ein-/Ausgabeklassen wie *FileRd*, *FileWr*, *TextRd*, *TextWr*

Modula-3 nutzt als Methode zur Datenein- und -ausgabe spezialisierte Prozeduren, die nicht notwendigerweise Teil der Sprache sein müssen.

Operating System: *Time*, *Date*, *Tick*, *OSError*, *File*, *Pipe*, *Process*, *Env* u.a. betriebssystemabhängige Module

Runtime: *WeakRef*, *RTType*, *RTAllocator* u.a. Laufzeitmodule

Generische Schnittstellen, wie z. B. *Sequence*, müssen die Prozeduren *Equal*, *Compare* und *Hash* enthalten. Instanzen über die Typen *Atom*, *Integer*, *Refany* und *Word* sind von diesen generischen Schnittstellen, soweit sie nicht bedeutungslos sind, standardmäßig vorhanden. *Equal* stellt die Äquivalenzrelation zur Verfügung, während *Compare* zwei Werte vergleicht, falls eine natürliche, totale Ordnung auf dem Typ definiert ist. Die *Hash*-Funktion muß zeitunabhängig sein, die Ordnung der *Equal*-Funktion erhalten und eine gleichmäßige Verteilung über den Wertebereich von *WORD* liefern.

Alle Schnittstellen verbergen für jede Datenstruktur die zugehörigen Operationsimplementationen. Dazu gehören auch die Einfüge- und Löschoperationen, die speziell für die jeweilige Datenstruktur gelten. Die Namen der Operationen sind nicht vereinheitlicht, so daß jede Schnittstelle eigene Namen verwendet.

4.2 Eiffel

Die ausführliche Darstellung aller Bibliotheksklassen findet sich in [Meyer 90b; Meyer 94]. Im Quelltext der Bibliotheken sind sämtliche Eigenschaften, die zur Wiederverwendung einer Klasse beitragen, verzeichnet.

Eiffel bietet eine Ausnahmebehandlung an, die jeweils bei Verletzung einer Zusicherung aufgerufen wird. Nach Auftreten der Ausnahme wird die Ausführung der fehlerhaften Routine unterbrochen, um daraufhin, je nach Art der Ausnahme, die Ablaufkontrolle an die aufrufende Routine weiterzugeben oder eine programmierte, alternative Strategie der fehlerhaften Routine auszuführen.

Der Sprachkern von Eiffel ist minimal gehalten und umfaßt die wichtigsten Konzepte wie Vererbung, Generizität und Schleifenkonstrukte. Grundsätzlich sind alle Datentypen dieser objektorientierten Sprache in Bibliotheken organisiert, so daß auch die grundlegenden Typen *integer*, *real*, *boolean* und *character* in der Kernbibliothek zu finden sind. Daher gehört die Basisbibliothek zum Lieferumfang der Sprache.

Um eine derart umfangreiche Bibliothek zu verwalten, benötigt man ein Schema zur Klassifizierung. In modulorientierten Sprachen, wie z. B. Ada, fehlt ein solches Konzept. In Eiffel ist der grundlegende Klassifizierungsmechanismus die Vererbung, welche eine Subtyphierarchie aufbaut. Als Mechanismus zur Wiederverwendung dient daneben die Kunden-Relation. Diese Unterscheidung ist nach [Snyd86] nötig, um bei einer Änderung der Implementation eines Objekts die Kunden dieses Objekts nicht zu beeinflussen. Ein Kunde verwendet lediglich die Spezifikation einer Klasse. Die Methoden dieser Klasse werden über die offizielle Schnittstelle benutzt. Dagegen hat ein Erbe Zugriff auf die Implementation. Alle Klasseneigenschaften können verwendet werden, um sie an einen speziellen Kontext anzupassen.

Eine Vererbungshierarchie ist in Eiffel durch Generalisierung erreicht worden. Diese Generalisierung erfolgt in einem fortwährenden Prozeß, der durch wiederholtes Abstrahieren und Extrahieren eine Vererbungshierarchie aufbaut. Durch Abstraktion werden Klassen entworfen, die noch nicht die Implementation enthalten und so als allgemeines Muster für konkrete Klassen zur Verfügung stehen. Durch Extrahieren von Gemeinsamkeiten in ähnlichen Komponenten werden abstrakte Klassen erreicht, von denen die konkreten Klassen erben. Dieser Aufbau der Vererbungsstruktur ist ständigen Änderungen unterworfen und sollte am Ende des Prozesses zu einer Vererbungshierarchie führen, die eine effektive Wiederverwendung von Komponenten ermöglicht. Die Bibliotheken nutzen Mehrfachvererbung und bilden daher statt eines Vererbungsbaumes ein Vererbungsnetz. Jeder Anwendungsprogrammierer ist dazu aufgerufen, selbst neue Klassen zu schreiben, welche die aufgestellten Kriterien erfüllen und daher für weitere Anwendungen verwendbar sind.

Die Bibliotheken von Eiffel können mittels eines C-Pakets gleichfalls für C-Programme verwendet werden. Da sie eine große Anzahl verschiedener Behälterklassen und zusätzlich Klassen für die grafische Darstellung und Fensteroberflächen beinhaltet, ist sie vielseitig einsetzbar. Ihr Umfang und ihre klare Hierarchisierung sind die Vorteile gegenüber den in Abschnitt 4.4 vorgestellten C++-Bibliotheken.

4.2.1 Strukturierung der Bibliotheken

Insgesamt sind in Eiffel zur Zeit folgende Bibliotheken verfügbar (Stand 1990: ca. 300 Klassen mit ca. 5000 sichtbaren Operationen):

Base Libraries sind die Basisbibliotheken mit ca. 150 Klassen, welche aufgegliedert sind in die Bibliotheken:

Kernel Library ist die Kernbibliothek mit fundamentalen Konzepten

Data Structure Library enthält Behälterklassen und Algorithmen

Lexical Library enthält Klassen für lexikalische Analyse

Parsing Library ist zuständig für die Syntaxanalyse

Iteration Library enthält Klassen zur systematischen Traversierung von Datenstrukturen und kann als Teil der Datenstruktur-Bibliothek angesehen werden

Vision Library enthält allgemeine, grafische Komponenten

Store Library enthält einen Persistenzmechanismus und Zugriff auf Datenbanken

Application Library enthält weitere anwendungsspezifische Klassen

In der folgenden Betrachtung der Eiffel-Bibliotheken werden die *Kernel* und *Data Structure Library* beachtet, da sie allein mit der Standardbibliothek von TLinhaltlich vergleichbar sind.

Die besonderen Strukturierungsprinzipien der Eiffel-Bibliotheken bestehen darin, daß die Klassen für Massendatentypen in der *Data Structure Library* nach den drei Eigenschaften

- Storage (Speicherung)
- Traversal (Durchlauf)
- Access method (Zugriffsmethode)

vollständig klassifiziert werden.

Ordnet man Klassen nach einem dieser Kriterien, entsteht eine Vererbungshierarchie. Mit Hilfe der Mehrfachvererbung kann eine spezielle Klasse durch Kombination von jeweils einer Klasse aus einer dieser drei Vererbungshierarchien erzeugt werden. So entsteht eine Architektur der *Data Structure Library* aus dieser Taxonomie der fundamentalen Programmierstrukturen. Weitere Kriterien, wie Lese-/Schreibberechtigung oder Persistenz, haben sich noch nicht als Grundlage für eine Klassifizierung durchgesetzt.

Kernbibliothek Die Klassen der Kernbibliothek (*Kernel Library*) umfassen diejenigen Strukturen, welche in dieser Arbeit besonders beachtet werden, da sie die Basisdatentypen und wesentliche Konzepte der Programmierung enthalten. Die zugehörigen Klassen lassen sich in fünf Bereiche gliedern. Diese werden in der Tabelle 4.2 dargestellt.

Jede benutzerdefinierte Klasse und jede weitere Bibliotheksklasse ist ein Erbe der universellen Klassen. Sie können als Wurzel der Vererbungshierarchie gesehen werden und definieren allgemeine Objekteigenschaften.

Die abstrakten Klassen *COMPARABLE, HASHABLE, PART_COMPARABLE* und *NUMERIC* stellen vererbbare, mathematische Eigenschaften für weitere Klassen zur Verfügung. Sie bilden eine Grundlage für Klassen der Datenstrukturbibliothek und werden daher zu der allgemeineren Kernbibliothek gezählt. *COMPARABLE* definiert eine totale Ordnung, *PART_COMPARABLE* eine partielle Ordnung auf Klassen. *HASHABLE* stellt eine Hash-Funktion für andere Klassen zur Verfügung. *NUMERIC* beschreibt Objekte, für welche die üblichen arithmetischen Operationen definiert sind. Diese vier Klassen können die generischen Parameter von Klassen einschränken. Mit der Angabe

Bibliothekskern	Klassen
universelle Klassen	ANY, PLATFORM, GENERAL
Sprachkonzepte	BOOLEAN, CHARACTER, INTEGER REAL, DOUBLE, ARRAY, STRING
Konversion	BASIC_ROUTINES
Mathematische Werte und Funktionen	MATH_CONST, SINGLE_MATH, DOUBLE_MATH
Primzahlen, Zufallszahlen, Zeichensatz	PRIMES, RANDOM, ASCII
Ein-/Ausgabe	STD_FILES, FILE, UNIX_FILE, UNIX_STD, DIRECTORY
Persistenz	STORABLE
Umgebung	EXCEPTION, MEMORY, MEM_INFO GC_INFO, ARGUMENTS, INTERNAL FILE_INFO
mathematische Eigenschaften	COMPARABLE, PART_COMPARABLE, HASHABLE, NUMERIC

Tabelle 4.2: Die Klassen der Kernbibliothek von Eiffel

class BINARY_SEARCH_TREE[G− >COMPARABLE] ...

wird verlangt, daß alle Elemente dieses speziellen Baumes vergleichbar sein müssen, d. h. sie müssen von der Klasse *COMPARABLE* erben.

Datenstrukturenbibliothek In diese Bibliothek gehören in weitestem Sinne Datenstrukturen für Behälter (Listen, Bäume, Queues, Kellerspeicher, Hashtabellen, Mengen, zirkulare Ketten usw.), Iteratoren und wichtige, aufzählbare Strukturen (Primzahlen, Pseudo-Zufallszahlen u. a.). Sie bilden die größte der vorhandenen Eiffel-Bibliotheken. Die Behälterklassen werden nach verschiedenen Eigenschaften strukturiert. Diese Struktur wird auch in der Speicherung der Bibliotheksquelltexte wiedergegeben. Sämtliche in Tabelle 4.3 aufgeführten Pakete bilden ein eigenes Unterverzeichnis im Verzeichnis der Standardbibliothek.

Alle Behälterklassen sind generisch definiert. Sie besitzen einen formalen Parameter, welcher einen Typ repräsentiert, der für die enthaltenen Elemente genutzt wird.

Die Klassen der Datenstrukturbibliothek stellen hauptsächlich die Datenstrukturen für Behälter und die drei vererbbaren Klassifizierungseigenschaften *Speicherung*, *Durchlauf* und *Zugriffsmethode* in den Paketen *storage*, *traversing* und *access* zur Verfügung. Die restlichen Klassen stellen einfache Konzepte, Iteratoren und weitere, aufzählbare Strukturen dar. Die Tabelle 4.3 gibt die Gliederung der Datenstrukturbibliothek in ihre einzelnen Pakete wieder.

Die Klasse *CONTAINER* ist die Superklasse aller Behälterklassen. Den drei Klassifizierungseigenschaften folgend, sind die direkten Erben dieser Klasse *BOX*, *COLLECTION* und *TRAVERSABLE*. Die Hierarchie wird in Abbildung 4.1 dargestellt.

Unter den Behälterklassen befinden sich ebenfalls abstrakte Klassen, z. B. *SEQUENCE*

Bibliothekspaket	Klassen
traversing Klassifizierungs-eigenschaften zum Durchlaufen	BIDIRECTIONAL, HIERARCHICAL, LINEAR, SEQUENTIAL, TRAVERSABLE
storage Klassifizierungs-eigenschaften zur Speicherung	BOUNDED, BOX, COUNTABLE, FINITE, FIXED, INFINITE, RESIZABLE, UNBOUNDED
access Klassifizierungs-eigenschaften zum Zugriff	ACTIVE, BAG, COLLECTION, CONTAINER, CURSOR_STRUCTURE, INDEXABLE, TABLE
list Lineare Strukturen mit verschiedenen Eigenschaften	ARRAYED_CIRCULAR, ARRAYED_LIST, CELL, PART_SORTED_LIST, BI_LINKABLE, CHAIN, CIRCULAR, DYNAMIC_CHAIN, DYNAMIC_CIRCULAR, DYNAMIC_LIST, LINKABLE, LINKED_CIRCULAR, LINKED_LIST, LIST, MULTI_ARRAY_LIST, SEQUENCE, SORTED_LIST, SORTED_TWO_WAY_LIST, TWO_WAY_CIRCULAR, TWO_WAY_LIST, FIXED_LIST, PART_SORTED_LIST
dispenser LIFO- und FIFO-Speicher	ARRAYED_QUEUE, ARRAYED_STACK, LINKED_QUEUE, PRIORITY_QUEUE, DISPENSER, SORTED_PRIORITY_QUEUE, HEAP, LINKED_STACK, QUEUE, STACK, BOUNDED_QUEUE, BOUNDED_STACK
tree Bäume versch. Implementationen	ARRAYED_TREE, BINARY_SEARCH_TREE, BINARY_TREE, DYNAMIC_TREE, FIXED_TREE, TREE, TWO_WAY_TREE
cursor_tree Zeigerstrukturen für rekursive Bäume	COMPACT_CURSOR_TREE, CURSOR_TREE, LINKED_CURSOR_TREE, RECURSIVE_CURSOR_TREE, TWO_WAY_CURSOR_TREE
set ungeordnete Mengen	BINARY_SEARCH_TREE_SET, COMPARABLE_SET, LINKED_SET, SET, SORTED_SET, SUBSET, PART_SORTED_SET, TWO_WAY_SORTED_SET
table Mengen mit Schlüsselwert	ARRAY2, HASH_TABLE
cursors Zeigerstrukturen auf eine durchlaufbare Datenstruktur	ARRAYED_LIST_CURSOR, COMPACT_TREE_CURSOR, CURSOR, LINKED_LIST_CURSOR, LINKED_TREE_CURSOR, MULTAR_LIST_CURSOR, RECURSIVE_TREE_CURSOR, TWO_WAY_TREE_CURSOR
iteration Iteratoren	ITERATOR, CURSOR_TREE_ITERATOR, LINEAR_ITERATOR, TWO_WAY_CHAIN_ITERATOR

Tabelle 4.3: Tabelle der Datenstrukturbibliothek von Eiffel

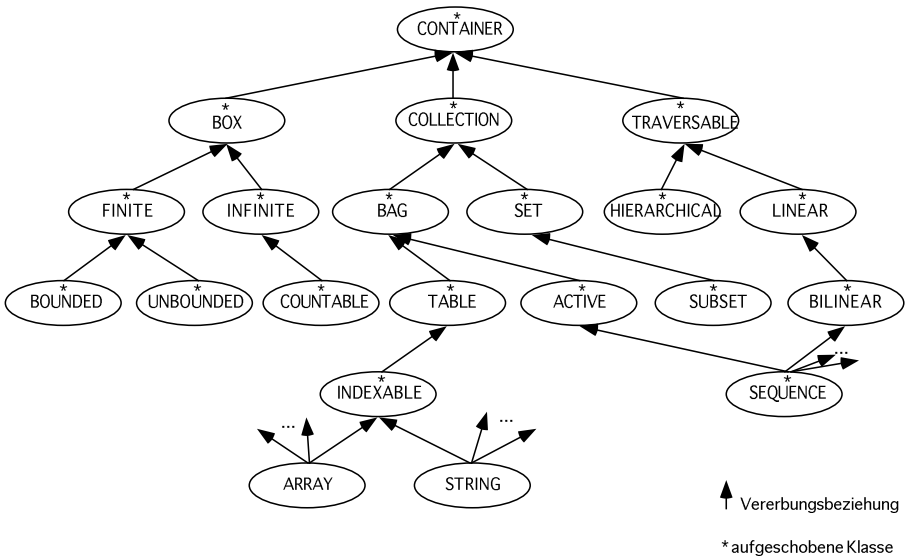


Abbildung 4.1: Hierarchie der Behälterklassen in Eiffel

und *QUEUE*, welche als *aufgeschoben* (engl. *deferred*) deklariert sind. Sie haben selbst keine Ausprägung als Objekt (Instanzen), sondern vererben bestimmte Eigenschaften mit ihren Methoden. In den erbenden Klassen, z.B. *PART_SORTED_LIST* und *ARRAYED_QUEUE*, werden die Operationen dann spezialisiert, um Objekte dieser Klassen erstellen zu können.

Eine Iteratorklasse wird von den Behälterklassen geerbt und kann ebenfalls von benutzerdefinierten Klassen geerbt werden. Die Methoden des Iterators werden in der erbenden Klasse überschrieben, um die geforderte Operation auszuführen. Für das Durchlaufen der Datenstruktur muß auf diese Weise keine Schleifenkontrollstruktur geschrieben werden. Eine Klasse kann von mehreren Iteratoren erben, um die verschiedenen Iterationsschemata zu verwenden. Jede der Iteratorklassen entspricht einem bestimmten Durchlaufschema. Somit gibt es zu jeder Iteratorklasse eine entsprechende Klasse in der *traversal*-Hierarchie.

4.2.2 Konvention der Methoden

Die Methoden innerhalb der Klassen sind kategorisiert. Dies ist eine wesentliche Hilfe für den Benutzer der Bibliotheken, um die Methoden der Klassen kennenzulernen und wiederzufinden. Es zeigt, daß eine Gesetzmäßigkeit der vorhandenen Methoden festgestellt werden kann, die von den Eigenschaften der jeweiligen Klasse abhängen. Die Eiffel-Kategorien werden hier aufgezeigt, da sie wesentlich die in Abschnitt 5.5.1 vorzustellenden Kategorien für Tycoon beeinflussen.

Die in den vorgestellten Eiffel-Bibliotheken benutzten Kategorien zur Strukturierung der Methoden innerhalb einer Klasse sind im folgenden dargestellt. Beispielhaft werden Methoden in Eiffel genannt, die besonders häufig vorkommen. Dabei werden in diesem Zusammenhang sämtliche Parameter der Methoden vernachlässigt. Defaultmäßig gelten alle Operationen in Eiffel für das **Current** Objekt, falls es nicht ausdrücklich anders angegeben wird. Das zu ändernde bzw. das zu behandelnde Objekt muß, im Gegensatz zu Tycoon, daher nicht als Parameter übergeben werden.

Initialization Die Initialisierung von Objekten einer Klasse ist nur in den nicht abstrakten, nicht aufgeschobenen (*deferred*) Klassen nötig. Die Methode zur Initialisierung wird in Eiffel für alle Klassen *make* genannt.

Access Der Zugriff auf die einzelnen Elemente des Datentyps ist nur für Behälterklassen relevant. Hierfür wird die Standardmethode *item* verwendet. Aber auch der Zugriff auf den Index ist bei einigen Klassen mit *index_of* möglich.

Measurement Die Größe eines Objekts ist wiederum für Behälterklassen von Bedeutung. Die Abfrage auf die Zahl der Elemente im Behälter lautet in Eiffel *count*. *occurences* zählt die Anzahl eines bestimmten Elements in einem Behälter mit Duplikaten.

Comparison Für fundamentale Datentypen existiert eine Ordnung innerhalb der möglichen Werte. Zum Vergleich zweier Werte gibt es die üblichen Vergleichsoperationen $<$, $<=$, $>$ und $>=$ mit booleschem Ergebnistyp. Die Tests auf Gleichheit lassen sich folgendermaßen spezifizieren:

- *equal* vergleicht zwei Objekte Feld für Feld auf gleichen Inhalt. Es ist die *shallow*-Version der *equal*-Methode und wird hier *Objekt-Gleichheit* genannt.
- *deep_equal* stellt fest, ob die Objektstrukturen rekursiv identisch sind.
- $=$, $/=$ vergleichen zwei Referenzen auf Objekte. Beziehen sich die Referenzen auf dasselbe Objekt, dann besteht *Referenz-Gleichheit*, welches mit der *deep*-Methode erreicht wird.

Basic operations Für die fundamentalen Datentypen werden die arithmetischen Operationen deklariert.

Status report Angaben über den Zustand eines Objekts betreffen z. B. die Frage, ob der Behälter leer ist (*empty*). Außerdem wird für viele Behälterklassen *has* verwendet, um festzustellen, ob ein bestimmtes Element enthalten ist. Weitere spezielle Eigenschaften eines Objekts können die Erweiterbarkeit (*extendible*) oder die Möglichkeit, enthaltene Elemente zusammenzustreichen (*prunable*), betreffen.

Status setting Die Eigenschaften eines Objekts können mit einigen Methoden geändert werden, z. B. setzt *compare_objects* die Vergleichsoperation *equal* statt $=$ für alle folgenden Vergleiche.

Cursor movement Für Behälterklassen mit definierter Ordnung werden Methoden benötigt, welche das aktuelle Element verändern. Der Zeiger kann dabei um eine Position (*forth*) oder i Positionen (*move*) vorbewegt werden. Ebenso kann auf das erste (*start*), letzte (*finish*) oder das erste Element mit bestimmten Eigenschaften (*search*) positioniert werden.

Element change Der Inhalt des Objekts kann insbesondere für Behälterklassen verändert werden. *put* ist die allgemeine Einfügemethode für ein Element, wenn es nicht schon vorhanden ist. *replace* ersetzt das aktuelle Element mit einem neuen Wert. *append* wird zum Anfügen eines Behälters gleicher Struktur verwendet, *prepend* fügt eine Datenstruktur vorne ein. Während *extend* den Behälter um ein angegebenes Element am Ende erweitert, wird mit *precede* ein Element am Anfang eingefügt.

Removal Das Entfernen von Elementen aus einem Behälter kann auf verschiedene Weise erfolgen. *prune* entfernt das angegebene Element einmal. In Behältern mit Duplikaten entfernt *prune_all* alle Elemente mit dem gegebenen Wert. *wipe_out* löscht alle Elemente eines Behälters. *clear_all* setzt alle Elemente auf den Standardwert. *remove* löscht das Element an der angegebenen Position im Behälter.

Resizing Für Behälter mit fester Größe werden Methoden definiert, welche seine Größe ändern.

Transformation An bestimmten Stellen innerhalb eines Behälters kann ein Austausch des Elements mit der Methode *swap* vorgenommen werden.

Conversion Die Transformation einer Datenstruktur in eine andere ist auf vielfältige Weise möglich. So kann z. B. eine Menge in eine lineare Struktur überführt werden oder eine ganze Zahl als Zeichenkette dargestellt werden (*to_string*). Die Methode *to_c* gibt für fast alle Datentypen eine Adresse des Speicherplatzes aus, um dieses Objekt außerhalb der Eiffel-Umgebung referenzieren zu können.

Duplication Ein Teil einer Datenstruktur kann kopiert werden und als neues Objekt weiterverarbeitet werden. Bei Zeichenketten können so z. B. Teile der Zeichenkette mit *substring* kopiert werden. Das Kopieren von Datenstrukturen kann in ein bestehendes Objekt (*copy*) oder ein neu erstelltes Objekt (*setup*) erfolgen. *resize* paßt die Größe des Kopie-Objekts automatisch dem kopierten Inhalt an. Die Standardkopiermethoden sind:

- *copy* kopiert die Felder eines Objekts in die Felder eines anderen, bereits existierenden Objekts. Es führt die Standardfunktion *copy* der Sprache C aus, wenn es nicht neu definiert wurde. Die Syntax ist wie folgt:

target.copy(source)

- *clone* kopiert die Felder des aktuellen Objekts in ein neu erstelltes Objekt. Es ist eine *shallow* Methode, welche die Zeiger auf die referenzierten Felder übernimmt. Es wird mit Hilfe von *copy* definiert und folgt damit jeder Änderung der *copy*-Methode automatisch. Die Syntax lautet:

target := clone(source)

- *deep_clone* produziert ein Duplikat einer gesamten Objektstruktur.

Output Eine druckbare Repräsentation des Objekts ist besonders für die fundamentalen Datentypen *INTEGER*, *CHARACTER* usw. von Bedeutung.

Expandierte Typen, deren Werte Objekte statt Referenzen sind, sind die Typen *BOOLEAN*, *INTEGER* und *CHARACTER*. Für sie wird immer die Objektgleichheit mit *equal* festgestellt. Wenn Elemente eines Behälters keine Objekte, sondern nur Referenzen auf Objekte sind, und der zu vergleichende Wert selbst eine Referenz ist, dann existieren zwei Möglichkeiten, die Gleichheit festzustellen:

1. *Referenzgleichheit* ($a = b$) wird erfüllt, wenn die verglichenen Referenzen mit demselben Objekt verknüpft sind. Diese Eigenschaft wird auch *Identität* genannt.

2. *Objektgleichheit* ($equal(a, b)$) wird erfüllt, wenn die beiden verglichenen Referenzen mit Objekten verknüpft sind, welche im Sinne der *equal*-Funktion der Klasse *ANY* gleich sind. Standardmäßig bestimmt *ANY* die Gleichheit Feld für Feld, diese Funktion kann aber neu definiert werden. Diese Eigenschaft wird auch *semantische Gleichheit* genannt.

Diese verschiedenen Eigenschaften müssen in der Bibliothek vollständig zur Verfügung gestellt werden. Ebenso müssen sie einheitlich und eindeutig definiert werden.

Die Methoden *copy*, *deep_copy* und *out* (Ausgabe von Datenstrukturen auf ein externes Medium) werden als Spracherweiterung angesehen. Von der Implementation her gesehen sind sie allerdings Eigenschaften der Standardbibliotheksklasse *ANY*, welche von jeder anderen Klasse beerbt wird. Die Erweiterung der Bibliothek hat in diesem Zusammenhang die Bedeutung, den Sprachumfang gering zu halten. Sie birgt andererseits die Schwierigkeit, die Bibliothek konsistent und einfach zu halten. Dieses Problem läßt sich in jedem Bibliotheksentwurf wiederfinden. Da jede Benutzeranwendung und Bibliothek auf der Sprache basieren, sollte der geringe Sprachumfang vorrangiges Ziel sein [Meyer 90a]. Die Bibliothek sollte allerdings ebenfalls nicht komplex werden, um überschaubar zu bleiben, aber die Komplexitätssteigerung wirkt sich hier weniger auf die Laufzeiteffizienz aus.

4.3 Smalltalk

In Smalltalk-80 werden eine Reihe von vordefinierten Klassen mitgeliefert, welche als eine Art ***Bibliothek*** angesehen werden können. Diese ca. 220 Klassen mit ca. 4500 Methoden [Hoffmann 87] umfassen die grundlegenden Bereiche einer Programmiersprache wie Zahlen, Datenstrukturen, Dateiverwaltung, Kontrollstrukturen, Prozeßverwaltung, Klassenverwaltung, Bildschirm und Grafik. Daher erlauben sie ein breites Spektrum von Anwendungsmöglichkeiten. Die Klassen sind hierarchisch angeordnet, d.h. es wird durch Spezialisierung einer Klasse eine Unterklasse gebildet. Smalltalk ist nicht typisiert und bietet daher keine generischen Klassen. Jede neue Klasse muß in die Klassenhierarchie integriert werden. Smalltalk besitzt kein Konzept zur Anbindung externer, benutzerdefinierter Bibliotheken.

Die Sprache wird mit sämtlichen, verfügbaren Klassen in dem Grundlagenbuch über Smalltalk [Goldberg, Robson 83] beschrieben. In einem Klassenbaum mit der Wurzel *Object* sind viele der Standardklassen von Smalltalk für einen Vergleich interessant. Sie werden in Abbildung 4.2 aufgelistet.[1] Die Massendatentypen *Array* und *String* gelten in einigen Sprachen als Basisdatentypen, werden in Smalltalk jedoch aufgrund ihrer Eigenschaften in einer Kategorie mit anderen, aggregierten Datenstrukturen angegeben. Da sämtliche Namen der Klassen selbsterklärend sind und zum Teil mit den Modulnamen von Modula-3 und Eiffel übereinstimmen, wird in der Tabelle auf eine Kurzbeschreibung verzichtet.

Weitere Klassen existieren für die Klassenverwaltung, Bildschirmdarstellung sowie Grafikerstellung.

In idealisierter Sichtweise geben die Klassen in Smalltalk einen *Add-On*-Ansatz wieder. Datenstrukturen und primitive Datentypen, wie *Integer*, *Real*, *Character* oder *String*, werden als vordefinierte Klassen zur Verfügung gestellt und sind damit prinzipiell austauschbar. In

[1] Die Einrückung einer Klasse bedeutet, daß sie von der vorherigen Klasse erbt. Klassen innerhalb einer Einrückungstiefe sind voneinander unabhängig.

Basisdatentypen:
Magnitude
- Character
- Date
- Time
- Number
  - Float
  - Fraction
  - Integer
    - LargeNegativeInteger
    - LargePositiveInteger
    - SmallInteger
- LookupKey
  - Association

UndefinedObject
Boolean
- False
- True

Prozesse:
Link
- Process

ProcessorScheduler
Delay
SharedQueue

Klasseneigenschaften:
Behavior
- ClassDescription
  - Class
  - MetaClass

Massendatentypen:
Collection
- SequencableCollection
  - LinkedList
  - ArrayedCollection
    - Array
    - String
      - Symbol
    - Text
  - Interval
  - OrderedCollection
    - SortedCollection
- Bag
- Set
  - Dictionary

Ein-/Ausgabe:
Stream
- PositionableStream
  - ReadStream
  - WriteStream
    - ReadWriteStream
      - ExternalStream
        - FileStream
- Random

File
FileDirectory
FilePage

Abbildung 4.2: Smalltalk Klassenhierarchie

der Smalltalk-Implementation werden einige dieser Klassen allerdings fest eingebaut, so daß sie für den Anwendungsprogrammierer nicht austauschbar sind [Hoffmann 87].

Anstelle eines Iterators stellt die Superklasse aller Behälterklassen, ***Collections***, die Mechanismen zur Iteration zur Verfügung. Die Operationen und ihre Erklärung werden in der Tabelle 4.4 zitiert.

4.4 C++

Die Probleme der Isolierung und Ineffizienz des Smalltalk-Systems (siehe Abschnitt 3.2.3) können mit der Einbindung von objektorientierten Konzepten in die Programmiersprache C gelöst werden, weil sie effizient benutzbar und auf vielen Rechnern verfügbar ist [Gorlen 87]. Um die Vorteile der objektorientierten Programmierung voll ausschöpfen zu können, wurden zahlreiche Bibliotheken für C++ entworfen, die sich zum größten Teil auf die Smalltalk-Bibliotheken stützen, um deren Funktionalität nachzubilden.

Operation	Beschreibung
add	Einfügen eines Elements
addAll	Einfügen einer Menge (Subtyp von *Collection*) von Elementen
remove	Entfernen eines Elements
removeAll	Entfernen einer Menge von Elementen
includes	Rückgabe eines boolschen Wertes, ob das Argument im Behälter steht
isEmpty	Rückgabe eines boolschen Wertes, ob der Behälter leer ist
occurencesOf	Anzahl der Duplikate des Arguments
do	Auswertung des Arguments für alle Elemente
select	Auswertung des Arguments für alle Elemente und Rückgabe eines Behälters mit den Elementen, die den Wert *true* bekommen
reject	wie select mit Rückgabe aller Elemente, die zu *false* evaluieren
collect	Auswertung des Arguments für alle Elemente und Rückgabe eines Behälters mit den errechneten Werten
detect	Auswertung des Arguments für alle Elemente und Rückgabe des ersten Elements, welches *true* wird
inject	Auswertung des Arguments für jedes Element, wobei die jeweils vorherige Auswertung als aktueller Argumentparameter einbezogen wird

Tabelle 4.4: Iteratoroperationen in Smalltalk

Verwendet eine Bibliothek keine eigenen Präfixe für globale Namen, so können bei einem gleichzeitigen Einsatz mehrerer Bibliotheken Namenskonflikte auftreten. Die Wahrscheinlichkeit für gleiche Namen in verschiedenen Bibliotheken ist für C++ nicht zu unterschätzen, da viele Bibliotheken die Hierarchie von Smalltalk nachbilden und daher die von Smalltalk verwendeten Namen benutzen. Zur Umgehung dieses Konflikts sind in dem neuesten Standard für C++ Namensbereiche (engl. *name spaces*) eingeführt. Ein Bibliotheksbenutzer kann diesen Konflikt in jedem Fall umgehen, indem er die Bibliotheksklassen selbst umbenennt.

In C++ sind die Datentypen *Integer* und *Character* eingebaute Basiskonstrukte. Sie bilden daher keine Klasse in dieser objektorientierten Sprache. Um einen vollständigen *Add-On*-Ansatz für alle Datentypen zu verfolgen, bilden einige der folgenden Bibliotheken auf der Grundlage dieser Basisdatentypen eine eigene Klasse, damit die Verwendung der Typen *Integer* und *Character* auch im objektorientierten Programmierstil gewährleistet wird.

Gerade für C++, für welches eine große Anzahl von Klassenbibliotheken für die verschiedensten Anwendungsbereiche erhältlich ist, wird in verschiedenen Publikationen beschrieben, welche Kriterien zur Strukturierung von Bibliotheken benutzt wurden [Möller, Müller 93; Coggins 90; Cohen et al. 92].

Ein großer Unterschied zwischen den im folgenden vorgestellten Bibliotheken für C++ besteht in ihrer Hierarchiestruktur. Während NIHCL und ET++ sich streng an ihre Vorlage von Smalltalk halten und ihre Klassen als Baum strukturieren, bilden libg++, LEDA und die Booch-Komponenten einen Klassenwald.

NIHCL. In der amerikanischen Gesundheitsbehörde ist auf Initiative von Keith Gorlen NIHCL[2] als ein frei verfügbares Produkt[3] entwickelt worden. Es besteht zu dieser Bibliothek keine offizielle Entwicklerunterstützung und keine Dokumentation.

NIHCL erweitert die Funktionalität von C++ mit der Implementation von ADTs, welche

[2] Abk. für *National Institute of Health Class Library*

[3] Via FTP ist die vollständige Bibliothek z. B. auf *alw.nih.gov* unter */pub/NIHCL/nihcl.tar.Z* verfügbar.

die objektorientierte Programmierung in C++ vereinfachen [Gorlen et al. 90]. Die Architektur orientiert sich an Smalltalk-80, wodurch alle Klassen als Baum mit einer gemeinsamen Wurzel *Object* als Basisklasse organisiert sind. Innerhalb dieses Klassenbaumes wird nur Einfachvererbung benutzt, aber die Mehrfachvererbung ist vom Anwender mit Hilfe eines Präprozessor-Schalters nutzbar.

Die Spezifikation und Implementation von Klassen erfolgt in getrennten Dateien. Für beide existieren Klassendateien als Schablonen, um dem Benutzer das Schreiben von eigenen Klassen zu erleichtern.

Die Entwurfsziele sind eine Generalisierung der Klassen, so daß ein Algorithmus mit jeder beliebigen Klasse ausführbar ist, und eine Spezialisierung der Klassen, welches eine flexible Anpassung der Klassen an die jeweilige Anwendung erlaubt.

Die Wurzel der Bibliothek ist die Klasse *NIHCL*, welche aus einer Ansammlung von statischen Funktionen besteht, um diese nicht global deklarieren zu müssen. Die eigentliche Basisklasse aller anderen Klassen ist deren Unterklasse *Object*. Sie realisiert allgemeine Dienste, wie das Kopieren komplexer Objekte mittels einer *shallowCopy*- oder *deepCopy*-Prozedur sowie Persistenz.

Die Ein- und Ausgabe von Objekten erfolgt über eine programm- und maschinenunabhängige Darstellung. Für benutzerdefinierte Klassen wird das Speicherformat automatisch erzeugt. Eine Ausgabedatei ist eine Instanz der Klasse *fstream* und wird mit der Funktion *storeOn()* beschrieben. Ebenso ist eine Eingabedatei eine Instanz von *fstream* und wird mit *readFrom()* gelesen. Auf diese Weise kann eine komplexe Datenstruktur, welche aus mehreren, von *Object* abgeleiteten Klassen besteht, in eine portable Darstellung konvertiert werden.

NIHCL enthält allgemein nützliche Datentypen wie *String*, *Float*, *Integer*, *Date* und *Time*. Im Gegensatz zu Smalltalk werden die grundlegenden Datentypen, welche in C fest eingebaut sind wie z. B. *Integer* und *Character*, nicht als Objekte behandelt. Es werden zwar Klassen wie *Integer* und *Float* vorgesehen, aber für sie werden keine arithmetischen Operationen eingebaut, um die Basisdatentypen effizient benutzen zu können.

Daneben stehen eine Menge von Behälterklassen als Subklassen der abstrakten Klasse *Collection* zur Verfügung. Das Angebot an Behälterklassen in NIHCL ist sehr viel größer als in Smalltalk. Die Hierarchie der Behälterklassen beinhaltet die in Tabelle 4.5 genannten Klassen.[4]

Zu diesen Klassen existiert eine Iteratorklasse, welche für das Iterieren innerhalb der enthaltenen Objekte zuständig ist. Obwohl keine stabilen Iteratoren implementiert wurden, sind zwei oder mehrere Iteratoren pro Klasse anwendbar. Für eine typsichere Verwendung ist jedoch eine anwendungsspezifische Ableitung beider Klassen, der Behälterklasse und der Iteratorklasse, nötig. Dies kann mit Hilfe von Schablonen (engl. *templates*) geschehen, soweit der Compiler das Konzept der Schablonen unterstützt. Auf eine Behälterklasse sind mehrere Iteratoren anwendbar, das Konzept der stabilen Iteratoren wird dabei jedoch nicht unterstützt.

Für asynchrone Prozesse und Multiprogrammierung mit Koroutinen stehen die Klassen *Process* (Prozeßobjekt für Koroutinen), *Scheduler* (Prozeß-Scheduler für Koroutinen), *Semaphore* (zur Prozeßsynchronisation) und *SharedQueue* (Queue von Objekten mit gleichzeitigem Zugriff) zur Verfügung.

Die Klassen *Vector* (abstrakte Klasse für Vektoren), *Random* (Zufallszahlengenerator), *Range* (Bereich der Integerzahlen) und viele weitere Klassen assistieren bei arithmetischen und mathematischen Problemen.

[4] Eine Einrückung bedeutet, daß die Klasse von der vorhergehenden Klasse erbt.

Collection — Abstrakte Klasse für Behälterklassen
- *Arraychar* — Byte Array
- *ArrayOb* — Array von Objektzeigern
- *Bag* — Ungeordnete Sammlung von Objekten
- *SeqCltn* — Abstrakte Klasse für geordnete, indizierte Behälterklassen
  - *Heap* — Haufen von Objektzeigern
  - *LinkedList* — Einfach verkettete Liste
  - *OrderedCltn* — Geordneter Behälter von Objektzeigern, ein indiziertes Feld
    - *SortedCltn* — Sortierter Behälter
      - *KeySortCltn* — Sortierter Behälter mit Schlüssel
  - *Stack* — Stapel von Objektzeigern
- *Set* — Ungeordneter Behälter von Objekten ohne Duplikate (Hash-Tabelle)
  - *Dictionary* — Menge von Assoziationen
    - *IdentDict* — Dictionary mit Objektadresse als Schlüssel
  - *IdentSet* — Set mit Objektadresse als Schlüssel

Tabelle 4.5: Behälterklassen der NIH Klassenbibliothek

Generische Klassen stehen nicht allgemein zur Verfügung. Beschränkt auf Arrays der elementaren Typen wird ein Script für den Unix Makroprozessor M4 mitgeliefert, um generische Klassen zu erzeugen.

Innerhalb der NIHCL wird ein Mechanismus zur Ausnahmebehandlung implementiert, der jedoch von den Autoren selbst als wenig effizient [Gorlen et al. 90, p. 352] eingeschätzt wird. Weiterhin wird keine automatische Speicherbereinigung (*garbage collection*) vorgesehen, so daß der Benutzer seinen Objektspeicher selbst verwalten muß.

Die Bibliothek ist für Unix-Systeme verfügbar. Die Portierung auf andere Systeme wird erschwert, weil betriebssystemspezifische Methoden über die gesamte Bibliothek verteilt sind und nicht zentral in einem eigenen Modul gehalten werden. Auch die Verwendung des speziellen Makro-Prozessors zur Implementation von generischen Klassen macht eine Übertragung auf andere Plattformen zeitaufwendig.

ET++. Diese Bibliothek ist für interaktive Anwendungen in UNIX-Umgebungen und mit einem herkömmlichen Fenstersystem an der Universität Zürich 1987 entwickelt worden [Weinand et al. 88]. ET++[5] soll die Funktionalität von MacApp und der Benutzerschnittstelle der nicht-objektorientierten Macintosh Toolbox in einem einzigen System vereinen. Die Integration von Klassen, wie sie in der Smalltalk-Umgebung bekannt sind, bedeutet nicht nur die Installierung einer Klassenhierarchie, sondern auch die Unterstützung dieser Klassen mit Mechanismen zur Bildschirmdarstellung und Interaktion. Daher beinhaltet ET++ Benutzerschnittstellenblöcke genauso wie Basisdatentypen. Für dieses System existiert eine Onlinedokumentation.

ET++ ist ein objektorientiertes System mit einer grafischen Benutzerschnittstelle. Es ist homogen und erweiterbar, um die Effizienz bei der Erstellung von komplexen Anwendungs-

[5] ET, Abk. für *Editor Toolkit*, ist das nicht-objektorientierte System der Universität Zürich, auf welchem ET++ basiert.

systemen zu steigern. Da für Unix, im Gegensatz zu Macintosh-Rechnern, kein Standard für Fenstersysteme existiert, wurde versucht, eine kleine Menge an Fenstersystemfunktionen zu finden, die in den meisten Fenstersystemen verfügbar sind oder durch eine kleine Schnittstelle implementiert werden können.

Die Architektur von ET++ besteht aus einer Klassenhierarchie und einer kleinen, geräteabhängigen Ebene, welche das abstrakte Fenstersystem auf ein darunterliegendes, reales Fenstersystem abbildet. Durch Anpassung dieses *System Interface* wird die Klassenbibliothek mit geringem Aufwand leicht portierbar. Außer den Klassen des *System Interface* und den *Iteratoren* sind alle Klassen von der Klasse *Object* abgeleitet, so daß sich ein Klassenbaum innerhalb eines Klassenwaldes ergibt.

Zur Klassenhierarchie von ET++ gehören die *User-Interface Classes*, *Application Classes* und *Foundation Classes*. Grafische Elemente werden in den *User-Interface Classes* implementiert. Dazu gehören Menues, Dialogboxen und Rollbalken. In den *Application Classes* befinden sich die Klassen auf höchster Ebene, welche die allgemeine Kontrollstruktur von Anwendungen in einer grafischen Umgebung bereitstellen.

Die *Foundation Classes* sind allgemein nützliche Datenstrukturen, die unabhängig von den weiteren Klassen in ET++ genutzt werden können. Sie beinhalten eine vereinfachte Version der *Collection Classes* von Smalltalk-80 und repräsentieren damit die Basisdatenstrukturen und grundlegenden Behälterklassen, wie *array*, *list*, *set* usw. Diese Klassen werden auch zur Implementation von ET++ selbst benutzt. Die vielfach grafisch benötigten Klassen *Point* und *Rectangle* zählen ebenfalls zu den *Foundation Classes*.

Für die Ein- und Ausgabe von Objekten sind zwei Methoden implementiert. Zusammen mit den flexiblen *stream*-Klassen von C++ erlauben sie eine allgemeine Implementation einer *DeepCopy*-Methode. Eine weitere Anwendung der Ein- und Ausgabemethoden ist eine Art *Inspector* für jedes Objekt. Der Zustand eines Objekts wird von der Methode *inspect* in einem eigenen Fenster dargestellt, welches bei Änderungen laufend aktualisiert wird.

ET++ unterstützt keine Mehrfachvererbung. Eine Nachbildung mit einfacher Vererbung durch die Anwendung von Objektkomposition mit Black-Box-Klassen ist möglich. Es bietet gleichfalls keine Generizität, um eine möglichst einfache Portierbarkeit auf andere Systeme zu erhalten, welches ein Hauptziel beim Entwurf von ET++ war.

Zur Fehlerbehandlung ist eine Fehlerbehandlungsmethode implementiert, welche dem Benutzer das Auftreten von Fehlern anzeigt. Dabei wird der Fehler in eine Fehlerklasse eingeordnet. Eine Fehlerkorrektur wird nicht durchgeführt. Sie liegt in der Verantwortung des Benutzers.

Es werden Iteratoren für jede Behälterklasse geliefert, die von der abstrakten Klasse *Iterator* abgeleitet sind. Diese abstrakte Klasse definiert ein abstraktes Protokoll für das Navigieren innerhalb eines Behälters, welches in der entsprechenden Iterator-Klasse spezialisiert werden kann. Auf einem Behälter können mehrere Iteratoren aktiv sein, da das Konzept der stabilen Iteratoren unterstützt wird.

LEDA. Die Klassen der LEDA-Bibliothek sind als Klassenwald aufgebaut [Möller, Müller 93]. Sie wurde am Max-Planck-Institut für Informatik von Stefan Näher für alle C-Front- und GNU-C++-Compiler entworfen und ist für Unix- oder DOS-Plattformen verfügbar. Enthalten sind einfache Datentypen, wie *Boolean*, *Real* und *String*, ebenso wie grundlegende Datenstrukturen (*Stacks, Lists, Arrays* etc.), spezialisierte Behälterklassen (*Dictionaries* etc.) und parametrisierbare Klassen (*Graphs* etc.).

Mehrfachvererbung wird nicht berücksichtigt. Vorbedingungen, die durch *if*-Konstrukte implementiert sind, werden für fast alle Operationen geprüft. Zur Erhaltung der Effizienz wird allerdings auf diesen Test verzichtet, wenn er eine vorgegebene, konstante Zeit überschreitet. Nach Verletzung einer Vorbedingung wird die Programmausführung mit einer Fehlermeldung abgebrochen, ohne eine Ausnahmebehandlung zu ermöglichen.

Es existieren für die meisten Datenstrukturen Iterationsmakros, die direkt in der Datenstruktur implementiert sind. Da keine stabilen Iteratoren unterstützt werden, zieht die Modifikation eines Behälters mit einem aktiven Iterator einen Programmabbruch mit Fehlermeldung nach sich.

OATH. Diese Bibliothek stellt eine Methode zum Entwurf einer Klassenhierarchie in C++ dar, die das Konzept des Subtyppolymorphismus ausnutzt. Sie sieht eine größere Implementationsunabhängigkeit vor und unterstützt implizit die Speicherverwaltung der von der Anwendung erzeugten Objekte. In OATH[6] wird eine strikte Subtypmethode zum Entwurf der Hierarchie angewendet, um eine abstrakte Typhierarchie zu erhalten, die mit den zu modellierenden Konzepten übereinstimmt [Kennedy 91]. Weiterhin soll diese Bibliothek größere Unabhängigkeit von der Implementation verwirklichen und implizites Speichermanagement ihrer Objekte unterstützen. Diese Eigenschaften sollen dem Entwickler von Applikationen eine wertvolle Hilfe bei der Entwicklung anwendungsspezifischer Klassen sein.

OATH steht für ein Experiment in der objektorientierten Programmierung mit dem Ziel der Erhöhung der Wiederverwendung. Daher gibt es zu dieser Bibliothek keine Benutzerunterstützung und externe Dokumentation.

OATH ist als Klassenbaum konzipiert, dessen Wurzel *obj* ist. Darunter gliedern sich nach den Subtypregeln u. a. die Klassen für die Basisdatentypen und Behälterklassen. Diese Typen sind einfach und verständlich, so daß sie keiner weiteren Erläuterung bedürfen. Um die Behälterklassen generell und flexibel zu gestalten, werden sie als generische Klassen angelegt. Die Vorteile dieser Bibliothek werden in ihrem ganzen Umfang sichtbar, wenn sie für komplexe, anwendungsspezifische Probleme verwendet wird [Kennedy 91]. In Tabelle 4.6 wird ein Teil der OATH wiedergegeben.[7]

Vererbung wird in C++ zum einen für die Vererbung der Funktionalität, der *Subtypisierung*, und zum anderen für die Vererbung der Implementation, der *Wiederverwendung*, benutzt. Diese trennbaren Konzepte sollen in der OATH unterschieden werden. Die Hierarchie der Bibliothek wird daher nur durch Subtypisierung erzeugt, um das Verhalten der modellierten Objekte zu reflektieren. Im Gegensatz zu anderen Bibliotheken wird die Repräsentation der Objekte auf dem Computer nicht auf die bequemste und speicherplatzsparendste Art widergespiegelt. Die OATH sieht eine Hierarchie von abstrakten Typen vor. Die Spezifikationen des Verhaltens korrespondieren zu den modellierten Konzepten. An den Blättern des Baumes der abstrakten Typen können Klassen angefügt werden, die das Verhalten der abstrakten Klassen implementieren.

Ein Beispiel für die von der Implementation und Wiederverwendung getrennte Subtypisierung ist die Beziehung zwischen den numerischen Typen der rationalen und ganzen Zahlen. In Smalltalk, NIHCL und libg++ wird *rational* als zusammengesetzte Integerzahl und damit als Schwester von *integer* in der Typhierarchie definiert. Diese Definition nutzt die Mächtigkeit der Wiederverwendung durch Zusammensetzung, gibt aber nicht das zu modellierende, ma-

[6] Abk. für *Object-oriented Abstract Type Hierarchy*

[7] Eine Einrückung bedeutet, daß die Klasse von der vorhergehenden Klasse erbt.

obj — Supertyp aller Objekte
- *pos* — Zeiger innerhalb eines Behälters
  - *listPos* — Zeiger innerhalb einer Liste
    - *stringPos* — Zeiger innerhalb einer Zeichenkette
- *bag* — allgemeine Behälterklasse, ungeordnete Menge mit Duplikaten
  - *set* — Menge
  - *queue* — Liste, auf der Löschen und Einfügen nur an erster bzw. letzter Stelle erlaubt ist
    - *seq* — Sequenz, bei der Einfügen oder Löschen den Index nicht verändert
      - *lifoQueue* — Queue mit lifo-Algorithmus
      - *fifoQueue* — Queue mit fifo-Algorithmus
        - *deq* — doppelt verkettete fifo-Queue mit zwei Enden
          - *list* — geordnete Liste
            - *string* — Zeichenkette als geordnete Liste von Zeichen
      - *ring* — zirkulare Queue
        - *grid* — Gitter
  - *table* — Tabelle
- *complex* — komplexe Zahlen
  - *real* — reelle Zahlen
    - *rational* — rationale Zahlen
      - *integer* — ganze Zahlen

Tabelle 4.6: Teilhierarchie von OATH

thematische Konzept wieder. Mathematisch gesehen ist *integer* ein Subtyp von *rational*. Die Hierarchie von OATH zeichnet diese Subtypbeziehungen zwischen den zu repräsentierenden Typen nach. Ein Vorteil ist der daraus resultierende Subtyp-Polymorphismus. Methoden, welche komplexe Zahlen verarbeiten, können genauso mit reellen oder ganzen Zahlen verwendet werden.

Ein weiteres Ziel dieser Hierarchie ist die robuste, automatische Speicherbereinigung (*garbage collection*) von Objekten. Diese ist innerhalb der portablen OATH-Klassenbibliothek implementiert, damit der Bibliotheksbenutzer die dynamischen Objekte nicht selbst verwalten muß.

libg++. Die GNU C++ Bibliothek *libg++* ist eine Sammlung von Klassen und Unterstützungswerkzeugen, die auf freiwilliger Basis mit Unterstützung des GNU Projekts der Free Software Foundation entstanden ist [Lea 88]. Diese Bibliothek wird mit dem GNU C++-Compiler *g++* benutzt. Daher ist sie nur soweit portabel, wie es der *g++*-Compiler ist.

Ein Ziel des Entwurfs und der Implementationsentscheidungen ist die Unterstützung verschiedener Programmierstile und -methoden, die wegen der Heterogenität von C++ mehr oder weniger objektorientiert sein können. Daher wird im Gegensatz zu NIHCL kein Klassenbaum erstellt, der das Konzept der Vererbung für alle enthaltenen Klassen ausnutzt, sondern die Klassen in libg++ bleiben generell unabhängig, um eine individuelle Auswahl und Benutzung zu ermöglichen. In diesem Klassenwald wird einfache Vererbung nur verwendet, wenn Klassen in ererbten Abhängigkeiten zueinander stehen.

Die Klassen von libg++ werden wie folgt kategorisiert:

Algebraische Typen sind zur Erweiterung von C++ mit zusätzlichen Basisdatentypen implementiert. Typen wie komplexe oder rationale Zahlen und Zeichenketten sind in den meisten Programmiersprachen fest eingebaut. Sie besitzen daher allgemein anerkannte Eigenschaften. Die Operationen dieser erweiterten Typen sollten dieselbe Syntax wie die Operationen der eingebauten Typen besitzen.

Behälterklassen sind abstrakte Datentypen, welche mehrere Elemente desselben Typs beinhalten. Zu den gekapselten Methoden zählen Einfügen, Löschen und das Untersuchen von Elementen mit Hilfe von Iteratoren.

Vektor-ähnliche Klassen verhalten sich zum einen wie algebraische Typen und zum anderen wie Behälterklassen, weil sie konstruktive Operatoren wie + und den Test, ob ein bestimmtes Element in der Klasse enthalten ist, gleichermaßen beinhalten.

Dienste zur Speicherverwaltung sind allgemein verwendbare Klassen, die den Benutzer der Bibliothek bei der Speicherverwaltung unterstützen. libg++ baut auf der Tatsache auf, daß die Programmierer effizienter als der Compiler oder das Laufzeitsystem die Speicherverwaltung vornehmen können.

Dienstklassen sind zur Kapselung einer bestimmten Menge an Diensten entworfen und haben einen für die benutzende Klasse irrelevanten, internen Zustand. Dies sind z. B. *Stream*-Klassen, Zufallszahlengeneratoren und andere anwendungsorientierte Klassen.

libg++ ist besonders für weniger umfangreiche Anwendungen effizient, in denen keine rein hierarchische Programmierumgebung gefordert ist. Für größere Softwareprojekte kann libg++ als fertige Basis einer größeren Hierarchie verwendet werden.

Booch Components. Die C++ Booch Components von G. Booch [Booch, Vilot 90] sind eine Übersetzung der Ada Booch Components nach C++. Sie sind für eine kommerzielle Nutzung erstellt und daher sowohl für verschiedene Unix-Systeme, als auch für PC- und Apple Macintosh-Systeme erhältlich [Stal, Berger 93b]. Ein Handbuch enthält den gesamten Klassenkatalog. Die Entwurfsstrategie beruht auf der objektorientierten Entwurfsmethode (*OOD*) von G. Booch [Booch 91].

Die Klassen der Booch Components gliedern sich in die vier Kategorien *Support*, *Tools*, *Structures* und *Tests* [Stal, Berger 93b].

Support enthält grundlegende Klassen für die Speicherverwaltung, Fehlerbehandlung, Prozeßsynchronisation und weitere elementare Klassen zur Realisierung von Behälterklassen. In der Version 1.4.7 gehören dazu:

Bound — elementare, auf dem Stack allozierte Listen
Except — Abstraktionen für die Fehlerbehandlung
Free — Freispeicherverwaltung
Heap — Implementierung stack-basierter Heaps
Node — Knotenklassen für Behälter
Shared — Referenzzählmechanismus
Synch — Prozeßsynchronisation
Unbound — elementare, auf dem Heap allozierte Listen
Vector — stack-basierte Vektoren

In der Version 2.1 werden Verfahren zum Bereitstellen von Persistenz hinzugefügt.

Tools realisieren die fundamentalen Algorithmen, die sich in Filterprogramme zum Bearbeiten von Datenströmen, Pattern-Matching-Algorithmen, Algorithmen zu Zeit- und Datumsangaben sowie Such- und Sortieralgorithmen einteilen lassen.

Structures besteht aus den Behälterklassen, die den wichtigsten Bestandteil der Bibliothek darstellen. Die Abstraktionen umfassen:

Bags — Sammlung von Elementen mit Wiederholung
Collections — allgemeine Behälter
Ordered_Collections — allgemeine Behälter mit Ordnung der Elemente
Deques — Double-Ended-Queues: Elemente dürfen vorn und hinten entnommen oder angefügt werden
Maps — Assoziatives Feld aus Schlüssel- und Elementtyp
Queues — FIFO-Puffer: Elemente werden vorn angefügt und hinten entnommen
Rings — Ringpuffer: Elemente werden vorn angefügt und entnommen
Sets — Sammlung von Elementen ohne Wiederholung
Stacks — Kellerspeicher
Strings — Folge von Elementen
Graphs — Struktur aus Kanten und Knoten, Zyklen sind erlaubt
Lists — Sequenz von Elementen
Trees — Struktur aus Kanten und Knoten, Zyklen sind nicht erlaubt
AVL_Trees — Bäume, nach dem Verfahren von Adelson-Velskii und Landis ausgeglichen

Queue — FIFO-Puffer
- *Bounded_Queue* — FIFO-Puffer auf dem Stack
  - *Guarded_Bounded_Queue* — *Bounded_Queue* mit Semaphorsynchronisation
  - *Synchronized_Bounded_Queue* — *Bounded_Queue* mit Monitorsynchronisation
- *Unbounded_Queue* — FIFO-Puffer auf dem Heap
  - *Guarded_Unbounded_Queue* — *Unbounded_Queue* mit Semaphorsynchronisation
  - *Synchronized_Unbounded_Queue* — *Unbounded_Queue* mit Monitorsynchronisation

Tabelle 4.7: Teilbaum der Booch Components für die Datenstruktur *Queue*

Die Behälter *Graphs*, *Lists* und *Trees* unterstützen strukturelle Referenz, welches bedeutet, daß zum Kopieren von Objekten eine *Shallow Copy*-Methode genügt. Alle anderen Behälterklassen liefern Kopien ihrer Unterstrukturen, so daß sie mit einem *Deep Copy*-Mechanismus Kopien anfertigen müssen.

Tests definiert spezielle Klassen zum Testen der *Structures* und *Tools*.

Es wird mit den Booch-Komponenten nicht genau die Hierarchie der Smalltalk-Umgebung nachgebildet, aber die Bibliothek NIHCL diente ebenso als Vorbild wie die libg++-Bibliothek.

In der neueren Version 2.1 der Booch Components wird den Deklarationen ein Präfix vorangestellt, um Namenskollisionen zu vermeiden.

Die Ausnahmebehandlung wird noch nicht von allen C++-Compilern geboten. Daher wird in den Booch Components ein Baum von Ausnahme-Klassen implementiert, wobei jeder Fehlerart eine Klasse zugeordnet wird. Beim Auftreten einer Ausnahmebedingung löst die Bibliothek eine entsprechende Ausnahmebehandlung aus. Diese ist vom Programmierer in einer *catch*-Anweisung zu behandeln, andernfalls kommt es zu einem Programmabbruch.

Um eine sorgfältig entworfene Sammlung von Bibliotheksklassen mit nützlichen Datenstrukturen zu erhalten, wurden hauptsächlich Effizienz, Einfachheit der Benutzung, Erweiterbarkeit und Adaptierbarkeit als Ziele verfolgt. Die strikte Trennung von Schnittstelle und Implementation wird konsequent durchgeführt. Diese Bibliothek unterstützt die nebenläufige Benutzung der Klassen und reflektiert dadurch den Trend zum Multiprozessing. Jede Datenstruktur wird auf verschiedene Arten implementiert, um verschiedene Komplexitätsanforderungen erfüllen zu können. So kann der Bibliotheksbenutzer die passende Datenstruktur für seine Anwendung wählen, die den gegebenen Rahmenbedingungen entspricht.

Als Beispiel erhält man zu der Klasse *Bounded_Queue* die Unterklasse *Guarded_Bounded-_Queue* mit der Verwendung von Semaphoren und die Unterklasse *Synchronized_Bounded-_Queue* mit der Verwendung von Monitoren in einer Multithread-Umgebung. Den verschiedenen Anforderungen an die Speicher- und Laufzeiteffizienz werden die Booch Components ebenfalls durch Ausprägung mehrerer Unterklassen zu derselben Oberklasse gerecht. *Unbounded_Queue* ist eine Unterklasse von *Queue*, die den Heap zum Speichern der Objekte benutzt, während *Bounded_Queue* ein auf dem Stack gespeichertes Feld fester Länge verwendet. Es ergibt sich für den Datentyp *Queue* ein Teilbaum des Klassenwaldes, der in Tabelle 4.7 dargestellt wird [Stal, Berger 93b].[8]

[8] Eine Einrückung bedeutet, daß die Klasse von der vorhergehenden Klasse erbt.

Zum Iterieren über die Elemente einer Behälterklasse stehen eigene Iteratorklassen zur Verfügung. Eine Iteratorklasse ist als *friend* der jeweils zugehörigen Behälterklasse definiert. *Aktive* Iteratoren dienen dabei zum Traversieren der Elemente im Behälter. *Passive* Iteratoren erlauben die Anwendung einer vorgegebenen Funktion auf alle Elemente des Behälters.

Die Bibliothek erfüllt die eigenen, gestellten Ziele [Booch, Vilot 90], da sie

- effizient ist, weil nur wenig zusätzliche Zeit gegenüber handkodierten Versionen in der Ausführung der Anwendungen verbraucht wird;
- leicht benutzbar ist, weil die implementierten Varianten der grundlegenden Datenstrukturen eine durchschaubare Organisation bieten;
- erweiterbar ist durch die Verwendung von Vererbungsmechanismen, die eine Orthogonalität der Bibliothek sichern;
- anpaßbar ist, weil die Anbindung an die Umgebung durch explizite Schnittstellen geschieht.

Zusammenfassung. Die wichtigsten Eigenschaften der vorgestellten C++-Bibliotheken werden in der Tabelle 4.8 dargestellt, um einen Überblick zu geben.[9]

4.5 Tycoon

Für das Tycoon System werden zunächst diejenigen Konzepte vorgestellt, welche die Strukturierung von Anwendungsprogrammen über deren Teilkomponenten hinweg ermöglichen. Die heutzutage entwickelten, umfangreichen Softwaresysteme fordern eine möglichst überschaubare Verwaltung ihrer Teilprogramme. Darauf aufbauend werden in einem weiteren Abschnitt die aktuellen Bibliotheken mit ihren Schnittstellen vorgestellt. Weiterhin werden die für TL bekannten Kapselungskonzepte kurz erläutert und in ihrer Verwendung für eine Standardbibliothek eingeschätzt.

4.5.1 Programmierung im Großen

Um Programmierung im Großen zu ermöglichen, müssen Konsistenzbedingungen auch über Systemkomponentengrenzen hinweg erzwungen werden. Besonders wichtig ist dabei das Konzept der separaten Kompilierung, welches verlangt, daß der Übersetzungsaufwand für ein Modul nur proportional zu der Komplexität dieses Moduls und nicht proportional zur Komplexität des Gesamtsystems wachsen darf.

Ebenso bedeutend ist die Lokalisierbarkeit von Information, welche die Typkorrektheit bestimmt. Wird die Lokalisierbarkeit nicht gewährleistet, dann ist das Typsystem nicht für die Systemprogrammierung geeignet. Ein Ansatz zur Vermeidung von Typunsicherheit für Eiffel in [Meyer 89] ist ein Beispiel für die Nicht-Lokalisierbarkeit [Matthes 93].

In TL werden die Konzepte von Modul, Modulschnittstelle und Bibliothek (*module, interface, library*) zur Modularisierung verwendet. Diese separaten Kompilationseinheiten unterstützen die Entwicklung großer Softwaresysteme. Module kapseln vordefinierte und externe

[9] Freie Felder besagen, daß die entsprechende Eigenschaft ungewiß ist, da sie nicht aus der vorliegenden Literatur entnommen werden konnte.

Eigenschaft	NIHCL	ET++	LEDA	OATH	libg++	Booch
Hierarchie	Baum	Baum	Wald	Baum	Wald	Wald
Vererbung	einfach, für benutzerdefinierte Klassen mehrfach	einfach	einfach	einfach, starke Subtypisierung	einfach	mehrfach
Portierbarkeit	aufwendig	ja	mehrere Plattformen verfügbar		nur für g++-Compiler	ja
Ausnahmebehandlung	ja, aber ineffizient	nein	nein, aber Vorbedingungen		nein, nur einfache Fehlerfunktion für jede Klasse	ja
Generizität	nur mit Makro-Prozessor	nein	ja		nur mit Makro-Prozessor	ja
Iteratoren	ein globaler	stabil	einfach		ja	aktive und passive
Dokumentation	nein, nur im Programmcode	ja, online	ja	nein	nein	ja
Basistypen als Klasse	ja, ohne arithmetische Operationen	ja		ja		nein

Tabelle 4.8: Zusammenfassung der Eigenschaften von C++-Bibliotheken

Systemfunktionalität, welches die Spracherweiterbarkeit und Portabilität von TL Anwendungen unterstützt [Matthes 93]. Gegenüber den anderen in dieser Arbeit zitierten Sprachen, welche kein explizites Bibliothekskonzept vorweisen können, ist dies ein großer Vorteil für den Anwendungsprogrammierer.

Eine Modulschnittstelle ist ein benannter Tupeltyp, der eine *import*-Klausel enthalten kann, um sich auf explizit importierte Module und Schnittstellen in einem globalen Sichtbarkeitsbereich zu beziehen.

Ein Modul definiert einen Tupelwert, der Bindungen gemäß der Signaturen seiner Schnittstelle aggregiert und sich ebenfalls auf importierte Module und Schnittstellen beziehen kann. Es entspricht einer mit den importierten Modulen parametrisierten Funktion. Durch diese Äquivalenz kann der Modulrumpf auch nicht-exportierte Bindungen und seiteneffekterzeugende Initialisierungsoperationen für das Modul durchführen. Die Reihenfolge der exportierten Bindungen muß nicht mit der Reihenfolge in der Signatur übereinstimmen. Für einen Entwickler von Modulen kann es allerdings nur hilfreich sein, wenn die Reihenfolgen in Signatur und Modulrumpf übereinstimmen.

Eine Bibliothek definiert den Sichtbarkeitsbereich für Modul- und Schnittstellennamen. Die Reihenfolge dieser Namen ist signifikant, da ein Modul nur importiert werden kann, wenn es örtlich vorher deklariert wurde. Zyklische Abhängigkeiten von Modulen sind in TL unzulässig, um die sequentielle Initialisierung zu gewährleisten und einen Zugriff auf uninitialisierte Wertbindungen in importierten Modulen zu verhindern.

Bezeichner müssen innerhalb einer Bibliothek eindeutig sein. Diese Einschränkung erlaubt eine Lösung des Problems des Rautenimports [Cardelli 89; Milner et al. 90] durch Rückführung auf die Namensäquivalenzregeln für abstrakte Typvariable [Matthes 93]. Ein Beispiel für TList schematisch in Abbildung 4.3 dargestellt[10]. Im Modul *d* kann von der Gleichheit der Typbezeichner *a.T* in den Sichtbarkeitsbereichen von *b* und *c* auf die Gleichheit der gebundenen Typausdrücke geschlossen werden, so daß für die Funktionsapplikation *c.f(b.y)* kein Typkonflikt auftritt.

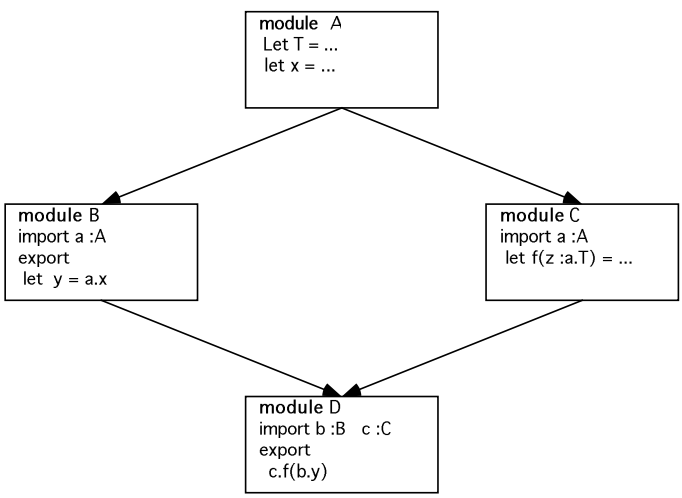


Abbildung 4.3: Rautenimport mit eindeutiger Typherleitung

[10] Aus Gründen der Vereinfachung wurden die jeweils zugehörigen Modulschnittstellen weggelassen.

Innerhalb der Bibliotheksdefinition werden Schnittstellen- und Modulbezeichner durch das Schlüsselwort import an Werte gebunden. Eine beispielhafte Importklausel

import *:Int :List*

bindet die Schnittstellenbezeichner *Int* und *List* an Tupeltypen, die den Schnittstellensignaturen entsprechen. Eine Importklausel für Module

import *int list*

bindet die Modulbezeichner *int* und *list* an Tupelwerte.

Bibliotheken, auf die von verschiedenen Benutzergruppen zugegriffen wird, beziehen sich auf gemeinsame Module, Schnittstellen und Bibliotheken. Um die *ad hoc*-Mechanismen in der Programmierung zu vermeiden, werden Bibliotheken in TLhierarchisch in Subbibliotheken gegliedert [Matthes 93]. Als Beispiel dient hier die Wurzel aller Bibliotheken, *Root*, und die zugeordnete Bibliothek *bulkenv*.

```
library Root
with
  library stdenv bulkenv
  interface Test
  module test
end

library bulkenv
import arrayOp :ArrayOp  list :List  iter :Iter
with
  interface Set Bag Assoc Dictionary VarList
  module linkedSet :Set bitSet :Set hashedSet :Set
  module assoc :Assoc varList :VarList bTree :Set
  hide varList VarList
end
```

Die Deklaration einer Bibliothek als Komponente einer anderen Bibliothek macht alle ihre Modul- und Schnittstellenbezeichner in der umfassenden Bibliothek sichtbar. Auch die in der *hide*-Klausel genannten Bezeichner werden global sichtbar, können aber nicht von nichtlokalen Modulen importiert werden. Es ist zu berücksichtigen, daß Bibliotheken mit Importlisten nur in Bibliotheken mit kompatiblen Modul- und Schnittstellenbindungen deklariert werden dürfen. Im Beispiel werden die von *bulkenv* erwarteten Bindungen

array :ArrayOp list :List iter :Iter

in der Bibliothek *stdenv* definiert, die daher in der Komponentenliste der Bibliothek *Root* vor der Deklaration von *bulkenv* stehen muß.

Die Eindeutigkeit von Modul-, Schnittstellen- und Bibliotheksbezeichnern innerhalb einer Bibliothek erleichtert die Reorganisation von Bibliotheken und vermeidet unerwünschte Übereinstimmungen von Modul- und Schnittstellennamen. Zusätzlich hilfreich bei der Reorganisation ist auch die Möglichkeit der Schachtelung von Bibliotheken. Andererseits führt diese Einschränkung in der Namensgebung zu einer restriktiveren Namenskonvention, in der festgelegt wird, daß Schnittstellennamen mit Großbuchstaben, Modulnamen mit Kleinbuchstaben beginnen sollten.

4.5.2 Aufbau der Bibliotheken

Im Tycoon System werden verschiedene Bibliotheken zur Verfügung gestellt, die jeweils einen Anwendungsbereich abdecken sollen. Diese Bibliotheken sind bedingt portierbar, weil sie ein lauffähiges Tycoon System erfordern. Dieses ist für das Unix-Betriebssystem SunOS entwickelt worden und wird zur Zeit im Rahmen von Diplomarbeiten am Arbeitsbereich DBIS auf das Macintosh Betriebssystem [Breilmann 95] und MS-Windows für PCs [Shen 96] portiert, so daß die Bibliotheken ebenfalls auf diesen Plattformen zum Einsatz kommen.

Innerhalb der verschiedenen Tycoon Bibliotheken ist die Struktur abhängig von ihrem Anwendungsbereich. Daher stehen in der Standardbibliothek viele Module unassoziiert nebeneinander, weil grundlegende Datenstrukturen und Methoden für andere Bibliotheken geliefert werden und nur wenig innerer Zusammenhang besteht.

In der jetzigen Version der Tycoon-Bibliotheken existieren neben zahlreichen anwendungsspezifischen Bibliotheken die folgenden, allgemeinen Bibliotheken:

- stdenv: Iter, IterRep, ArrayOp, Ascii, Time, Optional, sowie weitere Schnittstellen für Ein-/Ausgabe, Standardtypen, Basisdatenstrukturen
- bulkenv: Schnittstellen für Massendatentypen wie z. B. Tree, BitSet, Dictionary, Graph, HashTable, List, Queue, Set, Stack
- machineenv: maschinenabhängige Schnittstellen wie z. B. Address, CPUTime, Date, Directory, Image, Process, Store, UserTime, UxEnvironment, UxFile, Word

Die Module in der Standardbibliothek *stdenv* werden in dieser Arbeit unter dem Gesichtspunkt eines guten Entwurfs betrachtet. Dabei kommt es zum einen zu Überschneidungen mit den Behältern der *bulkenv*, weil eine Zeichenkette sowohl als Basisdatentyp, als auch als ein Behälter von mehreren Zeichen angesehen werden kann und damit die Frage aufwirft, ob die Schnittstelle *String* in der *stdenv* oder *bulkenv* deklariert werden muß. Die gleiche Frage entsteht für die Schnittstelle *ArrayOp*. Zum anderen entsprechen einige Module einem bestimmten Zweck, der anderen Bibliotheken wie der *machineenv* ähnlich ist. Daher ist aus Gründen der Vollständigkeit hier eine Übersicht über diese drei Bibliotheken gegeben, obwohl die *machineenv* und die *bulkenv* im Rahmen dieser Arbeit nicht überarbeitet werden.

Standardumgebung

Die Sprache TL selbst benötigt einige Datenstrukturen, um zur Übersetzungszeit Auswertungen vornehmen zu können. So wird beispielsweise der Datentyp *Int*.T für Feldindizes benutzt, wogegen *Bool*.T in bedingten Abfragen nötig ist. Auch für die Auswertung von Spracherweiterungen, wie z. B. das *case*-Konstrukt, werden die in die Sprache eingebauten Datentypen benötigt.

Aus diesem Grund bilden folgende ***Basisdatenstrukturen*** in einer standardmäßigen Umgebung den minimalen Umfang von TL, indem sie fest in der Sprache verankert sind.

Bool	Typ von Wahrheitswerten
Char	Typ von Zeichenliteralen
Int	Typ von Integerzahlen, Argumenttyp für for-Schleifen
Real	Typ von Floatingpointzahlen
Longreal	Typ von Floatingpointzahlen, deren Genauigkeit erweitert ist

CString	Typ von Zeichenketten in C, die mit der Zeichenbegrenzung von *Strings* in der Sprache C terminieren
String	Typ von Zeichenketten in TL, auch zur Konstruktion der Ausnahmewerte benötigt
Array	Typ von Feldern mit veränderbaren Elementen Arrays sind eine Spezialisierung von Vector
Word	Typ der Worte der Zielmaschine (maschinenabhängig), meist 32 Bit unsigned integer
Locality somewhere	die "don't care" Locality, wenn der Benutzer keine Lokalität bestimmt hat

Mit den folgenden Funktionen werden Operationen auf diesen Basisdatenstrukturen in ein Tycoon-Programm eingebunden:

builtin	Rückgabe einer eingebauten Funktion, z. B. "Int ="
bind	Externe Bindung zu einer C-Funktion.

Standardbibliothek

In ***stdenv*** sind alle fundamentalen Schnittstellen zusammengefaßt. Diese Bibliothek wird automatisch für jede Anwendung importiert, so daß jeder Anwender die grundlegenden Konzepte zur Verfügung hat. Es werden in den folgenden Abschnitten sämtliche Schnittstellen dieser Bibliothek erläutert. Zu jeder Schnittstelle existiert eine passende Modulimplementation, die in der Regel den gleichen Namen in kleiner Schreibweise trägt.

Basisdatentypen — Entsprechend dem tiefgreifenden *Add-On*-Konzept von Tycoon werden möglichst alle Operationen auf den Basisdatenstrukturen und weitere, grundlegende Datentypen in Bibliotheken zur Verfügung gestellt [Matthes 93]. Aus diesem Grund werden die entsprechenden Schnittstellen und zugehörigen Module auf der Basis der Standardumgebung in der Standardbibliothek bereitgestellt. Ein ähnliches Konzept wird in Eiffel realisiert.[11] Die Standardbibliothek im Tycoon System umfaßt folgende Schnittstellen zur Deklaration von Basisdatentypen:[12]

ArrayOp	Array fester Größe mit veränderlichem Zustand
BoolOp	Operationen auf boolschen Werten
CharOp	erweiterter ASCII-Zeichensatz mit Prädikaten und definierter Ordnung
IntOp	Operationen auf Integer
RealOp	Operationen für Floating Point Zahlen mit geringer Genauigkeit
StringOp	Zeichenketten mit veränderlichem Zustand

Ein- und Ausgabe — Als weiteres Konzept, welches in fast allen Programmiersprachen zu finden ist, enthält *stdenv* Schnittstellen zur Ein- und Ausgabe von Daten während der

[11] Die Implementation erfolgt dort in den eingebauten Klassen *integer_ref*, *character_ref* usw. Die vom Benutzer weiterzuverwendenden Klassen *integer*, *character* usw. bilden eine Schnittstelle zu diesen versteckt deklarierten Klassen, um eine einheitliche Oberfläche herzustellen [Meyer 90b].

[12] Die Postfixe *Op* dieser Schnittstellen vermeiden Namenskonflikte mit den Schlüsselwörtern und den Datentypen der Standardumgebung.

Programmausführung. Die Ein- und Ausgabe von Daten nimmt einen Sonderstatus ein. Sie bildet die Schnittstelle zwischen verschiedenen Programmen sowie den Programmen und dem menschlichen Benutzer. Daher stößt die rein abstrakte Sichtweise der Programmiersprache auf die kompromißreiche, reale Welt. Aus diesem Grund existieren verschiedene Standardmethoden zur Datenein- und -ausgabe. TLsieht die Grundlage der Ein- und Ausgabe in Streams. Hierzu existieren die Schnittstellen:

Print	formatierte Ausgabe auf writer.stdout
Reader	Lesezugriff auf Zeichenstrom
Writer	Schreibzugriff auf Zeichenströme

Formatierung, Konvertierung — Die Umwandlung von einem Datentyp in einen anderen ist insbesondere für Anwendungen mit Ein- und Ausgabe von Daten hilfreich, da die Schnittstelle meist nur Daten vom Typ *string*.T verarbeiten kann. Aus diesem Grund besteht ein Bedarf an Konvertierungsroutinen von allen Standarddatentypen in den Typ *string*.T. Konvertierungen in andere Datentypen werden ebenfalls in der folgenden Schnittstelle deklariert:

Fmt	formatierte Ausgabe der Standarddatenstrukturen

Iteration — Zur Iteration über Massendatentypen existiert eine Abstraktion in Form von Iteratoren.

Iter	generische Operationen für Iterationen über homogene Folgen
IterRep	einheitliche Repräsentation für verschiedene Iteratoren

Weitere Schnittstellen — Sie dienen der Bereitstellung allgemeiner Konzepte, die für viele Anwendungen nützlich scheinen. Dieses sind:

Ascii	Standardnamen für ASCII-Kontrollzeichen
Dynamic	Operationen auf Objekt-Typ-Paaren
Fingerprint	Abbildung von Bytestrings auf 64Bit-Hashvalues, die eine Verteilung für Quellcode-ähnliche Eingabe darstellt
Main	Vorlage für ein Hauptprogramm mit Kommandozeilen-Argumenten
Optional	unveränderliche, optionale Werte
Order	Operationen für eine lineare Ordnung
Pair	inhomogene, unveränderliche Paare von Werten
Poly	polymorphe Funktionen und Werte
RuntimeCore	Information über das Tycoon Runtime System, z. B. Pfadnamen für Bibliotheken
Scan	Zeichenfolgenzugriff mit einem Zeichen Vorausschau
Time	Systemzeit

Bibliothek von Massendatenstrukturen

Die Bibliothek ***bulkenv*** enthält in der aktuellen Version Schnittstellen zur Kapselung von Massendatenstrukturen sowie einige Eigenschaften, die für diese spezielle Art von Datenstrukturen nützlich sind. Folgende Schnittstellen gehören dazu:

Assoc	polymorphe, binäre Verbindungen mit veränderlichem Zustand
BTree	polymorpher, ausgeglichener Baum ohne Duplikate mit
BTreeDelete	Löschoperationen,
BTreeElements	Iterationen,
BTreeInsert	Einfügeoperationen,
BTreeReorg	Operationen zur Reorganisation und
BTreeUtil	Hilfsfunktionen.
BitSet	Bitvektoren fester Länge
BndStack	veränderlicher Stack mit fester Kapazität
Cache	Cache von willkürlichen Daten mit Ersetzungsstrategien
Cell	polymorphe, veränderliche Listen
Dictionary	polymorphes Dictionary
DirectedGraph	parametrisierte Graphstruktur
DynArray	Array mit dynamisch wachsender Größe
DynBitSet	Bitmengen mit dynamischer Größe
Environment	Sprachumgebung mit veränderlichen Variablen
Graph	wiederverwendbare Algorithmen für *DirectedGraph*
HashTable	polymorphe, veränderliche Hashtabelle
List	polymorphe Liste im funktionalen Stil
Position	Positionen in geordneten Datenstrukturen
Queue	polymorphe FIFO-Liste
Set	polymorphe Mengen mit veränderlichem Zustand im funktionalen Stil
Stack	polymorphe, veränderliche Stacks im funktionalen Stil
VarList	polymorphe, veränderliche Liste

Maschinennahe Bibliothek

Die Bibliothek ***machineenv*** enthält folgende Schnittstellen, um die maschinenabhängigen Implementationen des Tycoon Systems an zentraler Stelle zusammenzufassen:

Address	Hauptspeicherzugriff auf den Massenspeicher (engl. *heap*)
Audio	Ein- und Ausgabe von Audiodaten
CCallback	Callbacks von C nach Tycoon
CPUTime	Timer
Date	Darstellung des Datums nach dem gregorianischen Kalender
Directory	Dateinamenverwaltung
Image	Bilddatendarstellung auf Sun Sparc Workstations
LongIntOp	Ganzzahlenarithmetik mit mehrfacher Präzision
Process	Ausführung von Betriebssystem-Subprozessen
Store	Sichere Schnittstelle zum Speicher
UserTime	Darstellung der Zeit
UxEnvironment	Zugriff auf die Unix-Umgebung von Variablen
UxFile	Operationen auf Dateien im Unix-Format
Word	32-Bit-Maschinenwort als Bitmuster oder Zahl ohne Vorzeichen
Word2	Ganzzahl als Bitmuster oder Zahl ohne Vorzeichen

Daneben existiert ein weiterer Vorschlag von F. Matthes für eine Version der Bibliotheken, die sich vor allem durch die Definition eines Domain Descriptors, einige Umbenennungen und neue Zuordnung von Modulen zu Bibliotheken von der aktuellen Version unterscheidet:

- StdLib: Iter, IterRep, Domain, sowie weitere Klassen für I/O, Standardtypen, Datenstrukturen
- BulkLib: Bag, BitSet, DynArray, Relation, Sequence, Sort, Stack, Table u.a.
- MachineLib: Address, Data, Foreign, Process u.a.
- NumericLib: BitVector u.a.

Diese Module, insbesondere in der Standardbibliothek, werden in Kapitel 5 auf ihre allgemeine Verwendbarkeit hin untersucht. Dabei sind die gegenseitigen Abhängigkeiten zu berücksichtigen.

4.5.3 Kapselungskonzepte in TL-Bibliotheken

Innerhalb des Tycoon Systems steht dem Benutzer die Programmiersprache TL zur Verfügung, die in Abschnitt 3.2.5 näher erläutert wurde. Die Flexibilität von TL ermöglicht die Programmierung in verschiedenen Modellierungsstilen [Münnix 94], die im folgenden kurz dargestellt werden. Jedes Kapselungskonzept zieht einen eigenen Programmierstil nach sich. Der Programmierer, der gern funktionale Programme schreibt, wird Schwierigkeiten haben, ein Modul in sein Programm zu integrieren, welches nach dem imperativen Kapselungskonzept implementiert ist. Dies gilt entsprechend auch für andere Kapselungskonzepte. Die Probleme, welche verschiedene Modellierungsstile innerhalb einer Bibliothek mit sich bringen, werden anschließend erläutert.

Die TL Basiskonstrukte zur Benennung, Bindung und Typisierung können benutzt werden, um die klassischen Konzepte der abstrakten Datentypen und der objektorientierten Kapselung zu modellieren. Durch unterschiedliche Kombination der TL Kernprimitive kann diese Realisierung auf drei Weisen erfolgen. Diese Kapselungskonzepte werden als *funktional*, *imperativ* und *methodenbasiert* bezeichnet [Wetzel 94].

In TL sind die beiden erstgenannten Kapselungskonzepte durch direkt in die Sprache integrierte, funktionale und imperative Sprachkonstrukte modellierbar. Ein abstrakter Datentyp läßt sich durch Bereitstellung eines opaken Typs sowie darauf arbeitenden Methoden charakterisieren. Diese Methoden können entweder funktional, welches das *funktionale* Kapselungskonzept realisiert, oder zustandsbasiert für das *imperative* Kapselungskonzept implementiert sein. Die sprachliche Neutralität von TL ist so groß, daß zusätzlich eine Art der objektorientierten Programmierung in TL modellierbar ist. Für die *methodenbasierte* Kapselung werden Methoden aggregiert, die auf einen nach außen verborgenen Zustand zugreifen, der nur intern verändert werden darf. Am Beispiel des Behälterdatentyps *Stack* werden diese drei Techniken in [Wetzel 94] dargestellt.

In der *funktionalen Kapselung* wird bei Änderungsoperationen die geänderte Datenstruktur als Ausgabewert zurückgegeben. Dies ist an der Signatur der Änderungsoperation sichtbar, die einen Rückgabewert vom selben Typ wie der Eingabeparameter definiert. Beim Aufruf der Änderungsoperation muß daher das Operationsergebnis einer Variablen zugewiesen werden.

In der *imperativen Kapselungstechnik* werden die Methoden zustandsbasiert gekapselt. Die Änderungsoperationen ändern den Zustand des übergebenen Objekts und besitzen daher

nur **Ok** als Ausgabetyp. Das übergebene Objekt ist nach erfolgter Ausführung verändert. In der Implementation erfolgt die Zustandsänderung über eine Zuweisung. Es entsteht ein Seiteneffekt, da im Aufruf nicht erkennbar ist, daß sich die in dem abstrakten Datentyp verborgene Variable geändert hat.

Die *methodenbasierte Technik* zur Kapselung wird auch *objektorientiert* genannt, da das Konzept der *Datenkapselung* ein charakteristischer Mechanismus der Objektorientierung ist [Booch 86; Cox 86; Meyer 88; Wegner 87]. Dabei werden die Zustandsvariable individueller Objekte durch Kapselung mit Hilfe der eigenen Methoden gelesen und geändert [Matthes 93]. Die methodenbasierte Kapselung stellt einen Tupeltyp über Funktionssignaturen zur Verfügung. Eine **new**-Funktion zur Erzeugung von neuen Objekten ist außerhalb dieses Typs zu definieren. Änderungen an diesen Objekten erfolgen über Funktionen, die mit dem Objekt gekapselt sind. Interne Variable stellen den aktuellen Zustand dar. Es gibt daher für die Operationen keine Eingabevariable. Auch muß im Gegensatz zu den beiden anderen Techniken kein opaker Typ definiert sein.

4.5.4 Bibliotheken und Kapselungskonzepte

Anwendungen werden in den verschiedenen Kapselungskonzepten erstellt. Für die Wiederverwendung von Modulen ist es wichtig, daß sie in den verschiedenen Kapselungskonzepten vorhanden sind. Nur so kann jeder Anwender, gleich welchen Stil er bevorzugt, einen vorhandenen Bibliotheksbaustein sofort in seinen Programmcode einbinden. Es ist daher zu untersuchen, für welche Bausteine es sinnvoll ist, verschiedene Realisierungsarten anzubieten. Dabei ist eine Lösung zu finden, in welcher Weise diese Module zur Verfügung gestellt werden. Außerdem muß festgestellt werden, ob Module verschiedener Programmierstile in einer Anwendung nebeneinander eingesetzt werden können.

Da die Bibliotheken für die Wiederverwendung und damit auch für einen einfachen Austausch der Konzepte durch Austausch der Bibliotheksmodule erstellt wurden, ist es nötig, sämtliche Massendatentypen in allen drei Stilen zur Verfügung zu stellen. Nur so wird es einem Anwender ermöglicht, innerhalb seiner z. B. imperativ programmierten Anwendung einen Behälter gegen einen anderen auszutauschen. Liegt das neu zu verwendende Datenstrukturkonzept nur funktional vor, so bedeutet das einen erheblichen Änderungsaufwand in der Applikation, da jeder Aufruf einer Operation dieses Behälters von der imperativen zur funktionalen Deklaration geändert werden muß.

Imperative Bibliotheksmodule Der Datentyp *Array* ist in der *set*-Methode imperativ deklariert. Diese Methode, die genau ein Element des Arrays ändert, ist die grundlegende Änderungsfunktion dieser Datenstruktur. Die weitere Änderungsfunktion *setSub* ist ebenfalls imperativ. Zusätzlich sind für *Array* aber auch funktionale Methoden deklariert, z. B. *sub*, die von ihrer Bedeutung her eine Extraktion vornehmen und daher auch eine Ausgabe eines Wertes vom gleichen Typ haben müssen.

Die Basisdatentypen *Int*, *Char*, *Real* und *Bool* sind von ihrem Konzept her nicht änderbar, da sie nur jeweils einen bestimmten Wert darstellen. Somit können sie auch nicht in diese Klassifikation einbezogen werden.

Die dem funktionalen Stil zuzuordnenden Schnittstellen stehen in der Massendatenbibliothek *bulkLib*. Die Massendatentypen sind in der Mehrzahl funktional deklariert, da die Sprache TLin ihrem Ursprung funktional angelegt ist. Ausnahmen davon bilden z. B. *Dictionary*, welches imperativ deklariert ist und *DirectedGraph*, welches methodenbasiert deklariert

ist.

4.6 Zusammenfassung der Anforderungen an Tycoon

Für objektorientierte Sprachen ergibt sich wegen ihrer Möglichkeit zur Vererbung eine Hierarchiestruktur innerhalb ihrer Bibliotheken. Die meisten von ihnen sind wie Smalltalk in einem Klassenbaum organisiert. Auch die Bibliotheken mit Klassenwald, wie z. B. libg++, bestehen in Teilen aus einer Baumhierarchie. Bei den nicht-objektorientierten Sprachen entsteht keine derartige Vererbungshierarchie. Diese Sprachen sind darauf angewiesen, daß die Implementierer ihre Bibliotheken selbst strukturieren, indem sie möglichst generelle Module erstellen, welche nach verschiedenen Anforderungskriterien in weiteren Modulen spezialisiert werden. Die Beziehungen zwischen verschiedenen Modulen müssen dabei deutlich in der Dokumentation hervorgehoben werden.

Um die Defizite der nicht streng hierarchisch gegliederten Bibliotheksstruktur und der bisher wenig ausführlichen Dokumentation für TLauszugleichen, wird ein *Style Guide* erstellt. Durch eine derartige Vereinheitlichung werden die Beziehungen zwischen den Bibliotheksmodulen deutlich. Weiterhin sind Datenstrukturen in einer Anwendung leicht austauschbar, wenn semantisch gleiche Operationen in allen Modulen gleich benannt sind. Der Anwendungsprogrammierer kann immer wiederkehrende Namen leichter merken und erkennt damit schneller die Semantik einer Schnittstelle. Darüber hinaus soll mit diesen Richtlinien die Portierung der Bibliotheken vereinfacht werden. Die in Smalltalk nicht konsequent verfolgte Benennung von Operationen mit einem Postfix zur Kennzeichnung einer speziellen Eigenschaft dieser Operation soll auch in TLeingeführt werden. Die Regeln sind in Abschnitt 5.5 niedergelegt.

Ähnlich wie in C++ sind einige Basisdatentypen, wie *Integer* und *Character*, in Tycoon eingebaute Basiskonstrukte. Um einen vollständigen *Add-On*-Ansatz für alle Datentypen zu verfolgen, werden, vergleichbar mit einigen C++-Bibliotheken, auf der Grundlage dieser Basisdatentypen eigene Module gebildet, damit die Verwendung der Typen *Integer* und *Character* sich nicht von anderen Bibliotheksdatentypen unterscheidet.

Die in Modula-3 für generische Schnittstellen geforderten Operationen sollen für TLin einem eigenen Modul *value* zur Verfügung gestellt werden.

Die Tabelle 4.9 zeigt in einer Zusammenfassung die bemerkenswertesten Konzepte der untersuchten Sprachen. Dabei werden auch die in Abschnitt 3.2 vorgestellten Sprachkonzepte berücksichtigt.[13]

Als Verbindung der Programmiersprache zur realen Welt existieren verschiedene Standardmethoden zur Datenein- und -ausgabe. Modula-3 nutzt spezialisierte Prozeduren, die nicht notwendigerweise Teil der Sprache sein müssen. TLsieht die Grundlage der Ein- und Ausgabe in Streams. Dieses bewährte Konzept soll für die zu entwerfende Bibliothek beibehalten werden.

[13] Für einen Überblick über die unterschiedlichen C++-Bibliotheken siehe Abschnitt 4.4 mit Tabelle ??.

	Modula-3	Eiffel	Smalltalk	C++ (mit Bibl.)	Tycoon
Sprachkonzepte					
Programmiermodell	imperativ	objekt-orientiert	objekt-orientiert	objekt-orientiert	funktional
Berechnungsmethode	sequentiell	sequentiell	sequentiell	sequentiell	sequentiell
Typisierung	strikt	strikt	untypisiert	strikt	strikt
Implementation	Compiler	Compiler	Semi-Compiler	Compiler	Interpreter
Modularisierung	explizit	implizit	implizit	implizit	explizit
Bibliothekskonzepte					
Bibliotheksstruktur	Menge	Netz	Baum	bibl.abhängig	Menge
Erweiterbarkeit	ja	benutzerdef. Klassen	nein	ja	ja
Portierbarkeit	gut	gut	beschränkt	breite Verfügbarkeit	bedingt
Generizität	ja	ja	nein	ja	ja
Vererbung	nein	ja	ja	ja (bibl. abhängig)	nein
Anwendungsumfang	allgemein	vielseitig	allgemein	bibl.abhängig	bibl.abhängig
Dokumentation	ja	ja	ja	bibl.abhängig	bibl.intern
Ausnahmebehandlung	sprachintern	sprachintern	Abbruch nach Fehler	bibl.abhängig	sprachintern
Iteratoren	nein	ja	nein	bibl.abhängig	ja
Basisdatenstrukturen	sprachintern, zusätzlich in Bibl.	bibl.intern	bibl.intern	sprachintern	eingeschränkt bibl.intern

Tabelle 4.9: Konzepte aller vorgestellten Sprachen im Vergleich

Kapitel 5

Überarbeiteter Entwurf der Tycoon-Bibliotheken

In modulorientierten Systemen werden mit der Definition jedes Typs Namenskonventionen definiert, die meist in einer getrennten Dokumentation zusammengefaßt sind. In objektorientierten Klassenbibliotheken wird durch Vererbung die Konvention von der Superklasse automatisch übernommen. Um eine Art *Datenwörterbuch* mit allen Namen und ihrer Bedeutung zu erstellen, sind Programmier- und Dokumentationsrichtlinien von Bedeutung. Aus diesem Grund wird in den folgenden Abschnitten dargestellt, in welcher Weise die in den vorhergehenden Kapiteln geforderten Konzepte im Tycoon System realisiert werden.

Zuerst wird die externe Sicht auf die Bibliotheken erläutert. Dazu gehört die Zusammenfassung der für Applikationen fundamentalen Module in verschiedenen Bibliotheken mit dem Ziel, sämtliche Module, die ähnlichen Inhalt haben und einem ähnlichen Zweck dienen, in einer Bibliothek zu vereinen. Anschließend werden die Spezifikationen einiger ausgewählter Module, die bereits im Abschnitt 3.3 besprochen wurden, in TL konkretisiert.

Im darauffolgenden Abschnitt wird die Ergänzung der Sprache TL mit dem in Abschnitt 3.3.4 vorgestellten Konzept der *Zusicherungen* dargestellt. Die Syntax in TL und deren Umsetzung wird beschrieben und Grenzen der gegenwärtigen Implementation aufgezeigt.

In einem weiteren Abschnitt werden als interne Sicht auf die Bibliotheken deren Inhalte untersucht. Hier wird dargestellt, welche Operationen für welche Module nötig sind und welche Konzepte hinter der Zuordnung von Operationen zu Modulen stehen. Die Operationen werden kategorisiert und mit einer einheitlichen Namensgebung belegt, damit eine einheitliche Schnittstelle der abstrakten Datentypen für den Benutzer entsteht.

Im letzten Abschnitt wird das Dokumentationskonzept erläutert. Es wird ein *Style Guide* (eine Anleitung mit Stilrichtlinien) erstellt, der die in Kapitel 2 aufgestellten Richtlinien berücksichtigt und eine einheitliche Benennung für alle Bibliotheken von TL gewährleisten soll, um gleiche und ähnliche Strukturkonzepte für den Anwendungsprogrammierer erkennbar zu machen.

5.1 Strukturierung der Bibliotheken

Ein Grundsatz des Entwurfs der Bibliotheken für Tycoon ist der *Add-On*-Ansatz, welcher zur Folge hat, daß alle Datentypen mit ihren Operatoren außerhalb des Sprachkerns deklariert werden sollen. Daher ist ein Ziel, sämtliche Datentypen, die nicht vom Compiler benötigt

werden, aus der Standardumgebung auszulagern und diese dadurch auf die grundlegenden Datentypen zu reduzieren.

Ebenfalls soll die Vordefinition von arithmetischen Operationen für den Datentyp *Int* nicht aufrecht erhalten werden. Da jedoch der Compiler die Datentypen *Int* und *Bool* zur Übersetzung von if-Abfragen und for-Schleifen benötigt, kommt diesen Typen eine Sonderrolle zu. Ihre arithmetischen Operationen können mit den Konstrukten *bind* und *builtin* angebunden werden, um die wechselseitigen Referenzen in den Modulen von *Real*, *Bool* und *Int* zu lösen.

Im Weiteren wird dargestellt, welche Module der TL-Bibliotheken inhaltlich zusammengehören. Im Vergleich mit anderen Sprachen wird eine minimale Konfiguration der Bibliotheken festgelegt. Ein Ziel ist die einfache Erweiterbarkeit, so daß die Module für möglichst alle Anwendungen wiederverwendet werden können. Zusätzlich soll eine Plattformunabhängigkeit erreicht werden, damit eine auf allen Plattformen gleiche Basis eine einfache Portierung von Applikationen ermöglicht. Aus diesem Grund sollen sämtliche Schnittstellen, welche plattformabhängig sind, in einer eigenen Bibliothek gehalten werden.

Es reicht nicht aus, daß die einzelnen Module der Bibliothek für ihre Wiederverwendung gut geeignet sind. Der zusammenhängende Entwurf einer Bibliothek wird dadurch charakterisiert, daß das Ganze, nämlich die Bibliothek, mehr als die Summe seiner Teile sein sollte [Meyer 94]. Jedes Modul muß an passender Stelle in der Bibliothek zu finden sein und eine zu anderen Modulen homogene Schnittstelle besitzen. Die Richtlinien zur Erstellung der Bibliotheken, welche in diesem Kapitel dargelegt werden, sollen einen zusammenhängenden, überschaubaren Entwurf gewährleisten.

Gerade in den Standardbibliotheken des Tycoon Systems lassen sich ebenso wie in anderen Sprachen wenig Gemeinsamkeiten zwischen den Schnittstellen und Modulen entdecken. In Tycoon sind die meisten von ihnen keine abstraken Datentypen (ADTs), weil die Datentypen aus technischen Gründen für den Sprachkern offengelegt werden müssen. An der Oberfläche sollen sie aber möglichst als ADTs dargestellt werden, so daß der Anwender von Bibliotheken alle angebotenen Datenstrukturen auf die gleiche Weise behandeln kann.

Die Standardmodule decken die verschiedensten Anwendungsbereiche, wie Basistypen der Sprache, Ein-/Ausgabe oder Dateiverwaltung, ab. Daher verfügen diese Module über die unterschiedlichsten Eigenschaften. Ein Ziel dieser Arbeit ist, trotz der Heterogenität einen gemeinsamen Rahmen zur Definition von Standardmodulen festzulegen.

Im Gegensatz zu den Standardmodulen sind die Module mit Behälterdatenstrukturen, wie die Massendatentypen *Set*, *Bag*, *Array*, *BTree* oder *List*, durch allgemein übliche Schnittstellen besser vergleichbar. Es gibt umfangreiche Literatur, in welcher die Konstruktion von Bibliotheken mit Massendatentypen sehr strukturiert dargestellt wird. Behälterklassen in Eiffel sind beispielsweise mit Hilfe der Taxonomie aus den drei Kriterien Zugriffsmethode, Speichermethode und Durchlaufmethode hierarchisch strukturiert [Meyer 90b]. Durch diese Hierarchie wird der Aufbau der Bibliotheken homogen.

Eine derartig einheitliche Struktur kann für die Basisbibliothek von TL wegen ihrer Vielschichtigkeit nicht erwartet werden. Im folgenden Abschnitt werden die in der Standardbibliothek (*baseLib*) zusammengefaßten Module verzeichnet. Da die Änderungen gegenüber der bisherigen Bibliotheksversion eng mit der Bibliothek der maschinenabhängigen Module (*machineLib*) und der Bibliothek der Behälterdatentypen (*bulkLib*) zusammenhängen, werden diese in einer kurzen Beschreibung ebenfalls für eine Überarbeitung vorgeschlagen. Daran anschließend folgt die genaue Beschreibung der *baseLib*-Modulschnittstellen *Date* und *Time*, weil sie im Vergleich zu ihrer aktuellen Version einschneidenden Änderungen unterzogen werden.

5.1.1 Spezifizierung der Bibliotheken *baseLib*, *machineLib* und *bulkLib*

Aus den im vorigen Abschnitt beschriebenen Grundsätzen ergibt sich ein Rahmen für die Zusammensetzung der Bibliotheken des Tycoon Systems. Als Grundlage werden die allgemein verwendeten Datentypen in der Bibliothek *baseLib* mit den Basistypen (*Integer*, *Real*, *Char* etc.), Modulen zur Ein- und Ausgabe sowie weiteren, allgemein verwendeten Modulen zusammengefaßt. Diese Datentypen sind für jede Anwendung, ebenso wie für andere Bibliotheken, insbesondere auch für die Behälterbibliothek, fundamental.

Darauf aufbauend gibt es eine weitere, portable Bibliothek *bulkLib*, welche sämtliche Module zur Modellierung von Behälterdatentypen beinhaltet.

Daneben enthält die Bibliothek *machineLib* Schnittstellen zum Betriebssystem und anderen, plattformabhängigen Diensten, so daß bei einer Portierung des Tycoon Systems auf andere Plattformen nur die Schnittstellen dieser Bibliothek angepaßt werden müssen.

baseLib. Die Module in der Basisbibliothek orientieren sich an der aktuellen Version. Die bereits vorhandenen Schnittstellen und Module werden auch in der überarbeiteten Version weitestgehend beibehalten, um eine einfache Portierung bereits bestehender Tycoon-Programme auf die neue Bibliothek zu gewährleisten.

In der Bibliothek *baseLib* sind im wesentlichen die in Tabelle 5.1 aufgezeigten Kategorien von Modulen enthalten (dazu werden jeweils die Namen der zugehörigen Schnittstellen aufgelistet).

Die Schnittstelle *Fmt* entfällt, da die Konvertierungsroutinen in der Schnittstelle für den jeweiligen Datentyp enthalten sind (siehe Abschnitt 5.2.3).

Ein Modul *value* wird neu erstellt und enthält allgemeine Funktionen, die auf beliebigen Werten operieren und daher für alle Datentypen verwendet werden können. Dazu gehören die Kopier- und Vergleichsfunktionen, welche jeweils für die Äquivalenz und die Identität zur Verfügung gestellt werden. Die Namen *copy*, *clone* und *equal* werden vom Benutzer vergeben und bezeichnen die jeweils für die Datenstruktur geltende Funktion, welche mittels der in *Value* definierten Operationen implementiert werden kann. Die Operationen *ordinal*, *identity*, *identical* und *notIdentical* der Schnittstelle *Poly* aus der aktuellen Bibliotheksversion werden in *Value* übernommen, um eine Kompatibilität für bestehende Anwendungsprogramme herzustellen. Weitere Funktionen für *Value* werden in Abschnitt 5.2.5 erläutert.

Die Module *array* und *string* können aufgrund ihrer Eigenschaften auf verschiedene Weise in die Bibliotheken eingebracht werden. Zum einen sind Felder und Zeichenketten Datenstrukturen, die einen Elementtyp besitzen und eine "Sammlung" mehrerer Elemente darstellen. Dies wäre ein Grund, sie in die Bibliothek der Behälterdatentypen *bulkLib* einzubringen. Zum anderen stellen sie derartig grundlegende Konzepte zur Verfügung, daß sie in den meisten Programmiersprachen als eingebaute Datentypen definiert sind. Aus diesem Grund sollen sie auch in den TL-Bibliotheken in der fundamentalen *baseLib* deklariert werden.

Wegen ihres gerade genannten Charakters als Behälterdatentyp, lassen sich für die Module *array* und *string* eindeutig Kapselungskonzepte (vgl. Abschnitt 4.5.3) festlegen. Beide Schnittstellen werden imperativ deklariert, wie es in der aktuellen Version der Bibliotheken durchgeführt wurde. Bei Bedarf ist es dem Bibliotheksbenutzer freigestellt, seine eigene Schnittstelle für *array* oder *string* funktional zu entwerfen. Im Unterschied zu den genannten Modulen sind die übrigen Datentypen der *baseLib* so allgemein entworfen, daß sie keinem Kapselungskonzept eindeutig zugeordnet werden können. Eindeutige Zuordnungen zu den verschiedenen Kapselungskonzepten lassen sich nur für die Behälterdatentypen der *bulkLib*

Basisdatenstrukturen:

Array	indiziertes Feld mit Elementen gleichen Typs
Boolean	boolesche Werte
Char	Zeichen (nach Ascii-Zeichensatz)
Integer	ganze Zahlen
Real	reelle Zahlen mit doppelter Genauigkeit (engl. *double precision*)
String	Zeichenketten mit veränderlichem Zustand
ImmutString	Zeichenketten mit unveränderlichem (engl. *immutable*) Zustand

Ein- und Ausgabe:

Print	formatierte Ausgabe auf die Standardausgabe als Abkürzung für *write.stdout*
Read	Lesezugriff auf Zeichenströme
Write	Schreibzugriff auf Zeichenströme

Verschiedene:

Ascii	Zeichensatz
Date	Datum und Uhrzeit (siehe Abschnitt 5.1.2)
Directory	vom Betriebssystem abstrahierte Operationen auf Verzeichnissen
Domain	abstrakte Beschreibung für beliebige Werte (siehe Abschnitt 5.4.2)
Dynamic	Operationen für dynamische Typen
Environment	Wartung der Umgebungsvariablen, welche das Programmverhalten beeinflussen können
File	Dateibehandlung
Fingerprint	Abbildung von Bytestrings auf Hashwerte
Float	reelle Zahlen mit beliebiger Genauigkeit
Iter	Operationen auf Iteratoren
IterRep	Repräsentation eines Iterators
Optional	optionale Werte
Order	Ordnungsfunktion für beliebige Datentypen
Parameters	Zugriff auf Kommandozeilenparameter
Pair	polymorphe Funktionen für willkürliche Werte
Path	portable Schnittstelle zum Zugriff auf Datei- und Verzeichnisnamen
POrder	partielle Ordnungsfunktionen
RuntimeCore	Lesezugriff auf allgemeine Namen von Verzeichnissen der Tycoon-Umgebung
Scan	Zugriff auf eine Zeichenfolge mit Vorausschau
Time	Zeitangabe in Minuten seit einem festgelegten Startzeitpunkt (siehe Abschnitt 5.1.2)
Value	generelle Wertoperationen (siehe Abschnitt 5.2.5)

Tabelle 5.1: Schnittstellen der Basisbibliothek *stdLib* von Tycoon

treffen. Da diese Arbeit auf die Überarbeitung der Schnittstellen der Standardbibliothek beschränkt ist, wird nicht weiter auf die verschiedenen Kapselungskonzepte in TL eingegangen.

machineLib. Die Implementation von Basisdatentypen existiert in verschiedenen Ausprägungen auf den unterschiedlichen Maschinen. Daher muß z. B. für die Typen *Date* oder *File* in der Basisbibliothek eine allgemeine Repräsentation gefunden werden, die für alle Rechner gilt und deren Implementation rechnerabhängig in Modulen wie *unixDate*, *dosFile* u. a. erfolgt. Diese Implementationen werden jeweils mit einem Präfix, welcher die Plattform kennzeichnet, benannt. Somit sind die Schnittstellen der Bibliothek *machineLib* jeweils maschinenabhängig. Die Bibliothek *machineLib* enthält im wesentlichen die folgenden Schnittstellen zum Betriebssystem[1]:

Address	Zugriff auf den Hauptspeicher
Audio	Ein- und Ausgabe von Audiodaten
CCallback	Unterstützung der Callbacks von C nach Tycoon
CPUTime	maschineninterner Zeitmesser
Image	Bilddatenrepräsentation
Process	Betriebssystemaufrufe
Ticks	Zeitmessung der Maschine in Einheiten unter einer Sekunde
UxEnvironment	Schnittstelle des Moduls *environment* zu Unix
UxFile	Schnittstelle des Moduls *file* zum Unix-Betriebssystem
Word	Interpretation von Speicherworten als Zahlen ohne Vorzeichen oder Bitmuster bzw. Adressen

Word.T steht in *machineLib*, weil er in enger Beziehung zu Datentypen außerhalb von Tycoon steht und somit hauptsächlich in maschinennahen Zusammenhängen auf dem untersten Abstraktionsniveau benutzt wird, obwohl er ebenfalls als Basisdatentyp angesehen werden kann.

bulkLib. *bulkLib* stellt Behälterdatenstrukturen zur Verfügung, die auf den einfachen Datenstrukturen der *baseLib* aufbauen. Für fast jeden dieser Massendatentypen werden gleiche Operationen ähnlicher Ausprägung benötigt. Üblicherweise werden auf Massendatentypen Vergleichsfunktionen, wie beispielsweise $<$ und $==$, auf dem Elementtyp benötigt. Diese beruhen auf den Operationen *equal*, welche mittels der fundamentalen Schnittstelle *Value*, und *order*, welche mittels der abstrakten Schnittstelle *Domain* für alle Massendatentypen importiert wird. Ebenso umfassen viele dieser Behälterdatentypen eine Hash-Funktion, welche ebenfalls mit der Schnittstelle *Value* geliefert wird. Gleichfalls werden die Konvertierungsoperationen der Basisdatentypen in der *bulkLib* benötigt, weil die Formatierung eines Behälterdatentyps auf der Formatierung seines Elementtyps beruht. Viele der in der *bulkLib* benötigten Operationen werden somit bereits von der Basisbibliothek *baseLib* allen weiteren Bibliotheken zur Verfügung gestellt.

5.1.2 Schnittstellen für die Datums- und Zeitverwaltung

Date und *Time* befanden sich bislang in der Standardbibliothek *stdenv*. Sie waren aber nicht portabel und somit maschinenabhängig. Um diese Abhängigkeit zu beseitigen, wird ein Stan-

[1] Diese Liste stellt keinen Anspruch auf Vollständigkeit

dard für beide Datentypen erarbeitet, der portabel realisierbar ist und somit in der Bibliothek *baseLib* steht. Die dafür benötigten, systemabhängigen Datentypen werden in *machineLib* integriert. Die Deklarationen werden in Anlehnung an die Vereinbarungen in der Modula-3-Bibliothek festgelegt. Hinzugefügt werden nützliche Operationen zur Berechnung von verschiedenen Zeitpunkten und -intervallen, welche Umrechnungen zwischen verschiedenen Zeit- und Datumsdarstellungen unter Berücksichtigung des Kalenders erlauben.

Time. Ein Zeitpunkt wird mit dem Datentyp *Time.T* modelliert. Er repräsentiert die Anzahl an Sekunden seit einem festgelegten Startpunkt. Mit arithmetischen Operationen auf dem grundlegenden Datentyp *Real* können alle Zeitberechnungen durchgeführt werden. Eine Zeitspanne ist die Differenz zwischen zwei Zeitpunkten. Die Auflösung der meßbaren Zeitpunkte sollte so hoch sein, daß Zeitintervalle während der Ausführung eines Programms gemessen werden können. Die folgenden Operationen werden für den Datentyp *Time.T* definiert:

new(:Real) :T — Ein reeller Wert, der durch Zeitberechnungen ermittelt sein kann, wird als ein Zeitwert ab dem auf jeder Maschine verfügbaren Startzeitpunkt interpretiert.

now() :T — Die aktuelle Systemzeit wird ermittelt.

grain() :T — Die Granularität wird kleiner als 1 Sekunde garantiert. Zwei gemessene Zeitpunkte sind entweder gleich oder unterscheiden sich durch mindestens die Einheit der Granularität.

toString(:T) :String — Die Umwandlung eines Zeitpunkts in eine druckbare Zeichenkette.

Date. Der Datentyp *Date.T* stellt einen Zeitpunkt als ein Datum des gregorianischen Kalenders mit Uhrzeit dar. Er basiert auf dem Datentyp *Time.T* und enthält Regeln zur Konvertierung von Zeiten aus verschiedenen Zeitzonen in die lokale Zeit. Ein Tupel vom Typ *Date.T* ist ein Subtyp des Tupels

```
Tuple
        year: Int           (* z. B. 1999 *)
        month: Int          (* [jan ... dec] *)
        day: Int            (* [1 ... 31] *)
        hour: Int           (* [0 ... 23] *)
        minute: Int         (* [0 ... 59] *)
        second: Int         (* [0 ... 59] *)
        offset: Int         (* siehe Text *)
        zone: String        (* siehe Text *)
        weekDay: Int        (* [sun ... sat] *)
end
```

Die möglichen Werte für die Monate und Wochentage werden als Konstanten definiert. Das Feld *offset* spezifiziert die Differenz der lokalen Zeit zu der allgemeinen Zeit in UTC. Das Feld *zone* enthält die Bezeichnung der Zeitzone, für welche dieses Datum festgehalten wird. Folgende Operationen werden für diesen Datentyp deklariert:

error :Exception — Die Ausnahmebehandlung wird bei ungültigen Operationsaufrufen ausgelöst.

new(:T) :T — Erstellen eines Datums, welches im gregorianischen Kalender gültig ist.

fromTime(t :time.T zone :TimeZone) :T — Erzeugt aus dem Datentyp *Time.T* ein gültiges Datum.

getDate() :T — Erstellt das aktuelle Datum mit Zeit in der lokalen Zeitzone.

leapYear(x :T) :Bool — Die Abfrage gibt *true* zurück, wenn das Datum in einem Schaltjahr liegt.

addDays(x :T days :Int) :T — Addiert Tage zu einem Datum.

addMonths(x :T months :Int) :T — Addiert Monate zu einem Datum.

addYears(x :T years :Int) :T — Addiert Jahre zu einem Datum.

daysBetween(x, y :T) :Int — Berechnet die Anzahl der zwischen zwei Zeitpunkten liegenden Tage.

monthsBetween(x, y :T) :Int — Berechnet die Anzahl der zwischen zwei Zeitpunkten liegenden Monate.

yearsBetween(x, y :T) :Int — Berechnet die Anzahl der zwischen zwei Zeitpunkten liegenden Jahre.

secondsBetween(x, y :T) :Int — Berechnet die Anzahl der zwischen zwei Zeitpunkten liegenden Sekunden.

minutesBetween(x, y :T) :Int — Berechnet die Anzahl der zwischen zwei Zeitpunkten liegenden Minuten.

hoursBetween(x, y :T) :Int — Berechnet die Anzahl der zwischen zwei Zeitpunkten liegenden Stunden.

dateDifferenceBetween(x, y :T) :T — Berechnet die Differenz zwischen zwei Daten als absoluten Wert in der Datumsdarstellung.

timeDifferenceBetween(x, y :T) :time.T — Berechnet die Differenz zwischen zwei Daten als absoluten Wert in der Zeitdarstellung.

dayOfYear(x :T) :Int — Gibt den Kalendertag an.

fmtDate(x :T format :String) :String — Gibt eine Repräsentation des Datums (nur Tag, Monat, Jahr) als Zeichenkette.

fmtTime(x :T format :String) :String — Gibt eine Repräsentation der Zeit (nur Stunde, Minute, Sekunde) als Zeichenkette.

toTime(date :T) :time.T — Gibt das aktuelle Datum als Zeitdarstellung zurück.

toUTC(date :T) :T — Gibt die Zeit als Wert in der UTC an.

Für die beiden Operationen *fmtTime* und *fmtDate* werden Formatierungskennzeichen (z. B. *tt-mm-jjjj*) als eigener Datentyp vereinbart, welche die Darstellung der Ausgabe beeinflussen. Auf diese Weise sollen z. B. die Darstellungen

19. Juni 1996

und

06-19-96

aus demselben Wert erstellt werden können.

Ticks. Als Grundlage dieser Datentypen wird der Datentyp *Ticks.T* in der maschinenabhängigen Bibliothek *machineLib* erstellt. Dieser Datentyp repräsentiert den Wert einer Uhr mit einer Auflösung unter einer Sekunde. Diese Auflösung ist maschinenabhängig und üblicherweise eine sechzigstel Sekunde. Damit erlaubt dieser Datentyp genaue Messungen kleiner Intervalle. Folgende Operationen werden auf *Ticks.T* deklariert:

error :Exception — Eine Ausnahmebehandlung wird ausgelöst, wenn eine der folgenden Operationen fehlschlägt.

now() :T — Gibt den aktuellen Wert der Uhr in Anzahl von Ticks zurück.

toSeconds(ticks :T) :Real — Gibt die Anzahl Sekunden für die gegebene Anzahl an Ticks.

fromSeconds(seconds :Real) :T — Ermittelt die Anzahl Ticks für die gegebene Anzahl von Sekunden.

5.2 Konventionen für Methoden

In diesem Abschnitt wird zusammengestellt, welche Operationen oder Methoden in welchen Modulen benötigt werden. Das Konzept hinter dieser Zuordnung wird näher erläutert. Dazu wird versucht, diese Operationen in Gruppen zusammenzufassen, um dann für ähnliche Datenstrukturen auch die gleichen Operationen mit gleichen Namen zu erstellen, damit eine einheitliche Oberfläche der abstrakten Datentypen entsteht. Diese Einheitlichkeit dient der besseren Orientierung sowohl der Bibliotheksentwerfer als auch der Benutzer und sollte möglichst bibliotheksübergreifend sein. Es gibt generelle Konzepte für modulübergreifende Operationen, wie z. B. die Hash-Funktion oder Operationen zur Wandlung eines Massendatentyps in eine Iteration. Zusätzlich hat aber jeder Datentyp spezielle Eigenschaften, welche ebenso spezielle Operationen, die nur in diesem Datentyp definiert sind, erfordern.

Bei der Namensgebung in Bibliotheken muß auf Konsistenz und Gleichmäßigkeit geachtet werden, um es dem Benutzer leichter zu machen, die wesentlichen Eigenschaften einer Klasse zu erkennen. Außerdem ermöglicht die Konsistenz den Austausch von Implementationen ohne Änderungen im übrigen Programm, beispielsweise ein Ändern des Namens der aufgerufenen Operation, zu erfordern.

5.2.1 Fundamentale Operationsnamen

Die Namenswahl für Operationen wird als Ergebnis mehrerer Treffen zum Thema *Entwurf der Tycoon Bibliotheken* am Arbeitsbereich DBIS des Fachbereichs Informatik an der Universität Hamburg festgelegt. Sie dient der Vereinheitlichung der Bedeutung von Operationsnamen über die Schnittstellengrenzen hinweg.

equal, notEqual	Gleichheit des Wertes, standardmäßig als *shallowEqual* implementiert
==	Identität
!=	Negierte Identität
<	kleiner
>	größer
<=	kleiner oder gleich
>=	größer oder gleich
+	Addition
−	Subtraktion
*	Multiplikation
/	Division
and	boolsches *Und*
or	boolsches *Oder*
not	boolsche Negation
minimum	minimaler Wert des Wertebereichs, ersetzt *minVal* in einigen Schnittstellen
maximum	maximaler Wert des Wertebereichs, ersetzt *maxVal* in einigen Schnittstellen
min, max	das jeweils kleinere bzw. größere Element von zweien.
<>	Konkatenation. Für TooL wird das Symbol + vorgeschlagen, da dort ein Überladen von Funktionen möglich ist, womit die Namensinvasion eingedämmt würde.
negate	Umkehrung des Vorzeichens. Die Tilde ~ vor einer Zahl kennzeichnet sie als negativ. Eine Anpassung an die mathematische Schreibweise wäre wünschenswert, indem der Parser negative Zahlen am Vorzeichen − erkennt. Dies führt in der aktuellen Parser-Version zu Grammatik-Konflikten mit der Infix-Operation.

Tabelle 5.2: Funktionsnamen in TL

Festlegung der Bedeutung von Funktionsnamen In der Tabelle 5.2 werden die für TLallgemein gültigen Funktionen mit ihren Bedeutungen aufgeführt. Die bisherige doppelte Deklaration für arithmetische Basisoperationen durch Symbole und entsprechende Operationsnamen wird aufgehoben. Für Relationen und Funktionen sollen nur noch Symbole benutzt werden.

Sämtliche Inkrement- und Dekrementfunktionen werden ersatzlos gestrichen, da es keinen erheblichen Mehraufwand bedeutet, statt z. B. *inc(i)* den Ausdruck $i := i + 1$ zu schreiben.

Für die Datentypen *Word* und *Int* werden die in Tabelle 5.3 erläuterten Operationen benötigt. Die *unsigned*-Operationen interpretieren alle Bits gleichwertig, während die *signed*-Operationen das höchste Bit als Vorzeichenbit nicht mit in die Operation einbeziehen.

Die Tabelle 5.4 stellt einige grundsätzliche Operationen für Behälterdatentypen der *bulkLib* vor. Im Zusammenhang mit der Basisbibliothek *baseLib* werden diese Operationen für die Modulschnittstellen *Array* und *String* verwendet.

Erstellung von Datenstrukturen Datenstrukturen, welche sich komplett im Speicherbereich von Tycoon befinden, werden von der Freispeicherverwaltung (engl. *Garbage Collection*) gelöscht, andere Datenstrukturen, mit Referenzen aus dem Tycoon Speicher heraus, müssen vom Benutzer explizit wieder freigegeben werden (z. B. *File* und *Window*). Um diesen Un-

bitNot	Komplement jedes Bits
bitAnd	bitweises *Und*
bitOr	bitweises *Oder*
bitUnsignedShiftRight, bitSignedShiftRight	schiebe Bits rechts
bitUnsignedShiftLeft, bitSignedShiftLeft	schiebe Bits links

Tabelle 5.3: Bitoperationsnamen in TL

isEmpty(c :T(E)) :Bool	zeigt an, ob eine Datenstruktur leer ist
size(c :T(E)) :Int	gibt die Anzahl der Elemente eines ADT zurück. size >= 0 wird allgemein zugesichert. Wenn size = 0, dann ist die Datenstruktur leer (*isEmpty*).
clear(c :T(E)) :Ok	löscht alle Elemente. Die Nachbedingung ist *size*=0 , d. h. *isEmpty*.
includes(c :T(E) v :E) :Bool	stellt fest, ob das Element mit dem Wert v im Behälter c enthalten ist. Diese Abfrage wird häufig *isMember* genannt, mit dem Nachteil, daß der Massendatentyp nicht als erster Parameter aufgeführt werden kann, wenn die englische Grammatik berücksichtigt wird.
includesKey(c :T(E K) k :K) :Bool	stellt fest, ob das Element mit Schlüssel k in der Datenstruktur c enthalten ist.

Tabelle 5.4: Spezielle Funktionsnamen für Behälterdatenstrukturen

terschied für den Benutzer auch im Funktionsnamen deutlich zu machen, wird die folgende Konvention vereinbart.

new	erstellt Datenstrukturen, die durch *Garbage Collection* implizit wieder freigegeben werden.
create	erstellt Datenstrukturen, die vom Benutzer explizit wieder freigegeben werden müssen.
dispose	entfernt Datenstrukturen, die mit *create* erstellt wurden.
valid	prüft Datenstrukturen, die mit *create* erstellt wurden, auf ihre Existenz. Das Ergebnis ist *true*, wenn die Datenstruktur noch nicht mit *dispose* freigegeben wurde, und *false*, wenn *dispose* die Daten schon gelöscht hat.

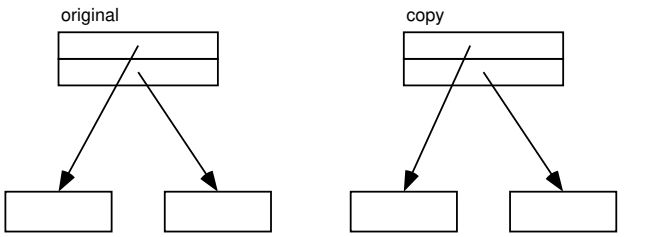
Durch diese Konvention müssen alle Funktionen, die in der Version 1 der Bibliotheken *destroy* genannt wurden, durch *dispose* ersetzt werden.

Kopieroperationen Die Kopierfunktionen in der Basisbibliothek werden in ihrer Bedeutung ebenfalls festgelegt. Die Unterscheidung von *copy* als imperative und *clone* als funktionale Kopierfunktion wird unter anderem in der Sprache Eiffel vorgenommen (siehe Abschnitt 4.2.2).

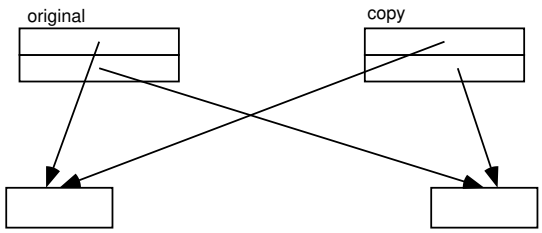
copy(from:T to:T):Ok	Die Inhalte eines Wertes eines ADT *from* werden in einen Wert eines ADT *to* desselben Typs kopiert. Die Funktion ist nicht konstruktiv und entspricht einer imperativen Schreibweise. Vor Aufruf muß eine neue Variable eines adäquaten Typs deklariert werden.
clone(x :T) :T	Ein *new* oder *create* erstellt einen neuen Wert eines ADT, der mit dem Inhalt von x gefüllt wird. Diese funktionale Version ist konstruktiv. Das Ergebnisobjekt wird von der Funktion erstellt.
equal(x, y :T) :Bool	Da die standardmäßige Kopierfunktion ein *shallowCopy* ist, wird die standardmäßige Gleichheitsfunktion auch *shallow* vergleichen.

Im Modul *value* werden die Operationen *shallowCopy* und *deepCopy* bereit gestellt. *copy* bzw. *clone* ist der Name, welcher dem Benutzer für seine Anwendung zur Verfügung steht. Abbildung 5.1 stellt die Unterschiede der Kopierfunktionen an den Beispielstrukturen *original* und *copy* dar. Für seine Anwendung darf er die Semantik (*shallow* oder *deep*) wählen, indem er auf die Funktionen des Moduls *value* zurückgreift. So wird in jeder Schnittstelle eines Datentyps eine *copy*-Funktion deklariert, welche unterschiedliche Ausprägung haben kann. In den meisten Fällen ist ein *shallowCopy* die standardmäßige Kopieroperation.

Um kopierte Objekte adäquat vergleichen zu können, werden ebenso *shallowEqual* und *deepEqual* im Modul *value* zur Verfügung gestellt. Im Datentyp-Modul wird gleichfalls eine Funktion *equal*, evtl. unter Benutzung der in *value* stehenden Operationen, definiert, die für diesen ADT sinnvoll ist. *shallowEqual* ist *true*, wenn für die beiden zu vergleichenden Objekte die Identität gilt, d. h. wenn sie auf dieselbe Struktur von Werten zeigen. *deepEqual* ist hingegen *true*, wenn die Struktur und der Inhalt beider Objekte gleich sind. Dies wird mit *semantischer Gleichheit* bezeichnet.



Speicherverweise nach Ausführung der Zuweisung *let copy = original.deepCopy()*



Speicherverweise nach Ausführung der Zuweisung *let copy = original.shallowCopy()*

Abbildung 5.1: Kopierfunktionen in der Speicherdarstellung

5.2.2 Klassifikation von Operationen für Massendatentypen

Das im Folgenden vorgestellte Klassifikationssystem für Operationen basiert auf der Möglichkeit, aufgrund verschiedener Eigenschaften von Operation und Datenstruktur Namen zusammensetzen zu können. So kann aufgrund des Namens auf die Eigenschaften der Operation geschlossen werden. Die Wahl der Namen wurde auf diese Weise systematisch durchgeführt. Bei der Klassifikation wird davon ausgegangen, daß verschiedene Kriterien auf Datentypen zutreffen und jedes dieser Kriterien einen Namensstamm erhält, welcher Teil des gesamten Operationsnamens wird. Die Reihenfolge der Kriterien in der nachfolgenden Aufzählung gibt die Reihenfolge der Zusammensetzung im Namen an. Die nachfolgend dargestellte Klassifikation ist für Massendatentypen erstellt worden. Ihre Anwendung im Bereich anderer Datenstrukturen ist sehr beschränkt. Sie ist daher als Empfehlung für eine Überarbeitung der *bulkLib* anzusehen, welche nicht Thema dieser Arbeit ist, und findet hier ausschließlich Verwendung für die Datenstrukturen *String* und *Array*, welche die Eigenschaften der fundamentalen Datentypen wie auch der Behälterdatentypen vereinen.

Es wird vereinbart, daß der Massendatentyp, auf dem die Operation ausgeführt wird, als erster Parameter steht. Die Reihenfolge der weiteren Parameter folgt dem Satzaufbau der englischen Sprache.

Die Operationen werden in die folgenden Klassen eingeordnet:

1. **Name** der Operation

   (a) **destruktive** Operationen — Zwei Datentypen werden als Parameter gegeben, der

erste Parameter wird mit dem Inhalt des zweiten verändert und somit überschrieben. Der Rückgabewert derartiger Operationen ist der Typ Ok. Die Operationen dieser Kategorie lauten:

delete löscht ein Element ohne Vorbedingung.
remove löscht ein Element mit der Vorbedingung, daß es vorhanden sein muß.
set setzt ein Element im Behälter ein, ohne Vorbedingung. Als Nachbedingung kann entweder auf das eingefügte Element mit *get* zugegriffen werden oder es wird eine Fehlermeldung ausgegeben. *set* ist die komplementäre Operation zu *get*.
replace ändert ein Element, mit der Vorbedingung, daß es vorhanden sein muß.
insert fügt ein Element in den Behälter ein, mit der Vorbedingung, daß es noch nicht vorhanden ist. Dies entspricht der Operation *set* mit Vorbedingung. Für eine geordnete Datenstruktur gilt bei *insert* im Gegensatz zu *set* die Vorbedingung, daß das Element an der gegebenen Stelle nicht vorhanden sein darf und daher auf jeden Fall eingefügt wird.

(b) **nicht destruktive** Operationen — Zwei Datentypen werden als Parameter gegeben und sind nach der Operation unverändert. Das geänderte Objekt wird in einem neuen Objekt als Ergebnis zurückgegeben. Die Operationen dieser Kategorie lauten[2]:

add fügt ein Element ein. Der Rückgabewert hat denselben Typ wie der erste Parameter.
get liefert ein Element des Behälters ohne Seiteneffekt.

Diese zwei Varianten von Operationen müssen unterschieden werden, da bei unveränderlichen Datentypen nur Variante 1b angewendet werden kann. Bei veränderlichen Datentypen sind beide Varianten möglich, aber meist wird 1a bevorzugt.

2. **Typ** des einzufügenden, zu löschenden oder zu ändernden Elements oder Behälters

(a) genau ein Element des Behälters — Dies ist die Standardbedeutung einer unter 1 genannten Operation, d. h. wenn genau ein Element eingefügt wird, wird auf eine spezielle Kennzeichnung der Operation mit zusätzlichen Bezeichnungen verzichtet.

(b) ein Teil des Behälterdatentyps —*All* als Zusatz zum Operationsnamen, z. B. *insertAll*, repräsentiert die Operation auf einem Teil des Objekts[3].

3. **Zugriffsart** auf ein Element des Behälters

(a) über den Wert des Elements — Dies ist die Standardbedeutung, es erfolgt kein Zusatz.

(b) über den Schlüssel des Elements — *Key* als Namenszusatz bezeichnet eine Operation, die den Ort des Änderungszugriffs über den auf dem Datentyp definierten Schlüssel aufsucht.

[2] *add* als prägnanter Bezeichner für das Einfügen wird gewählt, obwohl dieses Wort in anderen Sprachen für die arithmetische Addition gebraucht wird. Die Verwendung hier soll herausstellen, daß sich die Anzahl der Elemente in der Datenstruktur auf jeden Fall erhöht.

[3] Eine Bezeichnung mit *Range* statt *All* ist zu speziell, wenn es auf ungeordnete Datentypen angewendet wird.

(c) bei indizierten Datentypen — *Position* als Zusatz sucht einen gegebenen Wert über einen Index.

4. **Ort** des Einfügens, Löschens oder Änderns[4]

   (a) mitten im Container — Dies ist die Standardbedeutung, es erfolgt kein Zusatz.

   (b) am Beginn des Containers — *AtFirst* als Zusatz ist die treffendste und allgemeinste Form, diesen Ort zu bezeichnen.

   (c) am Ende des Containers — *AtLast* ist der entsprechende Zusatz, das Ende zu bezeichnen[5].

   4b und 4c sind bei indizierten Datentypen überflüssig, da in diesen Fällen der Ort der Einfügung mit Hilfe des Schlüssels oder Index als Parameter angegeben wird.

   3 und 4 sind Alternativen zur Auffindung des zu ändernden Elements. Wenn der genaue Schlüssel angegeben wird, darf nicht gleichzeitig der relative Ort aus 4b oder 4c vorgegeben werden. Umgekehrt gilt, wenn an einem relativen Ort gelöscht werden soll, dann darf zusätzlich kein konkreter Schlüssel zur Suche angegeben werden. Im Falle 3a kann daher zusätzlich mit 4 klassifiziert werden, im Falle 4a kann zusätzlich mit den Klassen aus 3 spezialisiert werden.

5. **Anzahl** der Löschungen bzw. Änderungen

   (a) genau eine Änderung bzw. Löschung am Ort mit angegebenen Eigenschaften — Dies ist die Standardbedeutung, es erfolgt kein Zusatz.

   Bei Datentypen, die keine Duplikate erlauben, ist dies der übliche Fall. Eventuell wird ein Namenszusatz aus 3 oder 4 eingesetzt.

   (b) alle Elemente, die den angegebenen Eigenschaften entsprechen, werden gelöscht oder mit der gegebenen Folge von Elementen (dies kann evtl. genau ein Element sein) überschrieben — *Each* als Zusatz zeigt an, daß jedes Element im Container, welches den Angaben entspricht, gelöscht bzw. geändert wird[6].

   Dies gilt nur für Datentypen mit Duplikaten. Bei dieser Variante wird für *delete* und *remove* die speziellere Nachbedingung, daß das gelöschte Element nicht in der Datenstruktur enthalten sein darf, auch für Datenstrukturen mit Duplikaten erfüllt.

   3, 4 und 5 sind sich gegenseitig ausschließende Möglichkeiten zum Auffinden des zu ändernden Elements. Wenn der genaue Ort angegeben wird, darf nicht gleichzeitig vorgegeben werden, daß alle Elemente gelöscht werden. Umgekehrt gilt, wenn alle auftretenden Elemente gelöscht werden sollen, dann darf zusätzlich kein Kriterium für den Ort

[4] Die in Eiffel verwendeten Operationsnamen *append*, *prepend*, *extend* und *precede* wurden diskutiert. Sie wurden nicht übernommen, weil sie nicht in das angegebene Klassifizierungsmuster passen.

[5] An Stelle der Bezeichner *AtFirst* und *AtLast* wurden auch die Vorschläge *First/Last*, *Begin/End* und *Front/End* als Gegensatzpaare in diesem Zusammenhang erwogen. Da diese Bezeichnungen häufig für Positionen auf dem Bildschirm gebraucht werden, sind diese Namenszusätze für die hiesige Verwendung nicht auf den ersten Blick verständlich.

[6] Die gebräuchliche Alternative *All* für *Each* wird in anderer Bedeutung für das Bezeichnen eines Teils einer Datenstruktur verwendet. Denkbar wäre auch eine prädikative Auffassung dieses Attributs *Each*, indem es als Eigenschaft der Datenstruktur gesehen wird, vergleichbar mit dem Vorhandensein von Duplikaten.

angegeben werden. Im Falle 4a kann also zusätzlich mit den Klassen aus 5 spezialisiert werden, im Falle 5a kann zusätzlich mit 4 klassifiziert werden.

6. **Rückgabe** des geänderten bzw. gelöschten Elements

   (a) ohne Rückgabe — Dies ist die Standardbedeutung, es erfolgt kein Zusatz.

   (b) mit Rückgabe — *Get* am Ende des Operationsnamens zeigt an, daß mit dieser Operation der geänderte oder gelöschte Wert zurückgegeben wird. Dies kann nach Kategorie 2 ein einzelnes Element oder eine Datensammlung sein.

   Diese Spezialisierung ist nur bei destruktiven Datentypen sinnvoll. Bei nicht-destruktiven Datentypen wird ohnehin eine *get*-Operation definiert (siehe Kategorie 1), die den geänderten Wert zurückgeben muß.

Ein Beispiel ist die Operation in der folgenden Zuweisung:

let myBulk = insertAllAtFirstGet(list updateList)

Nach der festgelegten Klassifikation fügt diese Operation die Behälterdatenstruktur *updateList* vorn in die Behälterdatenstruktur *list* ein und liefert das Ergebnis in *myBulk*. *list* bleibt durch diese Operation unverändert.

5.2.3 Konvertierung von Datentypen

Eine eigene Kategorie von Operationen in Modulen für Basisdatentypen bilden die Konvertierungsroutinen, die einen Datentyp auf einen anderen abbilden. Derartige Konvertierungen werden im Rahmen dieser Arbeit nur für Typen der *baseLib* betrachtet. Sie haben jedoch darüber hinaus Verwendung in der *bulkLib*, da sich die meisten Behälterdatentypen aus Basisdatentypen zusammensetzen. Zur sinnvollen Konvertierung von Behälterdatentypen ist der Abstraktionstyp *Iter* nützlich. Jeder Behälterdatentyp kann in eine Iteration verwandelt werden, welche wiederum als ein anderer Behälterdatentyp interpretiert wird. Diese Art der Iterationskonvertierung ist nur im Zusammenhang mit der *bulkLib* interessant und wird daher in dieser Arbeit nicht betrachtet.

Die Konvertierung in einen String dient hauptsächlich der Ausgabe von Werten am Bildschirm oder in eine Datei. Daher wird für jeden Basisdatentyp eine Routine gefordert, die diesen Datentyp in *String.T* wandelt, um eine Grundlage für die Darstellung aller auf Basistypen aufbauenden Datentypen zu erstellen.

Außer der Umwandlung in eine Zeichenkette gibt es noch weitere, sinnvolle Konvertierungen zwischen den Basisdatentypen. Alle Konvertierungsmöglichkeiten sind in der Tabelle 5.5 erfaßt. Im folgenden Abschnitt wird die Bedeutung der einzelnen Konvertierungsroutinen für das Tycoon System festgelegt. In einem weiteren Abschnitt wird dargelegt, an welcher Stelle diese Konvertierungsroutinen deklariert werden.

Die Semantik der Konvertierungsroutinen

Diese Konvertierungsfunktionen dienen verschiedenen Zwecken und haben verschiedene Semantik, die sich an der Implementation erkennen läßt.

ASCII-Code: Zur Konvertierung wird teilweise der ASCII-Code benutzt. Die Abbildungen von *Char* zu *Int* und *Int* zu *Char* sind die Funktionen, die den ASCII-Code benutzen und allgemein *ord* und *val* genannt werden. Sie bilden nur die ASCII-Tabelle ab:

from ↓ / to →	Int	Real	Bool	Char	String	Array	Word
Int		x	x	x	x		x
Real	x				x		
Bool	x				x		
Char	x				x		x
String	x					x	
Array					x		
Word	x				x		

Tabelle 5.5: Sinnvolle Konvertierungen der Basisdatentypen

Char.toInt(' ') ist 32, *Int.toChar*(32) ist ' '. In C++ wird dies auch im Namen der Funktion ausgedrückt: *atoi* heißt *ascii to int*.

Zifferndarstellung: Ebenso benötigt man eine Funktion, welche die Teilmenge 0..9 der Integerzahlen als Ziffern in die entsprechende Character-Darstellung wandelt. Daraus können dann Zeichenketten-Darstellungen der Ziffernfolgen von *Int*, *Real* u.a. zusammengestellt werden.

Unterbereichstypen: Ein Code ist auch die Aufzählung von Unterbereichstypen, wenn dieser Aufzählung eine automatische Numerierung von Integer, beginnend mit 0, zugeordnet wird. So wird z. B. im Typ *Bool* der Aufzählung (*false*, *true*) die Numerierung (0,1) zugeordnet, welches eine Teilmenge von *Int* ist. Diese Zuordnung wird für die Konvertierung von *Bool* zu *Int* und umgekehrt benutzt.

Elementtyp: Bei der Umwandlung von *Char* in *String* wird ausgenutzt, daß *Char* der *Elementtyp* des Behälterdatentyps *String* ist. Es wird ein Container mit genau einem Element erstellt. Die Umkehrfunktion ist nur auf einem Teil des Wertebereichs von *String* möglich, nämlich genau der Menge von Zeichenketten, die nur ein Element enthalten. Hier drückt sich die Sonderstellung der Datentypen *Array* und *String* aus, die wiederum aus einem Basisdatentyp zusammengesetzt sein können. Diese Eigenschaft läßt *Array* und *String* als Behälterdatentyp einordnen. Für Tycoon werden sie aber, wie auch in C++ und anderen Sprachen, zu den Basisdatentypen gezählt, da sie in den meisten Applikationen verwendet werden und in der Historie der Programmiersprachen immer zu den Basistypen gehörten.

Um eine mögliche Fehlinterpretation der Bedeutung einer Konvertierungsfunktion durch einen Bibliotheksbenutzer zu vermeiden, wird in diesem Abschnitt die Semantik aller Konvertierungsfunktionen für TL festgelegt:

Int nach String, Real nach String: Zur Interpretation eines *Int* oder *Real* als *String* benötigt man zwei Phasen:

1. *Int* wird als Folge von Elementen (Ziffern) interpretiert
2. Jedes dieser Elemente wird als *Char* interpretiert, wobei nicht direkt der ASCII-Code benutzt wird, sondern eine spezielle Abbildung der Ziffern 0..9 auf die Zeichen '0'..'9' stattfindet.

Int nach Real: *Int* ist mathematisch eine Teilmenge und damit ein Subtyp von *Real*. Es können darum alle ganzzahligen Werte als reelle Werte abgebildet werden, indem sie in eine andere Speicherdarstellung gebracht werden.

Real nach Int: Da nicht alle *Real*-Werte als *Int* darstellbar sind, ist ihre Konvertierung zu einem ganzzahligen Wert nur mit Hilfe einer Funktion möglich, welche den Bruchteil einer reellen Zahl vergißt bzw. auf- oder abrundet. Mit diesen Rundungsfunktionen geht Information verloren. Um die Semantik dieser Funktionen explizit zu unterscheiden, werden sie mit den mathematischen Funktionsnamen unterschieden:

trunc schneidet alle Nachkommastellen ab und rundet somit zum nächsten ganzzahligen Wert in Richtung 0.

floor gibt den nächsten Wert in Richtung −unendlich zurück.

ceiling gibt den nächsten Wert in Richtung +unendlich zurück.

round rundet auf den nächsten, ganzzahligen Wert.

sign gibt eine ganze Zahl für die Art des Vorzeichens wieder: −1 für ein negatives, 1 für ein positives Vorzeichen und 0 für den Wert 0.

Bool nach Int: Die Werte des Typs *Bool* werden als Aufzählung (*false*, *true*) angesehen, denen die Integer-Numerierung (0,1) zugeordnet wird.

Int nach Bool: Nur mit Hilfe einer Funktion ist die Umwandlung möglich. Dabei geht Information verloren, weil den Werten von *Bool* nur eine Teilmenge der Integerzahlen zugeordnet ist. Es werden alle positiven und negativen ganzen Zahlen auf den Wert *true* abgebildet, 0 wird auf *false* abgebildet.

String nach Int, String nach Real: Nur wenn eine eingeschränkte Menge von Zeichen in der Zeichenkette steht, ist eine Umwandlung möglich. Zuerst muß daher die Gültigkeit aller Zeichen geprüft werden. In *Int* kann nur umgewandelt werden, wenn alle Zeichen in der Teilmenge '0'..'9' liegen und somit Ziffern sind. Für reelle Werte darf zusätzlich ein '.', 'D' oder 'E' als Darstellung einer reellen Zahl in der Zeichenkette stehen. Daraufhin kann aus der Zeichenkette der *Int*- oder *Real*-Wert ermittelt werden.

Bool nach String: Eine spezielle Abbildung der Werte *true* und *false* auf die Zeichenketten 'true' und 'false' wird durchgeführt.

Char nach String: *Char* ist eine andere Darstellung einer einelementigen Teilmenge von *String*. Sieht man *String* als Behälterdatentyp an, dann ist *Char* der Elementtyp. Es werden dann alle Zeichenketten der Länge *eins* in den Typ *Char* gewandelt.

Ebenso kann ein *Char* als ein *Array* mit einem Element vom Typ *Char* angesehen werden. Hier erfolgt die Umwandlung entsprechend. Eine Funktion wird in der Basisbibliothek von TLallerdings nicht vorgesehen, da diese Funktion selten benötigt wird und einfach durch die Hintereinanderausführung der Funktionen

string.toArray(char.toString('a'))

ersetzt werden kann.

Der Deklarationsort der Konvertierungsroutinen

Es gibt verschiedene Orte, an denen die Konvertierungsroutinen deklariert werden können. Grundsätzlich besteht die Möglichkeit, alle Konvertierungsroutinen in einem Modul mit einer Schnittstelle zusammenzufassen. Dieser zentralisierte Ansatz hat den Nachteil, daß bei jeder Konvertierung im Anwendungsprogramm auch das Konvertierungsmodul importiert werden muß. Der Anwendungsprogrammierer denkt aber in den Strukturen seiner Anwendung und vergißt darum in vielen Fällen den Import-Befehl.

Ein besserer Ansatz ist die Verteilung der Konvertierungsroutinen auf die einzelnen Module, in denen die Datenstrukturen deklariert werden. In diesem Falle bieten sich die zwei Möglichkeiten an, die Routinen entweder zu den Routinen des Datentyps, der umgewandelt werden soll, zu schreiben, oder diese Routinen in dem Modul des Datentyps, in den verwandelt werden soll, zu integrieren. Im ersten Fall würden die Routinen durch das Wort *to* im Routinennamen kenntlich gemacht werden. Im zweiten Fall steht das Wort *from* im Namen der Routine.

Für Tycoon wird die erste Möglichkeit realisiert, da ein Anwendungsprogrammierer den Datentyp, den er umwandeln möchte, ohnehin in sein Programm importiert hat. Der Datentyp, der als Rückgabe der Routine dient, wird dann innerhalb des rufenden Datentyps importiert, sodaß der Programmierer sich nicht um den Import eines weiteren Moduls kümmern muß. Durch die initiale Umgebung von TL wird jeder Basisdatentyp vordefiniert, so daß sich bei der Konvertierung in einen Basisdatentyp das Importieren der jeweiligen Schnittstelle erübrigt. Als Beispiel wird in der Schnittstelle von *Int* eine Routine *toString* deklariert, die einen Integerwert in einen String wandelt.

```
Let T = Int
⋮
toString(x :T) :String
(*!OB
   Description: Return a string representation of the integer x.
                If x is negative, precede string by '-'.
   Parameters: In: integer x   Out: string
 !OE*)
```

5.2.4 Das div-mod-Problem

Während der Festlegung der Semantik aller Routinen für den Datentyp *Int* in der *baseLib* trat das Problem der Definition der Wertebereiche für eine Routine zur Restklassenbildung deutlich hervor.

Die Funktionen zur Division und Divisionsrestbildung bei ganzen Zahlen werden in der Literatur häufig mit *div* (für Division) und *mod* (für *Modulo*, d. h. Restklassenbildung) bezeichnet. Obwohl in TL die Funktionen nach der Konvention mit / und % bezeichnet werden, verwende ich innerhalb dieses Abschnittes die Bezeichner *div* und *mod*, um eine bessere Lesbarkeit zu erreichen und eine Unterscheidung der Funktionen für ganze und reelle Zahlen zu ermöglichen.

Wirth gibt in [Wirth 88] eine sprachliche Beschreibung des Funktionsverhaltens:

- x *div* y ist gleich dem ganzzahligen Anteil des Quotienten

- $x\ mod\ y$ ist gleich dem Rest der Division $x\ div\ y$ (für $y > 0$)

  Es gilt: $x = (x\ div\ y) * y + (x\ mod\ y)$

Einigkeit herrscht im Allgemeinen in der Literatur über die Definition dieser Funktionen für positive, ganze Zahlen. Sollen die *div*- und *mod*-Funktionen für alle ganzen Zahlen im Wertebereich von Integer, also auch für negative Werte, gelten, dann herrscht Uneinigkeit selbst in Fachkreisen darüber, welche Eigenschaften *div* und *mod* erfüllen müssen [Wichmann 79]. Daher schränken manche Autoren den Gültigkeitsbereich der *mod*-Funktion ein. Dem obigen Beispiel von Wirth folgend, werden nur positive Divisoren erlaubt. In TL-Syntax lautet eine einfache Implementation der *mod*-Funktion (Beispiel aus [Gnatz 77]):

```
let rec mod(a,b :Int) :Int =
  if a >= b then mod(a - b  b)
  else a
  end
```

Hier würde für ein positives a und ein negatives b eine Endlosschleife entstehen, da a immer größer als b bliebe, weil bei jedem rekursiven Aufruf von *mod* der absolute Wert von b zu a hinzuaddiert würde.

Wie sollen nun diese beiden Funktionen definiert werden? Der Wertebereich (positiv oder negativ) ist festzulegen und das Verhalten bei Division und Modulo-Bildung mit 0.

Die meisten Autoren fordern die *div-mod-Identität* [Wirth 88], in [Wichmann 79] auch *Remainder Identity* genannt:

$x = (x\ div\ y) * y + (x\ mod\ y)$
soll gelten für alle ganzzahligen Argumente x, y *mit* $y\ != 0$

Eine weitere Forderung bei Ausdehnung der Division auf negative Werte ist die Symmetrie von *div* [Nievergelt 91a]:

$$(-x)\ div\ y = x\ div\ (-y) = -(x\ div\ y)$$

Stellen wir jetzt zusätzlich eine schwache Forderung an die *mod*-Funktion:

Anzahl der Restwerte ist $|y|$, *d.h. bei festem* y *nimmt* $x\ mod\ y$ *genau* $|y|$ *verschiedene Werte an.*

oder wir fordern als Zusatz nichtnegative Reste:

$$0 <= x\ mod\ y < |y|,$$

dann gibt es mit der schwachen und der starken Forderung einen Widerspruch zur Symmetrieforderung.

Die zahlreichen Leserzuschriften in der Zeitschrift Informatik-Spektrum zum *div-mod*-Problem vertreten überzeugend zum Teil gegensätzliche Standpunkte zur Lösung dieses Problems. Wichmann [Wichmann 79] unterscheidet die Antworten auf seine intuitive Umfrage bzgl. des erwarteten Verhaltens von *div*- und *mod*-Operationen für Beispieloperatoren in sechs Kategorien:

1. Abschneiden in Richtung 0. Der reelle Teil wird erhalten, während der Bruchteil ignoriert wird. Das Vorzeichen des Restes ist dasselbe wie für x.

2. Vorzeichen löschen. Berechnung wie im 1. Fall, aber das Vorzeichen des Restes ist negativ, wenn x und y verschiedene Vorzeichen haben. Wenn y negativ ist, kann die allgemeine Multiplikationsregel nicht angewendet werden. Somit wird die div-mod-Identität nicht erhalten.

3. Rest ist positiv. Die Integerdivision hängt von den Vorzeichen von x und y ab und nicht vom reellen Wert x/y. Daher ist $(-4)\ div\ 5$ nicht gleich $4\ div\ (-5)$ und die Defintion nicht symmetrisch.

4. Abschneiden in Richtung $-$unendlich. Die mathematische Division wird durchgeführt und dann die ganze Zahl, die kleiner gleich diesem Wert ist, als Ergebnis genommen. Das Vorzeichen des Restes ist das Vorzeichen von y.

5. Andere. Die weite Streuung in dieser Kategorie ist bemerkenswert, besonders im Hinblick auf die enge Auswahl der befragten Leute.

6. Weiß nicht. Begründung: keine Eingebung oder würde immer die Dokumentation zu Rate ziehen.

Die meisten dieser Antworten (52%) hielten die div-mod-Identität ein, aber die anderen Kategorien waren nicht akzeptabel. Daher sind nur die Kategorien 1, 3 und 4 relevant für die weitere Diskussion.

Nievergelt [Nievergelt 91b] wagte es nicht, die gegensätzlichen Ansichten zu einem Standard zusammenzufassen. Auch für Tycoon ist nicht eindeutig zu vertreten, welcher Ansatz gefolgt werden sollte, da dies von der Applikation abhängt. Zum einen kann die Applikation die mathematische *Restklasse modulo y* erfordern, welche nur positive Restwerte kennt. Zum anderen kann man die Division als Berechnen eines Verhältnisses ansehen. Dann führt das Rechenbeispiel mit Elektronen, deren Ladung dargestellt wird, zu negativen Resten:
$(-14)\ div\ (-4) = 3\ Rest\ (-2)$

Für Tycoon wollen wir eine Minimalforderung erfüllen. Der Benutzer der Bibliothek wird selber dazu angehalten, sich bei Bedarf eine eigene Version der Division und Restbildung zu schreiben. In Programmiersprachen existieren auch hierzu einige Ansätze. In Ada und Portal wird als Ergänzung von *div* und *mod* die Funktion *rem* (für englisch *Remainder* = Rest) angeboten. Ein weiterer Vorschlag ist die Benennung der symmetrischen Varianten beider Funktionen mit *sdiv* und *smod*.

Der Katalog mit Minimalforderungen an Tycoon lautet:

1. $x\ div\ y$ und $x\ mod\ y$ haben denselben Definitionsbereich der ganzen Zahlen, mit der Ausnahme für $y = 0$.

2. div-mod-Identität

3. Normierung der Restklassen $x\ mod\ y$ zu genau $abs(y)$ Werten

4. Bewahrung möglichst vieler Symmetrien: $x\ div\ -y = -(x\ div\ y)$ etc.

5. Einfachheit der Definition

6. präzise Zusammenstellung der geltenden Rechenregeln und Sätze

Wenn die div-mod-Identität eingehalten wird, gibt es noch vier Varianten für die Definition der *div*-Funktion auf dem Bereich der ganzen Zahlen, die von Nievergelt in [Nievergelt 92] vorgestellt werden. Diese vier Definitionen der Division lauten:

1. $x\ div\ y := floor(x/y)$
2. $x\ div\ y := floor(x/y\ +\ 1/2)$
3. $x\ div\ y := sign(y)\ *\ floor(x/abs(y))$
4. $x\ div\ y := sign(y)\ *\ floor(x/abs(y)\ +\ 1/2)$

Für alle Definitionen gibt es dann für $y! = 0$ genau $abs(y)$ verschiedene Restklassen. Die Intervalle für $x\ mod\ y$ und die geltenden Symmetrien unterscheiden sich aber in allen Fällen.

Aufgrund der Vielzahl der möglichen Definitionen wird für Tycoon festgelegt, daß innerhalb der Basisbibliothek der Definitionsbereich auf positive ganze Zahlen eingeschränkt wird. Man erhält eine allgemeine Definition der Division mit ganzen Zahlen, die jeder Benutzer auf seine Weise spezialisieren und auf den negativen Zahlenbereich ausweiten kann. Durch die Einführung von Zusicherungen in Tycoon (siehe Abschnitte 3.3.4 und 5.3) kann jeder Benutzer der Division $x\ div\ y$ darauf hingewiesen werden, daß er unerlaubterweise negative Zahlen zur Division benutzt. Daraufhin hat er die Möglichkeit, sein Anwendungsprogramm zu überprüfen.

Sollte ein Anwendungsprogrammierer negative Zahlen benutzen wollen, muß er sich seine eigenen Routinen, basierend auf den Bibliotheksroutinen, schreiben. Durch die obige Festlegung wird verhindert, daß ein Benutzer darauf vertraut, daß er das gewünschte Ergebnis für die Division mit negativen Zahlen erhält. Die Ausnahmebehandlung nach einem falschen Aufruf weist ihn darauf hin, daß die vordefinierten Divisions- und Modulo-Routinen nicht vollständig im gesamten Zahlenbereich spezifiziert sind. Der Benutzer kann dann aus der Vielzahl der möglichen Definitionen die zu seiner Anwendung passende wählen.

5.2.5 Hash-Funktionen

Im Gegensatz zu der aktuellen Version der Tycoon Bibliotheken werden die Hash-Funktionen in der neuen Bibliotheksversion nicht für jede Schnittstelle deklariert. Sie werden zentral in der Schnittstellenimplementation *value* gehalten, um dann für jeden Datentyp benutzt werden zu können. Diese Implementation erzeugt automatisch für jeden Datentyp einen Hash-Wert. Die einzige Bedingung, welche an eine Hash-Funktion gestellt wird, ist die Erhaltung der Gleichheitsrelation. Da je nach Datentyp zwei Werte nach ihrer Identität (z. B. Basisdatentyp *int*) oder nach ihrer semantischen Gleichheit (z. B. Behälterdatentyp *list*) verglichen werden, muß für jede dieser Gleichheitsrelationen auch eine Hash-Funktion berücksichtigt werden.

Das Modul *value* wird neu erstellt und enthält alle allgemeinen Funktionen, die auf beliebigen Werten operieren und daher für wertbasierte Datentypen eingebunden werden. Die Operationen aus dem ehemaligen Modul *poly* werden weitestgehend übernommen.

Folgende Funktionen werden in der zugehörigen, abstrakten Schnittstelle *Value* unter Berücksichtigung von Referenzgleichheit und Identität[7] deklariert:

- *deepCopy* als Kopierfunktion zum Erzeugen einer identischen Datenstruktur,
  shallowCopy als Funktion zum Erstellen einer referenzgleichen Kopie,

[7] Vergleiche hierzu auch Abschnitt 4.2.2 zur Implementation in Eiffel.

- *deepEqual* als Vergleich auf Identität,
  shallowEqual als Vergleich auf gleiche Referenz innerhalb der Objekte,
- = als Vergleich auf Wertegleichheit,
- *hash* als eine *equality*-Hash-Funktion, welche die semantische Gleichheit der Werte erhält,
  identityHash als Hash-Funktion zur Erhaltung der Identität,
- sowie einige weitere Funktionen höherer Ordnung.

5.3 Zusicherungen in Tycoon

Vor- und Nachbedingungen für die Methoden eines Objekts sind nötig, um Programme möglichst korrekt in Bezug auf die definierten semantischen Eigenschaften und infolgedessen leicht wiederverwendbar zu erstellen. Bei Verletzung einer dieser Eigenschaften soll eine Ausnahmebehandlung durchgeführt werden. Auf diese Weise wird dem Anwendungsprogrammierer mitgeteilt, daß er ein Modul falsch benutzt hat. Nach dem Vertragsmodell von Eiffel sollte ein Bibliotheksimplementierer ebenfalls die Nachricht einer Ausnahme erhalten, wenn sein Modul nicht korrekt ausgeführt wurde.

Um die Methode des ***Programmierens durch Vertrag*** aus Eiffel (siehe Abschnitt 3.3.4) auch in TL zur Verfügung zu stellen, werden Konstruktoren für Vor- und Nachbedingungen in TL ergänzt. Durch die Einführung der Zusicherungen wird die Verantwortung für einen korrekten Routinenaufruf vom Modulimplementierer an den Anwendungsprogrammierer abgegeben. Dieser Ansatz ist sicherer als in Programmiersprachen ohne Zusicherungen, da nicht bei jedem Aufruf einer Routine die Vorbedingungen geprüft werden müssen. Sie sind zentral in der Routine implementiert und werden bei jedem Aufruf automatisch getestet. Ebenso müssen nicht nach jeder Ausführung der Routine sämtliche Fehlerbedingungen überprüft werden. Es werden stattdessen die in der Routine integrierten Bedingungen zugesichert. Vom Modulimplementierer wird durch diese Methode verlangt, daß er schon während des Entwurfs sorgfältig *sämtliche* Ein- und Ausgangsbedingungen der Routinen bedenkt und abfängt. Ist die Routine nach dem Vertragsmodell konzipiert, dann kann der Anwendungsprogrammierer anhand der Ausnahmen seinen Quelltext mittels eines Tests der Vorbedingungen an die Erfordernisse anpassen. Hat er die Bedingungen erfüllt, dann kann er sicher sein, daß die von ihm benutzte Routine nach ihrer Ausführung die Nachbedingungen erfüllt, die ihm als Grundlage für den weiteren Programmablauf dienen.

In [Mandrioli, Meyer 92] wird behauptet, daß das Konzept des *Programmierens durch Vertrag* im Gegensatz zum *defensiven Programmieren* steht. Unter *defensivem Programmieren* wird verstanden, daß der Programmierer versucht, alle Fehlerfälle in seinem Code abzufangen. Diese Arbeitsweise führt dazu, daß in der Modulimplementierung ebenso wie in der Anwendungsprogrammierung die gleichen Fehlerfälle abgefangen werden und der gesamte Programmcode expandiert. Das Vertragsmodell hingegen gibt vor, daß alle Verantwortlichkeiten auf den Anwendungsprogrammierer und den Modulimplementierer verteilt werden. Dabei ist die Grenze der Aufteilung variabel und muß für ein Projekt während der Entwurfsphase festgelegt werden. Damit steht das Vertragsmodell nicht im Widerspruch zum *defensiven Programmieren*, sondern stellt eine Möglichkeit dar, möglichst alle Fehlerfälle durch Aufgabenverteilung ohne Wiederholungen im Code abzufangen.

Das Konzept von modulumfassenden Invarianten ist grundsätzlich in der modularen Sprache TLim Gegensatz zu objektorientierten Sprachen nicht realisierbar, weil ein Invariantenausdruck auf das Objekt, für welches der Ausdruck gelten soll, zugreifen muß. Dieses Objekt ist jedoch außerhalb der Routinen in einem Modul nicht vorhanden. In objektorientierten Sprachen ist das Objekt, auf welches sich die Invariante bezieht, implizit bekannt, so daß es jederzeit zugreifbar ist. Obwohl die Klausel einer Invariante in TLfehlt, wird der Umfang der Korrektheitsprüfung nicht verringert, weil die Invarianten für jede Routine einer Schnittstelle als Vor- und gleichzeitig als Nachbedingungen formuliert werden können. Falls dies zu aufwendig erscheint, sollte der Implementierer verpflichtet werden, die Invarianten im Dokumentenkopf festzuhalten, damit ein Benutzer dieses Moduls weiß, welche Bedingungen seine Umgebung erfüllen muß.

Ebenso bereitet das Konzept des Zugriffs auf einen Variablenwert mit ***old***, der nur zu Beginn der Routine bekannt war, Schwierigkeiten in der Realisierung. Die Anfangswerte der Parametervariablen einer Operation sind nach Ausführung dieser Operation nicht mehr erhalten. Um eine aufwendige Kodierung mit Hilfsvariablen, die für jeden Parameter den Anfangswert speichern müssen, zu vermeiden, wird in TLauf das Konzept der *old*-Variable verzichtet. Ein Anwendungsprogrammierer kann in seiner Anwendung im Einzelfall dieses Konzept emulieren, wenn er eine derartige Nachbedingung prüfen will. Er muß in diesem Fall den entsprechenden Parameterwert in einer zusätzlichen Variablen abspeichern, um den Wert dann am Ende der Routine zu Prüfzwecken vergleichen zu können.

Zusätzlich muß jedem Implementierer von Modulen bewußt sein, daß auch die drei Zusicherungsarten aus Vorbedingung, Nachbedingung und Invariante zusammen nicht vollständig die Axiome eines ADTs abdecken. Eine komplette Zusicherungssprache würde die direkte Manipulation von Mengen, Folgen, Funktionen und Relationen, sowie Prädikate erster Ordnung und Quantoren benötigen. So ist für den Stack ein mathematisches Axiom

$$pop(push(x\ s)) == s$$

als Bedingung in TLnicht formulierbar. Um durch die Einführung dieser Konstrukte nicht vollständig die Eigenschaften und Ziele der Sprache zu verändern, welches das Erlernen und die Implementation erschweren und die Performanz verringern würde, wird der hier beschriebene, pragmatische Kompromiß für das Tycoon System gewählt. Es liegt in der Verantwortung der Programmierer, weitere Eigenschaften ihrer Schnittstellen und Module informell in einem Kommentar festzuhalten.

Grundsätzlich sind mehrere Lösungen zur Implementation der Zusicherungen in TLdenkbar. Zum einen kann die Sprache verändert werden. Dadurch werden Probleme der Kompatibilität entstehen, wenn Quelltexte nach der alten Sprachdefinition weiterverwendet werden sollen. Zudem steigt mit einer Erweiterung der Sprache auch die Komplexität des Compilers. Zum anderen können Generatoren geschrieben werden, wie sie in [Wetzel 94; Münnix 94] verwendet werden. Ein solches Konzept stünde außerhalb der Sprache und wird daher nicht realisiert. Als realisierbares *Add-On*-Konzept, welches die bisherige Sprache erhält, werden Syntaxerweiterungen zur Realisierung eingesetzt. Wie in [Schröder 93] beschrieben, erweitern sie die Grundkonzepte der Sprache mit zusätzlich definierbaren Konstrukten.

Um dem Benutzer der Bibliotheksmodule eine Auswahl zu erleichtern und die Spezifikation der Schnittstelle zu präzisieren, werden die Vor- und Nachbedingungen für jede Operation in der Schnittstelle als eigenes Konstrukt deklariert. Da diese Bedingungen bei der Übersetzung nicht von der Schnittstellendefinition her überprüft werden können, erzeugt das Tycoon System automatisch aus der Schnittstelle und der zugehörigen Modulimplementation

ein Prüfmodul, welches die Vorbedingungen prüft und bei ihrer Erfüllung die vom Anwender geschriebene Implementation der Routine ausführt. Im Fehlerfall wird eine Ausnahme ausgelöst. Nach erfolgreicher Ausführung der Routine wird die Nachbedingung getestet und im Falle der Nichterfüllung ebenfalls eine Ausnahme ausgelöst. Zur Übersetzungszeit können nur die Gültigkeit der Variablen und die Übereinstimmung der Typen statisch überprüft werden. Die Auswertung der Bedingungen kann erst zur Laufzeit erfolgen. In TL wird durch dieses Realisierungskonzept der Quelltext von Modulimplementationen nicht berührt. Sie werden wie gewohnt erzeugt. Die Realisierung der Überprüfung von Zusicherungen im Compiler ist in Sprachen, bei denen keine Trennung von Schnittstelle und Implementation erfolgt (z. B. Eiffel), einfacher, weil nur Deklarationen von Klassen bzw. Modulen bei der Auswertung der Zusicherungen berücksichtigt werden müssen.

Im Tycoon System schließen wir uns den Schlüsselworten an, die schon in [Hoare 72] für die Korrektheitsbeweise benutzt wurden, da sie, im Gegensatz zu den in Eiffel benutzten Schlüsselworten, welche auf das Vertragsmodell weisen, die mathematische Bedeutung betonen und ebenfalls in den Sprachen CLU, Alphard und Euclid eingesetzt werden. Bei diesen Prädikatentransformern werden die Vorbedingungen mit dem Schlüsselwort ***pre*** eingeführt, Nachbedingungen stehen hinter dem Schlüsselwort ***post***. Beide Konstrukte stehen nach einer Routinendeklaration und gelten jeweils für diese eine Routine.

Die boolschen Ausdrücke nach den Schlüsselworten ***pre*** und ***post*** bestehen aus einfachen Ausdrücken [Matthes, Müßig 93], die mit den boolschen Operatoren \/ (Konjunktion) und /\ (Disjunktion) verknüpft werden. Als Konvention für den *Style Guide* von TL wird festgelegt, daß jede Bedingung in einer eigenen Zeile steht. Der Standardwert für fehlende Bedingungen ist true. Obwohl Invarianten nicht realisiert sind, werden invariante Bedingungen nach dem Schlüsselwort invariant direkt vor dem end einer Schnittstellendeklaration aufgeführt, um dadurch für den Bibliotheksanwender die Eigenschaften der Schnittstelle zu verdeutlichen. Vom Übersetzer werden sie als Kommentar erkannt.

Als Beispiel der Implementation sei hier die Schnittstelle eines *Stacks* in der TL-Syntax aufgeführt:

```
interface Stack
export
  T <:Oper(E <:Ok) Ok
  error :Exception
  size(...)
  top(...)
  empty(...)
  full(...)
  ⋮
  push(E <:Ok  stack :T(E)  item :E) :Ok
    pre not(full(stack))
    post not(empty(stack))  /\
      top(stack) == item
end
```

Die Ausnahmen für auftretende Verletzungen der Vor- und Nachbedingungen werden in der Standardumgebung von TL deklariert. Wenn beispielsweise bei Aufruf

```
stack.push
```

die Bedingung

full(stack)

gilt, wird eine Ausnahme *preFailure* mit folgender Deklaration ausgelöst:

```
let preFailure = exception "Precondition failed" with moduleName :String end
```

Die Modulimplementation eines *Stacks* wird wie gewohnt geschrieben:

```
module stack
import ...
export
  Let T = ...
  let error = ...
  let size(...) = ...
  let top(...) = ...
  let empty(...) = ...
  let full(...) = ...
  let push(E <:Ok  stack :T(E)  item :E) :Ok = ...
  ⋮
end
```

Mit Hilfe einer generischen Funktion

newCheckedStack (_stack :Stack) :Stack

in dem vom System automatisch erstellten Modul *checkedStack* werden aus diesem vom Benutzer zu deklarierenden Paar von Schnittstelle und Modul bei der Kompilierung automatisch die Vor- und Nachbedingungen als ***if***-Abfragen modelliert. Die vom Benutzer implementierten Routinen werden unter den aufgestellten Bedingungen aufgerufen. Wenn der Anwendungsprogrammierer eine Prüfung vornehmen will, dann muß er in der Deklaration der Bibliothek und allen Modulen, die *stack* benutzen, den Aufruf jeweils durch

let stack = checkedStack.new(stack)

ersetzen. Für das Tycoon System wird eine Schalteroption gefordert, so daß das einfache oder das entsprechende, mit Zusicherungen überprüfbare Modul vom Compiler eingebunden wird.

5.4 Konzepte zur Abstraktionsbildung

Im Abschnitt 3.3.1 sind anhand der Vergleichssprachen einige Konzepte zur Bildung einer höheren Abstraktionsebene in den Bibliotheken identifiziert worden. Dazu sind allgemeine Operationen auf Werten für wertbasierte Datentypen erkannt worden, welche in dem Modul *value* für derartige Datentypmodule aggregiert werden. Die Beschreibung dieses Moduls, welches von den Werten der jeweiligen Datentypen abstrahiert, erfolgte bereits im Abschnitt 5.2.5, so daß es im folgenden nicht erläutert wird.

Eine weitere Abstraktion läßt sich für Behälterdatentypen, zu denen in diesem Zusammenhang aus den bereits erwähnten Gründen die Module *array* und *string* gehören, erkennen.

Operationen, welche die Eigenschaften der Behälterdatentypen konfigurieren, werden in einem Modul *domain* zur Verfügung gestellt.

Ebenfalls werden die Iteratoren als eine Abstraktion für die wiederholte Ausführung von Operationen angesehen. Da ihre Implementation auch für einige Basisdatentypen (z. B. *array* und *string*) benutzt wird und sie als Grundlage für die Bibliothek *bulkLib* dient, wird der Entwurf von Iteratoren hier im Zusammenhang mit der Standardbibliothek vorgeschlagen.

Weitere Abstraktionen, welche der Klassifizierung der Eiffel-Bibliothek in Speicherungs-, Hash- und Durchlaufskonzept entsprechen würden, werden in Tycoon nicht angestrebt, da ihre Realisierung wegen der vordefinierten Standardumgebung nur mittels komplizierter Strukturen erreicht werden kann. Eine derartige Aufblähung der Bibliotheksstrukturen dient nicht der Lesbarkeit und Verständlichkeit der Module. Weiterhin sind derartige Abstraktionen nur für Behälterdatenstrukturen anwendbar, so daß sie im Zusammenhang mit der Standardbibliothek in dieser Arbeit außer Acht gelassen werden.

5.4.1 Iteratoren

Für die Iteratoren einer Programmiersprache muß entschieden werden, ob sie der Datenkapselung oder der Kapselung von Kontrollstrukturen für eine Traversierung dienen sollen. Ein Iterator hat daher spezielle Voraussetzungen zu erfüllen. Er muß eine Datenstruktur mit spezieller Ordnung darstellen, eine Aktions- sowie eine Anfrageoperation beinhalten und eine Strategie bereitstellen, welche, basierend auf der Anfrageoperation, die Aktionsoperation ausführt. Damit ergibt sich die minimale Deklaration eines Iterator-Datentyps für Tycoon, wie sie in Abschnitt 3.3.2 für die abstrakte Schnittstelle *IterRep* dargestellt ist. Die Benutzerschnittstelle *Iter* behält die bereits in der aktuellen Version bestehenden Operationen, um dem Anwendungsprogrammierer eine vielfältige Auswahl an Iterator-Operationen zur Verfügung zu stellen.

Die Verwendung der Iteratoren in den Bibliotheken von Tycoon muß eingeschränkt werden. In der aktuellen Version der Bibliotheken wird der Typ *Iter* in der Schnittstelle des Typs *Int* benötigt, um eine Iteration über ganze Zahlen zu ermöglichen. Da wiederum der Typ *Int* fundamental für die Deklaration verschiedener Schnittstellen, u. a. *Iter*, ist, ergibt sich ein Zyklus in der Importierung, welches in den Tycoon Bibliotheken nicht erlaubt ist. In der aktuellen Version kann dieser Zyklus aufgelöst werden, da der Typ *Int* in der Standardumgebung vordefiniert wird. Da dies bei der Reorganisation jedoch vermieden werden soll, muß auf eine Verwendung von Iterationsoperationen in den Basisdatentypen wie *Int* verzichtet werden.

Die bisherigen Iteratoroperationen *create* und *elements*, welche in den Schnittstellen der Behälterdatentypen zur Erstellung von Iterationen deklariert sind, werden nach den allgemeinen Namenskonventionen (siehe Abschnitt 2.4.1) in *toIter* und *fromIter* umbenannt.

Die Trennung der Implementation des Iterators in eine Repräsentation *IterRep* und die abstrakte Oberfläche *Iter*, welche allein vom Benutzer gesehen werden soll, wird beibehalten. Mit Hilfe des Sprachkonstrukts hide kann diese Repräsentation vor dem Bibliotheksbenutzer versteckt werden. Diese Grundlage der Implementation muß nur dem Implementierer eines Behälterdatentyps bekannt sein, welcher Iteratoren verwenden will. Damit läßt sich die Implementation des Iterators ändern, ohne die Applikationsprogramme, welche die Iterator-Schnittstelle benutzen, zu berühren. Die Implementation der Iterator-Repräsentation kann rein funktional oder rein imperativ erfolgen und beeinflußt mittels ihrer zugehörigen Schnittstelle die Anwendungsprogramme. Die Vorteile der Verwendung verschiedener Kapselungskonzepte in Bibliotheken ist im Abschnitt 4.5.3 erläutert worden.

5.4.2 Domain

Mit *Domain* steht eine generelle, abstrakte Schnittstelle zur Verfügung, welche mit einer abstrakten Klasse in Eiffel vergleichbar ist. Diese Wertebereichsabstraktion basiert auf einem Vorschlag von F. Matthes und enthält die verbindenden Eigenschaften zwischen Basisdatentypen und Behälterdatentypen. Wegen ihrer universellen Nutzbarkeit wird sie in der Standardbibliothek *baseLib* eingebunden. Insbesondere für die Typen *array* und *string* der *baseLib* stellt *Domain* die Eigenschaften des Wertebereichs dar und bildet eine Abstraktion über Ordnungsfunktionen und Beziehungen der Behälterelemente.

Ein Wert vom Typ *Domain* wird in jeder Schnittstelle eines Basisdatentyps eingefügt. Dadurch wird sichergestellt, daß die in *Domain* deklarierten Operationen in jedem Datentyp zu finden sind. Für Datentypen, bei denen für einzelne *Domain*-Operationen keine Implementation zu finden ist, wird der unbestimmte Wert *nil* vom Typ *optional.T* deklariert[8]. Daher sind allgemeine Abfragen auf Datentypen leicht zu realisieren, weil sie auf Operationen der *Domain* ausgeführt werden und somit für alle Datentypen vorhanden sind. Ein Vorteil ist zusätzlich darin zu sehen, daß die für einen Datentyp nötigen Operationen mit ihren Standardnamen automatisch für eine Implementation vorgegeben werden. Es müssen nur deren Werte eingetragen werden.

Die subsumierten Eigenschaften von *Domain* werden in der Tabelle 5.6 wiedergegeben. Die für den Datentyp *Array* geltenden Werte sind exemplarisch hinzugefügt. Dabei basiert die Äquivalenzrelation ebenso wie die Hashfunktion und die Ordnungsfunktion auf den entsprechenden Funktionen des Feldelementtyps. Die Ordnungsfunktion gibt dabei die lexikografische Reihenfolge an.

Die Vorteile dieser Abstraktion können beim Implementieren von Massendatentypen ausgenutzt werden, weil die Ordnungs-, Äquivalenzfunktion etc. bereits durch *Domain* als eine zusätzliche Abstraktionsebene bereitgestellt werden. Statt der Übergabe einer Kopierfunktion kann der Datentyp die entsprechende Funktion aus *Domain* benutzen. Die jeweilige *new*-Funktion eines Massendatentyps wird an die speziellen Eigenschaften der *Domain*-Operationen angepaßt. Exemplarisch kann die Ordnungsfunktion auf dem Typ *String.T* die Groß- und Kleinschreibung von Buchstaben ignorieren, obwohl in der Deklaration von *String* die Groß- und Kleinschreibung unterschieden wird. Im folgenden wird ein Beispiel eines allgemeinen Massendatentyps aufgeführt:

```
let new(E, K <: Ok  keyOf(:E) :K  domain :Domain.T(K)) :Bulk(E K) =
  if not (optional.null(domain.order)) then
    ...        (* Ordnung von Domain ausnutzen, z. B. string.domain *)
  elsif not(optional.null(domain.hash)) then
    ...        (* Hashing von Domain ausnutzen *)
  ...
  end
```

Als Beispiel zur Benutzung dieser Funktion lautet eine Instanzbildung für einen exemplarischen Massendatentyp *bulk.T* auf einem Elementtyp zur Beschreibung einer Person:

```
let personBulk = bulk.new(:Person :String
                          let keyOf = fun(p :Person) p.name
                          let domain = string.domain)
```

[8] Der Datenyp *optional.T* stellt sicher, daß ein Wert auf jeden Fall vorhanden ist. Der Wert *optional.nil* bedeutet, daß der Wert nicht sicher angegeben werden kann, daß er also ungewiß ist.

Operationsname	Beschreibung	Wert für *array*
equiv	Äquivalenzrelation == auf dem Wertebereich mit der Eigenschaft $(a == b)/\backslash(b == c) ==> (a == c)$	==
mutable	boolsche Variable, die angibt, ob veränderliche Komponenten im Wertebereich existieren, d. h. ob das Ergebnis von == von Seiteneffekten beeinflußt wird	*true*
clone	Kopierfunktion, die eine vollständige Kopie erstellt, in welcher auch veränderliche Subkomponenten kopiert sind. Dies ist ein *deepClone* für Datentypen mit der *mutable*-Eigenschaft, sonst ein *shallowClone*	deepClone
default	Defaultwert, der bei Komponenten mit der Eigenschaft *mutable* mit Hilfe der Kopierfunktion gegen Seiteneffekte geschützt werden muß	nil
hash	Hashfunktion, welche die Äquivalenzrelation erhält und nicht auf den veränderlichen Objektbezeichnern (*OIDs*) des Objektspeichers aufbauen darf	hash
pOrder	partielle Ordnungsfunktion, welche die Transitivität, Asymmetrie und Reflexivität erfüllt, sowie konsistent mit der Äquivalenzrelation ist	pOrder
order	lineare Ordnungsfunktion, welche konsistent mit der partiellen Ordnung und der Äquivalenzrelation ist	order
toInt	Abbildungsfunktion vom Wertebereich in die Menge der ganzen Zahlen *Int* mit Erhaltung der Äquivalenzrelation	nil
minInt, MaxInt	ein minimaler bzw. maximaler Wert für die Abbildungsfunktion *toInt*	nil, nil

Tabelle 5.6: Operationen der Abstraktion *Domain*

Als eine Weiterführung dieser Idee wird es möglich, die Geschwindigkeit der Ausführung oder die Einhaltung der Ordnung einer Behälterstruktur als vorrangige Anforderung vorzugeben. Auf dieser Vorgabe basierend könnte von einem Generator automatisch ein passender Massendatentyp geliefert werden.

5.5 Spezifikations- und Dokumentationsrichtlinien

Im Hinblick auf eine einfache Wiederverwendung von Modulen und als Kommunikationshilfe in der Teamarbeit ist eine gute Dokumentation des Quelltextes von seiten des Entwerfers von Bibliotheken unerläßlich. Diese Dokumentation sollte so gestaltet sein, daß ein Werkzeug (*Browser*) leicht die Struktur des Quelltextes und seiner Dokumentation erkennen und wesentliche Abschnitte extrahieren kann.

Wichtige Information über ein Modul sollte für den Anwender, der für sein Anwendungssystem möglichst viele der schon bestehenden Module nutzen möchte, im Dokumentenkopf der Schnittstelle zusammengefaßt sein. Dazu gehört das Verhalten des Moduls und ein kurzer Überblick über das Leistungsvermögen. Besonderheiten sollten gleichfalls vermerkt sein. Jede Operation innerhalb des Moduls muß in der Schnittstelle dokumentiert werden. Der Zweck der Operation sowie die Ein- und Ausgabeparameter sollten hier aufgeführt werden. Ebenfalls sollten die Vorbedingungen dargelegt werden, die vor dem Operationsaufruf vom Benutzer sichergestellt werden müssen. In gleicher Weise sind die Nachbedingungen, die vom Entwerfer des Moduls garantiert werden, zu dokumentieren. Diese vollständigen, vom Entwerfer der Bibliothek zur Verfügung gestellten Mitteilungen müssen in allen Bibliotheken einheitlich sein, um ein Auffinden dieser Auskünfte auch für einen automatischen Browser möglich zu machen. Ein Vorschlag zur Formatierung und Einbindung dieser Dokumentation wird im Folgenden kurz dargelegt.

Weitere Dokumente enthalten ergänzende Mitteilungen über die Zusammenhänge der Bibliotheken, Schnittstellen und Implementationen, möglichst in grafischer Darstellung.

5.5.1 Dokumentenköpfe

Die DIN 66 230 [DIN 85] beschreibt als Teil für die Programmdokumentation Programmkenndaten, die im nächsten Abschnitt jeweils **fett** gedruckt werden. Die Anforderungen der DIN werden erläutert und im Zusammenhang mit dem Tycoon System auf ihre Verwendbarkeit hin bewertet. Die beschriebenen Kenndaten sind für Programme erstellt worden, aber sind übertragbar auf die Erfordernisse einer modularen oder objektorientierten Sprache, so daß ihre Anwendbarkeit für Module, Schnittstellen und Bibliotheken des Tycoon Systems hier kurz diskutiert werden soll.

Programmbezeichnung: Name und Version eines Programms sind zur eindeutigen Kennung eines Programms (bzw. einer Schnittstelle, Modulimplementation oder Bibliothek) nötig. Zur Identifizierung von Schnittstellen, Modulen und Bibliotheken ist die Angabe des **Programmnamens** unbedingt notwendig. Eventuell muß noch der Name des Moduls oder der Bibliothek angegeben werden, zu der dieses Modul gehört. Der **Autorenname** wird in der DIN nicht genannt, sollte aber anstelle einer Zuständigkeitsreferenz in allen Schnittstellen, Modulen und Bibliotheken eingetragen werden.

Zur Unterscheidung von Programmvarianten ist eine **Variantenbezeichnung** einzusetzen, wenn verschiedene Anwendungsbereiche oder Plattformen bestehen, für welche

dieselben Programmnamen gelten. Dies ist lediglich bei der Verwendung von plattformabhängigen Bibliotheken des Tycoon Systems (z. B. *newsenv*) nötig.

Zur Unterscheidung der Versionen wird eine **Versionsbezeichnung** für Schnittstellen, Module und Bibliotheken geführt. Hier sollte der Änderungsstand beschrieben sein. Für Tycoon ist ein **Freigabedatum** der ersten und der aktuellen Version überflüssig, weil es kein kommerzielles System ist.

Deskriptoren: Schlüsselwörter ermöglichen eine textuelle Suche. Sie charakterisieren die Aufgabe des Programms. Dieser Punkt kann für Tycoon wegfallen, da diese Wörter auch in der Kurzbeschreibung (siehe unter **Aufgabe**) gefunden werden können.

Aufgabe: Die Beschreibung des Zwecks eines Programms besteht aus einer **Kurzbeschreibung**, welche eine anwendungsbezogene Beschreibung der Aufgabe einschließlich der Ein- und Ausgabe erfordert. Sie ist im Tycoon System für Schnittstellen, Module und Bibliotheken notwendig. Ebenso dient eine derartige Beschreibung für Operationen zu ihrer präzisen Unterscheidung.

Darüber hinaus müssen verwendete **Methoden**, d. h. Theorien und Rechenverfahren zur Lösung eines Problems, aufgeführt werden. Sie können für das Tycoon System in der Kurzbeschreibung untergebracht werden.

Da das Tycoon System keine kommerzielle Anwendung hat, sind **Vorschriften** mit Hinweisen auf Gesetze, Normen, Richtlinien etc. in der Dokumentation nicht zu erwarten. Eine Ausnahme bilden Referenzen auf Standards wie die ISO-Norm oder Industristandards. Falls es für eine Schnittstelle nötig ist, gehört diese Erläuterung inhaltlich ebenfalls in die Kurzbeschreibung.

Bestimmte **Besonderheiten** als Abgrenzung des Anwendungsbereiches, insbesondere fachliche Voraussetzungen und Einschränkungen, können ebenfalls in die Kurzbeschreibung aufgenommen werden und bedürfen keines eigenen Eintrags im Dokumentenkopf von Tycoon-Dateien.

Programmbedarf: Die extern benötigten Programme können einerseits **Betriebssysteme** sein, welche in Tycoon auf eine Plattform beschränkt sind. Zum anderen können es **Unterprogramme** sein, die zum Ablauf benötigt werden. Diese Angaben sind im Tycoon System insbesondere für die C-Bibliotheken zur Anbindung externer Dienste (z. B. StarView) von Bedeutung und sollten in diesen Fällen innerhalb der Aufgabenbeschreibung im Dokumentenkopf aufgeführt werden. Direkte Abhängigkeiten zwischen TL-Bibliotheken werden durch die *import*-Konstrukte hergestellt und werden in einer eigenen Rubrik im Dokumentenkopf festgehalten.

Programmiersprachen: Die verwendete Sprache und die Übersetzungsumgebung sind grundlegende Daten für die Benutzung des Programms. Die Angaben der **Sprachen** werden für das Tycoon System zunehmend bedeutsam, da durch Syntaxerweiterungen und Implementierungen von weiteren, auf TLbasierenden Sprachen (z. B. TooL), eine Vielfalt an Bibliotheksmodulen mit unterschiedlichen Anforderungen an ihre Umgebung entstehen. Daher ist auch die Versionsangabe der benutzten **Übersetzer** für die Dokumentation wichtig.

Zuständigkeiten: Die für Entwicklung, Vertrieb und Pflege zuständigen Personen existieren für das Tycoon System nur in eingeschränktem Maße. Da Tycoon an einem wissenschaftlichen Institut mit wechselnden Projektteilnehmern erstellt wird, reicht es aus, für jede Bibliotheksdatei den Autoren und die letzten Bearbeiter festzuhalten.

Weitere Forderungen zur Dokumentation aus der DIN kommen für das Tycoon System nicht in Betracht, weil es nicht kommerziell vertrieben wird und die höhere Programmiersprache TL umfaßt, die von unteren Ebenen, wie z. B. dem Speicherbedarf und Dateizugriffsarten, abstrahiert.

Zu diesen entbehrlichen Angabe gehören die nötigen Gerätearten. Die Betriebssysteme, für die das System lauffähig ist, werden in der allgemeinen Dokumentation des Tycoon Systems angegeben und haben keinerlei Bedeutung für die einzelnen Bibliotheksdateien. Ebenso ist die Programmgröße und sein Speicherbedarf für das Tycoon System von untergeordneter Bedeutung, da es hauptsächlich nur für das Unix-Betriebssystem entwickelt wurde und darauf eingesetzt wird. Es müssen keine Betriebsarten unterschieden werden und die Angabe der Dateiorganisation(sequentiell, gestreut mit oder ohne Schlüssel) sowie der Zugriffsart (physisch-sequentiell, logisch-sequentiell oder wahlfrei) ist unnötig. Konventionen als rechtliche, technische oder organisatorische Bedingungen, wie Urheberrechte, Mitbenutzungsvereinbarungen etc., werden nur bei Verkauf in einer Begleitdokumentation angegeben und sind für einzelne Module des Tycoon Systems i. A. nicht anzuwenden. Zu den sonstigen Angaben über das erstellte Programm, die allein für den Vertrieb wichtig sind, zählen Installierungen mit einer Auflistung aller Geräte sowie die Preise und Kosten. Gesonderte Dokumentation hierüber muß für das Tycoon System im Bedarfsfall erstellt werden.

Gleichfalls müssen keine Angabe über Unterlagen in den Bibliotheksmodulen erfolgen. Handbücher oder Sprachberichte als externe Dokumentation existieren ohnehin nicht für alle Bibliotheken. Es gibt Technische Berichte und Referenzen auf weitere Literatur in den Seiten des Arbeitsbereichs DBIS im *WorldWideWeb*. Diese Information ist jedem Benutzer von Tycoon zugängig, so daß sich in den Quelltexten des Tycoon Systems ein derartiger Verweis erübrigt.

Die von der DIN geforderte Dokumentation ist insgesamt zu umfangreich, um sie für jede Bibliotheksdatei des Tycoon Systems zu erstellen. Die Implementierer der Bibliotheken sollen in den folgenden Abschnitten einen verbindlichen Vorschlag zur Dokumentation innerhalb der Quelldateien für das Tycoon System finden, der mit möglichst geringem Dokumentationsaufwand die für andere Programmierer wesentliche Information bereitstellt.

Schnittstellen und Module

Da für jeden Programmierer die Programmfunktionalität eine höhere Priorität genießt als die Kommentierung, ist es ungewiß, ob sämtliche Kommentare immer auf dem neuesten Stand gehalten werden. Aus diesem Grund soll soviel Information wie möglich in die Programmiersprache selbst eingebettet werden. Dazu zählt die Wahl der Namen und das Bereitstellen von Konstrukten für Zusicherungen oder importierte Module. Ein Wiederholen dieser Angaben im Dokumentenkopf dient nur einem besseren Überblick und soll dem Leser die Information kompakt darlegen. Es ist ebenfalls nützlich für ein Werkzeug, welches diese Information herausfiltern und Bibliotheksanwendern zur Verfügung stellen soll. Werden Dateiname, Autor, Datum, Versionsnummer, Kurzbeschreibung und die Zugehörigkeit zu einem System für alle Versionsstände von einem Versionskontrollsystem verwaltet, so bleibt als Information nur die

ausführliche Zweckbeschreibung jedes Moduls für den Dokumentenkopf. Da im Tycoon System derartige Werkzeuge nur eingeschränkt zur Verfügung stehen, müssen auch die obigen Angaben in den Dokumentenköpfen enthalten sein.

Alle Dokumentenköpfe werden im Quelltext mit dem Kommentarzeichen von TL geklammert, da sie nicht vom Typüberprüfer gelesen werden dürfen. Zusätzlich sollten sie zu Beginn und am Ende des Dokumentenkopfes mit einer kennzeichnenden Zeichenfolge geklammert werden, um diese Art von Kommentaren von den übrigen Kommentaren im Quelltext abzuheben. Mit Hilfe eines Werkzeugs können diese Kommentare anhand der festgelegten Schlüsselworte aus dem gesamten Quelltext herausgefiltert werden. Dies dient der Erstellung von Dokumentation. Ein derartiges Werkzeug ist dem Erstellen der *flat-short Form* von Eiffel-Klassen nachgebildet (siehe Abschnitt 3.2.2). Man erhält eine Übersicht über die gesamte Bibliothek oder bestimmte Schnittstellen bzw. Operationen, ohne die Details der Implementierung zu sehen. Der Nachteil gegenüber dem Eiffel-Werkzeug besteht darin, daß die Kurzübersicht nicht direkt aus dem Quelltext erzeugt werden kann, sondern darauf angewiesen ist, daß die Dokumentenköpfe auf den aktuellen Stand gebracht sind.

Als Schlüsselworte werden für die Dokumentenköpfe von Schnittstellen und Modulen die Zeichenfolge $!HB$ für *header begin* (Anfang des Kopfs) und $!HE$ für *header end* (Ende des Kopfs) vorgeschlagen. Diese Zeichen sind derart gewählt, daß sie weder der Anfang eines Schlüsselwortes der Sprache TL, noch der Anfang eines Bezeichners im Quelltext sind. Durch diese Maßnahme wird ausgeschlossen, daß das Werkzeug einen anderen Kommentar als Dokumentenkopf erkennt.

Für Tycoon wird eine Liste mit folgenden Feldern als Dokumentenkopf für jede Schnittstelle bzw. jede Modulimplementation als sinnvoll erachtet:

File: Name der Schnittstelle bzw. des Moduls

System: Name der Bibliothek, in der diese Schnittstelle bzw. dieses Modul eingebunden ist

Author, Date: Autor und Datum der Erstellung

Version: Versionsnummer (optional)

Interface: Name der Schnittstelle, die zu dem Modul gehört (nur für Modulimplementationen)

Purpose: Aufgabe dieser Schnittstelle bzw. dieses Moduls, dazu besondere Eigenschaften, eventuell Einschränkungen der Benutzung etc.

Imports: Importierte Schnittstellen und Module

Private: Private Operationen, d.h. eine Liste der Operationen, die nur innerhalb dieser Modulimplementation aufgerufen werden (optional)

Updates: Änderungsliste mit Autor, Datum, Version und Beschreibung

Die Reihenfolge dieser Felder muß als verbindlich angesehen werden, um eine Bearbeitung mit einem Werkzeug zu erleichtern. Zusätzlich sollte im Modul die Implementation beschrieben werden, z. B. welche Algorithmen benutzt wurden oder welches Programmierziel, beispielsweise Optimierung des Speicherbedarfs oder der Ausführungsgeschwindigkeit, verfolgt wurde. Die Information im Kopf der Module steht nur den Bibliotheksimplementierern zur

Verfügung. Ein Bibliotheksbenutzer sollte allein die Schnittstellenbeschreibungen sehen. Sie enthalten hinreichend Information zur Auswahl der passenden Schnittstelle für seine Applikation.

In der Änderungsliste muß genau aufgeführt werden, welche Operationen geändert, gelöscht oder hinzugefügt wurden. Dazu gehört der Name des Autors, das Datum und eine genaue Beschreibung der Änderung, die auch die neu vergebene Versionsnummer beinhaltet.

Die Versionsnummer der Schnittstelle kann entfallen, wenn die Datei in einem Versionskontrollsystem gehalten wird. Zur Kontrolle sollte aber jeder, der eine Änderung der Datei vornimmt, auch eine neue Versionsnummer vergeben. Diese Nummer muß nicht mit der Versionsnummer des Kontrollsystems übereinstimmen.

Vorgeschlagen wird eine aus drei ganzen Zahlen zusammengesetzte Versionsnummer $m.n.o$. Die Priorität der Zahlen nimmt von links nach rechts ab. Die Zahl an erster Stelle (m) wird jeweils erhöht, wenn die Schnittstelle geändert wurde. Die zweite Stelle (n) wird verändert, wenn die Implementation einer Operation sich geändert hat oder dem Modul eine neue Operation hinzugefügt wurde. Die dritte Stelle (o) wird jedes Mal erhöht, wenn sich eine Datenstruktur verändert hat oder ein interner Fehler behoben wurde. Dadurch wird eine Aufwärtskompatibilität von Anwendungsprogrammen erreicht. Sämtliche Versionen eines Bibliotheksmoduls, deren Versionsnummern, von links nach rechts entsprechend der Priorität verglichen, kleiner oder gleich der vom Anwendungsprogramm geforderten Versionsnummer sind, können benutzt werden.

Die importierten Schnittstellen sollten trotz eines eigenen Deklarationskonstruktes auch im Dokumentenkopf stehen, um eine vollständige Beschreibung im Kopf zu erhalten, die eventuell mit einem Werkzeug aus der Schnittstellen- bzw. Moduldatei ausgelesen werden kann. Mit Hilfe dieser Angabe kann von jeder beliebigen Anwendung aus der gesamte Import-Baum konstruiert werden. In diesem Fall soll die Wartbarkeit einen höheren Stellenwert einnehmen als die geforderte Vermeidung von Redundanz im Quelltext.

Allerdings sind die Angaben der exportierten Operationen in einem Modul überflüssig, weil alle Operationen, die von anderen Schnittstellen benutzt werden können, in der Schnittstelle aufgeführt sind. Hilfreich für einen Implementierer von Modulen ist dagegen die Angabe aller nicht-exportierten Operationen mit einer kurzen Zweckbeschreibung, z. B. Hilfsfunktion für eine Schnittstellenoperation. Es wird daher empfohlen, alle derartig privat vereinbarten Operationen eines Moduls im Kopf der Modulimplementation aufzuführen, um es anderen Programmierern zu erleichtern, diese Bibliothek zu pflegen.

Mit diesen Vereinbarungen sieht ein vollständiger Dokumentenkopf einer Schnittstelle aus der Standardbibliothek von TL beispielsweise folgendermaßen aus:

```
(*!HB
  File:     String.ti
  System:   BaseLib
  Author:   Florian Matthes
  Date:     11-JAN-94
  Version:  1
  Purpose:  Strings with a mutable state.
  Imports:  Iter, Order
  Updates:  (Date)      (Name)       (Description)
            27-AUG-95   Tanja Jung   Added classification of operations,
                                     new order of operations.
```

!HE)*

Der Dokumentenkopf des zugehörigen Moduls gleicht diesem Dokument bis auf eine zusätzliche Zeile mit der Angabe der Schnittstelle, für welche dieses Modul die Implementierung enthält.

Operationen

Die Implementationen der einzelnen Operationen werden in einem eigenen Dokumentenkopf beschrieben, welcher zu jeder Operation ausgeführt wird. Dieser Kommentar wird durch eine andere, eindeutig kennzeichnende Zeichenfolge geklammert, um eine automatische Auswahl der Schnittstellen-, Modul- und Operationsköpfe oder nur der Schnittstellen- bzw. Moduldokumentenköpfe zu erlauben. Hier kommt der strukturierte Ansatz eines Programmtextes, der schon in Abschnitt 2.2.3 beschrieben wurde, zum Ausdruck, da alle Operationen zu einem Modul gehören und im Sinne des Buch-Paradigmas dessen *Abschnitte* bilden. Um eine Liste der Beschreibungen der zu einer Schnittstelle gehörenden Operationen erstellen zu können, werden diese Dokumentenköpfe mit der Zeichenfolge *!OB* (für *operation begin*) eingeleitet und enden mit den Zeichen *!OE* (für *operation end*). Die Dokumentenköpfe der Schnittstellen bzw. der Module sind zu Rate zu ziehen, wenn der Bibliotheksbenutzer sich einen Überblick über die Aufteilung der Bibliotheken und deren Angebot machen will. Sie zeigen eine grobe Sicht der Bibliotheken, wohingegen die Dokumentenköpfe der Operationen detaillierte Kenntnisse über die vorhandenen Datenobjekte und Operationen einer Bibliothek vermitteln. Gerade die Kenntnisse auf dieser unteren Ebene benötigt ein Anwendungsprogrammierer, um effizient vorhandene Schnittstellen wiederverwenden zu können.

Die folgende Angabe ist für jede deklarierte Operation unerläßlich:

Description: Aufgabe und Durchführung dieser Operation, dazu evtl. Erläuterung und Herkunft des Algorithmus

Weitere Angaben sollten unbedingt dokumentiert werden. Für einige spezielle Operationen können diese Angaben jedoch nicht gemacht werden, da sie keinen Bedingungen unterliegen. Für den Fall, daß die Konstrukte zur Zusicherungsprüfung in die Sprache integriert sind, kann auf diese Angaben im Dokumentenkopf der Operation verzichtet werden:

Parameters: Ein- und Ausgabeparameter mit ihrer Bedeutung, sowie das Ergebnis einer Funktion

In: Eingabeparameter sowie Ein- und Ausgabeparameter

Out: Ausgabeparameter sowie Ein- und Ausgabeparameter

Result: Ergebnis der Funktion

Pre: Vorbedingungen, unter denen die Operation korrekt, d.h. wie beschrieben, abläuft

Post: Nachbedingungen, die nach Beendigung der Operation garantiert erfüllt sind, wenn die Vorbedingungen eingehalten wurden

Invariant: Invarianzen, die sowohl vor Beginn als auch nach Beendigung der Operation gelten

Exceptions: Möglicherweise auftretende Ausnahmen

Für die Angabe des Operationsnamens und der zugehörigen Parameterliste in einem Dokumentenkopf gibt es Vor- und Nachteile. Beide Angaben scheinen überflüssig zu sein, da sie aus der direkt vor dem Kommentar stehenden Signatur hervorgehen. Die Wiederholung dieser Angaben innerhalb des Kopfes kann dazu führen, daß eine Änderung hier vergessen wird und eine inkonsistente Beschreibung entsteht. Andererseits sollte ein automatisches Werkzeug den Namen und die Parameter aus dem Dokumentenkopf eindeutig entnehmen können, da im anderen Fall Signatur und zugehöriger Kommentar im Quelltext verstreut sind, welches vom Compiler nicht bemerkt würde, aber dieses Werkzeug vor unlösbare Probleme stellt. Ein weiteres Argument für die Aufnahme der Parameterliste in den Dokumentenkopf ist die Beschreibung der Parameter. Aus der Signatur allein geht nicht hervor, welcher Parameter nur Eingabeparameter ist und welcher Parameter der Ein- und Ausgabe dient und damit verändert wird, mit Ausnahme der Deklaration eines Parameters mit var. Aus diesem Grund sollte jeder Programmierer von Bibliotheksschnittstellen seine Parameter sorgfältig im Kommentar der jeweiligen Operation erläutern. Im Gegensatz dazu wird auf die Angabe des Operationsnamens in TL allgemein verzichtet, weil die Möglichkeit des Auseinanderreißens von Signatur und Kommentar einer Operation im Quelltext als gering eingestuft wird und daher das Software-Werkzeug den Operationsnamen aus der Deklaration direkt entnehmen kann.

Damit sieht der vollständige Dokumentenkopf für eine Operation der Schnittstelle *String* wie folgt aus:

```
isEmpty(string :T) :Bool
(*!OB
   Description: Test whether a string is empty.
   In: string
   Out: -
   Result: true, if empty, false otherwise
   Pre: -
   Post: -
   Invariant: -
   Exceptions: -
!OE*)
```

Kategorisierung von Operationen

Die Reihenfolge der Operationen innerhalb eines Moduls oder einer Schnittstelle unterliegt ebenfalls gewissen Regeln, um dem Anwender von Bibliotheken das Suchen einer passenden Operation zu vereinfachen. Es wird eine Klassifizierung der Operationen vorgenommen. Diese Operationsklassen hängen von der Art des Moduls bzw. der deklarierten Typen ab. Nicht für jeden Typ kann für jede dieser Klassen eine Operation deklariert werden.

Die folgende Aufzählung gibt die allgemeine Klassifizierung in den Tycoon-Bibliotheken wieder. Diese ist insbesondere für Massendatentypen anwendbar. Für Schnittstellen, deren Zweck sehr stark von dem der Standard- und Datenstrukturbibliotheken abweicht, müssen evtl. eigene Kategorien von Operationen eingeführt werden. Es ist sinnvoll für den Bibliothekimplementierer, wenn eine Datei mit sämtlichen Kategorien als Gerüst existiert. Diese kann

er kopieren und die einzelnen Kategorien mit adäquaten Operationen füllen. Die Reihenfolge der Kategorien soll die Häufigkeit ihres Auftretens in den Schnittstellen der Bibliotheken widerspiegeln. Die in allen Schnittstellen nötigen Kategorien, wie z. B. *Error*, *Creation* und *Duplication* stehen aus diesem Grund am Beginn der Dateien.

Aus den Betrachtungen der Bibliotheken anderer Programmiersprachen (vgl. Abschnitt 4.2.2 über Eiffel) und einer intensiven Beschäftigung mit den Operationen der Tycoon-Basisbibliotheken ergeben sich folgende Kategorien:

- **Error** umfaßt die Operationen zur Ausnahmebehandlung nach Auftreten von Fehlern in den übrigen Operationen des Datentyps. Der Implementierer muß darauf achten, daß es mindestens eine allgemeine Ausnahmebehandlung für jeden Datentyp gibt, um eine fehlerlose Weitergabe der Programmkontrolle zu ermöglichen.

- **Creation** umfaßt alle Operationen zur Erstellung von Datenstrukturen. Hierunter werden auch die Initialisierungen von Datenobjekten gefaßt.

- **Size** bedeutet die Feststellung der Größe einer Datenstruktur. Hierzu zählen bei Behälterdatentypen die Feststellung der Anzahl der Elemente und bei den Basisdatentypen die Feststellung des minimalen und maximalen Werts.

- **Duplication** sind alle Operationen, welche Datenstrukturen ganz oder teilweise kopieren. Eine allgemeine Kopierfunktion, welche das ganze Objekt kopiert, wird für jede Datenstruktur gefordert. Sie wird in der Schnittstelle *value* allgemein zur Verfügung gestellt. Weitere Operationen, welche Kopien von Objekten erzeugen, sind z. B. die *substring*-Operation zum Kopieren von Teilen einer Zeichenkette und ähnliche Operationen in fast allen Behälterdatentypen.

- **Order** bezeichnet die Ordnungsfunktionen, die auf dem Datentyp existieren.

- **Basic operations** beinhaltet alle spezifischen Routinen eines Datentyps. Für die fundamentalen Datentypen der *baseLib* werden in dieser Kategorie die arithmetischen Operationen deklariert.

- **Comparison** ist die Kategorie aller Vergleichsoperationen. Dazu zählen im wesentlichen die boolschen Vergleiche =, ! =, >, <, >= und <=. Es können aber auch andere, datentypspezifische Operationen deklariert werden. Voraussetzung für die Existenz derartiger Operationen ist das Vorhandensein einer Ordnung auf den Werten des Typs.

- **Status report** umfaßt alle Operationen, welche Angaben über den Zustand eines Datentyps betreffen. Die in allen Behälterdatentypen enthaltene Operation ist *isEmpty*, welche feststellt, ob der Behälter leer ist. Üblicherweise geben derartige Operationen einen Wert vom Typ *Bool* zurück.

- **Deletion** enthält alle Löschoperationen. Dies kann das Entfernen eines Elements aus dem Behälterdatentyp oder das vollständige Löschen eines ganzen Objekts betreffen. Hierzu gehört z. B. eine Art der *substring*-Operation, welche den Teil der Zeichenkette aus der Datenstruktur herausliest und dabei die ursprüngliche Datenstruktur verändert.

- **Update** umfaßt die Konkatenation von Teilzeichenketten und weitere Operationen, welche über Wert bzw. Schlüsselwert auf den Datentyp zugreifen und Elemente einfügen.

- **Concatenation** bezeichnet die Konkatenation von Datenstrukturen, wobei ein neues Objekt erstellt wird.
- **Element change** deklariert Operationen, die Änderungen innerhalb einer Datenstruktur vornehmen.
- **Searching** umfaßt alle Operationen, die nach einem Wert bzw. Index (bei indizierten Strukturen) in einer Datenstruktur suchen und entsprechend den zugehörigen Index oder Wert zurückgeben.
- **Status setting** umfaßt alle Operationen, welche die Angaben aus der Kategorie *Status report* über den Zustand eines Datentyps verändern können. Als Ausnahme hierzu wird die zu *isEmpty* gehörende Löschoperation, welche sämtliche Elemente einer Datenstruktur entfernt, üblicherweise in der Kategorie *Deletion* vermerkt.
- **Iteration** umfaßt alle Operationen, welche die Benutzung eines Iterators betreffen.
- **Conversion** klassifiziert Operationen, welche die Darstellung des Datentyps von einem Typ zu einem anderen verändern. Sämtliche Operationen aus Abschnitt 5.2.3 werden in dieser Kategorie zusammengefaßt.

5.5.2 Richtlinien für den Code

Die Nützlichkeit von Namenskonventionen und weiteren Richtlinien für ein Programmlayout wurde bereits in Kapitel 2 erläutert. In den Handbüchern über Tycoon [Matthes, Müßig 93; Mathiske et al. 93] werden Richtlinien zur Benennung und zum Layout genannt. Diese Vereinbarungen werden beibehalten und von den folgenden Vereinbarungen nicht berührt.

Des weiteren sind in den Bibliotheksmodulen einige Regeln zur Kommentierung und zum Layout befolgt worden, die bislang in keiner Anleitung schriftlich festgehalten sind. Dazu zählt unter anderem die Verwendung eines i. A. einheitlichen Dokumentenkopfs und von verschiedenen Implementierern uneinheitlich gehandhabte Klassifizierungen der Operationen pro Datenstruktur. Zusätzlich zu den Regeln der Handbücher werden im Folgenden die Richtlinien aus Kapitel 2 berücksichtigt, um eine vollständige Konvention für TL-Programme zu bilden. Dort sind die Begründungen und einige Beispiele für die Richtlinien enthalten. Die folgenden Richtlinien stellen als ein Ergebnis dieser Arbeit eine minimale Konvention dar, die für sämtliche in TLgeschriebenen Programme in Zukunft beachtet werden sollte. Dabei bleibt es jedem Programmierer selbst überlassen, sich eigenen, weitergehenden Konventionen zu unterwerfen.

Kommentare

- Die Dokumentationsköpfe von Schnittstellen, Modulen und Operationen müssen möglichst vollständig ausgefüllt sein. Die einzelnen Felder dieser speziellen Kommentare werden im Abschnitt 5.5.1 genau erläutert.
- Kommentare müssen verständlich geschrieben sein und sollten im wesentlichen keine Abkürzungen enthalten. Einfache Wiederholungen des zu kommentierenden Codes nützen einem Leser und möglichen Wiederverwender nicht. Aus den verschiedenen der in Abschnitt 2.2.1 dargestellten Kategorien von Kommentaren sind daher diejenigen zu bevorzugen, welche Codezeilen zusammenfassen und die Absicht des Codes beschreiben.

- Der Stil von Kommentaren muß möglichst änderungsfreundlich sein. Daher soll er möglichst nahe bei dem Code stehen, welchen er beschreibt. Eine Zeile soll nicht bis zu einer von anderen Zeilen bestimmten Position gefüllt werden, weil damit die Änderung der vorgebenden Zeile weitere Änderungen nach sich ziehen würde. Aus demselben Grund sollen keine Unterstreichungen bestimmter Länge vorgenommen werden und auch die Kommentare am Ende einer Zeile sollen vermieden werden. Ausführliche Beispiele zur Kommentierung stehen in [McConnell 93].

- Kommentare müssen klar und deutlich formuliert sein. Ihre Bedeutung muß aus dem Programm und der übrigen Dokumentation hervorgehen. Grundsätzlich sollen sie die Vorgänge auf der Ebene der Problemstellung beschreiben (das *Warum*) und nicht die Lösung auf der Ebene der Programmiersprache (das *Wie*). Die Absicht und der Zweck der Instruktionen muß erläutert werden.

- Die Kategorien für Operationen werden jeweils in einer eigenen Zeile vor den deklarierten Operationen vermerkt. Weil diese Kategorien fest sind und nicht mit dem Inhalt der Operationen, der sich häufig ändern kann, zusammenhängen, kann hierfür eine Layout-Darstellung gewählt werden, die von den aufgestellten Richtlinien abweicht. Eine Kategorie wird in Kommentarklammern gesetzt und zur Absetzung vom übrigen Text mit Gedankenstrichen versehen. Sie wird linksbündig geschrieben:

(— Status report — *)*

- Die Einheit und der Wertebereich einer numerischen Größe soll im Kommentar festgehalten werden.

- Kontrollstrukturen werden vor dem Block kommentiert. Der Zweck und das Ergebnis müssen daraus hervorgehen.

- Jeder Programmabschnitt, der nach den vorstehenden Regeln nicht gut kommentiert werden kann, sollte neu entworfen werden.

Layout

- In einer let-Bindung, einer Zuweisung und bei Infix-Operatoren stehen vor und nach dem Operator jeweils ein Leerzeichen.

- Vor, aber nicht nach einem Typ- (:) und einem Subtypzeichen (<:) steht ein Leerzeichen.

- Nach, aber nicht vor einem Komma oder Semikolon steht ein Leerzeichen.

- Die Operatoren ".", "?", und "!" stehen ohne Leerzeichen.

- Vor und nach Klammern stehen keine Leerzeichen.

- Zwischen zwei Signaturen stehen zwei Leerzeichen.

- Jede Verschachtelungsebene wird zwei Leerzeichen eingerückt.

- Eine Anweisungsfolge wird unterhalb der Anweisung, die sie einführt, eingerückt. Eine kurze Folge kann auch in einer Zeile stehen. In [Mathiske et al. 93, Anhang C] werden Beispiele für die Einrückung sämtlicher Kontrollstrukturen von TL gegeben.

- Kommentare werden unterhalb des ersten Wortes eingerückt, falls sie nicht in eine Zeile passen.
- Kommentare, die zu einer Folge von Anweisungen gehören, werden vor diese Folge gesetzt. Die Folge endet mit einer Leerzeile.
- Kommentare, die zu einer einzigen Deklaration oder Definition gehören, werden unmittelbar danach gesetzt und beginnen in derselben Tiefe der Verschachtelung.

Namenskonvention

Da TLdie Länge von Namen nicht beschränkt, können eine Reihe von Regeln aufgestellt werden, ohne Abkürzungen für Namen zu reglementieren. Obwohl die Dateinamen unter DOS, Windows, Windows NT und Apple Macintosh eingeschränkt sind, werden diese Restriktionen wegen der höheren Aussagekraft langer Namen nicht eingeführt. Damit ist ein Programmierer mit TLin der Lage, Programme mit hoher Lesbarkeit zu erstellen. Für die Portierung von TL-Programmen auf Betriebssysteme mit beschränkter Namenslänge können z. B. über Schlüsselwörter für das jeweilige Betriebssystem die entsprechenden Namen für die jeweilige Zielmaschine aus einer Tabelle bestimmt werden. Die folgenden Richtlinien geben den Rahmen für gut lesbare Programme an. Sie stellen eine Empfehlung an alle TL-Programmierer dar.

- Namen für Werte, wie Variable, Funktionen, Module und Ausnahmen, beginnen mit einem Kleinbuchstaben. Dies gilt auch für vordefinierte Schlüsselworte, welche im Kontext eines Werts benutzt werden.
- Namen für Typen, wie Typoperatoren und Schnittstellen, beginnen mit einem Großbuchstaben. Dies gilt auch für vordefinierte Schlüsselworte, welche im Kontext eines Typs benutzt werden.
- Namen der Bibliotheken werden klein geschrieben, weil eine Bibliothek ein *First Class* Objekt ist und als Bindeglied zwischen der Laufzeitumgebung und den eingebauten Typen gelten kann. Ihre Namen werden im Singular geschrieben, weil besonders in der objektorientierten Welt eine Bibliothek wieder als ein Objekt eines umfassenderen Konzepts aufgefaßt werden kann. Ein Postfix *Lib* soll dabei hervorheben, daß es sich hierbei nicht um die Implementation eines Datentyps handelt, sondern um eine Bibliothek. Dieses Postfix wird für alle weiteren Bibliotheksnamen gefordert.
- Die nachfolgenden Buchstaben aller Namen sind klein, mit der Ausnahme, daß bei zusammengesetzten Namen die jeweils ersten Buchstaben einer Komponente groß geschrieben werden.
- Routinennamen bestehen üblicherweise aus einem Verb. Dies drückt aus, welche Aktion auf den Parametern ausgeführt wird. Der Modulname wird nicht im Routinennamen aufgeführt, da bei abstrakten Datentypen eindeutig ist, auf welches Modul sich die Routine bezieht. Der Rückgabewert einer Funktion muß mit dem Namen präzise beschrieben sein.
- Namen für Funktionen zur Abfrage von Eigenschaften werden mit einem Adjektiv beschrieben. Dieses wird mit einem Präfix *is* versehen, um hervorzuheben, daß eine derartige Funktion einen boolschen Rückgabewert hat. Die Vorsilbe *is* wird für Tycoon als

Wortpaar	Bedeutung
add/remove, *insert/delete*	Einfügen bzw. Löschen von Elementen
increment/decrement	Erhöhen bzw. Verringern eines Werts um einen bestimmten Wert
lock/unlock	Sperren bzw. Freigeben
source/destination	Quelle bzw. Ziel einer Änderungsoperation
begin/end	Anfang bzw. Ende eines Abschnitts
first/last	erstes bzw. letztes Element
AtFirst/AtLast	vor dem ersten bzw. nach dem letzten Element in einem Behälter (nur als Namenszusatz)
set/get	Ändern bzw. Lesen eines Elements
min/max	minimaler bzw. maximaler Wert
open/close	Öffnen bzw. Schließen
source/target	Quelle bzw. Ziel einer (Lese-/Schreib-)Operation
create/dispose	Erstellen bzw. Zerstören einer Datenstruktur
new	Erstellen einer Datenstruktur, die mit der automatischen Speicherbereinigung wieder gelöscht wird und daher keine gegensätzliche Operation besitzt
up/down	Rollen nach oben bzw. unten
next/previous	nächstes bzw. vorheriges Element
show/hide	Anzeigen bzw. Verstecken einer Variablen
start/stop	Starten bzw. Beenden einer Aktivität

Tabelle 5.7: Gegensatzpaare

Schlüsselwort gewählt, da dieses Wort meist Fragen (im Englischen) einleitet und direkt auszusprechen ist.

- Routinennamen müssen so lang sein, daß sie alle Ausgaben und Seiteneffekte der Routine exakt beschreiben.

- Routinenparameter sollten bei ähnlichen Routinen in jeweils konsistenter Reihenfolge stehen. Es werden als aufeinanderfolgende Kategorien Eingabeparameter, Ein- und Ausgabeparameter und am Schluß nur Ausgabeparamter empfohlen. Im Fall eines ADT sollte dieser Typ als erster Parameter in der Parameterliste stehen.

- Variablennamen sollen leicht lesbar, treffend und leicht merkbar sein. Sie beschreiben das Problem selbst und nicht den gewählten Lösungsweg.

- Eine Variable wird nur zu einem einzigen Zweck innerhalb ihres Gültigkeitsbereichs verwendet.

- In der Tabelle 5.7 werden die für Variablen und Routinen verwendeten Paare mit gegensätzlicher Bedeutung aufgelistet. Diese sollen für alle Anwendungen bindend sein. Einige Worte in den Gegensatzpaaren, wie *begin*, *hide* oder *open*, sind nur in zusammen-

gesetzten Bezeichnern erlaubt, weil sie in TL als Schlüsselworte definiert sind.

Die Paare für Einfüge- und Löschoperationen haben im Zusammenhang mit der Klassifikation von Operationsnamen eine festgelegte, unterschiedliche Bedeutung. Diese wird in Abschnitt 5.2.2 erläutert.

- Bislang wurde die Schleifenvariable in *for*-Schleifen lt. Namenskonvention mit *i* bezeichnet. Um den Code besser lesbar zu machen und den Sinn der Schleife herauszustellen, sollte besser ein aussagekräftiger Name gewählt werden.
- Temporäre Variable mit nichtssagenden, kurzen Namen sollten vermieden werden.
- Konstante, welche in der Bedeutung eines Aufzählungstyps benutzt werden, werden durch ein gemeinsames Präfix gekennzeichnet.
- Konstantennamen werden nicht mit dem Wert benannt, den sie darstellen, um änderungsfreundlich zu bleiben.
- Im Falle eines ADT wird der exportierte Typ eines Bibliotheksmoduls mit *T* bezeichnet.
- Die Hauptausnahme eines Moduls wird mit *error* bezeichnet.

In Ergänzung der vorigen Regeln folgen zum Abschluß der Namenskonventionen für das Tycoon System ein paar Punkte zur Erklärung, welche Arten von Namen nicht verwendet werden sollten:

- Namen sollen aussagekräftig sein, d.h. es sollen statt Abkürzungen ganze Worte benutzt werden. Eine Ausnahme besteht für häufig verwendete Worte, wie beispielsweise Typnamen. In der initialen Umgebung werden die Basistypen für ganze Zahlen, Zeichen und boolsche Werte daher weiterhin in abgekürzter Form deklariert.
- Namen mit ähnlicher Bedeutung sollen vermieden werden.
- Namen mit unterschiedlicher Bedeutung, aber ähnlichen Namen werden vermieden.
- Ähnlich klingende Namen (*Homonyme*) werden vermieden.
- Numerierungen in Namen können durch eine besser gewählte Datenstruktur, u. U. Felder, vermieden werden.
- Syntaktisch inkorrekte Schreibweisen, die lediglich phonetisch interpretierbar sind, werden vermieden.
- Namen, die nur durch schwer unterscheidbare Buchstaben auseinandergehalten werden können, werden vermieden.
- Namen von Standardbibliotheksroutinen und vordefinierte Worte werden nicht verwendet.

Kapitel 6

Zusammenfassung und Ausblick

Bei der Erstellung von Bibliotheken ist eine leicht durchschaubare Hierarchisierung für ein besseres Verständnis von Vorteil. Mit Hilfe von objektorientierten Konzepten wie der Vererbung wird eine Hierarchie gebildet, die dem Verwender von Bibliotheken das Erfassen der Bedeutung der einzelnen Bausteine erleichtert. Eine Realisierung dieses Ansatzes war in TL nicht möglich, da dieser Sprache das objektorientierte Konzept der Vererbung fehlt. Allein die in TL zur Modellierung von objektorientierten Konzepten geeignete Subtypisierung ist nicht ausreichend, um unter Beibehaltung statischer Typkorrektheit einen hohen Grad an Wiederverwendbarkeit zu erreichen. Das Erben von Code im Sinne des objektorientierten Ansatzes erfordert neue Typrelationen wie das *Type-Matching*, welches im Rahmen einer Diplomarbeit [Lotter, Römer 95] für die Sprache TooL [1] [Gawecki, Matthes 95] angewendet wurde. Diese Sprache ermöglicht eine vollständig objektorientierte Programmierung mit Mehrfachvererbung und enthält wie TL ein Konzept für Bibliotheken.

Um trotz des fehlenden Klassenkonzepts in TL homogene und aussagekräftige Bibliotheken zu erstellen, wurden in dieser Arbeit Richtlinien zur Benennung von Schnittstellen, Modulimplementationen und Operationen der Standardbibliothek des Tycoon Systems entwickelt. Weiterhin wurden die in Tycoon bestehenden Funktionen der Bibliotheksschnittstellen vereinheitlicht und ergänzt, so daß dem Benutzer sinnvoll strukturierte Bibliotheken zur Verfügung stehen.

Für die Zukunft ist wünschenswert, daß die in dieser Arbeit entwickelten Richtlinien des Style Guides weiterhin in anderen Bibliotheken und Sprachdialekten Beachtung finden. Da die Massendatenbibliothek *bulkLib* am häufigsten in Anwendungsprogrammen benutzt wird, ist die Effektivität bei Einhaltung der Style Guides hier am größten. Um die Konsistenz in Tycoon zu fördern und damit die Verständlichkeit zu erhöhen, ist eine einheitliche Beschreibung notwendig. Daher werden diese Empfehlungen, soweit die Sprachkonzepte es erlauben, ebenfalls für TooL angewendet.

In einer integrierten Umgebung ist es ferner wünschenswert, daß der Verwender von Bibliotheken durch automatische Werkzeuge direkt am Bildschirm in der Auswahl von Modulen unterstützt wird. Das Ziel einer effizienten Wiederverwendung ist nur erreichbar, wenn durch verschiedenartige Darstellungen der Information aus den Modulen ein schneller Zugriff auf die Module, die den gewünschten Anforderungen entsprechen, gewährleistet wird. Der Aufwand zur Änderung in einer nicht streng hierarchisch gegliederten Bibliotheksstruktur wird hierdurch reduziert. Die vom Benutzer gesteuerte Auswahl und Darstellung sollte folgende

[1] Tycoon object-oriented Language

Werkzeuge umfassen:

Ein Bibliotheksbrowser gibt einen Gesamtüberblick über alle Module einer Bibliothek. Die Abhängigkeiten der Bibliotheken untereinander werden dargestellt. Ein Benutzer dieser Bibliothek kann einzelne Schnittstellen interaktiv auswählen und die enthaltenen Operationen anzeigen lassen. Ebenso kann er die Importabhängigkeiten der Schnittstellen und Module auf einfache Weise verfolgen. Diese geforderten Darstellungen können wahlweise als Text oder als Grafik erstellt werden.

In anderen Systemen gibt es bereits Werkzeuge, die einen Teil der gewünschten Aufgaben erledigen. Für Smalltalk existiert zur Darstellung der Klassenbibliothek ein *System Browser* (siehe Abschnitt 3.2.3), der die Abhängigkeiten der Klassen in verschiedenen Fenstern am Bildschirm textuell darstellt. In Eiffel kann für eine kompakte Darstellung der Klassenschnittstelle eine *short form* (siehe Abschnitt 3.2.2) erstellt werden. Eine Übersicht über mehrere Klassen mit ähnlichen Eigenschaften ist in der *short form* der Klassendarstellung mit wenig Aufwand zu erhalten.

Ein erster Schritt zu einem solchen Werkzeug für das Tycoon System ist das am Arbeitsbereich DBIS entwickelte Werkzeug zur Dokumentation der Modulverwaltung, welches es erlaubt, aus TL-Quelltexten anhand von Schlüsselworten Namen herauszufiltern und deren Abhängigkeiten in verschiedenen Modulen zu dokumentieren.

Ein Template-Generator unterstützt den Programmierer bei der Erstellung von einheitlichen Modulen und Schnittstellen. Der Dokumentenkopf wird automatisch beim Erstellen einer neuen Schnittstelle oder eines neuen Moduls als Vorlage zur Verfügung gestellt. Dieses Template gibt dem Implementierer einen Dokumenationsrahmen vor, den er nur noch auszufüllen braucht. Ebenso werden die Kategorien möglicher Methoden für die Schnittstellen vorgegeben, so daß der Implementierer einen Leitfaden hat, welche Operationen für den aktuellen Datentyp eingefügt werden müssen und bei Einhaltung dieses Templates alle Schnittstellen die gleiche Reihenfolge in den Kategorien und Parametern der Methoden aufweisen.

Ein Bibliotheksmanager verwaltet die Bibliotheken mit ihren Abhängigkeiten zwischen den Schnittstellen und Modulen automatisch. So kann gewährleistet werden, daß Änderungen in den Bibliotheken ohne zeitlichen Aufwand und korrekt in das System übernommen werden. Dem Implementierer wird die Änderung aller abhängigen Module abgenommen. Auch die Dateien, welche die Bibliotheken definieren, werden automatisch bei Änderung eines Moduls auf den neuesten Stand gebracht. Diese Automatisierung sichert die Konsistenz der Bibliotheken.

Die fundamentale Motivation dieser Arbeit, Wiederverwendung von Bausteinen für möglichst alle Anwendungen so einfach wie möglich zu gestalten, findet sich auch in [Bertram 95]. Hier wird die Wiederverwendung von abstrakten Teilschemata, die bereits in den frühen Phasen der Softwareentwicklung anfallen, gefordert. Die hierzu notwendigen Qualitätskriterien entsprechen im wesentlichen den in dieser Arbeit beschriebenen Anforderungen an Software-Bibliotheken. Die in Abschnitt 1.2.2 aufgestellten Richtlinien zur Unterstützung der Wiederverwendung sind auf andere Phasen der Softwareentwicklung übertragbar. Dies sollte in zunehmendem Maße in der Zukunft genutzt werden, um Kosten bei der Softwareentwicklung insbesondere für im Wettbewerb stehende Unternehmen zu reduzieren. Die systematische Wiederverwendung von Wissen, die bis vor nicht allzu langer Zeit dadurch geschah,

"daß einzelne Mitarbeiter ihr angehäuftes Wissen in neue Projekte einbrachten, quasi als Wiederverwendung durch Mitarbeitertransfer" [Bertram 95], muß in Zukunft systematisiert und personenunabhängig durchgeführt werden. Zu diesem Zweck wird die Wiederverwendung von abstrakten Schemata im Rahmen von Datenmodellen vorgeschlagen. Diese Schemata bilden eine Bibliothek, vergleichbar mit den Softwarebausteinen von TL, und sind somit für alle Projekte zugänglich. Im Unterschied zu den in dieser Arbeit beschriebenen Bibliotheken werden die Schema-Bibliotheken im Softwareentwurfsprozeß in einer früheren Phase als der Programmierung eingesetzt. Das Konzept von Bibliotheken ist somit universell und nicht auf Programmiersprachen beschränkt.

Literaturverzeichnis

Baker 93: Henry G. Baker. „Iterators: Signs of Weakness in Object-Oriented Languages". *ACM SIGPLAN OOPS Messenger*, 4(3):18–25, Juli 1993.

Beaudouin-Lafon 94: Michel Beaudouin-Lafon. *Object-oriented Languages. Basic Principles and Programming Techniques.* Chapman & Hall, London, 1994.

Bertram 95: Martin Bertram. „Qualitätsaspekte der Wiederverwendung von Schemata". *EMISA FORUM*, 2:18–24, 1995. Fachgruppenveröffentlichung der GI.

Biggerstaff, Perlis 89: T. Biggerstaff und A. Perlis. *Software Reusability. 1. Concepts and Models.* Addison-Wesley, 1989.

Boehm, others 84: Barry W. Boehm et al. „ Software Development Environment for Improving Productivity". *Computer*, Seite 30–44, Juni 1984.

Booch, Vilot 90: Grady Booch und Michael Vilot. „The Design of the C++ Booch Components". *ACM SIGPLAN Notices*, 25(10):1–11, 1990. Proceedings OOPSLA/ECOOP '90.

Booch 86: G. Booch. „Object-Oriented Development". *IEEE Transactions on Software Engineering*, SE-12(2):211–221, Februar 1986.

Booch 91: G. Booch. *Object-oriented Design with Applications.* Benjamin/Cummings, Redwood City, Ca, 1991.

Börstler 89: Jürgen Börstler. *Wiederverwendbarkeit und Softwareentwicklung - Probleme, Lösungsansaetze und Bibliographie.* Aachener Informatik-Berichte Nr. 89-5, RWTH Aachen, 1989.

Breilmann 95: Markus Breilmann. „Portierung des Tycoon-Systems auf das Macintosh Betriebssystem". Studienarbeit. Universität Hamburg, Fachbereich Informatik, 1995. In Vorbereitung.

Cardelli, Wegner 85: Luca Cardelli und P. Wegner. „On Understanding Types, Data Abstraction and Polymorphism". *ACM Computing Surveys*, 17(4):471–522, Dezember 1985.

Cardelli 89: L. Cardelli. „Typeful Programming". Technischer Bericht, Digital Systems Research Center Reports, Palo Alto, Mai 1989.

Cardelli 90: L. Cardelli. „The Quest Language and System (Tracking Draft)". Technischer Bericht, Digital Systems Research Center Reports, Palo Alto, Mai 1990.

Coggins 90: James M. Coggins. „Designing C++ Libraries“. In: *USENIX C++ Conference*, Seite 25–35, April 1990.

Cohen et al. 92: Bruce Cohen, Douglas Hahn, und Neil Soiffer. „Pragmatic Issues in the Implementation of Flexible Libraries for C++“. In: *USENIX C++ Conference*, August 1992.

Cox 86: B. J. Cox. *Object-Oriented Programming. An Evolutionary Approach.* Addison-Wesley, Reading, MA, 1986.

Dahl et al. 72: O.-J. Dahl, E.W. Dijkstra, und C.A.R. Hoare. *Srtuctured Programming.* Academic Press, London, 1972.

Date, White 89: C. J. Date und Colin J. White. *A Guide To SQL/DS.* Addison-Wesley, 1989.

Dijkstra 76: E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

DIN 85: DIN. *Software-Entwurf, Programmierung, Dokumentation, Schaltzeichen. Normen.* DIN-Taschenbuch 166. Deutsches Institut für Normung e. V., Berlin, 2. Auflage, 1985.

Ellis, Stroustrup 90: M. Ellis und B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

Engesser 93: Hermann Engesser, Hrsg. *DUDEN Informatik.* B.I.-Wissenschaftsverlag, Mannheim, 2. Auflage, 1993. Bearbeitet von Prof. Dr. Volker Claus und Dr. Andreas Schwill.

Floyd 67: R. W. Floyd. „Assigning Meanings to Programs“. In: *American Mathematical Society Symposium in Applied Mathematics*, Band 19, Seite 19–31, 1967.

Gall et al. 95: Harald Gall, Manfred Hauswirth, und René Klösch. „Objektorientierte Konzepte in Smalltalk, C++, Objective-C und Modula-3“. *Informatik-Spektrum*, 18(4):195–202, August 1995.

Gawecki, Matthes 95: Andreas Gawecki und Florian Matthes. „TooL: A Persistent Language Integrating Subtyping, Matching and Type Quantification“. FIDE Technical Report Series FIDE/95/135, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, März 1995.

Glass 82: Robert L. Glass. *Modern Programming Practices: A Report from Industry.* Prentice-Hall, Englewood-Cliffs, N.J., 1982.

Gnatz 77: R. Gnatz. „mod-Funktion und Maximumbestimmung“. Technischer Bericht TUM-INFO-7712, TU München, 1977.

Goldberg, Robson 83: Adele Goldberg und David Robson. *Smalltalk-80 - The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

Gorlen et al. 90: Keith E. Gorlen, Sanford M. Orlow, und Perry S. Plexico. *Data Abstraction and Object-Oriented Programming in C++.* Wiley, 1990.

Gorlen 87: Keith E. Gorlen. „An Object-Oriented Class Library for C++ Programs“. *Software - Practice and Experience*, 17(12):899–922, Dezember 1987.

Hoare 69: C. A. R. Hoare. „An Axiomatic Basis for Computer Programming“. *Communications of the ACM*, 12(10):576–580, Oktober 1969.

Hoare 72: C. A. R. Hoare. „Proof of Correctness of Data Representations“. *Acta Informatica*, 1:271–281, 1972.

Hoffmann 87: H.-J. Hoffmann. *Smalltalk verstehen und anwenden.* Hanser, 1987.

Horning et al. 93: Jim Horning, Bill Kalsow, Paul McJones, und Greg Nelson. „Some Useful Modula-3 Interfaces“. Technischer Bericht 113, digital Systems Research Center, Palo Alto, Ca, Dezember 1993.

Horowitz, Munson 84: E. Horowitz und J. B. Munson. „An Expansive View of Reusable Software“. *IEEE Transactions on Software Engineering*, SE-10(5):477–487, September 1984.

Kennedy 91: Brian M. Kennedy. „The Features of the Object-oriented Abstract Type Hierarchy (OATH)“. In: *USENIX C++ Conference*, April 1991.

Kernighan, Plauger 74: Brian W. Kernighan und P. J. Plauger. *The Elements of Programming Style.* McGraw-Hill, 1974.

Knuth 92: D. Knuth. *Literate Programming.* Center for the Study of Language and Information (CSLI), Leland Stanford Junior University, 1992.

Lamb 87: David Alex Lamb. „Specification of Iterators“. Technischer Bericht TR 87-195, Kingston Queen's University, Ontario, Canada, Juli 1987.

Lampson et al. 77: Butler W. Lampson, Jim J. Horning, Ralph L. London, J. G. Mitchell, und Gerard L. Popek. „Report on the Programming Language Euclid“. *ACM SIGPLAN Notices*, 12(2):1–79, Februar 1977.

Lea 88: Douglas Lea. „libg++, The GNU C++ Library“. In: *USENIX C++ Conference in Denver, CO*, Seite 243–256, Berkeley, CA, Oktober 1988. Usenix Association.

Lenz et al. 87: Manfred Lenz, Hans Albrecht Schmid, und Peter F. Wolf. „Software Reuse through Building Blocks“. *IEEE Software*, Seite 34–42, Juli 1987.

Lotter, Römer 95: Björn Lotter und Thorsten Römer. „Entwurf und Implementation von TooL-Bibliotheken“. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 1995. In Vorbereitung.

Mandrioli, Meyer 92: Dino Mandrioli und Bertrand Meyer, Hrsg. *Advances in Object-Oriented Software Engineering*, Object-Oriented Series, New York, 1992. Prentice-Hall. Advanced Seminar, Capri, June 1988.

Marcotty, Ledgard 87: Michael Marcotty und Henry Ledgard. *The World of Programming Languages.* Springer-Verlag, New York, 1987.

Martin 91: Bruce Martin. „The Separation of Interface and Implementation in C++“. In: *USENIX C++ Conference*, 1991.

Mathiske et al. 93: Bernd Mathiske, Florian Matthes, und Sven Müßig. „The Tycoon System and Library Manual". Technischer Bericht, DBIS Tycoon Report, Fachbereich Informatik, Universität Hamburg, Dezember 1993.

Matthes et al. 94: Florian Matthes, Sven Müßig, und J. W. Schmidt. „Persistent Polymorphic Programming in Tycoon: An Introduction". Technischer Bericht FIDE/94/106, Universität Hamburg, 1994.

Matthes, Müßig 93: Florian Matthes und Sven Müßig. „The Tycoon Language TL - An Introduction". Technischer Bericht, DBIS Tycoon Report, Fachbereich Informatik, Universität Hamburg, Dezember 1993.

Matthes, Schmidt 91: F. Matthes und J. W. Schmidt. „Bulk Types: Built-In or Add-On?". In: P. Kanellakis und J. W. Schmidt, Hrsg., *Bulk Types and Persistent Data. Proceedings of the Third International Workshop on Database Programming Languages.* Morgan Kaufmann Publishers, September 1991.

Matthes 91: Florian Matthes. „P-Quest: Installation and User Manual". Technischer Bericht 101-91, Fachbereich Informatik, Universität Hamburg, Germany, Oktober 1991.

Matthes 93: Florian Matthes. *Persistente Objektsysteme.* Springer, Berlin, 1993.

McConnell 93: Steven C. McConnell. *Code Complete.* Microsoft Press, 1993.

Melton, Simon 93: Jim Melton und Alan R. Simon. *Understanding the New SQL. A Complete Guide.* Series in data management systems. Morgan Kaufmann, San Francisco, CA, 1993.

Meyer 86: Bertrand Meyer. „Genericity versus Inheritance". In: *ACM Conference on OOPSLA (OO Programming Systems, Languages and Applications)*, Band 21 II, Seite 391–405. ACM Sigplan Notices, 1986.

Meyer 88: Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

Meyer 89: Bertrand Meyer. „Static Typing for Eiffel". Technischer Bericht TR-EI-18/ST, Interactive Software Engineering Inc., 1989.

Meyer 90a: Bertrand Meyer. *Eiffel - The Language.* Prentice-Hall, 1990.

Meyer 90b: Bertrand Meyer. *Eiffel - The Libraries.* Prentice-Hall, 1990.

Meyer 90c: Bertrand Meyer. *Introduction to the Theory of Programming Languages.* International Series in Computer Science. Prentice-Hall, New York, 1990. Reprint 1991.

Meyer 90d: Bertrand Meyer. „Lessons from the Design of Eiffel". *Communications of the ACM*, 33(9):69–88, September 1990.

Meyer 94: Bertrand Meyer. *Reusable Software - The Base Object-Oriented Component Libraries.* Prentice-Hall Int. , Hertfordshire, 1994.

Milner et al. 90: R. Milner, M. Tofte, und R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

Möller, Müller 93: Torsten Möller und Michael Müller. „Konzepte von Basisklassenbibliotheken in C++“. Studienarbeit. Universität Hamburg, Fachbereich Informatik, 1993.

Münnix 94: Petra Münnix. „Objektorientierte Datenmodellierung: Generierung statisch typisierter Repräsentationen“. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 1994.

Nelson 91: G. Nelson, Hrsg. *Systems Programming with Modula-3.* Prentice-Hall Series in innovative technology. Prentice-Hall, Englewood Cliffs, NJ, 1991.

Niederée et al. 92: Claudia Niederée, Sven Müssig, und Florian Matthes. „P-Quest User Manual“. DBIS Tycoon Report 102-92, Fachbereich Informatik, Universität Hamburg, Germany, Februar 1992.

Nievergelt 91a: J. Nievergelt. „Die fünf Grundoperationen der ganzzahligen Arithmetik und das div-mod-Problem“. *Informatik-Spektrum*, 14(2):43–44, Februar 1991.

Nievergelt 91b: J. Nievergelt. „Über das div-mod-Problem und die Normierung ganzzahliger Arithmetik“. *Informatik-Spektrum*, 14(6):351–354, Dezember 1991.

Nievergelt 92: J. Nievergelt. „Entscheidungsgrundlagen für die Normierung der ganzzahligen Arithmetik: Varianten der div- und mod-Operationen“. *Informatik-Spektrum*, 15(2):107–109, April 1992.

Oman, Cook 90a: Paul W. Oman und Curtis R. Cook. „The Book Paradigm for Improved Maintenance“. *IEEE Software*, Seite 39–45, Januar 1990.

Oman, Cook 90b: Paul W. Oman und Curtis R. Cook. „Typographic Style Is More Than Cosmetic“. *Communications of the ACM*, 33(5):506–520, Mai 1990.

Parnas 72: D. L. Parnas. „On the Criteria To Be Used in Decomposing Systems into Modules“. *Communications of the ACM*, 15(12):1053–1058, Dezember 1972.

Pomberger, Blaschek 93: Gustav Pomberger und Günther Blaschek. *Software Engineering - Prototyping und objektorientierte Software-Entwicklung.* Hanser, 1993.

Rees 82: Michael J. Rees. „Automatic Assessment Aids for Pascal Programs“. *ACM Sigplan Notices*, 17(10):33–42, Oktober 1982.

Reiser, Wirth 94: M. Reiser und N. Wirth. *Programmieren in Oberon. Das neue Pascal.* Addison-Wesley, Bonn, 1994.

Rovner et al. 85: P. Rovner, R. Levin, und J. Wick. „On Extending Modula-2 for Building Large, Integrated Systems“. Technischer Bericht, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, Januar 1985.

Schmid 94a: Hans Albrecht Schmid. „Reuse Inhibitors with, and Requirements for Collection Class Libraries“. In: B. Magnusson, B. Meyer, J.-M. Nerson, und J.-F. Perrot, Hrsg., *TOOLS 13 (Technology of Object-Oriented Languages and Systems), Proceedings of the 13th International Conference TOOLS EUROPE '94. Versailles, France*, Seite 243–252, Hertfordshire, 1994. Prentice-Hall.

Schmid 94b: Hans Albrecht Schmid. „Safe Iterators“. In: B. Magnusson, B. Meyer, J.-M. Nerson, und J.-F. Perrot, Hrsg., *TOOLS 13 (Technology of Object-Oriented Languages and Systems), Proceedings of the 13th International Conference TOOLS EUROPE '94. Versailles, France*, Seite 407–418, Hertfordshire, 1994. Prentice-Hall.

Schröder 93: Gerald Schröder. „Syntaktische Erweiterbarkeit von Programmiersprachen unter Benennumgs-, Bindungs- und Typisierungsinvarianzen“. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, August 1993.

Schulz 78: Arno Schulz. *Methoden des Softwareentwurfs und Strukturierte Programmierung.* W. de Gruyter, Berlin, 1978.

Shaw 81: Mary Shaw. *Alphard: Form and Content.* Springer, New York, 1981.

Shen 96: Kai Shen. „Portierung und Erweiterung des Tycoon-Systems auf IBM-PCs unter MS-Windows“. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 1996. In Vorbereitung.

Shneiderman 80: Ben Shneiderman. *Software Psychology: Human Factors in Computer and Information Systems.* Winthrop, Cambridge, Ma., 1980.

Stal, Berger 93a: Michael Stal und Wolfgang Berger. „Aus dem Vollen - Zum Design von C++-Klassenbibliotheken“. *iX Magazin*, Dezember 1993.

Stal, Berger 93b: Michael Stal und Wolfgang Berger. „Wettstreit der Bibliomanen - C++-Klassenbibliotheken“. *iX Magazin*, Dezember 1993.

Tracz 88: Will Tracz. „Software Reuse: Motivators and Inhibitors“. In: W. Tracz, Hrsg., *Software Reuse: Emerging Technology. Tutorial*, Seite 62–67. ComputerSociety Press, Washington, DC, 1988.

Traub 95: Horst-Peter Traub. „Objektorientierte Behälterklassen-Bibliotheken - Konzepte, Entwurf und Implementation“. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Mai 1995.

Uhl, Schmid 90: J. Uhl und H. A. Schmid. *A systematic Catalogue of Reusable Abstract Data Types.* LNCS 460. Springer, 1990.

Wegner 84: Peter Wegner. „Capital-Intensive Software Technology“. *IEEE Software*, 1(3), Juli 1984.

Wegner 87: P. Wegner. „Dimensions of Object-Based Language Design“. In: N. Meyrowitz, Hrsg., *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA)*, Orlando, Florida, 1987.

Weinand et al. 88: André Weinand, Erich Gamma, und Rudolf Marty. „ET++ - An Object-Oriented Application Framework in C++“. *Conference OOPSLA (Object-Oriented Systems, Languages and Applications)*, Seite 46–57, 1988.

Wetzel 94: Ingrid Wetzel. *STYLE: Systematische Entwicklung von Programmierumgebungen.* Dissertation, Universität Hamburg, Fachbereich Informatik, September 1994.

Wichmann 79: B. A. Wichmann. „Integer Division“. *Software - practice and experience*, 9:507–512, 1979.

Wirth 72: Niklaus Wirth. *Systematisches Programmieren. Eine Einführung.* B. G. Teubner, Stuttgart, 1972.

Wirth 88: Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, Berlin, 4. Auflage, 1988.